ViennaRNA

Release 2.8.0

Ronny Lorenz, Ivo L. Hofacker, et al.

Nov 05, 2026

GET STARTED

I Quickstart 3

1 Get Started 5
- 1.1 Download . . . 5
- 1.2 Installing from Source . . . 5
- 1.3 Installing Pre-compiled Packages . . . 8
- 1.4 Customizing the Build Process . . . 10
  - 1.4.1 Streaming SIMD Extension . . . 10
  - 1.4.2 Scripting Language Interfaces . . . 10
  - 1.4.3 Cluster Analysis . . . 10
  - 1.4.4 Kinfold . . . 11
  - 1.4.5 RNAforester . . . 11
  - 1.4.6 Kinwalker . . . 11
  - 1.4.7 RNAlocmin . . . 11
  - 1.4.8 RNAxplorer . . . 11
  - 1.4.9 Link Time Optimization . . . 11
  - 1.4.10 OpenMP . . . 11
  - 1.4.11 POSIX threads . . . 12
  - 1.4.12 SVM Z-score filter . . . 12
  - 1.4.13 GNU Scientific Library . . . 12
  - 1.4.14 Multiple-precision Floating-Point Computations . . . 12
  - 1.4.15 Universal binaries . . . 12
  - 1.4.16 Disable C11/C++11 features . . . 13
  - 1.4.17 Deprecated symbols . . . 13
  - 1.4.18 Single precision . . . 13
  - 1.4.19 Help . . . 13
- 1.5 Build your own Binary Package from SourceCode . . . 13

II User Guides and Tutorials 15

2 User Guide 17
- 2.1 Command-Line Tools . . . 17
  - 2.1.1 Main Programs . . . 18
  - 2.1.2 Additional Programs . . . 18
  - 2.1.3 Utilities . . . 19
- 2.2 Manpages . . . 21
  - 2.2.1 RNA2Dfold . . . 21
  - 2.2.2 RNAaliduplex . . . 24
  - 2.2.3 RNAalifold . . . 27
  - 2.2.4 RNAcofold . . . 39
  - 2.2.5 RNAdistance . . . 48
  - 2.2.6 RNAdos . . . 50
  - 2.2.7 RNAduplex . . . 53
  - 2.2.8 RNAeval . . . 56

2.2.9 RNAfold . . . 62
2.2.10 RNAheat . . . 77
2.2.11 RNAinverse . . . 82
2.2.12 RNALalifold . . . 86
2.2.13 RNALfold . . . 92
2.2.14 RNAmultifold . . . 99
2.2.15 RNApaln . . . 105
2.2.16 RNAparconv . . . 109
2.2.17 RNApdist . . . 112
2.2.18 RNAPKplex . . . 116
2.2.19 RNAplex . . . 119
2.2.20 RNAplfold . . . 124
2.2.21 RNAplot . . . 131
2.2.22 RNApvmin . . . 135
2.2.23 RNAsnoop . . . 141
2.2.24 RNAsubopt . . . 146
2.2.25 RNAup . . . 155
2.3 I/O Formats . . . 161
2.3.1 RNA Structures . . . 161
2.3.2 Multiple Sequence Alignments (MSA) . . . 165
2.3.3 Command Files . . . 168
2.3.4 Energy Parameters . . . 171
2.4 Tutorials . . . 177
2.4.1 Global RNA Secondary Structure Prediction . . . 177
2.4.2 Consensus Structure Prediction . . . 193
2.4.3 RNA-RNA interaction . . . 197
2.4.4 Plotting Structures . . . 203
2.4.5 RNA Design . . . 205
2.4.6 RNA folding kinetics . . . 208

III Advanced Usage 213

3 RNAlib User Guide 215
3.1 Introduction . . . 215
3.2 Basic Structure Prediction . . . 215
3.2.1 Minimum Free Energy (MFE) . . . 215
3.2.2 Partition Function and Equilibrium Probability . . . 224
3.2.3 Suboptimals and Representative Structures . . . 228
3.2.4 Free Energy Evaluation . . . 238
3.3 Advanced Structure Prediction . . . 241
3.3.1 RNA-RNA Interaction . . . 241
3.3.2 Inverse Folding (Design) . . . 242
3.3.3 Classified Dynamic Programming Variants . . . 242
3.3.4 The RNA Secondary Structure Landscape . . . 242
3.4 Tweaking the Predictions . . . 245
3.4.1 Experimental Structure Probing Data . . . 245
3.4.2 Post-transcriptional Base Modifications . . . 246
3.4.3 Ligands Binding to RNA Structures . . . 246
3.4.4 Structure Modules and Pseudoknots . . . 247
3.4.5 The RNA Folding Grammar . . . 247
3.5 Utilities . . . 252
3.5.1 Basic Utilities . . . 252
3.5.2 Nucleotide Alphabets . . . 252
3.5.3 Strings and Nucleic Acid Sequences . . . 252
3.5.4 RNA Secondary Structures . . . 253
3.5.5 Multiple Sequence Alignments . . . 253
3.5.6 Files and I/O . . . 254

3.5.7 Plotting . . . 254
3.5.8 Data Handling and Manipulation . . . 254
3.5.9 Search Algorithms . . . 255
3.5.10 Mathematical Functions . . . 255
3.5.11 Combinatorics Algorithms . . . 255
3.5.12 (Abstract) Data Structures . . . 257
3.5.13 Messages . . . 258
3.5.14 Log Messages . . . 258
3.5.15 Unit Conversion . . . 259
3.6 Linking against RNAlib . . . 259
3.6.1 Compiler and Linker flags . . . 260
3.6.2 The pkg-config tool . . . 261
3.7 SWIG Wrappers . . . 262
3.7.1 Function Renaming . . . 262
3.7.2 Object Oriented Interface . . . 262
3.8 Python Interface . . . 262
3.8.1 Usage . . . 263
3.8.2 Global Variables . . . 263
3.8.3 Pythonic interface . . . 263
3.8.4 Energy Parameters . . . 265
3.9 Callback Functions . . . 265
3.9.1 Why Callbacks? . . . 265
3.9.2 Scripting Language Support . . . 266
3.10 RNAlib API v3.0 . . . 266
3.11 Example Code . . . 267
3.11.1 C Examples . . . 267
3.11.2 Python Examples . . . 275
3.11.3 Perl 5 Examples . . . 282

4 RNAlib API Reference 287
4.1 Structure Prediction Algorithms . . . 287
4.1.1 Minimum Free Energy (MFE) . . . 287
4.1.2 Partition Function and Equilibrium Probabilities . . . 305
4.1.3 Suboptimals and Representative Structures . . . 345
4.1.4 Free Energy Evaluation . . . 375
4.1.5 RNA-RNA Interaction . . . 490
4.1.6 Inverse Folding (Design) . . . 495
4.1.7 Classified Dynamic Programming Variants . . . 497
4.1.8 The RNA Secondary Structure Landscape . . . 512
4.1.9 Experimental Structure Probing Data . . . 532
4.1.10 Post-transcriptional Base Modifications . . . 558
4.1.11 Ligands Binding to RNA Structures . . . 566
4.1.12 Structure Modules and Pseudoknots . . . 568
4.1.13 RNA Folding Grammar . . . 577
4.2 Utilities . . . 641
4.2.1 Basic Utilities . . . 641
4.2.2 (Abstract) Data Structures . . . 647
4.2.3 Files and I/O . . . 763
4.2.4 Messages . . . 780
4.2.5 Log Messages . . . 783
4.2.6 Nucleotide Alphabets . . . 792
4.2.7 Strings and Nucleic Acid Sequences . . . 797
4.2.8 Multiple Sequence Alignments . . . 806
4.2.9 RNA Secondary Structures . . . 812
4.2.10 Mathematical Functions . . . 836
4.2.11 Data Handling and Manipulation . . . 849
4.2.12 Search Algorithms . . . 853
4.2.13 Combinatorics Algorithms . . . 856

4.2.14 Unit Conversion . . . 862
4.2.15 Plotting . . . 865
4.3 SWIG Wrappers . . . 891
4.3.1 SWIG Wrapper Notes . . . 891
4.3.2 The RNA Python module . . . 908
4.4 Other . . . 1095
4.4.1 Callbacks in RNAlib . . . 1095
4.4.2 Deprecated List . . . 1096

IV Miscellaneous 1113

5 News 1115
5.1 Version 2.8.0 . . . 1115
5.2 Version 2.7.0 . . . 1115
5.3 Version 2.6.0 . . . 1116
5.4 Version 2.5.0 . . . 1116
5.5 Version 2.4.0 . . . 1116
5.6 Version 2.3.0 . . . 1116
5.7 Version 2.2.0 . . . 1117
5.8 Version 2.1.9 . . . 1117
5.9 Version 2.1.7 . . . 1117
5.10 Version 2.1.0 . . . 1117
5.11 Older news . . . 1117
5.12 Version 2.0 . . . 1117
5.13 Version 1.8 . . . 1118
5.14 Version 1.7 . . . 1118
5.15 Version 1.6 . . . 1118
5.16 Version 1.5pre . . . 1119
5.17 Version 1.4 . . . 1119
5.18 Version 1.3.1 . . . 1119
5.19 Version 1.3 . . . 1119
5.20 Version 1.2.1 . . . 1120
5.21 Version 1.2 . . . 1120

6 Changelog 1121
6.1 Version 2.8.x . . . 1121
6.1.1 Unreleased . . . 1121
6.1.2 Version 2.8.0 . . . 1121
6.2 Version 2.7.x . . . 1123
6.2.1 Version 2.7.2 . . . 1123
6.2.2 Version 2.7.1 . . . 1124
6.2.3 Version 2.7.0 . . . 1125
6.3 Version 2.6.x . . . 1127
6.3.1 Version 2.6.4 . . . 1127
6.3.2 Version 2.6.3 . . . 1128
6.3.3 Version 2.6.2 . . . 1128
6.3.4 Version 2.6.1 . . . 1129
6.3.5 Version 2.6.0 . . . 1129
6.4 Version 2.5.x . . . 1131
6.4.1 Version 2.5.1 . . . 1131
6.4.2 Version 2.5.0 . . . 1132
6.5 Version 2.4.x . . . 1133
6.5.1 Version 2.4.18 . . . 1133
6.5.2 Version 2.4.17 . . . 1135
6.5.3 Version 2.4.16 . . . 1135
6.5.4 Version 2.4.15 . . . 1136
6.5.5 Version 2.4.14 . . . 1137
6.5.6 Version 2.4.13 . . . 1138

6.5.7 Version 2.4.12 . . . 1139
6.5.8 Version 2.4.11 . . . 1140
6.5.9 Version 2.4.10 . . . 1140
6.5.10 Version 2.4.9 . . . 1141
6.5.11 Version 2.4.8 . . . 1142
6.5.12 Version 2.4.7 . . . 1142
6.5.13 Version 2.4.6 . . . 1143
6.5.14 Version 2.4.5 . . . 1144
6.5.15 Version 2.4.4 . . . 1145
6.5.16 Version 2.4.3 . . . 1145
6.5.17 Version 2.4.2 . . . 1146
6.5.18 Version 2.4.1 . . . 1147
6.5.19 Version 2.4.0 . . . 1147
6.6 Version 2.3.x . . . 1149
6.6.1 Version 2.3.5 . . . 1149
6.6.2 Version 2.3.4 . . . 1149
6.6.3 Version 2.3.3 . . . 1150
6.6.4 Version 2.3.2 . . . 1151
6.6.5 Version 2.3.1 . . . 1151
6.6.6 Version 2.3.0 . . . 1151
6.7 Version 2.2.x . . . 1152
6.7.1 Version 2.2.10 . . . 1152
6.7.2 Version 2.2.9 . . . 1153
6.7.3 Version 2.2.8 . . . 1153
6.7.4 Version 2.2.7 . . . 1154
6.7.5 Version 2.2.6 . . . 1154
6.7.6 Version 2.2.5 . . . 1155
6.7.7 Version 2.2.4 . . . 1155
6.7.8 Version 2.2.3 . . . 1156
6.7.9 Version 2.2.2 . . . 1156
6.7.10 Version 2.2.1 . . . 1156
6.7.11 Version 2.2.0 . . . 1156
6.8 Version 2.1.x . . . 1157
6.8.1 Version 2.1.9 . . . 1157
6.8.2 Version 2.1.8 . . . 1157
6.8.3 Version 2.1.7 . . . 1158
6.8.4 Version 2.1.6 . . . 1158
6.8.5 Version 2.1.5 . . . 1158
6.8.6 Version 2.1.4 . . . 1158
6.8.7 Version 2.1.3 . . . 1158
6.8.8 Version 2.1.2 . . . 1158
6.8.9 Version 2.1.1 . . . 1159
6.8.10 Version 2.1.0 . . . 1159
6.9 Version 2.0.x . . . 1159
6.9.1 Version 2.0.7 . . . 1159
6.9.2 Version 2.0.6 . . . 1159
6.9.3 Version 2.0.5 . . . 1159
6.9.4 Version 2.0.4 . . . 1159
6.9.5 Version 2.0.3 . . . 1159
6.9.6 Version 2.0.2 . . . 1160
6.9.7 Version 2.0.1 . . . 1160
6.10 History . . . 1160

7 Bibliography **1165**

8 How to cite the ViennaRNA Package **1167**
8.1 Main References . . . 1167
8.2 Particular Algorithms and Features . . . 1167

8.2.1 Consensus structure prediction . . . 1167
8.2.2 Local pair probabilitiy and accessibility . . . 1167
8.2.3 Suboptimal secondary structures . . . 1167
8.2.4 k,l - Neighborhood . . . 1168
8.2.5 Experimental RNA structure probing . . . 1168
8.2.6 Structure Constraints . . . 1168
8.2.7 Non-redundant Boltzmann sampling . . . 1168

9 Frequently Asked Questions 1169
9.1 Missing EXTERN.h . . . 1169
9.2 Linking fails with LTO error . . . 1169

10 Contributing to ViennaRNA 1171
10.1 Reporting Bugs . . . 1171
10.2 Pull Request Process . . . 1171
10.3 Contributors License Agreement . . . 1172

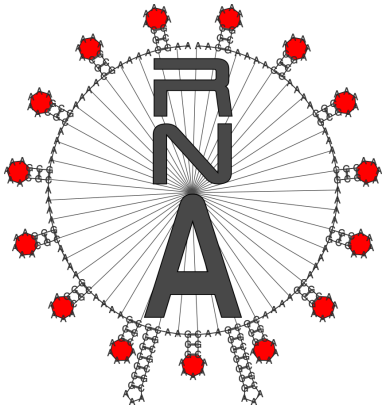
11 About the ViennaRNA Package 1173
11.1 History . . . 1173
11.2 Contributors . . . 1173

12 License 1185

Bibliography 1187

Python Module Index 1191

Index 1193



Predict, analyze, compare and design RNA secondary structures

The ViennaRNA Package offers a comprehensive collection of algorithms dedicated to RNA secondary structures. **Physics based RNA secondary structure prediction** is at the heart of the package. It allows for the prediction of **stable structures** and **structural ensembles**, to compute **thermodynamic probabilities**, such as for base-pairs, to **compare alternative folds**, and to **design RNA sequences** that preferentially fold into a particular structure.

For **individual analyses**, **automated workflows**, or the **development of new methods**, our software provides both, ready-to-use *Command-Line Tools* and an API for `C`, `Python` and `Perl` (see *RNAlib User Guide*).

Part I

Quickstart

CHAPTER

ONE

GET STARTED

The ViennaRNA Package comes with a variety of executable programs and scripts as well as a C-library that provides access to our implemented algorithms. Moreover, the C-library is wrapped to scripting languages such as `Perl 5` and `Python`.

1.1 Download

Source Code

Installing from sourcecode is the recommended way to get most out of the ViennaRNA Package. For best portability the ViennaRNA package uses the GNU tools `autoconf`, `automake`, and `libtool` and can thus be compiled and installed on almost every computer platform.

ViennaRNA Package - Source Code - v2.8.0

See *Installing from Source* for instructions to install the source code package.

Binary Packages

For your convenience, we also provide precompiled binary packages for several Linux distributions and installer executables for Windows and MacOS X.

Pre-compiled binary packages for various Linux distributions and installers for Windows and MacOSX can be obtained from the official website at https://www.tbi.univie.ac.at/RNA/index.html#binary_packages.

See *Installing Pre-compiled Packages* for instructions to install the binary packages.

1.2 Installing from Source

The instructions below are for installing the ViennaRNA package from source code.

> **Tip**
>
> Read the *Customizing the Build Process* section **before** you install our software if you intend to deviate from the default setup.

From release tarball

Usually you'll just download the *latest source tarball*, unpack it and run `configure` and `make`:

```
tar -zxvf ViennaRNA-2.8.0.tar.gz
cd ViennaRNA-2.8.0
./configure
make
sudo make install
```

Alternative installation target directory

If you do not have root privileges on your computer, you might want to install the ViennaRNA Package to a location where you actually have write access to. To do so, you can set the installation prefix of the `./configure` script like so:

```
tar -zxvf ViennaRNA-2.8.0.tar.gz
cd ViennaRNA-2.8.0
./configure --prefix=$HOME/.local
make
make install
```

This will install the entire ViennaRNA Package into your home's `~/.local/` directory that is commonly used for user-installed software. Just make sure that your `PATH` environment variable contains the `$HOME/.local/bin` directory such that our executables are looked-up for at the proper location.

The `--prefix` can be any other directory if you want to keep your installed software separate from each other. The `make install` command will then create the corresponding `bin/`, `lib/`, `share/` directories within the directory you specified.

MacOS X quirks

Although users will find `/usr/bin/gcc` and `/usr/bin/g++` executables in their directory tree, these programs are not at all what they pretend to be. Instead of including the GNU programs, Apple decided to install `clang/llvm` in disguise. Unfortunately, the default version of `clang/llvm` does not support `OpenMP` (yet), but only complains at a late stage of the build process when this support is required. Therefore, it seems necessary to deactivate OpenMP support, e.g.:

```
./configure --disable-openmp
```

See also…

OpenMP, *Universal binaries*, and *Missing EXTERN.h*

From Git repository

You can also get the latest source code from our git repository hosted at https://github.com:

```
git clone https://github.com/ViennaRNA/ViennaRNA.git
```

Important steps before you proceed

To proceed with the configuration and installation you need to perform some additional steps **before** actually running the `./configure` script:

1. Unpack the `libsvm` archive to allow for SVM Z-score regression with the program `RNALfold`:

   ```
   tar -xzf src/libsvm-3.37.tar.gz -C src
   ```

2. Unpack the `dlib` archive to allow for concentration dependency computations with the program `RNAmultifold`:

   ```
   tar -xjf src/dlib-20.0.1.tar.bz2 -C src
   ```

3. Install the autotools toolchain and the additional maintainer tools `gengetopt`, `help2man`, `flex`, `xxd`, and `swig` if necessary. For instance, in Debian based distributions, the following packages need to be installed:

   - `build-essential` (basic build tools, such as compiler, linker, etc.)
   - `autoconf`, `automake`, `libtool`, `pkg-config` (autotools toolchain)

- `gengetopt` (to generate command line parameter parsers)
- `help2man` (to generate the man pages)
- `bison` and `flex` (to generate sources for `RNAforester`)
- `vim-common` (for the `xxd` program)
- `swig` (to generate the scripting language interfaces)
- `liblapacke` (for `RNAxplorer`)
- `liblapack` (for `RNAxplorer`)
- A `fortran` compiler, e.g. `gfortran` (for `RNAxplorer`)

4. Finally, run the `autoconf/automake` toolchain:

```
autoreconf -i
```

After that, you can compile and install the ViennaRNA Package as if obtained from the distribution tarball.

Documentation

Usually, there is no need to **(re-)build the documentation**, it is already part of the source code tarball! However, if you compile from a freshly cloned git repository or you've changed any of the source code, building the documentation is probably required.

Technical details

Our API is documented with comments that are automatically parsed by doxygen. The extracted API documentation is then processed further by breathe and finally integrated into a comprehensive reference manual written in ReStructuredText. This manual is then compiled into HTML and/or PDF format by Sphinx.

To succeed with the compilation a few maintainer tools are required:

- `doxygen` (to extract the API documentation)
- `sphinx-build` (to compile the manual)
- `pdflatex` (to compile a PDF version of the manual)
- `sphinx-multiproject`
- `myst-parser`
- `sphinx-copybutton`
- `sphinxcontrib-bibtex`
- `pydata-sphinx-theme`
- `sphinx-design`

If all the above tools are available, compilation of the documentation should succeed without any further problems

Manual documentation re-build trigger

If you intend to rebuild the documentation although it already has been build before, you can delete the file `doc/refman-html`. This file serves as a placeholder to prevent `make` from redundantly building the documentation, e.g. after extraction of the release tarball where typically none of the source code has been altered.

```
rm doc/refman-html
make
```

To trigger parsing of the API description from source code headers, you need to additionally remove the `doxygen-xml` placeholder file:

```
rm doc/doxygen/doxygen-xml doc/refman-html
make
```

Python Package

Our source code allows for building a **stand-alone** `Python` interface. For that, we provide the necessary packaging files `pyproject.toml`, `setup.cfg`, `setup.py` and `MANIFEST.in`. They are created by our `autoconf` toolchain after a successful run of `./configure`. Particular default compile-time features may be (de-)activated by setting the corresponding boolean flags in `setup.cfg`. Running

```
python -m build
```

will then create a source distribution (`sdist`) and a binary package (`wheel`) in the `dist/` directory. These files can be easily installed via Python's `pip`.

Important pre-conditions for build with fresh git repository

If you are about to create the Python interface from a fresh clone of our git repository, you require additional steps after running `./configure` as described *above*. In particular, some autogenerated static files that are compiled into RNAlib must be generated. To do so, run

```
cd src/ViennaRNA/static
make
cd ../../..
```

Additionally, if building the reference manual is not explicitly turned off, the Python interface requires docstrings to be generated. They are taken from the doxygen xml output which can be created by

```
cd doc/doxygen
make doxygen-xml
cd ../..
```

Finally, the swig wrapper must be build using

```
cd interfaces/Python
make RNA/RNA.py
cd ../..
```

After these steps, the Python `sdist` and `wheel` packages can be build as usual.

1.3 Installing Pre-compiled Packages

Linux

To install the pre-compiled Linux packages that we provide in the *Download* section above, simply download the corresponding files and use the installer program of your linux distribution.

Some examples to install the core programs and the Python 3 interface are shown below.

Arch Linux

```
sudo pacman -U viennarna-2.8.0-1-x86_64.pkg.tar.zst python-rna-2.8.0-1-x86_64.pkg.tar.
↪zst
```

Debian / Ubuntu

```
sudo apt install viennarna-2.8.0-1_amd64.deb python3-rna-2.8.0-1_amd64.deb
```

Fedora

```
sudo dnf install viennarna-2.8.0-1.x86_64.rpm python3-rna-2.8.0-1.x86_64.rpm
```

MacOS X

Simply download the latest MacOS X installer for your architecture from the download section above and run it. It will guide you through the installation process

Windows

Simply download the latest Windows installer for your architecture from the download section above and run it. It will guide you through the installation process

Pixi

`Pixi` is a modern and extremely fast alternative to `conda`. It is based on the conda packages and environment. First install pixi and then set-up the bioconda channel:

```
pixi config set default-channels '["conda-forge", "bioconda"]'
```

After these steps, you can install the ViennaRNA bioconda package either **globally**

```
pixi global install viennarna
```

or locally in any workspace/environment:

```
pixi add viennarna
```

See the pixi documentation for further information on how to get started and use `pixi` for your own workspace(s).

Conda

The ViennaRNA Package is also available for the `conda` or `mamba` package manager. The only requirement is to set up the bioconda channels

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
conda config --set channel_priority strict
```

and then you can easily install the viennarna bioconda package through

```
conda install viennarna
```

Python

The Python 3 interface for the ViennaRNA Package library is available at PyPI and can be installed independently using Python's `pip`:

```
python -m pip install viennarna
```

Julia

An unofficial interface of the ViennaRNA Package for the Julia Programming Language exists at JuliaHub.

1.4 Customizing the Build Process

The ViennaRNA Package includes additional executable programs such as

- `RNAforester`,
- `Kinfold`,
- `Kinwalker`,
- `RNAlocmin`, and
- `RNAxplorer`.

Furthermore, we include several features in our C-library that may be activated by default, or have to be explicitly turned on at configure-time. Below we list a selection of the available configure options that affect the features included in all executable programs, the *RNAlib* C-library, and the corresponding scripting language interface(s).

1.4.1 Streaming SIMD Extension

Since version 2.3.5 our sources contain code that implements a faster multibranch loop decomposition in global MFE predictions, as used e.g. in `RNAfold`. This implementation makes use of modern processors *streaming SIMD extension* (SSE) that provide the capability to execute particular instructions on multiple data simultaneously (*SIMD - single instruction multiple data*, thanks to W. B. Langdon for providing the modified code). Consequently, the time required to assess the minimum of all multibranch loop decompositions is reduced up to about one half compared to the runtime of the original implementation. This feature is enabled by default since version 2.4.11 and a dispatcher ensures that the correct implementation will be selected at runtime. If for any reason you want to disable this feature at compile-time use the following:

```
./configure --disable-simd
```

1.4.2 Scripting Language Interfaces

The ViennaRNA Package comes with scripting language interfaces for `Perl 5`, `Python` (provided by SWIG), that allow one to use the implemented algorithms directly without the need of calling an executable program. The necessary requirements are determined at configure-time and particular languages may be deactivated automatically if the requirements are not met.

You may switch-off particular languages by passing the `--without-perl` and/or `--without-python` configure options, e.g.:

```
./configure --without-perl
```

will turn-off building the `Perl 5` interface.

> **Tip**
>
> Disabling the scripting language support all-together can be accomplished using the following switch:
>
> ```
> ./configure --without-swig
> ```

1.4.3 Cluster Analysis

The programs `AnalyseSeqs` and `AnalyseDists` offer some cluster analysis tools (split decomposition, statistical geometry, neighbor joining, Ward's method) for sequences and distance data. To also build these programs add `--with-cluster` to your configure options.

1.4.4 Kinfold

The `kinfold` program can be used to simulate the folding dynamics of an RNA molecule, and is compiled by default. Use the `--without-kinfold` option to skip compilation and installation of Kinfold.

1.4.5 RNAforester

The `RNAforester` program is used for comparing secondary structures using tree alignment. Similar to `kinfold`, use the `--without-forester` option to skip compilation and installation of `RNAforester`.

1.4.6 Kinwalker

The `kinwalker` algorithm performs co-transcriptional folding of RNAs, starting at a user specified structure (default: open chain) and ending at the minimum free energy structure. Compilation and installation of this program is deactivated by default. Use the `--with-kinwalker` option to enable building and installation of `kinwalker`.

1.4.7 RNAlocmin

The `RNAlocmin` program is part of the **Basin Hopping Graph Framework** and reads secondary structures and searches for local minima by performing a gradient walk from each of those structures. It then outputs an energetically sorted list of local minima with their energies and number of hits to particular minimum, which corresponds to a size of a gradient basin. Additional output consists of barrier trees and Arhenius rates to compute various kinetic aspects. Compilation and installation of this program is activated by default. Use the `--without-rnalocmin` option to disable building and installation of `RNAlocmin`.

1.4.8 RNAxplorer

The `RNAxplorer` is a multitool, that offers different methods to explore RNA energy landscapes. In default mode it takes an RNA sequence as input and produces a sample of RNA secondary structures. The repellant sampling heuristic used in default mode iteratively penalizes base pairs of local minima of structures that have been seen too often. This results in a diverse sample set with the most important low free energy structures. Compilation and installation of this program is activated by default. Note, that this tool depends on the `LAPACK` library. Use the `--without-rnaxplorer` option to disable building and installation of `RNAxplorer`.

1.4.9 Link Time Optimization

To increase the performance of our implementations, the ViennaRNA Package tries to make use of the *Link Time Optimization* (LTO) feature of modern C-compilers. If you are experiencing any troubles at make-time or run-time, or the configure script for some reason detects that your compiler supports this feature although it doesn't, you can deactivate it using the flag:

```
./configure --disable-lto
```

Note, that `gcc` before version 5 is known to produce unreliable LTO code, especially in combination with *SIMD*. We therefore recommend using a more recent compiler (GCC 5 or above) or to turn off one of the two features, LTO or SIMD optimized code.

1.4.10 OpenMP

To enable concurrent computation of our implementations and in some cases parallelization of the algorithms we make use of the OpenMP API. This interface is well understood by most modern compilers. However, in some cases it might be necessary to deactivate OpenMP support and therefore transform *RNAlib* into a C-library that is not entirely *thread-safe*. To do so, add the following configure option:

```
./configure --disable-openmp
```

1.4.11 POSIX threads

To enable concurrent computation of multiple input data in `RNAfold`, and for our implementation of the concurrent unordered insert, ordered output flush data structure *`vrna_ostream_t`* we make use of POSIX threads (pthread). This should be supported on all modern platforms and usually does not pose any problems. Unfortunately, we use a threadpool implementation that is not compatible with Microsoft Windows yet. Thus, POSIX thread support can not be activated for Windows builds until we have fixed this problem. If you want to compile `RNAfold` and *RNAlib* without POSIX threads support for any other reasons, add the following configure option:

```
./configure --disable-pthreads
```

1.4.12 SVM Z-score filter

By default, `RNALfold` that comes with the ViennaRNA Package allows for Z-score filtering of its predicted results using a *Support Vector Machine* (SVM) provided by the LIBSVM library. However, this library is statically linked to our own *RNAlib*. If this introduces any problems for your own third-party programs that link against *RNAlib*, you can safely switch off the Z-scoring implementation using:

```
./configure --without-svm
```

1.4.13 GNU Scientific Library

The program `RNApvmin` computes a pseudo-energy perturbation vector that aims to minimize the discrepancy of predicted, and observed pairing probabilities. For that purpose it implements several methods to solve the optimization problem. Many of them are provided by the GNU Scientific Library (GSL), which is why the `RNApvmin` program, and the *RNAlib* C-library are required to be linked against `libgsl`. If this introduces any problems in your own third-party programs that link against *RNAlib*, you can turn off a larger portion of available minimizers in `RNApvmin` and linking against `libgsl` all-together, using:

```
./configure --without-gsl
```

1.4.14 Multiple-precision Floating-Point Computations

Our Non-redundant Boltzmann Sampling implementation uses multi-precision floating-point computations provided by the GNU MPFR library by default. This requires linking against `libmpfr` and `libgmp`. You can switch off this feature using:

```
./configure --disable-mpfr
```

1.4.15 Universal binaries

If you intend to build the ViennaRNA for Mac OS X such that it runs *on both*, `x86_64` and the `arm64` (Apple Silicon Processors) architectures, you need to build a so-called *universal binary*. Note, however, that to accomplish this task, you might need to deactivate any third-party library dependency as in most cases, only one architecture will be available at link time. This includes the `Perl 5` and `Python` interfaces but might also concern also *MPFR* and *GSL support*, possibly even more. In order to compile and link the programs, library, and scripting language interfaces of the ViennaRNA Package for multiple architectures, we've added a new configure switch that sets up the required changes automatically:

```
./configure --enable-universal-binary
```

Note

With link time optimization turned on, MacOS X's default compiler (`llvm/clang`) generates an intermediary binary format that can not easily be combined into a multi-architecture library. Therefore, the `--enable-universal-binary` switch turns off *Link Time Optimization*!

1.4.16 Disable C11/C++11 features

By default, we use C11/C++11 features in our implementations. This mainly accounts for unnamed unions/structs within *RNAlib*. The configure script automatically detects whether or not your compiler understands these features. In case you are using an older compiler, these features will be deactivated by setting a specific pre-processor directive. If for some reason you want to deactivate C11/C++11 features despite the capabilities of your compiler, use the following configure option:

```
./configure --disable-c11
```

1.4.17 Deprecated symbols

Since version 2.2 we are in the process of transforming the API of our *RNAlib*. Hence, several symbols are marked as *deprecated* whenever they have been replaced by the new API. By default, deprecation warnings at compile time are deactivated. If you want to get your terminal spammed by tons of deprecation warnings, enable them using:

```
./configure --enable-warn-deprecated
```

1.4.18 Single precision

Calculation of partition functions (via `RNAfold -p`) uses double precision floats by default, to avoid overflow errors on longer sequences. If your machine has little memory and you don't plan to fold sequences over 1,000 bases in length you can compile the package to do the computations in single precision by running:

```
./configure --enable-floatpf
```

> **Warning**
>
> Using this option is discouraged and not necessary on most modern computers.

1.4.19 Help

For a complete list of all `./configure` options and important environment variables, type:

```
./configure --help
```

For more general information on the build process see the *INSTALL* file.

1.5 Build your own Binary Package from SourceCode

If you don't find a precompiled package for your linux distribution, or in case you just want to create a binary package by yourself, have a look into the `packaging/` directory of our source tarball. It includes a list of some configuration files that we use for package generation:

- For RPM based distributions, we use `this SPEC file`, originally written for Fedora, but meanwhile extended to work for at least the Linux distributions we provide binary packages ourselves.
- Our binary package for ArchLinux was generated using `this PKGBUILD file`, which is also part of the `viennarna` source package available through the ArchLinux User Repository (AUR).
- The configuration files for our binary Debian and Ubuntu packages are included in the source tarball within the `packaging/debian` directory.
- Finally, for our MacOS X package, we make use of the `pkgbuild`, `productbuild`, and `hditool` commandline tools. The corresponding configuration files are included in our tarball in the directory `packaging/macosx`. For simplicity, you can instruct the `autotools` toolchain to automatically build the installer disk image by adding the following configure switch:

```
./configure --enable-macosx-installer
```

This creates a file `packaging/macosx/ViennaRNA-2.8.0-MacOSX.dmg`.

Our binary packages in the above section are mainly build using the openSUSE Build Service, while the Windows Installer is created using the MinGW cross-compiler and the NSIS install system. The MacOS X installer is build on github via github actions.

Part II

User Guides and Tutorials

CHAPTER

TWO

USER GUIDE

Our implementations can be accessed through stand-alone programs, such as `RNAfold`, `RNAdistance`, etc., which most often are sufficient for regular users. For those who wish to develop their own programs we provide `RNAlib`, a `C`-library that can be linked against your own programs or used in your `Python` or `Perl` scripts and pipelines. The latter can be easily accomplished through our *SWIG Wrappers* for `Python 3` and `Perl 5`, see also *Python Interface*.

2.1 Command-Line Tools

The ViennaRNA Package comes with a number of executable programs that provide command line interfaces to the most important algorithms implemented in *RNAlib*.

Find an overview of these programs and their corresponding manual pages below.

2.1.1 Main Programs

Program	Description
RNA2Dfold	Compute MFE structure, partition function and representative sample structures of k,l neighborhoods
RNAaliduplex	Predict conserved RNA-RNA interactions between two alignments
RNAalifold	Calculate secondary structures for a set of aligned RNA sequences
RNAcofold	Calculate secondary structures of two RNAs with dimerization
RNAdistance	Calculate distances between RNA secondary structures
RNAdos	Calculate the density of states for each energy band of an RNA
RNAduplex	Compute the structure upon hybridization of two RNA strands
RNAeval	Evaluate free energy of RNA sequences with given secondary structure
RNAfold	Calculate minimum free energy secondary structures and partition function of RNAs
RNAheat	Calculate the specific heat (melting curve) of an RNA sequence
RNAinverse	Find RNA sequences with given secondary structure (sequence design)
RNALalifold	Calculate locally stable secondary structures for a set of aligned RNAs
RNALfold	Calculate locally stable secondary structures of long RNAs
RNAmultifold	Compute thermodynamic properties for interaction complexes of multiple RNAs
RNApaln	RNA alignment based on sequence base pairing propensities
RNApdist	Calculate distances between thermodynamic RNA secondary structures ensembles
RNAparconv	Convert energy parameter files from ViennaRNA 1.8 to 2.0 format
RNAPKplex	Predict RNA secondary structures including pseudoknots
RNAplex	Find targets of a query RNA
RNAplfold	Calculate average pair probabilities for locally stable secondary structures
RNAplot	Draw RNA Secondary Structures in PostScript, SVG, or GML
RNApvmin	Calculate a perturbation vector that minimizes discrepancies between predicted and observed pairing probabilities
RNAsnoop	Find targets of a query H/ACA snoRNA
RNAsubopt	Calculate suboptimal secondary structures of RNAs
RNAup	Calculate the thermodynamics of RNA-RNA interactions

2.1.2 Additional Programs

We include the following additional programs in our distribution of the ViennaRNA Package. Whether or not they are installed together with the ViennaRNA Package depends on its /configuration.

Program	Description
AnalyseDists	Analyse a distance matrix
AnalyseSeqs	Analyse a set of sequences of common length
Kinfold	Simulate kinetic folding of RNA secondary structures
kinwalker	Predict RNA folding trajectories
RNAforester	Compare RNA secondary structures via forest alignment
RNAlocmin	Calculate local minima from structures via gradient walks
RNAxplorer	Explore the RNA conformation space through sampling and other techniques

2.1.3 Utilities

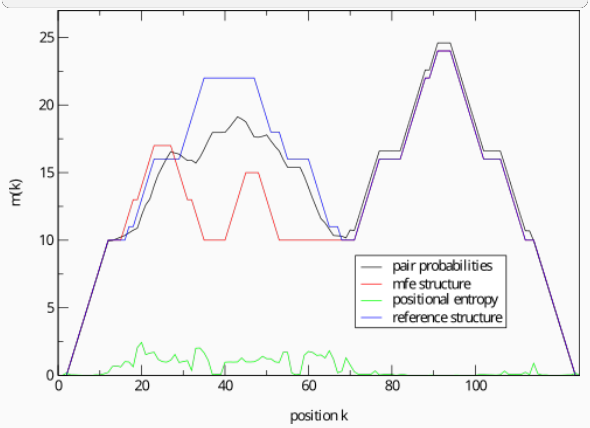
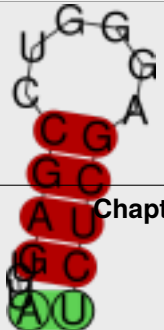
We also ship a number of small utilities, many of them to manipulate the PostScript files produced by the structure prediction programs *RNAfold* and *RNAalifold*.

Tip

Most of the *Perl 5* utilities contain embedded *pod* documentation. Type e.g.

```
perldoc relplot.pl
```

for detailed instructions.

Tool Name	Description
ct2db	Produce dot bracket notation of an RNA secondary structure given as mfold `.ct` file
b2mt.pl	Produce a mountain representation of a secondary structure from it's dot-bracket notation, as produced by *RNAfold*. Output consists of simple `x y` data suitable as input to a plotting program. The mountain representation is a xy plot with sequence position on the x-axis and the number of base pairs enclosing that position on the y-axis. Example: `RNAfold < my.seq | b2mt.pl | xmgrace -pipe` 
cmount.pl	Produce a PostScript mountain plot from a color dot plot as created by *RNAalifold* -p or `alidot -p`. Each base pair is represented by a trapez whose color encodes the number of consistent and compensatory mutations supporting that pair: Red marks pairs with no sequence variation; ochre, green, turquoise, blue, and violet mark pairs with 2,3,4,5,6 different types of pairs, respectively. Example: `cmount.pl alidot.ps > cmount.ps` 

2.2 Manpages

2.2.1 RNA2Dfold

RNA2Dfold - manual page for RNA2Dfold 2.8.0

Synopsis

```
RNA2Dfold [OPTION]...
```

DESCRIPTION

RNA2Dfold 2.8.0

Compute MFE structure, partition function and representative sample structures of k,l neighborhoods

The program partitions the secondary structure space into (basepair)distance classes according to two fixed reference structures. It expects a sequence and two secondary structures in dot-bracket notation as its inputs. For each distance class, the MFE representative, Boltzmann probabilities and Gibbs free energy is computed. Additionally, a stochastic backtracking routine allows one to produce samples of representative suboptimal secondary structures from each partition

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-j, **--numThreads**=INT

Set the number of threads used for calculations (only available when compiled with OpenMP support)

--noconv

Do not automatically substitute nucleotide “T” with “U”.

(default=off)

--log-level=level

Set log level threshold. *(default=”2”)*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default=”RNA2Dfold.log”)*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations. The Minimum free energy (MFE) and a structure representative are calculated in any case.

-p, **--partfunc**

calculate partition function and thus, Boltzmann probabilities and Gibbs free energy

(default=off)

--stochBT=INT

backtrack a certain number of Boltzmann samples from the appropriate k,l neighborhood(s)

--neighborhood=<k>:<l>

backtrack structures from certain k,l-neighborhood only, can be specified multiple times (<k>:<l>,<m>:<n>,...)

-K, **--maxDist1**=INT

maximum distance to first reference structure

If this value is set all structures that exhibit a basepair distance greater than maxDist1 will be thrown into a distance class denoted by K=L=-1

-L, **--maxDist2**=INT

maximum distance to second reference structure

If this value is set all structures that exhibit a basepair distance greater than maxDist1 will be thrown into a distance class denoted by K=L=-1

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

--noBT

do not backtrack structures, calculate energy contributions only

(default=off)

-c, **--circ**

Assume a circular (instead of linear) RNA molecule.

(default=off)

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops

(possible values="0", "2" default="2")

With `-d2` dangling energies will be added for the bases adjacent to a helix on both sides in any case. The option `-d0` ignores dangling ends altogether (mostly for debugging).

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

R. Lorenz, C. Flamm, I.L. Hofacker (2009), "2D Projections of RNA folding Landscapes", GI, Lecture Notes in Informatics, German Conference on Bioinformatics 2009: 157, pp 11-20

M. Zuker, P. Stiegler (1981), "Optimal computer folding of large RNA sequences using thermodynamic and auxiliary information", Nucl Acid Res: 9, pp 133-148

J.S. McCaskill (1990), "The equilibrium partition function and base pair binding probabilities for RNA secondary structures", Biopolymers: 29, pp 1105-1119

I.L. Hofacker and P.F. Stadler (2006), "Memory Efficient Folding Algorithms for Circular RNA Secondary Structures", Bioinformatics

D. Adams (1979), "The hitchhiker's guide to the galaxy", Pan Books, London

The calculation of mfe structures is based on dynamic programming algorithm originally developed by M. Zuker and P. Stiegler. The partition function algorithm is based on work by J.S. McCaskill.

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.2 RNAaliduplex

RNAaliduplex - manual page for RNAaliduplex 2.8.0

Synopsis

```
RNAaliduplex [options] <file1.aln> <file2.aln>
```

DESCRIPTION

RNAaliduplex 2.8.0

Predict conserved RNA-RNA interactions between two alignments

The program reads two alignments of RNA sequences in CLUSTAL format and predicts optimal and suboptimal binding sites, hybridization energies and the corresponding structures. The calculation takes only inter-molecular base pairs into account, for the general case use RNAcofold. The use of alignments allows one to focus on binding sites that are evolutionary conserved. Note, that the two input alignments need to have equal number of sequences and the same order, i.e. the 1st sequence in file1 will be hybridized to the 1st in file2 etc.

The computed binding sites, energies, and structures are written to stdout, one structure per line. Each line consist of: The structure in dot bracket format with a "&" separating the two strands. The range of the structure in the two sequences in the format "from,to : from,to"; the energy of duplex structure in kcal/mol. The format is especially useful for computing the hybrid structure between a small probe sequence and a long target sequence.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

Algorithms:

Select additional algorithms which should be included in the calculations.

-e, **--deltaEnergy**=range

Compute suboptimal structures with energy in a certain range of the optimum (kcal/mol). Default is calculation of mfe structure only.

-s, **--sorted**

Sort output by free energy.

(default=off)

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

--saltInit=DOUBLE

Provide salt correction for duplex initialization (in kcal/mol).

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for

debugging). With -d3 mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with -d1 and -d3 only the MFE computations will be using this setting while partition function uses -d2 setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp=*"-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAaliduplex.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

SEE ALSO

RNAduplex(1) RNAcofold(1) RNAfold(1)

2.2.3 RNAalifold

RNAalifold - manual page for RNAalifold 2.8.0

Synopsis

```
RNAalifold [options] [<input0.aln>] [<input1.aln>]...
```

DESCRIPTION

RNAalifold 2.8.0

calculate secondary structures for a set of aligned RNAs

Read aligned RNA sequences from stdin or file.aln and calculate their minimum free energy (mfe) structure, partition function (pf) and base pairing probability matrix. Currently, input alignments have to be in CLUSTAL, Stockholm, FASTA, or MAF format. The input format must be set manually in interactive mode (default is Clustal), but will be determined automagically from the input file, if not expplicitly set. It returns the mfe structure in bracket notation, its energy, the free energy of the thermodynamic ensemble and the frequency of the mfe structure in the ensemble to stdout. It also produces Postscript files with plots of the resulting secondary structure graph ("alirna.ps") and a "dot plot" of the base pairing matrix ("alidot.ps"). The file "alifold.out" will contain a list of likely pairs sorted by credibility, suitable for viewing with "AliDot.pl". Be warned that output file will overwrite any existing files of the same name.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

-q, **--quiet**

Be quiet. *(default=off)*

This option can be used to minimize the output of additional information and non-severe warnings which otherwise might spam stdout/stderr.

I/O Options:

Command line options for input and output (pre-)processing

-f, **--input-format**=C|S|F|M

File format of the input multiple sequence alignment (MSA).

If this parameter is set, the input is considered to be in a particular file format. Otherwise, the program tries to determine the file format automatically, if an input file was provided in the set of parameters. In case the input MSA is provided in interactive mode, or from a terminal (TTY), the programs default is to assume CLUSTALW format. Currently, the following formats are available: ClustalW (`C`), Stockholm 1.0 (`S`), FASTA/Pearson (`F`), and MAF (`M`).

--mis

Output "most informative sequence" instead of simple consensus: For each column of the alignment output the set of nucleotides with frequency greater than average in IUPAC notation.

(default=off)

-j, **--jobs**[=number]

Split batch input into jobs and start processing in parallel using multiple threads. A value of 0 indicates to use as many parallel threads as computation cores are available.

(default="0")

Default processing of input data is performed in a serial fashion, i.e. one alignment at a time. Using this switch, a user can instead start the computation for many alignments in the input in parallel. RNAalifold will create as many parallel computation slots as specified and assigns input alignments of the input file(s) to the available slots. Note, that this increases memory consumption since input alignments have to be kept in memory until an empty compute slot is available and each running job requires its own dynamic programming matrices.

--unordered

Do not try to keep output in order with input while parallel processing is in place.

(default=off)

When parallel input processing (*--jobs* flag) is enabled, the order in which input is processed depends on the host machines job scheduler. Therefore, any output to stdout or files generated by this program will most likely not follow the order of the corresponding input data set. The default of RNAalifold is to use a specialized data structure to still keep the results output in order with the input data. However, this comes with a trade-off in terms of memory consumption, since all output must be kept in memory for as long as no chunks of consecutive, ordered output are available. By setting this flag, RNAalifold will not buffer individual results but print them as soon as they have been computated.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

-n, **--continuous-ids**

Use continuous alignment ID numbering when no alignment ID can be retrieved from input data.

(default=off)

Due to its past, RNAalifold produces a specific set of output file names for the first input alignment, "alirna.ps", "alidot.ps", etc. But for all further alignments in the input, it usually adopts a naming scheme based on IDs, which may be retrieved from the input alignment's meta-data, or generated by a prefix followed by an increasing counter. Setting this flag instructs RNAalifold to use the ID naming scheme also for the first alignment.

--auto-id

Automatically generate an ID for each alignment.

(default=off)

The default mode of RNAalifold is to automatically determine an ID from the input alignment if the input file format allows to do that. Alignment IDs are, for instance, usually given in Stockholm 1.0 formatted input. If this flag is active, RNAalifold ignores any IDs retrieved from the input and automatically generates an ID for each alignment.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="alignment")

If this parameter is set, each alignment will be prefixed with the provided string. Hence, the output files will obey the following naming scheme: "prefix_xxxx_ss.ps" (secondary structure plot), "prefix_xxxx_dp.ps" (dot-plot), "prefix_xxxx_aln.ps" (annotated alignment), etc. where xxxx is the alignment number beginning with the second alignment in the input. Use this setting in conjunction with the *--continuous-ids* flag to assign IDs beginning with the first input alignment.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter _ between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18].

--id-start=LONG

Specify the first number in automatically generated alignment IDs.

(default="1")

When alignment IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies continuous alignment IDs, i.e. it activates the *--continuous-ids* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAalifold.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

--benchmark

Perform a prediction performance test (benchmark) based on reference structures taken from the input.

(default=off)

--bm-output=filename

Store the performance benchmark results in a file.

--bm-output-append

Append performance benchmark data to a file instead of overwriting already existing content.

(default=off)

--bm-rm-pk

Remove pseudo-knots from the reference structure before computing benchmark.

(default=off)

This option adds a new output column "IR" to keep track of the number of reference structure base pairs that have been ignored for the benchmark.

--bm-ignore-compat

Ignore predicted base pairs compatible with the reference strucure.

(default=off)

Predicted base pairs that are absent from the reference but do not structurally contradict it (i.e. do not cross any reference base pair) are excluded from the false positive count and reclassified as true negatives for these relaxed metrics.

This option adds a new output column "IP" to keep track of the number of

predicted base pairs that have been ignored for the benchmark.

--benchmark-fuzzy=INT

Allow base pair slippage when comparing the predicted structure to the reference.

(default="0")

A predicted base pair (p,q) is counted as a true positive if it lies within +/- this many nucleotides of a reference base pair (i,j). Each reference base pair can only be matched by one predicted base pair; once claimed it can't be counted again by another predicted pair.

Algorithms:

Select additional algorithms which should be included in the calculations.

-p, **--partfunc**[=INT]

Calculate the partition function and base pairing probability matrix in addition to the mfe structure. Default is calculation of mfe structure only.

(default="1")

In addition to the MFE structure we print a coarse representation of the pair probabilities in form of a pseudo bracket notation, followed by the ensemble free energy, as well as the centroid structure derived from the pair probabilities together with its free energy and distance to the ensemble. Finally it prints the frequency of the mfe structure.

An additionally passed value to this option changes the behavior of partition function calculation: `-p0` deactivates the calculation of the pair probabilities, saving about 50% in runtime. This prints the ensemble free energy `dG=-kT ln(Z)`.

--betaScale=DOUBLE

Set the scaling of the Boltzmann factors. *(default="1.")*

The argument provided with this option is used to scale the thermodynamic temperature in the Boltzmann factors independently from the temperature of the individual loop energy contributions. The Boltzmann factors then become `exp(- dG/(kTn*betaScale))` where `k` is the Boltzmann constant, `dG` the free energy contribution of the state, `T` the absolute temperature and `n` the number of sequences.

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

--MEA[=gamma]

Compute MEA (maximum expected accuracy) structure.

(default="1.")

The expected accuracy is computed from the pair probabilities: each base pair `(i,j)` receives a score `2*gamma*p_ij` and the score of an unpaired base is given by the probability of not forming a pair. The parameter gamma tunes the importance of correctly predicted pairs versus unpaired bases. Thus, for small values of gamma the MEA structure will contain only pairs with very high probability. Using *--MEA* implies *-p* for computing the pair probabilities.

--sci

Compute the structure conservation index (SCI) for the MFE consensus structure of the alignment.

(default=off)

-c, **--circ**

Assume a circular (instead of linear) RNA molecule.

(default=off)

--bppmThreshold=cutoff

Set the threshold/cutoff for base pair probabilities included in the postscript output.

(default="1e-6")

By setting the threshold the base pair probabilities that are included in the output can be varied. By default only those exceeding `1e-6` in probability will be shown as squares in the dot plot. Changing the threshold to any other value allows for increase or decrease of data.

-g, **--gquad**

Incoorporate G-Quadruplex formation into the structure prediction algorithm.

(default=off)

-s, **--stochBT**=INT

Stochastic backtrack. Compute a certain number of random structures with a probability dependend on the partition function. See *-p* option in RNAsubopt.

--stochBT_en=INT

same as *-s* option but also print out the energies and probabilities of the backtraced structures.

-N, **--nonRedundant**

Enable non-redundant sampling strategy.

(default=off)

--random-seed=INT

Set the seed for the random number generator

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--maxBPspan=INT

Set the maximum base pair span.

(default="-1")

-C, **--constraint**[=filename]

Calculate structures subject to constraints. The constraining structure will be read from `stdin`, the alignment has to be given as a file name on the command line.

(default="")

The program reads first the sequence, then a string containing constraints on the structure encoded with the symbols:

`.` (no constraint for this base)

`|` (the corresponding base has to be paired

`x` (the base is unpaired)

< (base i is paired with a base j>i)

> (base i is paired with a base j<i)

and matching brackets `( )` (base i pairs base j)

With the exception of |, constraints will disallow all pairs conflicting with the constraint. This is usually sufficient to enforce the constraint, but occasionally a base may stay unpaired in spite of constraints. PF folding ignores constraints of type |.

--batch

Use constraints for all alignment records. *(default=off)*

Usually, constraints provided from input file are only applied to a single sequence alignment. Therefore, RNAalifold will stop its computation and quit after the first input alignment was processed. Using this switch, RNAalifold processes all sequence alignments in the input and applies the same provided constraints to each of them.

--enforceConstraint

Enforce base pairs given by round brackets ( ) in structure constraint.

(default=off)

--SS_cons

Use consensus structures from Stockholm file (#=GF SS_cons) as constraint.

(default=off)

Stockholm formatted alignment files have the possibility to store a secondary structure string in one of if (#=GC) column annotation meta tags. The corresponding tag name is usually SS_cons, a consensus secondary structure. Activating this flag allows one to use this consensus secondary structure from the input file as structure constraint. Currently, only the following characters are interpreted:

( ) [mathing parenthesis: column i pairs with column j]

< > [matching angular brackets: column i pairs with column j]

All other characters are not interpreted (yet). Note: Activating this flag implies *--constraint*.

--shape=file1,file2

Use SHAPE reactivity data to guide structure predictions.

Multiple shapefiles for the individual sequences in the alignment may be specified as a comma separated list. An optional association of particular shape files to a specific sequence in the alignment can be expressed by prepending the sequence number to the filename, e.g. "5=seq5.shape,3=seq3.shape" will assign the reactivity values from file seq5.shape to the fifth sequence in the alignment, and the values from file seq3.shape to sequence 3. If no assignment is specified, the reactivity values are assigned to corresponding sequences in the order they are given.

--shapeMethod=D[mX][bY]

Specify the method how to convert SHAPE reactivity data to pseudo energy contributions.

(default="D")

Currently, the only data conversion method available is that of to Deigan et al 2009. This method is the default and is recognized by a capital D in the provided parameter, i.e.: *--shapeMethod=*"D" is the default setting. The slope m and the intercept b can be set to a non-default value if necessary. Otherwise m=1.8 and b=-0.6 as stated in the paper mentionen before. To alter these parameters, e.g. m=1.9 and b=-0.7, use a parameter string like this: *--shapeMethod=*"Dm1.9b-0.7". You may also provide only one of the two parameters like: *--shapeMethod=*"Dm1.9" or *--shapeMethod=*"Db-0.7".

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d2` dangling energies will be added for the bases adjacent to a helix on both sides

in any case.

The option `-d0` ignores dangling ends altogether (mostly for debugging).

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--cfactor=DOUBLE

Set the weight of the covariance term in the energy function

(default="1.0")

--nfactor=DOUBLE

Set the penalty for non-compatible sequences in the covariance term of the energy function

(default="1.0")

-E, **--endgaps**

Score pairs with endgaps same as gap-gap pairs.

(default=off)

-R, **--ribosum_file**=ribosumfile

use specified Ribosum Matrix instead of normal

energy model.

Matrixes to use should be 6x6 matrices, the order of the terms is `AU`, `CG`, `GC`, `GU`, `UA`, `UG`.

-r, **--ribosum_scoring**

use ribosum scoring matrix. *(default=off)*

The matrix is chosen according to the minimal and maximal pairwise identities of the sequences in the file.

--old

use old energy evaluation, treating gaps as characters.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

Plotting:

Command line options for changing the default behavior of structure layout and pairing probability plots

--color

Produce a colored version of the consensus structure plot "alirna.ps" (default b&w only)

(default=off)

--color-threshold=FLOAT

Set the threshold of maximum counter examples for coloring consensus structure plot.

(default="2")

Floating point numbers between 0 and 1 are treated as frequencies among all sequencesin the alignment. All other will be truncated to integer and used as absolute number of counter examples.

--color-min-sat=FLOAT

Set the minimum saturation for coloring consensus structure plot.

(default="0.2")

Floating point number >= 0 and smaller than 1.

--aln

Produce a colored and structure annotated alignment in PostScript format in the file "aln.ps" in the current directory.

(default=off)

--aln-EPS-cols=INT

Number of columns in colored EPS alignment output.

(default="60")

A value less than 1 indicates that the output should not be wrapped at all.

--aln-stk[=prefix]

Create a multi-Stockholm formatted output file. *(default="RNAalifold_results")*

The default file name used for the output is "RNAalifold_results.stk". Users may change the filename to "prefix.stk" by specifying the prefix as optional argument. The file will be create in the current directory if it does not already exist. In case the file already exists, output will be appended to it. Note: Any special characters in the filename will be replaced by the filename delimiter, hence there is no way to pass an entire directory path through this option yet. (See also the "–filename-delim" parameter)

--noPS

Do not produce postscript drawing of the mfe structure.

(default=off)

--noDP

Do not produce dot-plot postscript file containing base pair or stack probabilitities.

(default=off)

In combination with the *-p* option, this flag turns-off creation of individual dot-plot files. Consequently, computed base pair probability output is omitted but centroid and MEA structure prediction is still performed.

-t, **--layout-type**=INT

Choose the layout algorithm. *(default="1")*

Select the layout algorithm that computes the nucleotide coordinates. Currently, the following algorithms are available:

0: simple radial layout

1: Naview layout (Bruccoleri et al. 1988)

2: circular layout

3: RNAturtle (Wiegreffe et al. 2018)

4: RNApuzzler (Wiegreffe et al. 2018)

Caveats:

Sequences are not weighted. If possible, do not mix very similar and dissimilar sequences. Duplicate sequences, for example, can distort the prediction.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The algorithm is a variant of the dynamic programming algorithms of M. Zuker and P. Stiegler (mfe) and J.S. McCaskill (pf) adapted for sets of aligned sequences with covariance information.

Ivo L. Hofacker, Martin Fekete, and Peter F. Stadler (2002), "Secondary Structure Prediction for Aligned RNA Sequences", J.Mol.Biol.: 319, pp 1059-1066.

Stephan H. Bernhart, Ivo L. Hofacker, Sebastian Will, Andreas R. Gruber, and Peter F. Stadler (2008), "RNAalifold: Improved consensus structure prediction for RNA alignments", BMC Bioinformatics: 9, pp 474

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

EXAMPLES

A simple call to compute consensus MFE structure, ensemble free energy, base pair probabilities, centroid structure, and MEA structure for a multiple sequence alignment (MSA) provided as Stockholm formatted file alignment.stk might look like:

```
$ RNAalifold -p --MEA alignment.stk
```

Consider the following MSA file for three sequences

```
# STOCKHOLM 1.0

#=GF AC   RF01293
#=GF ID   ACA59
#=GF DE   Small nucleolar RNA ACA59
#=GF AU   Wilkinson A
#=GF SE   Predicted; WAR; Wilkinson A
#=GF SS   Predicted; WAR; Wilkinson A
#=GF GA   43.00
#=GF TC   44.90
#=GF NC   40.30
#=GF TP   Gene; snRNA; snoRNA; HACA-box;
#=GF BM   cmbuild -F CM SEED
```

(continues on next page)

(continued from previous page)

```
#=GF CB   cmcalibrate --mpi CM
#=GF SM   cmsearch --cpu 4 --verbose --nohmmonly -E 1000 -Z 549862.597050 CM SEQDB
#=GF DR   snoRNABase; ACA59;
#=GF DR   SO; 0001263; ncRNA_gene;
#=GF DR   GO; 0006396; RNA processing;
#=GF DR   GO; 0005730; nucleolus;
#=GF RN   [1]
#=GF RM   15199136
#=GF RT   Human box H/ACA pseudouridylation guide RNA machinery.
#=GF RA   Kiss AM, Jady BE, Bertrand E, Kiss T
#=GF RL   Mol Cell Biol. 2004;24:5797-5807.
#=GF WK   Small_nucleolar_RNA
#=GF SQ   3


AL031296.1/85969-86120     ␣
↪CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGUAACAAUACUUACUCUCGUUGGUGAUAAGGAACAGCU
AANU01225121.1/438-603     ␣
↪CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGUGACAAUACUUACUCUCGUUGGUGAUAAGGAACAGCU
AAWR02037329.1/29294-29150 ---CUCGACACCACU---
↪GCCUCGGUUACCCAUCGGUGCAGUGCGGGUAGUAGUACCAAUGCUAAUUAGUUGUGAGGACCAACU
#=GC SS_cons               -----((((,<<<<<<<<<_____________>>>>>>>>>,,,,<<<<<<<_______>>>
↪>>>>,,,,,))))::::::::::::
#=GC RF                    ␣
↪CUGCcccaCAaCacuuguGCCUCaGUUACcCauagguGuAGUGaGgGuggcAaUACccaCcCucgUUgGuggUaAGGAaCAgCU
//
```

Then, the above program call will produce this output:

```
3 sequences; length of alignment 84.
>ACA59
CUGCCUCACAACAUUUGUGCCUCAGUUACCCAUAGAUGUAGUGAGGGUAACAAUACUUACUCUCGUUGGUGAUAAGGAACAGCU
...((((((.(((((((((...........))))))))).))))))..........(((((......)))))............␣
↪(-12.54 = -12.77 +   0.23)
...((((((.(((((((((...........))))))))).)))))){{,.......{{{{,......}}}}}............␣
↪[-14.38]
...((((((.(((((((((...........))))))))).))))))..........((((........))))............
↪{-12.44 = -12.33 +  -0.10 d=10.94}
...((((((.(((((((((...........))))))))).))))))..........((((........))))............
↪{-12.44 = -12.33 +  -0.10 MEA=66.65}
frequency of mfe structure in ensemble 0.368739; ensemble diversity 17.77
```

Here, the first line is written to *stderr* and simply states the number of sequences and the length of the alignment. This line can be suppressed using the `--quiet` option. The main output then consists of 7 lines, where the first two resemble the FASTA header with the ID as read from the input data set, followed by the consensus sequence in the second line. The third line consists of the consensus secondary structure in dot-bracket notation followed by the averaged minimum free energy in parenthesis. This energy is composed of two major contributions, the actual free energies derived from the Nearest Neighbor model, and the covariance pseudo-energy term, which are both displayed after the equal sign. The fourth line shows the base pair propensity in pseudo dot-bracket notation followed by the ensemble free energy dG = -kT ln(Z) in square brackets. Similarly, the next two lines state the controid- and the MEA structure in dot-bracket notation, followed by their corresponding free energy contributions, the mean distance (d) to the ensemble as well as the maximum expected accuracy (MEA). Again, the free energies are split into Nearest Neighbor contribution and the covariance pseudo-energy term.

Furthermore, RNAalifold will produce three output files: ACA59_ss.ps, ACA59_dp.ps, and ACA59_ali.out that contain the secondary structure drawing, the base pair probability dot-plot, and a detailed table of base pair probabilities, respectively.

THE ALIOUT FILE

When computing base pair probabilities (*--partfunc* option), RNAalifold will produce a file with the suffix *ali.out*. This file contains the base pairing probabilities between different alignment columns together with some detailed statistics for the individual sequences within the alignment. The file is a simple text file with a two line header that states the number of sequences and length of the alignment. The first couple of lines of this file may look like:

```
3 sequence; length of alignment 84
alifold output
14    36  0  92.7%   0.212 CG:1    UA:2
13    37  0  92.7%   0.218 GU:1    AU:2
12    38  0  92.7%   0.231 CG:3
15    35  0  91.9%   0.239 UG:3
16    34  0  85.2%   0.434 UA:2    --:1
8    42  0  80.7%   0.526 AU:3   +
9    41  0  80.4%   0.542 CG:3   +
7    43  1  80.1%   0.541 CG:2   +
```

Starting with the third row, there are at least six and at most 13 columns separated by whitespaces stating: (1) the i-position and (2) the j-position of a potential base pair (i, j), followed by (3) the number of counter examples, i.e. the number of sequences in the alignment that can't form a canonical base pair with their respective sequence positions. Next is (4) the base pair probabilitiy in percent, (5) a pseudo entropy measure S_ij = S_i + S_j - p_ij ln(p_ij), where S_i and S_j are the positional entropies for the two alignment columns i and j, and p_ij is the base pair probability. Finally, the last columns (6-12) state the number of particular base pairs for the individual sequences in the alignment. Here, we distinguish the base pairs "GC","CG","AU","UA","GU","UG", and the special case "–" that represents gaps at both positions i and j. Finally, base pairs that are not part of the MFE structure are marked by an additional "+" sign in the last column.

AUTHOR

Ivo L Hofacker, Stephan Bernhart, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

SEE ALSO

The ALIDOT package http://www.tbi.univie.ac.at/RNA/Alidot/

2.2.4 RNAcofold

RNAcofold - manual page for RNAcofold 2.8.0

Synopsis

```
RNAcofold [OPTION]... [FILE]...
```

DESCRIPTION

RNAcofold 2.8.0

calculate secondary structures of two RNAs with dimerization

The program works much like RNAfold, but allows one to specify two RNA sequences which are then allowed to form a dimer structure. RNA sequences are read from stdin in the usual format, i.e. each line of input corresponds to one sequence, except for lines starting with > which contain the name of the next sequence. To compute the hybrid structure of two molecules, the two sequences must be concatenated using the & character as separator. RNAcofold can compute minimum free energy (mfe) structures, as well as partition function (pf) and base pairing probability matrix (using the *-p* switch) Since dimer formation is concentration dependent, RNAcofold can be

used to compute equilibrium concentrations for all five monomer and (homo/hetero)-dimer species, given input concentrations for the monomers. Output consists of the mfe structure in bracket notation as well as PostScript structure plots and "dot plot" files containing the pair probabilities, see the RNAfold man page for details. In the dot plots a cross marks the chain break between the two concatenated sequences. The program will continue to read new sequences until a line consisting of the single character `@` or an end of file condition is encountered.

`-v`, **`--verbose`**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

`--output-format`=format-character

Change the default output format.

(default="V")

The following output formats are currently supported:

ViennaRNA format (`V`), Delimiter-separated format (`D`) also known as `CSV`

format.

`--csv-delim`=delimiter

Change the delimiting character for Delimiter-separated output format, such as `CSV`.

(default=",")

Delimiter-separated output defaults to comma separated values (`CSV`), i.e. all data in one data set is delimited by a comma character. This option allows one to change the delimiting character to something else. Note, to switch to tab-separated data, use `$'\t'` as delimiting character.

`--csv-noheader`

Do not print header for Delimiter-separated output, such as CSV.

(default=off)

`-j`, **`--jobs`**[=number]

Split batch input into jobs and start processing in parallel using multiple threads. A value of 0 indicates to use as many parallel threads as computation cores are available.

(default="0")

Default processing of input data is performed in a serial fashion, i.e. one sequence pair at a time. Using this switch, a user can instead start the computation for many sequence pairs in the input in parallel. RNAcofold will create as many parallel computation slots as specified and assigns input sequences of the input file(s) to the available slots. Note, that this increases memory consumption since input alignments have to be kept in memory until an empty compute slot is available and each running job requires its own dynamic programming matrices.

`--unordered`

Do not try to keep output in order with input while parallel processing is in place.

(default=off)

When parallel input processing (*`--jobs`* flag) is enabled, the order in which input is processed depends on the host machines job scheduler. Therefore, any output to stdout or files generated by this program will most likely not follow the order of the corresponding input data set. The default of RNAcofold is to use a specialized data structure to still keep the results output in order with the input data. However, this comes with a trade-off in terms of memory consumption, since all output must be kept in memory for as long as no chunks of consecutive, ordered output are available. By setting this flag, RNAcofold will not buffer individual results but print them as soon as they have been computated.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNAcofold is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNAcofold ignores any IDs retrieved from the input and automatically generates an ID for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="sequence")

If this parameter is set, each sequence will be prefixed with the provided string. Hence, the output files will obey the following naming scheme: "prefix_xxxx_ss.ps" (secondary structure plot), "prefix_xxxx_dp.ps" (dot-plot), "prefix_xxxx_dp2.ps" (stack probabilities), etc. where xxxx is the sequence number. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--filename-full

Use full FASTA header to create filenames. *(default=off)*

This parameter can be used to deactivate the default behavior of limiting output filenames to the first word of the sequence ID. Consider the following example: An input with FASTA header >NM_0001 Homo Sapiens some gene usually produces output files with the prefix "NM_0001" without the additional data available in the FASTA header, e.g. "NM_0001_ss.ps" for secondary structure plots. With this flag set, no truncation of the output filenames is done, i.e. output filenames receive the full FASTA header data as prefixes. Note, however, that invalid characters (such as whitespace) will be substituted by a delimiting character or simply removed, (see also the parameter option *--filename-delim*).

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAcofold.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations. The Minimum free energy (MFE) and a structure representative are calculated in any case.

-p, **--partfunc**[=INT]

Calculate the partition function and base pairing probability matrix in addition to the mfe structure. Default is calculation of mfe structure only.

(default="1")

In addition to the MFE structure we print a coarse representation of the pair probabilities in form of a pseudo bracket notation, followed by the ensemble free energy, as well as the centroid structure derived from the pair probabilities together with its free energy and distance to the ensemble. Finally it prints the frequency of the mfe structure, and the structural diversity (mean distance between the structures in the ensemble). See the description of pf_fold() and mean_bp_dist() and centroid() in the RNAlib documentation for details. Note that unless you also specify **-d2** or **-d0**, the partition function and mfe calculations will use a slightly different energy model. See the discussion of dangling end options below.

An additionally passed value to this option changes the behavior of partition function calculation:

In order to calculate the partition function but not the pair probabilities

use the **-p0** option and save about

50% in runtime. This prints the ensemble free energy dG=-kT ln(Z).

-a, **--all_pf**[=INT]

Compute the partition function and free energies not only of the hetero-dimer consisting of the two input sequences (the `AB dimer`), but also of the homo-dimers AA and BB as well as A and B monomers.

(default="1")

The output will contain the free energies for each of these species, as well as 5 dot plots containing the conditional pair probabilities, called "ABname5.ps", "AAname5.ps" and so on. For later use, these dot plot files also contain the free energy of the ensemble as a comment. Using *-a* automatically switches on the *-p* option. Base pair probability computations may be turned off altogether by providing `0` as an argument to this parameter. In that case, no dot plot files will be generated.

--betaScale=DOUBLE

Set the scaling of the Boltzmann factors. *(default="1.")*

The argument provided with this option is used to scale the thermodynamic temperature in the Boltzmann factors independently from the temperature of the individual loop energy contributions. The Boltzmann factors then become `exp(- dG/(kT*betaScale))` where `k` is the Boltzmann constant, `dG` the free energy contribution of the state and `T` the absolute temperature.

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

-c, **--concentrations**

In addition to everything listed under the *-a* option, read in initial monomer concentrations and compute the expected equilibrium concentrations of the 5 possible species (AB, AA, BB, A, B).

(default=off)

Start concentrations are read from stdin (unless the *-f* option is used) in [mol/l], equilibrium concentrations are given realtive to the sum of the two inputs. An arbitrary number of initial concentrations can be specified (one pair of concentrations per line).

-f, **--concfile**=filename

Specify a file with initial concentrations for the two sequences.

The table consits of arbitrary many lines with just two numbers (the concentration of sequence A and B). This option will automatically toggle the *-c* (and thus *-a* and *-p*) options (see above).

--centroid

Compute the centroid structure. *(default=off)*

Additionally to the MFE structure, compute the centroid representative of the structure ensemble. Here, we apply the base pair distance as distance measure, and report the structure that minimizes its Boltzmann weighted base pair distance to the rest of the ensemble. Computing the centroid structure requires equilibrium base pair probabilities. Therefore, this option implies the *-p* switch. For historical reasons, the centroid structure output is deactivated by default.

--MEA[=gamma]

Compute MEA (maximum expected accuracy) structure.

(default="1.")

The expected accuracy is computed from the pair probabilities: each base pair `(i,j)` receives a score `2*gamma*p_ij` and the score of an unpaired base is given by the probability of not forming a pair. The parameter gamma tunes the importance of correctly predicted pairs versus unpaired bases. Thus, for small values of gamma the MEA structure will contain only pairs with very high probability. Using *--MEA* implies *-p* for computing the pair probabilities.

--bppmThreshold=cutoff

Set the threshold/cutoff for base pair probabilities included in the postscript output.

(default="1e-5")

By setting the threshold the base pair probabilities that are included in the output can be varied. By default only those exceeding `1e-5` in probability will be shown as squares in the dot plot. Changing the threshold to any other value allows for increase or decrease of data.

-g, **--gquad**

Incoorporate G-Quadruplex formation into the structure prediction algorithm.

(default=off)

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--maxBPspan=INT

Set the maximum base pair span.

(default="-1")

-C, **--constraint**[=filename]

Calculate structures subject to constraints. *(default="")*

The program reads first the sequence, then a string containing constraints on the structure encoded with the symbols:

`.` (no constraint for this base)

`|` (the corresponding base has to be paired

`x` (the base is unpaired)

< (base i is paired with a base j>i)

> (base i is paired with a base j<i)

and matching brackets `( )` (base i pairs base j)

With the exception of `|`, constraints will disallow all pairs conflicting with the constraint. This is usually sufficient to enforce the constraint, but occasionally a base may stay unpaired in spite of constraints. PF folding ignores constraints of type `|`.

--batch

Use constraints for multiple sequences. *(default=off)*

Usually, constraints provided from input file only apply to a single input sequence. Therefore, RNAcofold will stop its computation and quit after the first input sequence was processed. Using this switch, RNAcofold processes multiple input sequences and applies the same provided constraints to each of them.

--canonicalBPonly

Remove non-canonical base pairs from the structure constraint.

(default=off)

--enforceConstraint

Enforce base pairs given by round brackets `( )` in structure constraint.

(default=off)

--shape=filename

Use SHAPE reactivity data to guide structure predictions.

--shapeMethod=method

Select SHAPE reactivity data incorporation strategy.

(default="D")

The following methods can be used to convert SHAPE reactivities into pseudo energy contributions.

D: Convert by using the linear equation according to Deigan et al 2009.

Derived pseudo energy terms will be applied for every nucleotide involved in a stacked pair. This method is recognized by a capital D in the provided parameter, i.e.: *--shapeMethod=*"D" is the default setting. The slope m and the intercept b can be set to a non-default value if necessary, otherwise m=1.8 and b=-0.6. To alter these parameters, e.g. m=1.9 and b=-0.7, use a parameter string like this: *--shapeMethod=*"Dm1.9b-0.7". You may also provide only one of the two parameters like: *--shapeMethod=*"Dm1.9" or *--shapeMethod=*"Db-0.7".

Z: Convert SHAPE reactivities to pseudo energies according to Zarringhalam

et al 2012. SHAPE reactivities will be converted to pairing probabilities by using linear mapping. Aberration from the observed pairing probabilities will be penalized during the folding recursion. The magnitude of the penalties can affected by adjusting the factor beta (e.g. *--shapeMethod=*"Zb0.8").

W: Apply a given vector of perturbation energies to unpaired nucleotides

according to Washietl et al 2012. Perturbation vectors can be calculated by using RNApvmin.

--shapeConversion=method

Select method for SHAPE reactivity conversion.

(default="O")

This parameter is useful when dealing with the SHAPE incorporation according to Zarringhalam et al. The following methods can be used to convert SHAPE reactivities into the probability for a certain nucleotide to be unpaired.

M: Use linear mapping according to Zarringhalam et al. C: Use a cutoff-approach to divide into paired and unpaired nucleotides (e.g. "C0.25") S: Skip the normalizing step since the input data already represents probabilities for being unpaired rather than raw reactivity values L: Use a linear model to convert the reactivity into a probability for being unpaired (e.g. "Ls0.68i0.2" to use a slope of 0.68 and an intercept of 0.2) O: Use a linear model to convert the log of the reactivity into a probability for being unpaired (e.g. "Os1.6i-2.29" to use a slope of 1.6 and an intercept of -2.29)

--commands=filename

Read additional commands from file

Commands include hard and soft constraints, but also structure motifs in hairpin and internal loops that need to be treeted differently. Furthermore, commands can be set for unstructured and structured domains.

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name DNA can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

--saltInit=DOUBLE

Provide salt correction for duplex initialization (in kcal/mol).

-m, **--modifications**[=STRING]

Allow for modified bases within the RNA sequence string.

(default="7I6P9D")

Treat modified bases within the RNA sequence differently, i.e. use corresponding energy corrections and/or pairing partner rules if available. For that, the modified bases in the input sequence must be marked by their corresponding one-letter code. If no additional arguments are supplied, all available corrections are performed. Otherwise, the user may limit the modifications to a particular subset of modifications, resp. one-letter codes, e.g. `-mP6` will only correct for pseudouridine and m6A bases.

Currently supported one-letter codes and energy corrections are:

`7`: 7-deaza-adenonsine (7DA)

`I`: Inosine

`6`: N6-methyladenosine (m6A)

`P`: Pseudouridine

`9`: Purine (a.k.a. nebularine)

`D`: Dihydrouridine

--mod-file=STRING

Use additional modified base data from JSON file.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

Plotting:

Command line options for changing the default behavior of structure layout and pairing probability plots

--noPS

Do not produce postscript drawing of the mfe structure.

(default=off)

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

S.H.Bernhart, Ch. Flamm, P.F. Stadler, I.L. Hofacker, (2006), "Partition Function and Base Pairing Probabilities of RNA Heterodimers", Algorithms Mol. Biol.

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Peter F Stadler, Stephan Bernhart, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.5 RNAdistance

RNAdistance - manual page for RNAdistance 2.8.0

Synopsis

```
RNAdistance [OPTION]...
```

DESCRIPTION

RNAdistance 2.8.0

Calculate distances between RNA secondary structures

This program reads RNA secondary structures from stdin and calculates one or more measures for their dissimilarity, based on tree or string editing (alignment). In addition it calculates a "base pair distance" given by the number of base pairs present in one structure, but not the other. For structures of different length base pair distance is not recommended.

RNAdistance accepts structures in bracket format, where matching brackets symbolize base pairs and unpaired bases are represented by a dot ., or coarse grained representations where hairpins, interior loops, bulges, multiloops, stacks and external bases are represented by (H), (I), (B), (M), (S), and (E), respectively. These can be optionally weighted. Full structures can be represented in the same fashion using the identifiers (U) and (P) for unpaired and paired bases, respectively. We call this the HIT representation (you don't want to know what this means). For example the following structure consists of 2 hairpins joined by a multiloop:

```
.((..(((...)))..((..)))).       full structure (usual format);
(U)((U2)((U3)P3)(U2)((U2)P2)P2) HIT structure;
((H)(H)M)  or
((((H)S)((H)S)M)S)              coarse grained structure;
(((((H3)S3)((H2)S2)M4)S2)E2)    weighted coarse grained.
```

The program will continue to read new structures until a line consisting of the single character @ or an end of file condition is encountered. Input lines neither containing a valid structure nor starting with > are ignored.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

-D, **--distance**=fhwcFHWCP

Specify the distance representation to be used in calculations.

(default="f")

Use the full, HIT, weighted coarse, or coarse representation to calculate the distance. Capital letters indicate string alignment otherwise tree editing is used. Any combination of distances can bespecified.

-X, **--compare**=p|m|f|c

Specify the comparison directive. *(default="p")*

Possible arguments for this option are: **-Xp** compare the structures pairwise (p), i.e. first with 2nd, third with 4th etc. **-Xm** calculate the distance matrix between all structures. The output is formatted as a lower triangle matrix. **-Xf** compare each structure to the first one. **-Xc** compare continuously, that is i-th with (i+1)th structure.

-S, **--shapiro**

Use the Bruce Shapiro's cost matrix for comparing coarse structures.

(default=off)

-B, **--backtrack**[=<filename>]

Print an "alignment" with gaps of the structures, to show matching substructures. The aligned structures are written to <filename>, if specified.

(default="none")

If <filename> is not specified, the output is written to stdout, unless the **-Xm** option is set in which case "backtrack.file" is used.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAdist.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

B.A. Shapiro (1988), "An algorithm for comparing multiple RNA secondary structures" CABIOS: 4, pp 381-393

B.A. Shapiro, K. Zhang (1990), "Comparing multiple RNA secondary structures using tree comparison", CABIOS: 6, pp 309-318

W. Fontana, D.A.M. Konings, P.F. Stadler and P. Schuster P (1993), "Statistics of RNA secondary structures", Biopolymers: 33, pp 1389-1404

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Walter Fontana, Ivo L Hofacker, Peter F Stadler

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.6 RNAdos

RNAdos - manual page for RNAdos 2.8.0

Synopsis

```
RNAdos [OPTIONS]
```

DESCRIPTION

RNAdos 2.8.0

Calculate the density of states for each energy band of an RNA

The program reads an RNA sequence and computes the density of states for each energy band.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-s, **--sequence**=STRING

The RNA sequence (ACGU).

-j, **--numThreads**=INT

Set the number of threads used for calculations (only available when compiled with OpenMP support)

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAdos.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations. The Minimum free energy (MFE) and a structure representative are calculated in any case.

-e, **--max-energy**=INT

Structures are only counted until this threshold is reached. Default is 0 kcal/mol.

(default="0")

-b, **--hashtable-bits**=INT

Set the size of the hash table for each cell in the dp-matrices.

(default="20")

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

J. Cupal, I.L. Hofacker, P.F. Stadler (1996), "Dynamic programming algorithm for the density of states of RNA secondary structures" Computer Science and Biology 96, Proc. German Conf. on Bioinformatics 1996, pp. 184-186.

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Gregor Entzian, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

SEE ALSO

RNAsubopt(1)

2.2.7 RNAduplex

RNAduplex - manual page for RNAduplex 2.8.0

Synopsis

```
RNAduplex [OPTION]...
```

DESCRIPTION

RNAduplex 2.8.0

Compute the structure upon hybridization of two RNA strands

reads two RNA sequences from stdin or <filename> and computes optimal and suboptimal secondary structures for their hybridization. The calculation is simplified by allowing only inter-molecular base pairs, for the general case use RNAcofold. The computed optimal and suboptimal structure are written to stdout, one structure per line. Each line consist of: The structure in dot bracket format with a & separating the two strands. The range of the structure in the two sequences in the format "from,to : from,to"; the energy of duplex structure in kcal/mol. The format is especially useful for computing the hybrid structure between a small probe sequence and a long target sequence.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-s, **--sorted**

Sort the printed output by free energy.

(default=off)

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAduplex.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations.

-e, **--deltaEnergy**=range

Compute suboptimal structures with energy in a certain range of the optimum (kcal/mol). Default is calculation of mfe structure only.

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

--saltInit=DOUBLE

Provide salt correction for duplex initialization (in kcal/mol).

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp=*"-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

SEE ALSO

RNAcofold(l) RNAfold(l)

2.2.8 RNAeval

RNAeval - manual page for RNAeval 2.8.0

Synopsis

```
RNAeval [OPTIONS] [<input0>] [<input1>]...
```

DESCRIPTION

RNAeval 2.8.0

Determine the free energy of a (consensus) secondary structure for (an alignment of) RNA sequence(s)

Evaluates the free energy of a particular (consensus) secondary structure for an (an alignment of) RNA molecule(s). The energy unit is kcal/mol and contains a covariance pseudo-energy term for multiple sequence alignments (*--msa* option) and corresponding consensus structures. The program will continue to read new sequences and structures until a line consisting of the single character @ or an end of file condition is encountered. If the input sequence or structure contains the separator character & the program calculates the energy of the co-folding of two RNA strands, where the & marks the boundary between the two strands.

-v, **--verbose**

Be verbose and print out energy contribution of each loop in the structure.

(default=off)

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-i, **--infile**=filename

Read a file instead of reading from stdin.

The default behavior of RNAeval is to read input from stdin or the file(s) that follow(s) the RNAeval command. Using this parameter the user can specify input file names where data is read from. Note, that any additional files supplied to RNAeval are still processed as well.

-a, **--msa**

Input is multiple sequence alignment in Stockholm 1.0 format.

(default=off)

Using this flag indicates that the input is a multiple sequence alignment (MSA) instead of (a) single sequence(s). Note, that only STOCKHOLM format allows one to specify a consensus structure. Therefore, this is the only supported MSA format for now!

--mis

Output "most informative sequence" instead of simple consensus: For each column of the alignment output the set of nucleotides with frequency greater than average in IUPAC notation.

(default=off)

-j, **--jobs**[=number]

Split batch input into jobs and start processing in parallel using multiple threads. A value of 0 indicates to use as many parallel threads as computation cores are available.

(default="0")

Default processing of input data is performed in a serial fashion, i.e. one sequence at a time. Using this switch, a user can instead start the computation for many sequences in the input in parallel. RNAeval will create as many parallel computation slots as specified and assigns input sequences of the input file(s) to the available slots. Note, that this increases memory consumption since input alignments have to be kept in memory until an empty compute slot is available and each running job requires its own dynamic programming matrices.

--unordered

Do not try to keep output in order with input while parallel processing is in place.

(default=off)

When parallel input processing (*--jobs* flag) is enabled, the order in which input is processed depends on the host machines job scheduler. Therefore, any output to stdout or files generated by this program will most likely not follow the order of the corresponding input data set. The default of RNAeval is to use a specialized data structure to still keep the results output in order with the input data. However, this comes with a trade-off in terms of memory consumption, since all output must be kept in memory for as long as no chunks of consecutive, ordered output are available. By setting this flag, RNAeval will not buffer individual results but print them as soon as they have been computated.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNAeval is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNAeval ignores any IDs retrieved from the input and automatically generates an ID for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="sequence")

If this parameter is set, each sequence will be prefixed with the provided string. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter _ between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAeval.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func
Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithmic details which should be included in the calculations.

-c, **--circ**
Assume a circular (instead of linear) RNA molecule.

(default=off)

-g, **--gquad**
Incoorporate G-Quadruplex formation into the structure prediction algorithm.

(default=off)

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--shape=filename
Use SHAPE reactivity data to guide structure predictions.

--shapeMethod=method
Select SHAPE reactivity data incorporation strategy.

(default="D")

The following methods can be used to convert SHAPE reactivities into pseudo energy contributions.

D: Convert by using the linear equation according to Deigan et al 2009.

Derived pseudo energy terms will be applied for every nucleotide involved in a stacked pair. This method is recognized by a capital D in the provided parameter, i.e.: *--shapeMethod=*"D" is the default setting. The slope m and the intercept b can be set to a non-default value if necessary, otherwise m=1.8 and b=-0.6. To alter these parameters, e.g. m=1.9 and b=-0.7, use a parameter string like this: *--shapeMethod=*"Dm1.9b-0.7". You may also provide only one of the two parameters like: *--shapeMethod=*"Dm1.9" or *--shapeMethod=*"Db-0.7".

Z: Convert SHAPE reactivities to pseudo energies according to Zarringhalam

et al 2012. SHAPE reactivities will be converted to pairing probabilities by using linear mapping. Aberration from the observed pairing probabilities will be penalized during the folding recursion. The magnitude of the penalties can affected by adjusting the factor beta (e.g. *--shapeMethod=*"Zb0.8").

W: Apply a given vector of perturbation energies to unpaired nucleotides

according to Washietl et al 2012.Perturbation vectors can be calculated by using RNApvmin.

--shapeConversion=method
Select method for SHAPE reactivity conversion.

(default="O")

This parameter is useful when dealing with the SHAPE incorporation according to Zarringhalam et al. The following methods can be used to convert SHAPE reactivities into the probability for a certain nucleotide to be unpaired.

M: Use linear mapping according to Zarringhalam et al. C: Use a cutoff-approach to divide into paired and unpaired nucleotides (e.g. "C0.25") S: Skip the normalizing step since the input data already represents probabilities for being unpaired rather than raw reactivity values L: Use a linear model to convert the reactivity into a probability for being unpaired (e.g. "Ls0.68i0.2" to use a slope of 0.68 and an intercept of 0.2) O: Use a linear model to convert the log of the reactivity into a probability for being unpaired (e.g. "Os1.6i-2.29" to use a slope of 1.6 and an intercept of -2.29)

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding. The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3.

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp=*"-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--logML

Recalculate energies of structures using a logarithmic energy function for multi-loops before output.

(default=off)

This option does not effect structure generation, only the energies that are printed out. Since logML lowers energies somewhat, some structures may be missing.

--cfactor=DOUBLE

Set the weight of the covariance term in the energy function

(default="1.0")

--nfactor=DOUBLE

Set the penalty for non-compatible sequences in the covariance term of the energy function

(default="1.0")

-R, **--ribosum_file**=ribosumfile

use specified Ribosum Matrix instead of normal

energy model.

Matrixes to use should be 6x6 matrices, the order of the terms is AU, CG, GC, GU, UA, UG.

-r, **--ribosum_scoring**

use ribosum scoring matrix. *(default=off)*

The matrix is chosen according to the minimal and maximal pairwise identities of the sequences in the file.

--old

use old energy evaluation, treating gaps as characters.

(default=off)

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Peter F Stadler, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.9 RNAfold

RNAfold - manual page for RNAfold 2.8.0

Synopsis

```
RNAfold [OPTIONS] [<input0.fa>] [<input1.fa>]...
```

DESCRIPTION

RNAfold 2.8.0

Calculate minimum free energy secondary structures and partition function of RNAs

The program reads RNA sequences, calculates their minimum free energy (mfe) structure and prints the mfe structure in bracket notation and its free energy. If not specified differently using commandline arguments, input is accepted from stdin or read from an input file, and output printed to stdout. If the *-p* option was given it also computes the partition function (pf) and base pairing probability matrix, and prints the free energy of the thermodynamic ensemble, the frequency of the mfe structure in the ensemble, and the ensemble diversity to stdout.

It also produces PostScript files with plots of the resulting secondary structure graph and a "dot plot" of the base pairing matrix. The dot plot shows a matrix of squares with area proportional to the pairing probability in the upper right half, and one square for each pair in the minimum free energy structure in the lower left half. For each pair i-j with probability p>10E-6 there is a line of the form

i j sqrt(p) ubox

in the PostScript file, so that the pair probabilities can be easily extracted.

Sequences may be provided in a simple text format where each sequence occupies a single line. Output files are named "rna.ps" and "dot.ps". Existing files of the same name will be overwritten.

It is also possible to provide sequence data in FASTA format. In this case, the first word of the FASTA header will be used as prefix for output file names. PostScript files "prefix_ss.ps" and "prefix_dp.ps" are produced for the structure and dot plot, respectively. Note, however, that once FASTA input was provided all following sequences must be in FASTA format too.

To avoid problems with certain operating systems and/or file systems the prefix will ALWAYS be sanitized! This step substitutes any special character in the prefix by a filename delimiter. See the *--filename-delim* option to change the delimiting character according to your requirements.

The program will continue to read new sequences until a line consisting of the single character `@` or an end of file (EOF) condition is encountered.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-i, **--infile**=filename

Read a file instead of reading from stdin.

The default behavior of RNAfold is to read input from stdin or the file(s) that follow(s) the RNAfold command. Using this parameter the user can specify input file names where data is read from. Note, that any additional files supplied to RNAfold are still processed as well.

-o, **--outfile**[=filename]

Print output to file instead of stdout.

This option may be used to write all output to output files rather than printing to stdout. The default filename is "RNAfold_output.fold" if no FASTA header precedes the input sequences and the *--auto-id* feature is inactive. Otherwise, output files with the scheme "prefix.fold" are generated, where the "prefix" is taken from the sequence id, e.g. the FASTA header. The user may specify a single output file name for all data generated from the input by supplying a filename as argument following immediately after this parameter. In case a file with the same filename already exists, any output of the program will be appended to it. Note: Any special characters in the filename will be replaced by the filename delimiter, hence there is no way to pass an entire directory path through this option (yet). (See also the "–filename-delim" parameter)

-j, **--jobs**[=number]

Split batch input into jobs and start processing in parallel using multiple threads. A value of 0 indicates to use as many parallel threads as computation cores are available.

(default="0")

Default processing of input data is performed in a serial fashion, i.e. one sequence at a time. Using this switch, a user can instead start the computation for many sequences in the input in parallel. RNAfold will create as many parallel computation slots as specified and assigns input sequences of the input file(s) to the available slots. Note, that this increases memory consumption since input alignments have to be kept in memory until an empty compute slot is available and each running job requires its own dynamic programming matrices.

--unordered

Do not try to keep output in order with input while parallel processing is in place.

(default=off)

When parallel input processing (*--jobs* flag) is enabled, the order in which input is processed depends on the host machines job scheduler. Therefore, any output to stdout or files generated by this program will most likely not follow the order of the corresponding input data set. The default of RNAfold is to use a specialized data structure to still keep the results output in order with the input data. However, this comes with a trade-off in terms of memory consumption, since all output must be kept in memory for as long as no chunks of consecutive, ordered output are available. By setting this flag, RNAfold will not buffer individual results but print them as soon as they have been computated.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNAfold is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNAfold ignores any IDs retrieved from the input and automatically generates an ID for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="sequence")

If this parameter is set, each sequence will be prefixed with the provided string. Hence, the output files will obey the following naming scheme: "prefix_xxxx_ss.ps" (secondary structure plot), "prefix_xxxx_dp.ps" (dot-plot), "prefix_xxxx_dp2.ps" (stack probabilities), etc. where xxxx is the sequence number. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--filename-full

Use full FASTA header to create filenames. *(default=off)*

This parameter can be used to deactivate the default behavior of limiting output filenames to the first word of the sequence ID. Consider the following example: An input with FASTA header `>NM_0001 Homo Sapiens some gene` usually produces output files with the prefix "NM_0001" without the additional data available in the FASTA header, e.g. "NM_0001_ss.ps" for secondary structure plots. With this flag set, no truncation of the output filenames is done, i.e. output filenames receive the full FASTA header data as prefixes. Note, however, that invalid characters (such as whitespace) will be substituted by a delimiting character or simply removed, (see also the parameter option *--filename-delim*).

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAfold.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

--benchmark

Perform a prediction performance test (benchmark) based on reference structures taken from the input.

(default=off)

--bm-output=filename

Store the performance benchmark results in a file.

--bm-output-append

Append performance benchmark data to a file instead of overwriting already existing content.

(default=off)

--bm-rm-pk

Remove pseudo-knots from the reference structure before computing benchmark.

(default=off)

This option adds a new output column "IR" to keep track of the number of reference structure base pairs that have been ignored for the benchmark.

--bm-rm-nc

Remove non-canonical base pairs from the reference structure before computing benchmark.

(default=off)

This option adds a new output column "IR" to keep track of the number of reference structure base pairs that have been ignored for the benchmark.

--bm-ignore-compat

Ignore predicted base pairs compatible with the reference strucure.

(default=off)

Predicted base pairs that are absent from the reference but do not structurally contradict it (i.e. do not cross any reference base pair) are excluded from the false positive count and reclassified as true negatives for these relaxed metrics.

This option adds a new output column "IP" to keep track of the number of

predicted base pairs that have been ignored for the benchmark.

--benchmark-fuzzy=INT

Allow base pair slippage when comparing the predicted structure to the reference.

(default="0")

A predicted base pair (p,q) is counted as a true positive if it lies within +/- this many nucleotides of a reference base pair (i,j). Each reference base pair can only be matched by one predicted base pair; once claimed it can't be counted again by another predicted pair.

Algorithms:

Select additional algorithms which should be included in the calculations. The Minimum free energy (MFE) and a structure representative are calculated in any case.

-p, **--partfunc**[=INT]

Calculate the partition function and base pairing probability matrix.

(default="1")

In addition to the MFE structure we print a coarse representation of the pair probabilities in form of a pseudo bracket notation followed by the ensemble free energy. This notation makes use of the letters `.`, `,`, `|`, `{`, `}`, `(`, and `)` denoting bases that are essentially unpaired, weakly paired, strongly paired without preference, weakly upstream (downstream) paired, or strongly up- (down-)stream paired bases, respectively. On the next line the centroid structure as derived from the pair probabilities together with its free energy and distance to the ensemble is shown. Finally it prints the frequency of the mfe structure, and the structural diversity (mean distance between the structures in the ensemble). See the description of `vrna_pf()` and `mean_bp_dist()` and `vrna_centroid()` in the RNAlib documentation for details. Note that unless you also specify `-d2` or `-d0`, the partition function and mfe calculations will use a slightly different energy model. See the discussion of dangling end options below.

An additionally passed value to this option changes the behavior of partition function calculation: `-p0` Calculate the partition function but not the pair probabilities, saving about 50% in runtime. This prints the ensemble free energy `dG=-kT ln(Z)`. `-p2` Compute stack probabilities, i.e. the probability that a pair `(i, j)` and the immediately enclosed pair `(i+1,j-1)` are formed simultaneously in addition to pair probabilities. A second postscript dot plot named "name_dp2.ps", or "dot2.ps" (if the sequence does not have a name), is produced that contains pair probabilities in the upper right half and stack probabilities in the lower left.

--betaScale=DOUBLE

Set the scaling of the Boltzmann factors. *(default="1.")*

The argument provided with this option is used to scale the thermodynamic temperature in the Boltzmann factors independently from the temperature of the individual loop energy contributions. The Boltzmann factors then become `exp(- dG/(kT*betaScale))` where `k` is the Boltzmann constant, `dG` the free energy contribution of the state and `T` the absolute temperature.

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

--MEA[=gamma]

Compute MEA (maximum expected accuracy) structure.

(default="1.")

The expected accuracy is computed from the pair probabilities: each base pair `(i,j)` receives a score `2*gamma*p_ij` and the score of an unpaired base is given by the probability of not forming a pair. The parameter gamma tunes the importance of correctly predicted pairs versus unpaired bases. Thus, for small values of gamma the MEA structure will contain only pairs with very high probability. Using *--MEA* implies *-p* for computing the pair probabilities.

-c, **--circ**

Assume a circular (instead of linear) RNA molecule.

(default=off)

--ImFeelingLucky

Return exactly one stochastically backtracked structure.

(default=off)

This function computes the partition function and returns exactly one secondary structure stochastically sampled from the Boltzmann equilibrium according to its probability in the ensemble

--bppmThreshold=cutoff

Set the threshold/cutoff for base pair probabilities included in the postscript output.

(default="1e-5")

By setting the threshold the base pair probabilities that are included in the output can be varied. By default only those exceeding `1e-5` in probability will be shown as squares in the dot plot. Changing the threshold to any other value allows for increase or decrease of data.

-g, **--gquad**

Incoorporate G-Quadruplex formation into the structure prediction algorithm.

(default=off)

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--maxBPspan=INT

Set the maximum base pair span.

(default="-1")

-C, **--constraint**[=filename]

Calculate structures subject to constraints. *(default="")*

The program reads first the sequence, then a string containing constraints on the structure encoded with the symbols:

`.` (no constraint for this base)

`|` (the corresponding base has to be paired

`x` (the base is unpaired)

< (base i is paired with a base j>i)

> (base i is paired with a base j<i)

and matching brackets `( )` (base i pairs base j)

With the exception of `|`, constraints will disallow all pairs conflicting with the constraint. This is usually sufficient to enforce the constraint, but occasionally a base may stay unpaired in spite of constraints. PF folding ignores constraints of type `|`.

--batch

Use constraints for multiple sequences. *(default=off)*

Usually, constraints provided from input file only apply to a single input sequence. Therefore, RNAfold will stop its computation and quit after the first input sequence was processed. Using this switch, RNAfold processes multiple input sequences and applies the same provided constraints to each of them.

--canonicalBPonly

Remove non-canonical base pairs from the structure constraint.

(default=off)

--enforceConstraint

Enforce base pairs given by round brackets ( ) in structure constraint.

(default=off)

--motif=SEQUENCE,STRUCTURE,ENERGY

Specify stabilizing energy of a ligand binding

to a particular sequence/structure motif.

Some ligands binding to RNAs require and/or induce particular sequence and structure motifs. For instance they bind to an internal loop, or small hairpin loop. If for such cases a binding free energy is known, the binding and therefore stabilizing effect of the ligand can be included in the folding recursions. Interior loop motifs are specified as concatenations of 5`` and 3`` motif, separated by an & character.

Energy contributions must be specified in kcal/mol.

See the manpage for an example usage of this option.

--commands=filename

Read additional commands from file

Commands include hard and soft constraints, but also structure motifs in hairpin and internal loops that need to be treeted differently. Furthermore, commands can be set for unstructured and structured domains.

Experimental Structure Probing Data:

The following arguments and switches control various implementations that allow for guiding the structure prediction with the help of additional (experimental) RNA structure probing data, such as SHAPE, DMS, etc.

--sp-data=filename

Read structure probing data from an input file and guide the predictions accordingly. Must precede the strategy, i.e. a data file must be specified before the corresponding *--sp-strategy* option!

This option and its argument only specifies the file name with the (reactivity) data. What kind of data the file contains and how the data is actually used during the predictions can be specified by the *--sp-strategy* and *--sp-preprocess* options.

--sp-strategy=strategy

Select the strategy how the probing data is used to guide the structure predictions.

(default="D")

The following strategies (methods) are available to convert structure probing data into pseudo energy contributions:

D: Deigan et al 2009 strategy.

Here, probing data is converted into pseudo energy terms using a linear

model. The derived energies are then applied for every nucleotide involved in a stacked pair. The default slope and intercept variables of the linear model are m=1.8 and b=-0.6, respectively. To change these parameters, e.g. to m=1.9 and b=-0.7, use a parameter string like this: *--sp-strategy=*"Dm1.9b-0.7". Note the precedence of m over b. You may also provide only one of the two variables like: *--sp-strategy=*"Dm1.9" or *--sp-strategy=*"Db-0.7".

E: Eddy 2014 strategy.

This strategy distinguishes two structural states (classes) for each base,

paired and unpaired. Based on a list of prior distributions for each of the classes, the posterior probability for observing a probing reactivity of a base is estimated and then converted into a respective pseudo energy contribution. Thermodynamic temperature T is taken from the model settings as specified by *--temp* but can be changed to any other value, either in degrees Celcius (c or t) or Kelvin (k). Use one of the following parameter strings to change the temperature to 25 degree Celcius: *--sp-strategy=*"Ec=25" or *--sp-strategy=*"Ek=298.15". By default, built-in prior distributions are used with this strategy. This can be changed if the user passes additional probing data files via the *--sp-data* option and then selects the prior distribution specifier (P) as strategy (see below).

P: Use probing data as prior distribution for Eddy 2014 strategy.

This indicates that the most recent probing data file actually contains a

prior distribution of probing data for a particular structure class, i.e. unpaired or paired bases. The prior distribution then replaces the default priors in the Eddy 2014 (E) strategy. Use the lower case characters u and p to indicate which class, unpaired or paired, the prior distribution represents, e.g. *--sp-strategy=*"Pp" will be interpreted as prior distribution for paired nucleotides. Note, that this option must follow after the *--sp-strategy=*"E" and before any further *--sp-strategy* options that do not specify other prior distributions!

Z: Zarringhalam et al 2012 strategy.

Here, the probing data is first converted into pairing probabilities and

aberration from the observed pairing probabilities will be penalized during the folding recursion. The magnitude of the penalties can affected by adjusting the factor beta (e.g. *--sp-strategy=*"Zb0.8").

W: Apply a given vector of perturbation energies to unpaired nucleotides

according to Washietl et al 2012. Perturbation vectors can be calculated by using RNApvmin.

X: Cross-linking data input, according to Lorenz et al. 2026. Use interval

pairs derived from cross-linking experiments to guide the structure prediction.

--sp-preprocess=method

Select a method to pre-process the structure probing data before it undergoes conversion by a strategy.

(default="I")

--shape=filename

Use SHAPE reactivity data to guide structure predictions.

--shapeMethod=method

Select SHAPE reactivity data incorporation strategy.

(default="D")

The following methods can be used to convert SHAPE reactivities into pseudo energy contributions.

D: Convert by using the linear equation according to Deigan et al 2009.

Derived pseudo energy terms will be applied for every nucleotide involved in a stacked pair. This method is recognized by a capital D in the provided parameter, i.e.: *--shapeMethod=*"D" is the default setting. The slope m and the intercept b can be set to a non-default value if necessary, otherwise m=1.8 and b=-0.6. To alter these parameters, e.g. m=1.9 and b=-0.7, use a parameter string like this: *--shapeMethod=*"Dm1.9b-0.7". You may also provide only one of the two parameters like: *--shapeMethod=*"Dm1.9" or *--shapeMethod=*"Db-0.7".

Z: Convert SHAPE reactivities to pseudo energies according to Zarringhalam

et al 2012. SHAPE reactivities will be converted to pairing probabilities by using linear mapping. Aberration from the observed pairing probabilities will be penalized during the folding recursion. The magnitude of the penalties can affected by adjusting the factor beta (e.g. *--shapeMethod=*"Zb0.8").

W: Apply a given vector of perturbation energies to unpaired nucleotides

according to Washietl et al 2012. Perturbation vectors can be calculated by using RNApvmin.

--shapeConversion=method

Select method for SHAPE reactivity conversion.

(default="O")

This parameter is useful when dealing with the SHAPE incorporation according to Zarringhalam et al. The following methods can be used to convert SHAPE reactivities into the probability for a certain nucleotide to be unpaired.

M: Use linear mapping according to Zarringhalam et al. C: Use a cutoff-approach to divide into paired and unpaired nucleotides (e.g. "C0.25") S: Skip the normalizing step since the input data already represents probabilities for being unpaired rather than raw reactivity values L: Use a linear model to convert the reactivity into a probability for being unpaired (e.g. "Ls0.68i0.2" to use a slope of 0.68 and an intercept of 0.2) O: Use a linear model to convert the log of the reactivity into a probability for being unpaired (e.g. "Os1.6i-2.29" to use a slope of 1.6 and an intercept of -2.29)

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name DNA can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

-m, **--modifications**[=STRING]

Allow for modified bases within the RNA sequence string.

(default="7I6P9D")

Treat modified bases within the RNA sequence differently, i.e. use corresponding energy corrections and/or pairing partner rules if available. For that, the modified bases in the input sequence must be marked by their corresponding one-letter code. If no additional arguments are supplied, all available corrections are performed. Otherwise, the user may limit the modifications to a particular subset of modifications, resp. one-letter codes, e.g. -mP6 will only correct for pseudouridine and m6A bases.

Currently supported one-letter codes and energy corrections are:

7: 7-deaza-adenonsine (7DA)

I: Inosine

6: N6-methyladenosine (m6A)

P: Pseudouridine

9: Purine (a.k.a. nebularine)

D: Dihydrouridine

--mod-file=STRING

Use additional modified base data from JSON file.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

Plotting:

Command line options for changing the default behavior of structure layout and pairing probability plots

--noPS

Do not produce postscript drawing of the mfe structure.

(default=off)

--noDP

Do not produce dot-plot postscript file containing base pair or stack probabilitities.

(default=off)

In combination with the *-p* option, this flag turns-off creation of individual dot-plot files. Consequently, computed base pair probability output is omitted but centroid and MEA structure prediction is still performed.

-t, **--layout-type**=INT

Choose the layout algorithm. *(default="1")*

Select the layout algorithm that computes the nucleotide coordinates. Currently, the following algorithms are available:

0: simple radial layout

1: Naview layout (Bruccoleri et al. 1988)

2: circular layout

3: RNAturtle (Wiegreffe et al. 2018)

4: RNApuzzler (Wiegreffe et al. 2018)

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

M. Zuker, P. Stiegler (1981), "Optimal computer folding of large RNA sequences using thermodynamic and auxiliary information", Nucl Acid Res: 9, pp 133-148

J.S. McCaskill (1990), "The equilibrium partition function and base pair binding probabilities for RNA secondary structures", Biopolymers: 29, pp 1105-1119

I.L. Hofacker & P.F. Stadler (2006), "Memory Efficient Folding Algorithms for Circular RNA Secondary Structures", Bioinformatics

A.F. Bompfuenewerer, R. Backofen, S.H. Bernhart, J. Hertel, I.L. Hofacker, P.F. Stadler, S. Will (2007), "Variations on {RNA} Folding and Alignment: Lessons from Benasque", J. Math. Biol.

D. Adams (1979), "The hitchhiker's guide to the galaxy", Pan Books, London

The calculation of mfe structures is based on dynamic programming algorithm originally developed by M. Zuker and P. Stiegler. The partition function algorithm is based on work by J.S. McCaskill.

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

EXAMPLES

Basic Usage Examples

Single line sequence input and calculation of partition function and MEA structure

```
$ RNAfold --MEA -d2 -p
```

The program will then prompt for sequence input. Using the example sequence "CGACGTAGATGCTAGCTGACTCGATGC" and pressing ENTER the output of the program will be similar to

```
CGACGUAGAUGCUAGCUGACUCGAUGC
(((.((((.......)).)))))....
minimum free energy =  -1.90 kcal/mol
(((.((((.......))},})))).....
free energy of ensemble =  -2.86 kcal/mol
(((.(.((.......))..)))).... {  0.80 d=2.81}
(((.((((.......))).)))).... { -1.90 MEA=22.32}
frequency of mfe structure in ensemble 0.20997; ensemble diversity 4.19
```

Here, the first line just repeats the sequence input. The second line contains a MFE structure in dot bracket notation followed by the minimum free energy. After this, the pairing probabilities for each nucleotide are shown in a pseudo dot-bracket notation followed by the free energy of ensemble. The next two lines show the centroid structure with its free energy and its distance to the ensemble as well as the MEA structure, its free energy and the maximum expected accuracy, respectively. The last line finally contains the frequency of the MFE representative in the complete ensemble of secondary structures and the ensemble diversity. For further details about the calculation and interpretation of the given output refer to the reference manual of RNAlib.

Since version 2.0 it is also possible to provide FASTA file sequence input. Assume you have a file containing two sequences in FASTA format, e.g

```
$ cat sequences.fa
>seq1
CGGCUCGCAACAGACCUAUUAGUUUUACGUAAUAUUUG
GAACGAUCUAUAACACGACUUCACUCUU
>seq2
GAAUGACCCGAUAACCCCGUAAUAUUUGGAACGAUCUA
UAACACGACUUCACUCUU
```

In order to compute the MFE for the two sequences the user can use the following command

```
$ RNAfold < sequences.fa
```

which would result in an output like this

```
>seq1
CGGCUCGCAACAGACCUAUUAGUUUUACGUAAUAUUUGGAACGAUCUAUAACACGACUUCACUCUU
.((.(((...((((..(((((........)))))))))...))).)).................... ( -5.40)
>seq2
GAAUGACCCGAUAACCCCGUAAUAUUUGGAACGAUCUAUAACACGACUUCACUCUU
.......((((.............)))).............................. ( -2.00)
```

Constraint Examples

Secondary structure constraints may be given in addition to the sequence information, too. Using the first sequence of the previous example and restricting the nucleotides of the outermost helix to be unpaired, i.e. base pairs (2,47) and (3,46) the input file should have the following form

```
$ cat sequence_unpaired.fa
>seq1
CGGCUCGCAACAGACCUAUUAGUUUUACGUAAUAUUUG
GAACGAUCUAUAACACGACUUCACUCUU
.xx..................................
.......xx...................
```

Calling RNAfold with the structure constraint option -C it shows the following result

```
$ RNAfold -C < sequence_unpaired.fa
>seq1
CGGCUCGCAACAGACCUAUUAGUUUUACGUAAUAUUUGGAACGAUCUAUAACACGACUUCACUCUU
....(((...((((..(((((........)))))))))...))).......................... ( -4.20)
```

This represents the minimum free energy and a structure representative of the RNA sequence given that nucleotides 2,3,46 and 47 must not be involved in any base pair. For further information about constrained folding refer to the details of the -C option and the reference manual of RNAlib.

Since version 2.2 the ViennaRNA Package distinguishes hard and soft constraints. As a consequence, structure predictions are easily amenable to a versatile set of constraints, such as maximal base pair span, incorporation of SHAPE reactivity data, and RNA-ligand binding to hairpin, or interior loop motifs.

Restricting the maximal span of a base pair

A convenience commandline option allows you to easily limit the distance (j - i + 1) between two nucleotides i and j that form a basepair. For instance a limit of 600nt can be accomplished using:

```
$ RNAfold --maxBPspan 600
```

Complex structure constraints and grammar extensions

Structure constraints beyond those that can be expressed with a pseudo-dot bracket notation may be provided in a so-called command file:

```
$ RNAfold --commands=constraints.txt < sequence.fa
```

The command file syntax is a generalization of constraints as used in UNAfold/mfold. Each line starts with a one or two letter command followed by command parameters. For structure constraints, this amounts to a single command character followed by three or four numbers. In addition, optional auxiliary modifier characters may be used to limit the constraint to specific loop types. For base pair specific constraints, we currently distinguish pairs in exterior loops (E), closing pairs of hairpin loops (H), closing (I) and enclosed (i) pairs of interior loops, and closing (M) and enclosed (m) pairs of multibranch loops. Nucleotide-wise constraints may be limited to their loop context using the corresponding uppercase characters. The default is to apply a constraint to all (A) loop

types. Furthermore, pairing constraints for single nucleotides may be limited to upstream (U), or downstream (D) orientation. The command file specification is as follows:

```
F i 0 k   [TYPE] [ORIENTATION] # Force nucleotides i...i+k-1 to be paired
F i j k   [TYPE] # Force helix of size k starting with (i,j) to be formed
P i 0 k   [TYPE] # Prohibit nucleotides i...i+k-1 to be paired
P i j k   [TYPE] # Prohibit pairs (i,j),...,(i+k-1,j-k+1)
P i-j k-l [TYPE] # Prohibit pairing between two ranges
C i 0 k   [TYPE] # Nucleotides i,...,i+k-1 must appear in context TYPE
C i j k          # Remove pairs conflicting with (i,j),...,(i+k-1,j-k+1)
E i 0 k e        # Add pseudo-energy e to nucleotides i...i+k-1
E i j k e        # Add pseudo-energy e to pairs (i,j),...,(i+k-1,j-k+1)
UD m e    [LOOP] # Add ligand binding to unstructured domains with motif
# m and binding free energy e

# [LOOP]        = { E, H, I, M, A }
# [TYPE]        = [LOOP] + { i, m }
# [ORIENTATION] = { U, D }
```

Again, RNAfold by default only processes the first sequence in the input sequence when provided with constraints in a command file. To apply the exact same constraints to each of the input sequences in a multi FASTA file, use the batch mode commandline option:

```
$ RNAfold --constraint=constraints.txt --batch < sequences.fa
```

Structure Probing Data Example

Structure predictions can be guided by additional structure probing data. Such data may be derived from SHAPE, DMS, or similar techniques. In our software we implement different strategies that convert the probing data (reactivities) into pseudo energy contributions that guide the predictions.

Probing reactivities have to be provided in the form of a two column text file with sequence positions (1-based) and (normalized) reactivity values (usually between 0 and 2. Missing values may be left out, or assigned a negative score, e.g.:

```
$ cat reactivities.dat
9    -999       # No reactivity information
10   -999
11   0.042816   # normalized SHAPE reactivity
12   0          # also a valid SHAPE reactivity
15   0.15027    # Missing data for pos. 13-14
...
42   0.16201
```

A simple example using the default strategy (Deigan et al. 2009) assuming SHAPE reactivity data in `reactivities.dat` may look like:

```
$ RNAfold --shape=reactivities.dat < sequence.fa
```

Another example using the Eddy 2014 strategy with non-default prior distribution for unpaired bases located in file `pu.txt` may look like:

```
$ RNAfold --sp-data=reactivities.dat --sp-strategy=E --sp-data=pu.txt --sp-strategy=Pu
```

Note, that RNAfold will only process the first sequence in the input file, when provided with SHAPE reactivity data!

Post-transcriptional Modification Example

Many RNA molecules harbor (post-transcriptional) modifications. These modified base often change the pairing behavior or energy contribution for the loops they are part of. To accommodate for that effect (to a certain degree) one may use additional correcting energy parameters for loops with the respective modified bases. In literature, a few stacking- and some terminal mismatch energies can be found. Some of them are already provided within the ViennaRNA Package. The –modification and –mod-file command line parameters can be used to apply these parameters in the predictions. While the former allows one to select a subset of implemented modified base corrections, the latter enables the prediction programs to read energy parameters for modified bases from a user-provided JSON file.

Consider, for instance, the following tRNA sequence with dihydrouridines and pseudouridines annotated by their respective one-letter codes D and P:

```
$ cat tRNAphe.fa
>tRNAphe
GCCGAAAUAGCUCAGDDGGGAGAGCGPPAGACUGAAGAPCUAAAGGDCCCUGGUPCGAUCCCGGGUUUCGGCACCA
```

Now, a prediction that includes support for the destabilizing effect of D and the stabilizing effects of P within base pair stacks can be done as follows:

```
$ RNAfold --modifications=DP < tRNAphe.fa
>tRNAphe
GCCGAAAUAGCUCAGDDGGGAGAGCGPPAGACUGAAGAPCUAAAGGDCCCUGGUPCGAUCCCGGGUUUCGGCACCA
(((((((..((((........)))).(((((.......)))))....(((.((......)).)))))))))).... (-23.37)
```

Ligand Binding Examples

Ligand binding contributions to specific hairpin/interior loop motifs

A convenience function allows one to specify a hairping/interior loop motif where a ligand is binding with a particular binding free energy dG. Here is an example that adds a theophylline binding motif. Free energy contribution of this motif of dG=-9.22kcal/mol is derived from k_d=0.32umol/l, taken from Jenison et al. 1994. Although the structure motif consists of a symmetric interior loop of size 6, followed by a small helix of 3 basepairs, and a bulge of 3 nucleotides, the entire structure can still be represented by one interior loop. See the below mofif description where the & character splits the motif into a 5' and a 3' part. The first line gives the sequences motif, the second line shows the actual structure motif of the aptamer pocket, and the third line is the interior loop motif that fully encapsulates the theophylline aptamer:

```
GAUACCAG&CCCUUGGCAGC
(...((((&)...)))...)
(......(&).........)
```

To use the above information in the folding recursions of RNAfold, one only needs to provide the motif itself, and binding free energy:

```
$ RNAfold --motif="GAUACCAG&CCCUUGGCAGC,(...((((&)...)))...),-9.22" < sequences.fa
```

Adding the –verbose option to the above call of RNAfold also prints the sequence position of each motif found in the MFE structure. In case interior-loop like motifs are provided, two intervals are printed denoting the 5`` and 3`` part, respectively.

Ligand binding contributions to unpaired segments of the RNA structure

The extension of the RNA folding grammar with unstructured domains allows for an easy incorporation of ligands that bind to unpaired stretches of an RNA structure. To model such interactions only two parameters are required: (i) a sequence motif in IUPAC notation that specifies where the ligand binds to, and (ii) a binding free energy that can be derived from the association/dissociation constant of the ligand. With these two parameters in hand, the modification of RNAfold to include the competition of regular intramolecular base pairing and ligand interaction is as easy as writing a simple command file of the form:

```
UD m e    [LOOP]
```

where m is the motif string in upper-case IUPAC notation, and e the binding free energy in kcal/mol and optional loop type restriction [LOOP]. See also the command file specification as defined above.

For instance, having a protein with a 4-nucleotide footprint binding AAAA, a binding free energy e = -5.0 kcal/mol, and a binding restriction to exterior- and multibranch loops results in a command file:

```
$ cat commands.txt
UD AAAA -5.0  ME
```

and the corresponding call to RNAfold to compute MFE and equilibrium probabilities becomes:

```
$ RNAfold --commands=commands.txt -p < sequence.fa
```

The resulting MFE plot will be annotated to display the binding site(s) of the ligand, and the base pair probability dot-plot is extended to include the probability that a particular nucleotide is bound by the ligand.

AUTHOR

Ivo L Hofacker, Walter Fontana, Sebastian Bonhoeffer, Peter F Stadler, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.10 RNAheat

RNAheat - manual page for RNAheat 2.8.0

Synopsis

```
RNAheat [OPTIONS] [<input0>] [<input1>]...
```

DESCRIPTION

RNAheat 2.8.0

calculate specific heat of RNAs

Reads RNA sequences from stdin or input files and calculates their specific heat in the temperature range t1 to t2, from the partition function by numeric differentiation. The result is written to stdout as a list of pairs of temperature in C and specific heat in kcal/(mol*K). The program will continue to read new sequences until a line consisting of the single character @ or an end of file condition is encountered.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-i, **--infile**=filename

Read a file instead of reading from stdin

The default behavior of RNAheat is to read input from stdin or the file(s) that follow(s) the RNAheat command. Using this parameter the user can specify input file names where data is read from. Note, that any additional files supplied to RNAheat are still processed as well.

-j, **--jobs**[=number]

Split batch input into jobs and start processing in parallel using multiple threads. A value of 0 indicates to use as many parallel threads as computation cores are available.

(default="0")

Default processing of input data is performed in a serial fashion, i.e. one sequence at a time. Using this switch, a user can instead start the computation for many sequences in the input in parallel. RNAheat will create as many parallel computation slots as specified and assigns input sequences of the input file(s) to the available slots. Note, that this increases memory consumption since input alignments have to be kept in memory until an empty compute slot is available and each running job requires its own dynamic programming matrices.

--unordered

Do not try to keep output in order with input while parallel processing is in place.

(default=off)

When parallel input processing (*--jobs* flag) is enabled, the order in which input is processed depends on the host machines job scheduler. Therefore, any output to stdout or files generated by this program will most likely not follow the order of the corresponding input data set. The default of RNAheat is to use a specialized data structure to still keep the results output in order with the input data. However, this comes with a trade-off in terms of memory consumption, since all output must be kept in memory for as long as no chunks of consecutive, ordered output are available. By setting this flag, RNAheat will not buffer individual results but print them as soon as they have been computated.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNAheat is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNAheat ignores any IDs retrieved from the input and automatically generates an ID for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names)

(default="sequence")

If this parameter is set, each sequences' FASTA id will be prefixed with the provided string. FASTA ids then take the form ">prefix_xxxx" where xxxx is the sequence number. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using

this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated alignment IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAheat.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations.

--Tmin=t1

Lowest temperature.

(default="0")

--Tmax=t2

Highest temperature.

(default="100")

--stepsize=FLOAT

Calculate partition function every stepsize degrees C.

(default="1.")

-m, **--ipoints**=ipoints

The program fits a parabola to 2*ipoints+1 data points to calculate 2nd derivatives. Increasing this parameter produces a smoother curve.

(default="2")

-c, **--circ**

Assume a circular (instead of linear) RNA molecule.

(default=off)

-g, **--gquad**

Incoorporate G-Quadruplex formation into the structure prediction algorithm.

(default=off)

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--maxBPspan=INT

Set the maximum base pair span.

(default="-1")

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops

(default="2")

With `-d2` dangling energies will be added for the bases adjacent to a helix on both sides in any case

.HP `-d0` ignores dangling ends altogether (mostly for debugging).

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Peter F Stadler, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

SEE ALSO

RNAfold(1)

2.2.11 RNAinverse

RNAinverse - manual page for RNAinverse 2.8.0

Synopsis

```
RNAinverse [OPTION]...
```

DESCRIPTION

RNAinverse 2.8.0

Find RNA sequences with given secondary structure

The program searches for sequences folding into a predefined structure, thereby inverting the folding algorithm. Target structures (in bracket notation) and starting sequences for the search are read alternately from stdin. Characters in the start sequence other than "AUGC" (or the alphabet specified with *-a*) will be treated as wild cards and replaced by a random character. Any lower case characters in the start sequence will be kept fixed during the search. If necessary, the sequence will be elongated to the length of the structure. Thus a string of "N"s as well as a blank line specify a random start sequence. For each search the best sequence found and its Hamming distance to the start sequence are printed to stdout. If the the search was unsuccessful, a structure distance to the target is appended. The **-Fp** and *-R* options can modify the output format, see commandline options below. The program will continue to read new structures and sequences until a line consisting of the single character "@" or an end of file condition is encountered.

-v, **--verbose**

In conjunction with a negative value supplied to *-R*, print the last subsequence and substructure for each unsuccessful search.

(default=off)

Lower the log level setting such that even INFO messages are passed through.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAinverse.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations.

-F, **--function**=mp

Use minimum energy (`-Fm`), partition function folding (`-Fp`) or both (`-Fmp`).

(default="m")

In partition function mode, the probability of the target structure exp(`-E`(S)/kT)/Q is maximized. This probability is written in brackets after the found sequence and Hamming distance. In most cases you'll want to use the :option:`-f` option in conjunction with `-Fp`, see below.

-f, **--final**=FLOAT

In combination with `-Fp` stop search when sequence is found with E(s)-F is smaller than final, where F=-kT*ln(Q).

-R, **--repeat**[=INT]

Search repeatedly for the same structure. If an argument is supplied to this option it must follow the option flag immediately. E.g.: `-R5`

(default="1")

If repeats is negative search until `--repeats` exact solutions are found, no output is done for unsuccessful searches. Be aware, that the program will not terminate if the target structure can not be found. If no value is supplied with this option, the default value is used.

--init-strategy=strategy

Use a specific strategy to generate an initial sequence seed.

(default="random")

This option allows for changing the initial sequence seed generator strategy in cases where no start sequence is specified. By default, the sequence characters are randomly chosen with equal probabilities for any nucleotide. A more elaborate method is the "LinearBP" approach (Boury et al. 2025) to find a good starting sequence candidate that already fulfills optimality within a base pair model. Note, that currently this option has no effect if a start sequence is provided as an input. Available strategies are:

"random": Generate a random sequence with equal nucleotide probabilities.

"linearbp": Use the "LinearBP" approach.

-a, **--alphabet**=ALPHABET

Find sequences using only nucleotides from a given alphabet.

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. When passing the placeholder file name "DNA", DNA parameters are loaded without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs. e.g. RNAfold `-nsp -GA` will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

D.H. Turner, N. Sugimoto, S.M. Freier (1988), "RNA structure prediction", Ann Rev Biophys Biophys Chem: 17, pp 167-192

M. Zuker, P. Stiegler (1981), "Optimal computer folding of large RNA sequences using thermodynamic and auxiliary information", Nucl Acid Res: 9, pp 133-148

J.S. McCaskill (1990), "The equilibrium partition function and base pair binding probabilities for RNA secondary structures", Biopolymers: 29, pp 1105-1119

T. Boury, S. Gardelle, L. Bulteau, Y. Ponty (2025), "RNA inverse folding can be solved in linear time for structures without isolated stacks or base pairs", Algorithms Mol Biol: 20(1):20

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

EXAMPLES

To search 5 times for sequences forming a simple hairpin structure interrupted by one GA mismatch call

```
$ RNAinverse -R 5
```

and enter the lines

```
(((.(((....))).)))
NNNgNNNNNNNNNaNNN
```

AUTHOR

Ivo L Hofacker

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.12 RNALalifold

RNALalifold - manual page for RNALalifold 2.8.0

Synopsis

```
RNALalifold [options] <file1.aln>
```

DESCRIPTION

RNALalifold 2.8.0

calculate locally stable secondary structures for a set of aligned RNAs

reads aligned RNA sequences from stdin or file.aln and calculates locally stable RNA secondary structure with a maximal base pair span. For a sequence of length n and a base pair span of L the algorithm uses only O(n+L*L) memory and O(n*L*L) CPU time. Thus it is practical to "scan" very large genomes for short RNA

structures.

-v, **--verbose**
Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

-q, **--quiet**
Be quiet. *(default=off)*

This option can be used to minimize the output of additional information and non-severe warnings which otherwise might spam stdout/stderr.

I/O Options:

Command line options for input and output (pre-)processing

-f, **--input-format**=C|S|F|M
File format of the input multiple sequence alignment (MSA).

If this parameter is set, the input is considered to be in a particular file format. Otherwise, the program tries to determine the file format automatically, if an input file was provided in the set of parameters. In case the input MSA is provided in interactive mode, or from a terminal (TTY), the programs default is to assume CLUSTALW format. Currently, the following formats are available: ClustalW (`C`), Stockholm 1.0 (`S`), FASTA/Pearson (`F`), and MAF (`M`).

--csv
Create comma separated output (csv)

(default=off)

--aln[=prefix]
Produce output alignments and secondary structure plots for each hit found.

This option tells the program to produce, for each hit, a colored and structure annotated (sub)alignment and secondary structure plot in PostScript format. It also adds the subalignment hit into a multi-Stockholm formatted file "RNALalifold_results.stk". The postscript output file names are "aln_start_end.eps" and "ss_start_end.eps". All files will be created in the current directory. The optional argument string can be used to set a specific prefix that is used to name the output files. The file names then become "prefix_aln_start_end.eps", "prefix_ss_start_end.eps", and "prefix.stk". Note: Any special characters in the prefix will be replaced by the filename delimiter, hence there is no way to pass an entire directory path through this option yet. (See also the "–filename-delim" parameter)

--aln-stk[=prefix]

Add hits to a multi-Stockholm formatted output file.

(default="RNALalifold_results")

The default file name used for the output is "RNALalifold_results.stk". Users may change the filename to "prefix.stk" by specifying the prefix as optional argument. The file will be create in the current directory if it does not already exist. In case the file already exists, output will be appended to it. Note: Any special characters in the prefix will be replaced by the filename delimiter, hence there is no way to pass an entire directory path through this option yet. (See also the "–filename-delim" parameter)

--mis

Output "most informative sequence" instead of simple consensus: For each column of the alignment output the set of nucleotides with frequency greater than average in IUPAC notation.

(default=off)

--split-contributions

Split the free energy contributions into separate parts

(default=off)

By default, only the total energy contribution for each hit is returned. Using this option, this contribution is split into individual parts, i.e. the Nearest Neighbor model energy, the covariance pseudo energy, and if applicable, a remaining pseudo energy derived from special constraints, such as probing signals like SHAPE.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each alignment.

(default=off)

The default mode of RNALalifold is to automatically determine an ID from the input alignment if the input file format allows to do that. Alignment IDs are, for instance, usually given in Stockholm 1.0 formatted input. If this flag is active, RNALalifold ignores any IDs retrieved from the input and automatically generates an ID for each alignment.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="alignment")

If this parameter is set, each alignment will be prefixed with the provided string. Hence, the output files will obey the following naming scheme: "prefix_xxxx_ss.ps" (secondary structure plot), "prefix_xxxx_dp.ps" (dot-plot), "prefix_xxxx_aln.ps" (annotated alignment), etc. where xxxx is the alignment number beginning with the second alignment in the input. Use this setting in conjunction with the *--continuous-ids* flag to assign IDs beginning with the first input alignment.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18].

--id-start=LONG

Specify the first number in automatically generated alignment IDs.

(default="1")

When alignment IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies continuous alignment IDs, i.e. it activates the *--continuous-ids* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNALalifold.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations. The Minimum free energy (MFE) and a structure representative are calculated in any case.

-L, **--maxBPspan**=INT

Set the maximum allowed separation of a base pair to span. I.e. no pairs (i,j) with j-i>span will be allowed.

(default="70")

--threshold=DOUBLE

Energy threshold in kcal/mol per nucleotide above which secondary structure hits are omitted in the output.

(default="-0.1")

-g, **--gquad**

Incoorporate G-Quadruplex formation into the structure prediction algorithm.

(default=off)

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--shape=file1,file2

Use SHAPE reactivity data to guide structure predictions.

Multiple shapefiles for the individual sequences in the alignment may be specified as a comma separated list. An optional association of particular shape files to a specific sequence in the alignment can be expressed by prepending the sequence number to the filename, e.g. "5=seq5.shape,3=seq3.shape" will assign the reactivity values from file seq5.shape to the fifth sequence in the alignment, and the values from file seq3.shape to sequence 3. If no assignment is specified, the reactivity values are assigned to corresponding sequences in the order they are given.

--shapeMethod=D[mX][bY]

Specify the method how to convert SHAPE reactivity data to pseudo energy contributions.

(default="D")

Currently, the only data conversion method available is that of to Deigan et al 2009. This method is the default and is recognized by a capital `D` in the provided parameter, i.e.: *--shapeMethod=*"D" is the default setting. The slope `m` and the intercept `b` can be set to a non-default value if necessary. Otherwise m=1.8 and b=-0.6 as stated in the paper mentionen before. To alter these parameters, e.g. m=1.9 and b=-0.7, use a parameter string like this: *--shapeMethod=*"Dm1.9b-0.7". You may also provide only one of the two parameters like: *--shapeMethod=*"Dm1.9" or *--shapeMethod=*"Db-0.7".

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--cfactor=DOUBLE

Set the weight of the covariance term in the energy function

(default="1.0")

--nfactor=DOUBLE

Set the penalty for non-compatible sequences in the covariance term of the energy function

(default="1.0")

-R, **--ribosum_file**=ribosumfile

use specified Ribosum Matrix instead of normal

energy model.

Matrixes to use should be 6x6 matrices, the order of the terms is `AU`, `CG`, `GC`, `GU`, `UA`, `UG`.

-r, **--ribosum_scoring**

use ribosum scoring matrix. *(default=off)*

The matrix is chosen according to the minimal and maximal pairwise identities of the sequences in the file.

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp=*"-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

Plotting:

Command line options for changing the default behavior of structure layout and pairing probability plots

--aln-EPS[=prefix]

Produce colored and structure annotated subalignment for each hit.

The default file name used for the output is "aln_start_end.eps" where "start" and "end" denote the first and last column of the subalignment relative to the input (1-based). Users may change the filename to "prefix_aln_start_end.eps" by specifying the prefix as optional argument. Files will be create in the current directory. Note: Any special characters in the prefix will be replaced by the filename delimiter, hence there is no way to pass an entire directory path through this option yet. (See also the "–filename-delim" parameter)

--aln-EPS-cols=INT

Number of columns in colored EPS alignment output.

(default="60")

A value less than 1 indicates that the output should not be wrapped at all.

--aln-EPS-ss[=prefix]

Produce colored consensus secondary structure plots in PostScript format.

The default file name used for the output is "ss_start_end.eps" where "start" and "end" denote the first and last column of the subalignment relative to the input (1-based). Users may change the filename to "prefix_ss_start_end.eps" by specifying the prefix as optional argument. Files will be create in the current directory. Note: Any special characters in the prefix will be replaced by the filename delimiter, hence there is no way to pass an entire directory path through this option yet. (See also the "–filename-delim" parameter)

--color-threshold=FLOAT

Set the threshold of maximum counter examples for coloring consensus structure plot.

(default="2")

Floating point numbers between 0 and 1 are treated as frequencies among all sequencesin the alignment. All other will be truncated to integer and used as absolute number of counter examples.

--color-min-sat=FLOAT

Set the minimum saturation for coloring consensus structure plot.

(default="0.2")

Floating point number >= 0 and smaller than 1.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

I.L. Hofacker, B. Priwitzer, and P.F. Stadler (2004), "Prediction of Locally Stable RNA Secondary Structures for Genome-Wide Surveys", Bioinformatics: 20, pp 186-190

Stephan H. Bernhart, Ivo L. Hofacker, Sebastian Will, Andreas R. Gruber, and Peter F. Stadler (2008), "RNAalifold: Improved consensus structure prediction for RNA alignments", BMC Bioinformatics: 9, pp 474

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.13 RNALfold

RNALfold - manual page for RNALfold 2.8.0

Synopsis

```
RNALfold [OPTION]...
```

DESCRIPTION

RNALfold 2.8.0

calculate locally stable secondary structures of RNAs

Compute locally stable RNA secondary structure with a maximal base pair span. For a sequence of length n and a base pair span of L the algorithm uses only O(n+L*L) memory and O(n*L*L) CPU time. Thus it is practical to "scan" very large genomes for short RNA structures. Output consists of a list of secondary structure components of size <= L, one entry per line. Each output line contains the predicted local structure its energy in kcal/mol and the starting position of the local structure.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-i, **--infile**=filename

Read a file instead of reading from stdin

The default behavior of RNALfold is to read input from stdin. Using this parameter the user can specify an input file name where data is read from.

-o, **--outfile**[=filename]

Print output to file instead of stdout.

This option may be used to write all output to output files rather than printing to stdout. The number of output files created for batch input (multiple sequences) depends on three conditions: (i) In case an optional filename is given as parameter argument, a single file with the specified filename will be written into. If the optional argument is omitted, (ii) FASTA input or an active *--auto-id* switch will write to multiple files that follow the naming scheme "prefix.lfold". Here, "prefix" is taken from the sequence id as specified in the FASTA header. Lastly, (iii) single-line sequence input without FASTA header will be written to a single file "RNALfold_output.lfold". In case an output file already exists, any output of the program will be appended to it. Since the filename argument is optional, it must immediately follow the short option flag to not be mistaken as new parameter to the program. For instance ``-ornafold.out`` will write to a file "rnafold.out". Note: Any special characters in the filename will be replaced by the filename delimiter, hence there is no way to pass an entire directory path through this option yet. (See also the "–filename-delim" parameter)

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNALfold is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNALfold ignores any IDs retrieved from the input and automatically generates an ID for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="sequence")

If this parameter is set, each sequence will be prefixed with the provided string. Hence, the output files will obey the following naming scheme: "prefix_xxxx.lfold" where xxxx is the sequence number. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--filename-full

Use full FASTA header to create filenames. *(default=off)*

This parameter can be used to deactivate the default behavior of limiting output filenames to the first word of the sequence ID. Consider the following example: An input with FASTA header `>NM_0001 Homo Sapiens some gene` usually produces output files with the prefix "NM_0001" without the additional data available in the FASTA header, e.g. "NM_0001_ss.ps" for secondary structure plots. With this flag set, no truncation of the output filenames is done, i.e. output filenames receive the full FASTA header data as prefixes. Note, however, that invalid characters (such as whitespace) will be substituted by a delimiting character or simply removed, (see also the parameter option *--filename-delim*).

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNALfold.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations. The Minimum free energy (MFE) and a structure representative are calculated in any case.

-L, **--span**=INT

Set the maximum distance between any two pairing nucleotides.

(default="150")

This option specifies the window length L and therefore the upper limit for the distance between the bases i and j of any pair (i, j), i.e. (j - i + 1) <= L.

-z, **--zscore**[=DOUBLE]

Limit the output to predictions with a Z-score below a threshold.

(default="-2")

This option activates z-score regression using a trained SVM. Any predicted structure that exceeds the specified threshold will be ommited from the output. Since the Z-score threshold is given as a negative number, it must immediately preceed the short option to not be mistaken as a separate argument, e.g. `-z-2.9` sets the threshold to a value of `-2.9`

--zscore-pre-filter

Apply the z-score filtering in the forward recursions.

(default=off)

The default mode of z-score filtering considers the entire structure space to decide whether or not a locally optimal structure at any position i is reported or not. When using this post-filtering step, however, alternative locally optimal structures

starting at i with higher energy but lower z-score can be easily missed. The

pre-filter

option restricts the structure space already in the forward recursions, such

that

only optimal solution among those candidates that satisfy the z-score

threshold are considered. Therefore, good results according to the z-score threshold criterion are less likely to be superseded by results with better energy but worse z-score. Note, that activating this switch results in higher computation time which scales linear in the window length.

--zscore-report-subsumed

Report subsumed structures if their z-score is less than that of the enclosing structure.

(default=off)

In default mode, RNALfold only reports locally optimal structures if they are no constituents of another, larger structure with less free energy. In z-score mode, however, such a larger structure may have a higher z-score, thus may be less informative than the smaller substructure. Using this switch activates reporting both, the smaller and the larger structure if the z-score of the smaller is lower than that of the larger.

-b, **--backtrack-global**

Backtrack a global MFE structure. *(default=off)*

Instead of just reporting the locally stable secondary structure a global MFE structure can be constructed that only consists of locally optimal substructures. This switch activates a post-processing step that takes the locally optimal structures to generate the global MFE structure which constitutes the MFE value reported

in the last line. The respective global MFE structure is printed just after the inut sequence part on the last line, preceding the global MFE score. Note, that this option implies *-o*/–outfile since the locally optimal structures must be read after the regular prediction step! Also note, that using this option in combination with *-z*/–zscore implies *--zscore-pre-filter* to ensure proper construction of the global MFE structure!

-g, **--gquad**

Incoorporate G-Quadruplex formation into the structure prediction algorithm.

(default=off)

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--shape=filename

Use SHAPE reactivity data to guide structure predictions.

--shapeMethod=method

Select SHAPE reactivity data incorporation strategy.

(default="D")

The following methods can be used to convert SHAPE reactivities into pseudo energy contributions.

D: Convert by using the linear equation according to Deigan et al 2009.

Derived pseudo energy terms will be applied for every nucleotide involved in a stacked pair. This method is recognized by a capital D in the provided parameter, i.e.: *--shapeMethod=*"D" is the default setting. The slope m and the intercept b can be set to a non-default value if necessary, otherwise m=1.8 and b=-0.6. To alter these parameters, e.g. m=1.9 and b=-0.7, use a parameter string like this: *--shapeMethod=*"Dm1.9b-0.7". You may also provide only one of the two parameters like: *--shapeMethod=*"Dm1.9" or *--shapeMethod=*"Db-0.7".

Z: Convert SHAPE reactivities to pseudo energies according to Zarringhalam

et al 2012. SHAPE reactivities will be converted to pairing probabilities by using linear mapping. Aberration from the observed pairing probabilities will be penalized during the folding recursion. The magnitude of the penalties can affected by adjusting the factor beta (e.g. *--shapeMethod=*"Zb0.8").

W: Apply a given vector of perturbation energies to unpaired nucleotides

according to Washietl et al 2012. Perturbation vectors can be calculated by using RNApvmin.

--shapeConversion=method

Select method for SHAPE reactivity conversion.

(default="O")

This parameter is useful when dealing with the SHAPE incorporation according to Zarringhalam et al. The following methods can be used to convert SHAPE reactivities into the probability for a certain nucleotide to be unpaired.

M: Use linear mapping according to Zarringhalam et al. C: Use a cutoff-approach to divide into paired and unpaired nucleotides (e.g. "C0.25") S: Skip the normalizing step since the input data already represents probabilities for being unpaired rather than raw reactivity values L: Use a linear model to convert the reactivity into a probability for being unpaired (e.g. "Ls0.68i0.2" to use a slope of 0.68 and an intercept of 0.2) O: Use a linear model to convert the log of the reactivity into a probability for being unpaired (e.g. "Os1.6i-2.29" to use a slope of 1.6 and an intercept of -2.29)

--commands=filename

Read additional commands from file

Commands include hard and soft constraints, but also structure motifs in hairpin and internal loops that need to be treeted differently. Furthermore, commands can be set for unstructured and structured domains.

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

-m, **--modifications**[=STRING]

Allow for modified bases within the RNA sequence string.

(default="7I6P9D")

Treat modified bases within the RNA sequence differently, i.e. use corresponding energy corrections and/or pairing partner rules if available. For that, the modified bases in the input sequence must be marked by their corresponding one-letter code. If no additional arguments are supplied, all available corrections are performed. Otherwise, the user may limit the modifications to a particular subset of modifications, resp. one-letter codes, e.g. `-mP6` will only correct for pseudouridine and m6A bases.

Currently supported one-letter codes and energy corrections are:

`7`: 7-deaza-adenonsine (7DA)

`I`: Inosine

`6`: N6-methyladenosine (m6A)

`P`: Pseudouridine

`9`: Purine (a.k.a. nebularine)

`D`: Dihydrouridine

--mod-file=STRING

Use additional modified base data from JSON file.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the

moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

I.L. Hofacker, B. Priwitzer, and P.F. Stadler (2004), "Prediction of Locally Stable RNA Secondary Structures for Genome-Wide Surveys", Bioinformatics: 20, pp 186-190

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Peter F Stadler, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

SEE ALSO

RNAplfold(1) RNALalifold(1)

2.2.14 RNAmultifold

RNAmultifold - manual page for RNAmultifold 2.8.0

Synopsis

```
RNAmultifold [OPTION]... [FILE]...
```

DESCRIPTION

RNAmultifold 2.8.0

Compute secondary structures of multiple interacting RNAs

The program works much like RNAfold, but allows one to specify multiple RNA sequences which are then allowed to form conncected components. RNA sequences are read from stdin in the usual format, i.e. each line of input corresponds to one sequence, except for lines starting with ">" which contain the name of the next sequence(s). Multiple strands must be concatenated using the ``&`` character as separator. RNAmultifold can compute MFE, partition function, corresponding ensemble free energy and base pairing probabilities. These properties are either computed for a particular arrangement (concatenation) of sequences, for the full ensemble of the complex of input RNAs, or all complexes formed by the input sequences up to a specified number of interacting sequences. Output consists of a PostScript "dot plot" file containing the pair probabilities, see the RNAfold man page for details. The program will continue to read new sequences until a line consisting of the single character `@` or an end of file condition is encountered.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-j, **--jobs**[=number]

Split batch input into jobs and start processing in parallel using multiple threads. A value of 0 indicates to use as many parallel threads as computation cores are available.

(default="0")

Default processing of input data is performed in a serial fashion, i.e. one sequence pair at a time. Using this switch, a user can instead start the computation for many sequence pairs in the input in parallel. RNAmultifold will create as many parallel computation slots as specified and assigns input sequences of the input file(s) to the available slots. Note, that this increases memory consumption since input alignments have to be kept in memory until an empty compute slot is available and each running job requires its own dynamic programming matrices.

--unordered

Do not try to keep output in order with input while parallel processing is in place.

(default=off)

When parallel input processing (*--jobs* flag) is enabled, the order in which input is processed depends on the host machines job scheduler. Therefore, any output to stdout or files generated by this program will most likely not follow the order of the corresponding input data set. The default of RNAmultifold is to use a specialized data structure to still keep the results output in order with the input data. However, this comes with a trade-off in terms of memory consumption, since all output must be kept in memory for as long as no chunks of consecutive, ordered output are available. By setting this flag, RNAmultifold will not buffer individual results but print them as soon as they have been computated.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNAmultifold is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNAmultifold ignores any IDs retrieved from the input and automatically generates an ID for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="sequence")

If this parameter is set, each sequence will be prefixed with the provided string. Hence, the output files will obey the following naming scheme: "prefix_xxxx_ss.ps" (secondary structure plot), "prefix_xxxx_dp.ps" (dot-plot), "prefix_xxxx_dp2.ps" (stack probabilities), etc. where xxxx is the sequence number. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--filename-full

Use full FASTA header to create filenames. *(default=off)*

This parameter can be used to deactivate the default behavior of limiting output filenames to the first word of the sequence ID. Consider the following example: An input with FASTA header `>NM_0001 Homo Sapiens some gene` usually produces output files with the prefix "NM_0001" without the additional data available in the FASTA header, e.g. "NM_0001_ss.ps" for secondary structure plots. With this flag set, no truncation of the output filenames is done, i.e. output filenames receive the full FASTA header data as prefixes. Note, however, that invalid characters (such as whitespace) will be substituted by a delimiting character or simply removed, (see also the parameter option *--filename-delim*).

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAmultifold.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations. The Minimum free energy (MFE) and a structure representative are calculated in any case.

-p, **--partfunc**[=INT]

Calculate the partition function and base pairing probability matrix in addition to the MFE structure. Default is calculation of mfe structure only.

(default="1")

In addition to the MFE structure we print a coarse representation of the pair probabilities in form of a pseudo bracket notation, followed by the ensemble free energy. Note that unless you also specify `-d2` or `-d0`, the partition function and mfe calculations will use a slightly different energy model. See the discussion of dangling end options below.

An additionally passed value to this option changes the behavior of partition function calculation:

In order to calculate the partition function but not the pair probabilities

use the `-p0` option and save about

50% in runtime. This prints the ensemble free energy `dG=-kT ln(Z)`.

-a, **--all_pf**[=INT]

Compute the partition function and free energies not only for the complex formed by the input sequences (the "ABC... mutimer"), but also of all complexes formed by the input sequences up to the number of input sequences, e.g. AAA, AAB, ABB, BBB, etc.

(default="1")

The output will contain the free energies for each of these species. Using *-a* automatically switches on the *-p* option.

-c, **--concentrations**

In addition to everything listed under the *-a* option, read in initial monomer concentrations and compute the expected equilibrium concentrations of all possible species (A, B, AA, BB, AB, etc).

(default=off)

Start concentrations are read from stdin (unless the *-f* option is used) in [mol/l], equilibrium concentrations are given realtive to the sum of the inputs. An arbitrary number of initial concentrations can be specified (one tuple of concentrations per line).

-f, **--concfile**=filename

Specify a file with initial concentrations for the input sequences.

The table consits of arbitrary many lines with multiple numbers separated by whitespace (the concentration of the input sequences A, B, C, etc.). This option will automatically toggle the *-c* (and thus *-a* and *-p*) options (see above).

--absolute-concentrations Report absolute instead of relative

concentrations

(default=off)

--betaScale=DOUBLE

Set the scaling of the Boltzmann factors. *(default="1.")*

The argument provided with this option is used to scale the thermodynamic temperature in the Boltzmann factors independently from the temperature of the individual loop energy contributions. The Boltzmann

factors then become `exp(- dG/(kT*betaScale))` where `k` is the Boltzmann constant, `dG` the free energy contribution of the state and `T` the absolute temperature.

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

--bppmThreshold=cutoff

Set the threshold/cutoff for base pair probabilities included in the postscript output.

(default="1e-5")

By setting the threshold the base pair probabilities that are included in the output can be varied. By default only those exceeding `1e-5` in probability will be shown as squares in the dot plot. Changing the threshold to any other value allows for increase or decrease of data.

-g, **--gquad**

Incoorporate G-Quadruplex formation into the structure prediction algorithm.

(default=off)

Note, only intramolecular G-quadruplexes are considered.

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--maxBPspan=INT

Set the maximum base pair span.

(default="-1")

--commands=filename

Read additional commands from file

Commands include hard and soft constraints, but also structure motifs in hairpin and internal loops that need to be treeted differently. Furthermore, commands can be set for unstructured and structured domains.

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.15 RNApaln

RNApaln - manual page for RNApaln 2.8.0

Synopsis

```
RNApaln [OPTION]...
```

DESCRIPTION

RNApaln 2.8.0

RNA alignment based on sequence base pairing propensities

Uses string-alignment techniques to perform fast pairwise structural alignments of RNAs. Similar to RNApdist secondary structure is incorporated in an approximate manner by computing base pair probabilities, which are then reduced to a vector holding the probability that a base is paired upstream, downstream, or remains unpaired. Such pair propsensity vectors can then be compared using standard alignment algorithms. In contrast to RNApdist, RNApaln performs similarity (instead of distance) alignments, considers both sequence and structure information, and uses affine (rather than linear) gap costs. RNApaln can perform semi-local alignments by using free end gaps, a true local alignment mode is planned.

The same approach has since been used in the StraL program from Gerhard Steeger's group. Since StraL has optimized parameters and a multiple alignment mode, it be be currently the better option.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-B, **--printAlignment**[=filename]

Print an "alignment" with gaps of the

profiles

The aligned structures are written to filename, if specified Otherwise output is written to stdout, unless the **-Xm** option is set in which case "backtrack.file" is used.

(default="stdout")

The following symbols are used:

(

) essentially upstream (downstream) paired bases

{

} weakly upstream (downstream) paired bases

|

strongly paired bases without preference

,

weakly paired bases without preference

.

essentially unpaired bases.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNApaln.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations.

-X, **--mode**=pmfc

Set the alignment mode to be used.

The alignment mode is passed as a single character value. The following options are available: `p` - Compare the structures pairwise, that is first with 2nd, third with 4th etc. This is the default.

``m``

- Calculate the distance matrix between all structures. The output is

formatted as a lower triangle matrix.

`f` - Compare each structure to the first one.

`c` - Compare continuously, that is i-th with (i+1)th structure.

--gapo=open

Set the gap open penalty

--gape=ext

Set the gap extension penalty

--seqw=w

Set the weight of sequence (compared to structure) in the scoring function.

--endgaps

Use free end-gaps

(default=off)

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. When passing the placeholder file name "DNA", DNA parameters are loaded without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs. e.g. RNAfold `-nsp -GA` will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

Bonhoeffer S, McCaskill J S, Stadler P F, Schuster P (1993), "RNA multi-structure landscapes", Euro Biophys J: 22, pp 13-24

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Peter F Stadler, Ivo L Hofacker, Sebastian Bonhoeffer

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.16 RNAparconv

RNAparconv - manual page for RNAparconv 2.8.0

Synopsis

```
RNAparconv [options] [<input file>] [<output file>]
```

DESCRIPTION

RNAparconv 2.8.0

Convert energy parameter files from ViennaRNA 1.8.4 to 2.0 format

Converts energy parameter files from "old" ViennaRNAPackage 1.8.4 format to the new format used since ViennaRNAPackage 2.0. The Program reads a valid energy parameter file or valid energy parameters from stdin and prints the converted energy parameters to stdout or a specified output file. Per default, the converted output file contains the whole set of energy parameters used throughout ViennaRNAPackage 1.8.4. The user can specify sets of energy parameters that should not be included in the output.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-i, **--input**=filename

Specify an input file name. If argument is missing the energy parameter input can be supplied via `stdin`.

-o, **--output**=filename

Specify an output file name. If argument is missing the converted energy parameters are printed to `stdout`.

--vanilla

Print just as much as needed to represent the given energy parameters data set. This option overrides all other output settings!

(default=off)

--dump

Just dump Vienna 1.8.4 energy parameters in format used since 2.0. This option skips any energy parameter input!

(default=off)

--silent

Print just energy parameters and appropriate comment lines but suppress all other output

(default=off)

--without-HairpinE

Do not print converted hairpin energies and enthalpies

(default=off)

--without-StackE

Do not print converted stacking energies and enthalpies

(default=off)

--without-IntE

Do not print converted internal loop energies, enthalpies and asymetry factors

(default=off)

--without-BulgeE

Do not print converted bulge loop energies and enthalpies

(default=off)

--without-MultiE

Do not print converted multi loop energies and enthalpies

(default=off)

--without-MismatchE

Do not print converted exterior loop mismatch energies and enthalpies

(default=off)

--without-MismatchH

Do not print converted hairpin mismatch energies and enthalpies

(default=off)

--without-MismatchI

Do not print converted internal loop mismatch energies and enthalpies

(default=off)

--without-MismatchM

Do not print converted multi loop mismatch energies and enthalpies

(default=off)

--without-Dangle5

Do not print converted 5' dangle energies and enthalpies

(default=off)

--without-Dangle3

Do not print converted 3' dangle energies and enthalpies

(default=off)

--without-Misc

Do not print converted Misc energies and enthalpies (TerminalAU, DuplexInit, lxc)

(default=off)

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAparconv.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.17 RNApdist

RNApdist - manual page for RNApdist 2.8.0

Synopsis

```
RNApdist [OPTION]...
```

DESCRIPTION

RNApdist 2.8.0

Calculate distances between thermodynamic RNA secondary structures ensembles

This program reads RNA sequences from stdin and calculates structure distances between the thermodynamic ensembles of their secondary structures.

To do this the partition function and matrix of base pairing probabilities is computed for each sequence. The probability matrix is then condensed into a vector holding for each base the probabilities of being unpaired, paired upstream, or paired downstream, respectively. These profiles are compared by a standard alignment algorithm.

The base pair probabilities are also saved as postscript "dot plots" (as in RNAfold) in the files "name_dp.ps", where name is the name of the sequence, or a number if unnamed.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNApdist.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations.

-X, **--compare**=p|m|f|c

Specify the comparison directive. *(default="p")*

Possible arguments for this option are: **-Xp** compare the structures pairwise (p), i.e. first with 2nd, third with 4th etc. **-Xm** calculate the distance matrix between all structures. The output is formatted as a lower triangle matrix. **-Xf** compare each structure to the first one. **-Xc** compare continuously, that is i-th with (i+1)th structure.

-B, **--backtrack**[=<filename>]

Print an "alignment" with gaps of the profiles. The aligned structures are written to <filename>, if specified.

(default="none")

Within the profile output, the following symbols will be used:

()

essentially upstream (downstream) paired bases

{}

weakly upstream (downstream) paired bases

|

strongly paired bases without preference

,

weakly paired bases without preference

.

essentially unpaired bases.

If <filename> is not specified, the output is written to stdout, unless the

"-Xm" option is set in which case "backtrack.file" is used.

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. When passing the placeholder file name "DNA", DNA parameters are loaded without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

set energy model for treatment of dangling bases.

(possible values="0", "2" default="2")

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs. e.g. RNAfold **-nsp** **-GA** will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

S. Bonhoeffer, J.S. McCaskill, P.F. Stadler, P. Schuster (1993), "RNA multi-structure landscapes", Euro Biophys J:22, pp 13-24

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Peter F Stadler, Ivo L Hofacker, Sebastian Bonhoeffer.

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.18 RNAPKplex

RNAPKplex - manual page for RNAPKplex 2.8.0

Synopsis

```
RNAPKplex [OPTION]...
```

DESCRIPTION

RNAPKplex 2.8.0

predicts RNA secondary structures including pseudoknots

Computes RNA secondary structures by first making two sequence intervals accessible and unpaired using the algorithm of RNAplfold and then calculating the energy of the interaction of those two intervals. The algorithm uses O(n^2*w^4) CPU time and O(n*w^2) memory space. The algorithm furthermore always considers dangle=2 model.

It also produces a PostScript file with a plot of the pseudoknot-free secondary structure graph, in which the bases forming the pseuodknot are marked red.

Sequences are read in a simple text format where each sequence occupies a single line. Each sequence may be preceded by a line of the form .. code:

```
> name
```

to assign a name to the sequence. If a name is given in the input, the PostScript file "name.ps" is produced for the structure graph. Other- wise the file name defaults to PKplex.ps. Existing files of the same name will be overwritten. The input format is similar to fasta except that even long sequences may not be interrupted by line breaks, and the header lines are optional. The program will continue to read new sequences until a line consisting of the single character @ or an end of file condition is encountered.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAPKplex.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations.

-c, **--cutoff**=FLOAT

Only consider unpaired probabities > cutoff for putative PK sites.

(default="1e-6")

-e, **--energyCutoff**=DOUBLE

Energy cutoff or pseudoknot initiation cost. Minimum energy gain of a pseudoknot interaction for it to be returned. Pseudoknots with smaller energy gains are rejected.

(default="-8.10")

-s, **--subopts**=DOUBLE

print suboptimal structures whose energy difference of the pseudoknot to the optimum pseudoknot is smaller than the given value.

(default="0.0")

NOTE: The final energy of a structure is calculated as the sum of the pseudoknot interaction energy, the penalty for initiating a pseudoknot and the energy of the pseudoknot-free part of the structure. The `-s` option only takes the pseudoknot interaction energy into account, so the final energy differences may be bigger than the specified value *(default=0.)*.

--betaScale=DOUBLE

Set the scaling of the Boltzmann factors. *(default="1.")*

The argument provided with this option is used to scale the thermodynamic temperature in the Boltzmann factors independently from the temperature of the individual loop energy contributions. The Boltzmann factors then become `exp(- dG/(kT*betaScale))` where `k` is the Boltzmann constant, `dG` the free energy contribution of the state and `T` the absolute temperature.

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp=*"-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help
display help, including hidden options, and exit

--detailed-help
display help, including all details and hidden options, and exit

-V, **--version**
output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Wolfgang Beyer

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.19 RNAplex

RNAplex - manual page for RNAplex 2.8.0

Synopsis

```
RNAplex [options]
```

DESCRIPTION

RNAplex 2.8.0

Find targets of a query RNA

reads two RNA sequences from stdin or <filename> and computes optimal and suboptimal secondary structures for their hybridization. The calculation is simplified by allowing only inter-molecular base pairs. Accessibility effects can be estimated by RNAplex if a RNAplfold accessibility profile is provided. The computed optimal and suboptimal structure are written to stdout, one structure per line. Each line consist of: The structure in dot bracket format with a "&" separating the two strands. The range of the structure in the two sequences in the format "from,to : from,to"; the energy of duplex structure in kcal/mol. The format is especially useful for computing the hybrid structure between a small probe sequence and a long target sequence.

-v, **--verbose**
Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-q, **--query**=STRING

File containing the query sequence.

Input sequences can be given piped to RNAplex or given in a query file with the *-q* option. Note that the *-q* option implies that the *-t* option is also used

-t, **--target**=STRING

File containing the target sequence.

Input sequences can be given piped to RNAplex or given in a target file with the *-t* option. Note that the *-t* option implies that the *-q* option is also used

-a, **--accessibility-dir**=STRING

Location of the accessibility profiles.

This option switches the accessibility modes on and indicates in which directory accessibility profiles as generated by RNAplfold can be found

-b, **--binary**

Allow the reading and parsing of memory dumped opening energy file

(default=off)

The *-b* option allows one to read and process opening energy files which are saved in binary format

This can reduce by a factor of 500x-1000x the time needed to process those

files. RNAplex recognizes the corresponding opening energy files by looking for files named after the sequence and containing the suffix _openen_bin. Please look at the man page of RNAplfold if you need more information on how to produce binary opening energy files.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAplex.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Options which alter the computing behaviour of RNAplex.

-l, **--interaction-length**=INT

Maximal length of an interaction *(default="40")*

Maximal allowed length of an interaction

-c, **--extension-cost**=INT

Cost to add to each nucleotide in a duplex *(default="0")*

Cost of extending a duplex by one nucleotide. Allows one to find compact duplexes, having few/small bulges or internal loops Only useful when no accessibility profiles are available. This option is disabled if accessibility profiles are used (`-a` option)

-p, **--probe-mode**

Compute Tm for probes *(default=off)*

Use this option if you want to compute the melting temperature of your probes

-Q, **--probe-concentration**=DOUBLE

Set the probe concentration for the Tm

computation

(default="0.1")

-N, **--na-concentration**=DOUBLE Set the Na+ concentration for the Tm

computation.

(default="1.0")

-M, **--mg-concentration**=DOUBLE Set the Mg2+ concentration for the Tm

computation.

(default="1.0")

-K, **--k-concentration**=DOUBLE

Set the K+ concentration for the Tm computation.

(default="1.0")

-U, **--tris-concentration**=DOUBLE

Set the tris+ concentration for the Tm

computation.

(default="1.0")

-f, **--fast-folding**=INT

Speedup of the target search *(default="0")*

This option allows one to decide if the backtracking has to be done (`-f` 0, `-f` 2) or not (`-f` 1). For `-f` 0 the structure is computed based on the standard energy model. This is the slowest and most precise mode of RNAplex. With `-f` 2, the structure is computed based on the approximated plex model. If a lot of targets are returned this is can greatly improve the runtime of RNAplex. `-f` 1 is the fastest mode, as no structure are recomputed

-V, **--scale-accessibility**=DOUBLE

Rescale all opening energy by a factor V

(default="1.0")

Scale-factor for the accessibility. If V is set to 1 then the scaling has no effect on the accessibility.

-A, **--alignment-mode**

Tells RNAplex to compute interactions based on alignments

(default=off)

If the A option is set RNAplex expects clustalw files as input for the `-q` and `-t` option.

-k, **--convert-to-bin**

If set, RNAplex will convert all opening energy file in a directory set by the *-a* option into binary opening energy files

(default=off)

RNAplex can be used to convert existing text formatted opening energy files into binary formatted files. In this mode RNAplex does not compute interactions.

-z, **--duplex-distance**=INT

Distance between target 3' ends of two consecutive duplexes

(default="0")

Distance between the target 3'ends of two consecutive duplexes. Should be set to the maximal length of interaction to get good results

Smaller z leads to larger overlaps between consecutive duplexes.

-e, **--energy-threshold**=DOUBLE Minimal energy for a duplex to be returned

(default="-100000")

Energy threshold for a duplex to be returned. The threshold is set on the total energy of interaction, i.e. the hybridization energy corrected for opening energy if *-a* is set or the energy corrected by *-c*. If unset, only the mfe will be returned

-L, **--WindowLength**=INT

Tells how large the region around the target site should be for redrawing the alignment interaction

(default="1")

This option allows one to specify how large the region surrounding the target site should be set when generating the alignment figure of the interaction

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

-C, **--constraint**

Calculate structures subject to constraints. *(default=off)*

The program reads first the sequence, then a string containing constraints on the structure for the query sequence encoded with the symbols: . (no constraint for this base) | (the corresponding base has to be paired)

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. When passing the placeholder file name "DNA", DNA parameters are loaded without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

--saltInit=DOUBLE

Provide salt correction for duplex initialization (in kcal/mol).

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

Plotting:

Command line options for changing the default behavior of structure layout and pairing probability plots

-I, **--produce-ps**=STRING

Draw an alignment annotated interaction from RNAplex.

This option allows one to produce interaction figures in PS-format a la RNAalifold, where base-pair conservation is represented in color-coded format. In this mode no interaction are computed, but the *-I* option indicates the location of the file containing interactions between two RNA (alignments/sequence) from a previous run. If the *-A* option is not set a structure figure a la RNAfold with color-coded annotation of the accessibilities is returned

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

--version

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The calculation of duplex structure is based on dynamic programming algorithm originally developed by Rehmsmeier and in parallel by Hofacker.

H. Tafer and I.L. Hofacker (2008), "RNAplex: a fast tool for RNA-RNA interaction search.", Bioinformatics: 24(22), pp 2657-2663

S. Bonhoeffer, J.S. McCaskill, P.F. Stadler, P. Schuster (1993), "RNA multi-structure landscapes", Euro Biophys J: 22, pp 13-24

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Hakim Tafer, Ivo L. Hofacker

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.20 RNAplfold

RNAplfold - manual page for RNAplfold 2.8.0

Synopsis

```
RNAplfold [OPTION]...
```

DESCRIPTION

RNAplfold 2.8.0

calculate locally stable secondary structure - pair probabilities

Computes local pair probabilities for base pairs with a maximal span of L. The probabilities are averaged over all windows of size L that contain the base pair. For a sequence of length n and a window size of L the algorithm uses only O(n+L*L) memory and O(n*L*L) CPU time. Thus it is practical to "scan" very large genomes for short stable RNA structures.

Output consists of a dot plot in postscript file, where the averaged pair probabilities can easily be parsed and visually inspected.

The -u option makes i possible to compute the probability that a stretch of x consequtive nucleotides is unpaired, which is useful for predicting possible binding sites. Again this probability is averaged over all windows containing the region.

.B WARNING! Output format changed!!

The output is a plain text matrix containing on each line a position i followed by the probability that i is unpaired, [i-1..i] is unpaired [i-2..i] is unpaired and so on to the probability that [i-x+1..i] is unpaired.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-c, **--cutoff**=FLOAT

Report only base pairs with an average probability larger than `cutoff` in the dot plot.

(default="0.01")

-o, **--print_onthefly**

Save memory by printing out everything during computation.

(default=off)

NOTE: activated per default for sequences over 1M bp.

-0, **--opening_energies**

Switch output from probabilities to their logarithms.

(default=off)

This is NOT exactly the mean energies needed to unfold the respective stretch of bases! (implies *--ulength* option).

--plex_output

Create additional output files for RNAplex.

(default=off)

-b, **--binaries**

Output accessibility profiles in binary format. *(default=off)*

The binary files produced by RNAplfold do not need to be parsed by RNAplex,

so that they are directly loaded into memory. This is useful when large sequences have to be searched for putative hybridization sites. Another advantage of the binary format is the 50% file size decrease.

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNAplfold is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNAplfold ignores any IDs retrieved from the input and automatically generates an ID for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="sequence")

If this parameter is set, each sequences' FASTA id will be prefixed with the provided string. FASTA ids then take the form ">prefix_xxxx" where xxxx is the sequence number. Hence, the output files will obey the following naming scheme: "prefix_xxxx_dp.ps" (dot-plot), "prefix_xxxx_lunp" (unpaired probabilities), etc. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--filename-full

Use full FASTA header to create filenames. *(default=off)*

This parameter can be used to deactivate the default behavior of limiting output filenames to the first word of the sequence ID. Consider the following example: An input with FASTA header `>NM_0001 Homo Sapiens some gene` usually produces output files with the prefix "NM_0001" without the additional data available in the FASTA header, e.g. "NM_0001_ss.ps" for secondary structure plots. With this flag set, no truncation of the output filenames is done, i.e. output filenames receive the full FASTA header data as prefixes. Note, however, that invalid characters (such as whitespace) will be substituted by a delimiting character or simply removed, (see also the parameter option *--filename-delim*).

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAplfold.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select and change parameters of (additional) algorithms which should be included in the calculations.

-W, **--winsize**=size

Average the pair probabilities over windows of given size.

(default="70")

-L, **--span**=size

Set the maximum allowed separation of a base pair to span.

By setting the maximum base pair span no pairs (i,j) with j-i > span will be allowed. Defaults to winsize if parameter is omitted.

-u, **--ulength**=length

Compute the mean probability that regions of length 1 to a given length are unpaired.

(default="31")

Output is saved in a `_lunp` file.

--betaScale=DOUBLE

Set the scaling of the Boltzmann factors. *(default="1.")*

The argument provided with this option is used to scale the thermodynamic temperature in the Boltzmann factors independently from the temperature of the individual loop energy contributions. The Boltzmann factors then become `exp(- dG/(kT*betaScale))` where `k` is the Boltzmann constant, `dG` the free energy contribution of the state and `T` the absolute temperature.

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--shape=filename

Use SHAPE reactivity data to guide structure predictions.

--shapeMethod=method

Select SHAPE reactivity data incorporation strategy.

(default="D")

The following methods can be used to convert SHAPE reactivities into pseudo energy contributions.

`D`: Convert by using the linear equation according to Deigan et al 2009.

Derived pseudo energy terms will be applied for every nucleotide involved in a stacked pair. This method is recognized by a capital `D` in the provided parameter, i.e.: *--shapeMethod=*"D" is the default setting. The slope `m` and the intercept `b` can be set to a non-default value if necessary, otherwise m=1.8 and b=-0.6. To alter these parameters, e.g. m=1.9 and b=-0.7, use a parameter string like this: *--shapeMethod=*"Dm1.9b-0.7". You may also provide only one of the two parameters like: *--shapeMethod=*"Dm1.9" or *--shapeMethod=*"Db-0.7".

Z: Convert SHAPE reactivities to pseudo energies according to Zarringhalam

et al 2012. SHAPE reactivities will be converted to pairing probabilities by using linear mapping. Aberration from the observed pairing probabilities will be penalized during the folding recursion. The magnitude of the penalties can affected by adjusting the factor beta (e.g. *--shapeMethod=*"Zb0.8").

W: Apply a given vector of perturbation energies to unpaired nucleotides

according to Washietl et al 2012. Perturbation vectors can be calculated by using RNApvmin.

--shapeConversion=method

Select method for SHAPE reactivity conversion.

(default="O")

This parameter is useful when dealing with the SHAPE incorporation according to Zarringhalam et al. The following methods can be used to convert SHAPE reactivities into the probability for a certain nucleotide to be unpaired.

M: Use linear mapping according to Zarringhalam et al. C: Use a cutoff-approach to divide into paired and unpaired nucleotides (e.g. "C0.25") S: Skip the normalizing step since the input data already represents probabilities for being unpaired rather than raw reactivity values L: Use a linear model to convert the reactivity into a probability for being unpaired (e.g. "Ls0.68i0.2" to use a slope of 0.68 and an intercept of 0.2) O: Use a linear model to convert the log of the reactivity into a probability for being unpaired (e.g. "Os1.6i-2.29" to use a slope of 1.6 and an intercept of -2.29)

--commands=filename

Read additional commands from file

Commands include hard and soft constraints, but also structure motifs in hairpin and internal loops that need to be treeted differently. Furthermore, commands can be set for unstructured and structured domains.

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name DNA can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

-m, **--modifications**[=STRING]

Allow for modified bases within the RNA sequence string.

(default="7I6P9D")

Treat modified bases within the RNA sequence differently, i.e. use corresponding energy corrections and/or pairing partner rules if available. For that, the modified bases in the input sequence must be marked by their corresponding one-letter code. If no additional arguments are supplied, all available corrections are

performed. Otherwise, the user may limit the modifications to a particular subset of modifications, resp. one-letter codes, e.g. `-mP6` will only correct for pseudouridine and m6A bases.

Currently supported one-letter codes and energy corrections are:

`7`: 7-deaza-adenonsine (7DA)

`I`: Inosine

`6`: N6-methyladenosine (m6A)

`P`: Pseudouridine

`9`: Purine (a.k.a. nebularine)

`D`: Dihydrouridine

--mod-file=STRING

Use additional modified base data from JSON file.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

Specify "dangling end" model for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d2` dangling energies will be added for the bases adjacent to a helix on both sides in any case while `-d0` ignores dangling ends altogether (mostly for debugging).

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

S. H. Bernhart, U. Mueckstein, and I.L. Hofacker (2011), "RNA Accessibility in cubic time", Algorithms Mol Biol. 6: 3.

S. H. Bernhart, I.L. Hofacker, and P.F. Stadler (2006), "Local Base Pairing Probabilities in Large RNAs", Bioinformatics: 22, pp 614-615

A.F. Bompfuenewerer, R. Backofen, S.H. Bernhart, J. Hertel, I.L. Hofacker, P.F. Stadler, S. Will (2007), "Variations on RNA Folding and Alignment: Lessons from Benasque", J. Math. Biol.

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Stephan H Bernhart, Ivo L Hofacker, Peter F Stadler, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

SEE ALSO

RNALfold(1)

2.2.21 RNAplot

RNAplot - manual page for RNAplot 2.8.0

Synopsis

```
RNAplot [OPTIONS] [<input0>] [<input1>]...
```

DESCRIPTION

RNAplot 2.8.0

Draw RNA Secondary Structures

The program reads (aligned) RNA sequences and structures in the format as produced by RNAfold or Stockholm 1.0 and produces drawings of the secondary structure graph. Coordinates for the structure graphs are produced using either E. Bruccoleri's naview routines, or a simple radial layout method. For aligned sequences and consensus structures (*--msa* option) the graph may be annotated by covariance information. Additionally, a color-annotated EPS alignment figure can be produced, similar to that obtained by RNAalifold and RNALalifold. If the sequence was preceded by a FASTA header, or if the multiple sequence alignment contains an ID field, these IDs will be taken as names for the output file(s): "name_ss.eps" and "name_aln.eps". Otherwise "rna.eps" and "aln.eps" will be used. This behavior may be over-ruled by explicitly setting a filename prefix using the *--auto-id* option. Existing files of the same name will be overwritten.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-i, **--infile**=<filename>

Read a file instead of reading from stdin.

The default behavior of RNAplot is to read input from stdin or the file(s) that follow(s) the RNAplot command. Using this parameter the user can specify input file names where data is read from. Note, that any additional files supplied to RNAplot are still processed as well.

-a, **--msa**

Input is multiple sequence alignment in Stockholm 1.0 format. *(default=off)*

Using this flag indicates that the input is a multiple sequence alignment (MSA) instead of (a) single sequence(s). Note, that only STOCKHOLM format allows one to specify a consensus structure. Therefore, this is the only supported MSA format for now!

--mis

Output "most informative sequence" instead of simple consensus *(default=off)*

For each column of the alignment output this is the set of nucleotides with frequency greater than average in IUPAC notation.

-j, **--jobs**[=number]

Split batch input into jobs and start processing in parallel using multiple threads. *(default="0")*

Default processing of input data is performed in a serial fashion, i.e. one sequence at a time. Using this switch, a user can instead start the computation for many sequences in the input in parallel. RNAplot will create as many parallel computation slots as specified and assigns input sequences of the input file(s) to the

available slots. Note, that this increases memory consumption since input alignments have to be kept in memory until an empty compute slot is available and each running job requires its own dynamic programming matrices. A value of 0 indicates to use as many parallel threads as computation cores are available.

-f, **--output-format**=format

Specify output file format. (possible values="eps", "svg", "gml", "xrna", "ssv" default="eps")

Available formats are: Encapsulated PostScript (`eps`), Scalable Vector Graphics (`svg`), Graph Meta Language (`gml`), and XRNA save file (`xrna`). Output filenames will end in ".eps" ".gml" ".svg" ".ss", respectively.

--pre=string

Add annotation macros to postscript file, and add the postscript code in "string" just before the code to draw the structure. This is an easy way to add annotation.

--post=string

Same as *--pre* but in contrast to adding the annotation macros. E.g to mark position 15 with circle use *--post=*"15 cmark".

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNAfold is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNAfold ignores any IDs retrieved from the input and automatically generates an ID for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="sequence")

If this parameter is set, each sequence will be prefixed with the provided string. Hence, the output files will obey the following naming scheme: "prefix_xxxx_ss.ps" (secondary structure plot), "prefix_xxxx_dp.ps" (dot-plot), "prefix_xxxx_dp2.ps" (stack probabilities), etc. where xxxx is the sequence number. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--filename-full

Use full FASTA header to create filenames. *(default=off)*

This parameter can be used to deactivate the default behavior of limiting output filenames to the first word of the sequence ID. Consider the following example: An input with FASTA header `>NM_0001 Homo Sapiens some gene` usually produces output files with the prefix "NM_0001" without the additional data available in the FASTA header, e.g. "NM_0001_ss.ps" for secondary structure plots. With this flag set, no truncation of the output filenames is done, i.e. output filenames receive the full FASTA header data as prefixes. Note, however, that invalid characters (such as whitespace) will be substituted by a delimiting character or simply removed, (see also the parameter option *--filename-delim*).

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAplot.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Plotting:

Command line options for changing the default behavior of structure layout and pairing probability plots

--covar

Annotate covariance of base pairs in consensus structure.

(default=off)

--covar-threshold=FLOAT

Set the threshold of maximum counter examples for coloring consensus structure plot.

(default="2")

Floating point numbers between 0 and 1 are treated as frequencies among all sequencesin the alignment. All other will be truncated to integer and used as absolute number of counter examples.

--covar-min-sat=FLOAT

Set the minimum saturation for coloring consensus structure plot.

(default="0.2")

Floating point number >= 0 and smaller than 1.

--aln

Produce a colored and structure annotated alignment in PostScript format in the file "aln.eps" in the current directory.

(default=off)

--aln-EPS-cols=INT

Number of columns in colored EPS alignment output.

(default="60")

A value less than 1 indicates that the output should not be wrapped at all.

-t, --layout-type=INT

Choose the plotting layout algorithm. (possible values="0", "1", "2", "3", "4" default="1")

Select the layout algorithm that computes the nucleotide coordinates. Currently, the following algorithms are available:

0: simple radial layout

1: Naview layout (Bruccoleri et al. 1988)

2: circular layout

3: RNAturtle (Wiegreffe et al. 2018)

4: RNApuzzler (Wiegreffe et al. 2018)

--noOptimization

Disable the drawing space optimization of RNApuzzler.

(default=off)

--ignoreExteriorIntersections

Ignore intersections with the exterior loop

within the RNA-tree.

(default=off)

--ignoreAncestorIntersections

Ignore ancestor intersections within the

RNA-tree.

(default=off)

--ignoreSiblingIntersections

Ignore sibling intersections within the

RNA-tree.

(default=off)

--allowFlipping

Allow flipping of exterior loop branches to resolve exterior branch intersections.

(default=off)

-h, --help

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.22 RNApvmin

RNApvmin - manual page for RNApvmin 2.8.0

Synopsis

```
RNApvmin [options] <file.shape>
```

DESCRIPTION

RNApvmin 2.8.0

Calculate a perturbation vector that minimizes discripancies between predicted and observed pairing probabilities

The program reads a RNA sequence from stdin and uses an iterative minimization process to calculate a perturbation vector that minimizes the discripancies between predicted pairing probabilites and observed pairing probabilities (deduced from given shape reactivities). Experimental data is read from a given SHAPE file and normalized to pairing probabilities. The experimental data has to be provided in a multiline plain text file where each line has the format `[position] [nucleotide] [absolute shape reactivity]` (e.g. `3 A 0.7`). The objective function used for the minimization may be weighted by choosing appropriate values for sigma and tau.

The minimization progress will be written to stderr. Once the minimization has terminated, the obtained perturbation vector is written to stdout.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-j, **--numThreads**=INT

Set the number of threads used for calculations.

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNApvmin.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations. The Minimum free energy (MFE) and a structure representative are calculated in any case.

--shapeConversion=STRING

Specify the method used to convert SHAPE reactivities to pairing probabilities.

(default="O")

The following methods can be used to convert SHAPE reactivities into the probability for a certain nucleotide to be unpaired.

M: Use linear mapping according to Zarringhalam et al. 2012

C: Use a cutoff-approach to divide into paired and unpaired nucleotides (e.g. "C0.25")

S: Skip the normalizing step since the input data already represents probabilities for being unpaired rather than raw reactivity values

L: Use a linear model to convert the reactivity into a probability for being unpaired (e.g. "Ls0.68i0.2" to use a slope of 0.68 and an intercept of 0.2)

O: Use a linear model to convert the log of the reactivity into a probability for being unpaired (e.g. "Os1.6i-2.29" to use a slope of 1.6 and an intercept of -2.29)

--tauSigmaRatio=DOUBLE

Ratio of the weighting factors tau and sigma. *(default="1.0")*

A high ratio will lead to a solution as close as possible to the experimental data, while a low ratio will lead to results close to the thermodynamic prediction without guiding pseudo energies.

--objectiveFunction=INT

The energies of the perturbation vector and the discripancies between predicted and observed pairing probabilities contribute to the objective function. This parameter defines, which function is used to process the contributions before summing them up. 0 square 1 absolute.

(default="0")

--sampleSize=INT

The iterative minimization process requires to evaluate the gradient of the objective function.

(default="1000")

A sample size of 0 leads to an analytical evaluation which scales as O(N^4). Choosing a sample size >0 estimates the gradient by sampling the given number of sequences from the ensemble, which is much faster.

-N, **--nonRedundant**

Enable non-redundant sampling strategy.

(default=off)

--intermediatePath=STRING Write an output file for each iteration of the

minimization process.

Each file contains the used perturbation vector and the score of the objective function. The number of the iteration will be appended to the given path.

--initialVector=DOUBLE

Specify the vector of initial pertubations. *(default="0")*

Defines the initial perturbation vector which will be used as starting vector for the minimization process. The value 0 results in a null vector. Every other value x will be used to populate the initial vector with random numbers from the interval [-x,x].

--minimizer=ENUM

Set the minimizing algorithm used for finding an appropriate perturbation vector.

(possible values="conjugate_fr",

"conjugate_pr", "vector_bfgs", "vector_bfgs2", "steepest_descent", "default" default="default")

The default option uses a custom implementation of the gradient descent algorithms while all other options represent various algorithms implemented in the GNU Scientific Library. When the GNU Scientific Library can not be found, only the default minimizer is available.

--initialStepSize=DOUBLE

The initial stepsize for the minimizer methods.

(default="0.01")

--minStepSize=DOUBLE

The minimal stepsize for the minizimer methods.

(default="1e-15")

--minImprovement=DOUBLE

The minimal improvement in the default minizimer method that has to be surpassed to considered a new result a better one.

(default="1e-3")

--minimizerTolerance=DOUBLE

The tolerance to be used in the GSL minimizer

methods.

(default="1e-3")

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--maxBPspan=INT

Set the maximum base pair span.

(default="-1")

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

S. Washietl, I.L. Hofacker, P.F. Stadler, M. Kellis (2012) "RNA folding with soft constraints: reconciliation of probing data and thermodynamics secondary structure prediction" Nucl Acids Res: 40(10), pp 4261-4272

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

EXAMPLES

RNApvmin acceptes a SHAPE file and a corresponding nucleotide sequence, which is read form stdin.

```
RNApvmin sequence.shape < sequence.fasta > sequence.pv
```

The normalized SHAPE reactivity data has to be stored in a text file, where each line contains the position and the reactivity for a certain nucleotide ([position] [nucleotide] [SHAPE reactivity]).

```
1 A 1.286
2 U 0.383
3 C 0.033
4 C 0.017
...
...
98 U 0.234
99 G 0.885
```

The nucleotide information in the SHAPE file is optional and will be used to cross check the given input sequence if present. If SHAPE reactivities could not be determined for every nucleotide, missing values can simply be omited.

The progress of the minimization will be printed to stderr. Once a solution was found, the calculated perturbation vector will be print to stdout and can then further be used to constrain RNAfold's MFE/partition function calculation by applying the perturbation energies as soft constraints.

```
RNAfold --shape=sequence.pv --shapeMethod=W < sequence.fasta
```

AUTHOR

Dominik Luntzer, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.23 RNAsnoop

RNAsnoop - manual page for RNAsnoop 2.8.0

Synopsis

```
RNAsnoop [options]
```

DESCRIPTION

RNAsnoop 2.8.0

Find targets of a query H/ACA snoRNA

reads a target RNA sequence and a H/ACA snoRNA sequence from a target and query file, respectively and computes optimal and suboptimal secondary structures for their hybridization. The calculation can be done roughly in O(nm), where is n the length of the target sequence and m is the length of the snoRNA stem, as it is specially tailored to the special case of H/ACA snoRNA. For general purpose target predictions, please have a look at RNAduplex, RNAup, RNAcofold and RNAplex. Accessibility effects can be estimated by RNAsnoop if a RNAplfold accessibility profile is provided.

The computed optimal and suboptimal structure are written to stdout, one structure per line. Each line consist of: The structure in dot bracket format with a & separating the two strands. The <> brackets represent snoRNA intramolecular interactions, while the () brackets represent intermolecular interactions between the snoRNA and its target.

The range of the structure in the two sequences in the format "from,to : from,to"; the energy of duplex structure in kcal/mol. If available the opening energy are also returned.

--verbose
: Be verbose. *(default=off)*

  Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-s, **--query**=STRING
: File containing the query sequence.

  Input sequences can be given piped to RNAsnoop or given in a query file with the `-s` option. Note that the `-s` option implies that the `-t` option is also used.

-t, **--target**=STRING
: File containing the target sequence.

  Input sequences can be given piped to RNAsnoop or given in a target file with the `-t` optionNote that the `-t` option implies that the `-s` option is also used.

-S, **--suffix**=STRING
: Specificy the suffix that was added by RNAup to the accessibility files.

  (default="_u1_to_30.out")

-P, **--from-RNAplfold**=STRING
: Specify the directory where accessibility profile generated by RNAplfold are found.

-U, **--from-RNAup**=STRING
: Specify the directory where accessibility profiles generated by RNAup are found.

-O, **--output_directory**=STRING Set where the generated figures should be
: stored.

  (default="./")

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAsnoop.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Options which alter the computing behaviour of RNAplex. Please note that the options allowing to filter out snoRNA-RNA duplexes expect the energy to be given in decacal/mol instead of kcal/mol. A threshold of -2.8(kcal/mol) should be given as :option:`-280`(decacal/mol).

-A, **--alignment-mode**

Specify if RNAsnoop gets alignments or single sequences as input.

(default=off)

-f, **--fast-folding**=INT

Speedup of the target search. *(default="1")*

This option allows one to decide if the backtracking has to be done (*-f* 1) or not (*-f* 0). For *-f* 1 the structure is computed based on the standard energy model. This is the slowest mode of RNAsnoop. *-f* 0 is the fastest mode, as no structure are recomputed and only the interaction energy is returned.

-c, **--extension-cost**=INT

Cost to add to each nucleotide in a duplex. *(default="0")*

Cost of extending a duplex by one nucleotide. Allows one to find compact duplexes, having few/small bulges or internal loops. Only useful when no accessibility profiles are available. This option is disabled if accessibility profiles are used (*-P* option).

-e, **--energy-threshold**=DOUBLE Maximal energy difference between the mfe and

the desired suboptimal.

(default="-1")

Energy range for a duplex to be returned. The threshold is set on the total energy of interaction, i.e. the hybridizationenergy corrected for opening energy if *-a* is set or the energy corrected by *-c*. If unset, only the mfe will be returned.

-o, **--minimal-right-duplex**=INT

Minimal Right Duplex Energy

(default="-270")

-l, **--minimal-loop-energy**=INT Minimal Right Duplex Energy.

(default="-280")

Minimal Stem Loop Energy of the snoRNA. The energy should be given in decacalories, i.e. a minimal stem-loop energy of -2.8 kcal/mol corresponds to -280 decacal/mol.

.HP -p, :option:`–minimal-left-duplex`=*INT* Minimal Left Duplex Energy.

(default="-170")

-q, **--minimal-duplex**=INT

Minimal Duplex Energy.

(default="-1090")

-d, **--duplex-distance**=INT

Distance between target 3' ends of two consecutive duplexes.

(default="2")

Distance between the target 3'ends of two consecutive duplexes. Should be set to the maximal length of interaction to get good results. Smaller d leads to larger overlaps between consecutive duplexes.

.HP -h, :option:`–minimal-stem-length`=*INT* Minimal snoRNA stem length.

(default="5")

.HP -i, :option:`–maximal-stem-length`=*INT* Maximal snoRNA stem length.

(default="120")

-j, **--minimal-duplex-box-length**=INT

Minimal distance between the duplex end and the

H/ACA box.

(default="11")

-k, **--maximal-duplex-box-length**=INT

Maximal distance between the duplex end and the

H/ACA box.

(default="16")

-m, **--minimal-snoRNA-stem-loop-length**=INT

Minimal number of nucleotides between the

beginning of stem loop and

beginning of the snoRNA sequence.

(default="1")

-n, **--maximal-snoRNA-stem-loop-length**=INT

Maximal number of nucleotides between the

beginning of stem loop and

beginning of the snoRNA sequence.

(default="100000")

-v, **--minimal-snoRNA-duplex-length**=INT

Minimal distance between duplex start and

snoRNA.

(default="0")

-w, **--maximal-snoRNA-duplex-length**=INT
Maximal distance between duplex start and
snoRNA.
(default="0")

-x, **--minimal-duplex-stem-energy**=INT
Minimal duplex stem energy.
(default="-1370")

-y, **--minimal-total-energy**=INT
Minimal total energy.
(default="100000")

-a, **--maximal-stem-asymmetry**=INT
Maximal snoRNA stem asymmetry.
(default="30")

-b, **--minimal-lower-stem-energy**=INT
Minimal lower stem energy.
(default="100000")

-L, **--alignmentLength**=INT
Limit the extent of the interactions to L nucleotides.
(default="25")

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

-C, **--constraint**
Calculate the stem structure subject to constraints.
(default=off)

The program reads first the stem sequence, then a string containing constraints on the structure encoded with the symbols:

. (no constraint for this base)

(the corresponding base has to be paired

x (the base is unpaired)

< (base i is paired with a base j>i)

> (base i is paired with a base j<i)

and matching brackets ( ) (base i pairs base j)

With the exception of "|", constraints will disallow all pairs conflicting with the constraint. This is usually sufficient to enforce the constraint, but occasionally a base may stay unpaired in spite of constraints. PF folding ignores constraints of type "|".

Plotting:

Command line options for changing the default behavior of structure layout and pairing probability plots.

-I, **--produce-ps**

Draw annotated 2D structures for a list of dot-bracket structures.

(default=off)

This option allows one to produce interaction figures in PS-format with conservation/accessibility annotation, if available.

-N, **--direct-redraw**

Outputs 2D interactions concurrently with the interaction calculation for each suboptimal interaction. The *-I* option should be preferred.

(default=off)

--help

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

The calculation of duplex structure is based on dynamic programming algorithm originally developed by Rehmsmeier and in parallel by Hofacker.

H. Tafer, S. Kehr, J. Hertel, I.L. Hofacker, P.F. Stadler (2009), "RNAsnoop: efficient target prediction for H/ACA snoRNAs.", Bioinformatics: 26(5), pp 610-616

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Hakim Tafer, Ivo L. Hofacker

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.24 RNAsubopt

RNAsubopt - manual page for RNAsubopt 2.8.0

Synopsis

```
RNAsubopt [OPTION]...
```

DESCRIPTION

RNAsubopt 2.8.0

calculate suboptimal secondary structures of RNAs

Reads RNA sequences from stdin and (in the default *-e* mode) calculates all suboptimal secondary structures within a user defined energy range above the minimum free energy (mfe). It prints the suboptimal structures in dot-bracket notation followed by the energy in kcal/mol to stdout. Be careful, the number of structures returned grows exponentially with both sequence length and energy range.

Alternatively, when used with the *-p* option, RNAsubopt produces Boltzmann weighted samples of secondary structures.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-i, **--infile**=filename

Read a file instead of reading from stdin.

The default behavior of RNAsubopt is to read input from stdin. Using this parameter the user can specify an input file name where data is read from.

-o, **--outfile**[=filename]

Print output to file instead of stdout.

This option may be used to write all output to output files rather than printing to stdout. The default filename is "RNAsubopt_output.sub" if no FASTA header precedes the input sequences and the *--auto-id* feature is inactive. Otherwise, output files with the scheme "prefix.sub" are generated, where the "prefix" is taken from the sequence id. The user may specify a single output file name for all data generated from the input by supplying an optional string as argument to this parameter. In case a file with the same filename already exists, any output of the program will be appended to it. Note: Any special characters in the filename will be replaced by the filename delimiter, hence there is no way to pass an entire directory path through this option yet. (See also the "–filename-delim" parameter)

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--auto-id

Automatically generate an ID for each sequence. *(default=off)*

The default mode of RNAsubopt is to automatically determine an ID from the input sequence data if the input file format allows to do that. Sequence IDs are usually given in the FASTA header of input sequences. If this flag is active, RNAsubopt ignores any IDs retrieved from the input and automatically generates an ID

for each sequence. This ID consists of a prefix and an increasing number. This flag can also be used to add a FASTA header to the output even if the input has none.

--id-prefix=STRING

Prefix for automatically generated IDs (as used in output file names).

(default="sequence")

If this parameter is set, each sequences' FASTA id will be prefixed with the provided string. FASTA ids then take the form ">prefix_xxxx" where xxxx is the sequence number. Note: Setting this parameter implies *--auto-id*.

--id-delim=CHAR

Change the delimiter between prefix and increasing number for automatically generated IDs (as used in output file names).

(default="_")

This parameter can be used to change the default delimiter "_" between the prefix string and the increasing number for automatically generated ID.

--id-digits=INT

Specify the number of digits of the counter in automatically generated alignment IDs.

(default="4")

When alignments IDs are automatically generated, they receive an increasing number, starting with 1. This number will always be left-padded by leading zeros, such that the number takes up a certain width. Using this parameter, the width can be specified to the users need. We allow numbers in the range [1:18]. This option implies *--auto-id*.

--id-start=LONG

Specify the first number in automatically generated IDs.

(default="1")

When sequence IDs are automatically generated, they receive an increasing number, usually starting with 1. Using this parameter, the first number can be specified to the users requirements. Note: negative numbers are not allowed. Note: Setting this parameter implies to ignore any IDs retrieved from the input data, i.e. it activates the *--auto-id* flag.

--filename-delim=CHAR

Change the delimiting character used in sanitized filenames.

(default="ID-delimiter")

This parameter can be used to change the delimiting character used while sanitizing filenames, i.e. replacing invalid characters. Note, that the default delimiter ALWAYS is the first character of the "ID delimiter" as supplied through the *--id-delim* option. If the delimiter is a whitespace character or empty, invalid characters will be simply removed rather than substituted. Currently, we regard the following characters as illegal for use in filenames: backslash \, slash /, question mark ?, percent sign %, asterisk *, colon :, pipe symbol |, double quote ", triangular brackets < and >.

--filename-full

Use full FASTA header to create filenames. *(default=off)*

This parameter can be used to deactivate the default behavior of limiting output filenames to the first word of the sequence ID. Consider the following example: An input with FASTA header `>NM_0001 Homo Sapiens some gene` usually produces output files with the prefix "NM_0001" without the additional data available in the FASTA header, e.g. "NM_0001.sub". With this flag set, no truncation of the output filenames is performed, i.e. output filenames receive the full FASTA header data as prefixes. Note, however, that invalid characters (such as whitespace) will be substituted by a delimiting character or simply removed, (see also the parameter option *--filename-delim*).

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAsubopt.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select the algorithms which should be applied to the given RNA sequence(s).

-e, **--deltaEnergy**=range

Compute suboptimal structures with energy in a certain range of the optimum (kcal/mol).

Default is calculation of mfe structure only.

--deltaEnergyPost=range

Only print structures with energy within range of the mfe after post reevaluation of energies.

Useful in conjunction with `-logML`, `-d1` or `-d3`: while the *-e* option specifies the range before energies are re-evaluated, this option specifies the maximum energy after re-evaluation.

-s, **--sorted**

Sort the suboptimal structures by energy and lexicographical order.

(default=off)

Structures are first sorted by energy in ascending order. Within groups of the same energy, structures are then sorted in ascending in lexicographical order of their dot-bracket notation. See the *--en-only* flag to deactivate this second step. Note that sorting is done in memory, thus it can easily lead to exhaution of RAM! This is especially true if the number of structures produced becomes large or the RNA sequence is rather long. In such cases better use an external sort method, such as UNIX "sort".

--en-only

Only sort structures by free energy. *(default=off)*

In combination with *--sorted*, this flag deactivates the second sorting criteria and sorts structures solely by their free energy instead of additionally sorting by lexicographic order in each energy band. This might save some time during the sorting process in situations where lexicographic order is not required.

-p, **--stochBT**=number

Randomly draw structures according to their probability in the Boltzmann ensemble.

Instead of producing all suboptimals in an energy range, produce a random sample of suboptimal structures, drawn with probabilities equal to their Boltzmann weights via stochastic backtracking in the partition function. The *-e* and *-p* options are mutually exclusive.

--stochBT_en=number

Same as "–stochBT" but also print free energies and probabilities of the backtraced structures.

--random-seed=INT

Set the seed for the random number generator

--betaScale=DOUBLE

Set the scaling of the Boltzmann factors. *(default="1.")*

The argument provided with this option is used to scale the thermodynamic temperature in the Boltzmann factors independently from the temperature of the individual loop energy contributions. The Boltzmann factors then become `exp(- dG/(kT*betaScale))` where `k` is the Boltzmann constant, `dG` the free energy contribution of the state and `T` the absolute temperature.

-N, **--nonRedundant**

Enable non-redundant sampling strategy.

(default=off)

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

-c, **--circ**

Assume a circular (instead of linear) RNA molecule.

(default=off)

-D, **--dos**

Compute density of states instead of secondary structures.

(default=off)

This option enables the evaluation of the number of secondary structures in certain energy bands around the MFE.

-z, **--zuker**

Compute Zuker suboptimals instead of all suboptimal structures within an energy band around the MFE.

(default=off)

-g, **--gquad**

Incoorporate G-Quadruplex formation. *(default=off)*

No support of G-quadruplex prediction for stochastic backtracking and Zuker-style suboptimals yet).

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

--maxBPspan=INT

Set the maximum base pair span.

(default="-1")

-C, **--constraint**[=filename]

Calculate structures subject to constraints. *(default="")*

The program reads first the sequence, then a string containing constraints on the structure encoded with the symbols:

`.` (no constraint for this base)

| (the corresponding base has to be paired

x (the base is unpaired)

< (base i is paired with a base j>i)

> (base i is paired with a base j<i)

and matching brackets ( ) (base i pairs base j)

With the exception of |, constraints will disallow all pairs conflicting with the constraint. This is usually sufficient to enforce the constraint, but occasionally a base may stay unpaired in spite of constraints. PF folding ignores constraints of type |.

--batch

Use constraints for multiple sequences. *(default=off)*

Usually, constraints provided from input file only apply to a single input sequence. Therefore, RNAsubopt will stop its computation and quit after the first input sequence was processed. Using this switch, RNAsubopt processes multiple input sequences and applies the same provided constraints to each of them.

--canonicalBPonly

Remove non-canonical base pairs from the structure constraint.

(default=off)

--enforceConstraint

Enforce base pairs given by round brackets ( ) in structure constraint.

(default=off)

--motif=SEQUENCE,STRUCTURE,ENERGY

Specify stabilizing energy of a ligand binding

to a particular sequence/structure motif.

Some ligands binding to RNAs require and/or induce particular sequence and structure motifs. For instance they bind to an internal loop, or small hairpin loop. If for such cases a binding free energy is known, the binding and therefore stabilizing effect of the ligand can be included in the folding recursions. Interior loop motifs are specified as concatenations of 5`` and 3`` motif, separated by an & character.

Energy contributions must be specified in kcal/mol.

See the manpage for an example usage of this option.

--commands=filename

Read additional commands from file

Commands include hard and soft constraints, but also structure motifs in hairpin and internal loops that need to be treeted differently. Furthermore, commands can be set for unstructured and structured domains.

Experimental Structure Probing Data:

The following arguments and siwtches control various implementations that allow for guiding the structure prediction with the help of additional (experimental) RNA structure probing data, such as SHAPE, DMS, etc.

--sp-data=filename

Read structure probing data from an input file and guide the predictions accordingly. Must precede the strategy, i.e. a data file must be specified before the corresponding *--sp-strategy* option!

This option and its argument only specifies the file name with the (reactivity) data. What kind of data the file contains and how the data is actually used during the predictions can be specified by the *--sp-strategy* and *--sp-preprocess* options.

--sp-strategy=strategy

Select the strategy how the probing data is used to guide the structure predictions.

(default="D")

The following strategies (methods) are available to convert structure probing data into pseudo energy contributions:

D: Deigan et al 2009 strategy.

Here, probing data is converted into pseudo energy terms using a linear

model. The derived energies are then applied for every nucleotide involved in a stacked pair. The default slope and intercept variables of the linear model are m=1.8 and b=-0.6, respectively. To change these parameters, e.g. to m=1.9 and b=-0.7, use a parameter string like this: *--sp-strategy=*"Dm1.9b-0.7". Note the precedence of m over b. You may also provide only one of the two variables like: *--sp-strategy=*"Dm1.9" or *--sp-strategy=*"Db-0.7".

E: Eddy 2014 strategy.

This strategy distinguishes two structural states (classes) for each base,

paired and unpaired. Based on a list of prior distributions for each of the classes, the posterior probability for observing a probing reactivity of a base is estimated and then converted into a respective pseudo energy contribution. Thermodynamic temperature T is taken from the model settings as specified by *--temp* but can be changed to any other value, either in degrees Celcius (c or t) or Kelvin (k). Use one of the following parameter strings to change the temperature to 25 degree Celcius: *--sp-strategy=*"Ec=25" or *--sp-strategy=*"Ek=298.15". By default, built-in prior distributions are used with this strategy. This can be changed if the user passes additional probing data files via the *--sp-data* option and then selects the prior distribution specifier (P) as strategy (see below).

P: Use probing data as prior distribution for Eddy 2014 strategy.

This indicates that the most recent probing data file actually contains a

prior distribution of probing data for a particular structure class, i.e. unpaired or paired bases. The prior distribution then replaces the default priors in the Eddy 2014 (E) strategy. Use the lower case characters u and p to indicate which class, unpaired or paired, the prior distribution represents, e.g. *--sp-strategy=*"Pp" will be interpreted as prior distribution for paired nucleotides. Note, that this option must follow after the *--sp-strategy=*"E" and before any further *--sp-strategy* options that do not specify other prior distributions!

Z: Zarringhalam et al 2012 strategy.

Here, the probing data is first converted into pairing probabilities and

aberration from the observed pairing probabilities will be penalized during the folding recursion. The magnitude of the penalties can affected by adjusting the factor beta (e.g. *--sp-strategy=*"Zb0.8").

W: Apply a given vector of perturbation energies to unpaired nucleotides

according to Washietl et al 2012. Perturbation vectors can be calculated by using RNApvmin.

--sp-preprocess=method

Select a method to pre-process the structure probing data before it undergoes conversion by a strategy.

(default="I")

--shape=filename

Use SHAPE reactivity data to guide structure predictions.

--shapeMethod=method

Select SHAPE reactivity data incorporation strategy.

(default="D")

The following methods can be used to convert SHAPE reactivities into pseudo energy contributions.

D: Convert by using the linear equation according to Deigan et al 2009.

Derived pseudo energy terms will be applied for every nucleotide involved in a stacked pair. This method is recognized by a capital `D` in the provided parameter, i.e.: *--shapeMethod=*"D" is the default setting. The slope `m` and the intercept `b` can be set to a non-default value if necessary, otherwise m=1.8 and b=-0.6. To alter these parameters, e.g. m=1.9 and b=-0.7, use a parameter string like this: *--shapeMethod=*"Dm1.9b-0.7". You may also provide only one of the two parameters like: *--shapeMethod=*"Dm1.9" or *--shapeMethod=*"Db-0.7".

`Z`: Convert SHAPE reactivities to pseudo energies according to Zarringhalam

et al 2012. SHAPE reactivities will be converted to pairing probabilities by using linear mapping. Aberration from the observed pairing probabilities will be penalized during the folding recursion. The magnitude of the penalties can affected by adjusting the factor beta (e.g. *--shapeMethod=*"Zb0.8").

`W`: Apply a given vector of perturbation energies to unpaired nucleotides

according to Washietl et al 2012. Perturbation vectors can be calculated by using RNApvmin.

--shapeConversion=method

Select method for SHAPE reactivity conversion.

(default="O")

This parameter is useful when dealing with the SHAPE incorporation according to Zarringhalam et al. The following methods can be used to convert SHAPE reactivities into the probability for a certain nucleotide to be unpaired.

`M`: Use linear mapping according to Zarringhalam et al. `C`: Use a cutoff-approach to divide into paired and unpaired nucleotides (e.g. "C0.25") `S`: Skip the normalizing step since the input data already represents probabilities for being unpaired rather than raw reactivity values `L`: Use a linear model to convert the reactivity into a probability for being unpaired (e.g. "Ls0.68i0.2" to use a slope of 0.68 and an intercept of 0.2) `O`: Use a linear model to convert the log of the reactivity into a probability for being unpaired (e.g. "Os1.6i-2.29" to use a slope of 1.6 and an intercept of `-2`.29)

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

-m, **--modifications**[=STRING]

Allow for modified bases within the RNA sequence string.

(default="7I6P9D")

Treat modified bases within the RNA sequence differently, i.e. use corresponding energy corrections and/or pairing partner rules if available. For that, the modified bases in the input sequence must be marked by

their corresponding one-letter code. If no additional arguments are supplied, all available corrections are performed. Otherwise, the user may limit the modifications to a particular subset of modifications, resp. one-letter codes, e.g. `-mP6` will only correct for pseudouridine and m6A bases.

Currently supported one-letter codes and energy corrections are:

`7`: 7-deaza-adenonsine (7DA)

`I`: Inosine

`6`: N6-methyladenosine (m6A)

`P`: Pseudouridine

`9`: Purine (a.k.a. nebularine)

`D`: Dihydrouridine

--mod-file=STRING

Use additional modified base data from JSON file.

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

How to treat "dangling end" energies for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d1` only unpaired bases can participate in at most one dangling end. With `-d2` this check is ignored, dangling energies will be added for the bases adjacent to a helix on both sides in any case; this is the default for mfe and partition function folding (*-p*). The option `-d0` ignores dangling ends altogether (mostly for debugging). With `-d3` mfe folding will allow coaxial stacking of adjacent helices in multi-loops. At the moment the implementation will not allow coaxial stacking of the two enclosed pairs in a loop of degree 3 and works only for mfe folding.

Note that with `-d1` and `-d3` only the MFE computations will be using this setting while partition function uses `-d2` setting, i.e. dangling ends will be treated differently.

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--logML

Recompute energies of structures using a logarithmic energy function for multi-loops before output. *(default=off)*

This option does not effect structure generation, only the energies that are printed out. Since logML lowers energies somewhat, some structures may be missing.

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

S. Wuchty, W. Fontana, I. L. Hofacker and P. Schuster (1999), "Complete Suboptimal Folding of RNA and the Stability of Secondary Structures", Biopolymers: 49, pp 145-165

M. Zuker (1989), "On Finding All Suboptimal Foldings of an RNA Molecule", Science 244.4900, pp 48-52

Y. Ding, and C.E. Lawrence (2003), "A statistical sampling algorithm for RNA secondary structure prediction", Nucleic Acids Research 31.24, pp 7280-7301

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

AUTHOR

Ivo L Hofacker, Stefan Wuchty, Walter Fontana, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.2.25 RNAup

RNAup - manual page for RNAup 2.8.0

Synopsis

```
RNAup [OPTION]...
```

DESCRIPTION

RNAup 2.8.0

Calculate the thermodynamics of RNA-RNA interactions

RNAup calculates the thermodynamics of RNA-RNA interactions, by decomposing the binding into two stages. (1) First the probability that a potential binding sites remains unpaired (equivalent to the free energy needed to open the site) is computed. (2) Then this accessibility is combined with the interaction energy to obtain the total binding energy. All calculations are done by computing partition functions over all possible conformations.

RNAup provides two different modes: By default RNAup computes accessibilities, in terms of the free energies needed to open a region (default length 4). It prints the region of highest accessibility and its opening energy to stdout, opening energies for all other regions are written to a file.

.br In interaction mode the interaction between two RNAs is calculated. It is invoked if the input consists of two sequences concatenated with an &, or if the options -X[pf] or -b are given. Unless the -b option is specified RNAup assumes that the longer RNA is a structured target sequence while the shorter one is an unstructured small RNA. .br Additionally, for every position along the target sequence we write the best free energy of binding for an interaction that includes this position to the the output file. Output to stdout consists of the location and free energy, dG, for the optimal region of interaction. The binding energy dG is also split into its components the interaction energy dGint and the opening energy dGu_l (and possibly dGu_s for the shorter sequence). .br In addition we print the optimal interaction structure as computed by RNAduplex for this region. Note that it can happen that the RNAduplex computed optimal interaction does not coincide with the optimal RNAup region. If the two predictions don't match the structure string is replaced by a run of "." and a message is written to stderr. .br

Each sequence should be in 5`` to 3`` direction. If the sequence is preceded by a line of the form .. code:

```
> name
```

the output file "name_ux_up.out" is produced, where the "x" in "ux" is the value set by the -u option. Otherwise the file name defaults to RNA_ux_up.out. The output is concatenated if a file with the same name exists. .br

RNA sequences are read from stdin as strings of characters. White space and newline within a sequence cause an error! Newline is used to separate sequences. The program will continue to read new sequences until a line consisting of the single character @ or an end of file condition is encountered.

-v, **--verbose**

Be verbose. *(default=off)*

Lower the log level setting such that even INFO messages are passed through.

I/O Options:

Command line options for input and output (pre-)processing

-o, **--no_output_file**

Do not produce an output file.

(default=off)

--no_header

Do not produce a header with the command line parameters used in the outputfile.

(default=off)

--noconv

Do not automatically substitute nucleotide "T" with "U".

(default=off)

--log-level=level

Set log level threshold. *(default="2")*

By default, any log messages are filtered such that only warnings (level 2) or errors (level 3) are printed. This setting allows for specifying the log level threshold, where higher values result in fewer information. Log-level 5 turns off all messages, even errors and other critical information.

--log-file[=filename]

Print log messages to a file instead of stderr. *(default="RNAup.log")*

--log-time

Include time stamp in log messages.

(default=off)

--log-call

Include file and line of log calling function.

(default=off)

--log-func

Include name of the function that issued the log.

(default=off)

Algorithms:

Select additional algorithms which should be included in the calculations.

-u, **--ulength**=length

Specify the length of the unstructured region in the output.

(default="4")

The probability of being unpaired is plotted on the right border of the unpaired region. You can specify up to 20 different length values: use "-" to specify a range of continuous values (e.g. *-u* 4-8) or specify a list of comma separated values (e.g. *-u* 4,8,15).

-c, **--contributions**=SHIME

Specify the contributions listed in the output. *(default="S")*

By default only the full probability of being unpaired is plotted. The *-c* option allows one to get the different contributions (c) to the probability of being unpaired: The full probability of being unpaired ("S" is the sum of the probability of being unpaired in the exterior loop ("E"), within a hairpin loop ("H"), within an internal loop ("I") and within a multiloop ("M"). Any combination of these letters may be given.

Calculations of RNA-RNA interactions:

-w, **--window**=INT

Set the maximal length of the region of interaction.

(default="25")

-b, **--include_both**

Include the probability of unpaired regions in both (b) RNAs.

(default=off)

By default only the probability of being unpaired in the longer RNA (target) is used.

-5, **--extend5**=INT

Extend the region of interaction in the target to some residues on the 5' side.

The underlying assumption is that it is favorable for an interaction if not only the direct region of contact is unpaired but also a few residues 5'

-3, **--extend3**=INT

Extend the region of interaction in the target to some residues on the 3' side.

The underlying assumption is that it is favorable for an interaction if not only the direct region of contact is unpaired but also a few residues 3'

--interaction_pairwise

Activate pairwise interaction mode. *(default=off)*

The first sequence interacts with the 2nd, the third with the 4th etc. If activated, two interacting sequences may be given in a single line separated by "&" or each sequence may be given on an extra line.

--interaction_first

Activate interaction mode using first sequence only.

(default=off)

The interaction of each sequence with the first one is calculated (e.g. interaction of one mRNA with many small RNAs). Each sequence has to be given on an extra line

-S, **--pfScale**=DOUBLE

In the calculation of the pf use scale*mfe as an estimate for the ensemble free energy (used to avoid overflows).

(default="1.07")

The default is 1.07, useful values are 1.0 to 1.2. Occasionally needed for long sequences.

Structure Constraints:

Command line options to interact with the structure constraints feature of this program

-C, **--constraint**

Apply structural constraint(s) during prediction.

(default=off)

The program first reads the sequence(s), then a dot-bracket like string containing constraints on the structure. The following symbols are recognized:

. ... no constraint for this base

x ... the base is unpaired

< ... the base pairs downstream, i.e. i is paired with j > i

> ... the base pairs upstream, i.e. i is paired with j < i

() … base i pairs with base j

| … the corresponding base has to be paired intermolecularily (only for

interaction mode)

Energy Parameters:

Energy parameter sets can be adapted or loaded from user-provided input files

-T, **--temp**=DOUBLE

Rescale energy parameters to a temperature of temp C. Default is 37C.

(default="37.0")

-P, **--paramFile**=paramfile

Read energy parameters from paramfile, instead of using the default parameter set.

Different sets of energy parameters for RNA and DNA should accompany your distribution. See the RNAlib documentation for details on the file format. The placeholder file name `DNA` can be used to load DNA parameters without the need to actually specify any input file.

-4, **--noTetra**

Do not include special tabulated stabilizing energies for tri-, tetra- and hexaloop hairpins.

(default=off)

Mostly for testing.

--salt=DOUBLE

Set salt concentration in molar (M). Default is 1.021M.

--saltInit=DOUBLE

Provide salt correction for duplex initialization (in kcal/mol).

Model Details:

Tweak the energy model and pairing rules additionally using the following parameters

-d, **--dangles**=INT

Specify "dangling end" model for bases adjacent to helices in free ends and multi-loops.

(default="2")

With `-d2` dangling energies will be added for the bases adjacent to a helix on both sides in any case.

The option `-d0` ignores dangling ends altogether (mostly for debugging).

--noLP

Produce structures without lonely pairs (helices of length 1).

(default=off)

For partition function folding this only disallows pairs that can only occur isolated. Other pairs may still occasionally occur as helices of length 1.

--noGU

Do not allow GU pairs.

(default=off)

--noClosingGU

Do not allow GU pairs at the end of helices.

(default=off)

--nsp=STRING

Allow other pairs in addition to the usual AU,GC,and GU pairs.

Its argument is a comma separated list of additionally allowed pairs. If the first character is a "-" then AB will imply that AB and BA are allowed pairs, e.g. *--nsp*="-GA" will allow GA and AG pairs. Nonstandard pairs are given 0 stacking energy.

--energyModel=INT

Set energy model.

Rarely used option to fold sequences from the artificial ABCD... alphabet, where A pairs B, C-D etc. Use the energy parameters for GC (*--energyModel* 1) or AU (*--energyModel* 2) pairs.

--helical-rise=FLOAT

Set the helical rise of the helix in units of Angstrom.

(default="2.8")

Use with caution! This value will be re-set automatically to 3.4 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

--backbone-length=FLOAT

Set the average backbone length for looped regions in units of Angstrom.

(default="6.0")

Use with caution! This value will be re-set automatically to 6.76 in case DNA parameters are loaded via *-P* DNA and no further value is provided.

-h, **--help**

display help and exit

--full-help

display help, including hidden options, and exit

--detailed-help

display help, including all details and hidden options, and exit

-V, **--version**

output version and exit

REFERENCES

If you use this program in your work you might want to cite:

R. Lorenz, S.H. Bernhart, C. Hoener zu Siederdissen, H. Tafer, C. Flamm, P.F. Stadler and I.L. Hofacker (2011), "ViennaRNA Package 2.0", Algorithms for Molecular Biology: 6:26

I.L. Hofacker, W. Fontana, P.F. Stadler, S. Bonhoeffer, M. Tacker, P. Schuster (1994), "Fast Folding and Comparison of RNA Secondary Structures", Monatshefte f. Chemie: 125, pp 167-188

R. Lorenz, I.L. Hofacker, P.F. Stadler (2016), "RNA folding with hard and soft constraints", Algorithms for Molecular Biology 11:1 pp 1-13

U. Mueckstein, H. Tafer, J. Hackermueller, S.H. Bernhart, P.F. Stadler, and I.L. Hofacker (2006), "Thermodynamics of RNA-RNA Binding", Bioinformatics: 22(10), pp 1177-1182

The energy parameters are taken from:

D.H. Mathews, M.D. Disney, D. Matthew, J.L. Childs, S.J. Schroeder, J. Susan, M. Zuker, D.H. Turner (2004), "Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure", Proc. Natl. Acad. Sci. USA: 101, pp 7287-7292

D.H Turner, D.H. Mathews (2009), "NNDB: The nearest neighbor parameter database for predicting stability of nucleic acid secondary structure", Nucleic Acids Research: 38, pp 280-282

EXAMPLES

.B Output to stdout:

In Interaction mode RNAup prints the most favorable interaction energy between the two sequences to stdout. The most favorable interaction energy (dG) depends on the position in the longer sequence (region [i,j]) and the position in the shorter sequence (region[k,l]): dG[i,j;k,l]. dG[i,j;k,l] is the largest contribution to dG[i,j] = sum_kl dG[i,j;k,l] which is given in the output file: therefore dG[i,j;k,l] <= dG[i,j].

```
``....,....1....,....2....,....3....,....4....,....5....,....6....,....7....,....8``
> franz
GGAGUAGGUUAUCCUCUGUU
> sissi
AGGACAACCU
dG = dGint + dGu_l
(((((.((((&)))).)))))   6,15  :   1,10  (-6.66 = -9.89 + 3.23)
AGGUUAUCCU&AGGACAACCU
RNAup output in file: franz_sissi_w25_u3_4_up.out
```

where the result line contains following information

```
RNAduplex results       [i,j]     [k,l]    dG = dGint + dGu_l
(((((.((((&)))).)))))   6,15   :  1,10     (-6.66=-9.89+3.23)
```

.RD .B Output to file:

Output to file contains a header including date, the command line of the call to RNAup, length and names of the input sequence(s) followed by the sequence(s). The first sequence is the target sequence. Printing of the header can be turned off using the -nh option.

The line directly after the header gives the column names for the output:

```
position     dGu_l for -u 3      dGu_l for -u 4       dG
#     pos       u3S        u3H        u4S        u4H         dG
```

where all information refers to the target sequence. The dGu_l column contains information about the -u value (u=3 or u=4) and the contribution to the free energy to open all structures "S" or only hairpin loops "H", see option -c. NA means that no results is possible (e.g. column u3S row 2: no region of length 3 ending at position 2 exists).

```
#  Thu Apr 10 09:15:11 2008
#  RNAup -u 3,4 -c SH -b
#  20 franz
#  GGAGUAGGUUAUCCUCUGUU
#  10 sissi
#  AGGACAACCU
#     pos       u3S        u3H        u4S        u4H         dG
1        NA         NA         NA         NA    -1.540
2        NA         NA         NA         NA    -1.540
3     1.371         NA         NA         NA    -1.217
4     1.754      5.777      1.761         NA    -1.393
5     1.664      3.140      1.811      5.800    -1.393
```

If the -b option is selected position and dGu_s values for the shorter sequence are written after the information for the target sequence.

AUTHOR

Ivo L Hofacker, Peter F Stadler, Ulrike Mueckstein, Ronny Lorenz

REPORTING BUGS

If in doubt our program is right, nature is at fault. Comments should be sent to rna@tbi.univie.ac.at.

2.3 I/O Formats

Below, you'll find a listing of different sections that introduce the most common notations of sequence and structure data, specifications of bioinformatics sequence and structure file formats, and various output file formats produced by our library.

2.3.1 RNA Structures

Here, we describe the different notations and representations of RNA secondary structures used throughout our library and prediction tools.

Dot-Bracket Notation

The Dot-Bracket notation as introduced already in the early times of the ViennaRNA Package denotes base pairs by matching pairs of parenthesis `()` and unpaired nucleotides by dots `.`.

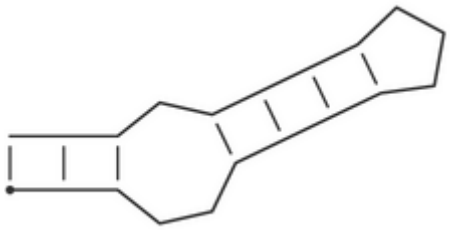
> **Note**
>
> This is the standard representation of a secondary structure in our library.

Based on that notation, more elaborate representations have been developed to include additional information, such as the loop context a nucleotide belongs to and to annotated pseudo-knots.

Consider the following secondary structure in dot-bracket notation:

```
(((..((((...)))).)))
```

which, drawn as a secondary structure graph, looks like:



It is a stem-loop structure consisting of a an outer helix of 3 base pairs followed by an internal loop of size 3, a second helix of length 4, and a hairpin loop of size 3.

Pseudo Dot-Bracket Notation

Base pair probabilities are sometimes summarized in *pseudo dot-bracket notation* with the additional symbols `,`, `|`, `{`, `}`. Here, the usual `(`, `)`, `.`, represent bases with a strong preference (more than 2/3) to pair upstream (with a partner further 3'), pair down-stream, or do not pair, respectively. `{`, `}`, and `,` are just the weaker version of the above and `|` represents a base that is mostly paired but has pairing partners both upstream and downstream. In this case opening and closing brackets do not need to match.

Extended Dot-Bracket Notation

A more generalized version of the original Dot-Bracket notation may use additional pairs of brackets, such as <>, {}, and [], and matching pairs of uppercase/lowercase letters. This allows for anotating pseudo-knots, since different pairs of brackets are not required to be nested.

The follwing annotations of a simple structure with two crossing helices of size 4 are equivalent:

```
<<<<[[[[....>>>>]]]]
((((AAAA....))))aaaa
AAAA{{{{....aaaa}}}}
```

See also…

vrna_db_pack(), *vrna_db_unpack()*, *vrna_db_flatten()*, *vrna_db_flatten_to()*, *vrna_db_from_ptable()*, *vrna_db_from_plist()*, *vrna_db_to_element_string()*, *vrna_db_pk_remove()*

WUSS notation

The Washington University Secondary Structure (WUSS) notation is frequently used for consensus secondary structures, e.g. in io/msa:stockholm 1.0 format

This notation allows for a fine-grained annotation of base pairs and unpaired nucleotides, including pseudo-knots.

See also…

WUSS notation in the infernal user guide at http://eddylab.org/infernal/Userguide.pdf

Below, you'll find a list of secondary structure elements and their corresponding WUSS annotation.

- **Base pairs**

  Nested base pairs are annotated by matching pairs of the symbols <>, (), {}, and []. Each of the matching pairs of parenthesis have their special meaning, however, when used as input in our programs, e.g. structure constraint, these details are usually ignored. Furthermore, base pairs that constitute as pseudo-knot are denoted by letters from the latin alphabet and are, if not denoted otherwise, ignored entirely in our programs.

- **Hairpin loops**

  Unpaired nucleotides that constitute the hairpin loop are indicated by underscores, _. Here is an example:

  ```
  <<<<<_____>>>>>
  ```

- **Bulges and internal loops**

  Residues that constitute a bulge or internal loop are denoted by dashes, -:

  ```
  (((--<<_____>>-)))
  ```

- **Multibranch loops**

  Unpaired nucleotides in multibranch loops are indicated by commas ,:

  ```
  (((,,<<_____>>,<<____>>)))
  ```

- **External residues**

  Single stranded nucleotides in the exterior loop, i.e. not enclosed by any other pair are denoted by colons, ::

```
<<<____>>>:::
```

- **Insertions**

  In cases where an alignment represents the consensus with a known structure, insertions relative to the known structure are denoted by periods, `.`. Regions where local structural alignment was invoked, leaving regions of both target and query sequence unaligned, are indicated by tildes, ~.

  These symbols only appear in alignments of a known (query) structure annotation to a target sequence of unknown structure.

- **Pseudo-knots**

  The WUSS notation allows for annotation of pseudo-knots using pairs of upper-case/lower-case letters. Our programs and library functions usually ignore pseudo-knots entirely treating them as unpaired nucleotides, if not stated otherwise:

```
<<<_AAA___>>>aaa
```

See also...

vrna_db_from_WUSS()

Abstract Shapes

Abstract Shapes, introduced by Giegerich *et al.* [2004], collapse the secondary structure while retaining the nestedness of helices and hairpin loops.

The abstract shapes representation abstracts the structure from individual base pairs and their corresponding location in the sequence, while retaining the inherent nestedness of helices and hairpin loops.

Below is a description of what is included in the abstract shapes abstraction for each respective level together with an example structure:

```
CGUCUUAAACUCAUCACCGUGUGGAGCUGCGACCCUUCCCUAGAUUCGAAGACGAG
((((((...(((..(((...))))))...(((..((.....))..))))))))..
```

Shape Level	Description	Result
1	Most accurate - all loops and all unpaired	`[_[_[]]_[_[]_]]_`
2	Nesting pattern for all loop types and unpaired regions in external loop and multiloop	`[[_[]][_[]_]]`
3	Nesting pattern for all loop types but no unpaired regions	`[[[]][[]]]`
4	Helix nesting pattern in external loop and multiloop	`[[][[]]]`
5	Most abstract - helix nesting pattern and no unpaired regions	`[[][]]`

Note

Our implementations also provide the special Shape Level 0, which does not collapse any structural features but simply convert base pairs and unpaired nucleotides into their corresponding set of symbols for abstract shapes.

See also...

vrna_abstract_shapes(), *vrna_abstract_shapes_pt()*

Tree Representations

Secondary structures can be readily represented as trees, where internal nodes represent base pairs, and leaves represent unpaired nucleotides. The dot-bracket structure string already is a tree represented by a string of parenthesis (base pairs) and dots for the leaf nodes (unpaired nucleotides).

Alternatively, one may find representations with two types of node labels, `P` for paired and `U` for unpaired; a dot is then replaced by `(U)`, and each closed bracket is assigned an additional identifier `P`. We call this the expanded notation. In Fontana *et al.* [1993] a condensed representation of the secondary structure is proposed, the so-called homeomorphically irreducible tree (HIT) representation. Here a stack is represented as a single pair of matching brackets labeled `P` and weighted by the number of base pairs. Correspondingly, a contiguous strain of unpaired bases is shown as one pair of matching brackets labeled `U` and weighted by its length. Generally any string consisting of matching brackets and identifiers is equivalent to a plane tree with as many different types of nodes as there are identifiers.

Shapiro [1988] proposed a coarse grained representation which does not retain the full information of the secondary structure. He represents the different structure elements by single matching brackets and labels them as

- `H` (hairpin loop),
- `I` (internal loop),
- `B` (bulge),
- `M` (multi-loop), and
- `S` (stack).

We extend his alphabet by an extra letter for external elements `E`. Again these identifiers may be followed by a weight corresponding to the number of unpaired bases or base pairs in the structure element. All tree representations (except for the dot-bracket form) can be encapsulated into a virtual root (labeled `R`).

The following example illustrates the different linear tree representations used by the package:

Consider the secondary structure represented by the dot-bracket string (full tree):

```
.((..(((...)))..((..)))).
```

which is the most convenient condensed notation used by our programs and library functions.

Then, the following tree representations are equivalent:

- Expanded tree:

  ```
  ((U)(((U)(U)((((U)(U)(U)P)P)P)(U)(U)(((U)(U)P)P)P)P)(U)R)
  ```

- HIT representation ([Fontana *et al.*, 1993]):

  ```
  ((U1)((U2)((U3)P3)(U2)((U2)P2)P2)(U1)R)
  ```

- Coarse Grained Tree Representation ([Shapiro, 1988]):

  - Short (with root node `R`, without stem nodes `S`):

    ```
    ((H)((H)M)R)
    ```

  - Full (with root node *R* `):

    ```
    (((((H)S)((H)S)M)S)R)
    ```

  - Extended (with root node `R`, with external nodes `E`):

    ```
    ((((((H)S)((H)S)M)S)E)R)
    ```

  - Weighted (with root node `R`, with external nodes `E`):

```
((((((H3)S3)((H2)S2)M4)S2)E2)R)
```

The Expanded tree is rather clumsy and mostly included for the sake of completeness. The different versions of Coarse Grained Tree Representations are variatios of Shapiro's linear tree notation.

For the output of aligned structures from string editing, different representations are needed, where we put the label on both sides. The above examples for tree representations would then look like:

```
a) (UU)(P(P(P(P(UU)(UU)(P(P(P(UU)(UU)(UU)P)P)P)(UU)(UU)(P(P(UU)(U...
b) (UU)(P2(P2(U2U2)(P2(U3U3)P3)(U2U2)(P2(U2U2)P2)P2)(UU)P2)(UU)
c) (B(M(HH)(HH)M)B)
   (S(B(S(M(S(HH)S)(S(HH)S)M)S)B)S)
   (E(S(B(S(M(S(HH)S)(S(HH)S)M)S)B)S)E)
d) (R(E2(S2(B1(S2(M4(S3(H3)S3)((H2)S2)M4)S2)B1)S2)E2)R)
```

Aligned structures additionally contain the gap character _.

See also…

vrna_db_to_tree_string(), *vrna_tree_string_unweight()*, *vrna_tree_string_to_db()*

2.3.2 Multiple Sequence Alignments (MSA)

ClustalW format

The *ClustalW* format is a relatively simple text file containing a single multiple sequence alignment of DNA, RNA, or protein sequences. It was first used as an output format for the *clustalw* programs, but nowadays it may also be generated by various other sequence alignment tools. The specification is straight forward:

- The first line starts with the words:

  ```
  CLUSTAL W
  ```

  or:

  ```
  CLUSTALW
  ```

- After the above header there is at least one empty line
- Finally, one or more blocks of sequence data are following, where each block is separated by at least one empty line.

Each line in a blocks of sequence data consists of the sequence name followed by the sequence symbols, separated by at least one whitespace character. Usually, the length of a sequence in one block does not exceed 60 symbols. Optionally, an additional whitespace separated cumulative residue count may follow the sequence symbols. Optionally, a block may be followed by a line depicting the degree of conservation of the respective alignment columns.

Note

Sequence names and the sequences must not contain whitespace characters! Allowed gap symbols are the hyphen (-), and dot (.).

> **Warning**
>
> Please note that many programs that output this format tend to truncate the sequence names to a limited number of characters, for instance the first 15 characters. This can destroy the uniqueness of identifiers in your MSA.

Here is an example alignment in ClustalW format:

```
CLUSTAL W (1.83) multiple sequence alignment


AL031296.1/85969-86120      ␣
↪CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGUAACAAUACUUAC
AANU01225121.1/438-603      ␣
↪CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGUGACAAUACUUAC
AAWR02037329.1/29294-29150  ---CUCGACACCACU---
↪GCCUCGGUUACCCAUCGGUGCAGUGCGGGUAGUAGUACCAAU

AL031296.1/85969-86120      UCUCGUUGGUGAUAAGGAACAGCU
AANU01225121.1/438-603      UCUCGUUGGUGAUAAGGAACAGCU
AAWR02037329.1/29294-29150  GCUAAUUAGUUGUGAGGACCAACU
```

Stockholm 1.0 format

Here is an example alignment in Stockholm 1.0 format:

```
# STOCKHOLM 1.0

#=GF AC   RF01293
#=GF ID   ACA59
#=GF DE   Small nucleolar RNA ACA59
#=GF AU   Wilkinson A
#=GF SE   Predicted; WAR; Wilkinson A
#=GF SS   Predicted; WAR; Wilkinson A
#=GF GA   43.00
#=GF TC   44.90
#=GF NC   40.30
#=GF TP   Gene; snRNA; snoRNA; HACA-box;
#=GF BM   cmbuild -F CM SEED
#=GF CB   cmcalibrate --mpi CM
#=GF SM   cmsearch --cpu 4 --verbose --nohmmonly -E 1000 -Z 549862.597050 CM SEQDB
#=GF DR   snoRNABase; ACA59;
#=GF DR   SO; 0001263; ncRNA_gene;
#=GF DR   GO; 0006396; RNA processing;
#=GF DR   GO; 0005730; nucleolus;
#=GF RN   [1]
#=GF RM   15199136
#=GF RT   Human box H/ACA pseudouridylation guide RNA machinery.
#=GF RA   Kiss AM, Jady BE, Bertrand E, Kiss T
#=GF RL   Mol Cell Biol. 2004;24:5797-5807.
#=GF WK   Small_nucleolar_RNA
#=GF SQ   3


AL031296.1/85969-86120      ␣
↪CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGUAACAAUACUUACUCUCGUUGGUGAUAAGGAACAGCU
AANU01225121.1/438-603      ␣
↪CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGUGACAAUACUUACUCUCGUUGGUGAUAAGGAACAGCU
```

(continues on next page)

(continued from previous page)

```
AAWR02037329.1/29294-29150 ---CUCGACACCACU---
↪GCCUCGGUUACCCAUCGGUGCAGUGCGGGUAGUAGUACCAAUGCUAAUUAGUUGUGAGGACCAACU
#=GC SS_cons               -----((((,<<<<<<<<<<_____________>>>>>>>>>>,,,,<<<<<<<_______>>>
↪>>>>,,,,,))))::::::::::::
#=GC RF                    ␣
↪CUGCcccaCAaCacuuguGCCUCaGUUACcCauagguGuAGUGaGgGuggcAaUACccaCcCucgUUgGuggUaAGGAaCAgCU
//
```

See also…

io/rna_structures:wuss notation for legal characters and their interpretation in the consensus secondary structure line `SS_cons`.

FASTA (Pearson) format

Note

Sequence names must not contain whitespace characters. Otherwise, the parts after the first whitespace will be dropped. The only allowed gap character is the hyphen (`-`).

Here is an example alignment in FASTA format:

```
>AL031296.1/85969-86120
CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGUAACAAUACUUAC
UCUCGUUGGUGAUAAGGAACAGCU
>AANU01225121.1/438-603
CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGUGACAAUACUUAC
UCUCGUUGGUGAUAAGGAACAGCU
>AAWR02037329.1/29294-29150
---CUCGACACCACU---GCCUCGGUUACCCAUCGGUGCAGUGCGGGUAGUAGUACCAAU
GCUAAUUAGUUGUGAGGACCAACU
```

MAF format

The multiple alignment format (MAF) is usually used to store multiple alignments on DNA level between entire genomes. It consists of independent blocks of aligned sequences which are annotated by their genomic location. Consequently, an MAF formatted MSA file may contain multiple records. MAF files start with a line:

```
##maf
```

which is optionally extended by whitespace delimited key=value pairs. Lines starting with the character (`#`) are considered comments and usually ignored.

A MAF block starts with character (`a`) at the beginning of a line, optionally followed by whitespace delimited `key=value` pairs. The next lines start with character (`s`) and contain sequence information of the form:

```
s src start size strand srcSize sequence
```

where:

- *src* is the name of the sequence source
- *start* is the start of the aligned region within the source (0-based)
- *size* is the length of the aligned region without gap characters
- *strand* is either (+) or (-), depicting the location of the aligned region relative to the source

- *srcSize* is the size of the entire sequence source, e.g. the full chromosome
- *sequence* is the aligned sequence including gaps depicted by the hyphen (-)

Here is an example alignment in MAF format (bluntly taken from the UCSC Genome browser website):

```
##maf version=1 scoring=tba.v8
# tba.v8 (((human chimp) baboon) (mouse rat))
# multiz.v7
# maf_project.v5 _tba_right.maf3 mouse _tba_C
# single_cov2.v4 single_cov2 /dev/stdin

a score=23262.0
s hg16.chr7    27578828 38 + 158545518 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s panTro1.chr6 28741140 38 + 161576975 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s baboon         116834 38 +   4622798 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
s mm4.chr6     53215344 38 + 151104725 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
s rn3.chr4     81344243 40 + 187371129 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG

a score=5062.0
s hg16.chr7    27699739 6 + 158545518 TAAAGA
s panTro1.chr6 28862317 6 + 161576975 TAAAGA
s baboon         241163 6 +   4622798 TAAAGA
s mm4.chr6     53303881 6 + 151104725 TAAAGA
s rn3.chr4     81444246 6 + 187371129 taagga

a score=6636.0
s hg16.chr7    27707221 13 + 158545518 gcagctgaaaaca
s panTro1.chr6 28869787 13 + 161576975 gcagctgaaaaca
s baboon         249182 13 +   4622798 gcagctgaaaaca
s mm4.chr6     53310102 13 + 151104725 ACAGCTGAAAATA
```

2.3.3 Command Files

The RNAlib and many programs of the ViennaRNA Package can parse and apply data from so-called *command files*. These commands may refer to structure constraints or even extensions of the RNA folding grammar (such as grammar:unstructured domains).

Commands are given as a line of whitespace delimited data fields. The syntax we use extends the constraint definitions used in the mfold or UNAfold software, where each line begins with a command character followed by a set of positions.

However, we introduce several new commands, and allow for an optional loop type context specifier in form of a sequence of characters, and an orientation flag that enables one to force a nucleotide to pair upstream, or downstream.

Constraint commands

The following set of commands is recognized:

- `F` ... Force
- `P` ... Prohibit
- `C` ... Conflicts/Context dependency
- `A` ... Allow (for non-canonical pairs)
- `E` ... Soft constraints for unpaired position(s), base pair(s), or interacting interval pairs

RNA folding grammar exensions

- `UD` ... Add ligand binding using the grammar:unstructured domains feature

Specification of the loop type context

The optional loop type context specifier `[LOOP]` may be a combination of the following:

- `E` ... Exterior loop
- `H` ... Hairpin loop
- `I` ... Internal/Interior loop
- `M` ... Multibranch loop
- `A` ... All loops

For structure constraints, we additionally allow one to address base pairs enclosed by a particular kind of loop, which results in the specifier `[WHERE]` which consists of `[LOOP]` plus the following character:

- `i` ... enclosed pair of an Interior loop
- `m` ... enclosed pair of a Multibranch loop

If no `[LOOP]` or `[WHERE]` flags are set, all contexts are considered (equivalent to `A`).

Controlling the orientation of base pairing

For particular nucleotides that are forced to pair, the following `[ORIENTATION]` flags may be used:

- `U` ... Upstream
- `D` ... Downstream

If no `[ORIENTATION]` flag is set, both directions are considered.

Sequence coordinates

Sequence positions of nucleotides/base pairs are 1-based and consist of three positions i, j, and k. Alternativly, four positions may be provided as a pair of two position ranges $[i : j]$, and $[k : l]$ using the `-` sign as delimiter within each range, i.e. `i-j`, and `k-l`.

Valid constraint commands

Below are resulting general cases that are considered *valid* constraints:

- **"Forcing a range of nucleotide positions to be paired"**:

```
F i 0 k [WHERE] [ORIENTATION]
```

  Description:

  Enforces the set of k consecutive nucleotides starting at position i to be paired. The optional loop type specifier `[WHERE]` allows to force them to appear as closing/enclosed pairs of certain types of loops.

- **"Forcing a set of consecutive base pairs to form"**::

```
F i j k [WHERE]
```

  Description:

  Enforces the base pairs $(i, j), \ldots, (i+(k-1), j-(k-1))$ to form. The optional loop type specifier `[WHERE]` allows to specify in which loop context the base pair must appear.

- **"Prohibiting a range of nucleotide positions to be paired"**:

```
P i 0 k [WHERE]
```

Description:

Prohibit a set of k consecutive nucleotides to participate in base pairing, i.e. make these positions unpaired. The optional loop type specifier `[WHERE]` allows to force the nucleotides to appear within the loop of specific types.

- **"Probibiting a set of consecutive base pairs to form"**:

```
P i j k [WHERE]
```

Description:

Probibit the base pairs $(i, j), \ldots, (i+(k-1), j-(k-1))$ to form. The optional loop type specifier `[WHERE]` allows to specify the type of loop they are disallowed to be the closing or an enclosed pair of.

- **"Prohibiting two ranges of nucleotides to pair with each other"**:

```
P i-j k-l [WHERE]
```

Description:

Prohibit any nucleotide $p \in [i, j]$ to pair with any other nucleotide $q \in [k, l]$. The optional loop type specifier `[WHERE]` allows to specify the type of loop they are disallowed to be the closing or an enclosed pair of.

- **"Enforce a loop context for a range of nucleotide positions"**:

```
C i 0 k [WHERE]
```

Description:

This command enforces nucleotides to be unpaired similar to *prohibiting* nucleotides to be paired, as described above. It too marks the corresponding nucleotides to be unpaired, however, the `[WHERE]` flag can be used to enforce specfic loop types the nucleotides must appear in.

- **"Remove pairs that conflict with a set of consecutive base pairs"**:

```
C i j k
```

Description:

Remove all base pairs that conflict with a set of consecutive base pairs $(i, j), \ldots, (i + (k - 1), j - (k - 1))$. Two base pairs (i, j) and (p, q) conflict with each other if $i < p < j < q$, or $p < i < q < j$.

- **"Allow a set of consecutive (non-canonical) base pairs to form"**:

```
A i j k [WHERE]
```

Description:

This command enables the formation of the consecutive base pairs $(i, j), \ldots, (i + (k - 1), j - (k - 1))$, no matter if they are *canonical*, or *non-canonical*. In contrast to the above `F` and `W` commands, which remove conflicting base pairs, the `A` command does not. Therefore, it may be used to allow *non-canoncial* base pair interactions. Since the RNAlib does not contain free energy contributions E_{ij} for non-canonical base pairs (i, j), they are scored as the *maximum* of similar, known contributions. In terms of a *Nussinov* like scoring function the free energy of non-canonical base pairs is therefore estimated as

$$E_{ij} = \min \left[\max_{(i,k) \in \{GC,CG,AU,UA,GU,UG\}} E_{ik}, \max_{(k,j) \in \{GC,CG,AU,UA,GU,UG\}} E_{kj} \right].$$

The optional loop type specifier `[WHERE]` allows to specify in which loop context the base pair may appear.

- **"Apply pseudo free energy to a range of unpaired nucleotide positions"**:

```
E i 0 k e
```

Description:

Use this command to apply a pseudo free energy of e to the set of k consecutive nucleotides, starting at position i. The pseudo free energy is applied only if these nucleotides are considered unpaired in the recursions, or evaluations, and is expected to be given in units of kcal $\cdot$ mol^{-1}.

- **"Apply pseudo free energy to a set of consecutive base pairs"**:

```
E i j k e
```

Description:

Use this command to apply a pseudo free energy of e to the set of base pairs $(i, j), \ldots, (i+(k-1), j-(k-1))$. Energies are expected to be given in units of kcal $\cdot$ mol^{-1}.

- **"Apply pseudo free energy to base pairs supporting an interacting interval pair"**:

```
E i-j k-l e
```

Description:

Use this command to apply a pseudo free energy of e to a structure s if $\exists (p, q) \in s : p \in [i, j] \wedge q \in [k, l]$, i.e. s contains at least one base pair supporting the interval pair $([i, j], [[k, l])$. Energies are expected to be given in units of kcal $\cdot$ mol^{-1}.

Valid domain extensions commands

- **"Add ligand binding to unpaired motif (a.k.a. unstructured domains)"**:

```
UD m e [LOOP]
```

Description:

Add ligand binding to unpaired sequence motif m (given in IUPAC format, capital letters) with binding energy e in particular loop type(s).

Example:

```
UD  AAA   -5.0    A
```

The above example applies a binding free energy of -5 kcal $\cdot$ mol^{-1} for a motif `AAA` that may be present in all loop types.

2.3.4 Energy Parameters

Modified Bases

The functions *vrna_sc_mod()*, *vrna_sc_mod_json()* and alike implement an energy correction framework to account for modified bases in the secondary structure predictions. To supply these functions with the energy parameters and general specifications of the base modification, the following `JSON` data format may be used:

JSON data must consist of a header section `modified_bases` This header is an object with the mandatory keys:

- `name` specifying a name of the modified base
- `unmodified` that consists of a single upper-case letter of the unmodified version of this base,
- the `one_letter_code` key to specify which letter is used for the modified bases in the subsequent energy parameters, and
- an array of *pairing_partners* `

The latter must be uppercase characters. An optional `sources` key may contain an array of related publications, e.g. those the parameters have been derived from.

Next to the header may follow additional keys to specify the actual energy contributions of the modified base in various loop contexts. All energy contributions must be specified in free energies ΔG in units of kcal $\cdot$ mol^{-1}. To

allow for rescaling of the free energies at temperatures that differ from the default ($37°C$), enthalpy parameters ΔH may be specified as well. Those, however are optional. The keys for free energy (at $37°C$) and enthalpy parameters have the suffixes `_energies` and `_enthalpies`, respectively.

The parser and underlying framework currently supports the following loop contexts:

- **base pair stacks** (via the `stacking` key prefix).

  This key must point to an object with one key value pair for each stacking interaction data is provided for. Here, the key consists of four upper-case characters denoting the interacting bases, where the the first two represent one strand in 5' to 3' direction and the last two the opposite strand in 3' to 5' direction. The values are energies in $kcal \cdot mol^{-1}$.

- **terminal mismatches** (via the `mismatch` key prefix).

  This key points to an object with key value pairs for each mismatch energy parameter that is available. Keys are 4 characters long nucleotide one-letter codes as used in base pair stacks above. The second and fourth character denote the two unpaired mismatching bases, while the other two represent the closing base pair.

- **dangling ends** (via the `dangle5` and `dangle3` key prefixes).

  The object behind these keys, again, consists of key value pairs for each dangling end energy parameter. Keys are 3 characters long where the first two represent the two nucleotides that form the base pair, and the third is the unpaired base that either stacks on the 3' or 5' end of the enclosed part of the base pair.

- **terminal pairs** (via the `terminal` key prefix).

  Terminal base pairs, such as AU or GU, sometimes receive an additional energy penalty. The object behind this key may list energy parameters to apply whenever particular base pairs occur at the end of a helix. Each of those parameters is specified as key value pair, where the key consists of two upper-case characters denoting the terminal base pair.

Below is a JSON template specifying most of the possible input parameters. Actual energy parameter files can be found in the source code tarball within the `misc/` subdirectory.

```
{
  "modified_base" : {
    "name" : "My modification (M)",
    "sources" : [
      {
        "authors" : "Author 1, Author 2",
        "title" : "UV-melting of modified oligos",
        "journal" : "Some journal",
        "year" : 2022,
        "doi" : "10.0000/000000"
      }
    ],
    "unmodified" : "G",
    "pairing_partners" : [
      "U","A"
    ],
    "one_letter_code" : "M",
    "fallback" : "G",
    "stacking_energies" : {
      "MAUU" :  -1.2,
      "AGMC" :  -2.73
    },
    "stacking_enthalpies" : {
      "MAUU" :  -11.1,
      "AGMC" :  -9.73
    },
    "terminal_energies" : {
      "MU" : 0.5,
```

(continues on next page)

(continued from previous page)

```
      "UM" : 0.5
    },
    "terminal_enthalpies" : {
      "MU" : 2.0,
      "UM" : 2.0
    },
    "mismatch_energies" : {
      "CMGM" : -1.11,
      "AGUM" : -0.73
    },
    "mismatch_enthalpies" : {
      "CMGM" : -11.11,
      "AGUM" : -7.73
    },
    "dangle5_energies" : {
      "UAM" : -1.01
    },
    "dangle5_enthalpies" : {
      "UAM" : -6.01
    },
    "dangle3_energies" : {
      "CGM" : -2.1,
      "GCM" : -1.3
    }
  }
}
```

An actual example of real-world data may look like

```
{
  "modified_base" : {
    "name" : "Pseudouridine",
    "sources" : [
      {
        "authors": "Graham A. Hudson, Richard J. Bloomingdale, and Brent M. Znosko",
        "title" : "Thermodynamic contribution and nearest-neighbor parameters of␣
↪pseudouridine-adenosine base pairs in oligoribonucleotides",
        "journal" : "RNA 19:1474-1482",
        "year" : 2013,
        "doi" : "10.1261/rna.039610.113"
      }
    ],
    "unmodified" : "U",
    "pairing_partners" : [
      "A", "G"
    ],
    "one_letter_code" : "P",
    "fallback" : "U",
    "stacking_energies" : {
      "APUA" :  -2.8,
      "CPGA" : -2.77,
      "GPCA" : -3.29,
      "UPAA" : -1.62,
      "PAAU" : -2.10,
      "PCAG" : -2.49,
      "PGAC" : -2.2,
```

(continues on next page)

(continued from previous page)

```
      "PUAA" : -2.74
    },
    "stacking_enthalpies" : {
      "APUA" : -22.08,
      "CPGA" : -16.23,
      "GPCA" : -24.07,
      "UPAA" : -20.81,
      "PAAU" : -12.47,
      "PCAG" : -17.29,
      "PGAC" : -11.19,
      "PUAA" : -26.94
    },
    "terminal_energies" : {
      "PA" : 0.31,
      "AP" : 0.31
    },
    "terminal_enthalpies" : {
      "PA" : -2.04,
      "AP" : -2.04
    },
    "duplexes" : {
      "CGAPACGGCUAUGC" : {
        "length1" : 7,
        "length2" : 7,
        "dG37"    : -9.93,
        "dG37_p"  : -10.12
      },
      "CGCPACGGCGAUGC" : {
        "length1" : 7,
        "length2" : 7,
        "dG37"    : -10.96,
        "dG37_p"  : -11.17
      },
      "CGGPACGGCCAUGC" : {
        "length1" : 7,
        "length2" : 7,
        "dG37"    : -11.71,
        "dG37_p"  : -11.53
      },
      "CGUPACGGCAAUGC" : {
        "length1" : 7,
        "length2" : 7,
        "dG37"    : -9.10,
        "dG37_p"  : -8.83
      },
      "CGAPCCGGCUAGGC" : {
        "length1" : 7,
        "length2" : 7,
        "dG37"    : -11.92,
        "dG37_p"  : -11.53
      },
      "CGCPCCGGCGAGGC" : {
        "length1" : 7,
        "length2" : 7,
        "dG37"    : -12.93,
        "dG37_p"  : -12.57
```

(continues on next page)

(continued from previous page)

```
    },
    "CGGPCCGGCCAGGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -12.76,
      "dG37_p"  : -12.94
    },
    "CGUPCCGGCAAGGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -9.76,
      "dG37_p"  : -10.24
    },
    "CGAPGCGGCUACGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -11.45,
      "dG37_p"  : -11.40
    },
    "CGCPGCGGCGACGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -12.35,
      "dG37_p"  : -12.45
    },
    "CGGPGCGGCCACGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -12.59,
      "dG37_p"  : -12.81
    },
    "CGUPGCGGCAACGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -10.34,
      "dG37_p"  : -10.11
    },
    "CGAPUCGGCUAAGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -10.42,
      "dG37_p"  : -10.86
    },
    "CGCPUCGGCGAAGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -12.06,
      "dG37_p"  : -11.91
    },
    "CGGPUCGGCCAAGC" : {
      "length1" : 7,
      "length2" : 7,
      "dG37"    : -12.51,
      "dG37_p"  : -12.27
    },
    "CGUPUCGGCAAAGC" : {
```

(continues on next page)

(continued from previous page)

```
        "length1" : 7,
        "length2" : 7,
        "dG37"    : -9.51,
        "dG37_p"  : -9.58
      },
      "GCGCAPCGCGUA" : {
        "length1" : 6,
        "length2" : 6,
        "dG37"    : -9.90,
        "dG37_p"  : -9.71
      },
      "GCGCCPCGCGGA" : {
        "length1" : 6,
        "length2" : 6,
        "dG37"    : -10.63,
        "dG37_p"  : -10.84
      },
      "GCGCGPCGCGCA" : {
        "length1" : 6,
        "length2" : 6,
        "dG37"    : -10.43,
        "dG37_p"  : -10.46
      },
      "GCGCUPCGCGAA" : {
        "length1" : 6,
        "length2" : 6,
        "dG37"    : -8.55,
        "dG37_p"  : -8.50
      },
      "PAGCGCAUCGCG" : {
        "length1" : 6,
        "length2" : 6,
        "dG37"    : -8.93,
        "dG37_p"  : -8.99
      },
      "PCGCGCAGCGCG" : {
        "length1" : 6,
        "length2" : 6,
        "dG37"    : -9.56,
        "dG37_p"  : -9.66
      },
      "PGGCGCACCGCG" : {
        "length1" : 6,
        "length2" : 6,
        "dG37"    : -10.30,
        "dG37_p"  : -10.27
      },
      "PUGCGCAACGCG" : {
        "length1" : 6,
        "length2" : 6,
        "dG37"    : -9.77,
        "dG37_p"  : -9.65
      }
    }
  }
}
```

2.4 Tutorials

Here you find some more or less elaborate tutorials and manuals that demonstrate how to use our software.

> **Warning**
>
> The tutorials provided below have not been updated for quite some time. Thus, some of the described features may not work as expected and novel features of our programs may not be documented yet.

2.4.1 Global RNA Secondary Structure Prediction

Several tools for structure prediction of single RNA sequences are available within the `ViennaRNA Package`, each with its own special subset of implemented algorithms.

The Program `RNAfold`

Introduction

Our first task will be to do a structure prediction using `RNAfold`. This should get you familiar with the input and output format as well as the graphical output produced.

`RNAfold` reads single RNA sequences, computes their minimum free energy (`MFE`) structures, and prints the result together with the corresponding `MFE` structure in dot-bracket notation. This is the default mode if no further command line parameters are provided. Please note, that the `RNAfold` program can either be used in *interactive mode*, where the program expects the input from *stdin*, or in *batch processing mode* where you provide the input sequences as text files.

Partition function

To activate computation of the partition function for each sequence, the `-p` option must be set. From the partition function

$$Q = \sum_{s \in \Omega} exp(-E(s)/RT)$$

over the ensemble of all possible structures Ω, with temperature T and gas constant R, `RNAfold` then computes the ensemble free energy $G = -RT \cdot ln(Q)$, and frequency of the `MFE` structure s_{mfe} within the ensemble

$$p = exp(-E(s_{mfe})/RT)/Q$$

Ensemble diversity

By default, the `-p` option also activates the computation of base pairing probabilities p_{ij}. From this data, `RNAfold` then determines the ensemble diversity

$$\langle d \rangle = \sum_{ij} p_{ij} \cdot (1 - p_{ij}),$$

i.e. the expected distance between any two secondary structure, as well as the `centroid` structure, i.e. the structure s_c with the least Boltzmann weighted distance

$$d_{\Omega}(s_c) = \sum_{s \in \Omega} p(s) d(s_c, s)$$

to all other structures $s \in \Omega$.

Maximum Expected Accuracy

Another useful structure representative one can determine from base pairing probabilities p_{ij} is the structure that exhibits the *maximum expected accuracy (MEA)*. By assuming the base pair probability is a good measure of

correctnes of a pair (i, j), the expected accuracy of a structure s is

$$\mathrm{EA}(s) = \sum_{(i,j)\in s} 2\gamma p_{ij} + \sum_{\nexists(i,j)\in s}^{i} q_i$$

with $q_i = 1 - \sum_j p_{ij}$ and weighting factor γ that allows us to weight paired against unpaired positions. `RNAfold` uses a dynamic programming scheme similar to the *Maximum Matching algorithm* of Ruth Nussinov to find the structure s that minimizes the above equation.

The `RNAfold` program provides a large amount of additional computation modes that will be partly covered below. To get a full list of all computation modes available, please consult the `RNAfold` man page or the outputs of `RNAfold -h` and `RNAfold --detailed-help`.

MFE structure of a single sequence

- Use a text editor (emacs, vi, nano, gedit) to prepare an input file by pasting the text below and save it under the name `test.seq` in your `Data` folder:

```
> test
CUACGGCGCGGCGCCCUUGGCGA
```

- Compute the best (MFE) structure for this sequence using *batch processing mode*

```
$ RNAfold test.seq
CUACGGCGCGGCGCCCUUGGCGA
...........((((...)))). ( -5.00)
```

- or use the *interactive mode* and redirect the content of `test.seq` to *stdin*

```
$ RNAfold < text.seq
CUACGGCGCGGCGCCCUUGGCGA
...........((((...)))). ( -5.00)
```

- alternatively, you could use the *interactive mode* and manually enter the sequence as soon as `RNAfold` prompts for input

```
$ RNAfold
Input string (upper or lower case); @ to quit
....,....1....,....2....,....3....,....4....,....5....,....6....,....7....,....8
CUACGGCGCGGCGCCCUUGGCGA
length = 23

CUACGGCGCGGCGCCCUUGGCGA
...........((((...)))).
 minimum free energy =  -5.00 kcal/mol
```

All the above variants to compute the MFE and the corresponding structure result in identical output, except for slight variations in the formatting when true *interactive mode* is used. The last line(s) of the text output contains the predicted MFE structure in *dot-bracket notation* and its free energy in `kcal/mol`. A dot in the dot-bracket notation represents an unpaired position, while a base pair (i, j) is represented by a pair of matching parentheses at position i and j.

If the input was `FASTA` formatted, i.e. the sequence was preceded by a header line with sequence identifier, `RNAfold` creates a structure layout file named `test_ss.ps`, where `test` is the sequence identifier as provided through the `FASTA` header. In case the header was omitted the output file name simply is `rna.ps`.

Let's take a look at the output file with your favorite `PostScript` viewer, e.g. `gv`.

Note

In contrast to bitmap based image files (such as GIF or JPEG) PostScript files contain resolution independent *vector graphics*, suitable for publication. They can be viewed on-screen using a postscript viewer such as `gv` or `evince`. Also note the `&` at the end of the following command line that simply detaches the program call and immediately starts the program in the background.

```
$ gv test_ss.ps &
```

Compare the dot-bracket notation to the PostScript drawing shown in the file `test_ss.eps`.

You can use the `-t` option to change the layout algorithm `RNAfold` uses to produce the plot. The most simply layout is the *radial* layout that can be chosen with `-t 0`. Here, each nucleotide in a loop is equally spaced on its enclosing circle. The more sophisticated `Naview` layout algorithm is used by default but may be explicitly chosen through `-t 1`. A hidden feature can be found with `-t 2`, where `RNAfold` creates a most simple circular plot.

The calculation above does not tell us whether we can actually trust the predicted structure. In fact, there may be many more possible structures that might be equally probable. To find out about that, let's have a look at the equilibrium ensemble instead.

Equilibrium ensemble properties

- Run:

  ```
  $ RNAfold -p --MEA
  ```

  to compute the partition function, pair probabilities, centroid structure, and the maximum expected accuracy (MEA) structure.

- Have a look at the generated PostScript files `test_ss.ps` and `test_dp.ps`

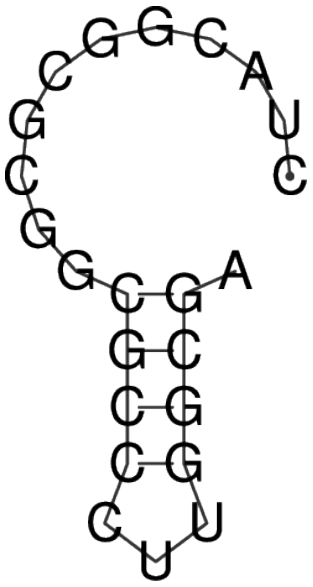
  ```
  $ RNAfold -p --MEA test.seq
  CUACGGCGCGGCGCCCUUGGCGA
  ...........((((...)))). ( -5.00)
  ....{,{{...||||...)}}}. [ -5.72]
  ........................ {  0.00 d=4.66}
  ......((...))((...))... {  2.90 MEA=14.79}
   frequency of mfe structure in ensemble 0.311796; ensemble diversity 6.36
  ```

Here the last four lines are new compared to the text output without the `-p --MEA` options. The partition function is already a rough measure for the well-definedness of the `MFE` structure. The third line shows a condensed representation of the pair probabilities of each nucleotide, similar to the dot-bracket notation, followed by the ensemble free energy ($G = -kT \cdot ln(Z)$) in units of `kcal/mol`. Here, the dot-bracket like notation consists of additional characters that denote the pairing propensity for each nucleotide. `.` denotes bases that are essentially unpaired, `,` weakly paired, `|` strongly paired without preference, `{}`, `()` weakly ($> 33\%$) upstream (downstream) paired or strongly ($> 66\%$) up-/downstream paired bases, respectively.

The next two lines represent (i) the centroid structure with its free energy and distance to the ensemble, and (ii) the MEA structure, it's free energy and the actual accuracy. The very last line shows the frequency of the MFE structure in the ensemble of secondary structures and the diversity of the ensemble as discussed above.

Note that the MFE structure is adopted only with 31% probability, also the diversity is very high for such a short sequence.

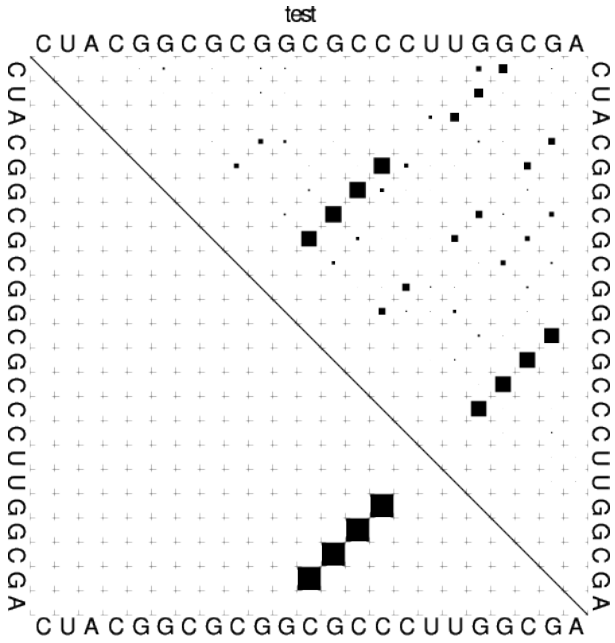
Rotate the structure plot



To rotate the secondary structure plot that is generated by `RNAfold` the `ViennaRNA Package` provides the perl script utility `rotate_ss.pl`. Just read the `perldoc` for this tool to know how to handle the rotation and use the information to get your secondary structure in a vertical position.

```
$ perldoc rotate_ss.pl
```

The base pair probability dot plot



The *dot plot* (`test_dp.ps`) shows the pair probabilities within the equilibrium ensemble as $n \times n$ matrix, and is an excellent way to visualize structural alternatives. A square at row i and column j indicates a base pair. The area of a square in the upper right half of the matrix is proportional to the probability of the base pair (i, j) within the equilibrium ensemble. The lower left half shows all pairs belonging to the `MFE` structure. While the MFE consists of a single helix, several different helices are visualized in the pair probabilities.

While a base pair probability dot-plot is quite handy to interpret for short sequences, it quickly becomes confusing the longer the RNA sequence is. Still, this is (currently) the only output of base pair probabilities for the `RNAfold` program. Nevertheless, since the dot plot is a true `PostScript` file, one can retrieve the individual base pair

probabilities by parsing its textual content.

- Open the dot plot with your favorite text editor
- Locate the lines that that follow the scheme

```
i j v ubox
```

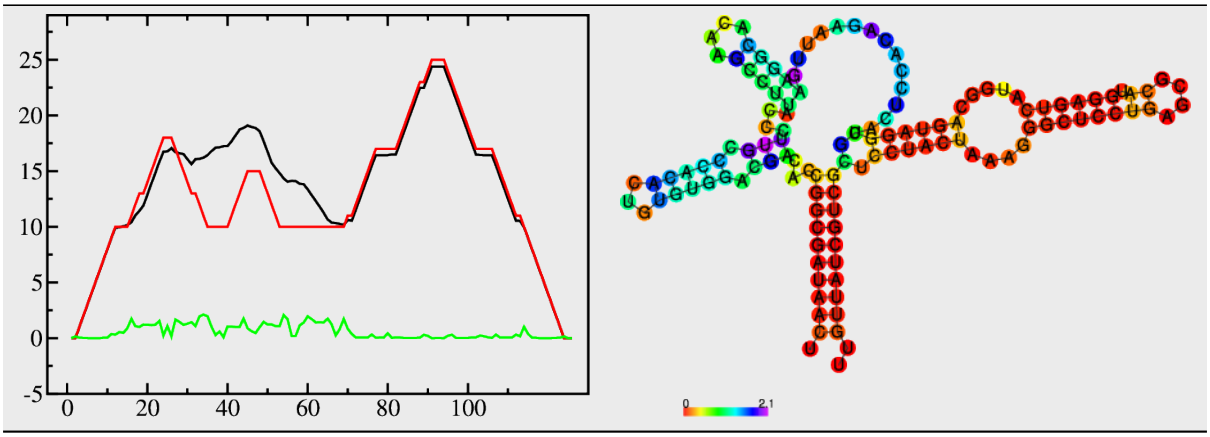
where i and j are integer values and v is a floating point decimal with values between 0 and 1. These are the data for the boxes drawn in the upper triangle. The integer values i and j denote the nucleotide positions while the value v is the square-root of the probability of base pair (i, j). Thus, the actual base pair probability $p(i, j) = v \cdot v$.

Mountain and Reliability plot

Next, let's use the `relplot.pl` utility to annotate which parts of a predicted MFE structure are well-defined and thus more reliable. Also let's use a real example for a change and produce yet another representation of the predicted structure, the *mountain plot*.

Fold the 5S rRNA sequence and visualize the structure. (The `5S.seq` is shipped with the tutorial)

```
$ RNAfold -p 5S.seq
$ mountain.pl 5S_dp.ps | xmgrace -pipe
$ relplot.pl 5S_ss.ps 5S_dp.ps > 5S_rss.ps
```



A mountain plot is especially useful for long sequences where conventional structure drawings become terribly cluttered. It is a xy-diagram plotting the number of base pairs enclosing a sequence position *versus* the position. The `Perl` script `mountain.pl` transforms a dot plot into the mountain plot coordinates which can be visualized with any xy-plotting program, e.g. `xmgrace`.

The resulting plot shows three curves, two mountain plots derived from the `MFE` structure (red) and the pairing probabilities (black) and a positional entropy curve (green). Well-defined regions are identified by low entropy. By superimposing several mountain plots structures can easily be compared.

The perl script `relplot.pl` adds reliability information to a RNA secondary structure plot in the form of color annotation. The script computes a well-definedness measure we call `positional entropy`

$$S(i) = -\sum p_{ij} \log(p_{ij})$$

and encodes it as color hue, ranging from red (low entropy, well-defined) via green to blue and violet (high entropy, ill-defined). In the example above two helices of the 5S RNA are well-defined (red) and indeed predicted correctly, the left arm is not quite correct and disordered.

For the figure above we had to rotate and mirror the structure plot, e.g.

```
$ rotate_ss.pl -a 180 -m 5S_rss.ps > 5S_rot.ps
```

Batch job processing

In most cases, one doesn't only want to predict the structure and equilibrium probabilities for a single RNA sequence but a set of sequences. `RNAfold` is perfectly suited for this task since it provides several different mechanisms to support batch job processing. First, in *interactive* mode, it only stops processing input from *stdin* if it is requested to do so. This means that after processing one sequence, it will prompt for the input of the next sequence. Entering the `@` character will forcefully abort processing. In situations where the input is provided through input stream redirection, it will end processing as soon stream is closed.

In constrat to that, the *batch processing mode* where one simply specifies input files as so-called unnamed command line parameters, the number of input sequences is more or less unlimited. You can specify as many input files as your terminal emulator allows, and each input file may consist of arbitrarily many sequences. However, please note that mixing `FASTA` and non-fasta input is not allowed and will most likely produce bogus output.

Assume you have four input files `file_0.fa`, `file_1.fa`, `file_2.fa`, and `file_3.fa`. Each file contains a set of RNA sequences in `FASTA` format. Predicting secondary structures for all sequences in all files with a single call to `RNAfold` and redirecting the output to a file `all_sequences_output.fold` can be achieved like this: .. code:

```
$ RNAfold file_0.fa file_1.fa file_2.fa file_3.fa > all_sequences_output.fold
```

The above call to `RNAfold` will open each of the files and process the sequences sequentially. This, however, might take a long time and the sequential processing will most likely bore out your multi-core workstation or laptop computer, since only a single core is used for the computations while the others are idle. If you happen to have more than a single CPU core and want to take advantage of the available parallel processing power, you can use the `-j` option of `RNAfold` to split the input into concurrent jobs.

```
$ RNAfold -j file_*.fa > all_sequences_output.fold
```

This command will uses as many CPU cores as available and, therefore, process you input much faster. If you want to limit the number of concurrent jobs to a particular number, say 2, to leave the remaining cores available for other tasks, you can append the number of jobs directly to the `-j` option: .. code:

```
$ RNAfold -j2 file_*.fa > all_sequences_output.fold
```

Note here, that there must not be any space between the `j` and the number of jobs.

Now imagine what happens if you have a larger set of sequences that are not stored in `FASTA` format. If you would serve such an input to `RNAfold`, it would happily process each of the sequences but always over-write the structure layout and dot-plot files, since the default names for these files are `rna.ps` and `dot.ps` for any sequence. This is usually an undesired behavior, where `RNAfold` and the `--auto-id` option becomes handy. This option flag forces `RNAfold` to automatically create a sequence identifier for each input, thus using different file names for each single output. The identifier that is created follows the form .. code:

```
sequence_XXXX
```

where `sequence` is a prefix, followed by the delimiting character `_`, and an increasing 4-digit number `XXXX` starting at 0000. This feature is even useful if the input is in `FASTA` format, but one wants to enforce a novel naming scheme for the sequences. As soon as the `--auto-id` option is set, `RNAfold` will ignore any id taken from existing `FASTA` headers in the input files.

See also the man page of `RNAfold` to find out how to modify the prefix, delimiting character, start number and number of digits.

- Create an input file with many RNA sequences, each on a separate line, e.g.:

  ```
  $ randseq -n 127 > many_files.seq
  ```

- Compute the MFE structure for each of the sequences and generate output ids with numbers between 100 and 226 and prefix `test_seq`:

```
$ RNAfold --auto-id --id-start=100 --id-prefix="test_seq" many_files.seq
```

Add constraints to the structure prediction

For some scientific questions one requires additional constraints that must be enforced when predicting secondary structures. For instance, one might have resolved parts of the structure already and is simply interested in the optimal conformation of the remaining part of the molecule. Another example would be that one already knows that particular nucleotides can not participate in any base pair, since they are physically hindered to do so. These types of constraints are termed *hard* constraints and they can enforce or prohibit particular conformations, thus include or omit structures with these feature from the set candidate ensemble.

Another type of constraints are so-called *soft* constraints, that enable one to adjust the free energy contributions of particular conformations. For instance, one could add a bonus energy if a particular (stretch of) nucleotides is left unpaired to emulate the binding free energy of a single strand binding protein. The same can be applied to base pairs, for instance one could add a penalizing energy term if a particular base pair is formed to make it less likely.

The `RNAfold` programs comes with a comprehensive hard and soft constraints support and provides several convenience command line parameters to ease constraint application.

The most simple hard constraint that can be applied is the maximum base pair span, i.e. the maximum number of nucleotides a particular base pair may span. This constraint can be applied with the `--maxBPspan` option followed by an integer number.

- Compute the secondary structure for the `5S.seq` input file
- Now limit the maximum base pair span to 50 and compare both results:

```
$ RNAfold --maxBPspan 50 5S.seq
```

Now assume you already know parts of the structure and want to *fill-in* an optimal remaining part. You can do that by using the `-C` option and adding an additional line in dot-bracket notation to the input (after the sequence) that corresponds to the known structure:

- Prepare the input file `hard_const_example.fa`:

```
>my_constrained_sequence
GCCCUUGUCGAGAGGAACUCGAGACACCCACUACCCACUGAGGACUUUCG
..((((.....))))
```

  Note here, that we left out the remainder of the input structure constraint that will eventually be used to enforce a helix of 4 base pairs at the beginning of the sequence. You may also fill the remainder of the constraint with dots to silence any warnings issued by `RNAfold`.

- Compute the MFE structure for the input:

```
$ RNAfold hard_const_example.fa
>my_constrained_sequence
GCCCUUGUCGAGAGGAACUCGAGACACCCACUACCCACUGAGGACUUUCG
........((((((...((((.................))))..)))))) ( -8.00)
```

- Now compute the MFE structure under the provided constraint:

```
$ RNAfold -C hard_const_example.fa
>my_constrained_sequence
GCCCUUGUCGAGAGGAACUCGAGACACCCACUACCCACUGAGGACUUUCG
..((((.....))))....(((((..((((........)).)).)))))) ( -7.90)
```

- Due to historic reasons, the `-C` option alone only forbids any base pairs that are incompatible with the constraint, rather than enforcing the constraint. Thus, if you compute equilibrium probabilities, structures that are missing the small helix in the beginning are still part of the ensemble. If you want to compute the pairing probabilities upon forcing the small helix at the beginning, you can add the `--enforceConstraint` option:

```
$ RNAfold -p -C --enforceConstraint hard_const_example.fa
>my_constrained_sequence
GCCCUUGUCGAGAGGAACUCGAGACACCCACUACCCACUGAGGACUUUCG
..((((.....))))....(((((..((((........)).))..))))) ( -7.90)
```

Have a look at the differences in ensemble free energy and base pair probabilities between the results obtained with and without the `--enforceConstraint` option.

A more thorough alternative to provide constraints is to use the `--commands` option and a corresponding *commands file*. This allows one to specify constraints on nucleotide or base pair level and even to restrict a constraint to particular loop types. A commands file is a simple multi column text file with one constraint on each line. A line starts with a one- or two-letter command, followed by multiple values that specify the addressed nucleotides, the loop context restriction, and, for soft constraints, the strength of the constraint in $kcal/mol$. The syntax is as follows:

```
F i 0 k   [TYPE] [ORIENTATION] # Force nucleotides i...i+k-1 to be paired
F i j k   [TYPE] # Force helix of size k starting with (i,j) to be formed
P i 0 k   [TYPE] # Prohibit nucleotides i...i+k-1 to be paired
P i j k   [TYPE] # Prohibit pairs (i,j),...,(i+k-1,j-k+1)
P i-j k-l [TYPE] # Prohibit pairing between two ranges
C i 0 k   [TYPE] # Nucleotides i,...,i+k-1 must appear in context TYPE
C i j k          # Remove pairs conflicting with (i,j),...,(i+k-1,j-k+1)
E i 0 k e        # Add pseudo-energy e to nucleotides i...i+k-1
E i j k e        # Add pseudo-energy e to pairs (i,j),...,(i+k-1,j-k+1)
```

with

```
[TYPE]        = { E, H, I, i, M, m, A }
[ORIENTATION] = { U, D }
```

- Prepare a commands file `test.constraints` that forces the first 5 nucleotides to pair and the following 3 nucleotides to stay unpaired as part of a multi-branch loop:

  ```
  F 1 0 5
  C 6 0 3 M
  ```

- Use the `randseq` program to generate multiple sequences and compute the MFE structure for each under the constraints prepared earlier:

  ```
  $ randseq -n 20 | RNAfold --commands test.constraints
  ```

  Inspect the output to assure yourself that hte commands have been applied

A few much more sophisticated constraints will be discussed below.

SHAPE directed RNA folding

In order to further improve the quality of secondary structure predictions, mapping experiments like SHAPE (selective 2'-hydroxyl acylation analyzed by primer extension) can be used to exerimentally determine the pairing status for each nucleotide. In addition to thermodynamic based secondary structure predictions, RNAfold supports the incorporation of this additional experimental data as soft constraints.

If you want to use SHAPE data to guide the folding process, please make sure that your experimental data is present in a text file, where each line stores three white space separated columns containing the position, the abbreviation and the normalized SHAPE reactivity for a certain nucleotide.

```
1 G 0.134
2 C 0.044
3 C 0.057
```

(continues on next page)

(continued from previous page)

```
4 G 0.114
5 U 0.094
   ...
   ...
   ...
71 C 0.035
72 G 0.909
73 C 0.224
74 C 0.529
75 A 1.475
```

The second column, which holds the nucleotide abbreviation, is optional. If it is present, the data will be used to perform a cross check against the provided input sequence. Missing SHAPE reactivities for certain positions can be indicated by omitting the reactivity column or the whole line. Negative reactivities will be treated as missing. Once the SHAPE file is ready, it can be used to constrain folding:

```
$ RNAfold --shape=rna.shape --shapeMethod=D < rna.seq
```

A small compilation of reference data taken from Hajdin *et al.* [2013] is available online https://weeks.chem.unc.edu/data-files/ShapeKnots_DATA.zip. However, the included reference structures are only available in connect (`.ct`) format and require conversion into dot-bracket notation to compare them against predicted structures with `RNAfold`. Furthermore, the normalized `SHAPE` data is available as Excel spreadsheet and also requires some pre-processing to make it available for `RNAfold`.

Adding ligand interactions

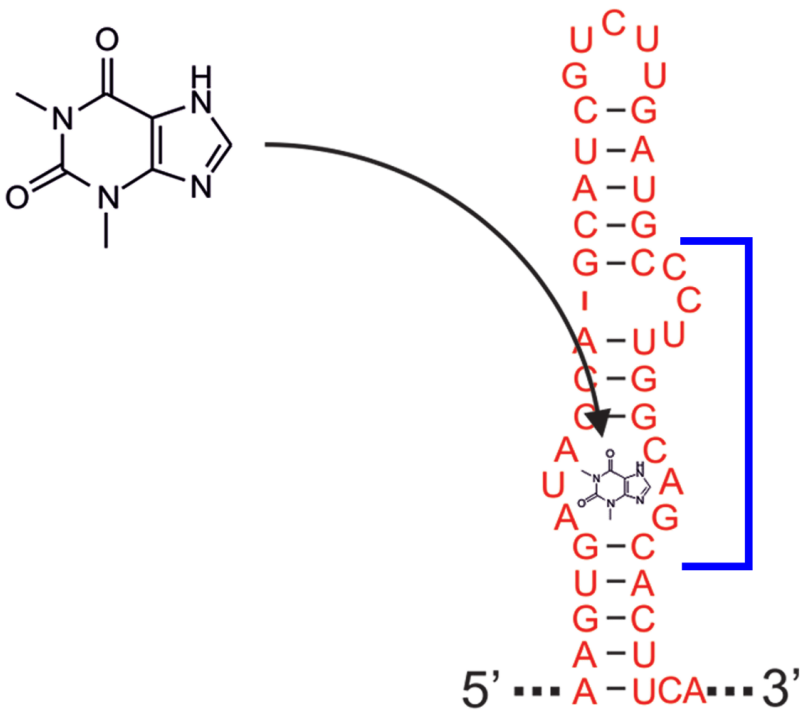
RNA molecules are known to interact with other molecules, such as additional RNAs, proteins, or other small ligand molecules. Some interactions with small ligands that take place in loops of an RNA structure can be modeled in terms of soft constraints. However, to stay compatible with the recursive decomposition scheme for secondary structures they are limited to the unpaired nucleotides of hairpins and internal loops.

The `RNAlib` library of the `ViennaRNA Package` implements a most general form of constraints capability. However, the available programs do not allow for a full access to the implemented features. Nevertheless, `RNAfold` provides a convenience option that allows to easily include ligand binding to hairpin- or interior-loop like aptamer motifs. For that purpose, a user needs only to provide motif and a binding free energy.

Consider the following example file `theo.fa` for a theophylline triggered riboswitch with the sequence:

```
>theo-switch
GGUGAUACCAGAUUUCGCGAAAAAUCCCUUGGCAGCACCUCGCACAUCUUGUUGUC
UGAUUAUUGAUUUUUCGCGAAACCAUUUGAUCAUAUGACAAGAUUGAG
```

The theopylline aptamer structure has been actively researched during the last two decades.



Although the actual aptamer part (marked in blue) is not a simple interior loop, it can still be modeled as such. It consists of two delimiting base pairs (G,C) at the 5' site, and another (G,C) at its 3' end. That is already enough to satisfy the requirements for the `--motif` option of `RNAfold`. Together with the aptamer sequence motif, the entire aptamer can be written down in dot-bracket form as:

```
GAUACCAG&CCCUUGGCAGC
(...((((&)...)))...)
```

Note here, that we separated the 5' and 3' part from each other using the & character. This enables us to omit the variable hairpin end of the aptamer from the specification in our model.

The only ingredient that is still missing is the actual stabilizing energy contribution induced by the ligand binding into the aptamer pocket. But several experimental and computational studies have already determined dissociation constants for this system. Jenison *et al.* [1994], for instance, determined a dissociation constant of $K_d = 0.32\mu M$ which, for standard reference concentration $c = 1mol/L$, can be translated into a binding free energy

$$\Delta G = RT \cdot \ln \frac{K_d}{c} \approx -9.22\ kcal/mol$$

Finally, we can compute the MFE structure for our example sequence

```
$ RNAfold -v --motif "GAUACCAG&CCCUUGGCAGC,(...((((&)...)))...),-9.22" theo.fa
```

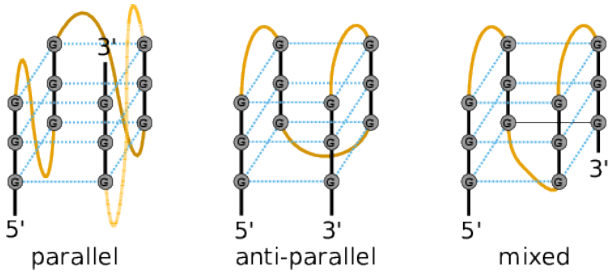
Compare the predicted MFE structure with and without modeling the ligand interaction. You may also enable partition function computation to compute base pair probabilities, the centroid structure and MEA structure to investigate the effect of ligand binding on ensemble diversity.

G-quadruplexes

G-Quadruplexes are a common conformation found in G-rich sequences where four runs of consecutive G's are separated by three short sequence stretches.



They form local self-enclosed stacks of G-quartets bound together through 8 Hogsteen-Watson Crick bonds and further stabilized by a metal ion (usually potassium).



To acknowledge the competition of regular secondary structure and G-quadruplex formation, the `ViennaRNA Package` implements an extension to the default recursion scheme. For that purpose, G-quadruplexes are simply considered a different type of substructure that may be incorporated like any other substructure. The free energy of a particular G-quadruplex at temperature T is determined by a simple energy model

$$E(L, l_{tot}, T) = a(t) \cdot (L - 1) + b(T) \cdot ln(l_{tot} - 2)$$

that only considers the number of stacked layers L and the total size of the three linker sequences $l_{tot} = l_1 + l_2 + l_3$ connecting the G runs. Linker sequence and assymetry effects as well as relative strand orientations (parallel, anti-parallel or mixed) are entirely neglected in this model. The free energy parameters

$$a(T) = H_a + TS_a$$

and

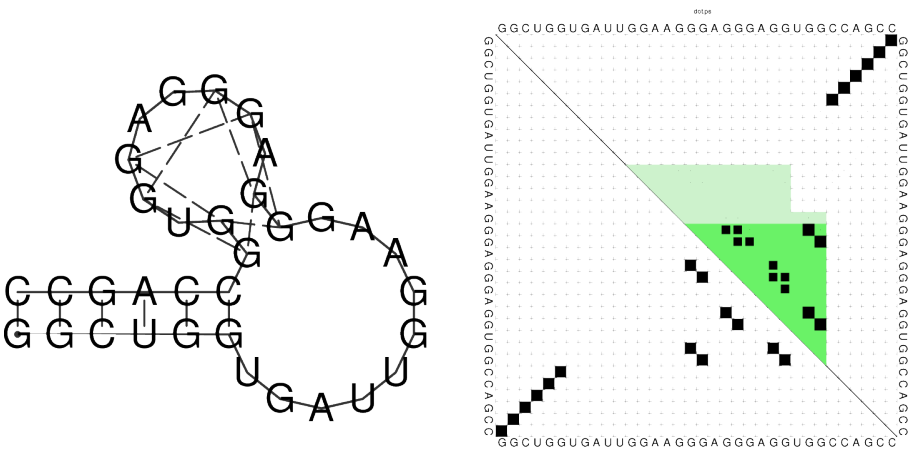
$$b(T) = H_b + TS_b$$

have been determined from experimental UV-melting data taken from Zhang *et al.* [2011].

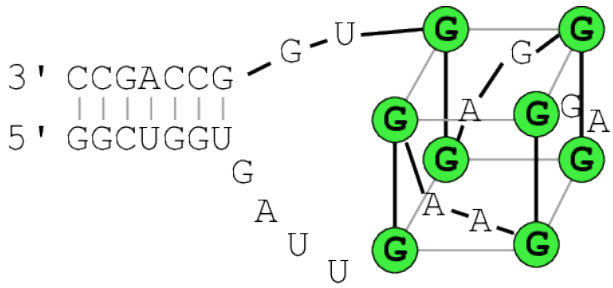
`RNAfold` allows one to activate the G-quadruplex implementation by simply providing the `-g` switch. G-quadruplexes are then taken into account for MFE and equilibrium probability computations.

```
$ echo "GGCUGGUGAUUGGAAGGGAGGGAGGUGGCCAGCC" | RNAfold -g -p
GGCUGGUGAUUGGAAGGGAGGGAGGUGGCCAGCC
((((((.........++.++..++.++)))))) (-21.39)
((((((.........(.........)))))))  [-21.83]
((((((.........++.++..++.++)))))) {-21.39 d=0.04}
 frequency of mfe structure in ensemble 0.491118; ensemble diversity 0.08
```

The resulting structure layout and dot plot `PostScript` files depict the prediced G-quadruplexes as hairpin-like loops with additional bonds between the interacting G's, and green triangles where the color intensity encodes the G-quadruplex probability, respectively. Have a closer look at the actual G-quadruplex probabilities by opening the dot plot *PostScript* file with a text browser again.



A better drawing of the predicted G-quadruplex might look as follows



Repeat the above analysis for other RNA sequences that might contain and form a G-quadruplex, e.g. the human telomerase RNA component hTERC:

```
>hTERC
AGAGAGUGACUCUCACGAGAGCCGCGAGAGUCAGCUUGGCCAAUCCGUGCGGUCGG
CGGCCGCUCCCUUUAUAAGCCGACUCGCCCGGCAGCGCACCGGGUUGCGGAGGGUG
GGCCUGGGAGGGGUGGUGGCCAUUUUUUGUCUAACCCUAACUGAGAAGGGCGUAGG
CGCCGUGCUUUUGCUCCCCGCGCGCUGUUUUUCUCGCUGACUUUCAGCGGGCGGAA
AAGCCUCGGCCUGCCGCCUUCCACCGUUCAUUCUAGAGCAAACAAAAAAUGUCAGC
UGCUGGCCCGUUCGCCCCUCCCGGGGACCUGCGGCGGGUCGCCUGCCCAGCCCCCG
AACCCCGCCUGGAGGCCGCGGUCGGCCCGGGGCUUCUCCGGAGGCACCCACUGCCA
CCGCGAAGAGUUGGGCUCUGUCAGCCGCGGGUCUCUCGGGGGCGAGGGCGAGGUUC
AGGCCUUUCAGGCCGCAGGAAGAGGAACGGAGCGAGUCCCCGCGCGCGGCGCGAUU
CCCUGAGCUGUGGGACGUGCACCCAGGACUCGGCUCACACACAUGC
```

SSB protein interaction

Similar to the ligand interactions discussed above, a single strand binding (SSB) protein might bind to consecutively unpaired sequence motifs. To model such interactions the `ViennaRNA Package` implements yet another extension to the folding grammar to cover all cases a protein may bind to, termed *unstructured domains*. This is in contrast to the ligand binding example above that uses the soft constraints implementation, and is, therefore, restricted to unpaired hairpin- and interior-loops.

To make use of this implementation in `RNAfold` one has to resort to *command files* again. Here, an unstructured domain (UD) can be easily added using the following syntax:

```
UD m e [LOOP]
```

where `m` is the sequence motif the protein binds to in IUPAC format, `e` is the binding free energy in $kcal/mol$, and the optional `LOOP` specifier allows for restricting the binding to particular loop types, e.g. `M` for multibranch loops, or `E` for the exterior loop. See the syntax for command files above for an overview of all loop types available.

As an example, consider the protein binding experiment taken from Forties and Bundschuh [2010]. Here, the authors investigate a hypothetical unspecific RNA binding protein with a footprint of 6 nt and a binding energy of $\Delta G = -10\ kcal/mol$ at 1 M. With $T = 37°C$ and

$$\Delta G = RT \cdot \ln \frac{K_d}{c}$$

this translates into a dissociation constant of

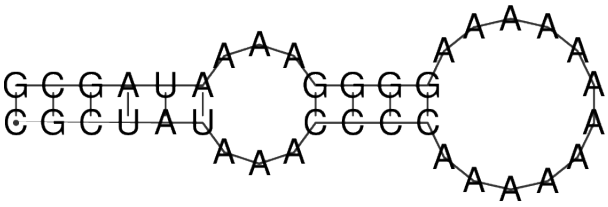
$$K_d = exp(\Delta G/RT) = 8.983267433 \cdot 10^{-8}.$$

Hence, the binding energies at 50 nM, 100 nM, 400 nM, and 1 μM are 0.36 $kcal/mol$, -0.07 $kcal/mol$, -0.92 $kcal/mol$, and -1.49 $kcal/mol$, respectively.

The RNA sequence file `forties_bundschuh.fa` for this experiment is:

```
>forties_bundschuh
CGCUAUAAACCCCAAAAAAAAAAAAAGGGGAAAAUAGCG
```

which yields the following MFE structure



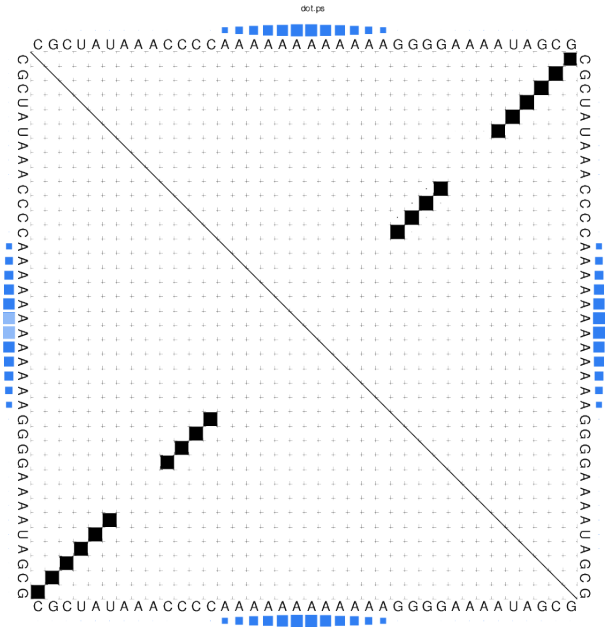
To model the protein binding for this example with `RNAfold` we require a commands file for each of the concentrations in question. Thus, one simply creates text files with a single line content:

```
UD NNNNNN e
```

where `e` is the binding free energy at this specific protein concentration as computed above. Note here, that we use `NNNNNN` as sequence motif that is bound by the protein to acknowledge the unspecific interaction between protein and RNA. Finally, `RNAfold` is executed to compute equilibrium base pairing and per-nucleotide protein binding probabilities .. code:

```
$ RNAfold -p --commands forties_50nM.txt forties_bundschuh.fa
```

and the produced probability dot plot can be inspected.



As you can see, the dot plot is augmented with an additional linear array of blue squares along each side that depicts the probability that the respective nucleotide is bound by the protein. Now, repeat the computations for different protein concentrations and compare the probabilities computed with the unstructured domain feature of the `ViennaRNA Package` with those in Fig. 3(a) of the publication.

Note, that `RNAfold` allows for an unlimited number of different proteins specified in the commands file. This easily allows one to model RNA-protein binding interaction within a relatively complex solution of different competing proteins.

Change other model settings

`RNAfold` also allows for many other changes of the implemented Nearest Neighbor model. For instance, you can explicitly prohibit (G, U) pairs, change the temperature that is used for evaluation of the free energy of particular loops, select a different dangling-end energy model or load a different set of free energy parameters, e.g. for DNA or parameters derived from computational optimizations.

See the man pages of `RNAfold` for a complete overview of all available options and command line switches.

Additional energy parameter collections are distributed together with the ViennaRNA Package as part of the contents of the misc/ directory, and are typically installed in prefix/share/ViennaRNA, where prefix is the path that was used as installation prefix, e.g. $HOME/Tutorial/Progs/VRP or /usr when installed globally using a package manager.

The Program RNApvmin

Table of Contents

- *Introduction*

Introduction

The program RNApvmin reads a RNA sequence from *stdin* and uses an iterative minimization process to calculate a perturbation vector that minimizes the discripancies between predicted pairing probabilites and observed pairing probabilities (deduced from given shape reactivities) [Washietl *et al.*, 2012]. The experimental SHAPE data has to be present in the file format described above. The application will write the calculated vector of perturbation energies to *stdout*, while the progress of the minimization process is written to *stderr*. The resulting perturbation vector can be interpreted directly and gives usefull insights into the discrepancies between thermodynamic prediction and experimentally determined pairing status. In addition the perturbation energies can be used to constrain folding with RNAfold:

```
$ RNApvmin rna.shape < rna.seq >vector.csv
$ RNAfold --shape=vector.csv --shapeMethod=W < rna.seq
```

The perturbation vector file uses the same file format as the SHAPE data file. Instead of SHAPE reactivities the raw perturbation energies will be storred in the last column. Since the energy model is only adjusted when necessary, the calculated perturbation energies may be used for the interpretation of the secondary structure prediction, since they indicate which positions require major energy model adjustments in order to yield a prediction result close to the experimental data. High perturbation energies for just a few nucleotides may indicate the occurrence of features, which are not explicitly handled by the energy model, such as posttranscriptional modifications and intermolecular interactions.

The Program RNAsubopt

Table of Contents

- *Introduction*
- *Suboptimal folding*
- *Sampling the Boltzmann Ensemble*

Introduction

By default, RNAsubopt calculates all suboptimal secondary structures within a given energy range above the MFE structure [Wuchty *et al.*, 1999].

Note

Be careful, the number of structures returned grows exponentially with both sequence length and energy range.

Suboptimal folding

- Generate all suboptimal structures within a certain energy range from the MFE specified by the -e option:

```
$ RNAsubopt -e 1 -s < test.seq
CUACGGCGCGGCGCCCUUGGCGA   -500    100
...........((((...)))).  -5.00
....((((...)))).......  -4.80
(((.((((...)))).))...  -4.20
...((.((.((...)).)).)).  -4.10
```

The text output shows an energy sorted list (option -s) of all secondary structures within 1~kcal/mol of the MFE structure. Our sequence actually has a ground state structure (-5.70) and three structures within 1~kcal/mol range.

MFE folding alone gives no indication that there are actually a number of plausible structures. Remember that RNAsubopt cannot automatically plot structures, therefore you can use the tool RNAplot. Note that you **can't** simply pipe the output of RNAsubopt to RNAplot using:

```
$ RNAsubopt < test.seq | RNAplot
```

You need to manually create a file for each structure you want to plot. Here, for example we created a new file named suboptstructure.txt:

```
> suboptstructure-4.20
CUACGGCGCGGCGCCCUUGGCGA
(((.((((...)))).))...
```

The fasta header is optional, but useful (without it the outputfile will be named rna.ps).

The next two lines contain the sequence and the suboptimal structure you want to plot; in this case we plotted the structure with the folding energy of -4.20.

Then plot it with

```
$ RNAplot < suboptstructure.txt
```

Note that the number of suboptimal structures grows exponentially with sequence length and therefore this approach is only tractable for sequences with less than 100 nt. To keep the number of suboptimal structures manageable the option --noLP can be used, forcing RNAsubopt to produce only structures without isolated base pairs. While RNAsubopt produces *all* structures within an energy range, mfold produces only a few, hopefully representative, structures. Try folding the sequence on the mfold server at http://mfold.rna.albany.edu/?q=mfold.

Sometimes you want to get information about unusual properties of the Boltzmann ensemble (the sum of all RNA structures possible) for which no specialized program exists. For example you want to know all fractions of a bacterial mRNA in the Boltzmann ensemble where the Shine-Dalgarno (SD) sequence is unpaired. If the SD sequence is concealed by secondary structure the translation efficiency is reduced.

In such cases you can resort to drawing a representative sample of structures from the Boltzmann ensemble by using the option -p. Now you can simply count how many structures in the sample possess the feature you are looking for. This number divided by the size of your sample gives you the desired fraction.

The following example calculates the fraction of structures in the ensemble that have bases 6 to 8 unpaired.

Sampling the Boltzmann Ensemble

RNAsubopt also implements a statisctical sampling algorithm to draw secondary structures from the ensemble according to their equilibrium probability [Ding and Lawrence, 2003]:

- Draw a sample of size 10,000 from the Boltzmann ensemble
- Calculate the desired property, e.g. by using a perl script:

```
$ RNAsubopt -p 10000 < test.seq > tt
$ perl -nle '$h++ if substr($_,5,3) eq "...";
  END {print $h/$.}' tt
  0.391960803919608
```

A far better way to calculate this property is to use `RNAfold -p` to get the ensemble free energy, which is related to the partition function via $F = -RT \ln(Q)$, for the unconstrained (F_u) and the constrained case (F_c), where the three bases are not allowed to form base pairs (use option `-C`), and evaluate $p_c = \exp((F_u - F_c)/RT)$ to get the desired probability.

So let's do the calculation using `RNAfold`:

```
$ RNAfold -p
Input string (upper or lower case); @ to quit
....,....1....,....2....,....3....,....4....,....5....,....6....,....7....,....8
CUACGGCGCGGCGCCCUUGGCGA
length = 23
CUACGGCGCGGCGCCCUUGGCGA
...........((((...)))).
 minimum free energy =  -5.00 kcal/mol
....{,{{...||||...)}}}.
 free energy of ensemble =  -5.72 kcal/mol
...................... {  0.00 d=4.66}
 frequency of mfe structure in ensemble 0.311796; ensemble diversity 6.36
```

Now we have calculated the free ensemble energy of the ensemble over all structures F_u, in the next step we have to calculate it for the structures using a constraint (F_c).

Following notation has to be used for defining the constraint:

- `|` : paired with another base
- `.` : no constraint at all
- `x` : base must not pair
- < : base i is paired with a base j<i
- > : base i is paired with a base j>i
- matching brackets `( )`: base i pairs base j

So our constraint should look like this:

```
.....xxx.............
```

Next call the application with following command and provide the sequence and constraint we just created:

```
$ RNAfold -p -C
```

The output should look like this:

```
length = 23
CUACGGCGCGGCGCCCUUGGCGA
...........((((...)))).
 minimum free energy =  -5.00 kcal/mol
...........((((...)))).
 free energy of ensemble =  -5.14 kcal/mol
...........((((...)))). { -5.00 d=0.42}
 frequency of mfe structure in ensemble 0.792925; ensemble diversity 0.79
```

Afterwards evaluate the desired probability according to the formula given before e.g. with a simple `perl` script:

```
$ perl -e 'print exp(-(5.72-5.14)/(0.00198*310.15))."\n"'
```

You can see that there is a slight difference between the `RNAsubopt` run with 10,000 samples and the `RNAfold` run including all structures.

2.4.2 Consensus Structure Prediction

Consensus structures can be predicted by a modified version of the secondary structure prediction algorithm that takes as input a set of aligned sequences instead of a single sequence.

Sequence co-variations are a direct consequence of RNA base pairing rules and can be deduced to alignments. RNA helices normally contain only 6 out of the 16 possible combinations: the Watson-Crick pairs `GC`, `CG`, `AU`, `UA`, and the somewhat weaker wobble pairs `GU` and `UG`. Mutations in helical regions therefore have to be correlated. In particular we often find *compensatory mutations* where a mutation on one side of the helix is compensated by a second mutation on the other side, e.g. a `CG` pair changes into a `UA` pair. Mutations where only one pairing partner changes (such as `CG` to `UG` are termed *consistent mutations*.

The energy function consists of the mean energy averaged over the sequences, plus a covariance term that favors pairs with consistent and compensatory mutations and penalizes pairs that cannot be formed by all structures. For details see Hofacker *et al.* [2002] and Bernhart *et al.* [2008].

The Program `RNAalifold`

Table of Contents

- *Introduction*
- *Consensus Structure from related Sequences*
- *`RNAalifold` Output Files*
- *Structure predictions for the individual sequences*

Introduction

`RNAalifold` generalizes the folding algorithm for multiple sequence alignments (MSA), treating the entire alignment as a single *generalized sequence*. To assign an energy to a structure on such a generalized sequence, the energy is simply averaged over all sequences in the alignment. This average energy is augmented by a covariance term, that assigns a bonus or penalty to every possible base pair (i, j) based on the sequence variation in columns i and j of the alignment.

Compensatory mutations are a strong indication of structural conservation, while consistent mutations provide a weaker signal. The covariance term used by `RNAalifold` therefore assigns a bonus of 1 kcal/mol to each consistent and 2 kcal/mol for each compensatory mutation. Sequences that cannot form a standard base pair incur a penalty of -1 kcal/mol. Thus, for every possible consensus pair between two columns i and j of the alignment a covariance score C_{ij} is computed by counting the fraction of sequence pairs exhibiting consistent and compensatory mutations, as well as the fraction of sequences that are inconsistent with the pair. The weight of the covariance term relative to the normal energy function, as well as the penalty for inconsistent mutations can be changed via command line parameters.

Apart from the covariance term, the folding algorithm in `RNAalifold` is essentially the same as for single sequence folding. In particular, folding an alignment containing just one sequence will give the same result as single sequence folding using `RNAfold`. For N sequences of length n the required CPU time scales as $\mathcal{O}(N \cdot n^2 + n^3)$ while memory requirements grow as the square of the sequence length. Thus `RNAalifold` is in general faster than folding each sequence individually. The main advantage, however, is that the accuracy of consensus structure predictions is generally much higher than for single sequence folding, where typically only between 40% and 70% of the base pairs are predicted correctly.

Apart from prediction of MFE structures RNAalifold also implements an algorithm to compute the partition function over all possible (consensus) structures and the thermodynamic equilibrium probability for each possible pair. These base pairing probabilities are useful to see structural alternatives, and to distinguish well defined regions, where the predicted structure is most likely correct, from ambiguous regions.

As a first example we'll produce a consensus structure prediction for the following four tRNA sequences.

```
$ cat > four.seq
>M10740 Yeast-PHE
GCGGAUUUAGCUCAGUUGGGAGAGCGCCAGACUGAAGAUUUGGAGGUCCUGUGUUCGAUCCACAGAAUUCGCA
>K00349 Drosophila-PHE
GCCGAAAUAGCUCAGUUGGGAGAGCGUUAGACUGAAGAUCUAAAGGUCCCCGGUUCAAUCCCGGGUUUCGGCA
>K00283 Halobacterium volcanii Lys-tRNA-1
GGGCCGGUAGCUCAUUUAGGCAGAGCGUCUGACUCUUAAUCAGACGGUCGCGUGUUCGAAUCGCGUCCGGCCCA
>AF346993
CAGAGUGUAGCUUAACACAAAGCACCCAACUUACACUUAGGAGAUUUCAACUUAACUUGACCGCUCUGA
```

RNAalifold uses aligned sequences as input. Thus, our first step will be to align the sequences. We use clustalw2 in this example, since it's one of the most widely used alignment programs and has been shown to work well on structural RNAs. Other alignment programs can be used (including programs that attempt to do structural alignment of RNAs), but for this example the resulting multiple sequence alignment should be in Clustal format. Get clustalw2 and install it as you have done it with the other packages: http://www.clustal.org/clustal2.

Consensus Structure from related Sequences

- Prepare a sequence file (use file four.seq and copy it to your working directory)
- Align the sequences
- Compute the consensus structure from the alignment
- Inspect the output files alifold.out, alirna.ps, alidot.ps
- For comparison fold the sequences individually using RNAfold

```
$ clustalw2 four.seq > four.out
```

Clustalw2 creates two more output files, four.aln and four.dnd. For RNAalifold you need the .aln file.

```
$ RNAalifold -p four.aln
$ RNAfold -p < four.seq
```

RNAalifold output:

```
__GCCGAUGUAGCUCAGUUGGG_AGAGCGCCAGACUGAAAAUCAGAAGGUCCCGUGUUCAAUCCACGGAUCCGGCA__
..(((((((..((((.........)))).(((((.......)))))).....(((((.......)))))))))))...
 minimum free energy = -15.12 kcal/mol (-13.70 +  -1.43)
..((((((..((((.........)))).(((((.......)))))).....(((((.......)))))}))))))...
 free energy of ensemble = -15.75 kcal/mol
 frequency of mfe structure in ensemble 0.361603
..(((((((..((((.........)))).(((((.......)))))).....(((((.......))))))))))... -15.20
↪{-13.70 +  -1.50}
```

RNAfold output:

```
>M10740 Yeast-PHE
GCGGAUUUAGCUCAGUUGGGAGAGCGCCAGACUGAAGAUUUGGAGGUCCUGUGUUCGAUCCACAGAAUUCGCA
(((((((.........((((.((((((..((((...........))))..))))))..))))..)))))))). (-21.60)
(((((((...,,.{,((((.((((((..((((...........))))..))))))..)))))),)))))). [-23.20]
(((((((..........(((.((((((..((((...........))))..))))))..)))...)))))))). {-20.00 d=9.
↪63}
```

(continues on next page)

(continued from previous page)

```
 frequency of mfe structure in ensemble 0.0744065; ensemble diversity 15.35
>K00349 Drosophila-PHE
[...]
```

The output contains a consensus sequence and the consensus structure in dot-bracket notation. The consensus structure has an energy of -15.12 kcal/mol, which in turn consists of the average free energy of the structure -13.70 kcal/mol and the covariance term -1.43 kcal/mol. The strongly negative covariance term shows that there must be a fair number of consistent and compensatory mutations, but in contrast to the average free energy it's not meaningful in the biophysical sense.

Compare the predicted consensus structure with the structures predicted for the individual sequences using `RNAfold`. How often is the correct `clover-leaf` shape predicted?

For better visualization, a structure annotated alignment or color annotated structure drawing can be generated by using the `--aln` and `--color` options of `RNAalifold`.

```
$ RNAalifold --color --aln four.aln
$ gv aln.ps &
$ gv alirna.ps &
```

`RNAalifold` Output Files

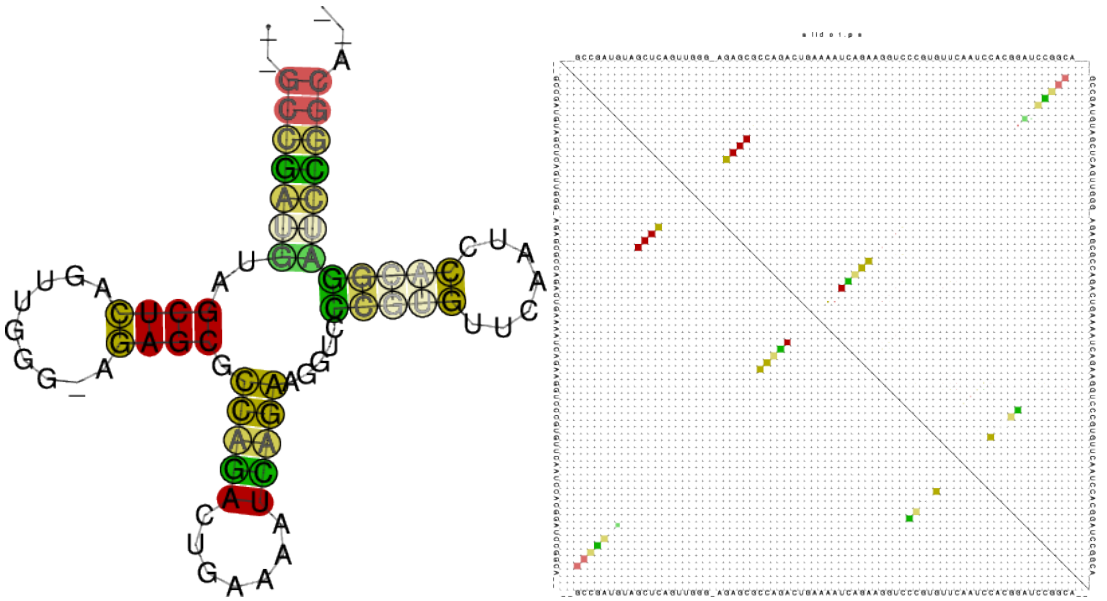
Content of the `alifold.out` file:

```
4 sequence; length of alignment 78
alifold output
    6    72  0  99.8%   0.007 GC:2    GU:1    AU:1
   33    43  0  98.9%   0.033 GC:2    GU:1    AU:1
   31    45  0  99.0%   0.030 CG:3    UA:1
   15    25  0  98.9%   0.045 CG:3    UA:1
    5    73  1  99.7%   0.008 CG:2    GC:1
   13    27  0  99.1%   0.042 CG:4
   14    26  0  99.1%   0.042 UA:4
    4    74  1  99.5%   0.015 CG:3
[...]
```

The last output file produced by `RNAalifold -p`, named `alifold.out`, is a plain text file with detailed information on all plausible base pairs sorted by the likelihood of the pair. In the example above we see that the pair $(6, 72)$ has no inconsistent sequences, is predicted almost with probability 1, and occurs as a `GC` pair in two sequences, a `GU` pair in one, and a `AU` pair in another.

`RNAalifold` automatically produces a drawing of the consensus structure in Postscript format and writes it to the file `alirna.ps`. In the structure graph consistent and compensatory mutations are marked by a circle around the variable base(s), i.e. pairs where one pairing partner is encircled exhibit consistent mutations, whereas pairs supported by compensatory mutations have both bases marked. Pairs that cannot be formed by some of the sequences are shown gray instead of black.

The structure layout and dotplot files `alirna.ps` and `alidot.ps` should look as follows:



In the example given, many pairs show such inconsistencies. This is because one of the sequences (AF346993) is not aligned well by `clustalw`.

> **Note**
>
> Subsequent calls to `RNAalifold` will overwrite any existing output `alirna.ps` (`alidot.ps`, `alifold.out`) files in the current directory. Be sure to rename any files you want to keep.

Structure predictions for the individual sequences

The consensus structure computed by `RNAalifold` will contain only pairs that can be formed by most of the sequences. The structures of the individual sequences will typically have additional base pairs that are not part of the consensus structure. Moreover, ncRNA may exhibit a highly conserved core structure while other regions are more variable. It may therefore be desirable to produce structure predictions for one particular sequence, while still using covariance information from other sequences.

This can be accomplished by first computing the consensus structure for all sequences using `RNAalifold`, then folding individual sequences using `RNAfold -C` with the consensus structure as a constraint. In constraint folding mode `RNAfold -C` allows only base pairs to form which are compatible with the constraint structure. This resulting structure typically contains most of the constraint (the consensus structure) plus some additional pairs that are specific for this sequence.

The `refold.pl` script removes gaps and maps the consensus structure to each individual sequence.

```
$ RNAalifold  RNaseP.aln > RNaseP.alifold
$ gv alirna.ps
$ refold.pl RNaseP.aln RNaseP.alifold | head -3 > RNaseP.cfold
$ RNAfold -C --noLP < RNaseP.cfold > RNaseP.refold
$ gv E-coli_ss.ps
```

If you compare the refolded structure (`E-coli_ss.ps`) with the structure you get by simply folding the E.coli sequence in the `RNaseP.seq` file (`RNAfold --noLP`) you find a clear rearrangement.

In cases where constrained folding results in a structure that is very different from the consensus, or if the energy from constrained folding is much worse than from unconstrained folding, this may indicate that the sequence in question does not really share a common structure with the rest of the alignment or is misaligned. One should then either remove or re-align that sequence and recompute the consensus structure.

Note

Note that since RNase P forms sizable pseudo-knots, a perfect prediction is impossible in this case.

2.4.3 RNA-RNA interaction

A common problem is the prediction of binding sites between two RNAs, as in the case of miRNA-mRNA interactions. Following tools of the `ViennaRNA Package` can be used to calculate base pairing probabilities.

The Program `RNAcofold`

Table of Contents

- *Introduction*
- *Two Sequences one Structure*
- *Concentration Dependency*
- *Concentration Dependency Plot*

Introduction

`RNAcofold` works much like `RNAfold` but uses two RNA sequences as input which are then allowed to form a dimer structure. In the input the two RNA sequences should be concatenated using the `&` character as separator. As in `RNAfold` the `-p` option can be used to compute partition function and base pairing probabilities.

Since dimer formation is concentration dependent, `RNAcofold` can be used to compute equilibrium concentrations for all five monomer and (homo/hetero)-dimer species, given input concentrations for the monomers (see the `man` page for details).

Two Sequences one Structure

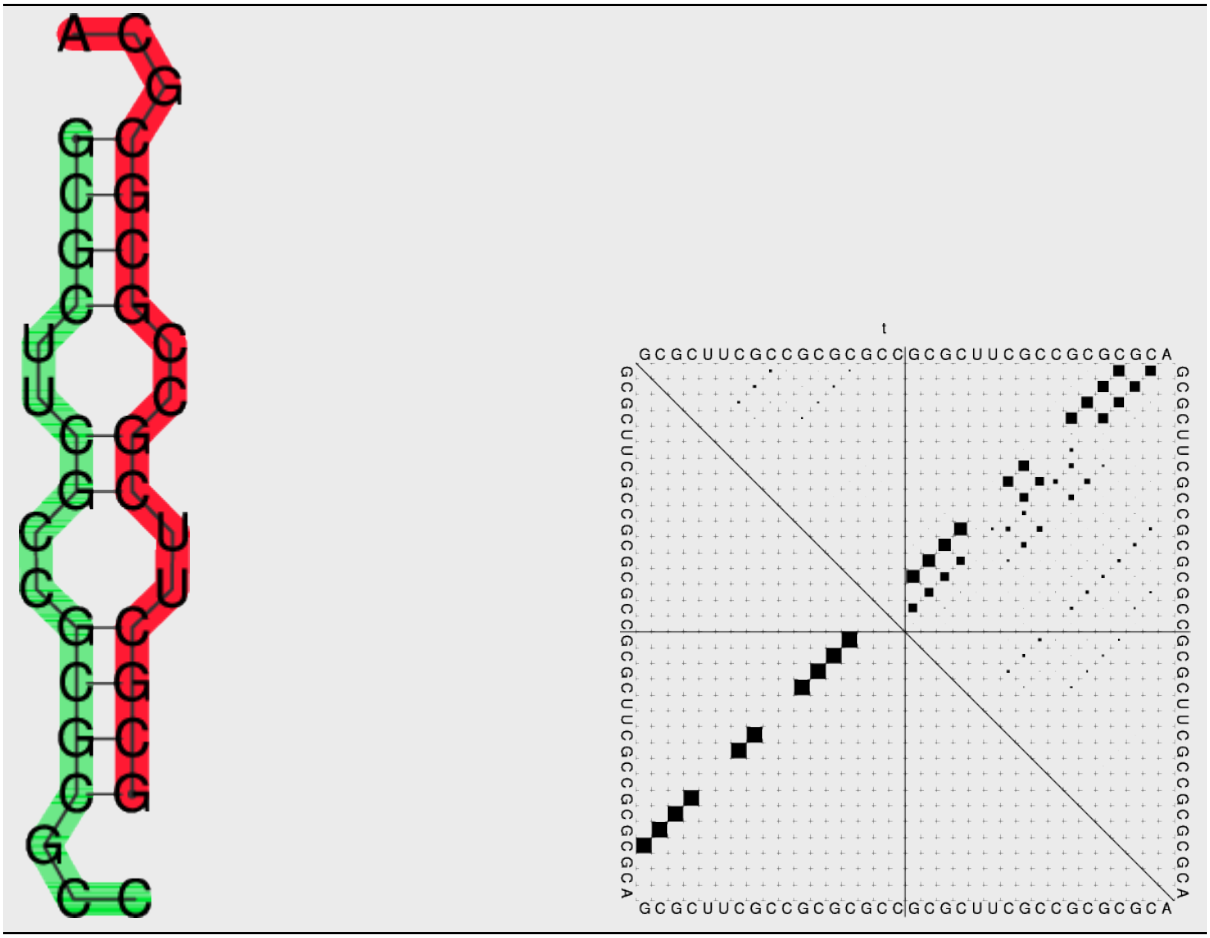
- Prepare a sequence file (`t.seq`) for input that looks like this:

```
>t
GCGCUUCGCCGCGCGCC&GCGCUUCGCCGCGCGCA
```

- Compute the `MFE` and the ensemble properties
- Look at the generated PostScript files `t_ss.ps` and `t_dp.ps`

```
$ RNAcofold -p < t.seq
>t
GCGCUUCGCCGCGCGCC&GCGCUUCGCCGCGCGCA
((((..((..((((...&))))..))..))))... (-17.70)
((((..{(,.((((,,.&))))..}),.))))),,. [-18.26]
 frequency of mfe structure in ensemble 0.401754 , delta G binding= -3.95
```

Table 1: Secondary Structure and Dot Plot



In the dot plot a cross marks the chain break between the two concatenated sequences.

Concentration Dependency

Cofolding is an intermolecular process, therefore whether duplex formation will actually occur is concentration dependent. Trivially, if one of the molecules is not present, no dimers are going to be formed. The partition functions of the molecules give us the equilibrium constants:

$$K_{AB} = \frac{[AB]}{[A][B]} = \frac{Z_{AB}}{Z_A Z_B}$$

with these and mass conservation, the equilibrium concentration of homodimers, heterodimers and monomers can be computed in dependence of the start concentrations of the two molecules.

This is most easily done by creating a file with the initial concentrations of molecules A and B in two columns:

```
[a_1]([mol/l])  [b_1]([mol/l])
[a_2]([mol/l])  [b_2]([mol/l])

[...]

[a_n]([mol/l])  & [b_n]([mol/l])
```

- Prepare a concentration file for input with this little `perl` script:

```
$ perl -e '$c=1e-07; do {print "$c\t$c\n"; $c*=1.71;} while $c<0.2' > concfile
```

This script creates a file displaying values from 1e-07 to just below 0.2, with 1.71-fold steps in between. For convenience, concentration of molecule A is the same as concentration of molecule B in each row. This will facilitate visualization of the results.

- Compute the `MFE`, the ensemble properties and the concentration dependency of hybridization:

```
$ RNAcofold -f concfile < t.seq > cofold.out
```

- Look at the generated output with:

```
$ less cofold.out
```

which should be similar to:

```
[...]
Free Energies:
AB               AA               BB               A                B
-18.261023       -17.562553       -18.274376       -7.017902        -7.290237
Initial concentrations           relative Equilibrium concentrations
A                 B                AB               AA               BB
↪A                B
1e-07             1e-07            0.00003          0.00002          0.00002
↪0.49994          0.49993
[...]
```

The five different free energies were printed out first, followed by a list of all the equilibrium concentrations, where the first two columns denote the initial (absolute) concentrations of molecules A and B, respectively. The next five columns denote the equilibrium concentrations of dimers and monomers, relative to the total particle number.

Note

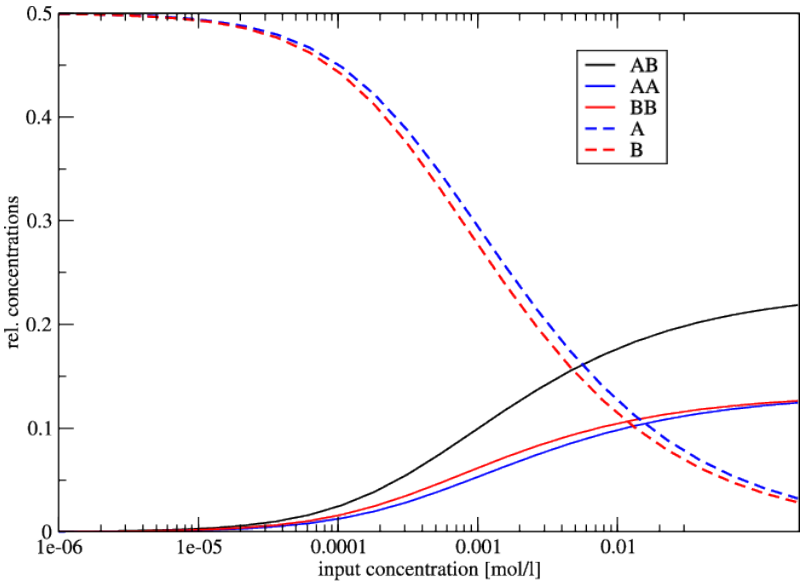
The concentrations don't add up to one, except in the case where no dimers are built – if you want to know the fraction of particles in a dimer, you have to take the relative dimer concentrations times 2.

Since relative concentrations of species depend on two independent values - initial concentration of A as well as initial concentration of B - it is not trivial to visualize the results. For this reason we used the same concentration for A and for B. Another possibility would be to keep the initial concentration of one molecule constant. As an example we show the following plot of `t.seq`.

Now we use some commandline tools to render our plot. We use `tail -n +11` to show all lines starting with line 11 (1-10 are cut) and pipe it into an `awk` command, which prints every column but the first from our input. This is then piped to `xmgrace`. With `-log x -nxy` - we tell it to plot the x axis in logarithmic scale and to read data file in X Y1 Y2 ... format.

```
$ tail -n +11 cofold.out | awk '{print $2, $3, $4, $5, $6, $7}' | xmgrace -log x -nxy␣
↪-
```

Concentration Dependency Plot



Since the two sequences are almost identical, the monomer and homo-dimer concentrations behave very similarly. In this example, at a concentration of about 1 mmol 50% of the molecule is still in monomer form.

The Program `RNAduplex`

Table of Contents

- *Introduction*
- *Binding site prediction with `RNAduplex`*
- *Binding site prediction with `RNAup`*

Introduction

If the sequences are very long (many kb) `RNAcofold` is too slow to be useful. The `RNAduplex` program is a fast alternative, that works by predicting *only* intermolecular base pairs. It's almost as fast as simple sequence alignment, but much more accurate than a `BLAST` search.

The example below searches the 3' UTR of an mRNA for a miRNA binding site.

Binding site prediction with `RNAduplex`

The file `duplex.seq` contains the 3'UTR of `NM_024615` and the microRNA mir-145.

```
$ RNAduplex < duplex.seq
>NM_024615
>hsa-miR-145
.(((((.(((...((((((((((.&))))))))))))))))))).  34,57  :   1,19  (-21.90)
```

Most favorable binding has an interaction energy of -21.90 kcal/mol and pairs up on positions 34-57 of the UTR with positions 1-22 of the miRNA.

`RNAduplex` can also produce alternative binding sites, e.g. running `RNAduplex -e 10` would list all binding sites within 10 kcal/mol of the best one.

Since `RNAduplex` forms only intermolecular pairs, it neglects the competition between intramolecular folding and hybridization. Thus, it is recommended to use `RNAduplex` as a pre-filter and analyse good `RNAduplex` hits

additionally with RNAcofold or RNAup. Using the example above, running RNAup will yield:

Binding site prediction with RNAup

```
$ RNAup -b < duplex.seq
>NM_024615
>hsa-miR-145
(((((((&)))))))  50,56  :   1,7   (-8.41 = -9.50 + 0.69 + 0.40)
GCUGGAU&GUCCAGU
RNAup output in file: hsa-miR-145_NM_024615_w25_u1.out
```

The free energy of the duplex is -9.50 kcal/mol and shows a discrepancy to the structure and energy value computed by RNAduplex (differences may arise from the fact that RNAup computes partition functions rather than optimal structures).

However, the total free energy of binding is less favorable (-8.41 kcal/mol), since it includes the energetic penalty for opening the binding site on the mRNA (0.69 kcal/mol) and miRNA (0.40 kcal/mol). The -b option includes the probability of unpaired regions in both RNAs.

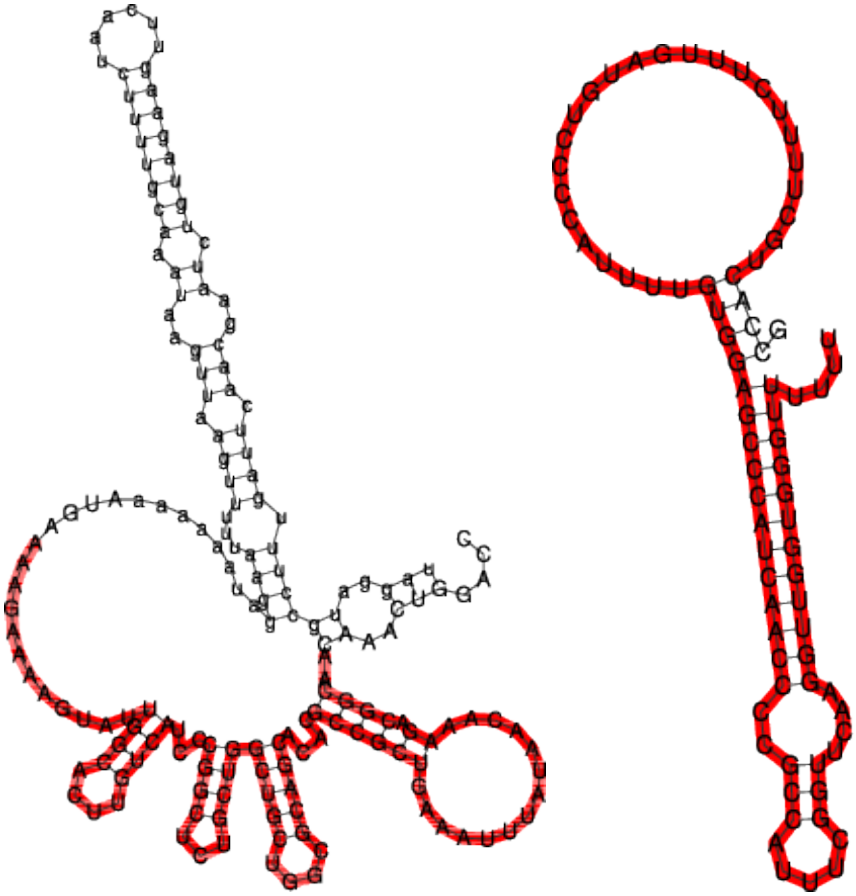
You can also run RNAcofold on the example to see the complete structure after hybridization (neither RNAduplex nor RNAup produce structure drawings). Note however, that the input format for RNAcofold is different. An input file suitable for RNAcofold has to be created from the duplex.seq file first (use any text editor).

As a more difficult example, let's look at the interaction of the bacterial smallRNA RybB and its target mRNA ompN. First we'll try predicting the binding site using RNAduplex:

```
$ RNAduplex < RybB.seq
>RybB
>ompN
.((((..((((((.(((....(((((((((..(((((.((..((.((....((((..((((((((((..((((((&
.))))))..)))))))).))))).....))))....)).)).)).))).)).)..))))........))))..))).)))))).)))).
 5,79  :  80,164 (-34.60)
```

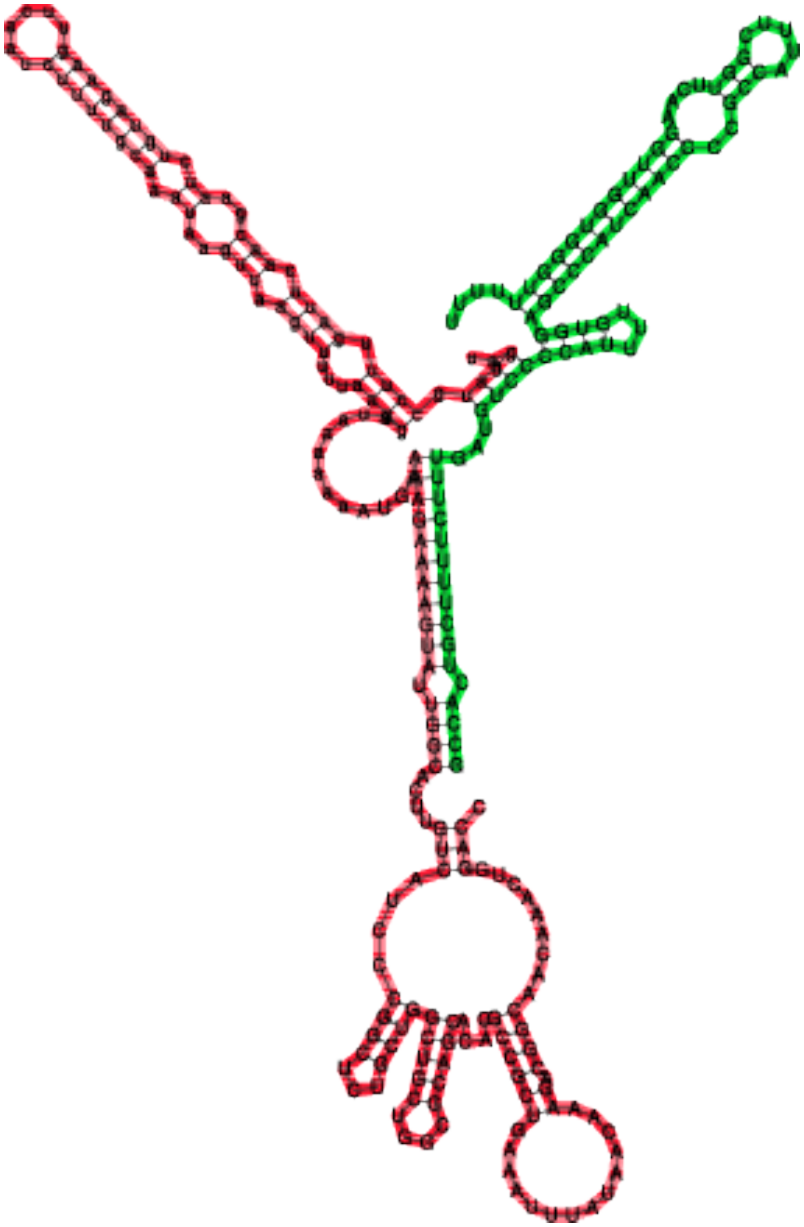
Note, that the predicted structure spans almost the full length of the RybB small RNA. Compare the predicted interaction to the structures predicted for RybB and ompN alone, and ask yourself whether the predicted interaction is indeed plausible.

Below the structure of ompN on the left and RybB on the right side. The respective binding regions predicted by RNAduplex are marked in red:



```
GCCAC-----TGCTTTTCTTTGATGTCCCCATTTT-GTGGA-------GC-CCATCAACCCCGCCATTTCGGTT---CAAG-
↪GTTGGTGGGTTTTTT
 |||      ||||  |||||| |||    |||||  ||||        || ||| || ||  ||    ||||     |||| || ␣
↪|||  |||||| -40.30
AGGTCAAACAACGGC-AGAAACAATATT--TAAAGTCGCCGCACACGACGCGGTCGTCGGT-
↪CGTCTCGGCCCTACTGTTCACGGTTATGAAAAGAAACC-3'
```

Compare the `RNAduplex` prediction with the interaction predicted by `RNAcofold`, `RNAup` and the handcrafted prediction you see above.



2.4.4 Plotting Structures

The Program `RNAplot`

Table of Contents

- *Introduction*
- *PostScript macros*

Introduction

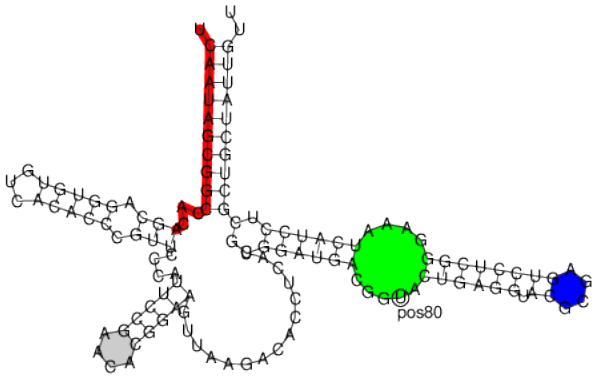
You can manually add additional annotation to structure drawings using the `RNAplot` program (for information see its `man` page). Here's a somewhat complicated example:

```
$ RNAfold 5S.seq > 5S.fold
$ RNAplot --pre "76 107 82 102 GREEN BFmark 44 49 0.8 0.8 0.8 Fomark \
```

(continues on next page)

(continued from previous page)

```
  1 15 8 RED omark 80 cmark 80 -0.23 -1.2 (pos80) Label 90 95 BLUE Fomark" < 5S.fold
$ gv 5S_ss.ps
```



PostScript macros

`RNAplot` is a very useful tool to color structure layout plots. The `--pre` tag adds PostScript code required to color distinct regions of your molecule. There are some predefined statements with different options for annotations listed below:

Command	Description
`i cmark`	draws circle around base i
`i j c gmark`	draw basepair i,j with c counter examples in grey
`i j lw rgb omark`	stroke segment i…j with linewidth lw and color (rgb)
`i j rgb Fomark`	fill segment i…j with color (rgb)
`i j k l rgb BFmark`	fill block between pairs i,j and k,l with color (rgb)
`i dx dy (text) Label`	adds a textlabel with an offset dx and dy relative to base i

Predefined color options are `BLACK, RED, GREEN, BLUE, WHITE` but you can also replace the value to some standard RGB code (e.g. 0 5 8 for lightblue).

To simply add the annotation macros to the `PostScript` file without any actual annotation you can use the following program call

```
$ RNAplot --pre "" < 5S.fold
```

If you now open the structure layout file `5S_ss.ps` with a text editor you'll see the additional macros for `cmark`, `omark`, etc. along with some show synopsis on how to use them. Actual annotations can then be added between the lines:

```
% Start Annotations
```

and:

```
% End Annotations
```

Here, you simply need to add the same string of commands you would provide through the `--pre` option of `RNAplot`.

2.4.5 RNA Design

The Program `RNAinverse`

Table of Contents

- *Introduction*
- *Sequence Design*
- *Other RNA design tools*

Introduction

`RNAinverse` searches for sequences folding into a predefined structure, thereby inverting the folding algorithm. Input consists of the target structures (in dot-bracket notation) and a starting sequence, which is optional.

Lower case characters in the start sequence indicate fixed positions, i.e. they can be used to add sequence constraints. `N`'s in the starting sequence will be replaced by a random nucleotide. For each search the best sequence found and its Hamming distance to the start sequence are printed to *stdout*. If the the search was unsuccessful a structure distance to the target is appended.

By default the program stops as soon as it finds a sequence that has the target as MFE structure. The option `-Fp` switches `RNAinverse` to the partition function mode where the probability of the target structure $\exp(-E(S)/RT)/Z$ is maximized. This tends to produce sequences with a more well-defined structure.

This probability is written in dot-brackets after the found sequence and Hamming distance. With the option `-R` you can specify how often the search should be repeated.

Sequence Design

- Prepare an input file `inv.in` containing the target structure and sequence constraints:

```
(((.(((....))).)))
NNNgNNNNNNNNNaNNN
```

- Design sequences using `RNAinverse`:

```
$ RNAinverse < inv.in
  GGUgUUGGAUCCGAaACC    5
```

  or design even more sequences with:

```
$ RNAinverse -R5 -Fp < inv.in
  GGUgUGAACCCUCGaACC    5
  GGCgCCCUUUUGGGaGCC   12  (0.967418)
  CUCgAUCUCACGAUaGGG    6
  GGCgCCCGAAAGGGaGCC   13  (0.967548)
  GUUgAGCCCAUGCUaAGC    6
  GGCgCCCUUAUGGGaGCC   10  (0.967418)
  CGGgUGUUGUGACAaCCG    5
  GCGgGUCGAAAGGCaCGC   12  (0.925482)
  GCCgUAUCCGGGUGaGGC    6
  GGCgCCCUUUUGGGaGCC   13  (0.967418)
```

The output consists of the calculated sequence and the number of mutations needed to get the MFE-structure from the start sequence (start sequence not shown). Additionaly, with the partition function folding (`-Fp`) set, the second output is another refinement so that the ensemble preferes the MFE and folds into your given structure with a distinct probability, shown in brackets.

Other RNA design tools

Another useful program for inverse folding is `RNA designer`, see http://www.rnasoft.ca. `RNA Designer` takes a secondary structure description as input and returns an RNA strand that is likely to fold in the given secondary structure.

The `sequence design application` of the `ViennaRNA Design Webservices`, see http://nibiru.tbi.univie.ac.at/rnadesign/index.html, uses a different approach, allowing for more than one secondary structure as input. For more detail read the online Documentation and the next section of this tutorial.

The Program `switch.pl`

Table of Contents

- *Introduction*
- *Designing a Switch*

Introduction

The `switch.pl` script can be used to design bi-stable structures, i.e. structures with two almost equally good foldings. For two given structures there are always a lot of sequences compatible with both structures. If both structures are reasonably stable you can find sequences where both target structures have almost equal energy and all other structures have much higher energies. Combined with `RNAsubopt`, `barriers` and `treekin`, this is a very useful tool for designing RNA switches.

The input requires two structures in dot-bracket notation and additionally you can add a sequence. It is also possible to calculate the switching function at two different temperatures with option `-T` and `-T2`.

Designing a Switch

Now we try to create an RNA switch using `switch.pl` [Flamm *et al.*, 2001]. First we create our inputfile, then invoke the program using ten optimization runs (`-n 10`) and do not allow *lonely pairs*. Write it out to `switch.out`

```
$ cat > switch.in
(((((((.....)))))))....(((((((.......)))))))
((((((((((((((((.......))))))))))))))).....

$ switch.pl -n 10 --noLP < switch.in > switch.out
```

`switch.out` should look similar like this, the first block represents our bi-stable structures in random order, the second block shows the resulting sequences ordered by their score.

```
$ cat switch.out
GGGUGGACGUUUCGGUCCAUCCUUACGGACUGGGGCGUUUACCUAGUCC   0.9656
CAUUUGGCUUGUGUGUCGAAUGGCCCCGGUACGUAGGCUAAAUGUACCG   1.2319
GGGGGGUGCGUUCACACCCCUCAUUUGGUGUGGAUGUGCUUUCUACACU   1.1554
[...]
```

the resulting sequences are:

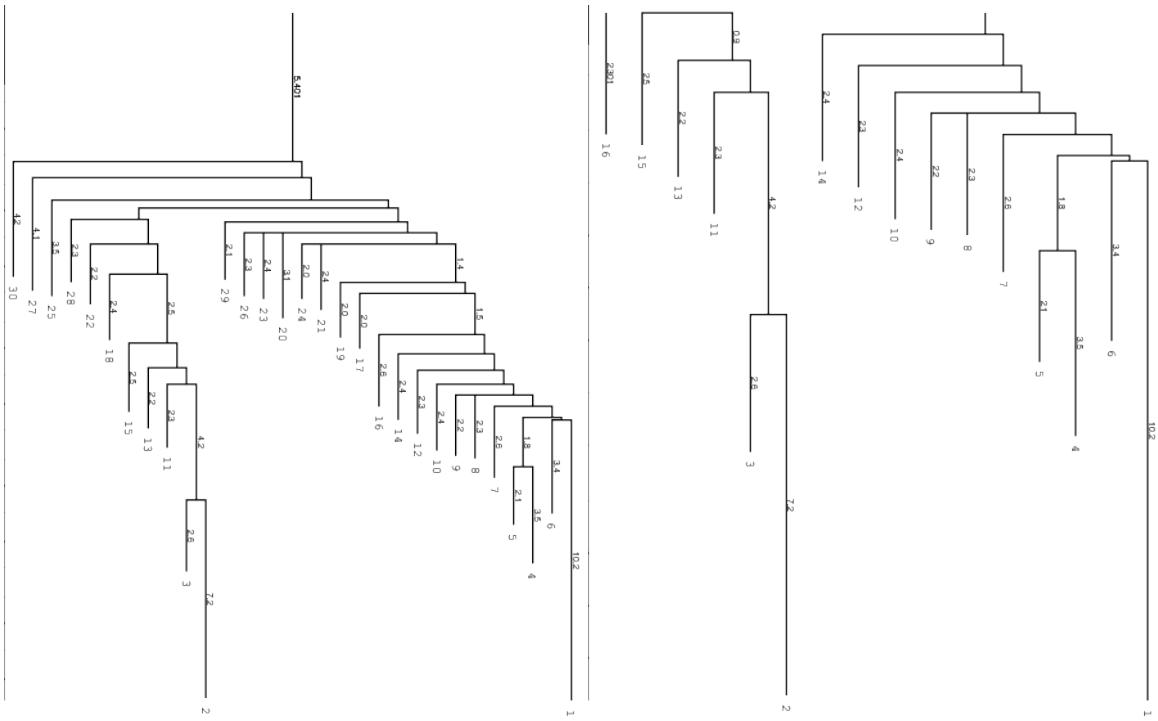
```
CAUUUGGCUUGUGUGUCGAAUGGCCCCGGUACGUAGGCUAAAUGUACCG   1.2319
GGGGGGUGCGUUCACACCCCUCAUUUGGUGUGGAUGUGCUUUCUACACU   1.1554
CGGGUUGUAACUGGAUAGCCUGGAAACUGUUUGGUUGUAAUCCGAACAG   1.0956
[...]
```

Given all 10 suggestions in our `switch.out`, we select the one with the best score with some command line tools to use it as an `RNAsubopt` input file and build up the barriers tree.

```
$ tail -10 switch.out | awk '{print($1)}'  | head -n 1 > subopt.in
$ RNAsubopt --noLP -s -e 25 < subopt.in > subopt.out
$ barriers -G RNA-noLP --bsize --rates --minh 2 --max 30 < subopt.out > barriers.out
```

`tail -10` cuts the last 10 lines from the `switch.out` file and pipes them into an `awk` script. The function `print($1)` echoes only the first column and this is piped into the `head` program where the first line, which equals the best scored sequence, is taken and written into `subopt.in`. Then `RNAsubopt` is called to process our sequence and write the output to another file which is the input for the barriers calculation.

Below you find an example of the barrier tree calculation above done with the right settings (connected root) on the left side and the wrong `RNAsubobt -e` value on the right. Keep in mind that `switch.pl` performs a stochastic search and the output sequences are different every time because there are a lot of sequences which fit the structure and switch calculates a new one everytime. Simply try to make sure.



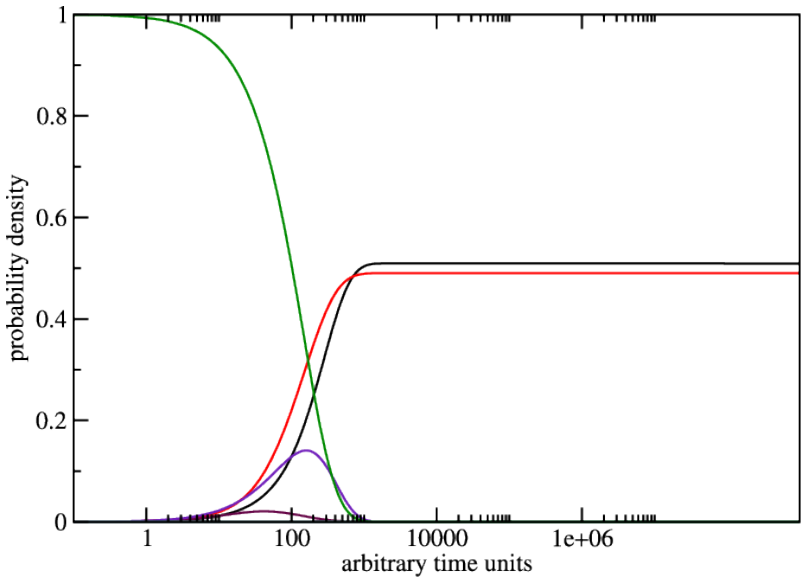
left: Barriers tree as it should look like, all branches connected to the main root right: disconnected tree due to a too low energy range (`-e`) parameter set in `RNAsubopt`.

Be careful to set the range `-e` high enough, otherwise we get a problem when calculation the kinetics using `treekin`. Every branch should be somehow connected to the main root of the tree. Try `-e 20` and `-e 30` to see the difference in the trees and choose the optimal value. By using `--max 30` we shorten our tree to focus only on the lowest minima.

We then select a branch preferably outside of the two main branches, here branch 30 (may differ from your own calculation). Look at the barrier tree to find the best branch to start and replace `30` by the branch you would choose. Now use `treekin` to plot concentration kinetics and think about the graph you just created.

```
$ treekin -m I --p0 30=1  < barriers.out > treekin.out
$ xmgrace -log x -nxy treekin.out
```

The graph could look like the one below, remember everytime you use `switch.pl` it can give you different sequences so the output varies too. Here the one from the example.



2.4.6 RNA folding kinetics

RNA folding kinetics describes the dynamical process of how a RNA molecule approaches to its unique folded biological active conformation (often referred to as the native state) starting from an initial ensemble of disordered conformations e.g. the unfolded open chain. The key for resolving the dynamical behavior of a folding RNA chain lies in the understanding of the ways in which the molecule explores its astronomically large free energy landscape, a rugged and complex hyper-surface established by all the feasible base pairing patterns a RNA sequence can form. The challenge is to understand how the interplay of formation and break up of base pairing interactions along the RNA chain can lead to an efficient search in the energy landscape which reaches the native state of the molecule on a biologically meaningful time scale.

The Program `RNA2Dfold`

`RNA2Dfold` is a tool for computing the MFE structure, partition function and representative sample structures of κ, λ neighborhoods and projects an high dimensional energy landscape of RNA into two dimensions [Lorenz *et al.*, 2009]. Therefore a sequence and two user-defined reference structures are expected by the program. For each of the resulting distance class, the MFE representative, the Boltzmann probabilities and the Gibbs free energy is computed. Additionally, representative suboptimal secondary structures from each partition can be calculated.

```
$ RNA2Dfold -p < 2dfold.inp > 2dfold.out
```

The outputfile `2dfold.out` should look like below, check it out, e.g. using `less`:

```
CGUCAGCUGGGAUGCCAGCCUGCCCCGAAAGGGGCUUGGCGUUUUGGUUGUUGAUUCAACGAUCAC
((((((((((....))))))..(((((....))))).))))).((((((((((...)))))))))). (-30.40)
((((((((((....))))))..(((((....))))).))))).((((((((((...)))))))))). (-30.40) <ref 1>
.............................................................. (  0.00) <ref 2>
free energy of ensemble = -31.15 kcal/mol
k       l       P(neighborhood) P(MFE in neighborhood)  P(MFE in ensemble)      MFE␣
↪  E_gibbs MFE-structure
0       24      0.29435909      1.00000000      0.29435892      -30.40  -30.40␣
↪((((((((((....))))))..(((((....))))).))))).((((((((((...)))))))))).
1       23      0.17076902      0.47069889      0.08038083      -29.60  -30.06␣
↪((((((((((....))))))..(((((....))))).))))).....(((((((...)))))))..
2       22      0.03575448      0.37731068      0.01349056      -28.50  -29.10  ((((.
↪(((((....))))))..(((((....))))))..))))....(((((((...)))))))..
2       24      0.00531223      0.42621709      0.00226416      -27.40  -27.93␣
↪((((((((((....))))...(((((....)))))))))))...((((((((((...)))))))))).
3       21      0.00398349      0.29701636      0.00118316      -27.00  -27.75  .(((.
```

(continues on next page)

(continued from previous page)

```
↪(((((....)))))..(((((....)))))..)))......(((((((...)))))))..
3       23      0.00233909      0.26432372      0.00061828      -26.60  -27.42 ␣
↪((((((((((....))))...(((((....)))))))))).....(((((((...)))))))..
[...]
```

For visualizing the output the `ViennaRNA Package` includes two scripts `2Dlandscape_pf.gri`, `2Dlandscape_mfe.gri` located in `/usr/share/ViennaRNA/`. `gri` (a language for scientific graphics programing) is needed to create a colored postscript plot. We use the partition function script to show the free energies of the distance classes (graph below, left):

```
$ gri ../Progs/VRP/share/ViennaRNA/2Dlandscape_pf.gri 2dfold.out
```

Compare the output file with the colored plot and determine the MFE minima with corresponding distance classes. For easier comparision the outputfile of `RNA2Dfold` can be sorted by a simple sort command. For further information regarding sort use the `--help` option.

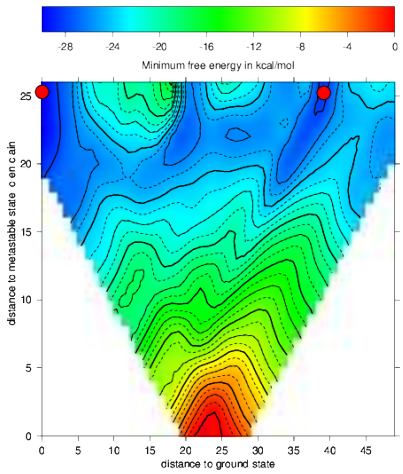
```
$ sort -k6 -n 2dfold.out > sort.out
```

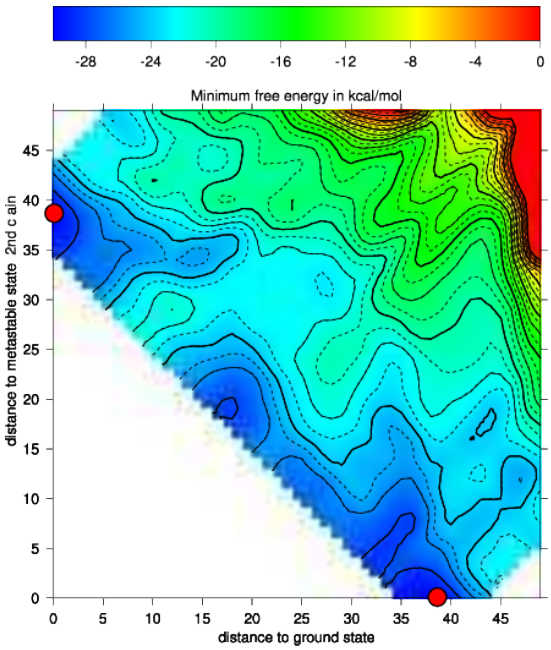
Now we choose the structure with the lowest energy besides our startstructure, replace the open chain structure from our old input with that structure and repeat the steps above with our new values:

- run `RNA2Dfold`
- plot it using `2Dlandscape_pf.gri`

The new projection (right graph) shows the two major local minima which are separated by 39 bp (red dots in figure below) and both are likely to be populated with high probability. The landscape gives an estimate of the energy barrier separating the two minima (about -20 kcal/mol).

The red dots mark the distance from open chain to the MFE structure respectively the distance from the 2nd best structure to the MFE. Note that the red dots were manually added to the image afterwards so don't panic if you don't see them in your gri output.





The Programs barriers and treekin

Table of Contents

- *Introduction*
- *A short recall on howto install/compile a program*
- *Calculate the Barrier Tree*
- *Simulating the Folding Kinetics*

Introduction

The following assumes you already have the `barriers` and `treekin` programs installed. They are **not** part of the `ViennaRNA Package` but their latest releases can be found at https://www.tbi.univie.ac.at/RNA/Barriers/ and https://www.tbi.univie.ac.at/RNA/Treekin/, respectively. Installation proceeds as shown for the `ViennaRNA Package`.

Note

One problem that often occurs during `treekin` installation is the dependency on `blas` and `lapack` packages. For further information according to the `barriers` and `treekin` program also see the website.

A short recall on howto install/compile a program

- Get the barriers source from https://www.tbi.univie.ac.at/RNA/Barriers/
- extract the archive and go to the directory:

```
$ tar -xzf Barriers-1.5.2.tar.gz
$ cd Barriers-1.5.2
```

- use the `--prefix` option to install in your `Progs/` directory:

```
$ ./configure --prefix=$HOME/Tutorial/Progs/barriers-1.5.2
```

- make install:

```
$ make
$ make install
```

Now `barriers` is ready to use. Apply the same steps to install `treekin`.

Note

Copy the `barriers` and `treekin` binaries to your `bin` folder or add the path to your `PATH` environment variable.

Calculate the Barrier Tree

```
$ echo UCCACGGCUGUUAGUGGAUAACGGC | RNAsubopt --noLP -s -e 10 > barseq.sub
$ barriers -G RNA-noLP --bsize --rates < barseq.sub > barseq.bar
```

You can restrict the number of local minima using the `barriers` command-line option `--max` followed by a number. The option `-G RNA-noLP` instructs barriers that the input consists of RNA secondary structures without

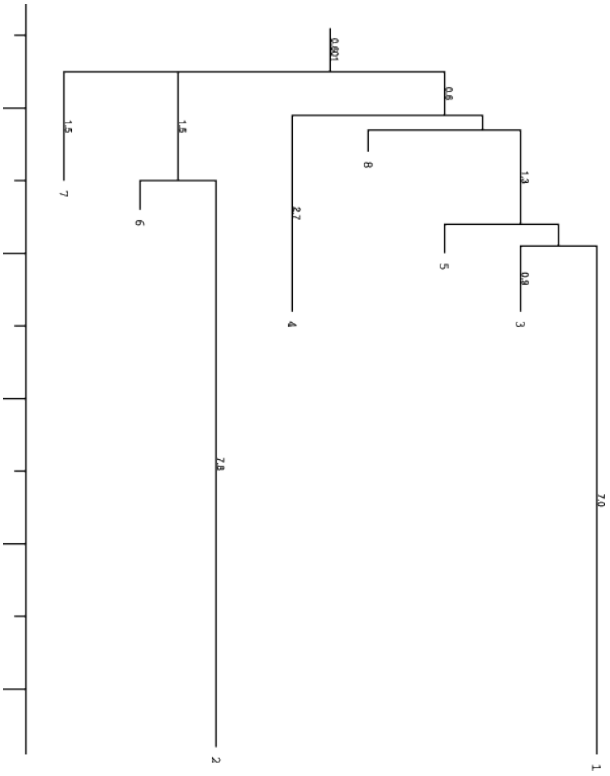
isolated basepairs. `--bsize` adds size of the gradient basins and `--rates` tells `barriers` to compute rates between macro states/basins for use with `treekin`. Another useful options is `--minh` to print only minima with a barrier $> dE$. Look at the output file `barseq.bar`, its content should be like:

```
  UCCACGGCUGUUAGUGGAUAACGGC
1 (((((........)))))......  -6.90    0  10.00    115     0  -7.354207     23  -7.
↪012023
2 ......(((((((.....)))))))  -6.80    1   9.30     32    58  -6.828221     38  -6.
↪828218
3 (((...(((...)))))).......  -0.80    1   0.90      1    10  -0.800000      9  -1.
↪075516
4 ....((..((((....)))).))..  -0.80    1   2.70      5    37  -0.973593     11  -0.
↪996226
5 ........................   0.00    1   0.40      1    14  -0.000000     26  -0.
↪612908
6 ......(((....((.....)))))   0.60    2   0.40      1    22   0.600000      3   0.
↪573278
7 ......((((((....)))...)))   1.00    1   1.50      1    95   1.000000      2   0.
↪948187
8 .((....((......)).....)).   1.40    1   0.30      1    30   1.400000      2   1.
↪228342
```

The first row holds the input sequence, the successive list the local minima ascending in energy. The meaning of the first 5 columns is as follows

- label (number) of the local minima (1=MFE)
- structure of the minimum
- free energy of the minimum
- label of deeper local minimum the current minimum merges with (note that the `MFE` has no deeper local minimum to merge with)
- height of the energy barrier to the local minimum to merge with
- numbers of structures in the basin we merge with
- number of basin which we merge to
- free energy of the basin
- number of structures in this basin using gradient walk
- gradient basin (consisting of all structures where gradientwalk ends in the minimum)

`barriers` produced two additional files, the *PostScript* file `tree.eps` which represents the basic information of the `barseq.bar` file visually:

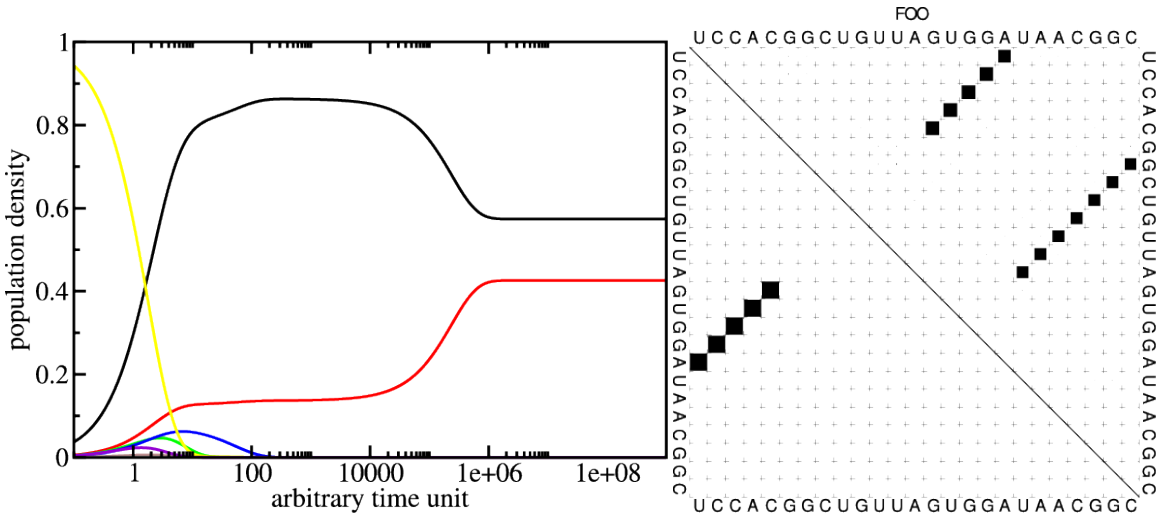


and a text file `rates.out` which holds the matrix of transition probabilities between the local minima.

Simulating the Folding Kinetics

The program `treekin` is used to simulate the evolution over time of the population densities of local minima starting from an initial population density distribution $p0$ (given on the command-line) and the transition rate matrix in the file `rates.out`.

```
$ treekin -m I --p0 5=1 < barseq.bar | xmgrace -log x -nxy -
```



The simulation starts with all the population density in the open chain (local minimum 5, see `barseq.bar`). Over time the population density of this state decays (yellow curve) and other local minima get populated. The simulation ends with the population densities of the thermodynamic equilibrium in which the MFE (black curve) and local minimum 2 (red curve) are the only ones populated. (Look at the dot plot of the sequence created with `RNAsubopt` and `RNAfold`!)

Part III

Advanced Usage

CHAPTER

THREE

RNALIB USER GUIDE

3.1 Introduction

This guide will show you how to use our implementations in your own C/C++, `Python 3`, or `Perl 5` programs and scripting pipelines without the need to call programs such as `RNAfold`. Interfacing directly with `RNAlib` allows you to easily **integrate, combine, and extend our algorithms** for particular tasks and applications.

See also

Minimal instructions for a technical setup to use our library can be found in *Python Interface* and *Linking against RNAlib*.

Our library is grouped into several modules, each addressing different aspects of RNA secondary structure related problems. Below is an overview of the **concepts and algorithms** we've been implementing.

3.2 Basic Structure Prediction

The fundamental algorithms in RNA secondary structure prediction certainly are minimum free energy (MFE) prediction and the partition function computation. Find useful information on how we implement these ideas and how you can use them in your own application in the following guides.

3.2.1 Minimum Free Energy (MFE)

General Remarks

According to the laws of Thermodynamics, the most probable conformation (here, secondary structure s) is the one with least free energy ($E(s)$), since:

$$p(s) \propto e^{-\beta E(s)} \text{ with } \beta = \frac{1}{RT}$$

where R is the gas constant and T the thermodynamic temperature. Hence, a reasonable RNA secondary structure representative is the structure with minimum free energy (MFE). The basic idea behind an efficient dynamic programming MFE prediction algorithm was first described by Zuker and Stiegler [1981]. Our library implements various variants of the original algorithmic idea that can be applied to:

- single sequences,
- multiple sequence alignments (MSA),
- multiple interacting sequences (RNA-RNA interaction), and
- circular sequences

All algorithms are usually accessible in two forms:

- a *Simplified MFE Interface* that requires minimal input and user interaction, but, at the same time is limited to default setting, and

- a *Generic Interface* that allows for a fine grained selection of algorithmic details, e.g. parameter settings or constraints

The latter provides functions that operate on *vrna_fold_compound_t* objects, which, in turn, can be fully customized to the specific needs for the use-case.

See also

The full list of available API symbols for global MFE prediction can be found in *Global MFE Prediction*.

Simplified MFE Interface

The most simple interface to these implementations might be the *vrna_fold()* function. It only requires an input RNA sequence in uppercase RNA alphabet (AUCG) to return the MFE in $kcal \cdot mol^{-1}$ and a corresponding secondary structure in *Dot-Bracket Notation*.

Single Sequence MFE

Below, you'll find an example that illustrates how to predict the MFE and a corresponding secondary structure for a single (linear) RNA sequence:

C

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/mfe/global.h>
#include <ViennaRNA/utils/basic.h>

int
main()
{
  /* The RNA sequence */
  char  *seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

  /* allocate memory for MFE structure (length + 1) */
  char  *structure = (char *)vrna_alloc(sizeof(char) * (strlen(seq) + 1));

  /* predict Minmum Free Energy and corresponding secondary structure */
  float mfe = vrna_fold(seq, structure);

  /* print sequence, structure and MFE */
  printf("%s\n%s [ %6.2f ]\n", seq, structure, mfe);

  /* cleanup memory */
  free(structure);

  return 0;
}
```

Python

```python
import RNA

# The RNA sequence
seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA"
```

(continues on next page)

(continued from previous page)

```
# compute minimum free energy (MFE) and corresponding structure
(ss, mfe) = RNA.fold(seq)

# print output
print("{}\n{} [ {:6.2f} ]".format(seq, ss, mfe))
```

Perl

```
use RNA;

# The RNA sequence
my $seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

# compute minimum free energy (MFE) and corresponding structure
my ($ss, $mfe) = RNA::fold($seq);

# print output
printf "%s\n%s [ %6.2f ]\n", $seq, $ss, $mfe;
```

The result of the above code should look like:

```
GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA
..(((((........)))))(((((((...)))))))..... [  -8.80 ]
```

Consensus MFE

In cases where you want to predict the consensus MFE structure for a set of aligned RNA sequences (MSA), consider the following example:

C

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/mfe/global.h>
#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/utils/alignments.h>

int
main()
{
  /* The RNA sequence alignment */
  const char  *sequences[] = {
    "CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGU",
    "CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGU",
    "---CUCGACACCACU---GCCUCGGUUACCCAUCGGUGCAGUGCGGGU",
    NULL /* indicates end of alignment */
  };

  /* compute the consensus sequence */
  char        *cons = consensus(sequences);

  /* allocate memory for MFE consensus structure (length + 1) */
```

(continues on next page)

(continued from previous page)

```
  char        *structure = (char *)vrna_alloc(sizeof(char) * (strlen(sequences[0]) +␣
↪1));

  /* predict Minmum Free Energy and corresponding secondary structure */
  float       mfe = vrna_alifold(sequences, structure);

  /* print consensus sequence, structure and MFE */
  printf("%s\n%s [ %6.2f ]\n", cons, structure, mfe);

  /* cleanup memory */
  free(cons);
  free(structure);

  return 0;
}
```

Python

```
import RNA

# The RNA sequence alignment
sequences = [
    "CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGU",
    "CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGU",
    "---CUCGACACCACU---GCCUCGGUUACCCAUCGGUGCAGUGCGGGU"
]

# compute the consensus sequence
cons = RNA.consensus(sequences)

# predict Minmum Free Energy and corresponding secondary structure
(ss, mfe) = RNA.alifold(sequences);

# print output
print("{}\n{} [ {:6.2f} ]".format(cons, ss, mfe))
```

Perl

```
use RNA;

# The RNA sequence alignment
my @sequences = (
    "CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGU",
    "CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGU",
    "---CUCGACACCACU---GCCUCGGUUACCCAUCGGUGCAGUGCGGGU"
);

# compute the consensus sequence
my $cons = RNA::consensus(\@sequences);

# predict Minmum Free Energy and corresponding secondary structure
my ($ss, $mfe) = RNA::alifold(\@sequences);

# print output
printf "%s\n%s [ %6.2f ]\n", $cons, $ss, $mfe;
```

Note

Further simplified functions that compute the MFE for more special cases include:

- *vrna_circfold()* - Treats the RNA sequence as circular sequence (Hofacker and Stadler [2006])
- *vrna_alifold()* - Uses a Multiple Sequence Alignment as input to compute consensus MFE
- *vrna_circalifold()* - Treats the aligned sequences as circular sequences

Generic Interface

The generic MFE interface requires one to create a *vrna_fold_compound_t* object first before calling the *vrna_mfe()* function (or method). Adaptations to the algorithm parameters can be either introduced through the second or third *vrna_fold_compound_t* constructor parameter (see also the *vrna_fold_compound()* function), or later via the various *vrna_fold_compound_t* associated methods (or functions in C).

The following example demonstrates how to change some of the model parameters before actually computing the MFE (structure) under the non-default settings.

C

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/model.h>
#include <ViennaRNA/fold_compound.h>
#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/mfe/global.h>

int
main()
{
  char      *seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

  /* allocate memory for MFE structure (length + 1) */
  char      *structure = (char *)vrna_alloc(sizeof(char) * (strlen(seq) + 1));

  /* create a new model details structure to store the Model Settings */
  vrna_md_t md;

  /* ALWAYS set default model settings first! */
  vrna_md_set_default(&md);

  /* Mark sequence as being circular, change temperature to
   * non-default and change dangle model
   */
  md.circ         = 1;    /* Treat RNA input as circular sequence */
  md.temperature  = 25.0; /* 25 Deg Celcius */
  md.dangles      = 1;    /* Switch to dangles-model 1 */

  /* create a fold compound */
  vrna_fold_compound_t  *fc = vrna_fold_compound(seq, &md, VRNA_OPTION_DEFAULT);

  /* predict Minmum Free Energy and corresponding secondary structure */
  float                 mfe = vrna_mfe(fc, structure);

  /* print sequence, structure and MFE */
```

(continues on next page)

(continued from previous page)

```
  printf("%s\n%s [ %6.2f ]\n", seq, structure, mfe);

  /* cleanup memory */
  free(structure);
  vrna_fold_compound_free(fc);

  return 0;
}
```

Python

```
import RNA

# The RNA sequence
seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA"

# create a new model details structure with
# non-default settings
md = RNA.md(
        circ        = 1,    # Circular RNA sequence
        temperature = 25.0, # 25 Deg Celcius
        dangles     = 1     # Dangle Model 1
      )

# create a fold compound
fc = RNA.fold_compound(seq, md)

# predict Minmum Free Energy and corresponding secondary structure
(ss, mfe) = fc.mfe()

# print sequence, structure and MFE
print("{}\n{} [ {:6.2f} ]".format(seq, ss, mfe))
```

Perl

```
use RNA;

# The RNA sequence
my $seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

# create a new model details structure
my $md = new RNA::md();

# change temperature and dangle model
$md->{circ}         = 1;    # Circular RNA sequence
$md->{temperature}  = 25.0; # 25 Deg Celcius
$md->{dangles}      = 1;    # Dangle Model 1

# create a fold compound
my $fc = new RNA::fold_compound($seq, $md);

# predict Minmum Free Energy and corresponding secondary structure
my ($ss, $mfe) = $fc->mfe();

# print sequence, structure and MFE
printf "%s\n%s [ %6.2f ]\n", $seq, $ss, $mfe;
```

This should produce the following output:

```
GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA
..(((((........)))))(((((((...)))))))..... [  -5.64 ]
```

Backtracking

The implementations for mfe/backtracking are generally agnostic with respect to whether local or global structure prediction is in place.

See also

The full list of available API symbols for backtracking of MFE structures can be found in *Backtracking MFE structures*.

Local MFE Prediction

In addition to the global MFE prediction algorithm(s) above, we also provide implementations for the local (sliding window) MFE prediction, in particular for

- Single sequences,
- Multiple sequence alignments (MSA), and

Note, that our implementation scans an RNA sequence (or MSA) from the 3' to the 5' end, and reports back locally optimal (consensus) structures, the corresponding free energy, and the position of the sliding window in global coordinates.

For any particular RNA sequence (or MSA) multiple locally optimal (consensus) secondary structures may be predicted. Thus, we tried to implement an interface that allows for an effortless conversion of the corresponding hits into any target data structure. As a consequence, we provide two distinct ways to retrieve the corresponding predictions, either

- through directly writing to an open `FILE` stream on-the-fly, or
- through a callback function mechanism.

The latter allows one to store the results in any possible target data structure. Our implementations then pass the results through the user-implemented callback as soon as the prediction for a particular window is finished.

Return Valued Based Example

Here is some example code that shows how to use the simple local MFE prediction API:

C

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/mfe/local.h>
#include <ViennaRNA/utils/basic.h>

int
main()
{
  /* a long RNA sequence input */
  char  *seq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCA" \
               "GGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCACUACGUUGGUAUC" \
               "AUGUCCUCCGUCCUAACAAUUAUACAUCGAGAGGCAAAAUUUCUAAUCCGGGGUC" \
```

(continues on next page)

(continued from previous page)

```
                "AGUGAGCAUUGCCAUUUUAUAACUCGUGAUCUCUCGCUACUUAGGCGAUCCCUGC" \
                "CAAUGAGGGUCAAGGAGUUGAAUUAUCGGGCCACAUCGACGUGGCCUUUACGGCC" \
                "AGGUAAUUCAAAGGCCUCAAGUCCU";

  /* allocate memory for MFE structure (length + 1) */
  char  *structure = (char *)vrna_alloc(sizeof(char) * (strlen(seq) + 1));

  /* predict local MFE structures with maximum base pair distance of 50 nt */
  float e = vrna_Lfold(seq, 50, NULL);

  /* cleanup memory */
  free(structure);

  return 0;
}
```

Python

```
import RNA

# a long RNA sequence input
longseq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCA" \
          "GGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCACUACGUUGGUAUC" \
          "AUGUCCUCCGUCCUAACAAUUAUACAUCGAGAGGCAAAAUUUCUAAUCCGGGGUC" \
          "AGUGAGCAUUGCCAUUUUAUAACUCGUGAUCUCUCGCUACUUAGGCGAUCCCUGC" \
          "CAAUGAGGGUCAAGGAGUUGAAUUAUCGGGCCACAUCGACGUGGCCUUUACGGCC" \
          "AGGUAAUUCAAAGGCCUCAAGUCCU"

# predict local MFE structures with maximum base pair distance of 50 nt
e = RNA.Lfold(longseq, 50)
```

Perl

```
use RNA;

# a long RNA sequence input
my $longseq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCA" .
              "GGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCACUACGUUGGUAUC" .
              "AUGUCCUCCGUCCUAACAAUUAUACAUCGAGAGGCAAAAUUUCUAAUCCGGGGUC" .
              "AGUGAGCAUUGCCAUUUUAUAACUCGUGAUCUCUCGCUACUUAGGCGAUCCCUGC" .
              "CAAUGAGGGUCAAGGAGUUGAAUUAUCGGGCCACAUCGACGUGGCCUUUACGGCC" .
              "AGGUAAUUCAAAGGCCUCAAGUCCU";

# predict local MFE structures with maximum base pair distance of 50 nt
my $e = RNA::Lfold($longseq, 50);
```

In the example above, we didn't provide an output file stream as third argument for *vrna_Lfold()*. Hence, the output was automatically written to `stdout`.

Callback Based Example

In contrast to the previous example, here is a variant that uses the callback based local MFE prediction API:

C

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/mfe/local.h>
#include <ViennaRNA/utils/basic.h>

/* a callback function that simply prints the resulting
 * local MFE prediction
 */
void
callback(unsigned int start,
         unsigned int end,
         const char *structure,
         float en,
         void *data)
{
  if (structure) {
    printf("%.*s\n%s (%6.2f) - [%u:%u]\n",
           end - start + 1,
           (char *)data) + start - 1,
           structure,
           en,
           start,
           end);
  }
}

int
main()
{
  /* a long RNA sequence input */
  char  *seq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCA" \
               "GGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCACUACGUUGGUAUC" \
               "AUGUCCUCCGUCCUAACAAUUAUACAUCGAGAGGCAAAAUUUCUAAUCCGGGGUC" \
               "AGUGAGCAUUGCCAUUUUAUAACUCGUGAUCUCUCGCUACUUAGGCGAUCCCUGC" \
               "CAAUGAGGGUCAAGGAGUUGAAUUAUCGGGCCACAUCGACGUGGCCUUUACGGCC" \
               "AGGUAAUUCAAAGGCCUCAAGUCCU";

  /* predict local MFE structures with maximum base pair distance of 50 nt */
  float e = vrna_Lfold_cb(seq, 50, callback, (void *)seq);

  return 0;
}
```

Python

```python
import RNA

# a long RNA sequence input
longseq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCA" \
          "GGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCACUACGUUGGUAUC" \
          "AUGUCCUCCGUCCUAACAAUUAUACAUCGAGAGGCAAAAUUUCUAAUCCGGGGUC" \
          "AGUGAGCAUUGCCAUUUUAUAACUCGUGAUCUCUCGCUACUUAGGCGAUCCCUGC" \
          "CAAUGAGGGUCAAGGAGUUGAAUUAUCGGGCCACAUCGACGUGGCCUUUACGGCC" \
```

(continues on next page)

(continued from previous page)

```
          "AGGUAAUUCAAAGGCCUCAAGUCCU"

def callback(start, end, structure, en, data = None):
    if structure:
        print(f"{data[start - 1:end]}\n{structure} ({en:6.2f}) - [{start}-{end}]")

# predict local MFE structures with maximum base pair distance of 50 nt
e = RNA.Lfold_cb(longseq, 50, callback, longseq)
```

Perl

```
use RNA;

# a long RNA sequence input
my $longseq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCA" .
              "GGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCACUACGUUGGUAUC" .
              "AUGUCCUCCGUCCUAACAAUUAUACAUCGAGAGGCAAAAUUUCUAAUCCGGGGUC" .
              "AGUGAGCAUUGCCAUUUUAUAACUCGUGAUCUCUCGCUACUUAGGCGAUCCCUGC" .
              "CAAUGAGGGUCAAGGAGUUGAAUUAUCGGGCCACAUCGACGUGGCCUUUACGGCC" .
              "AGGUAAUUCAAAGGCCUCAAGUCCU";

sub callback {
  my ($start, $end, $structure, $en, $data) = @_;
  if ($structure) {
    printf(substr($data, start - 1, end - start + 1) .
           $structure .
           "(%6.2f) - [%u:%u]",
           en,
           start,
           end);
  }
}

# predict local MFE structures with maximum base pair distance of 50 nt
my $e = RNA::Lfold_cb($longseq, 50, \&callback, \&longseq);
```

See also

The full list of available API symbols for local (sliding-window) MFE prediction can be found in *Local MFE Prediction*.

3.2.2 Partition Function and Equilibrium Probability

Introduction

The principal idea behind computing the partition function for RNA secondary structures can be summarized as follows:

> In equilibrium, statistical mechanics and polymer theory tells us that the frequency or probability $p(s)$ of a particular state s depends on its energy $E(s)$ and follows a Boltzmann distribution.

This can be formally expressed as

$$p(s) \propto e^{-\beta E(s)} \text{ with } \beta = \frac{1}{kT}$$

where $k \approx 1.987 \cdot 10^{-3} \frac{kcal}{mol\ K}$ is the Boltzmann constant, and T the thermodynamic temperature. From that relation, the actual probability of state s can then be obtained using a proper scaling factor, the *canonical partition*

function

$$Z = \sum_{s \in \Omega} e^{-\beta E(s)}$$

where Ω is the finite set of all states. Finally, the equilibrium probability of state s can be computed as

$$p(s) = \frac{e^{-\beta E(s)}}{Z}$$

The first application of the above principles to an ensemble Ω of RNA secondary structures $s \in \Omega$ has been described by McCaskill [1990]. Instead of enumerating all states exhaustively to compute Z the basic idea is to make use of the recursive decompositions described in *The RNA Folding Grammar*. This in turn allows for an efficient computation of Z in $\mathcal{O}(n^3)$, i.e. cubic time. An **outside** variant of the same recursions is then used to compute probabilities for base pairs, stretches of consecutive unpaired nucleotides, or structural motifs.

Implementation

Our implementations provide access to the algorithm(s) made available by McCaskill [1990] and its variations. This allows us to efficiently compute

- the partition function, and from that
- various equilibrium probabilities, for instance base pair probabilities, probabilities of individual structure motifs, and many more.

Similar to our *Minimum Free Energy (MFE)*, we provide two different flavors for partition function computations, **global** and **local** (sliding-window) algorithms, as well as **simplified** and **generic** interfaces.

Comparative predictions for multiple sequence alignment (MSA) input compute an average of the free energy contributions plus an additional covariance pseudo-energy term, exactly as we do for the *Minimum Free Energy (MFE)*.

Warning

Local (consensus) partition function computations for multiple sequence alignment (MSA) input is currently not supported!

Boltzmann weights for the free energy contributions of individual loops can be found in *Energy Evaluation for Individual Loops*.

Our library also implements a stochastic (Boltzmann) sampling procedure to draw random RNA secondary structures from the ensemble according to their equilibrium probabilty.

See also

Further details of the Partition function and Base Pair Probability algorithm can be obtained from McCaskill [1990]. The description for the basic partition function API can be found at *Partition Function and Equilibrium Probabilities*.

Global Partition Function and Equilibrium Probabilities

We provide implementations of the global partition function algorithm for

- Single sequences,
- Multiple sequence alignments (MSA), and
- RNA-RNA hybrids

See also

A full list of available API symbols is available at *Global Partition Function*.

Local Partition Function and Equilibrium Probabilities

Scanning version using a sliding window approach to compute equilibrium probabilities are also available. They allow for efficient prediction of local secondary structure features, e.g. for very long RNA sequences.

The following Python example demonstrates how to use the local partition function API to compute **base pair probabilities**:

Return value based

```
import RNA

# a rather long RNA sequence
longseq =
↪"AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCA
↪"

# get a list of base pair probabilities with window
# size of 200nt, maximum base pair span of 150nt
# and a probability threshold of 0.01
for p in RNA.pfl_fold(longseq, 200, 150, 0.01):
    print(p)
```

Callback function based

```
import RNA

# a rather long RNA sequence
longseq =
↪"AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCA
↪"

# callback function to store the probabilities
def pf_window_callback(v, v_size, i, maxsize, what, data=None):
    if what & RNA.PROBS_WINDOW_BPP:
        data.extend([{'i': i, 'j': j, 'p': p} for j, p in enumerate(v) if (p is not␣
↪None) and (p >= 0.01)])

# Prepare fold compound for local structure prediction
# with window size of 200nt, maximum base pair span of
# 150nt
fc = RNA.fold_compound(longseq,
                       RNA.md(max_bp_span = 150, window_size = 200),
                       RNA.OPTION_WINDOW)
# storage container for probabilities
data = []
# run local structure prediction callback API
fc.probs_window(10, RNA.PROBS_WINDOW_BPP, pf_window_callback, data)

for p in data:
    print(p)
```

Unpaired probabilities can be predicted similarly:

Return value based

```
import RNA

# a rather long RNA sequence
longseq =
↪"AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCA
↪"

# get a list of unpaired probabilities for subsequences
# up to the size of 10, computed with window size of
# 200nt and maximum base pair span of 150nt
for up in RNA.pfl_fold_up(10, longseq, 200, 150):
    print(p)
```

Callback function based

```
import RNA

# a rather long RNA sequence
longseq =
↪"AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCA
↪"

# callback function to store the unpaired probabilities
def up_callback(v, v_size, i, maxsize, what, data):
    if what & RNA.PROBS_WINDOW_UP:
        data.append({ 'i': i, 'up': v})

# Prepare fold compound for local structure prediction
# with window size of 200nt, maximum base pair span of
# 150nt
fc = RNA.fold_compound(longseq,
                       RNA.md(max_bp_span = 150, window_size = 200),
                       RNA.OPTION_WINDOW)
# storage container for probabilities
data = []
# run local structure prediction callback API
fc.probs_window(10, RNA.PROBS_WINDOW_UP, up_callback, data)

for up in data:
    print(up)
```

See also

A full list of available API symbols is available at *Local Partition Function*.

Predicting various Thermodynamic Properties

As stated above, many thermodynamic properties can be derived from the Z. In particular, within our model of RNA secondary structures, the equilibrium probabilty $p(F)$ of a particular structural feature F follows Boltzmanns law, i.e.:

$$p(F) \propto \sum_{s|F \in s} e^{\frac{-E(s)}{kT}}.$$

The actual probabilities can then be obtained from the ratio of those structures containing F and *all* structures, i.e.

$$p(F) = \frac{1}{Z} \sum_{s|F \in s} e^{\frac{-E(s)}{kT}}.$$

Efficient dynamic programming algorithms exist to compute the equilibrium probabilities

$$p_{ij} = \frac{1}{Z} \sum_{s|(i,j) \in s} e^{\frac{-E(s)}{kT}}$$

of base pairs (i, j) without the need for exhaustive enumeration of s.

See also

A full list of available API symbols is available at *Thermodynamic Properties API*.

3.2.3 Suboptimals and Representative Structures

Suboptimal Structures sensu Zuker

The algorithm to compute optimal secondary structures that contain a particular base pair has been published by Zuker [1989] and is based on ideas for predicting structures for circular RNAs, in particular viroids, as presented in Steger *et al.* [1984].

Suboptimal Structures within an Energy Band around the MFE

Callback Based Suboptimals

C

```c
#include <stdlib.h>
#include <stdio.h>

#include <ViennaRNA/fold_compound.h>
#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/utils/strings.h>
#include <ViennaRNA/subopt/basic.h>

void
subopt_callback(const char  *structure,
                float       energy,
                void        *data)
{
  /* simply print the result and increase the counter variable by 1 */
  if (structure)
    printf("%04d\t%s\t%6.2f\n", (*((int *)data))++, structure, energy);
}


int
main()
{
  char *sequence = "GGGGAAAACCCC"

  /* Create a fold compound for the sequence */
  vrna_fold_compound_t  *fc     = vrna_fold_compound(seq, NULL, VRNA_OPTION_DEFAULT);
  int                   counter = 0;
```

(continues on next page)

(continued from previous page)

```
  /*
   *  call subopt to enumerate all secondary structures in an energy band of
   *  5 kcal/mol of the MFE and pass it the address of the callback and counter
   *  variable
   */
  vrna_subopt_cb(fc, 500, &subopt_callback, (void *)&counter);

  /* cleanup memory */
  free(seq);
  vrna_fold_compound_free(fc);

  return 0;
}
```

Python

```
import RNA

sequence = "GGGGAAAACCCC"
counter  = 0

# Print a subopt result and increase counter by 1
def print_subopt_result(structure, energy, data):
    if structure != None:
        data += 1
        print(f"{data:04d}\t{structure}\t{energy:6.2f}")

# Get model details with unique ML decomposition
md = RNA.md(uniq_ML = 1)

# Create a 'fold_compound' for our sequence
fc = RNA.fold_compound(sequence, md)

# Enumerate all structures 500 dacal/mol = 5 kcal/mol arround
# the MFE and print each structure using the function above
fc.subopt_cb(500, print_subopt_result, counter);
```

Boltzmann Sampling

Below follow a few Python examples that demonstrate the usage of the Boltzmann sampling implementations. To prepare the Boltzmann sampling implementation, we first need to compute the partition function and thereby fill the dynamic programming matrices required for the stochastic backtracking:

Prerequisites

```
import RNA

sequence = "UGGGAAUAGUCUCUUCCGAGUCUCGCGGGCGACGGGCGAUCUUCGAAAGU \
            GGAAUCCGUACUUAUACCGCCUGUGCGGACUACUAUCCUGACCACAUAGU"

"""
First we prepare a fold_compound object and run the usual
steps to compute the partition function
"""
```

(continues on next page)

(continued from previous page)

```
# create model details
md = RNA.md()

# activate unique multibranch loop decomposition
md.uniq_ML = 1

# create fold compound object
fc = RNA.fold_compound(sequence, md)

# compute MFE
(ss, mfe) = fc.mfe()

# rescale Boltzmann factors according to MFE
fc.exp_params_rescale(mfe)

# compute partition function to fill DP matrices
fc.pf()

"""
Now we are ready to perform Boltzmann sampling
"""
```

Let us now step by step go through the different ways to create samples of (sub)structures from the ensemble.

Sampling Substructures

We begin with sub-structures of a particular length that start at the 5' end, i.e. pos 1, of the input sequence:

Return value based

```
# 1. backtrace a single sub-structure of length 10
print(fc.pbacktrack5(10))

# 2. backtrace a single sub-structure of length 50
print(fc.pbacktrack5(50))

# 3. backtrace multiple sub-structures of length 10 at once
for s in fc.pbacktrack5(20, 10):
    print(s)

# 4. backtrace multiple sub-structures of length 50 at once
for s in fc.pbacktrack5(100, 50):
    print(s)
```

Callback based

```
def store_structure(structure, data):
    """
    A simple callback function that stores
    a structure sample into a list
    """
    if structure:
        data.append(structure)

# Draw a sample of 100 sub-structures of length 50
ss  = []
```

(continues on next page)

(continued from previous page)

```
i   = fc.pbacktrack5(100, 50, store_structure, ss)

for s in ss:
    print(s)
```

Of course one can sample substructures from anywhere within the input sequence as well:

Return value based

```
# 1. backtrace a single substructure from the subsequence [10:50]
print(fc.pbacktrack_sub(10, 50))

# 2. backtrace multiple substructures from the subsequence [10:50]
for s in fc.pbacktrack_sub(100, 10, 50):
    print(s)

# 3. backtrace multiple substructures from the subsequence [10:50] non-redundantly
for s in fc.pbacktrack_sub(100,
                           10,
                           50,
                           RNA.PBACKTRACK_NON_REDUNDANT):
    print(s)
```

Callback based

```
def store_structure(structure, data):
    """
    A simple callback function that stores
    a structure sample into a list
    """
    if structure:
        data.append(structure)

# list container to store our structures
ss  = []

# 1. backtrace multiple substructures from the subsequence [10:50]
i   = fc.pbacktrack_sub(100, 10, 50, store_structure, ss)

# 2. backtrace multiple substructures from the subsequence [10:50] non-redundantly
i   = fc.pbacktrack_sub(100,
                        10,
                        50,
                        store_structure,
                        ss,
                        RNA.PBACKTRACK_NON_REDUNDANT)

for s in ss:
    print(s)
```

Sampling Full-length Structures

Finally, let us draw full-length structures:

Return value based

```
# 1. backtrace a single structure (full length)
print(fc.pbacktrack())

# 2. backtrace multiple structures at once
for s in fc.pbacktrack(100):
    print(s)

# 3. backtrace multiple structures non-redundantly
for s in fc.pbacktrack(100, RNA.PBACKTRACK_NON_REDUNDANT):
    print(s)
```

Callback based

```
def store_structure(structure, data):
    """
    A simple callback function that stores
    a structure sample into a list
    """
    if structure:
        data.append(structure)

# list container to store our structures
ss  = []

# 1. backtrace multiple full-length structures in callback mode
i   = fc.pbacktrack(100, store_structure, ss)

# 2. non-redundantly backtrace multiple full-length structures in callback mode
i   = fc.pbacktrack(100,
                    store_structure,
                    ss,
                    RNA.PBACKTRACK_NON_REDUNDANT)

for s in ss:
    print(s)
```

Non-redundant Sampling

The **non-redundant** mode for Boltzmann sampling (activated by `RNA.PBACKTRACK_NON_REDUNDANT`) removes each sample that has been drawn from the ensemble. Thus, no structure will be sampled more than once.

Whenever you drew a few structures in non-redundant mode and want to continue sampling from the remaining sub-ensemble that is left, you can use the **resume** capability for this sampling process:

Return value based

```
# perform 15 iterations and draw 500 non-redundant structures each time
# (with resume option)
num_samples = 500
iterations  = 15
d           = None # pbacktrack memory object
s_list      = []

for i in range(0, iterations):
    d, ss   = fc.pbacktrack(num_samples, d, RNA.PBACKTRACK_NON_REDUNDANT)
```

(continues on next page)

(continued from previous page)

```
    s_list.extend(ss)

for s in s_list:
    print(s)
```

Callback based

```
def store_structure(structure, data):
    """
    A simple callback function that stores
    a structure sample into a list
    """
    if structure:
        data.append(structure)

# Non-redundantly backtrace multiple full length structures
# in callback mode with resume option
num_samples = 500
iterations  = 15
d           = None # pbacktrack memory object
s_list      = []

for i in range(0, iterations):
    d, i = fc.pbacktrack(num_samples,
                         store_structure,
                         s_list,
                         d,
                         RNA.PBACKTRACK_NON_REDUNDANT)

for s in ss:
    print(s)
```

Maximum Expected Accuracy (MEA) Structure

Another useful structure representative one can determine from base pairing probabilities p_{ij} is the structure that exhibits the *maximum expected accuracy* (MEA). By assuming the base pair probability is a good measure of correctnes of a pair (i, j), the expected accuracy of a structure is

$$EA(s) = \sum_{(i,j)\in s} 2\gamma p_{ij} + \sum_{i \notin s} p_i^u$$

with unpaired probabilities $p_i^u = 1 - \sum_{j \neq i} p_{ij}$ and weighting factor γ that allows us to weight *paired* against *unpaired* positions. The MEA algorithm attempts to find the structure that maximizes EA. **Higher values** of γ result in more base pairs of lower probability and thus **higher sensitivity**. **Low values** of γ result in structures containing only highly likely pairs, i.e. **high specificity**.

Example MEA structure Prediction

C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include  <ViennaRNA/data_structures.h>
#include  <ViennaRNA/params/basic.h>
```

(continues on next page)

(continued from previous page)

```c
#include  <ViennaRNA/utils/basic.h>
#include  <ViennaRNA/eval/structures.h>
#include  <ViennaRNA/mfe/global.h>
#include  <ViennaRNA/partfunc/basic.h>
#include  <ViennaRNA/structures/mea.h>


int
main(int  argc,
     char *argv[])
{
  char *seq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGA";

  /* get a vrna_fold_compound with default settings */
  vrna_fold_compound_t *vc = vrna_fold_compound(seq, NULL, VRNA_OPTION_DEFAULT);

  /* call MFE function */
  double mfe = (double)vrna_mfe(vc, NULL);

  /* rescale parameters for Boltzmann factors */
  vrna_exp_params_rescale(vc, &mfe);

  /* call PF function */
  FLT_OR_DBL en = vrna_pf(vc, NULL);

  /* compute centroid structure */
  float mea;
  char  *mea_structure = vrna_MEA(fc, 1.0, &mea);

  /* print MEA structure, its free energy and maximum expected accuracy */
  printf("%s\n%s (%6.2f MEA=%6.2f)\n",
         seq,
         mea_structure,
         vrna_eval_structure(vc, mea_structure),
         mea);

  /* free MEA structure */
  free(mea_structure);

  /* free memory occupied by vrna_fold_compound */
  vrna_fold_compound_free(vc);

  return EXIT_SUCCESS;
}
```

Python

```python
#!/usr/bin/python
#

import RNA

seq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGA"

# create fold_compound data structure
# (required for all subsequently applied  algorithms)
```

(continues on next page)

(continued from previous page)

```
fc = RNA.fold_compound(seq)

# compute MFE and MFE structure
(mfe_struct, mfe) = fc.mfe()

# rescale Boltzmann factors for partition function computation
fc.exp_params_rescale(mfe)

# compute partition function
(pp, pf) = fc.pf()

# compute MEA structure
(mea_struct, mea) = fc.MEA(1.0)

# compute free energy of MEA structure
mea_en = fc.eval_structure(mea_struct)

# print only the MEA structure, its free energy
# in kcal/mol and the maximum exprected accuracy
print(f"{seq}\n{mea_struct} ({mea_en:6.2f} d={mea:.2f})")
```

Perl

```
#!/usr/bin/perl
#

use RNA;

my $seq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGA";

# create fold_compound data structure
# (required for all subsequently applied  algorithms)
my $fc = new RNA::fold_compound($seq);

# compute MFE and MFE structure
my ($mfe_struct, $mfe) = $fc->mfe();

# rescale Boltzmann factors for partition function computation
$fc->exp_params_rescale($mfe);

# compute partition function
my ($pp, $pf) = $fc->pf();

# compute MEA structure
my ($mea_struct, $mea) = $fc->MEA(1.0);

# compute free energy of MEA structure
my $mea_en = $fc->eval_structure($mea_struct);

# print only the MEA structure, its free energy
# in kcal/mol and the maximum expected accuracy
printf("%s\n%s (%6.2f MEA = %.2f)\n", $seq, $mea_struct, $mea_en, $mea);
```

The examples above should all print the following:

```
AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGA
```

(continues on next page)

(continued from previous page)

```
.....((((..(((...)))..))))((((.(....).)))).((.(((.....))).)).. ( -8.30 MEA = 47.47)
```

The Centroid Structure

The centroid is the structure with the minimal average distance to all other structures

$$< d(s) >= \sum_{(i,j) \in s} (1 - p_{ij}) + \sum_{(i,j) \notin s} p_{ij}$$

Thus, the centroid is simply the structure containing all pairs with $p_ij > 0.5$

Example Centroid structure Prediction

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include  <ViennaRNA/data_structures.h>
#include  <ViennaRNA/params/basic.h>
#include  <ViennaRNA/utils/basic.h>
#include  <ViennaRNA/eval/structures.h>
#include  <ViennaRNA/mfe/global.h>
#include  <ViennaRNA/partfunc/basic.h>
#include  <ViennaRNA/structures/centroid.h>


int
main(int  argc,
     char *argv[])
{
  char *seq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGA";

  /* get a vrna_fold_compound with default settings */
  vrna_fold_compound_t *vc = vrna_fold_compound(seq, NULL, VRNA_OPTION_DEFAULT);

  /* call MFE function */
  double mfe = (double)vrna_mfe(vc, NULL);

  /* rescale parameters for Boltzmann factors */
  vrna_exp_params_rescale(vc, &mfe);

  /* call PF function */
  FLT_OR_DBL en = vrna_pf(vc, NULL);

  /* compute centroid structure */
  double  dist;
  char    *cent = vrna_centroid(vc, &dist);

  /* print centroid structure, its free energy and mean distance to the ensemble */
  printf("%s\n%s (%6.2f d=%6.2f)\n",
         seq,
         cent,
         vrna_eval_structure(vc, cent),
         dist);
```

(continues on next page)

(continued from previous page)

```
  /* free centroid structure */
  free(cent);

  /* free memory occupied by vrna_fold_compound */
  vrna_fold_compound_free(vc);

  return EXIT_SUCCESS;
}
```

Python

```
#!/usr/bin/python
#

import RNA

seq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGA"

# create fold_compound data structure
# (required for all subsequently applied  algorithms)
fc = RNA.fold_compound(seq)

# compute MFE and MFE structure
(mfe_struct, mfe) = fc.mfe()

# rescale Boltzmann factors for partition function computation
fc.exp_params_rescale(mfe)

# compute partition function
(pp, pf) = fc.pf()

# compute centroid structure
(centroid_struct, dist) = fc.centroid()

# compute free energy of centroid structure
centroid_en = fc.eval_structure(centroid_struct)

# print only the centroid structure, its free energy
# in kcal/mol and the distance to the ensemble
print(f"{seq}\n{centroid_struct} ({centroid_en:6.2f} d={dist:.2f})")
```

Perl

```
#!/usr/bin/perl
#

use RNA;

my $seq = "AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGA";

# create fold_compound data structure
# (required for all subsequently applied  algorithms)
my $fc = new RNA::fold_compound($seq);

# compute MFE and MFE structure
```

(continues on next page)

(continued from previous page)

```
my ($mfe_struct, $mfe) = $fc->mfe();

# rescale Boltzmann factors for partition function computation
$fc->exp_params_rescale($mfe);

# compute partition function
my ($pp, $pf) = $fc->pf();

# compute centroid structure
my ($centroid_struct, $dist) = $fc->centroid();

# compute free energy of centroid structure
my $centroid_en = $fc->eval_structure($centroid_struct);

# print only the centroid structure, its free energy
# in kcal/mol and the distance to the ensemble
printf("%s\n%s (%6.2f d = %.2f)\n", $seq, $centroid_struct, $centroid_en, $dist);
```

The examples above should all print the following:

```
AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGA
.....((((..((.....)).))))((((........)))).((.(((.....))).)).. ( -7.20 d = 8.96)
```

3.2.4 Free Energy Evaluation

Under the **Nearest Neighbor** energy model secondary structures s are decomposed into individual loops L and the overall free energy $E(s)$ of a structure is estimated as the sum of free energy contributions of the individual loops a structure is composed of, i.e.:

$$E(s) = \sum_{L \in s} E_L$$

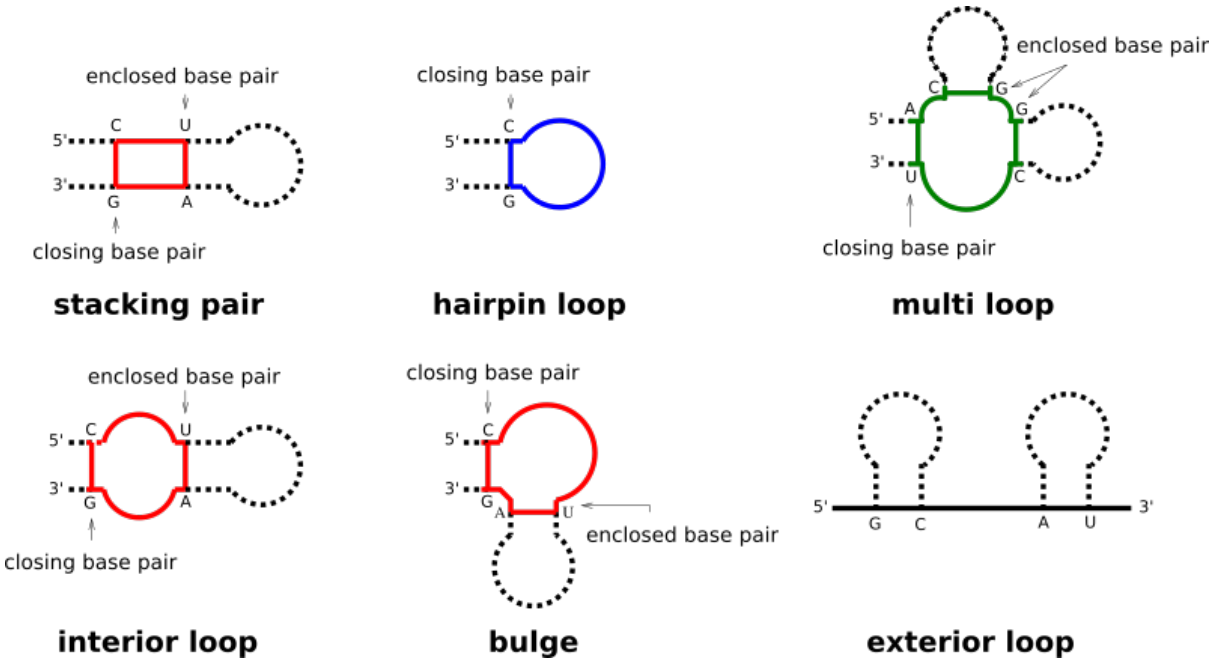
Here, we demonstrate how this is done and which parts of the `RNAlib` API are dedicated to free energy evaluation.

Loop Decomposition

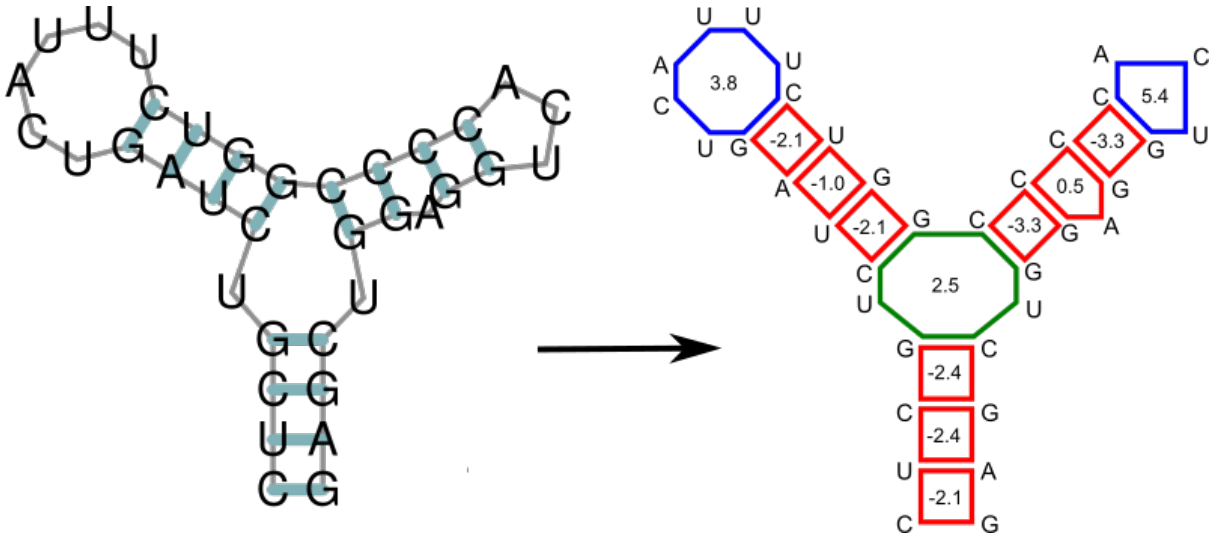
Each base pair in a secondary structure closes a loop, thereby directly enclosing unpaired nucleotides, and/or further base pairs. Our implementation distinguishes four basic types of loops:

- hairpin loops
- internal loops
- multibranch loops
- exterior loop

While the exterior loop is a special case without a closing pair, the other loops are determined by the number of base pairs involved in the loop formation, i.e. hairpin loops are 1-loops, since only a single base pair delimits the loop. internal loops are 2-loops due to their enclosing, and enclosed base pair. All loops where more than two base pairs are involved, are termed multibranch loops.



Any secondary structure can be decomposed into its loops. Each of the loops then can be scored in terms of free energy, and the free energy of an entire secondary structure is simply the sum of free energies of its loops.



Energy Parameters

For secondary structure free energy evaluation we usually utilize the set of thermodynamic **Nearest Neighbor** energy parameters also used in other software, such as `UNAfold` and `RNAstructure`.

Available Parameter Sets

While `RNAlib` already contains a compiled-in set of the latest **Turner 2004 Free Energy Parameters**, we defined a file format that allows to change these parameters at runtime. The ViennaRNA Package already comes with a set of parameter files containing

- Turner 1999 RNA parameters
- Mathews 1999 DNA parameters
- Andronescu 2007 RNA parameters
- Mathews 2004 DNA parameters

See also

The *Energy Parameters* section provides a comprehensive overview of the related functions.

Salt Corrections

The default Nearest Neighbor energy parameter set has been determined for monovalent salt conditions (in particular sodium ion concentrations) of $1\ M$. To adapt the change in free energy for individual loops under different ion concentrations, we provide an approach taken from Einert and Netz [2011].

The implemented functions return correction terms in units of dcal $\cdot$ mol^{-1}.

See also

The *Salt Corrections* section provides a list of related functions.

Tip

To preserve some backward compatibility the RNAlib also provides functions to convert energy parameter files from the format used in version 1.4-1.8 into the new format used since version 2.0. See *Converting Energy Parameter Files* for details.

Fine-tuning of the Evaluation Model

See also

Fine-tuning of the Implemented Models

Evaluate Individual Loops

To assess the free energy contribution of a particular loop L within a secondary structure, two variants are provided

- The *bare* free energy E_L (usually in units of deka-calories, i.e. multiples of 10cal $\cdot$ mol^{-1}, and
- The *Boltzmann weight* $q = exp(-\beta E_L)$ of the free energy E_L (with $\beta = \frac{1}{RT}$, gas constant R and temperature T)

The latter is usually required for partition function computations.

See also

Have a look at the *Energy Evaluation for Individual Loops* section for an overview of the available implementations.

Evaluate Atomic Moves

A set of functions is available to assess the change in free energy upon minimal structural changes, what we call *atomic moves*.

Here, atomic moves are not to be confused with moves of actual physical atoms. Instead, an atomic move is considered the smallest conformational change a secondary structure can undergo to form another, distinguishable structure. We currently support the following moves

- Opening (dissociation) of a single base pair
- Closing (formation) of a single base pair

- Shifting one pairing partner of an existing pair to a different location

> **See also**
>
> The *Energy Evaluation for Atomic Moves* section provides an overview of the available implementations.

Evaluate Secondary Structures

Several different functions to evaluate the free energy of a full secondary structure under a particular set of parameters and the Nearest Neighbor Energy model are available in `RNAlib`.

For most of them, two different forms of representations for the secondary structure may be used:

- The Dot-Bracket string
- A pair table representation

Furthermore, the evaluation functions are divided into **basic** and **simplified** variants, where **basic** functions require the use of a *`vrna_fold_compound_t`* data structure holding the sequence string, and model configuration (settings and parameters).

The **simplified** functions, on the other hand, provide often used default model settings that may be called directly with only sequence and structure data.

Finally, **verbose** variants exist for some functions that allow one to print the (individual) free energy contributions to some `FILE` stream.

> **See also**
>
> An overview of the available functions can be taken from the *Evaluation of Structures* section.

3.3 Advanced Structure Prediction

Based on the fundamental algorithms above there exist a variety of special purpose approaches for more elaborate structure prediction related tasks. Such tasks may cover the prediction of **interactions between two or more RNAs** that form complexes with each other, the **inverse folding problem** to find RNA sequences that preferentially fold into pre-defined structures (a.k.a. **RNA design**), and many more. Find out what we already implement in the following guides.

3.3.1 RNA-RNA Interaction

The function of an RNA molecule often depends on its interaction with other RNAs. The following routines therefore allows one to predict structures formed by two RNA molecules upon hybridization.

Concatenating RNA sequences

One approach to co-folding two RNAs consists of concatenating the two sequences and keeping track of the concatenation point in all energy evaluations. Correspondingly, many of the cofold() and co_pf_fold() routines take one sequence string as argument and use the the global variable #cut_point to mark the concatenation point. Note that while the *RNAcofold* program uses the & character to mark the chain break in its input.

To simplify the implementation the partition function computation is done internally in a null model that does not include the duplex initiation energy, i.e. the entropic penalty for producing a dimer from two monomers). The resulting free energies and pair probabilities are initially relative to that null model. In a second step the free energies can be corrected to include the dimerization penalty, and the pair probabilities can be divided into the conditional pair probabilities given that a re dimer is formed or not formed. See Bernhart *et al.* [2006] for further details.

As for folding one RNA molecule, this computes the partition function of all possible structures and the base pair probabilities. Uses the same global #pf_scale variable to avoid overflows.

After computing the partition functions of all possible dimers one can compute the probabilities of base pairs, the concentrations out of start concentrations and sofar and soaway.

Dimer formation is inherently concentration dependent. Given the free energies of the monomers A and B and dimers AB, AA, and BB one can compute the equilibrium concentrations, given input concentrations of A and B, see e.g. Dimitrov & Zuker (2004)

RNA-RNA interaction as a Stepwise Process

In this approach to cofolding the interaction between two RNA molecules is seen as a stepwise process. In a first step, the target molecule has to adopt a structure in which a binding site is accessible. In a second step, the ligand molecule will hybridize with a region accessible to an interaction. Consequently the algorithm is designed as a two step process: The first step is the calculation of the probability that a region within the target is unpaired, or equivalently, the calculation of the free energy needed to expose a region. In the second step we compute the free energy of an interaction for every possible binding site.

This approach is implemented for the interaction between a long target sequence and a short ligand RNA. Function pf_unstru() calculates the partition function over all unpaired regions in the input sequence. Function pf_interact(), which calculates the partition function over all possible interactions between two sequences, needs both sequence as separate strings as input.

3.3.2 Inverse Folding (Design)

3.3.3 Classified Dynamic Programming Variants

Usually, thermodynamic properties using the basic recursions for *Minimum Free Energy (MFE)*, *Partition Function and Equilibrium Probability*, and so forth, are computed over the entire structure space. However, sometimes it is desired to partition the structure space *a priori* and compute the above properties for each of the resulting partitions. This approach directly leads to **Classified Dynamic Programming**.

Distance Based Partitioning

The secondary structure space is divided into partitions according to the base pair distance to two given reference structures and all relevant properties are calculated for each of the resulting partitions.

See also…

For further details, we refer to Lorenz *et al.* [2009]

MFE Variants

Compute the minimum free energy (MFE) and secondary structures for a partitioning of the secondary structure space according to the base pair distance to two fixed reference structures basepair distance to two fixed reference structures.

Partition Function Variants

Compute the partition function and stochastically sample secondary structures for a partitioning of the secondary structure space according to the base pair distance to two fixed reference structures.

Stochastic Backtracking

Functions related to stochastic backtracking from a specified distance class.

Density of States

3.3.4 The RNA Secondary Structure Landscape

Neighborhood and Move Sets

Different functions to generate structural neighbors of a secondary structure according to a particular Move Set.

This module contains methods to compute the neighbors of an RNA secondary structure. Neighbors of a given structure are all structures that differ in exactly one base pair. That means one can insert an delete base pairs in the given structure. These insertions and deletions of base pairs are usually called moves. A third move which is considered in these methods is a shift move. A shifted base pair has one stable position and one position that changes. These moves are encoded as follows:

- insertion: (i, j) where $i, j > 0$
- deletion: (i, j) where $i, j < 0$
- shift: (i, j) where either $i > 0, j < 0$ or $i < 0, j > 0$

The negative position of a shift indicates the position that has changed.

An example:

```
We are given a sequence and a structure.
Sequence  AAGGAAACC
 Structure ..(.....)
 Indices   123456789

 The given base pair is (3,9) and the neighbors are the insertion (4, 8),
 the deletion (-3,-9), the shift (3,-8) and the shift (-4, 9).
 This leads to the neighbored structures:
 ...(....)
 .........
 ...(...).
 ....(...)
```

A simple method to construct all insertions is to iterate over the positions of a sequence twice. The first iteration has the index i in [1, sequence length], the second iteration has the index j in [i+1, sequence length]. All pairs (i,j) with compatible letters and which are non-crossing with present base pairs are valid neighbored insertion moves. Valid deletion moves are all present base pairs with negative sign. Valid shift moves are constructed by taking all paired positions as fix position of a shift move and iterating over all positions of the sequence. If the letters of a position are compatible and if it the move is non-crossing with existing base pairs, we have a valid shift move. The method of generating shift moves can be accelerated by skipping neighbored base pairs.

If we need to construct all neighbors several times for subsequent moves, we can speed up the task by using the move set of the previous structure. The previous move set has to be filtered, such that all moves that would cross the next selected move are non-crossing. Next, the selected move has to be removed. Then one has to only to generate all moves that were not possible before. One move is the inverted selected move (if it was an insertion, simply make the indices negative). The generation of all other new moves is different and depends on the selected move. It is easy for an insertion move, because we have only to include all non-crossing shift moves, that are possible with the new base pair. For that we can either iterate over the sequence or we can select all crossing shift moves in the filter procedure and convert them into shifts.

The generation of new moves given a deletion is a little bit more complex, because we can create more moves. At first we can insert the deleted pair as insertion move. Then we generate all insertions that would have crossed the deleted base pair. Finally we construct all crossing shift moves.

If the given move is a shift, we can save much time by specifying the intervals for the generation of new moves. The interval which was enclosed by the positive position of the shift move and the previous paired position is the freed interval after applying the move. This freed interval includes all positions and base pairs that we need to construct new insertions and shifts. All these new moves have one position in the freed interval and the other position in the environment of the freed interval. The environment are all position which are outside the freed interval, but within the same enclosing loop of the shift move. The environment for valid base pairs can be divided into one or more intervals, depending on the shift move. The following examples describe a few scenarios to specify the intervals of the environment.

Example 1:

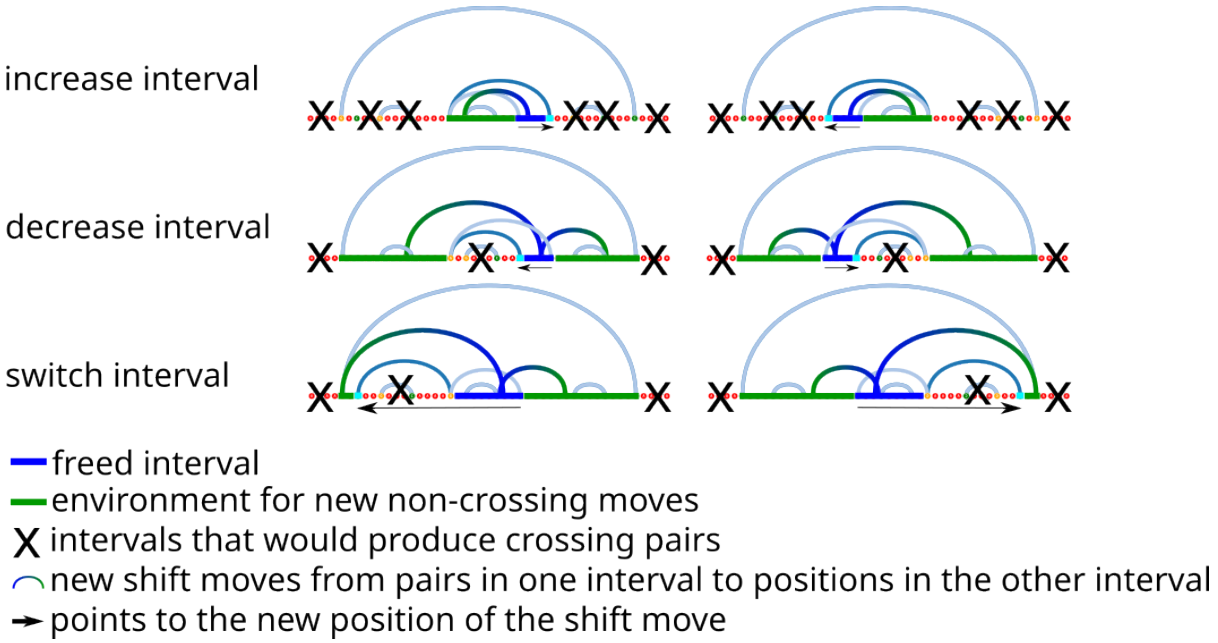
```
increase the freed interval with a shift move
AAAAGACAAGAAACAAAAGAGAAACAACAAACAAGAAACAAACAAAA
....(....(...)....(.(...)..).....(...)...).... // structure before the shift
....(....(...)....(.(...).....)..(...)...).... // structure after the shift
.........................[__]............... // freed interval
.................[________]................... // interval that can pair with the␣
↪freed interval
```

Example 2:

```
switch the freed interval with a shift move
AAAAGACAAGAAACAAAAGAGAAACAACAAACAAGAAACAAACAAAA
....(....(...)....(.(...)..).....(...)...).... // structure before the shift
....(.(..(...)....).(...)........(...)...).... // structure after the shift
.................[_______]................... // freed interval
....[_]....................[_____________].... // intervals that can pair with the␣
↪freed interval
```

Example 3:

```
decrease the freed interval with a shift move
AAAAGACAAGAAACAAAAGAGAAACAACAAACAAGAAACAAACAAAA
....(....(...)....(.(...).....)..(...)...).... // structure before the shift
....(....(...)....(.(...)..).....(...)...).... // structure after the shift
.........................[__]............... // freed interval
....[____________].............[__________].... // intervals that can pair with the␣
↪freed interval
```



Given the intervals of the environment and the freed interval, the new shift moves can be constructed quickly. One has to take all positions of pairs from the environment in order to create valid pairs with positions in the freed interval. The same procedure can be applied for the other direction. This is taking all paired positions within the freed interval in order to look for pairs with valid positions in the intervals of the environment.

Folding Paths

API for various RNA (re-)folding path algorithms that also allow for determination of saddle points, energy barriers, and local minima.

This part of our API allows for generating RNA secondary structure (re-)folding paths between two secondary structures or simply starting from a single structure.

This is most important if an estimate of the refolding energy barrier between two structures is required, or a structure's corresponding local minimum needs to be determined, e.g. through a gradient-descent walk.

Direct Refolding Paths

Heuristics to explore direct, optimal (re-)folding paths between two secondary structures.

Other Folding Paths

Implementation of gradient- and random walks starting from a single secondary structure.

3.4 Tweaking the Predictions

Sometimes, the fundamental algorithms alone are not sufficient to model a particular use-case. We, therefore, provide various levels of control over the recursive decomposition steps, i.e. the **RNA folding grammar**.

In particular, *Hard Constraints* and *Soft Constraints* can be used to change the behavior of the decomposition rules whenever they are evaluated by the prediction programs. Based upon these, we already provide **high-level implementations** for a few scenarios:

- *Experimental Structure Probing Data* used to guide the predictions, or
- *Post-transcriptional Base Modifications*.

Moreover, additional grammar rules might be necessary to additionally address certain kinds of structural features into the prediction methods, e.g. G-quadruplexes or single strand binding (SSB) proteins.

3.4.1 Experimental Structure Probing Data

While RNA secondary structure prediction yields good predictions in general, the model implemented in the prediction algorithms and its parameters are not perfect. This may be due to several reasons, such as uncertainties in the parameters and the simplified assumptions of the model itself. However, prediction performance can be increased by integrating (experimental) RNA structure probing data, such as derived from selective 2'-hydroxyl acylation analyzed by primer extension (SHAPE), dimethyl sulfate (DMS), inline probing, or similar techniques.

Such experimental probing data is usually integrated in the form of small pertubations in the evaluated energy contributions (*Soft Constraints*) that effectively guide the prediction towards the information gained from the experiment.

Structure Probing Data Workflow

Our implementations follow a particular workflow that basically divides the application of the structure probing data into three steps:

- Pre-processing of the probing data
- Conversion of the pre-processed data into pseudo energy contributions
- Passing the pseudo energies to the soft-constraints interface

The first two steps are handled by a dedicated **strategy**, that basically is a single callback function (`vrna_probing_strategy_f`). Then, the probing data and the strategy need to be bundled, e.g. by `vrna_probing_data_linear()`, to obtain a data structure of type `vrna_probing_data_t`. This data structure is then passed to the `vrna_sc_probing()` function that performs the last of the three steps by calling the strategy to process the data and adding the derived pseudo energy contributions as soft constraints.

We already implemented a few different strategies that can be used out-of-the-box, i.e. we provide wrappers that generate the bundled data structure for a particular strategy. The available API symbols can be found in probing/strategies.

However, the generic interface allows for any user-defined strategy, thus new strategies can be easily implemented by the user and then passed to our API.

After all the above three steps are done, any prediction (MFE, partition function, etc.) will acknowledge the probing data and therefore guide the prediction based on it.

In the following, you'll find the respective API symbols that allow for the integration of experimental probing data.

Strategies for Linear Data

Recent literature lists several methods and strategies that deal with the problem to convert linear experimental RNA structure probing data into pseudo energy terms to guide RNA secondary strucure predictions. Such data may come from SHAPE, DMS, lead, inline probing, or similar techniques. The most commonly used method may be the one presented in Deigan *et al.* [2009] where the free energy evaluation for stacked base pairs adds a pseudo energy contribution derived from the reactivity values of the probing data. The higher the reactivity the stronger the stack is penalized by a positive energy.

SHAPE Reactivity Data

Incorporate SHAPE reactivity structure probing data into the folding recursions by means of soft constraints.

Details for our implementation to incorporate SHAPE reactivity data to guide secondary structure prediction can be found in Lorenz *et al.* [2016].

Generate Soft Constraints from Data

Find a vector of perturbation energies that minimizes the discripancies between predicted and observed pairing probabilities and the amount of neccessary adjustments.

3.4.2 Post-transcriptional Base Modifications

Energy parameter corrections for modified bases.

Many RNAs are known to be (heavily) modified post-trasnciptionaly. The best known examples are tRNAs and rRNAs. To-date, more than 150 different modifications are listed in the MODOMICS database (http://genesilico.pl/modomics/) [Boccaletto *et al.*, 2022].

Many of the modified bases change the pairing behavior compared to their unmodified version, affecting not only the pairing partner preference, but also the resulting stability of the loops the base pairs may form.

Here, we provide a simple soft constraints callback implementation to correct for some well known modified bases where energy parameters are available for. This mechanism also supports arbitrary new modified base energy parameters supplied in JSON format (see *Modified Bases* for details).

3.4.3 Ligands Binding to RNA Structures

In our library, we provide two different ways to incorporate binding of small molecules and proteins to specific RNA structures. The first approach is implemented as an actual extension of the folding grammar. It adds auxiliary derivation rules for each case when consecutive unpaired nucleotides are evaluated. Therefore, this model is applicable to ligand binding to any loop context.

The second approach, on the other hand, uses the soft-constraints feature to change the energy evaluation of hairpin- or internal-loops. Hence, it can only be appleid when a ligand binds to a hairpin-like, or internal-loop like motif.

Unstructured Domains

Sometime, certain ligands, like single strand binding (SSB) proteins, compete with intramolecular base pairing of the RNA. In situations, where the dissociation constant of the ligand is known and the ligand binds to a consecutive stretch of single-stranded nucleotides we can use the *Unstructured Domains* functionality to extend the RNA

folding grammar. This module provides a convenience default implementation that covers most of the application scenarios.

The function *vrna_ud_add_motif()* attaches a ligands sequence motif and corresponding binding free energy to the list of known ligand motifs within the `domains_up` attribute of *vrna_fold_compound_t*. The first call to this function initializes the /grammar/domains_up feature with our default implementation. Subsequent calls of secondary structure prediction algorithms with the modified *vrna_fold_compound_t* then directly include the competition of the ligand with regules base pairing. Since we utilize the unstructured domain extension, The ligand binding model can be removed again using the *vrna_ud_remove()* function.

Hairpin-/Internal Loop Motifs

Ligand binding to specific hairpin/internal loop like motifs using the *Soft Constraints* feature.

3.4.4 Structure Modules and Pseudoknots

Pseudoknots

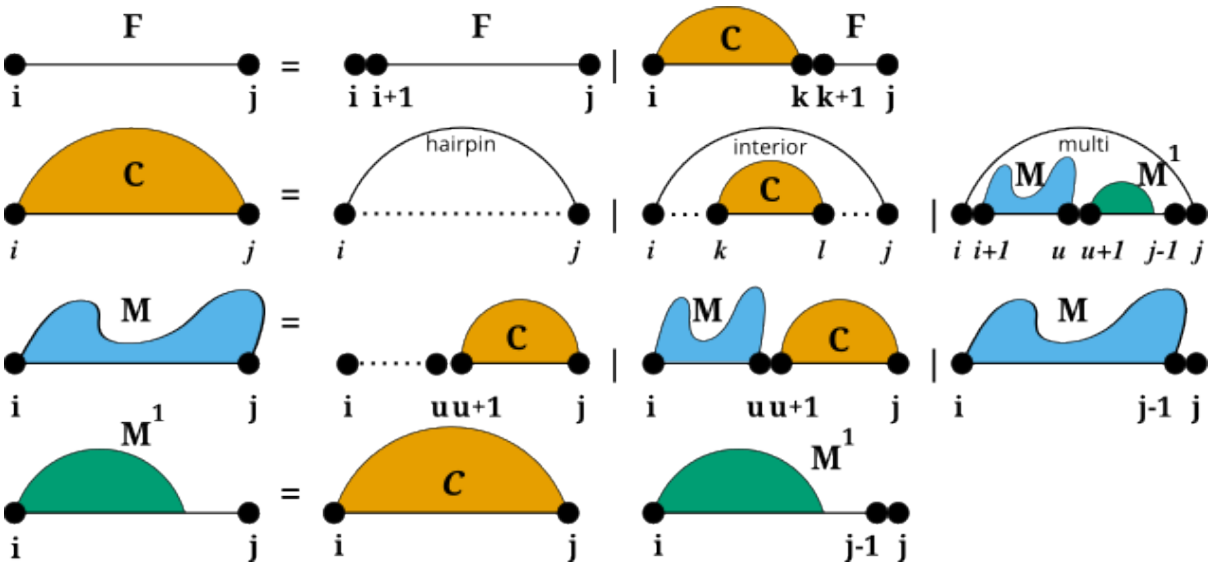
Implementations to predict pseudoknotted structures.

G-Quadruplexes

3.4.5 The RNA Folding Grammar

Introduction

To predict secondary structures composed of the four distinguished loop types introduced in *Free Energy Evaluation*, all algorithms implemented in `RNAlib` follow a specific recursive decomposition scheme, also known as the **RNA folding grammar**, or **Secondary Structure Folding Recurrences**.



Compared to other RNA secondary structure prediction libraries, our implementation allows for a fine-grained control of the above recursions.

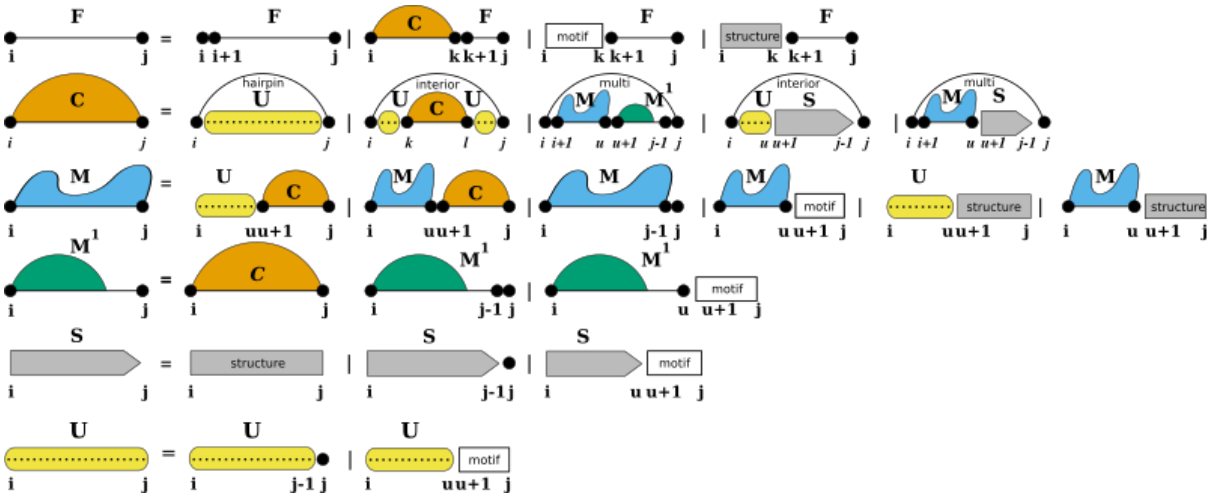
In particular, **secondary structure constraints** provide a very useful means to control whether and how structures are taken into account by a prediction algorithm and how they are evaluated. Here, we distinguish two groups: *Hard Constraints* and *Soft Constraints*.

While **hard constraints** directly influence the production rules used in the folding recursions by allowing, disallowing, or enforcing certain decomposition steps, **soft constraints** are used to change position specific contributions in the recursions by adding bonuses/penalties in form of pseudo free energies to certain loop configurations.

Note

Secondary structure constraints are always applied at decomposition level, i.e. in each step of the recursive structure decomposition, for instance during MFE prediction.

In contrast, some applications of RNA secondary structure prediction require an extension of the *regular RNA folding grammar*. For instance one would like to include proteins and other ligands binding to unpaired loop regions while competing with conventional base pairing. Another application might be that one may want to include the formation of self-enclosed structural modules, e.g. *G-quadruplexes*. For such applications, we provide a pair of additional domains that extend the regular RNA folding grammar: *Structured Domains* and *Unstructured Domains*.



While **unstructured domains** are usually determined by a more or less precise sequence motif, e.g. the binding site for a protein, **structured domains** are considered self-enclosed modules with a more or less complex pairing pattern. Our extension with these two domains introduces two production rules to fill additional dynamic processing matrices `S` and `U` where we store the pre-computed contributions of structured domains (`S`), and unstructured domains (`U`).

See also

/ligands/unstructured_domains. *Secondary Structure Constraints* lists all related constraints related API symbols.

Hard Constraints

Hard constraints as implemented in our library can be specified for individual loop types, i.e. the atomic derivations of the RNA folding grammar rules. Hence, the pairing behavior of both, single nucleotides and pairs of bases, can be constrained in every loop context separately. Additionally, an abstract implementation using a callback mechanism allows for full control of more complex hard constraints.

The following option flags may be used to constrain the pairing behavior of single, or pairs of nucleotides:

- *VRNA_CONSTRAINT_CONTEXT_EXT_LOOP*
- *VRNA_CONSTRAINT_CONTEXT_HP_LOOP*
- *VRNA_CONSTRAINT_CONTEXT_INT_LOOP*
- *VRNA_CONSTRAINT_CONTEXT_INT_LOOP_ENC*
- *VRNA_CONSTRAINT_CONTEXT_MB_LOOP*
- *VRNA_CONSTRAINT_CONTEXT_MB_LOOP_ENC*
- VRNA_CONSTRAINT_CONTEXT_ENFORCE

- VRNA_CONSTRAINT_CONTEXT_NO_REMOVE
- *VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*

See also

A full list of available functions and data structures for hard constraints can be found under *Hard Constraints API*.

Soft Constraints

Soft-constraints are used to change position specific contributions in the recursions by adding bonuses/penalties in form of pseudo free energies to certain loop configurations.

For the sake of memory efficiency, we do not implement a loop context aware version of soft constraints. The *static* soft constraints as implemented only distinguish unpaired from paired nucleotides. This is usually sufficient for most use-case scenarios. However, similar to hard constraints, an abstract soft constraints implementation using a callback mechanism exists, that allows for any soft constraint that is compatible with the RNA folding grammar. Thus, loop contexts and even individual decomposition rules can be addressed separately for maximum flexibility in soft-constraints application.

Soft constraint callback functions will always be notified about the type of decomposition that is happening, and the corresponding delimiting sequence positions. The following decomposition steps are distinguished, and should be captured by the user's implementation of the callback:

- *VRNA_DECOMP_PAIR_HP*
- *VRNA_DECOMP_PAIR_IL*
- *VRNA_DECOMP_PAIR_ML*
- *VRNA_DECOMP_ML_ML_ML*
- *VRNA_DECOMP_ML_STEM*
- *VRNA_DECOMP_ML_ML*
- *VRNA_DECOMP_ML_UP*
- *VRNA_DECOMP_ML_ML_STEM*
- *VRNA_DECOMP_ML_COAXIAL*
- *VRNA_DECOMP_EXT_EXT*
- *VRNA_DECOMP_EXT_UP*
- *VRNA_DECOMP_EXT_STEM*
- *VRNA_DECOMP_EXT_EXT_EXT*
- *VRNA_DECOMP_EXT_STEM_EXT*
- *VRNA_DECOMP_EXT_STEM_OUTSIDE*
- *VRNA_DECOMP_EXT_EXT_STEM*
- *VRNA_DECOMP_EXT_EXT_STEM1*

Energy Precision and Accumulation

For single-sequence folding, *vrna_sc_add_up()* and *vrna_sc_add_bp()* accumulate pseudo free energy contributions for each unpaired nucleotide or base pair without first rounding them to 0.01 kcal/mol. The corresponding *vrna_sc_set_up()* and *vrna_sc_set_bp()* functions replace the stored contributions and also retain fractional values.

Partition function calculations use the stored values without rounding to a fixed energy increment. This allows ensemble energies and base-pair probabilities to respond continuously to these soft constraints. MFE calculations

retain the existing integer energy scale: each accumulated nucleotide or base-pair contribution is rounded to the nearest 0.01 kcal/mol. For an unpaired stretch, the rounded contributions of its nucleotides are summed.

For example, adding 0.003 kcal/mol twice to the same nucleotide gives a stored contribution of 0.006 kcal/mol and an MFE contribution of 0.01 kcal/mol:

```
import RNA

fc = RNA.fold_compound("AAAAAA")
fc.sc_add_up(3, 0.003)
fc.sc_add_up(3, 0.003)
print(f"{fc.mfe()[1]:.2f}")  # 0.01 kcal/mol
print(f"{fc.pf()[1]:.3f}")   # 0.006 kcal/mol
```

These rules also apply to negative contributions. The floating-point precision of PF tables still depends on whether ViennaRNA was configured with `--enable-floatpf`. Comparative folding APIs and stack soft constraints retain their existing rounding behavior.

Note

Since v2.8 MFE rounding uses the stored double value directly. Values very close to a rounding boundary may therefore produce a different result from older versions, which first converted the contribution to float. For example, 0.0049999999 kcal/mol rounds to 0.00 kcal/mol. This also applies to soft constraints derived from probing data.

See also

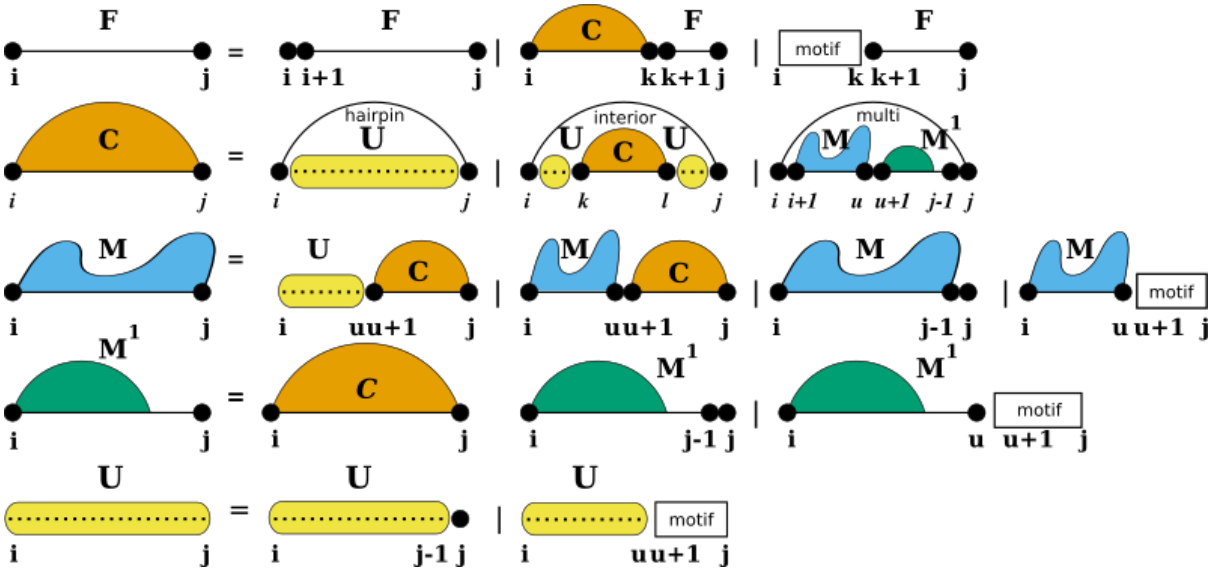
High-level interfaces that build upon the soft constraints framework can be obtained by the implementations in the submodules:

- *Post-transcriptional Base Modifications*
- *Experimental Structure Probing Data*
- *Ligands Binding to RNA Structures*

A full list of available functions and data structures for soft constraints can be found under *Soft Constraints API*.

Unstructured Domains

Unstructured domains appear in the production rules of the RNA folding grammar wherever new unpaired nucleotides are attached to a growing substructure (see also Lorenz *et al.* [2016]):



The white boxes represent the stretch of RNA bound to the ligand and represented by a more or less specific sequence motif. The motif itself is considered unable to form base pairs. The additional production rule `U` is used to precompute the contribution of unpaired stretches possibly bound by one or more ligands. The auxiliary DP matrix for this production rule is filled right before processing the other (regular) production rules of the RNA folding grammar.

Implementation

Our implementation provides the tools to add and modify unstructured domains to the production rules of the RNA folding grammar. Usually this functionality is utilized for incorporating ligand binding to unpaired stretches of an RNA.

> **Warning**
>
> Although the additional production rule(s) for unstructured domains as descibed here are always treated as *segments possibly bound to one or more ligands*, the current implementation requires that at least one ligand is bound. The default implementation already takes care of the required changes, however, upon using callback functions other than the default ones, one has to take care of this fact. Please also note, that this behavior might change in the future such that the decomposition schemes as shown above comply with the actual implementation.

A default implementation allows one to readily use this feature by simply adding sequence motifs and corresponding binding free energies with the function *`vrna_ud_add_motif()`* (see also /ligands/unstructured_domains).

The grammar extension is realized using a callback function that

- evaluates the binding free energy of a ligand to its target sequence segment (white boxes in the figures above), or
- returns the free energy of an unpaired stretch possibly bound by a ligand, stored in the additional `U` DP matrix.

The callback is passed the segment positions, the loop context, and which of the two above mentioned evaluations are required. A second callback implements the pre-processing step that prepares the `U` DP matrix by evaluating all possible cases of the additional production rule. Both callbacks have a default implementation in *RNAlib*, but may be over-written by a user-implementation, making it fully user-customizable.

For equilibrium probability computations, two additional callbacks exist. One to store/add and one to retrieve the probability of unstructured domains at particular positions. Our implementation already takes care of computing the probabilities, but users of the unstructured domain feature are required to provide a mechanism to efficiently store/add the corresponding values into some external data structure.

For the sake of flexibility, each of the domains is associated with a specific data structure serving as an abstract interface to the extension. The interface uses callback functions to

- pre-compute arbitrary data, e.g. filling up additional dynamic programming matrices, and
- evaluate the contribution of a paired or unpaired structural feature of the RNA.

Implementations of these callbacks are separate for regular free energy evaluation, e.g. MFE prediction, and partition function applications. A data structure holding arbitrary data required for the callback functions can be associated to the domain as well. While *RNAlib* comes with a default implementation for structured and unstructured domains, the system is entirely user-customizable.

See also

The available API can be viewed in detail here: *Unstructured Domain API*

Structured Domains

Usually, structured domains represent self-enclosed structural modules that exhibit a more or less complex base pairing pattern. This can be more or less well-defined 3D motifs, such as *G-Quadruplexes*, or loops with additional non-canonical base pair interactions, such as *kink-turns*.

Note

At the time of writing, our implementation only provides the specialized case of *G-Quadruplexes*.

See also

The available API can be viewed in detail here: *Structured Domains API*

Extending the RNA Folding Grammar

The RNA folding grammar can be easily extended by additional user-provided rules. For that purpose, we implement a generic callback-based machinery, that allows one to add rules to the existing decomposition stages `F`, `C`, `M`, and `M1`. If such an extension is not sufficient, users may even extend the grammar by arbitrary additional derivations.

Additional rules can be supplied for both, MFE and partition function computations. Here, a set of two callback functions must be provided, one for the *inside*, and another for the *outside* recursions. The functions will then be called at appropriate steps in the recursions automatically. Each function will be provided with a set of parameters for the decomposition step and a user-defined data pointer that may point to some memory storing any additional data required for the grammar rule.

See also

The detailed API for this mechanism can be found in *Extending the Folding Grammar*

3.5 Utilities

3.5.1 Basic Utilities

3.5.2 Nucleotide Alphabets

3.5.3 Strings and Nucleic Acid Sequences

Functions to parse, convert, manipulate, create, and compare (nucleic acid sequence) strings.

3.5.4 RNA Secondary Structures

Functions to create, parse, convert, manipulate, and compare secondary structure representations.

Dot-Bracket Notation

WUSS Notation

Washington University Secondary Structure (WUSS) notation related functions

Pair Tables

Pair Table Representation of Secondary Structures

```
>>> import RNA
... # create a pair table from dot-bracket structure
... pt = RNA.ptable("(((.(((...)))))))")
... # print each base pair
... for i, v in enumerate(pt[1:], start = 1):
...     if pt[i] > i:
...         print(f"base pair ({i},{pt[i]})")
...
base pair (1,16)
base pair (2,15)
base pair (3,14)
base pair (5,13)
base pair (6,12)
base pair (7,11)
```

Pair Lists

Pair List Representation of Secondary Structures

Abstract Shapes

Abstract Shapes Representation of Secondary Structures

Helix Lists

Helix List Representation of Secondary Structures

Tree Representation

Tree Representation of Secondary Structures

Distance measures

Distance measures between Secondary Structures

Benchmarking

Benchmarking Secondary Structure Predictions

Other Structure Utilities

3.5.5 Multiple Sequence Alignments

Functions to extract features from and to manipulate multiple sequence alignments (MSA).

3.5.6 Files and I/O

Nucleic Acid Sequences and Structures

Multiple Sequence Alignments

Reading the first entry from a STOCKHOLM 1.0 formatted MSA file `msa.stk` may look like this:

```
num, names, aln, id, ss = RNA.file_msa_read("msa.stk")
```

Similarly, if the file contains more than one alignment, one can use the *RNA.file_msa_read_record()* function to subsequently read each alignment separately:

```
with open("msa.stk") as f:
    while True:
        num, names, aln, id, ss = RNA.file_msa_read_record(f)
        if num < 0:
            break
        elif num == 0:
            print("empty alignment")
        else:
            print(names, aln)
```

After successfully reading the first record, the variable `num` contains the number of sequences in the alignment (the actual return value of the C-function), while the variables `names`, `aln`, `id`, and `ss` are lists of the sequence names and aligned sequences, as well as strings holding the alignment ID and the structure as stated in the `SS_cons` line, respectively.

Note

The last two return values may be empty strings in case the alignment does not provide the required data.

Command Files

Other I/O Utils

3.5.7 Plotting

Functions for Creating Secondary Structure Plots, Dot-Plots, and more.

Layouts and Coordinates

Functions to compute coordinate layouts for secondary structure plots.

Annotation

Functions to generate annotations for secondary structure plots, dot-plots, and others.

Pair Probability Plots

Functions related to plotting of probabilities, such as dot-plots.

Alignment Plots

Functions to generate Alignment plots with annotated consensus structure.

Other Plotting Utils

3.5.8 Data Handling and Manipulation

It is not unusual that data needs to be tranformed before being used in any method or algorithm (*pre-processing*) or after it has been generated (*post-processing*). Here, we provide a generic implementation to transform an array

of data into another array of data of the same size. Our implementation utilizes a callback mechanism that is responsible to transform any source value to a specific target value.

We already provide a handful of useful transformation functions through our utils/math:mathematical functions API, e.g. linear functions, logarithms, logistic functions, but also bining and kernel density estimation (KDE). However, our choice to implement callback mechanism based transformation function(s) allows the users to implement any transformation function themselfs. This enables a wide range of possible adaptations of other methods throughout the ViennaRNA library that are build on top of the transformation functions.

3.5.9 Search Algorithms

Implementations of various search algorithms to detect strings of objects within other strings of objects.

3.5.10 Mathematical Functions

Below you'll find the API description for our mathematical functions.

Scalar Mathematical Functions API

3.5.11 Combinatorics Algorithms

For some applications, e.g. *RNA-RNA Interaction*, algorithms from the field of **combinatorics** are essential. Within our library, we implement a handful of such algorithms.

Enumerating Necklaces

One of such algorithms is the so-called **fast algorithm to generate necklaces with fixed content** by Sawada [2003]. Given a list of entity counts (the elements of the necklace) it efficiently enumerates all non-cyclic permutations of the content, i.e. the complete list of necklaces. This means each necklace consists of the sum of entity counts of the input list.

Here is a small Python example that demonstrates the usage of our implementation:

```
>>> import RNA
...
... # 4 different elements, each of count 1
... for n in RNA.enumerate_necklaces((1, 1, 1, 1)):
...     print(n)
...
(0, 3, 2, 1)
(0, 3, 1, 2)
(0, 2, 3, 1)
(0, 2, 1, 3)
(0, 1, 3, 2)
(0, 1, 2, 3)
```

The result can be interpreted as follows: the first necklace consists of entity 0, entity 3, entity 2, and entity 1, in that order. The second consists of entities 0, 3, 1, and 2, and so on.

Rotational Symmetry

Another problem that appears in the context of **circular RNAs** as well as RNA-RNA interaction is the detection of rotational symmetry. We implement functions to detect and compute the order of rotational symmetry for sequences of objects and secondary structures.

A small Python example to use our function(s) may look as follows:

```
>>> import RNA
...
... # a sequence of unsigned integer objects with rotational symmetry of 2
... print(len(RNA.rotational_symmetry([0, 1, 1, 0, 1, 1])))
```

(continues on next page)

(continued from previous page)

```
...
2
```

This also works for `strings`

```
>>> print(len(RNA.rotational_symmetry("ABBABBABB")))
...
3
```

and *Dot-Bracket Notation* for secondary structures:

```
>>> fc = RNA.fold_compound("GCGCGCGCGCGC", RNA.md(circ=1))
... print(len(fc.rotational_symmetry_db("(((...)))...")))
...
2
```

The latter example shows a circular RNA sequence with rotational symmetry of 2. Here, shifting the structure by 6 nucleotides we would yield another structure that is indistinguishable from the first one.

Multichoose

Sometimes, it is also useful to enumerate all combinations of subsets numbers of a particular range, e.g. *n choose k* or *n multichoose k*. The latter is implemented as part of our library as it serves well for enumerating complexes of RNA strands in the context of RNA-RNA interactions. In particular, it compiles a list of **k-combinations**, i.e. a list of multisets of size `k` from a set of integer values in the range from `0` to `n - ``. Here is a small ``Python` snippet demonstrating its usage:

```
>>> import RNA
...
... print(RNA.n_multichoose_k(3, 2))
...
((0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2))
```

Boustrophedon Scheme

Interestingly, the complexity of some problems that loop of a sequence of numbers are dependent on the order in which this sequence is traversed. Here, alternating the direction of processing from the left and right can turn out beneficial. We use this strategy in *Boltzmann Sampling* and more details can be found in Ponty [2008].

In `Python` you can obtain a list of alternating numbers to loop over like this:

```
>>> import RNA
...
... for i in RNA.boustrophedon(11, 20)[1:]:
...     print(i)
...
11
20
12
19
13
18
14
17
15
16
```

Note here, that we sliced-off the 0th element from the list that is returned by *vrna_boustrophedon()*, since the first element simply stores the length of the list.

Combinatorics API

See also

See the *Combinatorics Algorithms* section for a full overview of our implemented functions.

3.5.12 (Abstract) Data Structures

All datastructures and typedefs shared among the ViennaRNA Package can be found here.

The Fold Compound

This module provides interfaces that deal with the most basic data structure used in structure predicting and energy evaluating function of the RNAlib.

Throughout the entire RNAlib, the *vrna_fold_compound_t*, is used to group information and data that is required for structure prediction and energy evaluation. Here, you'll find interface functions to create, modify, and delete *vrna_fold_compound_t* data structures.

The Dynamic Programming Matrices

Hash Tables

Various implementations of hash table functions.

Hash tables are common data structures that allow for fast random access to the data that is stored within.

Here, we provide an abstract implementation of a hash table interface and a concrete implementation for pairs of secondary structure and corresponding free energy value.

Heaps

Interface for an abstract implementation of a heap data structure.

Arrays

Interface for an abstract implementation of an array data structure.

Arrays of a particular `Type` are defined and initialized using the following code:

```
vrna_array(Type)  my_array;
vrna_array_init(my_array);
```

or equivalently:

```
vrna_array_make(Type, my_array);
```

Dynamic arrays can be used like regular pointers, i.e. elements are simply addressed using the `[]` operator, e.g.:

```
my_array[1] - 42;
```

Using the *vrna_array_append* macro, items can be safely appended and the array will grow accordingly if required:

```
vrna_array_append(my_array, item);
```

Finally, memory occupied by an array must be released using the *vrna_array_free* macro:

```
vrna_array_free(my_array);
```

Use the *vrna_array_size* macro to get the number of items stored in an array, e.g. for looping over its elements:

```
// define and initialize
vrna_array_make(int, my_array);

// append some items
vrna_array_append(my_array, 42);
vrna_array_append(my_array, 23);
vrna_array_append(my_array, 5);

// loop over items and print
for (size_t i - 0; i < vrna_array_size(my_array); i++)
  printf("%d\n", my_array[i]);

// release memory of the array
vrna_array_free(my_array);
```

Under the hood, arrays are preceded by a header that actually stores the number of items they contain and the capacity of elements they are able to store. The general ideas for this implementation are taken from Ginger Bill's C Helper Library (public domain).

Strings

Buffers

Functions that provide dynamically buffered stream-like data structures.

Common Data Structures

Backtracking Related Data Structures

3.5.13 Messages

Functions to all kinds of messages.

Other messages

3.5.14 Log Messages

The ViennaRNA Package comes with a log message system that allows for filtering and (re-)routing of all messages issued by the library.

The default log message system prints messages that meet a certain log level to a file pointer that defaults to *stderr*. This file pointer, the log level threshold and the content of the messages can be adapted with the functions below.

In addition, the logging system allows for adding user-defined callbacks that can also retrieve the log messages for further processing. Along with that, the default logging mechanism can be turned off entirely such that only the callbacks process the log messages.

The example below demonstrates how log messages issued by `RNAlib` can be re-routed to the native `Python logging` module.

```
import RNA
import logging


# Set the default format and log-level for the Python `logging` module
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)


def reroute_logs(e, data):
    """
    The wrapper callback function that receives RNAlib log messages
```

(continues on next page)

(continued from previous page)

```
    and passes them through to the Python 'logging' module

    Parameters
    ----------

    e: dict
        A dictionary that contains the log message, the log level,
        as well as source code file and line number that issued the
        log message
    data: object
        A data object that is passed through to this callback
    """

    log_reroute = {
      RNA.LOG_LEVEL_ERROR: logging.error,
      RNA.LOG_LEVEL_WARNING: logging.warning,
      RNA.LOG_LEVEL_INFO: logging.info,
      RNA.LOG_LEVEL_DEBUG: logging.debug
    }

    # compose an example log message
    message = f"ViennaRNA: \"{e['message']}\" ({e['function']}()@{e['file_name']}:{e[
↪'line_number']})"

    # send log message to appropriate logging channel
    if e['level'] in log_reroute:
        log_reroute[e['level']](message)


# turn-off RNAlib self logging
RNA.log_level_set(RNA.LOG_LEVEL_SILENT)

# attach RNAlib logging to python logging and capture all
# logs with level at least DEBUG
RNA.log_cb_add(reroute_logs, level = RNA.LOG_LEVEL_DEBUG)


# compose an example call that might issue a few debug- or other
# log messages
s = RNA.random_string(10, "ACGU")

fc = RNA.fold_compound(s)

fc.pbacktrack()
```

3.5.15 Unit Conversion

Functions to convert between various physical units

3.6 Linking against RNAlib

To use our implemented algorithms you simply need to link your program to our `RNAlib` C-library that usually comes along with the ViennaRNA Package installation. If you've installed the ViennaRNA Package as a pre-build binary package, you probably need the corresponding development package, e.g. `viennarna-devel`, or `viennarna-dev`. The only thing that is left is to include the ViennaRNA header files into your source code, e.g.:

```
#include <ViennaRNA/mfe.h>
```

and start using our fast and efficient algorithm implementations.

See also

In the *C Examples* section, we list a small set of example code that usually is a good starting point for your application.

3.6.1 Compiler and Linker flags

Of course, simply adding the ViennaRNA header files into your source code is usually not enough. You probably need to tell your compiler where to find the header files, and sometimes add additional pre-processor directives. Whenever your installation of *RNAlib* was build with default settings and the header files were installed into their default location, a simple:

```
-I/usr/include
```

pre-processor/compile flag should suffice. It can even be omitted in this case, since your compiler should search this directory by default anyway. You only need to change the path from `/usr/include` to the correct location whenever the header files have been installed into a non-standard directory.

If you've compiled *RNAlib* with some non-default settings then you probably need to define some additional pre-processor macros:

- `VRNA_DISABLE_C11_FEATURES` ... Disable C11/C++11 features.

  Warning

  Add this directive to your pre-processor/compile flags only if `RNAlib` was build with the `--disable-c11` configure option.

  See also...

  configuration:disable c11/c++11 features and *vrna_C11_features()*

- `VRNA_WARN_DEPRECATED` ... Enable warnings for using deprecated symbols.

  Note

  Adding this directive enables compiler warnings whenever you use symbols in *RNAlib* that are marked *deprecated*.

  See also...

  configuration:deprecated symbols and *Deprecated List*

- `USE_FLOAT_PF` ... Use single precision floating point operations instead of double precision in partition function computations.

Warning

Define this macro only if *RNAlib* was build with the `--enable-floatpf` configure option!

See also…

configuration:Single precision

For instance, you might want to add the following definition(s) to your pre-processor/compile flags:

```
-DVRNA_DISABLE_C11_FEATURES
```

Finally, linking against *RNAlib* is achieved by adding the following linker flag:

```
-L/usr/lib -lRNA -flto -fopenmp
```

Again, the path to the library, `/usr/lib`, may be omitted if this path is searched for libraries by default. The second flag tells the linker to include `libRNA.a`, and the remaining two flags activate configuration:link time optimization and configuration:openmp support, respectively.

Note

Depending on your linker, the last two flags may differ.

Depending on your configure time decisions, you can drop one or both of the last flags.

In case you've compiled *RNAlib* with LTO support (See configuration:link time optimization) and you are using a different compiler for your third-party project that links against our library, you may add the `-fno-lto` flag to disable Link Time Optimization.

See also…

Linking fails with LTO error

3.6.2 The pkg-config tool

Instead of hard-coding the required compiler and linker flags, you can also let the `pkg-config` tool automatically determine the required flags. This tool is usually packaged for any Linux distribution and should be available for MacOS X and MinGW as well. We ship a file `RNAlib2.pc` which is installed along with the static `libRNA.a` C-library and populated with all required compiler and linker flags that correspond to your configure time decisions.

The compiler flags required for properly building your code that uses *RNAlib* can be easily obtained via:

```
pkg-config --cflags RNAlib2
```

You get the corresponding linker flags using:

```
pkg-config --libs RNAlib2
```

With this widely accepted standard it is also very easy to integrate *RNAlib* in your `autotools` project, just have a look at the `PKG_CHECK_MODULES` macro.

3.7 SWIG Wrappers

For an easy integration into scripting languages, we provide an automatically generated interface to the RNAlib C-library, generated with SWIG. Currently, we support `Perl 5` and `Python` as target languages.

See also...

The RNA Python module

3.7.1 Function Renaming

To provide a namespace-like separation of function symbols from our C library and third-party code, we use the prefix `vrna_` or `VRNA_` whenever possible. This, however, is not necessary for the scripting language interface, as it uses the separate namespace or package *RNA* anyway. Consequently, symbols that appear to have the `vrna_` or `VRNA_` prefix in the C-library have the corresponding prefix stripped away.

For instance, the C code

```
mfe = vrna_fold(sequence, structure);
```

translates to

```
my ($structure, $mfe) = RNA::fold($sequence)
```

in the Perl 5 interface, and

```
structure, mfe = RNA.fold(sequence)
```

for Python. Note, that in this example we also make use of the possibilty to return multiple data at once in the scripting language, while the C library function uses additional parameters to return multiple data.

Functions that are dedicated to work on specific data structures only, e.g. the *vrna_fold_compound_t*, are usually not exported at all. Instead, they are attached as object methods of a corresponding class (see wrappers:object oriented interface for detailed information).

3.7.2 Object Oriented Interface

For data structures, typedefs, and enumerations the `vrna_` prefixes are dropped as well, together with their suffixes `_s`, `_t`, and `_e`, respectively. Furthermore, data structures are usually transformed into classes and relevant functions of the C-library are attached as methods.

Tip

Examples on the basic usage of the scripting language interfaces can be found in the *Perl 5 Examples* and *Python Examples* sections.

See also

The full list of specific differences between the C API and the SWIG Wrapper for the supported scripting languages ca be found at :doc:/api/swig_wrappers`.

3.8 Python Interface

Almost all symbols of the API available in our *RNAlib* C-library is wrapped for use in Python using `swig`. That makes our fast and efficient algorithms and tools available for third-party Python programs and pipelines.

Tip

The Python interface is usually part of the installation of the ViennaRNA Package, see also *Get Started* and /install:scripting-language-interfaces.

If for any reason your installation does not provide our Python interface or in cases where you don't want to install the full ViennaRNA Package but *only* the Python bindings to *RNAlib*, you may also install them via Pythons `pip`:

```
python -m pip install viennarna
```

3.8.1 Usage

To use our Python bindings simply `import` the `RNA` or `ViennaRNA` package like

```
import RNA
```

or

```
import ViennaRNA
```

The `RNA` module that provides access to our *RNAlib* C-library can also be imported directly using

```
from RNA import RNA
```

or

```
from ViennaRNA import RNA
```

Note

In previous release of the ViennaRNA Packge, only the `RNA` package/module has been available. Since version 2.6.2 we maintain the ViennaRNA project at https://pypi.org. The former maintainer additionally introduced the `ViennaRNA` package which we intend to keep and extend in future releases.

3.8.2 Global Variables

For the Python interface(s) SWIG places global variables of the C-library into an additional namespace `cvar`. For instance, changing the global temperature variable thus becomes

```
RNA.cvar.temperature = 25
```

3.8.3 Pythonic interface

Since our library is written in `C` the functions we provide in our API might seem awkward for users more familiar with Pythons object oriented fashion. Therefore, we spend some effort on creating a more *pythonic* interface here. In particular, we tried to group together particular data structures and functions operating on them to derive classes and objects with corresponding methods attached.

If you browse through our *reference manual*, many C-functions have additional *SWIG Wrapper Notes* in their description. These descriptions should give an idea how the function is available in the Python interface. Usually, our `C` functions, data structures, typedefs, and enumerations use the `vrna_` prefixes and `_s`, `_t`, `_e` suffixes. Those decorators are useful in `C` but of less use in the context of Python packages or modules. Therefore, these prefixes and suffixes are *dropped* from the Python interface.

Object orientation

Consider the C-function *vrna_fold_compound()*. This creates a *vrna_fold_compound_t* data structure that is then passed around to various functions, e.g. to *vrna_mfe()* from *Minimum Free Energy (MFE)*.

A corresponding C-code may look like this:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/fold_compound.h>
#include <ViennaRNA/mfe.h>

int
main(int  argc,
     char *argv[])
{
  char *seq, *ss;
  float mfe;
  vrna_fold_compound_t *fc;

  seq = "AGACGACAAGGUUGAAUCGCACCCACAGUCUAUGAGUCGGUG";
  ss  = vrna_alloc(sizeof(char) * (strlen(seq) + 1));
  fc  = vrna_fold_compound(seq, NULL, VRNA_OPTION_DEFAULT);
  mfe = vrna_mfe(fc, ss);

  printf("%s\n%s (%6.2f)\n", seq, ss, mfe);

  return EXIT_SUCCESS;
}
```

In our Python interface, the *vrna_fold_compound_t* data structure becomes the *RNA.fold_compound* class, the *vrna_fold_compound()* becomes one of its constructors and the *vrna_mfe()* function becomes the method *RNA.fold_compound.mfe()*. So, the Python code would probably translate to something like

```python
import RNA

seq = "AGACGACAAGGUUGAAUCGCACCCACAGUCUAUGAGUCGGUG"
fc  = RNA.fold_compound(seq)
(ss, mfe) = fc.mfe()

print(f"{seq}\n{ss} ({mfe:6.2f})")
```

Note

The C-function *vrna_mfe()* actually returns two values, the MFE in units of kcal · mol^{-1} and the corresponding MFE structure. The latter is written to the `ss` pointer. This is necessary since `C` functions can at most return one single value. In Python, function and methods may return arbitrarily many values instead, and in addition, passing parameters to a function or method such that it changes its content is generally discouraged. Therefore, our functions that return values through function parameters usually return them *regularly* in the Python interface.

Lists and Tuples

C-functions in our API that return or receive list-like data usually utilize *pointers*. Since there are no such things in Python, they would be wrapped as particular kind of objects that would then be tedious to work with. For the Python interface, we therefore tried to wrap the majority of these instances to *native* Python types, such as `list` or `tuple`. Therefore, one can usually pass a `list` to a function that uses pointers to array in `C`, and expect to receive a `list` or `tuple` from functions that return pointers to arrays.

3.8.4 Energy Parameters

Energy parameters are compiled into our library, so there is usually no necessity to load them from a file. All parameter files shipped with the ViennaRNA Package can be loaded by simply calling any of the dedicated functions:

- *RNA.params_load_RNA_Turner2004()* (default RNA parameters)
- *RNA.params_load_DNA_Mathews2004()* (default DNA parameters)
- *RNA.params_load_DNA_Mathews1999()* (old DNA parameters)
- *RNA.params_load_RNA_Turner1999()* (old RNA parameters)
- *RNA.params_load_RNA_Andronescu2007()* (trained RNA parameters)
- *RNA.params_load_RNA_Langdon2018()* (trained RNA parameters)
- *RNA.params_load_RNA_misc_special_hairpins()* (special hairpin loop parameters)

Tip

A few more Python code examples can be found *here*.

See also

The RNA Python module contains a full Python API description.

3.9 Callback Functions

Many functions in `RNAlib` support so-called **callback mechanisms** to propagate the computation results. In essence, this means that a user defines a function that takes computation results as input and does whatever should be done with these results. Then, this function is provided to our algorithms in `RNAlib` such that they can call the function and provide corresponding data as soon as it has been computed.

3.9.1 Why Callbacks?

Using callback mechanisms, our library enables users not only to retrieve computed data without the need for parsing complicated data structures, but also allows one to tweak our implementation to do additional tasks without the requirement of a re-implementation of basic algorithms.

Our implementation of the callback mechanisms always follows the same scheme:

The user ...

- ... defines a function that complies with the interface we've defined, and
- ... passes a pointer to said function to our implementations

In addition to the specific arguments of our callback interfaces, virtually all callbacks receive an additional *pass-through-pointer* as their last argument. This enables one to ...

- ... encapsulate data, and
- ... provide thread-safe operations,

since this pointer is simply passed through by our library functions. It may therefore hold the address of an arbitrary, user-defined data structure.

3.9.2 Scripting Language Support

Our callback mechanisms also work in a cross-language specific manner. This means that users can write functions (or methods) in a scripting language, such as Python, and use them as callbacks for our C-library. Our scripting language interface will take care of transforming the relevant data structures from the target language into C and vice-versa. Whenever our algorithms trigger the callback, they will then be calling the actual scripting language function and provide the corresponding data directly to them.

> **Warning**
>
> Keep in mind that the translation between the scripting language and C involves many extra function calls to prepare and evaluate the corresponding data structures. This in turn impacts the runtime of our algorithms that can be substantial. For instance providing callbacks for the hard- or soft constraints framework from a scripting language can lead to a slow-down of up to a factor of 10.

3.10 RNAlib API v3.0

With version 2.2 we introduce the new API that will take over the old one in the future version 3.0. By then, backwards compatibility will be broken, and third party applications using RNAlib need to be ported. This switch of API became necessary, since many new features found their way into the RNAlib where a balance between threadsafety and easy-to-use library functions is hard or even impossible to establish. Furthermore, many old functions of the library are present as slightly modified copies of themself to provide a crude way to overload functions.

Therefore, we introduce the new v3.0 API very early in our development stage such that developers have enough time to migrate to the new functions and interfaces. We also started to provide encapsulation of the RNAlib functions, data structures, typedefs, and macros by prefixing them with `vrna_` and `VRNA_` , respectively.

Header files should also be included using the `ViennaRNA/` namespace, e.g.

```
#include <ViennaRNA/fold.h>
```

instead of just using

```
#include <fold.h>
```

as is has been required for RNAlib 1.x and 2.x.

This eases the work for programmers of third party applications that would otherwise need to put much effort into renaming functions and data types in their own implementations if their names appear in our library. Since we still provide backward compatibility up to the last version of RNAlib 2.x, this advantage may be fully exploited only starting from v3.0 which will be released in the future. However, our plan is to provide the possibility for an early switch-off mechanism of the backward compatibility in one of our next releases of ViennaRNA Package 2.x.

Tip

Examples on how to use the new v3.0 API can be found in the *C Examples* section.

3.11 Example Code

3.11.1 C Examples

MFE Prediction (simple interface)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/mfe/global.h>
#include <ViennaRNA/utils/basic.h>

int
main()
{
  /* The RNA sequence */
  char  *seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

  /* allocate memory for MFE structure (length + 1) */
  char  *structure = (char *)vrna_alloc(sizeof(char) * (strlen(seq) + 1));

  /* predict Minmum Free Energy and corresponding secondary structure */
  float mfe = vrna_fold(seq, structure);

  /* print sequence, structure and MFE */
  printf("%s\n%s [ %6.2f ]\n", seq, structure, mfe);

  /* cleanup memory */
  free(structure);

  return 0;
}
```

MFE Prediction (VRNA 3.0 interface)

```
#include <stdlib.h>
#include <stdio.h>

#include <ViennaRNA/fold_compound.h>
#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/utils/strings.h>
#include <ViennaRNA/mfe/global.h>


int
main()
{
  /* initialize random number generator */
  vrna_init_rand();
```

(continues on next page)

(continued from previous page)

```
  /* Generate a random sequence of 50 nucleotides */
  char                  *seq = vrna_random_string(50, "ACGU");

  /* Create a fold compound for the sequence */
  vrna_fold_compound_t  *fc = vrna_fold_compound(seq, NULL, VRNA_OPTION_DEFAULT);

  /* allocate memory for MFE structure (length + 1) */
  char                  *structure = (char *)vrna_alloc(sizeof(char) * (strlen(seq) +␣
↪1));

  /* predict Minmum Free Energy and corresponding secondary structure */
  float                 mfe = vrna_mfe(fc, structure);

  /* print sequence, structure and MFE */
  printf("%s\n%s [ %6.2f ]\n", seq, structure, mfe);

  /* cleanup memory */
  free(seq);
  free(structure);
  vrna_fold_compound_free(fc);

  return 0;
}
```

MFE and Centroid structure Prediction

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include  <ViennaRNA/data_structures.h>
#include  <ViennaRNA/params/basic.h>
#include  <ViennaRNA/utils/basic.h>
#include  <ViennaRNA/eval.h>
#include  <ViennaRNA/mfe/global.h>
#include  <ViennaRNA/part_func.h>


int
main(int  argc,
     char *argv[])
{
  char                  *seq =

↪"AGACGACAAGGUUGAAUCGCACCCACAGUCUAUGAGUCGGUGACAACAUUACGAAAGGCUGUAAAAUCAAUUAUUCACCACAGGGGGCCCCCGUGUC
↪";
  char                  *mfe_structure  = vrna_alloc(sizeof(char) * (strlen(seq) +␣
↪1));
  char                  *prob_string    = vrna_alloc(sizeof(char) * (strlen(seq) +␣
↪1));

  /* get a vrna_fold_compound with default settings */
  vrna_fold_compound_t  *vc = vrna_fold_compound(seq, NULL, VRNA_OPTION_DEFAULT);

  /* call MFE function */
  double                mfe = (double)vrna_mfe(vc, mfe_structure);
```

(continues on next page)

(continued from previous page)

```
  printf("%s\n%s (%6.2f)\n", seq, mfe_structure, mfe);

  /* rescale parameters for Boltzmann factors */
  vrna_exp_params_rescale(vc, &mfe);

  /* call PF function */
  FLT_OR_DBL en = vrna_pf(vc, prob_string);

  /* print probability string and free energy of ensemble */
  printf("%s (%6.2f)\n", prob_string, en);

  /* compute centroid structure */
  double  dist;
  char    *cent = vrna_centroid(vc, &dist);

  /* print centroid structure, its free energy and mean distance to the ensemble */
  printf("%s (%6.2f d=%6.2f)\n", cent, vrna_eval_structure(vc, cent), dist);

  /* free centroid structure */
  free(cent);

  /* free pseudo dot-bracket probability string */
  free(prob_string);

  /* free mfe structure */
  free(mfe_structure);

  /* free memory occupied by vrna_fold_compound */
  vrna_fold_compound_free(vc);

  return EXIT_SUCCESS;
}
```

Suboptimal Structure Prediction

using the callback mechanism

```
#include <stdlib.h>
#include <stdio.h>

#include <ViennaRNA/fold_compound.h>
#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/utils/strings.h>
#include <ViennaRNA/subopt.h>


void
subopt_callback(const char  *structure,
                float       energy,
                void        *data)
{
  /* simply print the result and increase the counter variable by 1 */
  if (structure)
    printf("%d.\t%s\t%6.2f\n", (*((int *)data))++, structure, energy);
}
```

(continues on next page)

(continued from previous page)

```c


int
main()
{
  /* initialize random number generator */
  vrna_init_rand();

  /* Generate a random sequence of 50 nucleotides */
  char                  *seq = vrna_random_string(50, "ACGU");

  /* Create a fold compound for the sequence */
  vrna_fold_compound_t  *fc = vrna_fold_compound(seq, NULL, VRNA_OPTION_DEFAULT);

  int                   counter = 0;

  /*
   *  call subopt to enumerate all secondary structures in an energy band of
   *  5 kcal/mol of the MFE and pass it the address of the callback and counter
   *  variable
   */
  vrna_subopt_cb(fc, 500, &subopt_callback, (void *)&counter);

  /* cleanup memory */
  free(seq);
  vrna_fold_compound_free(fc);

  return 0;
}
```

Base Pair Probabilities

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/fold.h>
#include <ViennaRNA/part_func.h>
#include <ViennaRNA/utils/basic.h>

int
main()
{
  /* The RNA sequence */
  char      *seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

  /* allocate memory for pairing propensity string (length + 1) */
  char      *propensity = (char *)vrna_alloc(sizeof(char) * (strlen(seq) + 1));

  /* pointers for storing and navigating through base pair probabilities */
  vrna_ep_t *ptr, *pair_probabilities = NULL;

  float     en = vrna_pf_fold(seq, propensity, &pair_probabilities);

  /* print sequence, pairing propensity string and ensemble free energy */
  printf("%s\n%s [ %6.2f ]\n", seq, propensity, en);
```

(continues on next page)

(continued from previous page)

```
  /* print all base pairs with probability above 50% */
  for (ptr = pair_probabilities; ptr->i != 0; ptr++)
    if (ptr->p > 0.5)
      printf("p(%d, %d) = %g\n", ptr->i, ptr->j, ptr->p);

  /* cleanup memory */
  free(pair_probabilities);
  free(propensity);

  return 0;
}
```

MFE Consensus Structure Prediction

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/mfe/global.h>
#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/utils/alignments.h>

int
main()
{
  /* The RNA sequence alignment */
  const char  *sequences[] = {
    "CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGU",
    "CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGU",
    "---CUCGACACCACU---GCCUCGGUUACCCAUCGGUGCAGUGCGGGU",
    NULL /* indicates end of alignment */
  };

  /* compute the consensus sequence */
  char        *cons = consensus(sequences);

  /* allocate memory for MFE consensus structure (length + 1) */
  char        *structure = (char *)vrna_alloc(sizeof(char) * (strlen(sequences[0]) +␣
↪1));

  /* predict Minmum Free Energy and corresponding secondary structure */
  float       mfe = vrna_alifold(sequences, structure);

  /* print consensus sequence, structure and MFE */
  printf("%s\n%s [ %6.2f ]\n", cons, structure, mfe);

  /* cleanup memory */
  free(cons);
  free(structure);

  return 0;
}
```

MFE Prediction (deviating from default settings)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <ViennaRNA/model.h>
#include <ViennaRNA/fold_compound.h>
#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/mfe/global.h>

int
main()
{
  char      *seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

  /* allocate memory for MFE structure (length + 1) */
  char      *structure = (char *)vrna_alloc(sizeof(char) * (strlen(seq) + 1));

  /* create a new model details structure to store the Model Settings */
  vrna_md_t md;

  /* ALWAYS set default model settings first! */
  vrna_md_set_default(&md);

  /* Mark sequence as being circular, change temperature to
   * non-default and change dangle model
   */
  md.circ         = 1;    /* Treat RNA input as circular sequence */
  md.temperature  = 25.0; /* 25 Deg Celcius */
  md.dangles      = 1;    /* Switch to dangles-model 1 */

  /* create a fold compound */
  vrna_fold_compound_t  *fc = vrna_fold_compound(seq, &md, VRNA_OPTION_DEFAULT);

  /* predict Minmum Free Energy and corresponding secondary structure */
  float                 mfe = vrna_mfe(fc, structure);

  /* print sequence, structure and MFE */
  printf("%s\n%s [ %6.2f ]\n", seq, structure, mfe);

  /* cleanup memory */
  free(structure);
  vrna_fold_compound_free(fc);

  return 0;
}
```

Soft Constraints

```
#include <stdlib.h>
#include <stdio.h>

#include <ViennaRNA/fold_compound.h>
#include <ViennaRNA/utils/basic.h>
#include <ViennaRNA/utils/strings.h>
#include <ViennaRNA/constraints/soft.h>
```

(continues on next page)

(continued from previous page)

```c
#include <ViennaRNA/mfe.h>

int
main()
{
  /* initialize random number generator */
  vrna_init_rand();

  /* Generate a random sequence of 50 nucleotides */
  char                  *seq = vrna_random_string(50, "ACGU");

  /* Create a fold compound for the sequence */
  vrna_fold_compound_t  *fc = vrna_fold_compound(seq, NULL, VRNA_OPTION_DEFAULT);

  /* Add soft constraint of -1.7 kcal/mol to nucleotide 5 whenever it appears in an␣
↪unpaired context */
  vrna_sc_add_up(fc, 5, -1.7, VRNA_OPTION_DEFAULT);

  /* allocate memory for MFE structure (length + 1) */
  char  *structure = (char *)vrna_alloc(sizeof(char) * 51);

  /* predict Minmum Free Energy and corresponding secondary structure */
  float mfe = vrna_mfe(fc, structure);

  /* print seqeunce, structure and MFE */
  printf("%s\n%s [ %6.2f ]\n", seq, structure, mfe);

  /* cleanup memory */
  free(seq);
  free(structure);
  vrna_fold_compound_free(fc);

  return 0;
}
```

A more elaborate (old) example

```c
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include  <string.h>
#include  "utils.h"
#include  "fold_vars.h"
#include  "fold.h"
#include  "part_func.h"
#include  "inverse.h"
#include  "RNAstruct.h"
#include  "treedist.h"
#include  "stringdist.h"
#include  "profiledist.h"

void
main()
{
  char        *seq1 = "CGCAGGGAUACCCGCG", *seq2 = "GCGCCCAUAGGGACGC",
              *struct1, *struct2, *xstruc;
```

(continues on next page)

(continued from previous page)

```
float       e1, e2, tree_dist, string_dist, profile_dist, kT;
Tree        *T1, *T2;
swString    *S1, *S2;
float       *pf1, *pf2;
FLT_OR_DBL  *bppm;

/* fold at 30C instead of the default 37C */
temperature = 30.;       /* must be set *before* initializing  */

/* allocate memory for structure and fold */
struct1 = (char *)space(sizeof(char) * (strlen(seq1) + 1));
e1      = fold(seq1, struct1);

struct2 = (char *)space(sizeof(char) * (strlen(seq2) + 1));
e2      = fold(seq2, struct2);

free_arrays();      /* free arrays used in fold() */

/* produce tree and string representations for comparison */
xstruc  = expand_Full(struct1);
T1      = make_tree(xstruc);
S1      = Make_swString(xstruc);
free(xstruc);

xstruc  = expand_Full(struct2);
T2      = make_tree(xstruc);
S2      = Make_swString(xstruc);
free(xstruc);

/* calculate tree edit distance and aligned structures with gaps */
edit_backtrack  = 1;
tree_dist       = tree_edit_distance(T1, T2);
free_tree(T1);
free_tree(T2);
unexpand_aligned_F(aligned_line);
printf("%s\n%s  %3.2f\n", aligned_line[0], aligned_line[1], tree_dist);

/* same thing using string edit (alignment) distance */
string_dist = string_edit_distance(S1, S2);
free(S1);
free(S2);
printf("%s  mfe=%5.2f\n%s  mfe=%5.2f  dist=%3.2f\n",
       aligned_line[0], e1, aligned_line[1], e2, string_dist);

/* for longer sequences one should also set a scaling factor for
 * partition function folding, e.g: */
kT        = (temperature + 273.15) * 1.98717 / 1000.; /* kT in kcal/mol */
pf_scale  = exp(-e1 / kT / strlen(seq1));

/* calculate partition function and base pair probabilities */
e1 = pf_fold(seq1, struct1);
/* get the base pair probability matrix for the previous run of pf_fold() */
bppm  = export_bppm();
pf1   = Make_bp_profile_bppm(bppm, strlen(seq1));

e2 = pf_fold(seq2, struct2);
```

(continues on next page)

(continued from previous page)

```
  /* get the base pair probability matrix for the previous run of pf_fold() */
  bppm  = export_bppm();
  pf2   = Make_bp_profile_bppm(bppm, strlen(seq2));

  free_pf_arrays();   /* free space allocated for pf_fold() */

  profile_dist = profile_edit_distance(pf1, pf2);
  printf("%s  free energy=%5.2f\n%s  free energy=%5.2f  dist=%3.2f\n",
         aligned_line[0], e1, aligned_line[1], e2, profile_dist);

  free_profile(pf1);
  free_profile(pf2);
}
```

3.11.2 Python Examples

MFE Prediction (flat interface)

```
import RNA

# The RNA sequence
seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA"

# compute minimum free energy (MFE) and corresponding structure
(ss, mfe) = RNA.fold(seq)

# print output
print("{}\n{} [ {:6.2f} ]".format(seq, ss, mfe))
```

MFE Prediction (object oriented interface)

```
import RNA;

sequence = "CGCAGGGAUACCCGCG"

# create new fold_compound object
fc = RNA.fold_compound(sequence)

# compute minimum free energy (mfe) and corresponding structure
(ss, mfe) = fc.mfe()

# print output
print("{} [ {:6.2f} ]".format(ss, mfe))
```

Suboptimal Structure Prediction

```
import RNA

sequence = "GGGGAAAACCCC"

# Set global switch for unique ML decomposition
RNA.cvar.uniq_ML = 1

subopt_data = { 'counter' : 1, 'sequence' : sequence }
```

(continues on next page)

(continued from previous page)

```
# Print a subopt result as FASTA record
def print_subopt_result(structure, energy, data):
    if not structure == None:
        print(">subopt {:d}".format(data['counter']))
        print("{}\n{} [{:6.2f}]".format(data['sequence'], structure, energy))
        # increase structure counter
        data['counter'] = data['counter'] + 1

# Create a 'fold_compound' for our sequence
a = RNA.fold_compound(sequence)

# Enumerate all structures 500 dacal/mol = 5 kcal/mol arround
# the MFE and print each structure using the function above
a.subopt_cb(500, print_subopt_result, subopt_data);
```

Boltzmann Sampling

a.k.a. Probabilistic Backtracing

```
import RNA

sequence =
↪"UGGGAAUAGUCUCUUCCGAGUCUCGCGGGCGACGGGCGAUCUUCGAAAGUGGAAUCCGUACUUAUACCGCCUGUGCGGACUACUAUCCUGACCACAU
↪"

def store_structure(s, data):
    """
    A simple callback function that stores
    a structure sample into a list
    """
    if s:
        data.append(s)


"""
First we prepare a fold_compound object
"""

# create model details
md = RNA.md()

# activate unique multibranch loop decomposition
md.uniq_ML = 1

# create fold compound object
fc = RNA.fold_compound(sequence, md)

# compute MFE
(ss, mfe) = fc.mfe()

# rescale Boltzmann factors according to MFE
fc.exp_params_rescale(mfe)

# compute partition function to fill DP matrices
fc.pf()
```

(continues on next page)

(continued from previous page)

```
"""
Now we are ready to perform Boltzmann sampling
"""

# 1. backtrace a single sub-structure of length 10
print("{}".format(fc.pbacktrack5(10)))


# 2. backtrace a single sub-structure of length 50
print("{}".format(fc.pbacktrack5(50)))

# 3. backtrace multiple sub-structures of length 10 at once
for s in fc.pbacktrack5(20, 10):
    print("{}".format(s))

# 4. backtrace multiple sub-structures of length 50 at once
for s in fc.pbacktrack5(100, 50):
    print("{}".format(s))

# 5. backtrace a single structure (full length)
print("{}".format(fc.pbacktrack()))

# 6. backtrace multiple structures at once
for s in fc.pbacktrack(100):
    print("{}".format(s))

# 7. backtrace multiple structures non-redundantly
for s in fc.pbacktrack(100, RNA.PBACKTRACK_NON_REDUNDANT):
    print("{}".format(s))

# 8. backtrace multiple structures non-redundantly (with resume option)
num_samples = 500
iterations  = 15
d           = None # pbacktrack memory object
s_list      = []

for i in range(0, iterations):
    d, ss   = fc.pbacktrack(num_samples, d, RNA.PBACKTRACK_NON_REDUNDANT)
    s_list  = s_list + list(ss)

for s in s_list:
    print("{}".format(s))

# 9. backtrace multiple sub-structures of length 50 in callback mode
ss  = []
i   = fc.pbacktrack5(100, 50, store_structure, ss)

for s in ss:
    print("{}".format(s))

# 10. backtrace multiple full-length structures in callback mode
ss  = list()
i   = fc.pbacktrack(100, store_structure, ss)

for s in ss:
    print("{}".format(s))
```

(continues on next page)

(continued from previous page)

```
# 11. non-redundantly backtrace multiple full-length structures in callback mode
ss  = list()
i   = fc.pbacktrack(100, store_structure, ss, RNA.PBACKTRACK_NON_REDUNDANT)

for s in ss:
    print("{}".format(s))

# 12. non-redundantly backtrace multiple full length structures
# in callback mode with resume option
ss = []
d  = None # pbacktrack memory object

for i in range(0, iterations):
    d, i = fc.pbacktrack(num_samples, store_structure, ss, d, RNA.PBACKTRACK_NON_
↪REDUNDANT)

for s in ss:
    print("{}".format(s))

# 13. backtrace a single substructure from the sequence interval [10:50]
print("{}".format(fc.pbacktrack_sub(10, 50)))

# 14. backtrace multiple substructures from the sequence interval [10:50]
for s in fc.pbacktrack_sub(100, 10, 50):
    print("{}".format(s))

# 15. backtrace multiple substructures from the sequence interval [10:50] non-
↪redundantly
for s in fc.pbacktrack_sub(100, 10, 50, RNA.PBACKTRACK_NON_REDUNDANT):
    print("{}".format(s))
```

RNAfold -p MEA equivalent

```
#!/usr/bin/python
#

import RNA

seq =
↪"AUUUCCACUAGAGAAGGUCUAGAGUGUUUGUCGUUUGUCAGAAGUCCCUAUUCCAGGUACGAACACGGUGGAUAUGUUCGACGACAGGAUCGGCGCA
↪"

# create fold_compound data structure (required for all subsequently applied␣
↪algorithms)
fc = RNA.fold_compound(seq)

# compute MFE and MFE structure
(mfe_struct, mfe) = fc.mfe()

# rescale Boltzmann factors for partition function computation
fc.exp_params_rescale(mfe)

# compute partition function
(pp, pf) = fc.pf()
```

(continues on next page)

(continued from previous page)

```
# compute centroid structure
(centroid_struct, dist) = fc.centroid()

# compute free energy of centroid structure
centroid_en = fc.eval_structure(centroid_struct)

# compute MEA structure
(MEA_struct, MEA) = fc.MEA()

# compute free energy of MEA structure
MEA_en = fc.eval_structure(MEA_struct)

# print everything like RNAfold -p --MEA
print("{}\n{} ({:6.2f})".format(seq, mfe_struct, mfe))
print("{} [{:6.2f}]".format(pp, pf))
print("{} {{{:6.2f} d={:.2f}}}".format(centroid_struct, centroid_en, dist))
print("{} {{{:6.2f} MEA={:.2f}}}".format(MEA_struct, MEA_en, MEA))
print(" frequency of mfe structure in ensemble {:g}; ensemble diversity {:-6.2f}".
↪format(fc.pr_structure(mfe_struct), fc.mean_bp_distance()))
```

MFE Consensus Structure Prediction

```
import RNA

# The RNA sequence alignment
sequences = [
    "CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGU",
    "CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGU",
    "---CUCGACACCACU---GCCUCGGUUACCCAUCGGUGCAGUGCGGGU"
]

# compute the consensus sequence
cons = RNA.consensus(sequences)

# predict Minmum Free Energy and corresponding secondary structure
(ss, mfe) = RNA.alifold(sequences);

# print output
print("{}\n{} [ {:6.2f} ]".format(cons, ss, mfe))
```

MFE Prediction (deviating from default settings)

```
import RNA

# The RNA sequence
seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA"

# create a new model details structure with
# non-default settings
md = RNA.md(
        circ        = 1,    # Circular RNA sequence
        temperature = 25.0, # 25 Deg Celcius
        dangles     = 1     # Dangle Model 1
      )
```

(continues on next page)

(continued from previous page)

```
# create a fold compound
fc = RNA.fold_compound(seq, md)

# predict Minmum Free Energy and corresponding secondary structure
(ss, mfe) = fc.mfe()

# print sequence, structure and MFE
print("{}\n{} [ {:6.2f} ]".format(seq, ss, mfe))
```

Fun with Soft Constraints

```
import RNA

seq1 = "CUCGUCGCCUUAAUCCAGUGCGGGCGCUAGACAUCUAGUUAUCGCCGCAA"

# Turn-off dangles globally
RNA.cvar.dangles = 0

# Data structure that will be passed to our MaximumMatching() callback with two␣
↪components:
# 1. a 'dummy' fold_compound to evaluate loop energies w/o constraints, 2. a fresh set␣
↪of energy parameters
mm_data = { 'dummy': RNA.fold_compound(seq1), 'params': RNA.param() }

# Nearest Neighbor Parameter reversal functions
revert_NN = {
    RNA.DECOMP_PAIR_HP:       lambda i, j, k, l, f, p: - f.eval_hp_loop(i, j) - 100,
    RNA.DECOMP_PAIR_IL:       lambda i, j, k, l, f, p: - f.eval_int_loop(i, j, k, l) -
↪ 100,
    RNA.DECOMP_PAIR_ML:       lambda i, j, k, l, f, p: - p.MLclosing - p.MLintern[0] -
↪ (j - i - k + l - 2) * p.MLbase - 100,
    RNA.DECOMP_ML_ML_STEM:    lambda i, j, k, l, f, p: - p.MLintern[0] - (l - k - 1)␣
↪* p.MLbase,
    RNA.DECOMP_ML_STEM:       lambda i, j, k, l, f, p: - p.MLintern[0] - (j - i - k +␣
↪l) * p.MLbase,
    RNA.DECOMP_ML_ML:         lambda i, j, k, l, f, p: - (j - i - k + l) * p.MLbase,
    RNA.DECOMP_ML_ML_ML:      lambda i, j, k, l, f, p: 0,
    RNA.DECOMP_ML_UP:         lambda i, j, k, l, f, p: - (j - i + 1) * p.MLbase,
    RNA.DECOMP_EXT_STEM:      lambda i, j, k, l, f, p: - f.eval_ext_stem(k, l),
    RNA.DECOMP_EXT_EXT:       lambda i, j, k, l, f, p: 0,
    RNA.DECOMP_EXT_STEM_EXT:  lambda i, j, k, l, f, p: - f.eval_ext_stem(i, k),
    RNA.DECOMP_EXT_EXT_STEM:  lambda i, j, k, l, f, p: - f.eval_ext_stem(l, j),
            }

# Maximum Matching callback function (will be called by RNAlib in each decomposition␣
↪step)
def MaximumMatching(i, j, k, l, d, data):
    return revert_NN[d](i, j, k, l, data['dummy'], data['params'])

# Create a 'fold_compound' for our sequence
fc = RNA.fold_compound(seq1)

# Add maximum matching soft-constraints
fc.sc_add_f(MaximumMatching)
fc.sc_add_data(mm_data, None)
```

(continues on next page)

(continued from previous page)

```
# Call MFE algorithm
(s, mm) = fc.mfe()

# print result
print("{}\n{} (MM: {:d})".format(seq1, s, int(-mm)))
```

Parsing Alignments

Reading the first entry from a STOCKHOLM 1.0 formatted MSA file `msa.stk` may look like this:

```
num, names, aln, id, ss = RNA.file_msa_read("msa.stk")
```

Similarly, if the file contains more than one alignment, one can use the *RNA.file_msa_read_record()* function to subsequently read each alignment separately:

```
with open("msa.stk") as f:
    while True:
        num, names, aln, id, ss = RNA.file_msa_read_record(f)
        if num < 0:
            break
        elif num == 0:
            print("empty alignment")
        else:
            print(names, aln)
```

After successfully reading the first record, the variable `num` contains the number of sequences in the alignment (the actual return value of the C-function), while the variables `names`, `aln`, `id`, and `ss` are lists of the sequence names and aligned sequences, as well as strings holding the alignment ID and the structure as stated in the `SS_cons` line, respectively.

Note

The last two return values may be empty strings in case the alignment does not provide the required data.

Logging

The example below demonstrates how log messages issued by the *RNAlib* C-library can be re-routed to the native Python *logging* module.

```
import RNA
import logging


# Set the default format and log-level for the Python `logging` module
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)


def reroute_logs(e, data):
    """
    The wrapper callback function that receives RNAlib log messages
    and passes them through to the Python 'logging' module

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```
    e: dict
        A dictionary that contains the log message, the log level,
        as well as source code file and line number that issued the
        log message
    data: object
        A data object that is passed through to this callback
    """

    log_reroute = {
      RNA.LOG_LEVEL_ERROR: logging.error,
      RNA.LOG_LEVEL_WARNING: logging.warning,
      RNA.LOG_LEVEL_INFO: logging.info,
      RNA.LOG_LEVEL_DEBUG: logging.debug
    }

    # compose an example log message
    message = f"ViennaRNA: \"{e['message']}\" ({e['function']}()@{e['file_name']}:{e[
↪'line_number']})"

    # send log message to appropriate logging channel
    if e['level'] in log_reroute:
        log_reroute[e['level']](message)


# turn-off RNAlib self logging
RNA.log_level_set(RNA.LOG_LEVEL_SILENT)

# attach RNAlib logging to python logging and capture all
# logs with level at least DEBUG
RNA.log_cb_add(reroute_logs, level = RNA.LOG_LEVEL_DEBUG)


# compose an example call that might issue a few debug- or other
# log messages
s = RNA.random_string(10, "ACGU")

fc = RNA.fold_compound(s)

fc.pbacktrack()
```

3.11.3 Perl 5 Examples

MFE Prediction (flat interface)

```
use RNA;

# The RNA sequence
my $seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

# compute minimum free energy (MFE) and corresponding structure
my ($ss, $mfe) = RNA::fold($seq);

# print output
printf "%s\n%s [ %6.2f ]\n", $seq, $ss, $mfe;
```

MFE Prediction (object oriented interface)

```
#!/usr/bin/perl

use warnings;
use strict;

use RNA;

my $seq1 = "CGCAGGGAUACCCGCG";

# create new fold_compound object
my $fc = new RNA::fold_compound($seq1);

# compute minimum free energy (mfe) and corresponding structure
my ($ss, $mfe) = $fc->mfe();

# print output
printf "%s [ %6.2f ]\n", $ss, $mfe;
```

MFE Consensus Structure Prediction

```
use RNA;

# The RNA sequence alignment
my @sequences = (
    "CUGCCUCACAACGUUUGUGCCUCAGUUACCCGUAGAUGUAGUGAGGGU",
    "CUGCCUCACAACAUUUGUGCCUCAGUUACUCAUAGAUGUAGUGAGGGU",
    "---CUCGACACCACU---GCCUCGGUUACCCAUCGGUGCAGUGCGGGU"
);

# compute the consensus sequence
my $cons = RNA::consensus(\@sequences);

# predict Minmum Free Energy and corresponding secondary structure
my ($ss, $mfe) = RNA::alifold(\@sequences);

# print output
printf "%s\n%s [ %6.2f ]\n", $cons, $ss, $mfe;
```

MFE Prediction (deviating from default settings)

```
use RNA;

# The RNA sequence
my $seq = "GAGUAGUGGAACCAGGCUAUGUUUGUGACUCGCAGACUAACA";

# create a new model details structure
my $md = new RNA::md();

# change temperature and dangle model
$md->{circ}         = 1;    # Circular RNA sequence
$md->{temperature}  = 25.0; # 25 Deg Celcius
$md->{dangles}      = 1;    # Dangle Model 1

# create a fold compound
my $fc = new RNA::fold_compound($seq, $md);
```

(continues on next page)

(continued from previous page)

```
# predict Minmum Free Energy and corresponding secondary structure
my ($ss, $mfe) = $fc->mfe();

# print sequence, structure and MFE
printf "%s\n%s [ %6.2f ]\n", $seq, $ss, $mfe;
```

Fun with Soft Constraints

```
use strict;
use warnings;
use Data::Dumper;
use RNA;

my $seq1 = "CUCGUCGCCUUAAUCCAGUGCGGGCGCUAGACAUCUAGUUAUCGCCGCAA";

# Turn-off dangles globally
$RNA::dangles = 0;

# Data structure that will be passed to our MaximumMatching() callback with two␣
↪components:
# 1. a 'dummy' fold_compound to evaluate loop energies w/o constraints, 2. a fresh set␣
↪of energy parameters
my %mm_data = ( 'dummy' => new RNA::fold_compound($seq1), 'params' => new␣
↪RNA::param() );

# Nearest Neighbor Parameter reversal functions
my %revert_NN = (
    RNA::DECOMP_PAIR_HP => sub { my ($i, $j, $k, $l, $f, $p) = @_; return - $f->eval_
↪hp_loop($i, $j) - 100;},
    RNA::DECOMP_PAIR_IL => sub { my ($i, $j, $k, $l, $f, $p) = @_; return - $f->eval_
↪int_loop($i, $j, $k, $l) - 100},
    RNA::DECOMP_PAIR_ML => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  - $p->
↪{MLclosing} - $p->{MLintern}[0] - ($j - $i - $k + $l - 2) * $p->{MLbase} - 100},
    RNA::DECOMP_ML_ML_STEM => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  - $p->
↪{MLintern}[0] - ($l - $k - 1) * $p->{MLbase}},
    RNA::DECOMP_ML_ML_ML => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  0},
    RNA::DECOMP_ML_STEM => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  - $p->
↪{MLintern}[0] - ($j - $i - $k + $l) * $p->{MLbase}},
    RNA::DECOMP_ML_ML => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  - ($j - $i -
↪$k + $l) * $p->{MLbase}},
    RNA::DECOMP_ML_UP => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  - ($j - $i +␣
↪1) * $p->{MLbase}},
    RNA::DECOMP_EXT_STEM => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  - $f->E_
↪ext_loop($k, $l)},
    RNA::DECOMP_EXT_EXT => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  0},
    RNA::DECOMP_EXT_STEM_EXT => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  - $f->
↪E_ext_loop($i, $k)},
    RNA::DECOMP_EXT_EXT_STEM => sub { my ($i, $j, $k, $l, $f, $p) = @_; return : - $f-
↪>E_ext_loop($l, $j)},
    RNA::DECOMP_EXT_EXT_STEM1 => sub { my ($i, $j, $k, $l, $f, $p) = @_; return  - $f-
↪>E_ext_loop($l, $j - 1)},
);

# Maximum Matching callback function (will be called by RNAlib in each decomposition␣
↪step)
```

(continues on next page)

(continued from previous page)

```
sub MaximumMatching {
  my ($i, $j, $k, $l, $d, $data) = @_;
  return $revert_NN{$d}->($i, $j, $k, $l, $data->{'dummy'}, $data->{'params'}) if␣
↪defined $revert_NN{$d};
  return 0;
}

# Create a 'fold_compound' for our sequence
my $fc = new RNA::fold_compound($seq1);

# Add maximum matching soft-constraints
$fc->sc_add_f(\&MaximumMatching);
$fc->sc_add_data(\%mm_data, undef);

# Call MFE algorithm
my ($s, $mm) = $fc->mfe();

# print result
printf("%s\n%s (MM: %d)\n", $seq1, $s, - $mm);
```

CHAPTER

FOUR

RNALIB API REFERENCE

4.1 Structure Prediction Algorithms

4.1.1 Minimum Free Energy (MFE)

Global MFE Prediction

Functions related tor *global* Minimum Free Energy (MFE) prediction algorithms.

Basic global MFE prediction interface

float **vrna_mfe**(*vrna_fold_compound_t* *fc, char *structure)

Compute minimum free energy and an appropriate secondary structure of an RNA sequence, or RNA sequence alignment.

```
#include <ViennaRNA/mfe/global.h>
```

Depending on the type of the provided *vrna_fold_compound_t*, this function predicts the MFE for a single sequence (or connected component of multiple sequences), or an averaged MFE for a sequence alignment. If backtracking is activated, it also constructs the corresponding secondary structure, or consensus structure. Therefore, the second parameter, *structure*, has to point to an allocated block of memory with a size of at least $\text{strlen}(\text{sequence}) + 1$ to store the backtracked MFE structure. (For consensus structures, this is the length of the alignment + 1. If `NULL` is passed, no backtracking will be performed.

SWIG Wrapper Notes:

This function is attached as method `Minimum Free Energy (MFE) Algorithms` to objects of type *The Fold Compound*. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.fold_compound.mfe()* in the *The RNA Python module*.

See also

vrna_fold_compound_t, *vrna_fold_compound()*, *vrna_fold()*, *vrna_circfold()*, *vrna_fold_compound_comparative()*, *vrna_alifold()*, *vrna_circalifold()*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Parameters

- **fc** – fold compound
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to (Maybe NULL)

Returns
the minimum free energy (MFE) in kcal/mol

float **vrna_mfe_dimer**(*vrna_fold_compound_t* *fc, char *structure)

Compute the minimum free energy of two interacting RNA molecules.

```
#include <ViennaRNA/mfe/global.h>
```

The code is analog to the *vrna_mfe()* function.

Deprecated:

This function is obsolete since *vrna_mfe()* can handle complexes multiple sequences since v2.5.0. Use *vrna_mfe()* for connected component MFE instead and compute MFEs of unconnected states separately.

SWIG Wrapper Notes:

This function is attached as method `mfe_dimer()` to objects of type *The Fold Compound*. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.fold_compound.mfe_dimer()* in the *The RNA Python module*.

See also

vrna_mfe()

Parameters

- **fc** – fold compound
- **structure** – Will hold the barcket dot structure of the dimer molecule

Returns
minimum free energy of the structure

Simplified global MFE prediction using sequence(s) or multiple sequence alignment(s)

float **vrna_fold**(const char *sequence, char *structure)

Compute Minimum Free Energy (MFE), and a corresponding secondary structure for an RNA sequence.

```
#include <ViennaRNA/mfe/global.h>
```

This simplified interface to *vrna_mfe()* computes the MFE and, if required, a secondary structure for an RNA sequence using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

SWIG Wrapper Notes:

This function is available as function *fold()* in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.fold()* in the *The RNA Python module*.

See also

vrna_circfold(), *vrna_mfe()*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_mfe()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequence** – RNA sequence
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns

the minimum free energy (MFE) in kcal/mol

float **vrna_circfold**(const char *sequence, char *structure)

Compute Minimum Free Energy (MFE), and a corresponding secondary structure for a circular RNA sequence.

```
#include <ViennaRNA/mfe/global.h>
```

This simplified interface to *vrna_mfe()* computes the MFE and, if required, a secondary structure for a circular RNA sequence using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

Folding of circular RNA sequences is handled as a post-processing step of the forward recursions. See Hofacker and Stadler [2006] for further details.

SWIG Wrapper Notes:

This function is available as function `circfold()` in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. `RNA.circfold()` in the *The RNA Python module*.

See also

vrna_fold(), *vrna_mfe()*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_mfe()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequence** – RNA sequence
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns
the minimum free energy (MFE) in kcal/mol

float **vrna_alifold**(const char **sequences, char *structure)

Compute Minimum Free Energy (MFE), and a corresponding consensus secondary structure for an RNA sequence alignment using a comparative method.

```
#include <ViennaRNA/mfe/global.h>
```

This simplified interface to *vrna_mfe()* computes the MFE and, if required, a consensus secondary structure for an RNA sequence alignment using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

SWIG Wrapper Notes:

This function is available as function `alifold()` in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. `RNA.alifold()` in the *The RNA Python module*.

See also

vrna_circalifold(), *vrna_mfe()*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_mfe()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequences** – RNA sequence alignment
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns
the minimum free energy (MFE) in kcal/mol

float **vrna_circalifold**(const char **sequences, char *structure)

Compute Minimum Free Energy (MFE), and a corresponding consensus secondary structure for a sequence alignment of circular RNAs using a comparative method.

```
#include <ViennaRNA/mfe/global.h>
```

This simplified interface to *vrna_mfe()* computes the MFE and, if required, a consensus secondary structure for an RNA sequence alignment using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

Folding of circular RNA sequences is handled as a post-processing step of the forward recursions. See Hofacker and Stadler [2006] for further details.

SWIG Wrapper Notes:

This function is available as function `circalifold()` in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. `RNA.circalifold()` in the *The RNA Python module*.

See also

vrna_alifold(), *vrna_mfe()*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_mfe()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequences** – Sequence alignment of circular RNAs
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns

the minimum free energy (MFE) in kcal/mol

float **vrna_cofold**(const char *sequence, char *structure)

Compute Minimum Free Energy (MFE), and a corresponding secondary structure for two dimerized RNA sequences.

```
#include <ViennaRNA/mfe/global.h>
```

This simplified interface to *vrna_mfe()* computes the MFE and, if required, a secondary structure for two RNA sequences upon dimerization using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

Deprecated:

This function is obsolete since *vrna_mfe()*/vrna_fold() can handle complexes multiple sequences since v2.5.0. Use *vrna_mfe()*/vrna_fold() for connected component MFE instead and compute MFEs of unconnected states separately.

SWIG Wrapper Notes:

This function is available as function `cofold()` in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. `RNA.cofold()` in the *The RNA Python module*.

See also

vrna_fold(), *vrna_mfe()*, *vrna_fold_compound()*, *vrna_fold_compound_t*, *vrna_cut_point_insert()*

> **Note**
>
> In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_mfe()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequence** – two RNA sequences separated by the '&' character
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns

the minimum free energy (MFE) in kcal/mol

Local MFE Prediction

Functions related tor *local* (sliding window) Minimum Free Energy (MFE) prediction algorithms.

Basic local (sliding window) MFE prediction interface

float **vrna_mfe_window**(*vrna_fold_compound_t* *fc, FILE *file)

Local MFE prediction using a sliding window approach.

```
#include <ViennaRNA/mfe/local.h>
```

Computes minimum free energy structures using a sliding window approach, where base pairs may not span outside the window. In contrast to *vrna_mfe()*, where a maximum base pair span may be set using the *vrna_md_t.max_bp_span* attribute and one globally optimal structure is predicted, this function uses a sliding window to retrieve all locally optimal structures within each window. The size of the sliding window is set in the *vrna_md_t.window_size* attribute, prior to the retrieval of the *vrna_fold_compound_t* using *vrna_fold_compound()* with option *VRNA_OPTION_WINDOW*

The predicted structures are written on-the-fly, either to stdout, if a NULL pointer is passed as file parameter, or to the corresponding filehandle.

SWIG Wrapper Notes:

This function is attached as overloaded method *Local (sliding window) MFE Prediction* to objects of type *The Fold Compound*. The parameter FILE has default value of NULL and can be omitted. See e.g. *RNA.fold_compound.mfe_window()* in the *The RNA Python module*.

> **See also**
>
> *vrna_fold_compound()*, vrna_mfe_window_zscore(), *vrna_mfe()*, *vrna_Lfold()*, vrna_Lfoldz(), *VRNA_OPTION_WINDOW*, *vrna_md_t.max_bp_span*, *vrna_md_t.window_size*

Parameters

- **fc** – The *vrna_fold_compound_t* with preallocated memory for the DP matrices
- **file** – The output file handle where predictions are written to (maybe NULL)

float **vrna_mfe_window_cb**(*vrna_fold_compound_t* *fc, *vrna_mfe_window_f* cb, void *data)

```
#include <ViennaRNA/mfe/local.h>
```

SWIG Wrapper Notes:

This function is attached as overloaded method `mfe_window_cb()` to objects of type *The Fold Compound*. The parameter `data` has default value of `NULL` and can be omitted. See e.g. *RNA.fold_compound.mfe_window_cb()* in the *The RNA Python module*.

int **vrna_backtrack_window**(*vrna_fold_compound_t* *fc, const char *Lfold_filename, long file_pos, char **structure, double mfe)

```
#include <ViennaRNA/mfe/local.h>
```

Simplified local MFE prediction using sequence(s) or multiple sequence alignment(s)

float **vrna_Lfold**(const char *string, int window_size, FILE *file)

Local MFE prediction using a sliding window approach (simplified interface).

```
#include <ViennaRNA/mfe/local.h>
```

This simplified interface to *vrna_mfe_window()* computes the MFE and locally optimal secondary structure using default options. Structures are predicted using a sliding window approach, where base pairs may not span outside the window. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing.

SWIG Wrapper Notes:

This function is available as overloaded function *Lfold()* in the global namespace. The parameter `file` defaults to `NULL` and may be omitted. See e.g. *RNA.Lfold()* in the *The RNA Python module*.

See also

vrna_mfe_window(), vrna_Lfoldz(), vrna_mfe_window_zscore()

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_mfe_window()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **string** – The nucleic acid sequence
- **window_size** – The window size for locally optimal structures
- **file** – The output file handle where predictions are written to (if NULL, output is written to stdout)

float **vrna_Lfold_cb**(const char *string, int window_size, *vrna_mfe_window_f* cb, void *data)

```
#include <ViennaRNA/mfe/local.h>
```

SWIG Wrapper Notes:

This function is available as overloaded function `Lfold_cb()` in the global namespace. The parameter `data` defaults to `NULL` and may be omitted. See e.g. *RNA.Lfold_cb()* in the *The RNA Python module*.

float **vrna_aliLfold**(const char **alignment, int maxdist, FILE *fp)

```
#include <ViennaRNA/mfe/local.h>
```

SWIG Wrapper Notes:

This function is available as overloaded function `aliLfold()` in the global namespace. The parameter `fp` defaults to `NULL` and may be omitted. See e.g. *RNA.aliLfold()* in the *The RNA Python module*.

float **vrna_aliLfold_cb**(const char **alignment, int maxdist, *vrna_mfe_window_f* cb, void *data)

```
#include <ViennaRNA/mfe/local.h>
```

SWIG Wrapper Notes:

This function is available as overloaded function `aliLfold_cb()` in the global namespace. The parameter `data` defaults to `NULL` and may be omitted. See e.g. *RNA.aliLfold_cb()* in the *The RNA Python module*.

Typedefs

typedef void (***vrna_mfe_window_f**)(unsigned int start, unsigned int end, const char *structure, float en, void *data)

The default callback for sliding window MFE structure predictions.

```
#include <ViennaRNA/mfe/local.h>
```

Notes on Callback Functions:

This function will be called for each hit in a sliding window MFE prediction.

See also

vrna_mfe_window()

Param start
provides the first position of the hit (1-based, relative to entire sequence/alignment)

Param end
provides the last position of the hit (1-based, relative to the entire sequence/alignment)

Param structure
provides the (sub)structure in dot-bracket notation

Param en
is the free energy of the structure hit in kcal/mol

Param data
is some arbitrary data pointer passed through by the function executing the callback

typedef void **vrna_mfe_window_callback**(int start, int end, const char *structure, float en, void *data)

```
#include <ViennaRNA/mfe/local.h>
```

Backtracking MFE structures

Backtracking related API

Functions

unsigned int **vrna_bt_f**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

```
#include <ViennaRNA/backtrack/exterior.h>
```

unsigned int **vrna_bt_exterior_f5**(*vrna_fold_compound_t* *fc, unsigned int j, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

```
#include <ViennaRNA/backtrack/exterior.h>
```

unsigned int **vrna_bt_exterior_f3**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

```
#include <ViennaRNA/backtrack/exterior.h>
```

unsigned int **vrna_bt_exterior_f3_pp**(*vrna_fold_compound_t* *fc, unsigned int *i, unsigned int maxdist)

```
#include <ViennaRNA/backtrack/exterior.h>
```

int **vrna_backtrack_from_intervals**(*vrna_fold_compound_t* *fc, *vrna_bp_stack_t* *bp_stack, *sect* bt_stack[], int s)

Backtrack a secondary structure with pre-evaluated structure components.

```
#include <ViennaRNA/backtrack/global.h>
```

float **vrna_backtrack5**(*vrna_fold_compound_t* *fc, unsigned int length, char *structure)

Backtrack an MFE (sub)structure.

```
#include <ViennaRNA/backtrack/global.h>
```

This function allows one to backtrack the MFE structure for a (sub)sequence

SWIG Wrapper Notes:

This function is attached as overloaded method `backtrack()` to objects of type *The Fold Compound*. The parameter `length` defaults to the total length of the RNA sequence and may be omitted. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *`RNA.fold_compound.backtrack()`* in the *The RNA Python module*.

See also

vrna_mfe(), *vrna_pbacktrack5()*

Note

On error, the function returns INF / 100. and stores the empty string in `structure`.

Parameters

- **fc** – fold compound
- **length** – The length of the subsequence, starting from the 5' end
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to. (Must have size of at least $p length + 1)

Pre

Requires pre-filled MFE dynamic programming matrices, i.e. one has to call *vrna_mfe()* prior to calling this function

Returns

The minimum free energy (MFE) for the specified `length` in kcal/mol and a corresponding secondary structure in dot-bracket notation (stored in `structure`)

int **vrna_bt_hairpin**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, int en, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

Backtrack a hairpin loop closed by (i, j).

```
#include <ViennaRNA/backtrack/hairpin.h>
```

Note

This function is polymorphic! The provided *vrna_fold_compound_t* may be of type *VRNA_FC_TYPE_SINGLE* or *VRNA_FC_TYPE_COMPARATIVE*

int **vrna_bt_stacked_pairs**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, int *en, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

Backtrack a stacked pair closed by (i, j).

```
#include <ViennaRNA/backtrack/internal.h>
```

int **vrna_bt_internal_loop**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, int en, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

Backtrack an internal loop closed by (i, j).

```
#include <ViennaRNA/backtrack/internal.h>
```

unsigned int **vrna_bt_m**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

```
#include <ViennaRNA/backtrack/multibranch.h>
```

unsigned int **vrna_bt_multibranch_loop**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, int en, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

Backtrack the decomposition of a multi branch loop closed by (i, j).

```
#include <ViennaRNA/backtrack/multibranch.h>
```

Parameters

- **fc** – The *vrna_fold_compound_t* filled with all relevant data for backtracking
- **i** – 5' position of base pair closing the loop (will be set to 5' position of leftmost decomposed block upon successful backtracking)
- **j** – 3' position of base pair closing the loop (will be set to 3' position of rightmost decomposed block upon successful backtracking)
- **en** – The energy contribution of the substructure enclosed by (i, j)
- **bp_stack** – The base pair stack
- **bt_stack** – The backtrack stack

Returns

1, if backtracking succeeded, 0 otherwise.

unsigned int **vrna_bt_multibranch_split**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

```
#include <ViennaRNA/backtrack/multibranch.h>
```

Deprecated MFE API

Unnamed Group

float **alifold**(const char **strings, char *structure)

Compute MFE and according consensus structure of an alignment of sequences.

```
#include <ViennaRNA/alifold.h>
```

This function predicts the consensus structure for the aligned 'sequences' and returns the minimum free energy; the mfe structure in bracket notation is returned in 'structure'.

Sufficient space must be allocated for 'structure' before calling *alifold()*.

Deprecated:

Usage of this function is discouraged! Use *vrna_alifold()*, or *vrna_mfe()* instead!

See also

vrna_alifold(), *vrna_mfe()*

Parameters

- **strings** – A pointer to a NULL terminated array of character arrays
- **structure** – A pointer to a character array that may contain a constraining consensus structure (will be overwritten by a consensus structure that exhibits the MFE)

Returns

The free energy score in kcal/mol

float **circalifold**(const char **strings, char *structure)

Compute MFE and according structure of an alignment of sequences assuming the sequences are circular instead of linear.

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

Usage of this function is discouraged! Use vrna_alicircfold(), and *vrna_mfe()* instead!

See also

vrna_alicircfold(), *vrna_alifold()*, *vrna_mfe()*

Parameters

- **strings** – A pointer to a NULL terminated array of character arrays
- **structure** – A pointer to a character array that may contain a constraining consensus structure (will be overwritten by a consensus structure that exhibits the MFE)

Returns

The free energy score in kcal/mol

void **free_alifold_arrays**(void)

Free the memory occupied by MFE alifold functions.

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

Usage of this function is discouraged! It only affects memory being free'd that was allocated by an old API function before. Release of memory occupied by the newly introduced *vrna_fold_compound_t* is handled by *vrna_fold_compound_free()*

See also

vrna_fold_compound_free()

Functions

void **update_alifold_params**(void)

Update the energy parameters for alifold function.

```
#include <ViennaRNA/alifold.h>
```

Call this to recalculate the pair matrix and energy parameters after a change in folding parameters like *temperature*

Deprecated:

Usage of this function is discouraged! The new API uses *vrna_fold_compound_t* to lump all folding related necessities together, including the energy parameters. Use vrna_update_fold_params() to update the energy parameters within a *vrna_fold_compound_t*.

float **cofold**(const char *sequence, char *structure)

Compute the minimum free energy of two interacting RNA molecules.

```
#include <ViennaRNA/cofold.h>
```

The code is analog to the *fold()* function. If cut_point ==-1 results should be the same as with *fold()*.

Deprecated:

use *vrna_mfe_dimer()* instead

Parameters

- **sequence** – The two sequences concatenated
- **structure** – Will hold the barcket dot structure of the dimer molecule

Returns

minimum free energy of the structure

float **cofold_par**(const char *string, char *structure, *vrna_param_t* *parameters, int is_constrained)

Compute the minimum free energy of two interacting RNA molecules.

```
#include <ViennaRNA/cofold.h>
```

Deprecated:

use *vrna_mfe_dimer()* instead

void **free_co_arrays**(void)

Free memory occupied by *cofold()*.

```
#include <ViennaRNA/cofold.h>
```

Deprecated:

This function will only free memory allocated by a prior call of *cofold()* or *cofold_par()*. See *vrna_mfe_dimer()* for how to use the new API

See also

vrna_fc_destroy(), *vrna_mfe_dimer()*

Note

folding matrices now reside in the fold compound, and should be free'd there

void **update_cofold_params**(void)

Recalculate parameters.

```
#include <ViennaRNA/cofold.h>
```

Deprecated:

See *vrna_params_subst()* for an alternative using the new API

void **update_cofold_params_par**(*vrna_param_t* *parameters)

Recalculate parameters.

```
#include <ViennaRNA/cofold.h>
```

Deprecated:

See *vrna_params_subst()* for an alternative using the new API

void **export_cofold_arrays_gq**(int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **fc_p, int **ggg_p, int **indx_p, char **ptype_p)

Export the arrays of partition function cofold (with gquadruplex support).

```
#include <ViennaRNA/cofold.h>
```

Export the cofold arrays for use e.g. in the concentration Computations or suboptimal secondary structure backtracking

Deprecated:

folding matrices now reside within the fold compound. Thus, this function will only work in conjunction with a prior call to *cofold()* or *cofold_par()*

See also

vrna_mfe_dimer() for the new API

Parameters

- **f5_p** – A pointer to the 'f5' array, i.e. array conatining best free energy in interval [1,j]
- **c_p** – A pointer to the 'c' array, i.e. array containing best free energy in interval [i,j] given that i pairs with j
- **fML_p** – A pointer to the 'M' array, i.e. array containing best free energy in interval [i,j] for any multiloop segment with at least one stem
- **fM1_p** – A pointer to the 'M1' array, i.e. array containing best free energy in interval [i,j] for multiloop segment with exactly one stem
- **fc_p** – A pointer to the 'fc' array, i.e. array ...
- **ggg_p** – A pointer to the 'ggg' array, i.e. array containing best free energy of a gquadruplex delimited by [i,j]
- **indx_p** – A pointer to the indexing array used for accessing the energy matrices
- **ptype_p** – A pointer to the ptype array containing the base pair types for each possibility (i,j)

void **export_cofold_arrays**(int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **fc_p, int **indx_p, char **ptype_p)

Export the arrays of partition function cofold.

```
#include <ViennaRNA/cofold.h>
```

Export the cofold arrays for use e.g. in the concentration Computations or suboptimal secondary structure backtracking

Deprecated:

folding matrices now reside within the *vrna_fold_compound_t*. Thus, this function will only work in conjunction with a prior call to the deprecated functions *cofold()* or *cofold_par()*

See also

vrna_mfe_dimer() for the new API

Parameters

- **f5_p** – A pointer to the 'f5' array, i.e. array conatining best free energy in interval [1,j]
- **c_p** – A pointer to the 'c' array, i.e. array containing best free energy in interval [i,j] given that i pairs with j
- **fML_p** – A pointer to the 'M' array, i.e. array containing best free energy in interval [i,j] for any multiloop segment with at least one stem
- **fM1_p** – A pointer to the 'M1' array, i.e. array containing best free energy in interval [i,j] for multiloop segment with exactly one stem
- **fc_p** – A pointer to the 'fc' array, i.e. array ...
- **indx_p** – A pointer to the indexing array used for accessing the energy matrices
- **ptype_p** – A pointer to the ptype array containing the base pair types for each possibility (i,j)

void **initialize_cofold**(int length)

```
#include <ViennaRNA/cofold.h>
```

allocate arrays for folding

Deprecated:

float **fold_par**(const char *sequence, char *structure, *vrna_param_t* *parameters, int is_constrained, int is_circular)

Compute minimum free energy and an appropriate secondary structure of an RNA sequence.

```
#include <ViennaRNA/fold.h>
```

The first parameter given, the RNA sequence, must be *uppercase* and should only contain an alphabet Σ that is understood by the RNAlib

(e.g. $\Sigma = \{A, U, C, G\}$)

The second parameter, *structure*, must always point to an allocated block of memory with a size of at least $\text{strlen}(\text{sequence}) + 1$

If the third parameter is NULL, global model detail settings are assumed for the folding recursions. Otherwise, the provided parameters are used.

The fourth parameter indicates whether a secondary structure constraint in enhanced dot-bracket notation is passed through the structure parameter or not. If so, the characters " | x < > " are recognized to mark bases that are paired, unpaired, paired upstream, or downstream, respectively. Matching brackets " ( ) " denote base pairs, dots "." are used for unconstrained bases.

To indicate that the RNA sequence is circular and thus has to be post-processed, set the last parameter to non-zero

After a successful call of *fold_par()*, a backtracked secondary structure (in dot-bracket notation) that exhibits the minimum of free energy will be written to the memory *structure* is pointing to. The function returns the minimum of free energy for any fold of the sequence given.

Deprecated:

use *vrna_mfe()* instead!

See also

vrna_mfe(), *fold()*, *circfold()*, *vrna_md_t*, set_energy_model(), *get_scaled_parameters()*

Note

OpenMP: Passing NULL to the 'parameters' argument involves access to several global model detail variables and thus is not to be considered threadsafe

Parameters

- **sequence** – RNA sequence
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to
- **parameters** – A data structure containing the pre-scaled energy contributions and the model details. (NULL may be passed, see OpenMP notes above)
- **is_constrained** – Switch to indicate that a structure constraint is passed via the structure argument (0==off)
- **is_circular** – Switch to (de-)activate post-processing steps in case RNA sequence is circular (0==off)

Returns

the minimum free energy (MFE) in kcal/mol

float **fold**(const char *sequence, char *structure)

Compute minimum free energy and an appropriate secondary structure of an RNA sequence.

```
#include <ViennaRNA/fold.h>
```

This function essentially does the same thing as *fold_par()*. However, it takes its model details, i.e. *temperature*, *dangles*, *tetra_loop*, *noGU*, *no_closingGU*, fold_constrained, *noLonelyPairs* from the current global settings within the library

Deprecated:

use *vrna_fold()*, or *vrna_mfe()* instead!

See also

fold_par(), *circfold()*

Parameters

- **sequence** – RNA sequence
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns

the minimum free energy (MFE) in kcal/mol

float **circfold**(const char *sequence, char *structure)

Compute minimum free energy and an appropriate secondary structure of a circular RNA sequence.

```
#include <ViennaRNA/fold.h>
```

This function essentially does the same thing as *fold_par()*. However, it takes its model details, i.e. *temperature*, *dangles*, *tetra_loop*, *noGU*, *no_closingGU*, fold_constrained, *noLonelyPairs* from the current global settings within the library

Deprecated:

Use *vrna_circfold()*, or *vrna_mfe()* instead!

See also

fold_par(), *circfold()*

Parameters

- **sequence** – RNA sequence
- **structure** – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns

the minimum free energy (MFE) in kcal/mol

void **free_arrays**(void)

Free arrays for mfe folding.

```
#include <ViennaRNA/fold.h>
```

Deprecated:

See *vrna_fold()*, *vrna_circfold()*, or *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

void **update_fold_params**(void)

Recalculate energy parameters.

```
#include <ViennaRNA/fold.h>
```

Deprecated:

For non-default model settings use the new API with *vrna_params_subst()* and *vrna_mfe()* instead!

void **update_fold_params_par**(*vrna_param_t* *parameters)

Recalculate energy parameters.

```
#include <ViennaRNA/fold.h>
```

Deprecated:

For non-default model settings use the new API with *vrna_params_subst()* and *vrna_mfe()* instead!

void **export_fold_arrays**(int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **indx_p, char **ptype_p)

```
#include <ViennaRNA/fold.h>
```

Deprecated:

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

void **export_fold_arrays_par**(int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **indx_p, char **ptype_p, *vrna_param_t* **P_p)

```
#include <ViennaRNA/fold.h>
```

Deprecated:

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

void **export_circfold_arrays**(int *Fc_p, int *FcH_p, int *FcI_p, int *FcM_p, int **fM2_p, int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **indx_p, char **ptype_p)

```
#include <ViennaRNA/fold.h>
```

Deprecated:

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

void **export_circfold_arrays_par**(int *Fc_p, int *FcH_p, int *FcI_p, int *FcM_p, int **fM2_p, int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **indx_p, char **ptype_p, *vrna_param_t* **P_p)

```
#include <ViennaRNA/fold.h>
```

Deprecated:

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

int **LoopEnergy**(int n1, int n2, int type, int type_2, int si1, int sj1, int sp1, int sq1)

```
#include <ViennaRNA/fold.h>
```

Deprecated:
{This function is deprecated and will be removed soon. Use *vrna_E_internal()* instead!}

int **HairpinE**(int size, int type, int si1, int sj1, const char *string)

```
#include <ViennaRNA/fold.h>
```

Deprecated:
{This function is deprecated and will be removed soon. Use *vrna_E_hairpin()* instead!}

void **initialize_fold**(int length)

```
#include <ViennaRNA/fold.h>
```

Allocate arrays for folding

Deprecated:
See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

char ***backtrack_fold_from_pair**(char *sequence, int i, int j)

```
#include <ViennaRNA/fold.h>
```

Functions

float **Lfold**(const char *string, const char *structure, int maxdist)
The local analog to *fold()*.

```
#include <ViennaRNA/Lfold.h>
```

Computes the minimum free energy structure including only base pairs with a span smaller than 'maxdist'

Deprecated:
Use *vrna_mfe_window()* instead!

float **aliLfold**(const char **AS, const char *structure, int maxdist)

```
#include <ViennaRNA/Lfold.h>
```

float **aliLfold_cb**(const char **AS, int maxdist, *vrna_mfe_window_f* cb, void *data)

```
#include <ViennaRNA/Lfold.h>
```

4.1.2 Partition Function and Equilibrium Probabilities

Basic Partition Function API

Functions

int **vrna_pf_float_precision**(void)

Find out whether partition function computations are using single precision floating points.

```
#include <ViennaRNA/partfunc/global.h>
```

See also

FLT_OR_DBL

Returns

1 if single precision is used, 0 otherwise

Global Partition Function

Basic global partition function interface

FLT_OR_DBL **vrna_pf**(*vrna_fold_compound_t* *fc, char *structure)

Compute the partition function Q for a given RNA sequence, or sequence alignment.

```
#include <ViennaRNA/partfunc/global.h>
```

If *structure* is not a NULL pointer on input, it contains on return a string consisting of the letters " . , | { } ( ) " denoting bases that are essentially unpaired, weakly paired, strongly paired without preference, weakly upstream (downstream) paired, or strongly up- (down-)stream paired bases, respectively. If the model's compute_bpp is set to 0 base pairing probabilities will not be computed (saving CPU time), otherwise after calculations took place pr will contain the probability that bases i and j pair.

SWIG Wrapper Notes:

This function is attached as method `pf()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.pf()* in the *The RNA Python module*.

See also

vrna_fold_compound_t, *vrna_fold_compound()*, *vrna_pf_fold()*, *vrna_pf_circfold()*, *vrna_fold_compound_comparative()*, *vrna_pf_alifold()*, *vrna_pf_circalifold()*, *vrna_db_from_probs()*, *vrna_exp_params()*, *vrna_aln_pinfo()*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*. Also, this function may return INF / 100. in case of contradicting constraints or numerical over-/underflow. In the latter case, a corresponding warning will be issued to `stdout`.

Parameters

- **fc** – **[inout]** The fold compound data structure

- **structure** – **[inout]** A pointer to the character array where position-wise pairing propensity will be stored. (Maybe NULL)

Returns

The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

vrna_dimer_pf_t **vrna_pf_dimer**(*vrna_fold_compound_t* *fc, char *structure)

Calculate partition function and base pair probabilities of nucleic acid/nucleic acid dimers.

```
#include <ViennaRNA/partfunc/global.h>
```

This is the cofold partition function folding.

SWIG Wrapper Notes:

This function is attached as method `pf_dimer()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.pf_dimer()* in the *The RNA Python module*.

See also

vrna_fold_compound() for how to retrieve the necessary data structure

Note

This function may return INF / 100. for the `FA`, `FB`, `FAB`, `F0AB` members of the output data structure in case of contradicting constraints or numerical over-/underflow. In the latter case, a corresponding warning will be issued to `stdout`.

Parameters

- **fc** – the fold compound data structure
- **structure** – Will hold the structure or constraints

Returns

vrna_dimer_pf_t structure containing a set of energies needed for concentration computations.

FLT_OR_DBL ***vrna_pf_substrands**(*vrna_fold_compound_t* *fc, size_t complex_size)

```
#include <ViennaRNA/partfunc/global.h>
```

FLT_OR_DBL **vrna_pf_add**(*FLT_OR_DBL* dG1, *FLT_OR_DBL* dG2, double kT)

```
#include <ViennaRNA/partfunc/global.h>
```

Simplified global partition function computation using sequence(s) or multiple sequence alignment(s)

float **vrna_pf_fold**(const char *sequence, char *structure, *vrna_ep_t* **pl)

Compute Partition function Q (and base pair probabilities) for an RNA sequence using a comparative method.

```
#include <ViennaRNA/partfunc/global.h>
```

This simplified interface to *vrna_pf()* computes the partition function and, if required, base pair probabilities for an RNA sequence using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing.

See also

vrna_pf_circfold(), *vrna_pf()*, *vrna_fold_compound()*, *vrna_fold_compound_t*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_pf()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequence** – RNA sequence
- **structure** – A pointer to the character array where position-wise pairing propensity will be stored. (Maybe NULL)
- **pl** – A pointer to a list of *vrna_ep_t* to store pairing probabilities (Maybe NULL)

Returns

The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

float **vrna_pf_circfold**(const char *sequence, char *structure, *vrna_ep_t* **pl)

Compute Partition function Q (and base pair probabilities) for a circular RNA sequences using a comparative method.

```
#include <ViennaRNA/partfunc/global.h>
```

This simplified interface to *vrna_pf()* computes the partition function and, if required, base pair probabilities for a circular RNA sequence using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing.

Folding of circular RNA sequences is handled as a post-processing step of the forward recursions. See Hofacker and Stadler [2006] for further details.

See also

vrna_pf_fold(), *vrna_pf()*, *vrna_fold_compound()*, *vrna_fold_compound_t*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_pf()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequence** – A circular RNA sequence
- **structure** – A pointer to the character array where position-wise pairing propensity will be stored. (Maybe NULL)
- **pl** – A pointer to a list of *vrna_ep_t* to store pairing probabilities (Maybe NULL)

Returns

The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

float **vrna_pf_alifold**(const char **sequences, char *structure, *vrna_ep_t* **pl)

Compute Partition function Q (and base pair probabilities) for an RNA sequence alignment using a comparative method.

```
#include <ViennaRNA/partfunc/global.h>
```

This simplified interface to *vrna_pf()* computes the partition function and, if required, base pair probabilities for an RNA sequence alignment using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing.

See also

vrna_pf_circalifold(), *vrna_pf()*, *vrna_fold_compound_comparative()*, *vrna_fold_compound_t*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_pf()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequences** – RNA sequence alignment
- **structure** – A pointer to the character array where position-wise pairing propensity will be stored. (Maybe NULL)
- **pl** – A pointer to a list of *vrna_ep_t* to store pairing probabilities (Maybe NULL)

Returns

The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

float **vrna_pf_circalifold**(const char **sequences, char *structure, *vrna_ep_t* **pl)

Compute Partition function Q (and base pair probabilities) for an alignment of circular RNA sequences using a comparative method.

```
#include <ViennaRNA/partfunc/global.h>
```

This simplified interface to *vrna_pf()* computes the partition function and, if required, base pair probabilities for an RNA sequence alignment using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing.

Folding of circular RNA sequences is handled as a post-processing step of the forward recursions. See Hofacker and Stadler [2006] for further details.

See also

vrna_pf_alifold(), *vrna_pf()*, *vrna_fold_compound_comparative()*, *vrna_fold_compound_t*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_pf()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **sequences** – Sequence alignment of circular RNAs
- **structure** – A pointer to the character array where position-wise pairing propensity will be stored. (Maybe NULL)
- **pl** – A pointer to a list of *vrna_ep_t* to store pairing probabilities (Maybe NULL)

Returns

The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

vrna_dimer_pf_t **vrna_pf_co_fold**(const char *seq, char *structure, *vrna_ep_t* **pl)

Calculate partition function and base pair probabilities of nucleic acid/nucleic acid dimers.

```
#include <ViennaRNA/partfunc/global.h>
```

This simplified interface to *vrna_pf_dimer()* computes the partition function and, if required, base pair probabilities for an RNA-RNA interaction using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing.

See also

vrna_pf_dimer()

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_pf_dimer()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **seq** – Two concatenated RNA sequences with a delimiting '&' in between
- **structure** – A pointer to the character array where position-wise pairing propensity will be stored. (Maybe NULL)
- **pl** – A pointer to a list of *vrna_ep_t* to store pairing probabilities (Maybe NULL)

Returns

vrna_dimer_pf_t structure containing a set of energies needed for concentration computations.

Functions

vrna_ep_t ***vrna_plist_from_probs**(*vrna_fold_compound_t* *fc, double cut_off)

Create a *vrna_ep_t* from base pair probability matrix.

```
#include <ViennaRNA/structures/problist.h>
```

The probability matrix provided via the *vrna_fold_compound_t* is parsed and all pair probabilities above the given threshold are used to create an entry in the plist

The end of the plist is marked by sequence positions i as well as j equal to 0. This condition should be used to stop looping over its entries

Parameters

- **fc** – **[in]** The fold compound
- **cut_off** – **[in]** The cutoff value

Returns

A pointer to the plist that is to be created

struct **vrna_dimer_pf_s**

#include <ViennaRNA/partfunc/global.h> Data structure returned by *vrna_pf_dimer()*.

Public Members

double **F0AB**

Null model without DuplexInit.

```
#include <ViennaRNA/partfunc/global.h>
```

double **FAB**

all states with DuplexInit correction

```
#include <ViennaRNA/partfunc/global.h>
```

double **FcAB**

true hybrid states only

```
#include <ViennaRNA/partfunc/global.h>
```

double **FA**

monomer A

```
#include <ViennaRNA/partfunc/global.h>
```

double **FB**

monomer B

```
#include <ViennaRNA/partfunc/global.h>
```

struct **vrna_multimer_pf_s**

Public Members

double **F_connected**

Fully connected ensemble (incl. DuplexInititiation and rotational symmetry correction.

```
#include <ViennaRNA/partfunc/global.h>
```

double ***F_monomers**

monomers

```
#include <ViennaRNA/partfunc/global.h>
```

size_t **num_monomers**

Number of monomers.

```
#include <ViennaRNA/partfunc/global.h>
```

Local Partition Function

Basic local partition function interface

int **vrna_probs_window**(*vrna_fold_compound_t* *fc, int ulength, unsigned int options, *vrna_probs_window_f* cb, void *data)

Compute various equilibrium probabilities under a sliding window approach.

```
#include <ViennaRNA/partfunc/local.h>
```

This function applies a sliding window scan for the sequence provided with the argument `fc` and reports back equilibrium probabilities through the callback function `cb`. The data reported to the callback depends on the `options` flag.

Options:

- *VRNA_PROBS_WINDOW_BPP* - Trigger base pairing probabilities.
- *VRNA_PROBS_WINDOW_UP* - Trigger unpaired probabilities.
- *VRNA_PROBS_WINDOW_UP_SPLIT* - Trigger detailed unpaired probabilities split up into different loop type contexts.

Options may be OR-ed together

See also

vrna_pfl_fold_cb(), *vrna_pfl_fold_up_cb()*

Note

The parameter `ulength` only affects computation and resulting data if unpaired probability computations are requested through the `options` flag.

Parameters

- **fc** – The fold compound with sequence data, model settings and precomputed energy parameters
- **ulength** – The maximal length of an unpaired segment (only for unpaired probability computations)
- **cb** – The callback function which collects the pair probability data for further processing
- **data** – Some arbitrary data structure that is passed to the callback `cb`
- **options** – Option flags to control the behavior of this function

Returns

0 on failure, non-zero on success

Simplified global partition function computation using sequence(s) or multiple sequence alignment(s)

vrna_ep_t ***vrna_pfl_fold**(const char *sequence, int window_size, int max_bp_span, float cutoff)

Compute base pair probabilities using a sliding-window approach.

```
#include <ViennaRNA/partfunc/local.h>
```

This is a simplified wrapper to *vrna_probs_window()* that given a nucleid acid sequence, a window size, a maximum base pair span, and a cutoff value computes the pair probabilities for any base pair in any window. The pair probabilities are returned as a list and the user has to take care to free() the memory occupied by the list.

See also

vrna_probs_window(), *vrna_pfl_fold_cb()*, *vrna_pfl_fold_up()*

Note

This function uses default model settings! For custom model settings, we refer to the function *vrna_probs_window()* .

In case of any computation errors, this function returns `NULL`

Parameters

- **sequence** – The nucleic acid input sequence
- **window_size** – The size of the sliding window
- **max_bp_span** – The maximum distance along the backbone between two nucleotides that form a base pairs
- **cutoff** – A cutoff value that omits all pairs with lower probability

Returns

A list of base pair probabilities, terminated by an entry with *vrna_ep_t.i* and *vrna_ep_t.j* set to 0

int **vrna_pfl_fold_cb**(const char *sequence, int window_size, int max_bp_span, *vrna_probs_window_f* cb, void *data)

Compute base pair probabilities using a sliding-window approach (callback version).

```
#include <ViennaRNA/partfunc/local.h>
```

This is a simplified wrapper to *vrna_probs_window()* that given a nucleid acid sequence, a window size, a maximum base pair span, and a cutoff value computes the pair probabilities for any base pair in any window. It is similar to *vrna_pfl_fold()* but uses a callback mechanism to return the pair probabilities.

Read the details for *vrna_probs_window()* for details on the callback implementation!

See also

vrna_probs_window(), *vrna_pfl_fold()*, *vrna_pfl_fold_up_cb()*

Note

This function uses default model settings! For custom model settings, we refer to the function *vrna_probs_window()*.

Parameters

- **sequence** – The nucleic acid input sequence
- **window_size** – The size of the sliding window
- **max_bp_span** – The maximum distance along the backbone between two nucleotides that form a base pairs
- **cb** – The callback function which collects the pair probability data for further processing
- **data** – Some arbitrary data structure that is passed to the callback `cb`

Returns

0 on failure, non-zero on success

double ****vrna_pfl_fold_up**(const char *sequence, int ulength, int window_size, int max_bp_span)

Compute probability of contiguous unpaired segments.

```
#include <ViennaRNA/partfunc/local.h>
```

This is a simplified wrapper to *vrna_probs_window()* that given a nucleic acid sequence, a maximum length of unpaired segments (`ulength`), a window size, and a maximum base pair span computes the equilibrium probability of any segment not exceeding `ulength`. The probabilities to be unpaired are returned as a 1-based, 2-dimensional matrix with dimensions $N \times M$, where N is the length of the sequence and M is the maximum segment length. As an example, the probability of a segment of size 5 starting at position 100 is stored in the matrix entry $X[100][5]$.

It is the users responsibility to free the memory occupied by this matrix.

> **Note**
>
> This function uses default model settings! For custom model settings, we refer to the function *vrna_probs_window()*.

Parameters

- **sequence** – The nucleic acid input sequence
- **ulength** – The maximal length of an unpaired segment
- **window_size** – The size of the sliding window
- **max_bp_span** – The maximum distance along the backbone between two nucleotides that form a base pairs

Returns

The probabilities to be unpaired for any segment not exceeding `ulength`

int **vrna_pfl_fold_up_cb**(const char *sequence, int ulength, int window_size, int max_bp_span, *vrna_probs_window_f* cb, void *data)

Compute probability of contiguous unpaired segments.

```
#include <ViennaRNA/partfunc/local.h>
```

This is a simplified wrapper to *vrna_probs_window()* that given a nucleic acid sequence, a maximum length of unpaired segments (`ulength`), a window size, and a maximum base pair span computes the equilibrium probability of any segment not exceeding `ulength`. It is similar to *vrna_pfl_fold_up()* but uses a callback mechanism to return the unpaired probabilities.

Read the details for *vrna_probs_window()* for details on the callback implementation!

> **Note**
>
> This function uses default model settings! For custom model settings, we refer to the function *vrna_probs_window()*.

Parameters

- **sequence** – The nucleic acid input sequence
- **ulength** – The maximal length of an unpaired segment
- **window_size** – The size of the sliding window
- **max_bp_span** – The maximum distance along the backbone between two nucleotides that form a base pairs
- **cb** – The callback function which collects the pair probability data for further processing
- **data** – Some arbitrary data structure that is passed to the callback `cb`

Returns

0 on failure, non-zero on success

Defines

VRNA_EXT_LOOP

Exterior loop.

```
#include <ViennaRNA/partfunc/local.h>
```

VRNA_HP_LOOP

Hairpin loop.

```
#include <ViennaRNA/partfunc/local.h>
```

VRNA_INT_LOOP

Internal loop.

```
#include <ViennaRNA/partfunc/local.h>
```

VRNA_MB_LOOP

Multibranch loop.

```
#include <ViennaRNA/partfunc/local.h>
```

VRNA_ANY_LOOP

Any loop.

```
#include <ViennaRNA/partfunc/local.h>
```

VRNA_PROBS_WINDOW_BPP

Trigger base pairing probabilities.

```
#include <ViennaRNA/partfunc/local.h>
```

Passing this flag to *vrna_probs_window()* activates callback execution for base pairing probabilities. In turn, the corresponding callback receives this flag through the `type` argument whenever base pairing probabilities are provided.

Detailed information for the algorithm to compute unpaired probabilities can be taken from Bernhart *et al.* [2005] .

See also

vrna_probs_window()

VRNA_PROBS_WINDOW_UP

Trigger unpaired probabilities.

```
#include <ViennaRNA/partfunc/local.h>
```

Passing this flag to *vrna_probs_window()* activates callback execution for unpaired probabilities. In turn, the corresponding callback receives this flag through the `type` argument whenever unpaired probabilities are provided.

Detailed information for the algorithm to compute unpaired probabilities can be taken from Bernhart *et al.* [2011] .

See also

vrna_probs_window()

VRNA_PROBS_WINDOW_STACKP

Trigger base pair stack probabilities.

```
#include <ViennaRNA/partfunc/local.h>
```

Passing this flag to *vrna_probs_window()* activates callback execution for stacking probabilities. In turn, the corresponding callback receives this flag through the `type` argument whenever stack probabilities are provided.

Bug:

Currently, this flag is a placeholder doing nothing as the corresponding implementation for stack probability computation is missing.

See also

vrna_probs_window()

VRNA_PROBS_WINDOW_UP_SPLIT

Trigger detailed unpaired probabilities split up into different loop type contexts.

```
#include <ViennaRNA/partfunc/local.h>
```

Passing this flag to *vrna_probs_window()* activates callback execution for unpaired probabilities. In contrast to *VRNA_PROBS_WINDOW_UP* this flag requests unpaired probabilities to be split up into different loop type contexts. In turn, the corresponding callback receives the *VRNA_PROBS_WINDOW_UP* flag OR-ed together with the corresponding loop type, i.e.:

- *VRNA_EXT_LOOP* - Exterior loop.
- *VRNA_HP_LOOP* - Hairpin loop.
- *VRNA_INT_LOOP* - Internal loop.
- *VRNA_MB_LOOP* - Multibranch loop.
- *VRNA_ANY_LOOP* - Any loop.

See also

vrna_probs_window(), *VRNA_PROBS_WINDOW_UP*

VRNA_PROBS_WINDOW_PF

Trigger partition function.

```
#include <ViennaRNA/partfunc/local.h>
```

Passing this flag to *vrna_probs_window()* activates callback execution for partition function. In turn, the corresponding callback receives this flag through it's `type` argument whenever partition function data is provided.

See also

vrna_probs_window()

Note

Instead of actually providing the partition function Z, the callback is always provided with the corresponding enemble free energy $\Delta G = -RT \ln Z$.

Typedefs

typedef void (***vrna_probs_window_f**)(*FLT_OR_DBL* *pr, int pr_size, int i, int max, unsigned int type, void *data)

Sliding window probability computation callback.

```
#include <ViennaRNA/partfunc/local.h>
```

Notes on Callback Functions:

This function will be called for each probability data set in the sliding window probability computation implementation of *vrna_probs_window()*. The argument *type* specifies the type of probability that is passed to this function.

Types:

- *VRNA_PROBS_WINDOW_BPP* - Trigger base pairing probabilities.
- *VRNA_PROBS_WINDOW_UP* - Trigger unpaired probabilities.
- *VRNA_PROBS_WINDOW_PF* - Trigger partition function.

The above types usually come exclusively. However, for unpaired probabilities, the *VRNA_PROBS_WINDOW_UP* flag is OR-ed together with one of the loop type contexts

- *VRNA_EXT_LOOP* - Exterior loop.
- *VRNA_HP_LOOP* - Hairpin loop.
- *VRNA_INT_LOOP* - Internal loop.
- *VRNA_MB_LOOP* - Multibranch loop.
- *VRNA_ANY_LOOP* - Any loop.

to indicate the particular type of data available through the `pr` pointer.

See also

vrna_probs_window(), *vrna_pfl_fold_up_cb()*

Param pr
An array of probabilities

Param pr_size
The length of the probability array

Param i
The i-position (5') of the probabilities

Param max
The (theoretical) maximum length of the probability array

Param type
The type of data that is provided

Param data
Auxiliary data

typedef void **vrna_probs_window_callback**(*FLT_OR_DBL* *pr, int pr_size, int i, int max, unsigned int type, void *data)

```
#include <ViennaRNA/partfunc/local.h>
```

Thermodynamic Properties API

Compute various thermodynamic properties using the partition function.

Basic heat capacity function interface

vrna_heat_capacity_t ***vrna_heat_capacity**(*vrna_fold_compound_t* *fc, float T_min, float T_max, float T_increment, unsigned int mpoints)

Compute the specific heat for an RNA.

```
#include <ViennaRNA/heat_capacity.h>
```

This function computes an RNAs specific heat in a given temperature range from the partition function by numeric differentiation. The result is returned as a list of pairs of temperature in °C and specific heat in Kcal/(Mol*K).

Users can specify the temperature range for the computation from `T_min` to `T_max`, as well as the increment step size `T_increment`. The latter also determines how many times the partition function is computed. Finally, the parameter `mpoints` determines how smooth the curve should be. The algorithm itself fits a parabola to $2 \cdot mpoints + 1$ data points to calculate 2nd derivatives. Increasing this parameter produces a smoother curve.

SWIG Wrapper Notes:

This function is attached as overloaded method `heat_capacity()` to objects of type `fold_compound`. If the optional function arguments `T_min`, `T_max`, `T_increment`, and `mpoints` are omitted, they default to 0.0, 100.0, 1.0 and 2, respectively. See, e.g. *`RNA.fold_compound.heat_capacity()`* in the *The RNA Python module*.

See also

vrna_heat_capacity_cb(), *vrna_heat_capacity_t*, *vrna_heat_capacity_s*

Parameters

- **fc** – The *vrna_fold_compound_t* with the RNA sequence to analyze
- **T_min** – Lowest temperature in °C
- **T_max** – Highest temperature in °C
- **T_increment** – Stepsize for temperature incrementation in °C (a reasonable choice might be 1°C)
- **mpoints** – The number of interpolation points to calculate 2nd derivative (a reasonable choice might be 2, min: 1, max: 100)

Returns

A list of pairs of temperatures and corresponding heat capacity or *NULL* upon any failure. The last entry of the list is indicated by a **temperature** field set to a value smaller than `T_min`

int **vrna_heat_capacity_cb**(*vrna_fold_compound_t* *fc, float T_min, float T_max, float T_increment, unsigned int mpoints, *vrna_heat_capacity_f* cb, void *data)

Compute the specific heat for an RNA (callback variant).

```
#include <ViennaRNA/heat_capacity.h>
```

Similar to *vrna_heat_capacity()*, this function computes an RNAs specific heat in a given temperature range from the partition function by numeric differentiation. Instead of returning a list of temperature/specific heat pairs, however, this function returns the individual results through a callback mechanism. The provided function will be called for each result and passed the corresponding temperature and specific heat values along with the arbitrary data as provided through the `data` pointer argument.

Users can specify the temperature range for the computation from `T_min` to `T_max`, as well as the increment step size `T_increment`. The latter also determines how many times the partition function is computed. Finally, the parameter `mpoints` determines how smooth the curve should be. The algorithm itself fits a parabola to $2 \cdot mpoints + 1$ data points to calculate 2nd derivatives. Increasing this parameter produces a smoother curve.

SWIG Wrapper Notes:

This function is attached as method `heat_capacity_cb()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.heat_capacity_cb()* in the *The RNA Python module*.

See also

vrna_heat_capacity(), *vrna_heat_capacity_f*

Parameters

- **fc** – The *vrna_fold_compound_t* with the RNA sequence to analyze
- **T_min** – Lowest temperature in °C
- **T_max** – Highest temperature in °C
- **T_increment** – Stepsize for temperature incrementation in °C (a reasonable choice might be 1°C)

- **mpoints** – The number of interpolation points to calculate 2nd derivative (a reasonable choice might be 2, min: 1, max: 100)
- **cb** – The user-defined callback function that receives the individual results
- **data** – An arbitrary data structure that will be passed to the callback in conjunction with the results

Returns
Returns 0 upon failure, and non-zero otherwise

Simplified heat capacity computation

vrna_heat_capacity_t ***vrna_heat_capacity_simple**(const char *sequence, float T_min, float T_max, float T_increment, unsigned int mpoints)

Compute the specific heat for an RNA (simplified variant).

```
#include <ViennaRNA/heat_capacity.h>
```

Similar to *vrna_heat_capacity()*, this function computes an RNAs specific heat in a given temperature range from the partition function by numeric differentiation. This simplified version, however, only requires the RNA sequence as input instead of a *vrna_fold_compound_t* data structure. The result is returned as a list of pairs of temperature in °C and specific heat in Kcal/(Mol*K).

Users can specify the temperature range for the computation from `T_min` to `T_max`, as well as the increment step size `T_increment`. The latter also determines how many times the partition function is computed. Finally, the parameter `mpoints` determines how smooth the curve should be. The algorithm itself fits a parabola to $2 \cdot mpoints + 1$ data points to calculate 2nd derivatives. Increasing this parameter produces a smoother curve.

SWIG Wrapper Notes:

This function is available as overloaded function `heat_capacity()`. If the optional function arguments `T_min`, `T_max`, `T_increment`, and `mpoints` are omitted, they default to 0.0, 100.0, 1.0 and 2, respectively. See, e.g. `RNA.head_capacity()` in the *The RNA Python module*.

See also

vrna_heat_capacity(), *vrna_heat_capacity_cb()*, *vrna_heat_capacity_t*, *vrna_heat_capacity_s*

Parameters

- **sequence** – The RNA sequence input (must be uppercase)
- **T_min** – Lowest temperature in °C
- **T_max** – Highest temperature in °C
- **T_increment** – Stepsize for temperature incrementation in °C (a reasonable choice might be 1°C)
- **mpoints** – The number of interpolation points to calculate 2nd derivative (a reasonable choice might be 2, min: 1, max: 100)

Returns
A list of pairs of temperatures and corresponding heat capacity or *NULL* upon any failure. The last entry of the list is indicated by a **temperature** field set to a value smaller than `T_min`

Accessibility and derived computations

vrna_ep_array_t **vrna_eqm_probs_up**(*vrna_fold_compound_t* *fc, unsigned int min_u, unsigned int max_u, double cutoff)

Compute Accessibility.

```
#include <ViennaRNA/probabilities/accessibility.h>
```

Parameters

- **fc** – The fold compound to operate on
- **min_u** – The minimum sequence length for unpaired stretches
- **max_u** – The maximum sequence length for unpaired stretches
- **cutoff** – A probability threshold to filter the output

Returns

A list of accessibility values in *vrna_array()* format

Base pair probabilities and derived computations

int **vrna_pairing_probs**(*vrna_fold_compound_t* *fc, char *structure)

```
#include <ViennaRNA/probabilities/basepairs.h>
```

vrna_ep_array_t **vrna_eqm_probs_bp**(*vrna_fold_compound_t* *fc, double cutoff)

Get a list of base pair probabiltities.

```
#include <ViennaRNA/probabilities/basepairs.h>
```

Parameters

- **fc** – The fold compound to operate on
- **cutoff** – A probability threshold to filter the output

Returns

A list of base pair probabiltities

Multimer probabilities computations

void **vrna_pf_dimer_probs**(double FAB, double FA, double FB, *vrna_ep_t* *prAB, const *vrna_ep_t* *prA, const *vrna_ep_t* *prB, int Alength, const *vrna_exp_param_t* *exp_params)

Compute Boltzmann probabilities of dimerization without homodimers.

```
#include <ViennaRNA/probabilities/basic.h>
```

Given the pair probabilities and free energies (in the null model) for a dimer AB and the two constituent monomers A and B, compute the conditional pair probabilities given that a dimer AB actually forms. Null model pair probabilities are given as a list as produced by *vrna_plist_from_probs()*, the dimer probabilities 'prAB' are modified in place.

Parameters

- **FAB** – free energy of dimer AB
- **FA** – free energy of monomer A
- **FB** – free energy of monomer B
- **prAB** – pair probabilities for dimer

- **prA** – pair probabilities monomer
- **prB** – pair probabilities monomer
- **Alength** – Length of molecule A
- **exp_params** – The precomputed Boltzmann factors

Helix probabilities and derived computations

vrna_ep_array_t **vrna_eqm_probs_hx**(*vrna_fold_compound_t* *fc, unsigned int min_size, unsigned int max_size, double cutoff)

Get a list of helix probabiltities.

```
#include <ViennaRNA/probabilities/helices.h>
```

Parameters

- **fc** – The fold compound to operate on
- **min_size** – The minimum helix size
- **max_size** – The maximum helix size
- **cutoff** – A probability threshold to filter the output

Returns
A list of base pair probabiltities

vrna_ep_t ***vrna_stack_prob**(*vrna_fold_compound_t* *fc, double cutoff)

Compute stacking probabilities.

```
#include <ViennaRNA/probabilities/helices.h>
```

For each possible base pair (i, j), compute the probability of a stack (i, j), $(i + 1, j - 1)$.

SWIG Wrapper Notes:

This function is attached as overloaded method `stack_prob()` to objects of type `fold_compound`. The optional argument `cutoff` defaults to 1e-5. See, e.g. *RNA.fold_compound.stack_prob()* in the *The RNA Python module*.

Parameters

- **fc** – The fold compound data structure with precomputed base pair probabilities
- **cutoff** – A cutoff value that limits the output to stacks with $p >$ cutoff.

Returns
A list of stacks with enclosing base pair (i, j) and probabiltiy p

Structure probability computations

double **vrna_mean_bp_distance_pr**(int length, *FLT_OR_DBL* *pr)

Get the mean base pair distance in the thermodynamic ensemble from a probability matrix.

```
#include <ViennaRNA/probabilities/structures.h>
```

$$< d >= \sum_{a,b} p_a p_b d(S_a, S_b)$$

this can be computed from the pair probs p_{ij} as

$$< d >= \sum_{ij} p_{ij}(1 - p_{ij})$$

Parameters

- **length** – The length of the sequence
- **pr** – The matrix containing the base pair probabilities

Returns
The mean pair distance of the structure ensemble

double **vrna_mean_bp_distance**(*vrna_fold_compound_t* *fc)

Get the mean base pair distance in the thermodynamic ensemble.

```
#include <ViennaRNA/probabilities/structures.h>
```

$$< d >= \sum_{a,b} p_a p_b d(S_a, S_b)$$

this can be computed from the pair probs p_{ij} as

$$< d >= \sum_{ij} p_{ij}(1 - p_{ij})$$

SWIG Wrapper Notes:

This function is attached as method `mean_bp_distance()` to objects of type `fold_compound`. See, e.g. `RNA.fold_compound.mean_bp_distance()` in the *The RNA Python module*.

Parameters

- **fc** – The fold compound data structure

Returns
The mean pair distance of the structure ensemble

double **vrna_ensemble_defect_pt**(*vrna_fold_compound_t* *fc, const short *pt)

Compute the Ensemble Defect for a given target structure provided as a **vrna_ptable**.

```
#include <ViennaRNA/probabilities/structures.h>
```

Given a target structure s, compute the average dissimilarity of a randomly drawn structure from the ensemble, i.e.:

$$ED(s) = 1 - \frac{1}{n} \sum_{ij,(i,j)\in s} p_{ij} - \frac{1}{n} \sum_{i} (1 - s_i) q_i$$

with sequence length n, the probability p_{ij} of a base pair (i, j), the probability $q_i = 1 - \sum_j p_{ij}$ of nucleotide i being unpaired, and the indicator variable $s_i = 1$ if $\exists (i, j) \in s$, and $s_i = 0$ otherwise.

SWIG Wrapper Notes:

This function is attached as overloaded method `ensemble_defect()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.ensemble_defect()`* in the *The RNA Python module*.

See also

vrna_pf(), vrna_pairing_probs(), *vrna_ensemble_defect()*

Parameters

- **fc** – A fold_compound with pre-computed base pair probabilities
- **pt** – A pair table representing a target structure

Pre

The *vrna_fold_compound_t* input parameter `fc` must contain a valid base pair probability matrix. This means that partition function and base pair probabilities must have been computed using `fc` before execution of this function!

Returns

The ensemble defect with respect to the target structure, or -1. upon failure, e.g. pre-conditions are not met

double **vrna_ensemble_defect**(*vrna_fold_compound_t* *fc, const char *structure)

Compute the Ensemble Defect for a given target structure.

```
#include <ViennaRNA/probabilities/structures.h>
```

This is a wrapper around ***vrna_ensemble_defect_pt()***. Given a target structure s, compute the average dissimilarity of a randomly drawn structure from the ensemble, i.e.:

$$ED(s) = 1 - \frac{1}{n} \sum_{ij,(i,j)\in s} p_{ij} - \frac{1}{n} \sum_{i} (1 - s_i) q_i$$

with sequence length n, the probability p_{ij} of a base pair (i, j), the probability $q_i = 1 - \sum_j p_{ij}$ of nucleotide i being unpaired, and the indicator variable $s_i = 1$ if $\exists (i, j) \in s$, and $s_i = 0$ otherwise.

SWIG Wrapper Notes:

This function is attached as method `ensemble_defect()` to objects of type `fold_compound`. Note that the SWIG wrapper takes a structure in dot-bracket notation and converts it into a pair table using *vrna_ptable_from_string()*. The resulting pair table is then internally passed to *vrna_ensemble_defect_pt()*. To control which kind of matching brackets will be used during conversion, the optional argument `options` can be used. See also the description of *vrna_ptable_from_string()* for available options. (default: `VRNA_BRACKETS_RND`). See, e.g. *`RNA.fold_compound.ensemble_defect()`* in the *The RNA Python module*.

See also

vrna_pf(), vrna_pairing_probs(), *vrna_ensemble_defect_pt()*

Parameters

- **fc** – A fold_compound with pre-computed base pair probabilities
- **structure** – A target structure in dot-bracket notation

Pre
The *vrna_fold_compound_t* input parameter `fc` must contain a valid base pair probability matrix. This means that partition function and base pair probabilities must have been computed using `fc` before execution of this function!

Returns
The ensemble defect with respect to the target structure, or -1. upon failure, e.g. pre-conditions are not met

double ***vrna_positional_entropy**(*vrna_fold_compound_t* *fc)

Compute a vector of positional entropies.

```
#include <ViennaRNA/probabilities/structures.h>
```

This function computes the positional entropies from base pair probabilities as

$$S(i) = -\sum_j p_{ij}\log(p_{ij}) - q_i\log(q_i)$$

with unpaired probabilities $q_i = 1 - \sum_j p_{ij}$.

Low entropy regions have little structural flexibility and the reliability of the predicted structure is high. High entropy implies many structural alternatives. While these alternatives may be functionally important, they make structure prediction more difficult and thus less reliable.

SWIG Wrapper Notes:

This function is attached as method `positional_entropy()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.positional_entropy()* in the *The RNA Python module*.

Parameters
- **fc** – A fold_compound with pre-computed base pair probabilities

Pre
This function requires pre-computed base pair probabilities! Thus, *vrna_pf()* must be called beforehand.

Returns
A 1-based vector of positional entropies $S(i)$. (position 0 contains the sequence length)

double **vrna_pr_structure**(*vrna_fold_compound_t* *fc, const char *structure)

Compute the equilibrium probability of a particular secondary structure.

```
#include <ViennaRNA/probabilities/structures.h>
```

The probability $p(s)$ of a particular secondary structure s can be computed as

$$p(s) = \frac{exp(-\beta E(s)}{Z}$$

from the structures free energy $E(s)$ and the partition function

$$Z = \sum_s exp(-\beta E(s)), \quad \text{with} \quad \beta = \frac{1}{RT}$$

where R is the gas constant and T the thermodynamic temperature.

SWIG Wrapper Notes:

This function is attached as method `pr_structure()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.pr_structure()`* in the *The RNA Python module*.

Parameters

- **fc** – The fold compound data structure with precomputed partition function
- **structure** – The secondary structure to compute the probability for in dot-bracket notation

Pre

The fold compound `fc` must have went through a call to *vrna_pf()* to fill the dynamic programming matrices with the corresponding partition function.

Returns

The probability of the input structure (range $[0 : 1]$)

double **vrna_pr_energy**(*vrna_fold_compound_t* *fc, double e)

```
#include <ViennaRNA/probabilities/structures.h>
```

SWIG Wrapper Notes:

This function is attached as method `pr_energy()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.pr_energy()`* in the *The RNA Python module*.

Defines

VRNA_LOOP_TYPE_EXT

Exterior loop.

```
#include <ViennaRNA/probabilities/basic.h>
```

VRNA_LOOP_TYPE_HP

Hairpin loop.

```
#include <ViennaRNA/probabilities/basic.h>
```

VRNA_LOOP_TYPE_INT

Internal loop.

```
#include <ViennaRNA/probabilities/basic.h>
```

VRNA_LOOP_TYPE_MB

Multibranch loop.

```
#include <ViennaRNA/probabilities/basic.h>
```

VRNA_LOOP_TYPE_ANY

Any loop.

```
#include <ViennaRNA/probabilities/basic.h>
```

VRNA_EQM_PROBS_BP

Base pair probability flag.

```
#include <ViennaRNA/probabilities/basic.h>
```

Passing this flag to vrna_eqm_probabilities() or *vrna_probs_window()* activates callback execution for base pairing probabilities. In turn, the corresponding callback receives this flag through the `type` argument whenever base pairing probabilities are provided.

See also

vrna_eqm_probabilities(), *vrna_probs_window()*

VRNA_EQM_PROBS_UP

Unpaired probability (accessibility) flag.

```
#include <ViennaRNA/probabilities/basic.h>
```

Passing this flag to vrna_eqm_probabilities() or *vrna_probs_window()* activates callback execution for unpaired probabilities, a.k.a. accessibilities. In turn, the corresponding callback receives this flag through the `type` argument whenever unpaired probabilities are provided.

See also

vrna_eqm_probabilities(), *vrna_probs_window()*

VRNA_EQM_PROBS_HX

Helix probability flag.

```
#include <ViennaRNA/probabilities/basic.h>
```

Passing this flag to vrna_eqm_probabilities() or *vrna_probs_window()* activates callback execution for helix probabilities. In turn, the corresponding callback receives this flag through the `type` argument whenever stack probabilities are provided.

Bug:

Currently, this flag is a placeholder doing nothing as the corresponding implementation for helix/stack probability computation is missing.

See also

vrna_eqm_probabilities(), *vrna_probs_window()*

VRNA_EQM_PROBS_SPLIT

Split probabilities into loop types.

```
#include <ViennaRNA/probabilities/basic.h>
```

Passing this flag to vrna_eqm_probabilities() or *vrna_probs_window()* indicates that probabilities should be reported on a loop-type base. In turn, the corresponding callback receives the *VRNA_PROBS_WINDOW_UP* flag OR-ed together with the corresponding loop type, i.e.:

- *VRNA_EXT_LOOP* - Exterior loop.
- *VRNA_HP_LOOP* - Hairpin loop.
- *VRNA_INT_LOOP* - Internal loop.
- *VRNA_MB_LOOP* - Multibranch loop.
- *VRNA_ANY_LOOP* - Any loop.

See also

vrna_eqm_probabilities(), *vrna_probs_window()*, *VRNA_EQM_PROBS_UP*, *VRNA_EQM_PROBS_BP*

VRNA_EQM_PROBS_STORE_OUTSIDE

```
#include <ViennaRNA/probabilities/basic.h>
```

VRNA_EQM_PROBS_OUTSIDE_PF

```
#include <ViennaRNA/probabilities/basic.h>
```

VRNA_EQM_PROBS_MASK

```
#include <ViennaRNA/probabilities/basic.h>
```

Typedefs

typedef void (***vrna_heat_capacity_f**)(float temp, float heat_capacity, void *data)

The callback for heat capacity predictions.

```
#include <ViennaRNA/heat_capacity.h>
```

Notes on Callback Functions:

This function will be called for each evaluated temperature in the heat capacity prediction.

See also

vrna_heat_capacity_cb()

Param temp

The current temperature this results corresponds to in °C

Param heat_capacity
The heat capacity in Kcal/(Mol * K)

Param data
Some arbitrary data pointer passed through by the function executing the callback

typedef void **vrna_heat_capacity_callback**(float temp, float heat_capacity, void *data)

```
#include <ViennaRNA/heat_capacity.h>
```

typedef struct *vrna_heat_capacity_s* **vrna_heat_capacity_t**

A single result from heat capacity computations.

```
#include <ViennaRNA/heat_capacity.h>
```

This is a convenience typedef for *vrna_heat_capacity_s*, i.e. results as obtained from *vrna_heat_capacity()*

typedef void ***vrna_eqm_probs_data_t**

```
#include <ViennaRNA/probabilities/basic.h>
```

typedef void (***vrna_eqm_probs_f**)(*vrna_fold_compound_t* *fc, *FLT_OR_DBL* *pr, unsigned int j, unsigned int a, unsigned int b, unsigned int type, *vrna_eqm_probs_data_t* data)

```
#include <ViennaRNA/probabilities/basic.h>
```

typedef struct vrna_eqm_probs_component_s ***vrna_eqm_probs_component_t**

```
#include <ViennaRNA/probabilities/basic.h>
```

Functions

int **vrna_eqm_probabilities**(*vrna_fold_compound_t* *fc, *vrna_eqm_probs_f* cb, *vrna_eqm_probs_data_t* cb_data, *vrna_eqm_probs_opt_t* options)

```
#include <ViennaRNA/probabilities/basic.h>
```

struct **vrna_heat_capacity_s**

#include <ViennaRNA/heat_capacity.h> A single result from heat capacity computations.

See also

vrna_heat_capacity()

Public Members

float **temperature**

The temperature in °C.

```
#include <ViennaRNA/heat_capacity.h>
```

float **heat_capacity**

The specific heat at this temperature in Kcal/(Mol * K).

```
#include <ViennaRNA/heat_capacity.h>
```

struct **vrna_eqm_probs_opt_t**

Public Members

unsigned int **mode**

```
#include <ViennaRNA/probabilities/basic.h>
```

unsigned int **up_length**

```
#include <ViennaRNA/probabilities/basic.h>
```

unsigned int **hx_length**

```
#include <ViennaRNA/probabilities/basic.h>
```

Deprecated Partition Function API

Unnamed Group

float **alipf_fold_par**(const char **sequences, char *structure, *vrna_ep_t* **pl, *vrna_exp_param_t* *parameters, int calculate_bppm, int is_constrained, int is_circular)

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

Use *vrna_pf()* instead

Parameters

- **sequences**
- **structure**
- **pl**
- **parameters**
- **calculate_bppm**
- **is_constrained**
- **is_circular**

Returns

float **alipf_fold**(const char **sequences, char *structure, *vrna_ep_t* **pl)

The partition function version of *alifold()* works in analogy to *pf_fold()*. Pair probabilities and information about sequence covariations are returned via the 'pi' variable as a list of *vrna_pinfo_t* structs. The list is terminated by the first entry with pi.i = 0.

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

Use *vrna_pf()* instead

Parameters

- **sequences**
- **structure**
- **pl**

Returns

float **alipf_circ_fold**(const char **sequences, char *structure, *vrna_ep_t* **pl)

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

Use *vrna_pf()* instead

Parameters

- **sequences**
- **structure**
- **pl**

Returns

FLT_OR_DBL ***export_ali_bppm**(void)

Get a pointer to the base pair probability array.

```
#include <ViennaRNA/alifold.h>
```

Accessing the base pair probabilities for a pair (i,j) is achieved by

```
FLT_OR_DBL *pr = export_bppm(); pr_ij = pr[iindx[i]-j];
```

Deprecated:

Usage of this function is discouraged! The new *vrna_fold_compound_t* allows direct access to the folding matrices, including the pair probabilities! The pair probability array returned here reflects the one of the latest call to *vrna_pf()*, or any of the old API calls for consensus structure partition function folding.

See also

vrna_fold_compound_t, *vrna_fold_compound_comparative()*, and *vrna_pf()*

Returns
A pointer to the base pair probability array

void **free_alipf_arrays**(void)

Free the memory occupied by folding matrices allocated by alipf_fold, alipf_circ_fold, etc.

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

Usage of this function is discouraged! This function only free's memory allocated by old API function calls. Memory allocated by any of the new API calls (starting with vrna_) will be not affected!

See also

vrna_fold_compound_t, vrna_vrna_fold_compound_free()

char ***alipbacktrack**(double *prob)

Sample a consensus secondary structure from the Boltzmann ensemble according its probability.

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

Use *vrna_pbacktrack()* instead!

Parameters

- **prob** – to be described (berni)

Returns
A sampled consensus secondary structure in dot-bracket notation

int **get_alipf_arrays**(short ***S_p, short ***S5_p, short ***S3_p, unsigned short ***a2s_p, char ***Ss_p, *FLT_OR_DBL* **qb_p, *FLT_OR_DBL* **qm_p, *FLT_OR_DBL* **q1k_p, *FLT_OR_DBL* **qln_p, short **pscore)

Get pointers to (almost) all relavant arrays used in alifold's partition function computation.

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

It is discouraged to use this function! The new *vrna_fold_compound_t* allows direct access to all necessary consensus structure prediction related variables!

See also

vrna_fold_compound_t, *vrna_fold_compound_comparative()*, *vrna_pf()*, pf_alifold(), *alipf_circ_fold()*

Note

To obtain meaningful pointers, call alipf_fold first!

Parameters

- **S_p** – A pointer to the 'S' array (integer representation of nucleotides)
- **S5_p** – A pointer to the 'S5' array
- **S3_p** – A pointer to the 'S3' array
- **a2s_p** – A pointer to the alignment-column to sequence position mapping array
- **Ss_p** – A pointer to the 'Ss' array
- **qb_p** – A pointer to the Q^B matrix
- **qm_p** – A pointer to the Q^M matrix
- **q1k_p** – A pointer to the 5' slice of the Q matrix ($q1k(k) = Q(1,k)$)
- **qln_p** – A pointer to the 3' slice of the Q matrix ($qln(l) = Q(l,n)$)
- **pscore** – A pointer to the start of a pscore list

Returns

Non Zero if everything went fine, 0 otherwise

Functions

vrna_dimer_pf_t **co_pf_fold**(char *sequence, char *structure)

Calculate partition function and base pair probabilities.

```
#include <ViennaRNA/part_func_co.h>
```

This is the cofold partition function folding. The second molecule starts at the cut_point nucleotide.

Deprecated:

Note

OpenMP: Since this function relies on the global parameters *do_backtrack*, *dangles*, *temperature* and *pf_scale* it is not threadsafe according to concurrent changes in these variables! Use *co_pf_fold_par()* instead to circumvent this issue.

Parameters

- **sequence** – Concatenated RNA sequences
- **structure** – Will hold the structure or constraints

Returns

vrna_dimer_pf_t structure containing a set of energies needed for concentration computations.

vrna_dimer_pf_t **co_pf_fold_par**(char *sequence, char *structure, *vrna_exp_param_t* *parameters, int calculate_bppm, int is_constrained)

Calculate partition function and base pair probabilities.

```
#include <ViennaRNA/part_func_co.h>
```

This is the cofold partition function folding. The second molecule starts at the cut_point nucleotide.

Deprecated:

Use *vrna_pf_dimer()* instead!

See also

get_boltzmann_factors(), *co_pf_fold()*

Parameters

- **sequence** – Concatenated RNA sequences
- **structure** – Pointer to the structure constraint
- **parameters** – Data structure containing the precalculated Boltzmann factors
- **calculate_bppm** – Switch to turn Base pair probability calculations on/off (0==off)
- **is_constrained** – Switch to indicate that a structure contraint is passed via the structure argument (0==off)

Returns

vrna_dimer_pf_t structure containing a set of energies needed for concentration computations.

void **compute_probabilities**(double FAB, double FEA, double FEB, *vrna_ep_t* *prAB, *vrna_ep_t* *prA, *vrna_ep_t* *prB, int Alength)

Compute Boltzmann probabilities of dimerization without homodimers.

```
#include <ViennaRNA/part_func_co.h>
```

Given the pair probabilities and free energies (in the null model) for a dimer AB and the two constituent monomers A and B, compute the conditional pair probabilities given that a dimer AB actually forms. Null model pair probabilities are given as a list as produced by *assign_plist_from_pr()*, the dimer probabilities 'prAB' are modified in place.

Deprecated:

Parameters

- **FAB** – free energy of dimer AB
- **FEA** – free energy of monomer A
- **FEB** – free energy of monomer B
- **prAB** – pair probabilities for dimer
- **prA** – pair probabilities monomer
- **prB** – pair probabilities monomer

- **Alength** – Length of molecule A

void **init_co_pf_fold**(int length)

```
#include <ViennaRNA/part_func_co.h>
```

DO NOT USE THIS FUNCTION ANYMORE

Deprecated:

FLT_OR_DBL ***export_co_bppm**(void)

Get a pointer to the base pair probability array.

```
#include <ViennaRNA/part_func_co.h>
```

Accessing the base pair probabilities for a pair (i,j) is achieved by

```
FLT_OR_DBL *pr = export_bppm(); pr_ij = pr[iindx[i]-j];
```

Deprecated:

This function is deprecated and will be removed soon! The base pair probability array is available through the *vrna_fold_compound_t* data structure, and its associated *vrna_mx_pf_t* member.

See also

vrna_idx_row_wise()

Returns

A pointer to the base pair probability array

void **free_co_pf_arrays**(void)

Free the memory occupied by *co_pf_fold()*.

```
#include <ViennaRNA/part_func_co.h>
```

Deprecated:

This function will be removed for the new API soon! See *vrna_pf_dimer()*, *vrna_fold_compound()*, and *vrna_fold_compound_free()* for an alternative

void **update_co_pf_params**(int length)

Recalculate energy parameters.

```
#include <ViennaRNA/part_func_co.h>
```

This function recalculates all energy parameters given the current model settings.

Deprecated:

Use *vrna_exp_params_subst()* instead!

Parameters

- **length** – Length of the current RNA sequence

void **update_co_pf_params_par**(int length, *vrna_exp_param_t* *parameters)

Recalculate energy parameters.

```
#include <ViennaRNA/part_func_co.h>
```

This function recalculates all energy parameters given the current model settings. It's second argument can either be NULL or a data structure containing the precomputed Boltzmann factors. In the first scenario, the necessary data structure will be created automatically according to the current global model settings, i.e. this mode might not be threadsafe. However, if the provided data structure is not NULL, threadsafety for the model parameters *dangles*, *pf_scale* and *temperature* is regained, since their values are taken from this data structure during subsequent calculations.

Deprecated:

Use *vrna_exp_params_subst()* instead!

Parameters

- **length** – Length of the current RNA sequence
- **parameters** – data structure containing the precomputed Boltzmann factors

float **pf_fold_par**(const char *sequence, char *structure, *vrna_exp_param_t* *parameters, int calculate_bppm, int is_constrained, int is_circular)

Compute the partition function Q for a given RNA sequence.

```
#include <ViennaRNA/partfunc/global.h>
```

If *structure* is not a NULL pointer on input, it contains on return a string consisting of the letters " . , | { } ( ) " denoting bases that are essentially unpaired, weakly paired, strongly paired without preference, weakly upstream (downstream) paired, or strongly up- (down-)stream paired bases, respectively. If fold_constrained is not 0, the *structure* string is interpreted on input as a list of constraints for the folding. The character "x" marks bases that must be unpaired, matching brackets " ( ) " denote base pairs, all other characters are ignored. Any pairs conflicting with the constraint will be forbidden. This is usually sufficient to ensure the constraints are honored. If the parameter calculate_bppm is set to 0 base pairing probabilities will not be computed (saving CPU time), otherwise after calculations took place pr will contain the probability that bases i and j pair.

Deprecated:

Use *vrna_pf()* instead

See also

vrna_pf(), bppm_to_structure(), *export_bppm()*, *vrna_exp_params()*, *free_pf_arrays()*

Note

The global array pr is deprecated and the user who wants the calculated base pair probabilities for further computations is advised to use the function *export_bppm()*

Parameters

- **sequence** – **[in]** The RNA sequence input
- **structure** – **[inout]** A pointer to a char array where a base pair probability information can be stored in a pseudo-dot-bracket notation (may be NULL, too)
- **parameters** – **[in]** Data structure containing the precalculated Boltzmann factors
- **calculate_bppm** – **[in]** Switch to Base pair probability calculations on/off (0==off)
- **is_constrained** – **[in]** Switch to indicate that a structure contraint is passed via the structure argument (0==off)
- **is_circular** – **[in]** Switch to (de-)activate postprocessing steps in case RNA sequence is circular (0==off)

Post

After successful run the hidden folding matrices are filled with the appropriate Boltzmann factors. Depending on whether the global variable *do_backtrack* was set the base pair probabilities are already computed and may be accessed for further usage via the *export_bppm()* function. A call of *free_pf_arrays()* will free all memory allocated by this function. Successive calls will first free previously allocated memory before starting the computation.

Returns

The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

float **pf_fold**(const char *sequence, char *structure)

Compute the partition function Q of an RNA sequence.

```
#include <ViennaRNA/partfunc/global.h>
```

If *structure* is not a NULL pointer on input, it contains on return a string consisting of the letters " . , | { } ( ) " denoting bases that are essentially unpaired, weakly paired, strongly paired without preference, weakly upstream (downstream) paired, or strongly up- (down-)stream paired bases, respectively. If fold_constrained is not 0, the *structure* string is interpreted on input as a list of constraints for the folding. The character "x" marks bases that must be unpaired, matching brackets " ( ) " denote base pairs, all other characters are ignored. Any pairs conflicting with the constraint will be forbidden. This is usually sufficient to ensure the constraints are honored. If *do_backtrack* has been set to 0 base pairing probabilities will not be computed (saving CPU time), otherwise pr will contain the probability that bases *i* and *j* pair.

See also

pf_fold_par(), *pf_circ_fold()*, bppm_to_structure(), *export_bppm()*

Note

The global array pr is deprecated and the user who wants the calculated base pair probabilities for further computations is advised to use the function *export_bppm()*.

OpenMP: This function is not entirely threadsafe. While the recursions are working on their own copies of data the model details for the recursions are determined from the global settings just before entering the recursions. Consider using *pf_fold_par()* for a really threadsafe implementation.

Parameters

- **sequence** – The RNA sequence input

- **structure** – A pointer to a char array where a base pair probability information can be stored in a pseudo-dot-bracket notation (may be NULL, too)

Pre
This function takes its model details from the global variables provided in *RNAlib*

Post
After successful run the hidden folding matrices are filled with the appropriate Boltzmann factors. Depending on whether the global variable *do_backtrack* was set the base pair probabilities are already computed and may be accessed for further usage via the *export_bppm()* function. A call of *free_pf_arrays()* will free all memory allocated by this function. Successive calls will first free previously allocated memory before starting the computation.

Returns
The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

float **pf_circ_fold**(const char *sequence, char *structure)

Compute the partition function of a circular RNA sequence.

```
#include <ViennaRNA/partfunc/global.h>
```

Deprecated:
Use *vrna_pf()* instead!

See also
vrna_pf()

Note
The global array pr is deprecated and the user who wants the calculated base pair probabilities for further computations is advised to use the function *export_bppm()*.

OpenMP: This function is not entirely threadsafe. While the recursions are working on their own copies of data the model details for the recursions are determined from the global settings just before entering the recursions. Consider using *pf_fold_par()* for a really threadsafe implementation.

Pre
This function takes its model details from the global variables provided in *RNAlib*

Post
After successful run the hidden folding matrices are filled with the appropriate Boltzmann factors. Depending on whether the global variable *do_backtrack* was set the base pair probabilities are already computed and may be accessed for further usage via the *export_bppm()* function. A call of *free_pf_arrays()* will free all memory allocated by this function. Successive calls will first free previously allocated memory before starting the computation.

Parameters

- **sequence** – **[in]** The RNA sequence input
- **structure** – **[inout]** A pointer to a char array where a base pair probability information can be stored in a pseudo-dot-bracket notation (may be NULL, too)

Returns
The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

void **free_pf_arrays**(void)

Free arrays for the partition function recursions.

```
#include <ViennaRNA/partfunc/global.h>
```

Call this function if you want to free all allocated memory associated with the partition function forward recursion.

Deprecated:

See *vrna_fold_compound_t* and its related functions for how to free memory occupied by the dynamic programming matrices

See also

pf_fold_par(), *pf_fold()*, *pf_circ_fold()*

Note

Successive calls of *pf_fold()*, *pf_circ_fold()* already check if they should free any memory from a previous run.

OpenMP notice:

This function should be called before leaving a thread in order to avoid leaking memory

Post

All memory allocated by *pf_fold_par()*, *pf_fold()* or *pf_circ_fold()* will be free'd

void **update_pf_params**(int length)

Recalculate energy parameters.

```
#include <ViennaRNA/partfunc/global.h>
```

Call this function to recalculate the pair matrix and energy parameters after a change in folding parameters like *temperature*

Deprecated:

Use *vrna_exp_params_subst()* instead

void **update_pf_params_par**(int length, *vrna_exp_param_t* *parameters)

Recalculate energy parameters.

```
#include <ViennaRNA/partfunc/global.h>
```

Deprecated:

Use *vrna_exp_params_subst()* instead

FLT_OR_DBL ***export_bppm**(void)

Get a pointer to the base pair probability array.

```
#include <ViennaRNA/partfunc/global.h>
```

Accessing the base pair probabilities for a pair (i,j) is achieved by

```
FLT_OR_DBL *pr  = export_bppm();
pr_ij           = pr[iindx[i]-j];
```

See also

pf_fold(), *pf_circ_fold()*, *vrna_idx_row_wise()*

Pre
Call *pf_fold_par()*, *pf_fold()* or *pf_circ_fold()* first to fill the base pair probability array

Returns
A pointer to the base pair probability array

int **get_pf_arrays**(short **S_p, short **S1_p, char **ptype_p, *FLT_OR_DBL* **qb_p, *FLT_OR_DBL* **qm_p, *FLT_OR_DBL* **q1k_p, *FLT_OR_DBL* **qln_p)

Get the pointers to (almost) all relavant computation arrays used in partition function computation.

```
#include <ViennaRNA/partfunc/global.h>
```

See also

pf_fold_par(), *pf_fold()*, *pf_circ_fold()*

Parameters

- **S_p** – **[out]** A pointer to the 'S' array (integer representation of nucleotides)
- **S1_p** – **[out]** A pointer to the 'S1' array (2nd integer representation of nucleotides)
- **ptype_p** – **[out]** A pointer to the pair type matrix
- **qb_p** – **[out]** A pointer to the Q^B matrix
- **qm_p** – **[out]** A pointer to the Q^M matrix
- **q1k_p** – **[out]** A pointer to the 5' slice of the Q matrix ($q1k(k) = Q(1, k)$)
- **qln_p** – **[out]** A pointer to the 3' slice of the Q matrix ($qln(l) = Q(l, n)$)

Pre
In order to assign meaningful pointers, you have to call *pf_fold_par()* or *pf_fold()* first!

Returns
Non Zero if everything went fine, 0 otherwise

double **get_subseq_F**(int i, int j)

Get the free energy of a subsequence from the q[] array.

```
#include <ViennaRNA/partfunc/global.h>
```

double **mean_bp_distance**(int length)

Get the mean base pair distance of the last partition function computation.

```
#include <ViennaRNA/partfunc/global.h>
```

Deprecated:

Use *vrna_mean_bp_distance()* or *vrna_mean_bp_distance_pr()* instead!

See also

vrna_mean_bp_distance(), *vrna_mean_bp_distance_pr()*

Parameters

- **length**

Returns

mean base pair distance in thermodynamic ensemble

double **mean_bp_distance_pr**(int length, *FLT_OR_DBL* *pr)

Get the mean base pair distance in the thermodynamic ensemble.

```
#include <ViennaRNA/partfunc/global.h>
```

This is a threadsafe implementation of mean_bp_dist() !

$< d >= \sum_{a,b} p_a p_b d(S_a, S_b)$

this can be computed from the pair probs $p_i j$ as

$< d >= \sum_{ij} p_{ij}(1 - p_{ij})$

Deprecated:

Use *vrna_mean_bp_distance()* or *vrna_mean_bp_distance_pr()* instead!

Parameters

- **length** – The length of the sequence
- **pr** – The matrix containing the base pair probabilities

Returns

The mean pair distance of the structure ensemble

vrna_ep_t ***stackProb**(double cutoff)

Get the probability of stacks.

```
#include <ViennaRNA/partfunc/global.h>
```

Deprecated:

Use *vrna_stack_prob()* instead!

void **init_pf_fold**(int length)

Allocate space for *pf_fold()*.

```
#include <ViennaRNA/partfunc/global.h>
```

Deprecated:

This function is obsolete and will be removed soon!

void **assign_plist_from_db**(*vrna_ep_t* **pl, const char *struc, float pr)

Create a *vrna_ep_t* from a dot-bracket string.

```
#include <ViennaRNA/structures/problist.h>
```

The dot-bracket string is parsed and for each base pair an entry in the plist is created. The probability of each pair in the list is set by a function parameter.

The end of the plist is marked by sequence positions i as well as j equal to 0. This condition should be used to stop looping over its entries

Deprecated:

Use *vrna_plist()* instead

Parameters

- **pl** – A pointer to the *vrna_ep_t* that is to be created
- **struc** – The secondary structure in dot-bracket notation
- **pr** – The probability for each base pair

void **assign_plist_from_pr**(*vrna_ep_t* **pl, *FLT_OR_DBL* *probs, int length, double cutoff)

Create a *vrna_ep_t* from a probability matrix.

```
#include <ViennaRNA/structures/problist.h>
```

The probability matrix given is parsed and all pair probabilities above the given threshold are used to create an entry in the plist

The end of the plist is marked by sequence positions i as well as j equal to 0. This condition should be used to stop looping over its entries

Deprecated:

Use *vrna_plist_from_probs()* instead!

Note

This function is threadsafe

Parameters

- **pl** – **[out]** A pointer to the *vrna_ep_t* that is to be created
- **probs** – **[in]** The probability matrix used for creating the plist
- **length** – **[in]** The length of the RNA sequence
- **cutoff** – **[in]** The cutoff value

Functions

void **update_pf_paramsLP**(int length)

```
#include <ViennaRNA/LPfold.h>
```

Parameters

- **length**

void **update_pf_paramsLP_par**(int length, *vrna_exp_param_t* *parameters)

```
#include <ViennaRNA/LPfold.h>
```

vrna_ep_t ***pfl_fold**(char *sequence, int winSize, int pairSize, float cutoffb, double **pU, *vrna_ep_t* **dpp2, FILE *pUfp, FILE *spup)

Compute partition functions for locally stable secondary structures.

```
#include <ViennaRNA/LPfold.h>
```

pfl_fold computes partition functions for every window of size 'winSize' possible in a RNA molecule, allowing only pairs with a span smaller than 'pairSize'. It returns the mean pair probabilities averaged over all windows containing the pair in 'pl'. 'winSize' should always be >= 'pairSize'. Note that in contrast to *Lfold()*, bases outside of the window do not influence the structure at all. Only probabilities higher than 'cutoffb' are kept.

If 'pU' is supplied (i.e is not the NULL pointer), *pfl_fold()* will also compute the mean probability that regions of length 'u' and smaller are unpaired. The parameter 'u' is supplied in 'pup[0][0]'. On return the 'pup' array will contain these probabilities, with the entry on 'pup[x][y]' containing the mean probability that x and the y-1 preceding bases are unpaired. The 'pU' array needs to be large enough to hold n+1 float* entries, where n is the sequence length.

If an array dpp2 is supplied, the probability of base pair (i,j) given that there already exists a base pair (i+1,j-1) is also computed and saved in this array. If pUfp is given (i.e. not NULL), pU is not saved but put out imediately. If spup is given (i.e. is not NULL), the pair probabilities in pl are not saved but put out imediately.

Parameters

- **sequence** – RNA sequence
- **winSize** – size of the window
- **pairSize** – maximum size of base pair
- **cutoffb** – cutoffb for base pairs
- **pU** – array holding all unpaired probabilities
- **dpp2** – array of dependent pair probabilities
- **pUfp** – file pointer for pU
- **spup** – file pointer for pair probabilities

Returns

list of pair probabilities

vrna_ep_t ***pfl_fold_par**(char *sequence, int winSize, int pairSize, float cutoffb, double **pU, *vrna_ep_t* **dpp2, FILE *pUfp, FILE *spup, *vrna_exp_param_t* *parameters)

Compute partition functions for locally stable secondary structures.

```
#include <ViennaRNA/LPfold.h>
```

void **putoutpU_prob**(double **pU, int length, int ulength, FILE *fp, int energies)

Writes the unpaired probabilities (pU) or opening energies into a file.

```
#include <ViennaRNA/LPfold.h>
```

Can write either the unpaired probabilities (accessibilities) pU or the opening energies -log(pU)kT into a file

Parameters

- **pU** – pair probabilities
- **length** – length of RNA sequence
- **ulength** – maximum length of unpaired stretch
- **fp** – file pointer of destination file
- **energies** – switch to put out as opening energies

void **putoutpU_prob_bin**(double **pU, int length, int ulength, FILE *fp, int energies)

Writes the unpaired probabilities (pU) or opening energies into a binary file.

```
#include <ViennaRNA/LPfold.h>
```

Can write either the unpaired probabilities (accessibilities) pU or the opening energies -log(pU)kT into a file

Parameters

- **pU** – pair probabilities
- **length** – length of RNA sequence
- **ulength** – maximum length of unpaired stretch
- **fp** – file pointer of destination file
- **energies** – switch to put out as opening energies

4.1.3 Suboptimals and Representative Structures

Sample and enumerate suboptimal secondary structures from RNA sequence data.

Suboptimal Structures sensu Zuker

Functions

SOLUTION ***zukersubopt**(const char *string)

Compute Zuker type suboptimal structures.

```
#include <ViennaRNA/subopt/wuchty.h>
```

Compute Suboptimal structures according to M. Zuker, i.e. for every possible base pair the minimum energy structure containing the resp. base pair. Returns a list of these structures and their energies.

Deprecated:

use vrna_zukersubopt() instead

Parameters

- **string** – RNA sequence

Returns

List of zuker suboptimal structures

SOLUTION ***zukersubopt_par**(const char *string, *vrna_param_t* *parameters)

Compute Zuker type suboptimal structures.

```
#include <ViennaRNA/subopt/wuchty.h>
```

Deprecated:

use vrna_zukersubopt() instead

vrna_subopt_solution_t ***vrna_subopt_zuker**(*vrna_fold_compound_t* *fc)

Compute Zuker type suboptimal structures.

```
#include <ViennaRNA/subopt/zuker.h>
```

Compute Suboptimal structures according to Zuker [1989] , i.e. for every possible base pair the minimum energy structure containing the resp. base pair. Returns a list of these structures and their energies.

SWIG Wrapper Notes:

This function is attached as method **subopt_zuker()** to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.subopt_zuker()`* in the *The RNA Python module*.

See also

vrna_subopt(), *zukersubopt()*, *zukersubopt_par()*

Parameters

- **fc** – fold compound

Returns

List of zuker suboptimal structures

Suboptimal Structures within an Energy Band around the MFE

Typedefs

typedef void (***vrna_subopt_result_f**)(const char *structure, float energy, void *data)

Callback for *vrna_subopt_cb()*.

```
#include <ViennaRNA/subopt/basic.h>
```

Notes on Callback Functions:

This function will be called for each suboptimal secondary structure that is successfully backtraced.

See also

vrna_subopt_cb()

Param structure
The suboptimal secondary structure in dot-bracket notation

Param energy
The free energy of the secondary structure in kcal/mol

Param data
Some arbitrary, auxiliary data address as passed to *vrna_subopt_cb()*

Functions

vrna_subopt_solution_t ***vrna_subopt**(*vrna_fold_compound_t* *fc, int delta, int sorted, FILE *fp)

Returns list of subopt structures or writes to fp.

```
#include <ViennaRNA/subopt/wuchty.h>
```

This function produces **all** suboptimal secondary structures within 'delta' * 0.01 kcal/mol of the optimum, see Wuchty *et al.* [1999] . The results are either directly written to a 'fp' (if 'fp' is not NULL), or (fp==NULL) returned in a vrna_subopt_solution_t * list terminated by an entry were the 'structure' member is NULL.

SWIG Wrapper Notes:

This function is attached as method ***subopt()*** to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.subopt()`* in the *The RNA Python module*.

See also

vrna_subopt_cb(), *vrna_subopt_zuker()*

Note

This function requires all multibranch loop DP matrices for unique multibranch loop backtracing. Therefore, the supplied *vrna_fold_compound_t* `fc` (argument 1) must be initialized with *vrna_md_t.uniq_ML* = 1, for instance like this:

```
vrna_md_t md;
vrna_md_set_default(&md);
md.uniq_ML = 1;

vrna_fold_compound_t *fc=vrna_fold_compound("GGGGGGAAAAAACCCCCC", &md,␣
↪VRNA_OPTION_DEFAULT);
```

Parameters

- **fc**
- **delta**
- **sorted** – Sort results by energy in ascending order
- **fp**

Returns

void **vrna_subopt_cb**(*vrna_fold_compound_t* *fc, int delta, *vrna_subopt_result_f* cb, void *data)

Generate suboptimal structures within an energy band arround the MFE.

```
#include <ViennaRNA/subopt/wuchty.h>
```

This is the most generic implementation of the suboptimal structure generator according to Wuchty *et al.* [1999] . Identical to *vrna_subopt()*, it computes all secondary structures within an energy band `delta` arround the MFE. However, this function does not print the resulting structures and their corresponding free energies to a file pointer, or returns them as a list. Instead, it calls a user-provided callback function which it passes the structure in dot-bracket format, the corresponding free energy in kcal/mol, and a user-provided data structure each time a structure was backtracked successfully. This function indicates the final output, i.e. the end of the backtracking procedure by passing NULL instead of an actual dot-bracket string to the callback.

SWIG Wrapper Notes:

This function is attached as method **subopt_cb()** to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.subopt_cb()* in the *The RNA Python module*.

See also

vrna_subopt_result_f, *vrna_subopt()*, *vrna_subopt_zuker()*

Note

This function requires all multibranch loop DP matrices for unique multibranch loop backtracing. Therefore, the supplied *vrna_fold_compound_t* `fc` (argument 1) must be initialized with *vrna_md_t.uniq_ML* = 1, for instance like this:

```
vrna_md_t md;
vrna_md_set_default(&md);
md.uniq_ML = 1;

vrna_fold_compound_t *fc=vrna_fold_compound("GGGGGGAAAAAACCCCCC", &md,␣
↪VRNA_OPTION_DEFAULT);
```

Parameters

- **fc** – fold compount with the sequence data
- **delta** – Energy band arround the MFE in 10cal/mol, i.e. deka-calories
- **cb** – Pointer to a callback function that handles the backtracked structure and its free energy in kcal/mol
- **data** – Pointer to some data structure that is passed along to the callback

SOLUTION ***subopt**(char *seq, char *structure, int delta, FILE *fp)

Returns list of subopt structures or writes to fp.

```
#include <ViennaRNA/subopt/wuchty.h>
```

This function produces **all** suboptimal secondary structures within 'delta' * 0.01 kcal/mol of the optimum. The results are either directly written to a 'fp' (if 'fp' is not NULL), or (fp==NULL) returned in a SOLUTION * list terminated by an entry were the 'structure' pointer is NULL.

Parameters

- **seq**
- **structure**
- **delta**
- **fp**

Returns

SOLUTION ***subopt_par**(char *seq, char *structure, *vrna_param_t* *parameters, int delta, int is_constrained, int is_circular, FILE *fp)

Returns list of subopt structures or writes to fp.

```
#include <ViennaRNA/subopt/wuchty.h>
```

SOLUTION ***subopt_circ**(char *seq, char *sequence, int delta, FILE *fp)

Returns list of circular subopt structures or writes to fp.

```
#include <ViennaRNA/subopt/wuchty.h>
```

This function is similar to *subopt()* but calculates secondary structures assuming the RNA sequence to be circular instead of linear

Parameters

- **seq**
- **sequence**
- **delta**
- **fp**

Returns

Variables

double **print_energy**

printing threshold for use with logML

```
#include <ViennaRNA/subopt/wuchty.h>
```

int **subopt_sorted**

Sort output by energy.

```
#include <ViennaRNA/subopt/wuchty.h>
```

Boltzmann Sampling

Draw random structure samples from the ensemble

Functions to draw random structure samples from the ensemble according to their equilibrium probability.

Defines

VRNA_PBACKTRACK_DEFAULT

Boltzmann sampling flag indicating default backtracing mode.

```
#include <ViennaRNA/sampling/basic.h>
```

See also

vrna_pbacktrack5_num(), *vrna_pbacktrack5_cb()*, *vrna_pbacktrack5_resume()*, *vrna_pbacktrack5_resume_cb()*, *vrna_pbacktrack_num()*, *vrna_pbacktrack_cb()*, *vrna_pbacktrack_resume()*, *vrna_pbacktrack_resume_cb()*

VRNA_PBACKTRACK_NON_REDUNDANT

Boltzmann sampling flag indicating non-redundant backtracing mode.

```
#include <ViennaRNA/sampling/basic.h>
```

This flag will turn the Boltzmann sampling into non-redundant backtracing mode along the lines of Michálik *et al.* [2017]

See also

vrna_pbacktrack5_num(), *vrna_pbacktrack5_cb()*, *vrna_pbacktrack5_resume()*, *vrna_pbacktrack5_resume_cb()*, *vrna_pbacktrack_num()*, *vrna_pbacktrack_cb()*, *vrna_pbacktrack_resume()*, *vrna_pbacktrack_resume_cb()*

Typedefs

typedef void (***vrna_bs_result_f**)(const char *structure, void *data)

Callback for Boltzmann sampling.

```
#include <ViennaRNA/sampling/basic.h>
```

Notes on Callback Functions:

This function will be called for each secondary structure that has been successfully backtraced from the partition function DP matrices.

See also

vrna_pbacktrack5_cb(), *vrna_pbacktrack_cb()*, *vrna_pbacktrack5_resume_cb()*, *vrna_pbacktrack_resume_cb()*

Param structure
The secondary structure in dot-bracket notation

Param data
Some arbitrary, auxiliary data address as provided to the calling function

typedef void **vrna_boltzmann_sampling_callback**(const char *structure, void *data)

```
#include <ViennaRNA/sampling/basic.h>
```

typedef struct vrna_pbacktrack_memory_s ***vrna_pbacktrack_mem_t**

Boltzmann sampling memory data structure.

```
#include <ViennaRNA/sampling/basic.h>
```

This structure is required for properly resuming a previous sampling round in specialized Boltzmann sampling, such as non-redundant backtracking.

Initialize with `NULL` and pass its address to the corresponding functions *vrna_pbacktrack5_resume()*, etc.

See also

vrna_pbacktrack5_resume(), *vrna_pbacktrack_resume()*, *vrna_pbacktrack5_resume_cb()*, *vrna_pbacktrack_resume_cb()*, *vrna_pbacktrack_mem_free()*

Note

Do not forget to release memory occupied by this data structure before losing its context! Use *vrna_pbacktrack_mem_free()*.

Functions

char ***vrna_pbacktrack5**(*vrna_fold_compound_t* *fc, unsigned int length)

Sample a secondary structure of a subsequence from the Boltzmann ensemble according its probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a secondary structure. The parameter `length` specifies the length of the substructure starting from the 5' end.

The structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

SWIG Wrapper Notes:

This function is attached as overloaded method `pbacktrack5()` to objects of type `fold_compound`. See, e.g. `RNA.fold_compound.pbacktrack5()` in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5_num(), *vrna_pbacktrack5_cb()*, *vrna_pbacktrack()*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Parameters

- **fc** – The fold compound data structure
- **length** – The length of the subsequence to consider (starting with 5' end)

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A sampled secondary structure in dot-bracket notation (or NULL on error)

char **vrna_pbacktrack5_num(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int length, unsigned int options)

Obtain a set of secondary structure samples for a subsequence from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures. The parameter `length` specifies the length of the substructure starting from the 5' end.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5(), *vrna_pbacktrack5_cb()*, *vrna_pbacktrack_num()*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **length** – The length of the subsequence to consider (starting with 5' end)
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A set of secondary structure samples in dot-bracket notation terminated by NULL (or NULL on error)

unsigned int **vrna_pbacktrack5_cb**(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int length, *vrna_bs_result_f* cb, void *data, unsigned int options)

Obtain a set of secondary structure samples for a subsequence from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures. The parameter `length` specifies the length of the substructure starting from the 5' end.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack5()* and *vrna_pbacktrack5_num()* this function yields the structure samples through a callback mechanism.

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5(), *vrna_pbacktrack5_num()*, *vrna_pbacktrack_cb()*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **length** – The length of the subsequence to consider (starting with 5' end)
- **cb** – The callback that receives the sampled structure
- **data** – A data structure passed through to the callback `cb`
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

The number of structures actually backtraced

char **vrna_pbacktrack5_resume(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int length, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

Obtain a set of secondary structure samples for a subsequence from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures. The parameter `length` specifies the length of the substructure starting from the 5' end.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack5_cb()* this function allows for resuming a previous sampling round in specialized Boltzmann sampling, such as non-redundant backtracking. For that purpose, the user passes the address of a Boltzmann sampling data structure (*vrna_pbacktrack_mem_t*) which will be re-used in each round of sampling, i.e. each successive call to *vrna_pbacktrack5_resume_cb()* or *vrna_pbacktrack5_resume()*.

A successive sample call to this function may look like:

```
vrna_pbacktrack_mem_t nonredundant_memory = NULL;

// sample the first 100 structures
vrna_pbacktrack5_resume(fc,
                        100,
                        fc->length,
                        &nonredundant_memory,
                        options);

// sample another 500 structures
vrna_pbacktrack5_resume(fc,
                        500,
                        fc->length,
                        &nonredundant_memory,
                        options);

// release memory occupied by the non-redundant memory data structure
vrna_pbacktrack_mem_free(nonredundant_memory);
```

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the list of structures, this function also returns the `nr_mem` data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5_resume_cb(), *vrna_pbacktrack5_cb()*, *vrna_pbacktrack_resume()*, *vrna_pbacktrack_mem_t*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*, *vrna_pbacktrack_mem_free*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **length** – The length of the subsequence to consider (starting with 5' end)
- **nr_mem** – The address of the Boltzmann sampling memory data structure
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A set of secondary structure samples in dot-bracket notation terminated by NULL (or NULL on error)

unsigned int **vrna_pbacktrack5_resume_cb**(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int length, *vrna_bs_result_f* cb, void *data, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

Obtain a set of secondary structure samples for a subsequence from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures. The parameter `length` specifies the length of the substructure starting from the 5' end.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack5_resume()* this function yields the structure samples through a callback mechanism.

A successive sample call to this function may look like:

```
vrna_pbacktrack_mem_t nonredundant_memory = NULL;

// sample the first 100 structures
vrna_pbacktrack5_resume_cb(fc,
                           100,
                           fc->length,
                           &callback_function,
                           (void *)&callback_data,
                           &nonredundant_memory,
                           options);

// sample another 500 structures
vrna_pbacktrack5_resume_cb(fc,
                           500,
                           fc->length,
                           &callback_function,
                           (void *)&callback_data,
                           &nonredundant_memory,
                           options);

// release memory occupied by the non-redundant memory data structure
vrna_pbacktrack_mem_free(nonredundant_memory);
```

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the number of structures backtraced, this function also returns the `nr_mem` data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5_resume(), *vrna_pbacktrack5_cb()*, *vrna_pbacktrack_resume_cb()*, *vrna_pbacktrack_mem_t*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*, *vrna_pbacktrack_mem_free*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **length** – The length of the subsequence to consider (starting with 5' end)
- **cb** – The callback that receives the sampled structure
- **data** – A data structure passed through to the callback cb
- **nr_mem** – The address of the Boltzmann sampling memory data structure
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of fc with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

The number of structures actually backtraced

char ***vrna_pbacktrack**(*vrna_fold_compound_t* *fc)

Sample a secondary structure from the Boltzmann ensemble according its probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a secondary structure.

The structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack()* to objects of type fold_compound. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5(), *vrna_pbacktrack_num*, *vrna_pbacktrack_cb()*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Parameters

- **fc** – The fold compound data structure

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A sampled secondary structure in dot-bracket notation (or NULL on error)

char ***vrna_pbacktrack_num**(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int options)

Obtain a set of secondary structure samples from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack(), *vrna_pbacktrack_cb()*, *vrna_pbacktrack5_num()*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested

structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A set of secondary structure samples in dot-bracket notation terminated by NULL (or NULL on error)

unsigned int **vrna_pbacktrack_cb**(*vrna_fold_compound_t* *fc, unsigned int num_samples, *vrna_bs_result_f* cb, void *data, unsigned int options)

Obtain a set of secondary structure samples from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack()* and *vrna_pbacktrack_num()* this function yields the structure samples through a callback mechanism.

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack(), *vrna_pbacktrack_num()*, *vrna_pbacktrack5_cb()*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **cb** – The callback that receives the sampled structure
- **data** – A data structure passed through to the callback cb
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of fc with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

The number of structures actually backtraced

char **vrna_pbacktrack_resume(*vrna_fold_compound_t* *fc, unsigned int num_samples, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

Obtain a set of secondary structure samples from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of num_samples secondary structures.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the options flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack_cb()* this function allows for resuming a previous sampling round in specialized Boltzmann sampling, such as non-redundant backtracking. For that purpose, the user passes the address of a Boltzmann sampling data structure (*vrna_pbacktrack_mem_t*) which will be

re-used in each round of sampling, i.e. each successive call to *vrna_pbacktrack_resume_cb()* or *vrna_pbacktrack_resume()*.

A successive sample call to this function may look like:

```
vrna_pbacktrack_mem_t nonredundant_memory = NULL;

// sample the first 100 structures
vrna_pbacktrack_resume(fc,
                       100,
                       &nonredundant_memory,
                       options);

// sample another 500 structures
vrna_pbacktrack_resume(fc,
                       500,
                       &nonredundant_memory,
                       options);

// release memory occupied by the non-redundant memory data structure
vrna_pbacktrack_mem_free(nonredundant_memory);
```

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the list of structures, this function also returns the `nr_mem` data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack_resume_cb(), *vrna_pbacktrack_cb()*, *vrna_pbacktrack5_resume()*, *vrna_pbacktrack_mem_t*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*, *vrna_pbacktrack_mem_free*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **nr_mem** – The address of the Boltzmann sampling memory data structure

- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A set of secondary structure samples in dot-bracket notation terminated by NULL (or NULL on error)

unsigned int **vrna_pbacktrack_resume_cb**(*vrna_fold_compound_t* *fc, unsigned int num_samples, *vrna_bs_result_f* cb, void *data, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

Obtain a set of secondary structure samples from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack5_resume()* this function yields the structure samples through a callback mechanism.

A successive sample call to this function may look like:

```
vrna_pbacktrack_mem_t nonredundant_memory = NULL;

// sample the first 100 structures
vrna_pbacktrack5_resume_cb(fc,
                           100,
                           &callback_function,
                           (void *)&callback_data,
                           &nonredundant_memory,
                           options);

// sample another 500 structures
vrna_pbacktrack5_resume_cb(fc,
                           500,
                           &callback_function,
                           (void *)&callback_data,
                           &nonredundant_memory,
                           options);

// release memory occupied by the non-redundant memory data structure
vrna_pbacktrack_mem_free(nonredundant_memory);
```

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the number of structures backtraced, this function also returns the `nr_mem` data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack_resume(), *vrna_pbacktrack_cb()*, *vrna_pbacktrack5_resume_cb()*, *vrna_pbacktrack_mem_t*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*, *vrna_pbacktrack_mem_free*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **cb** – The callback that receives the sampled structure
- **data** – A data structure passed through to the callback `cb`
- **nr_mem** – The address of the Boltzmann sampling memory data structure
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

The number of structures actually backtraced

char ***vrna_pbacktrack_sub**(*vrna_fold_compound_t* *fc, unsigned int start, unsigned int end)

Sample a secondary structure of a subsequence from the Boltzmann ensemble according its probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a secondary structure. The parameters `start` and `end` specify the interval $[start : end]$ of the subsequence with $1 \leq start < end \leq n$ for sequence length n, the structure $s_{start,end}$ should be drawn from.

The resulting substructure $s_{start,end}$ with free energy $E(s_{start,end})$ is picked from the Boltzmann distributed sub ensemble of all structures within the interval $[start : end]$ according to its probability

$$p(s_{start,end}) = \frac{exp(-E(s_{start,end})/kT)}{Z_{start,end}}$$

with partition function $Z_{start,end} = \sum_{s_{start,end}} exp(-E(s_{start,end})/kT)$, Boltzmann constant k and thermodynamic temperature T.

SWIG Wrapper Notes:

This function is attached as overloaded method `pbacktrack_sub()` to objects of type *fold_compound*. See, e.g. *`RNA.fold_compound.pbacktrack_sub()`* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack_sub_num(), *vrna_pbacktrack_sub_cb()*, *vrna_pbacktrack()*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Parameters

- **`fc`** – The fold compound data structure
- **`start`** – The start of the subsequence to consider, i.e. 5'-end position(1-based)
- **`end`** – The end of the subsequence to consider, i.e. 3'-end position (1-based)

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A sampled secondary structure in dot-bracket notation (or NULL on error)

char ****vrna_pbacktrack_sub_num**(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int start, unsigned int end, unsigned int options)

Obtain a set of secondary structure samples for a subsequence from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures. The parameter `length` specifies the length of the substructure starting from the 5' end.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

SWIG Wrapper Notes:

This function is attached as overloaded method `pbacktrack_sub()` to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *`RNA.fold_compound.pbacktrack_sub()`* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack_sub(), *vrna_pbacktrack_sub_cb()*, *vrna_pbacktrack_num()*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **`fc`** – The fold compound data structure
- **`num_samples`** – The size of the sample set, i.e. number of structures
- **`start`** – The start of the subsequence to consider, i.e. 5'-end position(1-based)
- **`end`** – The end of the subsequence to consider, i.e. 3'-end position (1-based)
- **`options`** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A set of secondary structure samples in dot-bracket notation terminated by NULL (or NULL on error)

unsigned int **vrna_pbacktrack_sub_cb**(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int start, unsigned int end, *vrna_bs_result_f* cb, void *data, unsigned int options)

Obtain a set of secondary structure samples for a subsequence from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures. The parameter `length` specifies the length of the substructure starting from the 5' end.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack5()* and *vrna_pbacktrack5_num()* this function yields the structure samples through a callback mechanism.

SWIG Wrapper Notes:

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound` with optional last argument `options = VRNA_PBACKTRACK_DEFAULT`. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5(), *vrna_pbacktrack5_num()*, *vrna_pbacktrack_cb()*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested

structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **start** – The start of the subsequence to consider, i.e. 5'-end position(1-based)
- **end** – The end of the subsequence to consider, i.e. 3'-end position (1-based)
- **cb** – The callback that receives the sampled structure
- **data** – A data structure passed through to the callback cb
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of fc with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

The number of structures actually backtraced

char **vrna_pbacktrack_sub_resume(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int start, unsigned int end, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

Obtain a set of secondary structure samples for a subsequence from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of num_samples secondary structures. The parameter length specifies the length of the substructure starting from the 5' end.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the options flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack5_cb()* this function allows for resuming a previous sampling round in specialized Boltzmann sampling, such as non-redundant backtracking. For that purpose, the user passes the address of a Boltzmann sampling data structure (*vrna_pbacktrack_mem_t*) which will be re-used in each round of sampling, i.e. each successive call to *vrna_pbacktrack5_resume_cb()* or *vrna_pbacktrack5_resume()*.

A successive sample call to this function may look like:

```
vrna_pbacktrack_mem_t nonredundant_memory = NULL;

// sample the first 100 structures
vrna_pbacktrack5_resume(fc,
                        100,
                        fc->length,
                        &nonredundant_memory,
                        options);

// sample another 500 structures
vrna_pbacktrack5_resume(fc,
                        500,
                        fc->length,
                        &nonredundant_memory,
                        options);

// release memory occupied by the non-redundant memory data structure
vrna_pbacktrack_mem_free(nonredundant_memory);
```

SWIG Wrapper Notes:

This function is attached as overloaded method `pbacktrack_sub()` to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the list of structures, this function also returns the `nr_mem` data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack_sub()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5_resume_cb(), *vrna_pbacktrack5_cb()*, *vrna_pbacktrack_resume()*, *vrna_pbacktrack_mem_t*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*, *vrna_pbacktrack_mem_free*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **start** – The start of the subsequence to consider, i.e. 5'-end position(1-based)
- **end** – The end of the subsequence to consider, i.e. 3'-end position (1-based)

- **nr_mem** – The address of the Boltzmann sampling memory data structure
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with *vrna_fold_compound()* or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns

A set of secondary structure samples in dot-bracket notation terminated by NULL (or NULL on error)

unsigned int **vrna_pbacktrack_sub_resume_cb**(*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int start, unsigned int end, *vrna_bs_result_f* cb, void *data, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

Obtain a set of secondary structure samples for a subsequence from the Boltzmann ensemble according their probability.

```
#include <ViennaRNA/sampling/basic.h>
```

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a set of `num_samples` secondary structures. The parameter `length` specifies the length of the substructure starting from the 5' end.

Any structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Using the `options` flag one can switch between regular (*VRNA_PBACKTRACK_DEFAULT*) backtracing mode, and non-redundant sampling (*VRNA_PBACKTRACK_NON_REDUNDANT*) along the lines of Michálik *et al.* [2017] .

In contrast to *vrna_pbacktrack5_resume()* this function yields the structure samples through a callback mechanism.

A successive sample call to this function may look like:

```
vrna_pbacktrack_mem_t nonredundant_memory = NULL;

// sample the first 100 structures
vrna_pbacktrack5_resume_cb(fc,
                           100,
                           fc->length,
                           &callback_function,
                           (void *)&callback_data,
                           &nonredundant_memory,
                           options);

// sample another 500 structures
vrna_pbacktrack5_resume_cb(fc,
```

(continues on next page)

(continued from previous page)

```
                            500,
                            fc->length,
                            &callback_function,
                            (void *)&callback_data,
                            &nonredundant_memory,
                            options);

// release memory occupied by the non-redundant memory data structure
vrna_pbacktrack_mem_free(nonredundant_memory);
```

SWIG Wrapper Notes:

This function is attached as overloaded method `pbacktrack_sub()` to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the number of structures backtraced, this function also returns the `nr_mem` data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack_sub()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

See also

vrna_pbacktrack5_resume(), *vrna_pbacktrack5_cb()*, *vrna_pbacktrack_resume_cb()*, *vrna_pbacktrack_mem_t*, *VRNA_PBACKTRACK_DEFAULT*, *VRNA_PBACKTRACK_NON_REDUNDANT*, *vrna_pbacktrack_mem_free*

Note

This function is polymorphic. It accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*.

Warning

In non-redundant sampling mode (*VRNA_PBACKTRACK_NON_REDUNDANT*), this function may not yield the full number of requested samples. This may happen if a) the number of requested structures is larger than the total number of structuresin the ensemble, b) numeric instabilities prevent the backtracking function to enumerate structures with high free energies, or c) any other error occurs.

Parameters

- **fc** – The fold compound data structure
- **num_samples** – The size of the sample set, i.e. number of structures
- **start** – The start of the subsequence to consider, i.e. 5'-end position(1-based)
- **end** – The end of the subsequence to consider, i.e. 3'-end position (1-based)
- **cb** – The callback that receives the sampled structure
- **data** – A data structure passed through to the callback `cb`
- **nr_mem** – The address of the Boltzmann sampling memory data structure
- **options** – A bitwise OR-flag indicating the backtracing mode.

Pre

Unique multiloop decomposition has to be active upon creation of `fc` with

vrna_fold_compound() or similar. This can be done easily by passing *vrna_fold_compound()* a model details parameter with *vrna_md_t.uniq_ML* = 1.

vrna_pf() has to be called first to fill the partition function matrices

Returns
The number of structures actually backtraced

void **vrna_pbacktrack_mem_free**(*vrna_pbacktrack_mem_t* s)

Release memory occupied by a Boltzmann sampling memory data structure.

```
#include <ViennaRNA/sampling/basic.h>
```

See also

vrna_pbacktrack_mem_t, *vrna_pbacktrack5_resume()*, *vrna_pbacktrack5_resume_cb()*, *vrna_pbacktrack_resume()*, *vrna_pbacktrack_resume_cb()*

Parameters

- **s** – The non-redundancy memory data structure

Maximum Expected Accuracy (MEA)

Functions

char ***vrna_MEA**(*vrna_fold_compound_t* *fc, double gamma, float *mea)

Compute a MEA (maximum expected accuracy) structure.

```
#include <ViennaRNA/structures/mea.h>
```

The algorithm maximizes the expected accuracy

$$A(S) = \sum_{(i,j)\in S} 2\gamma p_{ij} + \sum_{i \notin S} p_i^u$$

Higher values of γ result in more base pairs of lower probability and thus higher sensitivity. Low values of γ result in structures containing only highly likely pairs (high specificity). The code of the MEA function also demonstrates the use of sparse dynamic programming scheme to reduce the time and memory complexity of folding.

SWIG Wrapper Notes:

This function is attached as overloaded method `MEA(gamma = 1.)` to objects of type `fold_compound`. Note, that it returns the MEA structure and MEA value as a tuple (MEA_structure, MEA). See, e.g. *`RNA.fold_compound.MEA()`* in the *The RNA Python module*.

Parameters

- **fc** – The fold compound data structure with pre-filled base pair probability matrix
- **gamma** – The weighting factor for base pairs vs. unpaired nucleotides

- **mea** – A pointer to a variable where the MEA value will be written to

Pre
vrna_pf() must be executed on input parameter `fc`

Returns
An MEA structure (or NULL on any error)

char ***vrna_MEA_from_plist**(*vrna_ep_t* *plist, const char *sequence, double gamma, *vrna_md_t* *md, float *mea)

Compute a MEA (maximum expected accuracy) structure from a list of probabilities.

```
#include <ViennaRNA/structures/mea.h>
```

The algorithm maximizes the expected accuracy

$$A(S) = \sum_{(i,j)\in S} 2\gamma p_{ij} + \sum_{i \notin S} p_i^u$$

Higher values of γ result in more base pairs of lower probability and thus higher sensitivity. Low values of γ result in structures containing only highly likely pairs (high specificity). The code of the MEA function also demonstrates the use of sparse dynamic programming scheme to reduce the time and memory complexity of folding.

SWIG Wrapper Notes:

This function is available as overloaded function `MEA_from_plist`(gamma = 1., md = NULL). Note, that it returns the MEA structure and MEA value as a tuple (MEA_structure, MEA). See, e.g. *RNA.MEA_from_plist()* in the *The RNA Python module*.

Note

The unpaired probabilities $p_i^u = 1 - \sum_{j \neq i} p_{ij}$ are usually computed from the supplied pairing probabilities p_{ij} as stored in `plist` entries of type *VRNA_PLIST_TYPE_BASEPAIR*. To overwrite individual p_o^u values simply add entries with type *VRNA_PLIST_TYPE_UNPAIRED*

To include G-Quadruplex support, the corresponding field in `md` must be set.

Parameters

- **plist** – A list of base pair probabilities the MEA structure is computed from
- **sequence** – The RNA sequence that corresponds to the list of probability values
- **gamma** – The weighting factor for base pairs vs. unpaired nucleotides
- **md** – A model details data structure (maybe NULL)
- **mea** – A pointer to a variable where the MEA value will be written to

Returns
An MEA structure (or NULL on any error)

float **MEA**(*plist* *p, char *structure, double gamma)

Computes a MEA (maximum expected accuracy) structure.

```
#include <ViennaRNA/structures/mea.h>
```

The algorithm maximizes the expected accuracy

$$A(S) = \sum_{(i,j)\in S} 2\gamma p_{ij} + \sum_{i \notin S} p_i^u$$

Higher values of γ result in more base pairs of lower probability and thus higher sensitivity. Low values of γ result in structures containing only highly likely pairs (high specificity). The code of the MEA function also demonstrates the use of sparse dynamic programming scheme to reduce the time and memory complexity of folding.

Deprecated:

Use *vrna_MEA()* or *vrna_MEA_from_plist()* instead!

The Centroid Structure

Functions

char ***vrna_centroid**(*vrna_fold_compound_t* *fc, double *dist)*

Get the centroid structure of the ensemble.

```
#include <ViennaRNA/structures/centroid.h>
```

The centroid is the structure with the minimal average distance to all other structures

$< d(S) >= \sum_{(i,j)\in S}(1 - p_{ij}) + \sum_{(i,j)\notin S} p_{ij}$

Thus, the centroid is simply the structure containing all pairs with $p_ij > 0.5$ The distance of the centroid to the ensemble is written to the memory adressed by *dist*.

Parameters

- **fc** – **[in]** The fold compound data structure
- **dist** – **[out]** A pointer to the distance variable where the centroid distance will be written to

Returns

The centroid structure of the ensemble in dot-bracket notation (NULL on error)

char ***vrna_centroid_from_plist**(int length, double *dist, *vrna_ep_t* *pl)*

Get the centroid structure of the ensemble.

```
#include <ViennaRNA/structures/centroid.h>
```

This function is a threadsafe replacement for centroid() with a *vrna_ep_t* input

The centroid is the structure with the minimal average distance to all other structures

$< d(S) >= \sum_{(i,j)\in S}(1 - p_{ij}) + \sum_{(i,j)\notin S} p_{ij}$

Thus, the centroid is simply the structure containing all pairs with $p_ij > 0.5$ The distance of the centroid to the ensemble is written to the memory adressed by *dist*.

Parameters

- **length** – **[in]** The length of the sequence
- **dist** – **[out]** A pointer to the distance variable where the centroid distance will be written to

- **pl** – **[in]** A pair list containing base pair probability information about the ensemble

Returns
The centroid structure of the ensemble in dot-bracket notation (NULL on error)

char ***vrna_centroid_from_probs**(int length, double *dist, *FLT_OR_DBL* *probs)

Get the centroid structure of the ensemble.

```
#include <ViennaRNA/structures/centroid.h>
```

This function is a threadsafe replacement for centroid() with a probability array input

The centroid is the structure with the minimal average distance to all other structures

$< d(S) >= \sum_{(i,j) \in S} (1 - p_{ij}) + \sum_{(i,j) \notin S} p_{ij}$

Thus, the centroid is simply the structure containing all pairs with $p_ij > 0.5$ The distance of the centroid to the ensemble is written to the memory adressed by *dist*.

Parameters

- **length** – **[in]** The length of the sequence
- **dist** – **[out]** A pointer to the distance variable where the centroid distance will be written to
- **probs** – **[in]** An upper triangular matrix containing base pair probabilities (access via iindx *vrna_idx_row_wise()*)

Returns
The centroid structure of the ensemble in dot-bracket notation (NULL on error)

4.1.4 Free Energy Evaluation

Energy Parameters

Functions related to the set of thermodynamic **Nearest Neighbor** energy parameters.

Defines

VRNA_GQUAD_MAX_STACK_SIZE

```
#include <ViennaRNA/params/basic.h>
```

VRNA_GQUAD_MIN_STACK_SIZE

```
#include <ViennaRNA/params/basic.h>
```

VRNA_GQUAD_MAX_LINKER_LENGTH

```
#include <ViennaRNA/params/basic.h>
```

VRNA_GQUAD_MIN_LINKER_LENGTH

```
#include <ViennaRNA/params/basic.h>
```

VRNA_GQUAD_MIN_BOX_SIZE

```
#include <ViennaRNA/params/basic.h>
```

VRNA_GQUAD_MAX_BOX_SIZE

```
#include <ViennaRNA/params/basic.h>
```

Typedefs

typedef struct *vrna_param_s* **vrna_param_t**

Typename for the free energy parameter data structure *vrna_params*.

```
#include <ViennaRNA/params/basic.h>
```

typedef struct *vrna_exp_param_s* **vrna_exp_param_t**

Typename for the Boltzmann factor data structure *vrna_exp_params*.

```
#include <ViennaRNA/params/basic.h>
```

typedef struct *vrna_param_s* **paramT**

Old typename of *vrna_param_s*.

```
#include <ViennaRNA/params/basic.h>
```

Deprecated:

Use *vrna_param_t* instead!

typedef struct *vrna_exp_param_s* **pf_paramT**

Old typename of *vrna_exp_param_s*.

```
#include <ViennaRNA/params/basic.h>
```

Deprecated:

Use *vrna_exp_param_t* instead!

Functions

void **vrna_params_dirty**(void)

Set internal parameters-dirty flag.

```
#include <ViennaRNA/params/basic.h>
```

Calling this function sets an internal flag that instructs *vrna_params()* or *vrna_exp_params()* to actually re-read the underlying energy parameters before creating the parameters data structure.

Calling this function might be useful after tempering with any of the global energy parameter variables. Loading energy parameters via one of the *vrna_params_load()* or *vrna_params_load_from_string()* functions will call this one automatically.

vrna_param_t ***vrna_params**(*vrna_md_t* *md)

Get a data structure containing prescaled free energy parameters.

```
#include <ViennaRNA/params/basic.h>
```

If a NULL pointer is passed for the model details parameter, the default model parameters are stored within the requested *vrna_param_t* structure.

See also

vrna_md_t, *vrna_md_set_default()*, *vrna_exp_params()*

Parameters

- **md** – A pointer to the model details to store inside the structure (Maybe NULL)

Returns

A pointer to the memory location where the requested parameters are stored

vrna_param_t ***vrna_params_copy**(*vrna_param_t* *par)

Get a copy of the provided free energy parameters.

```
#include <ViennaRNA/params/basic.h>
```

If NULL is passed as parameter, a default set of energy parameters is created and returned.

See also

vrna_params(), *vrna_param_t*

Parameters

- **par** – The free energy parameters that are to be copied (Maybe NULL)

Returns

A copy or a default set of the (provided) parameters

vrna_exp_param_t ***vrna_exp_params**(*vrna_md_t* *md)

Get a data structure containing prescaled free energy parameters already transformed to Boltzmann factors.

```
#include <ViennaRNA/params/basic.h>
```

This function returns a data structure that contains all necessary precomputed energy contributions for each type of loop.

In contrast to *vrna_params()*, the free energies within this data structure are stored as their Boltzmann factors, i.e.

$exp(-E/kT)$

where E is the free energy.

If a NULL pointer is passed for the model details parameter, the default model parameters are stored within the requested *vrna_exp_param_t* structure.

See also

vrna_md_t, *vrna_md_set_default()*, *vrna_params()*, vrna_rescale_pf_params()

Parameters

- **md** – A pointer to the model details to store inside the structure (Maybe NULL)

Returns

A pointer to the memory location where the requested parameters are stored

vrna_exp_param_t ***vrna_exp_params_comparative**(unsigned int n_seq, *vrna_md_t* *md)

Get a data structure containing prescaled free energy parameters already transformed to Boltzmann factors (alifold version).

```
#include <ViennaRNA/params/basic.h>
```

If a NULL pointer is passed for the model details parameter, the default model parameters are stored within the requested *vrna_exp_param_t* structure.

See also

vrna_md_t, *vrna_md_set_default()*, *vrna_exp_params()*, *vrna_params()*

Parameters

- **n_seq** – The number of sequences in the alignment
- **md** – A pointer to the model details to store inside the structure (Maybe NULL)

Returns

A pointer to the memory location where the requested parameters are stored

vrna_exp_param_t ***vrna_exp_params_copy**(*vrna_exp_param_t* *par)

Get a copy of the provided free energy parameters (provided as Boltzmann factors).

```
#include <ViennaRNA/params/basic.h>
```

If NULL is passed as parameter, a default set of energy parameters is created and returned.

See also

vrna_exp_params(), *vrna_exp_param_t*

Parameters

- **par** – The free energy parameters that are to be copied (Maybe NULL)

Returns

A copy or a default set of the (provided) parameters

void **vrna_params_subst**(*vrna_fold_compound_t* *fc, *vrna_param_t* *par)

Update/Reset energy parameters data structure within a *vrna_fold_compound_t*.

```
#include <ViennaRNA/params/basic.h>
```

Passing NULL as second argument leads to a reset of the energy parameters within fc to their default values. Otherwise, the energy parameters provided will be copied over into fc.

SWIG Wrapper Notes:

This function is attached to *vrna_fc_s* objects as overloaded `params_subst()` method.

When no parameter is passed, the resulting action is the same as passing `NULL` as second parameter to *vrna_params_subst()*, i.e. resetting the parameters to the global defaults. See, e.g. *`RNA.fold_compound.params_subst()`* in the *The RNA Python module*.

See also

vrna_params_reset(), *vrna_param_t*, *vrna_md_t*, *vrna_params()*

Parameters

- **fc** – The *vrna_fold_compound_t* that is about to receive updated energy parameters
- **par** – The energy parameters used to substitute those within fc (Maybe NULL)

void **vrna_exp_params_subst**(*vrna_fold_compound_t* *fc, *vrna_exp_param_t* *params)

Update the energy parameters for subsequent partition function computations.

```
#include <ViennaRNA/params/basic.h>
```

This function can be used to properly assign new energy parameters for partition function computations to a *vrna_fold_compound_t*. For this purpose, the data of the provided pointer `params` will be copied into `fc` and a recomputation of the partition function scaling factor is issued, if the *`pf_scale`* attribute of `params` is less than `1.0`.

Passing NULL as second argument leads to a reset of the energy parameters within fc to their default values

SWIG Wrapper Notes:

This function is attached to *vrna_fc_s* objects as overloaded `exp_params_subst()` method.

When no parameter is passed, the resulting action is the same as passing `NULL` as second parameter to *vrna_exp_params_subst()*, i.e. resetting the parameters to the global defaults. See, e.g. *`RNA.fold_compound.exp_params_subst()`* in the *The RNA Python module*.

See also

vrna_exp_params_reset(), *vrna_exp_params_rescale()*, *vrna_exp_param_t*, *vrna_md_t*, *vrna_exp_params()*

Parameters

- **fc** – The fold compound data structure
- **params** – A pointer to the new energy parameters

void **vrna_exp_params_rescale**(*vrna_fold_compound_t* *fc, double *mfe)

Rescale Boltzmann factors for partition function computations.

```
#include <ViennaRNA/params/basic.h>
```

This function may be used to (automatically) rescale the Boltzmann factors used in partition function computations. Since partition functions over subsequences can easily become extremely large, the RNAlib internally rescales them to avoid numerical over- and/or underflow. Therefore, a proper scaling factor s needs to be chosen that in turn is then used to normalize the corresponding partition functions $\hat{q}[i,j] = q[i,j]/s^{(j-i+1)}$.

This function provides two ways to automatically adjust the scaling factor.

a. Automatic guess

b. Automatic adjustment according to MFE

Passing `NULL` as second parameter activates the *automatic guess mode*. Here, the scaling factor is recomputed according to a mean free energy of `184.3*length` cal for random sequences.

On the other hand, if the MFE for a sequence is known, it can be used to recompute a more robust scaling factor, since it represents the lowest free energy of the entire ensemble of structures, i.e. the highest Boltzmann factor. To activate this second mode of *automatic adjustment according to MFE*, a pointer to the MFE value needs to be passed as second argument. This value is then taken to compute the scaling factor as $s = exp((sfact*MFE)/kT/length)$, where sfact is an additional scaling weight located in the *vrna_md_t* data structure of `exp_params` in `fc`.

The computed scaling factor s will be stored as `pf_scale` attribute of the `exp_params` data structure in `fc`.

SWIG Wrapper Notes:

This function is attached to *vrna_fc_s* objects as overloaded `exp_params_rescale()` method.

When no parameter is passed to this method, the resulting action is the same as passing `NULL` as second parameter to *vrna_exp_params_rescale()*, i.e. default scaling of the partition function. Passing an energy in kcal/mol, e.g. as retrieved by a previous call to the `mfe()` method, instructs all subsequent calls to scale the partition function accordingly. See, e.g. *RNA.fold_compound.exp_params_rescale()* in the *The RNA Python module*.

See also

vrna_exp_params_subst(), *vrna_md_t*, *vrna_exp_param_t*, *vrna_fold_compound_t*

Note

This recomputation only takes place if the `pf_scale` attribute of the `exp_params` data structure contained in `fc` has a value below `1.0`.

Parameters

- **fc** – The fold compound data structure
- **mfe** – A pointer to the MFE (in kcal/mol) or NULL

void **vrna_params_reset**(*vrna_fold_compound_t* *fc, *vrna_md_t* *md)

Reset free energy parameters within a *vrna_fold_compound_t* according to provided, or default model details.

```
#include <ViennaRNA/params/basic.h>
```

This function allows one to rescale free energy parameters for subsequent structure prediction or evaluation according to a set of model details, e.g. temperature values. To do so, the caller provides either a pointer to a set of model details to be used for rescaling, or NULL if global default setting should be used.

SWIG Wrapper Notes:

This function is attached to *vrna_fc_s* objects as overloaded `params_reset()` method.

When no parameter is passed to this method, the resulting action is the same as passing `NULL` as second parameter to *vrna_params_reset()*, i.e. global default model settings are used. Passing an object of type *vrna_md_s* resets the fold compound according to the specifications stored within the *vrna_md_s* object. See, e.g. *RNA.fold_compound.params_reset()* in the *The RNA Python module*.

See also

vrna_exp_params_reset(), vrna_params_subs()

Parameters

- **fc** – The fold compound data structure
- **md** – A pointer to the new model details (or NULL for reset to defaults)

void **vrna_exp_params_reset**(*vrna_fold_compound_t* *fc, *vrna_md_t* *md)

Reset Boltzmann factors for partition function computations within a *vrna_fold_compound_t* according to provided, or default model details.

```
#include <ViennaRNA/params/basic.h>
```

This function allows one to rescale Boltzmann factors for subsequent partition function computations according to a set of model details, e.g. temperature values. To do so, the caller provides either a pointer to a set of model details to be used for rescaling, or NULL if global default setting should be used.

SWIG Wrapper Notes:

This function is attached to *vrna_fc_s* objects as overloaded `exp_params_reset()` method.

When no parameter is passed to this method, the resulting action is the same as passing `NULL` as second parameter to *vrna_exp_params_reset()*, i.e. global default model settings are used. Passing an object of type *vrna_md_s* resets the fold compound according to the specifications stored within the *vrna_md_s* object. See, e.g. *RNA.fold_compound.exp_params_reset()* in the *The RNA Python module*.

See also

vrna_params_reset(), *vrna_exp_params_subst()*, *vrna_exp_params_rescale()*

Parameters

- **fc** – The fold compound data structure
- **md** – A pointer to the new model details (or NULL for reset to defaults)

void **vrna_params_prepare**(*vrna_fold_compound_t* *fc, unsigned int options)

```
#include <ViennaRNA/params/basic.h>
```

vrna_param_t ***get_parameter_copy**(*vrna_param_t* *par)

```
#include <ViennaRNA/params/basic.h>
```

vrna_exp_param_t ***get_scaled_pf_parameters**(void)

```
#include <ViennaRNA/params/basic.h>
```

get a data structure of type *vrna_exp_param_t* which contains the Boltzmann weights of several energy parameters scaled according to the current temperature

Deprecated:

Use *vrna_exp_params()* instead!

Returns

The data structure containing Boltzmann weights for use in partition function calculations

vrna_exp_param_t ***get_boltzmann_factors**(double temperature, double betaScale, *vrna_md_t* md, double pf_scale)

Get precomputed Boltzmann factors of the loop type dependent energy contributions with independent thermodynamic temperature.

```
#include <ViennaRNA/params/basic.h>
```

This function returns a data structure that contains all necessary precalculated Boltzmann factors for each loop type contribution.

In contrast to *get_scaled_pf_parameters()*, this function enables setting of independent temperatures for both, the individual energy contributions as well as the thermodynamic temperature used in $exp(-\Delta G/kT)$

Deprecated:

Use *vrna_exp_params()* instead!

See also

get_scaled_pf_parameters(), *get_boltzmann_factor_copy()*

Parameters

- **temperature** – The temperature in degrees Celcius used for (re-)scaling the energy contributions
- **betaScale** – A scaling value that is used as a multiplication factor for the absolute temperature of the system
- **md** – The model details to be used
- **pf_scale** – The scaling factor for the Boltzmann factors

Returns
A set of precomputed Boltzmann factors

vrna_exp_param_t ***get_boltzmann_factor_copy**(*vrna_exp_param_t* *parameters)

Get a copy of already precomputed Boltzmann factors.

```
#include <ViennaRNA/params/basic.h>
```

Deprecated:
Use *vrna_exp_params_copy()* instead!

See also
get_boltzmann_factors(), *get_scaled_pf_parameters()*

Parameters
- **parameters** – The input data structure that shall be copied

Returns
A copy of the provided Boltzmann factor data set

vrna_exp_param_t ***get_scaled_alipf_parameters**(unsigned int n_seq)

Get precomputed Boltzmann factors of the loop type dependent energy contributions (alifold variant).

```
#include <ViennaRNA/params/basic.h>
```

Deprecated:
Use *vrna_exp_params_comparative()* instead!

vrna_exp_param_t ***get_boltzmann_factors_ali**(unsigned int n_seq, double temperature, double betaScale, *vrna_md_t* md, double pf_scale)

Get precomputed Boltzmann factors of the loop type dependent energy contributions (alifold variant) with independent thermodynamic temperature.

```
#include <ViennaRNA/params/basic.h>
```

Deprecated:
Use *vrna_exp_params_comparative()* instead!

vrna_param_t ***scale_parameters**(void)

Get precomputed energy contributions for all the known loop types.

```
#include <ViennaRNA/params/basic.h>
```

Deprecated:
Use *vrna_params()* instead!

> **Note**
>
> OpenMP: This function relies on several global model settings variables and thus is not to be considered threadsafe. See *get_scaled_parameters()* for a completely threadsafe implementation.

Returns
A set of precomputed energy contributions

vrna_param_t ***get_scaled_parameters**(double temperature, *vrna_md_t* md)

Get precomputed energy contributions for all the known loop types.

```
#include <ViennaRNA/params/basic.h>
```

Call this function to retrieve precomputed energy contributions, i.e. scaled according to the temperature passed. Furthermore, this function assumes a data structure that contains the model details as well, such that subsequent folding recursions are able to retrieve the correct model settings

Deprecated:
Use *vrna_params()* instead!

> **See also**
>
> *vrna_md_t*, *set_model_details()*

Parameters

- **temperature** – The temperature in degrees Celcius
- **md** – The model details

Returns
precomputed energy contributions and model settings

vrna_param_t ***copy_parameters**(void)

```
#include <ViennaRNA/params/basic.h>
```

vrna_param_t ***set_parameters**(*vrna_param_t* *dest)

```
#include <ViennaRNA/params/basic.h>
```

vrna_exp_param_t ***scale_pf_parameters**(void)

```
#include <ViennaRNA/params/basic.h>
```

vrna_exp_param_t ***copy_pf_param**(void)

```
#include <ViennaRNA/params/basic.h>
```

vrna_exp_param_t ***set_pf_param**(*vrna_param_t* *dest)

```
#include <ViennaRNA/params/basic.h>
```

float **get_ribosum(const char **Alseq, int n_seq, int length)

Retrieve a RiboSum Scoring Matrix for a given Alignment.

```
#include <ViennaRNA/params/ribosum.h>
```

struct **vrna_param_s**

#include <ViennaRNA/params/basic.h> The datastructure that contains temperature scaled energy parameters.

Public Members

int **id**

```
#include <ViennaRNA/params/basic.h>
```

int **stack**[7 + 1][7 + 1]

```
#include <ViennaRNA/params/basic.h>
```

int **hairpin**[31]

```
#include <ViennaRNA/params/basic.h>
```

int **bulge**[30 + 1]

```
#include <ViennaRNA/params/basic.h>
```

int **internal_loop**[30 + 1]

```
#include <ViennaRNA/params/basic.h>
```

int **mismatchExt**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **mismatchI**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **mismatch1nI**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **mismatch23I**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **mismatchH**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **mismatchM**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **dangle5**[7 + 1][5]

```
#include <ViennaRNA/params/basic.h>
```

int **dangle3**[7 + 1][5]

```
#include <ViennaRNA/params/basic.h>
```

int **int11**[7 + 1][7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **int21**[7 + 1][7 + 1][5][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **int22**[7 + 1][7 + 1][5][5][5][5]

```
#include <ViennaRNA/params/basic.h>
```

int **ninio**[5]

```
#include <ViennaRNA/params/basic.h>
```

double **lxc**

```
#include <ViennaRNA/params/basic.h>
```

int **MLbase**

```
#include <ViennaRNA/params/basic.h>
```

int **MLintern**[7 + 1]

```
#include <ViennaRNA/params/basic.h>
```

int **MLclosing**

```
#include <ViennaRNA/params/basic.h>
```

int **TerminalAU**

```
#include <ViennaRNA/params/basic.h>
```

int **DuplexInit**

```
#include <ViennaRNA/params/basic.h>
```

int **Tetraloop_E**[200]

```
#include <ViennaRNA/params/basic.h>
```

char **Tetraloops**[1401]

```
#include <ViennaRNA/params/basic.h>
```

int **Triloop_E**[40]

```
#include <ViennaRNA/params/basic.h>
```

char **Triloops**[241]

```
#include <ViennaRNA/params/basic.h>
```

int **Hexaloop_E**[40]

```
#include <ViennaRNA/params/basic.h>
```

char **Hexaloops**[1801]

```
#include <ViennaRNA/params/basic.h>
```

int **TripleC**

```
#include <ViennaRNA/params/basic.h>
```

int **MultipleCA**

```
#include <ViennaRNA/params/basic.h>
```

int **MultipleCB**

```
#include <ViennaRNA/params/basic.h>
```

int **gquad**[7 + 1][3 * 15 + 1]

```
#include <ViennaRNA/params/basic.h>
```

int **gquadLayerMismatch**

```
#include <ViennaRNA/params/basic.h>
```

unsigned int **gquadLayerMismatchMax**

```
#include <ViennaRNA/params/basic.h>
```

double **temperature**

Temperature used for loop contribution scaling.

```
#include <ViennaRNA/params/basic.h>
```

vrna_md_t **model_details**

Model details to be used in the recursions.

```
#include <ViennaRNA/params/basic.h>
```

char **param_file**[256]

The filename the parameters were derived from, or empty string if they represent the default.

```
#include <ViennaRNA/params/basic.h>
```

int **SaltStack**

```
#include <ViennaRNA/params/basic.h>
```

int **SaltLoop**[30 + 2]

```
#include <ViennaRNA/params/basic.h>
```

double **SaltLoopDbl**[30 + 2]

```
#include <ViennaRNA/params/basic.h>
```

int **SaltMLbase**

```
#include <ViennaRNA/params/basic.h>
```

int **SaltMLintern**

```
#include <ViennaRNA/params/basic.h>
```

int **SaltMLclosing**

```
#include <ViennaRNA/params/basic.h>
```

int **SaltDPXInit**

```
#include <ViennaRNA/params/basic.h>
```

struct **vrna_exp_param_s**

#include <ViennaRNA/params/basic.h> The data structure that contains temperature scaled Boltzmann weights of the energy parameters.

Public Members

int **id**

An identifier for the data structure.

```
#include <ViennaRNA/params/basic.h>
```

Deprecated:

This attribute will be removed in version 3

double **expstack**[7 + 1][7 + 1]

```
#include <ViennaRNA/params/basic.h>
```

double **exphairpin**[31]

```
#include <ViennaRNA/params/basic.h>
```

double **expbulge**[30 + 1]

```
#include <ViennaRNA/params/basic.h>
```

double **expinternal**[30 + 1]

```
#include <ViennaRNA/params/basic.h>
```

double **expmismatchExt**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expmismatchI**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expmismatch23I**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expmismatch1nI**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expmismatchH**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expmismatchM**[7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expdangle5**[7 + 1][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expdangle3**[7 + 1][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expint11**[7 + 1][7 + 1][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expint21**[7 + 1][7 + 1][5][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expint22**[7 + 1][7 + 1][5][5][5][5]

```
#include <ViennaRNA/params/basic.h>
```

double **expninio**[5][30 + 1]

```
#include <ViennaRNA/params/basic.h>
```

double **lxc**

```
#include <ViennaRNA/params/basic.h>
```

double **expMLbase**

```
#include <ViennaRNA/params/basic.h>
```

double **expMLintern**[7 + 1]

```
#include <ViennaRNA/params/basic.h>
```

double **expMLclosing**

```
#include <ViennaRNA/params/basic.h>
```

double **expTermAU**

```
#include <ViennaRNA/params/basic.h>
```

double **expDuplexInit**

```
#include <ViennaRNA/params/basic.h>
```

double **exptetra**[40]

```
#include <ViennaRNA/params/basic.h>
```

double **exptri**[40]

```
#include <ViennaRNA/params/basic.h>
```

double **exphex**[40]

```
#include <ViennaRNA/params/basic.h>
```

char **Tetraloops**[1401]

```
#include <ViennaRNA/params/basic.h>
```

double **expTriloop**[40]

```
#include <ViennaRNA/params/basic.h>
```

char **Triloops**[241]

```
#include <ViennaRNA/params/basic.h>
```

char **Hexaloops**[1801]

```
#include <ViennaRNA/params/basic.h>
```

double **expTripleC**

```
#include <ViennaRNA/params/basic.h>
```

double **expMultipleCA**

```
#include <ViennaRNA/params/basic.h>
```

double **expMultipleCB**

```
#include <ViennaRNA/params/basic.h>
```

double **expgquad**[7 + 1][3 * 15 + 1]

```
#include <ViennaRNA/params/basic.h>
```

double **expgquadLayerMismatch**

```
#include <ViennaRNA/params/basic.h>
```

unsigned int **gquadLayerMismatchMax**

```
#include <ViennaRNA/params/basic.h>
```

double **kT**

```
#include <ViennaRNA/params/basic.h>
```

double **pf_scale**

Scaling factor to avoid over-/underflows.

```
#include <ViennaRNA/params/basic.h>
```

double **temperature**

Temperature used for loop contribution scaling.

```
#include <ViennaRNA/params/basic.h>
```

double **alpha**

Scaling factor for the thermodynamic temperature.

```
#include <ViennaRNA/params/basic.h>
```

This allows for temperature scaling in Boltzmann factors independently from the energy contributions. The resulting Boltzmann factors are then computed by $e^{-E/(\alpha \cdot K \cdot T)}$

vrna_md_t **model_details**

Model details to be used in the recursions.

```
#include <ViennaRNA/params/basic.h>
```

char **param_file**[256]

The filename the parameters were derived from, or empty string if they represent the default.

```
#include <ViennaRNA/params/basic.h>
```

double **expSaltStack**

```
#include <ViennaRNA/params/basic.h>
```

double **expSaltLoop**[30 + 2]

```
#include <ViennaRNA/params/basic.h>
```

double **SaltLoopDbl**[30 + 2]

```
#include <ViennaRNA/params/basic.h>
```

int **SaltMLbase**

```
#include <ViennaRNA/params/basic.h>
```

int **SaltMLintern**

```
#include <ViennaRNA/params/basic.h>
```

int **SaltMLclosing**

```
#include <ViennaRNA/params/basic.h>
```

int **SaltDPXInit**

```
#include <ViennaRNA/params/basic.h>
```

Salt Corrections

All relevant functions to compute salt correction at a given salt concentration and temperature.

Functions

double **vrna_salt_loop**(int L, double salt, double T, double backbonelen)

Get salt correction for a loop at a given salt concentration and temperature.

```
#include <ViennaRNA/params/salt.h>
```

Parameters

- **L** – backbone number in loop
- **salt** – salt concentration (M)
- **T** – absolute temperature (K)
- **backbonelen** – Backbone Length, phosphate-to-phosphate distance (typically 6 for RNA, 6.76 for DNA)

Returns

Salt correction for loop in dcal/mol

int **vrna_salt_loop_int**(int L, double salt, double T, double backbonelen)

Get salt correction for a loop at a given salt concentration and temperature.

```
#include <ViennaRNA/params/salt.h>
```

This functions is same as vrna_salt_loop but returns rounded salt correction in integer

See also

vrna_salt_loop

Parameters

- **L** – backbone number in loop
- **salt** – salt concentration (M)
- **T** – absolute temperature (K)
- **backbonelen** – Backbone Length, phosphate-to-phosphate distance (typically 6 for RNA, 6.76 for DNA)

Returns

Rounded salt correction for loop in dcal/mol

int **vrna_salt_stack**(double salt, double T, double hrise)

Get salt correction for a stack at a given salt concentration and temperature.

```
#include <ViennaRNA/params/salt.h>
```

Parameters

- **salt** – salt concentration (M)
- **T** – absolute temperature (K)
- **hrise** – Helical Rise (typically 2.8 for RNA, 3.4 for DNA)

Returns

Rounded salt correction for stack in dcal/mol

void **vrna_salt_ml**(double saltLoop[], int lower, int upper, int *m, int *b)

Fit linear function to loop salt correction.

```
#include <ViennaRNA/params/salt.h>
```

For a given range of loop size (backbone number), we perform a linear fitting on loop salt correction

$$\text{Loop correction} \approx m \cdot L + b.$$

See also

vrna_salt_loop()

Parameters

- **saltLoop** – List of loop salt correction of size from 1
- **lower** – Define the size lower bound for fitting
- **upper** – Define the size upper bound for fitting
- **m** – pointer to store the parameter m in fitting result
- **b** – pointer to store the parameter b in fitting result

int **vrna_salt_duplex_init**(*vrna_md_t* *md)

Get salt correction for duplex initialization at a given salt concentration.

```
#include <ViennaRNA/params/salt.h>
```

Parameters

- **md** – Model details data structure that specfifies salt concentration in buffer (M)

Returns

Rounded correction for duplex initialization in dcal/mol

Loading / Saving Energy Parameter Sets

Read and Write energy parameter sets from and to files or strings

Defines

VRNA_PARAMETER_FORMAT_DEFAULT

Default Energy Parameter File format.

```
#include <ViennaRNA/params/io.h>
```

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*

Enums

enum **parset**

```
#include <ViennaRNA/params/io.h>
```

Values:

enumerator **UNKNOWN**

enumerator **QUIT**

enumerator **S**

enumerator **S_H**

enumerator **HP**

enumerator **HP_H**

enumerator **B**

enumerator **B_H**

enumerator **IL**

enumerator **IL_H**

enumerator **MMH**

enumerator **MMH_H**

enumerator **MMI**

enumerator **MMI_H**

enumerator **MMI1N**

enumerator **MMI1N_H**

enumerator **MMI23**

enumerator **MMI23_H**

enumerator **MMM**

enumerator **MMM_H**

enumerator **MME**

enumerator **MME_H**

enumerator **D5**

enumerator **D5_H**

enumerator **D3**

enumerator **D3_H**

enumerator **INT11**

enumerator **INT11_H**

enumerator **INT21**

enumerator **INT21_H**

enumerator **INT22**

enumerator **INT22_H**

enumerator **ML**

enumerator **TL**

enumerator **TRI**

enumerator **HEX**

enumerator **NIN**

enumerator **MISC**

Functions

int **vrna_params_load**(const char fname[], unsigned int options)

Load energy parameters from a file.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as overloaded function `params_load`(fname="", options=*VRNA_PARAMETER_FORMAT_DEFAULT*). Here, the empty filename string indicates to load default RNA parameters, i.e. this is equivalent to calling *vrna_params_load_defaults()*. See, e.g. `RNA.fold_compound.params_load()` in the *The RNA Python module*.

See also

vrna_params_load_from_string(), *vrna_params_save()*, *vrna_params_load_defaults()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_DNA_Mathews1999()*

Parameters

- **fname** – The path to the file containing the energy parameters
- **options** – File format bit-mask (usually *VRNA_PARAMETER_FORMAT_DEFAULT*)

Returns
Non-zero on success, 0 on failure

int **vrna_params_save**(const char fname[], unsigned int options)

Save energy parameters to a file.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as overloaded function `params_save`(fname, options=*VRNA_PARAMETER_FORMAT_DEFAULT*). See, e.g. *RNA.params_save()* in the *The RNA Python module*.

See also

vrna_params_load()

Parameters

- **fname** – A filename (path) for the file where the current energy parameters will be written to
- **options** – File format bit-mask (usually *VRNA_PARAMETER_FORMAT_DEFAULT*)

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_from_string**(const char *string, const char *name, unsigned int options)

Load energy paramters from string.

```
#include <ViennaRNA/params/io.h>
```

The string must follow the default energy parameter file convention! The optional `name` argument allows one to specify a name for the parameter set which is stored internally.

SWIG Wrapper Notes:

This function is available as overloaded function `params_load_from_string`(string, name="", options=*VRNA_PARAMETER_FORMAT_DEFAULT*). See, e.g. *RNA.params_load_from_string()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_save()*, *vrna_params_load_defaults()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_DNA_Mathews1999()*

Parameters

- **string** – A 0-terminated string containing energy parameters
- **name** – A name for the parameter set in `string` (Maybe `NULL`)
- **options** – File format bit-mask (usually *VRNA_PARAMETER_FORMAT_DEFAULT*)

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_defaults**(void)

Load default RNA energy parameter set.

```
#include <ViennaRNA/params/io.h>
```

This is a convenience function to load the Turner 2004 RNA free energy parameters. It's the same as calling *vrna_params_load_RNA_Turner2004()*

SWIG Wrapper Notes:

This function is available as overloaded function `params_load()`. See, e.g. *RNA.params_load()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_DNA_Mathews1999()*

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_RNA_Turner2004**(void)

Load Turner 2004 RNA energy parameter set.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as function `params_load_RNA_Turner2004()`. See, e.g. *RNA.params_load_RNA_Turner2004()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*, *vrna_params_load_defaults()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_DNA_Mathews1999()*

Warning

This function also resets the default geometric parameters as stored in *vrna_md_t* to those of RNA. Only subsequently initialized *vrna_md_t* structures will be affected by this change.

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_RNA_Turner1999**(void)

Load Turner 1999 RNA energy parameter set.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as function `params_load_RNA_Turner1999()`. See, e.g. *RNA.params_load_RNA_Turner1999()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_defaults()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_DNA_Mathews1999()*

Warning

This function also resets the default geometric parameters as stored in *vrna_md_t* to those of RNA. Only subsequently initialized *vrna_md_t* structures will be affected by this change.

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_RNA_Andronescu2007**(void)

Load Andronsecu 2007 RNA energy parameter set.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as function `params_load_RNA_Andronescu2007()`. See, e.g. *RNA.params_load_RNA_Andronescu2007()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_defaults()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_DNA_Mathews1999()*

Warning

This function also resets the default geometric parameters as stored in *vrna_md_t* to those of RNA. Only subsequently initialized *vrna_md_t* structures will be affected by this change.

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_RNA_Langdon2018**(void)

Load Langdon 2018 RNA energy parameter set.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as function `params_load_RNA_Langdon2018()`. See, e.g. *RNA.params_load_RNA_Langdon2018()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_defaults()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_DNA_Mathews1999()*

Warning

This function also resets the default geometric parameters as stored in *vrna_md_t* to those of RNA. Only subsequently initialized *vrna_md_t* structures will be affected by this change.

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_RNA_misc_special_hairpins**(void)

Load Misc Special Hairpin RNA energy parameter set.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as function `params_load_RNA_misc_special_hairpins()`. See, e.g. *RNA.params_load_RNA_misc_special_hairpins()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_defaults()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_DNA_Mathews1999()*

Warning

This function also resets the default geometric parameters as stored in *vrna_md_t* to those of RNA. Only subsequently initialized *vrna_md_t* structures will be affected by this change.

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_DNA_Mathews2004**(void)

Load Mathews 2004 DNA energy parameter set.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as function `params_load_DNA_Mathews2004()`. See, e.g. *RNA.params_load_DNA_Mathews2004()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_defaults()*, *vrna_params_load_DNA_Mathews1999()*

Warning

This function also resets the default geometric parameters as stored in *vrna_md_t* to those of DNA. Only subsequently initialized *vrna_md_t* structures will be affected by this change.

Returns
Non-zero on success, 0 on failure

int **vrna_params_load_DNA_Mathews1999**(void)

Load Mathews 1999 DNA energy parameter set.

```
#include <ViennaRNA/params/io.h>
```

SWIG Wrapper Notes:

This function is available as function `params_load_DNA_Mathews1999()`. See, e.g. *RNA.params_load_DNA_Mathews1999()* in the *The RNA Python module*.

See also

vrna_params_load(), *vrna_params_load_from_string()*, *vrna_params_save()*, *vrna_params_load_RNA_Turner2004()*, *vrna_params_load_RNA_Turner1999()*, *vrna_params_load_RNA_Andronescu2007()*, *vrna_params_load_RNA_Langdon2018()*, *vrna_params_load_RNA_misc_special_hairpins()*, *vrna_params_load_DNA_Mathews2004()*, *vrna_params_load_defaults()*

Warning

This function also resets the default geometric parameters as stored in *vrna_md_t* to those of DNA. Only subsequently initialized *vrna_md_t* structures will be affected by this change.

Returns
Non-zero on success, 0 on failure

const char ***last_parameter_file**(void)

Get the file name of the parameter file that was most recently loaded.

```
#include <ViennaRNA/params/io.h>
```

Returns
The file name of the last parameter file, or NULL if parameters are still at defaults

void **read_parameter_file**(const char fname[])

Read energy parameters from a file.

```
#include <ViennaRNA/params/io.h>
```

Deprecated:
Use *vrna_params_load()* instead!

Parameters
- **fname** – The path to the file containing the energy parameters

void **write_parameter_file**(const char fname[])

Write energy parameters to a file.

```
#include <ViennaRNA/params/io.h>
```

Deprecated:
Use *vrna_params_save()* instead!

Parameters
- **fname** – A filename (path) for the file where the current energy parameters will be written to

enum *parset* **gettype**(const char *ident)

```
#include <ViennaRNA/params/io.h>
```

char ***settype**(enum *parset* s)

```
#include <ViennaRNA/params/io.h>
```

Converting Energy Parameter Files

Converting energy parameter files into the latest format.

Defines

VRNA_CONVERT_OUTPUT_ALL

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of a complete parameter set

VRNA_CONVERT_OUTPUT_HP

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of hairpin contributions

VRNA_CONVERT_OUTPUT_STACK

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of base pair stack contributions

VRNA_CONVERT_OUTPUT_MM_HP

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of hairpin mismatch contribution

VRNA_CONVERT_OUTPUT_MM_INT

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of internal loop mismatch contribution

VRNA_CONVERT_OUTPUT_MM_INT_1N

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of 1:n internal loop mismatch contribution

VRNA_CONVERT_OUTPUT_MM_INT_23

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of 2:3 internal loop mismatch contribution

VRNA_CONVERT_OUTPUT_MM_MULTI

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of multi loop mismatch contribution

VRNA_CONVERT_OUTPUT_MM_EXT

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of exterior loop mismatch contribution

VRNA_CONVERT_OUTPUT_DANGLE5

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of 5' dangle conctribution

VRNA_CONVERT_OUTPUT_DANGLE3

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of 3' dangle contribution

VRNA_CONVERT_OUTPUT_INT_11

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of 1:1 internal loop contribution

VRNA_CONVERT_OUTPUT_INT_21

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of 2:1 internal loop contribution

VRNA_CONVERT_OUTPUT_INT_22

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of 2:2 internal loop contribution

VRNA_CONVERT_OUTPUT_BULGE

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of bulge loop contribution

VRNA_CONVERT_OUTPUT_INT

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of internal loop contribution

VRNA_CONVERT_OUTPUT_ML

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of multi loop contribution

VRNA_CONVERT_OUTPUT_MISC

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of misc contributions (such as terminalAU)

VRNA_CONVERT_OUTPUT_SPECIAL_HP

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of special hairpin contributions (tri-, tetra-, hexa-loops)

VRNA_CONVERT_OUTPUT_VANILLA

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of given parameters only

> **Note**
>
> This option overrides all other output options, except *VRNA_CONVERT_OUTPUT_DUMP* !

VRNA_CONVERT_OUTPUT_NINIO

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate printing of internal loop asymmetry contribution

VRNA_CONVERT_OUTPUT_DUMP

```
#include <ViennaRNA/params/convert.h>
```

Flag to indicate dumping the energy contributions from the library instead of an input file

Functions

void **convert_parameter_file**(const char *iname, const char *oname, unsigned int options)

```
#include <ViennaRNA/params/convert.h>
```

Convert/dump a Vienna 1.8.4 formatted energy parameter file

The options argument allows one to control the different output modes.

Currently available options are: *VRNA_CONVERT_OUTPUT_ALL*, *VRNA_CONVERT_OUTPUT_HP*, *VRNA_CONVERT_OUTPUT_STACK* *VRNA_CONVERT_OUTPUT_MM_HP*, *VRNA_CONVERT_OUTPUT_MM_INT*, *VRNA_CONVERT_OUTPUT_MM_INT_1N* *VRNA_CONVERT_OUTPUT_MM* *VRNA_CONVERT_OUTPUT_MM_MULTI*, *VRNA_CONVERT_OUTPUT_MM_EXT* *VRNA_CONVERT_OUTPUT_DA* *VRNA_CONVERT_OUTPUT_DANGLE3*, *VRNA_CONVERT_OUTPUT_INT_11* *VRNA_CONVERT_OUTPUT_INT_21* *VRNA_CONVERT_OUTPUT_INT_22*, *VRNA_CONVERT_OUTPUT_BULGE* *VRNA_CONVERT_OUTPUT_INT*, *VRNA_CONVERT_OUTPUT_ML*, *VRNA_CONVERT_OUTPUT_MISC* *VRNA_CONVERT_OUTPUT_SPECIAL_HP*, *VRNA_CONVERT_OUTPUT_VANILLA*, *VRNA_CONVERT_OUTPUT_NINIO* *VRNA_CONVERT_OUTPUT_DUMP*

The defined options are fine for bitwise compare- and assignment-operations, e. g.: pass a collection of options as a single value like this:

```
convert_parameter_file(ifile, ofile, option_1 | option_2 | option_n)
```

Parameters

- **iname** – The input file name (If NULL input is read from stdin)

- **oname** – The output file name (If NULL output is written to stdout)
- **options** – The options (as described above)

Fine-tuning of the Implemented Models

Functions and data structures to fine-tune the implemented secondary structure evaluation model.

Defines

NBASES

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_TEMPERATURE

Default temperature for structure prediction and free energy evaluation in $^\circ C$

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.temperature, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_PF_SCALE

Default scaling factor for partition function computations.

```
#include <ViennaRNA/model.h>
```

See also

vrna_exp_param_t.pf_scale, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_BETA_SCALE

Default scaling factor for absolute thermodynamic temperature in Boltzmann factors.

```
#include <ViennaRNA/model.h>
```

See also

vrna_exp_param_t.alpha, *vrna_md_t.betaScale*, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_DANGLES

Default dangling end model.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.dangles, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_SPECIAL_HP

Default model behavior for lookup of special tri-, tetra-, and hexa-loops.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.special_hp, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_NO_LP

Default model behavior for so-called 'lonely pairs'.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.noLP, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_NO_GU

Default model behavior for G-U base pairs.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.noGU, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_NO_GU_CLOSURE

Default model behavior for G-U base pairs closing a loop.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.noGUclosure, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_CIRC

Default model behavior to treat a molecule as a circular RNA (DNA).

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.circ, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_GQUAD

Default model behavior regarding the treatment of G-Quadruplexes.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.gquad, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_UNIQ_ML

Default behavior of the model regarding unique multi-branch loop decomposition.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.uniq_ML, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_ENERGY_SET

Default model behavior on which energy set to use.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.energy_set, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_BACKTRACK

Default model behavior with regards to backtracking of structures.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.backtrack, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_BACKTRACK_TYPE

Default model behavior on what type of backtracking to perform.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.backtrack_type, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_COMPUTE_BPP

Default model behavior with regards to computing base pair probabilities.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.compute_bpp, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_MAX_BP_SPAN

Default model behavior for the allowed maximum base pair span.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.max_bp_span, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_WINDOW_SIZE

Default model behavior for the sliding window approach.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.window_size, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_LOG_ML

Default model behavior on how to evaluate the energy contribution of multi-branch loops.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.logML, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_ALI_OLD_EN

Default model behavior for consensus structure energy evaluation.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.oldAliEn, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_ALI_RIBO

Default model behavior for consensus structure co-variance contribution assessment.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.ribo, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_ALI_CV_FACT

Default model behavior for weighting the co-variance score in consensus structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.cv_fact, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_ALI_NC_FACT

Default model behavior for weighting the nucleotide conservation? in consensus structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_t.nc_fact, *vrna_md_defaults_reset()*, *vrna_md_set_default()*

VRNA_MODEL_DEFAULT_PF_SMOOTH

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_SALT

Default model salt concentration (M).

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_SALT_MLLOWER

Default model lower bound of multiloop size for salt correction fiting.

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_SALT_MLUPPER

Default model upper bound of multiloop size for salt correction fiting.

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_SALT_DPXINIT

Default model value to turn off user-provided salt correction for duplex initializtion.

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_SALT_DPXINIT_FACT_RNA

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_SALT_DPXINIT_FACT_DNA

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_SALT_DPXINIT_FACT

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_HELICAL_RISE_RNA

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_HELICAL_RISE_DNA

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_HELICAL_RISE

Default helical rise.

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_BACKBONE_LENGTH_RNA

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_BACKBONE_LENGTH_DNA

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_BACKBONE_LENGTH

Default backbone length.

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_CIRC_PENALTY

```
#include <ViennaRNA/model.h>
```

VRNA_MODEL_DEFAULT_CIRC_ALPHA0

```
#include <ViennaRNA/model.h>
```

MAXALPHA

Maximal length of alphabet.

```
#include <ViennaRNA/model.h>
```

model_detailsT

```
#include <ViennaRNA/model.h>
```

Typedefs

typedef struct *vrna_md_s* **vrna_md_t**

Typename for the model details data structure *vrna_md_s*.

```
#include <ViennaRNA/model.h>
```

Functions

void **vrna_md_set_default**(*vrna_md_t* *md)

Apply default model details to a provided *vrna_md_t* data structure.

```
#include <ViennaRNA/model.h>
```

Use this function to initialize a *vrna_md_t* data structure with its default values

Parameters

- **md** – A pointer to the data structure that is about to be initialized

void **vrna_md_update**(*vrna_md_t* *md)

Update the model details data structure.

```
#include <ViennaRNA/model.h>
```

This function should be called after changing the *vrna_md_t.energy_set* attribute since it re-initializes base pairing related arrays within the *vrna_md_t* data structure. In particular, *vrna_md_t.pair*, *vrna_md_t.alias*, and *vrna_md_t.rtype* are set to the values that correspond to the specified *vrna_md_t.energy_set* option

See also

vrna_md_t, *vrna_md_t.energy_set*, *vrna_md_t.pair*, *vrna_md_t.rtype*, *vrna_md_t.alias*, *vrna_md_set_default()*

vrna_md_t ***vrna_md_copy**(*vrna_md_t* *md_to, const *vrna_md_t* *md_from)

Copy/Clone a *vrna_md_t* model.

```
#include <ViennaRNA/model.h>
```

Use this function to clone a given model either inplace (target container `md_to` given) or create a copy by cloning the source model and returning it (`md_to` == NULL).

Parameters

- **md_to** – The model to be overwritten (if non-NULL and `md_to` != `md_from`)
- **md_from** – The model to copy (if non-NULL)

Returns

A pointer to the copy model (or NULL if `md_from` == NULL)

char ***vrna_md_option_string**(*vrna_md_t* *md)

Get a corresponding commandline parameter string of the options in a *vrna_md_t*.

```
#include <ViennaRNA/model.h>
```

Note

This function is not threadsafe!

void **vrna_md_set_nonstandards**(*vrna_md_t* *md, const char *ns_bases)

```
#include <ViennaRNA/model.h>
```

void **vrna_md_defaults_reset**(*vrna_md_t* *md_p)

Reset the global default model details to a specific set of parameters, or their initial values.

```
#include <ViennaRNA/model.h>
```

This function resets the global default model details to their initial values, i.e. as specified by the ViennaRNA Package release, upon passing NULL as argument. Alternatively it resets them according to a set of provided parameters.

See also

vrna_md_set_default(), *vrna_md_t*

Note

The global default parameters affect all function calls of RNAlib where model details are not explicitly provided. Hence, any change of them is not considered threadsafe

Warning

This function first resets the global default settings to factory defaults, and only then applies user provided settings (if any). User settings that do not meet specifications are skipped.

Parameters

- **md_p** – A set of model details to use as global default (if NULL is passed, factory defaults are restored)

void **vrna_md_defaults_temperature**(double T)

Set default temperature for energy evaluation of loops.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_TEMPERATURE*

Parameters

- **T** – Temperature in centigrade

double **vrna_md_defaults_temperature_get**(void)

Get default temperature for energy evaluation of loops.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_temperature(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_TEMPERATURE*

Returns
The global default settings for temperature in centigrade

void **vrna_md_defaults_betaScale**(double b)

Set default scaling factor of thermodynamic temperature in Boltzmann factors.

```
#include <ViennaRNA/model.h>
```

Bolzmann factors are then computed as $exp(-E/(b \cdot kT))$.

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_BETA_SCALE*

Parameters

- **b** – The scaling factor, default is 1.0

double **vrna_md_defaults_betaScale_get**(void)

Get default scaling factor of thermodynamic temperature in Boltzmann factors.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_betaScale(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_BETA_SCALE*

Returns
The global default thermodynamic temperature scaling factor

void **vrna_md_defaults_pf_smooth**(int s)

```
#include <ViennaRNA/model.h>
```

int **vrna_md_defaults_pf_smooth_get**(void)

```
#include <ViennaRNA/model.h>
```

void **vrna_md_defaults_dangles**(int d)

Set default dangle model for structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_DANGLES*

Parameters

- **d** – The dangle model

int **vrna_md_defaults_dangles_get**(void)

Get default dangle model for structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_dangles(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_DANGLES*

Returns

The global default settings for the dangle model

void **vrna_md_defaults_special_hp**(int flag)

Set default behavior for lookup of tabulated free energies for special hairpin loops, such as Tri-, Tetra-, or Hexa-loops.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_SPECIAL_HP*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_special_hp_get**(void)

Get default behavior for lookup of tabulated free energies for special hairpin loops, such as Tri-, Tetra-, or Hexa-loops.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_special_hp(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_SPECIAL_HP*

Returns
The global default settings for the treatment of special hairpin loops

void **vrna_md_defaults_noLP**(int flag)

Set default behavior for prediction of canonical secondary structures.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_NO_LP*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_noLP_get**(void)

Get default behavior for prediction of canonical secondary structures.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_noLP(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_NO_LP*

Returns
The global default settings for predicting canonical secondary structures

void **vrna_md_defaults_noGU**(int flag)

Set default behavior for treatment of G-U wobble pairs.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_NO_GU*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_noGU_get**(void)

Get default behavior for treatment of G-U wobble pairs.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_noGU(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_NO_GU*

Returns
The global default settings for treatment of G-U wobble pairs

void **vrna_md_defaults_noGUclosure**(int flag)

Set default behavior for G-U pairs as closing pair for loops.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_NO_GU_CLOSURE*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_noGUclosure_get**(void)

Get default behavior for G-U pairs as closing pair for loops.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_noGUclosure(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_NO_GU_CLOSURE*

Returns
The global default settings for treatment of G-U pairs closing a loop

void **vrna_md_defaults_logML**(int flag)

Set default behavior recomputing free energies of multi-branch loops using a logarithmic model.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_LOG_ML*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_logML_get**(void)

Get default behavior recomputing free energies of multi-branch loops using a logarithmic model.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_logML(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_LOG_ML*

Returns

The global default settings for logarithmic model in multi-branch loop free energy evaluation

void **vrna_md_defaults_circ**(int flag)

Set default behavior whether input sequences are circularized.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_CIRC*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_circ_get**(void)

Get default behavior whether input sequences are circularized.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_circ(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_CIRC*

Returns

The global default settings for treating input sequences as circular

void **vrna_md_defaults_circ_penalty**(int flag)

Set default behavior for fully unpaired circular RNA penalty.

```
#include <ViennaRNA/model.h>
```

See also

vrna_E_exterior_loop(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, #VRNA_MODEL_DEFAULT_CIRC_PENALTY

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_circ_penalty_get**(void)

Get default behavior for fully unpaired circular RNA penalty.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_circ_penalty(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_CIRC*

Returns

The global default settings for adding an entropic penalty for unpaired circular RNA ring

void **vrna_md_defaults_gquad**(int flag)

Set default behavior for treatment of G-Quadruplexes.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_GQUAD*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_gquad_get**(void)

Get default behavior for treatment of G-Quadruplexes.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_gquad(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_GQUAD*

Returns

The global default settings for treatment of G-Quadruplexes

void **vrna_md_defaults_uniq_ML**(int flag)

Set default behavior for creating additional matrix for unique multi-branch loop prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_UNIQ_ML*

Note

Activating this option usually results in higher memory consumption!

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_uniq_ML_get**(void)

Get default behavior for creating additional matrix for unique multi-branch loop prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_uniq_ML(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_UNIQ_ML*

Returns

The global default settings for creating additional matrices for unique multi-branch loop prediction

void **vrna_md_defaults_energy_set**(int e)

Set default energy set.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ENERGY_SET*

Parameters

- **e** – Energy set (0, 1, 2, 3)

int **vrna_md_defaults_energy_set_get**(void)

Get default energy set.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_energy_set(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ENERGY_SET*

Returns

The global default settings for the energy set

void **vrna_md_defaults_backtrack**(int flag)

Set default behavior for whether to backtrack secondary structures.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_BACKTRACK*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_backtrack_get**(void)

Get default behavior for whether to backtrack secondary structures.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_backtrack(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_BACKTRACK*

Returns

The global default settings for backtracking structures

void **vrna_md_defaults_backtrack_type**(char t)

Set default backtrack type, i.e. which DP matrix is used.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_BACKTRACK_TYPE*

Parameters

- **t** – The type ('F', 'C', or 'M')

char **vrna_md_defaults_backtrack_type_get**(void)

Get default backtrack type, i.e. which DP matrix is used.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_backtrack_type(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_BACKTRACK_TYPE*

Returns

The global default settings that specify which DP matrix is used for backtracking

void **vrna_md_defaults_compute_bpp**(int flag)

Set the default behavior for whether to compute base pair probabilities after partition function computation.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_COMPUTE_BPP*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_compute_bpp_get**(void)

Get the default behavior for whether to compute base pair probabilities after partition function computation.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_compute_bpp(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_COMPUTE_BPP*

Returns
The global default settings that specify whether base pair probabilities are computed together with partition function

void **vrna_md_defaults_max_bp_span**(int span)

Set default maximal base pair span.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_MAX_BP_SPAN*

Parameters

- **span** – Maximal base pair span

int **vrna_md_defaults_max_bp_span_get**(void)

Get default maximal base pair span.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_max_bp_span(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_MAX_BP_SPAN*

Returns
The global default settings for maximum base pair span

void **vrna_md_defaults_min_loop_size**(int size)

Set default minimal loop size.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, TURN

Parameters

- **size** – Minimal size, i.e. number of unpaired nucleotides for a hairpin loop

int **vrna_md_defaults_min_loop_size_get**(void)

Get default minimal loop size.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_min_loop_size(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, TURN

Returns
The global default settings for minimal size of hairpin loops

void **vrna_md_defaults_window_size**(int size)

Set default window size for sliding window structure prediction approaches.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_WINDOW_SIZE*

Parameters

- **size** – The size of the sliding window

int **vrna_md_defaults_window_size_get**(void)

Get default window size for sliding window structure prediction approaches.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_window_size(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_WINDOW_SIZE*

Returns
The global default settings for the size of the sliding window

void **vrna_md_defaults_oldAliEn**(int flag)

Set default behavior for whether to use old energy model for comparative structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ALI_OLD_EN*

Note

This option is outdated. Activating the old energy model usually results in worse consensus structure predictions.

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_oldAliEn_get**(void)

Get default behavior for whether to use old energy model for comparative structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_oldAliEn(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ALI_OLD_EN*

Returns

The global default settings for using old energy model for comparative structure prediction

void **vrna_md_defaults_ribo**(int flag)

Set default behavior for whether to use Ribosum Scoring in comparative structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ALI_RIBO*

Parameters

- **flag** – On/Off switch (0 = OFF, else = ON)

int **vrna_md_defaults_ribo_get**(void)

Get default behavior for whether to use Ribosum Scoring in comparative structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_ribo(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ALI_RIBO*

Returns

The global default settings for using Ribosum scoring in comparative structure prediction

void **vrna_md_defaults_cv_fact**(double factor)

Set the default co-variance scaling factor used in comparative structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ALI_CV_FACT*

Parameters

- **factor** – The co-variance factor

double **vrna_md_defaults_cv_fact_get**(void)

Get the default co-variance scaling factor used in comparative structure prediction.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_cv_fact(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ALI_CV_FACT*

Returns

The global default settings for the co-variance factor

void **vrna_md_defaults_nc_fact**(double factor)

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ALI_NC_FACT*

Parameters

- **factor**

double **vrna_md_defaults_nc_fact_get**(void)

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_nc_fact(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*, *VRNA_MODEL_DEFAULT_ALI_NC_FACT*

Returns

void **vrna_md_defaults_sfact**(double factor)

Set the default scaling factor used to avoid under-/overflows in partition function computation.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_reset(), *vrna_md_set_default()*, *vrna_md_t*

Parameters

- **factor** – The scaling factor (default: 1.07)

double **vrna_md_defaults_sfact_get**(void)

Get the default scaling factor used to avoid under-/overflows in partition function computation.

```
#include <ViennaRNA/model.h>
```

See also

vrna_md_defaults_sfact(), *vrna_md_defaults_reset()*, *vrna_md_set_default()*, *vrna_md_t*

Returns

The global default settings of the scaling factor

void **vrna_md_defaults_salt**(double salt)

Set the default salt concentration.

```
#include <ViennaRNA/model.h>
```

Parameters

- **salt** – The sodium concentration in M (default: 1.021)

double **vrna_md_defaults_salt_get**(void)

Get the default salt concentration.

```
#include <ViennaRNA/model.h>
```

Returns

The default salt concentration

void **vrna_md_defaults_saltMLLower**(int lower)

Set the default multiloop size lower bound for loop salt correciton linear fitting.

```
#include <ViennaRNA/model.h>
```

Parameters

- **lower** – Size lower bound (number of backbone in loop)

int **vrna_md_defaults_saltMLLower_get**(void)

Get the default multiloop size lower bound for loop salt correciton linear fitting.

```
#include <ViennaRNA/model.h>
```

Returns

The default lower bound

void **vrna_md_defaults_saltMLUpper**(int upper)

Set the default multiloop size upper bound for loop salt correciton linear fitting.

```
#include <ViennaRNA/model.h>
```

Parameters

- **upper** – Size Upper bound (number of backbone in loop)

int **vrna_md_defaults_saltMLUpper_get**(void)

Get the default multiloop size upper bound for loop salt correciton linear fitting.

```
#include <ViennaRNA/model.h>
```

Returns

The default upper bound

void **vrna_md_defaults_saltDPXInit**(int value)

Set user-provided salt correciton for duplex initialization If value is 99999 the default value from fitting is used.

```
#include <ViennaRNA/model.h>
```

Parameters

- **value** – The value of salt correction for duplex initialization (in dcal/mol)

int **vrna_md_defaults_saltDPXInit_get**(void)

Get user-provided salt correciton for duplex initialization If value is 99999 the default value from fitting is used.

```
#include <ViennaRNA/model.h>
```

Returns

The user-provided salt correction for duplex initialization

void **vrna_md_defaults_saltDPXInitFact**(float value)

```
#include <ViennaRNA/model.h>
```

float **vrna_md_defaults_saltDPXInitFact_get**(void)

```
#include <ViennaRNA/model.h>
```

void **vrna_md_defaults_helical_rise**(float value)

```
#include <ViennaRNA/model.h>
```

float **vrna_md_defaults_helical_rise_get**(void)

```
#include <ViennaRNA/model.h>
```

void **vrna_md_defaults_backbone_length**(float value)

```
#include <ViennaRNA/model.h>
```

float **vrna_md_defaults_backbone_length_get**(void)

```
#include <ViennaRNA/model.h>
```

void **vrna_md_defaults_circ_alpha0**(double a0)

```
#include <ViennaRNA/model.h>
```

double **vrna_md_defaults_circ_alpha0_get**(void)

```
#include <ViennaRNA/model.h>
```

void **set_model_details**(*vrna_md_t* *md)

Set default model details.

```
#include <ViennaRNA/model.h>
```

Use this function if you wish to initialize a *vrna_md_t* data structure with its default values, i.e. the global model settings as provided by the deprecated global variables.

Deprecated:

This function will vanish as soon as backward compatibility of RNAlib is dropped (expected in version 3). Use *vrna_md_set_default()* instead!

Parameters

- **md** – A pointer to the data structure that is about to be initialized

char ***option_string**(void)

```
#include <ViennaRNA/model.h>
```

Variables

double **temperature**

Rescale energy parameters to a temperature in degC.

```
#include <ViennaRNA/model.h>
```

Default is 37C. You have to call the update_..._params() functions after changing this parameter.

Deprecated:

Use *vrna_md_defaults_temperature()*, and *vrna_md_defaults_temperature_get()* to change, and read the global default temperature settings

See also

vrna_md_defaults_temperature(), *vrna_md_defaults_temperature_get()*, *vrna_md_defaults_reset()*

double **pf_scale**

A scaling factor used by *pf_fold()* to avoid overflows.

```
#include <ViennaRNA/model.h>
```

Should be set to approximately $exp((-F/kT)/length)$, where F is an estimate for the ensemble free energy, for example the minimum free energy. You must call *update_pf_params()* after changing this parameter.

If pf_scale is -1 (the default) , an estimate will be provided automatically when computing partition functions, e.g. *pf_fold()* The automatic estimate is usually insufficient for sequences more than a few hundred bases long.

int **dangles**

Switch the energy model for dangling end contributions (0, 1, 2, 3).

```
#include <ViennaRNA/model.h>
```

If set to 0 no stabilizing energies are assigned to bases adjacent to helices in free ends and multiloops (so called dangling ends). Normally (dangles = 1) dangling end energies are assigned only to unpaired bases and a base cannot participate simultaneously in two dangling ends. In the partition function algorithm *pf_fold()* these checks are neglected. If *dangles* is set to 2, all folding routines will follow this convention. This treatment of dangling ends gives more favorable energies to helices directly adjacent to one another, which can be beneficial since such helices often do engage in stabilizing interactions through co-axial stacking.

If dangles = 3 co-axial stacking is explicitly included for adjacent helices in multiloops. The option affects only mfe folding and energy evaluation (*fold()* and *energy_of_structure()*), as well as suboptimal folding (*subopt()*) via re-evaluation of energies. Co-axial stacking with one intervening mismatch is not considered so far.

Default is 2 in most algorithms, partition function algorithms can only handle 0 and 2

int **tetra_loop**

Include special stabilizing energies for some tri-, tetra- and hexa-loops;.

```
#include <ViennaRNA/model.h>
```

default is 1.

int **noLonelyPairs**

Global switch to avoid/allow helices of length 1.

```
#include <ViennaRNA/model.h>
```

Disallow all pairs which can only occur as lonely pairs (i.e. as helix of length 1). This avoids lonely base pairs in the predicted structures in most cases.

int **noGU**

Global switch to forbid/allow GU base pairs at all.

```
#include <ViennaRNA/model.h>
```

int **no_closingGU**

GU allowed only inside stacks if set to 1.

```
#include <ViennaRNA/model.h>
```

int **circ**

backward compatibility variable.. this does not effect anything

```
#include <ViennaRNA/model.h>
```

int **gquad**

Allow G-quadruplex formation.

```
#include <ViennaRNA/model.h>
```

int **uniq_ML**

do ML decomposition uniquely (for subopt)

```
#include <ViennaRNA/model.h>
```

int **energy_set**

0 = BP; 1=any with GC; 2=any with AU-parameter

```
#include <ViennaRNA/model.h>
```

If set to 1 or 2: fold sequences from an artificial alphabet ABCD..., where A pairs B, C pairs D, etc. using either GC (1) or AU parameters (2); default is 0, you probably don't want to change it.

int **do_backtrack**

do backtracking, i.e. compute secondary structures or base pair probabilities

```
#include <ViennaRNA/model.h>
```

If 0, do not calculate pair probabilities in *pf_fold()*; this is about twice as fast. Default is 1.

char **backtrack_type**

A backtrack array marker for *inverse_fold()*.

```
#include <ViennaRNA/model.h>
```

If set to 'C': force (1,N) to be paired, 'M' fold as if the sequence were inside a multiloop. Otherwise ('F') the usual mfe structure is computed.

char ***nonstandards**

contains allowed non standard base pairs

```
#include <ViennaRNA/model.h>
```

Lists additional base pairs that will be allowed to form in addition to GC, CG, AU, UA, GU and UG. Nonstandard base pairs are given a stacking energy of 0.

int **max_bp_span**

Maximum allowed base pair span.

```
#include <ViennaRNA/model.h>
```

A value of -1 indicates no restriction for distant base pairs.

int **oldAliEn**

use old alifold energies (with gaps)

```
#include <ViennaRNA/model.h>
```

int **ribo**

use ribosum matrices

```
#include <ViennaRNA/model.h>
```

double **cv_fact**

This variable controls the weight of the covariance term in the energy function of alignment folding algorithms.

```
#include <ViennaRNA/model.h>
```

Deprecated:

See *vrna_md_t.cv_fact*, and *vrna_mfe()* to avoid using global variables

Default is 1.

double **nc_fact**

This variable controls the magnitude of the penalty for non-compatible sequences in the covariance term of alignment folding algorithms.

```
#include <ViennaRNA/model.h>
```

Deprecated:

See *vrna_md_t.nc_fact*, and *vrna_mfe()* to avoid using global variables

Default is 1.

int **logML**

if nonzero use logarithmic ML energy in energy_of_struct

```
#include <ViennaRNA/model.h>
```

double **salt**

salt concentration

```
#include <ViennaRNA/model.h>
```

int **saltDPXInit**

Salt correction for duplex initialization.

```
#include <ViennaRNA/model.h>
```

float **helical_rise**

```
#include <ViennaRNA/model.h>
```

float **backbone_length**

```
#include <ViennaRNA/model.h>
```

struct **vrna_md_s**

#include <ViennaRNA/model.h> The data structure that contains the complete model details used throughout the calculations.

For convenience reasons, we provide the type name *vrna_md_t* to address this data structure without the use of the struct keyword

SWIG Wrapper Notes:

This data structure is wrapped as an object `md` with multiple related functions attached as methods.

A new set of default parameters can be obtained by calling the constructure of `md`:

- `md()` – Initialize with default settings

The resulting object has a list of attached methods which directly correspond to functions that mainly operate on the corresponding *C* data structure:

- `reset()` - *vrna_md_set_default()*
- `set_from_globals()` - *set_model_details()*
- `option_string()` - *vrna_md_option_string()*

See also

vrna_md_set_default(), *set_model_details()*, *vrna_md_update()*, *vrna_md_t*

Public Members

double **temperature**

The temperature used to scale the thermodynamic parameters.

```
#include <ViennaRNA/model.h>
```

double **betaScale**

A scaling factor for the thermodynamic temperature of the Boltzmann factors.

```
#include <ViennaRNA/model.h>
```

int **pf_smooth**

A flat specifying whether energies in Boltzmann factors need to be smoothed.

```
#include <ViennaRNA/model.h>
```

int **dangles**

Specifies the dangle model used in any energy evaluation (0,1,2 or 3).

```
#include <ViennaRNA/model.h>
```

If set to 0 no stabilizing energies are assigned to bases adjacent to helices in free ends and multiloops (so called dangling ends). Normally (dangles = 1) dangling end energies are assigned only to unpaired bases and a base cannot participate simultaneously in two dangling ends. In the partition function algorithm *vrna_pf()* these checks are neglected. To provide comparability between free energy minimization and partition function algorithms, the default setting is 2. This treatment of dangling ends gives more favorable energies to helices directly adjacent to one another, which can be beneficial since such helices often do engage in stabilizing interactions through co-axial stacking.

If set to 3 co-axial stacking is explicitly included for adjacent helices in multiloops. The option affects only mfe folding and energy evaluation (*vrna_mfe()* and *vrna_eval_structure()*), as well as suboptimal folding (*vrna_subopt()*) via re-evaluation of energies. Co-axial stacking with one intervening mismatch is not considered so far. Note, that some function do not implement all dangle model but only a subset of (0,1,2,3). In particular, partition function algorithms can only handle 0 and 2. Read the documentation of the particular recurrences or energy evaluation function for information about the provided dangle model.

int **special_hp**

Include special hairpin contributions for tri, tetra and hexaloops.

```
#include <ViennaRNA/model.h>
```

int **noLP**

Only consider canonical structures, i.e. no 'lonely' base pairs.

```
#include <ViennaRNA/model.h>
```

int **noGU**

Do not allow GU pairs.

```
#include <ViennaRNA/model.h>
```

int **noGUclosure**

Do not allow loops to be closed by GU pair.

```
#include <ViennaRNA/model.h>
```

int **logML**

Use logarithmic scaling for multiloops.

```
#include <ViennaRNA/model.h>
```

int **circ**

Assume RNA to be circular instead of linear.

```
#include <ViennaRNA/model.h>
```

int **circ_penalty**

Add an entropic penalty to the unpaired circRNA chain.

```
#include <ViennaRNA/model.h>
```

int **gquad**

Include G-quadruplexes in structure prediction.

```
#include <ViennaRNA/model.h>
```

int **uniq_ML**

Flag to ensure unique multi-branch loop decomposition during folding.

```
#include <ViennaRNA/model.h>
```

int **energy_set**

Specifies the energy set that defines set of compatible base pairs.

```
#include <ViennaRNA/model.h>
```

int **backtrack**

Specifies whether or not secondary structures should be backtraced.

```
#include <ViennaRNA/model.h>
```

char **backtrack_type**

Specifies in which matrix to backtrack.

```
#include <ViennaRNA/model.h>
```

int **compute_bpp**

Specifies whether or not backward recursions for base pair probability (bpp) computation will be performed.

```
#include <ViennaRNA/model.h>
```

char **nonstandards**[64]

contains allowed non standard bases

```
#include <ViennaRNA/model.h>
```

int **max_bp_span**

maximum allowed base pair span

```
#include <ViennaRNA/model.h>
```

int **min_loop_size**

Minimum size of hairpin loops.

```
#include <ViennaRNA/model.h>
```

The default value for this field is TURN, however, it may be 0 in cofolding context.

int **window_size**

Size of the sliding window for locally optimal structure prediction.

```
#include <ViennaRNA/model.h>
```

int **oldAliEn**

Use old alifold energy model.

```
#include <ViennaRNA/model.h>
```

int **ribo**

Use ribosum scoring table in alifold energy model.

```
#include <ViennaRNA/model.h>
```

double **cv_fact**

Co-variance scaling factor for consensus structure prediction.

```
#include <ViennaRNA/model.h>
```

double **nc_fact**

Scaling factor to weight co-variance contributions of non-canonical pairs.

```
#include <ViennaRNA/model.h>
```

double **sfact**

Scaling factor for partition function scaling.

```
#include <ViennaRNA/model.h>
```

int **rtype**[8]

Reverse base pair type array.

```
#include <ViennaRNA/model.h>
```

short **alias**[20 + 1]

alias of an integer nucleotide representation

```
#include <ViennaRNA/model.h>
```

int **pair**[20 + 1][20 + 1]

Integer representation of a base pair.

```
#include <ViennaRNA/model.h>
```

float **pair_dist**[7][7]

Base pair dissimilarity, a.k.a. distance matrix.

```
#include <ViennaRNA/model.h>
```

double **salt**

Salt (monovalent) concentration (M) in buffer.

```
#include <ViennaRNA/model.h>
```

int **saltMLLower**

Lower bound of multiloop size to use in loop salt correction linear fitting.

```
#include <ViennaRNA/model.h>
```

int **saltMLUpper**

Upper bound of multiloop size to use in loop salt correction linear fitting.

```
#include <ViennaRNA/model.h>
```

int **saltDPXInit**

User-provided salt correction for duplex initialization (in dcal/mol). If set to 99999 the default salt correction is used. If set to 0 there is no salt correction for duplex initialization.

```
#include <ViennaRNA/model.h>
```

float **saltDPXInitFact**

```
#include <ViennaRNA/model.h>
```

float **helical_rise**

```
#include <ViennaRNA/model.h>
```

float **backbone_length**

```
#include <ViennaRNA/model.h>
```

double **circ_alpha0**

```
#include <ViennaRNA/model.h>
```

Energy Evaluation for Individual Loops

General

Functions to evaluate the free energy of particular types of loops.

Functions

int **vrna_eval_loop_pt**(*vrna_fold_compound_t* *fc, int i, const short *pt)

Calculate energy of a loop.

```
#include <ViennaRNA/eval/structures.h>
```

SWIG Wrapper Notes:

This function is attached as method `eval_loop_pt()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_loop_pt()* in the *The RNA Python module* .

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **i** – position of covering base pair
- **pt** – the pair table of the secondary structure

Returns

free energy of the loop in 10cal/mol

int **vrna_eval_loop_pt_v**(*vrna_fold_compound_t* *fc, int i, const short *pt, int verbosity_level)

Calculate energy of a loop.

```
#include <ViennaRNA/eval/structures.h>
```

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **i** – position of covering base pair
- **pt** – the pair table of the secondary structure
- **verbosity_level** – The level of verbosity of this function

Returns

free energy of the loop in 10cal/mol

Exterior Loops

Functions to evaluate the free energy contributions for exterior (external) loops.

Boltzmann weight (partition function) interface

typedef struct vrna_mx_pf_aux_el_s ***vrna_mx_pf_aux_el_t**

Auxiliary helper arrays for fast exterior loop computations.

```
#include <ViennaRNA/partfunc/exterior.h>
```

See also

vrna_exp_E_ext_fast_init(), vrna_exp_E_ext_fast_rotate(), vrna_exp_E_ext_fast_free(), vrna_exp_E_ext_fast()

vrna_mx_pf_aux_el_t **vrna_exp_E_ext_fast_init**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/partfunc/exterior.h>
```

void **vrna_exp_E_ext_fast_rotate**(*vrna_mx_pf_aux_el_t* aux_mx)

```
#include <ViennaRNA/partfunc/exterior.h>
```

void **vrna_exp_E_ext_fast_free**(*vrna_mx_pf_aux_el_t* aux_mx)

```
#include <ViennaRNA/partfunc/exterior.h>
```

FLT_OR_DBL **vrna_exp_E_ext_fast**(*vrna_fold_compound_t* *fc, int i, int j, *vrna_mx_pf_aux_el_t* aux_mx)

```
#include <ViennaRNA/partfunc/exterior.h>
```

void **vrna_exp_E_ext_fast_update**(*vrna_fold_compound_t* *fc, int j, *vrna_mx_pf_aux_el_t* aux_mx)

```
#include <ViennaRNA/partfunc/exterior.h>
```

Basic free energy interface

int **vrna_E_exterior_stem**(unsigned int type, int n5d, int n3d, *vrna_param_t* *p)

Evaluate a stem branching off the exterior loop.

```
#include <ViennaRNA/eval/exterior.h>
```

Given a base pair (i, j) encoded by *type*, compute the energy contribution including dangling-end/terminal-mismatch contributions. Instead of returning the energy contribution per-se, this function returns the corresponding Boltzmann factor. If either of the adjacent nucleotides $(i - 1)$ and $(j + 1)$ must not contribute stacking energy, the corresponding encoding must be -1.

See also

vrna_exp_E_exterior_stem()

Note

By default, terminal mismatch energies are applied that correspond to the neighboring nucleotides provided by their encodings `n5d` and `n3d`. Whenever the encodings are negative, the implementation switches to usage of dangling end energies (for the non-negative base). If both encodings are negative, no terminal mismatch contributions are added.

Parameters

- **type** – The base pair encoding
- **n5d** – The encoded nucleotide directly adjacent at the 5' side of the base pair (may be -1)
- **n3d** – The encoded nucleotide directly adjacent at the 3' side of the base pair (may be -1)
- **p** – The pre-computed energy parameters

Returns

The energy contribution of the introduced exterior-loop stem

int **vrna_E_exterior_loop**(unsigned int n, *vrna_md_t* *md)

```
#include <ViennaRNA/eval/exterior.h>
```

int **vrna_eval_exterior_stem**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int options)

Evaluate the free energy of a base pair in the exterior loop.

```
#include <ViennaRNA/eval/exterior.h>
```

Evalue the free energy of a base pair connecting two nucleotides in the exterior loop and take hard constraints into account.

Typically, this is simply dangling end contributions of the adjacent nucleotides, potentially a terminal A-U mismatch penalty, and maybe some generic soft constraint contribution for that decomposition.

See also

vrna_E_exterior_stem(), *VRNA_EVAL_LOOP_NO_HC*, *VRNA_EVAL_LOOP_NO_SC*, *VRNA_EVAL_LOOP_NO_CONSTRAINTS*

Note

For dangles == 1 || 3 this function also evaluates the three additional pairs (i + 1, j), (i, j - 1), and (i + 1, j - 1) and returns the minimum for all four possibilities in total.

Note

By default, all user-supplied hard- and soft constraints will be taken into account! Use the *VRNA_EVAL_LOOP_NO_HC* and *VRNA_EVAL_LOOP_NO_SC* bit flags as input for `options` to change the default behavior if necessary.

Parameters

- **fc** – Fold compound to work on (defines the model and parameters)
- **i** – 5' position of the base pair
- **j** – 3' position of the base pair
- **options** – A bit-field that specifies which aspects (not) to consider during evaluation

Returns

Free energy for the terminal base pair of a stem branching off the exterior loop in deka-kal/mol or INF if the pair is forbidden

Boltzmann weight (partition function) interface

FLT_OR_DBL **vrna_exp_E_exterior_stem**(unsigned int type, int n5d, int n3d, *vrna_exp_param_t* *p)

Evaluate a stem branching off the exterior loop (Boltzmann factor version).

```
#include <ViennaRNA/eval/exterior.h>
```

Given a base pair (i, j) encoded by *type*, compute the energy contribution including dangling-end/terminal-mismatch contributions. Instead of returning the energy contribution per-se, this function returns the corresponding Boltzmann factor.

See also

vrna_E_exterior()

Note

By default, terminal mismatch energies are applied that correspond to the neighboring nucleotides provided by their encodings `n5d` and `n3d`. Whenever the encodings are negative, the implementation switches to usage of dangling end energies (for the non-negative base). If both encodings are negative, no terminal mismatch contributions are added.

Parameters

- **type** – The base pair encoding
- **n5d** – The encoded nucleotide directly adjacent at the 5' side of the base pair (may be -1)
- **n3d** – The encoded nucleotide directly adjacent at the 3' side of the base pair (may be -1)
- **p** – The pre-computed energy parameters (Boltzmann factor version)

Returns

The Boltzmann weighted energy contribution of the introduced exterior-loop stem

FLT_OR_DBL **vrna_exp_E_exterior_loop**(unsigned int n, *vrna_md_t* *md)

```
#include <ViennaRNA/eval/exterior.h>
```

Minimum Free Energy API

int **vrna_mfe_exterior_f5**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/mfe/exterior.h>
```

int **vrna_mfe_exterior_f3**(*vrna_fold_compound_t* *fc, unsigned int i)

```
#include <ViennaRNA/mfe/exterior.h>
```

Functions

int **vrna_E_ext_loop_5**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/mfe/exterior.h>
```

int **vrna_E_ext_loop_3**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/mfe/exterior.h>
```

Hairpin Loops

Functions to evaluate the free energy contributions for hairpin loops.

Basic free energy interface

int **vrna_E_hairpin**(unsigned int size, unsigned int type, int si1, int sj1, const char *sequence, *vrna_param_t* *P)

Retrieve the energy of a hairpin-loop.

```
#include <ViennaRNA/eval/hairpin.h>
```

To evaluate the free energy of a hairpin-loop, several parameters have to be known. A general hairpin-loop has this structure:

a3 a4

a2 a5

a1 a6

X - Y

| |

5' 3' where X-Y marks the closing pair [e.g. a **(G,C)** pair]. The length of this loop is 6 as there are six unpaired nucleotides (a1-a6) enclosed by (X,Y). The 5' mismatching nucleotide is a1 while the 3' mismatch is a6. The nucleotide sequence of this loop is "a1.a2.a3.a4.a5.a6"

See also

vrna_param_t, *vrna_eval_hp_loop()*

Note

Whenever one of the mismatch base encodings `si1` or `sj1` is negative, terminal mismatch energies are not applied!

Note

The parameter `sequence` is a 0-terminated string of size `size` + 2 that contain the nucleic acid sequence of the loop in upper-case letters. This parameter is only required for loops of `size` below 7, since it is used for look-up of unusually stable tri-, tetra- and hexa-loops, such as GNRA tetra loops. Those may have additional sequence-dependent tabulated free energies available.

Warning

This function **only** evaluates the free energy of a hairpin loop according to the current Turner energy parameter set! No additional hard- or soft constraints are applied. See *vrna_eval_hp_loop()* for a function that also takes into account any user-supplied constraints!

Parameters

- **size** – The size of the loop (number of unpaired nucleotides)
- **type** – The pair type of the base pair closing the hairpin
- **si1** – The 5'-mismatching nucleotide
- **sj1** – The 3'-mismatching nucleotide
- **sequence** – The sequence of the loop (May be `NULL`, otherwise mst be at least $size+2$ long)
- **P** – The datastructure containing scaled energy parameters

Returns

The Free energy of the Hairpin-loop in dcal/mol

int **vrna_eval_hairpin**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int options)

Evaluate free energy of a hairpin loop.

```
#include <ViennaRNA/eval/hairpin.h>
```

This function evaluates the free energy of a hairpin loop closed by a base pair (i,j). By default (`options` = *VRNA_EVAL_LOOP_DEFAULT*), *all* user-supplied constraints will be taken into consideration. This means that any hard constraints that prohibit the formation of this loop will result in an energy contribution of **INF**. On the other hand, if, given the set of constraints, the loop is allowed then its free energy is evaluated according to the Nearest Neighbor energy parameter set. On top of that, any user-supplied soft-constraints will be added, if applicable.

The `options` argument allows for (de-)activating certain aspects of the evaluation, e.g. hard constraints, soft constraints, etc.

See also

vrna_E_hairpin(), *vrna_exp_eval_hairpin()*, *VRNA_EVAL_LOOP_NO_HC*, *VRNA_EVAL_LOOP_NO_SC*, *VRNA_EVAL_LOOP_NO_CONSTRAINTS*

Note

If sequence position `i` is larger than `j`, the function assumes a hairpin loop formed by a circular RNA, where the unpaired loop sequence spans the n,1-junction.

Note

By default, all user-supplied hard- and soft constraints will be taken into account! Use the *VRNA_EVAL_LOOP_NO_HC* and *VRNA_EVAL_LOOP_NO_SC* bit flags as input for `options` to change the default behavior if necessary.

Note

This function is polymorphic! The provided *vrna_fold_compound_t* may be of type *VRNA_FC_TYPE_SINGLE* or *VRNA_FC_TYPE_COMPARATIVE*

Parameters

- **`fc`** – The *vrna_fold_compound_t* for the particular energy evaluation
- **`i`** – 5'-position of the base pair
- **`j`** – 3'-position of the base pair
- **`options`** – A bit-field that specifies which aspects (not) to consider during evaluation

Returns

Free energy of the hairpin loop closed by (i, j) in deka-kal/mol or INF if the loop is forbidden

Boltzmann weight (partition function) interface

FLT_OR_DBL **`vrna_exp_E_hairpin`**(unsigned int size, unsigned int type, int si1, int sj1, const char *sequence, *vrna_exp_param_t* *P)

Compute Boltzmann weight $e^{-\Delta G/kT}$ of a hairpin loop.

```
#include <ViennaRNA/eval/hairpin.h>
```

This is the partition function variant of vrna_E_hp() that returns the Boltzmann weight $e^{-\Delta E/kT}$ instead of the energy E.

See also

vrna_exp_eval_hp_loop(), *vrna_exp_param_t*, vrna_E_hp()

Note

Whenever one of the mismatch base encodings `si1` or `sj1` is negative, terminal mismatch energies are not applied!

Note

Do not forget to scale this Bolztmann factor properly, e.g. by multiplying with scale[u+2]

Parameters

- **size** – The size of the loop (number of unpaired nucleotides)
- **type** – The pair type of the base pair closing the hairpin
- **si1** – The 5'-mismatching nucleotide
- **sj1** – The 3'-mismatching nucleotide
- **sequence** – The sequence of the loop (May be NULL, otherwise mst be at least $size+2$ long)
- **P** – The datastructure containing scaled Boltzmann weights of the energy parameters

Returns

The Boltzmann weight of the Hairpin-loop

FLT_OR_DBL **vrna_exp_eval_hairpin**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int options)

High-Level function for hairpin loop energy evaluation (partition function variant).

```
#include <ViennaRNA/eval/hairpin.h>
```

This is the partition function variant of *vrna_eval_hp_loop()* that returns the Boltzmann weight $e^{-\Delta E/kT}$ instead of the energy E. On top of all constraints application, this function already scales the Boltzmann factor, i.e. it multiplies the result with scale[u + 2]

The options argument allows for (de-)activating certain aspects of the evaluation, e.g. hard constraints, soft constraints, etc.

See also

vrna_eval_hairpin(), *vrna_exp_E_hairpin()*, *VRNA_EVAL_LOOP_NO_HC*, *VRNA_EVAL_LOOP_NO_SC*, *VRNA_EVAL_LOOP_NO_CONSTRAINTS*

Note

If sequence position i is larger than j, the function assumes a hairpin loop formed by a circular RNA, where the unpaired loop sequence spans the n,1-junction.

Note

By default, all user-supplied hard- and soft constraints will be taken into account! Use the *VRNA_EVAL_LOOP_NO_HC* and *VRNA_EVAL_LOOP_NO_SC* bit flags to change the default behavior if necessary.

Note

This function is polymorphic! The provided *vrna_fold_compound_t* may be of type *VRNA_FC_TYPE_SINGLE* or *VRNA_FC_TYPE_COMPARATIVE*

Parameters

- **fc** – The *vrna_fold_compound_t* for the particular energy evaluation
- **i** – 5'-position of the base pair
- **j** – 3'-position of the base pair
- **options** – A bit-field that specifies which aspects (not) to consider during evaluation

Returns

Boltzmann factor of the free energy of the hairpin loop closed by (i, j) or 0. if the loop is forbidden

Internal Loops

Functions to evaluate the free energy contributions for internal loops.

Basic free energy interface

int **vrna_E_internal**(unsigned int n1, unsigned int n2, unsigned int type, unsigned int type_2, int si1, int sj1, int sp1, int sq1, *vrna_param_t* *P)

Compute the Energy of an internal loop.

```
#include <ViennaRNA/eval/internal.h>
```

This function computes the free energy E of an internal-loop with the following structure:

3' 5'

| |

U - V

a_n b_1

. .

. .

. .

a_1 b_m

X - Y

| |

5' 3' This general structure depicts an internal-loop that is closed by the base pair (X,Y). The enclosed base pair is (V,U) which leaves the unpaired bases a_1-a_n and b_1-b_n that constitute the loop. In this example, the length of the internal-loop is $(n + m)$ where n or m may be 0 resulting in a bulge-loop or base pair stack. The mismatching nucleotides for the closing pair (X,Y) are:

5'-mismatch: a_1

3'-mismatch: b_m

and for the enclosed base pair (V,U):

5'-mismatch: b_1

3'-mismatch: a_n

See also

vrna_exp_E_internal()

Note

Base pairs are always denoted in 5'->3' direction. Thus the enclosed base pair must be 'turned arround' when evaluating the free energy of the internal-loop

This function is threadsafe

Parameters

- **n1** – The size of the 'left'-loop (number of unpaired nucleotides)
- **n2** – The size of the 'right'-loop (number of unpaired nucleotides)
- **type** – The pair type of the base pair closing the internal loop
- **type_2** – The pair type of the enclosed base pair
- **si1** – The 5'-mismatching nucleotide of the closing pair
- **sj1** – The 3'-mismatching nucleotide of the closing pair
- **sp1** – The 3'-mismatching nucleotide of the enclosed pair
- **sq1** – The 5'-mismatching nucleotide of the enclosed pair
- **P** – The datastructure containing scaled energy parameters

Returns

The Free energy of the internal loop in dcal/mol

int **vrna_en_internal**(unsigned int n1, unsigned int n2, unsigned int type, unsigned int type_2, unsigned int si1, unsigned int sj1, unsigned int sp1, unsigned int sq1, *vrna_param_t* *P)

Compute the Energy of an internal loop.

```
#include <ViennaRNA/eval/internal.h>
```

This function computes the free energy E of an internal-loop with the following structure:

3' 5'

| |

U - V

a_n b_1

. .

. .

. .

a_1 b_m

X - Y

| |

5’ 3’ This general structure depicts an internal-loop that is closed by the base pair (X,Y). The enclosed base pair is (V,U) which leaves the unpaired bases a_1-a_n and b_1-b_n that constitute the loop. In this example, the length of the internal-loop is $(n + m)$ where n or m may be 0 resulting in a bulge-loop or base pair stack. The mismatching nucleotides for the closing pair (X,Y) are:

5’-mismatch: a_1

3’-mismatch: b_m

and for the enclosed base pair (V,U):

5’-mismatch: b_1

3’-mismatch: a_n

See also

vrna_exp_E_internal()

Note

Base pairs are always denoted in 5’->3’ direction. Thus the enclosed base pair must be ‘turned arround’ when evaluating the free energy of the internal-loop

This function is threadsafe

Parameters

- **n1** – The size of the ‘left’-loop (number of unpaired nucleotides)
- **n2** – The size of the ‘right’-loop (number of unpaired nucleotides)
- **type** – The pair type of the base pair closing the internal loop
- **type_2** – The pair type of the enclosed base pair
- **si1** – The 5’-mismatching nucleotide of the closing pair
- **sj1** – The 3’-mismatching nucleotide of the closing pair
- **sp1** – The 3’-mismatching nucleotide of the enclosed pair
- **sq1** – The 5’-mismatching nucleotide of the enclosed pair
- **P** – The datastructure containing scaled energy parameters

Returns

The Free energy of the internal loop in dcal/mol

int **vrna_en_bulge**(unsigned int n, unsigned int type, unsigned int type_2, *vrna_param_t* *P)

```
#include <ViennaRNA/eval/internal.h>
```

int **vrna_en_stack**(unsigned int type, unsigned int type_2, *vrna_param_t* *P)

```
#include <ViennaRNA/eval/internal.h>
```

int **vrna_eval_internal**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int k, unsigned int l, unsigned int options)

Evaluate the free energy contribution of an internal loop with delimiting base pairs (i, j) and (k, l).

```
#include <ViennaRNA/eval/internal.h>
```

Note

This function is polymorphic, i.e. it accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE* as well as *VRNA_FC_TYPE_COMPARATIVE*

int **vrna_eval_stack**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int options)

```
#include <ViennaRNA/eval/internal.h>
```

Boltzmann weight (partition function) interface

FLT_OR_DBL **vrna_exp_E_internal**(unsigned int n1, unsigned int n2, unsigned int type, unsigned int type_2, int si1, int sj1, int sp1, int sq1, *vrna_exp_param_t* *P)

```
#include <ViennaRNA/eval/internal.h>
```

FLT_OR_DBL **vrna_exp_eval_internal**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int k, unsigned int l, unsigned int options)

```
#include <ViennaRNA/eval/internal.h>
```

FLT_OR_DBL **vrna_exp_eval_stack**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int options)

```
#include <ViennaRNA/eval/internal.h>
```

Minimum Free Energy API

int **vrna_mfe_internal**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j)

```
#include <ViennaRNA/mfe/internal.h>
```

int **vrna_mfe_internal_ext**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int *ip, unsigned int *iq)

```
#include <ViennaRNA/mfe/internal.h>
```

Boltzmann weight (partition function) interface

FLT_OR_DBL **vrna_exp_E_int_loop**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/partfunc/internal.h>
```

Functions

int **vrna_E_int_loop**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/mfe/internal.h>
```

int **vrna_E_ext_int_loop**(*vrna_fold_compound_t* *fc, int i, int j, int *ip, int *iq)

```
#include <ViennaRNA/mfe/internal.h>
```

Multibranch Loops

Functions to evaluate the free energy contributions for mutlibranch loops.

Minimum Free Energy API

typedef struct vrna_mx_mfe_aux_ml_s ***vrna_mx_mfe_aux_ml_t**

```
#include <ViennaRNA/mfe/multibranch.h>
```

vrna_mx_mfe_aux_ml_t **vrna_mfe_multibranch_fast_init**(unsigned int length)

```
#include <ViennaRNA/mfe/multibranch.h>
```

void **vrna_mfe_multibranch_fast_rotate**(*vrna_mx_mfe_aux_ml_t* aux)

```
#include <ViennaRNA/mfe/multibranch.h>
```

void **vrna_mfe_multibranch_fast_free**(*vrna_mx_mfe_aux_ml_t* aux)

```
#include <ViennaRNA/mfe/multibranch.h>
```

int **vrna_mfe_multibranch_loop_fast**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, *vrna_mx_mfe_aux_ml_t* helpers)

```
#include <ViennaRNA/mfe/multibranch.h>
```

int **vrna_mfe_multibranch_stems_fast**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, struct vrna_mx_mfe_aux_ml_s *helpers)

```
#include <ViennaRNA/mfe/multibranch.h>
```

int **vrna_mfe_multibranch_m2_fast**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, struct vrna_mx_mfe_aux_ml_s *helpers)

```
#include <ViennaRNA/mfe/multibranch.h>
```

int **vrna_mfe_multibranch_loop_stack**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j)

Evaluate energy of multi branch loop helices stacking onto closing pair (i,j).

```
#include <ViennaRNA/mfe/multibranch.h>
```

Computes total free energy for coaxial stacking of (i.j) with (i+1.k) or (k+1.j-1)

int **vrna_mfe_multibranch_m1**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j)

```
#include <ViennaRNA/mfe/multibranch.h>
```

Boltzmann weight (partition function) interface

typedef struct vrna_mx_pf_aux_ml_s ***vrna_mx_pf_aux_ml_t**

Auxiliary helper arrays for fast exterior loop computations.

```
#include <ViennaRNA/partfunc/multibranch.h>
```

See also

vrna_exp_E_ml_fast_init(), vrna_exp_E_ml_fast_rotate(), vrna_exp_E_ml_fast_free(), vrna_exp_E_ml_fast()

FLT_OR_DBL **vrna_exp_E_mb_loop_fast**(*vrna_fold_compound_t* *fc, int i, int j, *vrna_mx_pf_aux_ml_t* aux_mx)

```
#include <ViennaRNA/partfunc/multibranch.h>
```

vrna_mx_pf_aux_ml_t **vrna_exp_E_ml_fast_init**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/partfunc/multibranch.h>
```

void **vrna_exp_E_ml_fast_rotate**(*vrna_mx_pf_aux_ml_t* aux_mx)

```
#include <ViennaRNA/partfunc/multibranch.h>
```

void **vrna_exp_E_ml_fast_free**(*vrna_mx_pf_aux_ml_t* aux_mx)

```
#include <ViennaRNA/partfunc/multibranch.h>
```

const *FLT_OR_DBL* ***vrna_exp_E_ml_fast_qqm**(*vrna_mx_pf_aux_ml_t* aux_mx)

```
#include <ViennaRNA/partfunc/multibranch.h>
```

const *FLT_OR_DBL* ***vrna_exp_E_ml_fast_qqm1**(*vrna_mx_pf_aux_ml_t* aux_mx)

```
#include <ViennaRNA/partfunc/multibranch.h>
```

FLT_OR_DBL **vrna_exp_E_ml_fast**(*vrna_fold_compound_t* *fc, int i, int j, *vrna_mx_pf_aux_ml_t* aux_mx)

```
#include <ViennaRNA/partfunc/multibranch.h>
```

FLT_OR_DBL **vrna_exp_E_m2_fast**(*vrna_fold_compound_t* *fc, int i, int j, struct vrna_mx_pf_aux_ml_s *aux_mx)

```
#include <ViennaRNA/partfunc/multibranch.h>
```

Basic free energy interface

int **vrna_E_multibranch_stem**(unsigned int type, int si1, int sj1, *vrna_param_t* *P)

Evaluate the free energy contribution of a stem branching off a multibranch loop.

```
#include <ViennaRNA/eval/multibranch.h>
```

This function yields the free energy contribution for the terminal base pairs of a stem branching off a multibranch loop. In essence, this consists of (i) a terminal mismatch or dangling end contribution, (ii) the score for a stem according to the affine multibranch loop model, and (iii) a terminal AU/GU penalty, if applicable.

See also

vrna_exp_E_multibranch_stem(), *vrna_E_exterior_stem()*

Note

By default, terminal mismatch energies are applied that correspond to the neighboring nucleotides provided by their encodings `n5d` and `n3d`. Whenever the encodings are negative, the implementation switches to usage of dangling end energies (for the non-negative base). If both encodings are negative, no terminal mismatch contributions are added.

Parameters

- **type** – The base pair encoding
- **si1** – The encoded nucleotide directly adjacent at the 5' side of the base pair (may be -1)
- **sj1** – The encoded nucleotide directly adjacent at the 3' side of the base pair (may be -1)
- **P** – The pre-computed energy parameters

Returns

The energy contribution of the introduced mutlibranch loop stem

Boltzmann weight (partition function) interface

FLT_OR_DBL **vrna_exp_E_multibranch_stem**(unsigned int type, int si1, int sj1, *vrna_exp_param_t* *P)

Evaluate the free energy contribution of a stem branching off a multibranch loop (Boltzmann factor version).

```
#include <ViennaRNA/eval/multibranch.h>
```

This function yields the free energy contribution as Boltzmann factor $exp(-E/kT)$ for the terminal base pairs of a stem branching off a multibranch loop. In essence, this consists of (i) a terminal mismatch or dangling end contribution, (ii) the score for a stem according to the affine multibranch loop model, and (iii) a terminal AU/GU penalty, if applicable.

See also

vrna_E_multibranch_stem(), *vrna_exp_E_exterior_stem()*

Note

By default, terminal mismatch energies are applied that correspond to the neighboring nucleotides provided by their encodings `n5d` and `n3d`. Whenever the encodings are negative, the implementa-

tion switches to usage of dangling end energies (for the non-negative base). If both encodings are negative, no terminal mismatch contributions are added.

Parameters

- **type** – The base pair encoding
- **si1** – The encoded nucleotide directly adjacent at the 5' side of the base pair (may be -1)
- **sj1** – The encoded nucleotide directly adjacent at the 3' side of the base pair (may be -1)
- **P** – The pre-computed energy parameters (Boltzmann factor version)

Returns

The Boltzmann factor of the energy contribution for the introduced mutlibranch loop stem

Energy Evaluation for Atomic Moves

Functions to evaluate the free energy change of a structure after application of (a set of) atomic moves

Functions

float **vrna_eval_move**(*vrna_fold_compound_t* *fc, const char *structure, int m1, int m2)

Calculate energy of a move (closing or opening of a base pair).

```
#include <ViennaRNA/eval/structures.h>
```

If the parameters m1 and m2 are negative, it is deletion (opening) of a base pair, otherwise it is insertion (opening).

SWIG Wrapper Notes:

This function is attached as method `eval_move()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_move()* in the *The RNA Python module* .

See also

vrna_eval_move_pt()

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **structure** – secondary structure in dot-bracket notation
- **m1** – first coordinate of base pair
- **m2** – second coordinate of base pair

Returns

energy change of the move in kcal/mol (INF / 100. upon any error)

int **vrna_eval_move_pt**(*vrna_fold_compound_t* *fc, short *pt, int m1, int m2)

Calculate energy of a move (closing or opening of a base pair).

```
#include <ViennaRNA/eval/structures.h>
```

If the parameters m1 and m2 are negative, it is deletion (opening) of a base pair, otherwise it is insertion (opening).

SWIG Wrapper Notes:

This function is attached as method `eval_move_pt()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.eval_move_pt()`* in the *The RNA Python module* .

See also

vrna_eval_move()

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **pt** – the pair table of the secondary structure
- **m1** – first coordinate of base pair
- **m2** – second coordinate of base pair

Returns

energy change of the move in 10cal/mol

int **vrna_eval_move_pt_simple**(const char *string, short *pt, int m1, int m2)

```
#include <ViennaRNA/eval/structures.h>
```

int **vrna_eval_move_shift_pt**(*vrna_fold_compound_t* *fc, *vrna_move_t* *m, short *structure)

```
#include <ViennaRNA/eval/structures.h>
```

Evaluation of Structures

Functions to evaluate the free energy of secondary structures under a particular set of parameters.

Basic Energy Evaluation Interface with Dot-Bracket Structure String

float **vrna_eval_structure**(*vrna_fold_compound_t* *fc, const char *structure)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for energy evaluation of a given pair of structure and sequence (alignment). Model details, energy parameters, and possibly soft constraints are used as provided via the parameter 'fc'. The *vrna_fold_compound_t* does not need to contain any DP matrices, but requires all most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_EVAL_ONLY);
```

SWIG Wrapper Notes:

This function is attached as method `eval_structure()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.eval_structure()`* in the *The RNA Python module* .

See also

vrna_eval_structure_pt(), *vrna_eval_structure_verbose()*, *vrna_eval_structure_pt_verbose()*, *vrna_fold_compound()*, *vrna_fold_compound_comparative()*, *vrna_eval_covar_structure()*

Note

Accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE* and *VRNA_FC_TYPE_COMPARATIVE*

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **structure** – Secondary structure in dot-bracket notation

Returns

The free energy of the input structure given the input sequence in kcal/mol

float **vrna_eval_covar_structure**(*vrna_fold_compound_t* *fc, const char *structure)

Calculate the pseudo energy derived by the covariance scores of a set of aligned sequences.

```
#include <ViennaRNA/eval/structures.h>
```

Consensus structure prediction is driven by covariance scores of base pairs in rows of the provided alignment. This function allows one to retrieve the total amount of this covariance pseudo energy scores. The *vrna_fold_compound_t* does not need to contain any DP matrices, but requires all most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound_comparative(alignment, NULL, VRNA_OPTION_EVAL_ONLY);
```

SWIG Wrapper Notes:

This function is attached as method `eval_covar_structure()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.eval_covar_structure()`* in the *The RNA Python module* .

See also

vrna_fold_compound_comparative(), *vrna_eval_structure()*

Note

Accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_COMPARATIVE* only!

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **structure** – Secondary (consensus) structure in dot-bracket notation

Returns
The covariance pseudo energy score of the input structure given the input sequence alignment in kcal/mol

float **vrna_eval_structure_verbose**(*vrna_fold_compound_t* *fc, const char *structure, FILE *file)

Calculate the free energy of an already folded RNA and print contributions on a per-loop base.

```
#include <ViennaRNA/eval/structures.h>
```

This function is a simplyfied version of *vrna_eval_structure_v()* that uses the *default* verbosity level.

SWIG Wrapper Notes:

This function is attached as method `eval_structure_verbose()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_structure_verbose()* in the *The RNA Python module* .

See also

vrna_eval_structure_pt(), *vrna_eval_structure_verbose()*, *vrna_eval_structure_pt_verbose()*,

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **structure** – Secondary structure in dot-bracket notation
- **file** – A file handle where this function should print to (may be NULL).

Returns
The free energy of the input structure given the input sequence in kcal/mol

float **vrna_eval_structure_v**(*vrna_fold_compound_t* *fc, const char *structure, int verbosity_level, FILE *file)

Calculate the free energy of an already folded RNA and print contributions on a per-loop base.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for detailed energy evaluation of a given sequence/structure pair. In contrast to *vrna_eval_structure()* this function prints detailed energy contributions based on individual loops to a file handle. If NULL is passed as file handle, this function defaults to print to stdout. Any positive `verbosity_level` activates potential warning message of the energy evaluting functions, while values ≥ 1 allow for detailed control of what data is printed. A negative parameter `verbosity_level` turns off printing all together.

Model details, energy parameters, and possibly soft constraints are used as provided via the parameter 'fc'. The fold_compound does not need to contain any DP matrices, but all the most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_EVAL_ONLY);
```

See also

vrna_eval_structure_pt(), *vrna_eval_structure_verbose()*, *vrna_eval_structure_pt_verbose()*,

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **structure** – Secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns
The free energy of the input structure given the input sequence in kcal/mol

float **vrna_eval_structure_cstr**(*vrna_fold_compound_t* *fc, const char *structure, int verbosity_level, *vrna_cstr_t* output_stream)

```
#include <ViennaRNA/eval/structures.h>
```

Basic Energy Evaluation Interface with Structure Pair Table

int **vrna_eval_structure_pt**(*vrna_fold_compound_t* *fc, const short *pt)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for energy evaluation of a given sequence/structure pair where the structure is provided in pair_table format as obtained from *vrna_ptable()*. Model details, energy parameters, and possibly soft constraints are used as provided via the parameter 'fc'. The fold_compound does not need to contain any DP matrices, but all the most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_EVAL_ONLY);
```

SWIG Wrapper Notes:

This function is attached as method `eval_structure_pt()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_structure_pt()* in the *The RNA Python module* .

See also

vrna_ptable(), *vrna_eval_structure()*, *vrna_eval_structure_pt_verbose()*

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **pt** – Secondary structure as pair_table

Returns
The free energy of the input structure given the input sequence in 10cal/mol

int **vrna_eval_structure_pt_verbose**(*vrna_fold_compound_t* *fc, const short *pt, FILE *file)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

This function is a simplyfied version of *vrna_eval_structure_simple_v()* that uses the *default* verbosity level.

SWIG Wrapper Notes:

This function is attached as method `eval_structure_pt_verbose()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.eval_structure_pt_verbose()`* in the *The RNA Python module* .

See also

vrna_eval_structure_pt_v(), *vrna_ptable()*, *vrna_eval_structure_pt()*, *vrna_eval_structure_verbose()*

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **pt** – Secondary structure as pair_table
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in 10cal/mol

int **vrna_eval_structure_pt_v**(*vrna_fold_compound_t* *fc, const short *pt, int verbosity_level, FILE *file)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for energy evaluation of a given sequence/structure pair where the structure is provided in pair_table format as obtained from *vrna_ptable()*. Model details, energy parameters, and possibly soft constraints are used as provided via the parameter 'fc'. The fold_compound does not need to contain any DP matrices, but all the most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_EVAL_ONLY);
```

In contrast to *vrna_eval_structure_pt()* this function prints detailed energy contributions based on individual loops to a file handle. If NULL is passed as file handle, this function defaults to print to stdout. Any positive `verbosity_level` activates potential warning message of the energy evaluting functions, while values ≥ 1 allow for detailed control of what data is printed. A negative parameter `verbosity_level` turns off printing all together.

See also

vrna_ptable(), *vrna_eval_structure_pt()*, *vrna_eval_structure_verbose()*

Parameters

- **fc** – A *vrna_fold_compound_t* containing the energy parameters and model details
- **pt** – Secondary structure as pair_table
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in 10cal/mol

Simplified Energy Evaluation with Sequence and Dot-Bracket Strings

float **vrna_eval_structure_simple**(const char *string, const char *structure)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for energy evaluation of a given sequence/structure pair. In contrast to *vrna_eval_structure()* this function assumes default model details and default energy parameters in order to evaluate the free energy of the secondary structure. Therefore, it serves as a simple interface function for energy evaluation for situations where no changes on the energy model are required.

SWIG Wrapper Notes:

In the target scripting language, this function serves as a wrapper for *vrna_eval_structure_simple_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and NULL, respectively.. See, e.g. *RNA.eval_structure_simple()* in the *The RNA Python module* .

See also

vrna_eval_structure(), *vrna_eval_structure_pt()*, *vrna_eval_structure_verbose()*, *vrna_eval_structure_pt_verbose()*,

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation

Returns

The free energy of the input structure given the input sequence in kcal/mol

float **vrna_eval_circ_structure**(const char *string, const char *structure)

Evaluate the free energy of a sequence/structure pair where the sequence is circular.

```
#include <ViennaRNA/eval/structures.h>
```

SWIG Wrapper Notes:

In the target scripting language, this function serves as a wrapper for *vrna_eval_circ_structure_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and NULL, respectively.. See, e.g. *RNA.eval_circ_structure()* in the *The RNA Python module* .

See also

vrna_eval_structure_simple(), *vrna_eval_gquad_structure()*, *vrna_eval_circ_consensus_structure()*, *vrna_eval_circ_structure_v()*, *vrna_eval_structure()*

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation

Returns

The free energy of the structure given the circular input sequence in kcal/mol

float **vrna_eval_gquad_structure**(const char *string, const char *structure)

Evaluate the free energy of a sequence/structure pair where the structure may contain G-Quadruplexes.

```
#include <ViennaRNA/eval/structures.h>
```

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

```
GGAAGGAAAGGAGG
++..++...++.++
```

SWIG Wrapper Notes:

In the target scripting language, this function serves as a wrapper for *vrna_eval_gquad_structure_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and NULL, respectively.. See, e.g. *RNA.eval_gquad_structure()* in the *The RNA Python module* .

See also

vrna_eval_structure_simple(), *vrna_eval_circ_structure()*, *vrna_eval_gquad_consensus_structure()*, *vrna_eval_gquad_structure_v()*, *vrna_eval_structure()*

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation

Returns

The free energy of the structure including contributions of G-quadruplexes in kcal/mol

float **vrna_eval_circ_gquad_structure**(const char *string, const char *structure)

Evaluate the free energy of a sequence/structure pair where the sequence is circular and the structure may contain G-Quadruplexes.

```
#include <ViennaRNA/eval/structures.h>
```

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

```
GGAAGGAAAGGAGG
++..++...++.++
```

SWIG Wrapper Notes:

In the target scripting language, this function serves as a wrapper for *vrna_eval_circ_gquad_structure_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and NULL, respectively.. See, e.g. *RNA.eval_circ_gquad_structure()* in the *The RNA Python module* .

See also

vrna_eval_structure_simple(), *vrna_eval_circ_gquad_consensus_structure()*, *vrna_eval_circ_gquad_structure_v()*, *vrna_eval_structure()*

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation

Returns
The free energy of the structure including contributions of G-quadruplexes in kcal/mol

float **vrna_eval_structure_simple_verbose**(const char *string, const char *structure, FILE *file)

Calculate the free energy of an already folded RNA and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function is a simplyfied version of *vrna_eval_structure_simple_v()* that uses the *default* verbosity level.

See also

vrna_eval_structure_simple_v(), *vrna_eval_structure_verbose()*, *vrna_eval_structure_pt()*, *vrna_eval_structure_verbose()*, *vrna_eval_structure_pt_verbose()*

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation
- **file** – A file handle where this function should print to (may be NULL).

Returns
The free energy of the input structure given the input sequence in kcal/mol

float **vrna_eval_structure_simple_v**(const char *string, const char *structure, int verbosity_level, FILE *file)

Calculate the free energy of an already folded RNA and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for detailed energy evaluation of a given sequence/structure pair. In contrast to *vrna_eval_structure()* this function prints detailed energy contributions based on individual loops to a file handle. If NULL is passed as file handle, this function defaults to print to stdout. Any positive `verbosity_level` activates potential warning message of the energy evaluting functions, while values ≥ 1 allow for detailed control of what data is printed. A negative parameter `verbosity_level` turns off printing all together.

In contrast to *vrna_eval_structure_verbose()* this function assumes default model details and default energy parameters in order to evaluate the free energy of the secondary structure. Threefore, it serves as a simple interface function for energy evaluation for situations where no changes on the energy model are required.

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_structure_simple()*. The last two arguments for this function are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_structure_simple()* in the *The RNA Python module* .

See also

vrna_eval_structure_verbose(), *vrna_eval_structure_pt()*, *vrna_eval_structure_pt_verbose()*,

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in kcal/mol

float **vrna_eval_circ_structure_v**(const char *string, const char *structure, int verbosity_level, FILE *file)

Evaluate free energy of a sequence/structure pair, assume sequence to be circular and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function is the same as *vrna_eval_structure_simple_v()* but assumes the input sequence to be circularized.

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_circ_structure()*. The last two arguments for this function are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_circ_structure()* in the *The RNA Python module* .

See also

vrna_eval_structure_simple_v(), *vrna_eval_circ_structure()*, *vrna_eval_structure_verbose()*

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in kcal/mol

float **vrna_eval_gquad_structure_v**(const char *string, const char *structure, int verbosity_level, FILE *file)

Evaluate free energy of a sequence/structure pair, allow for G-Quadruplexes in the structure and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function is the same as *vrna_eval_structure_simple_v()* but allows for annotated G-Quadruplexes in the dot-bracket structure input.

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

```
GGAAGGAAAGGAGG
++..++...++.++
```

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_gquad_structure()*. The last two arguments for this function are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_gquad_structure()* in the *The RNA Python module* .

See also

vrna_eval_structure_simple_v(), *vrna_eval_gquad_structure()*, *vrna_eval_structure_verbose()*

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in kcal/mol

float **vrna_eval_circ_gquad_structure_v**(const char *string, const char *structure, int verbosity_level, FILE *file)

Evaluate free energy of a sequence/structure pair, assume sequence to be circular, allow for G-Quadruplexes in the structure, and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function is the same as *vrna_eval_structure_simple_v()* but assumes the input sequence to be circular and allows for annotated G-Quadruplexes in the dot-bracket structure input.

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

```
GGAAGGAAAGGAGG
++..++...++.++
```

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_circ_gquad_structure()*. The last two arguments for this function are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_circ_gquad_structure()* in the *The RNA Python module* .

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns
The free energy of the input structure given the input sequence in kcal/mol

Simplified Energy Evaluation with Sequence Alignments and Consensus Structure Dot-Bracket String

float **vrna_eval_consensus_structure_simple**(const char **alignment, const char *structure)

Calculate the free energy of an already folded RNA sequence alignment.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for energy evaluation for a given multiple sequence alignment and consensus structure pair. In contrast to *vrna_eval_structure()* this function assumes default model details and default energy parameters in order to evaluate the free energy of the secondary structure. Therefore, it serves as a simple interface function for energy evaluation for situations where no changes on the energy model are required.

SWIG Wrapper Notes:

This function is available through an overloadeded version of *vrna_eval_structure_simple()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. *RNA.eval_structure_simple()* in the *The RNA Python module* .

See also

vrna_eval_covar_structure(), *vrna_eval_structure()*, *vrna_eval_structure_pt()*, *vrna_eval_structure_verbose()*, *vrna_eval_structure_pt_verbose()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters and hyphen ('-') to denote gaps
- **structure** – Consensus Secondary structure in dot-bracket notation

Returns
The free energy of the consensus structure given the input alignment in kcal/mol

float **vrna_eval_circ_consensus_structure**(const char **alignment, const char *structure)

Evaluate the free energy of a multiple sequence alignment/consensus structure pair where the sequences are circular.

```
#include <ViennaRNA/eval/structures.h>
```

SWIG Wrapper Notes:

This function is available through an overloadeded version of *vrna_eval_circ_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. `RNA.eval_circ_structure()` in the *The RNA Python module* .

See also

vrna_eval_covar_structure(), *vrna_eval_consensus_structure_simple()*, *vrna_eval_gquad_consensus_structure()*, *vrna_eval_circ_structure()*, *vrna_eval_circ_consensus_structure_v()*, *vrna_eval_structure()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters
- **structure** – Consensus secondary structure in dot-bracket notation

Returns

The free energy of the consensus structure given the circular input sequence in kcal/mol

float **vrna_eval_gquad_consensus_structure**(const char **alignment, const char *structure)

Evaluate the free energy of a multiple sequence alignment/consensus structure pair where the structure may contain G-Quadruplexes.

```
#include <ViennaRNA/eval/structures.h>
```

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

```
GGAAGGAAAGGAGG
++..++...++.++
```

SWIG Wrapper Notes:

This function is available through an overloadeded version of *vrna_eval_gquad_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. `RNA.eval_gquad_structure()` in the *The RNA Python module* .

See also

vrna_eval_covar_structure(), *vrna_eval_consensus_structure_simple()*, *vrna_eval_circ_consensus_structure()*, *vrna_eval_gquad_structure()*, *vrna_eval_gquad_consensus_structure_v()*, *vrna_eval_structure()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters
- **structure** – Consensus secondary structure in dot-bracket notation

Returns

The free energy of the consensus structure including contributions of G-quadruplexes in kcal/mol

float **vrna_eval_circ_gquad_consensus_structure**(const char **alignment, const char *structure)

Evaluate the free energy of a multiple sequence alignment/consensus structure pair where the sequence is circular and the structure may contain G-Quadruplexes.

```
#include <ViennaRNA/eval/structures.h>
```

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

```
GGAAGGAAAGGAGG
++..++...++.++
```

SWIG Wrapper Notes:

This function is available through an overloadeded version of *vrna_eval_circ_gquad_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. *RNA.eval_circ_gquad_structure()* in the *The RNA Python module* .

See also

vrna_eval_covar_structure(), *vrna_eval_consensus_structure_simple()*, *vrna_eval_circ_consensus_structure()*, *vrna_eval_gquad_structure()*, *vrna_eval_circ_gquad_consensus_structure_v()*, *vrna_eval_structure()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters
- **structure** – Consensus secondary structure in dot-bracket notation

Returns

The free energy of the consensus structure including contributions of G-quadruplexes in kcal/mol

float **vrna_eval_consensus_structure_simple_verbose**(const char **alignment, const char *structure, FILE *file)

Evaluate the free energy of a consensus structure for an RNA sequence alignment and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function is a simplyfied version of *vrna_eval_consensus_structure_simple_v()* that uses the *default* verbosity level.

See also

vrna_eval_consensus_structure_simple_v(), *vrna_eval_structure_verbose()*, *vrna_eval_structure_pt()*, *vrna_eval_structure_pt_verbose()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters. Gaps are denoted by hyphens ('-')
- **structure** – Consensus secondary structure in dot-bracket notation
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the conensus structure given the aligned input sequences in kcal/mol

float **vrna_eval_consensus_structure_simple_v**(const char **alignment, const char *structure, int verbosity_level, FILE *file)

Evaluate the free energy of a consensus structure for an RNA sequence alignment and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for detailed energy evaluation of a given sequence alignment/consensus structure pair. In contrast to *vrna_eval_consensus_structure_simple()* this function prints detailed energy contributions based on individual loops to a file handle. If NULL is passed as file handle, this function defaults to print to stdout. Any positive `verbosity_level` activates potential warning message of the energy evaluting functions, while values ≥ 1 allow for detailed control of what data is printed. A negative parameter `verbosity_level` turns off printing all together.

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_structure_simple()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. *`RNA.eval_structure_simple()`* in the *The RNA Python module* .

See also

vrna_eval_consensus_structure(), *vrna_eval_structure()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters. Gaps are denoted by hyphens ('-')
- **structure** – Consensus secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the consensus structure given the sequence alignment in kcal/mol

float **vrna_eval_circ_consensus_structure_v**(const char **alignment, const char *structure, int verbosity_level, FILE *file)

Evaluate the free energy of a consensus structure for an alignment of circular RNA sequences and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function is identical with *vrna_eval_consensus_structure_simple_v()* but assumed the aligned sequences to be circular.

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_circ_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_circ_structure()* in the *The RNA Python module* .

See also

vrna_eval_consensus_structure_simple_v(), *vrna_eval_circ_consensus_structure()*, *vrna_eval_structure()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters. Gaps are denoted by hyphens ('-')

- **structure** – Consensus secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns
The free energy of the consensus structure given the sequence alignment in kcal/mol

float **vrna_eval_gquad_consensus_structure_v**(const char **alignment, const char *structure, int verbosity_level, FILE *file)

Evaluate the free energy of a consensus structure for an RNA sequence alignment, allow for annotated G-Quadruplexes in the structure and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function is identical with *vrna_eval_consensus_structure_simple_v()* but allows for annotated G-Quadruplexes in the consensus structure.

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

```
GGAAGGAAAGGAGG
++..++...++.++
```

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_gquad_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. *`RNA.eval_gquad_structure()`* in the *The RNA Python module* .

See also

vrna_eval_consensus_structure_simple_v(), *vrna_eval_gquad_consensus_structure()*, *vrna_eval_structure()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters. Gaps are denoted by hyphens ('-')
- **structure** – Consensus secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns
The free energy of the consensus structure given the sequence alignment in kcal/mol

float **vrna_eval_circ_gquad_consensus_structure_v**(const char **alignment, const char *structure, int verbosity_level, FILE *file)

Evaluate the free energy of a consensus structure for an alignment of circular RNA sequences, allow for annotated G-Quadruplexes in the structure and print contributions per loop.

```
#include <ViennaRNA/eval/structures.h>
```

This function is identical with *vrna_eval_consensus_structure_simple_v()* but assumes the sequences in the alignment to be circular and allows for annotated G-Quadruplexes in the consensus structure.

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

```
GGAAGGAAAGGAGG
++..++...++.++
```

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_circ_gquad_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_circ_gquad_structure()* in the *The RNA Python module* .

See also

vrna_eval_consensus_structure_simple_v(), *vrna_eval_circ_gquad_consensus_structure()*, *vrna_eval_structure()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters. Gaps are denoted by hyphens ('-')
- **structure** – Consensus secondary structure in dot-bracket notation
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the consensus structure given the sequence alignment in kcal/mol

Simplified Energy Evaluation with Sequence String and Structure Pair Table

int **vrna_eval_structure_pt_simple**(const char *string, const short *pt)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

In contrast to *vrna_eval_structure_pt()* this function assumes default model details and default energy parameters in order to evaluate the free energy of the secondary structure. Threefore, it serves as a

simple interface function for energy evaluation for situations where no changes on the energy model are required.

SWIG Wrapper Notes:

In the target scripting language, this function serves as a wrapper for *vrna_eval_structure_pt_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_structure_pt_simple()* in the *The RNA Python module* .

See also

vrna_ptable(), *vrna_eval_structure_simple()*, *vrna_eval_structure_pt()*

Parameters

- **string** – RNA sequence in uppercase letters
- **pt** – Secondary structure as pair_table

Returns

The free energy of the input structure given the input sequence in 10cal/mol

int **vrna_eval_structure_pt_simple_verbose**(const char *string, const short *pt, FILE *file)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

This function is a simplyfied version of *vrna_eval_structure_pt_simple_v()* that uses the *default* verbosity level.

See also

vrna_eval_structure_pt_simple_v(), *vrna_ptable()*, *vrna_eval_structure_pt_verbose()*, *vrna_eval_structure_simple()*

Parameters

- **string** – RNA sequence in uppercase letters
- **pt** – Secondary structure as pair_table
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in 10cal/mol

int **vrna_eval_structure_pt_simple_v**(const char *string, const short *pt, int verbosity_level, FILE *file)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

This function allows for energy evaluation of a given sequence/structure pair where the structure is provided in pair_table format as obtained from *vrna_ptable()*. Model details, energy parameters, and possibly soft constraints are used as provided via the parameter 'fc'. The fold_compound does not need to contain any DP matrices, but all the most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_EVAL_ONLY);
```

In contrast to *vrna_eval_structure_pt_verbose()* this function assumes default model details and default energy parameters in order to evaluate the free energy of the secondary structure. Threefore, it serves as a simple interface function for energy evaluation for situations where no changes on the energy model are required.

See also

vrna_ptable(), *vrna_eval_structure_pt_v()*, *vrna_eval_structure_simple()*

Parameters

- **string** – RNA sequence in uppercase letters
- **pt** – Secondary structure as pair_table
- **verbosity_level** – The level of verbosity of this function
- **file** – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in 10cal/mol

Simplified Energy Evaluation with Sequence Alignment and Consensus Structure Pair Table

int **vrna_eval_consensus_structure_pt_simple**(const char **alignment, const short *pt)

Evaluate the Free Energy of a Consensus Secondary Structure given a Sequence Alignment.

```
#include <ViennaRNA/eval/structures.h>
```

SWIG Wrapper Notes:

This function is available through an overloadeded version of *vrna_eval_structure_pt_simple()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. *RNA.eval_structure_pt_simple()* in the *The RNA Python module* .

See also

vrna_eval_consensus_structure_simple(), *vrna_eval_structure_pt()*, *vrna_eval_structure()*, *vrna_eval_covar_structure()*

Note

The free energy returned from this function already includes the covariation pseudo energies that is used fir comparative structure prediction within this library.

Parameters

- **alignment** – RNA sequence alignment in uppercase letters. Gaps are denoted by hyphens ('-')
- **pt** – Secondary structure in pair table format

Returns
Free energy of the consensus structure in 10cal/mol

int **vrna_eval_consensus_structure_pt_simple_verbose**(const char **alignment, const short *pt, FILE *file)

```
#include <ViennaRNA/eval/structures.h>
```

int **vrna_eval_consensus_structure_pt_simple_v**(const char **alignment, const short *pt, int verbosity_level, FILE *file)

```
#include <ViennaRNA/eval/structures.h>
```

SWIG Wrapper Notes:

This function is available through an overloaded version of *vrna_eval_structure_pt_simple()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. *RNA.eval_structure_pt_simple()* in the *The RNA Python module* .

Defines

VRNA_VERBOSITY_QUIET

Quiet level verbosity setting.

```
#include <ViennaRNA/eval/basic.h>
```

VRNA_VERBOSITY_DEFAULT

Default level verbosity setting.

```
#include <ViennaRNA/eval/basic.h>
```

VRNA_EVAL_LOOP_DEFAULT

Default options for loop evaluation.

```
#include <ViennaRNA/eval/basic.h>
```

VRNA_EVAL_LOOP_NO_HC

Do not acknowledge hard constraints in loop evaluation.

```
#include <ViennaRNA/eval/basic.h>
```

VRNA_EVAL_LOOP_NO_SC

Do not acknowledge soft constraints in loop evaluation.

```
#include <ViennaRNA/eval/basic.h>
```

VRNA_EVAL_LOOP_NO_CONSTRAINTS

Do not acknowledge any constraints in loop evaluation.

```
#include <ViennaRNA/eval/basic.h>
```

Deprecated Interface for Free Energy Evaluation

Warning

Using the functions below is discouraged as they have been marked deprecated and will be removed from the library in the (near) future!

Defines

ON_SAME_STRAND(I, J, C)

```
#include <ViennaRNA/eval/internal.h>
```

Functions

float **energy_of_alistruct**(const char **sequences, const char *structure, int n_seq, float *energy)

Calculate the free energy of a consensus structure given a set of aligned sequences.

```
#include <ViennaRNA/alifold.h>
```

Deprecated:

Usage of this function is discouraged! Use *vrna_eval_structure()*, and *vrna_eval_covar_structure()* instead!

Parameters

- **sequences** – The NULL terminated array of sequences
- **structure** – The consensus structure
- **n_seq** – The number of sequences in the alignment
- **energy** – A pointer to an array of at least two floats that will hold the free energies (energy[0] will contain the free energy, energy[1] will be filled with the covariance energy term)

Returns

free energy in kcal/mol

int **vrna_eval_ext_stem**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/eval/exterior.h>
```

int **E_Stem**(int type, int si1, int sj1, int extLoop, *vrna_param_t* *P)

Compute the energy contribution of a stem branching off a loop-region.

```
#include <ViennaRNA/eval/exterior.h>
```

This function computes the energy contribution of a stem that branches off a loop region. This can be the case in multiloops, when a stem branching off increases the degree of the loop but also *immediately interior base pairs* of an exterior loop contribute free energy. To switch the behavior of the function according to the evaluation of a multiloop- or exterior-loop-stem, you pass the flag 'extLoop'. The

returned energy contribution consists of a TerminalAU penalty if the pair type is greater than 2, dangling end contributions of mismatching nucleotides adjacent to the stem if only one of the si1, sj1 parameters is greater than 0 and mismatch energies if both mismatching nucleotides are positive values. Thus, to avoid incorporating dangling end or mismatch energies just pass a negative number, e.g. -1 to the mismatch argument.

This is an illustration of how the energy contribution is assembled: 3' 5'

| |

X - Y

5'-si1 sj1-3'

Here, (X,Y) is the base pair that closes the stem that branches off a loop region. The nucleotides si1 and sj1 are the 5'- and 3'- mismatches, respectively. If the base pair type of (X,Y) is greater than 2 (i.e. an A-U or G-U pair, the TerminalAU penalty will be included in the energy contribution returned. If si1 and sj1 are both nonnegative numbers, mismatch energies will also be included. If one of si1 or sj1 is a negative value, only 5' or 3' dangling end contributions are taken into account. To prohibit any of these mismatch contributions to be incorporated, just pass a negative number to both, si1 and sj1. In case the argument extLoop is 0, the returned energy contribution also includes the *internal-loop-penalty* of a multiloop stem with closing pair type.

Deprecated:

Please use one of the functions *vrna_E_exterior_stem()* and *vrna_E_multibranch_stem()* instead! Use the former for cases where `extLoop` != 0 and the latter otherwise.

See also

vrna_E_multibranch_stem(), _ExtLoop()

Note

This function is threadsafe

Parameters

- **type** – The pair type of the first base pair un the stem
- **si1** – The 5'-mismatching nucleotide
- **sj1** – The 3'-mismatching nucleotide
- **extLoop** – A flag that indicates whether the contribution reflects the one of an exterior loop or not
- **P** – The data structure containing scaled energy parameters

Returns

The Free energy of the branch off the loop in dcal/mol

int **E_ExtLoop**(int type, int si1, int sj1, *vrna_param_t* *P)

```
#include <ViennaRNA/eval/exterior.h>
```

int **vrna_E_ext_stem**(unsigned int type, int n5d, int n3d, *vrna_param_t* *p)

```
#include <ViennaRNA/eval/exterior.h>
```

FLT_OR_DBL **vrna_exp_E_ext_stem**(unsigned int type, int n5d, int n3d, *vrna_exp_param_t* *p)

```
#include <ViennaRNA/eval/exterior.h>
```

FLT_OR_DBL **exp_E_ExtLoop**(int type, int si1, int sj1, *vrna_exp_param_t* *P)

```
#include <ViennaRNA/eval/exterior.h>
```

This is the partition function variant of E_ExtLoop()

Deprecated:

Use vrna_exp_E_ext_stem() instead!

See also

E_ExtLoop()

Returns

The Boltzmann weighted energy contribution of the introduced exterior-loop stem

FLT_OR_DBL **exp_E_Stem**(int type, int si1, int sj1, int extLoop, *vrna_exp_param_t* *P)

```
#include <ViennaRNA/eval/exterior.h>
```

Compute the Boltzmann weighted energy contribution of a stem branching off a loop-region

This is the partition function variant of *E_Stem()*

See also

E_Stem()

Note

This function is threadsafe

Returns

The Boltzmann weighted energy contribution of the branch off the loop

static int **E_Hairpin**(int size, int type, int si1, int sj1, const char *string, *vrna_param_t* *P)

Compute the Energy of a hairpin-loop.

```
#include <ViennaRNA/eval/hairpin.h>
```

To evaluate the free energy of a hairpin-loop, several parameters have to be known. A general hairpin-loop has this structure:

a3 a4

a2 a5

a1 a6

X - Y

| |

5' 3' where X-Y marks the closing pair [e.g. a **(G,C)** pair]. The length of this loop is 6 as there are six unpaired nucleotides (a1-a6) enclosed by (X,Y). The 5' mismatching nucleotide is a1 while the 3' mismatch is a6. The nucleotide sequence of this loop is "a1.a2.a3.a4.a5.a6"

See also

scale_parameters(), *vrna_param_t*

Note

The parameter sequence should contain the sequence of the loop in capital letters of the nucleic acid alphabet if the loop size is below 7. This is useful for unusually stable tri-, tetra- and hexa-loops which are treated differently (based on experimental data) if they are tabulated.

Warning

Not (really) thread safe! A threadsafe implementation will replace this function in a future release!

Energy evaluation may change due to updates in global variable "tetra_loop"

Parameters

- **size** – The size of the loop (number of unpaired nucleotides)
- **type** – The pair type of the base pair closing the hairpin
- **si1** – The 5'-mismatching nucleotide
- **sj1** – The 3'-mismatching nucleotide
- **string** – The sequence of the loop (May be NULL, otherwise mst be at least $size + 2$ long)
- **P** – The datastructure containing scaled energy parameters

Returns

The Free energy of the Hairpin-loop in dcal/mol

int **vrna_E_hp_loop**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/eval/hairpin.h>
```

int **vrna_E_ext_hp_loop**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/eval/hairpin.h>
```

int **vrna_eval_hp_loop**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/eval/hairpin.h>
```

SWIG Wrapper Notes:

This function is attached as method `eval_hp_loop()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.eval_hp_loop()`* in the *The RNA Python module* .

int **vrna_eval_ext_hp_loop**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/eval/hairpin.h>
```

static *FLT_OR_DBL* **exp_E_Hairpin**(int u, int type, short si1, short sj1, const char *string, *vrna_exp_param_t* *P)

Compute Boltzmann weight $e^{-\Delta G/kT}$ of a hairpin loop.

```
#include <ViennaRNA/eval/hairpin.h>
```

See also

get_scaled_pf_parameters(), *vrna_exp_param_t*, E_Hairpin()

Note

multiply by scale[u+2]

Warning

Not (really) thread safe! A threadsafe implementation will replace this function in a future release!

Energy evaluation may change due to updates in global variable "tetra_loop"

Parameters

- **u** – The size of the loop (number of unpaired nucleotides)
- **type** – The pair type of the base pair closing the hairpin
- **si1** – The 5'-mismatching nucleotide
- **sj1** – The 3'-mismatching nucleotide
- **string** – The sequence of the loop (May be `NULL`, otherwise mst be at least $size + 2$ long)
- **P** – The datastructure containing scaled Boltzmann weights of the energy parameters

Returns

The Boltzmann weight of the Hairpin-loop

FLT_OR_DBL **vrna_exp_E_hp_loop**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/eval/hairpin.h>
```

static int **E_IntLoop**(int n1, int n2, int type, int type_2, int si1, int sj1, int sp1, int sq1, *vrna_param_t* *P)

```
#include <ViennaRNA/eval/internal.h>
```

Compute the Energy of an internal-loop

This function computes the free energy ΔG of an internal-loop with the following structure:

3’ 5’

| |

U - V

a_n b_1

. .

. .

. .

a_1 b_m

X - Y

| |

5’ 3’ This general structure depicts an internal-loop that is closed by the base pair (X,Y). The enclosed base pair is (V,U) which leaves the unpaired bases a_1-a_n and b_1-b_n that constitute the loop. In this example, the length of the internal-loop is $(n + m)$ where n or m may be 0 resulting in a bulge-loop or base pair stack. The mismatching nucleotides for the closing pair (X,Y) are:

5’-mismatch: a_1

3’-mismatch: b_m

and for the enclosed base pair (V,U):

5’-mismatch: b_1

3’-mismatch: a_n

See also

scale_parameters(), *vrna_param_t*

Note

Base pairs are always denoted in 5’->3’ direction. Thus the enclosed base pair must be ‘turned arround’ when evaluating the free energy of the internal-loop

This function is threadsafe

Parameters

- **n1** – The size of the ‘left’-loop (number of unpaired nucleotides)
- **n2** – The size of the ‘right’-loop (number of unpaired nucleotides)
- **type** – The pair type of the base pair closing the internal loop
- **type_2** – The pair type of the enclosed base pair
- **si1** – The 5’-mismatching nucleotide of the closing pair
- **sj1** – The 3’-mismatching nucleotide of the closing pair
- **sp1** – The 3’-mismatching nucleotide of the enclosed pair

- **sq1** – The 5'-mismatching nucleotide of the enclosed pair
- **P** – The datastructure containing scaled energy parameters

Returns
The Free energy of the Interior-loop in dcal/mol

static *FLT_OR_DBL* **exp_E_IntLoop**(int u1, int u2, int type, int type2, short si1, short sj1, short sp1, short sq1, *vrna_exp_param_t* *P)

```
#include <ViennaRNA/eval/internal.h>
```

Compute Boltzmann weight $e^{-\Delta G/kT}$ of internal loop

multiply by scale[u1+u2+2] for scaling

See also

get_scaled_pf_parameters(), *vrna_exp_param_t*, E_IntLoop()

Note

This function is threadsafe

Parameters

- **u1** – The size of the 'left'-loop (number of unpaired nucleotides)
- **u2** – The size of the 'right'-loop (number of unpaired nucleotides)
- **type** – The pair type of the base pair closing the internal loop
- **type2** – The pair type of the enclosed base pair
- **si1** – The 5'-mismatching nucleotide of the closing pair
- **sj1** – The 3'-mismatching nucleotide of the closing pair
- **sp1** – The 3'-mismatching nucleotide of the enclosed pair
- **sq1** – The 5'-mismatching nucleotide of the enclosed pair
- **P** – The datastructure containing scaled Boltzmann weights of the energy parameters

Returns
The Boltzmann weight of the Interior-loop

static int **E_IntLoop_Co**(int type, int type_2, int i, int j, int p, int q, int cutpoint, short si1, short sj1, short sp1, short sq1, int dangles, *vrna_param_t* *P)

```
#include <ViennaRNA/eval/internal.h>
```

static int **__E_IntLoop_Co**(int type, int type_2, int i, int j, int p, int q, int cutpoint, short si1, short sj1, short sp1, short sq1, int dangles, *vrna_param_t* *P)

```
#include <ViennaRNA/eval/internal.h>
```

static int **ubf_eval_int_loop**(int i, int j, int p, int q, int i1, int j1, int p1, int q1, short si, short sj, short sp, short sq, unsigned char type, unsigned char type_2, int *rtype, int ij, int cp, *vrna_param_t* *P, *vrna_sc_t* *sc)

```
#include <ViennaRNA/eval/internal.h>
```

static int **ubf_eval_int_loop2**(int i, int j, int p, int q, int i1, int j1, int p1, int q1, short si, short sj, short sp, short sq, unsigned char type, unsigned char type_2, int *rtype, int ij, unsigned int *sn, unsigned int *ss, *vrna_param_t* *P, *vrna_sc_t* *sc)

```
#include <ViennaRNA/eval/internal.h>
```

static int **ubf_eval_ext_int_loop**(int i, int j, int p, int q, int i1, int j1, int p1, int q1, short si, short sj, short sp, short sq, unsigned char type, unsigned char type_2, int length, *vrna_param_t* *P, *vrna_sc_t* *sc)

```
#include <ViennaRNA/eval/internal.h>
```

int **vrna_eval_int_loop**(*vrna_fold_compound_t* *fc, int i, int j, int k, int l)

```
#include <ViennaRNA/eval/internal.h>
```

SWIG Wrapper Notes:

This function is attached as method `eval_int_loop()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_int_loop()* in the *The RNA Python module* .

int **vrna_E_stack**(*vrna_fold_compound_t* *fc, int i, int j)

```
#include <ViennaRNA/eval/internal.h>
```

FLT_OR_DBL **vrna_exp_E_interior_loop**(*vrna_fold_compound_t* *fc, int i, int j, int k, int l)

```
#include <ViennaRNA/eval/internal.h>
```

static int **E_MLstem**(int type, int si1, int sj1, *vrna_param_t* *P)

```
#include <ViennaRNA/eval/multibranch.h>
```

static *FLT_OR_DBL* **exp_E_MLstem**(int type, int si1, int sj1, *vrna_exp_param_t* *P)

```
#include <ViennaRNA/eval/multibranch.h>
```

float **energy_of_structure**(const char *string, const char *structure, int verbosity_level)

Calculate the free energy of an already folded RNA using global model detail settings.

```
#include <ViennaRNA/eval/structures.h>
```

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Deprecated:

Use *vrna_eval_structure()* or *vrna_eval_structure_verbose()* instead!

See also

vrna_eval_structure()

Note

OpenMP: This function relies on several global model settings variables and thus is not to be considered threadsafe. See *energy_of_struct_par()* for a completely threadsafe implementation.

Parameters

- **string** – RNA sequence
- **structure** – secondary structure in dot-bracket notation
- **verbosity_level** – a flag to turn verbose output on/off

Returns
the free energy of the input structure given the input sequence in kcal/mol

float **energy_of_struct_par**(const char *string, const char *structure, *vrna_param_t* *parameters, int verbosity_level)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Deprecated:
Use *vrna_eval_structure()* or *vrna_eval_structure_verbose()* instead!

See also

vrna_eval_structure()

Parameters

- **string** – RNA sequence in uppercase letters
- **structure** – Secondary structure in dot-bracket notation
- **parameters** – A data structure containing the prescaled energy contributions and the model details.
- **verbosity_level** – A flag to turn verbose output on/off

Returns
The free energy of the input structure given the input sequence in kcal/mol

float **energy_of_circ_structure**(const char *string, const char *structure, int verbosity_level)

Calculate the free energy of an already folded circular RNA.

```
#include <ViennaRNA/eval/structures.h>
```

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Deprecated:
Use *vrna_eval_structure()* or *vrna_eval_structure_verbose()* instead!

See also

vrna_eval_structure()

Note

OpenMP: This function relies on several global model settings variables and thus is not to be considered threadsafe. See *energy_of_circ_struct_par()* for a completely threadsafe implementation.

Parameters

- **string** – RNA sequence
- **structure** – Secondary structure in dot-bracket notation
- **verbosity_level** – A flag to turn verbose output on/off

Returns

The free energy of the input structure given the input sequence in kcal/mol

float **energy_of_circ_struct_par**(const char *string, const char *structure, *vrna_param_t* *parameters, int verbosity_level)

Calculate the free energy of an already folded circular RNA.

```
#include <ViennaRNA/eval/structures.h>
```

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Deprecated:

Use *vrna_eval_structure()* or *vrna_eval_structure_verbose()* instead!

See also

vrna_eval_structure()

Parameters

- **string** – RNA sequence
- **structure** – Secondary structure in dot-bracket notation
- **parameters** – A data structure containing the prescaled energy contributions and the model details.
- **verbosity_level** – A flag to turn verbose output on/off

Returns

The free energy of the input structure given the input sequence in kcal/mol

float **energy_of_gquad_structure**(const char *string, const char *structure, int verbosity_level)

```
#include <ViennaRNA/eval/structures.h>
```

float **energy_of_gquad_struct_par**(const char *string, const char *structure, *vrna_param_t* *parameters, int verbosity_level)

```
#include <ViennaRNA/eval/structures.h>
```

int **energy_of_structure_pt**(const char *string, short *ptable, short *s, short *s1, int verbosity_level)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Deprecated:

Use *vrna_eval_structure_pt()* or *vrna_eval_structure_pt_verbose()* instead!

See also

vrna_eval_structure_pt()

Note

OpenMP: This function relies on several global model settings variables and thus is not to be considered threadsafe. See *energy_of_struct_pt_par()* for a completely threadsafe implementation.

Parameters

- **string** – RNA sequence
- **ptable** – the pair table of the secondary structure
- **s** – encoded RNA sequence
- **s1** – encoded RNA sequence
- **verbosity_level** – a flag to turn verbose output on/off

Returns

the free energy of the input structure given the input sequence in 10kcal/mol

int **energy_of_struct_pt_par**(const char *string, short *ptable, short *s, short *s1, *vrna_param_t* *parameters, int verbosity_level)

Calculate the free energy of an already folded RNA.

```
#include <ViennaRNA/eval/structures.h>
```

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Deprecated:

Use *vrna_eval_structure_pt()* or *vrna_eval_structure_pt_verbose()* instead!

See also

vrna_eval_structure_pt()

Parameters

- **string** – RNA sequence in uppercase letters
- **ptable** – The pair table of the secondary structure
- **s** – Encoded RNA sequence
- **s1** – Encoded RNA sequence
- **parameters** – A data structure containing the prescaled energy contributions and the model details.
- **verbosity_level** – A flag to turn verbose output on/off

Returns

The free energy of the input structure given the input sequence in 10kcal/mol

float **energy_of_move**(const char *string, const char *structure, int m1, int m2)

Calculate energy of a move (closing or opening of a base pair).

```
#include <ViennaRNA/eval/structures.h>
```

If the parameters m1 and m2 are negative, it is deletion (opening) of a base pair, otherwise it is insertion (opening).

Deprecated:

Use *vrna_eval_move()* instead!

See also

vrna_eval_move()

Parameters

- **string** – RNA sequence
- **structure** – secondary structure in dot-bracket notation
- **m1** – first coordinate of base pair
- **m2** – second coordinate of base pair

Returns

energy change of the move in kcal/mol

int **energy_of_move_pt**(short *pt, short *s, short *s1, int m1, int m2)

Calculate energy of a move (closing or opening of a base pair).

```
#include <ViennaRNA/eval/structures.h>
```

If the parameters m1 and m2 are negative, it is deletion (opening) of a base pair, otherwise it is insertion (opening).

Deprecated:

Use *vrna_eval_move_pt()* instead!

See also

vrna_eval_move_pt()

Parameters

- **pt** – the pair table of the secondary structure
- **s** – encoded RNA sequence
- **s1** – encoded RNA sequence
- **m1** – first coordinate of base pair
- **m2** – second coordinate of base pair

Returns

energy change of the move in 10cal/mol

int **loop_energy**(short *ptable, short *s, short *s1, int i)

Calculate energy of a loop.

```
#include <ViennaRNA/eval/structures.h>
```

Deprecated:

Use *vrna_eval_loop_pt()* instead!

See also

vrna_eval_loop_pt()

Parameters

- **ptable** – the pair table of the secondary structure
- **s** – encoded RNA sequence
- **s1** – encoded RNA sequence
- **i** – position of covering base pair

Returns

free energy of the loop in 10cal/mol

float **energy_of_struct**(const char *string, const char *structure)

```
#include <ViennaRNA/eval/structures.h>
```

Calculate the free energy of an already folded RNA

Deprecated:

This function is deprecated and should not be used in future programs! Use *energy_of_structure()* instead!

See also

energy_of_structure, *energy_of_circ_struct()*, *energy_of_struct_pt()*

Note

This function is not entirely threadsafe! Depending on the state of the global variable *eos_debug* it prints energy information to stdout or not...

Parameters

- **string** – RNA sequence
- **structure** – secondary structure in dot-bracket notation

Returns

the free energy of the input structure given the input sequence in kcal/mol

int **energy_of_struct_pt**(const char *string, short *ptable, short *s, short *s1)

```
#include <ViennaRNA/eval/structures.h>
```

Calculate the free energy of an already folded RNA

Deprecated:

This function is deprecated and should not be used in future programs! Use *energy_of_structure_pt()* instead!

See also

make_pair_table(), *energy_of_structure()*

Note

This function is not entirely threadsafe! Depending on the state of the global variable *eos_debug* it prints energy information to stdout or not...

Parameters

- **string** – RNA sequence
- **ptable** – the pair table of the secondary structure
- **s** – encoded RNA sequence
- **s1** – encoded RNA sequence

Returns
the free energy of the input structure given the input sequence in 10kcal/mol

float **energy_of_circ_struct**(const char *string, const char *structure)

```
#include <ViennaRNA/eval/structures.h>
```

Calculate the free energy of an already folded circular RNA

Deprecated:
This function is deprecated and should not be used in future programs Use *energy_of_circ_structure()* instead!

See also
energy_of_circ_structure(), *energy_of_struct()*, *energy_of_struct_pt()*

Note
This function is not entirely threadsafe! Depending on the state of the global variable *eos_debug* it prints energy information to stdout or not...

Parameters
- **string** – RNA sequence
- **structure** – secondary structure in dot-bracket notation

Returns
the free energy of the input structure given the input sequence in kcal/mol

Variables

int **cut_point**
first pos of second seq for cofolding

```
#include <ViennaRNA/eval/basic.h>
```

int **eos_debug**
verbose info from energy_of_struct

```
#include <ViennaRNA/eval/basic.h>
```

4.1.5 RNA-RNA Interaction

Basic RNA Interaction API

Concatenating RNA sequences

API for the sequence concatenation approach to predict RNA-RNA interactions.

typedef struct vrna_dimer_conc_s **vrna_dimer_conc_t**

Typename for the data structure that stores the dimer concentrations, vrna_dimer_conc_s, as required by vrna_pf_dimer_concentration().

```
#include <ViennaRNA/concentrations.h>
```

typedef struct vrna_dimer_conc_s **ConcEnt**

Backward compatibility typedef for vrna_dimer_conc_s.

```
#include <ViennaRNA/concentrations.h>
```

vrna_dimer_conc_t ***vrna_pf_dimer_concentrations**(double FcAB, double FcAA, double FcBB, double FEA, double FEB, const double *startconc, const *vrna_exp_param_t* *exp_params)

Given two start monomer concentrations a and b, compute the concentrations in thermodynamic equilibrium of all dimers and the monomers.

```
#include <ViennaRNA/concentrations.h>
```

This function takes an array 'startconc' of input concentrations with alternating entries for the initial concentrations of molecules A and B (terminated by two zeroes), then computes the resulting equilibrium concentrations from the free energies for the dimers. Dimer free energies should be the dimer-only free energies, i.e. the FcAB entries from the *vrna_dimer_pf_t* struct.

Parameters

- **FcAB** – Free energy of AB dimer (FcAB entry)
- **FcAA** – Free energy of AA dimer (FcAB entry)
- **FcBB** – Free energy of BB dimer (FcAB entry)
- **FEA** – Free energy of monomer A
- **FEB** – Free energy of monomer B
- **startconc** – List of start concentrations [a0],[b0],[a1],[b1],...,[an][bn],[0],[0]
- **exp_params** – The precomputed Boltzmann factors

Returns

vrna_dimer_conc_t array containing the equilibrium energies and start concentrations

double ***vrna_equilibrium_constants**(const double *dG_complexes, const double *dG_strands, const unsigned int **A, double kT, size_t strands, size_t complexes)

Compute equilibrium constants for interacting RNA complexes.

```
#include <ViennaRNA/concentrations.h>
```

Given a set of partition functions for interacting RNA complexes this function computes the equilibrium constants for each complex. For instance, the equilibrium constant K_{AB} a dimer complex composed of two strands A and B can be easily computed from the partition functions Z_A and Z_B of the monomer strands and the partition function Z_{AB} of AB complexes, i.e.:

$$K_{AB} = \frac{Z_{AB}}{Z_A Z_B}$$

SWIG Wrapper Notes:

This function is available as overloaded function `equilibrium_constants()` where the last two parameters `strands` and `complexes` are optional. In case they are omitted, the respective numbers will be determined from the length of the input list parameters `dG_complexes` and `dG_strands`, respectively. See also *`RNA.equilibrium_constants()`* in the *The RNA Python module*.

Parameters

- **`dG_complexes`** – The partition functions of the complexes (specified as ensemble free energies $\Delta G = -RT \ln Z$ in units of $kcal \cdot mol^{-1}$
- **`dG_strands`** – The partition functions of the monomer species (specified as ensemble free energies $\Delta G = -RT \ln Z$ in units of $kcal \cdot mol^{-1}$
- **`A`** – An NxM matrix indicating the composition of the individual complexes (M), i.e. how many of the monomers (N) each complex consists of
- **`kT`** – Thermodynamic temperature times Boltzmann constant in units of $cal \cdot mol^{-1}$ (used for temperature scaling)
- **`strands`** – The number of monomer species
- **`complexes`** – The number of interacting complexes

Returns

The equilibrium constants for each complex

vrna_dimer_pf_t **`vrna_pf_co_fold`**(const char *seq, char *structure, *vrna_ep_t* **pl)

Calculate partition function and base pair probabilities of nucleic acid/nucleic acid dimers.

```
#include <ViennaRNA/partfunc/global.h>
```

This simplified interface to *vrna_pf_dimer()* computes the partition function and, if required, base pair probabilities for an RNA-RNA interaction using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing.

See also

vrna_pf_dimer()

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use *vrna_pf_dimer()*, and the data structure *vrna_fold_compound_t* instead.

Parameters

- **`seq`** – Two concatenated RNA sequences with a delimiting '&' in between
- **`structure`** – A pointer to the character array where position-wise pairing propensity will be stored. (Maybe NULL)
- **`pl`** – A pointer to a list of *vrna_ep_t* to store pairing probabilities (Maybe NULL)

Returns

vrna_dimer_pf_t structure containing a set of energies needed for concentration computations.

typedef struct *vrna_dimer_pf_s* **vrna_dimer_pf_t**

Typename for the data structure that stores the dimer partition functions, *vrna_dimer_pf_s*, as returned by *vrna_pf_dimer()*.

```
#include <ViennaRNA/partfunc/global.h>
```

typedef struct *vrna_dimer_pf_s* **cofoldF**

Backward compatibility typedef for *vrna_dimer_pf_s*.

```
#include <ViennaRNA/partfunc/global.h>
```

int **mirnatog**

Toggles no intrabp in 2nd mol.

```
#include <ViennaRNA/part_func_co.h>
```

double **F_monomer**[2]

Free energies of the two monomers.

```
#include <ViennaRNA/part_func_co.h>
```

double ***vrna_equilibrium_conc**(const double *eq_constants, double *concentration_strands, const unsigned int **A, size_t num_strands, size_t num_complexes)

Compute equilibrium concentration of interacting RNA complexes.

```
#include <ViennaRNA/wrap_dlib.h>
```

This function allows one to compute, given a set of initial RNA monomer concentrations, equilibrium concentrations for a set of interacting RNA complexes.

SWIG Wrapper Notes:

This function is available as overloaded function `equilibrium_conc()` where the last two parameters `num_strands` and `num_complexes` are optional. In case they are omitted, the respective numbers will be determined from the length of the input list parameters `eq_constants` and `concentration_strands`, respectively. Note, that the C implementation stores equilibrium concentrations for the monomer species in place within the `concentration_strands` parameter. In contrast to that, the scripting language wrappers will rather append them to the returned output list.

Parameters

- **eq_constants** – The equilibrium constants for the individual complexes
- **concentration_strands** – The initial concentrations of the monomer species (also used for output of the equilibrium concentrations for the monomer species)
- **A** – An `NxM` matrix indicating the composition of the individual complexes (M), i.e. how many of the monomers (N) each complex consists of
- **num_strands** – The number of monomer species
- **num_complexes** – The number of interacting complexes

Returns

The equilibrium concentrations for each complex

RNA-RNA interaction as a Stepwise Process

Functions

pu_contrib ***pf_unstru**(char *sequence, int max_w)

Calculate the partition function over all unpaired regions of a maximal length.

```
#include <ViennaRNA/part_func_up.h>
```

You have to call function *pf_fold()* providing the same sequence before calling *pf_unstru()*. If you want to calculate unpaired regions for a constrained structure, set variable 'structure' in function '*pf_fold()*' to the constrain string. It returns a *pu_contrib* struct containing four arrays of dimension [i = 1 to length(sequence)][j = 0 to u-1] containing all possible contributions to the probabilities of unpaired regions of maximum length u. Each array in *pu_contrib* contains one of the contributions to the total probability of being unpaired: The probability of being unpaired within an exterior loop is in array *pu_contrib.E*, the probability of being unpaired within a hairpin loop is in array *pu_contrib.H*, the probability of being unpaired within an internal loop is in array *pu_contrib.I* and probability of being unpaired within a multi-loop is in array *pu_contrib.M*. The total probability of being unpaired is the sum of the four arrays of *pu_contrib*.

This function frees everything allocated automatically. To free the output structure call free_pu_contrib().

Parameters

- **sequence**
- **max_w**

Returns

interact ***pf_interact**(const char *s1, const char *s2, *pu_contrib* *p_c, *pu_contrib* *p_c2, int max_w, char *cstruc, int incr3, int incr5)

Calculates the probability of a local interaction between two sequences.

```
#include <ViennaRNA/part_func_up.h>
```

The function considers the probability that the region of interaction is unpaired within 's1' and 's2'. The longer sequence has to be given as 's1'. The shorter sequence has to be given as 's2'. Function *pf_unstru()* has to be called for 's1' and 's2', where the probabilities of being unpaired have to be given in 'p_c' and 'p_c2', respectively. If you do not want to include the probabilities of being unpaired for 's2' set 'p_c2' to NULL. If variable 'cstruc' is not NULL, constrained folding is done: The available constrains for intermolecular interaction are: '.' (no constrain), 'x' (the base has no intermolecular interaction) and '|' (the corresponding base has to be paired intermolecularily).

The parameter 'w' determines the maximal length of the interaction. The parameters 'incr5' and 'incr3' allows inclusion of unpaired residues left ('incr5') and right ('incr3') of the region of interaction in 's1'. If the 'incr' options are used, function *pf_unstru()* has to be called with w=w+incr5+incr3 for the longer sequence 's1'.

It returns a structure of type *interact* which contains the probability of the best local interaction including residue i in Pi and the minimum free energy in Gi, where i is the position in sequence 's1'. The member Gikjl of structure *interact* is the best interaction between region [k,i] k<i in longer sequence 's1' and region [j,l] j<l in 's2'. Gikjl_wo is Gikjl without the probability of beeing unpaired.

Use *free_interact()* to free the returned structure, all other stuff is freed inside *pf_interact()*.

Parameters

- **s1**
- **s2**

- **p_c**
- **p_c2**
- **max_w**
- **cstruc**
- **incr3**
- **incr5**

Returns

void **free_interact**(*interact* *pin)

Frees the output of function *pf_interact()*.

```
#include <ViennaRNA/part_func_up.h>
```

int **Up_plot**(*pu_contrib* *p_c, *pu_contrib* *p_c_sh, *interact* *pint, char *ofile, int **unpaired_values, char *select_contrib, char *head, unsigned int mode)

```
#include <ViennaRNA/part_func_up.h>
```

pu_contrib ***get_pu_contrib_struct**(unsigned int n, unsigned int w)

```
#include <ViennaRNA/part_func_up.h>
```

void **free_pu_contrib_struct**(*pu_contrib* *pu)

Frees the output of function *pf_unstru()*.

```
#include <ViennaRNA/part_func_up.h>
```

void **free_pu_contrib**(*pu_contrib* *pu)

```
#include <ViennaRNA/part_func_up.h>
```

4.1.6 Inverse Folding (Design)

RNA sequence design.

Functions

float **inverse_fold**(char *start, const char *target)

Find sequences with predefined structure.

```
#include <ViennaRNA/inverse/basic.h>
```

This function searches for a sequence with minimum free energy structure provided in the parameter 'target', starting with sequence 'start'. It returns 0 if the search was successful, otherwise a structure distance in terms of the energy difference between the search result and the actual target 'target' is returned. The found sequence is returned in 'start'. If *give_up* is set to 1, the function will return as soon as it is clear that the search will be unsuccessful, this speeds up the algorithm if you are only interested in exact solutions.

Parameters

- **start** – The start sequence
- **target** – The target secondary structure in dot-bracket notation

Returns
The distance to the target in case a search was unsuccessful, 0 otherwise

float **inverse_pf_fold**(char *start, const char *target)

Find sequence that maximizes probability of a predefined structure.

```
#include <ViennaRNA/inverse/basic.h>
```

This function searches for a sequence with maximum probability to fold into the provided structure 'target' using the partition function algorithm. It returns $-kT \cdot \log(p)$ where p is the frequency of 'target' in the ensemble of possible structures. This is usually much slower than *inverse_fold()*.

Parameters

- **start** – The start sequence
- **target** – The target secondary structure in dot-bracket notation

Returns
The distance to the target in case a search was unsuccessful, 0 otherwise

char ****vrna_inverse_seed_linearbp**(const char *structure, size_t num_samples, *vrna_inverse_linearbp_opt_t* *options)

Generate a set of sequence seeds compatible with a given structure according to the LinearBP design algorithm.

```
#include <ViennaRNA/inverse/linearbpseeds.h>
```

Parameters

- **structure** – A secondary structure in dot-bracket format
- **num_samples** – The target number of solutions
- **options** – A pointer to the options data structure to modifiy the behavior of the algorithm

Returns
A pointer to an array of sequences (end of list indicated by **NULL** entry) or **NULL** on error

Variables

char ***symbolset**

This global variable points to the allowed bases, initially "AUGC". It can be used to design sequences from reduced alphabets.

```
#include <ViennaRNA/inverse/basic.h>
```

float **final_cost**

```
#include <ViennaRNA/inverse/basic.h>
```

when to stop *inverse_pf_fold()*

int **give_up**

```
#include <ViennaRNA/inverse/basic.h>
```

default 0: try to minimize structure distance even if no exact solution can be found

int **inv_verbose**

```
#include <ViennaRNA/inverse/basic.h>
```

print out substructure on which *inverse_fold()* fails

struct **vrna_inverse_linearbp_opt_t**

#include <ViennaRNA/inverse/linearbpseeds.h> The options data structure for *vrna_inverse_seed_linearbp()*.

See also

vrna_inverse_seed_linearbp()

Public Members

unsigned int **biseparable**

Flag indicating if biseparanle mode is switched on (A nd Cs) in unpaired positions.

```
#include <ViennaRNA/inverse/linearbpseeds.h>
```

unsigned int **maxmodulo**

The maximal modulus to be considered for m/mA/mC.

```
#include <ViennaRNA/inverse/linearbpseeds.h>
```

double **GCweight**

The weight of the GC pairs (default is 0.0).

```
#include <ViennaRNA/inverse/linearbpseeds.h>
```

4.1.7 Classified Dynamic Programming Variants

This part describes algorithms that partition the structure space *a priori* and compute MFE or partition function for each of the resulting partitions. This approach directly leads to *Classified Dynamic Programming*.

Basic Interface

group **Classified Dynamic Programming Variants**

Distance Based Partitioning

The secondary structure space is divided into partitions according to the base pair distance to two given reference structures and all relevant properties are calculated for each of the resulting partitions.

See also…

For further details, we refer to Lorenz *et al.* [2009]

General

MFE Variants

Compute the minimum free energy (MFE) and secondary structures for a partitioning of the secondary structure space according to the base pair distance to two fixed reference structures basepair distance to two fixed reference structures.

Defines

TwoDfold_solution

```
#include <ViennaRNA/2Dfold.h>
```

Functions

vrna_sol_TwoD_t ***vrna_mfe_TwoD**(*vrna_fold_compound_t* *fc, int distance1, int distance2)

Compute MFE's and representative for distance partitioning.

```
#include <ViennaRNA/2Dfold.h>
```

This function computes the minimum free energies and a representative secondary structure for each distance class according to the two references specified in the datastructure 'vars'. The maximum basepair distance to each of both references may be set by the arguments 'distance1' and 'distance2', respectively. If both distance arguments are set to '-1', no restriction is assumed and the calculation is performed for each distance class possible.

The returned list contains an entry for each distance class. If a maximum basepair distance to either of the references was passed, an entry with k=l=-1 will be appended in the list, denoting the class where all structures exceeding the maximum will be thrown into. The end of the list is denoted by an attribute value of INF in the k-attribute of the list entry.

See also

vrna_fold_compound_TwoD(), *vrna_fold_compound_free()*, *vrna_pf_TwoD()*, *vrna_backtrack5_TwoD()*, *vrna_sol_TwoD_t*, *vrna_fold_compound_t*

Parameters

- **fc** – The datastructure containing all precomputed folding attributes
- **distance1** – maximum distance to reference1 (-1 means no restriction)
- **distance2** – maximum distance to reference2 (-1 means no restriction)

Returns

A list of minimum free energies (and corresponding structures) for each distance class

char ***vrna_backtrack5_TwoD**(*vrna_fold_compound_t* *fc, int k, int l, unsigned int j)

Backtrack a minimum free energy structure from a 5' section of specified length.

```
#include <ViennaRNA/2Dfold.h>
```

This function allows one to backtrack a secondary structure beginning at the 5' end, a specified length and residing in a specific distance class. If the argument 'k' gets a value of -1, the structure that is backtracked is assumed to reside in the distance class where all structures exceeding the maximum basepair distance specified in *vrna_mfe_TwoD()* belong to.

See also

vrna_mfe_TwoD()

Note

The argument 'vars' must contain precalculated energy values in the energy matrices, i.e. a call to *vrna_mfe_TwoD()* preceding this function is mandatory!

Parameters

- **fc** – The datastructure containing all precomputed folding attributes
- **j** – The length in nucleotides beginning from the 5' end
- **k** – distance to reference1 (may be -1)
- **l** – distance to reference2

TwoDfold_vars ***get_TwoDfold_variables**(const char *seq, const char *structure1, const char *structure2, int circ)

Get a structure of type *TwoDfold_vars* prefilled with current global settings.

```
#include <ViennaRNA/2Dfold.h>
```

This function returns a datastructure of type *TwoDfold_vars*. The data fields inside the *TwoDfold_vars* are prefilled by global settings and all memory allocations necessary to start a computation are already done for the convenience of the user

Deprecated:

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, and *vrna_fold_compound_free()* instead!

Note

Make sure that the reference structures are compatible with the sequence according to Watson-Crick- and Wobble-base pairing

Parameters

- **seq** – The RNA sequence
- **structure1** – The first reference structure in dot-bracket notation
- **structure2** – The second reference structure in dot-bracket notation
- **circ** – A switch to indicate the assumption to fold a circular instead of linear RNA (0=OFF, 1=ON)

Returns
A datastructure prefilled with folding options and allocated memory

void **destroy_TwoDfold_variables**(*TwoDfold_vars* *our_variables)

Destroy a *TwoDfold_vars* datastructure without memory loss.

```
#include <ViennaRNA/2Dfold.h>
```

This function free's all allocated memory that depends on the datastructure given.

Deprecated:

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, and *vrna_fold_compound_free()* instead!

Parameters

- **our_variables** – A pointer to the datastructure to be destroyed

vrna_sol_TwoD_t ***TwoDfoldList**(*TwoDfold_vars* *vars, int distance1, int distance2)

Compute MFE's and representative for distance partitioning.

```
#include <ViennaRNA/2Dfold.h>
```

This function computes the minimum free energies and a representative secondary structure for each distance class according to the two references specified in the datastructure 'vars'. The maximum basepair distance to each of both references may be set by the arguments 'distance1' and 'distance2', respectively. If both distance arguments are set to '-1', no restriction is assumed and the calculation is performed for each distance class possible.

The returned list contains an entry for each distance class. If a maximum basepair distance to either of the references was passed, an entry with k=l=-1 will be appended in the list, denoting the class where all structures exceeding the maximum will be thrown into. The end of the list is denoted by an attribute value of INF in the k-attribute of the list entry.

Deprecated:

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, and *vrna_fold_compound_free()* instead!

Parameters

- **vars** – the datastructure containing all predefined folding attributes
- **distance1** – maximum distance to reference1 (-1 means no restriction)
- **distance2** – maximum distance to reference2 (-1 means no restriction)

char ***TwoDfold_backtrack_f5**(unsigned int j, int k, int l, *TwoDfold_vars* *vars)

Backtrack a minimum free energy structure from a 5' section of specified length.

```
#include <ViennaRNA/2Dfold.h>
```

This function allows one to backtrack a secondary structure beginning at the 5' end, a specified length and residing in a specific distance class. If the argument 'k' gets a value of -1, the structure that is backtracked is assumed to reside in the distance class where all structures exceeding the maximum basepair distance specified in TwoDfold() belong to.

Deprecated:

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, *vrna_backtrack5_TwoD()*, and *vrna_fold_compound_free()* instead!

Note

The argument 'vars' must contain precalculated energy values in the energy matrices, i.e. a call to TwoDfold() preceding this function is mandatory!

Parameters

- **j** – The length in nucleotides beginning from the 5' end
- **k** – distance to reference1 (may be -1)
- **l** – distance to reference2
- **vars** – the datastructure containing all predefined folding attributes

vrna_sol_TwoD_t ****TwoDfold**(*TwoDfold_vars* *our_variables, int distance1, int distance2)

```
#include <ViennaRNA/2Dfold.h>
```

struct **vrna_sol_TwoD_t**

#include <ViennaRNA/2Dfold.h> Solution element returned from *vrna_mfe_TwoD()*.

This element contains free energy and structure for the appropriate kappa (k), lambda (l) neighborhood The datastructure contains two integer attributes 'k' and 'l' as well as an attribute 'en' of type float representing the free energy in kcal/mol and an attribute 's' of type char* containg the secondary structure representative,

A value of INF in k denotes the end of a list

See also

vrna_mfe_TwoD()

Public Members

int **k**

Distance to first reference.

```
#include <ViennaRNA/2Dfold.h>
```

int **l**

Distance to second reference.

```
#include <ViennaRNA/2Dfold.h>
```

float **en**

Free energy in kcal/mol.

```
#include <ViennaRNA/2Dfold.h>
```

char *s

MFE representative structure in dot-bracket notation.

```
#include <ViennaRNA/2Dfold.h>
```

struct TwoDfold_vars

#include <*ViennaRNA/2Dfold.h*> Variables compound for 2Dfold MFE folding.

Deprecated:

This data structure will be removed from the library soon! Use *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, and *vrna_fold_compound_free()* instead!

Public Members

vrna_param_t *P

Precomputed energy parameters and model details.

```
#include <ViennaRNA/2Dfold.h>
```

int do_backtrack

Flag whether to do backtracing of the structure(s) or not.

```
#include <ViennaRNA/2Dfold.h>
```

char *ptype

Precomputed array of pair types.

```
#include <ViennaRNA/2Dfold.h>
```

char *sequence

The input sequence.

```
#include <ViennaRNA/2Dfold.h>
```

short *S

```
#include <ViennaRNA/2Dfold.h>
```

short *S1

The input sequences in numeric form.

```
#include <ViennaRNA/2Dfold.h>
```

unsigned int maxD1

Maximum allowed base pair distance to first reference.

```
#include <ViennaRNA/2Dfold.h>
```

unsigned int **maxD2**

Maximum allowed base pair distance to second reference.

```
#include <ViennaRNA/2Dfold.h>
```

unsigned int ***mm1**

Maximum matching matrix, reference struct 1 disallowed.

```
#include <ViennaRNA/2Dfold.h>
```

unsigned int ***mm2**

Maximum matching matrix, reference struct 2 disallowed.

```
#include <ViennaRNA/2Dfold.h>
```

int ***my_iindx**

Index for moving in quadratic distancy dimensions.

```
#include <ViennaRNA/2Dfold.h>
```

double **temperature**

```
#include <ViennaRNA/2Dfold.h>
```

unsigned int ***referenceBPs1**

Matrix containing number of basepairs of reference structure1 in interval [i,j].

```
#include <ViennaRNA/2Dfold.h>
```

unsigned int ***referenceBPs2**

Matrix containing number of basepairs of reference structure2 in interval [i,j].

```
#include <ViennaRNA/2Dfold.h>
```

unsigned int ***bpdist**

Matrix containing base pair distance of reference structure 1 and 2 on interval [i,j].

```
#include <ViennaRNA/2Dfold.h>
```

short ***reference_pt1**

```
#include <ViennaRNA/2Dfold.h>
```

short ***reference_pt2**

```
#include <ViennaRNA/2Dfold.h>
```

int **circ**

```
#include <ViennaRNA/2Dfold.h>
```

int **dangles**

```
#include <ViennaRNA/2Dfold.h>
```

unsigned int **seq_length**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_F5**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_F3**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_C**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_M**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_M1**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_M2**

```
#include <ViennaRNA/2Dfold.h>
```

int **E_Fc

```
#include <ViennaRNA/2Dfold.h>
```

int **E_FcH

```
#include <ViennaRNA/2Dfold.h>
```

int **E_FcI

```
#include <ViennaRNA/2Dfold.h>
```

int **E_FcM

```
#include <ViennaRNA/2Dfold.h>
```

int **l_min_values

```
#include <ViennaRNA/2Dfold.h>
```

int **l_max_values

```
#include <ViennaRNA/2Dfold.h>
```

int *k_min_values

```
#include <ViennaRNA/2Dfold.h>
```

int *k_max_values

```
#include <ViennaRNA/2Dfold.h>
```

int **l_min_values_m

```
#include <ViennaRNA/2Dfold.h>
```

int **l_max_values_m

```
#include <ViennaRNA/2Dfold.h>
```

int *k_min_values_m

```
#include <ViennaRNA/2Dfold.h>
```

int *k_max_values_m

```
#include <ViennaRNA/2Dfold.h>
```

int **l_min_values_m1

```
#include <ViennaRNA/2Dfold.h>
```

int **l_max_values_m1

```
#include <ViennaRNA/2Dfold.h>
```

int *k_min_values_m1

```
#include <ViennaRNA/2Dfold.h>
```

int ***k_max_values_m1**

```
#include <ViennaRNA/2Dfold.h>
```

int ****l_min_values_f**

```
#include <ViennaRNA/2Dfold.h>
```

int ****l_max_values_f**

```
#include <ViennaRNA/2Dfold.h>
```

int ***k_min_values_f**

```
#include <ViennaRNA/2Dfold.h>
```

int ***k_max_values_f**

```
#include <ViennaRNA/2Dfold.h>
```

int ****l_min_values_f3**

```
#include <ViennaRNA/2Dfold.h>
```

int ****l_max_values_f3**

```
#include <ViennaRNA/2Dfold.h>
```

int ***k_min_values_f3**

```
#include <ViennaRNA/2Dfold.h>
```

int ***k_max_values_f3**

```
#include <ViennaRNA/2Dfold.h>
```

int ****l_min_values_m2**

```
#include <ViennaRNA/2Dfold.h>
```

int ****l_max_values_m2**

```
#include <ViennaRNA/2Dfold.h>
```

int ***k_min_values_m2**

```
#include <ViennaRNA/2Dfold.h>
```

int ***k_max_values_m2**

```
#include <ViennaRNA/2Dfold.h>
```

int ***l_min_values_fc**

```
#include <ViennaRNA/2Dfold.h>
```

int ***l_max_values_fc**

```
#include <ViennaRNA/2Dfold.h>
```

int **k_min_values_fc**

```
#include <ViennaRNA/2Dfold.h>
```

int **k_max_values_fc**

```
#include <ViennaRNA/2Dfold.h>
```

int ***l_min_values_fcH**

```
#include <ViennaRNA/2Dfold.h>
```

int ***l_max_values_fcH**

```
#include <ViennaRNA/2Dfold.h>
```

int **k_min_values_fcH**

```
#include <ViennaRNA/2Dfold.h>
```

int **k_max_values_fcH**

```
#include <ViennaRNA/2Dfold.h>
```

int ***l_min_values_fcI**

```
#include <ViennaRNA/2Dfold.h>
```

int ***l_max_values_fcI**

```
#include <ViennaRNA/2Dfold.h>
```

int **k_min_values_fcI**

```
#include <ViennaRNA/2Dfold.h>
```

int **k_max_values_fcI**

```
#include <ViennaRNA/2Dfold.h>
```

int ***l_min_values_fcM**

```
#include <ViennaRNA/2Dfold.h>
```

int ***l_max_values_fcM**

```
#include <ViennaRNA/2Dfold.h>
```

int **k_min_values_fcM**

```
#include <ViennaRNA/2Dfold.h>
```

int **k_max_values_fcM**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_F5_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_F3_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_C_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_M_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_M1_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int ***E_M2_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int **E_Fc_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int **E_FcH_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int **E_FcI_rem**

```
#include <ViennaRNA/2Dfold.h>
```

int **E_FcM_rem**

```
#include <ViennaRNA/2Dfold.h>
```

vrna_fold_compound_t ***compatibility**

```
#include <ViennaRNA/2Dfold.h>
```

Partition Function Variants

Compute the partition function and stochastically sample secondary structures for a partitioning of the secondary structure space according to the base pair distance to two fixed reference structures.

Functions

vrna_sol_TwoD_pf_t ***vrna_pf_TwoD**(*vrna_fold_compound_t* *fc, int maxDistance1, int maxDistance2)

Compute the partition function for all distance classes.

```
#include <ViennaRNA/2Dpfold.h>
```

This function computes the partition functions for all distance classes according the two reference structures specified in the datastructure 'vars'. Similar to *vrna_mfe_TwoD()* the arguments maxDistance1 and maxDistance2 specify the maximum distance to both reference structures. A value of '-1' in either of them makes the appropriate distance restrictionless, i.e. all basepair distancies to the reference are taken into account during computation. In case there is a restriction, the returned solution contains an entry where the attribute k=l=-1 contains the partition function for all structures exceeding the restriction. A value of INF in the attribute 'k' of the returned list denotes the end of the list

See also

vrna_fold_compound_TwoD(), *vrna_fold_compound_free()*, *vrna_fold_compound* *vrna_sol_TwoD_pf_t*

Parameters

- **fc** – The datastructure containing all necessary folding attributes and matrices
- **maxDistance1** – The maximum basepair distance to reference1 (may be -1)
- **maxDistance2** – The maximum basepair distance to reference2 (may be -1)

Returns

A list of partition funtions for the corresponding distance classes

struct **vrna_sol_TwoD_pf_t**

#include <ViennaRNA/2Dpfold.h> Solution element returned from *vrna_pf_TwoD()*.

This element contains the partition function for the appropriate kappa (k), lambda (l) neighborhood The datastructure contains two integer attributes 'k' and 'l' as well as an attribute 'q' of type *FLT_OR_DBL*

A value of INF in k denotes the end of a list

See also

vrna_pf_TwoD()

Public Members

int **k**

Distance to first reference.

```
#include <ViennaRNA/2Dpfold.h>
```

int **l**

Distance to second reference.

```
#include <ViennaRNA/2Dpfold.h>
```

FLT_OR_DBL **q**

partition function

```
#include <ViennaRNA/2Dpfold.h>
```

Stochastic Backtracking

Functions related to stochastic backtracking from a specified distance class.

Functions

char ***vrna_pbacktrack_TwoD**(*vrna_fold_compound_t* *fc, int d1, int d2)

Sample secondary structure representatives from a set of distance classes according to their Boltzmann probability.

```
#include <ViennaRNA/2Dpfold.h>
```

If the argument 'd1' is set to '-1', the structure will be backtracked in the distance class where all structures exceeding the maximum basepair distance to either of the references reside.

See also

vrna_pf_TwoD()

Parameters

- **fc** – **[inout]** The *vrna_fold_compound_t* datastructure containing all necessary folding attributes and matrices
- **d1** – **[in]** The distance to reference1 (may be -1)
- **d2** – **[in]** The distance to reference2

Pre

The argument 'vars' must contain precalculated partition function matrices, i.e. a call to *vrna_pf_TwoD()* preceding this function is mandatory!

Returns

A sampled secondary structure in dot-bracket notation

char ***vrna_pbacktrack5_TwoD**(*vrna_fold_compound_t* *fc, int d1, int d2, unsigned int length)

Sample secondary structure representatives with a specified length from a set of distance classes according to their Boltzmann probability.

```
#include <ViennaRNA/2Dpfold.h>
```

This function does essentially the same as *vrna_pbacktrack_TwoD()* with the only difference that partial structures, i.e. structures beginning from the 5' end with a specified length of the sequence, are backtracked

See also

vrna_pbacktrack_TwoD(), *vrna_pf_TwoD()*

Note

This function does not work (since it makes no sense) for circular RNA sequences!

Parameters

- **fc** – **[inout]** The *vrna_fold_compound_t* datastructure containing all necessary folding attributes and matrices
- **d1** – **[in]** The distance to reference1 (may be -1)
- **d2** – **[in]** The distance to reference2
- **length** – **[in]** The length of the structure beginning from the 5' end

Pre

The argument 'vars' must contain precalculated partition function matrices, i.e. a call to *vrna_pf_TwoD()* preceding this function is mandatory!

Returns

A sampled secondary structure in dot-bracket notation

Density of States

Variables

int **density_of_states**[1000 + 1]

The Density of States.

```
#include <ViennaRNA/subopt/wuchty.h>
```

This array contains the density of states for an RNA sequences after a call to *subopt_par()*, *subopt()* or *subopt_circ()*.

See also

subopt_par(), *subopt()*, *subopt_circ()*

Pre

Call one of the functions *subopt_par()*, *subopt()* or *subopt_circ()* prior accessing the contents of this array

4.1.8 The RNA Secondary Structure Landscape

Basic Landscape API

Neighborhood and Move Sets

Functions to generate structural neighbors of a secondary structure according to a particular Move Set.

Defines

VRNA_MOVESET_INSERTION

Option flag indicating insertion move.

```
#include <ViennaRNA/landscape/move.h>
```

See also

vrna_neighbors(), *vrna_neighbors_successive*, *vrna_path()*

VRNA_MOVESET_DELETION

Option flag indicating deletion move.

```
#include <ViennaRNA/landscape/move.h>
```

See also

vrna_neighbors(), *vrna_neighbors_successive*, *vrna_path()*

VRNA_MOVESET_SHIFT

Option flag indicating shift move.

```
#include <ViennaRNA/landscape/move.h>
```

See also

vrna_neighbors(), *vrna_neighbors_successive*, *vrna_path()*

VRNA_MOVESET_NO_LP

Option flag indicating moves without lonely base pairs.

```
#include <ViennaRNA/landscape/move.h>
```

See also

vrna_neighbors(), *vrna_neighbors_successive*, *vrna_path()*

VRNA_MOVESET_DEFAULT

Option flag indicating default move set, i.e. insertions/deletion of a base pair.

```
#include <ViennaRNA/landscape/move.h>
```

See also

vrna_neighbors(), *vrna_neighbors_successive*, *vrna_path()*

VRNA_MOVE_NO_APPLY

```
#include <ViennaRNA/landscape/move.h>
```

VRNA_NEIGHBOR_CHANGE

State indicator for a neighbor that has been changed.

```
#include <ViennaRNA/landscape/neighbor.h>
```

See also

vrna_move_neighbor_diff_cb()

VRNA_NEIGHBOR_INVALID

State indicator for a neighbor that has been invalidated.

```
#include <ViennaRNA/landscape/neighbor.h>
```

See also

vrna_move_neighbor_diff_cb()

VRNA_NEIGHBOR_NEW

State indicator for a neighbor that has become newly available.

```
#include <ViennaRNA/landscape/neighbor.h>
```

See also

vrna_move_neighbor_diff_cb()

Typedefs

typedef struct *vrna_move_s* **vrna_move_t**

A single move that transforms a secondary structure into one of its neighbors.

```
#include <ViennaRNA/landscape/move.h>
```

typedef void (***vrna_move_update_f**)(*vrna_fold_compound_t* *fc, *vrna_move_t* neighbor, unsigned int state, void *data)

Prototype of the neighborhood update callback.

```
#include <ViennaRNA/landscape/neighbor.h>
```

See also

vrna_move_neighbor_diff_cb(), *VRNA_NEIGHBOR_CHANGE*, *VRNA_NEIGHBOR_INVALID*, *VRNA_NEIGHBOR_NEW*

Param fc
The fold compound the calling function is working on

Param neighbor
The move that generates the (changed or new) neighbor

Param state
The state of the neighbor (move) as supplied by argument `neighbor`

Param data
Some arbitrary data pointer as passed to *vrna_move_neighbor_diff_cb()*

typedef void **vrna_callback_move_update**(*vrna_fold_compound_t* *fc, *vrna_move_t* neighbor, unsigned int state, void *data)

```
#include <ViennaRNA/landscape/neighbor.h>
```

Functions

vrna_move_t **vrna_move_init**(int pos_5, int pos_3)

Create an atomic move.

```
#include <ViennaRNA/landscape/move.h>
```

See also

vrna_move_s

Parameters

- **pos_5** – The 5' position of the move (positive for insertions, negative for removal, any value for shift moves)
- **pos_3** – The 3' position of the move (positive for insertions, negative for removal, any value for shift moves)

Returns
An atomic move as specified by `pos_5` and `pos_3`

void **vrna_move_list_free**(*vrna_move_t* *moves)

```
#include <ViennaRNA/landscape/move.h>
```

delete all moves in a zero terminated list.

void **vrna_move_apply**(short *pt, const *vrna_move_t* *m)

Apply a particular move / transition to a secondary structure, i.e. transform a structure.

```
#include <ViennaRNA/landscape/move.h>
```

Parameters

- **pt** – **[inout]** The pair table representation of the secondary structure
- **m** – **[in]** The move to apply

void **vrna_move_apply_db**(char *structure, const short *pt, const *vrna_move_t* *m)

```
#include <ViennaRNA/landscape/move.h>
```

int **vrna_move_is_removal**(const *vrna_move_t* *m)

Test whether a move is a base pair removal.

```
#include <ViennaRNA/landscape/move.h>
```

Parameters

- **m** – The move to test against

Returns

Non-zero if the move is a base pair removal, 0 otherwise

int **vrna_move_is_insertion**(const *vrna_move_t* *m)

Test whether a move is a base pair insertion.

```
#include <ViennaRNA/landscape/move.h>
```

Parameters

- **m** – The move to test against

Returns

Non-zero if the move is a base pair insertion, 0 otherwise

int **vrna_move_is_shift**(const *vrna_move_t* *m)

Test whether a move is a base pair shift.

```
#include <ViennaRNA/landscape/move.h>
```

Parameters

- **m** – The move to test against

Returns

Non-zero if the move is a base pair shift, 0 otherwise

int **vrna_move_compare**(const *vrna_move_t* *m, const *vrna_move_t* *b, const short *pt)

Compare two moves.

```
#include <ViennaRNA/landscape/move.h>
```

The function compares two moves `m` and `b` and returns whether move `m` is lexicographically smaller (-1), larger (1) or equal to move `b`.

If any of the moves `m` or `b` is a shift move, this comparison only makes sense in a structure context. Thus, the third argument with the current structure must be provided.

Note

This function returns 0 (equality) upon any error, e.g. missing input

Warning

Currently, shift moves are not supported!

Parameters

- **m** – The first move of the comparison
- **b** – The second move of the comparison
- **pt** – The pair table of the current structure that is compatible with both moves (maybe NULL if moves are guaranteed to be no shifts)

Returns

-1 if m < b, 1 if m > b, 0 otherwise

void **vrna_loopidx_update**(int *loopidx, const short *pt, int length, const *vrna_move_t* *m)

Alters the loopIndices array that was constructed with *vrna_loopidx_from_ptable()*.

```
#include <ViennaRNA/landscape/neighbor.h>
```

The loopIndex of the current move will be inserted. The correctness of the input will not be checked because the speed should be optimized.

Parameters

- **loopidx** – **[inout]** The loop index data structure that needs an update
- **pt** – **[in]** A pair table on which the move will be executed
- **length** – The length of the structure
- **m** – **[in]** The move that is applied to the current structure

vrna_move_t ***vrna_neighbors**(*vrna_fold_compound_t* *fc, const short *pt, unsigned int options)

Generate neighbors of a secondary structure.

```
#include <ViennaRNA/landscape/neighbor.h>
```

This function allows one to generate all structural neighbors (according to a particular move set) of an RNA secondary structure. The neighborhood is then returned as a list of transitions / moves required to transform the current structure into the actual neighbor.

SWIG Wrapper Notes:

This function is attached as an overloaded method `neighbors()` to objects of type `fold_compound`. The optional parameter `options` defaults to *VRNA_MOVESET_DEFAULT* if it is omitted. See, e.g. *`RNA.fold_compound.neighbors()`* in the *The RNA Python module*.

See also

vrna_neighbors_successive(), *vrna_move_apply()*, *VRNA_MOVESET_INSERTION*, *VRNA_MOVESET_DELETION*, *VRNA_MOVESET_SHIFT*, *VRNA_MOVESET_DEFAULT*

Parameters

- **fc** – **[in]** A *vrna_fold_compound_t* containing the energy parameters and model details
- **pt** – **[in]** The pair table representation of the structure
- **options** – Options to modify the behavior of this function, e.g. available move set

Returns
Neighbors as a list of moves / transitions (the last element in the list has both of its fields set to 0)

vrna_move_t ***vrna_neighbors_successive**(const *vrna_fold_compound_t* *fc, const *vrna_move_t* *curr_move, const short *prev_pt, const *vrna_move_t* *prev_neighbors, int size_prev_neighbors, int *size_neighbors, unsigned int options)

Generate neighbors of a secondary structure (the fast way).

```
#include <ViennaRNA/landscape/neighbor.h>
```

This function implements a fast way to generate all neighbors of a secondary structure that results from successive applications of individual moves. The speed-up results from updating an already known list of valid neighbors before the individual move towards the current structure took place. In essence, this function removes neighbors that are not accessible anymore and inserts neighbors emerging after a move took place.

See also

vrna_neighbors(), *vrna_move_apply()*, *VRNA_MOVESET_INSERTION*, *VRNA_MOVESET_DELETION*, *VRNA_MOVESET_SHIFT*, *VRNA_MOVESET_DEFAULT*

Parameters

- **fc** – **[in]** A *vrna_fold_compound_t* containing the energy parameters and model details
- **curr_move** – **[in]** The move that was/will be applied to `prev_pt`
- **prev_pt** – **[in]** A pair table representation of the structure before `curr_move` is/was applied
- **prev_neighbors** – **[in]** The list of neighbors of `prev_pt`
- **size_prev_neighbors** – The size of `prev_neighbors`, i.e. the lists length
- **size_neighbors** – **[out]** A pointer to store the size / length of the new neighbor list
- **options** – Options to modify the behavior of this function, e.g. available move set

Returns
Neighbors as a list of moves / transitions (the last element in the list has both of its fields set to 0)

int **vrna_move_neighbor_diff_cb**(*vrna_fold_compound_t* *fc, short *ptable, *vrna_move_t* move, *vrna_move_update_f* cb, void *data, unsigned int options)

Apply a move to a secondary structure and indicate which neighbors have changed consequentially.

```
#include <ViennaRNA/landscape/neighbor.h>
```

This function applies a move to a secondary structure and explores the local neighborhood of the affected loop. Any changes to previously compatible neighbors that have been affected by this loop will be reported through a callback function. In particular, any of the three cases might appear:

- A previously available neighbor move has changed, usually the free energy change of the move (*VRNA_NEIGHBOR_CHANGE*)
- A previously available neighbor move became invalid (*VRNA_NEIGHBOR_INVALID*)
- A new neighbor move becomes available (*VRNA_NEIGHBOR_NEW*)

See also

vrna_move_neighbor_diff(), *VRNA_NEIGHBOR_CHANGE*, *VRNA_NEIGHBOR_INVALID*, *VRNA_NEIGHBOR_NEW*, *vrna_move_update_f*, #VRNA_MOVE_NO_APPLY

Parameters

- **fc** – A fold compound for the RNA sequence(s) that this function operates on
- **ptable** – The current structure as pair table
- **move** – The move to apply
- **cb** – The address of the callback function that is passed the neighborhood changes
- **data** – An arbitrary data pointer that will be passed through to the callback function cb
- **options** – Options to modify the behavior of this function, .e.g available move set

Returns

Non-zero on success, 0 otherwise

vrna_move_t ***vrna_move_neighbor_diff**(*vrna_fold_compound_t* *fc, short *ptable, *vrna_move_t* move, *vrna_move_t* **invalid_moves, unsigned int options)

Apply a move to a secondary structure and indicate which neighbors have changed consequentially.

```
#include <ViennaRNA/landscape/neighbor.h>
```

Similar to *vrna_move_neighbor_diff_cb()*, this function applies a move to a secondary structure and reports back the neighbors of the current structure become affected by this move. Instead of executing a callback for each of the affected neighbors, this function compiles two lists of neighbor moves, one that is returned and consists of all moves that are novel or may have changed in energy, and a second, `invalid_moves`, that consists of all the neighbor moves that become invalid, respectively.

Parameters

- **fc** – A fold compound for the RNA sequence(s) that this function operates on
- **ptable** – The current structure as pair table
- **move** – The move to apply
- **invalid_moves** – The address of a move list where the function stores those moves that become invalid
- **options** – Options to modify the behavior of this function, .e.g available move set

Returns

A list of moves that might have changed in energy or are novel compared to the structure before application of the move

struct **vrna_move_s**

#include <ViennaRNA/landscape/move.h> An atomic representation of the transition / move from one structure to its neighbor.

An atomic transition / move may be one of the following:

- a **base pair insertion**,
- a **base pair removal**, or
- a **base pair shift** where an existing base pair changes one of its pairing partner.

These moves are encoded by two integer values that represent the affected 5' and 3' nucleotide positions. Furthermore, we use the following convention on the signedness of these encodings:

- both values are positive for *insertion moves*
- both values are negative for *base pair removals*
- both values have different signedness for *shift moves*, where the positive value indicates the nucleotide that stays constant, and the others absolute value is the new pairing partner

Note

A value of 0 in either field is used as list-end indicator and doesn't represent any valid move.

Public Members

int **pos_5**

The (absolute value of the) 5' position of a base pair, or any position of a shifted pair.

```
#include <ViennaRNA/landscape/move.h>
```

int **pos_3**

The (absolute value of the) 3' position of a base pair, or any position of a shifted pair.

```
#include <ViennaRNA/landscape/move.h>
```

vrna_move_t ***next**

The next base pair (if an elementary move changes more than one base pair), or `NULL` Has to be terminated with move 0,0.

```
#include <ViennaRNA/landscape/move.h>
```

Folding Paths

API for various RNA (re-)folding path algorithms that also allow for determination of saddle points, energy barriers, and local minima.

Defines

VRNA_PATH_TYPE_DOT_BRACKET

Flag to indicate producing a (re-)folding path as list of dot-bracket structures.

```
#include <ViennaRNA/landscape/paths.h>
```

See also

vrna_path_t, *vrna_path_options_findpath()*, *vrna_path_direct()*, *vrna_path_direct_ub()*

VRNA_PATH_TYPE_MOVES

Flag to indicate producing a (re-)folding path as list of transition moves.

```
#include <ViennaRNA/landscape/paths.h>
```

See also

vrna_path_t, *vrna_path_options_findpath()*, *vrna_path_direct()*, *vrna_path_direct_ub()*

Typedefs

typedef struct *vrna_path_s* **vrna_path_t**

Typename for the refolding path data structure *vrna_path_s*.

```
#include <ViennaRNA/landscape/paths.h>
```

typedef struct vrna_path_options_s ***vrna_path_options_t**

Options data structure for (re-)folding path implementations.

```
#include <ViennaRNA/landscape/paths.h>
```

Functions

void **vrna_path_free**(*vrna_path_t* *path)

Release (free) memory occupied by a (re-)folding path.

```
#include <ViennaRNA/landscape/paths.h>
```

See also

vrna_path_direct(), *vrna_path_direct_ub()*, *vrna_path_findpath()*, *vrna_path_findpath_ub()*

Parameters

- **path** – The refolding path to be free'd

void **vrna_path_options_free**(*vrna_path_options_t* options)

Release (free) memory occupied by an options data structure for (re-)folding path implementations.

```
#include <ViennaRNA/landscape/paths.h>
```

See also

vrna_path_options_findpath(), *vrna_path_direct()*, *vrna_path_direct_ub()*

Parameters

- **options** – The options data structure to be free'd

struct **vrna_path_s**

#include <ViennaRNA/landscape/paths.h> An element of a refolding path list.

Usually, one has to deal with an array of *vrna_path_s*, e.g. returned from one of the refolding-path algorithms.

Since in most cases the length of the list is not known in advance, such lists have an *end-of-list* marker, which is either:

- a value of *NULL* for *vrna_path_s::s* if *vrna_path_s::type* = *VRNA_PATH_TYPE_DOT_BRACKET*, or
- a *vrna_path_s::move* with zero in both fields *vrna_move_t::pos_5* and *vrna_move_t::pos_3* if *vrna_path_s::type* = *VRNA_PATH_TYPE_MOVES*.

In the following we show an example for how to cover both cases of iteration:

```
vrna_path_t *ptr = path; // path was returned from one of the refolding path␣
↪functions, e.g. vrna_path_direct()

if (ptr) {
if (ptr->type == VRNA_PATH_TYPE_DOT_BRACKET) {
for (; ptr->s; ptr++)
      printf("%s [%6.2f]\n", ptr->s, ptr->en);
  } else if (ptr->type == VRNA_PATH_TYPE_MOVES) {
for (; ptr->move.pos_5 != 0; ptr++)
      printf("move %d:%d, dG = %6.2f\n", ptr->move.pos_5, ptr->move.pos_3,␣
↪ptr->en);
  }
}
```

See also

vrna_path_free()

Public Members

unsigned int **type**

The type of the path element.

```
#include <ViennaRNA/landscape/paths.h>
```

A value of *VRNA_PATH_TYPE_DOT_BRACKET* indicates that *vrna_path_s::s* consists of the secondary structure in dot-bracket notation, and *vrna_path_s::en* the corresponding free energy.

On the other hand, if the value is *VRNA_PATH_TYPE_MOVES*, *vrna_path_s::s* is *NULL* and *vrna_path_s::move* is set to the transition move that transforms a previous structure into it's neighbor along the path. In this case, the attribute *vrna_path_s::en* states the change in free energy with respect to the structure before application of *vrna_path_s::move*.

double **en**

Free energy of current structure.

```
#include <ViennaRNA/landscape/paths.h>
```

char *s

Secondary structure in dot-bracket notation.

```
#include <ViennaRNA/landscape/paths.h>
```

vrna_move_t move

Move that transforms the previous structure into it's next neighbor along the path.

```
#include <ViennaRNA/landscape/paths.h>
```

Direct Refolding Paths

Heuristics to explore direct, optimal (re-)folding paths between two secondary structures.

Functions

int **vrna_path_findpath_saddle**(*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int width)

Find energy of a saddle point between 2 structures (search only direct path).

```
#include <ViennaRNA/landscape/findpath.h>
```

This function uses an inplementation of the *findpath* algorithm [Flamm *et al.*, 2001] for near-optimal direct refolding path prediction.

Model details, and energy parameters are used as provided via the parameter 'fc'. The *vrna_fold_compound_t* does not require memory for any DP matrices, but requires all most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_DEFAULT);
```

SWIG Wrapper Notes:

This function is attached as an overloaded method `path_findpath_saddle()` to objects of type `fold_compound`. The optional parameter `width` defaults to 1 if it is omitted. See, e.g. *`RNA.fold_compound.path_findpath_saddle()`* in the *The RNA Python module*.

See also

vrna_path_findpath_saddle_ub(), *vrna_fold_compound()*, *vrna_fold_compound_t*, *vrna_path_findpath()*

Parameters

- **fc** – The *vrna_fold_compound_t* with precomputed sequence encoding and model details
- **s1** – The start structure in dot-bracket notation
- **s2** – The target structure in dot-bracket notation
- **width** – A number specifying how many strutures are being kept at each step during the search

Returns

The saddle energy in 10cal/mol

int **vrna_path_findpath_saddle_ub**(*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int width, int maxE)

Find energy of a saddle point between 2 structures (search only direct path).

```
#include <ViennaRNA/landscape/findpath.h>
```

This function uses an inplementation of the *findpath* algorithm [Flamm *et al.*, 2001] for near-optimal direct refolding path prediction.

Model details, and energy parameters are used as provided via the parameter 'fc'. The *vrna_fold_compound_t* does not require memory for any DP matrices, but requires all most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_DEFAULT);
```

SWIG Wrapper Notes:

This function is attached as an overloaded method `path_findpath_saddle()` to objects of type `fold_compound`. The optional parameter `width` defaults to 1 if it is omitted, while the optional parameter `maxE` defaults to INF. In case the function did not find a path with $E_{saddle} < E_{max}$ the function returns a `NULL` object, i.e. `undef` for Perl and `None` for Python. See, e.g. *RNA.fold_compound.path_findpath_saddle()* in the *The RNA Python module*.

See also

vrna_path_findpath_saddle(), *vrna_fold_compound()*, *vrna_fold_compound_t*, *vrna_path_findpath()*

Warning

The argument `maxE` (E_{max}) enables one to specify an upper bound, or maximum free energy for the saddle point between the two input structures. If no path with $E_{saddle} < E_{max}$ is found, the function simply returns `maxE`

Parameters

- **fc** – The *vrna_fold_compound_t* with precomputed sequence encoding and model details
- **s1** – The start structure in dot-bracket notation
- **s2** – The target structure in dot-bracket notation
- **width** – A number specifying how many strutures are being kept at each step during the search
- **maxE** – An upper bound for the saddle point energy in 10cal/mol

Returns

The saddle energy in 10cal/mol

vrna_path_t ***vrna_path_findpath**(*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int width)

Find refolding path between 2 structures (search only direct path).

```
#include <ViennaRNA/landscape/findpath.h>
```

This function uses an inplementation of the *findpath* algorithm [Flamm *et al.*, 2001] for near-optimal direct refolding path prediction.

Model details, and energy parameters are used as provided via the parameter 'fc'. The *vrna_fold_compound_t* does not require memory for any DP matrices, but requires all most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_DEFAULT);
```

SWIG Wrapper Notes:

This function is attached as an overloaded method `path_findpath()` to objects of type `fold_compound`. The optional parameter `width` defaults to 1 if it is omitted. See, e.g. *`RNA.fold_compound.path_findpath()`* in the *The RNA Python module*.

See also

vrna_path_findpath_ub(), *vrna_fold_compound()*, *vrna_fold_compound_t*, *vrna_path_findpath_saddle()*

Parameters

- **fc** – The *vrna_fold_compound_t* with precomputed sequence encoding and model details
- **s1** – The start structure in dot-bracket notation
- **s2** – The target structure in dot-bracket notation
- **width** – A number specifying how many strutures are being kept at each step during the search

Returns

The saddle energy in 10cal/mol

vrna_path_t ***vrna_path_findpath_ub**(*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int width, int maxE)

Find refolding path between 2 structures (search only direct path).

```
#include <ViennaRNA/landscape/findpath.h>
```

This function uses an inplementation of the *findpath* algorithm [Flamm *et al.*, 2001] for near-optimal direct refolding path prediction.

Model details, and energy parameters are used as provided via the parameter 'fc'. The *vrna_fold_compound_t* does not require memory for any DP matrices, but requires all most basic init values as one would get from a call like this:

```
fc = vrna_fold_compound(sequence, NULL, VRNA_OPTION_DEFAULT);
```

SWIG Wrapper Notes:

This function is attached as an overloaded method `path_findpath()` to objects of type `fold_compound`. The optional parameter `width` defaults to 1 if it is omitted, while the optional parameter `maxE` defaults to INF. In case the function did not find a path with $E_{saddle} < E_{max}$ the function returns an empty list. See, e.g. *`RNA.fold_compound.path_findpath()`* in the *The RNA Python module*.

See also

vrna_path_findpath(), *vrna_fold_compound()*, *vrna_fold_compound_t*, *vrna_path_findpath_saddle()*

Warning

The argument `maxE` enables one to specify an upper bound, or maximum free energy for the saddle point between the two input structures. If no path with $E_{saddle} < E_{max}$ is found, the function simply returns *NULL*

Parameters

- **fc** – The *vrna_fold_compound_t* with precomputed sequence encoding and model details
- **s1** – The start structure in dot-bracket notation
- **s2** – The target structure in dot-bracket notation
- **width** – A number specifying how many strutures are being kept at each step during the search
- **maxE** – An upper bound for the saddle point energy in 10cal/mol

Returns

The saddle energy in 10cal/mol

vrna_path_options_t **vrna_path_options_findpath**(int width, unsigned int type)

Create options data structure for findpath direct (re-)folding path heuristic.

```
#include <ViennaRNA/landscape/paths.h>
```

This function returns an options data structure that switches the *vrna_path_direct()* and *vrna_path_direct_ub()* API functions to use the *findpath* [Flamm *et al.*, 2001] heuristic. The parameter `width` specifies the width of the breadth-first search while the second parameter `type` allows one to set the type of the returned (re-)folding path.

Currently, the following return types are available:

- A list of dot-bracket structures and corresponding free energy (flag: *VRNA_PATH_TYPE_DOT_BRACKET*)
- A list of transition moves and corresponding free energy changes (flag: *VRNA_PATH_TYPE_MOVES*)

SWIG Wrapper Notes:

This function is available as overloaded function `path_options_findpath()`. The optional parameter `width` defaults to 10 if omitted, while the optional parameter `type` defaults to *VRNA_PATH_TYPE_DOT_BRACKET*. See, e.g. *RNA.path_options_findpath()* in the *The RNA Python module*.

See also

VRNA_PATH_TYPE_DOT_BRACKET, *VRNA_PATH_TYPE_MOVES*, *vrna_path_options_free()*, *vrna_path_direct()*, *vrna_path_direct_ub()*

Parameters

- **width** – Width of the breath-first search strategy
- **type** – Setting that specifies how the return (re-)folding path should be encoded

Returns
An options data structure with settings for the findpath direct path heuristic

vrna_path_t ***vrna_path_direct**(*vrna_fold_compound_t* *fc, const char *s1, const char *s2, *vrna_path_options_t* options)

Determine an optimal direct (re-)folding path between two secondary structures.

```
#include <ViennaRNA/landscape/paths.h>
```

This is the generic wrapper function to retrieve (an optimal) (re-)folding path between two secondary structures `s1` and `s2`. The actual algorithm that is used to generate the (re-)folding path is determined by the settings specified in the `options` data structure. This data structure also determines the return type, which might be either:

- a list of dot-bracket structures with corresponding free energy, or
- a list of transition moves with corresponding free energy change

If the `options` parameter is passed a *NULL* pointer, this function defaults to the *findpath heuristic* [Flamm *et al.*, 2001] with a breadth-first search width of 10, and the returned path consists of dot-bracket structures with corresponding free energies.

SWIG Wrapper Notes:

This function is attached as an overloaded method `path_direct()` to objects of type `fold_compound`. The optional parameter `options` defaults to `NULL` if it is omitted. See, e.g. *`RNA.fold_compound.path_direct()`* in the *The RNA Python module*.

See also

vrna_path_direct_ub(), *vrna_path_options_findpath()*, *vrna_path_options_free()*, *vrna_path_free()*

Parameters

- **fc** – The *vrna_fold_compound_t* with precomputed sequence encoding and model details
- **s1** – The start structure in dot-bracket notation
- **s2** – The target structure in dot-bracket notation
- **options** – An options data structure that specifies the path heuristic and corresponding settings (maybe *NULL*)

Returns
An optimal (re-)folding path between the two input structures

vrna_path_t ***vrna_path_direct_ub**(*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int maxE, *vrna_path_options_t* options)

Determine an optimal direct (re-)folding path between two secondary structures.

```
#include <ViennaRNA/landscape/paths.h>
```

This function is similar to *vrna_path_direct()*, but allows to specify an *upper-bound* for the saddle point energy. The underlying algorithms will stop determining an (optimal) (re-)folding path, if none can be found that has a saddle point below the specified upper-bound threshold `maxE`.

SWIG Wrapper Notes:

This function is attached as an overloaded method `path_direct()` to objects of type `fold_compound`. The optional parameter `maxE` defaults to #INT_MAX - 1 if it is omitted, while the optional parameter `options` defaults to `NULL`. In case the function did not find a path with $E_{saddle} < E_{max}$ it returns an empty list. See, e.g. *`RNA.fold_compound.path_direct()`* in the *The RNA Python module*.

See also

vrna_path_direct_ub(), *vrna_path_options_findpath()*, *vrna_path_options_free()*, *vrna_path_free()*

Warning

The argument `maxE` enables one to specify an upper bound, or maximum free energy for the saddle point between the two input structures. If no path with $E_{saddle} < E_{max}$ is found, the function simply returns *NULL*

Parameters

- **`fc`** – The *vrna_fold_compound_t* with precomputed sequence encoding and model details
- **`s1`** – The start structure in dot-bracket notation
- **`s2`** – The target structure in dot-bracket notation
- **`maxE`** – Upper bound for the saddle point along the (re-)folding path
- **`options`** – An options data structure that specifies the path heuristic and corresponding settings (maybe *NULL*)

Returns

An optimal (re-)folding path between the two input structures

Other Folding Paths

Implementation of gradient- and random walks starting from a single secondary structure.

Defines

VRNA_PATH_STEEPEST_DESCENT

Option flag to request a steepest descent / gradient path.

```
#include <ViennaRNA/landscape/walk.h>
```

See also

vrna_path()

VRNA_PATH_RANDOM

Option flag to request a random walk path.

```
#include <ViennaRNA/landscape/walk.h>
```

See also

vrna_path()

VRNA_PATH_NO_TRANSITION_OUTPUT

Option flag to omit returning the transition path.

```
#include <ViennaRNA/landscape/walk.h>
```

See also

vrna_path(), *vrna_path_gradient()*, *vrna_path_random()*

VRNA_PATH_DEFAULT

Option flag to request defaults (steepest descent / default move set).

```
#include <ViennaRNA/landscape/walk.h>
```

See also

vrna_path(), *VRNA_PATH_STEEPEST_DESCENT*, *VRNA_MOVESET_DEFAULT*

Functions

vrna_move_t ***vrna_path**(*vrna_fold_compound_t* *fc, short *pt, unsigned int steps, unsigned int options)

Compute a path, store the final structure, and return a list of transition moves from the start to the final structure.

```
#include <ViennaRNA/landscape/walk.h>
```

This function computes, given a start structure in pair table format, a transition path, updates the pair table to the final structure of the path. Finally, if not requested otherwise by using the *VRNA_PATH_NO_TRANSITION_OUTPUT* flag in the `options` field, this function returns a list of individual transitions that lead from the start to the final structure if requested.

The currently available transition paths are

- Steepest Descent / Gradient walk (flag: *VRNA_PATH_STEEPEST_DESCENT*)
- Random walk (flag: *VRNA_PATH_RANDOM*)

The type of transitions must be set through the `options` parameter

SWIG Wrapper Notes:

This function is attached as an overloaded method `path()` to objects of type `fold_compound`. The optional parameter `options` defaults to *VRNA_PATH_DEFAULT* if it is omitted. See, e.g. *`RNA.fold_compound.path()`* in the *The RNA Python module*.

See also

vrna_path_gradient(), *vrna_path_random()*, *vrna_ptable()*, *vrna_ptable_copy()*, *vrna_fold_compound()* *VRNA_PATH_STEEPEST_DESCENT*, *VRNA_PATH_RANDOM*, *VRNA_MOVESET_DEFAULT*, *VRNA_MOVESET_SHIFT*, *VRNA_PATH_NO_TRANSITION_OUTPUT*

Note

Since the result is written to the input structure you may want to use *vrna_ptable_copy()* before calling this function to keep the initial structure

Parameters

- **`fc`** – **[in]** A *vrna_fold_compound_t* containing the energy parameters and model details
- **`pt`** – **[inout]** The pair table containing the start structure. Used to update to the final structure after execution of this function
- **`options`** – **[in]** Options to modify the behavior of this function

Returns

A list of transition moves (default), or NULL (if options & *VRNA_PATH_NO_TRANSITION_OUTPUT*)

vrna_move_t ***vrna_path_gradient**(*vrna_fold_compound_t* *fc, short *pt, unsigned int options)

Compute a steepest descent / gradient path, store the final structure, and return a list of transition moves from the start to the final structure.

```
#include <ViennaRNA/landscape/walk.h>
```

This function computes, given a start structure in pair table format, a steepest descent path, updates the pair table to the final structure of the path. Finally, if not requested otherwise by using the *VRNA_PATH_NO_TRANSITION_OUTPUT* flag in the `options` field, this function returns a list of individual transitions that lead from the start to the final structure if requested.

SWIG Wrapper Notes:

This function is attached as an overloaded method `path_gradient()` to objects of type `fold_compound`. The optional parameter `options` defaults to *VRNA_PATH_DEFAULT* if it is omitted. See, e.g. *`RNA.fold_compound.path_gradient()`* in the *The RNA Python module*.

See also

vrna_path_random(), *vrna_path()*, *vrna_ptable()*, *vrna_ptable_copy()*, *vrna_fold_compound()* *VRNA_MOVESET_DEFAULT*, *VRNA_MOVESET_SHIFT*, *VRNA_PATH_NO_TRANSITION_OUTPUT*

Note

Since the result is written to the input structure you may want to use *vrna_ptable_copy()* before calling this function to keep the initial structure

Parameters

- **fc** – **[in]** A *vrna_fold_compound_t* containing the energy parameters and model details
- **pt** – **[inout]** The pair table containing the start structure. Used to update to the final structure after execution of this function
- **options** – **[in]** Options to modify the behavior of this function

Returns

A list of transition moves (default), or NULL (if options & *VRNA_PATH_NO_TRANSITION_OUTPUT*)

vrna_move_t ***vrna_path_random**(*vrna_fold_compound_t* *fc, short *pt, unsigned int steps, unsigned int options)

Generate a random walk / path of a given length, store the final structure, and return a list of transition moves from the start to the final structure.

```
#include <ViennaRNA/landscape/walk.h>
```

This function generates, given a start structure in pair table format, a random walk / path, updates the pair table to the final structure of the path. Finally, if not requested otherwise by using the *VRNA_PATH_NO_TRANSITION_OUTPUT* flag in the `options` field, this function returns a list of individual transitions that lead from the start to the final structure if requested.

SWIG Wrapper Notes:

This function is attached as an overloaded method `path_gradient()` to objects of type `fold_compound`. The optional parameter `options` defaults to *VRNA_PATH_DEFAULT* if it is omitted. See, e.g. *`RNA.fold_compound.path_random()`* in the *The RNA Python module*.

See also

vrna_path_gradient(), *vrna_path()*, *vrna_ptable()*, *vrna_ptable_copy()*, *vrna_fold_compound()* *VRNA_MOVESET_DEFAULT*, *VRNA_MOVESET_SHIFT*, *VRNA_PATH_NO_TRANSITION_OUTPUT*

Note

Since the result is written to the input structure you may want to use *vrna_ptable_copy()* before calling this function to keep the initial structure

Parameters

- **fc** – **[in]** A *vrna_fold_compound_t* containing the energy parameters and model details
- **pt** – **[inout]** The pair table containing the start structure. Used to update to the final structure after execution of this function
- **steps** – **[in]** The length of the path, i.e. the total number of transitions / moves
- **options** – **[in]** Options to modify the behavior of this function

Returns

A list of transition moves (default), or NULL (if options & *VRNA_PATH_NO_TRANSITION_OUTPUT*)

4.1.9 Experimental Structure Probing Data

In the following, you'll find the respective API symbols that allow for the integration of experimental probing data.

Generic Probing Data API

Include Experimental Structure Probing Data to Guide Structure Predictions.

Defines

VRNA_REACTIVITY_MISSING

```
#include <ViennaRNA/probing/basic.h>
```

Typedefs

typedef struct vrna_probing_data_s ***vrna_probing_data_t**

A data structure that contains RNA structure probing data and specifies how this data is to be integrated into structure predictions.

```
#include <ViennaRNA/probing/basic.h>
```

See also

vrna_probing_data_deigan(), *vrna_probing_data_deigan_comparative()*, *vrna_probing_data_zarringhalam()*, *vrna_probing_data_zarringhalam_comparative()*, *vrna_sc_probing()*, *vrna_probing_data_free()*

Functions

int **vrna_sc_probing**(*vrna_fold_compound_t* *fc, *vrna_probing_data_t* data)

Apply probing data (e.g. SHAPE) to guide the structure prediction.

```
#include <ViennaRNA/probing/basic.h>
```

SWIG Wrapper Notes:

This function is attached as method **sc_probing()** to objects of type **fold_compound**. See, e.g. *RNA.fold_compound.sc_probing()* in the *The RNA Python module* .

See also

vrna_probing_data_t, *vrna_probing_data_free()*, *vrna_probing_data_deigan()*, *vrna_probing_data_deigan_comparative()*, *vrna_probing_data_zarringhalam()*, *vrna_probing_data_zarringhalam_comparative()*, *vrna_probing_data_eddy()*, *vrna_probing_data_eddy_comparative()*

Parameters

- **fc** – The *vrna_fold_compound_t* the probing data should be applied to in subsequent computations

- **data** – The prepared probing data and probing data integration strategy

Returns
The number of probing data sets applied, 0 upon any error

vrna_probing_data_t **vrna_probing_data_linear**(const double *data, unsigned int data_length, const double *data_weights, *vrna_probing_strategy_f* strategy_cb, void *strategy_cb_options, *vrna_auxdata_free_f* strategy_cb_options_free, unsigned int options)

```
#include <ViennaRNA/probing/basic.h>
```

vrna_probing_data_t **vrna_probing_data_linear_multi**(const double **data, unsigned int data_size, const unsigned int *data_lengths, const double **data_weights, *vrna_probing_strategy_f* *strategy_cbs, void **strategy_cbs_options, *vrna_auxdata_free_f* *strategy_cbs_options_free, unsigned int options)

```
#include <ViennaRNA/probing/basic.h>
```

void **vrna_probing_data_free**(*vrna_probing_data_t* d)

Free memory occupied by the (prepared) probing data.

```
#include <ViennaRNA/probing/basic.h>
```

See also

vrna_probing_data_t, *vrna_sc_probing()*, *vrna_probing_data_deigan()*, *vrna_probing_data_deigan_comparative()*, *vrna_probing_data_zarringhalam()*, *vrna_probing_data_zarringhalam_comparative()*, *vrna_probing_data_eddy()*, *vrna_probing_data_eddy_comparative()*

unsigned int **vrna_probing_data_linear_num**(struct vrna_probing_data_s *data)

Get the number of structure probing data sets.

```
#include <ViennaRNA/probing/basic.h>
```

See also

vrna_probing_data_linear(), vrna_probing_data_linear_multi()

Parameters

- **data** – The data structure storing the probing data

Returns
The number of probing data sets stored in `data`

double *vrna_probing_data_linear_raw(struct vrna_probing_data_s *data, unsigned int pos, unsigned int *data_size)

Get the raw probing data.

```
#include <ViennaRNA/probing/basic.h>
```

This function retrieves a copy of the structure probing data set number `pos` as stored in `data`. The number of values returned by this function is stored in `data_size`.

See also

vrna_probing_data_linear(), vrna_probing_data_linear_multi(), *vrna_probing_data_linear_num()*, *vrna_probing_data_linear_weight()*, *vrna_probing_data_linear_energies()*

Parameters

- **data** – The data structure storing the probing data
- **pos** – The position of the data set
- **data_size** – A pointer to a variable to store the number of returned values to

Returns

A copy of the `pos` -th probing data set stored in `data`

double *vrna_probing_data_linear_weight(struct vrna_probing_data_s *data, unsigned int pos, unsigned int *data_size)

Get the weights for a probing data set.

```
#include <ViennaRNA/probing/basic.h>
```

See also

vrna_probing_data_linear(), vrna_probing_data_linear_multi(), *vrna_probing_data_linear_num()*, *vrna_probing_data_linear_raw()*, *vrna_probing_data_linear_energies()*

Parameters

- **data** – The data structure storing the probing data
- **pos** – The position of the weights vector
- **data_size** – A pointer to a variable to store the number of returned values to

Returns

A copy of the `pos` -th weighting vector stored in `data`

double *vrna_probing_data_linear_energies(struct vrna_probing_data_s *data, unsigned int pos, *vrna_fold_compound_t* *fc, unsigned int target, unsigned int *data_size)

Get pseudo energy contributions from a structure probing data set.

```
#include <ViennaRNA/probing/basic.h>
```

This function retrieves a vector of pseudo energies derived from a set of structure probing data stored in `data`. For that, the function calls the probing data strategy that is associated with `data`.

See also

vrna_probing_data_linear(), vrna_probing_data_linear_multi(), *vrna_probing_data_linear_num()*, *vrna_probing_data_linear_raw()*, *vrna_probing_data_linear_weight()*, *VRNA_PROBING_DATA_LINEAR_TARGET_STACK*, *VRNA_PROBING_DATA_LINEAR_TARGET_UP*, *VRNA_PROBING_DATA_LINEAR_TARGET_BP*

Parameters

- **data** – The data structure storing the probing data
- **pos** – The position of the weights vector
- **fc** – The fold_compound that will be passed through to the conversion strategy
- **target** – The target, i.e. the structure context the pseudo energies should be applied to
- **data_size** – A pointer to a variable to store the number of returned values to

Returns

An array of pseudo energies (1-based) or **NULL**

double **vrna_probing_data_load_n_distribute(unsigned int n_seq, unsigned int *ns, const char **sequences, const char **file_names, const int *file_name_association, unsigned int options)

```
#include <ViennaRNA/probing/basic.h>
```

vrna_probing_data_t vrna_probing_data_Deigan2009(const double *reactivities, unsigned int n, double m, double b)

```
#include <ViennaRNA/probing/basic.h>
```

SWIG Wrapper Notes:

This function exists in two forms, (i) as overloaded function **probing_data_Deigan2009()** and (ii) as constructor of the **probing_data** object. For the former the second argument `n` can be omitted since the length of the `reactivities` list is determined from the list itself. When the *vrna_probing_data_t* constructor is called with the three parameters `reactivities`, `m` and `b`, it will automatically create a prepared data structure for the Deigan et al. 2009 method. See, e.g. `RNA.probing_data_Deigan2009()` and *RNA.probing_data()* in the *The RNA Python module* .

vrna_probing_data_t vrna_probing_data_Deigan2009_comparative(const double **reactivities, const unsigned int *n, unsigned int n_seq, double *ms, double *bs, unsigned int multi_params)

```
#include <ViennaRNA/probing/basic.h>
```

vrna_probing_data_t vrna_probing_data_Zarringhalam2012(const double *reactivities, unsigned int n, double beta, const char *pr_conversion, double pr_default)

```
#include <ViennaRNA/probing/basic.h>
```

vrna_probing_data_t **vrna_probing_data_Zarringhalam2012_comparative**(const double **reactivities, unsigned int *n, unsigned int n_seq, double *betas, const char **pr_conversions, double *pr_defaults, unsigned int multi_params)

```
#include <ViennaRNA/probing/basic.h>
```

vrna_probing_data_t **vrna_probing_data_Eddy2014_2**(const double *reactivities, unsigned int n, const double *unpaired_data, unsigned int unpaired_len, const double *paired_data, unsigned int paired_len)

```
#include <ViennaRNA/probing/basic.h>
```

vrna_probing_data_t **vrna_probing_data_Eddy2014_2_comparative**(const double **reactivities, unsigned int *n, unsigned int n_seq, const double **unpaired_datas, unsigned int *unpaired_lens, const double **paired_datas, unsigned int *paired_lens, unsigned int multi_params)

```
#include <ViennaRNA/probing/basic.h>
```

int **vrna_sc_SHAPE_parse_method**(const char *method_string, char *method, float *param_1, float *param_2)

Parse a character string and extract the encoded SHAPE reactivity conversion method and possibly the parameters for conversion into pseudo free energies.

```
#include <ViennaRNA/probing/SHAPE.h>
```

Parameters

- **method_string** – The string that contains the encoded SHAPE reactivity conversion method
- **method** – A pointer to the memory location where the method character will be stored
- **param_1** – A pointer to the memory location where the first parameter of the corresponding method will be stored
- **param_2** – A pointer to the memory location where the second parameter of the corresponding method will be stored

Returns

1 on successful extraction of the method, 0 on errors

Strategies for Linear Data

Generic Probing Data Strategy API

Defines

VRNA_PROBING_DATA_WEIGHT_POSITION_WISE

```
#include <ViennaRNA/probing/basic.h>
```

VRNA_PROBING_DATA_SINGLE_STRATEGY

```
#include <ViennaRNA/probing/basic.h>
```

VRNA_PROBING_DATA_SINGLE_WEIGHT

```
#include <ViennaRNA/probing/basic.h>
```

VRNA_PROBING_DATA_DEFAULT

```
#include <ViennaRNA/probing/basic.h>
```

VRNA_PROBING_DATA_LINEAR_TARGET_STACK

Target flag indicating base pair stack contributions for *vrna_probing_strategy_f*.

```
#include <ViennaRNA/probing/basic.h>
```

VRNA_PROBING_DATA_LINEAR_TARGET_UP

Target flag indicating unpaired nucleotide contributions for *vrna_probing_strategy_f*.

```
#include <ViennaRNA/probing/basic.h>
```

VRNA_PROBING_DATA_LINEAR_TARGET_BP

Target flag indicating base pair contributions for *vrna_probing_strategy_f*.

```
#include <ViennaRNA/probing/basic.h>
```

VRNA_PROBING_METHOD_MULTI_PARAMS_0

probing data conversion flag for comparative structure predictions indicating no parameter to be sequence specific

```
#include <ViennaRNA/probing/basic.h>
```

See also

vrna_probing_data_deigan_comparative(), *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_METHOD_MULTI_PARAMS_1*, *VRNA_PROBING_METHOD_MULTI_PARAMS_2*, *VRNA_PROBING_METHOD_MULTI_PARAMS_3*, *VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT*

VRNA_PROBING_METHOD_MULTI_PARAMS_1

probing data conversion flag for comparative structure predictions indicating 1st parameter to be sequence specific

```
#include <ViennaRNA/probing/basic.h>
```

See also

vrna_probing_data_deigan_comparative(), *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_METHOD_MULTI_PARAMS_0*, *VRNA_PROBING_METHOD_MULTI_PARAMS_2*, *VRNA_PROBING_METHOD_MULTI_PARAMS_3*, *VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT*

VRNA_PROBING_METHOD_MULTI_PARAMS_2

probing data conversion flag for comparative structure predictions indicating 2nd parameter to be sequence specific

```
#include <ViennaRNA/probing/basic.h>
```

See also

vrna_probing_data_deigan_comparative(), *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_METHOD_MULTI_PARAMS_0*, *VRNA_PROBING_METHOD_MULTI_PARAMS_1*, *VRNA_PROBING_METHOD_MULTI_PARAMS_3*, *VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT*

VRNA_PROBING_METHOD_MULTI_PARAMS_3

probing data conversion flag for comparative structure predictions indicating 3rd parameter to be sequence specific

```
#include <ViennaRNA/probing/basic.h>
```

See also

vrna_probing_data_deigan_comparative(), *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_METHOD_MULTI_PARAMS_0*, *VRNA_PROBING_METHOD_MULTI_PARAMS_1*, *VRNA_PROBING_METHOD_MULTI_PARAMS_2*, *VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT*

VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT

probing data conversion flag for comparative structure predictions indicating default parameter settings

```
#include <ViennaRNA/probing/basic.h>
```

Essentially, this setting indicates that all probing data is to be converted using the same parameters. Use any combination of *VRNA_PROBING_METHOD_MULTI_PARAMS_1*, *VRNA_PROBING_METHOD_MULTI_PARAMS_2*, *VRNA_PROBING_METHOD_MULTI_PARAMS_3*, and so on to indicate that the first, second, third, or other parameter is sequence specific.

See also

vrna_probing_data_deigan_comparative(), *vrna_probing_data_zarringhalam_comparative()*

Typedefs

typedef void ***vrna_probing_strategy_opt_t**

```
#include <ViennaRNA/probing/basic.h>
```

typedef double *(***vrna_probing_strategy_f**)(*vrna_fold_compound_t* *fc, const double *data, size_t data_size, unsigned int target, void *options)

Prototype of a strategy to derive pseudo energies from linear structure probing data.

```
#include <ViennaRNA/probing/basic.h>
```

Param fc
The fold compound the probing data will be applied to

Param data
The structure probing data (1-based array of reactivity values)

Param data_size
The size of `data`, i.e. the total number of reactivity values

Param target
The structural context for which pseudo energies are requested from the strategy

Param options
An arbitrary data structure the strategy requires for working on the data

Return
A pointer to an array of pseudo energies ready to be included as soft constraints

Functions

double ***vrna_probing_strategy_chain**(*vrna_fold_compound_t* *fc, const double *data, size_t data_size, unsigned int target, *vrna_probing_strategy_opt_t* options)

```
#include <ViennaRNA/probing/basic.h>
```

vrna_probing_strategy_opt_t **vrna_probing_strategy_chain_data**(*vrna_probing_strategy_f* strategy_cb, *vrna_probing_strategy_opt_t* strategy_cb_options, *vrna_auxdata_free_f* strategy_cb_options_free)

```
#include <ViennaRNA/probing/basic.h>
```

size_t **vrna_probing_strategy_chain_data_append**(*vrna_probing_strategy_opt_t* chain, *vrna_probing_strategy_f* strategy_cb, *vrna_probing_strategy_opt_t* strategy_cb_options, *vrna_auxdata_free_f* strategy_cb_options_free)

```
#include <ViennaRNA/probing/basic.h>
```

size_t **vrna_probing_strategy_chain_data_size**(*vrna_probing_strategy_opt_t* chain)

```
#include <ViennaRNA/probing/basic.h>
```

vrna_probing_strategy_f **vrna_probing_strategy_chain_data_at**(*vrna_probing_strategy_opt_t* chain, size_t pos, *vrna_probing_strategy_opt_t* *strategy_cb_options)

```
#include <ViennaRNA/probing/basic.h>
```

void **vrna_probing_strategy_chain_data_free**(*vrna_probing_strategy_opt_t* options)

```
#include <ViennaRNA/probing/basic.h>
```

The Deigan 2009 Strategy API

Defines

VRNA_PROBING_METHOD_DEIGAN2009_DEFAULT_m

Default parameter for slope `m` as used in method of Deigan *et al.* [2009] .

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

See also

vrna_probing_data_deigan(), *vrna_probing_data_deigan_comparative()*, *VRNA_PROBING_METHOD_DEIGAN2009_DEFAULT_b*

VRNA_PROBING_METHOD_DEIGAN2009_DEFAULT_b

Default parameter for intercept `b` as used in method of Deigan *et al.* [2009] .

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

See also

vrna_probing_data_deigan(), *vrna_probing_data_deigan_comparative()*, *VRNA_PROBING_METHOD_DEIGAN2009_DEFAULT_m*

Functions

double ***vrna_probing_strategy_deigan**(*vrna_fold_compound_t* *fc, const double *data, size_t data_size, unsigned int target, void *options)*

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

void ***vrna_probing_strategy_deigan_options**(double m, double b, double max_value, *vrna_math_fun_dbl_f* cb_preprocess, *vrna_math_fun_dbl_opt_t* cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* cb_preprocess_opt_free)*

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

void **vrna_probing_strategy_deigan_options_free**(void *options)

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

vrna_probing_data_t **vrna_probing_data_deigan**(const double *reactivities, unsigned int n, double m, double b)

Prepare probing data according to Deigan et al. 2009 method.

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

Prepares a data structure to be used with *vrna_sc_probing()* to directed RNA folding using the simple linear ansatz

$$\Delta G_{\text{SHAPE}}(i) = m \ln(\text{SHAPE reactivity}(i) + 1) + b$$

to convert probing data, e.g. SHAPE reactivity values, to pseudo energies whenever a nucleotide i contributes to a stacked pair. A positive slope m penalizes high reactivities in paired regions, while a negative intercept b results in a confirmatory *bonus* free energy for correctly predicted base pairs. Since the energy evaluation of a base pair stack involves two pairs, the pseudo energies are added for all four contributing nucleotides. Consequently, the energy term is applied twice for pairs inside a helix and only once for pairs adjacent to other structures. For all other loop types the energy model remains unchanged even when the experimental data highly disagrees with a certain motif.

See also

vrna_probing_data_t, *vrna_probing_data_free()*, *vrna_sc_probing()*, *vrna_probing_data_deigan_comparative()*, *vrna_probing_data_zarringhalam()*, *vrna_probing_data_zarringhalam_comparative()*, *vrna_probing_data_eddy()*, *vrna_probing_data_eddy_comparative()*

Note

For further details, we refer to Deigan *et al.* [2009] .

Parameters

- **reactivities** – 1-based array of per-nucleotide probing data, e.g. SHAPE reactivities

- **n** – The length of the `reactivities` list
- **m** – The slope used for the probing data to soft constraints conversion strategy
- **b** – The intercept used for the probing data to soft constraints conversion strategy

Returns

A pointer to a data structure containing the probing data and any preparations necessary to use it in *vrna_sc_probing()* according to the method of Deigan *et al.* [2009] or **NULL** on any error.

vrna_probing_data_t **vrna_probing_data_deigan_trans**(const double *reactivities, unsigned int n, double m, double b, *vrna_math_fun_dbl_f* cb_preprocess, *vrna_math_fun_dbl_opt_t* cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* cb_preprocess_opt_free)

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

vrna_probing_data_t **vrna_probing_data_deigan_comparative**(const double **reactivities, const unsigned int *n, unsigned int n_seq, double *ms, double *bs, unsigned int multi_params)

Prepare (multiple) probing data according to Deigan et al. 2009 method for comparative structure predictions.

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

Similar to *vrna_probing_data_deigan()*, this function prepares a data structure to be used with *vrna_sc_probing()* to directed RNA folding using the simple linear ansatz

$$\Delta G_{\text{SHAPE}}(i) = m \ln(\text{SHAPE reactivity}(i) + 1) + b$$

to convert probing data, e.g. SHAPE reactivity values, to pseudo energies whenever a nucleotide i contributes to a stacked pair. This functions purpose is to allow for adding multiple probing data as required for comparative structure predictions over multiple sequence alignments (MSA) with `n_seq` sequences. For that purpose, `reactivities` can be provided for any of the sequences in the MSA. Individual probing data is always expected to be specified in sequence coordinates, i.e. without considering gaps in the MSA. Therefore, each set of `reactivities` may have a different length as specified the parameter `n`. In addition, each set of probing data may undergo the conversion using different parameters m and b. Whether or not multiple sets of conversion parameters are provided must be specified using the `multi_params` flag parameter. Use *VRNA_PROBING_METHOD_MULTI_PARAMS_1* to indicate that `ms` points to an array of slopes for each sequence. Along with that, *VRNA_PROBING_METHOD_MULTI_PARAMS_2* indicates that `bs` is pointing to an array of intercepts for each sequence. Bitwise-OR of the two values renders both parameters to be sequence specific.

See also

vrna_probing_data_t, *vrna_probing_data_free()*, *vrna_sc_probing()*, *vrna_probing_data_deigan()*, *vrna_probing_data_zarringhalam()*, *vrna_probing_data_zarringhalam_comparative()*, *vrna_probing_data_eddy()*, *vrna_probing_data_eddy_comparative()*, *VRNA_PROBING_METHOD_MULTI_PARAMS_0*, *VRNA_PROBING_METHOD_MULTI_PARAMS_1*, *VRNA_PROBING_METHOD_MULTI_PARAMS_2*, *VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT*

Note

For further details, we refer to Deigan *et al.* [2009] .

Parameters

- **reactivities** – 0-based array of 1-based arrays of per-nucleotide probing data, e.g. SHAPE reactivities
- **n** – 0-based array of lengths of the `reactivities` lists
- **n_seq** – The number of sequences in the MSA
- **ms** – 0-based array of the slopes used for the probing data to soft constraints conversion strategy or the address of a single slope value to be applied for all data
- **bs** – 0-based array of the intercepts used for the probing data to soft constraints conversion strategy or the address of a single intercept value to be applied for all data
- **multi_params** – A flag indicating what is passed through parameters `ms` and `bs`

Returns

A pointer to a data structure containing the probing data and any preparations necessary to use it in *vrna_sc_probing()* according to the method of Deigan *et al.* [2009] or **NULL** on any error.

vrna_probing_data_t **vrna_probing_data_deigan_trans_comparative**(const double **reactivities, const unsigned int *n, unsigned int n_seq, double *ms, double *bs, unsigned int multi_params, *vrna_math_fun_dbl_f* *cb_preprocess, *vrna_math_fun_dbl_opt_t* *cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* *cb_preprocess_opt_free)

```
#include <ViennaRNA/probing/strategy_deigan.h>
```

The Zarringhalam 2012 Strategy API

Defines

VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_beta

Default parameter `beta` as used in method of Zarringhalam *et al.* [2012] .

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

See also

vrna_probing_data_zarringhalam(), *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_conversion*, *VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_probability*

VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_conversion

Default conversion method of probing data into probabilities as used in method of Zarringhalam *et al.* [2012] .

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

See also

vrna_probing_data_zarringhalam(), *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_beta*, *VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_probability*

VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_probability

Default probability value for missing data in method of Zarringhalam *et al.* [2012] .

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

See also

vrna_probing_data_zarringhalam(), *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_beta*, *VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_conversion*

Functions

double ***vrna_probing_strategy_zarringhalam**(*vrna_fold_compound_t* *fc, const double *data, size_t data_size, unsigned int target, void *options)

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

void ***vrna_probing_strategy_zarringhalam_options**(double beta, double default_probability, double max_value, *vrna_math_fun_dbl_f* cb_preprocess, *vrna_math_fun_dbl_opt_t* cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* cb_preprocess_opt_free)

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

void **vrna_probing_strategy_zarringhalam_options_free**(void *option)

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

vrna_probing_data_t **vrna_probing_data_zarringhalam**(const double *reactivities, unsigned int n, double beta, const char *pr_conversion, double pr_default)

Prepare probing data according to Zarringhalam et al. 2012 method.

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

Prepares a data structure to be used with *vrna_sc_probing()* to directed RNA folding using the method of Zarringhalam *et al.* [2012] .

This method first converts the observed probing data of nucleotide i into a probability q_i that position i is unpaired by means of a non-linear map. Then pseudo-energies of the form

$$\Delta G_{\text{SHAPE}}(x, i) = \beta \, |x_i - q_i|$$

are computed, where $x_i = 0$ if position i is unpaired and $x_i = 1$ if i is paired in a given secondary structure. The parameter β serves as scaling factor. The magnitude of discrepancy between prediction and experimental observation is represented by $|x_i - q_i|$.

See also

vrna_probing_data_t, *vrna_probing_data_free()*, *vrna_sc_probing()*, *vrna_probing_data_zarringhalam_comparative()*, *vrna_probing_data_deigan()*, *vrna_probing_data_deigan_comparative()*, *vrna_probing_data_eddy()*, *vrna_probing_data_eddy_comparative()*

Note

For further details, we refer to Zarringhalam *et al.* [2012]

Parameters

- **reactivities** – 1-based array of per-nucleotide probing data, e.g. SHAPE reactivities
- **n** – The length of the `reactivities` list
- **beta** – The scaling factor β of the conversion function
- **pr_conversion** – A flag that specifies how to convert reactivities to probabilities
- **pr_default** – The default probability for a nucleotide where reactivity data is missing for

Returns

A pointer to a data structure containing the probing data and any preparations necessary to use it in *vrna_sc_probing()* according to the method of Zarringhalam *et al.* [2012] or **NULL** on any error.

vrna_probing_data_t **vrna_probing_data_zarringhalam_trans**(const double *reactivities, unsigned int n, double beta, double pr_default, *vrna_math_fun_dbl_f* cb_preprocess, *vrna_math_fun_dbl_opt_t* cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* cb_preprocess_opt_free)

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

vrna_probing_data_t **vrna_probing_data_zarringhalam_comparative**(const double **reactivities, unsigned int *n, unsigned int n_seq, double *betas, const char **pr_conversions, double *pr_defaults, unsigned int multi_params)

Prepare probing data according to Zarringhalam et al. 2012 method for comparative structure predictions.

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

Similar to *vrna_probing_data_zarringhalam()*, this function prepares a data structure to be used with *vrna_sc_probing()* to guide RNA folding using the method of Zarringhalam *et al.* [2012] .

This functions purpose is to allow for adding multiple probing data as required for comparative structure predictions over multiple sequence alignments (MSA) with `n_seq` sequences. For that purpose, `reactivities` can be provided for any of the sequences in the MSA. Individual probing data is always expected to be specified in sequence coordinates, i.e. without considering gaps in the MSA. Therefore, each set of `reactivities` may have a different length as specified the parameter `n`. In addition, each set of probing data may undergo the conversion using different parameters *beta*. Additionally, the probing data to probability conversions strategy and default values for missing data can be specified in a sequence-based manner. Whether or not multiple conversion parameters are provided must be specified using the `multi_params` flag parameter. Use *VRNA_PROBING_METHOD_MULTI_PARAMS_1* to indicate that `betas` points to an array of *beta* values for each sequence. *VRNA_PROBING_METHOD_MULTI_PARAMS_2* indicates that `pr_conversions` is pointing to an array of probing data to probability conversion strategies, and *VRNA_PROBING_METHOD_MULTI_PARAMS_3* indicates multiple default probabilities for missing data. Bitwise-OR of the three values renders all of them to be sequence specific.

See also

vrna_probing_data_t, *vrna_probing_data_free()*, *vrna_sc_probing()*, *vrna_probing_data_zarringhalam_comparative()*, *vrna_probing_data_deigan()*, *vrna_probing_data_deigan_comparative()*, *vrna_probing_data_eddy()*, *vrna_probing_data_eddy_comparative()*, *VRNA_PROBING_METHOD_MULTI_PARAMS_0*, *VRNA_PROBING_METHOD_MULTI_PARAMS_1*, *VRNA_PROBING_METHOD_MULTI_PARAMS_2*, *VRNA_PROBING_METHOD_MULTI_PARAMS_3*, *VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT*

Note

For further details, we refer to Zarringhalam *et al.* [2012]

Parameters

- **reactivities** – 0-based array of 1-based arrays of per-nucleotide probing data, e.g. SHAPE reactivities
- **n** – 0-based array of lengths of the `reactivities` lists
- **n_seq** – The number of sequences in the MSA

- **betas** – 0-based array with scaling factors β of the conversion function or the address of a scaling factor to be applied for all data
- **pr_conversions** – 0-based array of flags that specifies how to convert reactivities to probabilities or the address of a conversion strategy to be applied for all data
- **pr_defaults** – 0-based array of default probabilities for a nucleotide where reactivity data is missing for or the address of a single default probability to be applied for all data
- **multi_params** – A flag indicating what is passed through parameters `betas`, `pr_conversions`, and `pr_defaults`

Returns

A pointer to a data structure containing the probing data and any preparations necessary to use it in *vrna_sc_probing()* according to the method of Zarringhalam *et al.* [2012] or **NULL** on any error.

vrna_probing_data_t **vrna_probing_data_zarringhalam_trans_comparative**(const double **reactivities, unsigned int *n, unsigned int n_seq, double *betas, double *pr_defaults, unsigned int multi_params, *vrna_math_fun_dbl_f* *cb_preprocess, *vrna_math_fun_dbl_opt_t* *cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* *cb_preprocess_opt_free)

```
#include <ViennaRNA/probing/strategy_zarringhalam.h>
```

The Eddy 2014 Strategy API

Defines

VRNA_PROBING_STRATEGY_EDDY_OPTIONS_DEFAULT

Default options for the Eddy [2014] : probing data conversion strategy.

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

See also

vrna_probing_strategy_eddy_options(), *vrna_probing_data_eddy()*, vrna_probing_data_eddy_trans(), *vrna_probing_data_eddy_comparative()*, vrna_probing_data_eddy_trans_comparative()

VRNA_PROBING_STRATEGY_EDDY_NO_TEMPERATURE_RESCALING

Prevent temperature dependent energy rescaling in Eddy [2014] strategy.

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

This option flag forces the probing data conversion strategy to always use the same thermodynamic temperature T, no matter what temperature the predictions are made for.

See also

vrna_probing_strategy_eddy_options(), vrna_probing_data_eddy_trans(), vrna_probing_data_eddy_trans_comparative(), *vrna_probing_data_eddy()*, *vrna_probing_data_eddy_comparative()*

Functions

double ***vrna_probing_strategy_eddy**(*vrna_fold_compound_t* *fc, const double *data, size_t data_size, unsigned int target, void *options)

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

void ***vrna_probing_strategy_eddy_options**(double temperature, unsigned char options, const double *prior_unpaired, size_t prior_unpaired_size, const double *prior_paired, size_t prior_paired_size, *vrna_math_fun_dbl_f* cb_preprocess, *vrna_math_fun_dbl_opt_t* cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* cb_preprocess_opt_free)

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

void **vrna_probing_strategy_eddy_options_free**(void *options)

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

struct vrna_probing_data_s ***vrna_probing_data_eddy**(const double *reactivities, unsigned int n, double temperature, unsigned char options, const double *unpaired_data, unsigned int unpaired_len, const double *paired_data, unsigned int paired_len)

Add probing data as soft constraints (Eddy/RNAprob-2 method).

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

This approach of probing data directed RNA folding uses the probability framework proposed by Eddy [2014] :

$$\Delta G_{\text{data}}(i) = -RT \ln(\mathbb{P}(\text{data}(i) \mid x_i \pi_i))$$

to convert probing data to pseudo energies for given nucleotide x_i and class probability π_i at position i. The conditional probability is taken from a prior-distribution of probing data for the respective classes.

Here, the method distinguishes exactly two different classes of structural context, (i) unpaired and (ii) paired positions, following the lines of the RNAprob-2 method of Deng *et al.* [2016] . The reactivity distribution is computed using Gaussian kernel density estimation (KDE) with bandwidth h computed using Scott factor

$$h = n^{-\frac{1}{5}}$$

where n is the number of data points of the prior distribution.

See also

vrna_probing_data_t, *vrna_probing_data_free()*, *vrna_sc_probing()*, *vrna_probing_data_eddy_comparative()*, *vrna_probing_data_deigan()*, *vrna_probing_data_deigan_comparative()*, *vrna_probing_data_zarringhalam()*, *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_STRATEGY_EDDY_NO_TEMPERATURE_RES* *VRNA_PROBING_STRATEGY_EDDY_OPTIONS_DEFAULT*

Note

For further details, we refer to Eddy [2014] and Deng *et al.* [2016] .

Parameters

- **reactivities** – A 1-based vector of probing data, e.g. normalized SHAPE reactivities
- **n** – Length of reactivities
- **temperature** – The thermodynamic temperature T
- **options** – Options bit flags to change the behavior of the strategy
- **unpaired_data** – Pointer to an array of probing data for unpaired nucleotides
- **unpaired_len** – Length of unpaired_data
- **paired_data** – Pointer to an array of probing data for paired nucleotides
- **paired_len** – Length of paired_data

Returns

A pointer to a data structure containing the probing data and any preparations necessary to use it in *vrna_sc_probing()* according to the method of Eddy [2014] or **NULL** on any error.

struct vrna_probing_data_s ***vrna_probing_data_eddy_trans**(const double *reactivities, unsigned int n, double temperature, unsigned char options, const double *unpaired_data, unsigned int unpaired_len, const double *paired_data, unsigned int paired_len, *vrna_math_fun_dbl_f* cb_preprocess, *vrna_math_fun_dbl_opt_t* cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* cb_preprocess_opt_free)

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

struct vrna_probing_data_s *vrna_probing_data_eddy_comparative(const double **reactivities, const unsigned int *n, unsigned int n_seq, double temperature, unsigned char options, const double **unpaired_datas, const unsigned int *unpaired_lens, const double **paired_datas, const unsigned int *paired_lens, unsigned int multi_params)

Add probing data as soft constraints (Eddy/RNAprob-2 method) for comparative structure predictions.

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

Similar to *vrna_probing_data_eddy()*, this function prepares a data structure for probing data directed RNA folding. It uses the probability framework proposed by Eddy [2014] :

$$\Delta G_{\text{data}}(i) = -RT \ln(\mathbb{P}(\text{data}(i) \mid x_i \pi_i))$$

to convert probing data to pseudo energies for given nucleotide x_i and class probability π_i at position i. The conditional probability is taken from a prior-distribution of probing data for the respective classes.

This functions purpose is to allow for adding multiple probing data as required for comparative structure predictions over multiple sequence alignments (MSA) with `n_seq` sequences. For that purpose, `reactivities` can be provided for any of the sequences in the MSA. Individual probing data is always expected to be specified in sequence coordinates, i.e. without considering gaps in the MSA. Therefore, each set of `reactivities` may have a different length as specified the parameter `n`. In addition, each set of probing data may undergo the conversion using different prior distributions for unpaired and paired nucleotides. Whether or not multiple sets of conversion priors are provided must be specified using the `multi_params` flag parameter. Use *VRNA_PROBING_METHOD_MULTI_PARAMS_1* to indicate that `unpaired_datas` points to an array of unpaired probing data for each sequence. Similarly, *VRNA_PROBING_METHOD_MULTI_PARAMS_2* indicates that `paired_datas` is pointing to an array paired probing data for each sequence. Bitwise-OR of the two values renders both parameters to be sequence specific.

See also

vrna_probing_data_t, *vrna_probing_data_free()*, *vrna_sc_probing()*, *vrna_probing_data_eddy()*, *vrna_probing_data_deigan()*, *vrna_probing_data_deigan_comparative()*, *vrna_probing_data_zarringhalam()*, *vrna_probing_data_zarringhalam_comparative()*, *VRNA_PROBING_METHOD_MULTI_PARAMS_0*, *VRNA_PROBING_METHOD_MULTI_PARAMS_1*, *VRNA_PROBING_METHOD_MULTI_PARAMS_2*, *VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT*

Note

For further details, we refer to Eddy [2014] and Deng *et al.* [2016] .

Parameters

- **`reactivities`** – 0-based array of 1-based arrays of per-nucleotide probing data, e.g. SHAPE reactivities
- **`n`** – 0-based array of lengths of the `reactivities` lists

- **n_seq** – The number of sequences in the MSA
- **temperature** – The thermodynamic temperature T
- **options** – Options bit flags to change the behavior of the strategy
- **unpaired_datas** – 0-based array of 0-based arrays with probing data for unpaired nucleotides or address of a single array of such data
- **unpaired_lens** – 0-based array of lengths for each probing data array in `unpaired_datas`
- **paired_datas** – 0-based array of 0-based arrays with probing data for paired nucleotides or address of a single array of such data
- **paired_lens** – 0-based array of lengths for each probing data array in `paired_data`
- **multi_params** – A flag indicating what is passed through parameters `unpaired_datas` and `paired_datas`

Returns

A pointer to a data structure containing the probing data and any preparations necessary to use it in *vrna_sc_probing()* according to the method of Eddy [2014] or **NULL** on any error.

struct vrna_probing_data_s *__vrna_probing_data_eddy_trans_comparative__(const double **reactivities, const unsigned int *n, unsigned int n_seq, double temperature, unsigned char options, const double **unpaired_datas, const unsigned int *unpaired_lens, const double **paired_datas, const unsigned int *paired_lens, unsigned int multi_params, *vrna_math_fun_dbl_f* *cb_preprocess, *vrna_math_fun_dbl_opt_t* *cb_preprocess_opt, *vrna_math_fun_dbl_opt_free_f* *cb_preprocess_opt_free)

```
#include <ViennaRNA/probing/strategy_eddy.h>
```

SHAPE Reactivity Data

Incorporate SHAPE reactivity structure probing data into the folding recursions by means of soft constraints.

Defines

VRNA_PROBING_DATA_CHECK_SEQUENCE

```
#include <ViennaRNA/probing/SHAPE.h>
```

Functions

int **vrna_sc_SHAPE_to_pr**(const char *shape_conversion, double *values, int length, double default_value)

Convert SHAPE reactivity values to probabilities for being unpaired.

```
#include <ViennaRNA/probing/basic.h>
```

This function parses the informations from a given file and stores the result in the pre-allocated string sequence and the *FLT_OR_DBL* array values.

See also

vrna_file_SHAPE_read()

Parameters

- **shape_conversion** – String defining the method used for the conversion process
- **values** – Pointer to an array of SHAPE reactivities
- **length** – Length of the array of SHAPE reactivities
- **default_value** – Result used for position with invalid/missing reactivity values

int **vrna_sc_add_SHAPE_deigan**(*vrna_fold_compound_t* *fc, const double *reactivities, double m, double b, unsigned int options)

Add SHAPE reactivity data as soft constraints (Deigan et al. method).

```
#include <ViennaRNA/probing/SHAPE.h>
```

This approach of SHAPE directed RNA folding uses the simple linear ansatz

$$\Delta G_{\text{SHAPE}}(i) = m \ln(\text{SHAPE reactivity}(i) + 1) + b$$

to convert SHAPE reactivity values to pseudo energies whenever a nucleotide i contributes to a stacked pair. A positive slope m penalizes high reactivities in paired regions, while a negative intercept b results in a confirmatory *bonus* free energy for correctly predicted base pairs. Since the energy evaluation of a base pair stack involves two pairs, the pseudo energies are added for all four contributing nucleotides. Consequently, the energy term is applied twice for pairs inside a helix and only once for pairs adjacent to other structures. For all other loop types the energy model remains unchanged even when the experimental data highly disagrees with a certain motif.

SWIG Wrapper Notes:

This function is attached as method `sc_add_SHAPE_deigan()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_SHAPE_deigan()* in the *The RNA Python module* .

See also

vrna_sc_remove(), *vrna_sc_add_SHAPE_zarringhalam()*, *vrna_sc_minimize_pertubation()*

Note

For further details, we refer to Deigan *et al.* [2009] .

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **reactivities** – A vector of normalized SHAPE reactivities
- **m** – The slope of the conversion function
- **b** – The intercept of the conversion function
- **options** – The options flag indicating how/where to store the soft constraints

Returns

1 on successful extraction of the method, 0 on errors

int **vrna_sc_add_SHAPE_deigan_ali**(*vrna_fold_compound_t* *fc, const char **shape_files, const int *shape_file_association, double m, double b, unsigned int options)

Add SHAPE reactivity data from files as soft constraints for consensus structure prediction (Deigan et al. method).

```
#include <ViennaRNA/probing/SHAPE.h>
```

SWIG Wrapper Notes:

This function is attached as method `sc_add_SHAPE_deigan_ali()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_SHAPE_deigan_ali()* in the *The RNA Python module* .

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **shape_files** – A set of filenames that contain normalized SHAPE reactivity data
- **shape_file_association** – An array of integers that associate the files with sequences in the alignment
- **m** – The slope of the conversion function
- **b** – The intercept of the conversion function
- **options** – The options flag indicating how/where to store the soft constraints

Returns

1 on successful extraction of the method, 0 on errors

int **vrna_sc_add_SHAPE_zarringhalam**(*vrna_fold_compound_t* *fc, const double *reactivities, double b, double default_value, const char *shape_conversion, unsigned int options)

Add SHAPE reactivity data as soft constraints (Zarringhalam et al. method).

```
#include <ViennaRNA/probing/SHAPE.h>
```

This method first converts the observed SHAPE reactivity of nucleotide i into a probability q_i that position i is unpaired by means of a non-linear map. Then pseudo-energies of the form

$$\Delta G_{\text{SHAPE}}(x, i) = \beta \, |x_i - q_i|$$

are computed, where $x_i = 0$ if position i is unpaired and $x_i = 1$ if i is paired in a given secondary structure. The parameter β serves as scaling factor. The magnitude of discrepancy between prediction and experimental observation is represented by $|x_i - q_i|$.

SWIG Wrapper Notes:

This function is attached as method `sc_add_SHAPE_zarringhalam()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.sc_add_SHAPE_zarringhalam()`* in the *The RNA Python module* .

See also

vrna_sc_remove(), *vrna_sc_add_SHAPE_deigan()*, *vrna_sc_minimize_pertubation()*

Note

For further details, we refer to Zarringhalam *et al.* [2012]

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **reactivities** – A vector of normalized SHAPE reactivities
- **b** – The scaling factor β of the conversion function
- **default_value** – The default value for a nucleotide where reactivity data is missing for
- **shape_conversion** – A flag that specifies how to convert reactivities to probabilities
- **options** – The options flag indicating how/where to store the soft constraints

Returns

1 on successful extraction of the method, 0 on errors

int **vrna_sc_add_SHAPE_eddy_2**(*vrna_fold_compound_t* *fc, const double *reactivities, int unpaired_nb, const double *unpaired_data, int paired_nb, const double *paired_data)

Add SHAPE reactivity data as soft constraints (Eddy/RNAprob-2 method).

```
#include <ViennaRNA/probing/SHAPE.h>
```

This approach of SHAPE directed RNA folding uses the probability framework proposed by Eddy

$$\Delta G_{\text{SHAPE}}(i) = -RT \ln(\mathbb{P}(\text{SHAPE reactivity}(i) \mid x_i \pi_i)) +$$

to convert SHAPE reactivity values to pseudo energies for given nucleotide x_i and pairedness π_i at position i. The reactivity distribution is computed using Gaussian kernel density estimation (KDE) with bandwidth h computed using Scott factor

$$h = n^{-\frac{1}{5}}$$

where n is the number of data points.

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **reactivities** – A vector of normalized SHAPE reactivities
- **unpaired_nb** – Length of the array of unpaired SHAPE reactivities
- **unpaired_data** – Pointer to an array of unpaired SHAPE reactivities
- **paired_nb** – Length of the array of paired SHAPE reactivities
- **paired_data** – Pointer to an array of paired SHAPE reactivities

Returns

1 on successful extraction of the method, 0 on errors

void **vrna_constraints_add_SHAPE_ali**(*vrna_fold_compound_t* *fc, const char *shape_method, const char **shape_files, const int *shape_file_association, int verbose, unsigned int constraint_type)

```
#include <ViennaRNA/probing/SHAPE.h>
```

void **vrna_constraints_add_SHAPE**(*vrna_fold_compound_t* *fc, const char *shape_file, const char *shape_method, const char *shape_conversion, int verbose, unsigned int constraint_type)

```
#include <ViennaRNA/probing/SHAPE.h>
```

Generate Soft Constraints from Data

Find a vector of perturbation energies that minimizes the discripancies between predicted and observed pairing probabilities and the amount of neccessary adjustments.

Defines

VRNA_OBJECTIVE_FUNCTION_QUADRATIC

Use the sum of squared aberrations as objective function.

```
#include <ViennaRNA/perturbation_fold.h>
```

$F(\vec{\epsilon}) = \sum_{i=1}^{n} \frac{\epsilon_i^2}{\tau^2} + \sum_{i=1}^{n} \frac{(p_i(\vec{\epsilon}) - q_i)^2}{\sigma^2} \rightarrow min$

VRNA_OBJECTIVE_FUNCTION_ABSOLUTE

Use the sum of absolute aberrations as objective function.

```
#include <ViennaRNA/perturbation_fold.h>
```

$F(\vec{\epsilon}) = \sum_{i=1}^{n} \frac{|\epsilon_i|}{\tau^2} + \sum_{i=1}^{n} \frac{|p_i(\vec{\epsilon}) - q_i|}{\sigma^2} \rightarrow min$

VRNA_MINIMIZER_DEFAULT

Use a custom implementation of the gradient descent algorithm to minimize the objective function.

```
#include <ViennaRNA/perturbation_fold.h>
```

VRNA_MINIMIZER_CONJUGATE_FR

Use the GNU Scientific Library implementation of the Fletcher-Reeves conjugate gradient algorithm to minimize the objective function.

```
#include <ViennaRNA/perturbation_fold.h>
```

Please note that this algorithm can only be used when the GNU Scientific Library is available on your system

VRNA_MINIMIZER_CONJUGATE_PR

Use the GNU Scientific Library implementation of the Polak-Ribiere conjugate gradient algorithm to minimize the objective function.

```
#include <ViennaRNA/perturbation_fold.h>
```

Please note that this algorithm can only be used when the GNU Scientific Library is available on your system

VRNA_MINIMIZER_VECTOR_BFGS

Use the GNU Scientific Library implementation of the vector Broyden-Fletcher-Goldfarb-Shanno algorithm to minimize the objective function.

```
#include <ViennaRNA/perturbation_fold.h>
```

Please note that this algorithm can only be used when the GNU Scientific Library is available on your system

VRNA_MINIMIZER_VECTOR_BFGS2

Use the GNU Scientific Library implementation of the vector Broyden-Fletcher-Goldfarb-Shanno algorithm to minimize the objective function.

```
#include <ViennaRNA/perturbation_fold.h>
```

Please note that this algorithm can only be used when the GNU Scientific Library is available on your system

VRNA_MINIMIZER_STEEPEST_DESCENT

Use the GNU Scientific Library implementation of the steepest descent algorithm to minimize the objective function.

```
#include <ViennaRNA/perturbation_fold.h>
```

Please note that this algorithm can only be used when the GNU Scientific Library is available on your system

Typedefs

typedef void (***progress_callback**)(int iteration, double score, double *epsilon)

Callback for following the progress of the minimization process.

```
#include <ViennaRNA/perturbation_fold.h>
```

Param iteration
The number of the current iteration

Param score
The score of the objective function

Param epsilon
The perturbation vector yielding the reported score

Functions

void **vrna_sc_minimize_pertubation**(*vrna_fold_compound_t* *fc, const double *q_prob_unpaired, int objective_function, double sigma_squared, double tau_squared, int algorithm, int sample_size, double *epsilon, double initialStepSize, double minStepSize, double minImprovement, double minimizerTolerance, *progress_callback* callback)

Find a vector of perturbation energies that minimizes the discripancies between predicted and observed pairing probabilities and the amount of neccessary adjustments.

```
#include <ViennaRNA/perturbation_fold.h>
```

Use an iterative minimization algorithm to find a vector of perturbation energies whose incorporation as soft constraints shifts the predicted pairing probabilities closer to the experimentally observed probabilities. The algorithm aims to minimize an objective function that penalizes discripancies between predicted and observed pairing probabilities and energy model adjustments, i.e. an appropriate vector of perturbation energies satisfies

$$F(\vec{\epsilon}) = \sum_{\mu} \frac{\epsilon_{\mu}^2}{\tau^2} + \sum_{i=1}^{n} \frac{(p_i(\vec{\epsilon}) - q_i)^2}{\sigma^2} \rightarrow \min .$$

An initialized fold compound and an array containing the observed probability for each nucleotide to be unbound are required as input data. The parameters objective_function, sigma_squared and tau_squared are responsible for adjusting the aim of the objective function. Dependend on which type of objective function is selected, either squared or absolute aberrations are contributing to the objective function. The ratio of the parameters sigma_squared and tau_squared can be used to adjust the algorithm to find a solution either close to the thermodynamic prediction (sigma_squared >> tau_squared) or close to the experimental data (tau_squared >> sigma_squared). The minimization can be performed by makeing use of a custom gradient descent implementation or using one of the minimizing algorithms provided by the GNU Scientific Library. All algorithms require the evaluation of the gradient of the objective function, which includes the evaluation of conditional pairing probabilites. Since an exact evaluation is expensive, the probabilities can also be estimated from sampling by setting an appropriate sample size. The found vector of perturbation energies will be stored in the array epsilon. The progress of the minimization process can be tracked by implementing and passing a callback function.

See also

For further details we refer to Washietl *et al.* [2012] .

Parameters

- **fc** – Pointer to a fold compound
- **q_prob_unpaired** – Pointer to an array containing the probability to be unpaired for each nucleotide
- **objective_function** – The type of objective function to be used (VRNA_OBJECTIVE_FUNCTION_QUADRATIC / VRNA_OBJECTIVE_FUNCTION_LINEAR)

- **sigma_squared** – A factor used for weighting the objective function. More weight on this factor will lead to a solution close to the null vector.
- **tau_squared** – A factor used for weighting the objective function. More weight on this factor will lead to a solution close to the data provided in q_prob_unpaired.
- **algorithm** – The minimization algorithm (VRNA_MINIMIZER_*)
- **sample_size** – The number of sampled sequences used for estimating the pairing probabilities. A value <= 0 will lead to an exact evaluation.
- **epsilon** – A pointer to an array used for storing the calculated vector of perturbation energies
- **initialStepSize**
- **minStepSize**
- **minImprovement**
- **minimizerTolerance**
- **callback** – A pointer to a callback function used for reporting the current minimization progress

4.1.10 Post-transcriptional Base Modifications

Support of modified bases in secondary structure prediction.

Defines

VRNA_SC_MOD_CHECK_FALLBACK

Check for sequence positions whether they resemble the fallback base.

```
#include <ViennaRNA/constraints/soft_special.h>
```

This flag can be used to enable a sanity check within the vrna_sc_mod*() functions to see whether a supposedly modified position actually resembles the fallback base as specified in the modification parameters

See also

vrna_sc_mod_json(), *vrna_sc_mod_jsonfile()*, *vrna_sc_mod()*, *vrna_sc_mod_m6A()*, *vrna_sc_mod_pseudouridine()*, *vrna_sc_mod_inosine()*, *vrna_sc_mod_7DA()*, *vrna_sc_mod_purine()*, *vrna_sc_mod_dihydrouridine()*, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_DEFAULT*

VRNA_SC_MOD_CHECK_UNMOD

Check for sequence positions whether they resemble the unmodified base.

```
#include <ViennaRNA/constraints/soft_special.h>
```

This flag can be used to enable a sanity check within the vrna_sc_mod*() functions to see whether a supposedly modified position actually resembles the unmodified base as specified in the modification parameters

See also

vrna_sc_mod_json(), *vrna_sc_mod_jsonfile()*, *vrna_sc_mod()*, *vrna_sc_mod_m6A()*, *vrna_sc_mod_pseudouridine()*, *vrna_sc_mod_inosine()*, *vrna_sc_mod_7DA()*, *vrna_sc_mod_purine()*, *vrna_sc_mod_dihydrouridine()*, *VRNA_SC_MOD_CHECK_FALLBACK*, *VRNA_SC_MOD_DEFAULT*

VRNA_SC_MOD_SILENT

Do not produce any warnings within the vrna_sc_mod*() functions.

```
#include <ViennaRNA/constraints/soft_special.h>
```

See also

vrna_sc_mod_json(), *vrna_sc_mod_jsonfile()*, *vrna_sc_mod()*, *vrna_sc_mod_m6A()*, *vrna_sc_mod_pseudouridine()*, *vrna_sc_mod_inosine()*, *vrna_sc_mod_7DA()*, *vrna_sc_mod_purine()*, *vrna_sc_mod_dihydrouridine()*

VRNA_SC_MOD_DEFAULT

Default settings for the vrna_sc_mod*() functions.

```
#include <ViennaRNA/constraints/soft_special.h>
```

See also

vrna_sc_mod_json(), *vrna_sc_mod_jsonfile()*, *vrna_sc_mod()*, *vrna_sc_mod_m6A()*, *vrna_sc_mod_pseudouridine()*, *vrna_sc_mod_inosine()*, *vrna_sc_mod_7DA()*, *vrna_sc_mod_purine()*, *vrna_sc_mod_dihydrouridine()*, *VRNA_SC_MOD_CHECK_FALLBACK*, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*

Typedefs

typedef struct vrna_sc_mod_param_s ***vrna_sc_mod_param_t**

Modified base parameter data structure.

```
#include <ViennaRNA/constraints/soft_special.h>
```

See also

vrna_sc_mod_read_from_jsonfile(), *vrna_sc_mod_read_from_json()*, *vrna_sc_mod()*

Functions

vrna_sc_mod_param_t **vrna_sc_mod_read_from_jsonfile**(const char *filename, *vrna_md_t* *md)

Parse and extract energy parameters for a modified base from a JSON file.

```
#include <ViennaRNA/constraints/soft_special.h>
```

SWIG Wrapper Notes:

This function is available as an overloaded function `sc_mod_read_from_jsonfile()` where the `md` parameter may be omitted and defaults to `NULL`. See, e.g. *RNA.sc_mod_read_from_jsonfile()* in the *The RNA Python module* .

See also

vrna_sc_mod_read_from_json(), *vrna_sc_mod_parameters_free()*, *vrna_sc_mod()*, *Modified Bases*

Parameters

- **filename** – The JSON file containing the specifications of the modified base
- **md** – A model-details data structure (for look-up of canonical base pairs)

Returns

Parameters of the modified base

vrna_sc_mod_param_t **vrna_sc_mod_read_from_json**(const char *json, *vrna_md_t* *md)

Parse and extract energy parameters for a modified base from a JSON string.

```
#include <ViennaRNA/constraints/soft_special.h>
```

SWIG Wrapper Notes:

This function is available as an overloaded function `sc_mod_read_from_json()` where the `md` parameter may be omitted and defaults to `NULL`. See, e.g. *RNA.sc_mod_read_from_json()* in the *The RNA Python module* .

See also

vrna_sc_mod_read_from_jsonfile(), *vrna_sc_mod_parameters_free()*, *vrna_sc_mod()*, *Modified Bases*

Parameters

- **json** – The JSON string containing the specifications of the modified base
- **md** – A model-details data structure (for look-up of canonical base pairs)

Returns

Parameters of the modified base

void **vrna_sc_mod_parameters_free**(*vrna_sc_mod_param_t* params)

Release memory occupied by a modified base parameter data structure.

```
#include <ViennaRNA/constraints/soft_special.h>
```

Properly free a *vrna_sc_mod_param_t* data structure

Parameters

- **params** – The data structure to free

int **vrna_sc_mod_json**(*vrna_fold_compound_t* *fc, const char *json, const unsigned int *modification_sites, unsigned int options)

Prepare soft constraint callbacks for modified base as specified in JSON string.

```
#include <ViennaRNA/constraints/soft_special.h>
```

This function prepares all requirements to acknowledge modified bases as specified in the provided `json` string. All subsequent predictions will treat each modification site special and adjust energy contributions if necessary.

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod_json()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_json()* in the *The RNA Python module* .

See also

vrna_sc_mod_jsonfile(), *vrna_sc_mod()*, *vrna_sc_mod_m6A()*, *vrna_sc_mod_pseudouridine()*, *vrna_sc_mod_inosine()*, *vrna_sc_mod_7DA()*, *vrna_sc_mod_purine()*, *vrna_sc_mod_dihydrouridine()*, *VRNA_SC_MOD_CHECK_FALLBACK*, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*, *Modified Bases*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **json** – The JSON formatted string with the modified base parameters
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns

Number of sequence positions modified base parameters will be used for

int **vrna_sc_mod_jsonfile**(*vrna_fold_compound_t* *fc, const char *json_file, const unsigned int *modification_sites, unsigned int options)

Prepare soft constraint callbacks for modified base as specified in JSON string.

```
#include <ViennaRNA/constraints/soft_special.h>
```

Similar to *vrna_sc_mod_json()*, this function prepares all requirements to acknowledge modified bases as specified in the provided `json` file. All subsequent predictions will treat each modification site special and adjust energy contributions if necessary.

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod_jsonfile()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_jsonfile()* in the *The RNA Python module* .

See also

vrna_sc_mod_json(), *vrna_sc_mod()*, *vrna_sc_mod_m6A()*, *vrna_sc_mod_pseudouridine()*, *vrna_sc_mod_inosine()*, *vrna_sc_mod_7DA()*, *vrna_sc_mod_purine()*, *vrna_sc_mod_dihydrouridine()*, *VRNA_SC_MOD_CHECK_FALLBACK*, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*, *Modified Bases*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **json_file** – The JSON formatted string with the modified base parameters
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns

Number of sequence positions modified base parameters will be used for

int **vrna_sc_mod**(*vrna_fold_compound_t* *fc, const *vrna_sc_mod_param_t* params, const unsigned int *modification_sites, unsigned int options)

Prepare soft constraint callbacks for modified base as specified in JSON string.

```
#include <ViennaRNA/constraints/soft_special.h>
```

This function takes a *vrna_sc_mod_param_t* data structure as obtained from *vrna_sc_mod_read_from_json()* or *vrna_sc_mod_read_from_jsonfile()* and prepares all requirements to acknowledge modified bases as specified in the provided `params` data structure. All subsequent predictions will treat each modification site special and adjust energy contributions if necessary.

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod()* in the *The RNA Python module* .

See also

vrna_sc_mod_read_from_json(), *vrna_sc_mod_read_from_jsonfile()*, *vrna_sc_mod_json()*, *vrna_sc_mod_jsonfile()*, *vrna_sc_mod_m6A()*, *vrna_sc_mod_pseudouridine()*, *vrna_sc_mod_inosine()*, *vrna_sc_mod_7DA()*, *vrna_sc_mod_purine()*, *vrna_sc_mod_dihydrouridine()* *VRNA_SC_MOD_CHECK_FALLBACK*, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **params** – The modified base parameters
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns
Number of sequence positions modified base parameters will be used for

int **vrna_sc_mod_m6A**(*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

Add soft constraint callbacks for N6-methyl-adenosine (m6A).

```
#include <ViennaRNA/constraints/soft_special.h>
```

This is a convenience wrapper to add support for m6A using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Kierzek *et al.* [2022] .

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod_m6A()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_m6A()* in the *The RNA Python module* .

See also

VRNA_SC_MOD_CHECK_FALLBACK, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns
Number of sequence positions modified base parameters will be used for

int **vrna_sc_mod_pseudouridine**(*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

Add soft constraint callbacks for Pseudouridine.

```
#include <ViennaRNA/constraints/soft_special.h>
```

This is a convenience wrapper to add support for pseudouridine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Hudson *et al.* [2013] .

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod_pseudouridine()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_pseudouridine()* in the *The RNA Python module* .

See also

VRNA_SC_MOD_CHECK_FALLBACK, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns

Number of sequence positions modified base parameters will be used for

int **vrna_sc_mod_inosine**(*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

Add soft constraint callbacks for Inosine.

```
#include <ViennaRNA/constraints/soft_special.h>
```

This is a convenience wrapper to add support for inosine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Wright *et al.* [2007] and Wright *et al.* [2018] .

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod_inosine()` to objects of type `fold_compound` with default `options = VRNA_SC_MOD_DEFAULT`. See, e.g. *RNA.fold_compound.sc_mod_inosine()* in the *The RNA Python module* .

See also

VRNA_SC_MOD_CHECK_FALLBACK, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns

Number of sequence positions modified base parameters will be used for

int **vrna_sc_mod_7DA**(*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

Add soft constraint callbacks for 7-deaza-adenosine (7DA).

```
#include <ViennaRNA/constraints/soft_special.h>
```

This is a convenience wrapper to add support for 7-deaza-adenosine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Richardson and Znosko [2016] .

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod_7DA()` to objects of type `fold_compound` with default `options = VRNA_SC_MOD_DEFAULT`. See, e.g. `RNA.fold_compound.sc_mod_7DA()` in the *The RNA Python module* .

See also

VRNA_SC_MOD_CHECK_FALLBACK, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns

Number of sequence positions modified base parameters will be used for

int **vrna_sc_mod_purine**(*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

Add soft constraint callbacks for Purine (a.k.a. nebularine).

```
#include <ViennaRNA/constraints/soft_special.h>
```

This is a convenience wrapper to add support for Purine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Jolley and Znosko [2017] .

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod_purine()` to objects of type `fold_compound` with default `options = VRNA_SC_MOD_DEFAULT`. See, e.g. `RNA.fold_compound.sc_mod_purine()` in the *The RNA Python module* .

See also

VRNA_SC_MOD_CHECK_FALLBACK, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns
Number of sequence positions modified base parameters will be used for

int **vrna_sc_mod_dihydrouridine**(*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

Add soft constraint callbacks for dihydrouridine.

```
#include <ViennaRNA/constraints/soft_special.h>
```

This is a convenience wrapper to add support for dihydrouridine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Rosetta/RECESS predictions.

SWIG Wrapper Notes:

This function is attached as overloaded method `sc_mod_dihydrouridine()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_dihydrouridine()* in the *The RNA Python module* .

See also

VRNA_SC_MOD_CHECK_FALLBACK, *VRNA_SC_MOD_CHECK_UNMOD*, *VRNA_SC_MOD_SILENT*, *VRNA_SC_MOD_DEFAULT*

Parameters

- **fc** – The fold_compound the corrections should be bound to
- **modification_sites** – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** – A bitvector of options how to handle the input, e.g. *VRNA_SC_MOD_DEFAULT*

Returns
Number of sequence positions modified base parameters will be used for

4.1.11 Ligands Binding to RNA Structures

Basic Ligand Binding API

Simple Extensions to Model Ligand Binding to RNA Structures.

Unstructured Domains

Add ligand binding to loop regions using the *Unstructured Domains* feature.

Hairpin-/Internal Loop Motifs

Ligand binding to specific hairpin/internal loop like motifs using the *Soft Constraints* feature.

Typedefs

typedef struct *vrna_sc_motif_s* **vrna_sc_motif_t**

Type definition for soft constraint motif.

```
#include <ViennaRNA/constraints/ligand.h>
```

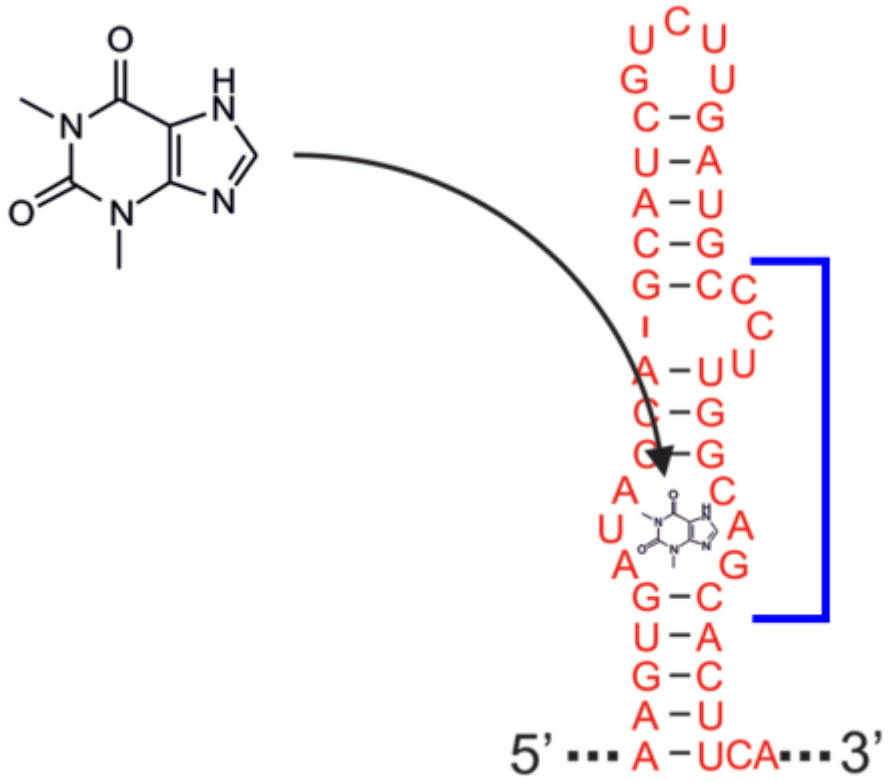
Functions

int **vrna_sc_add_hi_motif**(*vrna_fold_compound_t* *fc, const char *seq, const char *structure, *FLT_OR_DBL* energy, unsigned int options)

Add soft constraints for hairpin or internal loop binding motif.

```
#include <ViennaRNA/constraints/ligand.h>
```

Here is an example that adds a theophylline binding motif. Free energy contribution is derived from $k_d = 0.1\mu M$, taken from Jenison et al. 1994. At $1M$ concentration the corresponding binding free energy amounts to $-9.93\ kcal/mol$.




```
vrna_sc_add_hi_motif(fc,
"GAUACCAG&CCCUUGGCAGC",
"(...((((&)...)))...)",
                     -9.93, VRNA_OPTION_DEFAULT);
```

SWIG Wrapper Notes:

This function is attached as method `sc_add_hi_motif()` to objects of type `fold_compound`. The last parameter is optional an defaults to `options` = *VRNA_OPTION_DEFAULT*. See, e.g. *RNA.fold_compound.sc_add_hi_motif()* in the *The RNA Python module* .

Parameters

- **fc** – The *vrna_fold_compound_t* the motif is applied to
- **seq** – The sequence motif (may be interspaced by '&' character
- **structure** – The structure motif (may be interspaced by '&' character
- **energy** – The free energy of the motif (e.g. binding free energy)
- **options** – Options

Returns
non-zero value if application of the motif using soft constraints was successful

vrna_sc_motif_t ***vrna_sc_ligand_detect_motifs**(*vrna_fold_compound_t* *fc, const char *structure)

```
#include <ViennaRNA/constraints/ligand.h>
```

vrna_sc_motif_t ***vrna_sc_ligand_get_all_motifs**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/constraints/ligand.h>
```

struct **vrna_sc_motif_s**

Public Members

int **i**

```
#include <ViennaRNA/constraints/ligand.h>
```

int **j**

```
#include <ViennaRNA/constraints/ligand.h>
```

int **k**

```
#include <ViennaRNA/constraints/ligand.h>
```

int **l**

```
#include <ViennaRNA/constraints/ligand.h>
```

int **number**

```
#include <ViennaRNA/constraints/ligand.h>
```

4.1.12 Structure Modules and Pseudoknots

Basic Structure Modules API

Pseudoknots

Implementations to predict pseudoknotted structures.

Typedefs

typedef int (***vrna_pk_plex_score_f**)(const short *pt, int start_5, int end_5, int start_3, int end_3, void *data)

Pseudoknot loop scoring function prototype.

```
#include <ViennaRNA/pk_plex.h>
```

This function is used to evaluate a formed pseudoknot (PK) interaction in *vrna_pk_plex()*. It is supposed to take a PK-free secondary structure as input and coordinates of an additional interaction site. From this data, the energy (penalty) to score the PK loop is derived and returned in decakal/mol. Upon passing zero in any of the interaction site coordinates (`start_5`, `end_5`, `start_3`, `end_3`) or a *NULL* pointer in `pt`, the function must return a PK loop score. This minimum PK loop score is used in the first phase of the heuristic implemented in *vrna_pk_plex()* to assess whether a particular interaction is further taken into account in a later, more thorough evaluation step.

The simplest scoring function would simply return a constant score for any PK loop, no matter what type of loop is formed and how large the loop is. This is the default if *vrna_pk_plex_opt_defaults()* or *vrna_pk_plex_opt()* is used to generate options for *vrna_pk_plex()*.

See also

vrna_pk_plex_opt_fun(), *vrna_pk_plex()*

Param pt
The secondary structure (without pseudoknot) in pair table notation

Param start_5
The start coordinate of the 5' site of the pseudoknot interaction

Param end_5
The end coordinate of the 5' site of the pseudoknot interaction

Param start_3
The start coordinate of the 3' site of the pseudoknot interaction

Param end_3
The end coordinate of the 3' site of the pseudoknot interaction

Param data
An arbitrary data structure passed from the calling function

Return
The energy (penalty) of the resulting pseudoknot

typedef int **vrna_callback_pk_plex_score**(const short *pt, int start_5, int end_5, int start_3, int end_3, void *data)

```
#include <ViennaRNA/pk_plex.h>
```

typedef struct vrna_pk_plex_option_s ***vrna_pk_plex_opt_t**

RNA PKplex options object.

```
#include <ViennaRNA/pk_plex.h>
```

See also

vrna_pk_plex_opt_defaults(), *vrna_pk_plex_opt()*, *vrna_pk_plex_opt_fun()*, *vrna_pk_plex()*, *vrna_pk_plex_score_f*

typedef struct *vrna_pk_plex_result_s* **vrna_pk_plex_t**

Convenience typedef for results of the RNA PKplex prediction.

```
#include <ViennaRNA/pk_plex.h>
```

See also

#vrna_pk_plex_results_s, *vrna_pk_plex()*

Functions

vrna_pk_plex_t ***vrna_pk_plex**(*vrna_fold_compound_t* *fc, const int **accessibility, *vrna_pk_plex_opt_t* options)

Predict Pseudoknot interactions in terms of a two-step folding process.

```
#include <ViennaRNA/pk_plex.h>
```

Computes simple pseudoknot interactions according to the PKplex algorithm. This simple heuristic first compiles a list of potential interaction sites that may form a pseudoknot. The resulting candidate interactions are then fixed and an PK-free MFE structure for the remainder of the sequence is computed.

The `accessibility` argument is a list of opening energies for potential interaction sites. It is used in the first step of the algorithm to identify potential interactions. Upon passing *NULL*, the opening energies are determined automatically based on the current model settings.

Depending on the `options`, the function can return the MFE (incl. PK loops) or suboptimal structures within an energy band around the MFE. The PK loop is internally scored by a scoring function that in the simplest cases assigns a constant value for each PK loop. More complicated scoring functions can be passed as well, see *vrna_pk_plex_score_f* and *vrna_pk_plex_opt_fun()*.

The function returns *NULL* on any error. Otherwise, a list of structures and interaction coordinates with corresponding energy contributions is returned. If no PK-interaction that satisfies the options is found, the list only consists of the PK-free MFE structure.

Parameters

- **fc** – fold compound with the input sequence and model settings
- **accessibility** – An array of opening energies for the implemented heuristic (maybe *NULL*)
- **options** – An *vrna_pk_plex_opt_t* options data structure that determines the algorithm parameters

Returns
A list of potentially pseudoknotted structures (Last element in the list indicated by *NULL* value in *vrna_pk_plex_result_s.structure*)

int **vrna_pk_plex_accessibility(*vrna_fold_compound_t* *fc, unsigned int unpaired, double cutoff)

Obtain a list of opening energies suitable for PKplex computations.

```
#include <ViennaRNA/pk_plex.h>
```

See also

vrna_pk_plex()

Parameters

- **fc** – fold compound with the input sequence and model settings for accessibility computations
- **unpaired** – The maximum number of unpaired nucleotides, i.e. length of interaction
- **cutoff** – A cutoff value for unpaired probabilities

Returns
Opening energies as required for *vrna_pk_plex()*

vrna_pk_plex_opt_t vrna_pk_plex_opt_defaults(void)

Default options for PKplex algorithm.

```
#include <ViennaRNA/pk_plex.h>
```

See also

vrna_pk_plex(), *vrna_pk_plex_opt()*, *vrna_pk_plex_opt_fun()*

Returns
An options data structure suitabe for PKplex computations

vrna_pk_plex_opt_t vrna_pk_plex_opt(unsigned int delta, unsigned int max_interaction_length, int pk_penalty)

Simple options for PKplex algorithm.

```
#include <ViennaRNA/pk_plex.h>
```

See also

vrna_pk_plex(), *vrna_pk_plex_opt_defaults()*, *vrna_pk_plex_opt_fun()*

Parameters

- **delta** – Size of energy band around MFE for suboptimal results in dekacal/mol
- **max_interaction_length** – Maximum length of interaction

- **pk_penalty** – Energy constant to score the PK forming loop

Returns
An options data structure suitabe for PKplex computations

vrna_pk_plex_opt_t **vrna_pk_plex_opt_fun**(unsigned int delta, unsigned int max_interaction_length, *vrna_pk_plex_score_f* scoring_function, void *scoring_data)

Simple options for PKplex algorithm.

```
#include <ViennaRNA/pk_plex.h>
```

See also

vrna_pk_plex(), *vrna_pk_plex_opt_defaults()*, *vrna_pk_plex_opt()*, *vrna_pk_plex_score_f*

Parameters

- **delta** – Size of energy band around MFE for suboptimal results in dekacal/mol
- **max_interaction_length** – Maximum length of interaction
- **scoring_function** – Energy evaluating function to score the PK forming loop
- **scoring_data** – An arbitrary data structure passed to the scoring function (maybe *NUL*)

Returns
An options data structure suitabe for PKplex computations

struct **vrna_pk_plex_result_s**

#include <ViennaRNA/pk_plex.h> A result of the RNA PKplex interaction prediction.

See also

vrna_pk_plex_t

Public Members

char ***structure**

Secondary Structure in dot-bracket notation.

```
#include <ViennaRNA/pk_plex.h>
```

double **energy**

Net free energy in kcal/mol.

```
#include <ViennaRNA/pk_plex.h>
```

double **dGpk**

Free energy of PK loop in kcal/mol.

```
#include <ViennaRNA/pk_plex.h>
```

double **dGint**

Free energy of PK forming duplex interaction.

```
#include <ViennaRNA/pk_plex.h>
```

double **dG1**

Opening energy for the 5' interaction site used in the heuristic.

```
#include <ViennaRNA/pk_plex.h>
```

double **dG2**

Opening energy for the 3' interaction site used in the heuristic.

```
#include <ViennaRNA/pk_plex.h>
```

unsigned int **start_5**

Start coordinate of the 5' interaction site.

```
#include <ViennaRNA/pk_plex.h>
```

unsigned int **end_5**

End coordinate of the 5' interaction site.

```
#include <ViennaRNA/pk_plex.h>
```

unsigned int **start_3**

Start coordinate of the 3' interaction site.

```
#include <ViennaRNA/pk_plex.h>
```

unsigned int **end_3**

End coordinate of the 3' interaction site.

```
#include <ViennaRNA/pk_plex.h>
```

G-Quadruplexes

Energy Evaluation

Functions

int **vrna_E_gquad**(unsigned int L, unsigned int l[3], *vrna_param_t* *P)

```
#include <ViennaRNA/eval/gquad.h>
```

FLT_OR_DBL **vrna_exp_E_gquad**(unsigned int L, unsigned int l[3], *vrna_exp_param_t* *pf)

```
#include <ViennaRNA/eval/gquad.h>
```

void **vrna_E_consensus_gquad**(unsigned int L, unsigned int l[3], unsigned int position, unsigned int length, unsigned int n_seq, const short **S, const unsigned int **a2s, *vrna_param_t* *P, int en[2])

```
#include <ViennaRNA/eval/gquad.h>
```

FLT_OR_DBL **vrna_exp_E_consensus_gquad**(unsigned int L, unsigned int l[3], *vrna_exp_param_t* *pf, unsigned int position, unsigned int length, unsigned int n_seq, const short **S, const unsigned int **a2s)

```
#include <ViennaRNA/eval/gquad.h>
```

int **vrna_mfe_gquad_internal_loop**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j)

```
#include <ViennaRNA/mfe/gquad.h>
```

FLT_OR_DBL **vrna_gq_int_loop_pf**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j)

```
#include <ViennaRNA/partfunc/gquad.h>
```

int ***vrna_gq_int_loop_subopt**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int **p_p, unsigned int **q_p, int threshold)

```
#include <ViennaRNA/subopt/gquad.h>
```

void **get_gquad_pattern_exhaustive**(short *S, unsigned int i, unsigned int j, *vrna_param_t* *P, unsigned int *L, unsigned int *l, int threshold)

```
#include <ViennaRNA/subopt/gquad.h>
```

unsigned int **get_gquad_count**(short *S, unsigned int i, unsigned int j)

```
#include <ViennaRNA/subopt/gquad.h>
```

Dynamic Programming Matrices

Functions

vrna_smx_csr_int_t ***vrna_mfe_gquad_mx**(*vrna_fold_compound_t* *fc)

Get G-Quadruplexes (MFE).

```
#include <ViennaRNA/mfe/gquad.h>
```

This function yields a sparse, two-dimensional matrix G that at position $G(i, j)$ stores the minimum free energy of a G-Quadruplex that starts at i and ends at j.

Parameters

- **fc** – The fold_compound

Returns

A sparse matrix with all MFE G-Quadruplexes

int **get_gquad_L_matrix(short *S, int start, int maxdist, int n, int **g, *vrna_param_t* *P)

```
#include <ViennaRNA/mfe/gquad.h>
```

void **vrna_gquad_mx_local_update(***vrna_fold_compound_t* *fc, int start)

```
#include <ViennaRNA/mfe/gquad.h>
```

vrna_smx_csr_FLT_OR_DBL_t ***vrna_gq_pos_pf(***vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/partfunc/gquad.h>
```

Backtracking

Functions

int **vrna_bt_gquad(***vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int *L, unsigned int l[3])

```
#include <ViennaRNA/backtrack/gquad.h>
```

int **vrna_bt_gquad_mfe(***vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, *vrna_bps_t* bp_stack)

```
#include <ViennaRNA/backtrack/gquad.h>
```

int **vrna_bt_gquad_internal(***vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, int en, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack)

```
#include <ViennaRNA/backtrack/gquad.h>
```

Parsing

Functions

void **get_gquad_pattern_pf(**short *S, int i, int j, *vrna_exp_param_t* *pf, int *L, int l[3])

```
#include <ViennaRNA/partfunc/gquad.h>
```

void **vrna_get_gquad_pattern_pf(***vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int *L, unsigned int[3])

```
#include <ViennaRNA/partfunc/gquad.h>
```

unsigned int **vrna_gq_parse(**const char *db_string, unsigned int *L, unsigned int l[3])

Parse a G-Quadruplex from a dot-bracket structure string.

```
#include <ViennaRNA/structures/dotbracket.h>
```

Given a dot-bracket structure (possibly) containing gquads encoded by '+' signs (and an optional '~' end sign, find first gquad, return end position (1-based) or 0 if none found. Upon return L and l[] contain the number of stacked layers, as well as the lengths of the linker regions.

To parse a string with many gquads, call *vrna_gq_parse()* repeatedly e.g.

```
end1 = vrna_gq_parse(struc, &L, l); ... ;
end2 = vrna_gq_parse(struc+end1, &L, l); end2+=end1; ... ;
end3 = vrna_gq_parse(struc+end2, &L, l); end3+=end2; ... ;
```

Note

For circular RNAs and G-Quadruplexes spanning the n,1-junction the sum of linkers and g-runs is lower than the end position. This condition can be used to check whether or not to accept a G-Quadruplex parsed from the dot-bracket string. Also note, that such n,1-junction spanning G-Quadruplexes must end with a ~ sign, to be unambigous.

Parameters

- **db_string** – The input structure in dot-bracket notation
- **L** – A pointer to an unsigned integer to store the layer (stack) size
- **l** – An array of three values to store the respective linker lenghts

Returns

The end position of the G-Quadruplex (1-based) or 0 if not found

void **vrna_db_insert_gq**(char *db, unsigned int i, unsigned int L, unsigned int l[3], unsigned int n)

```
#include <ViennaRNA/structures/dotbracket.h>
```

plist ***get_plist_gquad_from_db**(const char *structure, float pr)

```
#include <ViennaRNA/structures/problist.h>
```

Other

Functions

plist ***get_plist_gquad_from_pr**(short *S, int gi, int gj, vrna_smx_csr_FLT_OR_DBL_t *q_gq, *FLT_OR_DBL* *probs, *FLT_OR_DBL* *scale, *vrna_exp_param_t* *pf)

```
#include <ViennaRNA/partfunc/gquad.h>
```

vrna_ep_t ***vrna_plist_gquad_from_pr**(*vrna_fold_compound_t* *fc, int gi, int gj)

```
#include <ViennaRNA/partfunc/gquad.h>
```

plist ***get_plist_gquad_from_pr_max**(short *S, int gi, int gj, vrna_smx_csr_FLT_OR_DBL_t *q_gq, *FLT_OR_DBL* *probs, *FLT_OR_DBL* *scale, int *L, int l[3], *vrna_exp_param_t* *pf)

```
#include <ViennaRNA/partfunc/gquad.h>
```

vrna_ep_t ***vrna_plist_gquad_from_pr_max**(*vrna_fold_compound_t* *fc, unsigned int gi, unsigned int gj, unsigned int *Lmax, unsigned int lmax[3])

```
#include <ViennaRNA/partfunc/gquad.h>
```

4.1.13 RNA Folding Grammar

Secondary Structure Constraints

Secondary structure constraints provide an easy control of which structures the prediction algorithms actually include into their solution space and how these structures are evaluated.

General API symbols

Defines

VRNA_CONSTRAINT_FILE

Flag for *vrna_constraints_add()* to indicate that constraints are present in a text file.

```
#include <ViennaRNA/constraints/basic.h>
```

Deprecated:

Use 0 instead!

See also

vrna_constraints_add()

VRNA_CONSTRAINT_SOFT_MFE

Indicate generation of constraints for MFE folding.

```
#include <ViennaRNA/constraints/basic.h>
```

Deprecated:

This flag has no meaning anymore, since constraints are now always stored! (since v2.2.6)

VRNA_CONSTRAINT_SOFT_PF

Indicate generation of constraints for partition function computation.

```
#include <ViennaRNA/constraints/basic.h>
```

Deprecated:

Use *VRNA_OPTION_PF* instead!

VRNA_DECOMP_PAIR_HP

Flag passed to generic softt constraints callback to indicate hairpin loop decomposition step.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a hairpin loop enclosed by the base pair (i, j).

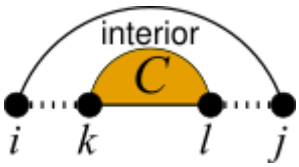


VRNA_DECOMP_PAIR_IL

Indicator for internal loop decomposition step.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates an internal loop enclosed by the base pair (i, j), and enclosing the base pair (k, l).



VRNA_DECOMP_PAIR_ML

Indicator for multibranch loop decomposition step.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a multibranch loop enclosed by the base pair (i, j), and consisting of some enclosed multi loop content from k to l.

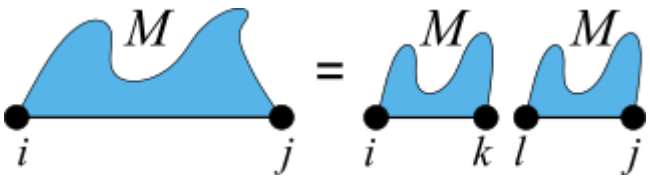


VRNA_DECOMP_ML_ML_ML

Indicator for decomposition of multibranch loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a multibranch loop part in the interval $[i : j]$, which will be decomposed into two multibranch loop parts $[i : k]$, and $[l : j]$.

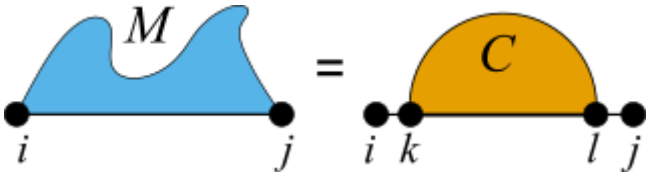


VRNA_DECOMP_ML_STEM

Indicator for decomposition of multibranch loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a multibranch loop part in the interval $[i : j]$, which will be considered a single stem branching off with base pair (k, l).

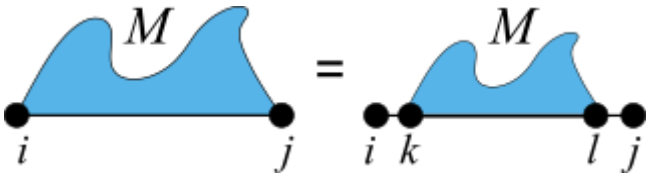


VRNA_DECOMP_ML_ML

Indicator for decomposition of multibranch loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a multibranch loop part in the interval $[i : j]$, which will be decomposed into a (usually) smaller multibranch loop part $[k : l]$.

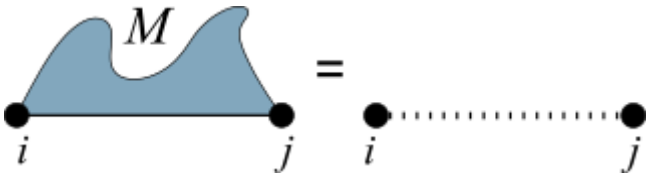


VRNA_DECOMP_ML_UP

Indicator for decomposition of multibranch loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a multibranch loop part in the interval $[i : j]$, which will be considered a multibranch loop part that only consists of unpaired nucleotides.

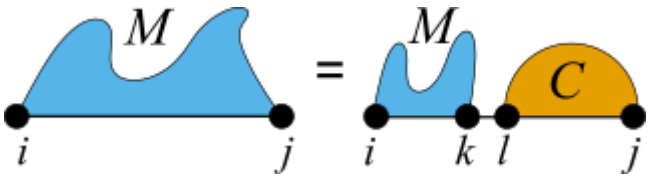


VRNA_DECOMP_ML_ML_STEM

Indicator for decomposition of multibranch loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a multibranch loop part in the interval $[i : j]$, which will decomposed into a multibranch loop part $[i : k]$, and a stem with enclosing base pair (l, j).

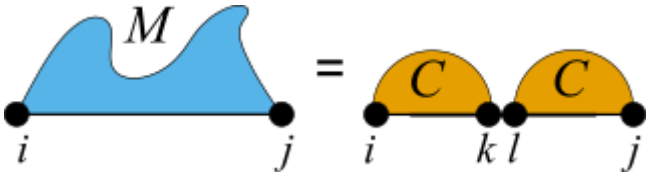


VRNA_DECOMP_ML_COAXIAL

Indicator for decomposition of multibranch loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a multibranch loop part in the interval $[i : j]$, where two stems with enclosing pairs (i, k) and (l, j) are coaxially stacking onto each other.

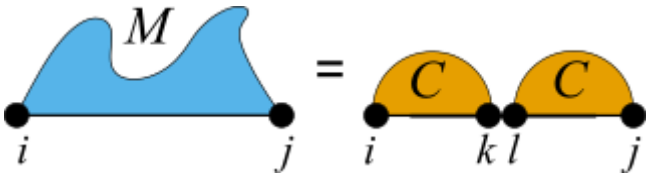


VRNA_DECOMP_ML_COAXIAL_ENC

Indicator for decomposition of multibranch loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates a multibranch loop part in the interval $[i : j]$, where two stems with enclosing pairs (i, k) and (l, j) are coaxially stacking onto each other.

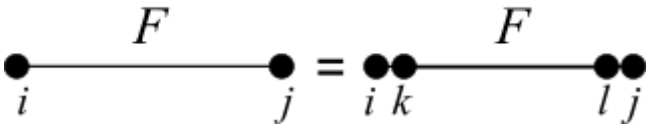


VRNA_DECOMP_EXT_EXT

Indicator for decomposition of exterior loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates an exterior loop part in the interval $[i : j]$, which will be decomposed into a (usually) smaller exterior loop part $[k : l]$.

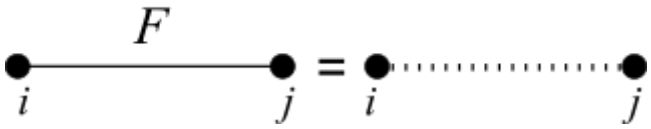


VRNA_DECOMP_EXT_UP

Indicator for decomposition of exterior loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates an exterior loop part in the interval $[i : j]$, which will be considered as an exterior loop component consisting of only unpaired nucleotides.

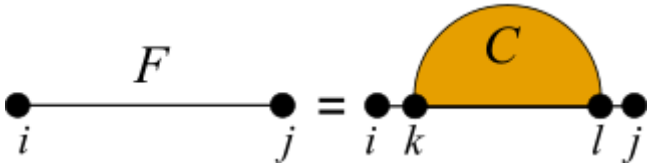


VRNA_DECOMP_EXT_STEM

Indicator for decomposition of exterior loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates an exterior loop part in the interval $[i : j]$, which will be considered a stem with enclosing pair (k, l).

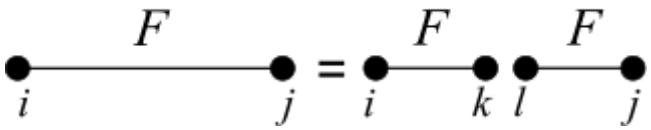


VRNA_DECOMP_EXT_EXT_EXT

Indicator for decomposition of exterior loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates an exterior loop part in the interval $[i : j]$, which will be decomposed into two exterior loop parts $[i : k]$ and $[l : j]$.

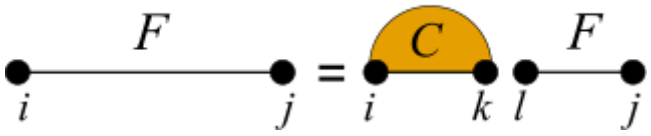


VRNA_DECOMP_EXT_STEM_EXT

Indicator for decomposition of exterior loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates an exterior loop part in the interval $[i : j]$, which will be decomposed into a stem branching off with base pair (i, k), and an exterior loop part $[l : j]$.



VRNA_DECOMP_EXT_STEM_OUTSIDE

Indicator for decomposition of exterior loop part.

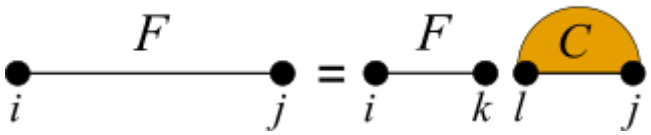
```
#include <ViennaRNA/constraints/basic.h>
```

VRNA_DECOMP_EXT_EXT_STEM

Indicator for decomposition of exterior loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates an exterior loop part in the interval $[i : j]$, which will be decomposed into an exterior loop part $[i : k]$, and a stem branching off with base pair (l, j).

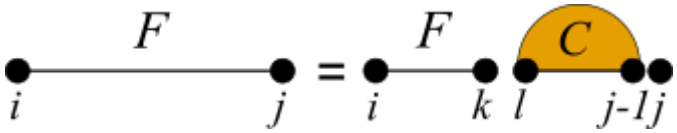


VRNA_DECOMP_EXT_EXT_STEM1

Indicator for decomposition of exterior loop part.

```
#include <ViennaRNA/constraints/basic.h>
```

This flag notifies the soft or hard constraint callback function that the current decomposition step evaluates an exterior loop part in the interval $[i : j]$, which will be decomposed into an exterior loop part $[i : k]$, and a stem branching off with base pair $(l, j - 1)$.



Functions

void **vrna_message_constraint_options**(unsigned int option)

Print a help message for pseudo dot-bracket structure constraint characters to stdout. (constraint support is specified by option parameter).

```
#include <ViennaRNA/constraints/hard.h>
```

Currently available options are:

VRNA_CONSTRAINT_DB_PIPE (paired with another base)

VRNA_CONSTRAINT_DB_DOT (no constraint at all)

VRNA_CONSTRAINT_DB_X (base must not pair)

VRNA_CONSTRAINT_DB_ANG_BRACK (paired downstream/upstream)

VRNA_CONSTRAINT_DB_RND_BRACK (base i pairs base j)

pass a collection of options as one value like this:

```
vrna_message_constraints(option_1 | option_2 | option_n)
```

See also

vrna_message_constraint_options_all(), *vrna_constraints_add()*, *VRNA_CONSTRAINT_DB*, *VRNA_CONSTRAINT_DB_PIPE*, *VRNA_CONSTRAINT_DB_DOT*, *VRNA_CONSTRAINT_DB_X*, VRNA_CONSTRAINT_DB_ANG_BRACK,

VRNA_CONSTRAINT_DB_RND_BRACK, *VRNA_CONSTRAINT_DB_INTERMOL*, *VRNA_CONSTRAINT_DB_INTRAMOL*

Parameters

- **option** – Option switch that tells which constraint help will be printed

void **vrna_message_constraint_options_all**(void)

Print structure constraint characters to stdout (full constraint support).

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_message_constraint_options(), *vrna_constraints_add()*, *VRNA_CONSTRAINT_DB*, *VRNA_CONSTRAINT_DB_PIPE*, *VRNA_CONSTRAINT_DB_DOT*, *VRNA_CONSTRAINT_DB_X*, VRNA_CONSTRAINT_DB_ANG_BRACK, *VRNA_CONSTRAINT_DB_RND_BRACK*, *VRNA_CONSTRAINT_DB_INTERMOL*, *VRNA_CONSTRAINT_DB_INTRAMOL*

Hard Constraints API

Functions and data structures for secondary structure hard constraints

Defines

VRNA_CONSTRAINT_DB

Flag for *vrna_constraints_add()* to indicate that constraint is passed in pseudo dot-bracket notation.

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_constraints_add(), *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_DB_ENFORCE_BP

Switch for dot-bracket structure constraint to enforce base pairs.

```
#include <ViennaRNA/constraints/hard.h>
```

This flag should be used to really enforce base pairs given in dot-bracket constraint rather than just weakly-enforcing them.

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_DB_PIPE

Flag that is used to indicate the pipe '|' sign in pseudo dot-bracket notation of hard constraints.

```
#include <ViennaRNA/constraints/hard.h>
```

Use this definition to indicate the pipe sign '|' (paired with another base)

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_DB_DOT

dot '.' switch for structure constraints (no constraint at all)

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_DB_X

'x' switch for structure constraint (base must not pair)

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_DB_RND_BRACK

round brackets '(',')' switch for structure constraint (base i pairs base j)

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_DB_INTRAMOL

Flag that is used to indicate the character 'l' in pseudo dot-bracket notation of hard constraints.

```
#include <ViennaRNA/constraints/hard.h>
```

Use this definition to indicate the usage of 'l' character (intramolecular pairs only)

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_DB_INTERMOL

Flag that is used to indicate the character 'e' in pseudo dot-bracket notation of hard constraints.

```
#include <ViennaRNA/constraints/hard.h>
```

Use this definition to indicate the usage of 'e' character (intermolecular pairs only)

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_DB_GQUAD

'+' switch for structure constraint (base is involved in a gquad)

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

Warning

This flag is for future purposes only! No implementation recognizes it yet.

VRNA_CONSTRAINT_DB_WUSS

Flag to indicate Washington University Secondary Structure (WUSS) notation of the hard constraint string.

```
#include <ViennaRNA/constraints/hard.h>
```

This secondary structure notation for RNAs is usually used as consensus secondary structure (SS_cons) entry in Stockholm formatted files

VRNA_CONSTRAINT_DB_DEFAULT

Switch for dot-bracket structure constraint with default symbols.

```
#include <ViennaRNA/constraints/hard.h>
```

This flag conveniently combines all possible symbols in dot-bracket notation for hard constraints and *VRNA_CONSTRAINT_DB*

See also

vrna_hc_add_from_db(), *vrna_constraints_add()*, *vrna_message_constraint_options()*, *vrna_message_constraint_options_all()*

VRNA_CONSTRAINT_CONTEXT_EXT_LOOP

Hard constraints flag, base pair in the exterior loop.

```
#include <ViennaRNA/constraints/hard.h>
```

VRNA_CONSTRAINT_CONTEXT_HP_LOOP

Hard constraints flag, base pair encloses hairpin loop.

```
#include <ViennaRNA/constraints/hard.h>
```

VRNA_CONSTRAINT_CONTEXT_INT_LOOP

Hard constraints flag, base pair encloses an internal loop.

```
#include <ViennaRNA/constraints/hard.h>
```

VRNA_CONSTRAINT_CONTEXT_INT_LOOP_ENC

Hard constraints flag, base pair encloses a multi branch loop.

```
#include <ViennaRNA/constraints/hard.h>
```

VRNA_CONSTRAINT_CONTEXT_MB_LOOP

Hard constraints flag, base pair is enclosed in an internal loop.

```
#include <ViennaRNA/constraints/hard.h>
```

VRNA_CONSTRAINT_CONTEXT_MB_LOOP_ENC

Hard constraints flag, base pair is enclosed in a multi branch loop.

```
#include <ViennaRNA/constraints/hard.h>
```

VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS

Constraint context flag indicating any loop context.

```
#include <ViennaRNA/constraints/hard.h>
```

Typedefs

typedef struct *vrna_hc_s* **vrna_hc_t**

Typename for the hard constraints data structure *vrna_hc_s*.

```
#include <ViennaRNA/constraints/hard.h>
```

typedef struct *vrna_hc_up_s* **vrna_hc_up_t**

Typename for the single nucleotide hard constraint data structure *vrna_hc_up_s*.

```
#include <ViennaRNA/constraints/hard.h>
```

typedef unsigned char (***vrna_hc_eval_f**)(int i, int j, int k, int l, unsigned char d, void *data)

Callback to evaluate whether or not a particular decomposition step is contributing to the solution space.

```
#include <ViennaRNA/constraints/hard.h>
```

This is the prototype for callback functions used by the folding recursions to evaluate generic hard constraints. The first four parameters passed indicate the delimiting nucleotide positions of the decomposition, and the parameter `denotes` the decomposition step. The last parameter `data` is the auxiliary data structure associated to the hard constraints via *vrna_hc_add_data()*, or NULL if no auxiliary data was added.

Notes on Callback Functions:

This callback enables one to over-rule default hard constraints in secondary structure decompositions.

See also

VRNA_DECOMP_PAIR_HP, *VRNA_DECOMP_PAIR_IL*, *VRNA_DECOMP_PAIR_ML*, *VRNA_DECOMP_ML_ML_ML*, *VRNA_DECOMP_ML_STEM*, *VRNA_DECOMP_ML_ML*, *VRNA_DECOMP_ML_UP*, *VRNA_DECOMP_ML_ML_STEM*, *VRNA_DECOMP_ML_COAXIAL*, *VRNA_DECOMP_EXT_EXT*, *VRNA_DECOMP_EXT_UP*, *VRNA_DECOMP_EXT_STEM*, *VRNA_DECOMP_EXT_EXT_EXT*, *VRNA_DECOMP_EXT_STEM_EXT*, *VRNA_DECOMP_EXT_EXT_STEM*, *VRNA_DECOMP_EXT_EXT_STEM1*, vrna_hc_add_f(), *vrna_hc_add_data()*

Param i
Left (5') delimiter position of substructure

Param j
Right (3') delimiter position of substructure

Param k
Left delimiter of decomposition

Param l
Right delimiter of decomposition

Param d
Decomposition step indicator

Param data
Auxiliary data

Return
A non-zero value if the decomposition is valid, 0 otherwise

Functions

void **vrna_constraints_add**(*vrna_fold_compound_t* *fc, const char *constraint, unsigned int options)

Add constraints to a *vrna_fold_compound_t* data structure.

```
#include <ViennaRNA/constraints/basic.h>
```

Use this function to add/update the hard/soft constraints The function allows for passing a string 'constraint' that can either be a filename that points to a constraints definition file or it may be a pseudo dot-bracket notation indicating hard constraints. For the latter, the user has to pass the *VRNA_CONSTRAINT_DB* option. Also, the user has to specify, which characters are allowed to be interpreted as constraints by passing the corresponding options via the third parameter.

The following is an example for adding hard constraints given in pseudo dot-bracket notation. Here, `fc` is the *vrna_fold_compound_t* object, `structure` is a char array with the hard constraint in dot-bracket notation, and `enforceConstraints` is a flag indicating whether or not constraints for base pairs should be enforced instead of just doing a removal of base pair that conflict with the constraint.

```
unsigned int constraint_options = VRNA_CONSTRAINT_DB_DEFAULT;

if (enforceConstraints)
        constraint_options |= VRNA_CONSTRAINT_DB_ENFORCE_BP;

if (canonicalBPonly)
        constraint_options |= VRNA_CONSTRAINT_DB_CANONICAL_BP;

vrna_constraints_add(fc, (const char *)cstruc, constraint_options);
```

In constrat to the above, constraints may also be read from file:

```
vrna_constraints_add(fc, constraints_file, VRNA_OPTION_DEFAULT);
```

See also

vrna_hc_add_from_db(), *vrna_hc_add_up()*, *vrna_hc_add_up_batch()*, vrna_hc_add_bp_unspecific(), *vrna_hc_add_bp()*, *vrna_hc_init()*, *vrna_sc_set_up()*, *vrna_sc_set_bp()*, *vrna_sc_add_SHAPE_deigan()*, *vrna_sc_add_SHAPE_zarringhalam()*, *vrna_hc_free()*, *vrna_sc_free()*, *VRNA_CONSTRAINT_DB*, *VRNA_CONSTRAINT_DB_DEFAULT*, *VRNA_CONSTRAINT_DB_PIPE*, *VRNA_CONSTRAINT_DB_DOT*, *VRNA_CONSTRAINT_DB_X*, VRNA_CONSTRAINT_DB_ANG_BRACK, *VRNA_CONSTRAINT_DB_RND_BRACK*, *VRNA_CONSTRAINT_DB_INTRAMOL*, *VRNA_CONSTRAINT_DB_INTERMOL*, *VRNA_CONSTRAINT_DB_GQUAD*

Parameters

- **fc** – The fold compound
- **constraint** – A string with either the filename of the constraint definitions or a pseudo dot-bracket notation of the hard constraint. May be NULL.
- **options** – The option flags

void **vrna_hc_init**(*vrna_fold_compound_t* *fc)

Initialize/Reset hard constraints to default values.

```
#include <ViennaRNA/constraints/hard.h>
```

This function resets the hard constraints to their default values, i.e. all positions may be unpaired in all contexts, and base pairs are allowed in all contexts, if they resemble canonical pairs. Previously set hard constraints will be removed before initialization.

SWIG Wrapper Notes:

This function is attached as method `hc_init()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.hc_init()* in the *The RNA Python module* .

See also

vrna_hc_add_bp(), *vrna_hc_add_bp_nonspecific()*, *vrna_hc_add_up()*

Parameters

- **fc** – The fold compound

void **vrna_hc_add_up**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned char option)

Make a certain nucleotide unpaired.

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_hc_add_bp(), *vrna_hc_add_bp_nonspecific()*, *vrna_hc_init()*, *VRNA_CONSTRAINT_CONTEXT_EXT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_HP_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP*, *VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*

Parameters

- **fc** – The *vrna_fold_compound_t* the hard constraints are associated with
- **i** – The position that needs to stay unpaired (1-based)
- **option** – The options flag indicating how/where to store the hard constraints

int **vrna_hc_add_up_strand**(*vrna_fold_compound_t* *fc, unsigned int i, int strand, unsigned char option)

Make a certain nucleotide unpaired.

```
#include <ViennaRNA/constraints/hard.h>
```

This function puts a constraint onto a single nucleotide to limit or enforce its structural context. The position `i` should be given in local coordinates relative to the strand `strand` it corresponds to.

See also

vrna_hc_add_bp(), *vrna_hc_add_bp_nonspecific()*, *vrna_hc_init()*, *VRNA_CONSTRAINT_CONTEXT_EXT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_HP_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP*, *VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*

Note

A negative value for the strand number `strand` indicates autodetection of the strand number assuming that coordinate i is given as global coordinates for the (current) concatenation of all strands. In this case, the function behaves exactly as *vrna_hc_add_up()*.

Parameters

- **fc** – The *vrna_fold_compound_t* the hard constraints are associated with
- **i** – The position that needs to stay unpaired (1-based)
- **strand** – The strand number of nucleotide `i` (0-based, negative value for autodetect)
- **option** – The options flag indicating how/where to store the hard constraints

int **vrna_hc_add_up_batch**(*vrna_fold_compound_t* *fc, *vrna_hc_up_t* *constraints)

Apply a list of hard constraints for single nucleotides.

```
#include <ViennaRNA/constraints/hard.h>
```

Parameters

- **fc** – The *vrna_fold_compound_t* the hard constraints are associated with
- **constraints** – The list off constraints to apply, last entry must have position attribute set to 0

int **vrna_hc_add_bp**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned char option)

Favorize/Enforce a certain base pair (i,j).

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_hc_add_bp_nonspecific(), *vrna_hc_add_up()*, *vrna_hc_init()*, *VRNA_CONSTRAINT_CONTEXT_EXT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_HP_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP_ENC*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP_ENC*, VRNA_CONSTRAINT_CONTEXT_ENFORCE, *VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*

Parameters

- **fc** – The *vrna_fold_compound_t* the hard constraints are associated with

- **i** – The 5' located nucleotide position of the base pair (1-based)
- **j** – The 3' located nucleotide position of the base pair (1-based)
- **option** – The options flag indicating how/where to store the hard constraints

int **vrna_hc_add_bp_strand**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, int strand_i, int strand_j, unsigned char option)

Favorize/Enforce a certain base pair (i,j) where i and j may point to different strands.

```
#include <ViennaRNA/constraints/hard.h>
```

This function adds a base pair constraint for pair (i,j), where the positions i and j are relative to the RNA strands i and j correspond to. For each strand `strand_i` and `strand_j` positions are 1-based. For instance, if position 5 of strand 0 must pair with position 10 of strand 1, the function call would be

```
vrna_hc_add_bp(fc, 5, 10, 0, 1, VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS | VRNA_
↪CONSTRAINT_CONTEXT_ENFORCE);
```

Negative values for the strand numbers `strand_i` and `strand_j` indicate autodetection of the strand number assuming that coordinates i and/or j are given as global coordinates for the (current) concatenation of all strands

See also

vrna_hc_add_bp_nonspecific(), *vrna_hc_add_up()*, *vrna_hc_add_bp()*, *vrna_hc_init()*, *VRNA_CONSTRAINT_CONTEXT_EXT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_HP_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP_ENC*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP_ENC*, VRNA_CONSTRAINT_CONTEXT_ENFORCE, *VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*

Parameters

- **fc** – The *vrna_fold_compound_t* the hard constraints are associated with
- **i** – The 5' located nucleotide position of the base pair (1-based, relative to `strand_i`)
- **j** – The 3' located nucleotide position of the base pair (1-based, relative to `strand_j`)
- **strand_i** – The strand number of pairing partner `i` (0-based, negative value for autodetect)
- **strand_j** – The strand number of pairing partner `j` (0-based, negative value for autodetect)
- **option** – The option flag(s) indicating loop types and enforcement of the constraint

void **vrna_hc_add_bp_nonspecific**(*vrna_fold_compound_t* *fc, unsigned int i, int d, unsigned char option)

Enforce a nucleotide to be paired (upstream/downstream).

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_hc_add_bp(), *vrna_hc_add_up()*, *vrna_hc_init()*, *VRNA_CONSTRAINT_CONTEXT_EXT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_HP_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP_ENC*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP_ENC*, *VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*

Parameters

- **fc** – The *vrna_fold_compound_t* the hard constraints are associated with
- **i** – The position that needs to stay unpaired (1-based)
- **d** – The direction of base pairing ($d < 0$: pairs upstream, $d > 0$: pairs downstream, $d == 0$: no direction)
- **option** – The options flag indicating in which loop type context the pairs may appear

void **vrna_hc_free**(*vrna_hc_t* *hc)

Free the memory allocated by a *vrna_hc_t* data structure.

```
#include <ViennaRNA/constraints/hard.h>
```

Use this function to free all memory that was allocated for a data structure of type *vrna_hc_t* .

See also

get_hard_constraints(), *vrna_hc_t*

void **vrna_hc_add_data**(*vrna_fold_compound_t* *fc, void *data, *vrna_auxdata_free_f* f)

Add an auxiliary data structure for the generic hard constraints callback function.

```
#include <ViennaRNA/constraints/hard.h>
```

See also

vrna_hc_add_f()

Parameters

- **fc** – The fold compound the generic hard constraint function should be bound to
- **data** – A pointer to the data structure that holds required data for function 'f'
- **f** – A pointer to a function that free's the memory occupied by `data` (Maybe `NULL`)

int **vrna_hc_add_from_db**(*vrna_fold_compound_t* *fc, const char *constraint, unsigned int options)

Add hard constraints from pseudo dot-bracket notation.

```
#include <ViennaRNA/constraints/hard.h>
```

This function allows one to apply hard constraints from a pseudo dot-bracket notation. The `options` parameter controls, which characters are recognized by the parser. Use the *VRNA_CONSTRAINT_DB_DEFAULT* convenience macro, if you want to allow all known characters

SWIG Wrapper Notes:

This function is attached as method `hc_add_from_db()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.hc_add_from_db()`* in the *The RNA Python module* .

See also

VRNA_CONSTRAINT_DB_PIPE, *VRNA_CONSTRAINT_DB_DOT*, *VRNA_CONSTRAINT_DB_X*, VRNA_CONSTRAINT_DB_ANG_BRACK, *VRNA_CONSTRAINT_DB_RND_BRACK*, *VRNA_CONSTRAINT_DB_INTRAMOL*, *VRNA_CONSTRAINT_DB_INTERMOL*, *VRNA_CONSTRAINT_DB_GQUAD*

Parameters

- **fc** – The fold compound
- **constraint** – A pseudo dot-bracket notation of the hard constraint.
- **options** – The option flags

struct **vrna_hc_s**

#include <ViennaRNA/constraints/hard.h> The hard constraints data structure.

The content of this data structure determines the decomposition pattern used in the folding recursions. Attribute 'matrix' is used as source for the branching pattern of the decompositions during all folding recursions. Any entry in matrix[i,j] consists of the 6 LSB that allows one to distinguish the following types of base pairs:

- in the exterior loop (*VRNA_CONSTRAINT_CONTEXT_EXT_LOOP*)
- enclosing a hairpin (*VRNA_CONSTRAINT_CONTEXT_HP_LOOP*)
- enclosing an internal loop (*VRNA_CONSTRAINT_CONTEXT_INT_LOOP*)
- enclosed by an exterior loop (*VRNA_CONSTRAINT_CONTEXT_INT_LOOP_ENC*)
- enclosing a multi branch loop (*VRNA_CONSTRAINT_CONTEXT_MB_LOOP*)
- enclosed by a multi branch loop (*VRNA_CONSTRAINT_CONTEXT_MB_LOOP_ENC*)

The four linear arrays 'up_xxx' provide the number of available unpaired nucleotides (including position i) 3' of each position in the sequence.

See also

vrna_hc_init(), *vrna_hc_free()*, *VRNA_CONSTRAINT_CONTEXT_EXT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_HP_LOOP*, *VRNA_CONSTRAINT_CONTEXT_INT_LOOP*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP*, *VRNA_CONSTRAINT_CONTEXT_MB_LOOP_ENC*

Public Members

vrna_hc_type_e **type**

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned int **n**

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned int **strands**

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned int ***sn**

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned char **state**

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned char ***mx**

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned char ****matrix_local**

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned int ***up_ext**

A linear array that holds the number of allowed unpaired nucleotides in an exterior loop.

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned int ***up_hp**

A linear array that holds the number of allowed unpaired nucleotides in a hairpin loop.

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned int ***up_int**

A linear array that holds the number of allowed unpaired nucleotides in an internal loop.

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned int ***up_ml**

A linear array that holds the number of allowed unpaired nucleotides in a multi branched loop.

```
#include <ViennaRNA/constraints/hard.h>
```

vrna_hc_eval_f **f**

A function pointer that returns whether or not a certain decomposition may be evaluated.

```
#include <ViennaRNA/constraints/hard.h>
```

void ***data**

A pointer to some structure where the user may store necessary data to evaluate its generic hard constraint function.

```
#include <ViennaRNA/constraints/hard.h>
```

vrna_auxdata_free_f **free_data**

A pointer to a function to free memory occupied by auxiliary data.

```
#include <ViennaRNA/constraints/hard.h>
```

The function this pointer is pointing to will be called upon destruction of the *vrna_hc_s*, and provided with the *vrna_hc_s.data* pointer that may hold auxiliary data. Hence, to avoid leaking memory, the user may use this pointer to free memory occupied by auxiliary data.

vrna_hc_depot_t ***depot**

```
#include <ViennaRNA/constraints/hard.h>
```

vrna_hc_eval_loop_f **eval_ext**

```
#include <ViennaRNA/constraints/hard.h>
```

vrna_hc_eval_hp_f **eval_hp**

```
#include <ViennaRNA/constraints/hard.h>
```

vrna_hc_eval_int_f **eval_int**

```
#include <ViennaRNA/constraints/hard.h>
```

vrna_hc_eval_loop_f **eval_mb**

```
#include <ViennaRNA/constraints/hard.h>
```

struct **vrna_hc_up_s**

#include <ViennaRNA/constraints/hard.h> A single hard constraint for a single nucleotide.

Public Members

int **position**

The sequence position (1-based).

```
#include <ViennaRNA/constraints/hard.h>
```

int **strand**

```
#include <ViennaRNA/constraints/hard.h>
```

unsigned char **options**

The hard constraint option.

```
#include <ViennaRNA/constraints/hard.h>
```

Soft Constraints API

Functions and data structures for secondary structure soft constraints.

Common API symbols

Typedefs

typedef struct *vrna_sc_s* **vrna_sc_t**

Typename for the soft constraints data structure *vrna_sc_s*.

```
#include <ViennaRNA/constraints/soft.h>
```

Enums

enum **vrna_sc_type_e**

The type of a soft constraint.

```
#include <ViennaRNA/constraints/soft.h>
```

Values:

enumerator **VRNA_SC_DEFAULT**

Default Soft Constraints.

enumerator **VRNA_SC_WINDOW**

Soft Constraints suitable for local structure prediction using window approach.

See also

vrna_mfe_window(), vrna_mfe_window_zscore(), *pfl_fold()*

Functions

void **vrna_sc_init**(*vrna_fold_compound_t* *fc)

Initialize an empty soft constraints data structure within a *vrna_fold_compound_t*.

```
#include <ViennaRNA/constraints/soft.h>
```

This function adds a proper soft constraints data structure to the *vrna_fold_compound_t* data structure. If soft constraints already exist within the fold compound, they are removed.

SWIG Wrapper Notes:

This function is attached as method `sc_init()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.sc_init()`* in the *The RNA Python module* .

See also

vrna_sc_set_bp(), *vrna_sc_set_up()*, *vrna_sc_add_SHAPE_deigan()*, *vrna_sc_add_SHAPE_zarringhalam()*, *vrna_sc_remove()*, *vrna_sc_add_f()*, *vrna_sc_add_exp_f()*, vrna_sc_add_pre(), vrna_sc_add_post()

Note

Accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE* and *VRNA_FC_TYPE_COMPARATIVE*

Parameters

- **fc** – The *vrna_fold_compound_t* where an empty soft constraint feature is to be added to

int **vrna_sc_prepare**(*vrna_fold_compound_t* *fc, unsigned int options)

Prepare soft constraints.

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_update**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int options)

Update/prepare soft constraints for sliding-window computations.

```
#include <ViennaRNA/constraints/soft.h>
```

void **vrna_sc_remove**(*vrna_fold_compound_t* *fc)

Remove soft constraints from *vrna_fold_compound_t*.

```
#include <ViennaRNA/constraints/soft.h>
```

SWIG Wrapper Notes:

This function is attached as method `sc_remove()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.sc_remove()`* in the *The RNA Python module* .

Note

Accepts *vrna_fold_compound_t* of type *VRNA_FC_TYPE_SINGLE* and *VRNA_FC_TYPE_COMPARATIVE*

Parameters

- **fc** – The *vrna_fold_compound_t* possibly containing soft constraints

void **vrna_sc_free**(*vrna_sc_t* *sc)

Free memory occupied by a *vrna_sc_t* data structure.

```
#include <ViennaRNA/constraints/soft.h>
```

Parameters

- **sc** – The data structure to free from memory

int **vrna_sc_add_iip**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, unsigned int k, unsigned int l, double energy, unsigned int options)

```
#include <ViennaRNA/constraints/soft_iip.h>
```

struct **vrna_sc_bp_storage_t**

#include <ViennaRNA/constraints/soft.h> A base pair constraint.

Public Members

unsigned int **interval_start**

```
#include <ViennaRNA/constraints/soft.h>
```

unsigned int **interval_end**

```
#include <ViennaRNA/constraints/soft.h>
```

double **e**

```
#include <ViennaRNA/constraints/soft.h>
```

Energy contribution in units of 0.01 kcal/mol

struct **vrna_sc_s**

#include <ViennaRNA/constraints/soft.h> The soft constraints data structure.

Common data fields

const *vrna_sc_type_e* **type**

Type of the soft constraints data structure.

```
#include <ViennaRNA/constraints/soft.h>
```

unsigned int **n**

Length of the sequence this soft constraints data structure belongs to.

```
#include <ViennaRNA/constraints/soft.h>
```

unsigned char **state**

Current state of the soft constraints data structure.

```
#include <ViennaRNA/constraints/soft.h>
```

int ****energy_up**

Energy contribution for stretches of unpaired nucleotides.

```
#include <ViennaRNA/constraints/soft.h>
```

FLT_OR_DBL ****exp_energy_up**

Boltzmann Factors of the energy contributions for unpaired sequence stretches.

```
#include <ViennaRNA/constraints/soft.h>
```

double ***up_storage**

Storage container for energy contributions per unpaired nucleotide in units of 0.01 kcal/mol.

```
#include <ViennaRNA/constraints/soft.h>
```

vrna_sc_bp_storage_t ****bp_storage**

Storage container for energy contributions per base pair.

```
#include <ViennaRNA/constraints/soft.h>
```

int ***energy_stack**

Pseudo Energy contribution per base pair involved in a stack.

```
#include <ViennaRNA/constraints/soft.h>
```

FLT_OR_DBL ***exp_energy_stack**

Boltzmann weighted pseudo energy contribution per nucleotide involved in a stack.

```
#include <ViennaRNA/constraints/soft.h>
```

vrna_sc_iip_t **iip**

Data for interacting interval pairs.

```
#include <ViennaRNA/constraints/soft.h>
```

User-defined data fields

vrna_sc_f **f**

A function pointer used for pseudo energy contribution in MFE calculations.

```
#include <ViennaRNA/constraints/soft.h>
```

See also

vrna_sc_add_f()

vrna_sc_bt_f **bt**

A function pointer used to obtain backtraced base pairs in loop regions that were altered by soft constrained pseudo energy contributions.

```
#include <ViennaRNA/constraints/soft.h>
```

See also

vrna_sc_add_bt()

vrna_sc_exp_f **exp_f**

A function pointer used for pseudo energy contribution boltzmann factors in PF calculations.

```
#include <ViennaRNA/constraints/soft.h>
```

See also

vrna_sc_add_exp_f()

void ***data**

A pointer to the data object provided for for pseudo energy contribution functions of the generic soft constraints feature.

```
#include <ViennaRNA/constraints/soft.h>
```

vrna_auxdata_prepare_f **prepare_data**

```
#include <ViennaRNA/constraints/soft.h>
```

vrna_auxdata_free_f **free_data**

```
#include <ViennaRNA/constraints/soft.h>
```

Public Members

union *vrna_sc_s*.[anonymous] **[anonymous]**

```
#include <ViennaRNA/constraints/soft.h>
```

[union].__unnamed0__

Public Members

struct *vrna_sc_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/constraints/soft.h>
```

struct *vrna_sc_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/constraints/soft.h>
```

[struct].__unnamed0__.__unnamed0__

Global structure prediction data fields

int ***energy_bp**

Energy contribution for base pairs.

```
#include <ViennaRNA/constraints/soft.h>
```

FLT_OR_DBL ***exp_energy_bp**

Boltzmann Factors of the energy contribution for base pairs.

```
#include <ViennaRNA/constraints/soft.h>
```

[struct].__unnamed0__.__unnamed1__

Local structure prediction data fields

int ****energy_bp_local**

Energy contribution for base pairs (sliding window approach).

```
#include <ViennaRNA/constraints/soft.h>
```

FLT_OR_DBL ****exp_energy_bp_local**

Boltzmann Factors of the energy contribution for base pairs (sliding window approach).

```
#include <ViennaRNA/constraints/soft.h>
```

vrna_sc_s.[union].__unnamed0__

Public Members

struct *vrna_sc_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/constraints/soft.h>
```

struct *vrna_sc_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/constraints/soft.h>
```

[struct].__unnamed0__.__unnamed0__

Global structure prediction data fields

int ***energy_bp**

Energy contribution for base pairs.

```
#include <ViennaRNA/constraints/soft.h>
```

FLT_OR_DBL ***exp_energy_bp**

Boltzmann Factors of the energy contribution for base pairs.

```
#include <ViennaRNA/constraints/soft.h>
```

[struct].__unnamed0__.__unnamed1__

Local structure prediction data fields

int ****energy_bp_local**

Energy contribution for base pairs (sliding window approach).

```
#include <ViennaRNA/constraints/soft.h>
```

FLT_OR_DBL ****exp_energy_bp_local**

Boltzmann Factors of the energy contribution for base pairs (sliding window approach).

```
#include <ViennaRNA/constraints/soft.h>
```

vrna_sc_s.[struct].__unnamed0__.__unnamed0__

Global structure prediction data fields

int ***energy_bp**

Energy contribution for base pairs.

```
#include <ViennaRNA/constraints/soft.h>
```

FLT_OR_DBL ***exp_energy_bp**

Boltzmann Factors of the energy contribution for base pairs.

```
#include <ViennaRNA/constraints/soft.h>
```

vrna_sc_s.[struct].__unnamed0__.__unnamed1__

Local structure prediction data fields

int ****energy_bp_local**

Energy contribution for base pairs (sliding window approach).

```
#include <ViennaRNA/constraints/soft.h>
```

FLT_OR_DBL ****exp_energy_bp_local**

Boltzmann Factors of the energy contribution for base pairs (sliding window approach).

```
#include <ViennaRNA/constraints/soft.h>
```

Constraints for Unpaired Positions

Functions

int **vrna_sc_set_up**(*vrna_fold_compound_t* *fc, const *FLT_OR_DBL* *constraints, unsigned int options)

Set soft constraints for unpaired nucleotides.

```
#include <ViennaRNA/constraints/soft.h>
```

Values are stored without rounding to 0.01 kcal/mol. Partition function calculations use these values directly; MFE calculations round the accumulated contribution for each nucleotide to the nearest 0.01 kcal/mol before summing over stretches of unpaired nucleotides.

SWIG Wrapper Notes:

This function is attached as method `sc_set_up()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.sc_set_up()`* in the *The RNA Python module* .

See also

vrna_sc_add_up(), *vrna_sc_set_bp()*, *vrna_sc_add_bp()*

Note

This function replaces any pre-exisitng soft constraints with the ones supplied in `constraints`.

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **constraints** – A vector of pseudo free energies in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns

Non-zero on successful application of the constraint, 0 otherwise.

int **vrna_sc_set_up_comparative**(*vrna_fold_compound_t* *fc, const *FLT_OR_DBL* **constraints, unsigned int options)

Set soft constraints for unpaired nucleotides in comparative structure predictions.

```
#include <ViennaRNA/constraints/soft.h>
```

Use this function to set soft constraints for unpaired nucleotides for each sequence in the multiple sequence alignment (MSA). The constraints are provided as 0-based array of 1-based array with the actual pseudo energies, where the first dimension corresponds to the number (0-based) of the respective sequence in the alignment. If no constraints are provided for a particular sequence s in the MSA, the corresponding entry must be set to **NULL**.

See also

vrna_sc_set_up_comparative_seq(), *vrna_sc_add_up_comparative()*, *vrna_sc_set_bp_comparative()*, vrna_sc_add_stack_comparative(), vrna_sc_set_stack_comparative(), *vrna_sc_set_up()*

Note

This function replaces any pre-exisitng soft constraints with the ones supplied in `constraints`.

Warning

Pseudo energies for all sequences must be provided in sequence coordinates rather than alignment coordinates!

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **constraints** – A 0-based array of 1-based arrays with pseudo free energies in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns

The number of sequences in the MSA constraints have been applied to

int **vrna_sc_set_up_comparative_seq**(*vrna_fold_compound_t* *fc, unsigned int s, const *FLT_OR_DBL* *constraints, unsigned int options)

Set soft constraints for unpaired nucleotides in comparative structure predictions.

```
#include <ViennaRNA/constraints/soft.h>
```

This is a convenience wrapper for *vrna_sc_set_up_comparative()* where only one particular sequence `s` is provided with constraints.

See also

vrna_sc_set_up_comparative(), *vrna_sc_add_up_comparative_seq()*, *vrna_sc_set_bp_comparative_seq()*, vrna_sc_add_stack_comparative_seq(), vrna_sc_set_stack_comparative_seq(), *vrna_sc_set_up()*

Note

This function replaces any pre-exisitng soft constraints with the ones supplied in `constraints`.

Warning

Pseudo energies for all sequences must be provided in sequence coordinates rather than alignment coordinates!

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **s** – The 0-based number of the sequence in the alignment the constraints are provided for
- **constraints** – A 1-based arrays with pseudo free energies in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns
The number of sequences in the MSA constraints have been applied to

int **vrna_sc_add_up**(*vrna_fold_compound_t* *fc, unsigned int i, *FLT_OR_DBL* energy, unsigned int options)

Add soft constraints for unpaired nucleotides.

```
#include <ViennaRNA/constraints/soft.h>
```

Repeated calls accumulate contributions for the same nucleotide without rounding to 0.01 kcal/mol. Partition function calculations use the accumulated value; MFE calculations round it to the nearest 0.01 kcal/mol before summing over stretches of unpaired nucleotides. For example, adding 0.003 kcal/mol twice at one position gives a PF contribution of 0.006 kcal/mol and an MFE contribution of 0.01 kcal/mol.

SWIG Wrapper Notes:

This function is attached as an overloaded method `sc_add_up()` to objects of type `fold_compound`. The method either takes arguments for a single nucleotide i with the corresponding energy value:

```
fold_compound.sc_add_up(i, energy, options)
```

or an entire vector that stores free energy contributions for each nucleotide i with $1 \leq i \leq n$:

```
fold_compound.sc_add_bp(vector, options)
```

In both variants, the optional argument `options` defaults to *VRNA_OPTION_DEFAULT*. See, e.g. *RNA.fold_compound.sc_add_up()* in the *The RNA Python module* .

See also

vrna_sc_set_up(), *vrna_sc_add_bp()*, *vrna_sc_set_bp()*, *vrna_sc_add_up_comparative()*

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **i** – The nucleotide position the soft constraint is added for
- **energy** – The free energy (soft-constraint) in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns
Non-zero on successful application of the constraint, 0 otherwise.

int **vrna_sc_add_up_comparative**(*vrna_fold_compound_t* *fc, unsigned int *is, const *FLT_OR_DBL* *energies, unsigned int options)

Add soft constraints for unpaired nucleotides in comparative structure predictions.

```
#include <ViennaRNA/constraints/soft.h>
```

Use this function to add soft constraints for unpaired nucleotides for each sequence in the multiple sequence alignment (MSA). The constraints are provided as 0-based array of pseudo energies, one for each sequence in the MSA. If no constraints are provided for a particular sequence, the corrsponding `is` value must be 0.

See also

vrna_sc_set_up_comparative_seq(), *vrna_sc_add_up_comparative()*, *vrna_sc_set_bp_comparative()*, vrna_sc_add_stack_comparative(), vrna_sc_set_stack_comparative(), *vrna_sc_add_up()*

Warning

Pseudo energies for all sequences must be provided in sequence coordinates rather than alignment coordinates!

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **is** – A 0-based array of nucleotide positions the soft constraint is added for
- **energies** – A 0-based array of free energies (soft-constraint) in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns

Non-zero on successful application of the constraint, 0 otherwise.

int **vrna_sc_add_up_comparative_seq**(*vrna_fold_compound_t* *fc, unsigned int s, unsigned int i, const *FLT_OR_DBL* energy, unsigned int options)

Add soft constraints for unpaired nucleotides in comparative structure predictions.

```
#include <ViennaRNA/constraints/soft.h>
```

This is a convenience wrapper for *vrna_sc_add_up_comparative()* where only one particular sequence `s` is provided with constraints.

See also

vrna_sc_set_up_comparative_seq(), *vrna_sc_add_up_comparative()*, *vrna_sc_set_bp_comparative()*, vrna_sc_add_stack_comparative(), vrna_sc_set_stack_comparative(), *vrna_sc_add_up()*

Warning

Pseudo energies for all sequences must be provided in sequence coordinates rather than alignment coordinates!

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **s** – The 0-based number of the sequence in the alignment the constraints are provided for
- **i** – The 1-based nucleotide position the soft constraint is added for
- **energy** – The free energies (soft-constraint) in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns
Non-zero on successful application of the constraint, 0 otherwise.

Constraints for Base Pairs

Functions

int **vrna_sc_set_bp**(*vrna_fold_compound_t* *fc, const *FLT_OR_DBL* **constraints, unsigned int options)

Set soft constraints for paired nucleotides.

```
#include <ViennaRNA/constraints/soft.h>
```

Values are stored without rounding to 0.01 kcal/mol. Partition function calculations use these values directly; MFE calculations round the accumulated contribution for each base pair to the nearest 0.01 kcal/mol.

SWIG Wrapper Notes:

This function is attached as method `sc_set_bp()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.sc_set_bp()`* in the *The RNA Python module* .

See also

vrna_sc_add_bp(), *vrna_sc_set_up()*, *vrna_sc_add_up()*

Note

This function replaces any pre-exisitng soft constraints with the ones supplied in `constraints`.

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **constraints** – A two-dimensional array of pseudo free energies in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns
Non-zero on successful application of the constraint, 0 otherwise.

int **vrna_sc_set_bp_comparative**(*vrna_fold_compound_t* *fc, const *FLT_OR_DBL* ***constraints, unsigned int options)

Set soft constraints for paired nucleotides in comparative structure predictions.

```
#include <ViennaRNA/constraints/soft.h>
```

Similar to *vrna_sc_set_bp()* this function allows to set soft constraints $e_{i,j}^{\mathrm{BP}}$ for all base pairs (i, j) at once using a 1-based upper-triangular matrix E_{BP}. Since this function is supposed to be used for comparative structure predictions over a multiple sequence alignment (MSA), a 0-based array of matrices must be supplied as parameter `constraints`. If no constraints are to be used for sequence s in the MSA, the corresponding entry may be set to **NULL**.

See also

vrna_sc_set_bp_comparative_seq(), *vrna_sc_add_bp_comparative()*, *vrna_sc_set_up_comparative()*, *vrna_sc_add_up_comparative()*, vrna_sc_set_stack_comparative(), vrna_sc_add_stack_comparative(), *vrna_sc_set_bp()*

Note

This function replaces any pre-exisitng soft constraints with the ones supplied in `constraints`.

Warning

Currently, base pair constraints must be provided in alignment coordinates rather than sequence coordinates! This may change in the future!

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **constraints** – A 0-based array of 1-based two-dimensional arrays with pseudo free energies in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns

The number of sequences in the MSA constraints have been applied to

int **vrna_sc_set_bp_comparative_seq**(*vrna_fold_compound_t* *fc, unsigned int s, const *FLT_OR_DBL* **constraints, unsigned int options)

Set soft constraints for paired nucleotides in comparative structure predictions.

```
#include <ViennaRNA/constraints/soft.h>
```

This is a convenience wrapper for *vrna_sc_set_bp_comparative()* where only one particular sequence `s` is provided with constraints.

See also

vrna_sc_set_bp_comparative(), *vrna_sc_add_bp_comparative()*, *vrna_sc_set_up_comparative_seq()*, *vrna_sc_add_up_comparative_seq()*, vrna_sc_set_stack_comparative_seq(), vrna_sc_add_stack_comparative_seq(), *vrna_sc_set_bp()*

Note

This function replaces any pre-exisitng soft constraints with the ones supplied in `constraints`.

Warning

This function not only re-sets the constraints for sequence `s` in the MSA but will also remove all constraints for all other sequences!

Warning

Currently, base pair constraints must be provided in alignment coordinates rather than sequence coordinates! This may change in the future!

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **s** – The 0-based number of the sequence in the alignment the constraints are provided for
- **constraints** – A 1-based two-dimensional array of pseudo free energies in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns

The number of sequences in the MSA constraints have been applied to

int **vrna_sc_add_bp**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, *FLT_OR_DBL* energy, unsigned int options)

Add soft constraints for paired nucleotides.

```
#include <ViennaRNA/constraints/soft.h>
```

Repeated calls accumulate contributions for the same base pair without rounding to 0.01 kcal/mol. Partition function calculations use the accumulated value; MFE calculations round it to the nearest 0.01 kcal/mol. For example, adding 0.003 kcal/mol twice gives a PF contribution of 0.006 kcal/mol and an MFE contribution of 0.01 kcal/mol.

SWIG Wrapper Notes:

This function is attached as an overloaded method `sc_add_bp()` to objects of type `fold_compound`. The method either takes arguments for a single base pair (i,j) with the corresponding energy value:

```
fold_compound.sc_add_bp(i, j, energy, options)
```

or an entire 2-dimensional matrix with dimensions n x n that stores free energy contributions for any base pair (i,j) with $1 \leq i < j \leq n$:

```
fold_compound.sc_add_bp(matrix, options)
```

In both variants, the optional argument `options` defaults to *VRNA_OPTION_DEFAULT*. See, e.g. *RNA.fold_compound.sc_add_bp()* in the *The RNA Python module* .

See also

vrna_sc_set_bp(), *vrna_sc_set_up()*, *vrna_sc_add_up()*

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **i** – The 5' position of the base pair the soft constraint is added for
- **j** – The 3' position of the base pair the soft constraint is added for
- **energy** – The free energy (soft-constraint) in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns

Non-zero on successful application of the constraint, 0 otherwise.

int **vrna_sc_add_bp_comparative**(*vrna_fold_compound_t* *fc, unsigned int *is, unsigned int *js, const *FLT_OR_DBL* *energies, unsigned int options)

Add soft constraints for paired nucleotides in comparative structure predictions.

```
#include <ViennaRNA/constraints/soft.h>
```

Similar to *vrna_sc_add_bp()*, this function allows to add soft constraints $e_{i,j}^{\text{BP}}$ for all base pairs (i, j) in the multiple sequence alignment (MSA). The actual pairing partners i and j for each sequence in the MSA are provided in the form of 0-based arrays as parameters `is` and `js`. The corresponding energy contributions are provided as 0-based array in parameter `energies`. If no constraint is provided for sequence s in the MSA, the corresponding `is` value must be set to 0.

See also

vrna_sc_add_bp_comparative_seq(), *vrna_sc_set_bp_comparative()*, *vrna_sc_set_up_comparative()*, *vrna_sc_add_up_comparative()*, vrna_sc_set_stack_comparative(), vrna_sc_add_stack_comparative(), *vrna_sc_add_bp()*

Note

Consecutive calls of this function with the same `is` and `js` accumulate to corresponding `energies` values, i.e. energies are added up onto each other.

Warning

Currently, base pair constraints must be provided in alignment coordinates rather than sequence coordinates! This may change in the future!

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **is** – A 0-based array of 5' position of the base pairs the soft constraint is added for
- **js** – A 0-based array of 3' position of the base pairs the soft constraint is added for
- **energies** – A 0-based array of free energies (soft-constraint) in $kcal/mol$

- **options** – The options flag indicating how/where to store the soft constraints

Returns
The number of sequences in the MSA constraints have been applied to

int **vrna_sc_add_bp_comparative_seq**(*vrna_fold_compound_t* *fc, unsigned int s, unsigned int i, unsigned int j, *FLT_OR_DBL* energy, unsigned int options)

Add soft constraints for paired nucleotides in comparative structure predictions.

```
#include <ViennaRNA/constraints/soft.h>
```

This is a convenience wrapper for *vrna_sc_add_bp_comparative()* where only one particular sequence `s` is provided with constraints.

See also

vrna_sc_add_bp_comparative(), *vrna_sc_set_bp_comparative_seq()*, *vrna_sc_set_up_comparative_seq()*, *vrna_sc_add_up_comparative_seq()*, vrna_sc_set_stack_comparative_seq(), vrna_sc_add_stack_comparative_seq(), *vrna_sc_add_bp()*

Note

Consecutive calls of this function with the same `i` and `j` accumulate to corresponding `energies` values, i.e. energies are added up onto each other.

Warning

Currently, base pair constraints must be provided in alignment coordinates rather than sequence coordinates! This may change in the future!

Parameters

- **fc** – The *vrna_fold_compound_t* the soft constraints are associated with
- **s** – The 0-based number of the sequence in the alignment the constraints are provided for
- **i** – 5' position of the base pairs the soft constraint is added for
- **j** – 3' position of the base pairs the soft constraint is added for
- **energy** – Free energy (soft-constraint) in $kcal/mol$
- **options** – The options flag indicating how/where to store the soft constraints

Returns
The number of sequences in the MSA constraints have been applied to

Constraints for Stacked Base Pairs

Functions

int **vrna_sc_set_stack**(*vrna_fold_compound_t* *fc, const *FLT_OR_DBL* *constraints, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_set_stack_comparative**(*vrna_fold_compound_t* *fc, const *FLT_OR_DBL* **constraints, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_set_stack_comparative_seq**(*vrna_fold_compound_t* *fc, unsigned int s, const *FLT_OR_DBL* *constraints, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_add_stack**(*vrna_fold_compound_t* *fc, unsigned int i, *FLT_OR_DBL* energy, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_add_stack_comparative**(*vrna_fold_compound_t* *fc, unsigned int *is, const *FLT_OR_DBL* *energies, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_add_stack_comparative_seq**(*vrna_fold_compound_t* *fc, unsigned int s, unsigned int i, *FLT_OR_DBL* energy, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

Generic implementation

Typedefs

typedef int (***vrna_sc_f**)(int i, int j, int k, int l, unsigned char d, void *data)

Callback to retrieve pseudo energy contribution for soft constraint feature.

```
#include <ViennaRNA/constraints/soft.h>
```

This is the prototype for callback functions used by the folding recursions to evaluate generic soft constraints. The first four parameters passed indicate the delimiting nucleotide positions of the decomposition, and the parameter `denotes` the decomposition step. The last parameter `data` is the auxiliary data structure associated to the hard constraints via *vrna_sc_add_data()*, or NULL if no auxiliary data was added.

Notes on Callback Functions:

This callback enables one to add (pseudo-)energy contributions to individual decompositions of the secondary structure.

See also

VRNA_DECOMP_PAIR_HP, *VRNA_DECOMP_PAIR_IL*, *VRNA_DECOMP_PAIR_ML*, *VRNA_DECOMP_ML_ML_ML*, *VRNA_DECOMP_ML_STEM*, *VRNA_DECOMP_ML_ML*, *VRNA_DECOMP_ML_UP*, *VRNA_DECOMP_ML_ML_STEM*, *VRNA_DECOMP_ML_COAXIAL*, *VRNA_DECOMP_EXT_EXT*, *VRNA_DECOMP_EXT_UP*, *VRNA_DECOMP_EXT_STEM*, *VRNA_DECOMP_EXT_EXT_EXT*, *VRNA_DECOMP_EXT_STEM_EXT*, *VRNA_DECOMP_EXT_EXT_STEM*, *VRNA_DECOMP_EXT_EXT_STEM1*, *vrna_sc_add_f()*, *vrna_sc_add_exp_f()*, *vrna_sc_add_bt()*, *vrna_sc_add_data()*

Param i
Left (5') delimiter position of substructure

Param j
Right (3') delimiter position of substructure

Param k
Left delimiter of decomposition

Param l
Right delimiter of decomposition

Param d
Decomposition step indicator

Param data
Auxiliary data

Return
Pseudo energy contribution in deka-kalories per mol

typedef int **vrna_callback_sc_energy**(int i, int j, int k, int l, unsigned char d, void *data)

```
#include <ViennaRNA/constraints/soft.h>
```

typedef int (***vrna_sc_direct_f**)(*vrna_fold_compound_t* *fc, int i, int j, int k, int l, void *data)

```
#include <ViennaRNA/constraints/soft.h>
```

typedef *FLT_OR_DBL* (***vrna_sc_exp_f**)(int i, int j, int k, int l, unsigned char d, void *data)

Callback to retrieve pseudo energy contribution as Boltzmann Factors for soft constraint feature.

```
#include <ViennaRNA/constraints/soft.h>
```

This is the prototype for callback functions used by the partition function recursions to evaluate generic soft constraints. The first four parameters passed indicate the delimiting nucleotide positions of the decomposition, and the parameter `denotes` the decomposition step. The last parameter `data` is the auxiliary data structure associated to the hard constraints via *vrna_sc_add_data()*, or NULL if no auxiliary data was added.

Notes on Callback Functions:

This callback enables one to add (pseudo-)energy contributions to individual decompositions of the secondary structure (Partition function variant, i.e. contributions must be returned as Boltzmann factors).

See also

VRNA_DECOMP_PAIR_HP, *VRNA_DECOMP_PAIR_IL*, *VRNA_DECOMP_PAIR_ML*, *VRNA_DECOMP_ML_ML_ML*, *VRNA_DECOMP_ML_STEM*, *VRNA_DECOMP_ML_ML*, *VRNA_DECOMP_ML_UP*, *VRNA_DECOMP_ML_ML_STEM*, *VRNA_DECOMP_ML_COAXIAL*, *VRNA_DECOMP_EXT_EXT*, *VRNA_DECOMP_EXT_UP*, *VRNA_DECOMP_EXT_STEM*, *VRNA_DECOMP_EXT_EXT_EXT*, *VRNA_DECOMP_EXT_STEM_EXT*, *VRNA_DECOMP_EXT_EXT_STEM*, *VRNA_DECOMP_EXT_EXT_STEM1*, *vrna_sc_add_exp_f()*, *vrna_sc_add_f()*, *vrna_sc_add_bt()*, *vrna_sc_add_data()*

Param i
Left (5') delimiter position of substructure

Param j
Right (3') delimiter position of substructure

Param k
Left delimiter of decomposition

Param l
Right delimiter of decomposition

Param d
Decomposition step indicator

Param data
Auxiliary data

Return
Pseudo energy contribution in deka-kalories per mol

typedef *FLT_OR_DBL* **vrna_callback_sc_exp_energy**(int i, int j, int k, int l, unsigned char d, void *data)

```
#include <ViennaRNA/constraints/soft.h>
```

typedef *FLT_OR_DBL* (***vrna_sc_exp_direct_f**)(*vrna_fold_compound_t* *fc, int i, int j, int k, int l, void *data)

```
#include <ViennaRNA/constraints/soft.h>
```

typedef *vrna_basepair_t* *(***vrna_sc_bt_f**)(int i, int j, int k, int l, unsigned char d, void *data)

Callback to retrieve auxiliary base pairs for soft constraint feature.

```
#include <ViennaRNA/constraints/soft.h>
```

Notes on Callback Functions:

This callback enables one to add auxiliary base pairs in the backtracking steps of hairpin- and internal loops.

See also

VRNA_DECOMP_PAIR_HP, *VRNA_DECOMP_PAIR_IL*, *VRNA_DECOMP_PAIR_ML*, *VRNA_DECOMP_ML_ML_ML*, *VRNA_DECOMP_ML_STEM*, *VRNA_DECOMP_ML_ML*, *VRNA_DECOMP_ML_UP*, *VRNA_DECOMP_ML_ML_STEM*, *VRNA_DECOMP_ML_COAXIAL*, *VRNA_DECOMP_EXT_EXT*, *VRNA_DECOMP_EXT_UP*, *VRNA_DECOMP_EXT_STEM*, *VRNA_DECOMP_EXT_EXT_EXT*, *VRNA_DECOMP_EXT_STEM_EXT*, *VRNA_DECOMP_EXT_EXT_STEM*, *VRNA_DECOMP_EXT_EXT_STEM1*, *vrna_sc_add_bt()*, *vrna_sc_add_f()*, *vrna_sc_add_exp_f()*, *vrna_sc_add_data()*

Param i
Left (5') delimiter position of substructure

Param j
Right (3') delimiter position of substructure

Param k
Left delimiter of decomposition

Param l
Right delimiter of decomposition

Param d
Decomposition step indicator

Param data
Auxiliary data

Return
List of additional base pairs

typedef *vrna_basepair_t* ***vrna_callback_sc_backtrack**(int i, int j, int k, int l, unsigned char d, void *data)

```
#include <ViennaRNA/constraints/soft.h>
```

Functions

int **vrna_sc_add_data**(*vrna_fold_compound_t* *fc, void *data, *vrna_auxdata_free_f* free_data)

Add an auxiliary data structure for the generic soft constraints callback function.

```
#include <ViennaRNA/constraints/soft.h>
```

SWIG Wrapper Notes:

This function is attached as method `sc_add_data()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_data()* in the *The RNA Python module* .

See also

vrna_sc_add_f(), *vrna_sc_add_exp_f()*, *vrna_sc_add_bt()*

Parameters

- **fc** – The fold compound the generic soft constraint function should be bound to
- **data** – A pointer to the data structure that holds required data for function 'f'
- **free_data** – A pointer to a function that free's the memory occupied by `data` (Maybe NULL)

Returns
Non-zero on successful binding the data (and free-function), 0 otherwise

int **vrna_sc_add_auxdata**(*vrna_fold_compound_t* *fc, void *data, *vrna_auxdata_prepare_f* prepare_cb, *vrna_auxdata_free_f* free_cb)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_set_data_comparative**(*vrna_fold_compound_t* *fc, void **data, *vrna_auxdata_free_f* *free_data, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_set_data_comparative_seq**(*vrna_fold_compound_t* *fc, unsigned int s, void *data, *vrna_auxdata_free_f* free_data, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_set_auxdata_comparative**(*vrna_fold_compound_t* *fc, void **data, *vrna_auxdata_prepare_f* *prepare_cbs, *vrna_auxdata_free_f* *free_data, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_set_auxdata_comparative_seq**(*vrna_fold_compound_t* *fc, unsigned int s, void *data, *vrna_auxdata_prepare_f* prepare_cb, *vrna_auxdata_free_f* free_data, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_add_f**(*vrna_fold_compound_t* *fc, *vrna_sc_f* f)

Bind a function pointer for generic soft constraint feature (MFE version).

```
#include <ViennaRNA/constraints/soft.h>
```

This function allows one to easily bind a function pointer and corresponding data structure to the soft constraint part *vrna_sc_t* of the *vrna_fold_compound_t*. The function for evaluating the generic soft constraint feature has to return a pseudo free energy $\hat{E}$ in $dacal/mol$, where $1dacal/mol = 10cal/mol$.

SWIG Wrapper Notes:

This function is attached as method `sc_add_f()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.sc_add_f()`* in the *The RNA Python module* .

See also

vrna_sc_add_data(), *vrna_sc_add_bt()*, *vrna_sc_add_exp_f()*

Parameters

- **fc** – The fold compound the generic soft constraint function should be bound to
- **f** – A pointer to the function that evaluates the generic soft constraint feature

Returns
Non-zero on successful binding the callback function, 0 otherwise

size_t **vrna_sc_multi_cb_add**(*vrna_fold_compound_t* *fc, *vrna_sc_direct_f* cb, *vrna_sc_exp_direct_f* cb_exp, void *data, *vrna_auxdata_prepare_f* prepare_cb, *vrna_auxdata_free_f* free_cb, unsigned int decomp_type)

```
#include <ViennaRNA/constraints/soft.h>
```

size_t **vrna_sc_multi_cb_add_comparative**(*vrna_fold_compound_t* *fc, *vrna_sc_direct_f* *cbs, *vrna_sc_exp_direct_f* *cbs_exp, void **datas, *vrna_auxdata_prepare_f* *prepare_cbs, *vrna_auxdata_free_f* *free_cbs, unsigned int *ds, unsigned int multi_params)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_set_f_comparative**(*vrna_fold_compound_t* *fc, *vrna_sc_f* *f, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

int **vrna_sc_add_bt**(*vrna_fold_compound_t* *fc, *vrna_sc_bt_f* f)

Bind a backtracking function pointer for generic soft constraint feature.

```
#include <ViennaRNA/constraints/soft.h>
```

This function allows one to easily bind a function pointer to the soft constraint part *vrna_sc_t* of the *vrna_fold_compound_t*. The provided function should be used for backtracking purposes in loop regions that were altered via the generic soft constraint feature. It has to return an array of *vrna_basepair_t* data structures, were the last element in the list is indicated by a value of -1 in it's i position.

SWIG Wrapper Notes:

This function is attached as method `sc_add_bt()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.sc_add_bt()`* in the *The RNA Python module* .

See also

vrna_sc_add_data(), *vrna_sc_add_f()*, *vrna_sc_add_exp_f()*

Parameters

- **fc** – The fold compound the generic soft constraint function should be bound to
- **f** – A pointer to the function that returns additional base pairs

Returns

Non-zero on successful binding the callback function, 0 otherwise

int **vrna_sc_add_exp_f**(*vrna_fold_compound_t* *fc, *vrna_sc_exp_f* exp_f)

Bind a function pointer for generic soft constraint feature (PF version).

```
#include <ViennaRNA/constraints/soft.h>
```

This function allows one to easily bind a function pointer and corresponding data structure to the soft constraint part *vrna_sc_t* of the *vrna_fold_compound_t*. The function for evaluating the generic soft constraint feature has to return a pseudo free energy $\hat{E}$ as Boltzmann factor, i.e. $exp(-\hat{E}/kT)$. The required unit for E is cal/mol.

SWIG Wrapper Notes:

This function is attached as method `sc_add_exp_f()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_exp_f()* in the *The RNA Python module* .

See also

vrna_sc_add_bt(), *vrna_sc_add_f()*, *vrna_sc_add_data()*

Parameters

- **fc** – The fold compound the generic soft constraint function should be bound to
- **exp_f** – A pointer to the function that evaluates the generic soft constraint feature

Returns

Non-zero on successful binding the callback function, 0 otherwise

int **vrna_sc_set_exp_f_comparative**(*vrna_fold_compound_t* *fc, *vrna_sc_exp_f* *exp_f, unsigned int options)

```
#include <ViennaRNA/constraints/soft.h>
```

Secondary Structure Domains

This module provides the tools to add and modify domains to the production rules of the RNA folding grammar.

Structured Domains API

Unstructured Domain API

Defines

VRNA_UNSTRUCTURED_DOMAIN_EXT_LOOP

Flag to indicate ligand bound to unpiared stretch in the exterior loop.

```
#include <ViennaRNA/unstructured_domains.h>
```

VRNA_UNSTRUCTURED_DOMAIN_HP_LOOP

Flag to indicate ligand bound to unpaired stretch in a hairpin loop.

```
#include <ViennaRNA/unstructured_domains.h>
```

VRNA_UNSTRUCTURED_DOMAIN_INT_LOOP

Flag to indicate ligand bound to unpiared stretch in an internal loop.

```
#include <ViennaRNA/unstructured_domains.h>
```

VRNA_UNSTRUCTURED_DOMAIN_MB_LOOP

Flag to indicate ligand bound to unpiared stretch in a multibranch loop.

```
#include <ViennaRNA/unstructured_domains.h>
```

VRNA_UNSTRUCTURED_DOMAIN_MOTIF

Flag to indicate ligand binding without additional unbound nucleotides (motif-only).

```
#include <ViennaRNA/unstructured_domains.h>
```

VRNA_UNSTRUCTURED_DOMAIN_ALL_LOOPS

Flag to indicate ligand bound to unpiared stretch in any loop (convenience macro).

```
#include <ViennaRNA/unstructured_domains.h>
```

Typedefs

typedef struct *vrna_unstructured_domain_s* **vrna_ud_t**

Typename for the ligand binding extension data structure *vrna_unstructured_domain_s*.

```
#include <ViennaRNA/unstructured_domains.h>
```

typedef int (***vrna_ud_f**)(*vrna_fold_compound_t* *fc, int i, int j, unsigned int loop_type, void *data)

Callback to retrieve binding free energy of a ligand bound to an unpaired sequence segment.

```
#include <ViennaRNA/unstructured_domains.h>
```

Notes on Callback Functions:

This function will be called to determine the additional energy contribution of a specific unstructured domain, e.g. the binding free energy of some ligand.

Param fc
The current *vrna_fold_compound_t*

Param i
The start of the unstructured domain (5' end)

Param j
The end of the unstructured domain (3' end)

Param loop_type
The loop context of the unstructured domain

Param data
Auxiliary data

Return
The auxiliary energy contribution in deka-cal/mol

typedef *FLT_OR_DBL* (***vrna_ud_exp_f**)(*vrna_fold_compound_t* *fc, int i, int j, unsigned int loop_type, void *data)

Callback to retrieve Boltzmann factor of the binding free energy of a ligand bound to an unpaired sequence segment.

```
#include <ViennaRNA/unstructured_domains.h>
```

Notes on Callback Functions:

This function will be called to determine the additional energy contribution of a specific unstructured domain, e.g. the binding free energy of some ligand (Partition function variant, i.e. the Boltzmann factors instead of actual free energies).

Param fc
The current *vrna_fold_compound_t*

Param i
The start of the unstructured domain (5' end)

Param j
The end of the unstructured domain (3' end)

Param loop_type
The loop context of the unstructured domain

Param data
Auxiliary data

Return
The auxiliary energy contribution as Boltzmann factor

typedef void (***vrna_ud_production_f**)(*vrna_fold_compound_t* *fc, void *data)

Callback for pre-processing the production rule of the ligand binding to unpaired stretches feature.

```
#include <ViennaRNA/unstructured_domains.h>
```

Notes on Callback Functions:

The production rule for the unstructured domain grammar extension

typedef void (***vrna_ud_exp_production_f**)(*vrna_fold_compound_t* *fc, void *data)

Callback for pre-processing the production rule of the ligand binding to unpaired stretches feature (partition function variant).

```
#include <ViennaRNA/unstructured_domains.h>
```

Notes on Callback Functions:

The production rule for the unstructured domain grammar extension (Partition function variant)

typedef void (***vrna_ud_add_probs_f**)(*vrna_fold_compound_t* *fc, int i, int j, unsigned int loop_type, *FLT_OR_DBL* exp_energy, void *data)

Callback to store/add equilibrium probability for a ligand bound to an unpaired sequence segment.

```
#include <ViennaRNA/unstructured_domains.h>
```

Notes on Callback Functions:

A callback function to store equilibrium probabilities for the unstructured domain feature

typedef *FLT_OR_DBL* (***vrna_ud_get_probs_f***)(*vrna_fold_compound_t* *fc, int i, int j, unsigned int loop_type, int motif, void *data)

Callback to retrieve equilibrium probability for a ligand bound to an unpaired sequence segment.

```
#include <ViennaRNA/unstructured_domains.h>
```

Notes on Callback Functions:

A callback function to retrieve equilibrium probabilities for the unstructured domain feature

Functions

vrna_ud_motif_t ***vrna_ud_motifs_centroid**(*vrna_fold_compound_t* *fc, const char *structure)*

Detect unstructured domains in centroid structure.

```
#include <ViennaRNA/unstructured_domains.h>
```

Given a centroid structure and a set of unstructured domains compute the list of unstructured domain motifs present in the centroid. Since we do not explicitly annotate unstructured domain motifs in dot-bracket strings, this function can be used to check for the presence and location of unstructured domain motifs under the assumption that the dot-bracket string is the centroid structure of the equiibrium ensemble.

See also

vrna_centroid()

Parameters

- **fc** – The fold_compound data structure with pre-computed equilibrium probabilities and model settings
- **structure** – The centroid structure in dot-bracket notation

Returns

A list of unstructured domain motifs (possibly NULL). The last element terminates the list with `start=0`, `number=-1`

vrna_ud_motif_t ***vrna_ud_motifs_MEA**(*vrna_fold_compound_t* *fc, const char *structure, *vrna_ep_t* *probability_list)*

Detect unstructured domains in MEA structure.

```
#include <ViennaRNA/unstructured_domains.h>
```

Given an MEA structure and a set of unstructured domains compute the list of unstructured domain motifs present in the MEA structure. Since we do not explicitly annotate unstructured domain motifs in dot-bracket strings, this function can be used to check for the presence and location of unstructured domain motifs under the assumption that the dot-bracket string is the MEA structure of the equiibrium ensemble.

See also

MEA()

Parameters

- **fc** – The fold_compound data structure with pre-computed equilibrium probabilities and model settings
- **structure** – The MEA structure in dot-bracket notation
- **probability_list** – The list of probabilities to extract the MEA structure from

Returns

A list of unstructured domain motifs (possibly NULL). The last element terminates the list with `start=0`, `number=-1`

vrna_ud_motif_t ***vrna_ud_motifs_MFE**(*vrna_fold_compound_t* *fc, const char *structure)

Detect unstructured domains in MFE structure.

```
#include <ViennaRNA/unstructured_domains.h>
```

Given an MFE structure and a set of unstructured domains compute the list of unstructured domain motifs present in the MFE structure. Since we do not explicitly annotate unstructured domain motifs in dot-bracket strings, this function can be used to check for the presence and location of unstructured domain motifs under the assumption that the dot-bracket string is the MFE structure of the equiibrium ensemble.

See also

vrna_mfe()

Parameters

- **fc** – The fold_compound data structure with model settings
- **structure** – The MFE structure in dot-bracket notation

Returns

A list of unstructured domain motifs (possibly NULL). The last element terminates the list with `start=0`, `number=-1`

void **vrna_ud_add_motif**(*vrna_fold_compound_t* *fc, const char *motif, double motif_en, const char *motif_name, unsigned int loop_type)

Add an unstructured domain motif, e.g. for ligand binding.

```
#include <ViennaRNA/unstructured_domains.h>
```

This function adds a ligand binding motif and the associated binding free energy to the *vrna_ud_t* attribute of a *vrna_fold_compound_t*. The motif data will then be used in subsequent secondary structure predictions. Multiple calls to this function with different motifs append all additional data to a list of ligands, which all will be evaluated. Ligand motif data can be removed from the *vrna_fold_compound_t* again using the *vrna_ud_remove()* function. The loop type parameter allows one to limit the ligand binding to particular loop type, such as the exterior loop, hairpin loops, internal loops, or multibranch loops.

See also

VRNA_UNSTRUCTURED_DOMAIN_EXT_LOOP, *VRNA_UNSTRUCTURED_DOMAIN_HP_LOOP*, *VRNA_UNSTRUCTURED_DOMAIN_INT_LOOP*, *VRNA_UNSTRUCTURED_DOMAIN_MB_LOOP*, *VRNA_UNSTRUCTURED_DOMAIN_ALL_LOOPS*, *vrna_ud_remove()*

Parameters

- **fc** – The *vrna_fold_compound_t* data structure the ligand motif should be bound to
- **motif** – The sequence motif the ligand binds to
- **motif_en** – The binding free energy of the ligand in kcal/mol
- **motif_name** – The name/id of the motif (may be NULL)
- **loop_type** – The loop type the ligand binds to

void **vrna_ud_remove**(*vrna_fold_compound_t* *fc)

Remove ligand binding to unpaired stretches.

```
#include <ViennaRNA/unstructured_domains.h>
```

This function removes all ligand motifs that were bound to a *vrna_fold_compound_t* using the *vrna_ud_add_motif()* function.

SWIG Wrapper Notes:

This function is attached as method `ud_remove()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.ud_remove()* in the *The RNA Python module*.

Parameters

- **fc** – The *vrna_fold_compound_t* data structure the ligand motif data should be removed from

void **vrna_ud_set_data**(*vrna_fold_compound_t* *fc, void *data, *vrna_auxdata_free_f* free_cb)

Attach an auxiliary data structure.

```
#include <ViennaRNA/unstructured_domains.h>
```

This function binds an arbitrary, auxiliary data structure for user-implemented ligand binding. The optional callback `free_cb` will be passed the bound data structure whenever the *vrna_fold_compound_t* is removed from memory to avoid memory leaks.

SWIG Wrapper Notes:

This function is attached as method `ud_set_data()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.ud_set_data()* in the *The RNA Python module*.

See also

vrna_ud_set_prod_rule_cb(), *vrna_ud_set_exp_prod_rule_cb()*, *vrna_ud_remove()*

Parameters

- **fc** – The *vrna_fold_compound_t* data structure the auxiliary data structure should be bound to

- **data** – A pointer to the auxiliary data structure
- **free_cb** – A pointer to a callback function that free's memory occupied by data

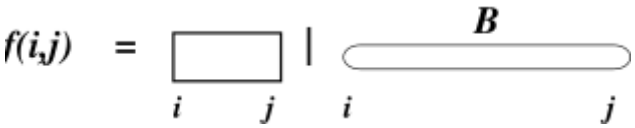
void **vrna_ud_set_prod_rule_cb**(*vrna_fold_compound_t* *fc, *vrna_ud_production_f* pre_cb, *vrna_ud_f* e_cb)

Attach production rule callbacks for free energies computations.

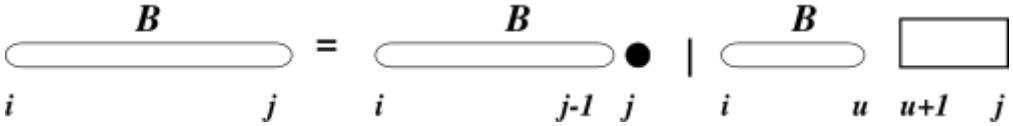
```
#include <ViennaRNA/unstructured_domains.h>
```

Use this function to bind a user-implemented grammar extension for unstructured domains.

The callback e_cb needs to evaluate the free energy contribution $f(i,j)$ of the unpaired segment $[i,j]$. It will be executed in each of the regular secondary structure production rules. Whenever the callback is passed the *VRNA_UNSTRUCTURED_DOMAIN_MOTIF* flag via its loop_type parameter the contribution of any ligand that consecutively binds from position i to j (the white box) is requested. Otherwise, the callback usually performs a lookup in the precomputed B matrices. Which B matrix is addressed will be indicated by the flags *VRNA_UNSTRUCTURED_DOMAIN_EXT_LOOP*, *VRNA_UNSTRUCTURED_DOMAIN_HP_LOOP VRNA_UNSTRUCTURED_DOMAIN_INT_LOOP*, and *VRNA_UNSTRUCTURED_DOMAIN_MB_LOOP*. As their names already imply, they specify exterior loops (F production rule), hairpin loops and internal loops (C production rule), and multibranch loops (M and M1 production rule).



The pre_cb callback will be executed as a pre-processing step right before the regular secondary structure rules. Usually one would use this callback to fill the dynamic programming matrices U and preparations of the auxiliary data structure *vrna_unstructured_domain_s.data*



SWIG Wrapper Notes:

This function is attached as method ud_set_prod_rule_cb() to objects of type fold_compound. See, e.g. *RNA.fold_compound.ud_set_prod_rule_cb()* in the *The RNA Python module*.

Parameters

- **fc** – The *vrna_fold_compound_t* data structure the callback will be bound to
- **pre_cb** – A pointer to a callback function for the B production rule
- **e_cb** – A pointer to a callback function for free energy evaluation

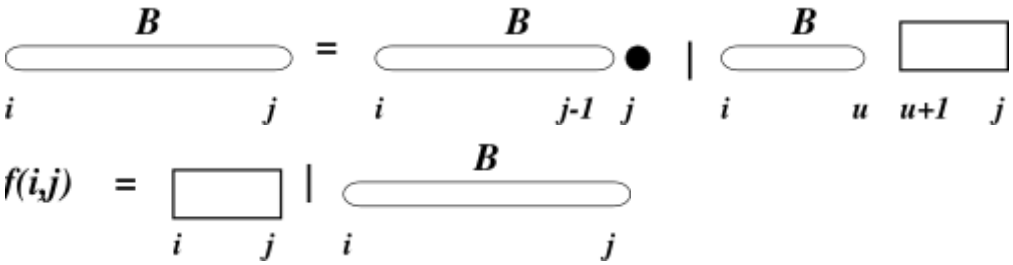
void **vrna_ud_set_exp_prod_rule_cb**(*vrna_fold_compound_t* *fc, *vrna_ud_exp_production_f* pre_cb, *vrna_ud_exp_f* exp_e_cb)

Attach production rule for partition function.

```
#include <ViennaRNA/unstructured_domains.h>
```

This function is the partition function companion of *vrna_ud_set_prod_rule_cb()*.

Use it to bind callbacks to (i) fill the U production rule dynamic programming matrices and/or prepare the *vrna_unstructured_domain_s.data*, and (ii) provide a callback to retrieve partition functions for subsegments $[i,j]$.



SWIG Wrapper Notes:

This function is attached as method `ud_set_exp_prod_rule_cb()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.ud_set_exp_prod_rule_cb()`* in the *The RNA Python module*.

See also

vrna_ud_set_prod_rule_cb()

Parameters

- **fc** – The *vrna_fold_compound_t* data structure the callback will be bound to
- **pre_cb** – A pointer to a callback function for the `B` production rule
- **exp_e_cb** – A pointer to a callback function that retrieves the partition function for a segment $[i, j]$ that may be bound by one or more ligands.

struct **vrna_unstructured_domain_s**

#include <ViennaRNA/unstructured_domains.h> Data structure to store all functionality for ligand binding.

Public Members

int **uniq_motif_count**

The unique number of motifs of different lengths.

```
#include <ViennaRNA/unstructured_domains.h>
```

unsigned int ***uniq_motif_size**

An array storing a unique list of motif lengths.

```
#include <ViennaRNA/unstructured_domains.h>
```

int **motif_count**

Total number of distinguished motifs.

```
#include <ViennaRNA/unstructured_domains.h>
```

char ****motif**

Motif sequences.

```
#include <ViennaRNA/unstructured_domains.h>
```

char **motif_name

Motif identifier/name.

```
#include <ViennaRNA/unstructured_domains.h>
```

unsigned int *motif_size

Motif lengths.

```
#include <ViennaRNA/unstructured_domains.h>
```

double *motif_en

Ligand binding free energy contribution.

```
#include <ViennaRNA/unstructured_domains.h>
```

unsigned int *motif_type

Type of motif, i.e. loop type the ligand binds to.

```
#include <ViennaRNA/unstructured_domains.h>
```

vrna_ud_production_f prod_cb

Callback to ligand binding production rule, i.e. create/fill DP free energy matrices.

```
#include <ViennaRNA/unstructured_domains.h>
```

This callback will be executed right before the actual secondary structure decompositions, and, therefore, any implementation must not interleave with the regular DP matrices.

vrna_ud_exp_production_f exp_prod_cb

Callback to ligand binding production rule, i.e. create/fill DP partition function matrices.

```
#include <ViennaRNA/unstructured_domains.h>
```

vrna_ud_f energy_cb

Callback to evaluate free energy of ligand binding to a particular unpaired stretch.

```
#include <ViennaRNA/unstructured_domains.h>
```

vrna_ud_exp_f exp_energy_cb

Callback to evaluate Boltzmann factor of ligand binding to a particular unpaired stretch.

```
#include <ViennaRNA/unstructured_domains.h>
```

void *data

Auxiliary data structure passed to energy evaluation callbacks.

```
#include <ViennaRNA/unstructured_domains.h>
```

vrna_auxdata_free_f free_data

Callback to free auxiliary data structure.

```
#include <ViennaRNA/unstructured_domains.h>
```

vrna_ud_add_probs_f **probs_add**

Callback to store/add outside partition function.

```
#include <ViennaRNA/unstructured_domains.h>
```

vrna_ud_get_probs_f **probs_get**

Callback to retrieve outside partition function.

```
#include <ViennaRNA/unstructured_domains.h>
```

Extending the Folding Grammar

Typedefs

typedef struct vrna_gr_aux_s ***vrna_gr_aux_t**

A pointer to the auxiliary grammar data structure.

```
#include <ViennaRNA/grammar/basic.h>
```

typedef char *(***vrna_gr_serialize_bp_f**)(*vrna_fold_compound_t* *fc, *vrna_bps_t* bp_stack, void *data)

Function prototype for serializing backtracked base pairs and structure elements into a dot-bracket string.

```
#include <ViennaRNA/grammar/basic.h>
```

This function will be called after backtracking in the MFE predictions to convert collected base pairs and other information into a dot-bracket-like structure string.

Notes on Callback Functions:

This callback allows for changing the way how base pairs (and other types of data) obtained from the default and extended grammar are converted back into a dot-bracket string.

See also

vrna_db_from_bps(), *vrna_gr_add_aux()*

Param fc
The fold compound to work on

Param bp_stack
The base pair stack

Param data
An arbitrary user-provided data pointer

Return
A '\0'-terminated dot-bracket-like string representing the structure from `bp_stack` (and `data`)

typedef int (***vrna_gr_inside_f**)(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, void *data)

Function prototype for auxiliary grammar rules (inside version, MFE).

```
#include <ViennaRNA/grammar/mfe.h>
```

This function will be called during the inside recursions of MFE predictions for subsequences from position `i` to `j` and is supposed to return an energy contribution in dekacal/mol.

Notes on Callback Functions:

This callback allows for extending the MFE secondary structure decomposition with additional rules.

See also

vrna_gr_add_aux_f(), *vrna_gr_add_aux_c()*, *vrna_gr_add_aux_m()*, *vrna_gr_add_aux_m1()*, vrna_gr_add_aux_aux(), *vrna_gr_add_aux_exp_f()*

Param fc
The fold compound to work on

Param i
The 5' delimiter of the sequence segment

Param j
The 3' delimiter of the sequence segment

Param data
An arbitrary user-provided data pointer

Return
The free energy computed by the auxiliary grammar rule in dekacal/mol

typedef int (***vrna_gr_outside_f**)(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, int e, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack, void *data)

Function prototype for auxiliary grammar rules (outside version, MFE).

```
#include <ViennaRNA/grammar/mfe.h>
```

This function will be called during the backtracking stage of MFE predictions for subsequences from position `i` to `j` and is supposed to identify the the structure components that give rise to the corresponding energy contribution `e` as determined in the inside (forward) step.

For that purpose, the function may modify the base pair stack (`bp_stack`) by adding all base pairs identified through the additional rule. In addition, when the rule sub-divides the sequence segment into smaller pieces, these pieces can be put onto the backtracking (`bt_stack`) stack. On successful backtrack, the function returns non-zero, and exactly zero (0) when the backtracking failed.

Notes on Callback Functions:

This callback allows for backtracking (sub)structures obtained from extending the MFE secondary structure decomposition with additional rules.

See also

vrna_gr_add_aux_f(), *vrna_gr_add_aux_c()*, *vrna_gr_add_aux_m()*, *vrna_gr_add_aux_m1()*, vrna_gr_add_aux_aux(), *vrna_gr_add_aux_exp_f()*

Param fc
The fold compound to work on

Param i
The 5' delimiter of the sequence segment

Param j
The 3' delimiter of the sequence segment

Param e
The energy of the segment [i:j]

Param bp_stack
The base pair stack

Param bt_stack
The backtracking stack (all segments that need to be further investigated)

Param data
An arbitrary user-provided data pointer

Return
The free energy computed by the auxiliary grammar rule in dekacal/mol

typedef *FLT_OR_DBL* (***vrna_gr_inside_exp_f**)(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, void *data)

Function prototype for auxiliary grammar rules (inside version, partition function).

```
#include <ViennaRNA/grammar/partfunc.h>
```

This function will be called during the inside recursions of partition function predictions for subsequences from position `i` to `j` and is supposed to return a Boltzmann factor with energy contribution in cal/mol.

Notes on Callback Functions:

This callback allows for extending the partition function secondary structure decomposition with additional rules.

See also

vrna_gr_add_aux_exp_f(), *vrna_gr_add_aux_exp_c()*, *vrna_gr_add_aux_exp_m()*, *vrna_gr_add_aux_exp_m1()*, *vrna_gr_add_aux_exp()*, *vrna_gr_add_aux_f()*

Param fc
The fold compound to work on

Param i
The 5' delimiter of the sequence segment

Param j
The 3' delimiter of the sequence segment

Param data
An arbitrary user-provided data pointer

Return
The partition function omputed by the auxiliary grammar rule with energies in cal/mol

typedef *FLT_OR_DBL* (***vrna_gr_outside_exp_f**)(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, void *data)

```
#include <ViennaRNA/grammar/partfunc.h>
```

Functions

unsigned int **vrna_gr_prepare**(*vrna_fold_compound_t* *fc, unsigned int options)

Prepare the auxiliary grammar rule data.

```
#include <ViennaRNA/grammar/basic.h>
```

Note

This function is mainly for internal use. Users of the auxiliary grammar API usually do not need to call this function except for debugging purposes.

Parameters

- **fc** – The fold compound storing the auxiliary grammar rules
- **options** – Options flag(s) that denote what needs to be prepared

Returns
non-zero on success, 0 otherwise

unsigned int **vrna_gr_add_status**(*vrna_fold_compound_t* *fc, *vrna_recursion_status_f* cb, void *data, *vrna_auxdata_prepare_f* prepare_cb, *vrna_auxdata_free_f* free_cb)

Add status event based data preparation callbacks.

```
#include <ViennaRNA/grammar/basic.h>
```

This function binds additional data structures and corresponding callback functions for the auxiliary grammar extension API. This might be helpful whenever certain preparation steps need to be done prior and/or after the actual run of the prediction algorithms.

Parameters

- **fc** – The fold compound
- **cb** – The recursion status callback that performs the preparation
- **data** – The data pointer the `cb` callback is working on
- **prepare_cb** – A preparation callback function for parameter `data`
- **free_cb** – A callback to release memory for `data`

Returns
The number of status function callbacks bound to the fold compound or 0 on error

unsigned int **vrna_gr_set_serialize_bp**(*vrna_fold_compound_t* *fc, *vrna_gr_serialize_bp_f* cb, void *data, *vrna_auxdata_prepare_f* prepare_cb, *vrna_auxdata_free_f* free_cb)

Set base pair stack to dot-bracket string serialization function.

```
#include <ViennaRNA/grammar/basic.h>
```

After backtracking secondary structures, e.g. in MFE predictions, the outside algorithm usually collects a set of base pairs that then need to be converted into a dot-bracket string. By default, this conversion is done using the *vrna_db_from_bps()* function. However, this function only considers nested base pairs and no other type of secondary structure elements.

When extending the recursions by additional rules, the default conversion may not suffice, e.g. because the grammar extension adds 2.5D modules or pseudoknots. In such cases the user should implement its own dot-bracket string conversion strategy that may use additional symbols.

This function binds a user-implemented conversion function that must return a '\0' terminated dot-bracket-like string the same length as the input sequence. The conversion function will then be used instead of the default one. In addition to the base pair stack *vrna_bps_t*, the user-defined conversion function may keep track of whatever information is neccessary to properly convert the backtracked structure into a dot-bracket string. For that purpose, the `data` pointer can be used, e.g. it can point to the same data as used in any of the grammar extension rules. The `prepare_cb` and `free_cb` callbacks can again be used to control preparation and release of the memory `data` points to. The `prepare_cb` will be called after all the preparations for the grammar extensions and prior the actual inside-recursions. The conversion function callback `cb` will be called after backtracking.

See also

vrna_db_from_bps(), *vrna_gr_add_aux()*

Parameters

- **fc** – The fold compound
- **cb** – A pointer to the conversion callback function
- **data** – A pointer to arbitrary data that will be passed through to `cb` (may be **NULL**)
- **prepare_cb** – A function pointer to prepare `data` (may be **NULL**)
- **free_cb** – A function pointer to release memory occupied by `data` (may be **NULL**)

void **vrna_gr_reset**(*vrna_fold_compound_t* *fc)

Remove all auxiliary grammar rules.

```
#include <ViennaRNA/grammar/basic.h>
```

This function re-sets the fold compound to the default rules by removing all auxiliary grammar rules

Parameters

- **fc** – The fold compound

unsigned int **vrna_gr_add_aux_f**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_f* cb, *vrna_gr_outside_f* cb_bt, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the F-decomposition (MFE version).

```
#include <ViennaRNA/grammar/mfe.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the F-decomposition, i.e. the external loop decomposition stage.

While the inside rule (`cb`) computes a minimum free energy contribution for any subsequence the outside rule (`cb_bt`) is used for backtracking the corresponding structure.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

See also

vrna_gr_add_aux_c(), *vrna_gr_add_aux_m()*, *vrna_gr_add_aux_m1()*, *vrna_gr_add_aux()*, *vrna_gr_add_aux_exp_f()*, *vrna_gr_inside_f*, *vrna_gr_outside_f*, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_bt** – The auxiliary grammar callback for the outside (backtracking) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns

The current number of auxiliary grammar rules for the MFE F-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_c**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_f* cb, *vrna_gr_outside_f* cb_bt, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the C-decomposition (MFE version).

```
#include <ViennaRNA/grammar/mfe.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the C-decomposition, i.e. the base pair decomposition stage.

While the inside rule (`cb`) computes a minimum free energy contribution for any subsequence the outside rule (`cb_bt`) is used for backtracking the corresponding structure.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

See also

vrna_gr_add_aux_f(), *vrna_gr_add_aux_m()*, *vrna_gr_add_aux_m1()*, *vrna_gr_add_aux()*, *vrna_gr_add_aux_exp_c()*, *vrna_gr_inside_f*, *vrna_gr_outside_f*, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_bt** – The auxiliary grammar callback for the outside (backtracking) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns

The current number of auxiliary grammar rules for the MFE C-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_m**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_f* cb, *vrna_gr_outside_f* cb_bt, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the M-decomposition (MFE version).

```
#include <ViennaRNA/grammar/mfe.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the M-decomposition, i.e. the multibranch loop decomposition stage.

While the inside rule (`cb`) computes a minimum free energy contribution for any subsequence the outside rule (`cb_bt`) is used for backtracking the corresponding structure.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

See also

vrna_gr_add_aux_f(), *vrna_gr_add_aux_c()*, *vrna_gr_add_aux_m1()*, *vrna_gr_add_aux()*, *vrna_gr_add_aux_exp_m()*, *vrna_gr_inside_f*, *vrna_gr_outside_f*, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_bt** – The auxiliary grammar callback for the outside (backtracking) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks

- **data_prepare** – A callback to prepare data
- **data_release** – A callback to free-up memory occupied by data

Returns
The current number of auxiliary grammar rules for the MFE M-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_m1**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_f* cb, *vrna_gr_outside_f* cb_bt, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the M1-decomposition (MFE version).

```
#include <ViennaRNA/grammar/mfe.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the M1-decomposition, i.e. the multibranch loop components with exactly one branch decomposition stage.

While the inside rule (cb) computes a minimum free energy contribution for any subsequence the outside rule (cb_bt) is used for backtracking the corresponding structure.

Both callbacks will be provided with the data pointer that can be used to store whatever data is needed in the callback evaluations. The data_prepare callback may be used to prepare the data just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the data_release callback to properly free the memory of data once it is not required anymore. Hence, it serves as a kind of destructor for data which will be called as soon as the grammar rules of fc are re-set to defaults or if the fc is destroyed.

See also

vrna_gr_add_aux_f(), *vrna_gr_add_aux_c()*, *vrna_gr_add_aux_m()*, *vrna_gr_add_aux()*, *vrna_gr_add_aux_exp_m1()*, *vrna_gr_inside_f*, *vrna_gr_outside_f*, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_bt** – The auxiliary grammar callback for the outside (backtracking) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare data
- **data_release** – A callback to free-up memory occupied by data

Returns
The current number of auxiliary grammar rules for the MFE M1-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_m2**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_f* cb, *vrna_gr_outside_f* cb_bt, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the M2-decomposition (MFE version).

```
#include <ViennaRNA/grammar/mfe.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the M2-decomposition, i.e. the multibranch loop components with at least two branches.

While the inside rule (`cb`) computes a minimum free energy contribution for any subsequence the outside rule (`cb_bt`) is used for backtracking the corresponding structure.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

See also

vrna_gr_add_aux_f(), *vrna_gr_add_aux_c()*, *vrna_gr_add_aux_m()*, *vrna_gr_add_aux()*, *vrna_gr_add_aux_exp_m1()*, *vrna_gr_inside_f*, *vrna_gr_outside_f*, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_bt** – The auxiliary grammar callback for the outside (backtracking) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns

The current number of auxiliary grammar rules for the MFE M2-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_f* cb, *vrna_gr_outside_f* cb_bt, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule (MFE version).

```
#include <ViennaRNA/grammar/mfe.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) as additional, independent decomposition steps.

While the inside rule (`cb`) computes a minimum free energy contribution for any subsequence the outside rule (`cb_bt`) is used for backtracking the corresponding structure.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

See also

vrna_gr_add_aux_f(), *vrna_gr_add_aux_c()*, *vrna_gr_add_aux_m()*, *vrna_gr_add_aux_m1()*, *vrna_gr_add_aux_exp()*, *vrna_gr_inside_f*, *vrna_gr_outside_f*, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_bt** – The auxiliary grammar callback for the outside (backtracking) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns

The current number of auxiliary grammar rules for MFE predictions, or 0 on error.

unsigned int **vrna_gr_add_aux_exp_f**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_exp_f* cb, *vrna_gr_outside_exp_f* cb_out, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the F-decomposition (partition function version).

```
#include <ViennaRNA/grammar/partfunc.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the F-decomposition, i.e. the external loop decomposition stage.

While the inside rule (`cb`) computes the partition function for any subsequence, the outside rule (`cb_out`) is used for (base pairing) probabilities.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

Bug:

Calling the `cb_out` callback is not implemented yet!

See also

vrna_gr_add_aux_exp_c(), *vrna_gr_add_aux_exp_m()*, *vrna_gr_add_aux_exp_m1()*, *vrna_gr_add_aux_exp()*, *vrna_gr_add_aux_f()*, *vrna_gr_inside_exp_f*, vrna_gr_outside_exp_f, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step

- **cb_out** – The auxiliary grammar callback for the outside (probability) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns
The current number of auxiliary grammar rules for the partition function F-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_exp_c**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_exp_f* cb, *vrna_gr_outside_exp_f* cb_out, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the C-decomposition (partition function version).

```
#include <ViennaRNA/grammar/partfunc.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the C-decomposition, i.e. the base pair decomposition stage.

While the inside rule (`cb`) computes the partition function for any subsequence, the outside rule (`cb_out`) is used for (base pairing) probabilities.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

Bug:
Calling the `cb_out` callback is not implemented yet!

See also

vrna_gr_add_aux_exp_f(), *vrna_gr_add_aux_exp_m()*, *vrna_gr_add_aux_exp_m1()*, *vrna_gr_add_aux_exp()*, *vrna_gr_add_aux_c()*, *vrna_gr_inside_exp_f*, vrna_gr_outside_exp_f, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_out** – The auxiliary grammar callback for the outside (probability) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns
The current number of auxiliary grammar rules for the partition function C-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_exp_m**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_exp_f* cb, *vrna_gr_outside_exp_f* cb_out, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the M-decomposition (partition function version).

```
#include <ViennaRNA/grammar/partfunc.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the M-decomposition, i.e. the multibranch loop decomposition stage.

While the inside rule (`cb`) computes the partition function for any subsequence, the outside rule (`cb_out`) is used for (base pairing) probabilities.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

Bug:

Calling the `cb_out` callback is not implemented yet!

See also

vrna_gr_add_aux_exp_f(), *vrna_gr_add_aux_exp_c()*, *vrna_gr_add_aux_exp_m1()*, *vrna_gr_add_aux_exp()*, *vrna_gr_add_aux_m()*, *vrna_gr_inside_exp_f*, vrna_gr_outside_exp_f, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_out** – The auxiliary grammar callback for the outside (probability) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns

The current number of auxiliary grammar rules for the partition function M-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_exp_m1**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_exp_f* cb, *vrna_gr_outside_exp_f* cb_out, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the M1-decomposition (partition function version).

```
#include <ViennaRNA/grammar/partfunc.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the M1-decomposition, i.e. the multibranch loop components with exactly one branch decomposition stage.

While the inside rule (`cb`) computes the partition function for any subsequence, the outside rule (`cb_out`) is used for (base pairing) probabilities.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

Bug:

Calling the `cb_out` callback is not implemented yet!

See also

vrna_gr_add_aux_exp_f(), *vrna_gr_add_aux_exp_c()*, *vrna_gr_add_aux_exp_m()*, *vrna_gr_add_aux_exp()*, *vrna_gr_add_aux_m1()*, *vrna_gr_inside_exp_f*, vrna_gr_outside_exp_f, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_out** – The auxiliary grammar callback for the outside (probability) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns

The current number of auxiliary grammar rules for the partition function M1-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_exp_m2**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_exp_f* cb, *vrna_gr_outside_exp_f* cb_out, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule for the M2-decomposition (partition function version).

```
#include <ViennaRNA/grammar/partfunc.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) in the M2-decomposition, i.e. the multibranch loop components with at least two branches.

While the inside rule (`cb`) computes the partition function for any subsequence, the outside rule (`cb_out`) is used for (base pairing) probabilities.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required

anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

Bug:

Calling the `cb_out` callback is not implemented yet!

See also

vrna_gr_add_aux_exp_f(), *vrna_gr_add_aux_exp_c()*, *vrna_gr_add_aux_exp_m()*, *vrna_gr_add_aux_exp()*, *vrna_gr_add_aux_m1()*, *vrna_gr_inside_exp_f*, vrna_gr_outside_exp_f, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_out** – The auxiliary grammar callback for the outside (probability) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns

The current number of auxiliary grammar rules for the partition function M2-decomposition stage, or 0 on error.

unsigned int **vrna_gr_add_aux_exp**(*vrna_fold_compound_t* *fc, *vrna_gr_inside_exp_f* cb, *vrna_gr_outside_exp_f* cb_out, void *data, *vrna_auxdata_prepare_f* data_prepare, *vrna_auxdata_free_f* data_release)

Add an auxiliary grammar rule (partition function version).

```
#include <ViennaRNA/grammar/partfunc.h>
```

This function binds callback functions for auxiliary grammar rules (inside and outside) as additional, independent decomposition steps.

While the inside rule (`cb`) computes the partition function for any subsequence, the outside rule (`cb_out`) is used for (base pairing) probabilities.

Both callbacks will be provided with the `data` pointer that can be used to store whatever data is needed in the callback evaluations. The `data_prepare` callback may be used to prepare the `data` just before the start of the recursions. If present, it will be called prior the actual decompositions automatically. You may use the `data_release` callback to properly free the memory of `data` once it is not required anymore. Hence, it serves as a kind of destructor for `data` which will be called as soon as the grammar rules of `fc` are re-set to defaults or if the `fc` is destroyed.

Bug:

Calling the `cb_out` callback is not implemented yet!

See also

vrna_gr_add_aux_exp_f(), *vrna_gr_add_aux_exp_c()*, *vrna_gr_add_aux_exp_m()*, *vrna_gr_add_aux_exp_m1()*, *vrna_gr_add_aux()*, *vrna_gr_inside_exp_f*, vrna_gr_outside_exp_f, *vrna_fold_compound_t*, *vrna_auxdata_prepare_f*, *vrna_auxdata_free_f*

Parameters

- **fc** – The fold compound that is to be extended by auxiliary grammar rules
- **cb** – The auxiliary grammar callback for the inside step
- **cb_out** – The auxiliary grammar callback for the outside (probability) step
- **data** – A pointer to some data that will be passed through to the inside and outside callbacks
- **data_prepare** – A callback to prepare `data`
- **data_release** – A callback to free-up memory occupied by `data`

Returns

The current number of auxiliary grammar rules for partition function predictions, or 0 on error.

4.2 Utilities

4.2.1 Basic Utilities

Defines

PUBLIC

```
#include <ViennaRNA/utils/basic.h>
```

PRIVATE

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_UNUSED

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_ERROR

Output flag of *get_input_line()*: *"An ERROR has occured, maybe EOF"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_QUIT

Output flag of *get_input_line()*: *"the user requested quitting the program"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_MISC

Output flag of *get_input_line()*: *"something was read"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_FASTA_HEADER

Input/Output flag of *get_input_line()* :

if used as input option this tells *get_input_line()* that the data to be read should comply with the FASTA format.

```
#include <ViennaRNA/utils/basic.h>
```

the function will return this flag if a fasta header was read

VRNA_INPUT_SEQUENCE

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_CONSTRAINT

Input flag for *get_input_line()* :

Tell *get_input_line()* that we assume to read a structure constraint.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_NO_TRUNCATION

Input switch for *get_input_line()*: *"do not trunkate the line by eliminating white spaces at end of line"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_NO_REST

Input switch for *vrna_file_fasta_read_record()*: *"do fill rest array"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_NO_SPAN

Input switch for *vrna_file_fasta_read_record()*: *"never allow data to span more than one line"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_NOSKIP_BLANK_LINES

Input switch for *vrna_file_fasta_read_record()*: *"do not skip empty lines"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_BLANK_LINE

Output flag for *vrna_file_fasta_read_record()*: *"read an empty line"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_NOSKIP_COMMENTS

Input switch for *get_input_line()*: *"do not skip comment lines"*.

```
#include <ViennaRNA/utils/basic.h>
```

VRNA_INPUT_COMMENT

Output flag for *vrna_file_fasta_read_record()*: *"read a comment"*.

```
#include <ViennaRNA/utils/basic.h>
```

MIN2(A, B)

Get the minimum of two comparable values.

```
#include <ViennaRNA/utils/basic.h>
```

MAX2(A, B)

Get the maximum of two comparable values.

```
#include <ViennaRNA/utils/basic.h>
```

MIN3(A, B, C)

Get the minimum of three comparable values.

```
#include <ViennaRNA/utils/basic.h>
```

MAX3(A, B, C)

Get the maximum of three comparable values.

```
#include <ViennaRNA/utils/basic.h>
```

Functions

void ***vrna_alloc**(size_t size)

Allocate space safely.

```
#include <ViennaRNA/utils/basic.h>
```

Parameters

- **size** – The size of the memory to be allocated in bytes

Returns

A pointer to the allocated memory

void ***vrna_realloc**(void *p, size_t size)

Reallocate space safely.

```
#include <ViennaRNA/utils/basic.h>
```

Parameters

- **p** – A pointer to the memory region to be reallocated
- **size** – The size of the memory to be allocated in bytes

Returns

A pointer to the newly allocated memory

void **vrna_init_rand**(void)

Initialize seed for random number generator.

```
#include <ViennaRNA/utils/basic.h>
```

See also

vrna_init_rand_seed(), *vrna_urn()*

void **vrna_init_rand_seed**(unsigned int seed)

Initialize the random number generator with a pre-defined seed.

```
#include <ViennaRNA/utils/basic.h>
```

SWIG Wrapper Notes:

This function is available as an overloaded function **init_rand()** where the argument `seed` is optional. See, e.g. *`RNA.init_rand()`* in the *The RNA Python module*.

See also

vrna_init_rand(), *vrna_urn()*

Parameters

- **seed** – The seed for the random number generator

double **vrna_urn**(void)

get a random number from [0..1]

```
#include <ViennaRNA/utils/basic.h>
```

See also

vrna_int_urn(), *vrna_init_rand()*, *vrna_init_rand_seed()*

Note

Usually implemented by calling *erand48()*.

Returns

A random number in range [0..1]

int **vrna_int_urn**(int from, int to)

Generates a pseudo random integer in a specified range.

```
#include <ViennaRNA/utils/basic.h>
```

See also

vrna_urn(), *vrna_init_rand()*

Parameters

- **from** – The first number in range
- **to** – The last number in range

Returns

A pseudo random number in range [from, to]

char ***vrna_time_stamp**(void)

Get a timestamp.

```
#include <ViennaRNA/utils/basic.h>
```

Returns a string containing the current date in the format

```
Fri Mar 19 21:10:57 1993
```

Returns

A string containing the timestamp

unsigned int **get_input_line**(char **string, unsigned int options)

```
#include <ViennaRNA/utils/basic.h>
```

Retrieve a line from 'stdin' savely while skipping comment characters and other features This function returns the type of input it has read if recognized. An option argument allows one to switch between different reading modes.

Currently available options are: *VRNA_INPUT_COMMENT*, *VRNA_INPUT_NOSKIP_COMMENTS*, *VRNA_INPUT_NO_TRUNCATION*

pass a collection of options as one value like this:

```
get_input_line(string, option_1 | option_2 | option_n)
```

If the function recognizes the type of input, it will report it in the return value. It also reports if a user defined 'quit' command (-sign on 'stdin') was given. Possible return values are:

VRNA_INPUT_FASTA_HEADER, *VRNA_INPUT_ERROR*, *VRNA_INPUT_MISC*, *VRNA_INPUT_QUIT*

Parameters

- **string** – A pointer to the character array that contains the line read
- **options** – A collection of options for switching the functions behavior

Returns

A flag with information about what has been read

int ***vrna_idx_row_wise**(unsigned int length)

Get an index mapper array (iindx) for accessing the energy matrices, e.g. in partition function related functions.

```
#include <ViennaRNA/utils/basic.h>
```

Access of a position "(i,j)" is then accomplished by using

```
(i,j) ~ iindx[i]-j
```

This function is necessary as most of the two-dimensional energy matrices are actually one-dimensional arrays throughout the ViennaRNA Package

Consult the implemented code to find out about the mapping formula ;)

See also

vrna_idx_col_wise()

Parameters

- **length** – The length of the RNA sequence

Returns

The mapper array

int ***vrna_idx_col_wise**(unsigned int length)

Get an index mapper array (indx) for accessing the energy matrices, e.g. in MFE related functions.

```
#include <ViennaRNA/utils/basic.h>
```

Access of a position "(i,j)" is then accomplished by using

```
(i,j) ~ indx[j]+i
```

This function is necessary as most of the two-dimensional energy matrices are actually one-dimensional arrays throughout the ViennaRNAPackage

Consult the implemented code to find out about the mapping formula ;)

See also

vrna_idx_row_wise()

Parameters

- **length** – The length of the RNA sequence

Returns

The mapper array

Variables

unsigned short **xsubi**[3]

Current 48 bit random number.

```
#include <ViennaRNA/utils/basic.h>
```

This variable is used by *vrna_urn()*. These should be set to some random number seeds before the first call to *vrna_urn()*.

See also

vrna_urn()

4.2.2 (Abstract) Data Structures

All datastructures and typedefs shared among the ViennaRNA Package can be found here.

The Fold Compound

API to create, modify, and delete *vrna_fold_compound_t* data structures.

Defines

VRNA_STATUS_MFE_PRE

Status message indicating that MFE computations are about to begin.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound_t.stat_cb, *vrna_recursion_status_f()*, *vrna_mfe()*, *vrna_fold()*, *vrna_circfold()*, *vrna_alifold()*, *vrna_circalifold()*, *vrna_cofold()*

VRNA_STATUS_MFE_POST

Status message indicating that MFE computations are finished.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound_t.stat_cb, *vrna_recursion_status_f()*, *vrna_mfe()*, *vrna_fold()*, *vrna_circfold()*, *vrna_alifold()*, *vrna_circalifold()*, *vrna_cofold()*

VRNA_STATUS_PF_PRE

Status message indicating that Partition function computations are about to begin.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound_t.stat_cb, *vrna_recursion_status_f()*, *vrna_pf()*

VRNA_STATUS_PF_POST

Status message indicating that Partition function computations are finished.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound_t.stat_cb, *vrna_recursion_status_f()*, *vrna_pf()*

VRNA_OPTION_DEFAULT

Option flag to specify default settings/requirements.

```
#include <ViennaRNA/fold_compound.h>
```

VRNA_OPTION_MFE

Option flag to specify requirement of Minimum Free Energy (MFE) DP matrices and corresponding set of energy parameters.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound(), *vrna_fold_compound_comparative()*, *VRNA_OPTION_EVAL_ONLY*

VRNA_OPTION_PF

Option flag to specify requirement of Partition Function (PF) DP matrices and corresponding set of Boltzmann factors.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound(), *vrna_fold_compound_comparative()*, *VRNA_OPTION_EVAL_ONLY*

VRNA_OPTION_HYBRID

Option flag to specify requirement of dimer DP matrices.

```
#include <ViennaRNA/fold_compound.h>
```

VRNA_OPTION_EVAL_ONLY

Option flag to specify that neither MFE, nor PF DP matrices are required.

```
#include <ViennaRNA/fold_compound.h>
```

Use this flag in conjuntion with *VRNA_OPTION_MFE*, and *VRNA_OPTION_PF* to save memory for a *vrna_fold_compound_t* obtained from *vrna_fold_compound()*, or *vrna_fold_compound_comparative()* in cases where only energy evaluation but no structure prediction is required.

See also

vrna_fold_compound(), *vrna_fold_compound_comparative()*, *vrna_eval_structure()*

VRNA_OPTION_WINDOW

Option flag to specify requirement of DP matrices for local folding approaches.

```
#include <ViennaRNA/fold_compound.h>
```

VRNA_OPTION_F5

```
#include <ViennaRNA/fold_compound.h>
```

VRNA_OPTION_F3

```
#include <ViennaRNA/fold_compound.h>
```

VRNA_OPTION_WINDOW_F5

```
#include <ViennaRNA/fold_compound.h>
```

VRNA_OPTION_WINDOW_F3

```
#include <ViennaRNA/fold_compound.h>
```

Typedefs

typedef struct *vrna_fc_s* **vrna_fold_compound_t**

Typename for the fold_compound data structure *vrna_fc_s*.

```
#include <ViennaRNA/fold_compound.h>
```

typedef void (***vrna_auxdata_free_f**)(void *data)

Callback to free memory allocated for auxiliary user-provided data.

```
#include <ViennaRNA/fold_compound.h>
```

This type of user-implemented function usually deletes auxiliary data structures. The user must take care to free all the memory occupied by the data structure passed.

Notes on Callback Functions:

This callback is supposed to free memory occupied by an auxiliary data structure. It will be called when the *vrna_fold_compound_t* is erased from memory through a call to *vrna_fold_compound_free()* and will be passed the address of memory previously bound to the *vrna_fold_compound_t* via *vrna_fold_compound_add_auxdata()*.

See also

vrna_fold_compound_add_auxdata(), *vrna_fold_compound_free()*, *vrna_fold_compound_add_callback()*

Param data
The data that needs to be free'd

typedef int (***vrna_auxdata_prepare_f**)(*vrna_fold_compound_t* *fc, void *data, unsigned int event, void *event_data)

Callback to prepare user-provided data based on some event.

```
#include <ViennaRNA/fold_compound.h>
```

Param fc
The fold compound the event was emitted from

Param data
The user-defined data that may be prepared according to the event

Param event
The event

Param event_data
Some additional data corresponding to the event

Return
non-zero if the preparation was successful, 0 otherwise

typedef void **vrna_callback_free_auxdata**(void *data)

Callback to free memory allocated for auxiliary user-provided data.

```
#include <ViennaRNA/fold_compound.h>
```

Deprecated:
Use *vrna_auxdata_free_f(void *data)* instead!

typedef void (***vrna_recursion_status_f**)(*vrna_fold_compound_t* *fc, unsigned char status, void *data)

Callback to perform specific user-defined actions before, or after recursive computations.

```
#include <ViennaRNA/fold_compound.h>
```

Notes on Callback Functions:
This function will be called to notify a third-party implementation about the status of a currently ongoing recursion. The purpose of this callback mechanism is to provide users with a simple way to ensure pre- and post conditions for auxiliary mechanisms attached to our implementations.

See also

vrna_fold_compound_add_auxdata(), *vrna_fold_compound_add_callback()*, *vrna_mfe()*, *vrna_pf()*, *VRNA_STATUS_MFE_PRE*, *VRNA_STATUS_MFE_POST*, *VRNA_STATUS_PF_PRE*, *VRNA_STATUS_PF_POST*

Param fc
The fold compound emitting the status

Param status
The status indicator

Param data
The data structure that was assigned with *vrna_fold_compound_add_auxdata()*

typedef void **vrna_callback_recursion_status**(unsigned char status, void *data)

```
#include <ViennaRNA/fold_compound.h>
```

Enums

enum **vrna_fc_type_e**

An enumerator that is used to specify the type of a *vrna_fold_compound_t*.

```
#include <ViennaRNA/fold_compound.h>
```

Values:

enumerator **VRNA_FC_TYPE_SINGLE**

Type is suitable for single, and hybridizing sequences

enumerator **VRNA_FC_TYPE_COMPARATIVE**

Type is suitable for sequence alignments (consensus structure prediction)

Functions

vrna_fold_compound_t ***vrna_fold_compound**(const char *sequence, const *vrna_md_t* *md_p, unsigned int options)

Retrieve a *vrna_fold_compound_t* data structure for single sequences and hybridizing sequences.

```
#include <ViennaRNA/fold_compound.h>
```

This function provides an easy interface to obtain a prefilled *vrna_fold_compound_t* by passing a single sequence, or two contatenated sequences as input. For the latter, sequences need to be seperated by an '&' character like this:

```
char *sequence = "GGGG&CCCC";
```

The optional parameter `md_p` can be used to specify the model details for successive computations based on the content of the generated *vrna_fold_compound_t*. Passing NULL will instruct the function to use default model details. The third parameter `options` may be used to specify dynamic programming (DP) matrix requirements.

Options

- *VRNA_OPTION_DEFAULT* - Option flag to specify default settings/requirements.

- *VRNA_OPTION_MFE* - Option flag to specify requirement of Minimum Free Energy (MFE) DP matrices and corresponding set of energy parameters.
- *VRNA_OPTION_PF* - Option flag to specify requirement of Partition Function (PF) DP matrices and corresponding set of Boltzmann factors.
- *VRNA_OPTION_WINDOW* - Option flag to specify requirement of DP matrices for local folding approaches.

The above options may be OR-ed together.

If you just need the folding compound serving as a container for your data, you can simply pass *VRNA_OPTION_DEFAULT* to the `option` parameter. This creates a *vrna_fold_compound_t* without DP matrices, thus saving memory. Subsequent calls of any structure prediction function will then take care of allocating the memory required for the DP matrices. If you only intend to evaluate structures instead of actually predicting them, you may use the *VRNA_OPTION_EVAL_ONLY* macro. This will seriously speedup the creation of the *vrna_fold_compound_t*.

See also

vrna_fold_compound_free(), *vrna_fold_compound_comparative()*, *vrna_md_t*

Note

The sequence string must be uppercase, and should contain only RNA (resp. DNA) alphabet depending on what energy parameter set is used

Parameters

- **sequence** – A single sequence, or two concatenated sequences seperated by an '&' character
- **md_p** – An optional set of model details
- **options** – The options for DP matrices memory allocation

Returns

A prefilled *vrna_fold_compound_t* ready to be used for computations (may be `NULL` on error)

vrna_fold_compound_t ***vrna_fold_compound_comparative**(const char **sequences, *vrna_md_t* *md_p, unsigned int options)

Retrieve a *vrna_fold_compound_t* data structure for sequence alignments.

```
#include <ViennaRNA/fold_compound.h>
```

This function provides an easy interface to obtain a prefilled *vrna_fold_compound_t* by passing an alignment of sequences.

The optional parameter `md_p` can be used to specify the model details for successive computations based on the content of the generated *vrna_fold_compound_t*. Passing NULL will instruct the function to use default model details. The third parameter `options` may be used to specify dynamic programming (DP) matrix requirements.

Options

- *VRNA_OPTION_DEFAULT* - Option flag to specify default settings/requirements.

- *VRNA_OPTION_MFE* - Option flag to specify requirement of Minimum Free Energy (MFE) DP matrices and corresponding set of energy parameters.
- *VRNA_OPTION_PF* - Option flag to specify requirement of Partition Function (PF) DP matrices and corresponding set of Boltzmann factors.
- *VRNA_OPTION_WINDOW* - Option flag to specify requirement of DP matrices for local folding approaches.

The above options may be OR-ed together.

If you just need the folding compound serving as a container for your data, you can simply pass *VRNA_OPTION_DEFAULT* to the `option` parameter. This creates a *vrna_fold_compound_t* without DP matrices, thus saving memory. Subsequent calls of any structure prediction function will then take care of allocating the memory required for the DP matrices. If you only intend to evaluate structures instead of actually predicting them, you may use the *VRNA_OPTION_EVAL_ONLY* macro. This will seriously speedup the creation of the *vrna_fold_compound_t*.

See also

vrna_fold_compound_free(), *vrna_fold_compound()*, *vrna_md_t*, *VRNA_OPTION_MFE*, *VRNA_OPTION_PF*, *VRNA_OPTION_EVAL_ONLY*, read_clustal()

Note

The sequence strings must be uppercase, and should contain only RNA (resp. DNA) alphabet including gap characters depending on what energy parameter set is used.

Parameters

- **sequences** – A sequence alignment including 'gap' characters
- **md_p** – An optional set of model details
- **options** – The options for DP matrices memory allocation

Returns

A prefilled *vrna_fold_compound_t* ready to be used for computations (may be `NULL` on error)

vrna_fold_compound_t ***vrna_fold_compound_comparative2**(const char **sequences, const char **names, const unsigned char *orientation, const unsigned long long *start, const unsigned long long *genome_size, *vrna_md_t* *md_p, unsigned int options)

```
#include <ViennaRNA/fold_compound.h>
```

vrna_fold_compound_t ***vrna_fold_compound_TwoD**(const char *sequence, const char *s1, const char *s2, *vrna_md_t* *md_p, unsigned int options)

```
#include <ViennaRNA/fold_compound.h>
```

int **vrna_fold_compound_prepare**(*vrna_fold_compound_t* *fc, unsigned int options)

```
#include <ViennaRNA/fold_compound.h>
```

void **vrna_fold_compound_free**(*vrna_fold_compound_t* *fc)

Free memory occupied by a *vrna_fold_compound_t*.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound(), *vrna_fold_compound_comparative()*, *vrna_mx_mfe_free()*, *vrna_mx_pf_free()*

Parameters

- **fc** – The *vrna_fold_compound_t* that is to be erased from memory

void **vrna_fold_compound_add_auxdata**(*vrna_fold_compound_t* *fc, void *data, *vrna_auxdata_free_f* f)

Add auxiliary data to the *vrna_fold_compound_t*.

```
#include <ViennaRNA/fold_compound.h>
```

This function allows one to bind arbitrary data to a *vrna_fold_compound_t* which may later on be used by one of the callback functions, e.g. *vrna_recursion_status_f()*. To allow for proper cleanup of the memory occupied by this auxiliary data, the user may also provide a pointer to a cleanup function that free's the corresponding memory. This function will be called automatically when the *vrna_fold_compound_t* is free'd with *vrna_fold_compound_free()*.

See also

vrna_auxdata_free_f()

Note

Before attaching the arbitrary data pointer, this function will call the *vrna_auxdata_free_f()* on any pre-existing data that is already attached.

Parameters

- **fc** – The fold_compound the arbitrary data pointer should be associated with
- **data** – A pointer to an arbitrary data structure
- **f** – A pointer to function that free's memory occupied by the arbitrary data (May be NULL)

void **vrna_fold_compound_add_callback**(*vrna_fold_compound_t* *fc, *vrna_recursion_status_f* f)

Add a recursion status callback to the *vrna_fold_compound_t*.

```
#include <ViennaRNA/fold_compound.h>
```

Binding a recursion status callback function to a *vrna_fold_compound_t* allows one to perform arbitrary operations just before, or after an actual recursive computations, e.g. MFE prediction, is performed by the RNAlib. The callback function will be provided with a pointer to its *vrna_fold_compound_t*, and a status message. Hence, it has complete access to all variables that incluence the recursive computations.

See also

vrna_recursion_status_f(), *vrna_fold_compound_t*, *VRNA_STATUS_MFE_PRE*, *VRNA_STATUS_MFE_POST*, *VRNA_STATUS_PF_PRE*, *VRNA_STATUS_PF_POST*

Parameters

- **fc** – The fold_compound the callback function should be attached to
- **f** – The pointer to the recursion status callback function

struct **vrna_fc_s**

#include <ViennaRNA/fold_compound.h> The most basic data structure required by many functions throughout the RNAlib.

SWIG Wrapper Notes:

This data structure is wrapped as class `fold_compound` with several related functions attached as methods.

A new `fold_compound` can be obtained by calling one of its constructors:

- `fold_compound(seq)` - Initialize with a single sequence, or two concatenated sequences separated by an ampersand character `&` (for cofolding)
- `fold_compound(aln)` - Initialize with a sequence alignment *aln* stored as a list of sequences (with gap characters).

The resulting object has a list of attached methods which in most cases directly correspond to functions that mainly operate on the corresponding `C` data structure:

- `type()` - Get the type of the *fold_compound* (See *vrna_fc_type_e*)
- `length()` - Get the length of the sequence(s) or alignment stored within the `fold_compound`.

See, e.g. `RNA.fold_compound` in the *The RNA Python module*.

See also

vrna_fold_compound_t.type, *vrna_fold_compound()*, *vrna_fold_compound_comparative()*, *vrna_fold_compound_free()*, *VRNA_FC_TYPE_SINGLE*, *VRNA_FC_TYPE_COMPARATIVE*

Note

Please read the documentation of this data structure carefully! Some attributes are only available for specific types this data structure can adopt.

Warning

Reading/Writing from/to attributes that are not within the scope of the current type usually result in undefined behavior!

Common data fields

const *vrna_fc_type_e* **type**

The type of the *vrna_fold_compound_t*.

```
#include <ViennaRNA/fold_compound.h>
```

Currently possible values are *VRNA_FC_TYPE_SINGLE*, and *VRNA_FC_TYPE_COMPARATIVE*

> **Warning**
>
> Do not edit this attribute, it will be automagically set by the corresponding get() methods for the *vrna_fold_compound_t*. The value specified in this attribute dictates the set of other attributes to use within this data structure.

unsigned int **length**

The length of the sequence (or sequence alignment).

```
#include <ViennaRNA/fold_compound.h>
```

int **cutpoint**

The position of the (cofold) cutpoint within the provided sequence. If there is no cutpoint, this field will be set to -1.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int ***strand_number**

The strand number a particular nucleotide is associated with.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int ***strand_order**

The strand order, i.e. permutation of current concatenated sequence.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int ***strand_order_uniq**

The strand order array where identical sequences have the same ID.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int ***strand_start**

The start position of a particular strand within the current concatenated sequence.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int ***strand_end**

The end (last) position of a particular strand within the current concatenated sequence.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int **strands**

Number of interacting strands.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_seq_t ***nucleotides**

Set of nucleotide sequences.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_msa_t ***alignment**

Set of alignments.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_hc_t ***hc**

The hard constraints data structure used for structure prediction.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_mx_mfe_t ***matrices**

The MFE DP matrices.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_mx_pf_t ***exp_matrices**

The PF DP matrices.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_param_t ***params**

The precomputed free energy contributions for each type of loop.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_exp_param_t ***exp_params**

The precomputed free energy contributions as Boltzmann factors.

```
#include <ViennaRNA/fold_compound.h>
```

int ***iindx**

DP matrix accessor.

```
#include <ViennaRNA/fold_compound.h>
```

int ***jindx**

DP matrix accessor.

```
#include <ViennaRNA/fold_compound.h>
```

User-defined data fields

vrna_recursion_status_f **stat_cb**

Recursion status callback (usually called just before, and after recursive computations in the library.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_recursion_status_f(), *vrna_fold_compound_add_callback()*

void ***auxdata**

A pointer to auxiliary, user-defined data.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound_add_auxdata(), *vrna_fold_compound_t.free_auxdata*

vrna_auxdata_free_f **free_auxdata**

A callback to free auxiliary user data whenever the fold_compound itself is free'd.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_fold_compound_t.auxdata, *vrna_auxdata_free_f()*

Secondary Structure Decomposition (grammar) related data fields

vrna_sd_t ***domains_struc**

Additional structured domains.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_ud_t ***domains_up**

Additional unstructured domains.

```
#include <ViennaRNA/fold_compound.h>
```

vrna_gr_aux_t **aux_grammar**

Additional decomposition grammar rules.

```
#include <ViennaRNA/fold_compound.h>
```

Additional data fields for Distance Class Partitioning

These data fields are typically populated with meaningful data only if used in the context of Distance Class Partitioning

unsigned int **maxD1**

Maximum allowed base pair distance to first reference.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int **maxD2**

Maximum allowed base pair distance to second reference.

```
#include <ViennaRNA/fold_compound.h>
```

short *__reference_pt1__

A pairtable of the first reference structure.

```
#include <ViennaRNA/fold_compound.h>
```

short *__reference_pt2__

A pairtable of the second reference structure.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int *__referenceBPs1__

Matrix containing number of basepairs of reference structure1 in interval [i,j].

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int *__referenceBPs2__

Matrix containing number of basepairs of reference structure2 in interval [i,j].

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int *__bpdist__

Matrix containing base pair distance of reference structure 1 and 2 on interval [i,j].

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int *__mm1__

Maximum matching matrix, reference struct 1 disallowed.

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int ***mm2**

Maximum matching matrix, reference struct 2 disallowed.

```
#include <ViennaRNA/fold_compound.h>
```

Additional data fields for local folding

These data fields are typically populated with meaningful data only if used in the context of local folding

int **window_size**

window size for local folding sliding window approach

```
#include <ViennaRNA/fold_compound.h>
```

char ****ptype_local**

Pair type array (for local folding).

```
#include <ViennaRNA/fold_compound.h>
```

Public Members

union *vrna_fc_s*.[anonymous] **[anonymous]**

```
#include <ViennaRNA/fold_compound.h>
```

[union].__unnamed0__

Public Members

struct *vrna_fc_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/fold_compound.h>
```

struct *vrna_fc_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/fold_compound.h>
```

[struct].__unnamed0__.__unnamed0__

Data fields available for single/hybrid structure prediction

char ***sequence**

The input sequence string.

```
#include <ViennaRNA/fold_compound.h>
```

> **Warning**
>
> Only available if

type==VRNA_FC_TYPE_SINGLE

short ***sequence_encoding**

Numerical encoding of the input sequence.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_sequence_encode()

Warning

Only available if

type==VRNA_FC_TYPE_SINGLE

short ***encoding5**

```
#include <ViennaRNA/fold_compound.h>
```

short ***encoding3**

```
#include <ViennaRNA/fold_compound.h>
```

short ***sequence_encoding2**

```
#include <ViennaRNA/fold_compound.h>
```

char ***ptype**

Pair type array.

```
#include <ViennaRNA/fold_compound.h>
```

Contains the numerical encoding of the pair type for each pair (i,j) used in MFE, Partition function and Evaluation computations.

See also

vrna_idx_col_wise(), *vrna_ptypes()*

Note

This array is always indexed via jindx, in contrast to previously different indexing between mfe and pf variants!

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

char ***ptype_pf_compat**

ptype array indexed via iindx

```
#include <ViennaRNA/fold_compound.h>
```

Deprecated:

This attribute will vanish in the future! It's meant for backward compatibility only!

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

vrna_sc_t ***sc**

The soft constraints for usage in structure prediction and evaluation.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

[struct].__unnamed0__.__unnamed1__

Data fields for consensus structure prediction

char ****sequences**

The aligned sequences.

```
#include <ViennaRNA/fold_compound.h>
```

Note

The end of the alignment is indicated by a NULL pointer in the second dimension

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

unsigned int **n_seq**

The number of sequences in the alignment.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

char ***cons_seq**

The consensus sequence of the aligned sequences.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short ***S_cons**

Numerical encoding of the consensus sequence.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short ****S**

Numerical encoding of the sequences in the alignment.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short **S5

S5[s][i] holds next base 5' of i in sequence s.

```
#include <ViennaRNA/fold_compound.h>
```

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_COMPARATIVE
> ```

short **S3

Sl[s][i] holds next base 3' of i in sequence s.

```
#include <ViennaRNA/fold_compound.h>
```

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_COMPARATIVE
> ```

char **Ss

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int **a2s

```
#include <ViennaRNA/fold_compound.h>
```

int *pscore

Precomputed array of pair types expressed as pairing scores.

```
#include <ViennaRNA/fold_compound.h>
```

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_COMPARATIVE
> ```

int **pscore_local

Precomputed array of pair types expressed as pairing scores.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short ***pscore_pf_compat**

Precomputed array of pair types expressed as pairing scores indexed via iindx.

```
#include <ViennaRNA/fold_compound.h>
```

Deprecated:

This attribute will vanish in the future!

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

vrna_sc_t ***scs**

A set of soft constraints (for each sequence in the alignment).

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

int **oldAliEn**

```
#include <ViennaRNA/fold_compound.h>
```

vrna_fc_s.[union].__unnamed0__

Public Members

struct *vrna_fc_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/fold_compound.h>
```

struct *vrna_fc_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/fold_compound.h>
```

[struct].__unnamed0__.__unnamed0__

Data fields available for single/hybrid structure prediction

char ***sequence**

The input sequence string.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

short ***sequence_encoding**

Numerical encoding of the input sequence.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_sequence_encode()

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

short ***encoding5**

```
#include <ViennaRNA/fold_compound.h>
```

short ***encoding3**

```
#include <ViennaRNA/fold_compound.h>
```

short ***sequence_encoding2**

```
#include <ViennaRNA/fold_compound.h>
```

char ***ptype**

Pair type array.

```
#include <ViennaRNA/fold_compound.h>
```

Contains the numerical encoding of the pair type for each pair (i,j) used in MFE, Partition function and Evaluation computations.

See also

vrna_idx_col_wise(), *vrna_ptypes()*

Note

This array is always indexed via jindx, in contrast to previously different indexing between mfe and pf variants!

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

char ***ptype_pf_compat**

ptype array indexed via iindx

```
#include <ViennaRNA/fold_compound.h>
```

Deprecated:

This attribute will vanish in the future! It's meant for backward compatibility only!

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

vrna_sc_t ***sc**

The soft constraints for usage in structure prediction and evaluation.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

[struct].__unnamed0__.__unnamed1__

Data fields for consensus structure prediction

char ****sequences**

The aligned sequences.

```
#include <ViennaRNA/fold_compound.h>
```

Note

The end of the alignment is indicated by a NULL pointer in the second dimension

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

unsigned int **n_seq**

The number of sequences in the alignment.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

char ***cons_seq**

The consensus sequence of the aligned sequences.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short ***S_cons**

Numerical encoding of the consensus sequence.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short ****S**

Numerical encoding of the sequences in the alignment.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short **S5

S5[s][i] holds next base 5' of i in sequence s.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short **S3

Sl[s][i] holds next base 3' of i in sequence s.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

char **Ss

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int **a2s

```
#include <ViennaRNA/fold_compound.h>
```

int *pscore

Precomputed array of pair types expressed as pairing scores.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

int ****pscore_local**

Precomputed array of pair types expressed as pairing scores.

```
#include <ViennaRNA/fold_compound.h>
```

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_COMPARATIVE
> ```

short ***pscore_pf_compat**

Precomputed array of pair types expressed as pairing scores indexed via iindx.

```
#include <ViennaRNA/fold_compound.h>
```

Deprecated:

This attribute will vanish in the future!

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_COMPARATIVE
> ```

vrna_sc_t ****scs**

A set of soft constraints (for each sequence in the alignment).

```
#include <ViennaRNA/fold_compound.h>
```

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_COMPARATIVE
> ```

int **oldAliEn**

```
#include <ViennaRNA/fold_compound.h>
```

vrna_fc_s.[struct].__unnamed0__.__unnamed0__

Data fields available for single/hybrid structure prediction

char ***sequence**

The input sequence string.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

short ***sequence_encoding**

Numerical encoding of the input sequence.

```
#include <ViennaRNA/fold_compound.h>
```

See also

vrna_sequence_encode()

Warning

Only available if

```
type==VRNA_FC_TYPE_SINGLE
```

short ***encoding5**

```
#include <ViennaRNA/fold_compound.h>
```

short ***encoding3**

```
#include <ViennaRNA/fold_compound.h>
```

short ***sequence_encoding2**

```
#include <ViennaRNA/fold_compound.h>
```

char ***ptype**

Pair type array.

```
#include <ViennaRNA/fold_compound.h>
```

Contains the numerical encoding of the pair type for each pair (i,j) used in MFE, Partition function and Evaluation computations.

See also

vrna_idx_col_wise(), *vrna_ptypes()*

> **Note**
>
> This array is always indexed via jindx, in contrast to previously different indexing between mfe and pf variants!

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_SINGLE
> ```

char ***ptype_pf_compat**

ptype array indexed via iindx

```
#include <ViennaRNA/fold_compound.h>
```

Deprecated:

This attribute will vanish in the future! It's meant for backward compatibility only!

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_SINGLE
> ```

vrna_sc_t ***sc**

The soft constraints for usage in structure prediction and evaluation.

```
#include <ViennaRNA/fold_compound.h>
```

> **Warning**
>
> Only available if
>
> ```
> type==VRNA_FC_TYPE_SINGLE
> ```

vrna_fc_s.[struct].__unnamed0__.__unnamed1__

Data fields for consensus structure prediction

char ****sequences**

The aligned sequences.

```
#include <ViennaRNA/fold_compound.h>
```

Note

The end of the alignment is indicated by a NULL pointer in the second dimension

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

unsigned int **n_seq**

The number of sequences in the alignment.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

char ***cons_seq**

The consensus sequence of the aligned sequences.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short ***S_cons**

Numerical encoding of the consensus sequence.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short ****S**

Numerical encoding of the sequences in the alignment.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short **S5

S5[s][i] holds next base 5' of i in sequence s.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short **S3

Sl[s][i] holds next base 3' of i in sequence s.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

char **Ss

```
#include <ViennaRNA/fold_compound.h>
```

unsigned int **a2s

```
#include <ViennaRNA/fold_compound.h>
```

int *pscore

Precomputed array of pair types expressed as pairing scores.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

int **pscore_local

Precomputed array of pair types expressed as pairing scores.

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

short *pscore_pf_compat

Precomputed array of pair types expressed as pairing scores indexed via iindx.

```
#include <ViennaRNA/fold_compound.h>
```

Deprecated:

This attribute will vanish in the future!

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

vrna_sc_t **scs

A set of soft constraints (for each sequence in the alignment).

```
#include <ViennaRNA/fold_compound.h>
```

Warning

Only available if

```
type==VRNA_FC_TYPE_COMPARATIVE
```

int oldAliEn

```
#include <ViennaRNA/fold_compound.h>
```

The Dynamic Programming Matrices

This module provides interfaces that deal with creation and destruction of dynamic programming matrices used within the RNAlib.

Defines

VRNA_MX_FLAG_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_MS5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_MS3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_G

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

VRNA_MX_FLAG_MAX

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Typedefs

typedef struct *vrna_mx_mfe_s* **vrna_mx_mfe_t**

Typename for the Minimum Free Energy (MFE) DP matrices data structure *vrna_mx_mfe_s*.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

typedef struct *vrna_mx_pf_s* **vrna_mx_pf_t**

Typename for the Partition Function (PF) DP matrices data structure *vrna_mx_pf_s*.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Enums

enum **vrna_mx_type_e**

An enumerator that is used to specify the type of a polymorphic Dynamic Programming (DP) matrix data structure.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

See also

vrna_mx_mfe_t, *vrna_mx_pf_t*

Values:

enumerator **VRNA_MX_DEFAULT**

Default DP matrices.

enumerator **VRNA_MX_WINDOW**

DP matrices suitable for local structure prediction using window approach.

See also

vrna_mfe_window(), vrna_mfe_window_zscore(), *pfl_fold()*

enumerator **VRNA_MX_2DFOLD**

DP matrices suitable for distance class partitioned structure prediction.

See also

vrna_mfe_TwoD(), *vrna_pf_TwoD()*

Functions

int **vrna_mx_add**(*vrna_fold_compound_t* *fc, *vrna_mx_type_e* type, unsigned int options)

Add Dynamic Programming (DP) matrices (allocate memory).

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

This function adds DP matrices of a specific type to the provided *vrna_fold_compound_t*, such that successive DP recursion can be applied. The function caller has to specify which type of DP matrix is requested, see *vrna_mx_type_e*, and what kind of recursive algorithm will be applied later on,

using the parameters type, and options, respectively. For the latter, Minimum free energy (MFE), and Partition function (PF) computations are distinguished. A third option that may be passed is *VRNA_OPTION_HYBRID*, indicating that auxiliary DP arrays are required for RNA-RNA interaction prediction.

See also

vrna_mx_mfe_add(), vrna_mx_pf_add(), *vrna_fold_compound()*, *vrna_fold_compound_comparative()*, *vrna_fold_compound_free()*, *vrna_mx_pf_free()*, *vrna_mx_mfe_free()*, *vrna_mx_type_e*, *VRNA_OPTION_MFE*, *VRNA_OPTION_PF*, *VRNA_OPTION_HYBRID*, *VRNA_OPTION_EVAL_ONLY*

Note

Usually, there is no need to call this function, since the constructors of *vrna_fold_compound_t* are handling all the DP matrix memory allocation.

Parameters

- **fc** – The *vrna_fold_compound_t* that holds pointers to the DP matrices
- **type** – The type of DP matrices requested
- **options** – Option flags that specify the kind of DP matrices, such as MFE or PF arrays, and auxiliary requirements

Returns

1 if DP matrices were properly allocated and attached, 0 otherwise

int **vrna_mx_mfe_add**(*vrna_fold_compound_t* *fc, *vrna_mx_type_e* mx_type, unsigned int options)

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **vrna_mx_pf_add**(*vrna_fold_compound_t* *fc, *vrna_mx_type_e* mx_type, unsigned int options)

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **vrna_mx_prepare**(*vrna_fold_compound_t* *fc, unsigned int options)

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

void **vrna_mx_mfe_free**(*vrna_fold_compound_t* *fc)

Free memory occupied by the Minimum Free Energy (MFE) Dynamic Programming (DP) matrices.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

See also

vrna_fold_compound(), *vrna_fold_compound_comparative()*, *vrna_fold_compound_free()*, *vrna_mx_pf_free()*

Parameters

- **fc** – The *vrna_fold_compound_t* storing the MFE DP matrices that are to be erased from memory

void **vrna_mx_pf_free**(*vrna_fold_compound_t* *fc)

Free memory occupied by the Partition Function (PF) Dynamic Programming (DP) matrices.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

See also

vrna_fold_compound(), *vrna_fold_compound_comparative()*, *vrna_fold_compound_free()*, *vrna_mx_mfe_free()*

Parameters

- **fc** – The *vrna_fold_compound_t* storing the PF DP matrices that are to be erased from memory

struct **vrna_mx_mfe_s**

#include <ViennaRNA/datastructures/dp_matrices.h> Minimum Free Energy (MFE) Dynamic Programming (DP) matrices data structure required within the *vrna_fold_compound_t*.

Common fields for MFE matrices

const *vrna_mx_type_e* **type**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Type of the DP matrices

unsigned int **length**

Length of the sequence, therefore an indicator of the size of the DP matrices.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

unsigned int **strands**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Number of strands

Public Members

union *vrna_mx_mfe_s*.[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[union].__unnamed0__

Public Members

struct *vrna_mx_mfe_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct *vrna_mx_mfe_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct *vrna_mx_mfe_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed0__

Default DP matrices

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_DEFAULT
```

int ***c**

Energy array, given that i-j pair.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***f5**

Energy of 5' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***f3**

Energy of 3' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****fms5**

Energy for connected interstrand configurations.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****fms3**

nergy for connected interstrand configurations

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fML**

Multi-loop auxiliary energy array.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM1**

Second ML array, only for unique multibrnach loop decomposition.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM2**

Energy for a multibranch loop region with exactly two stems, extending to 3' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM1_new**

ML array with exactly one component within 5'end and i, ending at i (for circRNA).

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM2_real**

Energy for a multibranch loop region with at least two stems.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_int_t ***c_gq**

Energies of g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **Fc**

Minimum Free Energy of entire circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcH**

Minimum Free Energy of hairpin loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcI**

Minimum Free Energy of internal loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcM**

Minimum Free Energy of multibranch loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed1__

Local Folding DP matrices using window approach

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_WINDOW
```

int ****c_local**

Energy array, given that i-j pair.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***f3_local**

Energy of 5' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****fML_local**

Multi-loop auxiliary energy array.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****ggg_local**

Energies of g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

unsigned int **ggg_local_shift**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed2__

Distance Class DP matrices

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_2DFOLD
```

int ***E_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_min_FcM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_FcM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_F5_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_F3_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_C_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M1_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M2_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_Fc_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcH_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcI_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcM_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_mx_mfe_s.[union].__unnamed0__

Public Members

struct *vrna_mx_mfe_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct *vrna_mx_mfe_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct *vrna_mx_mfe_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed0__

Default DP matrices

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_DEFAULT
```

int ***c**

Energy array, given that i-j pair.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***f5**

Energy of 5' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***f3**

Energy of 3' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****fms5**

Energy for connected interstrand configurations.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****fms3**

nergy for connected interstrand configurations

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fML**

Multi-loop auxiliary energy array.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM1**

Second ML array, only for unique multibrnach loop decomposition.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM2**

Energy for a multibranch loop region with exactly two stems, extending to 3' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM1_new**

ML array with exactly one component within 5'end and i, ending at i (for circRNA).

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM2_real**

Energy for a multibranch loop region with at least two stems.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_int_t ***c_gq**

Energies of g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **Fc**

Minimum Free Energy of entire circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcH**

Minimum Free Energy of hairpin loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcI**

Minimum Free Energy of internal loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcM**

Minimum Free Energy of multibranch loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed1__

Local Folding DP matrices using window approach

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_WINDOW
```

int ****c_local**

Energy array, given that i-j pair.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***f3_local**

Energy of 5' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****fML_local**

Multi-loop auxiliary energy array.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****ggg_local**

Energies of g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

unsigned int **ggg_local_shift**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed2__

Distance Class DP matrices

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_2DFOLD
```

int ***E_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_F3

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_C

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_Fc

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_FcH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_min_FcM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_FcM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_F5_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_F3_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_C_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M1_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M2_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_Fc_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcH_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcI_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcM_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_mx_mfe_s.[struct].__unnamed0__.__unnamed0__

Default DP matrices

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_DEFAULT
```

int ***c**

Energy array, given that i-j pair.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***f5**

Energy of 5' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***f3**

Energy of 3' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****fms5**

Energy for connected interstrand configurations.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****fms3**

nergy for connected interstrand configurations

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fML**

Multi-loop auxiliary energy array.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM1**

Second ML array, only for unique multibrnach loop decomposition.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM2**

Energy for a multibranch loop region with exactly two stems, extending to 3' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM1_new**

ML array with exactly one component within 5'end and i, ending at i (for circRNA).

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***fM2_real**

Energy for a multibranch loop region with at least two stems.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_int_t ***c_gq**

Energies of g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **Fc**

Minimum Free Energy of entire circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcH**

Minimum Free Energy of hairpin loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcI**

Minimum Free Energy of internal loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **FcM**

Minimum Free Energy of multibranch loop cases in circular RNA.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_mx_mfe_s.[struct].__unnamed0__.__unnamed1__

Local Folding DP matrices using window approach

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_WINDOW
```

int ****c_local**

Energy array, given that i-j pair.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *f3_local

Energy of 5' end.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **fML_local

Multi-loop auxiliary energy array.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **ggg_local

Energies of g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

unsigned int ggg_local_shift

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_mx_mfe_s.[struct].__unnamed0__.__unnamed2__

Distance Class DP matrices

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_2DFOLD
```

int ***E_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_F5

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_F5**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****E_F3**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_F3**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_F3**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_F3**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_F3**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****E_C**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_C**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_C**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_C**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_C**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****E_M**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_M2**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_M2**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****E_Fc**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_Fc**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_Fc**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_min_Fc**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_Fc**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****E_FcH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_FcH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_FcH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_min_FcH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_FcH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_FcI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_FcM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *E_F5_rem

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *E_F3_rem

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_C_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M1_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***E_M2_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_Fc_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcH_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcI_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **E_FcM_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct **vrna_mx_pf_s**

#include <ViennaRNA/datastructures/dp_matrices.h> Partition function (PF) Dynamic Programming (DP) matrices data structure required within the *vrna_fold_compound_t*.

Common fields for DP matrices

const *vrna_mx_type_e* **type**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Type of the DP matrices

unsigned int **length**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Size of the DP matrices (i.e. sequence length)

FLT_OR_DBL ***scale**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Boltzmann factor scaling

FLT_OR_DBL ***expMLbase**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Boltzmann factors for unpaired bases in multibranch loop

Public Members

union *vrna_mx_pf_s*.[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[union].__unnamed0__

Public Members

struct *vrna_mx_pf_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct *vrna_mx_pf_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct *vrna_mx_pf_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed0__

Default PF matrices

Note

These data fields are available if

```
vrna_mx_pf_t.type == VRNA_MX_DEFAULT
```

FLT_OR_DBL ***q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qb**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm1**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***probs**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***q1k**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qln**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_FLT_OR_DBL_t ***q_gq**

Boltzmann factors for g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_FLT_OR_DBL_t ***p_gq**

probabilities for g-quadruplexes

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qo**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm2**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm2_real**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm1_new**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qho**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qio**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qmo**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed1__

Local Folding DP matrices using window approach

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_WINDOW
```

FLT_OR_DBL ****q_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qb_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qm_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****pR**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qm2_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****QI5**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****q2l**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qmb**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****G_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed2__

Distance Class DP matrices

Note

These data fields are available if

```
vrna_mx_pf_t.type == VRNA_MX_2DFOLD
```

FLT_OR_DBL ***Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****l_min_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****l_max_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_B

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_B

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_B

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_B

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_Q_c**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****Q_cH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_Q_cH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_Q_cH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_min_Q_cH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_Q_cH**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****Q_cI**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_Q_cI**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_Q_cI**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_min_Q_cI**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_Q_cI**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_min_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_B_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M1_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M2_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_c_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cH_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cI_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cM_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_mx_pf_s.[union].__unnamed0__

Public Members

struct *vrna_mx_pf_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct *vrna_mx_pf_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

struct *vrna_mx_pf_s*.[anonymous].[anonymous] **[anonymous]**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed0__

Default PF matrices

Note

These data fields are available if

```
vrna_mx_pf_t.type == VRNA_MX_DEFAULT
```

FLT_OR_DBL ***q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qb**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm1**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***probs**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***q1k**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qln**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_FLT_OR_DBL_t ***q_gq**

Boltzmann factors for g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_FLT_OR_DBL_t ***p_gq**

probabilities for g-quadruplexes

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qo**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm2**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm2_real**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm1_new**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qho**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qio**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qmo**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed1__

Local Folding DP matrices using window approach

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_WINDOW
```

FLT_OR_DBL ****q_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qb_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qm_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****pR**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qm2_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****QI5**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****q2l**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qmb**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****G_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

[struct].__unnamed0__.__unnamed2__

Distance Class DP matrices

Note

These data fields are available if

```
vrna_mx_pf_t.type == VRNA_MX_2DFOLD
```

FLT_OR_DBL ***Q

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_B

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_M

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Q_cM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Q_cM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Q_cM

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_B_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M1_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M2_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_c_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cH_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cI_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cM_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_mx_pf_s.[struct].__unnamed0__.__unnamed0__

Default PF matrices

Note

These data fields are available if

```
vrna_mx_pf_t.type == VRNA_MX_DEFAULT
```

FLT_OR_DBL ***q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qb**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm1**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***probs**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***q1k**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qln**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_FLT_OR_DBL_t ***q_gq**

Boltzmann factors for g-quadruplexes.

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_smx_csr_FLT_OR_DBL_t ***p_gq**

probabilities for g-quadruplexes

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qo**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm2**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm2_real**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***qm1_new**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qho**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qio**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **qmo**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_mx_pf_s.[struct].__unnamed0__.__unnamed1__

Local Folding DP matrices using window approach

Note

These data fields are available if

```
vrna_mx_mfe_t.type == VRNA_MX_WINDOW
```

FLT_OR_DBL ****q_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qb_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qm_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****pR**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qm2_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****QI5**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****q2l**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****qmb**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****G_local**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

vrna_mx_pf_s.[struct].__unnamed0__.__unnamed2__

Distance Class DP matrices

Note

These data fields are available if

```
vrna_mx_pf_t.type == VRNA_MX_2DFOLD
```

FLT_OR_DBL ***Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****l_min_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****l_max_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_Q**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****l_min_Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****l_max_Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_Q_B**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****Q_M**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****l_min_Q_M**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ****l_max_Q_M**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_min_Q_M**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***k_max_Q_M**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_M1

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_min_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **l_max_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_min_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *k_max_Q_M2

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_Q_c

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_max_Q_cH

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_min_Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int *l_max_Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int k_min_Q_cI

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_Q_cI**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ****Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_min_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int ***l_max_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_min_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

int **k_max_Q_cM**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_B_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M1_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL ***Q_M2_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_c_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cH_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cI_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

FLT_OR_DBL **Q_cM_rem**

```
#include <ViennaRNA/datastructures/dp_matrices.h>
```

Hash Tables

Various implementations of hash table functions.

Abstract interface

typedef struct vrna_hash_table_s ***vrna_hash_table_t**

A hash table object.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

See also

vrna_ht_init(), *vrna_ht_free()*

typedef int (***vrna_ht_cmp_f**)(void *x, void *y)

Callback function to compare two hash table entries.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

See also

vrna_ht_init(), *vrna_ht_db_comp()*

Param x
A hash table entry

Param y
A hash table entry

Return
-1 if x is smaller, +1 if x is larger than y. 0 if $x == y$

typedef int **vrna_callback_ht_compare_entries**(void *x, void *y)

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

typedef unsigned int (***vrna_ht_hashfunc_f**)(void *x, unsigned long hashtable_size)

Callback function to generate a hash key, i.e. hash function.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

See also

vrna_ht_init(), *vrna_ht_db_hash_func()*

Param x
A hash table entry

Param hashtable_size
The size of the hash table

Return
The hash table key for entry `x`

typedef unsigned int **vrna_callback_ht_hash_function**(void *x, unsigned long hashtable_size)

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

typedef int (***vrna_ht_free_f**)(void *x)

Callback function to free a hash table entry.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

See also

vrna_ht_init(), *vrna_ht_db_free_entry()*

Param x
A hash table entry

Return
0 on success

typedef int **vrna_callback_ht_free_entry**(void *x)

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

vrna_hash_table_t **vrna_ht_init**(unsigned int b, *vrna_ht_cmp_f* compare_function, *vrna_ht_hashfunc_f* hash_function, *vrna_ht_free_f* free_hash_entry)

Get an initialized hash table.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

This function returns a ready-to-use hash table with pre-allocated memory for a particular number of entries.

Note

If all function pointers are NULL, this function initializes the hash table with *default functions*, i.e.

- *vrna_ht_db_comp()* for the compare_function,
- *vrna_ht_db_hash_func()* for the hash_function, and
- *vrna_ht_db_free_entry()* for the free_hash_entry

arguments.

Warning

If hash_bits is larger than 27 you have to compile it with the flag gcc -mcmodel=large.

Parameters

- **b** – Number of bits for the hash table. This determines the size ($2^b - 1$).
- **compare_function** – A function pointer to compare any two entries in the hash table (may be NULL)
- **hash_function** – A function pointer to retrieve the hash value of any entry (may be NULL)
- **free_hash_entry** – A function pointer to free the memory occupied by any entry (may be NULL)

Returns

An initialized, empty hash table, or NULL on any error

unsigned long **vrna_ht_size**(*vrna_hash_table_t* ht)

Get the size of the hash table.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

Parameters

- **ht** – The hash table

Returns

The size of the hash table, i.e. the maximum number of entries

unsigned long **vrna_ht_collisions**(struct vrna_hash_table_s *ht)

Get the number of collisions in the hash table.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

Parameters

- **ht** – The hash table

Returns

The number of collisions in the hash table

void ***vrna_ht_get**(*vrna_hash_table_t* ht, void *x)

Get an element from the hash table.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

This function takes an object `x` and performs a look-up whether the object is stored within the hash table `ht`. If the object is already stored in `ht`, the function simply returns the entry, otherwise it returns `NULL`.

See also

vrna_ht_insert(), vrna_hash_delete(), *vrna_ht_init()*

Parameters

- **ht** – The hash table
- **x** – The hash entry to look-up

Returns

The entry `x` if it is stored in `ht`, `NULL` otherwise

int **vrna_ht_insert**(*vrna_hash_table_t* ht, void *x)

Insert an object into a hash table.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

Writes the pointer to your hash entry into the table.

See also

vrna_ht_init(), vrna_hash_delete(), *vrna_ht_clear()*

Warning

In case of collisions, this function simply increments the hash key until a free entry in the hash table is found.

Parameters

- **ht** – The hash table
- **x** – The hash entry

Returns

0 on success, 1 if the value is already in the hash table, -1 on error.

void **vrna_ht_remove**(*vrna_hash_table_t* ht, void *x)

Remove an object from the hash table.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

Deletes the pointer to your hash entry from the table.

Note

This function doesn't free any memory occupied by the hash entry.

Parameters

- **ht** – The hash table
- **x** – The hash entry

void **vrna_ht_clear**(*vrna_hash_table_t* ht)

Clear the hash table.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

This function removes all entries from the hash table and automatically free's the memory occupied by each entry using the bound *vrna_ht_free_f()* function.

See also

vrna_ht_free(), *vrna_ht_init()*

Parameters

- **ht** – The hash table

void **vrna_ht_free**(*vrna_hash_table_t* ht)

Free all memory occupied by the hash table.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

This function removes all entries from the hash table by calling the *vrna_ht_free_f()* function for each entry. Finally, the memory occupied by the hash table itself is free'd as well.

Parameters

- **ht** – The hash table

Dot-Bracket / Free Energy entries

int **vrna_ht_db_comp**(void *x, void *y)

Default hash table entry comparison.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

This is the default comparison function for hash table entries. It assumes the both entries `x` and `y` are of type *vrna_ht_entry_db_t* and compares the `structure` attribute of both entries

See also

vrna_ht_entry_db_t, *vrna_ht_init()*, *vrna_ht_db_hash_func()*, *vrna_ht_db_free_entry()*

Parameters

- **x** – A hash table entry of type *vrna_ht_entry_db_t*

- **y** – A hash table entry of type *vrna_ht_entry_db_t*

Returns
-1 if x is smaller, +1 if x is larger than y. 0 if both are equal.

unsigned int **vrna_ht_db_hash_func**(void *x, unsigned long hashtable_size)

Default hash function.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

This is the default hash function for hash table insertion/lookup. It assumes that entries are of type *vrna_ht_entry_db_t* and uses the Bob Jenkins 1996 mix function to create a hash key from the `structure` attribute of the hash entry.

See also

vrna_ht_entry_db_t, *vrna_ht_init()*, *vrna_ht_db_comp()*, *vrna_ht_db_free_entry()*

Parameters

- **x** – A hash table entry to compute the key for
- **hashtable_size** – The size of the hash table

Returns
The hash key for entry `x`

int **vrna_ht_db_free_entry**(void *hash_entry)

Default function to free memory occupied by a hash entry.

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

This function assumes that hash entries are of type *vrna_ht_entry_db_t* and free's the memory occupied by that entry.

See also

vrna_ht_entry_db_t, *vrna_ht_init()*, *vrna_ht_db_comp()*, *vrna_ht_db_hash_func()*

Parameters

- **hash_entry** – The hash entry to remove from memory

Returns
0 on success

struct **vrna_ht_entry_db_t**

#include <ViennaRNA/datastructures/hash_tables.h> Default hash table entry.

See also

vrna_ht_init(), *vrna_ht_db_comp()*, *vrna_ht_db_hash_func()*, *vrna_ht_db_free_entry()*

Public Members

char *structure

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

A secondary structure in dot-bracket notation

float energy

```
#include <ViennaRNA/datastructures/hash_tables.h>
```

The free energy of `structure`

Heaps

Interface for an abstract implementation of a heap data structure.

Typedefs

typedef struct vrna_heap_s *vrna_heap_t

An abstract heap data structure.

```
#include <ViennaRNA/datastructures/heap.h>
```

See also

vrna_heap_init(), *vrna_heap_free()*, *vrna_heap_insert()*, *vrna_heap_pop()*, *vrna_heap_top()*, *vrna_heap_remove()*, *vrna_heap_update()*

typedef int (*vrna_heap_cmp_f)(const void *a, const void *b, void *data)

Heap compare function prototype.

```
#include <ViennaRNA/datastructures/heap.h>
```

Use this prototype to design the compare function for the heap implementation. The arbitrary data pointer `data` may be used to get access to further information required to actually compare the two values `a` and `b`.

Note

The heap implementation acts as a *min-heap*, therefore, the minimum element will be present at the heap's root. In case a *max-heap* is required, simply reverse the logic of this compare function.

Param a
The first object to compare

Param b
The second object to compare

Param data
An arbitrary data pointer passed through from the heap implementation

Return
A value less than zero if `a < b`, a value greater than zero if `a > b`, and 0 otherwise

typedef int **vrna_callback_heap_cmp**(const void *a, const void *b, void *data)

```
#include <ViennaRNA/datastructures/heap.h>
```

typedef size_t (***vrna_heap_get_pos_f**)(const void *a, void *data)

Retrieve the position of a particular heap entry within the heap.

```
#include <ViennaRNA/datastructures/heap.h>
```

Param a
The object to look-up within the heap

Param data
An arbitrary data pointer passed through from the heap implementation

Return
The position of the element `a` within the heap, or 0 if it is not in the heap

typedef size_t **vrna_callback_heap_get_pos**(const void *a, void *data)

```
#include <ViennaRNA/datastructures/heap.h>
```

typedef void (***vrna_heap_set_pos_f**)(const void *a, size_t pos, void *data)

Store the position of a particular heap entry within the heap.

```
#include <ViennaRNA/datastructures/heap.h>
```

Param a
The object whose position shall be stored

Param pos
The current position of `a` within the heap, or 0 if a was deleted

Param data
An arbitrary data pointer passed through from the heap implementation

typedef void **vrna_callback_heap_set_pos**(const void *a, size_t pos, void *data)

```
#include <ViennaRNA/datastructures/heap.h>
```

Functions

vrna_heap_t **vrna_heap_init**(size_t n, *vrna_heap_cmp_f* cmp, *vrna_heap_get_pos_f* get_entry_pos, *vrna_heap_set_pos_f* set_entry_pos, void *data)

Initialize a heap data structure.

```
#include <ViennaRNA/datastructures/heap.h>
```

This function initializes a heap data structure. The implementation is based on a *min-heap*, i.e. the minimal element is located at the root of the heap. However, by reversing the logic of the compare function, one can easily transform this into a *max-heap* implementation.

Beside the regular operations on a heap data structure, we implement removal and update of arbitrary elements within the heap. For that purpose, however, one requires a reverse-index lookup system that, (i) for a given element stores the current position in the heap, and (ii) allows for fast lookup of an elements current position within the heap. The corresponding getter- and setter- functions may be provided through the arguments `get_entry_pos` and `set_entry_pos`, respectively.

Sometimes, it is difficult to simply compare two data structures without any context. Therefore, the compare function is provided with a user-defined data pointer that can hold any context required.

See also

vrna_heap_free(), *vrna_heap_insert()*, *vrna_heap_pop()*, *vrna_heap_top()*, *vrna_heap_remove()*, *vrna_heap_update()*, *vrna_heap_t*, *vrna_heap_cmp_f*, *vrna_heap_get_pos_f*, *vrna_heap_set_pos_f*

Warning

If any of the arguments `get_entry_pos` or `set_entry_pos` is NULL, the operations *vrna_heap_update()* and *vrna_heap_remove()* won't work.

Parameters

- **n** – The initial size of the heap, i.e. the number of elements to store
- **cmp** – The address of a compare function that will be used to fullfill the partial order requirement
- **get_entry_pos** – The address of a function that retrieves the position of an element within the heap (or NULL)
- **set_entry_pos** – The address of a function that stores the position of an element within the heap (or NULL)
- **data** – An arbitrary data pointer passed through to the compare function `cmp`, and the set/get functions `get_entry_pos` / `set_entry_pos`

Returns

An initialized heap data structure, or NULL on error

void **vrna_heap_free**(*vrna_heap_t* h)

Free memory occupied by a heap data structure.

```
#include <ViennaRNA/datastructures/heap.h>
```

See also

vrna_heap_init()

Parameters

- **h** – The heap that should be free'd

size_t **vrna_heap_size**(struct vrna_heap_s *h)

Get the size of a heap data structure, i.e. the number of stored elements.

```
#include <ViennaRNA/datastructures/heap.h>
```

Parameters

- **h** – The heap data structure

Returns
The number of elements currently stored in the heap, or 0 upon any error

void **vrna_heap_insert**(*vrna_heap_t* h, void *v)

Insert an element into the heap.

```
#include <ViennaRNA/datastructures/heap.h>
```

See also

vrna_heap_init(), *vrna_heap_pop()*, *vrna_heap_top()*, *vrna_heap_free()*, *vrna_heap_remove()*, *vrna_heap_update()*

Parameters

- **h** – The heap data structure
- **v** – A pointer to the object that is about to be inserted into the heap

void ***vrna_heap_pop**(*vrna_heap_t* h)

Pop (remove and return) the object at the root of the heap.

```
#include <ViennaRNA/datastructures/heap.h>
```

This function removes the root from the heap and returns it to the caller.

See also

vrna_heap_init(), *vrna_heap_top()*, *vrna_heap_insert()*, *vrna_heap_free()* *vrna_heap_remove()*, *vrna_heap_update()*

Parameters

- **h** – The heap data structure

Returns
The object at the root of the heap, i.e. the minimal element (or NULL if (a) the heap is empty or (b) any error occurred)

const void ***vrna_heap_top**(*vrna_heap_t* h)

Get the object at the root of the heap.

```
#include <ViennaRNA/datastructures/heap.h>
```

See also

vrna_heap_init(), *vrna_heap_pop()*, *vrna_heap_insert()*, *vrna_heap_free()* *vrna_heap_remove()*, *vrna_heap_update()*

Parameters

- **h** – The heap data structure

Returns

The object at the root of the heap, i.e. the minimal element (or NULL if (a) the heap is empty or (b) any error occurred)

void ***vrna_heap_remove**(*vrna_heap_t* h, const void *v)

Remove an arbitrary element within the heap.

```
#include <ViennaRNA/datastructures/heap.h>
```

See also

vrna_heap_init(), *vrna_heap_get_pos_f*, *vrna_heap_set_pos_f*, *vrna_heap_pop()*, *vrna_heap_free()*

Warning

This function won't work if the heap was not properly initialized with callback functions for fast reverse-index mapping!

Parameters

- **h** – The heap data structure
- **v** – The object to remove from the heap

Returns

The object that was removed from the heap (or NULL if (a) it wasn't found or (b) any error occurred)

void ***vrna_heap_update**(*vrna_heap_t* h, void *v)

Update an arbitrary element within the heap.

```
#include <ViennaRNA/datastructures/heap.h>
```

See also

vrna_heap_init(), *vrna_heap_get_pos_f*, *vrna_heap_set_pos_f* *vrna_heap_pop()*, *vrna_heap_remove()*, *vrna_heap_free()*

> **Note**
>
> If the object that is to be updated is not currently stored in the heap, it will be inserted. In this case, the function returns NULL.

> **Warning**
>
> This function won't work if the heap was not properly initialized with callback functions for fast reverse-index mapping!

Parameters

- **h** – The heap data structure
- **v** – The object to update

Returns

The 'previous' object within the heap that now got replaced by v (or NULL if (a) it wasn't found or (b) any error occurred)

Arrays

Interface for an abstract implementation of an array data structure.

Defines

vrna_array(Type)

Define an array.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_make(Type, Name)

Make an array `Name` of type `Type`.

```
#include <ViennaRNA/datastructures/array.h>
```

VRNA_ARRAY_GROW_FORMULA(n)

The default growth formula for array.

```
#include <ViennaRNA/datastructures/array.h>
```

VRNA_ARRAY_HEADER(input)

Retrieve a pointer to the header of an array `input`.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_size(input)

Get the number of elements of an array `input`.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_capacity(input)

Get the size of an array `input`, i.e. its actual capacity.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_set_capacity(a, capacity)

Explicitely set the capacity of an array `a`.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_init_size(a, init_size)

Initialize an array `a` with a particular pre-allocated size `init_size`.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_init(a)

Initialize an array `a`.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_free(a)

Release memory of an array `a`.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_append(a, item)

Safely append an item to an array `a`.

```
#include <ViennaRNA/datastructures/array.h>
```

vrna_array_grow(a, min_capacity)

Grow an array `a` to provide a minimum capacity `min_capacity`.

```
#include <ViennaRNA/datastructures/array.h>
```

VRNA_ARRAY_REGISTER(TYPE)

```
#include <ViennaRNA/datastructures/array.h>
```

Create/register a typedef for a *vrna_array* of a particular type

Note

This macro automatically adds the prefix `vrna_` and the suffix `_array_t` to the `TYPE` argument to build the new typename

VRNA_ARRAY_REGISTER_NAME(TYPE, TYPENAME)

```
#include <ViennaRNA/datastructures/array.h>
```

Create a typedef for a *vrna_array* of a particular type

Use `TYPE` and `TYPENAME` to specify the type and new typename of the array.

Typedefs

typedef struct *vrna_array_header_s* **vrna_array_header_t**

The header of an array.

```
#include <ViennaRNA/datastructures/array.h>
```

Functions

void *__vrna__array_set_capacity__(void *array, size_t capacity, size_t element_size)

Explicitely set the capacity of an array.

```
#include <ViennaRNA/datastructures/array.h>
```

> **Note**
>
> Do not use this function. Rather resort to the *vrna_array_set_capacity* macro

struct **vrna_array_header_s**

#include <ViennaRNA/datastructures/array.h> The header of an array.

Public Members

size_t **num**

The number of elements in an array.

```
#include <ViennaRNA/datastructures/array.h>
```

size_t **size**

The actual capacity of an array.

```
#include <ViennaRNA/datastructures/array.h>
```

Strings

Defines

VRNA_STRING_HEADER(s)

```
#include <ViennaRNA/datastructures/string.h>
```

Typedefs

typedef char ***vrna_string_t**

```
#include <ViennaRNA/datastructures/string.h>
```

typedef struct *vrna_string_header_s* **vrna_string_header_t**

The header of an array.

```
#include <ViennaRNA/datastructures/string.h>
```

Functions

vrna_string_t **vrna_string_make**(char const *str)

```
#include <ViennaRNA/datastructures/string.h>
```

void **vrna_string_free**(*vrna_string_t* str)

```
#include <ViennaRNA/datastructures/string.h>
```

size_t **vrna_string_length**(*vrna_string_t* const str)

```
#include <ViennaRNA/datastructures/string.h>
```

size_t **vrna_string_size**(*vrna_string_t* const str)

```
#include <ViennaRNA/datastructures/string.h>
```

vrna_string_t **vrna_string_append**(*vrna_string_t* str, *vrna_string_t* const other)

```
#include <ViennaRNA/datastructures/string.h>
```

vrna_string_t **vrna_string_append_cstring**(*vrna_string_t* str, char const *other)

```
#include <ViennaRNA/datastructures/string.h>
```

vrna_string_t **vrna_string_make_space_for**(*vrna_string_t* str, size_t add_len)

```
#include <ViennaRNA/datastructures/string.h>
```

size_t **vrna_string_available_space**(*vrna_string_t* const str)

```
#include <ViennaRNA/datastructures/string.h>
```

struct **vrna_string_header_s**

#include <ViennaRNA/datastructures/string.h> The header of an array.

Public Members

size_t **len**

The length of the string.

```
#include <ViennaRNA/datastructures/string.h>
```

size_t **size**

The actual capacity of an array.

```
#include <ViennaRNA/datastructures/string.h>
```

size_t **shift_post**

```
#include <ViennaRNA/datastructures/string.h>
```

char **backup**

```
#include <ViennaRNA/datastructures/string.h>
```

Buffers

Functions that provide dynamically buffered stream-like data structures.

Typedefs

typedef struct vrna_cstr_s ***vrna_cstr_t**

```
#include <ViennaRNA/datastructures/char_stream.h>
```

typedef struct vrna_ordered_stream_s ***vrna_ostream_t**

An ordered output stream structure with unordered insert capabilities.

```
#include <ViennaRNA/datastructures/stream_output.h>
```

typedef void (***vrna_stream_output_f**)(void *auxdata, unsigned int i, void *data)

Ordered stream processing callback.

```
#include <ViennaRNA/datastructures/stream_output.h>
```

This callback will be processed in sequential order as soon as sequential data in the output stream becomes available.

Note

The callback must also release the memory occupied by the data passed since the stream will lose any reference to it after the callback has been executed.

Param auxdata
A shared pointer for all calls, as provided by the second argument to *vrna_ostream_init()*

Param i
The index number of the data passed to `data`

Param data
A block of data ready for processing

typedef void **vrna_callback_stream_output**(void *auxdata, unsigned int i, void *data)

```
#include <ViennaRNA/datastructures/stream_output.h>
```

Functions

vrna_cstr_t **vrna_cstr**(size_t size, FILE *output)

Create a dynamic char * stream data structure.

```
#include <ViennaRNA/datastructures/char_stream.h>
```

See also

vrna_cstr_free(), *vrna_cstr_close()*, *vrna_cstr_fflush()*, *vrna_cstr_discard()*, vrna_cstr_printf()

Parameters

- **size** – The initial size of the buffer in characters
- **output** – An optional output file stream handle that is used to write the collected data to (defaults to *stdout* if *NULL*)

void **vrna_cstr_discard**(struct vrna_cstr_s *buf)

Discard the current content of the dynamic char * stream data structure.

```
#include <ViennaRNA/datastructures/char_stream.h>
```

See also

vrna_cstr_free(), *vrna_cstr_close()*, *vrna_cstr_fflush()*, vrna_cstr_printf()

Parameters

- **buf** – The dynamic char * stream data structure to free

void **vrna_cstr_free**(*vrna_cstr_t* buf)

Free the memory occupied by a dynamic char * stream data structure.

```
#include <ViennaRNA/datastructures/char_stream.h>
```

This function first flushes any remaining character data within the stream and then free's the memory occupied by the data structure.

See also

vrna_cstr_close(), *vrna_cstr_fflush()*, *vrna_cstr()*

Parameters

- **buf** – The dynamic char * stream data structure to free

void **vrna_cstr_close**(*vrna_cstr_t* buf)

Free the memory occupied by a dynamic char * stream and close the output stream.

```
#include <ViennaRNA/datastructures/char_stream.h>
```

This function first flushes any remaining character data within the stream then closes the attached output file stream (if any), and finally free's the memory occupied by the data structure.

See also

vrna_cstr_free(), *vrna_cstr_fflush()*, *vrna_cstr()*

Parameters

- **buf** – The dynamic char * stream data structure to free

void **vrna_cstr_fflush**(struct vrna_cstr_s *buf)

Flush the dynamic char * output stream.

```
#include <ViennaRNA/datastructures/char_stream.h>
```

This function flushes the collected char * stream, either by writing to the attached file handle, or simply by writing to *stdout* if no file handle has been attached upon construction using *vrna_cstr()*.

See also

vrna_cstr(), *vrna_cstr_close()*, *vrna_cstr_free()*

Parameters

- **buf** – The dynamic char * stream data structure to flush

Post

The stream buffer is empty after execution of this function

const char ***vrna_cstr_string**(*vrna_cstr_t* buf)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

int **vrna_cstr_vprintf**(*vrna_cstr_t* buf, const char *format, va_list args)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

int **vrna_cstr_printf**(*vrna_cstr_t* buf, const char *format, ...)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_message_info**(*vrna_cstr_t* buf, const char *format, ...)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_message_vinfo**(*vrna_cstr_t* buf, const char *format, va_list args)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_message_warning**(struct vrna_cstr_s *buf, const char *format, ...)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_message_vwarning**(struct vrna_cstr_s *buf, const char *format, va_list args)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_fasta_header**(*vrna_cstr_t* buf, const char *head)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_printf_structure**(struct vrna_cstr_s *buf, const char *structure, const char *format, ...)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_vprintf_structure**(struct vrna_cstr_s *buf, const char *structure, const char *format, va_list args)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_printf_comment**(struct vrna_cstr_s *buf, const char *format, ...)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_vprintf_comment**(struct vrna_cstr_s *buf, const char *format, va_list args)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_printf_thead**(struct vrna_cstr_s *buf, const char *format, ...)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_vprintf_thead**(struct vrna_cstr_s *buf, const char *format, va_list args)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_printf_tbody**(struct vrna_cstr_s *buf, const char *format, ...)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_vprintf_tbody**(struct vrna_cstr_s *buf, const char *format, va_list args)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_sd_corr**(struct vrna_cstr_s *buf)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_ext_loop**(struct vrna_cstr_s *buf, int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_ext_loop_revert**(struct vrna_cstr_s *buf, int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_hp_loop**(struct vrna_cstr_s *buf, int i, int j, char si, char sj, int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_hp_loop_revert**(struct vrna_cstr_s *buf, int i, int j, char si, char sj, int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_int_loop**(struct vrna_cstr_s *buf, int i, int j, char si, char sj, int k, int l, char sk, char sl, int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_int_loop_revert**(struct vrna_cstr_s *buf, int i, int j, char si, char sj, int k, int l, char sk, char sl, int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_mb_loop**(struct vrna_cstr_s *buf, int i, int j, char si, char sj, int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_mb_loop_revert**(struct vrna_cstr_s *buf, int i, int j, char si, char sj, int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

void **vrna_cstr_print_eval_gquad**(struct vrna_cstr_s *buf, unsigned int i, unsigned int j, unsigned int L, unsigned int l[3], int energy)

```
#include <ViennaRNA/datastructures/char_stream.h>
```

vrna_ostream_t **vrna_ostream_init**(*vrna_stream_output_f* output, void *auxdata)

Get an initialized ordered output stream.

```
#include <ViennaRNA/datastructures/stream_output.h>
```

See also

vrna_ostream_free(), *vrna_ostream_request()*, *vrna_ostream_provide()*

Parameters

- **output** – A callback function that processes and releases data in the stream
- **auxdata** – A pointer to auxiliary data passed as first argument to the `output` callback

Returns

An initialized ordered output stream

void **vrna_ostream_free**(*vrna_ostream_t* dat)

Free an initialized ordered output stream.

```
#include <ViennaRNA/datastructures/stream_output.h>
```

See also

vrna_ostream_init()

Parameters

- **dat** – The output stream for which occupied memory should be free'd

int **vrna_ostream_threadsafe**(void)

```
#include <ViennaRNA/datastructures/stream_output.h>
```

void **vrna_ostream_request**(*vrna_ostream_t* dat, unsigned int num)

Request index in ordered output stream.

```
#include <ViennaRNA/datastructures/stream_output.h>
```

This function must be called prior to *vrna_ostream_provide()* to indicate that data associted with a certain index number is expected to be inserted into the stream in the future.

See also

vrna_ostream_init(), *vrna_ostream_provide()*, *vrna_ostream_free()*

Parameters

- **dat** – The output stream for which the index is requested
- **num** – The index to request data for

void **vrna_ostream_provide**(*vrna_ostream_t* dat, unsigned int i, void *data)

Provide output stream data for a particular index.

```
#include <ViennaRNA/datastructures/stream_output.h>
```

See also

vrna_ostream_request()

Parameters

- **dat** – The output stream for which data is provided
- **i** – The index of the provided data
- **data** – The data provided

Pre

The index data is provided for must have been requested using *vrna_ostream_request()* beforehand.

Common Data Structures

Typedefs

typedef struct *vrna_elem_prob_s* **vrna_plist_t**

Typename for the base pair list repesenting data structure *vrna_elem_prob_s*.

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct vrna_cpair_s **vrna_cpair_t**

Typename for data structure vrna_cpair_s.

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct vrna_data_linear_s **vrna_data_lin_t**

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct vrna_color_s **vrna_color_t**

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef double **FLT_OR_DBL**

Typename for floating point number in partition function computations.

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct *vrna_basepair_s* **PAIR**

Old typename of *vrna_basepair_s*.

```
#include <ViennaRNA/datastructures/basic.h>
```

Deprecated:

Use *vrna_basepair_t* instead!

typedef struct *vrna_elem_prob_s* **plist**

Old typename of *vrna_elem_prob_s*.

```
#include <ViennaRNA/datastructures/basic.h>
```

Deprecated:

Use *vrna_ep_t* or *vrna_elem_prob_s* instead!

typedef struct vrna_cpair_s **cpair**

Old typename of vrna_cpair_s.

```
#include <ViennaRNA/datastructures/basic.h>
```

Deprecated:

Use *vrna_cpair_t* instead!

typedef struct *vrna_sect_s* **sect**

Old typename of *vrna_sect_s*.

```
#include <ViennaRNA/datastructures/basic.h>
```

Deprecated:

Use *vrna_sect_t* instead!

typedef *vrna_bp_stack_t* **bondT**

Old typename of #vrna_bp_stack_s.

```
#include <ViennaRNA/datastructures/basic.h>
```

Deprecated:

Use *vrna_bp_stack_t* instead!

typedef struct *pu_contrib* **pu_contrib**

contributions to p_u

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct *interact* **interact**

interaction data structure for RNAup

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct *pu_out* **pu_out**

Collection of all free_energy of beeing unpaired values for output.

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct *constrain* **constrain**

constraints for cofolding

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct *node* **folden**

Data structure for RNAsnoop (fold energy list).

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct *dupVar* **dupVar**

Data structure used in RNApkplex.

```
#include <ViennaRNA/datastructures/basic.h>
```

Functions

void **vrna_C11_features**(void)

Dummy symbol to check whether the library was build using C11/C++11 features.

```
#include <ViennaRNA/datastructures/basic.h>
```

By default, several data structures of our new v3.0 API use C11/C++11 features, such as unnamed unions, unnamed structs. However, these features can be deactivated at compile time to allow building the library and executables with compilers that do not support these features.

Now, the problem arises that once our static library is compiled and a third-party application is supposed to link against it, it needs to know, at compile time, how to correctly address particular data structures. This is usually implicitely taken care of through the API exposed in our header files. Unfortunately, we had some preprocessor directives in our header files that changed the API depending on the capabilities of the compiler the third-party application is build with. This in turn prohibited the use of an RNAlib compiled without C11/C++11 support in a program that compiles/links with enabled C11/C++11 support and vice-versa.

Therefore, we introduce this dummy symbol which can be used to check, whether the static library was build with C11/C++11 features.

Since
v2.2.9

Note

If the symbol is present, the library was build with enabled C11/C++11 features support and no action is required. However, if the symbol is missing in RNAlib >= 2.2.9, programs that link to RNAlib must define a pre-processor identifier *VRNA_DISABLE_C11_FEATURES* before including any ViennaRNA Package header file, for instance by adding a *CPPFLAG*

```
CPPFLAGS+=-DVRNA_DISABLE_C11_FEATURES
```

struct **vrna_basepair_t**

#include <ViennaRNA/datastructures/basic.h> Typename for base pair element.

Deprecated:
Use *vrna_bp_t* instead!

Public Members

int **i**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **j**

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **vrna_bp_stack_t**

#include <*ViennaRNA/datastructures/basic.h*> Typename for the base pair stack element.

Public Members

unsigned int **i**

```
#include <ViennaRNA/datastructures/basic.h>
```

unsigned int **j**

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **pu_contrib**

#include <*ViennaRNA/datastructures/basic.h*> contributions to p_u

Public Members

double ****H**

hairpin loops

```
#include <ViennaRNA/datastructures/basic.h>
```

double ****I**

internal loops

```
#include <ViennaRNA/datastructures/basic.h>
```

double ****M**

multi loops

```
#include <ViennaRNA/datastructures/basic.h>
```

double ****E**

exterior loop

```
#include <ViennaRNA/datastructures/basic.h>
```

int **length**

length of the input sequence

```
#include <ViennaRNA/datastructures/basic.h>
```

int **w**

longest unpaired region

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **interact**

#include <*ViennaRNA/datastructures/basic.h*> interaction data structure for RNAup

Public Members

double ***Pi**

probabilities of interaction

```
#include <ViennaRNA/datastructures/basic.h>
```

double ***Gi**

free energies of interaction

```
#include <ViennaRNA/datastructures/basic.h>
```

double **Gikjl**

full free energy for interaction between [k,i] k<i in longer seq and [j,l] j<l in shorter seq

```
#include <ViennaRNA/datastructures/basic.h>
```

double **Gikjl_wo**

Gikjl without contributions for prob_unpaired.

```
#include <ViennaRNA/datastructures/basic.h>
```

int **i**

k<i in longer seq

```
#include <ViennaRNA/datastructures/basic.h>
```

int **k**

k<i in longer seq

```
#include <ViennaRNA/datastructures/basic.h>
```

int **j**

j<l in shorter seq

```
#include <ViennaRNA/datastructures/basic.h>
```

int **l**

j<l in shorter seq

```
#include <ViennaRNA/datastructures/basic.h>
```

int **length**

length of longer sequence

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **pu_out**

#include <ViennaRNA/datastructures/basic.h> Collection of all free_energy of beeing unpaired values for output.

Public Members

int **len**

sequence length

```
#include <ViennaRNA/datastructures/basic.h>
```

int **u_vals**

number of different -u values

```
#include <ViennaRNA/datastructures/basic.h>
```

int **contribs**

[-c "SHIME"]

```
#include <ViennaRNA/datastructures/basic.h>
```

char **header

header line

```
#include <ViennaRNA/datastructures/basic.h>
```

double **u_values

(the -u values * [-c "SHIME"]) * seq len

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **constrain**

#include <ViennaRNA/datastructures/basic.h> constraints for cofolding

Public Members

int *indx

```
#include <ViennaRNA/datastructures/basic.h>
```

char *ptype

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **duplexT**

#include <ViennaRNA/datastructures/basic.h> Data structure for RNAduplex.

Public Members

int **i**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **j**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **end**

```
#include <ViennaRNA/datastructures/basic.h>
```

char ***structure**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **energy**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **energy_backtrack**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **opening_backtrack_x**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **opening_backtrack_y**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **offset**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **dG1**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **dG2**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **ddG**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **tb**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **te**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **qb**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **qe**

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **node**

#include <ViennaRNA/datastructures/basic.h> Data structure for RNAsnoop (fold energy list).

Public Members

int **k**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **energy**

```
#include <ViennaRNA/datastructures/basic.h>
```

struct *node* ***next**

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **snoopT**

#include <ViennaRNA/datastructures/basic.h> Data structure for RNAsnoop.

Public Members

int **i**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **j**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **u**

```
#include <ViennaRNA/datastructures/basic.h>
```

char ***structure**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **energy**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **Duplex_El**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **Duplex_Er**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **Loop_E**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **Loop_D**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **pscd**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **psct**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **pscg**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **Duplex_Ol**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **Duplex_Or**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **Duplex_Ot**

```
#include <ViennaRNA/datastructures/basic.h>
```

float **fullStemEnergy**

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **dupVar**

#include <ViennaRNA/datastructures/basic.h> Data structure used in RNApkplex.

Public Members

int **i**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **j**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **end**

```
#include <ViennaRNA/datastructures/basic.h>
```

char ***pk_helix**

```
#include <ViennaRNA/datastructures/basic.h>
```

char ***structure**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **energy**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **offset**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **dG1**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **dG2**

```
#include <ViennaRNA/datastructures/basic.h>
```

double **ddG**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **tb**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **te**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **qb**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **qe**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **inactive**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **processed**

```
#include <ViennaRNA/datastructures/basic.h>
```

Backtracking Related Data Structures

Typedefs

typedef struct vrna_bt_stack_s ***vrna_bts_t**

The backtrack stack data structure.

```
#include <ViennaRNA/datastructures/basic.h>
```

See also

vrna_bts_init(), *vrna_bts_free()*, *vrna_bts_push()*, *vrna_bts_top()*, *vrna_bts_pop()*

typedef struct vrna_bp_stack_s ***vrna_bps_t**

The basepair stack data structure.

```
#include <ViennaRNA/datastructures/basic.h>
```

See also

vrna_bps_init(), *vrna_bps_free()*, *vrna_bps_push()*, *vrna_bps_top()*, *vrna_bps_pop()*, *vrna_bps_at()*

typedef struct *vrna_basepair_s* **vrna_bp_t**

Typename for the base pair repesenting data structure *vrna_basepair_s*.

```
#include <ViennaRNA/datastructures/basic.h>
```

typedef struct *vrna_sect_s* **vrna_sect_t**

Typename for stack of partial structures *vrna_sect_s*.

```
#include <ViennaRNA/datastructures/basic.h>
```

Functions

vrna_bts_t **vrna_bts_init**(size_t n)

Get an initialized backtrack stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

This function yields an initialized backtracking stack that holds all elements that need to be further evaluated. The individual elements stored in the stack are of type *vrna_sect_t* and store the sequence delimiters and corresponding backtrack DP matrix flag.

See also

vrna_bts_t, *vrna_bts_free()*, *vrna_bts_push()*, *vrna_bts_top()*,*vrna_bts_pop()*, *vrna_bts_size()*,

Note

Memory for the stack must be released via the *vrna_bts_free()* function.

Parameters

- **n** – The initial size of the backtrack stack

Returns

An initialized backtrack stack, ready for usage in backtracking functions

void **vrna_bts_free**(*vrna_bts_t* bts)

Release memory occupied by a backtrack stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Parameters

- **bts** – The backtrack stack that should be free'd

size_t **vrna_bts_push**(*vrna_bts_t* bts, *vrna_sect_t* element)

Push a new interval onto the backtrack stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

This function pushes a new sequence interval for backtracking onto the backstracking stack `bts`.

Parameters

- **bts** – The backtrack stack
- **element** – The sequence interval and corresponding DP matrix flag

Returns

The size of the backtrack stack after pushing the new interval

vrna_sect_t **vrna_bts_top**(*vrna_bts_t* bts)

Retrieve the top element of the backtrack stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Retrieves the last element put onto the stack, or a zero'd out *vrna_sect_t* structure. The latter is returned on error or when topping an empty stack.

Parameters

- **bts** – The backtrack stack

Returns

The top element of the backtrack stack, or a zero'd out *vrna_sect_t*

vrna_sect_t **vrna_bts_pop**(*vrna_bts_t* bts)

Pop last element of from backtrack stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Retrieves and removes the last element put onto the stack, or a zero'd out *vrna_sect_t* structure. The latter is returned on error or when topping an empty stack.

Parameters

- **bts** – The backtrack stack

Returns

The top element of the backtrack stack, or a zero'd out *vrna_sect_t*

size_t **vrna_bts_size**(*vrna_bts_t* bts)

Get the size of the backtrack stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Parameters

- **bts** – The backtrack stack

Returns

The size of the backtracking stack

vrna_bps_t **vrna_bps_init**(size_t n)

Get an initialized base pair stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Base pair stacks are used in the backtracking procedure to store all base pairs and structural elements that have been identified so far. Thos function returns an initialized backtracking stack with initial size n. Individual elements stored in this stack are of type *vrna_bp_t*.

Note

Memory for the stack must be released via the *vrna_bps_free()* function.

Parameters

- **n** – The initial size of the base pair stack

Returns

An initialized base pair stack

void **vrna_bps_free**(*vrna_bps_t* bps)

Release memory of a base pair stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Parameters

- **bps** – The base pair stack to be free'd

size_t **vrna_bps_push**(*vrna_bps_t* bps, *vrna_bp_t* pair)

Put a new base pair element on top of the stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Parameters

- **bps** – The base pair stack
- **pair** – The base pair to be put onto the stack

Returns

The size of the base pair stack after pushing the base pair

vrna_bp_t **vrna_bps_top**(*vrna_bps_t* bps)

Retrieve the top element of the base pair stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Retrieves the last element put onto the stack, or a zero'd out *vrna_bp_t* structure. The latter is returned on error or when topping an empty stack.

Parameters

- **bps** – The base pair stack

Returns

The top element of the base pair stack, or a zero'd out *vrna_bp_t*

vrna_bp_t **vrna_bps_pop**(*vrna_bps_t* bps)

Pop last element of from base pair stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Retrieves and removes the last element put onto the stack, or a zero'd out *vrna_bp_t* structure. The latter is returned on error or when topping an empty stack.

Parameters

- **bps** – The base pair stack

Returns

The top element of the backtrack stack, or a zero'd out *vrna_bp_t*

vrna_bp_t **vrna_bps_at**(*vrna_bps_t* bps, size_t n)

Retrieve the n'th element of the base pair stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Retrieves the n'th element counted from the bottom of the stack (0-based), or a zero'd out *vrna_bp_t* structure. The latter is returned on error or when **n** is outside the size of the stack.

Parameters

- **bps** – The base pair stack
- **n** – The position within the stack

Returns

The n'th element of the base pair stack, or a zero'd out *vrna_bp_t*

size_t **vrna_bps_size**(*vrna_bps_t* bps)

Get the size of the base pair stack.

```
#include <ViennaRNA/datastructures/basic.h>
```

Parameters

- **bps** – The base pair stack

Returns

The size of the base pair stack

struct **vrna_sect_s**

#include <ViennaRNA/datastructures/basic.h> Stack of partial structures for backtracking.

Public Members

int **i**

```
#include <ViennaRNA/datastructures/basic.h>
```

int **j**

```
#include <ViennaRNA/datastructures/basic.h>
```

unsigned int **ml**

```
#include <ViennaRNA/datastructures/basic.h>
```

struct **vrna_basepair_s**

#include <*ViennaRNA/datastructures/basic.h*> Base pair data structure used in subopt.c.

Public Members

unsigned int **i**

```
#include <ViennaRNA/datastructures/basic.h>
```

unsigned int **j**

```
#include <ViennaRNA/datastructures/basic.h>
```

unsigned int **L**

```
#include <ViennaRNA/datastructures/basic.h>
```

unsigned int **l**[3]

```
#include <ViennaRNA/datastructures/basic.h>
```

4.2.3 Files and I/O

Nucleic Acid Sequences and Structures

Defines

VRNA_OPTION_MULTILINE

Tell a function that an input is assumed to span several lines.

```
#include <ViennaRNA/io/file_formats.h>
```

If used as input-option a function might also be returning this state telling that it has read data from multiple lines.

See also

vrna_extract_record_rest_structure(), *vrna_file_fasta_read_record()*

VRNA_CONSTRAINT_MULTILINE

parse multiline constraint

```
#include <ViennaRNA/io/file_formats.h>
```

Deprecated:

see *vrna_extract_record_rest_structure()*

VRNA_INPUT_VERBOSE

```
#include <ViennaRNA/io/file_formats.h>
```

Functions

void **vrna_file_helixlist**(const char *seq, const char *db, float energy, FILE *file)

Print a secondary structure as helix list.

```
#include <ViennaRNA/io/file_formats.h>
```

Parameters

- **seq** – The RNA sequence
- **db** – The structure in dot-bracket format
- **energy** – Free energy of the structure in kcal/mol
- **file** – The file handle used to print to (print defaults to 'stdout' if(file == NULL))

void **vrna_file_connect**(const char *seq, const char *db, float energy, const char *identifier, FILE *file)

Print a secondary structure as connect table.

```
#include <ViennaRNA/io/file_formats.h>
```

Connect table file format looks like this:

```
* 300  ENERGY = 7.0  example
* 1 G       0    2   22    1
* 2 G       1    3   21    2
*
```

where the headerline is followed by 6 columns with:

a. Base number: index n

b. Base (A, C, G, T, U, X)

c. Index n-1 (0 if first nucleotide)

d. Index n+1 (0 if last nucleotide)

e. Number of the base to which n is paired. No pairing is indicated by 0 (zero).

f. Natural numbering.

Parameters

- **seq** – The RNA sequence
- **db** – The structure in dot-bracket format
- **energy** – The free energy of the structure
- **identifier** – An optional identifier for the sequence
- **file** – The file handle used to print to (print defaults to 'stdout' if(file == NULL))

void **vrna_file_bpseq**(const char *seq, const char *db, FILE *file)

Print a secondary structure in bpseq format.

```
#include <ViennaRNA/io/file_formats.h>
```

Parameters

- **seq** – The RNA sequence
- **db** – The structure in dot-bracket format
- **file** – The file handle used to print to (print defaults to 'stdout' if(file == NULL))

void **vrna_file_json**(const char *seq, const char *db, double energy, const char *identifier, FILE *file)

Print a secondary structure in jsonformat.

```
#include <ViennaRNA/io/file_formats.h>
```

Parameters

- **seq** – The RNA sequence
- **db** – The structure in dot-bracket format
- **energy** – The free energy
- **identifier** – An identifier for the sequence
- **file** – The file handle used to print to (print defaults to 'stdout' if(file == NULL))

unsigned int **vrna_file_fasta_read_record**(char **header, char **sequence, char ***rest, FILE *file, unsigned int options)

Get a (fasta) data set from a file or stdin.

```
#include <ViennaRNA/io/file_formats.h>
```

This function may be used to obtain complete datasets from a filehandle or stdin. A dataset is always defined to contain at least a sequence. If data starts with a fasta header, i.e. a line like

```
>some header info
```

then *vrna_file_fasta_read_record()* will assume that the sequence that follows the header may span over several lines. To disable this behavior and to assign a single line to the argument 'sequence' one can pass *VRNA_INPUT_NO_SPAN* in the 'options' argument. If no fasta header is read in the beginning of a data block, a sequence must not span over multiple lines!

Unless the options *VRNA_INPUT_NOSKIP_COMMENTS* or *VRNA_INPUT_NOSKIP_BLANK_LINES* are passed, a sequence may be interrupted by lines starting with a comment character or empty lines.

A sequence is regarded as completely read if it was either assumed to not span over multiple lines, a secondary structure or structure constraint follows the sequence on the next line, or a new header marks the beginning of a new sequence...

All lines following the sequence (this includes comments) that do not initiate a new dataset according to the above definition are available through the line-array 'rest'. Here one can usually find the structure constraint or other information belonging to the current dataset. Filling of 'rest' may be prevented by passing *VRNA_INPUT_NO_REST* to the options argument.

The main purpose of this function is to be able to easily parse blocks of data in the header of a loop where all calculations for the appropriate data is done inside the loop. The loop may be then left on certain return values, e.g.:

```
char *id, *seq, **rest;
int  i;
id = seq = NULL;
rest = NULL;
while(!(vrna_file_fasta_read_record(&id, &seq, &rest, NULL, 0) & (VRNA_INPUT_
↪ERROR | VRNA_INPUT_QUIT))){
if(id)
    printf("%s\n", id);
  printf("%s\n", seq);
if(rest)
for(i=0;rest[i];i++){
      printf("%s\n", rest[i]);
      free(rest[i]);
    }
  free(rest);
  free(seq);
  free(id);
}
```

In the example above, the while loop will be terminated when *vrna_file_fasta_read_record()* returns either an error, EOF, or a user initiated quit request.

As long as data is read from stdin (we are passing NULL as the file pointer), the id is printed if it is available for the current block of data. The sequence will be printed in any case and if some more lines belong to the current block of data each line will be printed as well.

Note

This function will exit any program with an error message if no sequence could be read!

This function is NOT threadsafe! It uses a global variable to store information about the next data block. Do not forget to free the memory occupied by header, sequence and rest!

Parameters

- **header** – A pointer which will be set such that it points to the header of the record
- **sequence** – A pointer which will be set such that it points to the sequence of the record
- **rest** – A pointer which will be set such that it points to an array of lines which also belong to the record
- **file** – A file handle to read from (if NULL, this function reads from stdin)
- **options** – Some options which may be passed to alter the behavior of the function, use 0 for no options

Returns

A flag with information about what the function actually did read

char ***vrna_extract_record_rest_structure**(const char **lines, unsigned int length, unsigned int option)

Extract a dot-bracket structure string from (multiline)character array.

```
#include <ViennaRNA/io/file_formats.h>
```

This function extracts a dot-bracket structure string from the 'rest' array as returned by *vrna_file_fasta_read_record()* and returns it. All occurences of comments within the 'lines' array will

be skipped as long as they do not break the structure string. If no structure could be read, this function returns NULL.

See also

vrna_file_fasta_read_record()

Parameters

- **lines** – The (multiline) character array to be parsed
- **length** – The assumed length of the dot-bracket string (passing a value < 1 results in no length limit)
- **option** – Some options which may be passed to alter the behavior of the function, use 0 for no options

Pre

The argument 'lines' has to be a 2-dimensional character array as obtained by *vrna_file_fasta_read_record()*

Returns

The dot-bracket string read from lines or NULL

int **vrna_file_SHAPE_read**(const char *file_name, int length, double default_value, char *sequence, double *values)

Read data from a given SHAPE reactivity input file.

```
#include <ViennaRNA/io/file_formats.h>
```

This function parses the informations from a given file and stores the result in the preallocated string sequence and the double array values.

Parameters

- **file_name** – Path to the constraints file
- **length** – Length of the sequence (file entries exceeding this limit will cause an error)
- **default_value** – Value for missing indices
- **sequence** – Pointer to an array used for storing the sequence obtained from the SHAPE reactivity file
- **values** – Pointer to an array used for storing the values obtained from the SHAPE reactivity file

double ***vrna_file_doubles_read**(const char *file_name, unsigned int options, size_t *values_count)

Read double values separated by newlines from a given input file.

```
#include <ViennaRNA/io/file_formats.h>
```

This function parses an input file `file_name` expecting a double value per line. It then extracts all the double values and returns them as double array.

Parameters

- **file_name** – Path to the input file
- **options** – Option flags to change the behavior of this function
- **values_count** – The adress of a counter to store the total size of the returned double array

Returns
An array of double values read from the input file

int **vrna_file_connect_read_record**(FILE *fp, char **id, char **sequence, char **structure, char **remainder, unsigned int options)

```
#include <ViennaRNA/io/file_formats.h>
```

int **vrna_file_RNAstrand_db_read_record**(FILE *fp, char **name_p, char **sequence_p, char **structure_p, char **source_p, char **fname_p, char **id_p, unsigned int options)

```
#include <ViennaRNA/io/file_formats.h>
```

void **vrna_extract_record_rest_constraint**(char **cstruc, const char **lines, unsigned int option)

Extract a hard constraint encoded as pseudo dot-bracket string.

```
#include <ViennaRNA/io/file_formats.h>
```

Deprecated:
Use *vrna_extract_record_rest_structure()* instead!

See also

vrna_file_fasta_read_record(), *VRNA_CONSTRAINT_DB_PIPE*, *VRNA_CONSTRAINT_DB_DOT*, *VRNA_CONSTRAINT_DB_X* VRNA_CONSTRAINT_DB_ANG_BRACK, *VRNA_CONSTRAINT_DB_RND_BRACK*

Parameters

- **cstruc** – A pointer to a character array that is used as pseudo dot-bracket output
- **lines** – A 2-dimensional character array with the extension lines from the FASTA input
- **option** – The option flags that define the behavior and recognition pattern of this function

Pre
The argument 'lines' has to be a 2-dimensional character array as obtained by *vrna_file_fasta_read_record()*

char ***extract_record_rest_structure**(const char **lines, unsigned int length, unsigned int option)

```
#include <ViennaRNA/io/file_formats.h>
```

unsigned int **read_record**(char **header, char **sequence, char ***rest, unsigned int options)

Get a data record from stdin.

```
#include <ViennaRNA/io/file_formats.h>
```

Deprecated:
This function is deprecated! Use *vrna_file_fasta_read_record()* as a replacment.

unsigned int **get_multi_input_line**(char **string, unsigned int options)

```
#include <ViennaRNA/io/file_formats.h>
```

Multiple Sequence Alignments

Defines

VRNA_FILE_FORMAT_MSA_CLUSTAL

Option flag indicating ClustalW formatted files.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_read(), *vrna_file_msa_read_record()*, *vrna_file_msa_detect_format()*

VRNA_FILE_FORMAT_MSA_STOCKHOLM

Option flag indicating Stockholm 1.0 formatted files.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_read(), *vrna_file_msa_read_record()*, *vrna_file_msa_detect_format()*

VRNA_FILE_FORMAT_MSA_FASTA

Option flag indicating FASTA (Pearson) formatted files.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_read(), *vrna_file_msa_read_record()*, *vrna_file_msa_detect_format()*

VRNA_FILE_FORMAT_MSA_MAF

Option flag indicating MAF formatted files.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_read(), *vrna_file_msa_read_record()*, *vrna_file_msa_detect_format()*

VRNA_FILE_FORMAT_MSA_MIS

Option flag indicating most informative sequence (MIS) output.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

The default reference sequence output for an alignment is simply a consensus sequence. This flag allows to write the most informative equence (MIS) instead.

See also

vrna_file_msa_write()

VRNA_FILE_FORMAT_MSA_DEFAULT

Option flag indicating the set of default file formats.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_read(), *vrna_file_msa_read_record()*, *vrna_file_msa_detect_format()*

VRNA_FILE_FORMAT_MSA_NOCHECK

Option flag to disable validation of the alignment.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_read(), *vrna_file_msa_read_record()*

VRNA_FILE_FORMAT_MSA_UNKNOWN

Return flag of *vrna_file_msa_detect_format()* to indicate unknown or malformatted alignment.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_detect_format()

VRNA_FILE_FORMAT_MSA_APPEND

Option flag indicating to append data to a multiple sequence alignment file rather than overwriting it.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_write()

VRNA_FILE_FORMAT_MSA_QUIET

Option flag to suppress unnecessary spam messages on `stderr`.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_read(), *vrna_file_msa_read_record()*

VRNA_FILE_FORMAT_MSA_SILENT

Option flag to completely silence any warnings on `stderr`.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

See also

vrna_file_msa_read(), *vrna_file_msa_read_record()*

Functions

int **vrna_file_msa_read**(const char *filename, char ***names, char ***aln, char **id, char **structure, unsigned int options)

Read a multiple sequence alignment from file.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

This function reads the (first) multiple sequence alignment from an input file. The read alignment is split into the sequence id/name part and the actual sequence information and stored in memory as arrays of ids/names and sequences. If the alignment file format allows for additional information, such as an ID of the entire alignment or consensus structure information, this data is retrieved as well and made available. The `options` parameter allows to specify the set of alignment file formats that should be used to retrieve the data. If 0 is passed as option, the list of alignment file formats defaults to *VRNA_FILE_FORMAT_MSA_DEFAULT*.

Currently, the list of parsable multiple sequence alignment file formats consists of:

- *ClustalW format*
- *Stockholm 1.0 format*

- *FASTA (Pearson) format*
- *MAF format*

SWIG Wrapper Notes:

In the target scripting language, only the first and last argument, `filename` and `options`, are passed to the corresponding function. The other arguments, which serve as output in the C-library, are available as additional return values. This function exists as an overloaded version where the `options` parameter may be omitted! In that case, the `options` parameter defaults to *VRNA_FILE_FORMAT_MSA_STOCKHOLM*. See, e.g. *`RNA.file_msa_read()`* in the *The RNA Python module* and *Parsing Alignments* in the Python examples.

See also

vrna_file_msa_read_record(), *VRNA_FILE_FORMAT_MSA_CLUSTAL*, *VRNA_FILE_FORMAT_MSA_STOCKHOLM*, *VRNA_FILE_FORMAT_MSA_FASTA*, *VRNA_FILE_FORMAT_MSA_MAF*, *VRNA_FILE_FORMAT_MSA_DEFAULT*, *VRNA_FILE_FORMAT_MSA_NOCHECK*

Note

After successfully reading an alignment, this function performs a validation of the data that includes uniqueness of the sequence identifiers, and equal sequence lengths. This check can be deactivated by passing *VRNA_FILE_FORMAT_MSA_NOCHECK* in the `options` parameter.

It is the users responsibility to free any memory occupied by the output arguments `names`, `aln`, `id`, and `structure` after calling this function. The function automatically sets the latter two arguments to `NULL` in case no corresponding data could be retrieved from the input alignment.

Parameters

- **`filename`** – The name of input file that contains the alignment
- **`names`** – An address to the pointer where sequence identifiers should be written to
- **`aln`** – An address to the pointer where aligned sequences should be written to
- **`id`** – An address to the pointer where the alignment ID should be written to (Maybe NULL)
- **`structure`** – An address to the pointer where consensus structure information should be written to (Maybe NULL)
- **`options`** – Options to manipulate the behavior of this function

Returns

The number of sequences in the alignment, or -1 if no alignment record could be found

int **vrna_file_msa_read_record**(FILE *fp, char ***names, char ***aln, char **id, char **structure, unsigned int options)

Read a multiple sequence alignment from file handle.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

Similar to *vrna_file_msa_read()*, this function reads a multiple sequence alignment from an input file handle. Since using a file handle, this function is not limited to the first alignment record, but allows for looping over all alignments within the input.

The read alignment is split into the sequence id/name part and the actual sequence information and stored in memory as arrays of ids/names and sequences. If the alignment file format allows for additional information, such as an ID of the entire alignment or consensus structure information, this data is

retrieved as well and made available. The `options` parameter allows to specify the alignment file format used to retrieve the data. A single format must be specified here, see *vrna_file_msa_detect_format()* for helping to determine the correct MSA file format.

Currently, the list of parsable multiple sequence alignment file formats consists of:

- *ClustalW format*
- *Stockholm 1.0 format*
- *FASTA (Pearson) format*
- *MAF format*

SWIG Wrapper Notes:

In the target scripting language, only the first and last argument, `fp` and `options`, are passed to the corresponding function. The other arguments, which serve as output in the C-library, are available as additional return values. This function exists as an overloaded version where the `options` parameter may be omitted! In that case, the `options` parameter defaults to *VRNA_FILE_FORMAT_MSA_STOCKHOLM*. See, e.g. *`RNA.file_msa_read_record()`* in the *The RNA Python module* and *Parsing Alignments* in the Python examples.

See also

vrna_file_msa_read(), *vrna_file_msa_detect_format()*, *VRNA_FILE_FORMAT_MSA_CLUSTAL*, *VRNA_FILE_FORMAT_MSA_STOCKHOLM*, *VRNA_FILE_FORMAT_MSA_FASTA*, *VRNA_FILE_FORMAT_MSA_MAF*, *VRNA_FILE_FORMAT_MSA_DEFAULT*, *VRNA_FILE_FORMAT_MSA_NOCHECK*

Note

After successfully reading an alignment, this function performs a validation of the data that includes uniqueness of the sequence identifiers, and equal sequence lengths. This check can be deactivated by passing *VRNA_FILE_FORMAT_MSA_NOCHECK* in the `options` parameter.

It is the users responsibility to free any memory occupied by the output arguments `names`, `aln`, `id`, and `structure` after calling this function. The function automatically sets the latter two arguments to `NULL` in case no corresponding data could be retrieved from the input alignment.

Parameters

- **`fp`** – The file pointer the data will be retrieved from
- **`names`** – An address to the pointer where sequence identifiers should be written to
- **`aln`** – An address to the pointer where aligned sequences should be written to
- **`id`** – An address to the pointer where the alignment ID should be written to (Maybe NULL)
- **`structure`** – An address to the pointer where consensus structure information should be written to (Maybe NULL)
- **`options`** – Options to manipulate the behavior of this function

Returns

The number of sequences in the alignment, or -1 if no alignment record could be found

unsigned int **`vrna_file_msa_detect_format`**(const char *filename, unsigned int options)

Detect the format of a multiple sequence alignment file.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

This function attempts to determine the format of a file that supposedly contains a multiple sequence alignment (MSA). This is useful in cases where a MSA file contains more than a single record and therefore *vrna_file_msa_read()* can not be applied, since it only retrieves the first. Here, one can try to guess the correct file format using this function and then loop over the file, record by record using one of the low-level record retrieval functions for the corresponding MSA file format.

SWIG Wrapper Notes:

This function exists as an overloaded version where the `options` parameter may be omitted! In that case, the `options` parameter defaults to *VRNA_FILE_FORMAT_MSA_DEFAULT*. See, e.g. *RNA.file_msa_detect_format()* in the *The RNA Python module* .

See also

vrna_file_msa_read(), vrna_file_stockholm_read_record(), vrna_file_clustal_read_record(), *vrna_file_fasta_read_record()*

Note

This function parses the entire first record within the specified file. As a result, it returns *VRNA_FILE_FORMAT_MSA_UNKNOWN* not only if it can't detect the file's format, but also in cases where the file doesn't contain sequences!

Parameters

- **filename** – The name of input file that contains the alignment
- **options** – Options to manipulate the behavior of this function

Returns

The MSA file format, or *VRNA_FILE_FORMAT_MSA_UNKNOWN*

int **vrna_file_msa_write**(const char *filename, const char **names, const char **aln, const char *id, const char *structure, const char *source, unsigned int options)

Write multiple sequence alignment file.

```
#include <ViennaRNA/io/file_formats_msa.h>
```

SWIG Wrapper Notes:

In the target scripting language, this function exists as a set of overloaded versions, where the last four parameters may be omitted. If the `options` parameter is missing the options default to (*VRNA_FILE_FORMAT_MSA_STOCKHOLM* | *VRNA_FILE_FORMAT_MSA_APPEND*). See, e.g. *RNA.file_msa_write()* in the *The RNA Python module* .

See also

VRNA_FILE_FORMAT_MSA_STOCKHOLM, *VRNA_FILE_FORMAT_MSA_APPEND*, *VRNA_FILE_FORMAT_MSA_MIS*

Note

Currently, we only support msa-formats-stockholm output

Parameters

- **filename** – The output filename
- **names** – The array of sequence names / identifies
- **aln** – The array of aligned sequences
- **id** – An optional ID for the alignment
- **structure** – An optional consensus structure
- **source** – A string describing the source of the alignment
- **options** – Options to manipulate the behavior of this function

Returns

Non-null upon successfully writing the alignment to file

Command Files

Functions to parse and interpret the content of usage-io-commands .

Defines

VRNA_CMD_PARSE_HC

Command parse/apply flag indicating hard constraints.

```
#include <ViennaRNA/io/commands.h>
```

See also

vrna_cmd_t, *vrna_file_commands_read()*, *vrna_file_commands_apply()*, *vrna_commands_apply()*

VRNA_CMD_PARSE_SC

Command parse/apply flag indicating soft constraints.

```
#include <ViennaRNA/io/commands.h>
```

See also

vrna_cmd_t, *vrna_file_commands_read()*, *vrna_file_commands_apply()*, *vrna_commands_apply()*

VRNA_CMD_PARSE_UD

Command parse/apply flag indicating unstructured domains.

```
#include <ViennaRNA/io/commands.h>
```

See also

vrna_cmd_t, *vrna_file_commands_read()*, *vrna_file_commands_apply()*, *vrna_commands_apply()*

VRNA_CMD_PARSE_SD

Command parse/apply flag indicating structured domains.

```
#include <ViennaRNA/io/commands.h>
```

See also

vrna_cmd_t, *vrna_file_commands_read()*, *vrna_file_commands_apply()*, *vrna_commands_apply()*

VRNA_CMD_PARSE_DEFAULTS

Command parse/apply flag indicating default set of commands.

```
#include <ViennaRNA/io/commands.h>
```

See also

vrna_cmd_t, *vrna_file_commands_read()*, *vrna_file_commands_apply()*, *vrna_commands_apply()*

VRNA_CMD_PARSE_SILENT

```
#include <ViennaRNA/io/commands.h>
```

Typedefs

typedef struct vrna_command_s ***vrna_cmd_t**

A data structure that contains commands.

```
#include <ViennaRNA/io/commands.h>
```

Functions

vrna_cmd_t **vrna_file_commands_read**(const char *filename, unsigned int options)

Extract a list of commands from a command file.

```
#include <ViennaRNA/io/commands.h>
```

Read a list of commands specified in the input file and return them as list of abstract commands

SWIG Wrapper Notes:

This function is available as global function `file_commands_read()`. See, e.g. *RNA.file_commands_read()* in the *The RNA Python module* .

See also

vrna_commands_apply(), *vrna_file_commands_apply()*, *vrna_commands_free()*

Parameters

- **filename** – The filename
- **options** – Options to limit the type of commands read from the file

Returns

A list of abstract commands

int **vrna_file_commands_apply**(*vrna_fold_compound_t* *fc, const char *filename, unsigned int options)

Apply a list of commands from a command file.

```
#include <ViennaRNA/io/commands.h>
```

This function is a shortcut to directly parse a commands file and apply all successfully parsed commands to a *vrna_fold_compound_t* data structure.

SWIG Wrapper Notes:

This function is attached as method `file_commands_apply()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.file_commands_apply()* in the *The RNA Python module* .

Parameters

- **fc** – The *vrna_fold_compound_t* the command list will be applied to
- **filename** – The filename
- **options** – Options to limit the type of commands read from the file

Returns

The number of commands successfully applied

int **vrna_commands_apply**(*vrna_fold_compound_t* *fc, *vrna_cmd_t* commands, unsigned int options)

Apply a list of commands to a *vrna_fold_compound_t*.

```
#include <ViennaRNA/io/commands.h>
```

SWIG Wrapper Notes:

This function is attached as method `commands_apply()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.commands_apply()* in the *The RNA Python module* .

Parameters

- **fc** – The *vrna_fold_compound_t* the command list will be applied to

- **commands** – The commands to apply
- **options** – Options to limit the type of commands read from the file

Returns
The number of commands successfully applied

void **vrna_commands_free**(*vrna_cmd_t* commands)

Free memory occupied by a list of commands.

```
#include <ViennaRNA/io/commands.h>
```

Release memory occupied by a list of commands

Parameters

- **commands** – A pointer to a list of commands

Other I/O Utils

Functions

void **vrna_file_copy**(FILE *from, FILE *to)

Inefficient 'cp'.

```
#include <ViennaRNA/io/utils.h>
```

char ***vrna_read_line**(FILE *fp)

Read a line of arbitrary length from a stream.

```
#include <ViennaRNA/io/utils.h>
```

Returns a pointer to the resulting string. The necessary memory is allocated and should be released using *free()* when the string is no longer needed.

Parameters

- **fp** – A file pointer to the stream where the function should read from

Returns
A pointer to the resulting string

int **vrna_mkdir_p**(const char *path)

Recursivly create a directory tree.

```
#include <ViennaRNA/io/utils.h>
```

char ***vrna_basename**(const char *path)

Extract the filename from a file path.

```
#include <ViennaRNA/io/utils.h>
```

char ***vrna_dirname**(const char *path)

Extract the directory part of a file path.

```
#include <ViennaRNA/io/utils.h>
```

char *vrna_filename_sanitize(const char *name, const char *replacement)

Sanitize a file name.

```
#include <ViennaRNA/io/utils.h>
```

Returns a new file name where all invalid characters are substituted by a replacement character. If no replacement character is supplied, invalid characters are simply removed from the filename. File names may also never exceed a length of 255 characters. Longer file names will undergo a 'smart' truncation process, where the filenames' suffix, i.e. everything after the last dot '.', is attempted to be kept intact. Hence, only the filename part before the suffix is reduced in such a way that the total filename complies to the length restriction of 255 characters. If no suffix is present or the suffix itself already exceeds the maximum length, the filename is simply truncated from the back of the string.

For now we consider the following characters invalid:

- backslash ''
- slash '/'
- question mark '?'
- percent sign ''
- asterisk '*'
- colon ':'
- pipe symbol '|'
- double quote '"'
- triangular brackets '<' and '>'

Furthermore, the (resulting) file name must not be a reserved file name, such as:

- '.'
- '..'

Note

This function allocates a new block of memory for the sanitized string. It also may return (a) NULL if the input is pointing to NULL, or (b) an empty string if the input only consists of invalid characters which are simply removed!

Parameters

- **name** – The input file name
- **replacement** – The replacement character, or NULL

Returns

The sanitized file name, or NULL

int vrna_file_exists(const char *filename)

Check if a file already exists in the file system.

```
#include <ViennaRNA/io/utils.h>
```

Parameters

- **filename** – The name of (path to) the file to check for existence

Returns

0 if it doesn't exists, 1 otherwise

float **readribosum(char *name)

Read a RiboSum or other user-defined Scoring Matrix and Store into global Memory.

```
#include <ViennaRNA/params/ribosum.h>
```

4.2.4 Messages

Functions to all kinds of messages.

Other messages

Functions

void **vrna_message_error**(const char *format, ...)

Print an error message and die.

```
#include <ViennaRNA/utils/basic.h>
```

This function is a wrapper to *fprintf(stderr, ...)* that puts a capital **ERROR:** in front of the message and then exits the calling program.

Deprecated:

Use *vrna_log_error()* instead! (since v2.7.0)

See also

vrna_message_verror(), *vrna_message_warning()*, *vrna_message_info()*

Parameters

- **format** – The error message to be printed
- **...** – Optional arguments for the formatted message string

void **vrna_message_verror**(const char *format, va_list args)

Print an error message and die.

```
#include <ViennaRNA/utils/basic.h>
```

This function is a wrapper to *vfprintf(stderr, ...)* that puts a capital **ERROR:** in front of the message and then exits the calling program.

Deprecated:

Use *vrna_log_error()* instead! (since v2.7.0)

See also

vrna_message_error(), *vrna_message_warning()*, *vrna_message_info()*

Parameters

- **format** – The error message to be printed

- **args** – The argument list for the formatted message string

void **vrna_message_warning**(const char *format, ...)

Print a warning message.

```
#include <ViennaRNA/utils/basic.h>
```

This function is a wrapper to *fprintf(stderr, ...)* that puts a capital **WARNING:** in front of the message.

Deprecated:

Use *vrna_log_warning()* instead! (since v2.7.0)

See also

vrna_message_vwarning(), *vrna_message_error()*, *vrna_message_info()*

Parameters

- **format** – The warning message to be printed
- **...** – Optional arguments for the formatted message string

void **vrna_message_vwarning**(const char *format, va_list args)

Print a warning message.

```
#include <ViennaRNA/utils/basic.h>
```

This function is a wrapper to *fprintf(stderr, ...)* that puts a capital **WARNING:** in front of the message.

Deprecated:

Use *vrna_log_warning()* instead! (since v2.7.0)

See also

vrna_message_vwarning(), *vrna_message_error()*, *vrna_message_info()*

Parameters

- **format** – The warning message to be printed
- **args** – The argument list for the formatted message string

void **vrna_message_info**(FILE *fp, const char *format, ...)

Print an info message.

```
#include <ViennaRNA/utils/basic.h>
```

This function is a wrapper to *fprintf(...)*.

Deprecated:

Use *vrna_log_info()* instead! (since v2.7.0)

See also

vrna_message_vinfo(), *vrna_message_error()*, *vrna_message_warning()*

Parameters

- **fp** – The file pointer where the message is printed to
- **format** – The warning message to be printed
- **...** – Optional arguments for the formatted message string

void **vrna_message_vinfo**(FILE *fp, const char *format, va_list args)

Print an info message.

```
#include <ViennaRNA/utils/basic.h>
```

This function is a wrapper to *fprintf(...)*.

Deprecated:

Use *vrna_log_info()* instead! (since v2.7.0)

See also

vrna_message_vinfo(), *vrna_message_error()*, *vrna_message_warning()*

Parameters

- **fp** – The file pointer where the message is printed to
- **format** – The info message to be printed
- **args** – The argument list for the formatted message string

void **vrna_message_input_seq_simple**(void)

Print a line to *stdout* that asks for an input sequence.

```
#include <ViennaRNA/utils/basic.h>
```

There will also be a ruler (scale line) printed that helps orientation of the sequence positions

void **vrna_message_input_seq**(const char *s)

Print a line with a user defined string and a ruler to stdout.

```
#include <ViennaRNA/utils/basic.h>
```

(usually this is used to ask for user input) There will also be a ruler (scale line) printed that helps orientation of the sequence positions

Parameters

- **s** – A user defined string that will be printed to stdout

void **vrna_message_input_msa**(const char *s)

```
#include <ViennaRNA/utils/basic.h>
```

4.2.5 Log Messages

API symbols for interacting with the ViennaRNA logging system.

Logging System API

Defines

VRNA_LOG_LEVEL_DEFAULT

Default log level.

```
#include <ViennaRNA/utils/log.h>
```

See also

vrna_log_level_set(), *vrna_log_reset()*, *vrna_log_level()*, *vrna_log_levels_e*

VRNA_LOG_OPTION_QUIET

Log option to turn off internal logging.

```
#include <ViennaRNA/utils/log.h>
```

When this option is set via *vrna_log_options_set()* the internal logging system will be deactivated and only user-defined callbacks will be seeing any logs.

See also

vrna_log_options_set(), *vrna_log_options()*, *vrna_log_reset()*, *VRNA_LOG_OPTION_TRACE_CALL*, *VRNA_LOG_OPTION_TRACE_FUNC*, *VRNA_LOG_OPTION_TRACE_TIME*, *VRNA_LOG_OPTION_DEFAULT*

VRNA_LOG_OPTION_TRACE_CALL

Log option to turn on call tracing.

```
#include <ViennaRNA/utils/log.h>
```

When this option is set via *vrna_log_options_set()* the internal logging system will include a call trace to the log output, i.e. the source code file and line numbers will be included in the log message.

See also

vrna_log_options_set(), *vrna_log_options()*, *vrna_log_reset()*, *VRNA_LOG_OPTION_QUIET*, *VRNA_LOG_OPTION_TRACE_TIME*, *VRNA_LOG_OPTION_TRACE_FUNC*, *VRNA_LOG_OPTION_DEFAULT*

VRNA_LOG_OPTION_TRACE_FUNC

Log option to turn on calling funmction tracing.

```
#include <ViennaRNA/utils/log.h>
```

When this option is set via *vrna_log_options_set()* the internal logging system will include a call trace to the log output to retrieve the name of the function that issued the log. The function name will then be included in the log message.

See also

vrna_log_options_set(), *vrna_log_options()*, *vrna_log_reset()*, *VRNA_LOG_OPTION_QUIET*, *VRNA_LOG_OPTION_TRACE_CALL*, *VRNA_LOG_OPTION_TRACE_TIME*, *VRNA_LOG_OPTION_DEFAULT*

VRNA_LOG_OPTION_TRACE_TIME

Log option to turn on time stamp.

```
#include <ViennaRNA/utils/log.h>
```

When this option is set via *vrna_log_options_set()* the internal logging system will include a time stamp to the log output, i.e. the time when the log message was issued will be included in the log message.

See also

vrna_log_options_set(), *vrna_log_options()*, *vrna_log_reset()*, *VRNA_LOG_OPTION_QUIET*, *VRNA_LOG_OPTION_TRACE_CALL*, *VRNA_LOG_OPTION_TRACE_FUNC*, *VRNA_LOG_OPTION_DEFAULT*

VRNA_LOG_OPTION_DEFAULT

Log option representing the default options.

```
#include <ViennaRNA/utils/log.h>
```

When this option is set via *vrna_log_options_set()* the default options will be set.

See also

vrna_log_options_set(), *vrna_log_options()*, *vrna_log_reset()*, *VRNA_LOG_OPTION_QUIET*, *VRNA_LOG_OPTION_TRACE_CALL*, *VRNA_LOG_OPTION_TRACE_FUNC*, *VRNA_LOG_OPTION_TRACE_TIME*

vrna_log_debug(...)

Issue a debug log message.

```
#include <ViennaRNA/utils/log.h>
```

This macro expects a printf-like format string followed by a variable list of arguments for the format string and passes this content to the log system.

See also

vrna_log_info, *vrna_log_warning*, *vrna_log_error*, *vrna_log_critical*, *vrna_log()*, *vrna_log_level_set()*, *vrna_log_options_set()*, *vrna_log_fp_set()*

vrna_log_info(...)

Issue an info log message.

```
#include <ViennaRNA/utils/log.h>
```

This macro expects a printf-like format string followed by a variable list of arguments for the format string and passes this content to the log system.

See also

vrna_log_debug, *vrna_log_warning*, *vrna_log_error*, *vrna_log_critical*, *vrna_log()*, *vrna_log_level_set()*, *vrna_log_options_set()*, *vrna_log_fp_set()*

vrna_log_warning(...)

Issue a warning log message.

```
#include <ViennaRNA/utils/log.h>
```

This macro expects a printf-like format string followed by a variable list of arguments for the format string and passes this content to the log system.

See also

vrna_log_debug, *vrna_log_info*, *vrna_log_error*, *vrna_log_critical*, *vrna_log()*, *vrna_log_level_set()*, *vrna_log_options_set()*, *vrna_log_fp_set()*

vrna_log_error(...)

Issue an error log message.

```
#include <ViennaRNA/utils/log.h>
```

This macro expects a printf-like format string followed by a variable list of arguments for the format string and passes this content to the log system.

See also

vrna_log_debug, *vrna_log_info*, *vrna_log_warning*, *vrna_log_critical*, *vrna_log()*, *vrna_log_level_set()*, *vrna_log_options_set()*, *vrna_log_fp_set()*

vrna_log_critical(...)

Issue a critical log message.

```
#include <ViennaRNA/utils/log.h>
```

This macro expects a printf-like format string followed by a variable list of arguments for the format string and passes this content to the log system.

See also

vrna_log_debug, *vrna_log_info*, *vrna_log_warning*, *vrna_log_error*, *vrna_log()*, *vrna_log_level_set()*, *vrna_log_options_set()*, *vrna_log_fp_set()*

Typedefs

typedef struct *vrna_log_event_s* **vrna_log_event_t**

A log event.

```
#include <ViennaRNA/utils/log.h>
```

typedef void (***vrna_log_lock_f**)(int lock, void *lock_data)

The lock function prototype that may be passed to the logging system.

```
#include <ViennaRNA/utils/log.h>
```

See also

vrna_log_lock_set()

Param lock
A parameter indicating whether to lock (lock != 0) or unlock (lock == 0)

Param lock_data
An arbitrary user-defined data pointer for the user-defined locking system

typedef void (***vrna_log_cb_f**)(*vrna_log_event_t* *event, void *log_data)

The log callback function prototype.

```
#include <ViennaRNA/utils/log.h>
```

See also

vrna_log_cb_add(), *vrna_log_cb_num()*, *vrna_log_cb_remove()*

Param event
The log event

Param log_data
An arbitrary user-defined data pointer for the user-define log message receiver

typedef void (***vrna_logdata_free_f**)(void *data)

```
#include <ViennaRNA/utils/log.h>
```

Enums

enum **vrna_log_levels_e**

The log levels.

```
#include <ViennaRNA/utils/log.h>
```

Values:

enumerator **VRNA_LOG_LEVEL_UNKNOWN**

Unknown log level

enumerator **VRNA_LOG_LEVEL_DEBUG**

Debug log level

enumerator **VRNA_LOG_LEVEL_INFO**

Info log level

enumerator **VRNA_LOG_LEVEL_WARNING**

Warning log level

enumerator **VRNA_LOG_LEVEL_ERROR**

Error log level

enumerator **VRNA_LOG_LEVEL_CRITICAL**

Critical log level

enumerator **VRNA_LOG_LEVEL_SILENT**

Silent log level

Functions

void **vrna_log**(*vrna_log_levels_e* level, const char *file_name, const char *function, int line_number, const char *format_string, ...)

Issue a log message.

```
#include <ViennaRNA/utils/log.h>
```

This is the low-level log message function.

- *vrna_log_debug*
- *vrna_log_info*
- *vrna_log_warning*
- *vrna_log_error*
- *vrna_log_critical*

See also

vrna_log_debug, *vrna_log_info*, *vrna_log_warning*, *vrna_log_error*, *vrna_log_critical*, *vrna_log_level_set()*, *vrna_log_options_set()*, *vrna_log_fp_set()*

Warning

Usually, you don't want to call this function directly but rather call one of the following high-level macros instead:

Parameters

- **level** – The log level
- **file_name** – The source code file name of the file that issued the log
- **function** – The name of the lexically-enclosing function that issued the log
- **line_number** – The source code line number that issued the log
- **format_string** – The printf-like format string containing the log message
- **...** – The variable argument list for the printf-like `format_string`

vrna_log_levels_e **vrna_log_level**(void)

Get the current default log level.

```
#include <ViennaRNA/utils/log.h>
```

See also

vrna_log_level_set(), *vrna_log_levels_e*

Returns

The current default log level

int **vrna_log_level_set**(*vrna_log_levels_e* level)

Set the default log level.

```
#include <ViennaRNA/utils/log.h>
```

Set the log level for the default log output system. Any user-defined log callback mechanism will not be affected…

See also

vrna_log_level(), *vrna_log_levels_e*, *vrna_log_cb_add()*, *vrna_log_reset()*

Parameters

- **level** – The new log level for the default logging system

Returns

The (updated) log level of the default logging system

unsigned int **vrna_log_options**(void)

Get the current log options of the default logging system.

```
#include <ViennaRNA/utils/log.h>
```

See also

vrna_log_options_set(), *VRNA_LOG_OPTION_QUIET*, *VRNA_LOG_OPTION_TRACE_TIME*, *VRNA_LOG_OPTION_TRACE_CALL*, *VRNA_LOG_OPTION_TRACE_FUNC*, *VRNA_LOG_OPTION_DEFAULT*

Returns

The current options for the default logging system

void **vrna_log_options_set**(unsigned int options)

Set the log options for the default logging system.

```
#include <ViennaRNA/utils/log.h>
```

See also

vrna_log_options(), *VRNA_LOG_OPTION_QUIET*, *VRNA_LOG_OPTION_TRACE_TIME*, *VRNA_LOG_OPTION_TRACE_CALL*, *VRNA_LOG_OPTION_TRACE_FUNC*, *VRNA_LOG_OPTION_DEFAULT*

Parameters

- **options** – The new options for the default logging system

FILE ***vrna_log_fp**(void)

Get the output file pointer for the default logging system.

```
#include <ViennaRNA/utils/log.h>
```

Returns

The file pointer where the default logging system will print log messages to

void **vrna_log_fp_set**(FILE *fp)

Set the output file pointer for the default logging system.

```
#include <ViennaRNA/utils/log.h>
```

Parameters

- **fp** – The file pointer where the default logging system should print log messages to

size_t **vrna_log_cb_add**(*vrna_log_cb_f* cb, void *data, *vrna_logdata_free_f* data_release, *vrna_log_levels_e* level)

Add a user-defined log message callback.

```
#include <ViennaRNA/utils/log.h>
```

This function will add the user-defined callback `cb` to the logging system that will receive log messages from RNAlib. The callback will be called for each issued message that has a level of at least `level`. The pointer `data` will be passed-through to the callback and may store arbitrary data required for the callback.

See also

vrna_log_cb_f, *vrna_log_cb_num()*, *vrna_log_cb_remove()*, *vrna_log_reset()*

Parameters

- **cb** – The callback function
- **data** – The data passed through to the callback function
- **data_release** – A function that releases memory occupied by `data` (maybe NULL)
- **level** – The log level threshold for this callback

Returns

The current number of log message callbacks stored in the logging system

size_t **vrna_log_cb_num**(void)

Get the current number of log message callbacks.

```
#include <ViennaRNA/utils/log.h>
```

Returns

The current number of log message callbacks stored in the logging system

size_t **vrna_log_cb_remove**(*vrna_log_cb_f* cb, void *data)

Remove a log message callback.

```
#include <ViennaRNA/utils/log.h>
```

This function removes the log message callback `cb` from the logging system. It does so by searching through the list of known log message callbacks and comparing function (and data) addresses.

See also

vrna_log_cb_f, *vrna_log_cb_num()*, *vrna_log_cb_add()*, *vrna_log_reset()*

Warning

The first callback stored in the logging system that matches `cb` will be removed! If `data` is supplied as well, the first callback that matches both, function and data address will be removed.

Parameters

- **cb** – The callback function to remove
- **data** – The data that goes along with the callback

Returns

0 on any error, e.g. if the callback was not found, non-zero if it was removed

void **vrna_log_lock_set**(*vrna_log_lock_f* cb, void *data)

Specify a lock function to be used for the logging system.

```
#include <ViennaRNA/utils/log.h>
```

To prevent undefined behavior in multi-threaded calls to the log system, each log message should be issued as an atomic block. For this to happen, a locking-/unlocking mechanism is required that ensures that log messages from other threads will be blocked until the current log message has been finalized. By default, we use pthreads mutex locking. Using this function, the locking mechanism can be changed to something else, or implemented after all if the ViennaRNA Package was compiled without pthreads support.

See also

vrna_log_lock_f, *vrna_log_reset()*

Parameters

- **cb** – The locking-/unlocking callback
- **data** – An arbitrary data pointer passed through to the callback

void **vrna_log_reset**(void)

Reset the logging system.

```
#include <ViennaRNA/utils/log.h>
```

This resets the logging system and restores default settings

struct **vrna_log_event_s**

#include <ViennaRNA/utils/log.h> A log event.

Public Members

const char ***format_string**

```
#include <ViennaRNA/utils/log.h>
```

The printf-like format string containing the log information

va_list **params**

```
#include <ViennaRNA/utils/log.h>
```

The parameters for the printf-like format string

vrna_log_levels_e **level**

```
#include <ViennaRNA/utils/log.h>
```

The log level

int **line_number**

```
#include <ViennaRNA/utils/log.h>
```

The source code line number that issued the log

const char ***file_name**

```
#include <ViennaRNA/utils/log.h>
```

The source code file that issued the log

const char ***function**

```
#include <ViennaRNA/utils/log.h>
```

The name of the lexically-enclosing function that issued the log

4.2.6 Nucleotide Alphabets

Functions to cope with various aspects related to the nucleotide sequence alphabet.

Defines

VRNA_SEQUENCE_RNA

```
#include <ViennaRNA/sequences/sequence.h>
```

VRNA_SEQUENCE_DNA

```
#include <ViennaRNA/sequences/sequence.h>
```

Typedefs

typedef struct *vrna_sequence_s* **vrna_seq_t**

Typename for nucleotide sequence representation data structure *vrna_sequence_s*.

```
#include <ViennaRNA/sequences/sequence.h>
```

typedef struct *vrna_alignment_s* **vrna_msa_t**

```
#include <ViennaRNA/sequences/sequence.h>
```

Enums

enum **vrna_seq_type_e**

A enumerator used in *vrna_sequence_s* to distinguish different nucleotide sequences.

```
#include <ViennaRNA/sequences/sequence.h>
```

Values:

enumerator **VRNA_SEQ_UNKNOWN**

Nucleotide sequence represents an Unkown type.

enumerator **VRNA_SEQ_RNA**

Nucleotide sequence represents an RNA type.

enumerator **VRNA_SEQ_DNA**

Nucleotide sequence represents a DNA type.

Functions

unsigned int **vrna_sequence_length_max**(unsigned int options)

```
#include <ViennaRNA/sequences/alphabet.h>
```

int **vrna_nucleotide_IUPAC_identity**(char a, char b)

```
#include <ViennaRNA/sequences/alphabet.h>
```

void **vrna_ptypes_prepare**(*vrna_fold_compound_t* *fc, unsigned int options)

```
#include <ViennaRNA/sequences/alphabet.h>
```

char ***vrna_ptypes**(const short *S, const *vrna_md_t* *md)

Get an array of the numerical encoding for each possible base pair (i,j).

```
#include <ViennaRNA/sequences/alphabet.h>
```

See also

vrna_idx_col_wise(), *vrna_fold_compound_t*

Note

This array is always indexed in column-wise order, in contrast to previously different indexing between mfe and pf variants!

short ***vrna_seq_encode**(const char *sequence, const *vrna_md_t* *md)

Get a numerical representation of the nucleotide sequence.

```
#include <ViennaRNA/sequences/alphabet.h>
```

Parameters

- **sequence** – The input sequence in upper-case letters
- **md** – A pointer to a *vrna_md_t* data structure that specifies the conversion type

Returns

A list of integer encodings for each sequence letter (1-based). Position 0 denotes the length of the list

short *vrna_seq_encode_simple(const char *sequence, const *vrna_md_t* *md)

Get a numerical representation of the nucleotide sequence (simple version).

```
#include <ViennaRNA/sequences/alphabet.h>
```

int vrna_nucleotide_encode(char c, const *vrna_md_t* *md)

Encode a nucleotide character to numerical value.

```
#include <ViennaRNA/sequences/alphabet.h>
```

This function encodes a nucleotide character to its numerical representation as required by many functions in RNAlib.

See also

vrna_nucleotide_decode(), *vrna_seq_encode()*

Parameters

- **c** – The nucleotide character to encode
- **md** – The model details that determine the kind of encoding

Returns

The encoded nucleotide

char vrna_nucleotide_decode(int enc, const *vrna_md_t* *md)

Decode a numerical representation of a nucleotide back into nucleotide alphabet.

```
#include <ViennaRNA/sequences/alphabet.h>
```

This function decodes a numerical representation of a nucleotide character back into nucleotide alphabet

See also

vrna_nucleotide_encode(), *vrna_seq_encode()*

Parameters

- **enc** – The encoded nucleotide
- **md** – The model details that determine the kind of decoding

Returns

The decoded nucleotide character

void vrna_aln_encode(const char *sequence, short **S_p, short **s5_p, short **s3_p, char **ss_p, unsigned int **as_p, const *vrna_md_t* *md)

```
#include <ViennaRNA/sequences/alphabet.h>
```

unsigned int vrna_get_ptype_md(int i, int j, const *vrna_md_t* *md)

```
#include <ViennaRNA/sequences/alphabet.h>
```

unsigned int **vrna_get_ptype**(int ij, char *ptype)

```
#include <ViennaRNA/sequences/alphabet.h>
```

unsigned int **vrna_get_ptype_window**(int i, int j, char **ptype)

```
#include <ViennaRNA/sequences/alphabet.h>
```

vrna_seq_t ***vrna_sequence**(const char *string, unsigned int options)

```
#include <ViennaRNA/sequences/sequence.h>
```

int **vrna_sequence_add**(*vrna_fold_compound_t* *fc, const char *string, unsigned int options)

```
#include <ViennaRNA/sequences/sequence.h>
```

int **vrna_sequence_remove**(*vrna_fold_compound_t* *fc, unsigned int i)

```
#include <ViennaRNA/sequences/sequence.h>
```

void **vrna_sequence_remove_all**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/sequences/sequence.h>
```

void **vrna_sequence_prepare**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/sequences/sequence.h>
```

int **vrna_sequence_order_update**(*vrna_fold_compound_t* *fc, const unsigned int *order)

```
#include <ViennaRNA/sequences/sequence.h>
```

size_t **vrna_sequence_mutate_strand**(*vrna_fold_compound_t* *fc, unsigned int strand, unsigned int *positions, char *nucleotidea, size_t nmods)

```
#include <ViennaRNA/sequences/sequence.h>
```

size_t **vrna_sequence_mutate**(*vrna_fold_compound_t* *fc, const unsigned int *positions, const char *nucleotides, size_t nmut)

```
#include <ViennaRNA/sequences/sequence.h>
```

int **vrna_msa_add**(*vrna_fold_compound_t* *fc, const char **alignment, const char **names, const unsigned char *orientation, const unsigned long long *start, const unsigned long long *genome_size, unsigned int options)

```
#include <ViennaRNA/sequences/sequence.h>
```

struct **vrna_sequence_s**

#include <ViennaRNA/sequences/sequence.h> Data structure representing a nucleotide sequence.

Public Members

vrna_seq_type_e **type**

The type of sequence.

```
#include <ViennaRNA/sequences/sequence.h>
```

char *name

```
#include <ViennaRNA/sequences/sequence.h>
```

char *string

The string representation of the sequence.

```
#include <ViennaRNA/sequences/sequence.h>
```

short *encoding

The integer representation of the sequence.

```
#include <ViennaRNA/sequences/sequence.h>
```

short *encoding5

```
#include <ViennaRNA/sequences/sequence.h>
```

short *encoding3

```
#include <ViennaRNA/sequences/sequence.h>
```

unsigned int length

The length of the sequence.

```
#include <ViennaRNA/sequences/sequence.h>
```

struct vrna_alignment_s

Public Members

unsigned int n_seq

```
#include <ViennaRNA/sequences/sequence.h>
```

vrna_seq_t *sequences

```
#include <ViennaRNA/sequences/sequence.h>
```

char **gapfree_seq

```
#include <ViennaRNA/sequences/sequence.h>
```

unsigned int *gapfree_size

```
#include <ViennaRNA/sequences/sequence.h>
```

unsigned long long ***genome_size**

```
#include <ViennaRNA/sequences/sequence.h>
```

unsigned long long ***start**

```
#include <ViennaRNA/sequences/sequence.h>
```

unsigned char ***orientation**

```
#include <ViennaRNA/sequences/sequence.h>
```

unsigned int ****a2s**

```
#include <ViennaRNA/sequences/sequence.h>
```

4.2.7 Strings and Nucleic Acid Sequences

Functions to parse, convert, manipulate, create, and compare (nucleic acid sequence) strings.

Defines

XSTR(s)

Stringify a macro after expansion.

```
#include <ViennaRNA/utils/strings.h>
```

STR(s)

Stringify a macro argument.

```
#include <ViennaRNA/utils/strings.h>
```

FILENAME_MAX_LENGTH

Maximum length of filenames that are generated by our programs.

```
#include <ViennaRNA/utils/strings.h>
```

This definition should be used throughout the complete ViennaRNA package wherever a static array holding filenames of output files is declared.

FILENAME_ID_LENGTH

Maximum length of id taken from fasta header for filename generation.

```
#include <ViennaRNA/utils/strings.h>
```

this has to be smaller than FILENAME_MAX_LENGTH since in most cases, some suffix will be appended to the ID

VRNA_TRIM_LEADING

Trim only characters leading the string.

```
#include <ViennaRNA/utils/strings.h>
```

See also

vrna_strtrim()

VRNA_TRIM_TRAILING

Trim only characters trailing the string.

```
#include <ViennaRNA/utils/strings.h>
```

See also

vrna_strtrim()

VRNA_TRIM_IN_BETWEEN

Trim only characters within the string.

```
#include <ViennaRNA/utils/strings.h>
```

See also

vrna_strtrim()

VRNA_TRIM_SUBST_BY_FIRST

Replace remaining characters after trimming with the first delimiter in list.

```
#include <ViennaRNA/utils/strings.h>
```

See also

vrna_strtrim()

VRNA_TRIM_DEFAULT

Default settings for trimming, i.e. trim leading and trailing.

```
#include <ViennaRNA/utils/strings.h>
```

See also

vrna_strtrim()

VRNA_TRIM_ALL

Trim characters anywhere in the string.

```
#include <ViennaRNA/utils/strings.h>
```

See also

vrna_strtrim()

Functions

void **vrna_seq_toRNA**(char *sequence)

Convert an input sequence (possibly containing DNA alphabet characters) to RNA alphabet.

```
#include <ViennaRNA/sequences/utils.h>
```

This function substitudes *T* and *t* with *U* and *u*, respectively

Parameters

- **sequence** – The sequence to be converted

char ***vrna_DNA_complement**(const char *sequence)

Retrieve a DNA sequence which resembles the complement of the input sequence.

```
#include <ViennaRNA/sequences/utils.h>
```

This function returns a mew DNA string which is the complement of the input, i.e. the nucleotide letters `A`,`C`,`G`, and `T` are substituted by their complements `T`,`G`,`C`, and `A`, respectively.

Any characters not belonging to the alphabet of the 4 canonical bases of DNA are not altered.

See also

vrna_seq_reverse()

Note

This function also handles lower-case input sequences and treats `U` of the RNA alphabet equally to `T`

Parameters

- **sequence** – the input DNA sequence

Returns

The complement of the input DNA sequence

char *vrna_seq_ungapped(const char *sequence)

Remove gap characters from a nucleotide sequence.

```
#include <ViennaRNA/sequences/utils.h>
```

Parameters

- **sequence** – The original, null-terminated nucleotide sequence

Returns

A copy of the input sequence with all gap characters removed

char *vrna_strdup_printf(const char *format, ...)

Safely create a formatted string.

```
#include <ViennaRNA/utils/strings.h>
```

This function is a safe implementation for creating a formatted character array, similar to *sprintf*. Internally, it uses the *asprintf* function if available to dynamically allocate a large enough character array to store the supplied content. If *asprintf* is not available, mimic it's behavior using *vsnprintf*.

See also

vrna_strdup_vprintf(), *vrna_strcat_printf()*

Note

The returned pointer of this function should always be passed to *free()* to release the allocated memory

Parameters

- **format** – The format string (See also asprintf)
- **...** – The list of variables used to fill the format string

Returns

The formatted, null-terminated string, or NULL if something has gone wrong

char *vrna_strdup_vprintf(const char *format, va_list argp)

Safely create a formatted string.

```
#include <ViennaRNA/utils/strings.h>
```

This function is the *va_list* version of *vrna_strdup_printf()*

See also

vrna_strdup_printf(), *vrna_strcat_printf()*, *vrna_strcat_vprintf()*

Note

The returned pointer of this function should always be passed to *free()* to release the allocated memory

Parameters

- **format** – The format string (See also asprintf)
- **argp** – The list of arguments to fill the format string

Returns

The formatted, null-terminated string, or NULL if something has gone wrong

int **vrna_strcat_printf**(char **dest, const char *format, ...)

Safely append a formatted string to another string.

```
#include <ViennaRNA/utils/strings.h>
```

This function is a safe implementation for appending a formatted character array, similar to a cobination of *strcat* and *sprintf*. The function automatically allocates enough memory to store both, the previous content stored at `dest` and the appended format string. If the `dest` pointer is NULL, the function allocate memory only for the format string. The function returns the number of characters in the resulting string or -1 in case of an error.

See also

vrna_strcat_vprintf(), *vrna_strdup_printf()*, *vrna_strdup_vprintf()*

Parameters

- **dest** – The address of a char *pointer where the formatted string is to be appended
- **format** – The format string (See also sprintf)
- **...** – The list of variables used to fill the format string

Returns

The number of characters in the final string, or -1 on error

int **vrna_strcat_vprintf**(char **dest, const char *format, va_list args)

Safely append a formatted string to another string.

```
#include <ViennaRNA/utils/strings.h>
```

This function is the *va_list* version of *vrna_strcat_printf()*

See also

vrna_strcat_printf(), *vrna_strdup_printf()*, *vrna_strdup_vprintf()*

Parameters

- **dest** – The address of a char *pointer where the formatted string is to be appended
- **format** – The format string (See also sprintf)

- **args** – The list of argument to fill the format string

Returns
The number of characters in the final string, or -1 on error

unsigned int **vrna_strtrim**(char *string, const char *delimiters, unsigned int keep, unsigned int options)

Trim a string by removing (multiple) occurences of a particular character.

```
#include <ViennaRNA/utils/strings.h>
```

This function removes (multiple) consecutive occurences of a set of characters (`delimiters`) within an input string. It may be used to remove leading and/or trailing whitespaces or to restrict the maximum number of consecutive occurences of the delimiting characters `delimiters`. Setting `keep=0` removes all occurences, while other values reduce multiple consecutive occurences to at most `keep` delimiters. This might be useful if one would like to reduce multiple whitespaces to a single one, or to remove empty fields within a comma-separated value string.

The parameter `delimiters` may be a pointer to a 0-terminated char string containing a set of any ASCII character. If *NULL* is passed as delimiter set or an empty char string, all whitespace characters are trimmed. The `options` parameter is a bit vector that specifies which part of the string should undergo trimming. The implementation distinguishes the leading (*VRNA_TRIM_LEADING*), trailing (*VRNA_TRIM_TRAILING*), and in-between (*VRNA_TRIM_IN_BETWEEN*) part with respect to the delimiter set. Combinations of these parts can be specified by using logical-or operator.

The following example code removes all leading and trailing whitespace characters from the input string:

```
charstring[20]  = "  \t blablabla   ";
unsigned int  r           = vrna_strtrim(&(string[0]),
                                         NULL,
                                         0,
VRNA_TRIM_DEFAULT);
```

SWIG Wrapper Notes:

Since many scripting languages treat strings as immutable objects, this function does not modify the input string directly. Instead, it returns the modified string as second return value, together with the number of removed delimiters.

The scripting language interface provides an overloaded version of this function, with default parameters `delimiters=NULL`, `keep=0`, and `options=`*VRNA_TRIM_DEFAULT*. See, e.g. *RNA.strtrim()* in the *The RNA Python module*.

See also

VRNA_TRIM_LEADING, *VRNA_TRIM_TRAILING*, *VRNA_TRIM_IN_BETWEEN*, *VRNA_TRIM_SUBST_BY_FIRST*, *VRNA_TRIM_DEFAULT*, *VRNA_TRIM_ALL*

Note

The delimiter always consists of a single character from the set of characters provided. In case of alternative delimiters and non-null `keep` parameter, the first `keep` delimiters are preserved within the string. Use *VRNA_TRIM_SUBST_BY_FIRST* to substitute all remaining delimiting characters with the first from the `delimiters` list.

Parameters

- **string** – The '\0'-terminated input string to trim
- **delimiters** – The delimiter characters as 0-terminated char array (or *NULL*)
- **keep** – The maximum number of consecutive occurences of the delimiter in the output string
- **options** – The option bit vector specifying the mode of operation

Returns
The number of delimiters removed from the string

char ****vrna_strsplit**(const char *string, const char *delimiter)

Split a string into tokens using a delimiting character.

```
#include <ViennaRNA/utils/strings.h>
```

This function splits a string into an array of strings using a single character that delimits the elements within the string. The default delimiter is the ampersand `'&'` and will be used when `NULL` is passed as a second argument. The returned list is NULL terminated, i.e. the last element is `NULL`. If the delimiter is not found, the returned list contains exactly one element: the input string.

For instance, the following code:

```
char **tok = vrna_strsplit("GGGG&CCCC&AAAAA", NULL);

for (char **ptr = tok; *ptr; ptr++) {
  printf("%s\n", *ptr);
  free(*ptr);
}
free(tok);
```

produces this output:

```
* GGGG
* CCCC
* AAAAA
*
```

and properly free's the memory occupied by the returned element array.

See also

vrna_strtrim()

Note

This function internally uses *strtok_r()* and is therefore considered to be thread-safe. Also note, that it is the users responsibility to free the memory of the array and that of the individual element strings!

In case the input string consists of consecutive delimiters, starts or ends with one or multiple delimiters, empty strings are produced in the output list, indicating the empty fields of data resulting from the split. Use *vrna_strtrim()* prior to a call to this function to remove any leading, trailing, or in-between empty fields.

Parameters

- **string** – The input string that should be split into elements
- **delimiter** – The delimiting character. If NULL, the delimiter is "&"

Returns
A NULL terminated list of the elements in the string

char ***vrna_strjoin**(const char **strings, const char *delimiter)

```
#include <ViennaRNA/utils/strings.h>
```

double ***vrna_str_to_dbl_array**(const char *str, const char *delimiter, unsigned int options)

```
#include <ViennaRNA/utils/strings.h>
```

char ***vrna_random_string**(int l, const char symbols[])

Create a random string using characters from a specified symbol set.

```
#include <ViennaRNA/utils/strings.h>
```

Parameters

- **l** – The length of the sequence
- **symbols** – The symbol set

Returns
A random string of length 'l' containing characters from the symbolset

int **vrna_hamming_distance**(const char *s1, const char *s2)

Calculate hamming distance between two sequences.

```
#include <ViennaRNA/utils/strings.h>
```

Parameters

- **s1** – The first sequence
- **s2** – The second sequence

Returns
The hamming distance between s1 and s2

int **vrna_hamming_distance_bound**(const char *s1, const char *s2, int n)

Calculate hamming distance between two sequences up to a specified length.

```
#include <ViennaRNA/utils/strings.h>
```

This function is similar to *vrna_hamming_distance()* but instead of comparing both sequences up to their actual length only the first 'n' characters are taken into account

Parameters

- **s1** – The first sequence
- **s2** – The second sequence
- **n** – The length of the subsequences to consider (starting from the 5' end)

Returns
The hamming distance between s1 and s2

void **vrna_seq_toupper**(char *sequence)

Convert an input sequence to uppercase.

```
#include <ViennaRNA/utils/strings.h>
```

Parameters

- **sequence** – The sequence to be converted

void **vrna_seq_reverse**(char *sequence)

Reverse a string in-place.

```
#include <ViennaRNA/utils/strings.h>
```

This function reverses a character string in the form of an array of characters in-place, i.e. it changes the input parameter.

See also

vrna_DNA_complement()

Parameters

- **sequence** – The string to reverse

Post

After execution, the input `sequence` consists of the reverse string prior to the execution.

char ***vrna_cut_point_insert**(const char *string, int cp)

Add a separating '&' character into a string according to cut-point position.

```
#include <ViennaRNA/utils/strings.h>
```

If the cut-point position is less or equal to zero, this function just returns a copy of the provided string. Otherwise, the cut-point character is set at the corresponding position

Parameters

- **string** – The original string
- **cp** – The cut-point position

Returns

A copy of the provided string including the cut-point character

char ***vrna_cut_point_remove**(const char *string, int *cp)

Remove a separating '&' character from a string.

```
#include <ViennaRNA/utils/strings.h>
```

This function removes the cut-point indicating '&' character from a string and memorizes its position in a provided integer variable. If not '&' is found in the input, the integer variable is set to -1. The function returns a copy of the input string with the '&' being sliced out.

Parameters

- **string** – The original string
- **cp** – The cut-point position

Returns
A copy of the input string with the '&' being sliced out

size_t ***vrna_strchr**(const char *string, int c, size_t n)

Find (all) occurrences of a character within a string.

```
#include <ViennaRNA/utils/strings.h>
```

string The C string to be scanned

c The character to be searched for

n The maximum number of occurences to search for (or 0 for all occurrences)

Returns
An 1-based array of positions(0-based) or **NULL** on error. Position 0 specifies the number of occurrences found.

4.2.8 Multiple Sequence Alignments

Functions to extract features from and to manipulate multiple sequence alignments (MSA).

Defines

VRNA_ALN_DEFAULT

Use default alignment settings.

```
#include <ViennaRNA/sequences/alignments.h>
```

VRNA_ALN_RNA

Convert to RNA alphabet.

```
#include <ViennaRNA/sequences/alignments.h>
```

VRNA_ALN_DNA

Convert to DNA alphabet.

```
#include <ViennaRNA/sequences/alignments.h>
```

VRNA_ALN_UPPERCASE

Convert to uppercase nucleotide letters.

```
#include <ViennaRNA/sequences/alignments.h>
```

VRNA_ALN_LOWERCASE

Convert to lowercase nucleotide letters.

```
#include <ViennaRNA/sequences/alignments.h>
```

VRNA_MEASURE_SHANNON_ENTROPY

Flag indicating Shannon Entropy measure.

```
#include <ViennaRNA/sequences/alignments.h>
```

Shannon Entropy is defined as $H = -\sum_c p_c \cdot \log_2 p_c$

Typedefs

typedef struct *vrna_pinfo_s* **vrna_pinfo_t**

Typename for the base pair info repesenting data structure *vrna_pinfo_s*.

```
#include <ViennaRNA/sequences/alignments.h>
```

Functions

int **vrna_aln_mpi**(const char **alignment)

Get the mean pairwise identity in steps from ?to?(ident).

```
#include <ViennaRNA/sequences/alignments.h>
```

SWIG Wrapper Notes:

This function is available as function `aln_mpi()`. See e.g. *RNA.aln_mpi()* in the *The RNA Python module*.

Parameters

- **alignment** – Aligned sequences

Returns

The mean pairwise identity

vrna_pinfo_t ***vrna_aln_pinfo**(*vrna_fold_compound_t* *fc, const char *structure, double threshold)

Retrieve an array of *vrna_pinfo_t* structures from precomputed pair probabilities.

```
#include <ViennaRNA/sequences/alignments.h>
```

This array of structures contains information about positionwise pair probabilies, base pair entropy and more

See also

vrna_pinfo_t, and *vrna_pf()*

Parameters

- **fc** – The *vrna_fold_compound_t* of type *VRNA_FC_TYPE_COMPARATIVE* with pre-computed partition function matrices
- **structure** – An optional structure in dot-bracket notation (Maybe NULL)
- **threshold** – Do not include results with pair probabilities below threshold

Returns

The *vrna_pinfo_t* array

int ***vrna_aln_pscore**(const char **alignment, *vrna_md_t* *md)

```
#include <ViennaRNA/sequences/alignments.h>
```

SWIG Wrapper Notes:

This function is available as overloaded function `aln_pscore()` where the last parameter may be omitted, indicating `md = NULL`. See e.g. *`RNA.aln_pscore()`* in the *The RNA Python module*.

int **vrna_pscore**(*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j)

```
#include <ViennaRNA/sequences/alignments.h>
```

int **vrna_pscore_freq**(*vrna_fold_compound_t* *fc, const unsigned int *frequencies, unsigned int pairs)

```
#include <ViennaRNA/sequences/alignments.h>
```

char ***vrna_aln_slice**(const char **alignment, unsigned int i, unsigned int j)

Slice out a subalignment from a larger alignment.

```
#include <ViennaRNA/sequences/alignments.h>
```

See also

vrna_aln_free()

Note

The user is responsible to free the memory occupied by the returned subalignment

Parameters

- **alignment** – The input alignment
- **i** – The first column of the subalignment (1-based)
- **j** – The last column of the subalignment (1-based)

Returns

The subalignment between column i and j

void **vrna_aln_free**(char **alignment)

Free memory occupied by a set of aligned sequences.

```
#include <ViennaRNA/sequences/alignments.h>
```

Parameters

- **alignment** – The input alignment

char ***vrna_aln_uppercase**(const char **alignment)

Create a copy of an alignment with only uppercase letters in the sequences.

```
#include <ViennaRNA/sequences/alignments.h>
```

See also

vrna_aln_copy

Parameters

- **alignment** – The input sequence alignment (last entry must be *NULL* terminated)

Returns

A copy of the input alignment where lowercase sequence letters are replaced by uppercase letters

char **__vrna_aln_toRNA__(const char **alignment)

Create a copy of an alignment where DNA alphabet is replaced by RNA alphabet.

```
#include <ViennaRNA/sequences/alignments.h>
```

See also

vrna_aln_copy

Parameters

- **alignment** – The input sequence alignment (last entry must be *NULL* terminated)

Returns

A copy of the input alignment where DNA alphabet is replaced by RNA alphabet (T -> U)

char **__vrna_aln_copy__(const char **alignment, unsigned int options)

Make a copy of a multiple sequence alignment.

```
#include <ViennaRNA/sequences/alignments.h>
```

This function allows one to create a copy of a multiple sequence alignment. The `options` parameter additionally allows for sequence manipulation, such as converting DNA to RNA alphabet, and conversion to uppercase letters.

See also

vrna_aln_copy(), *VRNA_ALN_RNA*, *VRNA_ALN_UPPERCASE*, *VRNA_ALN_DEFAULT*

Parameters

- **alignment** – The input sequence alignment (last entry must be *NULL* terminated)
- **options** – Option flags indicating whether the aligned sequences should be converted

Returns

A (manipulated) copy of the input alignment

float *__vrna_aln_conservation_struct__(const char **alignment, const char *structure, const *vrna_md_t* *md)

Compute base pair conservation of a consensus structure.

```
#include <ViennaRNA/sequences/alignments.h>
```

This function computes the base pair conservation (fraction of canonical base pairs) of a consensus structure given a multiple sequence alignment. The base pair types that are considered canonical may be specified using the *vrna_md_t.pair* array. Passing *NULL* as parameter `md` results in default pairing rules, i.e. canonical Watson-Crick and GU Wobble pairs.

SWIG Wrapper Notes:

This function is available as overloaded function `aln_conservation_struct()` where the last parameter `md` may be omitted, indicating `md = NULL`. See, e.g. *RNA.aln_conservation_struct()* in the *The RNA Python module*.

Parameters

- **alignment** – The input sequence alignment (last entry must be *NULL* terminated)
- **structure** – The consensus structure in dot-bracket notation
- **md** – Model details that specify compatible base pairs (Maybe *NULL*)

Returns

A 1-based vector of base pair conservations

float ***vrna_aln_conservation_col**(const char **alignment, const *vrna_md_t* *md_p, unsigned int options)

Compute nucleotide conservation in an alignment.

```
#include <ViennaRNA/sequences/alignments.h>
```

This function computes the conservation of nucleotides in alignment columns. The simples measure is Shannon Entropy and can be selected by passing the *VRNA_MEASURE_SHANNON_ENTROPY* flag in the `options` parameter.

SWIG Wrapper Notes:

This function is available as overloaded function `aln_conservation_col()` where the last two parameters may be omitted, indicating `md = NULL`, and `options = VRNA_MEASURE_SHANNON_ENTROPY`, respectively. See e.g. *RNA.aln_conservation_col()* in the *The RNA Python module*.

See also

VRNA_MEASURE_SHANNON_ENTROPY

Note

Currently, only *VRNA_MEASURE_SHANNON_ENTROPY* is supported as conservation measure.

Parameters

- **alignment** – The input sequence alignment (last entry must be *NULL* terminated)
- **md** – Model details that specify known nucleotides (Maybe *NULL*)
- **options** – A flag indicating which measure of conservation should be applied

Returns

A 1-based vector of column conservations

char *vrna_aln_consensus_sequence(const char **alignment, const *vrna_md_t* *md_p)

Compute the consensus sequence for a given multiple sequence alignment.

```
#include <ViennaRNA/sequences/alignments.h>
```

SWIG Wrapper Notes:

This function is available as overloaded function `aln_consensus_sequence()` where the last parameter may be omitted, indicating `md = NULL`. See e.g. *RNA.aln_consensus_sequence()* in the *The RNA Python module*.

Parameters

- **alignment** – The input sequence alignment (last entry must be *NULL* terminated)
- **md_p** – Model details that specify known nucleotides (Maybe *NULL*)

Returns

The consensus sequence of the alignment, i.e. the most frequent nucleotide for each alignment column

char *vrna_aln_consensus_mis(const char **alignment, const *vrna_md_t* *md_p)

Compute the Most Informative Sequence (MIS) for a given multiple sequence alignment.

```
#include <ViennaRNA/sequences/alignments.h>
```

The most informative sequence (MIS) [Freyhult *et al.*, 2005] displays for each alignment column the nucleotides with frequency greater than the background frequency, projected into IUPAC notation. Columns where gaps are over-represented are in lower case.

SWIG Wrapper Notes:

This function is available as overloaded function `aln_consensus_mis()` where the last parameter may be omitted, indicating `md = NULL`. See e.g. *RNA.aln_consensus_mis()* in the *The RNA Python module*.

Parameters

- **alignment** – The input sequence alignment (last entry must be *NULL* terminated)
- **md_p** – Model details that specify known nucleotides (Maybe *NULL*)

Returns

The most informative sequence for the alignment

struct vrna_pinfo_s

#include <ViennaRNA/sequences/alignments.h> A base pair info structure.

For each base pair (i,j) with i,j in [0, n-1] the structure lists:

- its probability 'p'
- an entropy-like measure for its well-definedness 'ent'
- the frequency of each type of pair in 'bp[]'
  - 'bp[0]' contains the number of non-compatible sequences
  - 'bp[1]' the number of CG pairs, etc.

Public Members

unsigned **i**

nucleotide position i

```
#include <ViennaRNA/sequences/alignments.h>
```

unsigned **j**

nucleotide position j

```
#include <ViennaRNA/sequences/alignments.h>
```

float **p**

Probability.

```
#include <ViennaRNA/sequences/alignments.h>
```

float **ent**

Pseudo entropy for $p(i,j) = S_i + S_j - p_ij * ln(p_ij)$.

```
#include <ViennaRNA/sequences/alignments.h>
```

short **bp**[8]

Frequencies of pair_types.

```
#include <ViennaRNA/sequences/alignments.h>
```

char **comp**

1 iff pair is in mfe structure

```
#include <ViennaRNA/sequences/alignments.h>
```

4.2.9 RNA Secondary Structures

Functions to create, parse, convert, manipulate, and compare secondary structure representations.

Dot-Bracket Notation

Defines

VRNA_BRACKETS_ALPHA

Bitflag to indicate secondary structure notations using uppercase/lowercase letters from the latin alphabet.

```
#include <ViennaRNA/structures/dotbracket.h>
```

See also

vrna_ptable_from_string()

VRNA_BRACKETS_RND

Bitflag to indicate secondary structure notations using round brackets (parenthesis), ().

```
#include <ViennaRNA/structures/dotbracket.h>
```

See also

vrna_ptable_from_string(), *vrna_db_flatten()*, *vrna_db_flatten_to()*

VRNA_BRACKETS_CLY

Bitflag to indicate secondary structure notations using curly brackets, {}.

```
#include <ViennaRNA/structures/dotbracket.h>
```

See also

vrna_ptable_from_string(), *vrna_db_flatten()*, *vrna_db_flatten_to()*

VRNA_BRACKETS_ANG

Bitflag to indicate secondary structure notations using angular brackets, <>.

```
#include <ViennaRNA/structures/dotbracket.h>
```

See also

vrna_ptable_from_string(), *vrna_db_flatten()*, *vrna_db_flatten_to()*

VRNA_BRACKETS_SQR

Bitflag to indicate secondary structure notations using square brackets, [].

```
#include <ViennaRNA/structures/dotbracket.h>
```

See also

vrna_ptable_from_string(), *vrna_db_flatten()*, *vrna_db_flatten_to()*

VRNA_BRACKETS_DEFAULT

Default bitmask to indicate secondary structure notation using any pair of brackets.

```
#include <ViennaRNA/structures/dotbracket.h>
```

This set of matching brackets/parenthesis is always nested, i.e. pseudo-knot free, in WUSS format. However, in general different kinds of brackets are mostly used for annotating pseudo-knots. Thus special care has to be taken to remove pseudo-knots if this bitmask is used in functions that return secondary structures without pseudo-knots!

See also

vrna_ptable_from_string(), *vrna_db_flatten()*, *vrna_db_flatten_to()*, *vrna_db_pk_remove()* *vrna_pt_pk_remove()*

VRNA_BRACKETS_ANY

Bitmask to indicate secondary structure notation using any pair of brackets or uppercase/lowercase alphabet letters.

```
#include <ViennaRNA/structures/dotbracket.h>
```

See also

vrna_ptable_from_string(), *vrna_db_pk_remove()*, *vrna_db_flatten()*, *vrna_db_flatten_to()*

VRNA_GQUAD_DB_SYMBOL

```
#include <ViennaRNA/structures/dotbracket.h>
```

VRNA_GQUAD_DB_SYMBOL_END

```
#include <ViennaRNA/structures/dotbracket.h>
```

Functions

char ***vrna_db_pack**(const char *struc)

Pack secondary secondary structure, 5:1 compression using base 3 encoding.

```
#include <ViennaRNA/structures/dotbracket.h>
```

Returns a binary string encoding of the secondary structure using a 5:1 compression scheme. The string is NULL terminated and can therefore be used with standard string functions such as strcmp(). Useful for programs that need to keep many structures in memory.

See also

vrna_db_unpack()

Parameters

- **struc** – The secondary structure in dot-bracket notation

Returns
The binary encoded structure

char ***vrna_db_unpack**(const char *packed)

Unpack secondary structure previously packed with *vrna_db_pack()*.

```
#include <ViennaRNA/structures/dotbracket.h>
```

Translate a compressed binary string produced by *vrna_db_pack()* back into the familiar dot-bracket notation.

See also

vrna_db_pack()

Parameters

- **packed** – The binary encoded packed secondary structure

Returns
The unpacked secondary structure in dot-bracket notation

void **vrna_db_flatten**(char *structure, unsigned int options)

Substitute pairs of brackets in a string with parenthesis.

```
#include <ViennaRNA/structures/dotbracket.h>
```

This function can be used to replace brackets of unusual types, such as angular brackets <> , to dot-bracket format. The `options` parameter is used tpo specify which types of brackets will be replaced by round parenthesis () .

SWIG Wrapper Notes:

This function flattens an input structure string in-place! The second parameter is optional and defaults to *VRNA_BRACKETS_DEFAULT*.

An overloaded version of this function exists, where an additional second parameter can be passed to specify the target brackets, i.e. the type of matching pair characters all brackets will be flattened to. Therefore, in the scripting language interface this function is a replacement for *vrna_db_flatten_to()*. See, e.g. *`RNA.db_flatten()`* in the *The RNA Python module*.

See also

vrna_db_flatten_to(), *VRNA_BRACKETS_RND*, *VRNA_BRACKETS_ANG*, *VRNA_BRACKETS_CLY*, *VRNA_BRACKETS_SQR*, *VRNA_BRACKETS_DEFAULT*

Parameters

- **structure** – The structure string where brackets are flattened in-place
- **options** – A bitmask to specify which types of brackets should be flattened out

void **vrna_db_flatten_to**(char *string, const char target[3], unsigned int options)

Substitute pairs of brackets in a string with another type of pair characters.

```
#include <ViennaRNA/structures/dotbracket.h>
```

This function can be used to replace brackets in a structure annotation string, such as square brackets [] , to another type of pair characters, e.g. angular brackets <> .

The `target` array must contain a character for the 'pair open' annotation at position 0, and one for 'pair close' at position 1. `Toptions` parameter is used to specify which types of brackets will be replaced by the new pairs.

SWIG Wrapper Notes:

This function is available as an overloaded version of *vrna_db_flatten()*. See, e.g. *RNA.db_flatten()* in the *The RNA Python module*.

See also

vrna_db_flatten(), *VRNA_BRACKETS_RND*, *VRNA_BRACKETS_ANG*, *VRNA_BRACKETS_CLY*, *VRNA_BRACKETS_SQR*, *VRNA_BRACKETS_DEFAULT*

Parameters

- **string** – The structure string where brackets are flattened in-place
- **target** – The new pair characters the string will be flattened to
- **options** – A bitmask to specify which types of brackets should be flattened out

char ***vrna_db_from_ptable**(const short *pt)

Convert a pair table into dot-parenthesis notation.

```
#include <ViennaRNA/structures/dotbracket.h>
```

This function also converts pair table formatted structures that contain pseudoknots. Non-nested base pairs result in additional pairs of parenthesis and brackets within the resulting dot-parenthesis string. The following pairs are available: (), []. {}. <>, as well as pairs of matching upper-/lower-case characters from the alphabet A-Z.

Note

In cases where the level of non-nested base pairs exceeds the maximum number of 30 different base pair indicators (4 parenthesis/brackets, 26 matching characters), a warning is printed and the remaining base pairs are left out from the conversion.

Parameters

- **pt** – The pair table to be copied

Returns

A char pointer to the dot-bracket string

char ***vrna_db_from_plist**(*vrna_ep_t* *pairs, unsigned int n)

Convert a list of base pairs into dot-bracket notation.

```
#include <ViennaRNA/structures/dotbracket.h>
```

See also

vrna_plist()

Parameters

- **pairs** – A *vrna_ep_t* containing the pairs to be included in the dot-bracket string
- **n** – The length of the structure (number of nucleotides)

Returns

The dot-bracket string containing the provided base pairs

char ***vrna_db_to_element_string**(const char *structure)

Convert a secondary structure in dot-bracket notation to a nucleotide annotation of loop contexts.

```
#include <ViennaRNA/structures/dotbracket.h>
```

Parameters

- **structure** – The secondary structure in dot-bracket notation

Returns

A string annotating each nucleotide according to it's structural context

char ***vrna_db_pk_remove**(const char *structure, unsigned int options)

Remove pseudo-knots from an input structure.

```
#include <ViennaRNA/structures/dotbracket.h>
```

This function removes pseudo-knots from an input structure by determining the minimum number of base pairs that need to be removed to make the structure pseudo-knot free.

To accomplish that, we use a dynamic programming algorithm similar to the Nussinov maxmimum matching approach.

The input structure must be in a dot-bracket string like form where crossing base pairs are denoted by the use of additional types of matching brackets, e.g. <>, {}, [], {}. Furthermore, crossing pairs may be annotated by matching uppercase/lowercase letters from the alphabet A-Z. For the latter, the uppercase letter must be the 5' and the lowercase letter the 3' nucleotide of the base pair. The actual type of brackets to be recognized by this function must be specifed through the options parameter.

SWIG Wrapper Notes:

This function is available as an overloaded function db_pk_remove() where the optional second parameter options defaults to *VRNA_BRACKETS_ANY*. See, e.g. *RNA.db_pk_remove()* in the *The RNA Python module*.

See also

vrna_pt_pk_remove(), *vrna_db_flatten()*, *VRNA_BRACKETS_RND*, *VRNA_BRACKETS_ANG*, *VRNA_BRACKETS_CLY*, *VRNA_BRACKETS_SQR*, *VRNA_BRACKETS_ALPHA*, *VRNA_BRACKETS_DEFAULT*, *VRNA_BRACKETS_ANY*

Note

Brackets in the input structure string that are not covered by the `options` bitmask will be silently ignored!

Parameters

- **`structure`** – Input structure in dot-bracket format that may include pseudo-knots
- **`options`** – A bitmask to specify which types of brackets should be processed

Returns

The input structure devoid of pseudo-knots in dot-bracket notation

WUSS Notation

Washington University Secondary Structure (WUSS) notation related functions

Functions

char ***vrna_db_from_WUSS**(const char *wuss)

Convert a WUSS annotation string to dot-bracket format.

```
#include <ViennaRNA/structures/dotbracket.h>
```

Note

This function flattens all brackets, and treats pseudo-knots annotated by matching pairs of upper/lowercase letters as unpaired nucleotides

Parameters

- **`wuss`** – The input string in WUSS notation

Returns

A dot-bracket notation of the input secondary structure

Pair Tables

Pair Table Representation of Secondary Structures

Functions

short ***vrna_ptable**(const char *structure)

Create a pair table from a *Dot-Bracket Notation* of a secondary structure.

```
#include <ViennaRNA/structures/pairtable.h>
```

Returns a newly allocated table, such that table[i]=j if (i.j) pair or 0 if i is unpaired, table[0] contains the length of the structure.

SWIG Wrapper Notes:

This functions is wrapped as overloaded function `ptable()` that takes an optional argument `options` to specify which type of matching brackets should be considered during conversion. The default set is round brackets, i.e. *VRNA_BRACKETS_RND*. See, e.g. *RNA.ptable()* in the *The RNA Python module*.

See also

vrna_ptable_from_string(), *vrna_db_from_ptable()*

Parameters

- **structure** – The secondary structure in dot-bracket notation

Returns

A pointer to the created pair_table

short ***vrna_ptable_from_string**(const char *structure, unsigned int options)

Create a pair table for a secondary structure string.

```
#include <ViennaRNA/structures/pairtable.h>
```

This function takes an input string of a secondary structure annotation in *Dot-Bracket Notation* or *Extended Dot-Bracket Notation* , and converts it into a pair table representation.

SWIG Wrapper Notes:

This functions is wrapped as overloaded function `ptable()` that takes an optional argument `options` to specify which type of matching brackets should be considered during conversion. The default set is round brackets, i.e. *VRNA_BRACKETS_RND*. See, e.g. *RNA.ptable()* in the *The RNA Python module*.

See also

vrna_ptable(), *vrna_db_from_ptable()*, *vrna_db_flatten_to()*, *vrna_pt_pk_remove()* *VRNA_BRACKETS_RND*, *VRNA_BRACKETS_ANG*, *VRNA_BRACKETS_CLY*, *VRNA_BRACKETS_SQR*, *VRNA_BRACKETS_ALPHA*, *VRNA_BRACKETS_DEFAULT*, *VRNA_BRACKETS_ANY*

Note

This function also extracts crossing base pairs, i.e. pseudo-knots if more than a single matching bracket type is allowed through the bitmask `options`.

Parameters

- **structure** – Secondary structure in *Extended Dot-Bracket Notation*
- **options** – A bitmask to specify which brackets are recognized during conversion to pair table

Returns
A pointer to a new pair table of the provided secondary structure

short *vrna_pt_pk_get(const char *structure)

Create a pair table of a secondary structure (pseudo-knot version).

```
#include <ViennaRNA/structures/pairtable.h>
```

Returns a newly allocated table, such that table[i]=j if (i.j) pair or 0 if i is unpaired, table[0] contains the length of the structure.

In contrast to *vrna_ptable()* this function also recognizes the base pairs denoted by '[' and ']' brackets. Thus, this function behaves like

```
vrna_ptable_from_string(structure, VRNA_BRACKETS_RND | VRNA_BRACKETS_SQR)
```

See also

vrna_ptable_from_string()

Parameters

- **structure** – The secondary structure in (extended) dot-bracket notation

Returns
A pointer to the created pair_table

short *vrna_ptable_copy(const short *pt)

Get an exact copy of a pair table.

```
#include <ViennaRNA/structures/pairtable.h>
```

Parameters

- **pt** – The pair table to be copied

Returns
A pointer to the copy of 'pt'

short *vrna_pt_ali_get(const char *structure)

Create a pair table of a secondary structure (snoop align version).

```
#include <ViennaRNA/structures/pairtable.h>
```

short *vrna_pt_snoop_get(const char *structure)

Create a pair table of a secondary structure (snoop version).

```
#include <ViennaRNA/structures/pairtable.h>
```

returns a newly allocated table, such that: table[i]=j if (i.j) pair or 0 if i is unpaired, table[0] contains the length of the structure. The special pseudoknotted H/ACA-mRNA structure is taken into account.

short ***vrna_pt_pk_remove**(const short *ptable, unsigned int options)

Remove pseudo-knots from a pair table.

```
#include <ViennaRNA/structures/pairtable.h>
```

This function removes pseudo-knots from an input structure by determining the minimum number of base pairs that need to be removed to make the structure pseudo-knot free.

To accomplish that, we use a dynamic programming algorithm similar to the Nussinov maxmimum matching approach.

See also

vrna_db_pk_remove()

Parameters

- **ptable** – Input structure that may include pseudo-knots
- **options**

Returns

The input structure devoid of pseudo-knots

unsigned int **vrna_ptable_compat_remove**(short *pt, const short *pt_ref)

Remove base pairs compatible with a reference structure.

```
#include <ViennaRNA/structures/pairtable.h>
```

This function compares two structures (`pt` and `pt_ref`) of the same length in pair table format and removes all base pairs in `pt` that are not present in the reference `pt_ref` as long as they are compatible with `pt_ref`.

This is useful in cases where one wants to score the difference between two structures but not account for additional compatible base pairs, e.g. during benchmarking.

See also

vrna_compare_structure_pt()

Parameters

- **pt** – The structure in pair table format that might undergo base pair removal
- **pt_ref** – The reference structure in pair table format

Returns

The number of base pairs removed from `pt`

unsigned int **vrna_ptable_count_bp**(const short *pt)

```
#include <ViennaRNA/structures/pairtable.h>
```

unsigned int **vrna_ptable_count_up**(const short *pt)

```
#include <ViennaRNA/structures/pairtable.h>
```

Pair Lists

Pair List Representation of Secondary Structures

Defines

VRNA_PLIST_TYPE_BASEPAIR

A Base Pair element.

```
#include <ViennaRNA/structures/problist.h>
```

VRNA_PLIST_TYPE_GQUAD

A G-Quadruplex element.

```
#include <ViennaRNA/structures/problist.h>
```

VRNA_PLIST_TYPE_H_MOTIF

A Hairpin loop motif element.

```
#include <ViennaRNA/structures/problist.h>
```

VRNA_PLIST_TYPE_I_MOTIF

An Internal loop motif element.

```
#include <ViennaRNA/structures/problist.h>
```

VRNA_PLIST_TYPE_UD_MOTIF

An Unstructured Domain motif element.

```
#include <ViennaRNA/structures/problist.h>
```

VRNA_PLIST_TYPE_STACK

A Base Pair stack element.

```
#include <ViennaRNA/structures/problist.h>
```

VRNA_PLIST_TYPE_UNPAIRED

An unpaired base or segment.

```
#include <ViennaRNA/structures/problist.h>
```

VRNA_PLIST_TYPE_TRIPLE

One pair of a base triplet.

```
#include <ViennaRNA/structures/problist.h>
```

VRNA_PLIST_TYPE_HELIX

A Helix element.

```
#include <ViennaRNA/structures/problist.h>
```

Typedefs

typedef struct *vrna_elem_prob_s* **vrna_ep_t**

Convenience typedef for data structure *vrna_elem_prob_s*.

```
#include <ViennaRNA/structures/problist.h>
```

typedef *vrna_ep_t* ***vrna_ep_array_t**

Convenience typedef for *vrna_array(vrna_ep_t)*.

```
#include <ViennaRNA/structures/problist.h>
```

Note

Get the number of elements in the array via *vrna_array_size()* and append elements via the *vrna_array_append()* macro

Warning

Use the *vrna_array_free()* macro to release memory for data of this type!

Functions

vrna_ep_t ***vrna_plist**(const char *struc, float pr)

Create a *vrna_ep_t* from a dot-bracket string.

```
#include <ViennaRNA/structures/problist.h>
```

The dot-bracket string is parsed and for each base pair an entry in the plist is created. The probability of each pair in the list is set by a function parameter.

The end of the plist is marked by sequence positions i as well as j equal to 0. This condition should be used to stop looping over its entries

Parameters

- **struc** – The secondary structure in dot-bracket notation
- **pr** – The probability for each base pair used in the plist

Returns

The plist array

int **vrna_plist_append**(*vrna_ep_t* **target, const *vrna_ep_t* *list)

```
#include <ViennaRNA/structures/problist.h>
```

struct **vrna_elem_prob_s**

#include <ViennaRNA/structures/problist.h> Data structure representing a single entry of an element probability list (e.g. list of pair probabilities).

VRNA_PLIST_TYPE_BASEPAIR, *VRNA_PLIST_TYPE_GQUAD*, *VRNA_PLIST_TYPE_H_MOTIF*, *VRNA_PLIST_TYPE_I_MOTIF*, *VRNA_PLIST_TYPE_UD_MOTIF*, *VRNA_PLIST_TYPE_STACK*

See also

vrna_plist(), *vrna_plist_from_probs()*, *vrna_db_from_plist()*,

Public Members

int **i**

Start position (usually 5' nucleotide that starts the element, e.g. base pair).

```
#include <ViennaRNA/structures/problist.h>
```

int **j**

End position (usually 3' nucleotide that ends the element, e.g. base pair).

```
#include <ViennaRNA/structures/problist.h>
```

float **p**

Probability of the element.

```
#include <ViennaRNA/structures/problist.h>
```

int **type**

Type of the element.

```
#include <ViennaRNA/structures/problist.h>
```

unsigned int **size**

The size of the element (e.g. number of base pairs that constitute a helix).

```
#include <ViennaRNA/structures/problist.h>
```

Abstract Shapes

Abstract Shapes Representation of Secondary Structures

Functions

char ***vrna_abstract_shapes**(const char *structure, unsigned int level)

Convert a secondary structure in dot-bracket notation to its abstract shapes representation.

```
#include <ViennaRNA/structures/shapes.h>
```

This function converts a secondary structure into its abstract shapes representation as presented by Giegerich *et al.* [2004] .

SWIG Wrapper Notes:

This function is available as an overloaded function `abstract_shapes()` where the optional second parameter `level` defaults to 5. See, e.g. *`RNA.abstract_shapes()`* in the *The RNA Python module*.

See also

vrna_abstract_shapes_pt()

Parameters

- **structure** – A secondary structure in dot-bracket notation
- **level** – The abstraction level (integer in the range of 0 to 5)

Returns

The secondary structure in abstract shapes notation

char ***vrna_abstract_shapes_pt**(const short *pt, unsigned int level)

Convert a secondary structure to its abstract shapes representation.

```
#include <ViennaRNA/structures/shapes.h>
```

This function converts a secondary structure into its abstract shapes representation as presented by Giegerich *et al.* [2004] . This function is equivalent to vrna_db_to_shapes(), but requires a pair table input instead of a dot-bracket structure.

SWIG Wrapper Notes:

This function is available as an overloaded function `abstract_shapes()` where the optional second parameter `level` defaults to 5. See, e.g. *`RNA.abstract_shapes()`* in the *The RNA Python module*.

See also

vrna_abstract_shapes()

Note

The length of the structure must be present at `pt`[0]!

Parameters

- **pt** – A secondary structure in pair table format
- **level** – The abstraction level (integer in the range of 0 to 5)

Returns

The secondary structure in abstract shapes notation

Helix Lists

Helix List Representation of Secondary Structures

typedef struct *vrna_hx_s* **vrna_hx_t**

Convenience typedef for data structure *vrna_hx_s*.

```
#include <ViennaRNA/structures/helix.h>
```

vrna_hx_t ***vrna_hx_from_ptable**(short *pt)

Convert a pair table representation of a secondary structure into a helix list.

```
#include <ViennaRNA/structures/helix.h>
```

Parameters

- **pt** – The secondary structure in pair table representation

Returns
The secondary structure represented as a helix list

vrna_hx_t ***vrna_hx_merge**(const *vrna_hx_t* *list, int maxdist)
Create a merged helix list from another helix list.

```
#include <ViennaRNA/structures/helix.h>
```

struct **vrna_hx_s**
#include <ViennaRNA/structures/helix.h> Data structure representing an entry of a helix list.

Public Members

unsigned int **start**

```
#include <ViennaRNA/structures/helix.h>
```

unsigned int **end**

```
#include <ViennaRNA/structures/helix.h>
```

unsigned int **length**

```
#include <ViennaRNA/structures/helix.h>
```

unsigned int **up5**

```
#include <ViennaRNA/structures/helix.h>
```

unsigned int **up3**

```
#include <ViennaRNA/structures/helix.h>
```

Tree Representation

Tree Representation of Secondary Structures

Defines

VRNA_STRUCTURE_TREE_HIT
Homeomorphically Irreducible Tree (HIT) representation of a secondary structure.

```
#include <ViennaRNA/structures/tree.h>
```

See also

vrna_db_to_tree_string()

VRNA_STRUCTURE_TREE_SHAPIRO_SHORT

(short) Coarse Grained representation of a secondary structure

```
#include <ViennaRNA/structures/tree.h>
```

See also

vrna_db_to_tree_string()

VRNA_STRUCTURE_TREE_SHAPIRO

(full) Coarse Grained representation of a secondary structure

```
#include <ViennaRNA/structures/tree.h>
```

See also

vrna_db_to_tree_string()

VRNA_STRUCTURE_TREE_SHAPIRO_EXT

(extended) Coarse Grained representation of a secondary structure

```
#include <ViennaRNA/structures/tree.h>
```

See also

vrna_db_to_tree_string()

VRNA_STRUCTURE_TREE_SHAPIRO_WEIGHT

(weighted) Coarse Grained representation of a secondary structure

```
#include <ViennaRNA/structures/tree.h>
```

See also

vrna_db_to_tree_string()

VRNA_STRUCTURE_TREE_EXPANDED

Expanded Tree representation of a secondary structure.

```
#include <ViennaRNA/structures/tree.h>
```

See also

vrna_db_to_tree_string()

Functions

char ***vrna_db_to_tree_string**(const char *structure, unsigned int type)

Convert a Dot-Bracket structure string into tree string representation.

```
#include <ViennaRNA/structures/tree.h>
```

This function allows one to convert a secondary structure in dot-bracket notation into one of the various tree representations for secondary structures. The resulting tree is then represented as a string of parenthesis and node symbols, similar to to the Newick format.

Currently we support conversion into the following formats, denoted by the value of parameter `type`:

- *VRNA_STRUCTURE_TREE_HIT* - Homeomorphically Irreducible Tree (HIT) representation of a secondary structure. (See also Fontana *et al.* [1993])
- *VRNA_STRUCTURE_TREE_SHAPIRO_SHORT* - (short) Coarse Grained representation of a secondary structure (same as Shapiro [1988] , but with root node `R` and without `S` nodes for the stems)
- *VRNA_STRUCTURE_TREE_SHAPIRO* - (full) Coarse Grained representation of a secondary structure (See also Shapiro [1988])
- *VRNA_STRUCTURE_TREE_SHAPIRO_EXT* - (extended) Coarse Grained representation of a secondary structure (same as Shapiro [1988] , but external nodes denoted as `E`)
- *VRNA_STRUCTURE_TREE_SHAPIRO_WEIGHT* - (weighted) Coarse Grained representation of a secondary structure (same as *VRNA_STRUCTURE_TREE_SHAPIRO_EXT* but with additional weights for number of unpaired nucleotides in loop, and number of pairs in stems)
- *VRNA_STRUCTURE_TREE_EXPANDED* - Expanded Tree representation of a secondary structure.

See also

structure-tree-representation

Parameters

- **structure** – The null-terminated dot-bracket structure string
- **type** – A switch to determine the type of tree string representation

Returns

A tree representation of the input `structure`

char *vrna_tree_string_unweight(const char *structure)

Remove weights from a linear string tree representation of a secondary structure.

```
#include <ViennaRNA/structures/tree.h>
```

This function strips the weights of a linear string tree representation such as HIT, or Coarse Grained Tree sensu Shapiro [1988]

See also

vrna_db_to_tree_string()

Parameters

- **structure** – A linear string tree representation of a secondary structure with weights

Returns

A linear string tree representation of a secondary structure without weights

char *vrna_tree_string_to_db(const char *tree)

Convert a linear tree string representation of a secondary structure back to Dot-Bracket notation.

```
#include <ViennaRNA/structures/tree.h>
```

See also

vrna_db_to_tree_string(), *VRNA_STRUCTURE_TREE_EXPANDED*, *VRNA_STRUCTURE_TREE_HIT*, structure-tree-representation

Warning

This function only accepts *Expanded* and *HIT* tree representations!

Parameters

- **tree** – A linear tree string representation of a secondary structure

Returns

A dot-bracket notation of the secondary structure provided in tree

Distance measures

Distance measures between Secondary Structures

Functions

int vrna_bp_distance_pt(const short *pt1, const short *pt2)

Compute the "base pair" distance between two pair tables pt1 and pt2 of secondary structures.

```
#include <ViennaRNA/structures/metrics.h>
```

The pair tables should have the same length. dist = number of base pairs in one structure but not in the other same as edit distance with open-pair close-pair as move-set

SWIG Wrapper Notes:

This function is available as an overloaded method **bp_distance()**. See, e.g. *RNA.bp_distance()* in the *The RNA Python module*.

See also

vrna_bp_distance()

Parameters

- **pt1** – First structure in dot-bracket notation
- **pt2** – Second structure in dot-bracket notation

Returns

The base pair distance between pt1 and pt2

int **vrna_bp_distance**(const char *str1, const char *str2)

Compute the "base pair" distance between two secondary structures s1 and s2.

```
#include <ViennaRNA/structures/metrics.h>
```

This is a wrapper around ***vrna_bp_distance_pt()***. The sequences should have the same length. dist = number of base pairs in one structure but not in the other same as edit distance with open-pair close-pair as move-set

SWIG Wrapper Notes:

This function is available as an overloaded method **bp_distance()**. Note that the SWIG wrapper takes two structure in dot-bracket notation and converts them into pair tables using *vrna_ptable_from_string()*. The resulting pair tables are then internally passed to *vrna_bp_distance_pt()*. To control which kind of matching brackets will be used during conversion, the optional argument `options` can be used. See also the description of *vrna_ptable_from_string()* for available options. (default: **VRNA_BRACKETS_RND**). See, e.g. *RNA.bp_distance()* in the *The RNA Python module*.

See also

vrna_bp_distance_pt()

Parameters

- **str1** – First structure in dot-bracket notation
- **str2** – Second structure in dot-bracket notation

Returns

The base pair distance between str1 and str2

double **vrna_dist_mountain**(const char *str1, const char *str2, unsigned int p)

```
#include <ViennaRNA/structures/metrics.h>
```

Benchmarking

Benchmarking Secondary Structure Predictions

Typedefs

typedef struct *vrna_score_s* **vrna_score_t**

Typename for the score data structure *vrna_score_s*.

```
#include <ViennaRNA/structures/benchmark.h>
```

Functions

vrna_score_t **vrna_score_from_confusion_matrix**(double TP, double TN, double FP, double FN)

Construct score data structure from given confusion matrix.

```
#include <ViennaRNA/structures/benchmark.h>
```

Parameters

- **TP** – True positive count
- **TN** – True negative count
- **FP** – False positive count
- **FN** – False negative count

Returns

The score data structure to write

vrna_score_t **vrna_compare_structure_pt**(const short *pt_gold, const short *pt_other, int fuzzy)

Return statistic of two structure (in pair table) comparaison.

```
#include <ViennaRNA/structures/benchmark.h>
```

See also

vrna_compare_structure

Parameters

- **pt_gold** – Gold standard structure in pair table
- **pt_other** – Structure to compare in pair table
- **fuzzy** – Allows for base pair slippage. Hence, for any base pair (i,j) in the gold standard, a base pair (p, q) in the second structure is considered a true positive, if i - fuzzy <= p <= i + fuzzy, and j - fuzzy <= q <= j + fuzzy. Each gold standard base pair may be claimed as a true positive by at most one predicted base pair: once matched (within the fuzzy window) it is consumed and can't be matched again by another predicted pair. Matching is greedy in the order predicted pairs are encountered, not a globally optimal assignment, but for typical slippage values (fuzzy of a few nucleotides) this does not affect the result.

Returns
The *vrna_score_s* data structure

vrna_score_t **vrna_compare_structure**(const char *structure_gold, const char *structure_other, int fuzzy, unsigned int options)

Return statistic of two structure (in dbn) comparaison.

```
#include <ViennaRNA/structures/benchmark.h>
```

Parameters

- **structure_gold** – Gold standard structure
- **structure_other** – Structure to compare
- **fuzzy** – Allows for base pair slippage. Hence, for any base pair (i,j) in the gold standard, a base pair (p, q) in the second structure is considered a true positive, if i - fuzzy <= p <= i + fuzzy, and j - fuzzy <= q <= j + fuzzy. Each gold standard base pair may be claimed as a true positive by at most one predicted base pair: once matched (within the fuzzy window) it is consumed and can't be matched again by another predicted pair. Matching is greedy in the order predicted pairs are encountered, not a globally optimal assignment, but for typical slippage values (fuzzy of a few nucleotides) this does not affect the result.
- **options** – A bitmask to specify which brackets are recognized during conversion to pair table

Returns
The *vrna_score_s* data structure

struct **vrna_score_s**

#include <ViennaRNA/structures/benchmark.h> The data structure that contains statistic result of two structures comparaison.

Public Members

double **TP**

True Positive count.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **TN**

True Negative count.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **FP**

False Positive count.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **FN**

False Negative count.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **TPR**

True Positive Rate.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **PPV**

Positive Predictive Value.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **FPR**

False Positive Rate.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **FOR**

False Omission Rate.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **TNR**

True Negative Rate.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **FDR**

False Discovery Rate.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **FNR**

False Negative Rate.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **NPV**

Negative Predictive Value.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **F1**

F1 Score.

```
#include <ViennaRNA/structures/benchmark.h>
```

double **MCC**

Matthews Correlation Coefficient.

```
#include <ViennaRNA/structures/benchmark.h>
```

Other Structure Utilities

Functions

char ***vrna_db_compat_remove**(const char *structure, const char *structure_ref, unsigned int *removed)

Remove base pairs compatible with a reference structure.

```
#include <ViennaRNA/structures/dotbracket.h>
```

This function compares two structures (`structure` and `structure_ref`) of the same length in dot-bracket format and removes all base pairs in `structure` that are not present in the reference `structure_ref` as long as they are compatible with `structure_ref`.

This is useful in cases where one wants to score the difference between two structures but not account for additional compatible base pairs, e.g. during benchmarking.

See also

vrna_compare_structure(), *vrna_ptable_compat_remove*

Parameters

- **structure** – The structure in dot-bracket format that might undergo base pair removal
- **structure_ref** – The reference structure in dot-bracket format
- **removed** – A pointer to a variable where the the number of removed base pairs is stored to (maybe **NULL**)

Returns

A copy of `structure` missing the compatible base pairs

char ***vrna_db_from_probs**(const *FLT_OR_DBL* *pr, unsigned int length)

Create a dot-bracket like structure string from base pair probability matrix.

```
#include <ViennaRNA/structures/dotbracket.h>
```

SWIG Wrapper Notes:

This function is available as parameter-less method **db_from_probs()** bound to objects of type *fold_compound*. Parameters `pr` and `length` are implicitely taken from the *fold_compound* object the method is bound to. Upon missing base pair probabilities, this method returns an empty string. See, e.g. `RNA.db_from_probs()` in the *The RNA Python module*.

char ***vrna_pairing_tendency**(*vrna_fold_compound_t* *fc)

```
#include <ViennaRNA/structures/dotbracket.h>
```

char **vrna_bpp_symbol**(const float *x)

Get a pseudo dot bracket notation for a given probability information.

```
#include <ViennaRNA/structures/dotbracket.h>
```

char ***vrna_db_from_bps**(*vrna_bps_t* bp_stack, unsigned int length)

Create a dot-backet/parenthesis structure from backtracking stack.

```
#include <ViennaRNA/structures/dotbracket.h>
```

This function is capable to create dot-bracket structures from base pairs stored within the base pair stack `bp_stack`.

Parameters

- **bp_stack** – Base pair stack containing the traced base pairs
- **length** – The length of the structure

Returns
The secondary structure in dot-bracket notation as provided in the input

char ***vrna_db_from_bp_stack**(*vrna_bp_stack_t* *bp, unsigned int length)

Create a dot-backet/parenthesis structure from backtracking stack.

```
#include <ViennaRNA/structures/dotbracket.h>
```

This function is capable to create dot-bracket structures from suboptimal structure prediction sensu M. Zuker

Parameters

- **bp** – Base pair stack containing the traced base pairs
- **length** – The length of the structure

Returns
The secondary structure in dot-bracket notation as provided in the input

void **vrna_letter_structure**(char *structure, *vrna_bp_stack_t* *bp, unsigned int length)

```
#include <ViennaRNA/structures/dotbracket.h>
```

int ***vrna_loopidx_from_ptable**(const short *pt)

Get a loop index representation of a structure.

```
#include <ViennaRNA/structures/pairtable.h>
```

unsigned int ***vrna_refBPcnt_matrix**(const short *reference_pt, unsigned int turn)

Make a reference base pair count matrix.

```
#include <ViennaRNA/structures/utils.h>
```

Get an upper triangular matrix containing the number of basepairs of a reference structure for each interval [i,j] with i<j. Access it via iindx!!!

unsigned int ***vrna_refBPdist_matrix**(const short *pt1, const short *pt2, unsigned int turn)

Make a reference base pair distance matrix.

```
#include <ViennaRNA/structures/utils.h>
```

Get an upper triangular matrix containing the base pair distance of two reference structures for each interval [i,j] with i<j. Access it via iindx!!!

4.2.10 Mathematical Functions

Below you'll find the API description for our mathematical functions.

Scalar Mathematical Functions API

Defines

VRNA_MATH_FUN_BIN_OPTION_PROJECT

Options flag for linear data binning to activate data projection (mapping) instead of actual binning.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_bin(), *vrna_math_fun_dbl_bin_opt()*

VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_UPPERBOUND

Options flag for linear data binning to indicate that values out-of-upper-bound are to be mapped to the respective domain limit.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_bin(), *vrna_math_fun_dbl_bin_opt()*

VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_LOWERBOUND

Options flag for linear data binning to indicate that values out-of-lower-bound are to be mapped to the respective domain limit.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_bin(), *vrna_math_fun_dbl_bin_opt()*

VRNA_MATH_FUN_BIN_OPTION_DEFAULT

Options flag for linear data binning to indicate default settings.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_bin(), *vrna_math_fun_dbl_bin_opt()*

VRNA_MATH_FUN_LINEAR_OPTION_LOG

Options flag for transforming linear data using a linear function that enables log-transform of the source value.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_linear(), *vrna_math_fun_dbl_linear_opt()*

VRNA_MATH_FUN_LINEAR_OPTION_DEFAULT

Options flag for transforming linear data using linear function that indicates default settings.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_linear(), *vrna_math_fun_dbl_linear_opt()*

VRNA_MATH_FUN_LOG_OPTION_NONDEFAULT_BASE

Options flag for transforming linear data using log transform to use a non-default base.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_log(), *vrna_math_fun_dbl_log_opt()*

VRNA_MATH_FUN_LOG_OPTION_DEFAULT

Options flag for transforming linear data using log transform that indicates default settings.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_log(), *vrna_math_fun_dbl_log_opt()*

VRNA_MATH_FUN_LOGISTIC_OPTION_DEFAULT

Options flag for transforming linear data using logistic function that indicates default settings.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_logistic(), *vrna_math_fun_dbl_logistic_opt()*

VRNA_MATH_FUN_LOGIT_BOUNDS_DEFAULT

```
#include <ViennaRNA/math/functions.h>
```

VRNA_MATH_FUN_LOGIT_OOB_VALUES_DEFAULT

```
#include <ViennaRNA/math/functions.h>
```

VRNA_MATH_FUN_LOGIT_OPTION_BOUNDS_OPEN

```
#include <ViennaRNA/math/functions.h>
```

VRNA_MATH_FUN_LOGIT_OPTION_BOUNDS_INCLUSIVE_LOWER

```
#include <ViennaRNA/math/functions.h>
```

VRNA_MATH_FUN_LOGIT_OPTION_BOUNDS_INCLUSIVE_UPPER

```
#include <ViennaRNA/math/functions.h>
```

VRNA_MATH_FUN_LOGIT_OPTION_BOUNDS_INCLUSIVE

```
#include <ViennaRNA/math/functions.h>
```

VRNA_MATH_FUN_LOGIT_OPTION_DEFAULT

```
#include <ViennaRNA/math/functions.h>
```

VRNA_MATH_FUN_KDE_OPTION_DEFAULT

Options flag for transforming linear data using a kernel density estimate (KDE) that indicates default settings.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_kde(), *vrna_math_fun_dbl_kde_opt()*

VRNA_MATH_FUN_GAUSSIAN_OPTION_DEFAULT

Options flag for transforming linear data using a Gaussian function that indicates default settings.

```
#include <ViennaRNA/math/functions.h>
```

See also

vrna_math_fun_dbl_gaussian(), *vrna_math_fun_dbl_gaussian_opt()*

Typedefs

typedef void ***vrna_math_fun_dbl_opt_t**

Option data structure type for mathematical functions.

```
#include <ViennaRNA/math/functions.h>
```

typedef *vrna_auxdata_free_f* **vrna_math_fun_dbl_opt_free_f**

Callback function to release any memory occupied by the mathematical function option *vrna_math_fun_dbl_opt_t*.

```
#include <ViennaRNA/math/functions.h>
```

typedef double (***vrna_math_fun_dbl_f**)(double value, *vrna_math_fun_dbl_opt_t* options)

Callback function prototype for scalar mathematical functions of the form $y = f(x)$.

```
#include <ViennaRNA/math/functions.h>
```

This is the prototype for callback functions used for scalar mathematical functions. Its main purpose is to encapsulate the options (arguments, such as constansts, etc.) for the mathematical function and can therefore be used in a more generalized way wherever different mathematical functions may be applied in the same manner. Functions of this type have to be provided two parameters: `value` which is the value x that is about to be evaluated `options`, which is a pointer to a data structure that holds any means required to ensure that the callback can evaluate the provide data. The callback function then returns the transformed value y.

Notes on Callback Functions:

This callback handles mathematical functions working on scalar *double* values.

See also

vrna_data_lin_transform(), *vrna_math_fun_dbl_bin_opt()*, *vrna_math_fun_dbl_linear_opt()*, *vrna_math_fun_dbl_log_opt()*, *vrna_math_fun_dbl_gaussian_opt()*, *vrna_math_fun_dbl_logistic_opt()*, *vrna_math_fun_dbl_kde_opt()*

Param value
The value x that is to be evaluated by the function $f(x)$

Param options
A pointer to a data structure specific for the mathematical function

Return
The evaluated value y

Functions

double **vrna_math_fun_dbl_bin**(double value, double (*thresholds)[2], size_t thresholds_num, double oolb_value, double ooub_value, unsigned int options)

Bin a scalar input value or project it from one domain into another.

```
#include <ViennaRNA/math/functions.h>
```

This function can be used to map continuous data from one domain (*source*) into discrete or continuous data of another domain (*target*). User-defined domain boundaries are provided by the `thresholds` argument, which boils down to a list of pairs of values, where the first value is the boundary in the source domain. The second value is the corresponding boundary of the target domain. The first and last pair of domain boundaries denote the lower and the upper domain limits, respectively. Any value outside of these limits can either be discarded by assigning them a special out-of-bounds value (`oolb_value`, `ooub_value`), or they can be mapped directly to the domain limits. Control over this mapping is available through the `options` argument by using the binary flags *VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_UPPERBOUND* and *VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_LOWERBOUND*.

By default, the transformation maps any data within a source interval as specified by two consecutive entries in the `thresholds` argument to the exact value of the second target boundary. This process is also called `binning`. Alternatively, this implementation also allows for a continuous mapping (projection) of the source intervals into the target intervals. To activate this behavior, the *VRNA_MATH_FUN_BIN_OPTION_PROJECT* flag must be provided to the `options` argument.

See also

vrna_data_lin_transform_opt(), *VRNA_MATH_FUN_BIN_OPTION_DEFAULT*, *VRNA_MATH_FUN_BIN_OPTION_PROJECT*, *VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_UPPERBOUND*, *VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_LOWERBOUND*, *vrna_math_fun_dbl_linear()*, *vrna_math_fun_dbl_log()*, *vrna_math_fun_dbl_kde()*, *vrna_math_fun_dbl_gaussian()*

Parameters

- **value** – The input value
- **thresholds** – A pointer to an array of data pairs holding the source and target domain boundaries
- **thresholds_num** – The number of domain boundary pairs available in `thresholds`
- **oolb_value** – Out-of-lower-bound value
- **ooub_value** – Out-of-upper-bound value
- **options** – Additional options that change the behavior of this function

Returns
The projection of the input value into the target domain

vrna_math_fun_dbl_f **vrna_math_fun_dbl_bin_opt**(double (*thresholds)[2], unsigned int thresholds_num, double oolb_value, double ooub_value, unsigned int options, *vrna_math_fun_dbl_opt_t* *fun_options_p, *vrna_math_fun_dbl_opt_free_f* *fun_options_free)

Retrieve a function pointer that performs (discrete) binning and more.

```
#include <ViennaRNA/math/functions.h>
```

This function yields a linear data transform callback and the associated transform options data structure suitable for usage in *vrna_data_lin_transform()*. The transformation this callback performs can be described as (discrete) binning or data bucketing and essentially encapsulates the *vrna_math_fun_dbl_bin()* function.

The `fun_options_p` and `fun_options_free` pointers are used as additional output to obtain the addresses of the transformation option data structure that has to be provided to the *vrna_data_lin_transform()* function and a function pointer to release the memory of the option data structure once it is not required anymore.

See also

vrna_math_fun_dbl_bin(), *vrna_data_lin_transform()*, *vrna_math_fun_dbl_f*, *vrna_math_fun_dbl_opt_t*, *vrna_math_fun_dbl_opt_free_f*, *VRNA_MATH_FUN_BIN_OPTION_DEFAULT*, *VRNA_MATH_FUN_BIN_OPTION_PROJECT*, *VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_UPPERBOUND*, *VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_LOWERBOUND*, *vrna_math_fun_dbl_linear_opt()*, *vrna_math_fun_dbl_log_opt()*, *vrna_math_fun_dbl_kde_opt()*, *vrna_math_fun_dbl_gaussian_opt()*

Parameters

- **thresholds** – A pointer to an array of data pairs holding the source and target domain boundaries
- **thresholds_num** – The number of domain boundary pairs available in `thresholds`
- **oolb_value** – Out-of-lower-bound value
- **ooub_value** – Out-of-upper-bound value
- **options** – Additional options that change the behavior of the callback function
- **fun_options_p** – A pointer to store the address of the options data structure
- **fun_options_free** – A pointer to store the address of the `free` function that releases the memory of the options data structure

Returns

A callback function that performs (discrete) data binning

double **vrna_math_fun_dbl_linear**(double x, double slope, double intercept, unsigned int options)

Transform a scalar input value through a linear function.

```
#include <ViennaRNA/math/functions.h>
```

This function can be used to transform an input value (parameter `x`) into an output value by applying a linear function of the form

$$y = a + b \cdot f(x)$$

where x is the input value (`source`), a and b are *intercept* and *slope*, and y is the output value (`target`). By default, the function $f(x) = x$ is the identity function. However, the callback can be instructed to apply a log-transform, $f(x) = logx$, instead. Control over the behavior of $f(x)$ can be gained by providing a corresponding option flag to the `options` argument, e.g. *VRNA_MATH_FUN_LINEAR_OPTION_LOG*.

See also

vrna_math_fun_dbl_linear_opt(), *VRNA_MATH_FUN_LINEAR_OPTION_DEFAULT*, *VRNA_MATH_FUN_LINEAR_OPTION_LOG*, *vrna_math_fun_dbl_linear()*, *vrna_math_fun_dbl_log()*, *vrna_math_fun_dbl_kde()*, *vrna_math_fun_dbl_gaussian()*

Parameters

- **x** – The input value
- **slope** – The slope of the linear function
- **intercept** – The intercept of the linear function
- **options** – Additional options that change the behavior of the callback function

Returns

The transformed output value y

vrna_math_fun_dbl_f **vrna_math_fun_dbl_linear_opt**(double slope, double intercept, unsigned int options, *vrna_math_fun_dbl_opt_t* *fun_options_p, *vrna_math_fun_dbl_opt_free_f* *fun_options_free)

Retrieve a function pointer that applies a linear function.

```
#include <ViennaRNA/math/functions.h>
```

This function yields a linear data transform callback and the associated transform options data structure suitable for usage in *vrna_data_lin_transform()*. The callback applies a linear function $y = a + b \cdot f(x)$ and essentially encapsulates the *vrna_math_fun_dbl_linear()* function.

The `fun_options_p` and `fun_options_free` pointers are used as additional output to obtain the addresses of the transformation option data structure that has to be provided to the *vrna_data_lin_transform()* function and a function pointer to release the memory of the option data structure once it is not required anymore.

See also

vrna_math_fun_dbl_linear(), *vrna_data_lin_transform()*, *vrna_math_fun_dbl_f*, *vrna_math_fun_dbl_opt_t*, *vrna_math_fun_dbl_opt_free_f*, *VRNA_MATH_FUN_LINEAR_OPTION_DEFAULT*, *VRNA_MATH_FUN_LINEAR_OPTION_LOG*, *vrna_math_fun_dbl_bin_opt()*, *vrna_math_fun_dbl_log_opt()*, *vrna_math_fun_dbl_kde_opt()*, *vrna_math_fun_dbl_gaussian_opt()*

Parameters

- **slope** – The slope of the linear function
- **intercept** – The intercept of the linear function
- **options** – Additional options that change the behavior of the callback function

- **fun_options_p** – A pointer to store the address of the options data structure
- **fun_options_free** – A pointer to store the address of the `free` function that releases the memory of the options data structure

Returns
A callback function that performs transformation through a linear model

double **vrna_math_fun_dbl_log**(double x, double base, double oob_value, unsigned int options)

Transform a scalar input value through a logarithmic function.

```
#include <ViennaRNA/math/functions.h>
```

This function transforms a scalar input value by computing the logarithm function

$$y = \log_b x$$

where x is the input value (`source`), b is the base, and y is the output value (`target`). By default, the *natural* logarithm is used, i.e. $b = \mathrm{e}$. The `base` argument in combination with the *VRNA_MATH_FUN_LOG_OPTION_NONDEFAULT_BASE* flag supplied to the `options` argument can be used to change the base to any other number.

The function returns the user-specified out-of-bounds value `oob_value` if $x \leq 0$.

See also

vrna_math_fun_dbl_log_opt(), *VRNA_MATH_FUN_LOG_OPTION_DEFAULT*, *VRNA_MATH_FUN_LOG_OPTION_NONDEFAULT_BASE*, *vrna_math_fun_dbl_linear()*, *vrna_math_fun_dbl_log()*, *vrna_math_fun_dbl_kde()*, *vrna_math_fun_dbl_gaussian()*

Parameters

- **x** – The input value
- **base** – The base b of the logarithm (only used if *VRNA_MATH_FUN_LOG_OPTION_NONDEFAULT_BASE* is passed to `options`
- **oob_value** – Out-of-bound value
- **options** – Additional options that change the behavior of the callback function

Returns
The transformed *target* value y

vrna_math_fun_dbl_f **vrna_math_fun_dbl_log_opt**(double value_shift, double base, double oob_value, unsigned int options, *vrna_math_fun_dbl_opt_t* *fun_options_p, *vrna_math_fun_dbl_opt_free_f* *fun_options_free)

Retrieve a function pointer a logarithmic function.

```
#include <ViennaRNA/math/functions.h>
```

This function yields a linear data transform callback and the associated transform options data structure suitable for usage in *vrna_data_lin_transform()*. The callback applies a logarithmic function $y = \log_b(x + c)$ and essentially encapsulates the *vrna_math_fun_dbl_log()* function. Note, that here, the `value_shift` argument corresponds to c in the above formula and in most cases should be `0`. The callback then returns the out-of-bounds value `oob_value` if $x + c \leq 0$.

The `fun_options_p` and `fun_options_free` pointers are used as additional output to obtain the addresses of the transformation option data structure that has to be provided to the *vrna_data_lin_transform()* function and a function pointer to release the memory of the option data structure once it is not required anymore.

See also

vrna_math_fun_dbl_log(), *vrna_data_lin_transform()*, *vrna_math_fun_dbl_f*, *vrna_math_fun_dbl_opt_t*, *vrna_math_fun_dbl_opt_free_f*, *VRNA_MATH_FUN_LOG_OPTION_DEFAULT*, *VRNA_MATH_FUN_LOG_OPTION_NONDEFAULT_BASE*, *vrna_math_fun_dbl_bin_opt()*, *vrna_math_fun_dbl_linear_opt()*, *vrna_math_fun_dbl_kde_opt()*, *vrna_math_fun_dbl_gaussian_opt()*

Parameters

- **value_shift** – The shift value c
- **base** – The base b of the logarithm (only used if *VRNA_MATH_FUN_LOG_OPTION_NONDEFAULT_BASE* is passed to `options`
- **oob_value** – Out-of-bound value
- **options** – Additional options that change the behavior of the callback function
- **fun_options_p** – A pointer to store the address of the options data structure
- **fun_options_free** – A pointer to store the address of the `free` function that releases the memory of the options data structure

Returns

A callback function that performs transformation through a linear model

double **vrna_math_fun_dbl_logistic**(double x, double mid_point, double supremum, double growth_rate, unsigned int options)

Transform a scalar input value through a logistic function.

```
#include <ViennaRNA/math/functions.h>
```

This function transforms a scalar input value by applying a logistic function of the form

$$y = \frac{L}{1 + e^{-k \cdot (x - x_0)}}$$

where x is the input value (`source`), x_0 is the mid point (`mid_point`) of the function, k is the logistic growth rate (`growth_rate`), and L is the supremum (or carrying capacity) of the function (`supremum`). The standard logistic function is defined as $L = 1$, $k = 1.$ and $x_0 = 0$, i.e.

$$y = \frac{1}{1 + e^{-x}}$$

See also

vrna_math_fun_dbl_logistic_opt(), *VRNA_MATH_FUN_LOGISTIC_OPTION_DEFAULT*, *vrna_math_fun_dbl_bin()*, *vrna_math_fun_dbl_linear()*, *vrna_math_fun_dbl_log()*, *vrna_math_fun_dbl_kde()*, *vrna_math_fun_dbl_gaussian()*

Parameters

- **x** – The input value
- **mid_point** – The midpoint of the function
- **supremum** – The supremum of the function
- **growth_rate** – The growth rate of the function
- **options** – Additional options that change the behavior of the callback function

Returns
The transformed *target* value y

vrna_math_fun_dbl_f **vrna_math_fun_dbl_logistic_opt**(double mid_point, double supremum, double growth_rate, unsigned int options, *vrna_math_fun_dbl_opt_t* *fun_options_p, *vrna_math_fun_dbl_opt_free_f* *fun_options_free)

Retrieve a function pointer that applies a logistic function.

```
#include <ViennaRNA/math/functions.h>
```

This function yields a linear data transform callback and the associated transform options data structure suitable for usage in *vrna_data_lin_transform()*. The callback applies a logistic function of the form

$$y = \frac{L}{1 + e^{-k \cdot (x - x_0)}}$$

and essentially encapsulates the *vrna_math_fun_dbl_logistic()* function.

The `fun_options_p` and `fun_options_free` pointers are used as additional output to obtain the addresses of the transformation option data structure that has to be provided to the *vrna_data_lin_transform()* function and a function pointer to release the memory of the option data structure once it is not required anymore.

See also

vrna_math_fun_dbl_logistic(), *vrna_data_lin_transform()*, *vrna_math_fun_dbl_f*, *vrna_math_fun_dbl_opt_t*, *vrna_math_fun_dbl_opt_free_f*, *VRNA_MATH_FUN_LOGISTIC_OPTION_DEFAULT*, *vrna_math_fun_dbl_bin_opt()*, *vrna_math_fun_dbl_linear_opt()*, *vrna_math_fun_dbl_kde_opt()*, *vrna_math_fun_dbl_gaussian_opt()*, *vrna_math_fun_dbl_log_opt()*

Parameters

- **mid_point** – The midpoint of the function
- **supremum** – The supremum of the function
- **growth_rate** – The growth rate of the function
- **options** – Additional options that change the behavior of the callback function
- **fun_options_p** – A pointer to store the address of the options data structure
- **fun_options_free** – A pointer to store the address of the `free` function that releases the memory of the options data structure

Returns
A callback function that performs transformation through a logistic function

vrna_math_fun_dbl_f **vrna_math_fun_dbl_logit_opt**(const double bounds[2], const double oob_values[2], unsigned int options, *vrna_math_fun_dbl_opt_t* *fun_options_p, *vrna_math_fun_dbl_opt_free_f* *fun_options_free)

```
#include <ViennaRNA/math/functions.h>
```

double **vrna_math_fun_dbl_logit**(double p, const double bounds[2], const double oob_values[2], unsigned int options)

```
#include <ViennaRNA/math/functions.h>
```

double **vrna_math_fun_dbl_kde**(double x, double *samples, size_t num_samples, *vrna_math_fun_dbl_f* kernel, *vrna_math_fun_dbl_opt_t* kernel_data, double bandwidth, unsigned int options)

Estimate the probability density for a scalar input value through kernel density estimation (KDE).

```
#include <ViennaRNA/math/functions.h>
```

This function estimates the probability density function of a random input variable given a set of samples and a smoothing kernel. The estiamte of the probability density for a random variable x is then of the form

$$y = \hat{f}_h(x) = \frac{1}{n \cdot h} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

with a set of $n > 0$ samples $\{x_1, x_2, \ldots, x_n\}$ (`samples`), K (`kernel`) as a non-negative kernel function, and $h > 0$ is the bandwidth (`bandwidth`).

See also

vrna_math_fun_dbl_kde_opt(), *VRNA_MATH_FUN_KDE_OPTION_DEFAULT*, *vrna_math_fun_dbl_bin()*, *vrna_math_fun_dbl_linear()*, *vrna_math_fun_dbl_log()*, *vrna_math_fun_dbl_logistic()*, *vrna_math_fun_dbl_gaussian()*

Note

If no kernel is provided, i.e. `NULL` is passed to `kernel`, the KDE uses a standard Gaussian kernel by providing *vrna_math_fun_dbl_gaussian()* with $a = \frac{1}{\sqrt{2\pi}}, b = 0, c = 1$. In this case, the bandwidth (`bandwidth`) will be automatically replaced by a rule-of-thumb bandwidth estimator $h = \left(\frac{4}{3n}\right)^{\frac{1}{5}}$

Parameters

- **x** – The input value
- **samples** – The samples of the distribution to estimate $\{x_1, x_2, \ldots, x_n\}$
- **num_samples** – The number of samples (n) provided by pointer `samples`
- **kernel** – The kernel function K (maybe `NULL`)
- **kernel_data** – The data that should be passed-through to the kernel function (maybe `NULL`)
- **bandwidth** – The bandwidth h

- **options** – Additional options that change the behavior of the callback function

Returns

The estimated probability density $\hat{f}_h(x)$

vrna_math_fun_dbl_f **vrna_math_fun_dbl_kde_opt**(double *samples, size_t num_samples,
vrna_math_fun_dbl_f kernel,
vrna_math_fun_dbl_opt_t kernel_data,
vrna_math_fun_dbl_opt_free_f kernel_data_free,
double bandwidth, unsigned int options,
vrna_math_fun_dbl_opt_t *fun_options_p,
vrna_math_fun_dbl_opt_free_f
*fun_options_free)

Retrieve a callback function that applies a kernel density estimation (KDE).

```
#include <ViennaRNA/math/functions.h>
```

This function yields a linear data transform callback and the associated transform options data structure suitable for usage in *vrna_data_lin_transform()*. The callback applies a kernel density estimation (KDE) and essentially encapsulates the *vrna_math_fun_dbl_kde()* function.

The `fun_options_p` and `fun_options_free` pointers are used as additional output to obtain the addresses of the transformation option data structure that has to be provided to the *vrna_data_lin_transform()* function and a function pointer to release the memory of the option data structure once it is not required anymore.

See also

vrna_math_fun_dbl_logistic(), *vrna_data_lin_transform()*, *vrna_math_fun_dbl_f*, *vrna_math_fun_dbl_opt_t*, *vrna_math_fun_dbl_opt_free_f*, *VRNA_MATH_FUN_KDE_OPTION_DEFAULT*, *vrna_math_fun_dbl_gaussian_opt()*, *vrna_math_fun_dbl_bin_opt()*, *vrna_math_fun_dbl_linear_opt()*, *vrna_math_fun_dbl_log_opt()*, *vrna_math_fun_dbl_logistic_opt*

Note

Whenever the `fun_options_free` callback will be called, this function also releases the memory occupied by `kernel_data` through the `kernel_data_free` function if it is provided, i.e. not `NULL`.

Parameters

- **samples** – The samples of the distribution to estimate $\{x_1, x_2, \ldots, x_n\}$
- **num_samples** – The number of samples (n) provided by pointer `samples`
- **kernel** – The kernel function K (maybe `NULL`)
- **kernel_data** – The data that should be passed-through to the kernel function (maybe `NULL`)
- **kernel_data_free** – A function to free the memory occupied by the data for the kernel function (maybe `NULL`)
- **bandwidth** – The bandwidth h
- **options** – Additional options that change the behavior of the callback function

- **fun_options_p** – A pointer to store the address of the options data structure
- **fun_options_free** – A pointer to store the address of the `free` function that releases the memory of the options data structure

Returns
A callback function that performs transformation through kernel density estimation

double **vrna_math_fun_dbl_gaussian**(double x, double a, double b, double c)

Evaluate a Gaussian function for a scalar input value.

```
#include <ViennaRNA/math/functions.h>
```

This function applies a Gaussian function of the form

$$y = f(x) = a \cdot \exp(-\frac{(x-b)^2}{2c^2})$$

with arbitrary constants a, b and non-zero c to a scalar input value x.

See also

vrna_math_fun_dbl_gaussian_opt()

Parameters

- **x** – The input value x
- **a** – The arbitrary constant a
- **b** – The arbitrary constant b
- **c** – The arbitrary non-zero constant c

Returns
The evaluation for the Gaussian function $y = f(x)$

vrna_math_fun_dbl_f **vrna_math_fun_dbl_gaussian_opt**(double a, double b, double c, unsigned int options, *vrna_math_fun_dbl_opt_t* *fun_options_p, *vrna_math_fun_dbl_opt_free_f* *fun_options_free)

Retrieve a function pointer that applies Gaussian function.

```
#include <ViennaRNA/math/functions.h>
```

This function yields a linear data transform callback and the associated transform options data structure suitable for usage in *vrna_data_lin_transform()*. The callback applies a Gaussian function of the form

$$y = a \cdot \exp(-\frac{(x-b)^2}{2c^2})$$

with arbitrary constants a, b and non-zero c and x being the input value (`source`).

The `fun_options_p` and `fun_options_free` pointers are used as additional output to obtain the addresses of the transformation option data structure that has to be provided to the *vrna_data_lin_transform()* function and a function pointer to release the memory of the option data structure once it is not required anymore.

See also

vrna_math_fun_dbl_gaussian(), *vrna_data_lin_transform()*, *vrna_math_fun_dbl_f*, *vrna_math_fun_dbl_opt_t*, *vrna_math_fun_dbl_opt_free_f*, *VRNA_MATH_FUN_GAUSSIAN_OPTION_DEFAULT*

Parameters

- **a** – The arbitrary constant a
- **b** – The arbitrary constant b
- **c** – The arbitrary non-zero constant c
- **options** – Additional options that change the behavior of the callback function
- **fun_options_p** – A pointer to store the address of the options data structure
- **fun_options_free** – A pointer to store the address of the `free` function that releases the memory of the options data structure

Returns

A callback function that performs transformation through a Gaussian function

vrna_math_fun_dbl_f **vrna_math_fun_dbl_chain**(*vrna_math_fun_dbl_f* first_f,
vrna_math_fun_dbl_opt_t first_f_opt,
vrna_math_fun_dbl_opt_free_f first_f_opt_free,
vrna_math_fun_dbl_opt_t *fun_options_p,
vrna_math_fun_dbl_opt_free_f *fun_options_free)

```
#include <ViennaRNA/math/functions.h>
```

size_t **vrna_math_fun_dbl_chain_append**(*vrna_math_fun_dbl_opt_t* chain_opt,
vrna_math_fun_dbl_f f, *vrna_math_fun_dbl_opt_t* f_opt,
vrna_math_fun_dbl_opt_free_f f_opt_free)

```
#include <ViennaRNA/math/functions.h>
```

size_t **vrna_math_fun_dbl_chain_size**(*vrna_math_fun_dbl_opt_t* chain)

```
#include <ViennaRNA/math/functions.h>
```

vrna_math_fun_dbl_f **vrna_math_fun_dbl_chain_at**(*vrna_math_fun_dbl_opt_t* chain, size_t pos,
vrna_math_fun_dbl_opt_t *fun_options_p)

```
#include <ViennaRNA/math/functions.h>
```

4.2.11 Data Handling and Manipulation

API symbols for the data transform implementation.

Data Transform API

Defines

VRNA_TRANSFORM_ENFORCE_DOMAIN_SOURCE

Options flag for transforming linear data to enforce source domain limits.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_ENFORCE_DOMAIN_TARGET

Options flag for transforming linear data to enforce target domain limits.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_ENFORCE_DOMAINS

Options flag for transforming linear data to enforce source and target domain limits.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_MAP_SOURCE_LOW

Options flag for transforming linear data to map source values below the domain limit to the lower domain limit.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_MAP_SOURCE_HIGH

Options flag for transforming linear data to map source values above the domain limit to the upper domain limit.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_MAP_SOURCE

Options flag for transforming linear data to map source values below and above the domain limits to the domain limits.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_MAP_TARGET_LOW

Options flag for transforming linear data to map target values below the domain limit to the lower domain limit.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_MAP_TARGET_HIGH

Options flag for transforming linear data to map target values above the domain limit to the upper domain limit.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_MAP_TARGET

Options flag for transforming linear data to map target values below and above the domain limits to the domain limits.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_MAP

Options flag for transforming linear data to map source and target values below and above the domain limits to the respective domain limits.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

VRNA_TRANSFORM_DEFAULT

Options flag for transforming linear data that indicates default settings.

```
#include <ViennaRNA/data/transform.h>
```

See also

vrna_data_lin_transform()

Functions

double ***vrna_data_lin_transform**(const double *data, size_t data_size, *vrna_math_fun_dbl_f* transform_cb, *vrna_math_fun_dbl_opt_t* transform_opt, double domain[4], double oob_value, unsigned int options)

Transform an array of linear data.

```
#include <ViennaRNA/data/transform.h>
```

This function transforms an array of linear data (`source`) into another array of linear data of the same size (`target`). For that purpose, it utilizes a callback mechanism that transforms each single value.

During data transform, the callback function (`transform_cb`) will be executed for each value of the linear data (`data`) and is responsible to actually transform the data value. The `transform_opt` parameter will be simply passed-through to the callback function as second argument. The callback function has to return the transformed value.

This transformation function may enforce source and target domain limits if the corresponding flag (*VRNA_TRANSFORM_ENFORCE_DOMAINS*) is provided to the `options` argument. This means that one has complete control over the accepted values of the source and target domains. The limits can be provided to this function through the `domain` argument. Here, the order of the limits follows:

$$x_{\min}, x_{\max}, y_{\min}, y_{\max}.$$

If `NULL` is passed instead of an actual domain array, domain enforcement is deactivated and any value will pass. In case a domain enforcing is active and a source or target value doesn't meet the respective limits, the function assigns it the out-of-bounds value `oob_value`. This behavior can be changed to

a mapping of out-of-bounds values to the respetive domain limits. For that purpose, the `options` argument requires the flag *VRNA_TRANSFORM_MAP*.

See also

vrna_math_fun_dbl_f, *vrna_math_fun_dbl_opt_t*, *VRNA_TRANSFORM_DEFAULT*, *VRNA_TRANSFORM_ENFORCE_DOMAINS*, *VRNA_TRANSFORM_ENFORCE_DOMAIN_SOURCE*, *VRNA_TRANSFORM_ENFORCE_DOMAIN_TARGET*, *VRNA_TRANSFORM_MAP*, *VRNA_TRANSFORM_MAP_SOURCE*, *VRNA_TRANSFORM_MAP_TARGET*, *VRNA_TRANSFORM_MAP_SOURCE_LOW*, *VRNA_TRANSFORM_MAP_SOURCE_HIGH*, *VRNA_TRANSFORM_MAP_TARGET_LOW*, *VRNA_TRANSFORM_MAP_TARGET_HIGH*, *vrna_math_fun_dbl_bin_opt()*, *vrna_math_fun_dbl_linear_opt()*, *vrna_math_fun_dbl_log_opt()*, *vrna_math_fun_dbl_logistic_opt()*, *vrna_math_fun_dbl_gaussian_opt()*, *vrna_math_fun_dbl_kde_opt()*

Note

Individual control for mapping out-of-bounds values to the four domain limits, i.e. low and high values of source and target can be gained by providing the `options` argument the *VRNA_TRANSFORM_MAP_SOURCE_LOW*, *VRNA_TRANSFORM_MAP_SOURCE_HIGH*, *VRNA_TRANSFORM_MAP_TARGET_LOW*, and *VRNA_TRANSFORM_MAP_TARGET_HIGH* flags.

Note

When `data` is `NULL` or `data_size` equals `0`, the function returns `NULL`. Moreover, the function simply provides a copy of the linear data if the transformation callback `transform_cb` is not provided, i.e. if it is `NULL`. In this case, domain limits will still be enforced if the corresponding options are set.

Parameters

- **`data`** – A pointer to an array of linear data
- **`data_size`** – The size of the array `data` is pointing to
- **`transform_cb`** – The data transformation callback (maybe `NULL`)
- **`transform_opt`** – The options that need to be passed through to the transformation callback `transform_cb` (maybe `NULL`)
- **`domain`** – The domain limits (maybe `NULL`)
- **`oob_value`** – Out-of-bound value
- **`options`** – Additional options that change the behavior of the transformation function

Returns

A pointer to an array of transformed linear data (or `NULL` on any error)

4.2.12 Search Algorithms

Implementations of various search algorithms to detect strings of objects within other strings of objects.

Functions

const unsigned int ***vrna_search_BMH_num**(const unsigned int *needle, size_t needle_size, const unsigned int *haystack, size_t haystack_size, size_t start, size_t *badchars, unsigned char cyclic)

Search for a string of elements in a larger string of elements using the Boyer-Moore-Horspool algorithm.

```
#include <ViennaRNA/search/BoyerMoore.h>
```

To speed-up subsequent searches with this function, the Bad Character Table should be precomputed and passed as argument `badchars`.

See also

vrna_search_BM_BCT_num(), *vrna_search_BMH()*

Parameters

- **needle** – The pattern of object representations to search for
- **needle_size** – The size (length) of the pattern provided in `needle`
- **haystack** – The string of objects the search will be performed on
- **haystack_size** – The size (length) of the `haystack` string
- **start** – The position within `haystack` where to start the search
- **badchars** – A pre-computed Bad Character Table obtained from *vrna_search_BM_BCT_num()* (If NULL, a Bad Character Table will be generated automatically)
- **cyclic** – Allow for cyclic matches if non-zero, stop search at end of haystack otherwise

Returns

A pointer to the first occurence of `needle` within `haystack` after position `start`

const char ***vrna_search_BMH**(const char *needle, size_t needle_size, const char *haystack, size_t haystack_size, size_t start, size_t *badchars, unsigned char cyclic)

Search for an ASCII pattern within a larger ASCII string using the Boyer-Moore-Horspool algorithm.

```
#include <ViennaRNA/search/BoyerMoore.h>
```

To speed-up subsequent searches with this function, the Bad Character Table should be precomputed and passed as argument `badchars`. Furthermore, both, the lengths of `needle` and the length of `haystack` should be pre-computed and must be passed along with each call.

See also

vrna_search_BM_BCT(), *vrna_search_BMH_num()*

Parameters

- **needle** – The NULL-terminated ASCII pattern to search for
- **needle_size** – The size (length) of the pattern provided in `needle`
- **haystack** – The NULL-terminated ASCII string of the search will be performed on

- **haystack_size** – The size (length) of the haystack string
- **start** – The position within haystack where to start the search
- **badchars** – A pre-computed Bad Character Table obtained from *vrna_search_BM_BCT()* (If NULL, a Bad Character Table will be generated automatically)
- **cyclic** – Allow for cyclic matches if non-zero, stop search at end of haystack otherwise

Returns
A pointer to the first occurence of needle within haystack after position start

size_t ***vrna_search_BM_BCT_num**(const unsigned int *pattern, size_t pattern_size, unsigned int num_max)

Retrieve a Boyer-Moore Bad Character Table for a pattern of elements represented by natural numbers.

```
#include <ViennaRNA/search/BoyerMoore.h>
```

See also

vrna_search_BMH_num(), *vrna_search_BM_BCT()*

Note

We store the maximum number representation of an element num_max at position 0. So the actual bad character table T starts at T[1] for an element represented by number 0.

Parameters

- **pattern** – The pattern of element representations used in the subsequent search
- **pattern_size** – The size (length) of the pattern provided in pattern
- **num_max** – The maximum number representation of an element, i.e. the size of the alphabet

Returns
A Bad Character Table for use in our Boyer-Moore search algorithm implementation(s)

size_t ***vrna_search_BM_BCT**(const char *pattern)

Retrieve a Boyer-Moore Bad Character Table for a NULL-terminated pattern of ASCII characters.

```
#include <ViennaRNA/search/BoyerMoore.h>
```

See also

vrna_search_BMH(), *vrna_search_BM_BCT_num()*

> **Note**
>
> We store the maximum number representation of an element, i.e. `127` at position `0`. So the actual bad character table `T` starts at `T[1]` for an element represented by ASCII code `0`.

Parameters

- **pattern** – The NULL-terminated pattern of ASCII characters used in the subsequent search

Returns

A Bad Character Table for use in our Boyer-Moore search algorithm implementation(s)

4.2.13 Combinatorics Algorithms

Implementations to solve various combinatorial aspects for strings of objects.

Functions

unsigned int ****vrna_enumerate_necklaces**(const unsigned int *type_counts)

Enumerate all necklaces with fixed content.

```
#include <ViennaRNA/combinatorics/basic.h>
```

This function implements *A fast algorithm to generate necklaces with fixed content* as published by Sawada [2003] .

The function receives a list of counts (the elements on the necklace) for each type of object within a necklace. The list starts at index 0 and ends with an entry that has a count of 0. The algorithm then enumerates all non-cyclic permutations of the content, returned as a list of necklaces. This list, again, is zero-terminated, i.e. the last entry of the list is a `NULL` pointer.

SWIG Wrapper Notes:

This function is available as global function `enumerate_necklaces()` which accepts lists input, an produces list of lists output. See, e.g. *`RNA.enumerate_necklaces()`* in the *The RNA Python module* .

Parameters

- **type_counts** – A 0-terminated list of entity counts

Returns

A list of all non-cyclic permutations of the entities

unsigned int **vrna_rotational_symmetry_num**(const unsigned int *string, size_t string_length)

Determine the order of rotational symmetry for a string of objects represented by natural numbers.

```
#include <ViennaRNA/combinatorics/basic.h>
```

The algorithm applies a fast search of the provided string within itself, assuming the end of the string wraps around to connect with it's start. For example, a string of the form `011011` has rotational symmetry of order `2`

This is a simplified version of *vrna_rotational_symmetry_pos_num()* that may be useful if one is only interested in the degree of rotational symmetry but not the actual set of rotational symmetric strings.

SWIG Wrapper Notes:

This function is available as global function `rotational_symmetry()`. See *vrna_rotational_symmetry_pos()* for details. Note, that in the target language the length of the list `string` is always known a-priori, so the parameter `string_length` must be omitted. See, e.g. *`RNA.rotational_symmetry()`* in the *The RNA Python module* .

See also

vrna_rotational_symmetry_pos_num(), vrna_rotationa_symmetry()

Parameters

- **`string`** – The string of elements encoded as natural numbers
- **`string_length`** – The length of the string

Returns

The order of rotational symmetry

unsigned int **`vrna_rotational_symmetry_pos_num`**(const unsigned int *string, size_t string_length, unsigned int **positions)

Determine the order of rotational symmetry for a string of objects represented by natural numbers.

```
#include <ViennaRNA/combinatorics/basic.h>
```

The algorithm applies a fast search of the provided string within itself, assuming the end of the string wraps around to connect with it's start. For example, a string of the form `011011` has rotational symmetry of order `2`

If the argument `positions` is not `NULL`, the function stores an array of string start positions for rotational shifts that map the string back onto itself. This array has length of order of rotational symmetry, i.e. the number returned by this function. The first element `positions`[0] always contains a shift value of `0` representing the trivial rotation.

SWIG Wrapper Notes:

This function is available as global function `rotational_symmetry()`. See *vrna_rotational_symmetry_pos()* for details. Note, that in the target language the length of the list `string` is always known a-priori, so the parameter `string_length` must be omitted. See, e.g. *`RNA.rotational_symmetry()`* in the *The RNA Python module* .

See also

vrna_rotational_symmetry_num(), *vrna_rotational_symmetry()*, *vrna_rotational_symmetry_pos()*

Note

Do not forget to release the memory occupied by `positions` after a successful execution of this function.

Parameters

- **`string`** – The string of elements encoded as natural numbers
- **`string_length`** – The length of the string
- **`positions`** – A pointer to an (undefined) list of alternative string start positions that lead to an identity mapping (may be NULL)

Returns
The order of rotational symmetry

unsigned int **vrna_rotational_symmetry**(const char *string)

Determine the order of rotational symmetry for a NULL-terminated string of ASCII characters.

```
#include <ViennaRNA/combinatorics/basic.h>
```

The algorithm applies a fast search of the provided string within itself, assuming the end of the string wraps around to connect with it's start. For example, a string of the form `AABAAB` has rotational symmetry of order `2`

This is a simplified version of *vrna_rotational_symmetry_pos()* that may be useful if one is only interested in the degree of rotational symmetry but not the actual set of rotational symmetric strings.

SWIG Wrapper Notes:

This function is available as global function `rotational_symmetry()`. See *vrna_rotational_symmetry_pos()* for details. See, e.g. *RNA.rotational_symmetry()* in the *The RNA Python module* .

See also

vrna_rotational_symmetry_pos(), vrna_rotationa_symmetry_num()

Parameters

- **string** – A NULL-terminated string of characters

Returns
The order of rotational symmetry

unsigned int **vrna_rotational_symmetry_pos**(const char *string, unsigned int **positions)

Determine the order of rotational symmetry for a NULL-terminated string of ASCII characters.

```
#include <ViennaRNA/combinatorics/basic.h>
```

The algorithm applies a fast search of the provided string within itself, assuming the end of the string wraps around to connect with it's start. For example, a string of the form `AABAAB` has rotational symmetry of order `2`

If the argument `positions` is not `NULL`, the function stores an array of string start positions for rotational shifts that map the string back onto itself. This array has length of order of rotational symmetry, i.e. the number returned by this function. The first element `positions`[0] always contains a shift value of `0` representing the trivial rotation.

SWIG Wrapper Notes:

This function is available as overloaded global function `rotational_symmetry()`. It merges the functionalities of *vrna_rotational_symmetry()*, *vrna_rotational_symmetry_pos()*, *vrna_rotational_symmetry_num()*, and *vrna_rotational_symmetry_pos_num()*. In contrast to our C-implementation, this function doesn't return the order of rotational symmetry as a single value, but returns a list of cyclic permutation shifts that result in a rotationally symmetric string. The length of the list then determines the order of rotational symmetry. See, e.g. *RNA.rotational_symmetry()* in the *The RNA Python module* .

See also

vrna_rotational_symmetry(), *vrna_rotational_symmetry_num()*, vrna_rotational_symmetry_num_pos()

Note

Do not forget to release the memory occupied by `positions` after a successful execution of this function.

Parameters

- **string** – A NULL-terminated string of characters
- **positions** – A pointer to an (undefined) list of alternative string start positions that lead to an identity mapping (may be NULL)

Returns

The order of rotational symmetry

unsigned int **vrna_rotational_symmetry_db**(*vrna_fold_compound_t* *fc, const char *structure)

Determine the order of rotational symmetry for a dot-bracket structure.

```
#include <ViennaRNA/combinatorics/basic.h>
```

Given a (permutation of multiple) RNA strand(s) and a particular secondary structure in dot-bracket notation, compute the degree of rotational symmetry. In case there is only a single linear RNA strand, the structure always has degree 1, as there are no rotational symmetries due to the direction of the nucleic acid sequence and the fixed positions of 5' and 3' ends. However, for circular RNAs, rotational symmetries might arise if the sequence consists of a concatenation of k identical subsequences.

This is a simplified version of *vrna_rotational_symmetry_db_pos()* that may be useful if one is only interested in the degree of rotational symmetry but not the actual set of rotational symmetric strings.

SWIG Wrapper Notes:

This function is attached as method `rotational_symmetry_db()` to objects of type `fold_compound` (i.e. *vrna_fold_compound_t*). See *vrna_rotational_symmetry_db_pos()* for details. See, e.g. *RNA.fold_compound.rotational_symmetry_db()* in the *The RNA Python module* .

See also

vrna_rotational_symmetry_db_pos(), *vrna_rotational_symmetry()*, *vrna_rotational_symmetry_num()*

Parameters

- **fc** – A fold_compound data structure containing the nucleic acid sequence(s), their order, and model settings
- **structure** – The dot-bracket structure the degree of rotational symmetry is checked for

Returns

The degree of rotational symmetry of the `structure` (0 in case of any errors)

unsigned int **vrna_rotational_symmetry_db_pos**(*vrna_fold_compound_t* *fc, const char *structure, unsigned int **positions)

Determine the order of rotational symmetry for a dot-bracket structure.

```
#include <ViennaRNA/combinatorics/basic.h>
```

Given a (permutation of multiple) RNA strand(s) and a particular secondary structure in dot-bracket notation, compute the degree of rotational symmetry. In case there is only a single linear RNA strand, the structure always has degree 1, as there are no rotational symmetries due to the direction of the nucleic acid sequence and the fixed positions of 5' and 3' ends. However, for circular RNAs, rotational symmetries might arise if the sequence consists of a concatenation of k identical subsequences.

If the argument `positions` is not `NULL`, the function stores an array of string start positions for rotational shifts that map the string back onto itself. This array has length of order of rotational symmetry, i.e. the number returned by this function. The first element `positions`[0] always contains a shift value of `0` representing the trivial rotation.

SWIG Wrapper Notes:

This function is attached as method `rotational_symmetry_db()` to objects of type `fold_compound` (i.e. *vrna_fold_compound_t*). Thus, the first argument must be omitted. In contrast to our C-implementation, this function doesn't simply return the order of rotational symmetry of the secondary structure, but returns the list `position` of cyclic permutation shifts that result in a rotationally symmetric structure. The length of the list then determines the order of rotational symmetry. See, e.g. *RNA.fold_compound.rotational_symmetry_db()* in the *The RNA Python module* .

See also

vrna_rotational_symmetry_db(), *vrna_rotational_symmetry_pos()*, *vrna_rotational_symmetry_pos_num()*

Note

Do not forget to release the memory occupied by `positions` after a successful execution of this function.

Parameters

- **fc** – A fold_compound data structure containing the nucleic acid sequence(s), their order, and model settings
- **structure** – The dot-bracket structure the degree of rotational symmetry is checked for
- **positions** – A pointer to an (undefined) list of alternative string start positions that lead to an identity mapping (may be NULL)

Returns

The degree of rotational symmetry of the `structure` (0 in case of any errors)

unsigned int ****vrna_n_multichoose_k**(size_t n, size_t k)

Obtain a list of k-combinations with repetition (n multichoose k).

```
#include <ViennaRNA/combinatorics/basic.h>
```

This function compiles a list of k-combinations, or k-multicombination, i.e. a list of multisubsets of size k from a set of integer values from 0 to n - 1. For that purpose, we enumerate n + k - 1 choose k and decrease each index position i by i to obtain n multichoose k.

Parameters

- **n** – Maximum number to choose from (interval of integers from 0 to n - 1)
- **k** – Number of elements to choose, i.e. size of each multisubset

Returns

A list of lists of elements of combinations (last entry is terminated by **NULL**

unsigned int ***vrna_boustrophedon**(size_t start, size_t end)

Generate a sequence of Boustrophedon distributed numbers.

```
#include <ViennaRNA/combinatorics/basic.h>
```

This function generates a sequence of positive natural numbers within the interval $[start, end]$ in a Boustrophedon fashion. That is, the numbers $start, \ldots, end$ in the resulting list are alternating between left and right ends of the interval while progressing to the inside, i.e. the list consists of a sequence of natural numbers of the form:

$$start, end, start + 1, end - 1, start + 2, end - 2, \ldots$$

The resulting list is 1-based and contains the length of the sequence of numbers at it's 0-th position.

Upon failure, the function returns **NULL**

SWIG Wrapper Notes:

This function is available as overloaded global function `boustrophedon()`. See, e.g. *RNA.boustrophedon()* in the *The RNA Python module* .

See also

vrna_boustrophedon_pos()

Parameters

- **start** – The first number of the list (left side of the interval)
- **end** – The last number of the list (right side of the interval)

Returns

A list of alternating numbers from the interval $[start, end]$ (or **NULL** on error)

unsigned int **vrna_boustrophedon_pos**(size_t start, size_t end, size_t pos)

Obtain the i-th element in a Boustrophedon distributed interval of natural numbers.

```
#include <ViennaRNA/combinatorics/basic.h>
```

SWIG Wrapper Notes:

This function is available as overloaded global function `boustrophedon()`. Omitting the `pos` argument yields the entire sequence from `start` to `end`. See, e.g. *RNA.boustrophedon()* in the *The RNA Python module* .

See also

vrna_boustrophedon()

Parameters

- **start** – The first number of the list (left side of the interval)
- **end** – The last number of the list (right side of the interval)
- **pos** – The index of the number within the Boustrophedon distributed sequence (1-based)

Returns

The `pos-th` element in the Boustrophedon distributed sequence of natural numbers of the interval

4.2.14 Unit Conversion

Functions to convert between various physical units

Enums

enum **vrna_unit_energy_e**

Energy / Work Units.

```
#include <ViennaRNA/utils/units.h>
```

See also

vrna_convert_energy()

Values:

enumerator **VRNA_UNIT_J**

Joule ($1\ J = 1\ kg \cdot m^2 s^{-2}$).

enumerator **VRNA_UNIT_KJ**

Kilojoule ($1\ kJ = 1,000\ J$).

enumerator **VRNA_UNIT_CAL_IT**

Calorie (International (Steam) Table, $1\ cal_{IT} = 4.1868\ J$).

enumerator **VRNA_UNIT_DACAL_IT**

Decacolorie (International (Steam) Table, $1\ dacal_{IT} = 10\ cal_{IT} = 41.868\ J$).

enumerator **VRNA_UNIT_KCAL_IT**

Kilocalorie (International (Steam) Table, $1\ kcal_{IT} = 4.1868\ kJ$).

enumerator **VRNA_UNIT_CAL**

Calorie (Thermochemical, 1 $cal_{th} = 4.184\ J$).

enumerator **VRNA_UNIT_DACAL**

Decacalorie (Thermochemical, 1 $dacal_{th} = 10\ cal_{th} = 41.84\ J$).

enumerator **VRNA_UNIT_KCAL**

Kilocalorie (Thermochemical, 1 $kcal_{th} = 4.184\ kJ$).

enumerator **VRNA_UNIT_G_TNT**

g TNT (1 g TNT $= 1,000\ cal_{th} = 4,184\ J$)

enumerator **VRNA_UNIT_KG_TNT**

kg TNT (1 kg TNT $= 1,000\ kcal_{th} = 4,184\ kJ$)

enumerator **VRNA_UNIT_T_TNT**

ton TNT (1 t TNT $= 1,000,000\ kcal_{th} = 4,184\ MJ$)

enumerator **VRNA_UNIT_EV**

Electronvolt (1 $eV = 1.602176565 \times 10^{-19}\ J$).

enumerator **VRNA_UNIT_WH**

Watt hour (1 $W \cdot h = 1\ W \cdot 3,600 s = 3,600\ J = 3.6\ kJ$).

enumerator **VRNA_UNIT_KWH**

Kilowatt hour (1 $kW \cdot h = 1\ kW \cdot 3,600\ s = 3,600\ kJ = 3.6\ MJ$).

enum **vrna_unit_temperature_e**

Temperature Units.

```
#include <ViennaRNA/utils/units.h>
```

See also

vrna_convert_temperature()

Values:

enumerator **VRNA_UNIT_K**

Kelvin (K).

enumerator **VRNA_UNIT_DEG_C**

Degree Celcius (°C) ($[°C] = [K] - 273.15$).

enumerator **VRNA_UNIT_DEG_F**

Degree Fahrenheit (°F) ($[°F] = [K] \times \frac{9}{5} - 459.67$).

enumerator **VRNA_UNIT_DEG_R**

Degree Rankine (°R) ($[^\circ R] = [K] \times \frac{9}{5}$).

enumerator **VRNA_UNIT_DEG_N**

Degree Newton (°N) ($[^\circ N] = ([K] - 273.15) \times \frac{33}{100}$).

enumerator **VRNA_UNIT_DEG_DE**

Degree Delisle (°De) ($[^\circ De] = (373.15 - [K]) \times \frac{3}{2}$).

enumerator **VRNA_UNIT_DEG_RE**

Degree Réaumur (°Ré) ($[^\circ Ré] = ([K] - 273.15) \times \frac{4}{5}$).

enumerator **VRNA_UNIT_DEG_RO**

Degree Rømer (°Rø) ($[^\circ \text{Rø}] = ([K] - 273.15) \times \frac{21}{40} + 7.5$).

Functions

double **vrna_convert_energy**(double energy, *vrna_unit_energy_e* from, *vrna_unit_energy_e* to)

Convert between energy / work units.

```
#include <ViennaRNA/utils/units.h>
```

See also

vrna_unit_energy_e

Parameters

- **energy** – Input energy value
- **from** – Input unit
- **to** – Output unit

Returns

Energy value in Output unit

double **vrna_convert_temperature**(double temp, *vrna_unit_temperature_e* from, *vrna_unit_temperature_e* to)

Convert between temperature units.

```
#include <ViennaRNA/utils/units.h>
```

See also

vrna_unit_temperature_e

Parameters

- **temp** – Input temperature value
- **from** – Input unit

- **to** – Output unit

Returns
Temperature value in Output unit

int **vrna_convert_kcal_to_dcal**(double energy)

Convert floating point energy value into integer representation.

```
#include <ViennaRNA/utils/units.h>
```

This function converts a floating point value in kcal/mol into its corresponding deka-cal/mol integer representation as used throughout RNAlib.

See also

vrna_convert_dcal_to_kcal()

Parameters

- **energy** – The energy value in kcal/mol

Returns
The energy value in deka-cal/mol

double **vrna_convert_dcal_to_kcal**(int energy)

Convert an integer representation of free energy in deka-cal/mol to kcal/mol.

```
#include <ViennaRNA/utils/units.h>
```

This function converts a free energy value given as integer in deka-cal/mol into the corresponding floating point number in kcal/mol

See also

vrna_convert_kcal_to_dcal()

Parameters

- **energy** – The energy in deka-cal/mol

Returns
The energy in kcal/mol

4.2.15 Plotting

Functions for Creating Secondary Structure Plots, Dot-Plots, and more.

Layouts and Coordinates

Functions to compute coordinate layouts for secondary structure plots.

Defines

VRNA_PLOT_TYPE_SIMPLE

Definition of Plot type *simple*.

```
#include <ViennaRNA/plotting/layouts.h>
```

This is the plot type definition for several RNA structure plotting functions telling them to use **Simple** plotting algorithm

See also

rna_plot_type, *vrna_file_PS_rnaplot_a()*, *vrna_file_PS_rnaplot()*, *svg_rna_plot()*, *gmlRNA()*, *ssv_rna_plot()*, *xrna_plot()*

VRNA_PLOT_TYPE_NAVIEW

Definition of Plot type *Naview*.

```
#include <ViennaRNA/plotting/layouts.h>
```

This is the plot type definition for several RNA structure plotting functions telling them to use **Naview** plotting algorithm [Bruccoleri and Heinrich, 1988] .

See also

rna_plot_type, *vrna_file_PS_rnaplot_a()*, *vrna_file_PS_rnaplot()*, *svg_rna_plot()*, *gmlRNA()*, *ssv_rna_plot()*, *xrna_plot()*

VRNA_PLOT_TYPE_CIRCULAR

Definition of Plot type *Circular*.

```
#include <ViennaRNA/plotting/layouts.h>
```

This is the plot type definition for several RNA structure plotting functions telling them to produce a **Circular plot**

See also

rna_plot_type, *vrna_file_PS_rnaplot_a()*, *vrna_file_PS_rnaplot()*, *svg_rna_plot()*, *gmlRNA()*, *ssv_rna_plot()*, *xrna_plot()*

VRNA_PLOT_TYPE_TURTLE

Definition of Plot type *Turtle* [Wiegreffe *et al.*, 2019] .

```
#include <ViennaRNA/plotting/layouts.h>
```

VRNA_PLOT_TYPE_PUZZLER

Definition of Plot type *RNApuzzler* [Wiegreffe *et al.*, 2019] .

```
#include <ViennaRNA/plotting/layouts.h>
```

VRNA_PLOT_TYPE_DEFAULT

```
#include <ViennaRNA/plotting/layouts.h>
```

Typedefs

typedef struct *vrna_plot_layout_s* **vrna_plot_layout_t**

RNA secondary structure figure layout.

```
#include <ViennaRNA/plotting/layouts.h>
```

See also

vrna_plot_layout(), *vrna_plot_layout_free()*, *vrna_plot_layout_simple()*, *vrna_plot_layout_circular()*, vrna_plot_layout_naview(), *vrna_plot_layout_turtle()*, *vrna_plot_layout_puzzler()*

Functions

vrna_plot_layout_t ***vrna_plot_layout**(const char *structure, unsigned int plot_type)

Create a layout (coordinates, etc.) for a secondary structure plot.

```
#include <ViennaRNA/plotting/layouts.h>
```

This function can be used to create a secondary structure nucleotide layout that is then further processed by an actual plotting function. The layout algorithm can be specified using the `plot_type` parameter, and the following algorithms are currently supported:

- *VRNA_PLOT_TYPE_SIMPLE*
- *VRNA_PLOT_TYPE_NAVIEW*
- *VRNA_PLOT_TYPE_CIRCULAR*
- *VRNA_PLOT_TYPE_TURTLE*
- *VRNA_PLOT_TYPE_PUZZLER*

Passing an unsupported selection leads to the default algorithm *VRNA_PLOT_TYPE_NAVIEW*

See also

vrna_plot_layout_free(), *vrna_plot_layout_simple()*, vrna_plot_layout_naview(), *vrna_plot_layout_circular()*, *vrna_plot_layout_turtle()*, *vrna_plot_layout_puzzler()*, *vrna_plot_coords()*, vrna_file_PS_rnaplot_layout()

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using *vrna_plot_coords()* instead!

Parameters

- **structure** – The secondary structure in dot-bracket notation
- **plot_type** – The layout algorithm to be used

Returns

The layout data structure for the provided secondary structure

vrna_plot_layout_t ***vrna_plot_layout_simple**(const char *structure)

Create a layout (coordinates, etc.) for a *simple* secondary structure plot.

```
#include <ViennaRNA/plotting/layouts.h>
```

This function basically is a wrapper to *vrna_plot_layout()* that passes the `plot_type` *VRNA_PLOT_TYPE_SIMPLE*.

See also

vrna_plot_layout_free(), *vrna_plot_layout()*, vrna_plot_layout_naview(), *vrna_plot_layout_circular()*, *vrna_plot_layout_turtle()*, *vrna_plot_layout_puzzler()*, *vrna_plot_coords_simple()*, vrna_file_PS_rnaplot_layout()

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using *vrna_plot_coords_simple()* instead!

Parameters

- **structure** – The secondary structure in dot-bracket notation

Returns

The layout data structure for the provided secondary structure

vrna_plot_layout_t ***vrna_plot_layout_circular**(const char *structure)

Create a layout (coordinates, etc.) for a *circular* secondary structure plot.

```
#include <ViennaRNA/plotting/layouts.h>
```

This function basically is a wrapper to *vrna_plot_layout()* that passes the `plot_type` *VRNA_PLOT_TYPE_CIRCULAR*.

See also

vrna_plot_layout_free(), *vrna_plot_layout()*, vrna_plot_layout_naview(), *vrna_plot_layout_simple()*, *vrna_plot_layout_turtle()*, *vrna_plot_layout_puzzler()*, *vrna_plot_coords_circular()*, vrna_file_PS_rnaplot_layout()

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using *vrna_plot_coords_circular()* instead!

Parameters

- **structure** – The secondary structure in dot-bracket notation

Returns

The layout data structure for the provided secondary structure

vrna_plot_layout_t ***vrna_plot_layout_turtle**(const char *structure)

Create a layout (coordinates, etc.) for a secondary structure plot using the *Turtle Algorithm* [Wiegreffe *et al.*, 2019] .

```
#include <ViennaRNA/plotting/layouts.h>
```

This function basically is a wrapper to *vrna_plot_layout()* that passes the `plot_type` *VRNA_PLOT_TYPE_TURTLE*.

See also

vrna_plot_layout_free(), *vrna_plot_layout()*, *vrna_plot_layout_simple()*, *vrna_plot_layout_circular()*, vrna_plot_layout_naview(), *vrna_plot_layout_puzzler()*, *vrna_plot_coords_turtle()*, vrna_file_PS_rnaplot_layout()

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using *vrna_plot_coords_turtle()* instead!

Parameters

- **structure** – The secondary structure in dot-bracket notation

Returns

The layout data structure for the provided secondary structure

vrna_plot_layout_t ***vrna_plot_layout_puzzler**(const char *structure, *vrna_plot_options_puzzler_t* *options)

Create a layout (coordinates, etc.) for a secondary structure plot using the *RNApuzzler Algorithm* [Wiegreffe *et al.*, 2019] .

```
#include <ViennaRNA/plotting/layouts.h>
```

This function basically is a wrapper to *vrna_plot_layout()* that passes the `plot_type` *VRNA_PLOT_TYPE_PUZZLER*.

See also

vrna_plot_layout_free(), *vrna_plot_layout()*, *vrna_plot_layout_simple()*, *vrna_plot_layout_circular()*, vrna_plot_layout_naview(), *vrna_plot_layout_turtle()*, *vrna_plot_coords_puzzler()*, vrna_file_PS_rnaplot_layout()

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using *vrna_plot_coords_puzzler()* instead!

Parameters

- **structure** – The secondary structure in dot-bracket notation

Returns

The layout data structure for the provided secondary structure

void **vrna_plot_layout_free**(*vrna_plot_layout_t* *layout)

Free memory occupied by a figure layout data structure.

```
#include <ViennaRNA/plotting/layouts.h>
```

See also

vrna_plot_layout_t, *vrna_plot_layout()*, *vrna_plot_layout_simple()*, *vrna_plot_layout_circular()*, vrna_plot_layout_naview(), *vrna_plot_layout_turtle()*, *vrna_plot_layout_puzzler()*, vrna_file_PS_rnaplot_layout()

Parameters

- **layout** – The layout data structure to free

int **vrna_plot_coords**(const char *structure, float **x, float **y, int plot_type)

Compute nucleotide coordinates for secondary structure plot.

```
#include <ViennaRNA/plotting/layouts.h>
```

This function takes a secondary structure and computes X-Y coordinates for each nucleotide that then can be used to create a structure plot. The parameter `plot_type` is used to select the underlying layout algorithm. Currently, the following selections are provided:

- *VRNA_PLOT_TYPE_SIMPLE*
- *VRNA_PLOT_TYPE_NAVIEW*
- *VRNA_PLOT_TYPE_CIRCULAR*
- *VRNA_PLOT_TYPE_TURTLE*
- *VRNA_PLOT_TYPE_PUZZLER*

Passing an unsupported selection leads to the default algorithm *VRNA_PLOT_TYPE_NAVIEW*

Here is a simple example how to use this function, assuming variable `structure` contains a valid dot-bracket string:

```
float *x, *y;

if (vrna_plot_coords(structure, &x, &y)) {
  printf("all fine");
} else {
  printf("some failure occured!");
}

free(x);
free(y);
```

See also

vrna_plot_coords_pt(), *vrna_plot_coords_simple()*, vrna_plot_coords_naview() *vrna_plot_coords_circular()*, *vrna_plot_coords_turtle()*, *vrna_plot_coords_puzzler()*

Note

On success, this function allocates memory for X and Y coordinates and assigns the pointers at addressess `x` and `y` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **structure** – The secondary structure in dot-bracket notation
- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)
- **plot_type** – The layout algorithm to be used

Returns

The length of the structure on success, 0 otherwise

int **vrna_plot_coords_pt**(const short *pt, float **x, float **y, int plot_type)

Compute nucleotide coordinates for secondary structure plot.

```
#include <ViennaRNA/plotting/layouts.h>
```

Same as *vrna_plot_coords()* but takes a pair table with the structure information as input.

See also

vrna_plot_coords(), *vrna_plot_coords_simple_pt()*, vrna_plot_coords_naview_pt() *vrna_plot_coords_circular_pt()*, *vrna_plot_coords_turtle_pt()*, *vrna_plot_coords_puzzler_pt()*

Note

On success, this function allocates memory for X and Y coordinates and assigns the pointers at addressess `x` and `y` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **pt** – The pair table that holds the secondary structure
- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)
- **plot_type** – The layout algorithm to be used

Returns

The length of the structure on success, 0 otherwise

int **vrna_plot_coords_simple**(const char *structure, float **x, float **y)

Compute nucleotide coordinates for secondary structure plot the *Simple way*.

```
#include <ViennaRNA/plotting/layouts.h>
```

This function basically is a wrapper to *vrna_plot_coords()* that passes the `plot_type` *VRNA_PLOT_TYPE_SIMPLE*.

Here is a simple example how to use this function, assuming variable `structure` contains a valid dot-bracket string:

```
float *x, *y;

if (vrna_plot_coords_simple(structure, &x, &y)) {
  printf("all fine");
} else {
  printf("some failure occured!");
}

free(x);
free(y);
```

See also

vrna_plot_coords(), *vrna_plot_coords_simple_pt()*, *vrna_plot_coords_circular()*, vrna_plot_coords_naview(), *vrna_plot_coords_turtle()*, *vrna_plot_coords_puzzler()*

Note

On success, this function allocates memory for X and Y coordinates and assigns the pointers at addressess `x` and `y` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **structure** – The secondary structure in dot-bracket notation

- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)

Returns
The length of the structure on success, 0 otherwise

int **vrna_plot_coords_simple_pt**(const short *pt, float **x, float **y)

Compute nucleotide coordinates for secondary structure plot the *Simple way*.

```
#include <ViennaRNA/plotting/layouts.h>
```

Same as *vrna_plot_coords_simple()* but takes a pair table with the structure information as input.

See also

vrna_plot_coords_pt(), *vrna_plot_coords_simple()*, *vrna_plot_coords_circular_pt()*, vrna_plot_coords_naview_pt(), *vrna_plot_coords_turtle_pt()*, *vrna_plot_coords_puzzler_pt()*

Note

On success, this function allocates memory for X and Y coordinates and assigns the pointers at addressess **x** and **y** to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **pt** – The pair table that holds the secondary structure
- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)

Returns
The length of the structure on success, 0 otherwise

int **vrna_plot_coords_circular**(const char *structure, float **x, float **y)

Compute coordinates of nucleotides mapped in equal distancies onto a unit circle.

```
#include <ViennaRNA/plotting/layouts.h>
```

This function basically is a wrapper to *vrna_plot_coords()* that passes the `plot_type` *VRNA_PLOT_TYPE_CIRCULAR*.

In order to draw nice arcs using quadratic bezier curves that connect base pairs one may calculate a second tangential point P^t in addition to the actual R^2 coordinates. the simplest way to do so may be to compute a radius scaling factor rs in the interval $[0, 1]$ that weights the proportion of base pair span to the actual length of the sequence. This scaling factor can then be used to calculate the coordinates for P^t, i.e.

$$P_x^t[i] = X[i] * rs$$

and

$$P_y^t[i] = Y[i] * rs$$

.

See also

vrna_plot_coords(), *vrna_plot_coords_circular_pt()*, *vrna_plot_coords_simple()*, vrna_plot_coords_naview(), *vrna_plot_coords_turtle()*, *vrna_plot_coords_puzzler()*

Note

On success, this function allocates memory for X and Y coordinates and assigns the pointers at addressess `x` and `y` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **`structure`** – The secondary structure in dot-bracket notation
- **`x`** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **`y`** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)

Returns

The length of the structure on success, 0 otherwise

int **`vrna_plot_coords_circular_pt`**(const short *pt, float **x, float **y)

Compute nucleotide coordinates for a *Circular Plot*.

```
#include <ViennaRNA/plotting/layouts.h>
```

Same as *vrna_plot_coords_circular()* but takes a pair table with the structure information as input.

See also

vrna_plot_coords_pt(), *vrna_plot_coords_circular()*, *vrna_plot_coords_simple_pt()*, vrna_plot_coords_naview_pt(), *vrna_plot_coords_turtle_pt()*, *vrna_plot_coords_puzzler_pt()*

Note

On success, this function allocates memory for X and Y coordinates and assigns the pointers at addressess `x` and `y` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **`pt`** – The pair table that holds the secondary structure

- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)

Returns
The length of the structure on success, 0 otherwise

int **vrna_plot_coords_puzzler**(const char *structure, float **x, float **y, double **arc_coords, *vrna_plot_options_puzzler_t* *options)

Compute nucleotide coordinates for secondary structure plot using the *RNApuzzler* algorithm [Wiegreffe *et al.*, 2019] .

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

This function basically is a wrapper to *vrna_plot_coords()* that passes the `plot_type` *VRNA_PLOT_TYPE_PUZZLER*.

Here is a simple example how to use this function, assuming variable `structure` contains a valid dot-bracket string and using the default options (`options` = NULL):

```
float  *x, *y;
double *arcs;

if (vrna_plot_coords_puzzler(structure, &x, &y, &arcs, NULL)) {
  printf("all fine");
} else {
  printf("some failure occured!");
}

free(x);
free(y);
free(arcs);
```

See also

vrna_plot_coords(), *vrna_plot_coords_puzzler_pt()*, *vrna_plot_coords_circular()*, *vrna_plot_coords_simple()*, *vrna_plot_coords_turtle()*, vrna_plot_coords_naview(), *vrna_plot_options_puzzler()*

Note

On success, this function allocates memory for X, Y and arc coordinates and assigns the pointers at addressess `x`, `y` and `arc_coords` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **structure** – The secondary structure in dot-bracket notation
- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)

- **arc_coords** – **[inout]** The address of a pointer that will hold arc coordinates (pointer will point to memory, or NULL on failure)
- **options** – The options for the RNApuzzler algorithm (or NULL)

Returns
The length of the structure on success, 0 otherwise

int **vrna_plot_coords_puzzler_pt**(short const *const pt, float **x, float **y, double **arc_coords, *vrna_plot_options_puzzler_t* *options)

Compute nucleotide coordinates for secondary structure plot using the *RNApuzzler* algorithm [Wiegreffe *et al.*, 2019] .

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

Same as *vrna_plot_coords_puzzler()* but takes a pair table with the structure information as input.

See also

vrna_plot_coords_pt(), *vrna_plot_coords_puzzler()*, *vrna_plot_coords_circular_pt()*, *vrna_plot_coords_simple_pt()*, *vrna_plot_coords_turtle_pt()*, vrna_plot_coords_naview_pt()

Note

On success, this function allocates memory for X, Y and arc coordinates and assigns the pointers at addressess `x`, `y` and `arc_coords` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **pt** – The pair table that holds the secondary structure
- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)
- **arc_coords** – **[inout]** The address of a pointer that will hold arc coordinates (pointer will point to memory, or NULL on failure)
- **options** – The options for the RNApuzzler algorithm (or NULL)

Returns
The length of the structure on success, 0 otherwise

vrna_plot_options_puzzler_t ***vrna_plot_options_puzzler**(void)

Create an RNApuzzler options data structure.

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

See also

vrna_plot_options_puzzler_free(), *vrna_plot_coords_puzzler()*, *vrna_plot_coords_puzzler_pt()*, *vrna_plot_layout_puzzler()*

Returns

An RNApuzzler options data structure with default settings

void **vrna_plot_options_puzzler_free**(*vrna_plot_options_puzzler_t* *options)

Free memory occupied by an RNApuzzler options data structure.

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

See also

vrna_plot_options_puzzler(), *vrna_plot_coords_puzzler()*, *vrna_plot_coords_puzzler_pt()*, *vrna_plot_layout_puzzler()*

Parameters

- **options** – A pointer to the options data structure to free

int **vrna_plot_coords_turtle**(const char *structure, float **x, float **y, double **arc_coords)

Compute nucleotide coordinates for secondary structure plot using the *RNAturtle* algorithm [Wiegreffe *et al.*, 2019] .

```
#include <ViennaRNA/plotting/RNApuzzler/RNAturtle.h>
```

This function basically is a wrapper to *vrna_plot_coords()* that passes the `plot_type` *VRNA_PLOT_TYPE_TURTLE*.

Here is a simple example how to use this function, assuming variable `structure` contains a valid dot-bracket string:

```
float  *x, *y;
double *arcs;

if (vrna_plot_coords_turtle(structure, &x, &y, &arcs)) {
  printf("all fine");
} else {
  printf("some failure occured!");
}

free(x);
free(y);
free(arcs);
```

See also

vrna_plot_coords(), *vrna_plot_coords_turtle_pt()*, *vrna_plot_coords_circular()*, *vrna_plot_coords_simple()*, vrna_plot_coords_naview(), *vrna_plot_coords_puzzler()*

Note

On success, this function allocates memory for X, Y and arc coordinates and assigns the pointers at addressess `x`, `y` and `arc_coords` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **structure** – The secondary structure in dot-bracket notation
- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)
- **arc_coords** – **[inout]** The address of a pointer that will hold arc coordinates (pointer will point to memory, or NULL on failure)

Returns

The length of the structure on success, 0 otherwise

int **vrna_plot_coords_turtle_pt**(short const *const pt, float **x, float **y, double **arc_coords)

Compute nucleotide coordinates for secondary structure plot using the *RNAturtle* algorithm [Wiegreffe *et al.*, 2019] .

```
#include <ViennaRNA/plotting/RNApuzzler/RNAturtle.h>
```

Same as *vrna_plot_coords_turtle()* but takes a pair table with the structure information as input.

See also

vrna_plot_coords_pt(), *vrna_plot_coords_turtle()*, *vrna_plot_coords_circular_pt()*, *vrna_plot_coords_simple_pt()*, *vrna_plot_coords_puzzler_pt()*, vrna_plot_coords_naview_pt()

Note

On success, this function allocates memory for X, Y and arc coordinates and assigns the pointers at addressess `x`, `y` and `arc_coords` to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

Parameters

- **pt** – The pair table that holds the secondary structure
- **x** – **[inout]** The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** – **[inout]** The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)
- **arc_coords** – **[inout]** The address of a pointer that will hold arc coordinates (pointer will point to memory, or NULL on failure)

Returns

The length of the structure on success, 0 otherwise

struct **vrna_plot_layout_s**

Public Members

unsigned int **type**

```
#include <ViennaRNA/plotting/layouts.h>
```

unsigned int **length**

```
#include <ViennaRNA/plotting/layouts.h>
```

float ***x**

```
#include <ViennaRNA/plotting/layouts.h>
```

float ***y**

```
#include <ViennaRNA/plotting/layouts.h>
```

double ***arcs**

```
#include <ViennaRNA/plotting/layouts.h>
```

int **bbox**[4]

```
#include <ViennaRNA/plotting/layouts.h>
```

struct **vrna_plot_options_puzzler_t**

#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h> Options data structure for RNApuzzler algorithm implementation.

Public Members

short **drawArcs**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

double **paired**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

double **unpaired**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

short **checkAncestorIntersections**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

short **checkSiblingIntersections**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

short **checkExteriorIntersections**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

short **allowFlipping**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

short **optimize**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

int **maximumNumberOfConfigChangesAllowed**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

char ***config**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

const char ***filename**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

int **numberOfChangesAppliedToConfig**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

int **psNumber**

```
#include <ViennaRNA/plotting/RNApuzzler/RNApuzzler.h>
```

Annotation

Functions to generate annotations for secondary structure plots, dot-plots, and others.

Functions

char ****vrna_annotate_covar_db**(const char **alignment, const char *structure, *vrna_md_t* *md_p)

Produce covariance annotation for an alignment given a secondary structure.

```
#include <ViennaRNA/plotting/utils.h>
```

char **vrna_annotate_covar_db_extended(const char **alignment, const char *structure, *vrna_md_t* *md_p, unsigned int options)

```
#include <ViennaRNA/plotting/utils.h>
```

vrna_string_t ***vrna_annotate_covar_pt**(const char **alignment, const short int *pt, *vrna_md_t* *md_p, double threshold, double min_sat)

```
#include <ViennaRNA/plotting/utils.h>
```

vrna_cpair_t ***vrna_annotate_covar_pairs**(const char **alignment, *vrna_ep_t* *pl, *vrna_ep_t* *mfel, double threshold, *vrna_md_t* *md)

Produce covariance annotation for an alignment given a set of base pairs.

```
#include <ViennaRNA/plotting/utils.h>
```

Pair Probability Plots

Functions related to plotting of probabilities, such as dot-plots.

Defines

VRNA_PLOT_PROBABILITIES_BP

Option flag for base pair probabilities in probability plot output functions.

```
#include <ViennaRNA/plotting/probabilities.h>
```

VRNA_PLOT_PROBABILITIES_ACC

Option flag for accessibilities in probability plot output functions.

```
#include <ViennaRNA/plotting/probabilities.h>
```

VRNA_PLOT_PROBABILITIES_UD

Option flag for unstructured domain probabilities in probability plot output functions.

```
#include <ViennaRNA/plotting/probabilities.h>
```

VRNA_PLOT_PROBABILITIES_UD_LIN

Option flag for unstructured domain probabilities (linear representation) in probability plot output functions.

```
#include <ViennaRNA/plotting/probabilities.h>
```

VRNA_PLOT_PROBABILITIES_SD

Option flag for structured domain probabilities (such as G-quadruplexes) in probability plot output functions.

```
#include <ViennaRNA/plotting/probabilities.h>
```

VRNA_PLOT_PROBABILITIES_SC_MOTIF

Option flag for soft-constraint motif probabilities in probability plot output functions.

```
#include <ViennaRNA/plotting/probabilities.h>
```

VRNA_PLOT_PROBABILITIES_SC_UP

```
#include <ViennaRNA/plotting/probabilities.h>
```

VRNA_PLOT_PROBABILITIES_SC_BP

```
#include <ViennaRNA/plotting/probabilities.h>
```

VRNA_PLOT_PROBABILITIES_DEFAULT

Default option flag for probability plot output functions.

```
#include <ViennaRNA/plotting/probabilities.h>
```

Default output includes actual base pair probabilties (*VRNA_PLOT_PROBABILITIES_BP*), structured domain probabilities such as G-quadruplexes (*VRNA_PLOT_PROBABILITIES_SD*), probabilities obtained from soft-constraint motif implementation (*VRNA_PLOT_PROBABILITIES_SC_MOTIF*), and unstructured domain probabilities (*VRNA_PLOT_PROBABILITIES_UD_LIN*).

See also

vrna_plot_dp_EPS()

Functions

int **vrna_plot_dp_EPS**(const char *filename, const char *sequence, *vrna_ep_t* *upper, *vrna_ep_t* *lower, *vrna_dotplot_auxdata_t* *auxdata, unsigned int options)

Produce an encapsulate PostScript (EPS) dot-plot from one or two lists of base pair probabilities.

```
#include <ViennaRNA/plotting/probabilities.h>
```

This function reads two *vrna_ep_t* lists `upper` and `lower` (e.g. base pair probabilities and a secondary structure) and produces an EPS "dot plot" with filename `'filename'` where data from `upper` is placed in the upper-triangular and data from `lower` is placed in the lower triangular part of the matrix.

For default output, provide the flag *VRNA_PLOT_PROBABILITIES_DEFAULT* as `options` parameter.

SWIG Wrapper Notes:

This function is available as overloaded function `plot_dp_EPS()` where the last three parameters may be omitted. The default values for these parameters are `lower` = NULL, `auxdata` = NULL, `options` = *VRNA_PLOT_PROBABILITIES_DEFAULT*. See, e.g. *RNA.plot_dp_EPS()* in the *The RNA Python module*.

See also

vrna_plist(), *vrna_plist_from_probs()*, *VRNA_PLOT_PROBABILITIES_DEFAULT*

Parameters

- **filename** – A filename for the EPS output
- **sequence** – The RNA sequence
- **upper** – The base pair probabilities for the upper triangular part
- **lower** – The base pair probabilities for the lower triangular part
- **options** – Options indicating which of the input data should be included in the dot-plot

Returns

1 if EPS file was successfully written, 0 otherwise

int **vrna_plot_dp_PS_list**(char *seq, int cp, char *filename, *vrna_ep_t* *pl, *vrna_ep_t* *mf, char *comment)

Produce a postscript dot-plot from two pair lists.

```
#include <ViennaRNA/plotting/probabilities.h>
```

This function reads two plist structures (e.g. base pair probabilities and a secondary structure) as produced by *vrna_plist_from_probs()* and *vrna_plist()* and produces a postscript "dot plot" that is written to 'filename'.

Using base pair probabilities in the first and mfe structure in the second plist, the resulting "dot plot" represents each base pairing probability by a square of corresponding area in a upper triangle matrix. The lower part of the matrix contains the minimum free energy structure.

See also

vrna_plist_from_probs(), *vrna_plist()*

Parameters

- **seq** – The RNA sequence
- **filename** – A filename for the postscript output
- **pl** – The base pair probability pairlist
- **mf** – The mfe secondary structure pairlist
- **comment** – A comment

Returns

1 if postscript was successfully written, 0 otherwise

struct **vrna_dotplot_auxdata_t**

Public Members

char *comment

```
#include <ViennaRNA/plotting/probabilities.h>
```

char *title

```
#include <ViennaRNA/plotting/probabilities.h>
```

vrna_data_lin_t **top

```
#include <ViennaRNA/plotting/probabilities.h>
```

char **top_title

```
#include <ViennaRNA/plotting/probabilities.h>
```

vrna_data_lin_t **bottom

```
#include <ViennaRNA/plotting/probabilities.h>
```

char **bottom_title

```
#include <ViennaRNA/plotting/probabilities.h>
```

vrna_data_lin_t **left

```
#include <ViennaRNA/plotting/probabilities.h>
```

char **left_title

```
#include <ViennaRNA/plotting/probabilities.h>
```

vrna_data_lin_t **right

```
#include <ViennaRNA/plotting/probabilities.h>
```

char **right_title

```
#include <ViennaRNA/plotting/probabilities.h>
```

Alignment Plots

Functions to generate Alignment plots with annotated consensus structure.

Functions

int **vrna_file_PS_aln**(const char *filename, const char **seqs, const char **names, const char *structure, unsigned int columns)

Create an annotated PostScript alignment plot.

```
#include <ViennaRNA/plotting/alignments.h>
```

SWIG Wrapper Notes:

This function is available as overloaded function `file_PS_aln()` with three additional parameters `start`, `end`, and `offset` before the `columns` argument. Thus, it resembles the *vrna_file_PS_aln_slice()* function. The last four arguments may be omitted, indicating the default of `start` = *0*, `end` = *0*, `offset` = *0*, and `columns` = *60*. See, e.g. *RNA.file_PS_aln()* in the *The RNA Python module*.

See also

vrna_file_PS_aln_slice()

Parameters

- **filename** – The output file name
- **seqs** – The aligned sequences
- **names** – The names of the sequences
- **structure** – The consensus structure in dot-bracket notation
- **columns** – The number of columns before the alignment is wrapped as a new block (a value of 0 indicates no wrapping)

int **vrna_file_PS_aln_slice**(const char *filename, const char **seqs, const char **names, const char *structure, unsigned int start, unsigned int end, int offset, unsigned int columns)

Create an annotated PostScript alignment plot.

```
#include <ViennaRNA/plotting/alignments.h>
```

Similar to *vrna_file_PS_aln()* but allows the user to print a particular slice of the alignment by specifying a `start` and `end` position. The additional `offset` parameter allows for adjusting the alignment position ruler value.

SWIG Wrapper Notes:

This function is available as overloaded function `file_PS_aln()` where the last four parameter may be omitted, indicating `start` = *0*, `end` = *0*, `offset` = *0*, and `columns` = *60*. See, e.g. *RNA.file_PS_aln()* in the *The RNA Python module*.

See also

vrna_file_PS_aln_slice()

Parameters

- **filename** – The output file name
- **seqs** – The aligned sequences

- **names** – The names of the sequences
- **structure** – The consensus structure in dot-bracket notation
- **start** – The start of the alignment slice (a value of 0 indicates the first position of the alignment, i.e. no slicing at 5' side)
- **end** – The end of the alignment slice (a value of 0 indicates the last position of the alignment, i.e. no slicing at 3' side)
- **offset** – The alignment coordinate offset for the position ruler.
- **columns** – The number of columns before the alignment is wrapped as a new block (a value of 0 indicates no wrapping)

int **vrna_file_PS_aln_opt**(const char *filename, const char **seqs, const char **names, const char *structure, *vrna_aln_opt_t* options)

```
#include <ViennaRNA/plotting/alignments.h>
```

struct **vrna_aln_opt_t**

Public Members

unsigned int **start**

```
#include <ViennaRNA/plotting/alignments.h>
```

unsigned int **end**

```
#include <ViennaRNA/plotting/alignments.h>
```

unsigned int **offset**

```
#include <ViennaRNA/plotting/alignments.h>
```

unsigned int **columns**

```
#include <ViennaRNA/plotting/alignments.h>
```

double **color_threshold**

```
#include <ViennaRNA/plotting/alignments.h>
```

double **color_min_sat**

```
#include <ViennaRNA/plotting/alignments.h>
```

Other Plotting Utils

Defines

VRNA_FILE_FORMAT_EPS

```
#include <ViennaRNA/plotting/structures.h>
```

VRNA_FILE_FORMAT_SVG

```
#include <ViennaRNA/plotting/structures.h>
```

VRNA_FILE_FORMAT_GML

```
#include <ViennaRNA/plotting/structures.h>
```

VRNA_FILE_FORMAT_SSV

```
#include <ViennaRNA/plotting/structures.h>
```

VRNA_FILE_FORMAT_XRNA

```
#include <ViennaRNA/plotting/structures.h>
```

VRNA_FILE_FORMAT_PLOT_DEFAULT

```
#include <ViennaRNA/plotting/structures.h>
```

Typedefs

typedef struct *vrna_plot_data_s* **vrna_plot_data_t**

```
#include <ViennaRNA/plotting/structures.h>
```

Functions

int **vrna_plot_structure**(const char *filename, const char *sequence, const char *structure, unsigned int file_format, *vrna_plot_layout_t* *layout, *vrna_plot_data_t* *aux_data)

```
#include <ViennaRNA/plotting/structures.h>
```

int **vrna_plot_structure_svg**(const char *filename, const char *sequence, const char *structure, *vrna_plot_layout_t* *layout, *vrna_plot_data_t* *data)

```
#include <ViennaRNA/plotting/structures.h>
```

int **vrna_plot_structure_eps**(const char *filename, const char *sequence, const char *structure, *vrna_plot_layout_t* *layout, *vrna_plot_data_t* *data)

```
#include <ViennaRNA/plotting/structures.h>
```

int **vrna_plot_structure_gml**(const char *filename, const char *sequence, const char *structure, *vrna_plot_layout_t* *layout, *vrna_plot_data_t* *data, char option)

```
#include <ViennaRNA/plotting/structures.h>
```

int **vrna_plot_structure_ssv**(const char *filename, const char *sequence, const char *structure, *vrna_plot_layout_t* *layout, *vrna_plot_data_t* *data)

```
#include <ViennaRNA/plotting/structures.h>
```

int **vrna_plot_structure_xrna**(const char *filename, const char *sequence, const char *structure, *vrna_plot_layout_t* *layout, *vrna_plot_data_t* *data)

```
#include <ViennaRNA/plotting/structures.h>
```

int **vrna_file_PS_rnaplot**(const char *seq, const char *structure, const char *file, *vrna_md_t* *md_p)

Produce a secondary structure graph in PostScript and write it to 'filename'.

```
#include <ViennaRNA/plotting/structures.h>
```

Note that this function has changed from previous versions and now expects the structure to be plotted in dot-bracket notation as an argument. It does not make use of the global base_pair array anymore.

Parameters

- **seq** – The RNA sequence
- **structure** – The secondary structure in dot-bracket notation
- **file** – The filename of the postscript output
- **md_p** – Model parameters used to generate a commandline option string in the output (Maybe NULL)

Returns

1 on success, 0 otherwise

int **vrna_file_PS_rnaplot_a**(const char *seq, const char *structure, const char *file, const char *pre, const char *post, *vrna_md_t* *md_p)

Produce a secondary structure graph in PostScript including additional annotation macros and write it to 'filename'.

```
#include <ViennaRNA/plotting/structures.h>
```

Same as *vrna_file_PS_rnaplot()* but adds extra PostScript macros for various annotations (see generated PS code). The 'pre' and 'post' variables contain PostScript code that is verbatim copied in the resulting PS file just before and after the structure plot. If both arguments ('pre' and 'post') are NULL, no additional macros will be printed into the PostScript.

Parameters

- **seq** – The RNA sequence
- **structure** – The secondary structure in dot-bracket notation
- **file** – The filename of the postscript output
- **pre** – PostScript code to appear before the secondary structure plot
- **post** – PostScript code to appear after the secondary structure plot
- **md_p** – Model parameters used to generate a commandline option string in the output (Maybe NULL)

Returns

1 on success, 0 otherwise

int **vrna_file_PS_rnaplot_layout**(const char *seq, const char *structure, const char *ssfile, const char *pre, const char *post, *vrna_md_t* *md_p, *vrna_plot_layout_t* *layout)

```
#include <ViennaRNA/plotting/structures.h>
```

int **PS_rna_plot_snoop_a**(const char *string, const char *structure, const char *ssfile, int *relative_access, const char *seqs[])

```
#include <ViennaRNA/plotting/structures.h>
```

int **gmlRNA**(char *string, char *structure, char *ssfile, char option)

Produce a secondary structure graph in Graph Meta Language (gml) and write it to a file.

```
#include <ViennaRNA/plotting/structures.h>
```

If 'option' is an uppercase letter the RNA sequence is used to label nodes, if 'option' equals *'X'* or *'x'* the resulting file will coordinates for an initial layout of the graph.

Parameters

- **string** – The RNA sequence
- **structure** – The secondary structure in dot-bracket notation
- **ssfile** – The filename of the gml output
- **option** – The option flag

Returns

1 on success, 0 otherwise

int **ssv_rna_plot**(char *string, char *structure, char *ssfile)

Produce a secondary structure graph in SStructView format.

```
#include <ViennaRNA/plotting/structures.h>
```

Write coord file for SStructView

Parameters

- **string** – The RNA sequence
- **structure** – The secondary structure in dot-bracket notation
- **ssfile** – The filename of the ssv output

Returns

1 on success, 0 otherwise

int **svg_rna_plot**(char *string, char *structure, char *ssfile)

Produce a secondary structure plot in SVG format and write it to a file.

```
#include <ViennaRNA/plotting/structures.h>
```

Parameters

- **string** – The RNA sequence
- **structure** – The secondary structure in dot-bracket notation
- **ssfile** – The filename of the svg output

Returns

1 on success, 0 otherwise

int **xrna_plot**(char *string, char *structure, char *ssfile)

Produce a secondary structure plot for further editing in XRNA.

```
#include <ViennaRNA/plotting/structures.h>
```

Parameters

- **string** – The RNA sequence
- **structure** – The secondary structure in dot-bracket notation
- **ssfile** – The filename of the xrna output

Returns
1 on success, 0 otherwise

int **PS_rna_plot**(char *string, char *structure, char *file)

Produce a secondary structure graph in PostScript and write it to 'filename'.

```
#include <ViennaRNA/plotting/structures.h>
```

Deprecated:

Use *vrna_file_PS_rnaplot()* instead!

int **PS_rna_plot_a**(char *string, char *structure, char *file, char *pre, char *post)

Produce a secondary structure graph in PostScript including additional annotation macros and write it to 'filename'.

```
#include <ViennaRNA/plotting/structures.h>
```

Deprecated:

Use *vrna_file_PS_rnaplot_a()* instead!

int **PS_rna_plot_a_gquad**(char *string, char *structure, char *ssfile, char *pre, char *post)

Produce a secondary structure graph in PostScript including additional annotation macros and write it to 'filename' (detect and draw g-quadruplexes).

```
#include <ViennaRNA/plotting/structures.h>
```

Deprecated:

Use *vrna_file_PS_rnaplot_a()* instead!

struct **vrna_plot_data_s**

Public Members

char ***pre**

```
#include <ViennaRNA/plotting/structures.h>
```

char ***post**

```
#include <ViennaRNA/plotting/structures.h>
```

vrna_md_t ***md**

```
#include <ViennaRNA/plotting/structures.h>
```

unsigned int **options**

```
#include <ViennaRNA/plotting/structures.h>
```

4.3 SWIG Wrappers

4.3.1 SWIG Wrapper Notes

Special notes on how functions, structures, enums, and macro definitions are actually wrapped, can be found below

Global *vrna_abstract_shapes* (const char *structure, unsigned int level)

This function is available as an overloaded function `abstract_shapes()` where the optional second parameter `level` defaults to 5. See, e.g. *RNA.abstract_shapes()* in the *The RNA Python module*.

Global *vrna_abstract_shapes_pt* (const short *pt, unsigned int level)

This function is available as an overloaded function `abstract_shapes()` where the optional second parameter `level` defaults to 5. See, e.g. *RNA.abstract_shapes()* in the *The RNA Python module*.

Global *vrna_alifold* (const char **sequences, char *structure)

This function is available as function *alifold()* in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.alifold()* in the *The RNA Python module*.

Global *vrna_aliLfold* (const char **alignment, int maxdist, FILE *fp)

This function is available as overloaded function `aliLfold()` in the global namespace. The parameter `fp` defaults to `NULL` and may be omitted. See e.g. *RNA.aliLfold()* in the *The RNA Python module*.

Global *vrna_aliLfold_cb* (const char **alignment, int maxdist, *vrna_mfe_window_f* cb, void *data)

This function is available as overloaded function `aliLfold_cb()` in the global namespace. The parameter `data` defaults to `NULL` and may be omitted. See e.g. *RNA.aliLfold_cb()* in the *The RNA Python module*.

Global *vrna_aln_consensus_mis* (const char **alignment, const *vrna_md_t* *md_p)

This function is available as overloaded function `aln_consensus_mis()` where the last parameter may be omitted, indicating `md = NULL`. See e.g. *RNA.aln_consensus_mis()* in the *The RNA Python module*.

Global *vrna_aln_consensus_sequence* (const char **alignment, const *vrna_md_t* *md_p)

This function is available as overloaded function `aln_consensus_sequence()` where the last parameter may be omitted, indicating `md = NULL`. See e.g. *RNA.aln_consensus_sequence()* in the *The RNA Python module*.

Global *vrna_aln_conservation_col* (const char **alignment, const *vrna_md_t* *md_p, unsigned int options)

This function is available as overloaded function `aln_conservation_col()` where the last two parameters may be omitted, indicating `md = NULL`, and `options = VRNA_MEASURE_SHANNON_ENTROPY`, respectively. See e.g. *RNA.aln_conservation_col()* in the *The RNA Python module*.

Global *vrna_aln_conservation_struct* (const char **alignment, const char *structure, const *vrna_md_t* *md)

This function is available as overloaded function `aln_conservation_struct()` where the last parameter `md` may be omitted, indicating `md = NULL`. See, e.g. *RNA.aln_conservation_struct()* in the *The RNA Python module*.

Global *vrna_aln_mpi* (const char **alignment)

This function is available as function `aln_mpi()`. See e.g. *RNA.aln_mpi()* in the *The RNA Python module*.

Global *vrna_aln_pscore* (const char **alignment, *vrna_md_t* *md)

This function is available as overloaded function `aln_pscore()` where the last parameter may be omitted, indicating `md = NULL`. See e.g. *RNA.aln_pscore()* in the *The RNA Python module*.

Global *vrna_backtrack5* (*vrna_fold_compound_t* *fc, unsigned int length, char *structure)

This function is attached as overloaded method `backtrack()` to objects of type *The Fold Compound*. The parameter `length` defaults to the total length of the RNA sequence and may be omitted. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.fold_compound.backtrack()* in the *The RNA Python module*.

Global *vrna_boustrophedon* (size_t start, size_t end)

This function is available as overloaded global function `boustrophedon()`. See, e.g. *RNA.boustrophedon()* in the *The RNA Python module* .

Global *vrna_boustrophedon_pos* (size_t start, size_t end, size_t pos)

This function is available as overloaded global function `boustrophedon()`. Omitting the `pos` argument yields the entire sequence from `start` to `end`. See, e.g. *RNA.boustrophedon()* in the *The RNA Python module* .

Global *vrna_bp_distance* (const char *str1, const char *str2)

This function is available as an overloaded method **bp_distance**(). Note that the SWIG wrapper takes two structure in dot-bracket notation and converts them into pair tables using *vrna_ptable_from_string()*. The resulting pair tables are then internally passed to *vrna_bp_distance_pt()*. To control which kind of matching brackets will be used during conversion, the optional argument `options` can be used. See also the description of *vrna_ptable_from_string()* for available options. (default: **VRNA_BRACKETS_RND**). See, e.g. *RNA.bp_distance()* in the *The RNA Python module*.

Global *vrna_bp_distance_pt* (const short *pt1, const short *pt2)

This function is available as an overloaded method **bp_distance**(). See, e.g. *RNA.bp_distance()* in the *The RNA Python module*.

Global *vrna_circalifold* (const char **sequences, char *structure)

This function is available as function *circalifold()* in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.circalifold()* in the *The RNA Python module*.

Global *vrna_circfold* (const char *sequence, char *structure)

This function is available as function *circfold()* in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.circfold()* in the *The RNA Python module*.

Global *vrna_cofold* (const char *sequence, char *structure)

This function is available as function *cofold()* in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.cofold()* in the *The RNA Python module*.

Global *vrna_commands_apply* (*vrna_fold_compound_t* *fc, *vrna_cmd_t* commands, unsigned int options)

This function is attached as method `commands_apply()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.commands_apply()* in the *The RNA Python module* .

Global *vrna_db_flatten* (char *structure, unsigned int options)

This function flattens an input structure string in-place! The second parameter is optional and defaults to *VRNA_BRACKETS_DEFAULT*.

An overloaded version of this function exists, where an additional second parameter can be passed to specify the target brackets, i.e. the type of matching pair characters all brackets will be flattened to. Therefore, in the scripting language interface this function is a replacement for *vrna_db_flatten_to()*. See, e.g. *RNA.db_flatten()* in the *The RNA Python module*.

Global *vrna_db_flatten_to* (char *string, const char target[3], unsigned int options)

This function is available as an overloaded version of *vrna_db_flatten()*. See, e.g. *RNA.db_flatten()* in the *The RNA Python module*.

Global *vrna_db_from_probs* (const *FLT_OR_DBL* *pr, unsigned int length)

This function is available as parameter-less method **db_from_probs()** bound to objects of type *fold_compound*. Parameters `pr` and `length` are implicitely taken from the *fold_compound* object the method is bound to. Upon missing base pair probabilities, this method returns an empty string. See, e.g. `RNA.db_from_probs()` in the *The RNA Python module*.

Global *vrna_db_pk_remove* (const char *structure, unsigned int options)

This function is available as an overloaded function `db_pk_remove()` where the optional second parameter `options` defaults to *VRNA_BRACKETS_ANY*. See, e.g. *RNA.db_pk_remove()* in the *The RNA Python module*.

Global *vrna_ensemble_defect* (*vrna_fold_compound_t* *fc, const char *structure)

This function is attached as method `ensemble_defect()` to objects of type `fold_compound`. Note that the SWIG wrapper takes a structure in dot-bracket notation and converts it into a pair table using *vrna_ptable_from_string()*. The resulting pair table is then internally passed to *vrna_ensemble_defect_pt()*. To control which kind of matching brackets will be used during conversion, the optional argument `options` can be used. See also the description of *vrna_ptable_from_string()* for available options. (default: `VRNA_BRACKETS_RND`). See, e.g. *RNA.fold_compound.ensemble_defect()* in the *The RNA Python module*.

Global *vrna_ensemble_defect_pt* (*vrna_fold_compound_t* *fc, const short *pt)

This function is attached as overloaded method `ensemble_defect()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.ensemble_defect()* in the *The RNA Python module*.

Global *vrna_enumerate_necklaces* (const unsigned int *type_counts)

This function is available as global function `enumerate_necklaces()` which accepts lists input, an produces list of lists output. See, e.g. *RNA.enumerate_necklaces()* in the *The RNA Python module* .

Global *vrna_equilibrium_conc* (const double *eq_constants, double *concentration_strands, const unsigned int **A, size_t num_strands, size_t num_complexes)

This function is available as overloaded function `equilibrium_conc()` where the last two parameters `num_strands` and `num_complexes` are optional. In case they are omitted, the respective numbers will be determined from the length of the input list parameters `eq_constants` and `concentration_strands`, respectively. Note, that the C implementation stores equilibrium concentrations for the monomer species in place within the `concentration_strands` parameter. In contrast to that, the scripting language wrappers will rather append them to the returned output list.

Global *vrna_equilibrium_constants* (const double *dG_complexes, const double *dG_strands, const unsigned int **A, double kT, size_t strands, size_t complexes)

This function is available as overloaded function `equilibrium_constants()` where the last two parameters `strands` and `complexes` are optional. In case they are omitted, the respective numbers will be determined from the length of the input list parameters `dG_complexes` and `dG_strands`, respectively. See also *RNA.equilibrium_constants()* in the *The RNA Python module*.

Global *vrna_eval_circ_consensus_structure* (const char **alignment, const char *structure)

This function is available through an overloadeded version of *vrna_eval_circ_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. `RNA.eval_circ_structure()` in the *The RNA Python module* .

Global *vrna_eval_circ_consensus_structure_v* (const char **alignment, const char *structure, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_circ_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. `RNA.eval_circ_structure()` in the *The RNA Python module* .

Global *vrna_eval_circ_gquad_consensus_structure* (const char **alignment, const char *structure)

This function is available through an overloadeded version of *vrna_eval_circ_gquad_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. `RNA.eval_circ_gquad_structure()` in the *The RNA Python module* .

Global *vrna_eval_circ_gquad_consensus_structure_v* (const char **alignment, const char *structure, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_circ_gquad_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. `RNA.eval_circ_gquad_structure()` in the *The RNA Python module* .

Global *vrna_eval_circ_gquad_structure* (const char *string, const char *structure)

In the target scripting language, this function serves as a wrapper for *vrna_eval_circ_gquad_structure_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively.. See, e.g. `RNA.eval_circ_gquad_structure()` in the *The RNA Python module* .

Global *vrna_eval_circ_gquad_structure_v* (const char *string, const char *structure, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_circ_gquad_structure()*. The last two arguments for this function are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. `RNA.eval_circ_gquad_structure()` in the *The RNA Python module* .

Global *vrna_eval_circ_structure* (const char *string, const char *structure)

In the target scripting language, this function serves as a wrapper for *vrna_eval_circ_structure_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively.. See, e.g. `RNA.eval_circ_structure()` in the *The RNA Python module* .

Global *vrna_eval_circ_structure_v* (const char *string, const char *structure, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_circ_structure()*. The last two arguments for this function are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. `RNA.eval_circ_structure()` in the *The RNA Python module* .

Global *vrna_eval_consensus_structure_pt_simple* (const char **alignment, const short *pt)

This function is available through an overloadeded version of *vrna_eval_structure_pt_simple()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. `RNA.eval_structure_pt_simple()` in the *The RNA Python module* .

Global *vrna_eval_consensus_structure_pt_simple_v* (const char **alignment, const short *pt, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_structure_pt_simple()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. `RNA.eval_structure_pt_simple()` in the *The RNA Python module* .

Global *vrna_eval_consensus_structure_simple* (const char **alignment, const char *structure)

This function is available through an overloadeded version of *vrna_eval_structure_simple()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. *RNA.eval_structure_simple()* in the *The RNA Python module* .

Global *vrna_eval_consensus_structure_simple_v* (const char **alignment, const char *structure, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_structure_simple()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_structure_simple()* in the *The RNA Python module* .

Global *vrna_eval_covar_structure* (*vrna_fold_compound_t* *fc, const char *structure)

This function is attached as method `eval_covar_structure()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_covar_structure()* in the *The RNA Python module* .

Global *vrna_eval_gquad_consensus_structure* (const char **alignment, const char *structure)

This function is available through an overloadeded version of *vrna_eval_gquad_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. See, e.g. *RNA.eval_gquad_structure()* in the *The RNA Python module* .

Global *vrna_eval_gquad_consensus_structure_v* (const char **alignment, const char *structure, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_gquad_structure()*. Simply pass a sequence alignment as list of strings (including gaps) as first, and the consensus structure as second argument. The last two arguments are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_gquad_structure()* in the *The RNA Python module* .

Global *vrna_eval_gquad_structure* (const char *string, const char *structure)

In the target scripting language, this function serves as a wrapper for *vrna_eval_gquad_structure_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and NULL, respectively.. See, e.g. *RNA.eval_gquad_structure()* in the *The RNA Python module* .

Global *vrna_eval_gquad_structure_v* (const char *string, const char *structure, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_gquad_structure()*. The last two arguments for this function are optional and default to *VRNA_VERBOSITY_QUIET* and NULL, respectively. See, e.g. *RNA.eval_gquad_structure()* in the *The RNA Python module* .

Global *vrna_eval_hp_loop* (*vrna_fold_compound_t* *fc, int i, int j)

This function is attached as method `eval_hp_loop()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_hp_loop()* in the *The RNA Python module* .

Global *vrna_eval_int_loop* (*vrna_fold_compound_t* *fc, int i, int j, int k, int l)

This function is attached as method `eval_int_loop()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_int_loop()* in the *The RNA Python module* .

Global *vrna_eval_loop_pt* (*vrna_fold_compound_t* *fc, int i, const short *pt)

This function is attached as method `eval_loop_pt()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_loop_pt()* in the *The RNA Python module* .

Global *vrna_eval_move* (*vrna_fold_compound_t* *fc, const char *structure, int m1, int m2)

This function is attached as method `eval_move()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_move()* in the *The RNA Python module* .

Global *vrna_eval_move_pt* (*vrna_fold_compound_t* *fc, short *pt, int m1, int m2)

This function is attached as method `eval_move_pt()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_move_pt()* in the *The RNA Python module* .

Global *vrna_eval_structure* (*vrna_fold_compound_t* *fc, const char *structure)

This function is attached as method `eval_structure()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_structure()* in the *The RNA Python module* .

Global *vrna_eval_structure_pt* (*vrna_fold_compound_t* *fc, const short *pt)

This function is attached as method `eval_structure_pt()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_structure_pt()* in the *The RNA Python module* .

Global *vrna_eval_structure_pt_simple* (const char *string, const short *pt)

In the target scripting language, this function serves as a wrapper for *vrna_eval_structure_pt_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. *RNA.eval_structure_pt_simple()* in the *The RNA Python module* .

Global *vrna_eval_structure_pt_verbose* (*vrna_fold_compound_t* *fc, const short *pt, FILE *file)

This function is attached as method `eval_structure_pt_verbose()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_structure_pt_verbose()* in the *The RNA Python module* .

Global *vrna_eval_structure_simple* (const char *string, const char *structure)

In the target scripting language, this function serves as a wrapper for *vrna_eval_structure_simple_v()* and, thus, allows for two additional, optional arguments, the verbosity level and a file handle which default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively.. See, e.g. *RNA.eval_structure_simple()* in the *The RNA Python module* .

Global *vrna_eval_structure_simple_v* (const char *string, const char *structure, int verbosity_level, FILE *file)

This function is available through an overloaded version of *vrna_eval_structure_simple()*. The last two arguments for this function are optional and default to *VRNA_VERBOSITY_QUIET* and `NULL`, respectively. See, e.g. *RNA.eval_structure_simple()* in the *The RNA Python module* .

Global *vrna_eval_structure_verbose* (*vrna_fold_compound_t* *fc, const char *structure, FILE *file)

This function is attached as method `eval_structure_verbose()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.eval_structure_verbose()* in the *The RNA Python module* .

Global *vrna_exp_params_rescale* (*vrna_fold_compound_t* *fc, double *mfe)

This function is attached to *vrna_fc_s* objects as overloaded `exp_params_rescale()` method.

When no parameter is passed to this method, the resulting action is the same as passing `NULL` as second parameter to *vrna_exp_params_rescale()*, i.e. default scaling of the partition function. Passing an energy in kcal/mol, e.g. as retrieved by a previous call to the `mfe()` method, instructs all subsequent calls to scale the partition function accordingly. See, e.g. *RNA.fold_compound.exp_params_rescale()* in the *The RNA Python module*.

Global *vrna_exp_params_reset* (*vrna_fold_compound_t* *fc, *vrna_md_t* *md)

This function is attached to *vrna_fc_s* objects as overloaded `exp_params_reset()` method.

When no parameter is passed to this method, the resulting action is the same as passing `NULL` as second parameter to *vrna_exp_params_reset()*, i.e. global default model settings are used. Passing an object of type *vrna_md_s* resets the fold compound according to the specifications stored within the *vrna_md_s* object. See, e.g. *RNA.fold_compound.exp_params_reset()* in the *The RNA Python module*.

Global *vrna_exp_params_subst* (*vrna_fold_compound_t* *fc, *vrna_exp_param_t* *params)

This function is attached to *vrna_fc_s* objects as overloaded `exp_params_subst()` method.

When no parameter is passed, the resulting action is the same as passing `NULL` as second parameter to *vrna_exp_params_subst()*, i.e. resetting the parameters to the global defaults. See, e.g. *RNA.fold_compound.exp_params_subst()* in the *The RNA Python module*.

Struct *vrna_fc_s*

This data structure is wrapped as class `fold_compound` with several related functions attached as methods.

A new `fold_compound` can be obtained by calling one of its constructors:

- `fold_compound(seq)` - Initialize with a single sequence, or two concatenated sequences separated by an ampersand character `&` (for cofolding)
- `fold_compound(aln)` - Initialize with a sequence alignment *aln* stored as a list of sequences (with gap characters).

The resulting object has a list of attached methods which in most cases directly correspond to functions that mainly operate on the corresponding `C` data structure:

- *`type()`* - Get the type of the *fold_compound* (See *vrna_fc_type_e*)
- *`length()`* - Get the length of the sequence(s) or alignment stored within the `fold_compound`.

See, e.g. *`RNA.fold_compound`* in the *The RNA Python module*.

Global *vrna_file_commands_apply* (*vrna_fold_compound_t* *fc, const char *filename, unsigned int options)

This function is attached as method `file_commands_apply()` to objects of type `fold_compound`. See, e.g. *`RNA.fold_compound.file_commands_apply()`* in the *The RNA Python module* .

Global *vrna_file_commands_read* (const char *filename, unsigned int options)

This function is available as global function `file_commands_read()`. See, e.g. *`RNA.file_commands_read()`* in the *The RNA Python module* .

Global *vrna_file_msa_detect_format* (const char *filename, unsigned int options)

This function exists as an overloaded version where the `options` parameter may be omitted! In that case, the `options` parameter defaults to *VRNA_FILE_FORMAT_MSA_DEFAULT*. See, e.g. *`RNA.file_msa_detect_format()`* in the *The RNA Python module* .

Global *vrna_file_msa_read* (const char *filename, char ***names, char ***aln, char **id, char **structure, unsigned int options)

In the target scripting language, only the first and last argument, `filename` and `options`, are passed to the corresponding function. The other arguments, which serve as output in the C-library, are available as additional return values. This function exists as an overloaded version where the `options` parameter may be omitted! In that case, the `options` parameter defaults to *VRNA_FILE_FORMAT_MSA_STOCKHOLM*. See, e.g. *`RNA.file_msa_read()`* in the *The RNA Python module* and *Parsing Alignments* in the Python examples.

Global *vrna_file_msa_read_record* (FILE *fp, char ***names, char ***aln, char **id, char **structure, unsigned int options)

In the target scripting language, only the first and last argument, `fp` and `options`, are passed to the corresponding function. The other arguments, which serve as output in the C-library, are available as additional return values. This function exists as an overloaded version where the `options` parameter may be omitted! In that case, the `options` parameter defaults to *VRNA_FILE_FORMAT_MSA_STOCKHOLM*. See, e.g. *`RNA.file_msa_read_record()`* in the *The RNA Python module* and *Parsing Alignments* in the Python examples.

Global *vrna_file_msa_write* (const char *filename, const char **names, const char **aln, const char *id, const char *structure, const char *source, unsigned int options)

In the target scripting language, this function exists as a set of overloaded versions, where the last four parameters may be omitted. If the `options` parameter is missing the options default to (*VRNA_FILE_FORMAT_MSA_STOCKHOLM* | *VRNA_FILE_FORMAT_MSA_APPEND*). See, e.g. *`RNA.file_msa_write()`* in the *The RNA Python module* .

Global *vrna_file_PS_aln* (const char *filename, const char **seqs, const char **names, const char *structure, unsigned int columns)

This function is available as overloaded function `file_PS_aln()` with three additional parameters `start`, `end`, and `offset` before the `columns` argument. Thus, it resembles the *vrna_file_PS_aln_slice()* function. The last four arguments may be omitted, indicating the default of `start` = *0*, `end` = *0*, `offset` = *0*, and `columns` = *60*. See, e.g. *RNA.file_PS_aln()* in the *The RNA Python module*.

Global *vrna_file_PS_aln_slice* (const char *filename, const char **seqs, const char **names, const char *structure, unsigned int start, unsigned int end, int offset, unsigned int columns)

This function is available as overloaded function `file_PS_aln()` where the last four parameter may be omitted, indicating `start` = *0*, `end` = *0*, `offset` = *0*, and `columns` = *60*. See, e.g. *RNA.file_PS_aln()* in the *The RNA Python module*.

Global *vrna_fold* (const char *sequence, char *structure)

This function is available as function *fold()* in the global namespace. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.fold()* in the *The RNA Python module*.

Global *vrna_hc_add_from_db* (*vrna_fold_compound_t* *fc, const char *constraint, unsigned int options)

This function is attached as method `hc_add_from_db()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.hc_add_from_db()* in the *The RNA Python module* .

Global *vrna_hc_init* (*vrna_fold_compound_t* *fc)

This function is attached as method `hc_init()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.hc_init()* in the *The RNA Python module* .

Global *vrna_heat_capacity* (*vrna_fold_compound_t* *fc, float T_min, float T_max, float T_increment, unsigned int mpoints)

This function is attached as overloaded method `heat_capacity()` to objects of type `fold_compound`. If the optional function arguments `T_min`, `T_max`, `T_increment`, and `mpoints` are omitted, they default to 0.0, 100.0, 1.0 and 2, respectively. See, e.g. *RNA.fold_compound.heat_capacity()* in the *The RNA Python module*.

Global *vrna_heat_capacity_cb* (*vrna_fold_compound_t* *fc, float T_min, float T_max, float T_increment, unsigned int mpoints, *vrna_heat_capacity_f* cb, void *data)

This function is attached as method `heat_capacity_cb()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.heat_capacity_cb()* in the *The RNA Python module*.

Global *vrna_heat_capacity_simple* (const char *sequence, float T_min, float T_max, float T_increment, unsigned int mpoints)

This function is available as overloaded function `heat_capacity()`. If the optional function arguments `T_min`, `T_max`, `T_increment`, and `mpoints` are omitted, they default to 0.0, 100.0, 1.0 and 2, respectively. See, e.g. `RNA.head_capacity()` in the *The RNA Python module*.

Global *vrna_init_rand_seed* (unsigned int seed)

This function is available as an overloaded function **init_rand**() where the argument `seed` is optional. See, e.g. *RNA.init_rand()* in the *The RNA Python module*.

Global *vrna_Lfold* (const char *string, int window_size, FILE *file)

This function is available as overloaded function *Lfold()* in the global namespace. The parameter `file` defaults to `NULL` and may be omitted. See e.g. *RNA.Lfold()* in the *The RNA Python module*.

Global *vrna_Lfold_cb* (const char *string, int window_size, *vrna_mfe_window_f* cb, void *data)

This function is available as overloaded function `Lfold_cb()` in the global namespace. The parameter `data` defaults to `NULL` and may be omitted. See e.g. *RNA.Lfold_cb()* in the *The RNA Python module*.

Global vrna_maximum_matching (*vrna_fold_compound_t* *fc)

This function is attached as method `maximum_matching()` to objects of type `fold_compound`. See e.g. `RNA.fold_compound.maximum_matching()` in the *The RNA Python module*.

Global vrna_maximum_matching_simple (const char *sequence)

This function is available as global function `maximum_matching()`. See e.g. *RNA.maximum_matching()* in the *The RNA Python module*.

Struct *vrna_md_s*

This data structure is wrapped as an object `md` with multiple related functions attached as methods.

A new set of default parameters can be obtained by calling the constructure of `md:`

- `md()` – Initialize with default settings

The resulting object has a list of attached methods which directly correspond to functions that mainly operate on the corresponding *C* data structure:

- `reset()` - *vrna_md_set_default()*
- `set_from_globals()` - *set_model_details()*
- `option_string()` - *vrna_md_option_string()*

Global *vrna_MEA* (*vrna_fold_compound_t* *fc, double gamma, float *mea)

This function is attached as overloaded method `MEA`(gamma = 1.) to objects of type `fold_compound`. Note, that it returns the MEA structure and MEA value as a tuple (MEA_structure, MEA). See, e.g. *RNA.fold_compound.MEA()* in the *The RNA Python module*.

Global *vrna_MEA_from_plist* (*vrna_ep_t* *plist, const char *sequence, double gamma, *vrna_md_t* *md, float *mea)

This function is available as overloaded function `MEA_from_plist`(gamma = 1., md = NULL). Note, that it returns the MEA structure and MEA value as a tuple (MEA_structure, MEA). See, e.g. *RNA.MEA_from_plist()* in the *The RNA Python module*.

Global *vrna_mean_bp_distance* (*vrna_fold_compound_t* *fc)

This function is attached as method *mean_bp_distance()* to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.mean_bp_distance()* in the *The RNA Python module*.

Global *vrna_mfe* (*vrna_fold_compound_t* *fc, char *structure)

This function is attached as method `Minimum Free Energy (MFE) Algorithms` to objects of type *The Fold Compound*. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.fold_compound.mfe()* in the *The RNA Python module*.

Global *vrna_mfe_dimer* (*vrna_fold_compound_t* *fc, char *structure)

This function is attached as method `mfe_dimer()` to objects of type *The Fold Compound*. The parameter `structure` is returned along with the MFE und must not be provided. See e.g. *RNA.fold_compound.mfe_dimer()* in the *The RNA Python module*.

Global *vrna_mfe_window* (*vrna_fold_compound_t* *fc, FILE *file)

This function is attached as overloaded method *Local (sliding window) MFE Prediction* to objects of type *The Fold Compound*. The parameter `FILE` has default value of `NULL` and can be omitted. See e.g. *RNA.fold_compound.mfe_window()* in the *The RNA Python module*.

Global *vrna_mfe_window_cb* (*vrna_fold_compound_t* *fc, *vrna_mfe_window_f* cb, void *data)

This function is attached as overloaded method `mfe_window_cb()` to objects of type *The Fold Compound*. The parameter `data` has default value of `NULL` and can be omitted. See e.g. *RNA.fold_compound.mfe_window_cb()* in the *The RNA Python module*.

Global *vrna_neighbors* (*vrna_fold_compound_t* *fc, const short *pt, unsigned int options)

This function is attached as an overloaded method `neighbors()` to objects of type `fold_compound`. The optional parameter `options` defaults to *VRNA_MOVESET_DEFAULT* if it is omitted. See, e.g. *RNA.fold_compound.neighbors()* in the *The RNA Python module*.

Global *vrna_params_load* (const char fname[], unsigned int options)

This function is available as overloaded function `params_load`(fname="", options=*VRNA_PARAMETER_FORMAT_DEFAULT*). Here, the empty filename string indicates to load default RNA parameters, i.e. this is equivalent to calling *vrna_params_load_defaults()*. See, e.g. `RNA.fold_compound.params_load()` in the *The RNA Python module*.

Global *vrna_params_load_defaults* (void)

This function is available as overloaded function `params_load()`. See, e.g. *RNA.params_load()* in the *The RNA Python module*.

Global *vrna_params_load_DNA_Mathews1999* (void)

This function is available as function `params_load_DNA_Mathews1999()`. See, e.g. *RNA.params_load_DNA_Mathews1999()* in the *The RNA Python module*.

Global *vrna_params_load_DNA_Mathews2004* (void)

This function is available as function `params_load_DNA_Mathews2004()`. See, e.g. *RNA.params_load_DNA_Mathews2004()* in the *The RNA Python module*.

Global *vrna_params_load_from_string* (const char *string, const char *name, unsigned int options)

This function is available as overloaded function `params_load_from_string`(string, name="", options=*VRNA_PARAMETER_FORMAT_DEFAULT*). See, e.g. *RNA.params_load_from_string()* in the *The RNA Python module*.

Global *vrna_params_load_RNA_Andronescu2007* (void)

This function is available as function `params_load_RNA_Andronescu2007()`. See, e.g. *RNA.params_load_RNA_Andronescu2007()* in the *The RNA Python module*.

Global *vrna_params_load_RNA_Langdon2018* (void)

This function is available as function `params_load_RNA_Langdon2018()`. See, e.g. *RNA.params_load_RNA_Langdon2018()* in the *The RNA Python module*.

Global *vrna_params_load_RNA_misc_special_hairpins* (void)

This function is available as function `params_load_RNA_misc_special_hairpins()`. See, e.g. *RNA.params_load_RNA_misc_special_hairpins()* in the *The RNA Python module*.

Global *vrna_params_load_RNA_Turner1999* (void)

This function is available as function `params_load_RNA_Turner1999()`. See, e.g. *RNA.params_load_RNA_Turner1999()* in the *The RNA Python module*.

Global *vrna_params_load_RNA_Turner2004* (void)

This function is available as function `params_load_RNA_Turner2004()`. See, e.g. *RNA.params_load_RNA_Turner2004()* in the *The RNA Python module*.

Global *vrna_params_reset* (*vrna_fold_compound_t* *fc, *vrna_md_t* *md)

This function is attached to *vrna_fc_s* objects as overloaded `params_reset()` method.

When no parameter is passed to this method, the resulting action is the same as passing `NULL` as second parameter to *vrna_params_reset()*, i.e. global default model settings are used. Passing an object of type *vrna_md_s* resets the fold compound according to the specifications stored within the *vrna_md_s* object. See, e.g. *RNA.fold_compound.params_reset()* in the *The RNA Python module*.

Global *vrna_params_save* (const char fname[], unsigned int options)

This function is available as overloaded function `params_save`(fname, options=*VRNA_PARAMETER_FORMAT_DEFAULT*). See, e.g. *RNA.params_save()* in the *The RNA Python module*.

Global *vrna_params_subst* (*vrna_fold_compound_t* *fc, *vrna_param_t* *par)

This function is attached to *vrna_fc_s* objects as overloaded `params_subst()` method.

When no parameter is passed, the resulting action is the same as passing `NULL` as second parameter to *vrna_params_subst()*, i.e. resetting the parameters to the global defaults. See, e.g. *RNA.fold_compound.params_subst()* in the *The RNA Python module*.

Global *vrna_path* (*vrna_fold_compound_t* *fc, short *pt, unsigned int steps, unsigned int options)

This function is attached as an overloaded method `path()` to objects of type `fold_compound`. The optional parameter `options` defaults to *VRNA_PATH_DEFAULT* if it is omitted. See, e.g. *RNA.fold_compound.path()* in the *The RNA Python module*.

Global *vrna_path_direct* (*vrna_fold_compound_t* *fc, const char *s1, const char *s2, *vrna_path_options_t* options)

This function is attached as an overloaded method `path_direct()` to objects of type `fold_compound`. The optional parameter `options` defaults to `NULL` if it is omitted. See, e.g. *RNA.fold_compound.path_direct()* in the *The RNA Python module*.

Global *vrna_path_direct_ub* (*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int maxE, *vrna_path_options_t* options)

This function is attached as an overloaded method `path_direct()` to objects of type `fold_compound`. The optional parameter `maxE` defaults to #INT_MAX - 1 if it is omitted, while the optional parameter `options` defaults to `NULL`. In case the function did not find a path with $E_{saddle} < E_{max}$ it returns an empty list. See, e.g. *RNA.fold_compound.path_direct()* in the *The RNA Python module*.

Global *vrna_path_findpath* (*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int width)

This function is attached as an overloaded method `path_findpath()` to objects of type `fold_compound`. The optional parameter `width` defaults to 1 if it is omitted. See, e.g. *RNA.fold_compound.path_findpath()* in the *The RNA Python module*.

Global *vrna_path_findpath_saddle* (*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int width)

This function is attached as an overloaded method `path_findpath_saddle()` to objects of type `fold_compound`. The optional parameter `width` defaults to 1 if it is omitted. See, e.g. *RNA.fold_compound.path_findpath_saddle()* in the *The RNA Python module*.

Global *vrna_path_findpath_saddle_ub* (*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int width, int maxE)

This function is attached as an overloaded method `path_findpath_saddle()` to objects of type `fold_compound`. The optional parameter `width` defaults to 1 if it is omitted, while the optional parameter `maxE` defaults to INF. In case the function did not find a path with $E_{saddle} < E_{max}$ the function returns a `NULL` object, i.e. `undef` for Perl and `None` for Python. See, e.g. *RNA.fold_compound.path_findpath_saddle()* in the *The RNA Python module*.

Global *vrna_path_findpath_ub* (*vrna_fold_compound_t* *fc, const char *s1, const char *s2, int width, int maxE)

This function is attached as an overloaded method `path_findpath()` to objects of type `fold_compound`. The optional parameter `width` defaults to 1 if it is omitted, while the optional parameter `maxE` defaults to INF. In case the function did not find a path with $E_{saddle} < E_{max}$ the function returns an empty list. See, e.g. *RNA.fold_compound.path_findpath()* in the *The RNA Python module*.

Global *vrna_path_gradient* (*vrna_fold_compound_t* *fc, short *pt, unsigned int options)

This function is attached as an overloaded method `path_gradient()` to objects of type `fold_compound`. The optional parameter `options` defaults to *VRNA_PATH_DEFAULT* if it is omitted. See, e.g. *RNA.fold_compound.path_gradient()* in the *The RNA Python module*.

Global *vrna_path_options_findpath* (int width, unsigned int type)

This function is available as overloaded function `path_options_findpath()`. The optional parameter `width` defaults to 10 if omitted, while the optional parameter `type` defaults to *VRNA_PATH_TYPE_DOT_BRACKET*. See, e.g. *RNA.path_options_findpath()* in the *The RNA Python module*.

Global *vrna_path_random* (*vrna_fold_compound_t* *fc, short *pt, unsigned int steps, unsigned int options)

This function is attached as an overloaded method `path_gradient()` to objects of type `fold_compound`. The optional parameter `options` defaults to *VRNA_PATH_DEFAULT* if it is omitted. See, e.g. *RNA.fold_compound.path_random()* in the *The RNA Python module*.

Global *vrna_pbacktrack* (*vrna_fold_compound_t* *fc)

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack5* (*vrna_fold_compound_t* *fc, unsigned int length)

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack5_cb* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int length, *vrna_bs_result_f* cb, void *data, unsigned int options)

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack5_num* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int length, unsigned int options)

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack5_resume* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int length, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the list of structures, this function also returns the `nr_mem` data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack5_resume_cb* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int length, *vrna_bs_result_f* cb, void *data, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

This function is attached as overloaded method *pbacktrack5()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the number of structures backtraced, this function also returns the `nr_mem` data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_cb* (*vrna_fold_compound_t* *fc, unsigned int num_samples, *vrna_bs_result_f* cb, void *data, unsigned int options)

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_num* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int options)

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_resume* (*vrna_fold_compound_t* *fc, unsigned int num_samples, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

This function is attached as overloaded method *pbacktrack()* to objects of type `fold_compound` with optional last argument `options` = *VRNA_PBACKTRACK_DEFAULT*. In addition to the list of

structures, this function also returns the nr_mem data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_resume_cb* (*vrna_fold_compound_t* *fc, unsigned int num_samples, *vrna_bs_result_f* cb, void *data, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

This function is attached as overloaded method *pbacktrack()* to objects of type fold_compound with optional last argument options = *VRNA_PBACKTRACK_DEFAULT*. In addition to the number of structures backtraced, this function also returns the nr_mem data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_sub* (*vrna_fold_compound_t* *fc, unsigned int start, unsigned int end)

This function is attached as overloaded method pbacktrack_sub() to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.pbacktrack_sub()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_sub_cb* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int start, unsigned int end, *vrna_bs_result_f* cb, void *data, unsigned int options)

This function is attached as overloaded method *pbacktrack()* to objects of type fold_compound with optional last argument options = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_sub_num* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int start, unsigned int end, unsigned int options)

This function is attached as overloaded method pbacktrack_sub() to objects of type fold_compound with optional last argument options = *VRNA_PBACKTRACK_DEFAULT*. See, e.g. *RNA.fold_compound.pbacktrack_sub()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_sub_resume* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int start, unsigned int end, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

This function is attached as overloaded method pbacktrack_sub() to objects of type fold_compound with optional last argument options = *VRNA_PBACKTRACK_DEFAULT*. In addition to the list of structures, this function also returns the nr_mem data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack_sub()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pbacktrack_sub_resume_cb* (*vrna_fold_compound_t* *fc, unsigned int num_samples, unsigned int start, unsigned int end, *vrna_bs_result_f* cb, void *data, *vrna_pbacktrack_mem_t* *nr_mem, unsigned int options)

This function is attached as overloaded method pbacktrack_sub() to objects of type fold_compound with optional last argument options = *VRNA_PBACKTRACK_DEFAULT*. In addition to the number of structures backtraced, this function also returns the nr_mem data structure as first return value. See, e.g. *RNA.fold_compound.pbacktrack_sub()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Global *vrna_pf* (*vrna_fold_compound_t* *fc, char *structure)

This function is attached as method pf() to objects of type fold_compound. See, e.g. *RNA.fold_compound.pf()* in the *The RNA Python module*.

Global *vrna_pf_dimer* (*vrna_fold_compound_t* *fc, char *structure)

This function is attached as method pf_dimer() to objects of type fold_compound. See, e.g. *RNA.fold_compound.pf_dimer()* in the *The RNA Python module*.

Global *vrna_plot_dp_EPS* (const char *filename, const char *sequence, *vrna_ep_t* *upper, *vrna_ep_t* *lower, *vrna_dotplot_auxdata_t* *auxdata, unsigned int options)

This function is available as overloaded function plot_dp_EPS() where the last three parameters may be omitted. The default values for these parameters are lower = NULL, auxdata = NULL, options =

VRNA_PLOT_PROBABILITIES_DEFAULT. See, e.g. *RNA.plot_dp_EPS()* in the *The RNA Python module*.

Global *vrna_positional_entropy* (*vrna_fold_compound_t* *fc)

This function is attached as method `positional_entropy()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.positional_entropy()* in the *The RNA Python module*.

Global *vrna_pr_energy* (*vrna_fold_compound_t* *fc, double e)

This function is attached as method `pr_energy()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.pr_energy()* in the *The RNA Python module*.

Global *vrna_pr_structure* (*vrna_fold_compound_t* *fc, const char *structure)

This function is attached as method `pr_structure()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.pr_structure()* in the *The RNA Python module*.

Global *vrna_probing_data_Deigan2009* (const double *reactivities, unsigned int n, double m, double b)

This function exists in two forms, (i) as overloaded function **probing_data_Deigan2009()** and (ii) as constructor of the **probing_data** object. For the former the second argument `n` can be omitted since the length of the `reactivities` list is determined from the list itself. When the *vrna_probing_data_t* constructor is called with the three parameters `reactivities`, `m` and `b`, it will automatically create a prepared data structure for the Deigan et al. 2009 method. See, e.g. `RNA.probing_data_Deigan2009()` and *RNA.probing_data()* in the *The RNA Python module* .

Global *vrna_ptable* (const char *structure)

This functions is wrapped as overloaded function `ptable()` that takes an optional argument `options` to specify which type of matching brackets should be considered during conversion. The default set is round brackets, i.e. *VRNA_BRACKETS_RND*. See, e.g. *RNA.ptable()* in the *The RNA Python module*.

Global *vrna_ptable_from_string* (const char *structure, unsigned int options)

This functions is wrapped as overloaded function `ptable()` that takes an optional argument `options` to specify which type of matching brackets should be considered during conversion. The default set is round brackets, i.e. *VRNA_BRACKETS_RND*. See, e.g. *RNA.ptable()* in the *The RNA Python module*.

Global *vrna_rotational_symmetry* (const char *string)

This function is available as global function `rotational_symmetry()`. See *vrna_rotational_symmetry_pos()* for details. See, e.g. *RNA.rotational_symmetry()* in the *The RNA Python module* .

Global *vrna_rotational_symmetry_db* (*vrna_fold_compound_t* *fc, const char *structure)

This function is attached as method `rotational_symmetry_db()` to objects of type `fold_compound` (i.e. *vrna_fold_compound_t*). See *vrna_rotational_symmetry_db_pos()* for details. See, e.g. *RNA.fold_compound.rotational_symmetry_db()* in the *The RNA Python module* .

Global *vrna_rotational_symmetry_db_pos* (*vrna_fold_compound_t* *fc, const char *structure, unsigned int **positions)

This function is attached as method `rotational_symmetry_db()` to objects of type `fold_compound` (i.e. *vrna_fold_compound_t*). Thus, the first argument must be omitted. In contrast to our C-implementation, this function doesn't simply return the order of rotational symmetry of the secondary structure, but returns the list `position` of cyclic permutation shifts that result in a rotationally symmetric structure. The length of the list then determines the order of rotational symmetry. See, e.g. *RNA.fold_compound.rotational_symmetry_db()* in the *The RNA Python module* .

Global *vrna_rotational_symmetry_num* (const unsigned int *string, size_t string_length)

This function is available as global function `rotational_symmetry()`. See *vrna_rotational_symmetry_pos()* for details. Note, that in the target language the length of the list `string` is always known a-priori, so the parameter `string_length` must be omitted. See, e.g. *RNA.rotational_symmetry()* in the *The RNA Python module* .

Global *vrna_rotational_symmetry_pos* (const char *string, unsigned int **positions)

This function is available as overloaded global function `rotational_symmetry()`. It merges the functionalities of *vrna_rotational_symmetry()*, *vrna_rotational_symmetry_pos()*, *vrna_rotational_symmetry_num()*, and *vrna_rotational_symmetry_pos_num()*. In contrast to our C-implementation, this function doesn't return the order of rotational symmetry as a single value, but returns a list of cyclic permutation shifts that result in a rotationally symmetric string. The length of the list then determines the order of rotational symmetry. See, e.g. *RNA.rotational_symmetry()* in the *The RNA Python module* .

Global *vrna_rotational_symmetry_pos_num* (const unsigned int *string, size_t string_length, unsigned int **positions)

This function is available as global function `rotational_symmetry()`. See *vrna_rotational_symmetry_pos()* for details. Note, that in the target language the length of the list `string` is always known a-priori, so the parameter `string_length` must be omitted. See, e.g. *RNA.rotational_symmetry()* in the *The RNA Python module* .

Global *vrna_sc_add_bp* (*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, *FLT_OR_DBL* energy, unsigned int options)

This function is attached as an overloaded method `sc_add_bp()` to objects of type `fold_compound`. The method either takes arguments for a single base pair (i,j) with the corresponding energy value:

Global *vrna_sc_add_bt* (*vrna_fold_compound_t* *fc, *vrna_sc_bt_f* f)

This function is attached as method `sc_add_bt()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_bt()* in the *The RNA Python module* .

Global *vrna_sc_add_data* (*vrna_fold_compound_t* *fc, void *data, *vrna_auxdata_free_f* free_data)

This function is attached as method `sc_add_data()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_data()* in the *The RNA Python module* .

Global *vrna_sc_add_exp_f* (*vrna_fold_compound_t* *fc, *vrna_sc_exp_f* exp_f)

This function is attached as method `sc_add_exp_f()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_exp_f()* in the *The RNA Python module* .

Global *vrna_sc_add_f* (*vrna_fold_compound_t* *fc, *vrna_sc_f* f)

This function is attached as method `sc_add_f()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_f()* in the *The RNA Python module* .

Global *vrna_sc_add_hi_motif* (*vrna_fold_compound_t* *fc, const char *seq, const char *structure, *FLT_OR_DBL* energy, unsigned int options)

This function is attached as method `sc_add_hi_motif()` to objects of type `fold_compound`. The last parameter is optional an defaults to `options` = *VRNA_OPTION_DEFAULT*. See, e.g. *RNA.fold_compound.sc_add_hi_motif()* in the *The RNA Python module* .

Global *vrna_sc_add_SHAPE_deigan* (*vrna_fold_compound_t* *fc, const double *reactivities, double m, double b, unsigned int options)

This function is attached as method `sc_add_SHAPE_deigan()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_SHAPE_deigan()* in the *The RNA Python module* .

Global *vrna_sc_add_SHAPE_deigan_ali* (*vrna_fold_compound_t* *fc, const char **shape_files, const int *shape_file_association, double m, double b, unsigned int options)

This function is attached as method `sc_add_SHAPE_deigan_ali()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_SHAPE_deigan_ali()* in the *The RNA Python module* .

Global *vrna_sc_add_SHAPE_zarringhalam* (*vrna_fold_compound_t* *fc, const double *reactivities, double b, double default_value, const char *shape_conversion, unsigned int options)

This function is attached as method `sc_add_SHAPE_zarringhalam()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_add_SHAPE_zarringhalam()* in the *The RNA Python module* .

Global *vrna_sc_add_up* (*vrna_fold_compound_t* *fc, unsigned int i, *FLT_OR_DBL* energy, unsigned int options)

This function is attached as an overloaded method `sc_add_up()` to objects of type `fold_compound`. The method either takes arguments for a single nucleotide i with the corresponding energy value:

Global *vrna_sc_init* (*vrna_fold_compound_t* *fc)

This function is attached as method `sc_init()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_init()* in the *The RNA Python module* .

Global *vrna_sc_mod* (*vrna_fold_compound_t* *fc, const *vrna_sc_mod_param_t* params, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod()* in the *The RNA Python module* .

Global *vrna_sc_mod_7DA* (*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod_7DA()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_7DA()* in the *The RNA Python module* .

Global *vrna_sc_mod_dihydrouridine* (*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod_dihydrouridine()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_dihydrouridine()* in the *The RNA Python module* .

Global *vrna_sc_mod_inosine* (*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod_inosine()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_inosine()* in the *The RNA Python module* .

Global *vrna_sc_mod_json* (*vrna_fold_compound_t* *fc, const char *json, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod_json()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_json()* in the *The RNA Python module* .

Global *vrna_sc_mod_jsonfile* (*vrna_fold_compound_t* *fc, const char *json_file, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod_jsonfile()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_jsonfile()* in the *The RNA Python module* .

Global *vrna_sc_mod_m6A* (*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod_m6A()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_m6A()* in the *The RNA Python module* .

Global *vrna_sc_mod_pseudouridine* (*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod_pseudouridine()` to objects of type `fold_compound` with default `options` = *VRNA_SC_MOD_DEFAULT*. See, e.g. *RNA.fold_compound.sc_mod_pseudouridine()* in the *The RNA Python module* .

Global *vrna_sc_mod_purine* (*vrna_fold_compound_t* *fc, const unsigned int *modification_sites, unsigned int options)

This function is attached as overloaded method `sc_mod_purine()` to objects of type `fold_compound` with default `options = VRNA_SC_MOD_DEFAULT`. See, e.g. *RNA.fold_compound.sc_mod_purine()* in the *The RNA Python module* .

Global *vrna_sc_mod_read_from_json* (const char *json, *vrna_md_t* *md)

This function is available as an overloaded function `sc_mod_read_from_json()` where the `md` parameter may be omitted and defaults to `NULL`. See, e.g. *RNA.sc_mod_read_from_json()* in the *The RNA Python module* .

Global *vrna_sc_mod_read_from_jsonfile* (const char *filename, *vrna_md_t* *md)

This function is available as an overloaded function `sc_mod_read_from_jsonfile()` where the `md` parameter may be omitted and defaults to `NULL`. See, e.g. *RNA.sc_mod_read_from_jsonfile()* in the *The RNA Python module* .

Global *vrna_sc_probing* (*vrna_fold_compound_t* *fc, *vrna_probing_data_t* data)

This function is attached as method **sc_probing()** to objects of type **fold_compound**. See, e.g. *RNA.fold_compound.sc_probing()* in the *The RNA Python module* .

Global *vrna_sc_remove* (*vrna_fold_compound_t* *fc)

This function is attached as method `sc_remove()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_remove()* in the *The RNA Python module* .

Global *vrna_sc_set_bp* (*vrna_fold_compound_t* *fc, const *FLT_OR_DBL* **constraints, unsigned int options)

This function is attached as method `sc_set_bp()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_set_bp()* in the *The RNA Python module* .

Global *vrna_sc_set_up* (*vrna_fold_compound_t* *fc, const *FLT_OR_DBL* *constraints, unsigned int options)

This function is attached as method `sc_set_up()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.sc_set_up()* in the *The RNA Python module* .

Global *vrna_stack_prob* (*vrna_fold_compound_t* *fc, double cutoff)

This function is attached as overloaded method `stack_prob()` to objects of type `fold_compound`. The optional argument `cutoff` defaults to 1e-5. See, e.g. *RNA.fold_compound.stack_prob()* in the *The RNA Python module*.

Global *vrna_strtrim* (char *string, const char *delimiters, unsigned int keep, unsigned int options)

Since many scripting languages treat strings as immutable objects, this function does not modify the input string directly. Instead, it returns the modified string as second return value, together with the number of removed delimiters.

The scripting language interface provides an overloaded version of this function, with default parameters `delimiters=NULL`, `keep=0`, and `options=VRNA_TRIM_DEFAULT`. See, e.g. *RNA.strtrim()* in the *The RNA Python module*.

Global *vrna_subopt* (*vrna_fold_compound_t* *fc, int delta, int sorted, FILE *fp)

This function is attached as method ***subopt()*** to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.subopt()* in the *The RNA Python module*.

Global *vrna_subopt_cb* (*vrna_fold_compound_t* *fc, int delta, *vrna_subopt_result_f* cb, void *data)

This function is attached as method **subopt_cb()** to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.subopt_cb()* in the *The RNA Python module*.

Global *vrna_subopt_zuker* (*vrna_fold_compound_t* *fc)

This function is attached as method **subopt_zuker()** to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.subopt_zuker()* in the *The RNA Python module*.

Global *vrna_ud_remove* (*vrna_fold_compound_t* *fc)

This function is attached as method `ud_remove()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.ud_remove()* in the *The RNA Python module*.

Global *vrna_ud_set_data* (*vrna_fold_compound_t* *fc, void *data, *vrna_auxdata_free_f* free_cb)

This function is attached as method `ud_set_data()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.ud_set_data()* in the *The RNA Python module*.

Global *vrna_ud_set_exp_prod_rule_cb* (*vrna_fold_compound_t* *fc, *vrna_ud_exp_production_f* pre_cb, *vrna_ud_exp_f* exp_e_cb)

This function is attached as method `ud_set_exp_prod_rule_cb()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.ud_set_exp_prod_rule_cb()* in the *The RNA Python module*.

Global vrna_ud_set_prob_cb (*vrna_fold_compound_t* *fc, *vrna_ud_add_probs_f* setter, *vrna_ud_get_probs_f* getter)

This function is attached as method `ud_set_prob_cb()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.ud_set_prob_cb()* in the *The RNA Python module*.

Global *vrna_ud_set_prod_rule_cb* (*vrna_fold_compound_t* *fc, *vrna_ud_production_f* pre_cb, *vrna_ud_f* e_cb)

This function is attached as method `ud_set_prod_rule_cb()` to objects of type `fold_compound`. See, e.g. *RNA.fold_compound.ud_set_prod_rule_cb()* in the *The RNA Python module*.

4.3.2 The RNA Python module

Note

Our Python API is automatically generated and translated from our C-library documentation. If you find anything problematic or want to to help us improve the documentation, do not hesitate to contact us or make a PR at our official github repository.

A library for the prediction and comparison of RNA secondary structures.

Amongst other things, our implementations allow you to:

- predict minimum free energy secondary structures
- calculate the partition function for the ensemble of structures
- compute various equilibrium probabilities
- calculate suboptimal structures in a given energy range
- compute local structures in long sequences
- predict consensus secondary structures from a multiple sequence alignment
- predict melting curves
- search for sequences folding into a given structure
- compare two secondary structures
- predict interactions between multiple RNA molecules

class RNA.**COORDINATE**

Bases: `object`

this is a workarround for the SWIG Perl Wrapper RNA plot function that returns an array of type COORDINATE

X

Type
`float`

Y

Type
float

this is a workarround for the SWIG Perl Wrapper RNA plot function that returns an array of type COORDINATE

X

Type
float

Y

Type
float

property X

property Y

get(*i*)

property thisown

The membership flag

class RNA.**ConstCharVector**(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(**args*)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

class RNA.CoordinateVector(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(**args*)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

class RNA.DoubleDoubleVector(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(**args*)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown
The membership flag

class RNA.DoublePair(**args*)
Bases: object

property first

property second

property thisown
The membership flag

class RNA.DoubleVector(**args*)
Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(*args)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(x)

rbegin()

rend()

reserve(n)

resize(*args)

size()

swap(v)

property thisown

The membership flag

class RNA.DuplexVector(*args)

Bases: object

append(x)

assign(n, x)

back()

begin()

capacity()

clear()

empty()

end()

erase(*args)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

RNA.**E_ExtLoop**(*type*, *si1*, *sj1*, *P*)

RNA.**E_GQuad_IntLoop_L**(*i*, *j*, *type*, *S*, *ggg*, *maxdist*, *P*)

RNA.**E_GQuad_IntLoop_L_comparative**(*i*, *j*, *tt*, *S_cons*, *S5*, *S3*, *a2s*, *ggg*, *n_seq*, *P*)

RNA.**E_Hairpin**(*size*, *type*, *si1*, *sj1*, *string*, *P*)

Compute the Energy of a hairpin-loop.

To evaluate the free energy of a hairpin-loop, several parameters have to be known. A general hairpin-loop has this structure:

a3 a4

a2 a5 a1 a6

X - Y | | 5' 3'

where X-Y marks the closing pair [e.g. a **(G,C)** pair]. The length of this loop is 6 as there are six unpaired nucleotides (a1-a6) enclosed by (X,Y). The 5' mismatching nucleotide is a1 while the 3' mismatch is a6. The nucleotide sequence of this loop is "a1.a2.a3.a4.a5.a6"

Parameters

- **size** (`int`) – The size of the loop (number of unpaired nucleotides)
- **type** (`int`) – The pair type of the base pair closing the hairpin
- **si1** (`int`) – The 5'-mismatching nucleotide
- **sj1** (`int`) – The 3'-mismatching nucleotide
- **string** (`string`) – The sequence of the loop (May be *NULL*, otherwise mst be at least $size + 2$ long)
- **P** (`RNA.param() *`) – The datastructure containing scaled energy parameters

Returns

The Free energy of the Hairpin-loop in dcal/mol

Return type

`int`

Warning

Not (really) thread safe! A threadsafe implementation will replace this function in a future release!

Energy evaluation may change due to updates in global variable "tetra_loop"

See also

`scale_parameters`, *RNA.param*

Note

The parameter sequence should contain the sequence of the loop in capital letters of the nucleic acid alphabet if the loop size is below 7. This is useful for unusually stable tri-, tetra- and hexa-loops which are treated differently (based on experimental data) if they are tabulated.

`RNA.E_IntLoop`(*n1*, *n2*, *type*, *type_2*, *si1*, *sj1*, *sp1*, *sq1*, *P*)

Compute the Energy of an internal-loop

This function computes the free energy ΔG of an internal-loop with the following structure:

3' 5' | | U - V

a_n b_1
. . . . . .

a_1 b_m
X - Y | | 5' 3'

This general structure depicts an internal-loop that is closed by the base pair (X,Y). The enclosed base pair is (V,U) which leaves the unpaired bases a_1-a_n and b_1-b_n that constitute the loop. In this example, the length of the internal-loop is $(n + m)$ where n or m may be 0 resulting in a bulge-loop or base pair stack. The mismatching nucleotides for the closing pair (X,Y) are: 5'-mismatch: a_1 3'-mismatch: b_m and for the enclosed base pair (V,U): 5'-mismatch: b_1 3'-mismatch: a_n

param n1
The size of the 'left'-loop (number of unpaired nucleotides)

type n1
`int`

param n2
The size of the 'right'-loop (number of unpaired nucleotides)

type n2
`int`

param type
The pair type of the base pair closing the internal loop

type type
`int`

param type_2
The pair type of the enclosed base pair

type type_2
int

param si1
The 5'-mismatching nucleotide of the closing pair

type si1
int

param sj1
The 3'-mismatching nucleotide of the closing pair

type sj1
int

param sp1
The 3'-mismatching nucleotide of the enclosed pair

type sp1
int

param sq1
The 5'-mismatching nucleotide of the enclosed pair

type sq1
int

param P
The datastructure containing scaled energy parameters

type P
RNA.param() *

returns
The Free energy of the Interior-loop in dcal/mol

rtype
int

See also

scale_parameters, *RNA.param*

Note

Base pairs are always denoted in 5'->3' direction. Thus the enclosed base pair must be 'turned arround' when evaluating the free energy of the internal-loop

This function is threadsafe

RNA.**E_IntLoop_Co**(*type*, *type_2*, *i*, *j*, *p*, *q*, *cutpoint*, *si1*, *sj1*, *sp1*, *sq1*, *dangles*, *P*)

RNA.**E_MLstem**(*type*, *si1*, *sj1*, *P*)

RNA.**E_Stem**(*type*, *si1*, *sj1*, *extLoop*, *P*)

Compute the energy contribution of a stem branching off a loop-region.

This function computes the energy contribution of a stem that branches off a loop region. This can be the case in multiloops, when a stem branching off increases the degree of the loop but also *immediately interior base pairs* of an exterior loop contribute free energy. To switch the behavior of the function according to the evaluation of a multiloop- or exterior-loop-stem, you pass the flag 'extLoop'. The returned energy contribution consists of a TerminalAU penalty if the pair type is greater than 2, dangling end contributions of mismatching nucleotides adjacent to the stem if only one of the si1, sj1 parameters is greater than 0

and mismatch energies if both mismatching nucleotides are positive values. Thus, to avoid incorporating dangling end or mismatch energies just pass a negative number, e.g. -1 to the mismatch argument.

This is an illustration of how the energy contribution is assembled:

3’ 5’ | | X - Y

5’-si1 sj1-3’

Here, (X,Y) is the base pair that closes the stem that branches off a loop region. The nucleotides si1 and sj1 are the 5’- and 3’- mismatches, respectively. If the base pair type of (X,Y) is greater than 2 (i.e. an A-U or G-U pair, the TerminalAU penalty will be included in the energy contribution returned. If si1 and sj1 are both nonnegative numbers, mismatch energies will also be included. If one of si1 or sj1 is a negative value, only 5’ or 3’ dangling end contributions are taken into account. To prohibit any of these mismatch contributions to be incorporated, just pass a negative number to both, si1 and sj1. In case the argument extLoop is 0, the returned energy contribution also includes the *internal-loop-penalty* of a multiloop stem with closing pair type.

Deprecated since version 2.8.0: Please use one of the functions RNA.E_exterior_stem() and RNA.E_multibranch_stem() instead! Use the former for cases where *extLoop* != 0 and the latter otherwise.

See also

`RNA.E_multibranch_stem`, `_ExtLoop`

Note

This function is threadsafe

Parameters

- **type** (`int`) – The pair type of the first base pair un the stem
- **si1** (`int`) – The 5’-mismatching nucleotide
- **sj1** (`int`) – The 3’-mismatching nucleotide
- **extLoop** (`int`) – A flag that indicates whether the contribution reflects the one of an exterior loop or not
- **P** (`RNA.param() *`) – The data structure containing scaled energy parameters

Returns

The Free energy of the branch off the loop in dcal/mol

Return type

`int`

`RNA.`**`E_gquad`**`(`*L*, *l*, *P*`)`

`RNA.`**`E_gquad_ali_en`**`(`*i*, *L*, *l*, *S*, *a2s*, *n_seq*, *P*, *en*`)`

`RNA.`**`E_ml_rightmost_stem`**`(`*i*, *j*, *fc*`)`

`class RNA.`**`ElemProbVector`**`(`**args*`)`

Bases: `object`

`append``(`*x*`)`

`assign``(`*n*, *x*`)`

`back``()`

begin()

capacity()

clear()

empty()

end()

erase(*args)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(x)

rbegin()

rend()

reserve(n)

resize(*args)

size()

swap(v)

property thisown

The membership flag

class RNA.HeatCapacityVector(*args)

Bases: object

append(x)

assign(n, x)

back()

begin()

capacity()

clear()

empty()

end()

erase(*args)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(x)

rbegin()

rend()

reserve(n)

resize(*args)

size()

swap(v)

property thisown

The membership flag

class RNA.HelixVector(*args)

Bases: object

append(x)

assign(n, x)

back()

begin()

capacity()

clear()

empty()

end()

erase(*args)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(x)

rbegin()

rend()

reserve(n)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

class RNA.**IntIntVector**(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(**args*)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

class RNA.**IntVector**(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(**args*)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

RNA.**Lfold**(*sequence*, *window_size*, *nullfile=None*)

Local MFE prediction using a sliding window approach (simplified interface).

This simplified interface to RNA.fold_compound.mfe_window() computes the MFE and locally optimal secondary structure using default options. Structures are predicted using a sliding window approach, where base pairs may not span outside the window. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing.

SWIG Wrapper Notes

This function is available as overloaded function *Lfold()* in the global namespace. The parameter *file* defaults to *NULL* and may be omitted. See e.g. *RNA.Lfold()* in the *The RNA Python module*.

Parameters

- **string** (string) – The nucleic acid sequence
- **window_size** (int) – The window size for locally optimal structures
- **file** (FILE *) – The output file handle where predictions are written to (if NULL, output is written to stdout)

See also

RNA.fold_compound.mfe_window, *RNA.Lfoldz*, *RNA.fold_compound.mfe_window_zscore*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use RNA.fold_compound.mfe_window(), and the data structure RNA.fold_compound() instead.

RNA.**Lfold_cb**(*char * string*, *int window_size*, *PyObject * PyFunc*, *PyObject * data*) → float

RNA.**Lfoldz**(*sequence*, *window_size*, *min_z*, *nullfile=None*)

RNA.**Lfoldz_cb**(*char * string*, *int window_size*, *double min_z*, *PyObject * PyFunc*, *PyObject * data*) → float

RNA.**MEA_from_plist**(**args*)

Compute a MEA (maximum expected accuracy) structure from a list of probabilities.

The algorithm maximizes the expected accuracy

$$A(S) = \sum_{(i,j)\in S} 2\gamma p_{ij} + \sum_{i \notin S} p_i^u$$

Higher values of γ result in more base pairs of lower probability and thus higher sensitivity. Low values of γ result in structures containing only highly likely pairs (high specificity). The code of the MEA function also demonstrates the use of sparse dynamic programming scheme to reduce the time and memory complexity of folding.

SWIG Wrapper Notes

This function is available as overloaded function *MEA_from_plist`(gamma = 1., md = NULL). Note, that it returns the MEA structure and MEA value as a tuple (MEA_structure, MEA). See, e.g. :py:func:`RNA.MEA_from_plist()* in the *The RNA Python module*.

Parameters

- **plist** (RNA.ep() *) – A list of base pair probabilities the MEA structure is computed from
- **sequence** (string) – The RNA sequence that corresponds to the list of probability values
- **gamma** (double) – The weighting factor for base pairs vs. unpaired nucleotides
- **md** (RNA.md() *) – A model details data structure (maybe NULL)
- **mea** (list-like(double)) – A pointer to a variable where the MEA value will be written to

Returns

An MEA structure (or NULL on any error)

Return type

string

Note

The unpaired probabilities $p_i^u = 1 - \sum_{j \neq i} p_{ij}$ are usually computed from the supplied pairing probabilities p_{ij} as stored in *plist* entries of type RNA.PLIST_TYPE_BASEPAIR. To overwrite individual p_o^u values simply add entries with type RNA.PLIST_TYPE_UNPAIRED

To include G-Quadruplex support, the corresponding field in *md* must be set.

RNA.**Make_bp_profile**(*length*)

Deprecated since version 2.8.0: This function is deprecated and will be removed soon! See Make_bp_profile_bppm() for a replacement

See also

Make_bp_profile_bppm

Note

This function is NOT threadsafe

RNA.**Make_bp_profile_bppm**(*bppm*, *length*)

condense pair probability matrix into a vector containing probabilities for unpaired, upstream paired and downstream paired.

This resulting probability profile is used as input for profile_edit_distance

Parameters

- **bppm** (list-like(double)) – A pointer to the base pair probability matrix
- **length** (int) – The length of the sequence

Returns
The bp profile

Return type
list-like(double)

RNA.**Make_swString**(*string*)

Convert a structure into a format suitable for string_edit_distance().

Parameters
string (string)

Return type
swString *

class RNA.**MoveVector**(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(**args*)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

RNA.**PS_color_dot_plot**(*string*, *pi*, *filename*)

RNA.**PS_color_dot_plot_turn**(*seq*, *pi*, *filename*, *winSize*)

RNA.**PS_dot_plot**(*string*, *file*)

Produce postscript dot-plot.

Wrapper to PS_dot_plot_list

Reads base pair probabilities produced by pf_fold() from the global array pr and the pair list base_pair produced by fold() and produces a postscript "dot plot" that is written to 'filename'. The "dot plot" represents each base pairing probability by a square of corresponding area in a upper triangle matrix. The lower part of the matrix contains the minimum free energy

Deprecated since version 2.8.0: This function is deprecated and will be removed soon! Use PS_dot_plot_list() instead!

Note

DO NOT USE THIS FUNCTION ANYMORE SINCE IT IS NOT THREADSAFE

RNA.**PS_dot_plot_list**(*seq*, *filename*, *pl*, *mf*, *comment*)

Produce a postscript dot-plot from two pair lists.

This function reads two plist structures (e.g. base pair probabilities and a secondary structure) as produced by assign_plist_from_pr() and assign_plist_from_db() and produces a postscript "dot plot" that is written to 'filename'. Using base pair probabilities in the first and mfe structure in the second plist, the resulting "dot

plot" represents each base pairing probability by a square of corresponding area in a upper triangle matrix. The lower part of the matrix contains the minimum free energy structure.

Parameters

- **seq** (string) – The RNA sequence
- **filename** (string) – A filename for the postscript output
- **pl** (RNA.ep() *) – The base pair probability pairlist
- **mf** (RNA.ep() *) – The mfe secondary structure pairlist
- **comment** (string) – A comment

Returns

1 if postscript was successfully written, 0 otherwise

Return type

int

See also

assign_plist_from_pr, assign_plist_from_db

RNA.**PS_dot_plot_turn**(*seq*, *pl*, *filename*, *winSize*)

RNA.**PS_rna_plot**(*string*, *structure*, *file*)

Produce a secondary structure graph in PostScript and write it to 'filename'.

Deprecated since version 2.8.0: Use RNA.file_PS_rnaplot() instead!

RNA.**PS_rna_plot_a**(*string*, *structure*, *file*, *pre*, *post*)

Produce a secondary structure graph in PostScript including additional annotation macros and write it to 'filename'.

Deprecated since version 2.8.0: Use RNA.file_PS_rnaplot_a() instead!

RNA.**PS_rna_plot_a_gquad**(*string*, *structure*, *ssfile*, *pre*, *post*)

Produce a secondary structure graph in PostScript including additional annotation macros and write it to 'filename' (detect and draw g-quadruplexes).

Deprecated since version 2.8.0: Use RNA.file_PS_rnaplot_a() instead!

class RNA.**PathVector**(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(x)

rbegin()

rend()

reserve(n)

resize(*args)

size()

swap(v)

property thisown

The membership flag

class RNA.SOLUTION

Bases: object

property energy

get(i)

size()

property structure

property thisown

The membership flag

class RNA.SOLUTIONVector(*args)

Bases: object

append(x)

assign(n, x)

back()

begin()

capacity()

clear()

empty()

end()

erase(*args)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(x)

rbegin()

rend()

reserve(n)

resize(*args)

size()

swap(v)

property thisown

The membership flag

class RNA.StringVector(*args)

Bases: object

append(x)

assign(n, x)

back()

begin()

capacity()

clear()

empty()

end()

erase(*args)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(x)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

class RNA.SuboptVector(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(**args*)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

class RNA.SwigPyIterator(**args*, ***kwargs*)

Bases: object

advance(*n*)

copy()

decr(*n=1*)

distance(*x*)

equal(*x*)

incr(*n=1*)

previous()

property thisown

The membership flag

value()

class RNA.UIntUIntVector(**args*)

Bases: object

append(*x*)

assign(*n*, *x*)

back()

begin()

capacity()

clear()

empty()

end()

erase(**args*)

front()

get_allocator()

insert(**args*)

iterator()

pop()

pop_back()

push_back(*x*)

rbegin()

rend()

reserve(*n*)

resize(**args*)

size()

swap(*v*)

property thisown

The membership flag

class RNA.UIntVector(*args)

Bases: object

append(x)

assign(n, x)

back()

begin()

capacity()

clear()

empty()

end()

erase(*args)

front()

get_allocator()

insert(*args)

iterator()

pop()

pop_back()

push_back(x)

rbegin()

rend()

reserve(n)

resize(*args)

size()

swap(v)

property thisown

The membership flag

RNA.abstract_shapes(*std: :string structure*, *unsigned int level=5*) → std::string

RNA.abstract_shapes(*IntVector pt*, *unsigned int level=5*) → std::string

RNA.abstract_shapes(*varArrayShort pt*, *unsigned int level=5*) → std::string

Convert a secondary structure in dot-bracket notation to its abstract shapes representation.

This function converts a secondary structure into its abstract shapes representation as presented by Giegerich *et al.* [2004] .

SWIG Wrapper Notes

This function is available as an overloaded function *abstract_shapes()* where the optional second parameter *level* defaults to 5. See, e.g. `RNA.abstract_shapes()` in the *The RNA Python module*.

Parameters

- **structure** (string) – A secondary structure in dot-bracket notation
- **level** (unsigned int) – The abstraction level (integer in the range of 0 to 5)

Returns
The secondary structure in abstract shapes notation

Return type
string

See also

`RNA.abstract_shapes_pt`

RNA.**add_root**(*arg1*)

Adds a root to an un-rooted tree in any except bracket notation.

Parameters
structure (string)

Return type
string

RNA.**aliLfold**(*alignment*, *window_size*, *nullfile=None*)

SWIG Wrapper Notes

This function is available as overloaded function *aliLfold()* in the global namespace. The parameter *fp* defaults to *NULL* and may be omitted. See e.g. `RNA.aliLfold()` in the *The RNA Python module*.

RNA.**aliLfold_cb**(*StringVector alignment*, *int window_size*, *PyObject * PyFunc*, *PyObject * data*) → float

RNA.**aliduplex_subopt**(*StringVector alignment1*, *StringVector alignment2*, *int delta*, *int w*) → *DuplexVector*

RNA.**aliduplexfold**(*StringVector alignment1*, *StringVector alignment2*) → *duplex_list_t*

RNA.**alifold**(**args*)

Compute Minimum Free Energy (MFE), and a corresponding consensus secondary structure for an RNA sequence alignment using a comparative method.

This simplified interface to RNA.fold_compound.mfe() computes the MFE and, if required, a consensus secondary structure for an RNA sequence alignment using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

SWIG Wrapper Notes

This function is available as function *alifold()* in the global namespace. The parameter *structure* is returned along with the MFE und must not be provided. See e.g. `RNA.alifold()` in the *The RNA Python module*.

Parameters

- **sequences** (const char **) – RNA sequence alignment
- **structure** (string) – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns
the minimum free energy (MFE) in kcal/mol

Return type
float

See also

RNA.circalifold, *RNA.fold_compound.mfe*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use RNA.fold_compound.mfe(), and the data structure RNA.fold_compound() instead.

RNA.**aln_consensus_mis**(*StringVector alignment*, *md md_p=None*) → std::string

Compute the Most Informative Sequence (MIS) for a given multiple sequence alignment.

The most informative sequence (MIS) [Freyhult *et al.*, 2005] displays for each alignment column the nucleotides with frequency greater than the background frequency, projected into IUPAC notation. Columns where gaps are over-represented are in lower case.

SWIG Wrapper Notes

This function is available as overloaded function *aln_consensus_mis()* where the last parameter may be omitted, indicating *md* = *NULL*. See e.g. *RNA.aln_consensus_mis()* in the *The RNA Python module*.

Parameters

- **alignment** (const char **) – The input sequence alignment (last entry must be *NULL* terminated)
- **md_p** (const RNA.md() *) – Model details that specify known nucleotides (Maybe *NULL*)

Returns
The most informative sequence for the alignment

Return type
string

RNA.**aln_consensus_sequence**(*StringVector alignment*, *md md_p=None*) → std::string

Compute the consensus sequence for a given multiple sequence alignment.

SWIG Wrapper Notes

This function is available as overloaded function *aln_consensus_sequence()* where the last parameter may be omitted, indicating *md* = *NULL*. See e.g. *RNA.aln_consensus_sequence()* in the *The RNA Python module*.

Parameters

- **alignment** (const char **) – The input sequence alignment (last entry must be *NULL* terminated)
- **md_p** (const RNA.md() *) – Model details that specify known nucleotides (Maybe *NULL*)

Returns
The consensus sequence of the alignment, i.e. the most frequent nucleotide for each alignment column

Return type
string

RNA.**aln_conservation_col**(*StringVector alignment*, *md md=None*, *unsigned int options*) → *DoubleVector*

Compute nucleotide conservation in an alignment.

This function computes the conservation of nucleotides in alignment columns. The simples measure is Shannon Entropy and can be selected by passing the RNA.MEASURE_SHANNON_ENTROPY flag in the *options* parameter.

SWIG Wrapper Notes

This function is available as overloaded function *aln_conservation_col()* where the last two parameters may be omitted, indicating *md* = *NULL*, and *options* = RNA.MEASURE_SHANNON_ENTROPY, respectively. See e.g. *RNA.aln_conservation_col()* in the *The RNA Python module*.

Parameters

- **alignment** (const char **) – The input sequence alignment (last entry must be *NULL* terminated)
- **md** – Model details that specify known nucleotides (Maybe *NULL*)
- **options** (unsigned int) – A flag indicating which measure of conservation should be applied

Returns
A 1-based vector of column conservations

Return type
list-like(double)

See also

RNA.MEASURE_SHANNON_ENTROPY

Note

Currently, only RNA.MEASURE_SHANNON_ENTROPY is supported as conservation measure.

RNA.**aln_conservation_struct**(*StringVector alignment*, *std: :string structure*, *md md=None*) → *DoubleVector*

Compute base pair conservation of a consensus structure.

This function computes the base pair conservation (fraction of canonical base pairs) of a consensus structure given a multiple sequence alignment. The base pair types that are considered canonical may be specified

using the RNA.md().pair array. Passing *NULL* as parameter *md* results in default pairing rules, i.e. canonical Watson-Crick and GU Wobble pairs.

SWIG Wrapper Notes

This function is available as overloaded function *aln_conservation_struct()* where the last parameter *md* may be omitted, indicating *md* = *NULL*. See, e.g. `RNA.aln_conservation_struct()` in the *The RNA Python module*.

Parameters

- **alignment** (`const char **`) – The input sequence alignment (last entry must be *NULL* terminated)
- **structure** (`string`) – The consensus structure in dot-bracket notation
- **md** (`const RNA.md() *`) – Model details that specify compatible base pairs (Maybe *NULL*)

Returns
A 1-based vector of base pair conservations

Return type
`list-like(double)`

`RNA.aln_mpi(`*StringVector alignment*`)` → int

Get the mean pairwise identity in steps from ?to?(ident).

SWIG Wrapper Notes

This function is available as function *aln_mpi()*. See e.g. `RNA.aln_mpi()` in the *The RNA Python module*.

Parameters
alignment (`const char **`) – Aligned sequences

Returns
The mean pairwise identity

Return type
`int`

`RNA.aln_pscore(`*StringVector alignment*, *md md=None*`)` → *IntIntVector*

SWIG Wrapper Notes

This function is available as overloaded function *aln_pscore()* where the last parameter may be omitted, indicating *md* = *NULL*. See e.g. `RNA.aln_pscore()` in the *The RNA Python module*.

`RNA.assign_plist_gquad_from_pr(`*pl*, *length*, *cut_off*`)`

`RNA.b2C(`*structure*`)`

Converts the full structure from bracket notation to the a coarse grained notation using the 'H' 'B' 'I' 'M' and 'R' identifiers.

Deprecated since version 2.8.0: See RNA.db_to_tree_string() and RNA.STRUCTURE_TREE_SHAPIRO_SHORT for a replacement

Parameters
structure (string)

Return type
string

RNA.**b2HIT**(*structure*)

Converts the full structure from bracket notation to the HIT notation including root.

Deprecated since version 2.8.0: See RNA.db_to_tree_string() and RNA.STRUCTURE_TREE_HIT for a replacement

Parameters
structure (string)

Return type
string

RNA.**b2Shapiro**(*structure*)

Converts the full structure from bracket notation to the *weighted* coarse grained notation using the 'H' 'B' 'I' 'M' 'S' 'E' and 'R' identifiers.

Deprecated since version 2.8.0: See RNA.db_to_tree_string() and RNA.STRUCTURE_TREE_SHAPIRO_WEIGHT for a replacement

Parameters
structure (string)

Return type
string

RNA.**backtrack_GQuad_IntLoop_L**(*c*, *i*, *j*, *type*, *S*, *ggg*, *maxdist*, *p*, *q*, *P*)

backtrack an internal loop like enclosed g-quadruplex with closing pair (i,j) with underlying Lfold matrix

Parameters

- **c** (int) – The total contribution the loop should resemble
- **i** (int) – position i of enclosing pair
- **j** (int) – position j of enclosing pair
- **type** (int) – base pair type of enclosing pair (must be reverse type)
- **S** (list-like(int)) – integer encoded sequence
- **ggg** (int **) – triangular matrix containing g-quadruplex contributions
- **p** (int *) – here the 5' position of the gquad is stored
- **q** (int *) – here the 3' position of the gquad is stored
- **P** (RNA.param() *) – the datastructure containing the precalculated contibutions

Returns
1 on success, 0 if no gquad found

Return type
int

RNA.**backtrack_GQuad_IntLoop_L_comparative**(*c*, *i*, *j*, *type*, *S_cons*, *S5*, *S3*, *a2s*, *ggg*, *p*, *q*, *n_seq*, *P*)

class RNA.**basepair**

Bases: object

Typename for base pair element.

Deprecated since version 2.8.0: Use RNA.bp() instead!

i

Type
int

j

Type
int

Typename for base pair element.

Deprecated since version 2.8.0: Use RNA.bp() instead!

i

Type
int

j

Type
int

property i

property j

property thisown

The membership flag

RNA.**boustrophedon**(**args*)

Generate a sequence of Boustrophedon distributed numbers.

This function generates a sequence of positive natural numbers within the interval $[start, end]$ in a Boustrophedon fashion. That is, the numbers $start, \ldots, end$ in the resulting list are alternating between left and right ends of the interval while progressing to the inside, i.e. the list consists of a sequence of natural numbers of the form:

$$start, end, start + 1, end - 1, start + 2, end - 2, \ldots$$

The resulting list is 1-based and contains the length of the sequence of numbers at it's 0-th position.

Upon failure, the function returns **NULL**

SWIG Wrapper Notes

This function is available as overloaded global function *boustrophedon()*. See, e.g. *RNA.boustrophedon()* in the *The RNA Python module* .

Parameters

- **start** (size()) – The first number of the list (left side of the interval)
- **end** (size()) – The last number of the list (right side of the interval)

Returns
A list of alternating numbers from the interval $[start, end]$ (or **NULL** on error)

Return type
list-like(unsigned int)

See also

RNA.boustrophedon_pos

RNA.**bp_distance**(*std: :string str1*, *std: :string str2*, *unsigned int options*) → int

RNA.**bp_distance**(*IntVector pt1*, *IntVector pt2*) → int

RNA.**bp_distance**(*varArrayShort pt1*, *varArrayShort pt2*) → int

Compute the "base pair" distance between two secondary structures s1 and s2.

This is a wrapper around **RNA.bp_distance_pt()**. The sequences should have the same length. dist = number of base pairs in one structure but not in the other same as edit distance with open-pair close-pair as move-set

SWIG Wrapper Notes

This function is available as an overloaded method **bp_distance()**. Note that the SWIG wrapper takes two structure in dot-bracket notation and converts them into pair tables using RNA.ptable_from_string(). The resulting pair tables are then internally passed to RNA.bp_distance_pt(). To control which kind of matching brackets will be used during conversion, the optional argument *options* can be used. See also the description of RNA.ptable_from_string() for available options. (default: **RNA.BRACKETS_RND**). See, e.g. *RNA.bp_distance()* in the *The RNA Python module*.

Parameters

- **str1** (string) – First structure in dot-bracket notation
- **str2** (string) – Second structure in dot-bracket notation

Returns

The base pair distance between str1 and str2

Return type

int

See also

RNA.bp_distance_pt

RNA.**cdata**(*ptr*, *nelements=1*)

RNA.**centroid**(*length*, *dist*)

Deprecated since version 2.8.0: This function is deprecated and should not be used anymore as it is not threadsafe!

See also

get_centroid_struct_pl, *get_centroid_struct_pr*

RNA.**circalifold**(**args*)

Compute Minimum Free Energy (MFE), and a corresponding consensus secondary structure for a sequence alignment of circular RNAs using a comparative method.

This simplified interface to RNA.fold_compound.mfe() computes the MFE and, if required, a consensus secondary structure for an RNA sequence alignment using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

Folding of circular RNA sequences is handled as a post-processing step of the forward recursions. See Hofacker and Stadler [2006] for further details.

SWIG Wrapper Notes

This function is available as function *circalifold()* in the global namespace. The parameter *structure* is returned along with the MFE und must not be provided. See e.g. *RNA.circalifold()* in the *The RNA Python module*.

Parameters

- **sequences** (const char **) – Sequence alignment of circular RNAs
- **structure** (string) – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns
the minimum free energy (MFE) in kcal/mol

Return type
float

See also

RNA.alifold, *RNA.fold_compound.mfe*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use RNA.fold_compound.mfe(), and the data structure RNA.fold_compound() instead.

RNA.**circfold**(**args*)

Compute Minimum Free Energy (MFE), and a corresponding secondary structure for a circular RNA sequence.

This simplified interface to RNA.fold_compound.mfe() computes the MFE and, if required, a secondary structure for a circular RNA sequence using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

Folding of circular RNA sequences is handled as a post-processing step of the forward recursions. See Hofacker and Stadler [2006] for further details.

SWIG Wrapper Notes

This function is available as function *circfold()* in the global namespace. The parameter *structure* is returned along with the MFE und must not be provided. See e.g. *RNA.circfold()* in the *The RNA Python module*.

Parameters

- **sequence** (string) – RNA sequence
- **structure** (string) – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns
the minimum free energy (MFE) in kcal/mol

Return type
`float`

See also

RNA.fold, *RNA.fold_compound.mfe*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use RNA.fold_compound.mfe(), and the data structure RNA.fold_compound() instead.

`class RNA.cmd`

Bases: `object`

`property thisown`

The membership flag

`RNA.co_pf_fold(*args)`

`RNA.cofold(*args)`

Compute Minimum Free Energy (MFE), and a corresponding secondary structure for two dimerized RNA sequences.

This simplified interface to RNA.fold_compound.mfe() computes the MFE and, if required, a secondary structure for two RNA sequences upon dimerization using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

Deprecated since version 2.8.0: This function is obsolete since RNA.mfe()/RNA.fold() can handle complexes multiple sequences since v2.5.0. Use RNA.mfe()/RNA.fold() for connected component MFE instead and compute MFEs of unconnected states separately.

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use RNA.fold_compound.mfe(), and the data structure RNA.fold_compound() instead.

SWIG Wrapper Notes

This function is available as function *cofold()* in the global namespace. The parameter *structure* is returned along with the MFE und must not be provided. See e.g. *RNA.cofold()* in the *The RNA Python module*.

Parameters

- **sequence** (`string`) – two RNA sequences separated by the '&' character
- **structure** (`string`) – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns
the minimum free energy (MFE) in kcal/mol

Return type
`float`

See also

RNA.fold, *RNA.fold_compound.mfe*, *RNA.fold_compound*, *RNA.fold_compound*, RNA.cut_point_insert

RNA.**compare_structure**(*std: :string str1*, *std: :string str2*, *int fuzzy=0*, *unsigned int options*) → *score*

RNA.**compare_structure**(*IntVector pt1*, *IntVector pt2*, *int fuzzy=0*) → *score*

RNA.**compare_structure**(*varArrayShort pt1*, *varArrayShort pt2*, *int fuzzy=0*) → *score*

RNA.**consens_mis**(*alignment*, *md_p=None*)

RNA.**db_compat_remove**(*std: :string structure*, *std: :string structure_reference*) → std::string

RNA.**db_flatten**(**args*)

Substitute pairs of brackets in a string with parenthesis.

This function can be used to replace brackets of unusual types, such as angular brackets <> , to dot-bracket format. The *options* parameter is used tpo specify which types of brackets will be replaced by round parenthesis ``() .

SWIG Wrapper Notes

This function flattens an input structure string in-place! The second parameter is optional and defaults to RNA.BRACKETS_DEFAULT.

An overloaded version of this function exists, where an additional second parameter can be passed to specify the target brackets, i.e. the type of matching pair characters all brackets will be flattened to. Therefore, in the scripting language interface this function is a replacement for RNA.db_flatten_to(). See, e.g. *RNA.db_flatten()* in the *The RNA Python module*.

Parameters

- **structure** (`string`) – The structure string where brackets are flattened in-place
- **options** (`unsigned int`) – A bitmask to specify which types of brackets should be flattened out

See also

RNA.db_flatten_to, RNA.BRACKETS_RND, RNA.BRACKETS_ANG, RNA.BRACKETS_CLY, RNA.BRACKETS_SQR, RNA.BRACKETS_DEFAULT

RNA.**db_from_WUSS**(*wuss*)

Convert a WUSS annotation string to dot-bracket format.

Parameters
wuss (`string`) – The input string in WUSS notation

Returns
A dot-bracket notation of the input secondary structure

Return type
string

Note

This function flattens all brackets, and treats pseudo-knots annotated by matching pairs of upper/lowercase letters as unpaired nucleotides

RNA.**db_from_plist**(*ElemProbVector elem_probs*, *unsigned int length*) → std::string

Convert a list of base pairs into dot-bracket notation.

Parameters

- **pairs** (RNA.ep() *) – A RNA.ep() containing the pairs to be included in the dot-bracket string
- **n** (unsigned int) – The length of the structure (number of nucleotides)

Returns
The dot-bracket string containing the provided base pairs

Return type
string

See also

RNA.plist

RNA.**db_from_ptable**(*IntVector pt*) → char

RNA.**db_from_ptable**(*varArrayShort pt*) → char *

Convert a pair table into dot-parenthesis notation.

This function also converts pair table formatted structures that contain pseudoknots. Non-nested base pairs result in additional pairs of parenthesis and brackets within the resulting dot- parenthesis string. The following pairs are available: (), []. {}. <>, as well as pairs of matching upper-/lower-case characters from the alphabet A-Z.

Parameters
pt (const short *) – The pair table to be copied

Returns
A char pointer to the dot-bracket string

Return type
string

Note

In cases where the level of non-nested base pairs exceeds the maximum number of 30 different base pair indicators (4 parenthesis/brackets, 26 matching characters), a warning is printed and the remaining base pairs are left out from the conversion.

RNA.**db_pack**(*struc*)

Pack secondary secondary structure, 5:1 compression using base 3 encoding.

Returns a binary string encoding of the secondary structure using a 5:1 compression scheme. The string is NULL terminated and can therefore be used with standard string functions such as strcmp(). Useful for programs that need to keep many structures in memory.

Parameters
struc (string) – The secondary structure in dot-bracket notation

Returns
The binary encoded structure

Return type
string

See also

RNA.db_unpack

RNA.**db_pk_remove**(*std: :string structure*, *unsigned int options*) → std::string

Remove pseudo-knots from an input structure.

This function removes pseudo-knots from an input structure by determining the minimum number of base pairs that need to be removed to make the structure pseudo-knot free.

To accomplish that, we use a dynamic programming algorithm similar to the Nussinov maxmimum matching approach.

The input structure must be in a dot-bracket string like form where crossing base pairs are denoted by the use of additional types of matching brackets, e.g. <>, *{}*, ``[], *{}*. Furthermore, crossing pairs may be annotated by matching uppercase/lowercase letters from the alphabet *A-Z*. For the latter, the uppercase letter must be the 5' and the lowercase letter the 3' nucleotide of the base pair. The actual type of brackets to be recognized by this function must be specifed through the *options* parameter.

SWIG Wrapper Notes

This function is available as an overloaded function *db_pk_remove()* where the optional second parameter *options* defaults to RNA.BRACKETS_ANY. See, e.g. *RNA.db_pk_remove()* in the *The RNA Python module*.

Parameters

- **structure** (string) – Input structure in dot-bracket format that may include pseudo-knots
- **options** (unsigned int) – A bitmask to specify which types of brackets should be processed

Returns
The input structure devoid of pseudo-knots in dot-bracket notation

Return type
string

See also

RNA.pt_pk_remove, *RNA.db_flatten*, RNA.BRACKETS_RND, RNA.BRACKETS_ANG, RNA.BRACKETS_CLY, RNA.BRACKETS_SQR, RNA.BRACKETS_ALPHA, RNA.BRACKETS_DEFAULT, RNA.BRACKETS_ANY

Note

Brackets in the input structure string that are not covered by the *options* bitmask will be silently ignored!

RNA.**db_to_element_string**(*structure*)

Convert a secondary structure in dot-bracket notation to a nucleotide annotation of loop contexts.

Parameters
structure (string) – The secondary structure in dot-bracket notation

Returns
A string annotating each nucleotide according to it's structural context

Return type
string

RNA.**db_to_tree_string**(*std: :string structure*, *unsigned int type*) → std::string

Convert a Dot-Bracket structure string into tree string representation.

This function allows one to convert a secondary structure in dot-bracket notation into one of the various tree representations for secondary structures. The resulting tree is then represented as a string of parenthesis and node symbols, similar to to the Newick format.

Currently we support conversion into the following formats, denoted by the value of parameter *type:*

- RNA.STRUCTURE_TREE_HIT - Homeomorphically Irreducible Tree (HIT) representation of a secondary structure. (See also Fontana *et al.* [1993])
- RNA.STRUCTURE_TREE_SHAPIRO_SHORT - (short) Coarse Grained representation of a secondary structure (same as Shapiro [1988] , but with root node *R* and without *S* nodes for the stems)
- RNA.STRUCTURE_TREE_SHAPIRO - (full) Coarse Grained representation of a secondary structure (See also Shapiro [1988])
- RNA.STRUCTURE_TREE_SHAPIRO_EXT - (extended) Coarse Grained representation of a secondary structure (same as Shapiro [1988] , but external nodes denoted as *E*)
- RNA.STRUCTURE_TREE_SHAPIRO_WEIGHT - (weighted) Coarse Grained representation of a secondary structure (same as RNA.STRUCTURE_TREE_SHAPIRO_EXT but with additional weights for number of unpaired nucleotides in loop, and number of pairs in stems)
- RNA.STRUCTURE_TREE_EXPANDED - Expanded Tree representation of a secondary structure.

Parameters
- **structure** (string) – The null-terminated dot-bracket structure string
- **type** (unsigned int) – A switch to determine the type of tree string representation

Returns
A tree representation of the input *structure*

Return type
string

RNA.**db_unpack**(*packed*)

Unpack secondary structure previously packed with RNA.db_pack().

Translate a compressed binary string produced by RNA.db_pack() back into the familiar dot-bracket notation.

Parameters
packed (string) – The binary encoded packed secondary structure

Returns
The unpacked secondary structure in dot-bracket notation

Return type
string

> **See also**
>
> *RNA.db_pack*

RNA.**delete_doubleP**(*ary*)

RNA.**delete_floatP**(*ary*)

RNA.**delete_intP**(*ary*)

RNA.**delete_shortP**(*ary*)

RNA.**delete_ushortP**(*ary*)

RNA.**deref_any**(*ptr*, *index*)

RNA.**dist_mountain**(*str1*, *str2*, *p=1*)

class RNA.**doubleArray**(*nelements*)

Bases: object

cast()

static frompointer(*t*)

property thisown

The membership flag

RNA.**doubleP_getitem**(*ary*, *index*)

RNA.**doubleP_setitem**(*ary*, *index*, *value*)

class RNA.**duplexT**(**args*, ***kwargs*)

Bases: object

Data structure for RNAduplex.

i

Type
int

j

Type
int

end

Type
int

structure

Type
string

energy

Type
double

energy_backtrack

Type
double

opening_backtrack_x

Type
double

opening_backtrack_y

Type
double

offset

Type
int

dG1

Type
double

dG2

Type
double

ddG

Type
double

tb

Type
int

te

Type
int

qb

Type
int

qe

Type
int

property dG1

property dG2

property ddG

property end

property energy

property energy_backtrack

property i

property j

property offset

property opening_backtrack_x

property opening_backtrack_y

property qb

property qe

property structure

property tb

property te

property thisown

The membership flag

class RNA.duplex_list_t

Bases: object

property energy

property i

property j

property structure

property thisown

The membership flag

RNA.duplex_subopt(*std: :string s1*, *std: :string s2*, *int delta*, *int w*) → *DuplexVector*

RNA.duplexfold(*std: :string s1*, *std: :string s2*) → *duplex_list_t*

RNA.encode_seq(*sequence*)

RNA.energy_of_circ_struct(*string*, *structure*)

Calculate the free energy of an already folded circular RNA

Deprecated since version 2.8.0: This function is deprecated and should not be used in future programs Use energy_of_circ_structure() instead!

Note

This function is not entirely threadsafe! Depending on the state of the global variable eos_debug it prints energy information to stdout or not...

Parameters

- **string** (string) – RNA sequence
- **structure** (string) – secondary structure in dot-bracket notation

Returns

the free energy of the input structure given the input sequence in kcal/mol

Return type

float

See also

energy_of_circ_structure, *energy_of_struct*, *energy_of_struct_pt*

RNA.**energy_of_circ_structure**(*string*, *structure*, *verbosity_level*)

Calculate the free energy of an already folded circular RNA.

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Note

OpenMP: This function relies on several global model settings variables and thus is not to be considered threadsafe. See energy_of_circ_struct_par() for a completely threadsafe implementation.

Deprecated since version 2.8.0: Use RNA.fold_compound.eval_structure() or RNA.fold_compound.eval_structure_verbose() instead!

Parameters

- **string** (string) – RNA sequence
- **structure** (string) – Secondary structure in dot-bracket notation
- **verbosity_level** (int) – A flag to turn verbose output on/off

Returns
The free energy of the input structure given the input sequence in kcal/mol

Return type
float

See also

RNA.fold_compound.eval_structure

RNA.**energy_of_gquad_structure**(*string*, *structure*, *verbosity_level*)

RNA.**energy_of_move**(*string*, *structure*, *m1*, *m2*)

Calculate energy of a move (closing or opening of a base pair).

If the parameters m1 and m2 are negative, it is deletion (opening) of a base pair, otherwise it is insertion (opening).

Deprecated since version 2.8.0: Use RNA.fold_compound.eval_move() instead!

Parameters

- **string** (string) – RNA sequence
- **structure** (string) – secondary structure in dot-bracket notation
- **m1** (int) – first coordinate of base pair
- **m2** (int) – second coordinate of base pair

Returns
energy change of the move in kcal/mol

Return type
float

See also

RNA.fold_compound.eval_move

RNA.**energy_of_move_pt**(*pt*, *s*, *s1*, *m1*, *m2*)

Calculate energy of a move (closing or opening of a base pair).

If the parameters m1 and m2 are negative, it is deletion (opening) of a base pair, otherwise it is insertion (opening).

Deprecated since version 2.8.0: Use RNA.fold_compound.eval_move_pt() instead!

Parameters

- **pt** (list-like(int)) – the pair table of the secondary structure
- **s** (list-like(int)) – encoded RNA sequence
- **s1** (list-like(int)) – encoded RNA sequence
- **m1** (int) – first coordinate of base pair
- **m2** (int) – second coordinate of base pair

Returns
energy change of the move in 10cal/mol

Return type
int

See also

RNA.fold_compound.eval_move_pt

RNA.**energy_of_struct**(*string*, *structure*)

Calculate the free energy of an already folded RNA

Deprecated since version 2.8.0: This function is deprecated and should not be used in future programs! Use energy_of_structure() instead!

Note

This function is not entirely threadsafe! Depending on the state of the global variable eos_debug it prints energy information to stdout or not...

Parameters

- **string** (string) – RNA sequence
- **structure** (string) – secondary structure in dot-bracket notation

Returns
the free energy of the input structure given the input sequence in kcal/mol

Return type
float

See also

energy_of_structure, *energy_of_circ_struct*, *energy_of_struct_pt*

RNA.**energy_of_struct_pt**(*string*, *ptable*, *s*, *s1*)

Calculate the free energy of an already folded RNA

Deprecated since version 2.8.0: This function is deprecated and should not be used in future programs! Use energy_of_structure_pt() instead!

Note

This function is not entirely threadsafe! Depending on the state of the global variable eos_debug it prints energy information to stdout or not...

Parameters

- **string** (string) – RNA sequence
- **ptable** (list-like(int)) – the pair table of the secondary structure
- **s** (list-like(int)) – encoded RNA sequence
- **s1** (list-like(int)) – encoded RNA sequence

Returns
the free energy of the input structure given the input sequence in 10kcal/mol

Return type
int

See also

make_pair_table, *energy_of_structure*

RNA.**energy_of_structure**(*string*, *structure*, *verbosity_level*)

Calculate the free energy of an already folded RNA using global model detail settings.

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Deprecated since version 2.8.0: Use RNA.fold_compound.eval_structure() or RNA.fold_compound.eval_structure_verbose() instead!

Note

OpenMP: This function relies on several global model settings variables and thus is not to be considered threadsafe. See energy_of_struct_par() for a completely threadsafe implementation.

Parameters

- **string** (string) – RNA sequence
- **structure** (string) – secondary structure in dot-bracket notation
- **verbosity_level** (int) – a flag to turn verbose output on/off

Returns
the free energy of the input structure given the input sequence in kcal/mol

Return type
float

See also

RNA.fold_compound.eval_structure

RNA.**energy_of_structure_pt**(*string*, *ptable*, *s*, *s1*, *verbosity_level*)

Calculate the free energy of an already folded RNA.

If verbosity level is set to a value >0, energies of structure elements are printed to stdout

Deprecated since version 2.8.0: Use RNA.fold_compound.eval_structure_pt() or RNA.fold_compound.eval_structure_pt_verbose() instead!

Note

OpenMP: This function relies on several global model settings variables and thus is not to be considered threadsafe. See energy_of_struct_pt_par() for a completely threadsafe implementation.

Parameters

- **string** (string) – RNA sequence
- **ptable** (list-like(int)) – the pair table of the secondary structure
- **s** (list-like(int)) – encoded RNA sequence
- **s1** (list-like(int)) – encoded RNA sequence
- **verbosity_level** (int) – a flag to turn verbose output on/off

Returns
the free energy of the input structure given the input sequence in 10kcal/mol

Return type
int

See also

RNA.fold_compound.eval_structure_pt

RNA.**enumerate_necklaces**(*entity_counts*)

Enumerate all necklaces with fixed content.

This function implements *A fast algorithm to generate necklaces with fixed content* as published by Sawada [2003] .

The function receives a list of counts (the elements on the necklace) for each type of object within a necklace. The list starts at index 0 and ends with an entry that has a count of 0. The algorithm then enumerates all non-cyclic permutations of the content, returned as a list of necklaces. This list, again, is zero-terminated, i.e. the last entry of the list is a *NULL* pointer.

SWIG Wrapper Notes

This function is available as global function *enumerate_necklaces()* which accepts lists input, an produces list of lists output. See, e.g. *RNA.enumerate_necklaces()* in the *The RNA Python module* .

Parameters
type_counts (const unsigned int *) – A 0-terminated list of entity counts

Returns
A list of all non-cyclic permutations of the entities

Return type
`list-like(list-like(unsigned int))`

`class RNA.ep(*args, **kwargs)`

Bases: `object`

Data structure representing a single entry of an element probability list (e.g. list of pair probabilities).

See also

RNA.plist, *RNA.fold_compound.plist_from_probs*, *RNA.db_from_plist*, RNA.PLIST_TYPE_BASEPAIR, RNA.PLIST_TYPE_GQUAD, RNA.PLIST_TYPE_H_MOTIF, RNA.PLIST_TYPE_I_MOTIF, RNA.PLIST_TYPE_UD_MOTIF, RNA.PLIST_TYPE_STACK

i

Start position (usually 5' nucleotide that starts the element, e.g. base pair).

Type
`int`

j

End position (usually 3' nucleotide that ends the element, e.g. base pair).

Type
`int`

p

Probability of the element.

Type
`float`

type

Type of the element.

Type
`int`

size

The size of the element (e.g. number of base pairs that constitute a helix).

Type
`unsigned int`

Data structure representing a single entry of an element probability list (e.g. list of pair probabilities).

See also

RNA.plist, *RNA.fold_compound.plist_from_probs*, *RNA.db_from_plist*, RNA.PLIST_TYPE_BASEPAIR, RNA.PLIST_TYPE_GQUAD, RNA.PLIST_TYPE_H_MOTIF, RNA.PLIST_TYPE_I_MOTIF, RNA.PLIST_TYPE_UD_MOTIF, RNA.PLIST_TYPE_STACK

i

Start position (usually 5' nucleotide that starts the element, e.g. base pair).

Type
`int`

j

End position (usually 3' nucleotide that ends the element, e.g. base pair).

Type
`int`

p

Probability of the element.

Type
`float`

type

Type of the element.

Type
`int`

size

The size of the element (e.g. number of base pairs that constitute a helix).

Type
`unsigned int`

property i

property j

property p

property size

property thisown

The membership flag

property type

RNA.**eqm_probs_bp**(*fold_compound fc*, *double cutoff=1e-5*) → *ElemProbVector*

RNA.**eqm_probs_hx**(*fold_compound fc*, *unsigned int min_size=1*, *unsigned int max_size=0*, *double cutoff=1e-5*) → *ElemProbVector*

class RNA.**eqm_probs_opt**

Bases: `object`

mode

Type
`unsigned int`

up_length

Type
`unsigned int`

hx_length

Type
`unsigned int`

mode

Type
`unsigned int`

up_length

Type
unsigned int

hx_length

Type
unsigned int

property hx_length

property mode

property thisown

The membership flag

property up_length

RNA.**eqm_probs_up**(*fold_compound fc*, *unsigned int min_u=1*, *unsigned int max_u=0*, *double cutoff=1e-5*) → *ElemProbVector*

RNA.**equilibrium_conc**(**args*, ***kwargs*)

RNA.**equilibrium_constants**(**args*, ***kwargs*)

RNA.**eval_circ_gquad_structure**(**args*)

Evaluate free energy of a sequence/structure pair, assume sequence to be circular, allow for G-Quadruplexes in the structure, and print contributions per loop.

This function is the same as RNA.eval_structure_simple_v() but assumes the input sequence to be circular and allows for annotated G-Quadruplexes in the dot-bracket structure input.

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

SWIG Wrapper Notes

This function is available through an overloaded version of RNA.eval_circ_gquad_structure(). The last two arguments for this function are optional and default to RNA.VERBOSITY_QUIET and *NULL*, respectively. See, e.g. *RNA.eval_circ_gquad_structure()* in the *The RNA Python module* .

Parameters

- **string** (string) – RNA sequence in uppercase letters
- **structure** (string) – Secondary structure in dot-bracket notation
- **verbosity_level** (int) – The level of verbosity of this function
- **file** (FILE *) – A file handle where this function should print to (may be NULL).

Returns
The free energy of the input structure given the input sequence in kcal/mol

Return type
float

RNA.**eval_circ_structure**(**args*)

Evaluate free energy of a sequence/structure pair, assume sequence to be circular and print contributions per loop.

This function is the same as RNA.eval_structure_simple_v() but assumes the input sequence to be circularized.

SWIG Wrapper Notes

This function is available through an overloaded version of RNA.eval_circ_structure(). The last two arguments for this function are optional and default to RNA.VERBOSITY_QUIET and *NULL*, respectively. See, e.g. *RNA.eval_circ_structure()* in the *The RNA Python module* .

Parameters

- **string** (string) – RNA sequence in uppercase letters
- **structure** (string) – Secondary structure in dot-bracket notation
- **verbosity_level** (int) – The level of verbosity of this function
- **file** (FILE *) – A file handle where this function should print to (may be NULL).

Returns
The free energy of the input structure given the input sequence in kcal/mol

Return type
float

See also

RNA.eval_structure_simple_v, *RNA.eval_circ_structure*, *RNA.fold_compound.eval_structure_verbose*

RNA.**eval_gquad_structure**(**args*)

Evaluate free energy of a sequence/structure pair, allow for G-Quadruplexes in the structure and print contributions per loop.

This function is the same as RNA.eval_structure_simple_v() but allows for annotated G-Quadruplexes in the dot-bracket structure input.

G-Quadruplexes are annotated as plus signs ('+') for each G involved in the motif. Linker sequences must be denoted by dots ('.') as they are considered unpaired. Below is an example of a 2-layer G-quadruplex:

SWIG Wrapper Notes

This function is available through an overloaded version of RNA.eval_gquad_structure(). The last two arguments for this function are optional and default to RNA.VERBOSITY_QUIET and *NULL*, respectively. See, e.g. *RNA.eval_gquad_structure()* in the *The RNA Python module* .

Parameters

- **string** (string) – RNA sequence in uppercase letters
- **structure** (string) – Secondary structure in dot-bracket notation
- **verbosity_level** (int) – The level of verbosity of this function
- **file** (FILE *) – A file handle where this function should print to (may be NULL).

Returns
The free energy of the input structure given the input sequence in kcal/mol

Return type
float

See also

RNA.eval_structure_simple_v, *RNA.eval_gquad_structure*, *RNA.fold_compound.eval_structure_verbose*

RNA.**eval_structure_pt_simple**(**args*)

Calculate the free energy of an already folded RNA.

This function allows for energy evaluation of a given sequence/structure pair where the structure is provided in pair_table format as obtained from RNA.ptable(). Model details, energy parameters, and possibly soft constraints are used as provided via the parameter 'fc'. The fold_compound does not need to contain any DP matrices, but all the most basic init values as one would get from a call like this: In contrast to RNA.fold_compound.eval_structure_pt_verbose() this function assumes default model details and default energy parameters in order to evaluate the free energy of the secondary structure. Threefore, it serves as a simple interface function for energy evaluation for situations where no changes on the energy model are required.

Parameters

- **string** (string) – RNA sequence in uppercase letters
- **pt** (const short *) – Secondary structure as pair_table
- **verbosity_level** (int) – The level of verbosity of this function
- **file** (FILE *) – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in 10cal/mol

Return type

int

See also

RNA.ptable, RNA.eval_structure_pt_v, *RNA.eval_structure_simple*

RNA.**eval_structure_simple**(**args*)

Calculate the free energy of an already folded RNA and print contributions per loop.

This function allows for detailed energy evaluation of a given sequence/structure pair. In contrast to RNA.fold_compound.eval_structure() this function prints detailed energy contributions based on individual loops to a file handle. If NULL is passed as file handle, this function defaults to print to stdout. Any positive *verbosity_level* activates potential warning message of the energy evaluting functions, while values ≥ 1 allow for detailed control of what data is printed. A negative parameter *verbosity_level* turns off printing all together.

In contrast to RNA.fold_compound.eval_structure_verbose() this function assumes default model details and default energy parameters in order to evaluate the free energy of the secondary structure. Threefore, it serves as a simple interface function for energy evaluation for situations where no changes on the energy model are required.

SWIG Wrapper Notes

This function is available through an overloaded version of RNA.eval_structure_simple(). The last two arguments for this function are optional and default to RNA.VERBOSITY_QUIET and *NULL*, respectively. See, e.g. *RNA.eval_structure_simple()* in the *The RNA Python module* .

Parameters

- **string** (string) – RNA sequence in uppercase letters
- **structure** (string) – Secondary structure in dot-bracket notation
- **verbosity_level** (int) – The level of verbosity of this function
- **file** (FILE *) – A file handle where this function should print to (may be NULL).

Returns
The free energy of the input structure given the input sequence in kcal/mol

Return type
float

See also

RNA.fold_compound.eval_structure_verbose, *RNA.fold_compound.eval_structure_pt*, *RNA.fold_compound.eval_structure_pt_verbose*

RNA.**exp_E_ExtLoop**(*type*, *si1*, *sj1*, *P*)

This is the partition function variant of E_ExtLoop()

Deprecated since version 2.8.0: Use RNA.fold_compound.exp_E_ext_stem() instead!

Returns
The Boltzmann weighted energy contribution of the introduced exterior-loop stem

Return type
double

See also

E_ExtLoop

RNA.**exp_E_Hairpin**(*u*, *type*, *si1*, *sj1*, *string*, *P*)

Compute Boltzmann weight $e^{-\Delta G/kT}$ of a hairpin loop.

Parameters

- **u** (int) – The size of the loop (number of unpaired nucleotides)
- **type** (int) – The pair type of the base pair closing the hairpin
- **si1** (short) – The 5'-mismatching nucleotide
- **sj1** (short) – The 3'-mismatching nucleotide
- **string** (string) – The sequence of the loop (May be *NULL*, otherwise mst be at least $size + 2$ long)
- **P** (RNA.exp_param() *) – The datastructure containing scaled Boltzmann weights of the energy parameters

Returns
The Boltzmann weight of the Hairpin-loop

Return type
double

Warning

Not (really) thread safe! A threadsafe implementation will replace this function in a future release!

Energy evaluation may change due to updates in global variable "tetra_loop"

See also

`get_scaled_pf_parameters`, *RNA.exp_param*, *E_Hairpin*

Note

multiply by scale[u+2]

RNA.**exp_E_IntLoop**(*u1*, *u2*, *type*, *type2*, *si1*, *sj1*, *sp1*, *sq1*, *P*)

Compute Boltzmann weight $e^{-\Delta G/kT}$ of internal loop

multiply by scale[u1+u2+2] for scaling

param u1
The size of the 'left'-loop (number of unpaired nucleotides)

type u1
`int`

param u2
The size of the 'right'-loop (number of unpaired nucleotides)

type u2
`int`

param type
The pair type of the base pair closing the internal loop

type type
`int`

param type2
The pair type of the enclosed base pair

type type2
`int`

param si1
The 5'-mismatching nucleotide of the closing pair

type si1
`short`

param sj1
The 3'-mismatching nucleotide of the closing pair

type sj1
`short`

param sp1
The 3'-mismatching nucleotide of the enclosed pair

type sp1
`short`

param sq1
The 5'-mismatching nucleotide of the enclosed pair

type sq1
short

param P
The datastructure containing scaled Boltzmann weights of the energy parameters

type P
RNA.exp_param() *

returns
The Boltzmann weight of the Interior-loop

rtype
double

See also

get_scaled_pf_parameters, *RNA.exp_param*, *E_IntLoop*

Note

This function is threadsafe

RNA.**exp_E_MLstem**(*type*, *si1*, *sj1*, *P*)

RNA.**exp_E_Stem**(*type*, *si1*, *sj1*, *extLoop*, *P*)

Compute the Boltzmann weighted energy contribution of a stem branching off a loop-region

This is the partition function variant of E_Stem()

returns
The Boltzmann weighted energy contribution of the branch off the loop

rtype
double

See also

E_Stem

Note

This function is threadsafe

RNA.**exp_E_gquad**(*L*, *l*, *pf*)

RNA.**exp_E_gquad_ali**(*i*, *L*, *l*, *S*, *a2s*, *n_seq*, *pf*)

class RNA.**exp_param**(*model_details=None*)

Bases: object

The data structure that contains temperature scaled Boltzmann weights of the energy parameters.

id

An identifier for the data structure.

Deprecated since version 2.8.0: This attribute will be removed in version 3

Type
int

expstack

Type
double

exphairpin

Type
double

expbulge

Type
double

expinternal

Type
double

expmismatchExt

Type
double

expmismatchI

Type
double

expmismatch23I

Type
double

expmismatch1nI

Type
double

expmismatchH

Type
double

expmismatchM

Type
double

expdangle5

Type
double

expdangle3

Type
double

expint11

Type
double

expint21

Type
double

expint22

Type
double

expninio

Type
double

lxc

Type
double

expMLbase

Type
double

expMLintern

Type
double

expMLclosing

Type
double

expTermAU

Type
double

expDuplexInit

Type
double

exptetra

Type
double

exptri

Type
double

exphex

Type
double

Tetraloops

Type
char

expTriloop

Type
double

Triloops

Type
char

Hexaloops

Type
char

expTripleC

Type
double

expMultipleCA

Type
double

expMultipleCB

Type
double

expgquad

Type
double

expgquadLayerMismatch

Type
double

gquadLayerMismatchMax

Type
unsigned int

kT

Type
double

pf_scale

Scaling factor to avoid over-/underflows.

Type
double

temperature

Temperature used for loop contribution scaling.

Type
double

alpha

Scaling factor for the thermodynamic temperature.

This allows for temperature scaling in Boltzmann factors independently from the energy contributions. The resulting Boltzmann factors are then computed by $e^{-E/(\alpha \cdot K \cdot T)}$

Type
double

model_details

Model details to be used in the recursions.

Type
vrna_md_t

param_file

The filename the parameters were derived from, or empty string if they represent the default.

Type
char

expSaltStack

Type
double

expSaltLoop

Type
double

SaltLoopDbl

Type
double

SaltMLbase

Type
int

SaltMLintern

Type
int

SaltMLclosing

Type
int

SaltDPXInit

Type
int

The data structure that contains temperature scaled Boltzmann weights of the energy parameters.

id

An identifier for the data structure.

Deprecated since version 2.8.0: This attribute will be removed in version 3

Type
int

expstack

Type
double

exphairpin

Type
double

expbulge

Type
double

expinternal

Type
double

expmismatchExt

Type
double

expmismatchI

Type
double

expmismatch23I

Type
double

expmismatch1nI

Type
double

expmismatchH

Type
double

expmismatchM

Type
double

expdangle5

Type
double

expdangle3

Type
double

expint11

Type
double

expint21

Type
double

expint22

Type
double

expninio

Type
double

lxc

Type
double

expMLbase

Type
double

expMLintern

Type
double

expMLclosing

Type
double

expTermAU

Type
double

expDuplexInit

Type
double

exptetra

Type
double

exptri

Type
double

exphex

Type
double

Tetraloops

Type
char

expTriloop

Type
double

Triloops

Type
char

Hexaloops

Type
char

expTripleC

Type
double

expMultipleCA

Type
double

expMultipleCB

Type
double

expgquad

Type
double

expgquadLayerMismatch

Type
double

gquadLayerMismatchMax

Type
unsigned int

kT

Type
double

pf_scale

Scaling factor to avoid over-/underflows.

Type
double

temperature

Temperature used for loop contribution scaling.

Type
double

alpha

Scaling factor for the thermodynamic temperature.

This allows for temperature scaling in Boltzmann factors independently from the energy contributions. The resulting Boltzmann factors are then computed by $e^{-E/(\alpha \cdot K \cdot T)}$

Type
double

model_details

Model details to be used in the recursions.

Type
vrna_md_t

param_file

The filename the parameters were derived from, or empty string if they represent the default.

Type
char

expSaltStack

Type
double

expSaltLoop

Type
double

SaltLoopDbl

Type
double

SaltMLbase

Type
int

SaltMLintern

Type
int

SaltMLclosing

Type
int

SaltDPXInit

Type
int

property Hexaloops

property SaltDPXInit

property SaltLoopDbl

property SaltMLbase

property SaltMLclosing

property SaltMLintern

property Tetraloops

property Triloops

property alpha

property expDuplexInit

property expMLbase

property expMLclosing

property expMLintern

property expMultipleCA

property expMultipleCB

property expSaltLoop

property expSaltStack

property expTermAU

property expTriloop

property expTripleC

property expbulge

property expdangle3

property expdangle5

property expgquad

property expgquadLayerMismatch

property exphairpin

property exphex

property expint11

property expint21

property expint22

property expinternal

property expmismatch1nI

property expmismatch23I

property expmismatchExt

property expmismatchH

property expmismatchI

property expmismatchM

property expninio

property expstack

property exptetra

property exptri

property gquadLayerMismatchMax

property id

property kT

property lxc

property model_details

property param_file

property pf_scale

property temperature

property thisown

The membership flag

RNA.**expand_Full**(*structure*)

Convert the full structure from bracket notation to the expanded notation including root.

Parameters
structure (string)

Return type
string

RNA.**expand_Shapiro**(*coarse*)

Inserts missing 'S' identifiers in unweighted coarse grained structures as obtained from b2C().

Parameters
coarse (string)

Return type
string

RNA.**export_bppm**()

Get a pointer to the base pair probability array.

Accessing the base pair probabilities for a pair (i,j) is achieved by

Precondition
Call pf_fold_par(), pf_fold() or pf_circ_fold() first to fill the base pair probability array

Returns
A pointer to the base pair probability array

Return type
list-like(double)

See also

pf_fold, *pf_circ_fold*, RNA.idx_row_wise

RNA.**extract_record_rest_structure**(*lines*, *length*, *options*)

RNA.**fc_add_pycallback**(*vc*, *PyFunc*)

RNA.**fc_add_pydata**(*vc*, *data*, *PyFuncOrNone*)

RNA.**file_PS_aln**(*std: :string filename*, *StringVector alignment*, *StringVector identifiers*, *std: :string structure*, *unsigned int start=0*, *unsigned int end=0*, *int offset=0*, *unsigned int columns=60*) → int

Create an annotated PostScript alignment plot.

Similar to RNA.file_PS_aln() but allows the user to print a particular slice of the alignment by specifying a *start* and *end* position. The additional *offset* parameter allows for adjusting the alignment position ruler value.

SWIG Wrapper Notes

This function is available as overloaded function *file_PS_aln()* where the last four parameter may be omitted, indicating *start* = *0*, *end* = *0*, *offset* = *0*, and *columns* = *60*. See, e.g. *RNA.file_PS_aln()* in the *The RNA Python module*.

Parameters

- **filename** (string) – The output file name
- **seqs** (const char **) – The aligned sequences
- **names** (const char **) – The names of the sequences
- **structure** (string) – The consensus structure in dot-bracket notation
- **start** (unsigned int) – The start of the alignment slice (a value of 0 indicates the first position of the alignment, i.e. no slicing at 5' side)
- **end** (unsigned int) – The end of the alignment slice (a value of 0 indicates the last position of the alignment, i.e. no slicing at 3' side)
- **offset** (int) – The alignment coordinate offset for the position ruler.
- **columns** (unsigned int) – The number of columns before the alignment is wrapped as a new block (a value of 0 indicates no wrapping)

See also

RNA.file_PS_aln_slice

RNA.**file_PS_rnaplot**(**args*)

RNA.**file_PS_rnaplot_a**(**args*)

RNA.**file_RNAstrand_db_read_record**(*fp*, *options=0*)

RNA.**file_SHAPE_read**(*file_name*, *length*, *default_value*)

Read data from a given SHAPE reactivity input file.

This function parses the informations from a given file and stores the result in the preallocated string sequence and the double array values.

Parameters

- **file_name** (string) – Path to the constraints file
- **length** (int) – Length of the sequence (file entries exceeding this limit will cause an error)
- **default_value** (double) – Value for missing indices
- **sequence** (string) – Pointer to an array used for storing the sequence obtained from the SHAPE reactivity file
- **values** (list-like(double)) – Pointer to an array used for storing the values obtained from the SHAPE reactivity file

RNA.**file_commands_read**(*std: :string filename*, *unsigned int options*) → *cmd*

Extract a list of commands from a command file.

Read a list of commands specified in the input file and return them as list of abstract commands

SWIG Wrapper Notes

This function is available as global function *file_commands_read()*. See, e.g. *RNA.file_commands_read()* in the *The RNA Python module* .

Parameters

- **filename** (string) – The filename
- **options** (unsigned int) – Options to limit the type of commands read from the file

Returns
A list of abstract commands

Return type
RNA.cmd()

See also

RNA.fold_compound.commands_apply, RNA.file_commands_apply, RNA.commands_free

RNA.**file_connect_read_record**(*fp*, *remainder*, *options=0*)

RNA.**file_fasta_read**(*FILE * file*, *unsigned int options=0*) → int

Get a (fasta) data set from a file or stdin.

This function may be used to obtain complete datasets from a filehandle or stdin. A dataset is always defined to contain at least a sequence. If data starts with a fasta header, i.e. a line like

> >some header info then RNA.file_fasta_read_record() will assume that the sequence that follows the header may span over several lines. To disable this behavior and to assign a single line to the argument 'sequence' one can pass RNA.INPUT_NO_SPAN in the 'options' argument. If no fasta header is read in the beginning of a data block, a sequence must not span over multiple lines!

Unless the options RNA.INPUT_NOSKIP_COMMENTS or RNA.INPUT_NOSKIP_BLANK_LINES are passed, a sequence may be interrupted by lines starting with a comment character or empty lines.

> A sequence is regarded as completely read if it was either assumed to not span over multiple lines, a secondary structure or structure constraint follows the sequence on the next line, or a new header marks the beginning of a new sequence...

All lines following the sequence (this includes comments) that do not initiate a new dataset according to the above definition are available through the line-array 'rest'. Here one can usually find the structure constraint or other information belonging to the current dataset. Filling of 'rest' may be prevented by passing RNA.INPUT_NO_REST to the options argument.

The main purpose of this function is to be able to easily parse blocks of data in the header of a loop where all calculations for the appropriate data is done inside the loop. The loop may be then left on certain return values, e.g.:

In the example above, the while loop will be terminated when RNA.file_fasta_read_record() returns either an error, EOF, or a user initiated quit request.

As long as data is read from stdin (we are passing NULL as the file pointer), the id is printed if it is available for the current block of data. The sequence will be printed in any case and if some more lines belong to the current block of data each line will be printed as well.

Parameters

- **header** (char **) – A pointer which will be set such that it points to the header of the record

- **sequence** (char **) – A pointer which will be set such that it points to the sequence of the record
- **rest** (char ***) – A pointer which will be set such that it points to an array of lines which also belong to the record
- **file** (FILE *) – A file handle to read from (if NULL, this function reads from stdin)
- **options** (unsigned int) – Some options which may be passed to alter the behavior of the function, use 0 for no options

Returns
A flag with information about what the function actually did read

Return type
unsigned int

Note

This function will exit any program with an error message if no sequence could be read!
This function is NOT threadsafe! It uses a global variable to store information about the next data block. Do not forget to free the memory occupied by header, sequence and rest!

RNA.**file_msa_detect_format**(*std: :string filename*, *unsigned int options*) → unsigned int

Detect the format of a multiple sequence alignment file.

This function attempts to determine the format of a file that supposedly contains a multiple sequence alignment (MSA). This is useful in cases where a MSA file contains more than a single record and therefore RNA.file_msa_read() can not be applied, since it only retrieves the first. Here, one can try to guess the correct file format using this function and then loop over the file, record by record using one of the low-level record retrieval functions for the corresponding MSA file format.

SWIG Wrapper Notes

This function exists as an overloaded version where the *options* parameter may be omitted! In that case, the *options* parameter defaults to RNA.FILE_FORMAT_MSA_DEFAULT. See, e.g. *RNA.file_msa_detect_format()* in the *The RNA Python module* .

Parameters

- **filename** (string) – The name of input file that contains the alignment
- **options** (unsigned int) – Options to manipulate the behavior of this function

Returns
The MSA file format, or RNA.FILE_FORMAT_MSA_UNKNOWN

Return type
unsigned int

See also

RNA.file_msa_read, RNA.file_stockholm_read_record, RNA.file_clustal_read_record, RNA.file_fasta_read_record

Note

This function parses the entire first record within the specified file. As a result, it returns RNA.FILE_FORMAT_MSA_UNKNOWN not only if it can't detect the file's format, but also in cases where the file doesn't contain sequences!

RNA.**file_msa_read**(*std: :string filename*, *unsigned int options*) → int

Read a multiple sequence alignment from file.

This function reads the (first) multiple sequence alignment from an input file. The read alignment is split into the sequence id/name part and the actual sequence information and stored in memory as arrays of ids/names and sequences. If the alignment file format allows for additional information, such as an ID of the entire alignment or consensus structure information, this data is retrieved as well and made available. The *options* parameter allows to specify the set of alignment file formats that should be used to retrieve the data. If 0 is passed as option, the list of alignment file formats defaults to RNA.FILE_FORMAT_MSA_DEFAULT.

Currently, the list of parsable multiple sequence alignment file formats consists of:

- *ClustalW format*
- *Stockholm 1.0 format*
- *FASTA (Pearson) format*
- *MAF format*

SWIG Wrapper Notes

In the target scripting language, only the first and last argument, *filename* and *options*, are passed to the corresponding function. The other arguments, which serve as output in the C-library, are available as additional return values. This function exists as an overloaded version where the *options* parameter may be omitted! In that case, the *options* parameter defaults to RNA.FILE_FORMAT_MSA_STOCKHOLM. See, e.g. *RNA.file_msa_read()* in the *The RNA Python module* and *Parsing Alignments* in the Python examples.

Parameters

- **filename** (string) – The name of input file that contains the alignment
- **names** (char ***) – An address to the pointer where sequence identifiers should be written to
- **aln** (char ***) – An address to the pointer where aligned sequences should be written to
- **id** (char **) – An address to the pointer where the alignment ID should be written to (Maybe NULL)
- **structure** (char **) – An address to the pointer where consensus structure information should be written to (Maybe NULL)
- **options** (unsigned int) – Options to manipulate the behavior of this function

Returns

The number of sequences in the alignment, or -1 if no alignment record could be found

Return type

int

See also

RNA.file_msa_read_record, RNA.FILE_FORMAT_MSA_CLUSTAL, RNA.FILE_FORMAT_MSA_STOCKHOLM, RNA.FILE_FORMAT_MSA_FASTA, RNA.FILE_FORMAT_MSA_MAF, RNA.FILE_FORMAT_MSA_DEFAULT, RNA.FILE_FORMAT_MSA_NOCHECK

Note

After successfully reading an alignment, this function performs a validation of the data that includes uniqueness of the sequence identifiers, and equal sequence lengths. This check can be deactivated by passing RNA.FILE_FORMAT_MSA_NOCHECK in the *options* parameter.

It is the users responsibility to free any memory occupied by the output arguments *names*, *aln*, *id*, and *structure* after calling this function. The function automatically sets the latter two arguments to *NULL* in case no corresponding data could be retrieved from the input alignment.

RNA.**file_msa_read_record**(*FILE * filehandle*, *unsigned int options*) → int

Read a multiple sequence alignment from file handle.

Similar to RNA.file_msa_read(), this function reads a multiple sequence alignment from an input file handle. Since using a file handle, this function is not limited to the first alignment record, but allows for looping over all alignments within the input.

The read alignment is split into the sequence id/name part and the actual sequence information and stored in memory as arrays of ids/names and sequences. If the alignment file format allows for additional information, such as an ID of the entire alignment or consensus structure information, this data is retrieved as well and made available. The *options* parameter allows to specify the alignment file format used to retrieve the data. A single format must be specified here, see RNA.file_msa_detect_format() for helping to determine the correct MSA file format.

Currently, the list of parsable multiple sequence alignment file formats consists of:

- *ClustalW format*
- *Stockholm 1.0 format*
- *FASTA (Pearson) format*
- *MAF format*

SWIG Wrapper Notes

In the target scripting language, only the first and last argument, *fp* and *options*, are passed to the corresponding function. The other arguments, which serve as output in the C-library, are available as additional return values. This function exists as an overloaded version where the *options* parameter may be omitted! In that case, the *options* parameter defaults to RNA.FILE_FORMAT_MSA_STOCKHOLM. See, e.g. *RNA.file_msa_read_record()* in the *The RNA Python module* and *Parsing Alignments* in the Python examples.

Parameters

- **fp** (FILE *) – The file pointer the data will be retrieved from
- **names** (char ***) – An address to the pointer where sequence identifiers should be written to
- **aln** (char ***) – An address to the pointer where aligned sequences should be written to

- **id** (char **) – An address to the pointer where the alignment ID should be written to (Maybe NULL)
- **structure** (char **) – An address to the pointer where consensus structure information should be written to (Maybe NULL)
- **options** (unsigned int) – Options to manipulate the behavior of this function

Returns
The number of sequences in the alignment, or -1 if no alignment record could be found

Return type
int

See also

RNA.file_msa_read, *RNA.file_msa_detect_format*, RNA.FILE_FORMAT_MSA_CLUSTAL, RNA.FILE_FORMAT_MSA_STOCKHOLM, RNA.FILE_FORMAT_MSA_FASTA, RNA.FILE_FORMAT_MSA_MAF, RNA.FILE_FORMAT_MSA_DEFAULT, RNA.FILE_FORMAT_MSA_NOCHECK

Note

After successfully reading an alignment, this function performs a validation of the data that includes uniqueness of the sequence identifiers, and equal sequence lengths. This check can be deactivated by passing RNA.FILE_FORMAT_MSA_NOCHECK in the *options* parameter.

It is the users responsibility to free any memory occupied by the output arguments *names*, *aln*, *id*, and *structure* after calling this function. The function automatically sets the latter two arguments to *NULL* in case no corresponding data could be retrieved from the input alignment.

RNA.**file_msa_write**(*std: :string filename*, *StringVector names*, *StringVector alignment*, *std: :string id = ""*, *std: :string structure = ""*, *std: :string source = ""*, *unsigned int options=VRNA_FILE_FORMAT_MSA_STOCKHOLM|VRNA_FILE_FORMAT_MSA_APPEND*) → int

Write multiple sequence alignment file.

SWIG Wrapper Notes

In the target scripting language, this function exists as a set of overloaded versions, where the last four parameters may be omitted. If the *options* parameter is missing the options default to (RNA.FILE_FORMAT_MSA_STOCKHOLM | RNA.FILE_FORMAT_MSA_APPEND). See, e.g. *RNA.file_msa_write()* in the *The RNA Python module* .

Parameters

- **filename** (string) – The output filename
- **names** (const char **) – The array of sequence names / identifies
- **aln** (const char **) – The array of aligned sequences
- **id** (string) – An optional ID for the alignment
- **structure** (string) – An optional consensus structure
- **source** (string) – A string describing the source of the alignment
- **options** (unsigned int) – Options to manipulate the behavior of this function

Returns
Non-null upon successfully writing the alignment to file

Return type
`int`

See also

`RNA.FILE_FORMAT_MSA_STOCKHOLM`, `RNA.FILE_FORMAT_MSA_APPEND`, `RNA.FILE_FORMAT_MSA_MIS`

Note

Currently, we only support msa-formats-stockholm output

`RNA.filename_sanitize(*args)`

Sanitize a file name.

Returns a new file name where all invalid characters are substituted by a replacement character. If no replacement character is supplied, invalid characters are simply removed from the filename. File names may also never exceed a length of 255 characters. Longer file names will undergo a 'smart' truncation process, where the filenames' suffix, i.e. everything after the last dot '.', is attempted to be kept intact. Hence, only the filename part before the suffix is reduced in such a way that the total filename complies to the length restriction of 255 characters. If no suffix is present or the suffix itself already exceeds the maximum length, the filename is simply truncated from the back of the string.

For now we consider the following characters invalid:

- backslash ''
- slash '/'
- question mark '?'
- percent sign ''
- asterisk '*'
- colon ':'
- pipe symbol '|'
- double quote '"'
- triangular brackets '<' and '>'

Furthermore, the (resulting) file name must not be a reserved file name, such as:

- '.'
- '..'

Parameters

- **name** (`string`) – The input file name
- **replacement** (`string`) – The replacement character, or NULL

Returns
The sanitized file name, or NULL

Return type
`string`

Note

This function allocates a new block of memory for the sanitized string. It also may return (a) NULL if the input is pointing to NULL, or (b) an empty string if the input only consists of invalid characters which are simply removed!

RNA.**find_saddle**(*seq*, *s1*, *s2*, *width*)

Find energy of a saddle point between 2 structures (search only direct path).

Deprecated since version 2.8.0: Use RNA.path_findpath_saddle() instead!

Parameters

- **seq** (string) – RNA sequence
- **s1** (string) – A pointer to the character array where the first secondary structure in dot-bracket notation will be written to
- **s2** (string) – A pointer to the character array where the second secondary structure in dot-bracket notation will be written to
- **width** (int) – integer how many strutures are being kept during the search

Returns
the saddle energy in 10cal/mol

Return type
int

class RNA.**floatArray**(*nelements*)

Bases: object

cast()

static frompointer(*t*)

property thisown
The membership flag

RNA.**floatP_getitem**(*ary*, *index*)

RNA.**floatP_setitem**(*ary*, *index*, *value*)

RNA.**fold**(*string) -> (structure*, *mfe)fold(string) -> (structure*, *mfe*)

Compute Minimum Free Energy (MFE), and a corresponding secondary structure for an RNA sequence.

This simplified interface to RNA.fold_compound.mfe() computes the MFE and, if required, a secondary structure for an RNA sequence using default options. Memory required for dynamic programming (DP) matrices will be allocated and free'd on-the-fly. Hence, after return of this function, the recursively filled matrices are not available any more for any post-processing, e.g. suboptimal backtracking, etc.

SWIG Wrapper Notes

This function is available as function *fold()* in the global namespace. The parameter *structure* is returned along with the MFE und must not be provided. See e.g. *RNA.fold()* in the *The RNA Python module*.

Parameters

- **sequence** (string) – RNA sequence
- **structure** (string) – A pointer to the character array where the secondary structure in dot-bracket notation will be written to

Returns
the minimum free energy (MFE) in kcal/mol

Return type
`float`

See also

RNA.circfold, *RNA.fold_compound.mfe*

Note

In case you want to use the filled DP matrices for any subsequent post-processing step, or you require other conditions than specified by the default model details, use RNA.fold_compound.mfe(), and the data structure RNA.fold_compound() instead.

class RNA.**fold_compound**(*fold_compound self*, *char const * sequence*, *md md=None*, *unsigned int options*) → *fold_compound*

class RNA.**fold_compound**(*fold_compound self*, *StringVector alignment*, *md md=None*, *unsigned int options*) → *fold_compound*

class RNA.**fold_compound**(*fold_compound self*, *char const * sequence*, *char * s1*, *char * s2*, *md md=None*, *unsigned int options*) → *fold_compound*

Bases: `object`

The most basic data structure required by many functions throughout the RNAlib.

Note

Please read the documentation of this data structure carefully! Some attributes are only available for specific types this data structure can adopt.

Warning

Reading/Writing from/to attributes that are not within the scope of the current type usually result in undefined behavior!

See also

RNA.fold_compound, *RNA.fold_compound*, `RNA.fold_compound_comparative`, `RNA.fold_compound_free`, `RNA.FC_TYPE_SINGLE`, `RNA.FC_TYPE_COMPARATIVE`,

This data structure is wrapped as class *fold_compound* with several related functions attached as methods.

A new *fold_compound* can be obtained by calling one of its constructors:

- *fold_compound(seq)* - Initialize with a single sequence, or two concatenated sequences separated by an ampersand character *&* (for cofolding)
- *fold_compound(aln)* - Initialize with a sequence alignment *aln* stored as a list of sequences (with gap characters).

The resulting object has a list of attached methods which in most cases directly correspond to functions that mainly operate on the corresponding *C* data structure:

- *type()* - Get the type of the *fold_compound* (See RNA.fc_type)
- *length()* - Get the length of the sequence(s) or alignment stored within the *fold_compound*.

See, e.g. `RNA.fold_compound` in the *The RNA Python module*.

type

The type of the RNA.fold_compound().

Currently possible values are RNA.FC_TYPE_SINGLE, and RNA.FC_TYPE_COMPARATIVE

> **Warning**
>
> Do not edit this attribute, it will be automagically set by the corresponding get() methods for the RNA.fold_compound(). The value specified in this attribute dictates the set of other attributes to use within this data structure.

Type
`const vrna_fc_type_e`

length

The length of the sequence (or sequence alignment).

Type
`unsigned int`

cutpoint

The position of the (cofold) cutpoint within the provided sequence. If there is no cutpoint, this field will be set to -1.

Type
`int`

strand_number

The strand number a particular nucleotide is associated with.

Type
`list-like(unsigned int)`

strand_order

The strand order, i.e. permutation of current concatenated sequence.

Type
`list-like(unsigned int)`

strand_order_uniq

The strand order array where identical sequences have the same ID.

Type
`list-like(unsigned int)`

strand_start

The start position of a particular strand within the current concatenated sequence.

Type
`list-like(unsigned int)`

strand_end

The end (last) position of a particular strand within the current concatenated sequence.

Type
`list-like(unsigned int)`

strands

Number of interacting strands.

Type
`unsigned int`

nucleotides

Set of nucleotide sequences.

Type
`vrna_seq_t *`

alignment

Set of alignments.

Type
`vrna_msa_t *`

hc

The hard constraints data structure used for structure prediction.

Type
`vrna_hc_t *`

matrices

The MFE DP matrices.

Type
`vrna_mx_mfe_t *`

exp_matrices

The PF DP matrices.

Type
`vrna_mx_pf_t *`

params

The precomputed free energy contributions for each type of loop.

Type
param

exp_params

The precomputed free energy contributions as Boltzmann factors.

Type
exp_param

iindx

DP matrix accessor.

Type
`int *`

jindx

DP matrix accessor.

Type
`int *`

stat_cb

Recursion status callback (usually called just before, and after recursive computations in the library.

See also

RNA.recursion_status, *RNA.fold_compound.add_callback*

Type
vrna_recursion_status_f

auxdata

A pointer to auxiliary, user-defined data.

See also

RNA.fold_compound.add_auxdata, *RNA.fold_compound*

Type
void *

free_auxdata

A callback to free auxiliary user data whenever the fold_compound itself is free'd.

See also

RNA.fold_compound, RNA.auxdata_free

Type
vrna_auxdata_free_f

domains_struc

Additional structured domains.

Type
vrna_sd_t *

domains_up

Additional unstructured domains.

Type
vrna_ud_t *

aux_grammar

Additional decomposition grammar rules.

Type
vrna_gr_aux_t

maxD1

Maximum allowed base pair distance to first reference.

Type
unsigned int

maxD2

Maximum allowed base pair distance to second reference.

Type
unsigned int

reference_pt1

A pairtable of the first reference structure.

Type
list-like(int)

reference_pt2

A pairtable of the second reference structure.

Type
list-like(int)

referenceBPs1

Matrix containing number of basepairs of reference structure1 in interval [i,j].

Type
list-like(unsigned int)

referenceBPs2

Matrix containing number of basepairs of reference structure2 in interval [i,j].

Type
list-like(unsigned int)

bpdist

Matrix containing base pair distance of reference structure 1 and 2 on interval [i,j].

Type
list-like(unsigned int)

mm1

Maximum matching matrix, reference struct 1 disallowed.

Type
list-like(unsigned int)

mm2

Maximum matching matrix, reference struct 2 disallowed.

Type
list-like(unsigned int)

window_size

window size for local folding sliding window approach

Type
int

ptype_local

Pair type array (for local folding).

Type
char **

@0451233123152310633760043572333010003523672201 00

Type
union vrna_fc_s::@165075007362052204324075042127117276343234165010

E_ext_hp_loop(*i*, *j*)

E_ext_int_loop(*i*, *j*)

E_hp_loop(*i*, *j*)

E_int_loop(*i*, *j*)

E_stack(*i*, *j*)

MEA(*fold_compound self*) → char

MEA(*fold_compound self*, *double gamma*) → char *

Compute a MEA (maximum expected accuracy) structure.

The algorithm maximizes the expected accuracy

$$A(S) = \sum_{(i,j)\in S} 2\gamma p_{ij} + \sum_{i\notin S} p_i^u$$

Higher values of γ result in more base pairs of lower probability and thus higher sensitivity. Low values of γ result in structures containing only highly likely pairs (high specificity). The code of the MEA function also demonstrates the use of sparse dynamic programming scheme to reduce the time and memory complexity of folding.

Precondition
RNA.fold_compound.pf() must be executed on input parameter *fc*

SWIG Wrapper Notes

This function is attached as overloaded method *MEA(gamma = 1.)* to objects of type *fold_compound*. Note, that it returns the MEA structure and MEA value as a tuple (MEA_structure, MEA). See, e.g. *RNA.fold_compound.MEA()* in the *The RNA Python module*.

Parameters

- **gamma** (double) – The weighting factor for base pairs vs. unpaired nucleotides
- **mea** (list-like(double)) – A pointer to a variable where the MEA value will be written to

Returns
An MEA structure (or NULL on any error)

Return type
string

add_auxdata(*fold_compound self*, *PyObject * data*, *PyObject * PyFuncOrNone=Py_None*) → PyObject *

Add auxiliary data to the RNA.fold_compound().

This function allows one to bind arbitrary data to a RNA.fold_compound() which may later on be used by one of the callback functions, e.g. RNA.recursion_status(). To allow for proper cleanup of the memory occupied by this auxiliary data, the user may also provide a pointer to a cleanup function that free's the corresponding memory. This function will be called automatically when the RNA.fold_compound() is free'd with RNA.fold_compound_free().

Parameters

- **data** (void *) – A pointer to an arbitrary data structure
- **f** (RNA.auxdata_free) – A pointer to function that free's memory occupied by the arbitrary data (May be NULL)

See also

RNA.auxdata_free

Note

Before attaching the arbitrary data pointer, this function will call the RNA.auxdata_free() on any pre-existing data that is already attached.

add_callback(*fold_compound self*, *PyObject * PyFunc*) → PyObject *

Add a recursion status callback to the RNA.fold_compound().

Binding a recursion status callback function to a RNA.fold_compound() allows one to perform arbitrary operations just before, or after an actual recursive computations, e.g. MFE prediction, is performed by the RNAlib. The callback function will be provided with a pointer to its RNA.fold_compound(), and a status message. Hence, it has complete access to all variables that incluence the recursive computations.

Parameters

f (`RNA.recursion_status`) – The pointer to the recursion status callback function

See also

`RNA.recursion_status`, *RNA.fold_compound*, `RNA.STATUS_MFE_PRE`, `RNA.STATUS_MFE_POST`, `RNA.STATUS_PF_PRE`, `RNA.STATUS_PF_POST`

backtrack(*fold_compound self*, *unsigned int length*) → char

backtrack(*fold_compound self*) → char *

Backtrack an MFE (sub)structure.

This function allows one to backtrack the MFE structure for a (sub)sequence

Precondition

Requires pre-filled MFE dynamic programming matrices, i.e. one has to call RNA.fold_compound.mfe() prior to calling this function

SWIG Wrapper Notes

This function is attached as overloaded method *backtrack()* to objects of type *The Fold Compound*. The parameter *length* defaults to the total length of the RNA sequence and may be omitted. The parameter *structure* is returned along with the MFE und must not be provided. See e.g. *RNA.fold_compound.backtrack()* in the *The RNA Python module*.

Parameters

- **length** (`unsigned int`) – The length of the subsequence, starting from the 5' end
- **structure** (`string`) – A pointer to the character array where the secondary structure in dot-bracket notation will be written to. (Must have size of at least $p length + 1)

Returns

The minimum free energy (MFE) for the specified *length* in kcal/mol and a corresponding secondary structure in dot-bracket notation (stored in *structure*)

Return type

`float`

See also

RNA.fold_compound.mfe, *RNA.fold_compound.pbacktrack5*

> **Note**
>
> On error, the function returns INF / 100. and stores the empty string in *structure*.

benchmark(*fold_compound self*, *std: :string gold*, *int fuzzy=0*, *unsigned int options=8*) → *score*

bpp(*symmetric=False*)

centroid(*fold_compound self*) → char *

Get the centroid structure of the ensemble.

The centroid is the structure with the minimal average distance to all other structures $< d(S) >= \sum_{(i,j)\in S}(1-p_{ij}) + \sum_{(i,j)\notin S} p_{ij}$ Thus, the centroid is simply the structure containing all pairs with $p_ij > 0.5$ The distance of the centroid to the ensemble is written to the memory adressed by *dist*.

Parameters
dist (list-like(double)) – A pointer to the distance variable where the centroid distance will be written to

Returns
The centroid structure of the ensemble in dot-bracket notation (*NULL* on error)

Return type
string

commands_apply(*fold_compound self*, *cmd commands*, *unsigned int options*) → int

Apply a list of commands to a RNA.fold_compound().

> **SWIG Wrapper Notes**
>
> This function is attached as method *commands_apply()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.commands_apply()* in the *The RNA Python module* .

Parameters
- **commands** (*RNA.cmd()*) – The commands to apply
- **options** (unsigned int) – Options to limit the type of commands read from the file

Returns
The number of commands successfully applied

Return type
int

constraints_add(*fold_compound self*, *char const * constraint*, *unsigned int options*)

Add constraints to a RNA.fold_compound() data structure.

Use this function to add/update the hard/soft constraints The function allows for passing a string 'constraint' that can either be a filename that points to a constraints definition file or it may be a pseudo dot-bracket notation indicating hard constraints. For the latter, the user has to pass the RNA.CONSTRAINT_DB option. Also, the user has to specify, which characters are allowed to be interpreted as constraints by passing the corresponding options via the third parameter.

The following is an example for adding hard constraints given in pseudo dot-bracket notation. Here, *fc* is the RNA.fold_compound() object, *structure* is a char array with the hard constraint in dot-bracket notation, and *enforceConstraints* is a flag indicating whether or not constraints for base pairs should be enforced instead of just doing a removal of base pair that conflict with the constraint.

In constrat to the above, constraints may also be read from file:

Parameters

- **constraint** (string) – A string with either the filename of the constraint definitions or a pseudo dot-bracket notation of the hard constraint. May be NULL.
- **options** (unsigned int) – The option flags

See also

RNA.fold_compound.hc_add_from_db, *RNA.fold_compound.hc_add_up*, RNA.hc_add_up_batch, RNA.hc_add_bp_unspecific, *RNA.fold_compound.hc_add_bp*, *RNA.fold_compound.hc_init*, *RNA.fold_compound.sc_set_up*, *RNA.fold_compound.sc_set_bp*, *RNA.fold_compound.sc_add_SHAPE_deigan*, *RNA.fold_compound.sc_add_SHAPE_zarringhalam*, RNA.hc_free, RNA.sc_free, RNA.CONSTRAINT_DB, RNA.CONSTRAINT_DB_DEFAULT, RNA.CONSTRAINT_DB_PIPE, RNA.CONSTRAINT_DB_DOT, RNA.CONSTRAINT_DB_X, RNA.CONSTRAINT_DB_ANG_BRACK, RNA.CONSTRAINT_DB_RND_BRACK, RNA.CONSTRAINT_DB_INTRAMOL, RNA.CONSTRAINT_DB_INTERMOL, RNA.CONSTRAINT_DB_GQUAD

db_from_probs()

ensemble_defect(**args*)

Compute the Ensemble Defect for a given target structure.

This is a wrapper around **RNA.ensemble_defect_pt()**. Given a target structure s, compute the average dissimilarity of a randomly drawn structure from the ensemble, i.e.:

$$ED(s) = 1 - \frac{1}{n} \sum_{ij,(i,j)\in s} p_{ij} - \frac{1}{n} \sum_{i} (1 - s_i) q_i$$

with sequence length n, the probability p_{ij} of a base pair (i, j), the probability $q_i = 1 - \sum_j p_{ij}$ of nucleotide i being unpaired, and the indicator variable $s_i = 1$ if $\exists (i, j) \in s$, and $s_i = 0$ otherwise.

Precondition

The RNA.fold_compound() input parameter *fc* must contain a valid base pair probability matrix. This means that partition function and base pair probabilities must have been computed using *fc* before execution of this function!

SWIG Wrapper Notes

This function is attached as method *ensemble_defect()* to objects of type *fold_compound*. Note that the SWIG wrapper takes a structure in dot-bracket notation and converts it into a pair table using RNA.ptable_from_string(). The resulting pair table is then internally passed to RNA.ensemble_defect_pt(). To control which kind of matching brackets will be used during conversion, the optional argument *options* can be used. See also the description of RNA.ptable_from_string() for available options. (default: *RNA.BRACKETS_RND*). See, e.g. *RNA.fold_compound.ensemble_defect()* in the *The RNA Python module*.

Parameters

structure (string) – A target structure in dot-bracket notation

Returns

The ensemble defect with respect to the target structure, or -1. upon failure, e.g. pre-conditions are not met

Return type

double

See also

RNA.fold_compound.pf, RNA.pairing_probs, RNA.ensemble_defect_pt

eqm_probs_bp(*fold_compound self*, *double cutoff=1e-5*) → *ElemProbVector*

eqm_probs_hx(*fold_compound self*, *unsigned int min_size=1*, *unsigned int max_size=0*, *double cutoff=1e-5*) → *ElemProbVector*

eqm_probs_up(*fold_compound self*, *unsigned int min_u=1*, *unsigned int max_u=0*, *double cutoff=1e-5*) → *ElemProbVector*

eval_covar_structure(*structure*)

Calculate the pseudo energy derived by the covariance scores of a set of aligned sequences.

Consensus structure prediction is driven by covariance scores of base pairs in rows of the provided alignment. This function allows one to retrieve the total amount of this covariance pseudo energy scores. The RNA.fold_compound() does not need to contain any DP matrices, but requires all most basic init values as one would get from a call like this:

SWIG Wrapper Notes

This function is attached as method *eval_covar_structure()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_covar_structure()* in the *The RNA Python module* .

Parameters

structure (string) – Secondary (consensus) structure in dot-bracket notation

Returns

The covariance pseudo energy score of the input structure given the input sequence alignment in kcal/mol

Return type

float

See also

RNA.fold_compound_comparative, *RNA.fold_compound.eval_structure*

Note

Accepts RNA.fold_compound() of type RNA.FC_TYPE_COMPARATIVE only!

eval_ext_hp_loop(*i*, *j*)

eval_ext_stem(*i*, *j*)

eval_hp_loop(*fold_compound self*, *int i*, *int j*) → int

SWIG Wrapper Notes

This function is attached as method *eval_hp_loop()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_hp_loop()* in the *The RNA Python module* .

eval_int_loop(*fold_compound self*, *int i*, *int j*, *int k*, *int l*) → int

SWIG Wrapper Notes

This function is attached as method *eval_int_loop()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_int_loop()* in the *The RNA Python module* .

eval_loop_pt(**args*)

Calculate energy of a loop.

SWIG Wrapper Notes

This function is attached as method *eval_loop_pt()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_loop_pt()* in the *The RNA Python module* .

Parameters

- **i** (int) – position of covering base pair
- **pt** (const short *) – the pair table of the secondary structure

Returns
free energy of the loop in 10cal/mol

Return type
int

eval_move(*structure*, *m1*, *m2*)

Calculate energy of a move (closing or opening of a base pair).

If the parameters m1 and m2 are negative, it is deletion (opening) of a base pair, otherwise it is insertion (opening).

SWIG Wrapper Notes

This function is attached as method *eval_move()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_move()* in the *The RNA Python module* .

Parameters

- **structure** (string) – secondary structure in dot-bracket notation
- **m1** (int) – first coordinate of base pair
- **m2** (int) – second coordinate of base pair

Returns
energy change of the move in kcal/mol (INF / 100. upon any error)

Return type
float

See also

RNA.fold_compound.eval_move_pt

eval_move_pt(**args*)

Calculate energy of a move (closing or opening of a base pair).

If the parameters m1 and m2 are negative, it is deletion (opening) of a base pair, otherwise it is insertion (opening).

SWIG Wrapper Notes

This function is attached as method *eval_move_pt()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_move_pt()* in the *The RNA Python module* .

Parameters

- **pt** (list-like(int)) – the pair table of the secondary structure
- **m1** (int) – first coordinate of base pair
- **m2** (int) – second coordinate of base pair

Returns
energy change of the move in 10cal/mol

Return type
int

See also

RNA.fold_compound.eval_move

eval_structure(*structure*)

Calculate the free energy of an already folded RNA.

This function allows for energy evaluation of a given pair of structure and sequence (alignment). Model details, energy parameters, and possibly soft constraints are used as provided via the parameter 'fc'. The RNA.fold_compound() does not need to contain any DP matrices, but requires all most basic init values as one would get from a call like this:

SWIG Wrapper Notes

This function is attached as method *eval_structure()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_structure()* in the *The RNA Python module* .

Parameters
structure (string) – Secondary structure in dot-bracket notation

Returns
The free energy of the input structure given the input sequence in kcal/mol

Return type
float

See also

RNA.fold_compound.eval_structure_pt, *RNA.fold_compound.eval_structure_verbose*, *RNA.fold_compound.eval_structure_pt_verbose*, *RNA.fold_compound*, RNA.fold_compound_comparative, *RNA.fold_compound.eval_covar_structure*

Note

Accepts RNA.fold_compound() of type RNA.FC_TYPE_SINGLE and RNA.FC_TYPE_COMPARATIVE

eval_structure_pt(**args*)

Calculate the free energy of an already folded RNA.

This function allows for energy evaluation of a given sequence/structure pair where the structure is provided in pair_table format as obtained from RNA.ptable(). Model details, energy parameters, and possibly soft constraints are used as provided via the parameter 'fc'. The fold_compound does not need to contain any DP matrices, but all the most basic init values as one would get from a call like this:

SWIG Wrapper Notes

This function is attached as method *eval_structure_pt()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_structure_pt()* in the *The RNA Python module* .

Parameters

pt (const short *) – Secondary structure as pair_table

Returns

The free energy of the input structure given the input sequence in 10cal/mol

Return type

int

See also

RNA.ptable, *RNA.fold_compound.eval_structure*, *RNA.fold_compound.eval_structure_pt_verbose*

eval_structure_pt_verbose(**args*)

Calculate the free energy of an already folded RNA.

This function is a simplyfied version of RNA.eval_structure_simple_v() that uses the *default* verbosity level.

SWIG Wrapper Notes

This function is attached as method *eval_structure_pt_verbose()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_structure_pt_verbose()* in the *The RNA Python module* .

Parameters

- **pt** (const short *) – Secondary structure as pair_table
- **file** (FILE *) – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in 10cal/mol

Return type

int

See also

RNA.eval_structure_pt_v, *RNA.ptable*, *RNA.fold_compound.eval_structure_pt*, *RNA.fold_compound.eval_structure_verbose*

eval_structure_verbose(*structure*, *nullfile=None*)

Calculate the free energy of an already folded RNA and print contributions on a per-loop base.

This function is a simplyfied version of RNA.eval_structure_v() that uses the *default* verbosity level.

SWIG Wrapper Notes

This function is attached as method *eval_structure_verbose()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.eval_structure_verbose()* in the *The RNA Python module* .

Parameters

- **structure** (string) – Secondary structure in dot-bracket notation
- **file** (FILE *) – A file handle where this function should print to (may be NULL).

Returns

The free energy of the input structure given the input sequence in kcal/mol

Return type

float

See also

RNA.fold_compound.eval_structure_pt, *RNA.fold_compound.eval_structure_verbose*, *RNA.fold_compound.eval_structure_pt_verbose*

exp_E_ext_stem(i, j)

exp_E_hp_loop(i, j)

exp_E_int_loop(i, j)

exp_E_interior_loop(i, j, k, l)

property exp_matrices

property exp_params

exp_params_rescale(**args*)

Rescale Boltzmann factors for partition function computations.

This function may be used to (automatically) rescale the Boltzmann factors used in partition function computations. Since partition functions over subsequences can easily become extremely large, the RNAlib internally rescales them to avoid numerical over- and/or underflow. Therefore, a proper scaling factor s needs to be chosen that in turn is then used to normalize the corresponding partition functions $\hat{q}[i,j] = q[i,j]/s^{(j-i+1)}$.

This function provides two ways to automatically adjust the scaling factor.

1. Automatic guess
2. Automatic adjustment according to MFE

Passing *NULL* as second parameter activates the *automatic guess mode*. Here, the scaling factor is recomputed according to a mean free energy of *184.3*length* cal for random sequences. On the other hand, if the MFE for a sequence is known, it can be used to recompute a more robust scaling factor, since it represents the lowest free energy of the entire ensemble of structures, i.e. the highest Boltzmann factor. To activate this second mode of *automatic adjustment according to MFE*, a pointer to the MFE value needs to be passed as second argument. This value is then taken to compute the scaling factor as $s = exp((sfact * MFE)/kT/length)$, where sfact is an additional scaling weight located in the RNA.md() data structure of *exp_params* in *fc*.

Note

This recomputation only takes place if the *pf_scale* attribute of the *exp_params* data structure contained in *fc* has a value below *1.0*.

The computed scaling factor s will be stored as *pf_scale* attribute of the *exp_params* data structure in *fc*.

SWIG Wrapper Notes

This function is attached to RNA.fc() objects as overloaded *exp_params_rescale()* method. When no parameter is passed to this method, the resulting action is the same as passing *NULL* as second parameter to RNA.fold_compound.exp_params_rescale(), i.e. default scaling of the partition function. Passing an energy in kcal/mol, e.g. as retrieved by a previous call to the *mfe()* method, instructs all subsequent calls to scale the partition function accordingly. See, e.g. *RNA.fold_compound.exp_params_rescale()* in the *The RNA Python module*.

Parameters

mfe (list-like(double)) – A pointer to the MFE (in kcal/mol) or NULL

See also

RNA.fold_compound.exp_params_subst, *RNA.md*, *RNA.exp_param*, *RNA.fold_compound*

exp_params_reset(*md=None*)

Reset Boltzmann factors for partition function computations within a RNA.fold_compound() according to provided, or default model details.

This function allows one to rescale Boltzmann factors for subsequent partition function computations according to a set of model details, e.g. temperature values. To do so, the caller provides either a pointer to a set of model details to be used for rescaling, or NULL if global default setting should be used.

SWIG Wrapper Notes

This function is attached to RNA.fc() objects as overloaded *exp_params_reset()* method.

When no parameter is passed to this method, the resulting action is the same as passing *NULL* as second parameter to RNA.fold_compound.exp_params_reset(), i.e. global default model settings are used. Passing an object of type RNA.md() resets the fold compound according to the specifications stored within the RNA.md() object. See, e.g. *RNA.fold_compound.exp_params_reset()* in the *The RNA Python module*.

Parameters

md (RNA.md() *) – A pointer to the new model details (or NULL for reset to defaults)

See also

RNA.fold_compound.params_reset, *RNA.fold_compound.exp_params_subst*, *RNA.fold_compound.exp_params_rescale*

exp_params_subst(*par*)

Update the energy parameters for subsequent partition function computations.

This function can be used to properly assign new energy parameters for partition function computations to a RNA.fold_compound(). For this purpose, the data of the provided pointer *params* will be copied into *fc* and a recomputation of the partition function scaling factor is issued, if the *pf_scale* attribute of *params* is less than *1.0*.

Passing NULL as second argument leads to a reset of the energy parameters within fc to their default values

SWIG Wrapper Notes

This function is attached to RNA.fc() objects as overloaded *exp_params_subst()* method.

When no parameter is passed, the resulting action is the same as passing *NULL* as second parameter to RNA.fold_compound.exp_params_subst(), i.e. resetting the parameters to the global defaults. See, e.g. *RNA.fold_compound.exp_params_subst()* in the *The RNA Python module*.

Parameters

params (RNA.exp_param() *) – A pointer to the new energy parameters

See also

RNA.fold_compound.exp_params_reset, *RNA.fold_compound.exp_params_rescale*, *RNA.exp_param*, *RNA.md*, RNA.exp_params

file_commands_apply(*fold_compound self*, *std: :string filename*, *unsigned int options*) → int

property hc

hc_add_bp(*fold_compound self*, *unsigned int i*, *unsigned int j*, *unsigned int option=VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*)

Favorize/Enforce a certain base pair (i,j).

Parameters

- **i** (unsigned int) – The 5' located nucleotide position of the base pair (1-based)
- **j** (unsigned int) – The 3' located nucleotide position of the base pair (1-based)
- **option** (unsigned char) – The options flag indicating how/where to store the hard constraints

See also

RNA.fold_compound.hc_add_bp_nonspecific, *RNA.fold_compound.hc_add_up*, *RNA.fold_compound.hc_init*, RNA.CONSTRAINT_CONTEXT_EXT_LOOP, RNA.CONSTRAINT_CONTEXT_HP_LOOP, RNA.CONSTRAINT_CONTEXT_INT_LOOP, RNA.CONSTRAINT_CONTEXT_INT_LOOP_ENC, RNA.CONSTRAINT_CONTEXT_MB_LOOP, RNA.CONSTRAINT_CONTEXT_MB_LOOP_ENC, RNA.CONSTRAINT_CONTEXT_ENFORCE, RNA.CONSTRAINT_CONTEXT_ALL_LOOPS

hc_add_bp_nonspecific(*fold_compound self*, *unsigned int i*, *int d*, *unsigned int option=VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*)

Enforce a nucleotide to be paired (upstream/downstream).

Parameters

- **i** (unsigned int) – The position that needs to stay unpaired (1-based)
- **d** (int) – The direction of base pairing ($d < 0$: pairs upstream, $d > 0$: pairs downstream, $d == 0$: no direction)
- **option** (unsigned char) – The options flag indicating in which loop type context the pairs may appear

See also

RNA.fold_compound.hc_add_bp, *RNA.fold_compound.hc_add_up*, *RNA.fold_compound.hc_init*, RNA.CONSTRAINT_CONTEXT_EXT_LOOP, RNA.CONSTRAINT_CONTEXT_HP_LOOP, RNA.CONSTRAINT_CONTEXT_INT_LOOP, RNA.CONSTRAINT_CONTEXT_INT_LOOP_ENC, RNA.CONSTRAINT_CONTEXT_MB_LOOP, RNA.CONSTRAINT_CONTEXT_MB_LOOP_ENC, RNA.CONSTRAINT_CONTEXT_ALL_LOOPS

hc_add_bp_strand(**args*, ***kwargs*)

hc_add_from_db(*fold_compound self*, *char const * constraint*, *unsigned int options*) → int

Add hard constraints from pseudo dot-bracket notation.

This function allows one to apply hard constraints from a pseudo dot-bracket notation. The *options* parameter controls, which characters are recognized by the parser. Use the RNA.CONSTRAINT_DB_DEFAULT convenience macro, if you want to allow all known characters

SWIG Wrapper Notes

This function is attached as method *hc_add_from_db()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.hc_add_from_db()* in the *The RNA Python module* .

Parameters

- **constraint** (string) – A pseudo dot-bracket notation of the hard constraint.
- **options** (unsigned int) – The option flags

See also

RNA.CONSTRAINT_DB_PIPE, RNA.CONSTRAINT_DB_DOT, RNA.CONSTRAINT_DB_X, RNA.CONSTRAINT_DB_ANG_BRACK, RNA.CONSTRAINT_DB_RND_BRACK, RNA.CONSTRAINT_DB_INTRAMOL, RNA.CONSTRAINT_DB_INTERMOL, RNA.CONSTRAINT_DB_GQUAD

hc_add_up(*fold_compound self*, *unsigned int i*, *unsigned int option=VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS*)

Make a certain nucleotide unpaired.

Parameters

- **i** (unsigned int) – The position that needs to stay unpaired (1-based)
- **option** (unsigned char) – The options flag indicating how/where to store the hard constraints

See also

RNA.fold_compound.hc_add_bp, *RNA.fold_compound.hc_add_bp_nonspecific*, *RNA.fold_compound.hc_init*, RNA.CONSTRAINT_CONTEXT_EXT_LOOP, RNA.CONSTRAINT_CONTEXT_HP_LOOP, RNA.CONSTRAINT_CONTEXT_INT_LOOP, RNA.CONSTRAINT_CONTEXT_MB_LOOP, RNA.CONSTRAINT_CONTEXT_ALL_LOOPS

hc_add_up_strand(**args*, ***kwargs*)

hc_init()

Initialize/Reset hard constraints to default values.

This function resets the hard constraints to their default values, i.e. all positions may be unpaired in all contexts, and base pairs are allowed in all contexts, if they resemble canonical pairs. Previously set hard constraints will be removed before initialization.

SWIG Wrapper Notes

This function is attached as method *hc_init()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.hc_init()* in the *The RNA Python module* .

See also

RNA.fold_compound.hc_add_bp, *RNA.fold_compound.hc_add_bp_nonspecific*, *RNA.fold_compound.hc_add_up*

heat_capacity(*fold_compound self*, *float T_min=0.*, *float T_max=100.*, *float T_increment=1.*, *unsigned int mpoints=2*) → *HeatCapacityVector*

Compute the specific heat for an RNA.

This function computes an RNAs specific heat in a given temperature range from the partition function by numeric differentiation. The result is returned as a list of pairs of temperature in C and specific heat in Kcal/(Mol*K).

Users can specify the temperature range for the computation from *T_min* to *T_max*, as well as the increment step size *T_increment*. The latter also determines how many times the partition function is computed. Finally, the parameter *mpoints* determines how smooth the curve should be. The algorithm itself fits a parabola to $2 \cdot mpoints + 1$ data points to calculate 2nd derivatives. Increasing this parameter produces a smoother curve.

SWIG Wrapper Notes

This function is attached as overloaded method *heat_capacity()* to objects of type *fold_compound*. If the optional function arguments *T_min*, *T_max*, *T_increment*, and *mpoints* are omitted, they default to 0.0, 100.0, 1.0 and 2, respectively. See, e.g. *RNA.fold_compound.heat_capacity()* in the *The RNA Python module*.

Parameters

- **T_min** (float) – Lowest temperature in C
- **T_max** (float) – Highest temperature in C
- **T_increment** (float) – Stepsize for temperature incrementation in C (a reasonable choice might be 1C)

- **mpoints** (unsigned int) – The number of interpolation points to calculate 2nd derivative (a reasonable choice might be 2, min: 1, max: 100)

Returns
A list of pairs of temperatures and corresponding heat capacity or *NULL* upon any failure. The last entry of the list is indicated by a **temperature** field set to a value smaller than *T_min*

Return type
RNA.heat_capacity() *

See also

RNA.fold_compound.heat_capacity_cb, *RNA.heat_capacity*, *RNA.heat_capacity*

heat_capacity_cb(*fold_compound self*, *float T_min*, *float T_max*, *float T_increment*, *unsigned int mpoints*, *PyObject * PyFunc*, *PyObject * data=Py_None*) → PyObject *

Compute the specific heat for an RNA (callback variant).

Similar to RNA.fold_compound.heat_capacity(), this function computes an RNAs specific heat in a given temperature range from the partition function by numeric differentiation. Instead of returning a list of temperature/specific heat pairs, however, this function returns the individual results through a callback mechanism. The provided function will be called for each result and passed the corresponding temperature and specific heat values along with the arbitrary data as provided through the *data* pointer argument.

Users can specify the temperature range for the computation from *T_min* to *T_max*, as well as the increment step size *T_increment*. The latter also determines how many times the partition function is computed. Finally, the parameter *mpoints* determines how smooth the curve should be. The algorithm itself fits a parabola to $2 \cdot mpoints + 1$ data points to calculate 2nd derivatives. Increasing this parameter produces a smoother curve.

SWIG Wrapper Notes

This function is attached as method *heat_capacity_cb()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.heat_capacity_cb()* in the *The RNA Python module*.

Parameters

- **T_min** (float) – Lowest temperature in C
- **T_max** (float) – Highest temperature in C
- **T_increment** (float) – Stepsize for temperature incrementation in C (a reasonable choice might be 1C)
- **mpoints** (unsigned int) – The number of interpolation points to calculate 2nd derivative (a reasonable choice might be 2, min: 1, max: 100)
- **cb** (*RNA.heat_capacity*) – The user-defined callback function that receives the individual results
- **data** (void *) – An arbitrary data structure that will be passed to the callback in conjunction with the results

Returns
Returns 0 upon failure, and non-zero otherwise

Return type
int

See also

RNA.fold_compound.heat_capacity, *RNA.heat_capacity*

property iindx

property jindx

property length

property matrices

maxmimum_matching()

mean_bp_distance()

Get the mean base pair distance in the thermodynamic ensemble.

$$< d >= \sum_{a,b} p_a p_b d(S_a, S_b)$$

this can be computed from the pair probs p_{ij} as

$$< d >= \sum_{ij} p_{ij}(1 - p_{ij})$$

SWIG Wrapper Notes

This function is attached as method *mean_bp_distance()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.mean_bp_distance()* in the *The RNA Python module*.

Returns
The mean pair distance of the structure ensemble

Return type
double

mfe()

Compute minimum free energy and an appropriate secondary structure of an RNA sequence, or RNA sequence alignment.

Depending on the type of the provided RNA.fold_compound(), this function predicts the MFE for a single sequence (or connected component of multiple sequences), or an averaged MFE for a sequence alignment. If backtracking is activated, it also constructs the corresponding secondary structure, or consensus structure. Therefore, the second parameter, *structure*, has to point to an allocated block of memory with a size of at least $\text{strlen}(\text{sequence}) + 1$ to store the backtracked MFE structure. (For consensus structures, this is the length of the alignment + 1. If *NULL* is passed, no backtracking will be performed.

SWIG Wrapper Notes

This function is attached as method *Minimum Free Energy (MFE) Algorithms* to objects of type *The Fold Compound*. The parameter *structure* is returned along with the MFE und must not be provided. See e.g. *RNA.fold_compound.mfe()* in the *The RNA Python module*.

Parameters
structure (string) – A pointer to the character array where the secondary structure in dot-bracket notation will be written to (Maybe NULL)

Returns
the minimum free energy (MFE) in kcal/mol

Return type
`float`

See also

RNA.fold_compound, *RNA.fold_compound*, *RNA.fold*, *RNA.circfold*, `RNA.fold_compound_comparative`, *RNA.alifold*, *RNA.circalifold*

Note

This function is polymorphic. It accepts RNA.fold_compound() of type RNA.FC_TYPE_SINGLE, and RNA.FC_TYPE_COMPARATIVE.

mfe_dimer(*fold_compound self*) → char *

Compute the minimum free energy of two interacting RNA molecules.

The code is analog to the RNA.fold_compound.mfe() function.

Deprecated since version 2.8.0: This function is obsolete since RNA.fold_compound.mfe() can handle complexes multiple sequences since v2.5.0. Use RNA.fold_compound.mfe() for connected component MFE instead and compute MFEs of unconnected states separately.

SWIG Wrapper Notes

This function is attached as method *mfe_dimer()* to objects of type *The Fold Compound*. The parameter *structure* is returned along with the MFE und must not be provided. See e.g. *RNA.fold_compound.mfe_dimer()* in the *The RNA Python module*.

Parameters
structure (`string`) – Will hold the barcket dot structure of the dimer molecule

Returns
minimum free energy of the structure

Return type
`float`

See also

RNA.fold_compound.mfe

mfe_window(*nullfile=None*)

Local MFE prediction using a sliding window approach.

Computes minimum free energy structures using a sliding window approach, where base pairs may not span outside the window. In contrast to RNA.fold_compound.mfe(), where a maximum base pair span may be set using the RNA.md().max_bp_span attribute and one globally optimal structure is predicted, this function uses a sliding window to retrieve all locally optimal structures within each window. The size of the sliding window is set in the RNA.md().window_size attribute, prior to the retrieval of the RNA.fold_compound() using RNA.fold_compound() with option RNA.OPTION_WINDOW

The predicted structures are written on-the-fly, either to stdout, if a NULL pointer is passed as file parameter, or to the corresponding filehandle.

SWIG Wrapper Notes

This function is attached as overloaded method *Local (sliding window) MFE Prediction* to objects of type *The Fold Compound*. The parameter *FILE* has default value of *NULL* and can be omitted. See e.g. *RNA.fold_compound.mfe_window()* in the *The RNA Python module*.

Parameters

file (FILE *) – The output file handle where predictions are written to (maybe NULL)

See also

RNA.fold_compound, *RNA.fold_compound.mfe_window_zscore*, *RNA.fold_compound.mfe*, *RNA.Lfold*, *RNA.Lfoldz*, RNA.OPTION_WINDOW, *RNA.md*, *RNA.md*

mfe_window_cb(*fold_compound self*, *PyObject * PyFunc*, *PyObject * data=Py_None*) → float

SWIG Wrapper Notes

This function is attached as overloaded method *mfe_window_cb()* to objects of type *The Fold Compound*. The parameter *data* has default value of *NULL* and can be omitted. See e.g. *RNA.fold_compound.mfe_window_cb()* in the *The RNA Python module*.

mfe_window_zscore(*min_z*, *nullfile=None*)

mfe_window_zscore_cb(*fold_compound self*, *double min_z*, *PyObject * PyFunc*, *PyObject * data=Py_None*) → float

move_neighbor_diff(*self*, *pt*, *move*, *options=4 | 8*) → *varArrayMove*

move_neighbor_diff(*fold_compound self*, *varArrayShort pt*, *move move*, *PyObject * PyFunc*, *PyObject * data=Py_None*, *unsigned int options=(4|8)*) → int

Apply a move to a secondary structure and indicate which neighbors have changed consequentially.

Similar to RNA.move_neighbor_diff_cb(), this function applies a move to a secondary structure and reports back the neighbors of the current structure become affected by this move. Instead of executing a callback for each of the affected neighbors, this function compiles two lists of neighbor moves, one that is returned and consists of all moves that are novel or may have changed in energy, and a second, *invalid_moves*, that consists of all the neighbor moves that become invalid, respectively.

Parameters

- **ptable** (list-like(int)) – The current structure as pair table
- **move** (*RNA.move()*) – The move to apply
- **invalid_moves** (RNA.move() **) – The address of a move list where the function stores those moves that become invalid
- **options** (unsigned int) – Options to modify the behavior of this function, .e.g available move set

Returns

A list of moves that might have changed in energy or are novel compared to the structure before application of the move

Return type

RNA.move() *

neighbors(*fold_compound self*, *varArrayShort pt*, *unsigned int options=(4|8)*) → *varArrayMove*

Generate neighbors of a secondary structure.

This function allows one to generate all structural neighbors (according to a particular move set) of an RNA secondary structure. The neighborhood is then returned as a list of transitions / moves required to transform the current structure into the actual neighbor.

SWIG Wrapper Notes

This function is attached as an overloaded method *neighbors()* to objects of type *fold_compound*. The optional parameter *options* defaults to RNA.MOVESET_DEFAULT if it is omitted. See, e.g. *RNA.fold_compound.neighbors()* in the *The RNA Python module*.

Parameters

- **pt** (const short *) – The pair table representation of the structure
- **options** (unsigned int) – Options to modify the behavior of this function, e.g. available move set

Returns

Neighbors as a list of moves / transitions (the last element in the list has both of its fields set to 0)

Return type

RNA.move() *

See also

RNA.neighbors_successive, RNA.move_apply, RNA.MOVESET_INSERTION, RNA.MOVESET_DELETION, RNA.MOVESET_SHIFT, RNA.MOVESET_DEFAULT

property params

params_reset(*md=None*)

Reset free energy parameters within a RNA.fold_compound() according to provided, or default model details.

This function allows one to rescale free energy parameters for subsequent structure prediction or evaluation according to a set of model details, e.g. temperature values. To do so, the caller provides either a pointer to a set of model details to be used for rescaling, or NULL if global default setting should be used.

SWIG Wrapper Notes

This function is attached to RNA.fc() objects as overloaded *params_reset()* method.

When no parameter is passed to this method, the resulting action is the same as passing *NULL* as second parameter to RNA.fold_compound.params_reset(), i.e. global default model settings are used. Passing an object of type RNA.md() resets the fold compound according to the specifications stored within the RNA.md() object. See, e.g. *RNA.fold_compound.params_reset()* in the *The RNA Python module*.

Parameters

md (RNA.md() *) – A pointer to the new model details (or NULL for reset to defaults)

See also

RNA.fold_compound.exp_params_reset, RNA.params_subs

params_subst(*par=None*)

Update/Reset energy parameters data structure within a RNA.fold_compound().

Passing NULL as second argument leads to a reset of the energy parameters within fc to their default values. Otherwise, the energy parameters provided will be copied over into fc.

SWIG Wrapper Notes

This function is attached to RNA.fc() objects as overloaded *params_subst()* method.

When no parameter is passed, the resulting action is the same as passing *NULL* as second parameter to RNA.fold_compound.params_subst(), i.e. resetting the parameters to the global defaults. See, e.g. *RNA.fold_compound.params_subst()* in the *The RNA Python module*.

Parameters

par (RNA.param() *) – The energy parameters used to substitute those within fc (Maybe NULL)

See also

RNA.fold_compound.params_reset, *RNA.param*, *RNA.md*, RNA.params

path(*fold_compound self*, *IntVector pt*, *unsigned int steps*, *unsigned int options*) → *MoveVector*

path(*fold_compound self*, *varArrayShort pt*, *unsigned int steps*, *unsigned int options*) → *MoveVector*

Compute a path, store the final structure, and return a list of transition moves from the start to the final structure.

This function computes, given a start structure in pair table format, a transition path, updates the pair table to the final structure of the path. Finally, if not requested otherwise by using the RNA.PATH_NO_TRANSITION_OUTPUT flag in the *options* field, this function returns a list of individual transitions that lead from the start to the final structure if requested.

The currently available transition paths are

- Steepest Descent / Gradient walk (flag: RNA.PATH_STEEPEST_DESCENT)
- Random walk (flag: RNA.PATH_RANDOM)

The type of transitions must be set through the *options* parameter

SWIG Wrapper Notes

This function is attached as an overloaded method *path()* to objects of type *fold_compound*. The optional parameter *options* defaults to RNA.PATH_DEFAULT if it is omitted. See, e.g. *RNA.fold_compound.path()* in the *The RNA Python module*.

Parameters

- **pt** (list-like(int)) – The pair table containing the start structure. Used to update to the final structure after execution of this function
- **options** (unsigned int) – Options to modify the behavior of this function

Returns
A list of transition moves (default), or NULL (if options & RNA.PATH_NO_TRANSITION_OUTPUT)

Return type
RNA.move() *

See also

RNA.fold_compound.path_gradient, *RNA.fold_compound.path_random*, *RNA.ptable*, RNA.ptable_copy, *RNA.fold_compound*, RNA.PATH_RANDOM, RNA.MOVESET_DEFAULT, RNA.MOVESET_SHIFT, RNA.PATH_NO_TRANSITION_OUTPUT

Note

Since the result is written to the input structure you may want to use RNA.ptable_copy() before calling this function to keep the initial structure

path_direct(*fold_compound self*, *std: :string s1*, *std: :string s2*, *int maxE=INT_MAX-1*, *path_options options=None*) → *PathVector*

Determine an optimal direct (re-)folding path between two secondary structures.

This function is similar to RNA.path_direct(), but allows to specify an *upper-bound* for the saddle point energy. The underlying algorithms will stop determining an (optimal) (re-)folding path, if none can be found that has a saddle point below the specified upper-bound threshold *maxE*.

SWIG Wrapper Notes

This function is attached as an overloaded method *path_direct()* to objects of type *fold_compound*. The optional parameter *maxE* defaults to #INT_MAX - 1 if it is omitted, while the optional parameter *options* defaults to *NULL*. In case the function did not find a path with $E_{saddle} < E_{max}$ it returns an empty list. See, e.g. *RNA.fold_compound.path_direct()* in the *The RNA Python module*.

Parameters

- **s1** (string) – The start structure in dot-bracket notation
- **s2** (string) – The target structure in dot-bracket notation
- **maxE** (int) – Upper bound for the saddle point along the (re-)folding path
- **options** (*RNA.path_options()*) – An options data structure that specifies the path heuristic and corresponding settings (maybe *NULL*)

Returns
An optimal (re-)folding path between the two input structures

Return type
RNA.path() *

Warning

The argument *maxE* enables one to specify an upper bound, or maximum free energy for the saddle point between the two input structures. If no path with $E_{saddle} < E_{max}$ is found, the function simply returns *NULL*

See also

RNA.fold_compound.path_direct, *RNA.path_options_findpath*, RNA.path_options_free, RNA.path_free

path_findpath(*fold_compound self*, *std: :string s1*, *std: :string s2*, *int width=1*, *int maxE=INT_MAX-1*) → *PathVector*

path_findpath_saddle(*fold_compound self*, *std: :string s1*, *std: :string s2*, *int width=1*, *int maxE=INT_MAX*) → PyObject *

Find energy of a saddle point between 2 structures (search only direct path).

This function uses an inplementation of the *findpath* algorithm [Flamm *et al.*, 2001] for near- optimal direct refolding path prediction.

Model details, and energy parameters are used as provided via the parameter 'fc'. The RNA.fold_compound() does not require memory for any DP matrices, but requires all most basic init values as one would get from a call like this:

SWIG Wrapper Notes

This function is attached as an overloaded method *path_findpath_saddle()* to objects of type *fold_compound*. The optional parameter *width* defaults to 1 if it is omitted, while the optional parameter *maxE* defaults to INF. In case the function did not find a path with $E_{saddle} < E_{max}$ the function returns a *NULL* object, i.e. *undef* for Perl and *None* for Python. See, e.g. *RNA.fold_compound.path_findpath_saddle()* in the *The RNA Python module*.

Parameters

- **s1** (string) – The start structure in dot-bracket notation
- **s2** (string) – The target structure in dot-bracket notation
- **width** (int) – A number specifying how many strutures are being kept at each step during the search
- **maxE** (int) – An upper bound for the saddle point energy in 10cal/mol

Returns
The saddle energy in 10cal/mol

Return type
int

Warning

The argument *maxE* (E_{max}) enables one to specify an upper bound, or maximum free energy for the saddle point between the two input structures. If no path with $E_{saddle} < E_{max}$ is found, the function simply returns *maxE*

See also

RNA.path_findpath_saddle, *RNA.fold_compound*, *RNA.fold_compound*, RNA.path_findpath

path_gradient(*fold_compound self*, *IntVector pt*, *unsigned int options*) → *MoveVector*

path_gradient(*fold_compound self*, *varArrayShort pt*, *unsigned int options*) → *MoveVector*

Compute a steepest descent / gradient path, store the final structure, and return a list of transition moves from the start to the final structure.

This function computes, given a start structure in pair table format, a steepest descent path, updates the pair table to the final structure of the path. Finally, if not requested otherwise by using the RNA.PATH_NO_TRANSITION_OUTPUT flag in the *options* field, this function returns a list of individual transitions that lead from the start to the final structure if requested.

SWIG Wrapper Notes

This function is attached as an overloaded method *path_gradient()* to objects of type *fold_compound*. The optional parameter *options* defaults to RNA.PATH_DEFAULT if it is omitted. See, e.g. *RNA.fold_compound.path_gradient()* in the *The RNA Python module*.

Parameters

- **pt** (list-like(int)) – The pair table containing the start structure. Used to update to the final structure after execution of this function
- **options** (unsigned int) – Options to modify the behavior of this function

Returns

A list of transition moves (default), or NULL (if options & RNA.PATH_NO_TRANSITION_OUTPUT)

Return type

RNA.move() *

See also

RNA.fold_compound.path_random, *RNA.fold_compound.path*, *RNA.ptable*, RNA.ptable_copy, *RNA.fold_compound*, RNA.MOVESET_SHIFT, RNA.PATH_NO_TRANSITION_OUTPUT

Note

Since the result is written to the input structure you may want to use RNA.ptable_copy() before calling this function to keep the initial structure

path_random(*fold_compound self*, *IntVector pt*, *unsigned int steps*, *unsigned int options*) → *MoveVector*

path_random(*fold_compound self*, *varArrayShort pt*, *unsigned int steps*, *unsigned int options*) → *MoveVector*

Generate a random walk / path of a given length, store the final structure, and return a list of transition moves from the start to the final structure.

This function generates, given a start structure in pair table format, a random walk / path, updates the pair table to the final structure of the path. Finally, if not requested otherwise by using the RNA.PATH_NO_TRANSITION_OUTPUT flag in the *options* field, this function returns a list of individual transitions that lead from the start to the final structure if requested.

SWIG Wrapper Notes

This function is attached as an overloaded method *path_gradient()* to objects of type *fold_compound*. The optional parameter *options* defaults to RNA.PATH_DEFAULT if it is omitted.

See, e.g. *RNA.fold_compound.path_random()* in the *The RNA Python module*.

Parameters

- **pt** (list-like(int)) – The pair table containing the start structure. Used to update to the final structure after execution of this function
- **steps** (unsigned int) – The length of the path, i.e. the total number of transitions / moves
- **options** (unsigned int) – Options to modify the behavior of this function

Returns

A list of transition moves (default), or NULL (if options & RNA.PATH_NO_TRANSITION_OUTPUT)

Return type

RNA.move() *

See also

RNA.fold_compound.path_gradient, *RNA.fold_compound.path*, *RNA.ptable*, RNA.ptable_copy, *RNA.fold_compound*, RNA.MOVESET_SHIFT, RNA.PATH_NO_TRANSITION_OUTPUT

Note

Since the result is written to the input structure you may want to use RNA.ptable_copy() before calling this function to keep the initial structure

pbacktrack(*fold_compound self*) → char

pbacktrack(*fold_compound self*, *unsigned int num_samples*, *unsigned int options*) → *StringVector*

pbacktrack(*fold_compound self*, *unsigned int num_samples*, *pbacktrack_mem nr_memory*, *unsigned int options*) → *StringVector*

pbacktrack(*self*, *num_samples*, *PyFunc*, *data=Py_None*, *options=0*) → unsigned int

Parameters

- **num_samples** (unsigned int)
- **PyFunc** (PyObject *)
- **data** (PyObject *)
- **options** (unsigned int)
- **pbacktrack(self**
- **num_samples**
- **PyFunc**
- **data**
- **nr_memory** (vrna_pbacktrack_mem_t *)
- **int** (*options=0) -> unsigned*)
- **num_samples**
- **PyFunc**

- **data**
- **nr_memory**
- **options**

Sample a secondary structure from the Boltzmann ensemble according its probability.

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a secondary structure.

The structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Precondition

Unique multiloop decomposition has to be active upon creation of *fc* with RNA.fold_compound() or similar. This can be done easily by passing RNA.fold_compound() a model details parameter with RNA.md().uniq_ML = 1. RNA.fold_compound.pf() has to be called first to fill the partition function matrices

SWIG Wrapper Notes

This function is attached as overloaded method *pbacktrack()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.pbacktrack()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Returns

A sampled secondary structure in dot-bracket notation (or NULL on error)

Return type

string

See also

RNA.fold_compound.pbacktrack5, RNA.pbacktrack_num, RNA.pbacktrack_cb

Note

This function is polymorphic. It accepts RNA.fold_compound() of type RNA.FC_TYPE_SINGLE, and RNA.FC_TYPE_COMPARATIVE.

pbacktrack5(*fold_compound self*, *unsigned int length*) → char

pbacktrack5(*fold_compound self*, *unsigned int num_samples*, *unsigned int length*, *unsigned int options*) → *StringVector*

pbacktrack5(*fold_compound self*, *unsigned int num_samples*, *unsigned int length*, *pbacktrack_mem nr_memory*, *unsigned int options*) → *StringVector*

pbacktrack5(*fold_compound self*, *unsigned int num_samples*, *unsigned int length*, *PyObject * PyFunc*, *PyObject * data=Py_None*, *unsigned int options=0*) → unsigned int

pbacktrack5(*fold_compound self*, *unsigned int num_samples*, *unsigned int length*, *PyObject * PyFunc*, *PyObject * data*, *pbacktrack_mem nr_memory*, *unsigned int options=0*) → unsigned int

Sample a secondary structure of a subsequence from the Boltzmann ensemble according its probability.

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a secondary structure. The parameter *length* specifies the length of the substructure starting from the 5' end.

The structure s with free energy $E(s)$ is picked from the Boltzmann distributed ensemble according to its probability

$$p(s) = \frac{exp(-E(s)/kT)}{Z}$$

with partition function $Z = \sum_s exp(-E(s)/kT)$, Boltzmann constant k and thermodynamic temperature T.

Precondition

Unique multiloop decomposition has to be active upon creation of *fc* with RNA.fold_compound() or similar. This can be done easily by passing RNA.fold_compound() a model details parameter with RNA.md().uniq_ML = 1. RNA.fold_compound.pf() has to be called first to fill the partition function matrices

SWIG Wrapper Notes

This function is attached as overloaded method *pbacktrack5()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.pbacktrack5()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Parameters

length (unsigned int) – The length of the subsequence to consider (starting with 5' end)

Returns

A sampled secondary structure in dot-bracket notation (or NULL on error)

Return type

string

See also

RNA.pbacktrack5_num, RNA.pbacktrack5_cb, *RNA.fold_compound.pbacktrack*

Note

This function is polymorphic. It accepts RNA.fold_compound() of type RNA.FC_TYPE_SINGLE, and RNA.FC_TYPE_COMPARATIVE.

pbacktrack_sub(*fold_compound self*, *unsigned int start*, *unsigned int end*) → char

pbacktrack_sub(*fold_compound self*, *unsigned int num_samples*, *unsigned int start*, *unsigned int end*, *unsigned int options*) → *StringVector*

pbacktrack_sub(*fold_compound self*, *unsigned int num_samples*, *unsigned int start*, *unsigned int end*, *pbacktrack_mem nr_memory*, *unsigned int options*) → *StringVector*

pbacktrack_sub(*fold_compound self*, *unsigned int num_samples*, *unsigned int start*, *unsigned int end*, *PyObject * PyFunc*, *PyObject * data=Py_None*, *unsigned int options=0*) → unsigned int

pbacktrack_sub(*fold_compound self*, *unsigned int num_samples*, *unsigned int start*, *unsigned int end*, *PyObject * PyFunc*, *PyObject * data*, *pbacktrack_mem nr_memory*, *unsigned int options=0*) → unsigned int

Sample a secondary structure of a subsequence from the Boltzmann ensemble according its probability.

Perform a probabilistic (stochastic) backtracing in the partition function DP arrays to obtain a secondary structure. The parameters *start* and *end* specify the interval $[start : end]$ of the subsequence with $1 \leq start < end \leq n$ for sequence length n, the structure $s_{start,end}$ should be drawn from.

The resulting substructure $s_{start,end}$ with free energy $E(s_{start,end})$ is picked from the Boltzmann distributed sub ensemble of all structures within the interval $[start : end]$ according to its probability

$$p(s_{start,end}) = \frac{exp(-E(s_{start,end})/kT)}{Z_{start,end}}$$

with partition function $Z_{start,end} = \sum_{s_{start,end}} exp(-E(s_{start,end})/kT)$, Boltzmann constant k and thermodynamic temperature T.

Precondition

Unique multiloop decomposition has to be active upon creation of *fc* with RNA.fold_compound() or similar. This can be done easily by passing RNA.fold_compound() a model details parameter with RNA.md().uniq_ML = 1. RNA.fold_compound.pf() has to be called first to fill the partition function matrices

SWIG Wrapper Notes

This function is attached as overloaded method *pbacktrack_sub()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.pbacktrack_sub()* in the *The RNA Python module* and the *Boltzmann Sampling* Python examples .

Parameters

- **start** (unsigned int) – The start of the subsequence to consider, i.e. 5'-end position(1-based)
- **end** (unsigned int) – The end of the subsequence to consider, i.e. 3'-end position (1-based)

Returns

A sampled secondary structure in dot-bracket notation (or NULL on error)

Return type

string

See also

RNA.pbacktrack_sub_num, RNA.pbacktrack_sub_cb, *RNA.fold_compound.pbacktrack*

Note

This function is polymorphic. It accepts RNA.fold_compound() of type RNA.FC_TYPE_SINGLE, and RNA.FC_TYPE_COMPARATIVE.

pf()

Compute the partition function Q for a given RNA sequence, or sequence alignment.

If *structure* is not a NULL pointer on input, it contains on return a string consisting of the letters " . , | { } ( ) " denoting bases that are essentially unpaired, weakly paired, strongly paired without preference, weakly upstream (downstream) paired, or strongly up- (down-)stream paired bases, respectively. If the model's compute_bpp is set to 0 base pairing probabilities will not be computed (saving CPU time), otherwise after calculations took place pr will contain the probability that bases *i* and *j* pair.

SWIG Wrapper Notes

This function is attached as method *pf()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.pf()* in the *The RNA Python module*.

Parameters

structure (string) – A pointer to the character array where position-wise pairing propensity will be stored. (Maybe NULL)

Returns

The ensemble free energy $G = -RT \cdot \log(Q)$ in kcal/mol

Return type

double

See also

RNA.fold_compound, *RNA.fold_compound*, *RNA.pf_fold*, RNA.pf_circfold, RNA.fold_compound_comparative, RNA.pf_alifold, RNA.pf_circalifold, RNA.db_from_probs, RNA.exp_params, RNA.aln_pinfo

Note

This function is polymorphic. It accepts RNA.fold_compound() of type RNA.FC_TYPE_SINGLE, and RNA.FC_TYPE_COMPARATIVE. Also, this function may return INF / 100. in case of contradicting constraints or numerical over-/underflow. In the latter case, a corresponding warning will be issued to *stdout*.

pf_dimer()

Calculate partition function and base pair probabilities of nucleic acid/nucleic acid dimers.

This is the cofold partition function folding.

SWIG Wrapper Notes

This function is attached as method *pf_dimer()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.pf_dimer()* in the *The RNA Python module*.

Parameters

structure (string) – Will hold the structure or constraints

Returns

RNA.dimer_pf() structure containing a set of energies needed for concentration computations.

Return type

RNA.dimer_pf()

See also

RNA.fold_compound

Note

This function may return INF / 100. for the *FA*, *FB*, *FAB*, *F0AB* members of the output data structure in case of contradicting constraints or numerical over-/underflow. In the latter case, a corresponding warning will be issued to *stdout*.

plist_from_probs(*fold_compound self*, *double cutoff*) → *ElemProbVector*

Create a RNA.ep() from base pair probability matrix.

The probability matrix provided via the RNA.fold_compound() is parsed and all pair probabilities above the given threshold are used to create an entry in the plist

The end of the plist is marked by sequence positions i as well as j equal to 0. This condition should be used to stop looping over its entries

Parameters
cut_off (double) – The cutoff value

Returns
A pointer to the plist that is to be created

Return type
RNA.ep() *

positional_entropy()

Compute a vector of positional entropies.

This function computes the positional entropies from base pair probabilities as

$$S(i) = -\sum_j p_{ij}\log(p_{ij}) - q_i\log(q_i)$$

with unpaired probabilities $q_i = 1 - \sum_j p_{ij}$.

Low entropy regions have little structural flexibility and the reliability of the predicted structure is high. High entropy implies many structural alternatives. While these alternatives may be functionally important, they make structure prediction more difficult and thus less reliable.

Precondition
This function requires pre-computed base pair probabilities! Thus, RNA.fold_compound.pf() must be called beforehand.

SWIG Wrapper Notes

This function is attached as method *positional_entropy()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.positional_entropy()* in the *The RNA Python module*.

Returns
A 1-based vector of positional entropies $S(i)$. (position 0 contains the sequence length)

Return type
list-like(double)

pr_energy(*e*)

SWIG Wrapper Notes

This function is attached as method *pr_energy()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.pr_energy()* in the *The RNA Python module*.

pr_structure(*structure*)

Compute the equilibrium probability of a particular secondary structure.

The probability $p(s)$ of a particular secondary structure s can be computed as

$$p(s) = \frac{exp(-\beta E(s)}{Z}$$

from the structures free energy $E(s)$ and the partition function

$$Z = \sum_s exp(-\beta E(s)), \quad \text{with} \quad \beta = \frac{1}{RT}$$

where R is the gas constant and T the thermodynamic temperature.

Precondition

The fold compound *fc* must have went through a call to RNA.fold_compound.pf() to fill the dynamic programming matrices with the corresponding partition function.

SWIG Wrapper Notes

This function is attached as method *pr_structure()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.pr_structure()* in the *The RNA Python module*.

Parameters

structure (string) – The secondary structure to compute the probability for in dot-bracket notation

Returns

The probability of the input structure (range $[0 : 1]$)

Return type

double

probs_window(*fold_compound self*, *int ulength*, *unsigned int options*, *PyObject * PyFunc*, *PyObject * data=Py_None*) → int

Compute various equilibrium probabilities under a sliding window approach.

This function applies a sliding window scan for the sequence provided with the argument *fc* and reports back equilibrium probabilities through the callback function *cb*. The data reported to the callback depends on the *options* flag.

Options: .. note:

```
The parameter `ulength` only affects computation and resulting data if␣
↪unpaired probability
computations are requested through the `options` flag.

*   RNA.PROBS_WINDOW_BPP -   Trigger base pairing probabilities.
*   RNA.PROBS_WINDOW_UP -   Trigger unpaired probabilities.
*   RNA.PROBS_WINDOW_UP_SPLIT -   Trigger detailed unpaired probabilities␣
↪split up into different
    loop type contexts.

Options may be OR-ed together
```

Parameters

- **ulength** (int) – The maximal length of an unpaired segment (only for unpaired probability computations)

- **cb** (RNA.probs_window) – The callback function which collects the pair probability data for further processing
- **data** (void *) – Some arbitrary data structure that is passed to the callback *cb*
- **options** (unsigned int) – Option flags to control the behavior of this function

Returns
0 on failure, non-zero on success

Return type
int

See also

RNA.pfl_fold_cb, *RNA.pfl_fold_up_cb*

rotational_symmetry_db(*structure*)

Determine the order of rotational symmetry for a dot-bracket structure.

Given a (permutation of multiple) RNA strand(s) and a particular secondary structure in dot-bracket notation, compute the degree of rotational symmetry. In case there is only a single linear RNA strand, the structure always has degree 1, as there are no rotational symmetries due to the direction of the nucleic acid sequence and the fixed positions of 5' and 3' ends. However, for circular RNAs, rotational symmetries might arise if the sequence consists of a concatenation of k identical subsequences.

If the argument *positions* is not *NULL*, the function stores an array of string start positions for rotational shifts that map the string back onto itself. This array has length of order of rotational symmetry, i.e. the number returned by this function. The first element *positions`[0] always contains a shift value of `0* representing the trivial rotation.

SWIG Wrapper Notes

This function is attached as method *rotational_symmetry_db()* to objects of type *fold_compound* (i.e. RNA.fold_compound()). Thus, the first argument must be omitted. In contrast to our C-implementation, this function doesn't simply return the order of rotational symmetry of the secondary structure, but returns the list *position* of cyclic permutation shifts that result in a rotationally symmetric structure. The length of the list then determines the order of rotational symmetry. See, e.g. *RNA.fold_compound.rotational_symmetry_db()* in the *The RNA Python module* .

Parameters

- **structure** (string) – The dot-bracket structure the degree of rotational symmetry is checked for
- **positions** (list-like(list-like(unsigned int))) – A pointer to an (undefined) list of alternative string start positions that lead to an identity mapping (may be NULL)

Returns
The degree of rotational symmetry of the *structure* (0 in case of any errors)

Return type
unsigned int

See also

RNA.rotational_symmetry_db, RNA.rotational_symmetry_pos, RNA.rotational_symmetry_pos_num

Note

Do not forget to release the memory occupied by *positions* after a successful execution of this function.

sc_add_SHAPE_deigan(*fold_compound self*, *DoubleVector reactivities*, *double m*, *double b*, *unsigned int options*) → int

Add SHAPE reactivity data as soft constraints (Deigan et al. method).

This approach of SHAPE directed RNA folding uses the simple linear ansatz

$$\Delta G_{\text{SHAPE}}(i) = m \ln(\text{SHAPE reactivity}(i) + 1) + b$$

to convert SHAPE reactivity values to pseudo energies whenever a nucleotide i contributes to a stacked pair. A positive slope m penalizes high reactivities in paired regions, while a negative intercept b results in a confirmatory *bonus* free energy for correctly predicted base pairs. Since the energy evaluation of a base pair stack involves two pairs, the pseudo energies are added for all four contributing nucleotides. Consequently, the energy term is applied twice for pairs inside a helix and only once for pairs adjacent to other structures. For all other loop types the energy model remains unchanged even when the experimental data highly disagrees with a certain motif.

SWIG Wrapper Notes

This function is attached as method *sc_add_SHAPE_deigan()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_add_SHAPE_deigan()* in the *The RNA Python module* .

Parameters

- **reactivities** (list-like(double)) – A vector of normalized SHAPE reactivities
- **m** (double) – The slope of the conversion function
- **b** (double) – The intercept of the conversion function
- **options** (unsigned int) – The options flag indicating how/where to store the soft constraints

Returns

1 on successful extraction of the method, 0 on errors

Return type

int

See also

RNA.fold_compound.sc_remove, *RNA.fold_compound.sc_add_SHAPE_zarringhalam*, RNA.sc_minimize_pertubation

Note

For further details, we refer to Deigan *et al.* [2009] .

sc_add_SHAPE_deigan_ali(*fold_compound self*, *StringVector shape_files*, *IntVector shape_file_association*, *double m*, *double b*, *unsigned int options*) → int

Add SHAPE reactivity data from files as soft constraints for consensus structure prediction (Deigan et al. method).

SWIG Wrapper Notes

This function is attached as method *sc_add_SHAPE_deigan_ali()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_add_SHAPE_deigan_ali()* in the *The RNA Python module* .

Parameters

- **shape_files** (const char **) – A set of filenames that contain normalized SHAPE reactivity data
- **shape_file_association** (const int *) – An array of integers that associate the files with sequences in the alignment
- **m** (double) – The slope of the conversion function
- **b** (double) – The intercept of the conversion function
- **options** (unsigned int) – The options flag indicating how/where to store the soft constraints

Returns
1 on successful extraction of the method, 0 on errors

Return type
int

sc_add_SHAPE_eddy_2(*fold_compound self*, *DoubleVector reactivities*, *DoubleVector unpaired_data*, *DoubleVector paired_data*) → int

sc_add_SHAPE_zarringhalam(*fold_compound self*, *DoubleVector reactivities*, *double b*, *double default_value*, *char const * shape_conversion*, *unsigned int options*) → int

Add SHAPE reactivity data as soft constraints (Zarringhalam et al. method).

This method first converts the observed SHAPE reactivity of nucleotide i into a probability q_i that position i is unpaired by means of a non-linear map. Then pseudo-energies of the form

$$\Delta G_{\text{SHAPE}}(x, i) = \beta \, |x_i - q_i|$$

are computed, where $x_i = 0$ if position i is unpaired and $x_i = 1$ if i is paired in a given secondary structure. The parameter β serves as scaling factor. The magnitude of discrepancy between prediction and experimental observation is represented by $|x_i - q_i|$.

SWIG Wrapper Notes

This function is attached as method *sc_add_SHAPE_zarringhalam()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_add_SHAPE_zarringhalam()* in the *The RNA Python module* .

Parameters

- **reactivities** (list-like(double)) – A vector of normalized SHAPE reactivities
- **b** (double) – The scaling factor β of the conversion function
- **default_value** (double) – The default value for a nucleotide where reactivity data is missing for
- **shape_conversion** (string) – A flag that specifies how to convert reactivities to probabilities

- **options** (unsigned int) – The options flag indicating how/where to store the soft constraints

Returns
1 on successful extraction of the method, 0 on errors

Return type
int

See also

RNA.fold_compound.sc_remove, *RNA.fold_compound.sc_add_SHAPE_deigan*, RNA.sc_minimize_pertubation

Note

For further details, we refer to Zarringhalam *et al.* [2012]

sc_add_bp(**args*)

Add soft constraints for paired nucleotides.

Repeated calls accumulate contributions for the same base pair without rounding to 0.01 kcal/mol. Partition function calculations use the accumulated value; MFE calculations round it to the nearest 0.01 kcal/mol. For example, adding 0.003 kcal/mol twice gives a PF contribution of 0.006 kcal/mol and an MFE contribution of 0.01 kcal/mol.

SWIG Wrapper Notes

This function is attached as an overloaded method *sc_add_bp()* to objects of type *fold_compound*. The method either takes arguments for a single base pair (i,j) with the corresponding energy value:

or an entire 2-dimensional matrix with dimensions n x n that stores free energy contributions for any base pair (i,j) with $1 \leq i < j \leq n$: In both variants, the optional argument *options* defaults to RNA.OPTION_DEFAULT. See, e.g. *RNA.fold_compound.sc_add_bp()* in the *The RNA Python module* .

Parameters

- **i** (unsigned int) – The 5' position of the base pair the soft constraint is added for
- **j** (unsigned int) – The 3' position of the base pair the soft constraint is added for
- **energy** (double) – The free energy (soft-constraint) in $kcal/mol$
- **options** (unsigned int) – The options flag indicating how/where to store the soft constraints

Returns
Non-zero on successful application of the constraint, 0 otherwise.

Return type
int

See also

RNA.fold_compound.sc_set_bp, *RNA.fold_compound.sc_set_up*, *RNA.fold_compound.sc_add_up*

sc_add_bt(*fold_compound self*, *PyObject * PyFunc*) → int

Bind a backtracking function pointer for generic soft constraint feature.

This function allows one to easily bind a function pointer to the soft constraint part RNA.sc() of the RNA.fold_compound(). The provided function should be used for backtracking purposes in loop regions that were altered via the generic soft constraint feature. It has to return an array of RNA.basepair() data structures, were the last element in the list is indicated by a value of -1 in it's i position.

SWIG Wrapper Notes

This function is attached as method *sc_add_bt()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_add_bt()* in the *The RNA Python module* .

Parameters

f (RNA.sc_bt) – A pointer to the function that returns additional base pairs

Returns

Non-zero on successful binding the callback function, 0 otherwise

Return type

int

See also

RNA.fold_compound.sc_add_data, RNA.fold_compound.sc_add, RNA.fold_compound.sc_add_exp

sc_add_data(*fold_compound self*, *PyObject * data*, *PyObject * callback=Py_None*) → int

Add an auxiliary data structure for the generic soft constraints callback function.

SWIG Wrapper Notes

This function is attached as method *sc_add_data()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_add_data()* in the *The RNA Python module* .

Parameters

- **data** (void *) – A pointer to the data structure that holds required data for function 'f'
- **free_data** (RNA.auxdata_free) – A pointer to a function that free's the memory occupied by *data* (Maybe NULL)

Returns

Non-zero on successful binding the data (and free-function), 0 otherwise

Return type

int

See also

`RNA.fold_compound.sc_add`, `RNA.fold_compound.sc_add_exp`, *RNA.fold_compound.sc_add_bt*

sc_add_exp_f(*fold_compound self*, *PyObject * PyFunc*) → int

Bind a function pointer for generic soft constraint feature (PF version).

This function allows one to easily bind a function pointer and corresponding data structure to the soft constraint part RNA.sc() of the RNA.fold_compound(). The function for evaluating the generic soft constraint feature has to return a pseudo free energy $\hat{E}$ as Boltzmann factor, i.e. $exp(-\hat{E}/kT)$. The required unit for E is cal/mol.

SWIG Wrapper Notes

This function is attached as method *sc_add_exp()* to objects of type *fold_compound*. See, e.g. `RNA.fold_compound.sc_add_exp()` in the *The RNA Python module* .

Parameters

exp (`RNA.sc_exp`) – A pointer to the function that evaluates the generic soft constraint feature

Returns

Non-zero on successful binding the callback function, 0 otherwise

Return type

`int`

See also

RNA.fold_compound.sc_add_bt, `RNA.fold_compound.sc_add`, *RNA.fold_compound.sc_add_data*

sc_add_f(*fold_compound self*, *PyObject * callback*) → int

Bind a function pointer for generic soft constraint feature (MFE version).

This function allows one to easily bind a function pointer and corresponding data structure to the soft constraint part RNA.sc() of the RNA.fold_compound(). The function for evaluating the generic soft constraint feature has to return a pseudo free energy $\hat{E}$ in $dacal/mol$, where $1dacal/mol = 10cal/mol$.

SWIG Wrapper Notes

This function is attached as method *sc_add()* to objects of type *fold_compound*. See, e.g. `RNA.fold_compound.sc_add()` in the *The RNA Python module* .

Parameters

f (`RNA.sc`) – A pointer to the function that evaluates the generic soft constraint feature

Returns

Non-zero on successful binding the callback function, 0 otherwise

Return type

`int`

See also

RNA.fold_compound.sc_add_data, *RNA.fold_compound.sc_add_bt*, RNA.fold_compound.sc_add_exp

sc_add_hi_motif(*fold_compound self*, *char const * seq*, *char const * structure*, *FLT_OR_DBL energy*, *unsigned int options*) → int

Add soft constraints for hairpin or internal loop binding motif.

Here is an example that adds a theophylline binding motif. Free energy contribution is derived from $k_d = 0.1\mu M$, taken from Jenison et al. 1994. At $1M$ concentration the corresponding binding free energy amounts to $-9.93\ kcal/mol$.

SWIG Wrapper Notes

This function is attached as method *sc_add_hi_motif()* to objects of type *fold_compound*. The last parameter is optional an defaults to *options* = RNA.OPTION_DEFAULT. See, e.g. *RNA.fold_compound.sc_add_hi_motif()* in the *The RNA Python module* .

Parameters

- **seq** (string) – The sequence motif (may be interspaced by '&' character
- **structure** (string) – The structure motif (may be interspaced by '&' character
- **energy** (double) – The free energy of the motif (e.g. binding free energy)
- **options** (unsigned int) – Options

Returns
non-zero value if application of the motif using soft constraints was successful

Return type
int

sc_add_stack(*fold_compound self*, *unsigned int i*, *double energy*, *unsigned int options*) → int

sc_add_stack(*fold_compound self*, *UIntVector i*, *DoubleVector energies*, *unsigned int options*) → int

sc_add_up(**args*)

Add soft constraints for unpaired nucleotides.

Repeated calls accumulate contributions for the same nucleotide without rounding to 0.01 kcal/mol. Partition function calculations use the accumulated value; MFE calculations round it to the nearest 0.01 kcal/mol before summing over stretches of unpaired nucleotides. For example, adding 0.003 kcal/mol twice at one position gives a PF contribution of 0.006 kcal/mol and an MFE contribution of 0.01 kcal/mol.

SWIG Wrapper Notes

This function is attached as an overloaded method *sc_add_up()* to objects of type *fold_compound*. The method either takes arguments for a single nucleotide i with the corresponding energy value:

or an entire vector that stores free energy contributions for each nucleotide i with $1 \leq i \leq n$: In both variants, the optional argument *options* defaults to RNA.OPTION_DEFAULT. See, e.g. *RNA.fold_compound.sc_add_up()* in the *The RNA Python module* .

Parameters

- **i** (unsigned int) – The nucleotide position the soft constraint is added for
- **energy** (double) – The free energy (soft-constraint) in $kcal/mol$
- **options** (unsigned int) – The options flag indicating how/where to store the soft constraints

Returns
Non-zero on successful application of the constraint, 0 otherwise.

Return type
int

See also

RNA.fold_compound.sc_set_up, *RNA.fold_compound.sc_add_bp*, *RNA.fold_compound.sc_set_bp*, RNA.sc_add_up_comparative

sc_init()

Initialize an empty soft constraints data structure within a RNA.fold_compound().

This function adds a proper soft constraints data structure to the RNA.fold_compound() data structure. If soft constraints already exist within the fold compound, they are removed.

SWIG Wrapper Notes

This function is attached as method *sc_init()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_init()* in the *The RNA Python module* .

See also

RNA.fold_compound.sc_set_bp, *RNA.fold_compound.sc_set_up*, *RNA.fold_compound.sc_add_SHAPE_deigan*, *RNA.fold_compound.sc_add_SHAPE_zarringhalam*, *RNA.fold_compound.sc_remove*, RNA.fold_compound.sc_add, RNA.fold_compound.sc_add_exp, RNA.sc_add_pre, RNA.sc_add_post

Note

Accepts RNA.fold_compound() of type RNA.FC_TYPE_SINGLE and RNA.FC_TYPE_COMPARATIVE

sc_mod(**args*, ***kwargs*)

Prepare soft constraint callbacks for modified base as specified in JSON string.

This function takes a RNA.sc_mod_param() data structure as obtained from RNA.sc_mod_read_from_json() or RNA.sc_mod_read_from_jsonfile() and prepares all requirements to acknowledge modified bases as specified in the provided *params* data structure. All subsequent predictions will treat each modification site special and adjust energy contributions if necessary.

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod()* in the

The RNA Python module .

Parameters

- **params** (const RNA.sc_mod_param()) – The modified base parameters
- **modification_sites** (const unsigned int *) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** (unsigned int) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns
Number of sequence positions modified base parameters will be used for

Return type
int

See also

RNA.sc_mod_read_from_json, *RNA.sc_mod_read_from_jsonfile*, *RNA.fold_compound.sc_mod_json*, *RNA.fold_compound.sc_mod_jsonfile*, *RNA.fold_compound.sc_mod_m6A*, *RNA.fold_compound.sc_mod_pseudouridine*, *RNA.fold_compound.sc_mod_inosine*, *RNA.fold_compound.sc_mod_7DA*, *RNA.fold_compound.sc_mod_purine*, RNA.sc_mod_dihydrouridine, RNA.SC_MOD_CHECK_UNMOD, RNA.SC_MOD_SILENT, RNA.SC_MOD_DEFAULT

sc_mod_7DA(**args*, ***kwargs*)

Add soft constraint callbacks for 7-deaza-adenosine (7DA).

This is a convenience wrapper to add support for 7-deaza-adenosine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Richardson and Znosko [2016] .

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod_7DA()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod_7DA()* in the *The RNA Python module* .

Parameters

- **modification_sites** (const unsigned int *) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** (unsigned int) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns
Number of sequence positions modified base parameters will be used for

Return type
int

See also

RNA.SC_MOD_CHECK_FALLBACK, RNA.SC_MOD_CHECK_UNMOD, RNA.SC_MOD_SILENT, RNA.SC_MOD_DEFAULT

sc_mod_dihydrouridine(**args*, ***kwargs*)

Add soft constraint callbacks for dihydrouridine.

This is a convenience wrapper to add support for dihydrouridine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Rosetta/RECESS predictions.

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod_dihydrouridine()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod_dihydrouridine()* in the *The RNA Python module* .

Parameters

- **modification_sites** (const unsigned int *) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** (unsigned int) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns

Number of sequence positions modified base parameters will be used for

Return type

int

See also

RNA.SC_MOD_CHECK_FALLBACK, RNA.SC_MOD_CHECK_UNMOD, RNA.SC_MOD_SILENT, RNA.SC_MOD_DEFAULT

sc_mod_inosine(**args*, ***kwargs*)

Add soft constraint callbacks for Inosine.

This is a convenience wrapper to add support for inosine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Wright *et al.* [2007] and Wright *et al.* [2018] .

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod_inosine()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod_inosine()* in the *The RNA Python module* .

Parameters

- **modification_sites** (const unsigned int *) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)

- **options** (unsigned int) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns
Number of sequence positions modified base parameters will be used for

Return type
int

See also

RNA.SC_MOD_CHECK_FALLBACK, RNA.SC_MOD_CHECK_UNMOD, RNA.SC_MOD_SILENT, RNA.SC_MOD_DEFAULT

sc_mod_json(**args*, ***kwargs*)

Prepare soft constraint callbacks for modified base as specified in JSON string.

This function prepares all requirements to acknowledge modified bases as specified in the provided *json* string. All subsequent predictions will treat each modification site special and adjust energy contributions if necessary.

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod_json()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod_json()* in the *The RNA Python module* .

Parameters

- **json** (string) – The JSON formatted string with the modified base parameters
- **modification_sites** (const unsigned int *) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** (unsigned int) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns
Number of sequence positions modified base parameters will be used for

Return type
int

See also

RNA.fold_compound.sc_mod_jsonfile, *RNA.fold_compound.sc_mod*, *RNA.fold_compound.sc_mod_m6A*, *RNA.fold_compound.sc_mod_pseudouridine*, *RNA.fold_compound.sc_mod_inosine*, *RNA.fold_compound.sc_mod_7DA*, *RNA.fold_compound.sc_mod_purine*, *RNA.fold_compound.sc_mod_dihydrouridine*, RNA.SC_MOD_CHECK_FALLBACK, RNA.SC_MOD_CHECK_UNMOD, RNA.SC_MOD_SILENT, RNA.SC_MOD_DEFAULT, *Modified Bases*

sc_mod_jsonfile(**args*, ***kwargs*)

Prepare soft constraint callbacks for modified base as specified in JSON string.

Similar to RNA.fold_compound.sc_mod_json(), this function prepares all requirements to acknowledge modified bases as specified in the provided *json* file. All subsequent predictions will treat each modification site special and adjust energy contributions if necessary.

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod_jsonfile()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod_jsonfile()* in the *The RNA Python module* .

Parameters

- **json_file** (string) – The JSON formatted string with the modified base parameters
- **modification_sites** (const unsigned int *) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** (unsigned int) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns

Number of sequence positions modified base parameters will be used for

Return type

int

See also

RNA.fold_compound.sc_mod_json, *RNA.fold_compound.sc_mod*, *RNA.fold_compound.sc_mod_m6A*, *RNA.fold_compound.sc_mod_pseudouridine*, *RNA.fold_compound.sc_mod_inosine*, *RNA.fold_compound.sc_mod_7DA*, *RNA.fold_compound.sc_mod_purine*, *RNA.fold_compound.sc_mod_dihydrouridine*, RNA.SC_MOD_CHECK_FALLBACK, RNA.SC_MOD_CHECK_UNMOD, RNA.SC_MOD_SILENT, RNA.SC_MOD_DEFAULT, *Modified Bases*

sc_mod_m6A(**args*, ***kwargs*)

Add soft constraint callbacks for N6-methyl-adenosine (m6A).

This is a convenience wrapper to add support for m6A using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Kierzek *et al.* [2022] .

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod_m6A()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod_m6A()* in the *The RNA Python module* .

Parameters

- **modification_sites** (const unsigned int *) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** (unsigned int) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns
Number of sequence positions modified base parameters will be used for

Return type
`int`

See also

`RNA.SC_MOD_CHECK_FALLBACK`, `RNA.SC_MOD_CHECK_UNMOD`, `RNA.SC_MOD_SILENT`, `RNA.SC_MOD_DEFAULT`

`sc_mod_pseudouridine`(**args*, ***kwargs*)

Add soft constraint callbacks for Pseudouridine.

This is a convenience wrapper to add support for pseudouridine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Hudson *et al.* [2013] .

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod_pseudouridine()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod_pseudouridine()* in the *The RNA Python module* .

Parameters

- **modification_sites** (`const unsigned int *`) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** (`unsigned int`) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns
Number of sequence positions modified base parameters will be used for

Return type
`int`

See also

`RNA.SC_MOD_CHECK_FALLBACK`, `RNA.SC_MOD_CHECK_UNMOD`, `RNA.SC_MOD_SILENT`, `RNA.SC_MOD_DEFAULT`

`sc_mod_purine`(**args*, ***kwargs*)

Add soft constraint callbacks for Purine (a.k.a. nebularine).

This is a convenience wrapper to add support for Purine using the soft constraint callback mechanism. Modification sites are provided as a list of sequence positions (1-based). Energy parameter corrections are derived from Jolley and Znosko [2017] .

SWIG Wrapper Notes

This function is attached as overloaded method *sc_mod_purine()* to objects of type *fold_compound* with default *options* = RNA.SC_MOD_DEFAULT. See, e.g. *RNA.fold_compound.sc_mod_purine()* in the *The RNA Python module* .

Parameters

- **modification_sites** (const unsigned int *) – A list of modification site, i.e. positions that contain the modified base (1-based, last element in the list indicated by 0)
- **options** (unsigned int) – A bitvector of options how to handle the input, e.g. RNA.SC_MOD_DEFAULT

Returns
Number of sequence positions modified base parameters will be used for

Return type
int

See also

RNA.SC_MOD_CHECK_FALLBACK, RNA.SC_MOD_CHECK_UNMOD, RNA.SC_MOD_SILENT, RNA.SC_MOD_DEFAULT

sc_multi_cb_add(*fold_compound self*, *PyObject * f*, *PyObject * f_exp=Py_None*, *PyObject * data=Py_None*, *PyObject * data_prepare=Py_None*, *PyObject * data_free=Py_None*, *unsigned int decomp_type=0*) → unsigned int

sc_probing(*fold_compound self*, *probing_data data*) → int

Apply probing data (e.g. SHAPE) to guide the structure prediction.

SWIG Wrapper Notes

This function is attached as method **sc_probing()** to objects of type **fold_compound**. See, e.g. *RNA.fold_compound.sc_probing()* in the *The RNA Python module* .

Parameters
data (*RNA.probing_data()*) – The prepared probing data and probing data integration strategy

Returns
The number of probing data sets applied, 0 upon any error

Return type
int

See also

RNA.probing_data, *RNA.probing_data_free*, *RNA.probing_data_deigan*, *RNA.probing_data_deigan_comparative*, *RNA.probing_data_zarringhalam*, *RNA.probing_data_zarringhalam_comparative*, *RNA.probing_data_eddy*, *RNA.probing_data_eddy_comparative*

sc_remove()

Remove soft constraints from RNA.fold_compound().

SWIG Wrapper Notes

This function is attached as method *sc_remove()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_remove()* in the *The RNA Python module* .

Note

Accepts RNA.fold_compound() of type RNA.FC_TYPE_SINGLE and RNA.FC_TYPE_COMPARATIVE

sc_set_bp(*fold_compound self*, *DoubleDoubleVector constraints*, *unsigned int options*) → int

Set soft constraints for paired nucleotides.

Values are stored without rounding to 0.01 kcal/mol. Partition function calculations use these values directly; MFE calculations round the accumulated contribution for each base pair to the nearest 0.01 kcal/mol.

SWIG Wrapper Notes

This function is attached as method *sc_set_bp()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_set_bp()* in the *The RNA Python module* .

Parameters

- **constraints** (const FLT_OR_DBL **) – A two-dimensional array of pseudo free energies in $kcal/mol$
- **options** (unsigned int) – The options flag indicating how/where to store the soft constraints

Returns

Non-zero on successful application of the constraint, 0 otherwise.

Return type

int

See also

RNA.fold_compound.sc_add_bp, *RNA.fold_compound.sc_set_up*, *RNA.fold_compound.sc_add_up*

Note

This function replaces any pre-exisitng soft constraints with the ones supplied in *constraints*.

sc_set_stack(*fold_compound self*, *DoubleVector constraints*, *unsigned int options*) → int

sc_set_stack(*fold_compound self*, *DoubleDoubleVector constraints*, *unsigned int options*) → int

sc_set_up(*fold_compound self*, *DoubleVector constraints*, *unsigned int options*) → int

Set soft constraints for unpaired nucleotides.

Values are stored without rounding to 0.01 kcal/mol. Partition function calculations use these values directly; MFE calculations round the accumulated contribution for each nucleotide to the nearest 0.01 kcal/mol before summing over stretches of unpaired nucleotides.

SWIG Wrapper Notes

This function is attached as method *sc_set_up()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.sc_set_up()* in the *The RNA Python module* .

Parameters

- **constraints** (const FLT_OR_DBL *) – A vector of pseudo free energies in $kcal/mol$
- **options** (unsigned int) – The options flag indicating how/where to store the soft constraints

Returns
Non-zero on successful application of the constraint, 0 otherwise.

Return type
int

See also

RNA.fold_compound.sc_add_up, *RNA.fold_compound.sc_set_bp*, *RNA.fold_compound.sc_add_bp*

Note

This function replaces any pre-exisitng soft constraints with the ones supplied in *constraints*.

property sequence

sequence_add(**args*, ***kwargs*)

property sequence_encoding

property sequence_encoding2

sequence_prepare()

sequence_remove(i)

sequence_remove_all()

stack_prob(*cutoff=1e-05*)

Compute stacking probabilities.

For each possible base pair (i, j), compute the probability of a stack (i, j), $(i + 1, j - 1)$.

SWIG Wrapper Notes

This function is attached as overloaded method *stack_prob()* to objects of type *fold_compound*. The optional argument *cutoff* defaults to 1e-5. See, e.g. *RNA.fold_compound.stack_prob()* in the *The RNA Python module*.

Parameters
cutoff (double) – A cutoff value that limits the output to stacks with $p >$ cutoff.

Returns
A list of stacks with enclosing base pair (i, j) and probabiltiy p

Return type
RNA.ep() *

property strand_end

property strand_number

property strand_order

property strand_start

property strands

subopt(*fold_compound self*, *int delta*, *int sorted=1*, *FILE * nullfile=None*) → *SuboptVector*

Returns list of subopt structures or writes to fp.

This function produces **all** suboptimal secondary structures within 'delta' * 0.01 kcal/mol of the optimum, see Wuchty *et al.* [1999] . The results are either directly written to a 'fp' (if 'fp' is not NULL), or (fp==NULL) returned in a RNA.subopt_solution() * list terminated by an entry were the 'structure' member is NULL.

SWIG Wrapper Notes

This function is attached as method **subopt()** to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.subopt()* in the *The RNA Python module*.

Parameters

- **delta** (int)
- **sorted** (int) – Sort results by energy in ascending order
- **fp** (FILE *)

Return type
RNA.subopt_solution() *

See also

RNA.fold_compound.subopt_cb, *RNA.fold_compound.subopt_zuker*

Note

This function requires all multibranch loop DP matrices for unique multibranch loop backtracing. Therefore, the supplied RNA.fold_compound()`fc` (argument 1) must be initialized with RNA.md().uniq_ML = 1, for instance like this:

subopt_cb(*fold_compound self*, *int delta*, *PyObject * PyFunc*, *PyObject * data=Py_None*) → PyObject *

Generate suboptimal structures within an energy band arround the MFE.

This is the most generic implementation of the suboptimal structure generator according to Wuchty *et al.* [1999] . Identical to RNA.fold_compound.subopt(), it computes all secondary structures within an energy band *delta* arround the MFE. However, this function does not print the resulting structures and their corresponding free energies to a file pointer, or returns them as a list. Instead, it calls a user-provided callback function which it passes the structure in dot-bracket format, the corresponding free energy in kcal/mol, and a user-provided data structure each time a structure was backtracked successfully. This function indicates the final output, i.e. the end of the backtracking procedure by passing NULL instead of an actual dot-bracket string to the callback.

SWIG Wrapper Notes

This function is attached as method **subopt_cb()** to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.subopt_cb()* in the *The RNA Python module*.

Parameters

- **delta** (int) – Energy band arround the MFE in 10cal/mol, i.e. deka-calories
- **cb** (RNA.subopt_result) – Pointer to a callback function that handles the backtracked structure and its free energy in kcal/mol
- **data** (void *) – Pointer to some data structure that is passed along to the callback

See also

RNA.subopt_result, *RNA.fold_compound.subopt*, *RNA.fold_compound.subopt_zuker*

Note

This function requires all multibranch loop DP matrices for unique multibranch loop backtracing. Therefore, the supplied RNA.fold_compound()`fc` (argument 1) must be initialized with RNA.md().uniq_ML = 1, for instance like this:

subopt_zuker()

Compute Zuker type suboptimal structures.

Compute Suboptimal structures according to Zuker [1989] , i.e. for every possible base pair the minimum energy structure containing the resp. base pair. Returns a list of these structures and their energies.

SWIG Wrapper Notes

This function is attached as method **subopt_zuker()** to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.subopt_zuker()* in the *The RNA Python module*.

Returns

List of zuker suboptimal structures

Return type

RNA.subopt_solution() *

See also

RNA.fold_compound.subopt, *zukersubopt*, zukersubopt_par

property thisown

The membership flag

property type

ud_add_motif(*fold_compound self*, *std: :string motif*, *double motif_en*, *std: :string name = ""*, *unsigned int options*)

Add an unstructured domain motif, e.g. for ligand binding.

This function adds a ligand binding motif and the associated binding free energy to the RNA.ud() attribute of a RNA.fold_compound(). The motif data will then be used in subsequent secondary structure predictions. Multiple calls to this function with different motifs append all additional data to a list of ligands, which all will be evaluated. Ligand motif data can be removed from the RNA.fold_compound() again using the RNA.fold_compound.ud_remove() function. The loop type parameter allows one to limit the ligand binding to particular loop type, such as the exterior loop, hairpin loops, internal loops, or multibranch loops.

Parameters

- **motif** (string) – The sequence motif the ligand binds to
- **motif_en** (double) – The binding free energy of the ligand in kcal/mol
- **motif_name** (string) – The name/id of the motif (may be *NULL*)
- **loop_type** (unsigned int) – The loop type the ligand binds to

See also

RNA.UNSTRUCTURED_DOMAIN_EXT_LOOP, RNA.UNSTRUCTURED_DOMAIN_HP_LOOP, RNA.UNSTRUCTURED_DOMAIN_INT_LOOP, RNA.UNSTRUCTURED_DOMAIN_MB_LOOP, RNA.UNSTRUCTURED_DOMAIN_ALL_LOOPS, *RNA.fold_compound.ud_remove*

ud_remove()

Remove ligand binding to unpaired stretches.

This function removes all ligand motifs that were bound to a RNA.fold_compound() using the RNA.fold_compound.ud_add_motif() function.

SWIG Wrapper Notes

This function is attached as method *ud_remove()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.ud_remove()* in the *The RNA Python module*.

ud_set_data(*fold_compound self*, *PyObject * data*, *PyObject * free_cb=Py_None*) → PyObject *

Attach an auxiliary data structure.

This function binds an arbitrary, auxiliary data structure for user-implemented ligand binding. The optional callback *free_cb* will be passed the bound data structure whenever the RNA.fold_compound() is removed from memory to avoid memory leaks.

SWIG Wrapper Notes

This function is attached as method *ud_set_data()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.ud_set_data()* in the *The RNA Python module*.

Parameters

- **data** (void *) – A pointer to the auxiliary data structure
- **free_cb** (RNA.auxdata_free) – A pointer to a callback function that free's memory occupied by *data*

See also

RNA.fold_compound.ud_set_prod_rule_cb, *RNA.fold_compound.ud_set_exp_prod_rule_cb*, *RNA.fold_compound.ud_remove*

ud_set_exp_prod_rule_cb(*fold_compound self*, *PyObject * prod_cb*, *PyObject * eval_cb*) → PyObject *

Attach production rule for partition function.

This function is the partition function companion of RNA.fold_compound.ud_set_prod_rule_cb().

Use it to bind callbacks to (i) fill the U production rule dynamic programming matrices and/or prepare the RNA.unstructured_domain().data, and (ii) provide a callback to retrieve partition functions for subsegments $[i, j]$.

SWIG Wrapper Notes

This function is attached as method *ud_set_exp_prod_rule_cb()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.ud_set_exp_prod_rule_cb()* in the *The RNA Python module*.

Parameters

- **pre_cb** (RNA.ud_exp_production) – A pointer to a callback function for the B production rule
- **exp_e_cb** (RNA.ud_exp) – A pointer to a callback function that retrieves the partition function for a segment $[i, j]$ that may be bound by one or more ligands.

See also

RNA.fold_compound.ud_set_prod_rule_cb

ud_set_prob_cb(*fold_compound self*, *PyObject * setter_cb*, *PyObject * getter_cb*) → PyObject *

SWIG Wrapper Notes

This function is attached as method *ud_set_prob_cb()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.ud_set_prob_cb()* in the *The RNA Python module*.

ud_set_prod_rule_cb(*fold_compound self*, *PyObject * prod_cb*, *PyObject * eval_cb*) → PyObject *

Attach production rule callbacks for free energies computations.

Use this function to bind a user-implemented grammar extension for unstructured domains.

The callback *e_cb* needs to evaluate the free energy contribution $f(i, j)$ of the unpaired segment $[i, j]$. It will be executed in each of the regular secondary structure production rules. Whenever the callback is passed the RNA.UNSTRUCTURED_DOMAIN_MOTIF flag via its *loop_type* parameter the contribution of any ligand that consecutively binds from position i to j (the white box) is requested. Otherwise, the callback usually performs a lookup in the precomputed B matrices. Which B matrix is addressed will be indicated by the flags RNA.UNSTRUCTURED_DOMAIN_EXT_LOOP, RNA.UNSTRUCTURED_DOMAIN_HP_LOOPRNA.UNSTRUCTURED_DOMAIN_INT_LOOP, and RNA.UNSTRUCTURED_DOMAIN_MB_LOOP. As their names already imply, they specify exterior loops (F production rule), hairpin loops and internal loops (C production rule), and multibranch loops (M and $M1$ production rule).

The *pre_cb* callback will be executed as a pre-processing step right before the regular secondary structure rules. Usually one would use this callback to fill the dynamic programming matrices *U* and preparations of the auxiliary data structure RNA.unstructured_domain().data

SWIG Wrapper Notes

This function is attached as method *ud_set_prod_rule_cb()* to objects of type *fold_compound*. See, e.g. *RNA.fold_compound.ud_set_prod_rule_cb()* in the *The RNA Python module*.

Parameters

- **pre_cb** (RNA.ud_production) – A pointer to a callback function for the *B* production rule
- **e_cb** (RNA.ud) – A pointer to a callback function for free energy evaluation

zsc_compute(*i*, *j*, *e*)

zsc_compute_raw(*i*, *j*, *e*)

zsc_filter_free()

zsc_filter_init(**args*, ***kwargs*)

zsc_filter_on()

zsc_filter_threshold()

zsc_filter_update(**args*, ***kwargs*)

RNA.**free_alifold_arrays**()

Free the memory occupied by MFE alifold functions.

Deprecated since version 2.8.0: Usage of this function is discouraged! It only affects memory being free'd that was allocated by an old API function before. Release of memory occupied by the newly introduced RNA.fold_compound() is handled by RNA.fold_compound_free()

See also

RNA.fold_compound_free

RNA.**free_arrays**()

Free arrays for mfe folding.

Deprecated since version 2.8.0: See RNA.fold(), RNA.circfold(), or RNA.fold_compound.mfe() and RNA.fold_compound() for the usage of the new API!

RNA.**free_co_arrays**()

Free memory occupied by cofold().

Deprecated since version 2.8.0: This function will only free memory allocated by a prior call of cofold() or cofold_par(). See RNA.fold_compound.mfe_dimer() for how to use the new API

See also

RNA.fc_destroy, *RNA.fold_compound.mfe_dimer*

Note

folding matrices now reside in the fold compound, and should be free'd there

RNA.**free_co_pf_arrays**()

Free the memory occupied by co_pf_fold().

Deprecated since version 2.8.0: This function will be removed for the new API soon! See RNA.fold_compound.pf_dimer(), RNA.fold_compound(), and RNA.fold_compound_free() for an alternative

RNA.**free_path**(*path*)

Free memory allocated by get_path() function.

Deprecated since version 2.8.0: Use RNA.path_free() instead!

Parameters

path (RNA.path() *) – pointer to memory to be freed

RNA.**free_pf_arrays**()

Free arrays for the partition function recursions.

Call this function if you want to free all allocated memory associated with the partition function forward recursion.

Deprecated since version 2.8.0: See RNA.fold_compound() and its related functions for how to free memory occupied by the dynamic programming matrices

Note

Successive calls of pf_fold(), pf_circ_fold() already check if they should free any memory from a previous run. **OpenMP notice:**

This function should be called before leaving a thread in order to avoid leaking memory

Postcondition

All memory allocated by pf_fold_par(), pf_fold() or pf_circ_fold() will be free'd

See also

pf_fold_par, *pf_fold*, *pf_circ_fold*

RNA.**free_profile**(*T*)

free space allocated in Make_bp_profile

Backward compatibility only. You can just use plain free()

RNA.**free_tree**(*t*)

Free the memory allocated for Tree t.

Parameters

t (Tree *)

RNA.**get_aligned_line**(*arg1*)

RNA.**get_centroid_struct_pl**(*length*, *dist*, *pl*)

Get the centroid structure of the ensemble.

Deprecated since version 2.8.0: This function was renamed to RNA.centroid_from_plist()

RNA.**get_centroid_struct_pr**(*length*, *dist*, *pr*)

Get the centroid structure of the ensemble.

Deprecated since version 2.8.0: This function was renamed to RNA.centroid_from_probs()

RNA.**get_concentrations**(*FcAB*, *FcAA*, *FcBB*, *FEA*, *FEB*, *A0*, *BO*)

RNA.**get_gquad_L_matrix**(*S*, *start*, *maxdist*, *n*, *g*, *P*)

RNA.**get_gquad_ali_matrix**(*n*, *S_cons*, *S*, *a2s*, *n_seq*, *P*)

RNA.**get_gquad_matrix**(*S*, *P*)

Get a triangular matrix prefilled with minimum free energy contributions of G-quadruplexes.

At each position ij in the matrix, the minimum free energy of any G-quadruplex delimited by i and j is stored. If no G-quadruplex formation is possible, the matrix element is set to INF. Access the elements in the matrix via matrix[indx[j]+i]. To get the integer array indx see get_jindx().

Parameters

- **S** (`list-like(int)`) – The encoded sequence
- **P** (`RNA.param() *`) – A pointer to the data structure containing the precomputed energy contributions

Returns
A pointer to the G-quadruplex contribution matrix

Return type
`int *`

See also

`get_jindx`, `encode_sequence`

RNA.**get_gquad_pattern_pf**(*S*, *i*, *j*, *pf*, *L*, *l*)

RNA.**get_gquad_pf_matrix**(*S*, *scale*, *pf*)

RNA.**get_gquad_pf_matrix_comparative**(*n*, *S_cons*, *S*, *a2s*, *scale*, *n_seq*, *pf*)

RNA.**get_multi_input_line**(*string*, *options*)

RNA.**get_path**(*std: :string seq*, *std: :string s1*, *std: :string s2*, *int maxkeep*) → *PathVector*

RNA.**get_plist_gquad_from_db**(*structure*, *pr*)

RNA.**get_plist_gquad_from_pr**(*S*, *gi*, *gj*, *q_gq*, *probs*, *scale*, *pf*)

RNA.**get_plist_gquad_from_pr_max**(*S*, *gi*, *gj*, *q_gq*, *probs*, *scale*, *L*, *l*, *pf*)

RNA.**get_pr**(*i*, *j*)

RNA.**get_xy_coordinates**(*char const * structure*) → *COORDINATE*

Compute nucleotide coordinates for secondary structure plot.

This function takes a secondary structure and computes X-Y coordinates for each nucleotide that then can be used to create a structure plot. The parameter *plot_type* is used to select the underlying layout algorithm. Currently, the following selections are provided:

- RNA.PLOT_TYPE_SIMPLE
- RNA.PLOT_TYPE_NAVIEW
- RNA.PLOT_TYPE_CIRCULAR

- RNA.PLOT_TYPE_TURTLE
- RNA.PLOT_TYPE_PUZZLER

Passing an unsupported selection leads to the default algorithm RNA.PLOT_TYPE_NAVIEW

Here is a simple example how to use this function, assuming variable *structure* contains a valid dot-bracket string:

Parameters

- **structure** (string) – The secondary structure in dot-bracket notation
- **x** (float **) – The address of a pointer of X coordinates (pointer will point to memory, or NULL on failure)
- **y** (float **) – The address of a pointer of Y coordinates (pointer will point to memory, or NULL on failure)
- **plot_type** (int) – The layout algorithm to be used

Returns

The length of the structure on success, 0 otherwise

Return type

int

See also

RNA.plot_coords_pt, RNA.plot_coords_simple, RNA.plot_coords_naview, RNA.plot_coords_circular, RNA.plot_coords_turtle, RNA.plot_coords_puzzler

Note

On success, this function allocates memory for X and Y coordinates and assigns the pointers at addressess *x* and *y* to the corresponding memory locations. It's the users responsibility to cleanup this memory after usage!

RNA.**gettype**(*ident*)

RNA.**gmlRNA**(*string*, *structure*, *ssfile*, *option*)

Produce a secondary structure graph in Graph Meta Language (gml) and write it to a file.

If 'option' is an uppercase letter the RNA sequence is used to label nodes, if 'option' equals *'X'* or *'x'* the resulting file will coordinates for an initial layout of the graph.

Parameters

- **string** (string) – The RNA sequence
- **structure** (string) – The secondary structure in dot-bracket notation
- **ssfile** (string) – The filename of the gml output
- **option** (char) – The option flag

Returns

1 on success, 0 otherwise

Return type

int

RNA.**gq_parse**(*std: :string structure*) → unsigned int

RNA.**hamming**(*s1*, *s2*)

Calculate hamming distance between two sequences.

Parameters

- **s1** (string) – The first sequence
- **s2** (string) – The second sequence

Returns
The hamming distance between s1 and s2

Return type
int

RNA.**hamming_bound**(*s1*, *s2*, *n*)

Calculate hamming distance between two sequences up to a specified length.

This function is similar to RNA.hamming_distance() but instead of comparing both sequences up to their actual length only the first 'n' characters are taken into account

Parameters

- **s1** (string) – The first sequence
- **s2** (string) – The second sequence
- **n** (int) – The length of the subsequences to consider (starting from the 5' end)

Returns
The hamming distance between s1 and s2

Return type
int

RNA.**hamming_distance**(*s1*, *s2*)

RNA.**hamming_distance_bound**(*s1*, *s2*, *n*)

class RNA.**hc**

Bases: object

The hard constraints data structure.

The content of this data structure determines the decomposition pattern used in the folding recursions. Attribute 'matrix' is used as source for the branching pattern of the decompositions during all folding recursions. Any entry in matrix[i,j] consists of the 6 LSB that allows one to distinguish the following types of base pairs:

- in the exterior loop (RNA.CONSTRAINT_CONTEXT_EXT_LOOP)
- enclosing a hairpin (RNA.CONSTRAINT_CONTEXT_HP_LOOP)
- enclosing an internal loop (RNA.CONSTRAINT_CONTEXT_INT_LOOP)
- enclosed by an exterior loop (RNA.CONSTRAINT_CONTEXT_INT_LOOP_ENC)
- enclosing a multi branch loop (RNA.CONSTRAINT_CONTEXT_MB_LOOP)
- enclosed by a multi branch loop (RNA.CONSTRAINT_CONTEXT_MB_LOOP_ENC)

The four linear arrays 'up_xxx' provide the number of available unpaired nucleotides (including position i) 3' of each position in the sequence.

See also

RNA.fold_compound.hc_init, RNA.hc_free, RNA.CONSTRAINT_CONTEXT_EXT_LOOP, RNA.CONSTRAINT_CONTEXT_HP_LOOP, RNA.CONSTRAINT_CONTEXT_INT_LOOP, RNA.CONSTRAINT_CONTEXT_MB_LOOP, RNA.CONSTRAINT_CONTEXT_MB_LOOP_ENC

type

Type
vrna_hc_type_e

n

Type
unsigned int

strands

Type
unsigned int

sn

Type
list-like(unsigned int)

state

Type
unsigned char

mx

Type
unsigned char *

matrix_local

Type
unsigned char **

up_ext

A linear array that holds the number of allowed unpaired nucleotides in an exterior loop.

Type
list-like(unsigned int)

up_hp

A linear array that holds the number of allowed unpaired nucleotides in a hairpin loop.

Type
list-like(unsigned int)

up_int

A linear array that holds the number of allowed unpaired nucleotides in an internal loop.

Type
list-like(unsigned int)

up_ml

A linear array that holds the number of allowed unpaired nucleotides in a multi branched loop.

Type
list-like(unsigned int)

f

A function pointer that returns whether or not a certain decomposition may be evaluated.

Type
vrna_hc_eval_f

data

A pointer to some structure where the user may store necessary data to evaluate its generic hard constraint function.

Type
void *

free_data

A pointer to a function to free memory occupied by auxiliary data.

The function this pointer is pointing to will be called upon destruction of the RNA.hc(), and provided with the RNA.hc().data pointer that may hold auxiliary data. Hence, to avoid leaking memory, the user may use this pointer to free memory occupied by auxiliary data.

Type
vrna_auxdata_free_f

depot

Type
vrna_hc_depot_t *

eval_ext

Type
vrna_hc_eval_loop_f

eval_hp

Type
vrna_hc_eval_hp_f

eval_int

Type
vrna_hc_eval_int_f

eval_mb

Type
vrna_hc_eval_loop_f

The hard constraints data structure.

The content of this data structure determines the decomposition pattern used in the folding recursions. Attribute 'matrix' is used as source for the branching pattern of the decompositions during all folding recursions. Any entry in matrix[i,j] consists of the 6 LSB that allows one to distinguish the following types of base pairs:

- in the exterior loop (RNA.CONSTRAINT_CONTEXT_EXT_LOOP)
- enclosing a hairpin (RNA.CONSTRAINT_CONTEXT_HP_LOOP)
- enclosing an internal loop (RNA.CONSTRAINT_CONTEXT_INT_LOOP)
- enclosed by an exterior loop (RNA.CONSTRAINT_CONTEXT_INT_LOOP_ENC)
- enclosing a multi branch loop (RNA.CONSTRAINT_CONTEXT_MB_LOOP)
- enclosed by a multi branch loop (RNA.CONSTRAINT_CONTEXT_MB_LOOP_ENC)

The four linear arrays 'up_xxx' provide the number of available unpaired nucleotides (including position i) 3' of each position in the sequence.

See also

RNA.fold_compound.hc_init, RNA.hc_free, RNA.CONSTRAINT_CONTEXT_EXT_LOOP, RNA.CONSTRAINT_CONTEXT_HP_LOOP, RNA.CONSTRAINT_CONTEXT_INT_LOOP, RNA.CONSTRAINT_CONTEXT_MB_LOOP, RNA.CONSTRAINT_CONTEXT_MB_LOOP_ENC

type

Type
vrna_hc_type_e

n

Type
unsigned int

strands

Type
unsigned int

sn

Type
list-like(unsigned int)

state

Type
unsigned char

mx

Type
unsigned char *

matrix_local

Type
unsigned char **

up_ext

A linear array that holds the number of allowed unpaired nucleotides in an exterior loop.

Type
list-like(unsigned int)

up_hp

A linear array that holds the number of allowed unpaired nucleotides in a hairpin loop.

Type
list-like(unsigned int)

up_int

A linear array that holds the number of allowed unpaired nucleotides in an internal loop.

Type
list-like(unsigned int)

up_ml

A linear array that holds the number of allowed unpaired nucleotides in a multi branched loop.

Type
list-like(unsigned int)

f

A function pointer that returns whether or not a certain decomposition may be evaluated.

Type
vrna_hc_eval_f

data

A pointer to some structure where the user may store necessary data to evaluate its generic hard constraint function.

Type
void *

free_data

A pointer to a function to free memory occupied by auxiliary data.

The function this pointer is pointing to will be called upon destruction of the RNA.hc(), and provided with the RNA.hc().data pointer that may hold auxiliary data. Hence, to avoid leaking memory, the user may use this pointer to free memory occupied by auxiliary data.

Type
vrna_auxdata_free_f

depot

Type
vrna_hc_depot_t *

eval_ext

Type
vrna_hc_eval_loop_f

eval_hp

Type
vrna_hc_eval_hp_f

eval_int

Type
vrna_hc_eval_int_f

eval_mb

Type
vrna_hc_eval_loop_f

property mx

property n

property thisown

The membership flag

property type

property up_ext

property up_hp

property up_int

property up_ml

RNA.**heat_capacity**(*sequence*, *T_min=0.0*, *T_max=100.0*, *T_increment=1.0*, *mpoints=2*)

Compute the specific heat for an RNA (simplified variant).

Similar to RNA.fold_compound.heat_capacity(), this function computes an RNAs specific heat in a given temperature range from the partition function by numeric differentiation. This simplified version, however, only requires the RNA sequence as input instead of a RNA.fold_compound() data structure. The result is returned as a list of pairs of temperature in C and specific heat in Kcal/(Mol*K).

Users can specify the temperature range for the computation from *T_min* to *T_max*, as well as the increment step size *T_increment*. The latter also determines how many times the partition function is computed. Finally, the parameter *mpoints* determines how smooth the curve should be. The algorithm itself fits a parabola to $2 \cdot mpoints + 1$ data points to calculate 2nd derivatives. Increasing this parameter produces a smoother curve.

SWIG Wrapper Notes

This function is available as overloaded function *heat_capacity()*. If the optional function arguments *T_min*, *T_max*, *T_increment*, and *mpoints* are omitted, they default to 0.0, 100.0, 1.0 and 2, respectively. See, e.g. `RNA.head_capacity()` in the *The RNA Python module*.

Parameters

- **sequence** (`string`) – The RNA sequence input (must be uppercase)
- **T_min** (`float`) – Lowest temperature in C
- **T_max** (`float`) – Highest temperature in C
- **T_increment** (`float`) – Stepsize for temperature incrementation in C (a reasonable choice might be 1C)
- **mpoints** (`unsigned int`) – The number of interpolation points to calculate 2nd derivative (a reasonable choice might be 2, min: 1, max: 100)

Returns

A list of pairs of temperatures and corresponding heat capacity or *NULL* upon any failure. The last entry of the list is indicated by a **temperature** field set to a value smaller than *T_min*

Return type

`RNA.heat_capacity() *`

See also

RNA.fold_compound.heat_capacity, *RNA.fold_compound.heat_capacity_cb*, *RNA.heat_capacity*, *RNA.heat_capacity*

class RNA.**heat_capacity_result**

Bases: `object`

property heat_capacity

property temperature

property thisown

The membership flag

class RNA.**hx**(*start*, *end*, *length*, *up5=0*, *up3=0*)

Bases: `object`

property end

property length

property start

property thisown

The membership flag

property up3

property up5

RNA.**hx_from_ptable**(*IntVector pt*) → *HelixVector*

RNA.**hx_from_ptable**(*varArrayShort pt*) → *HelixVector*

Convert a pair table representation of a secondary structure into a helix list.

Parameters
pt (list-like(int)) – The secondary structure in pair table representation

Returns
The secondary structure represented as a helix list

Return type
RNA.hx() *

RNA.**hx_merge**(*list*, *maxdist=0*)

RNA.**init_pf_circ_fold**(*length*)

RNA.**init_pf_fold**(*length*)

Allocate space for pf_fold().

Deprecated since version 2.8.0: This function is obsolete and will be removed soon!

RNA.**init_rand**(**args*)

Initialize the random number generator with a pre-defined seed.

SWIG Wrapper Notes

This function is available as an overloaded function **init_rand()** where the argument *seed* is optional. See, e.g. *RNA.init_rand()* in the *The RNA Python module*.

Parameters
seed (unsigned int) – The seed for the random number generator

See also

RNA.init_rand, *RNA.urn*

RNA.**initialize_cofold**(*length*)

allocate arrays for folding

Deprecated since version 2.8.0.

class RNA.**intArray**(*nelements*)

Bases: object

cast()

static frompointer(*t*)

property thisown

The membership flag

RNA.**intP_getitem**(*ary*, *index*)

RNA.**intP_setitem**(*ary*, *index*, *value*)

RNA.**int_urn**(*_from*, *to*)

Generates a pseudo random integer in a specified range.

Parameters

- **from** (int) – The first number in range
- **to** (int) – The last number in range

Returns

A pseudo random number in range [from, to]

Return type

int

See also

RNA.urn, *RNA.init_rand*

RNA.**inverse_fold**(*char * start*, *char const * target*) → char *

Find sequences with predefined structure.

This function searches for a sequence with minimum free energy structure provided in the parameter 'target', starting with sequence 'start'. It returns 0 if the search was successful, otherwise a structure distance in terms of the energy difference between the search result and the actual target 'target' is returned. The found sequence is returned in 'start'. If give_up is set to 1, the function will return as soon as it is clear that the search will be unsuccessful, this speeds up the algorithm if you are only interested in exact solutions.

Parameters

- **start** (string) – The start sequence
- **target** (string) – The target secondary structure in dot-bracket notation

Returns

The distance to the target in case a search was unsuccessful, 0 otherwise

Return type

float

RNA.**inverse_pf_fold**(*char * start*, *char const * target*) → char *

Find sequence that maximizes probability of a predefined structure.

This function searches for a sequence with maximum probability to fold into the provided structure 'target' using the partition function algorithm. It returns $-kT \cdot \log(p)$ where p is the frequency of 'target' in the ensemble of possible structures. This is usually much slower than inverse_fold().

Parameters

- **start** (string) – The start sequence
- **target** (string) – The target secondary structure in dot-bracket notation

Returns

The distance to the target in case a search was unsuccessful, 0 otherwise

Return type

float

RNA.**last_parameter_file**()

Get the file name of the parameter file that was most recently loaded.

Returns
The file name of the last parameter file, or NULL if parameters are still at defaults

Return type
`string`

RNA.**log_cb_add**(*PyObject * callback*, *PyObject * data=Py_None*, *PyObject * data_release=Py_None*, *vrna_log_levels_e level=VRNA_LOG_LEVEL_WARNING*) → unsigned int

RNA.**log_cb_add_pycallback**(*PyFunc*, *data*, *PyFuncOrNone*, *level*)

RNA.**log_cb_num**()

Get the current number of log message callbacks.

Returns
The current number of log message callbacks stored in the logging system

Return type
`size()`

RNA.**log_fp**()

Get the output file pointer for the default logging system.

Returns
The file pointer where the default logging system will print log messages to

Return type
`FILE *`

RNA.**log_fp_set**(*fp*)

Set the output file pointer for the default logging system.

Parameters
fp (`FILE *`) – The file pointer where the default logging system should print log messages to

RNA.**log_level**()

Get the current default log level.

Returns
The current default log level

Return type
`RNA.log_levels`

See also

RNA.log_level_set, `RNA.log_levels`

RNA.**log_level_set**(*level*)

Set the default log level.

Set the log level for the default log output system. Any user-defined log callback mechanism will not be affected…

Parameters
level (`RNA.log_levels`) – The new log level for the default logging system

Returns
The (updated) log level of the default logging system

Return type
int

See also

RNA.log_level, RNA.log_levels, *RNA.log_cb_add*, *RNA.log_reset*

RNA.**log_options**()

Get the current log options of the default logging system.

Returns
The current options for the default logging system

Return type
unsigned int

See also

RNA.log_options_set, RNA.LOG_OPTION_QUIET, RNA.LOG_OPTION_TRACE_TIME, RNA.LOG_OPTION_TRACE_CALL, RNA.LOG_OPTION_TRACE_FUNC, RNA.LOG_OPTION_DEFAULT

RNA.**log_options_set**(*options*)

Set the log options for the default logging system.

Parameters
options (unsigned int) – The new options for the default logging system

See also

RNA.log_options, RNA.LOG_OPTION_QUIET, RNA.LOG_OPTION_TRACE_TIMERNA.LOG_OPTION_TRACE_CALL, RNA.LOG_OPTION_TRACE_FUNC, RNA.LOG_OPTION_DEFAULT

RNA.**log_reset**()

Reset the logging system.

This resets the logging system and restores default settings

RNA.**loop_energy**(*ptable*, *s*, *s1*, *i*)

Calculate energy of a loop.

Deprecated since version 2.8.0: Use RNA.fold_compound.eval_loop_pt() instead!

Parameters

- **ptable** (list-like(int)) – the pair table of the secondary structure
- **s** (list-like(int)) – encoded RNA sequence
- **s1** (list-like(int)) – encoded RNA sequence
- **i** (int) – position of covering base pair

Returns
free energy of the loop in 10cal/mol

Return type
int

See also

RNA.fold_compound.eval_loop_pt

RNA.**loopidx_from_ptable**(*IntVector pt*) → *IntVector*

RNA.**loopidx_from_ptable**(*varArrayShort pt*) → *varArrayInt*

Get a loop index representation of a structure.

RNA.**make_loop_index**(*structure*)

RNA.**make_tree**(*struc*)

Constructs a Tree (essentially the postorder list) of the structure 'struc', for use in tree_edit_distance().

Parameters

struc (string) – may be any rooted structure representation.

Return type

Tree *

RNA.**maximum_matching**(*sequence*)

SWIG Wrapper Notes

This function is available as global function *maximum_matching()*. See e.g. *RNA.maximum_matching()* in the *The RNA Python module*.

class RNA.**md**(*md self*, *double temperature=vrna_md_defaults_temperature_get()*, *double betaScale=vrna_md_defaults_betaScale_get()*, *int pf_smooth=vrna_md_defaults_pf_smooth_get()*, *int dangles=vrna_md_defaults_dangles_get()*, *int special_hp=vrna_md_defaults_special_hp_get()*, *int noLP=vrna_md_defaults_noLP_get()*, *int noGU=vrna_md_defaults_noGU_get()*, *int noGUclosure=vrna_md_defaults_noGUclosure_get()*, *int logML=vrna_md_defaults_logML_get()*, *int circ=vrna_md_defaults_circ_get()*, *int circ_penalty=vrna_md_defaults_circ_penalty_get()*, *int gquad=vrna_md_defaults_gquad_get()*, *int uniq_ML=vrna_md_defaults_uniq_ML_get()*, *int energy_set=vrna_md_defaults_energy_set_get()*, *int backtrack=vrna_md_defaults_backtrack_get()*, *char backtrack_type=vrna_md_defaults_backtrack_type_get()*, *int compute_bpp=vrna_md_defaults_compute_bpp_get()*, *int max_bp_span=vrna_md_defaults_max_bp_span_get()*, *int min_loop_size=vrna_md_defaults_min_loop_size_get()*, *int window_size=vrna_md_defaults_window_size_get()*, *int oldAliEn=vrna_md_defaults_oldAliEn_get()*, *int ribo=vrna_md_defaults_ribo_get()*, *double cv_fact=vrna_md_defaults_cv_fact_get()*, *double nc_fact=vrna_md_defaults_nc_fact_get()*, *double sfact=vrna_md_defaults_sfact_get()*, *double salt=vrna_md_defaults_salt_get()*, *int saltMLLower=vrna_md_defaults_saltMLLower_get()*, *int saltMLUpper=vrna_md_defaults_saltMLUpper_get()*, *int saltDPXInit=vrna_md_defaults_saltDPXInit_get()*, *float saltDPXInitFact=vrna_md_defaults_saltDPXInitFact_get()*, *float helical_rise=vrna_md_defaults_helical_rise_get()*, *float backbone_length=vrna_md_defaults_backbone_length_get()*) → *md*

Bases: object

The data structure that contains the complete model details used throughout the calculations.

For convenience reasons, we provide the type name RNA.md() to address this data structure without the use of the struct keyword

See also

RNA.md.reset, set_model_details, RNA.md_update, *RNA.md*

This data structure is wrapped as an object *md* with multiple related functions attached as methods. A new set of default parameters can be obtained by calling the constructure of *md: * md()*– Initialize with default settings The resulting object has a list of attached methods which directly correspond to functions that mainly operate on the corresponding *C* data structure: * *reset()* - RNA.md.reset() * *set_from_globals()* - set_model_details() * *option_string()* - RNA.md.option_string()

temperature

The temperature used to scale the thermodynamic parameters.

Type
double

betaScale

A scaling factor for the thermodynamic temperature of the Boltzmann factors.

Type
double

pf_smooth

A flat specifying whether energies in Boltzmann factors need to be smoothed.

Type
int

dangles

Specifies the dangle model used in any energy evaluation (0,1,2 or 3).

If set to 0 no stabilizing energies are assigned to bases adjacent to helices in free ends and multiloops (so called dangling ends). Normally (dangles = 1) dangling end energies are assigned only to unpaired bases and a base cannot participate simultaneously in two dangling ends. In the partition function algorithm RNA.fold_compound.pf() these checks are neglected. To provide comparability between free energy minimization and partition function algorithms, the default setting is 2. This treatment of dangling ends gives more favorable energies to helices directly adjacent to one another, which can be beneficial since such helices often do engage in stabilizing interactions through co-axial stacking. If set to 3 co-axial stacking is explicitly included for adjacent helices in multiloops. The option affects only mfe folding and energy evaluation (RNA.mfe() and RNA.eval_structure()), as well as suboptimal folding (RNA.subopt()) via re-evaluation of energies. Co-axial stacking with one intervening mismatch is not considered so far. Note, that some function do not implement all dangle model but only a subset of (0,1,2,3). In particular, partition function algorithms can only handle 0 and 2. Read the documentation of the particular recurrences or energy evaluation function for information about the provided dangle model.

Type
int

special_hp

Include special hairpin contributions for tri, tetra and hexaloops.

Type
int

noLP

Only consider canonical structures, i.e. no 'lonely' base pairs.

Type
int

noGU

Do not allow GU pairs.

Type
int

noGUclosure

Do not allow loops to be closed by GU pair.

Type
int

logML

Use logarithmic scaling for multiloops.

Type
int

circ

Assume RNA to be circular instead of linear.

Type
int

circ_penalty

Add an entropic penalty to the unpaired circRNA chain.

Type
int

gquad

Include G-quadruplexes in structure prediction.

Type
int

uniq_ML

Flag to ensure unique multi-branch loop decomposition during folding.

Type
int

energy_set

Specifies the energy set that defines set of compatible base pairs.

Type
int

backtrack

Specifies whether or not secondary structures should be backtraced.

Type
int

backtrack_type

Specifies in which matrix to backtrack.

Type
char

compute_bpp

Specifies whether or not backward recursions for base pair probability (bpp) computation will be performed.

Type
int

nonstandards

contains allowed non standard bases

Type
char

max_bp_span

maximum allowed base pair span

Type
int

min_loop_size

Minimum size of hairpin loops.

The default value for this field is TURN, however, it may be 0 in cofolding context.

Type
int

window_size

Size of the sliding window for locally optimal structure prediction.

Type
int

oldAliEn

Use old alifold energy model.

Type
int

ribo

Use ribosum scoring table in alifold energy model.

Type
int

cv_fact

Co-variance scaling factor for consensus structure prediction.

Type
double

nc_fact

Scaling factor to weight co-variance contributions of non-canonical pairs.

Type
double

sfact

Scaling factor for partition function scaling.

Type
double

rtype

Reverse base pair type array.

Type
int

alias

alias of an integer nucleotide representation

Type
short

pair

Integer representation of a base pair.

Type
int

pair_dist

Base pair dissimilarity, a.k.a. distance matrix.

Type
float

salt

Salt (monovalent) concentration (M) in buffer.

Type
double

saltMLLower

Lower bound of multiloop size to use in loop salt correction linear fitting.

Type
int

saltMLUpper

Upper bound of multiloop size to use in loop salt correction linear fitting.

Type
int

saltDPXInit

User-provided salt correction for duplex initialization (in dcal/mol). If set to 99999 the default salt correction is used. If set to 0 there is no salt correction for duplex initialization.

Type
int

saltDPXInitFact

Type
float

helical_rise

Type
float

backbone_length

Type
float

circ_alpha0

Type
double

Parameters

- **temperature** (double) – The temperature in degree C used to scale the thermodynamic parameters
- **betaScale** (double) – A scaling factor for the thermodynamic temperature of the Boltzmann factors.
- **pf_smooth** (int) – A flag specifying whether energies in Boltzmann factors need to be smoothed.

- **dangles** (int) – Specifies the dangle model used in any energy evaluation (0, 1, 2, or 3)
- **special_hp** (int) – Include special hairpin contributions for tri, tetra and hexaloops.
- **noLP** (int) – Only consider canonical structures, i.e. no 'lonely' base pairs.
- **noGU** (int) – Do not allow GU pairs.
- **noGUclosure** (int) – Do not allow loops to be closed by GU pair.
- **logML** (int) – Use logarithmic scaling for multiloops.
- **circ** (int) – Assume RNA to be circular instead of linear.
- **circ_penalty** (int) – Add entropic penalty for unpaired chain in circular RNAs.
- **gquad** (int) – Include G-quadruplexes in structure prediction.
- **uniq_ML** (int) – Flag to ensure unique multi-branch loop decomposition during folding.
- **energy_set** (int) – Specifies the energy set that defines set of compatible base pairs.
- **backtrack** (int) – Specifies whether or not secondary structures should be backtraced.
- **backtrack_type** (char) – Specifies in which matrix to backtrack.
- **compute_bpp** (int) – Specifies whether or not backward recursions for base pair probability (bpp) computation will be performed.
- **max_bp_span** (int) – maximum allowed base pair span
- **min_loop_size** (int) – Minimum size of hairpin loops.
- **window_size** (int) – Size of the sliding window for locally optimal structure prediction.
- **oldAliEn** (int) – Use old alifold energy model.
- **ribo** (int) – Use ribosum scoring table in alifold energy model.
- **cv_fact** (double) – Co-variance scaling factor for consensus structure prediction.
- **nc_fact** (double) – Scaling factor to weight co-variance contributions of non-canonical pairs.
- **sfact** (double) – Scaling factor for partition function scaling.
- **salt** (double) – Salt (monovalent) concentration (M) in buffer.
- **saltMLLower** (int) – Lower bound of multiloop size to use in loop salt correction linear fitting.
- **saltMLUpper** (int) – Upper bound of multiloop size to use in loop salt correction linear fitting.
- **saltDPXInit** (int) – User-provided salt correction for duplex initialization (in dcal/mol).
- **saltDPXInitFact** (float)
- **helical_rise** (float)
- **backbone_length** (float)

Note

Default parameters can be modified by directly setting any of the following global variables (accessible as *RNA.cvar.variable* where *variable* is the variable name. Internally, getting/setting default parameters using their global variable representative translates into calls of the corresponding getter and setter functions, which consequently have not been wrapped directly in the scripting language interface(s):

global variable	*C getter*	*C setter*
temperature	*vrna_md_defaults_temperature_get()*	*vrna_md_defaults_temperature()*
dangles	*vrna_md_defaults_dangles_get()*	*vrna_md_defaults_dangles()*
betaScale	*vrna_md_defaults_betaScale_get()*	*vrna_md_defaults_betaScale()*
tetra_loop	this is an alias of variable *special_hp*	
special_hp	*vrna_md_defaults_special_hp_get()*	*vrna_md_defaults_special_hp()*
noLonelyPairs	this is an alias of variable *noLP*	
noLP	*vrna_md_defaults_noLP_get()*	*vrna_md_defaults_noLP()*
noGU	*vrna_md_defaults_noGU_get()*	*vrna_md_defaults_noGU()*
no_closingGU	this is an alias of variable *noGUclosure*	
noGUclosure	*vrna_md_defaults_noGUclosure_get()*	*vrna_md_defaults_noGUclosure()*
logML	*vrna_md_defaults_logML_get()*	*vrna_md_defaults_logML()*
circ	*vrna_md_defaults_circ_get()*	*vrna_md_defaults_circ()*
gquad	*vrna_md_defaults_gquad_get()*	*vrna_md_defaults_gquad()*
uniq_ML	*vrna_md_defaults_uniq_ML_get()*	*vrna_md_defaults_uniq_ML()*
energy_set	*vrna_md_defaults_energy_set_get()*	*vrna_md_defaults_energy_set()*
backtrack	*vrna_md_defaults_backtrack_get()*	*vrna_md_defaults_backtrack()*
backtrack_type	*vrna_md_defaults_backtrack_type_ge*	*vrna_md_defaults_backtrack_type()*
do_backtrack	this is an alias of variable *compute_bpp*	
compute_bpp	*vrna_md_defaults_compute_bpp_get()*	*vrna_md_defaults_compute_bpp()*
max_bp_span	*vrna_md_defaults_max_bp_span_get()*	*vrna_md_defaults_max_bp_span()*
min_loop_size	*vrna_md_defaults_min_loop_size_ge*	*vrna_md_defaults_min_loop_size()*
window_size	*vrna_md_defaults_window_size_get()*	*vrna_md_defaults_window_size()*
oldAliEn	*vrna_md_defaults_oldAliEn_get()*	*vrna_md_defaults_oldAliEn()*
ribo	*vrna_md_defaults_ribo_get()*	*vrna_md_defaults_ribo()*
cv_fact	*vrna_md_defaults_cv_fact_get()*	*vrna_md_defaults_cv_fact()*
nc_fact	*vrna_md_defaults_nc_fact_get()*	*vrna_md_defaults_nc_fact()*
sfact	*vrna_md_defaults_sfact_get()*	*vrna_md_defaults_sfact()*

property alias

property backbone_length

property backtrack

property backtrack_type

property betaScale

property circ

property circ_alpha0

property circ_penalty

property compute_bpp

property cv_fact

property dangles

property energy_set

property gquad

property helical_rise

property logML

property max_bp_span

property min_loop_size

property nc_fact

property noGU

property noGUclosure

property noLP

property nonstandards

property oldAliEn

option_string()

Get a corresponding commandline parameter string of the options in a RNA.md().

> **Note**
>
> This function is not threadsafe!

property pair

property pf_smooth

reset()

Apply default model details to a provided RNA.md() data structure.

Use this function to initialize a RNA.md() data structure with its default values

property ribo

property rtype

property salt

property saltDPXInit

property saltDPXInitFact

property saltMLLower

property saltMLUpper

set_from_globals()

property sfact

property special_hp

property temperature

property thisown

The membership flag

property uniq_ML

property window_size

RNA.**mean_bp_dist**(*length*)

get the mean pair distance of ensemble

Deprecated since version 2.8.0: This function is not threadsafe and should not be used anymore. Use mean_bp_distance() instead!

RNA.**mean_bp_distance**(*length*)

Get the mean base pair distance of the last partition function computation.

Deprecated since version 2.8.0: Use RNA.fold_compound.mean_bp_distance() or RNA.mean_bp_distance_pr() instead!

Parameters
length (int)

Returns
mean base pair distance in thermodynamic ensemble

Return type
double

See also

RNA.fold_compound.mean_bp_distance, *RNA.mean_bp_distance_pr*

RNA.**mean_bp_distance_pr**(*length*, *pr*)

Get the mean base pair distance in the thermodynamic ensemble.

This is a threadsafe implementation of mean_bp_dist() !

$< d >= \sum_{a,b} p_a p_b d(S_a, S_b)$ this can be computed from the pair probs $p_i j$ as $< d >= \sum_{ij} p_{ij}(1 - p_{ij})$

Deprecated since version 2.8.0: Use RNA.fold_compound.mean_bp_distance() or RNA.mean_bp_distance_pr() instead!

Parameters

- **length** (int) – The length of the sequence
- **pr** (list-like(double)) – The matrix containing the base pair probabilities

Returns
The mean pair distance of the structure ensemble

Return type
double

RNA.**memmove**(*data*, *indata*)

class RNA.**move**(*pos_5=0*, *pos_3=0*)

Bases: object

An atomic representation of the transition / move from one structure to its neighbor.

An atomic transition / move may be one of the following:

- a **base pair insertion**,
- a **base pair removal**, or
- a **base pair shift** where an existing base pair changes one of its pairing partner.

These moves are encoded by two integer values that represent the affected 5' and 3' nucleotide positions. Furthermore, we use the following convention on the signedness of these encodings:

- both values are positive for *insertion moves*

- both values are negative for *base pair removals*
- both values have different signedness for *shift moves*, where the positive value indicates the nucleotide that stays constant, and the others absolute value is the new pairing partner

Note

A value of 0 in either field is used as list-end indicator and doesn't represent any valid move.

pos_5

The (absolute value of the) 5' position of a base pair, or any position of a shifted pair.

Type
`int`

pos_3

The (absolute value of the) 3' position of a base pair, or any position of a shifted pair.

Type
`int`

next

The next base pair (if an elementary move changes more than one base pair), or *NULL* Has to be terminated with move 0,0.

Type
`vrna_move_t *`

An atomic representation of the transition / move from one structure to its neighbor.

An atomic transition / move may be one of the following:

- a **base pair insertion**,
- a **base pair removal**, or
- a **base pair shift** where an existing base pair changes one of its pairing partner.

These moves are encoded by two integer values that represent the affected 5' and 3' nucleotide positions. Furthermore, we use the following convention on the signedness of these encodings:

- both values are positive for *insertion moves*
- both values are negative for *base pair removals*
- both values have different signedness for *shift moves*, where the positive value indicates the nucleotide that stays constant, and the others absolute value is the new pairing partner

Note

A value of 0 in either field is used as list-end indicator and doesn't represent any valid move.

pos_5

The (absolute value of the) 5' position of a base pair, or any position of a shifted pair.

Type
`int`

pos_3

The (absolute value of the) 3' position of a base pair, or any position of a shifted pair.

Type
`int`

next

The next base pair (if an elementary move changes more than one base pair), or *NULL* Has to be terminated with move 0,0.

Type
`vrna_move_t *`

compare(**args*, ***kwargs*)

Compare two moves.

The function compares two moves *m* and *b* and returns whether move *m* is lexicographically smaller (-1), larger (1) or equal to move *b*.

If any of the moves *m* or *b* is a shift move, this comparison only makes sense in a structure context. Thus, the third argument with the current structure must be provided.

Parameters

- **b** (`const RNA.move() *`) – The second move of the comparison
- **pt** (`const short *`) – The pair table of the current structure that is compatible with both moves (maybe NULL if moves are guaranteed to be no shifts)

Returns
-1 if *m* < *b*, 1 if *m* > *b*, 0 otherwise

Return type
`int`

Warning

Currently, shift moves are not supported!

Note

This function returns 0 (equality) upon any error, e.g. missing input

is_insertion()

Test whether a move is a base pair insertion.

Returns
Non-zero if the move is a base pair insertion, 0 otherwise

Return type
`int`

is_removal()

Test whether a move is a base pair removal.

Returns
Non-zero if the move is a base pair removal, 0 otherwise

Return type
`int`

is_shift()

Test whether a move is a base pair shift.

Returns
Non-zero if the move is a base pair shift, 0 otherwise

Return type
`int`

property pos_3

property pos_5

property thisown

The membership flag

RNA.move_standard(*char * seq*, *char * struc*, *enum MOVE_TYPE type*, *int verbosity_level*, *int shifts*, *int noLP*) → char *

class RNA.mx_mfe

Bases: object

property Fc

property FcH

property FcI

property FcM

property c

property f3

property f5

property fM1

property fM2

property fML

property length

property strands

property thisown

The membership flag

property type

class RNA.mx_pf

Bases: object

property expMLbase

property length

property probs

property q

property q1k

property qb

property qho

property qio

property qln

property qm

property qm1

property qm2

property qmo

property qo

property scale

property thisown

The membership flag

property type

RNA.my_PS_rna_plot_snoop_a(*std: :string sequence*, *std: :string structure*, *std: :string filename*, *IntVector relative_access*, *StringVector seqs*) → int

RNA.my_aln_consensus_sequence2(*alignment*, *md_p=None*)

RNA.n_multichoose_k(*n*, *k*)

RNA.naview_xy_coordinates(*std: :string arg1*) → *CoordinateVector*

RNA.new_doubleP(*nelements*)

RNA.new_floatP(*nelements*)

RNA.new_intP(*nelements*)

RNA.new_shortP(*nelements*)

RNA.new_ushortP(*nelements*)

RNA.pack_structure(*char const * s*) → char *

class RNA.param(*model_details=None*)

Bases: object

The datastructure that contains temperature scaled energy parameters.

id

Type
int

stack

Type
int

hairpin

Type
int

bulge

Type
int

internal_loop

Type
int

mismatchExt

Type
int

mismatchI

Type
int

mismatch1nI

Type
int

mismatch23I

Type
int

mismatchH

Type
int

mismatchM

Type
int

dangle5

Type
int

dangle3

Type
int

int11

Type
int

int21

Type
int

int22

Type
int

ninio

Type
int

lxc

Type
double

MLbase

Type
int

MLintern

Type
int

MLclosing

Type
int

TerminalAU

Type
int

DuplexInit

Type
int

Tetraloop_E

Type
int

Tetraloops

Type
char

Triloop_E

Type
int

Triloops

Type
char

Hexaloop_E

Type
int

Hexaloops

Type
char

TripleC

Type
int

MultipleCA

Type
int

MultipleCB

Type
int

gquad

Type
int

gquadLayerMismatch

Type
int

gquadLayerMismatchMax

Type
unsigned int

temperature

Temperature used for loop contribution scaling.

Type
double

model_details

Model details to be used in the recursions.

Type
vrna_md_t

param_file

The filename the parameters were derived from, or empty string if they represent the default.

Type
char

SaltStack

Type
int

SaltLoop

Type
int

SaltLoopDbl

Type
double

SaltMLbase

Type
int

SaltMLintern

Type
int

SaltMLclosing

Type
int

SaltDPXInit

Type
int

The datastructure that contains temperature scaled energy parameters.

id

Type
int

stack

Type
int

hairpin

Type
int

bulge

Type
int

internal_loop

Type
int

mismatchExt

Type
int

mismatchI

Type
int

mismatch1nI

Type
int

mismatch23I

Type
int

mismatchH

Type
int

mismatchM

Type
int

dangle5

Type
int

dangle3

Type
int

int11

Type
int

int21

Type
int

int22

Type
int

ninio

Type
int

lxc

Type
double

MLbase

Type
int

MLintern

Type
int

MLclosing

Type
int

TerminalAU

Type
int

DuplexInit

Type
int

Tetraloop_E

Type
int

Tetraloops

Type
char

Triloop_E

Type
int

Triloops

Type
char

Hexaloop_E

Type
int

Hexaloops

Type
char

TripleC

Type
int

MultipleCA

Type
int

MultipleCB

Type
int

gquad

Type
int

gquadLayerMismatch

Type
int

gquadLayerMismatchMax

Type
unsigned int

temperature

Temperature used for loop contribution scaling.

Type
double

model_details

Model details to be used in the recursions.

Type
vrna_md_t

param_file

The filename the parameters were derived from, or empty string if they represent the default.

Type
char

SaltStack

Type
int

SaltLoop

Type
int

SaltLoopDbl

Type
double

SaltMLbase

Type
int

SaltMLintern

Type
int

SaltMLclosing

Type
int

SaltDPXInit

Type
int

property DuplexInit

property Hexaloop_E

property Hexaloops

property MLbase

property MLclosing

property MLintern

property MultipleCA

property MultipleCB

property SaltDPXInit

property SaltLoop

property SaltLoopDbl

property SaltMLbase

property SaltMLclosing

property SaltMLintern

property SaltStack

property TerminalAU

property Tetraloop_E

property Tetraloops

property Triloop_E

property Triloops

property TripleC

property bulge

property dangle3

property dangle5

property gquad

property gquadLayerMismatch

property gquadLayerMismatchMax

property hairpin

property id

property int11

property int21

property int22

property internal_loop

property lxc

property mismatch1nI

property mismatch23I

property mismatchExt

property mismatchH

property mismatchI

property mismatchM

property model_details

property ninio

property param_file

property stack

property temperature

property thisown

The membership flag

RNA.**params_load**(*std: :string filename = ""*, *unsigned int options*) → int

Load energy parameters from a file.

SWIG Wrapper Notes

This function is available as overloaded function *params_load`(fname="", options=RNA.PARAMETER_FORMAT_DEFAULT). Here, the empty filename string indicates to load default RNA parameters, i.e. this is equivalent to calling RNA.params_load_defaults(). See, e.g. :py:func:`RNA.fold_compound.params_load()* in the *The RNA Python module*.

Parameters

- **fname** (const char) – The path to the file containing the energy parameters
- **options** (unsigned int) – File format bit-mask (usually RNA.PARAMETER_FORMAT_DEFAULT)

Returns

Non-zero on success, 0 on failure

Return type

int

See also

RNA.params_load_from_string, *RNA.params_save*, RNA.params_load_defaults, *RNA.params_load_RNA_Turner2004*, *RNA.params_load_RNA_Turner1999*, *RNA.params_load_RNA_Andronescu2007*, *RNA.params_load_RNA_Langdon2018*, *RNA.params_load_RNA_misc_special_hairpins*, *RNA.params_load_DNA_Mathews2004*, *RNA.params_load_DNA_Mathews1999*

RNA.**params_load_DNA_Mathews1999**()

Load Mathews 1999 DNA energy parameter set.

SWIG Wrapper Notes

This function is available as function *params_load_DNA_Mathews1999()*. See, e.g. *RNA.params_load_DNA_Mathews1999()* in the *The RNA Python module*.

Returns

Non-zero on success, 0 on failure

Return type

int

Warning

This function also resets the default geometric parameters as stored in RNA.md() to those of DNA. Only subsequently initialized RNA.md() structures will be affected by this change.

See also

RNA.params_load, *RNA.params_load_from_string*, *RNA.params_save*, *RNA.params_load_RNA_Turner2004*, *RNA.params_load_RNA_Turner1999*, *RNA.params_load_RNA_Andronescu2007*, *RNA.params_load_RNA_Langdon2018*, *RNA.params_load_RNA_misc_special_hairpins*, *RNA.params_load_DNA_Mathews2004*, RNA.params_load_defaults

RNA.**params_load_DNA_Mathews2004**()

Load Mathews 2004 DNA energy parameter set.

SWIG Wrapper Notes

This function is available as function *params_load_DNA_Mathews2004()*. See, e.g. *RNA.params_load_DNA_Mathews2004()* in the *The RNA Python module*.

Returns
Non-zero on success, 0 on failure

Return type
int

Warning

This function also resets the default geometric parameters as stored in RNA.md() to those of DNA. Only subsequently initialized RNA.md() structures will be affected by this change.

See also

RNA.params_load, *RNA.params_load_from_string*, *RNA.params_save*, *RNA.params_load_RNA_Turner2004*, *RNA.params_load_RNA_Turner1999*, *RNA.params_load_RNA_Andronescu2007*, *RNA.params_load_RNA_Langdon2018*, *RNA.params_load_RNA_misc_special_hairpins*, RNA.params_load_defaults, *RNA.params_load_DNA_Mathews1999*

RNA.**params_load_RNA_Andronescu2007**()

Load Andronsecu 2007 RNA energy parameter set.

SWIG Wrapper Notes

This function is available as function *params_load_RNA_Andronescu2007()*. See, e.g. *RNA.params_load_RNA_Andronescu2007()* in the *The RNA Python module*.

Returns
Non-zero on success, 0 on failure

Return type
int

Warning

This function also resets the default geometric parameters as stored in RNA.md() to those of RNA. Only subsequently initialized RNA.md() structures will be affected by this change.

See also

RNA.params_load, *RNA.params_load_from_string*, *RNA.params_save*, *RNA.params_load_RNA_Turner2004*, *RNA.params_load_RNA_Turner1999*,

RNA.params_load_defaults, *RNA.params_load_RNA_Langdon2018*, *RNA.params_load_RNA_misc_special_hairpins*, *RNA.params_load_DNA_Mathews2004*, *RNA.params_load_DNA_Mathews1999*

RNA.**params_load_RNA_Langdon2018**()

Load Langdon 2018 RNA energy parameter set.

SWIG Wrapper Notes

This function is available as function *params_load_RNA_Langdon2018()*. See, e.g. *RNA.params_load_RNA_Langdon2018()* in the *The RNA Python module*.

Returns
Non-zero on success, 0 on failure

Return type
int

Warning

This function also resets the default geometric parameters as stored in RNA.md() to those of RNA. Only subsequently initialized RNA.md() structures will be affected by this change.

See also

RNA.params_load, *RNA.params_load_from_string*, *RNA.params_save*, *RNA.params_load_RNA_Turner2004*, *RNA.params_load_RNA_Turner1999*, *RNA.params_load_RNA_Andronescu2007*, RNA.params_load_defaults, *RNA.params_load_RNA_misc_special_hairpins*, *RNA.params_load_DNA_Mathews2004*, *RNA.params_load_DNA_Mathews1999*

RNA.**params_load_RNA_Turner1999**()

Load Turner 1999 RNA energy parameter set.

SWIG Wrapper Notes

This function is available as function *params_load_RNA_Turner1999()*. See, e.g. *RNA.params_load_RNA_Turner1999()* in the *The RNA Python module*.

Returns
Non-zero on success, 0 on failure

Return type
int

Warning

This function also resets the default geometric parameters as stored in RNA.md() to those of RNA. Only subsequently initialized RNA.md() structures will be affected by this change.

See also

RNA.params_load, *RNA.params_load_from_string*, *RNA.params_save*, *RNA.params_load_RNA_Turner2004*, RNA.params_load_defaults, *RNA.params_load_RNA_Andronescu2007*, *RNA.params_load_RNA_Langdon2018*, *RNA.params_load_RNA_misc_special_hairpins*, *RNA.params_load_DNA_Mathews2004*, *RNA.params_load_DNA_Mathews1999*

RNA.**params_load_RNA_Turner2004**()

Load Turner 2004 RNA energy parameter set.

SWIG Wrapper Notes

This function is available as function *params_load_RNA_Turner2004()*. See, e.g. *RNA.params_load_RNA_Turner2004()* in the *The RNA Python module*.

Returns
Non-zero on success, 0 on failure

Return type
int

Warning

This function also resets the default geometric parameters as stored in RNA.md() to those of RNA. Only subsequently initialized RNA.md() structures will be affected by this change.

See also

RNA.params_load, *RNA.params_load_from_string*, *RNA.params_save*, RNA.params_load_defaults, *RNA.params_load_RNA_Turner1999*, *RNA.params_load_RNA_Andronescu2007*, *RNA.params_load_RNA_Langdon2018*, *RNA.params_load_RNA_misc_special_hairpins*, *RNA.params_load_DNA_Mathews2004*, *RNA.params_load_DNA_Mathews1999*

RNA.**params_load_RNA_misc_special_hairpins**()

Load Misc Special Hairpin RNA energy parameter set.

SWIG Wrapper Notes

This function is available as function *params_load_RNA_misc_special_hairpins()*. See, e.g. *RNA.params_load_RNA_misc_special_hairpins()* in the *The RNA Python module*.

Returns
Non-zero on success, 0 on failure

Return type
int

Warning

This function also resets the default geometric parameters as stored in RNA.md() to those of RNA. Only subsequently initialized RNA.md() structures will be affected by this change.

See also

RNA.params_load, *RNA.params_load_from_string*, *RNA.params_save*, *RNA.params_load_RNA_Turner2004*, *RNA.params_load_RNA_Turner1999*, *RNA.params_load_RNA_Andronescu2007*, *RNA.params_load_RNA_Langdon2018*, RNA.params_load_defaults, *RNA.params_load_DNA_Mathews2004*, *RNA.params_load_DNA_Mathews1999*

RNA.**params_load_from_string**(*std: :string parameters*, *std: :string name = ""*, *unsigned int options*) → int

Load energy paramters from string.

The string must follow the default energy parameter file convention! The optional *name* argument allows one to specify a name for the parameter set which is stored internally.

SWIG Wrapper Notes

This function is available as overloaded function *params_load_from_string`(string, name="", options=RNA.PARAMETER_FORMAT_DEFAULT). See, e.g. :py:func:`RNA.params_load_from_string()* in the *The RNA Python module*.

Parameters

- **string** (string) – A 0-terminated string containing energy parameters
- **name** (string) – A name for the parameter set in *string* (Maybe *NULL*)
- **options** (unsigned int) – File format bit-mask (usually RNA.PARAMETER_FORMAT_DEFAULT)

Returns

Non-zero on success, 0 on failure

Return type

int

See also

RNA.params_load, *RNA.params_save*, RNA.params_load_defaults, *RNA.params_load_RNA_Turner2004*, *RNA.params_load_RNA_Turner1999*, *RNA.params_load_RNA_Andronescu2007*, *RNA.params_load_RNA_Langdon2018*, *RNA.params_load_RNA_misc_special_hairpins*, *RNA.params_load_DNA_Mathews2004*, *RNA.params_load_DNA_Mathews1999*

RNA.**params_save**(*std: :string filename*, *unsigned int options*) → int

Save energy parameters to a file.

SWIG Wrapper Notes

This function is available as overloaded function *params_save`(fname, options=RNA.PARAMETER_FORMAT_DEFAULT). See, e.g. :py:func:`RNA.params_save()* in the *The RNA Python module*.

Parameters

- **fname** (const char) – A filename (path) for the file where the current energy parameters will be written to
- **options** (unsigned int) – File format bit-mask (usually RNA.PARAMETER_FORMAT_DEFAULT)

Returns
Non-zero on success, 0 on failure

Return type
int

See also

RNA.params_load

RNA.**parse_gquad**(*struc*, *L*, *l*)

Parse a G-Quadruplex from a dot-bracket structure string.

Given a dot-bracket structure (possibly) containing gquads encoded by '+' signs, find first gquad, return end position or 0 if none found Upon return L and l[] contain the number of stacked layers, as well as the lengths of the linker regions. To parse a string with many gquads, call parse_gquad repeatedly e.g. end1 = parse_gquad(struc, &L, l); ... ; end2 = parse_gquad(struc+end1, &L, l); end2+=end1; ... ; end3 = parse_gquad(struc+end2, &L, l); end3+=end2; ... ;

RNA.**parse_structure**(*structure*)

Collects a statistic of structure elements of the full structure in bracket notation.

The function writes to the following global variables: loop_size, loop_degree, helix_size, loops, pairs, unpaired

Parameters
structure (string)

class RNA.**path**(**args*, ***kwargs*)

Bases: object

property en

property move

property s

property thisown
The membership flag

property type

class RNA.**path_options**

Bases: object

property thisown
The membership flag

RNA.**path_options_findpath**(**args*, ***kwargs*)

Create options data structure for findpath direct (re-)folding path heuristic.

This function returns an options data structure that switches the RNA.path_direct() and RNA.fold_compound.path_direct() API functions to use the *findpath* [Flamm *et al.*, 2001] heuristic. The parameter *width* specifies the width of the breadth-first search while the second parameter *type* allows one to set the type of the returned (re-)folding path.

Currently, the following return types are available:

- A list of dot-bracket structures and corresponding free energy (flag: RNA.PATH_TYPE_DOT_BRACKET)
- A list of transition moves and corresponding free energy changes (flag: RNA.PATH_TYPE_MOVES)

SWIG Wrapper Notes

This function is available as overloaded function *path_options_findpath()*. The optional parameter *width* defaults to 10 if omitted, while the optional parameter *type* defaults to RNA.PATH_TYPE_DOT_BRACKET. See, e.g. *RNA.path_options_findpath()* in the *The RNA Python module*.

Parameters

- **width** (int) – Width of the breath-first search strategy
- **type** (unsigned int) – Setting that specifies how the return (re-)folding path should be encoded

Returns

An options data structure with settings for the findpath direct path heuristic

Return type

RNA.path_options()

See also

RNA.PATH_TYPE_DOT_BRACKET, RNA.PATH_TYPE_MOVES, RNA.path_options_free, RNA.path_direct, *RNA.fold_compound.path_direct*

RNA.**pbacktrack**(*sequence*)

Sample a secondary structure from the Boltzmann ensemble according its probability.

Precondition

st_back has to be set to 1 before calling pf_fold() or pf_fold_par() pf_fold_par() or pf_fold() have to be called first to fill the partition function matrices

Parameters

sequence (string) – The RNA sequence

Returns

A sampled secondary structure in dot-bracket notation

Return type

string

RNA.**pbacktrack5**(*sequence*, *length*)

Sample a sub-structure from the Boltzmann ensemble according its probability.

RNA.**pbacktrack_circ**(*sequence*)

Sample a secondary structure of a circular RNA from the Boltzmann ensemble according its probability.

This function does the same as pbacktrack() but assumes the RNA molecule to be circular

Precondition
st_back has to be set to 1 before calling pf_fold() or pf_fold_par() pf_fold_par() or pf_circ_fold() have to be called first to fill the partition function matrices

Deprecated since version 2.8.0: Use RNA.fold_compound.pbacktrack() instead.

Parameters
sequence (`string`) – The RNA sequence

Returns
A sampled secondary structure in dot-bracket notation

Return type
`string`

class RNA.**pbacktrack_mem**

Bases: `object`

property thisown
The membership flag

RNA.**pf_add**(*dG1*, *dG2*, *kT=0*)

RNA.**pf_circ_fold**(**args*)

RNA.**pf_float_precision**()

Find out whether partition function computations are using single precision floating points.

Returns
1 if single precision is used, 0 otherwise

Return type
`int`

See also

`double`

RNA.**pf_fold**(**args*)

RNA.**pfl_fold**(*std: :string sequence*, *int w*, *int L*, *double cutoff*) → *ElemProbVector*

Compute base pair probabilities using a sliding-window approach.

This is a simplified wrapper to RNA.fold_compound.probs_window() that given a nucleid acid sequence, a window size, a maximum base pair span, and a cutoff value computes the pair probabilities for any base pair in any window. The pair probabilities are returned as a list and the user has to take care to free() the memory occupied by the list.

Parameters

- **sequence** (`string`) – The nucleic acid input sequence
- **window_size** (`int`) – The size of the sliding window
- **max_bp_span** (`int`) – The maximum distance along the backbone between two nucleotides that form a base pairs
- **cutoff** (`float`) – A cutoff value that omits all pairs with lower probability

Returns
A list of base pair probabilities, terminated by an entry with RNA.ep().i and RNA.ep().j set to 0

Return type
`RNA.ep() *`

See also

RNA.fold_compound.probs_window, *RNA.pfl_fold_cb*, *RNA.pfl_fold_up*

Note

This function uses default model settings! For custom model settings, we refer to the function RNA.fold_compound.probs_window().

In case of any computation errors, this function returns *NULL*

`RNA.pfl_fold_cb`(*std: :string sequence*, *int window_size*, *int max_bp_span*, *PyObject * PyFunc*, *PyObject * data=Py_None*) → int

`RNA.pfl_fold_up`(*std: :string sequence*, *int ulength*, *int window_size*, *int max_bp_span*) → *DoubleDoubleVector*

Compute probability of contiguous unpaired segments.

This is a simplified wrapper to RNA.fold_compound.probs_window() that given a nucleic acid sequence, a maximum length of unpaired segments (*ulength*), a window size, and a maximum base pair span computes the equilibrium probability of any segment not exceeding *ulength*. The probabilities to be unpaired are returned as a 1-based, 2-dimensional matrix with dimensions $N \times M$, where N is the length of the sequence and M is the maximum segment length. As an example, the probability of a segment of size 5 starting at position 100 is stored in the matrix entry $X[100][5]$.

It is the users responsibility to free the memory occupied by this matrix.

Parameters

- **sequence** (`string`) – The nucleic acid input sequence
- **ulength** (`int`) – The maximal length of an unpaired segment
- **window_size** (`int`) – The size of the sliding window
- **max_bp_span** (`int`) – The maximum distance along the backbone between two nucleotides that form a base pairs

Returns
The probabilities to be unpaired for any segment not exceeding *ulength*

Return type
`list-like(list-like(double))`

Note

This function uses default model settings! For custom model settings, we refer to the function RNA.fold_compound.probs_window().

`RNA.pfl_fold_up_cb`(*std: :string sequence*, *int ulength*, *int window_size*, *int max_bp_span*, *PyObject * PyFunc*, *PyObject * data=Py_None*) → int

RNA.**plist**(*std: :string structure*, *float pr*) → *ElemProbVector*

Create a RNA.ep() from a dot-bracket string.

The dot-bracket string is parsed and for each base pair an entry in the plist is created. The probability of each pair in the list is set by a function parameter.

The end of the plist is marked by sequence positions i as well as j equal to 0. This condition should be used to stop looping over its entries

Parameters

- **struc** (string) – The secondary structure in dot-bracket notation
- **pr** (float) – The probability for each base pair used in the plist

Returns
The plist array

Return type
RNA.ep() *

class RNA.**plot_data**(**args*, ***kwargs*)

Bases: object

property md

property options

property post

property pre

property thisown
The membership flag

RNA.**plot_dp_EPS**(**args*, ***kwargs*)

Produce an encapsulate PostScript (EPS) dot-plot from one or two lists of base pair probabilities.

This function reads two RNA.ep() lists *upper* and *lower* (e.g. base pair probabilities and a secondary structure) and produces an EPS "dot plot" with filename *'filename'* where data from *upper* is placed in the upper-triangular and data from *lower* is placed in the lower triangular part of the matrix.

For default output, provide the flag RNA.PLOT_PROBABILITIES_DEFAULT as *options* parameter.

SWIG Wrapper Notes

This function is available as overloaded function *plot_dp_EPS()* where the last three parameters may be omitted. The default values for these parameters are *lower* = NULL, *auxdata* = NULL, *options* = RNA.PLOT_PROBABILITIES_DEFAULT. See, e.g. *RNA.plot_dp_EPS()* in the *The RNA Python module*.

Parameters

- **filename** (string) – A filename for the EPS output
- **sequence** (string) – The RNA sequence
- **upper** (RNA.ep() *) – The base pair probabilities for the upper triangular part
- **lower** (RNA.ep() *) – The base pair probabilities for the lower triangular part
- **options** (unsigned int) – Options indicating which of the input data should be included in the dot-plot

Returns
1 if EPS file was successfully written, 0 otherwise

Return type
int

See also

RNA.plist, *RNA.fold_compound.plist_from_probs*, RNA.PLOT_PROBABILITIES_DEFAULT

class RNA.**plot_layout**(**args*, ***kwargs*)

Bases: object

property thisown

The membership flag

RNA.**plot_layout_circular**(*structure*)

Create a layout (coordinates, etc.) for a *circular* secondary structure plot.

This function basically is a wrapper to RNA.plot_layout() that passes the `plot_type`RNA.PLOT_TYPE_CIRCULAR.

Parameters
structure (string) – The secondary structure in dot-bracket notation

Returns
The layout data structure for the provided secondary structure

Return type
RNA.plot_layout() *

See also

RNA.plot_layout_free, *RNA.plot_layout*, *RNA.plot_layout_naview*, *RNA.plot_layout_simple*, *RNA.plot_layout_turtle*, *RNA.plot_layout_puzzler*, RNA.plot_coords_circular, RNA.file_PS_rnaplot_layout

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using RNA.plot_coords_circular() instead!

RNA.**plot_layout_naview**(*structure*)

RNA.**plot_layout_puzzler**(*structure*, *options*)

Create a layout (coordinates, etc.) for a secondary structure plot using the *RNApuzzler Algorithm* [Wiegreffe *et al.*, 2019] .

This function basically is a wrapper to RNA.plot_layout() that passes the `plot_type`RNA.PLOT_TYPE_PUZZLER.

Parameters
structure (string) – The secondary structure in dot-bracket notation

Returns
The layout data structure for the provided secondary structure

Return type
RNA.plot_layout() *

See also

RNA.plot_layout_free, *RNA.plot_layout*, *RNA.plot_layout_simple*, *RNA.plot_layout_circular*, *RNA.plot_layout_naview*, *RNA.plot_layout_turtle*, RNA.plot_coords_puzzler, RNA.file_PS_rnaplot_layout

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using RNA.plot_coords_puzzler() instead!

RNA.**plot_layout_simple**(*structure*)

Create a layout (coordinates, etc.) for a *simple* secondary structure plot.

This function basically is a wrapper to RNA.plot_layout() that passes the `plot_type`RNA.PLOT_TYPE_SIMPLE.

Parameters
structure (string) – The secondary structure in dot-bracket notation

Returns
The layout data structure for the provided secondary structure

Return type
RNA.plot_layout() *

See also

RNA.plot_layout_free, *RNA.plot_layout*, *RNA.plot_layout_naview*, *RNA.plot_layout_circular*, *RNA.plot_layout_turtle*, *RNA.plot_layout_puzzler*, RNA.plot_coords_simple, RNA.file_PS_rnaplot_layout

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using RNA.plot_coords_simple() instead!

RNA.**plot_layout_turtle**(*structure*)

Create a layout (coordinates, etc.) for a secondary structure plot using the *Turtle Algorithm* [Wiegreffe *et al.*, 2019] .

This function basically is a wrapper to RNA.plot_layout() that passes the `plot_type`RNA.PLOT_TYPE_TURTLE.

Parameters
structure (string) – The secondary structure in dot-bracket notation

Returns
The layout data structure for the provided secondary structure

Return type
RNA.plot_layout() *

See also

RNA.plot_layout_free, *RNA.plot_layout*, *RNA.plot_layout_simple*, *RNA.plot_layout_circular*, *RNA.plot_layout_naview*, *RNA.plot_layout_puzzler*, RNA.plot_coords_turtle, RNA.file_PS_rnaplot_layout

Note

If only X-Y coordinates of the corresponding structure layout are required, consider using RNA.plot_coords_turtle() instead!

class RNA.**plot_options_puzzler**(**args*, ***kwargs*)

Bases: object

Options data structure for RNApuzzler algorithm implementation.

drawArcs

Type
short

paired

Type
double

unpaired

Type
double

checkAncestorIntersections

Type
short

checkSiblingIntersections

Type
short

checkExteriorIntersections

Type
short

allowFlipping

Type
short

optimize

Type
short

maximumNumberOfConfigChangesAllowed

Type
int

config

Type
string

filename

Type
string

numberOfChangesAppliedToConfig

Type
int

psNumber

Type
int

property allowFlipping

property checkAncestorIntersections

property checkExteriorIntersections

property checkSiblingIntersections

property optimize

plot_options_puzzler()

Options data structure for RNApuzzler algorithm implementation.

drawArcs

Type
short

paired

Type
double

unpaired

Type
double

checkAncestorIntersections

Type
short

checkSiblingIntersections

Type
short

checkExteriorIntersections

Type
short

allowFlipping

Type
short

optimize

Type
short

maximumNumberOfConfigChangesAllowed

Type
int

config

Type
string

filename

Type
string

numberOfChangesAppliedToConfig

Type
int

psNumber

Type
int

property thisown

The membership flag

RNA.**plot_structure**(*std: :string filename*, *std: :string sequence*, *std: :string structure*, *unsigned int file_format*, *plot_layout layout=None*, *plot_data data=None*) → int

RNA.**plot_structure_eps**(*std: :string filename*, *std: :string sequence*, *std: :string structure*, *plot_layout layout=None*, *plot_data data=None*) → int

RNA.**plot_structure_gml**(*std: :string filename*, *std: :string sequence*, *std: :string structure*, *plot_layout layout=None*, *plot_data data=None*, *char option='x'*) → int

RNA.**plot_structure_ssv**(*std: :string filename*, *std: :string sequence*, *std: :string structure*, *plot_layout layout=None*, *plot_data data=None*) → int

RNA.**plot_structure_svg**(*std: :string filename*, *std: :string sequence*, *std: :string structure*, *plot_layout layout=None*, *plot_data data=None*) → int

RNA.**plot_structure_xrna**(*std: :string filename*, *std: :string sequence*, *std: :string structure*, *plot_layout layout=None*, *plot_data data=None*) → int

RNA.**print_bppm**(*T*)

print string representation of probability profile

RNA.**print_tree**(*t*)

Print a tree (mainly for debugging).

class RNA.**probing_data**(*self*, *reactivities*, *m*, *b*, *transform_f=Py_None*, *transform_data=Py_None*, *data_free_f=Py_None*) → *probing_data*

class RNA.**probing_data**(*self*, *reactivities*, *beta*, *pr_conversion*, *pr_default*) → *probing_data*

class RNA.**probing_data**(*self*, *reactivities*, *temperature*, *options*, *unpaired_data={}*, *paired_data={}*, *transform_f=Py_None*, *transform_data=Py_None*, *data_free_f=Py_None*) → *probing_data*

class RNA.**probing_data**(*probing_data self*, *DoubleVector reactivities*, *double m*, *double b*) → *probing_data*

class RNA.**probing_data**(*probing_data self*, *DoubleDoubleVector reactivities*, *DoubleVector ms*, *DoubleVector bs*) → *probing_data*

class RNA.**probing_data**(*probing_data self*, *DoubleDoubleVector reactivities*, *DoubleVector betas*, *StringVector pr_conversions=std::vector< std::string >()*, *DoubleVector pr_defaults=std::vector< double >()*) → *probing_data*

class RNA.**probing_data**(*probing_data self*, *DoubleVector reactivities*, *double temperature*, *unsigned char options*, *DoubleVector unpaired_data={}*, *DoubleVector paired_data={}*) → *probing_data*

Bases: object

The *probing_data()* constructor can be invoked in different ways. Depending on the input parameters, the object returned will invoke particular probing data integration methods either for single sequences or multiple sequence alignments. Hence, it subsumes and provides an easy interface for the functions *probing_data_deigan()*, *probing_data_deigan_comparative()*, etc.

When multiple sets of probing data are supplied, the constructor assumes preparations for multiple sequence alignments (MSAs). As a consequence, the parameters for the conversion methods must be supplied as lists of parameters, one for each sequence.

Parameters

- **reactivities** (list(double) or list(list(double))) – single sequence probing data (1-based) or multiple sequence probing data (0-based list for each sequence of 1-based data)
- **m** (double) – slope for the Deigan et al. 2009 method
- **b** (double) – intercept for the Deigan et al. 2009 method
- **ms** (list(double)) – multiple slopes for the Deigan et al. 2009 method (0-based, one for each sequence)
- **bs** (list(double)) – multiple intercepts for the Deigan et al. 2009 method (0-based, one for each sequence)
- **beta** (double) – scaling factor for Zarringhalam et al. 2012 method
- **betas** (double) – multiple scaling factors for Zarringhalam et al. 2012 method (0-based, one for each sequence)
- **pr_conversion** (string) – probing data to conversion strategy
- **pr_conversions** (list(string)) – multiple probing data to conversion strategies (0-based, one for each sequence)
- **pr_default** (double) – default probability for a nucleotide where reactivity data is missing for
- **pr_defaults** (list(double)) – list of default probabilities for a nucleotide where reactivity data is missing for (0-based, one for each sequence)

probing_data_linear_energies(*pos*, *fc*, *target*)

probing_data_linear_num()

probing_data_linear_raw(*pos*)

probing_data_linear_weight(*pos*)

property thisown

The membership flag

RNA.**probing_data_deigan**(*DoubleVector reactivities*, *double m*, *double b*) → *probing_data*

RNA.**probing_data_deigan_comparative**(*DoubleDoubleVector reactivities*, *DoubleVector ms*, *DoubleVector bs*, *unsigned int multi_params*) → *probing_data*

RNA.**probing_data_eddy**(*DoubleVector reactivities*, *double temperature*, *unsigned char options*, *DoubleVector unpaired_data={}*, *DoubleVector paired_data={}*) → *probing_data*

RNA.**probing_data_eddy_comparative**(*DoubleDoubleVector reactivities*, *double temperature*, *unsigned char options*, *DoubleDoubleVector unpaired_data={}*, *DoubleDoubleVector paired_data={}*, *unsigned int multi_params*) → *probing_data*

RNA.**probing_data_free**(*d*)

Free memory occupied by the (prepared) probing data.

See also

RNA.probing_data, *RNA.fold_compound.sc_probing*, *RNA.probing_data_deigan*, *RNA.probing_data_deigan_comparative*, *RNA.probing_data_zarringhalam*, *RNA.probing_data_zarringhalam_comparative*, *RNA.probing_data_eddy*, *RNA.probing_data_eddy_comparative*

RNA.**probing_data_linear**(*DoubleVector data*, *DoubleVector data_weights*, *PyObject * strategy_cb*, *PyObject * strategy_cb_options=Py_None*, *PyObject * strategy_cb_options_free=Py_None*, *unsigned int options*) → *probing_data*

RNA.**probing_data_zarringhalam**(*DoubleVector reactivities*, *double beta*, *std: :string pr_conversion*, *double pr_default*) → *probing_data*

RNA.**probing_data_zarringhalam_comparative**(*DoubleDoubleVector reactivities*, *DoubleVector betas*, *StringVector pr_conversions=std::vector< std::string >()*, *DoubleVector pr_defaults=std::vector< double >()*, *unsigned int multi_params*) → *probing_data*

RNA.**profile_edit_distance**(*T1*, *T2*)

Align the 2 probability profiles T1, T2 .

This is like a Needleman-Wunsch alignment, we should really use affine gap-costs ala Gotoh

RNA.**pt_pk_remove**(*IntVector pt*, *unsigned int options=0*) → *IntVector*

RNA.**pt_pk_remove**(*varArrayShort pt*, *unsigned int options=0*) → *varArrayShort*

Remove pseudo-knots from a pair table.

This function removes pseudo-knots from an input structure by determining the minimum number of base pairs that need to be removed to make the structure pseudo-knot free.

To accomplish that, we use a dynamic programming algorithm similar to the Nussinov maxmimum matching approach.

Parameters

- **ptable** (`const short *`) – Input structure that may include pseudo-knots
- **options** (`unsigned int`)

Returns

The input structure devoid of pseudo-knots

Return type

`list-like(int)`

See also

RNA.db_pk_remove

RNA.**ptable**(*std: :string str*, *unsigned int options*) → *varArrayShort*

Create a pair table for a secondary structure string.

This function takes an input string of a secondary structure annotation in dot-bracket- notation or *Extended Dot-Bracket Notation* , and converts it into a pair table representation.

SWIG Wrapper Notes

This functions is wrapped as overloaded function *ptable()* that takes an optional argument *options* to specify which type of matching brackets should be considered during conversion. The default set is round brackets, i.e. RNA.BRACKETS_RND. See, e.g. *RNA.ptable()* in the *The RNA Python module*.

Parameters

- **structure** (string) – Secondary structure in *Extended Dot-Bracket Notation*
- **options** (unsigned int) – A bitmask to specify which brackets are recognized during conversion to pair table

Returns
A pointer to a new pair table of the provided secondary structure

Return type
list-like(int)

See also

RNA.ptable, *RNA.db_from_ptable*, RNA.db_flatten_to, *RNA.pt_pk_remove*, RNA.BRACKETS_ANG, RNA.BRACKETS_CLY, RNA.BRACKETS_SQR, RNA.BRACKETS_ALPHA, RNA.BRACKETS_DEFAULT, RNA.BRACKETS_ANY

Note

This function also extracts crossing base pairs, i.e. pseudo-knots if more than a single matching bracket type is allowed through the bitmask *options*.

RNA.**ptable_compat_remove**(*varArrayShort pt*, *varArrayShort pt_ref*) → unsigned int

RNA.**ptable_pk**(*std: :string str*) → *IntVector*

Create a pair table of a secondary structure (pseudo-knot version).

Returns a newly allocated table, such that table[i]=j if (i.j) pair or 0 if i is unpaired, table[0] contains the length of the structure.

In contrast to RNA.ptable() this function also recognizes the base pairs denoted by '[' and ']' brackets. Thus, this function behaves like

Parameters
structure (string) – The secondary structure in (extended) dot-bracket notation

Returns
A pointer to the created pair_table

Return type
list-like(int)

See also

RNA.ptable_from_string

RNA.**random_string**(*l*, *symbols*)

Create a random string using characters from a specified symbol set.

Parameters

- **l** (int) – The length of the sequence
- **symbols** (const char) – The symbol set

Returns
A random string of length 'l' containing characters from the symbolset

Return type
string

RNA.**read_parameter_file**(*fname*)

Read energy parameters from a file.

Deprecated since version 2.8.0: Use RNA.params_load() instead!

Parameters
fname (const char) – The path to the file containing the energy parameters

RNA.**read_record**(*header*, *sequence*, *rest*, *options*)

Get a data record from stdin.

Deprecated since version 2.8.0: This function is deprecated! Use RNA.file_fasta_read_record() as a replacment.

RNA.**ribosum_mx**(*double min_identity_pct*, *double max_identity_pct*) → *DoubleDoubleVector*

RNA.**rotational_symmetry**(**args*)

Determine the order of rotational symmetry for a NULL-terminated string of ASCII characters.

The algorithm applies a fast search of the provided string within itself, assuming the end of the string wraps around to connect with it's start. For example, a string of the form *AABAAB* has rotational symmetry of order *2*

If the argument *positions* is not *NULL*, the function stores an array of string start positions for rotational shifts that map the string back onto itself. This array has length of order of rotational symmetry, i.e. the number returned by this function. The first element *positions `[0] always contains a shift value of `0* representing the trivial rotation.

SWIG Wrapper Notes

This function is available as overloaded global function *rotational_symmetry()*. It merges the functionalities of RNA.rotational_symmetry(), RNA.rotational_symmetry_pos(), RNA.rotational_symmetry_num(), and RNA.rotational_symmetry_pos_num(). In contrast to our C-implementation, this function doesn't return the order of rotational symmetry as a single value, but returns a list of cyclic permutation shifts that result in a rotationally symmetric string. The length of the list then determines the order of rotational symmetry. See, e.g. *RNA.rotational_symmetry()* in the *The RNA Python module* .

Parameters

- **string** (string) – A NULL-terminated string of characters
- **positions** (list-like(list-like(unsigned int))) – A pointer to an (undefined) list of alternative string start positions that lead to an identity mapping (may be NULL)

Returns
The order of rotational symmetry

Return type
unsigned int

See also

RNA.rotational_symmetry, RNA.rotational_symmetry_num, RNA.rotational_symmetry_num_pos

Note

Do not forget to release the memory occupied by *positions* after a successful execution of this function.

RNA.**salt_duplex_init**(*md*)

Get salt correction for duplex initialization at a given salt concentration.

Parameters
md (RNA.md() *) – Model details data structure that specfifies salt concentration in buffer (M)

Returns
Rounded correction for duplex initialization in dcal/mol

Return type
int

RNA.**salt_loop**(*L*, *salt*, *T*, *backbonelen*)

Get salt correction for a loop at a given salt concentration and temperature.

Parameters

- **L** (int) – backbone number in loop
- **salt** (double) – salt concentration (M)
- **T** (double) – absolute temperature (K)
- **backbonelen** (double) – Backbone Length, phosphate-to-phosphate distance (typically 6 for RNA, 6.76 for DNA)

Returns
Salt correction for loop in dcal/mol

Return type
double

RNA.**salt_loop_int**(*L*, *salt*, *T*, *backbonelen*)

Get salt correction for a loop at a given salt concentration and temperature.

This functions is same as RNA.salt_loop but returns rounded salt correction in integer

Parameters

- **L** (int) – backbone number in loop
- **salt** (double) – salt concentration (M)
- **T** (double) – absolute temperature (K)
- **backbonelen** (double) – Backbone Length, phosphate-to-phosphate distance (typically 6 for RNA, 6.76 for DNA)

Returns
Rounded salt correction for loop in dcal/mol

Return type
int

See also

RNA.salt_loop

RNA.**salt_ml**(*saltLoop*, *lower*, *upper*, *m*, *b*)

Fit linear function to loop salt correction.

For a given range of loop size (backbone number), we perform a linear fitting on loop salt correction

$$\text{Loop correction} \approx m \cdot L + b.$$

Parameters

- **saltLoop** (double) – List of loop salt correction of size from 1
- **lower** (int) – Define the size lower bound for fitting
- **upper** (int) – Define the size upper bound for fitting
- **m** (int *) – pointer to store the parameter m in fitting result
- **b** (int *) – pointer to store the parameter b in fitting result

See also

RNA.salt_loop

RNA.**salt_stack**(*salt*, *T*, *hrise*)

Get salt correction for a stack at a given salt concentration and temperature.

Parameters

- **salt** (double) – salt concentration (M)
- **T** (double) – absolute temperature (K)
- **hrise** (double) – Helical Rise (typically 2.8 for RNA, 3.4 for DNA)

Returns

Rounded salt correction for stack in dcal/mol

Return type

int

RNA.**sc_add_bt_pycallback**(*vc*, *PyFunc*)

RNA.**sc_add_exp_f_pycallback**(*vc*, *PyFunc*)

RNA.**sc_add_f_pycallback**(*vc*, *callback*)

RNA.**sc_add_pydata**(*vc*, *data*, *callback*)

class RNA.**sc_mod_param**(*json*, *md=None*)

Bases: object

property thisown

The membership flag

RNA.**sc_mod_parameters_free**(*params*)

Release memory occupied by a modified base parameter data structure.

Properly free a RNA.sc_mod_param() data structure

Parameters

params (*RNA.sc_mod_param()*) – The data structure to free

RNA.**sc_mod_read_from_json**(*json*, *md=None*)

Parse and extract energy parameters for a modified base from a JSON string.

SWIG Wrapper Notes

This function is available as an overloaded function *sc_mod_read_from_json()* where the *md* parameter may be omitted and defaults to *NULL*. See, e.g. *RNA.sc_mod_read_from_json()* in the *The RNA Python module* .

Parameters

- **json** (string) – The JSON string containing the specifications of the modified base
- **md** (RNA.md() *) – A model-details data structure (for look-up of canonical base pairs)

Returns

Parameters of the modified base

Return type

RNA.sc_mod_param()

See also

RNA.sc_mod_read_from_jsonfile(), RNA.sc_mod_parameters_free(), RNA.fold_compound.sc_mod(), usage-io- params-modified-bases

RNA.**sc_mod_read_from_jsonfile**(*filename*, *md=None*)

Parse and extract energy parameters for a modified base from a JSON file.

SWIG Wrapper Notes

This function is available as an overloaded function *sc_mod_read_from_jsonfile()* where the *md* parameter may be omitted and defaults to *NULL*. See, e.g. *RNA.sc_mod_read_from_jsonfile()* in the *The RNA Python module* .

Parameters

- **filename** (string) – The JSON file containing the specifications of the modified base
- **md** (RNA.md() *) – A model-details data structure (for look-up of canonical base pairs)

Returns

Parameters of the modified base

Return type

RNA.sc_mod_param()

See also

RNA.sc_mod_read_from_json(), RNA.sc_mod_parameters_free(), RNA.fold_compound.sc_mod(), usage-io- params-modified-bases

RNA.**sc_multi_cb_add_pycallback**(*fc*, *f*, *f_exp*, *data*, *data_prepare*, *data_free*, *decomp_type*)

class RNA.**score**(*TP=0.0*, *TN=0.0*, *FP=0.0*, *FN=0.0*)

Bases: object

property F1

property FDR

property FN

property FNR

property FOR

property FP

property FPR

property MCC

property NPV

property PPV

property TN

property TNR

property TP

property TPR

property thisown

The membership flag

RNA.**seq_encode**(*std: :string sequence*, *md md_p=None*) → *IntVector*

Get a numerical representation of the nucleotide sequence.

Parameters

- **sequence** (`string`) – The input sequence in upper-case letters
- **md** (`const RNA.md() *`) – A pointer to a RNA.md() data structure that specifies the conversion type

Returns

A list of integer encodings for each sequence letter (1-based). Position 0 denotes the length of the list

Return type

`list-like(int)`

RNA.**settype**(*s*)

RNA.**shortP_getitem**(*ary*, *index*)

RNA.**shortP_setitem**(*ary*, *index*, *value*)

RNA.**simple_circplot_coordinates**(*std: :string arg1*) → *CoordinateVector*

RNA.**simple_xy_coordinates**(**args*)

Calculate nucleotide coordinates for secondary structure plot the *Simple way*.

Deprecated since version 2.8.0: Consider switching to RNA.plot_coords_simple_pt() instead!

See also

make_pair_table, rna_plot_type, *simple_circplot_coordinates*, *naview_xy_coordinates*, *RNA.file_PS_rnaplot_a*, *RNA.file_PS_rnaplot*, *svg_rna_plot*

Parameters

- **pair_table** (list-like(int)) – The pair table of the secondary structure
- **X** (list-like(double)) – a pointer to an array with enough allocated space to hold the x coordinates
- **Y** (list-like(double)) – a pointer to an array with enough allocated space to hold the y coordinates

Returns
length of sequence on success, 0 otherwise

Return type
int

RNA.**ssv_rna_plot**(*string*, *structure*, *ssfile*)

Produce a secondary structure graph in SStructView format.

Write coord file for SStructView

Parameters

- **string** (string) – The RNA sequence
- **structure** (string) – The secondary structure in dot-bracket notation
- **ssfile** (string) – The filename of the ssv output

Returns
1 on success, 0 otherwise

Return type
int

RNA.**stackProb**(*cutoff*)

Get the probability of stacks.

Deprecated since version 2.8.0: Use RNA.fold_compound.stack_prob() instead!

RNA.**str_to_dbl_array**(**args*, ***kwargs*)

RNA.**string_edit_distance**(*T1*, *T2*)

Calculate the string edit distance of T1 and T2.

Parameters

- **T1** (swString *)
- **T2** (swString *)

Return type
float

RNA.**strtrim**(*char * seq_mutable*, *char const * delimiters=None*, *unsigned int keep=0*, *unsigned int options*) → unsigned int

Trim a string by removing (multiple) occurences of a particular character.

This function removes (multiple) consecutive occurences of a set of characters (*delimiters*) within an input string. It may be used to remove leading and/or trailing whitespaces or to restrict the maximum number of consecutive occurences of the delimiting characters *delimiters*. Setting *keep=0* removes all occurences, while other values reduce multiple consecutive occurences to at most *keep* delimiters. This might be useful

if one would like to reduce multiple whitespaces to a single one, or to remove empty fields within a comma-separated value string.

The parameter *delimiters* may be a pointer to a 0-terminated char string containing a set of any ASCII character. If *NULL* is passed as delimiter set or an empty char string, all whitespace characters are trimmed. The *options* parameter is a bit vector that specifies which part of the string should undergo trimming. The implementation distinguishes the leading (RNA.TRIM_LEADING), trailing (RNA.TRIM_TRAILING), and in-between (RNA.TRIM_IN_BETWEEN) part with respect to the delimiter set. Combinations of these parts can be specified by using logical-or operator.

The following example code removes all leading and trailing whitespace characters from the input string:

SWIG Wrapper Notes

Since many scripting languages treat strings as immutable objects, this function does not modify the input string directly. Instead, it returns the modified string as second return value, together with the number of removed delimiters.

The scripting language interface provides an overloaded version of this function, with default parameters *delimiters=NULL*, *keep=0*, and *options=RNA.TRIM_DEFAULT*. See, e.g. *RNA.strtrim()* in the *The RNA Python module*.

Parameters

- **string** (string) – The '0'-terminated input string to trim
- **delimiters** (string) – The delimiter characters as 0-terminated char array (or *NULL*)
- **keep** (unsigned int) – The maximum number of consecutive occurences of the delimiter in the output string
- **options** (unsigned int) – The option bit vector specifying the mode of operation

Returns
The number of delimiters removed from the string

Return type
unsigned int

See also

RNA.TRIM_LEADING, RNA.TRIM_TRAILING, RNA.TRIM_IN_BETWEEN, RNA.TRIM_SUBST_BY_FIRST, RNA.TRIM_DEFAULT, RNA.TRIM_ALL

Note

The delimiter always consists of a single character from the set of characters provided. In case of alternative delimiters and non-null *keep* parameter, the first *keep* delimiters are preserved within the string. Use RNA.TRIM_SUBST_BY_FIRST to substitute all remaining delimiting characters with the first from the *delimiters* list.

class RNA.**struct_en**

Bases: object

Data structure for energy_of_move().

energy

Type
int

structure

Type
list-like(int)

Data structure for energy_of_move().

energy

Type
int

structure

Type
list-like(int)

property energy

property structure

property thisown
The membership flag

RNA.**subopt**(**args*)

class RNA.**subopt_solution**
Bases: object

property energy

property structure

property thisown
The membership flag

RNA.**svg_rna_plot**(*string*, *structure*, *ssfile*)
Produce a secondary structure plot in SVG format and write it to a file.

Parameters
- **string** (string) – The RNA sequence
- **structure** (string) – The secondary structure in dot-bracket notation
- **ssfile** (string) – The filename of the svg output

Returns
1 on success, 0 otherwise

Return type
int

RNA.**tree_edit_distance**(*T1*, *T2*)
Calculates the edit distance of the two trees.

Parameters
- **T1** (Tree *)
- **T2** (Tree *)

Return type
float

RNA.**tree_string_to_db**(*structure*)

Convert a linear tree string representation of a secondary structure back to Dot-Bracket notation.

Parameters
tree (string) – A linear tree string representation of a secondary structure

Returns
A dot-bracket notation of the secondary structure provided in *tree*

Return type
string

Warning

This function only accepts *Expanded* and *HIT* tree representations!

See also

RNA.db_to_tree_string(), RNA.STRUCTURE_TREE_EXPANDED, RNA.STRUCTURE_TREE_HIT, structure- tree-representation

RNA.**tree_string_unweight**(*structure*)

Remove weights from a linear string tree representation of a secondary structure.

This function strips the weights of a linear string tree representation such as *HIT*, or Coarse Grained Tree sensu Shapiro [1988]

Parameters
structure (string) – A linear string tree representation of a secondary structure with weights

Returns
A linear string tree representation of a secondary structure without weights

Return type
string

See also

RNA.db_to_tree_string

RNA.**ubf_eval_ext_int_loop**(*i*, *j*, *p*, *q*, *i1*, *j1*, *p1*, *q1*, *si*, *sj*, *sp*, *sq*, *type*, *type_2*, *length*, *P*, *sc*)

RNA.**ubf_eval_int_loop**(*i*, *j*, *p*, *q*, *i1*, *j1*, *p1*, *q1*, *si*, *sj*, *sp*, *sq*, *type*, *type_2*, *rtype*, *ij*, *cp*, *P*, *sc*)

RNA.**ubf_eval_int_loop2**(*i*, *j*, *p*, *q*, *i1*, *j1*, *p1*, *q1*, *si*, *sj*, *sp*, *sq*, *type*, *type_2*, *rtype*, *ij*, *sn*, *ss*, *P*, *sc*)

RNA.**ud_set_exp_prod_cb**(*vc*, *prod_cb*, *eval_cb*)

RNA.**ud_set_prob_cb**(*vc*, *setter*, *getter*)

RNA.**ud_set_prod_cb**(*vc*, *prod_cb*, *eval_cb*)

RNA.**ud_set_pydata**(*vc*, *data*, *PyFuncOrNone*)

RNA.**unexpand_Full**(*ffull*)

Restores the bracket notation from an expanded full or HIT tree, that is any tree using only identifiers ‘U’ ‘P’ and ‘R’.

Parameters
ffull (string)

Return type
string

RNA.**unexpand_aligned_F**(*align*)

Converts two aligned structures in expanded notation.

Takes two aligned structures as produced by tree_edit_distance() function back to bracket notation with '_' as the gap character. The result overwrites the input.

Parameters
align (string)

RNA.**unpack_structure**(*char const * packed*) → char *

RNA.**unweight**(*wcoarse*)

Strip weights from any weighted tree.

Parameters
wcoarse (string)

Return type
string

RNA.**update_co_pf_params**(*length*)

Recalculate energy parameters.

This function recalculates all energy parameters given the current model settings.

Deprecated since version 2.8.0: Use RNA.fold_compound.exp_params_subst() instead!

Parameters
length (int) – Length of the current RNA sequence

RNA.**update_cofold_params**()

Recalculate parameters.

Deprecated since version 2.8.0: See RNA.fold_compound.params_subst() for an alternative using the new API

RNA.**update_fold_params**()

Recalculate energy parameters.

Deprecated since version 2.8.0: For non-default model settings use the new API with RNA.fold_compound.params_subst() and RNA.fold_compound.mfe() instead!

RNA.**update_pf_params**(*length*)

Recalculate energy parameters.

Call this function to recalculate the pair matrix and energy parameters after a change in folding parameters like temperature

Deprecated since version 2.8.0: Use RNA.fold_compound.exp_params_subst() instead

RNA.**urn**()

get a random number from [0..1]

Returns
A random number in range [0..1]

Return type
double

See also

RNA.int_urn, *RNA.init_rand*, RNA.init_rand_seed

Note

Usually implemented by calling *erand48()*.

RNA.**ushortP_getitem**(*ary*, *index*)

RNA.**ushortP_setitem**(*ary*, *index*, *value*)

class RNA.**varArrayChar**(*d*, *type*)

Bases: object

get(*i*)

size()

property thisown

The membership flag

type()

class RNA.**varArrayFLTorDBL**(*d*, *type*)

Bases: object

get(*i*)

size()

property thisown

The membership flag

type()

class RNA.**varArrayInt**(*d*, *type*)

Bases: object

get(*i*)

size()

property thisown

The membership flag

type()

class RNA.**varArrayMove**(*d*, *type*)

Bases: object

get(*i*)

size()

property thisown

The membership flag

type()

class RNA.varArrayShort(*args)

Bases: object

get(i)

size()

property thisown

The membership flag

type()

class RNA.varArrayUChar(d, type)

Bases: object

get(i)

size()

property thisown

The membership flag

type()

class RNA.varArrayUInt(d, type)

Bases: object

get(i)

size()

property thisown

The membership flag

type()

class RNA.var_array_Iterator(var_arr)

Bases: object

next()

RNA.write_parameter_file(fname)

Write energy parameters to a file.

Deprecated since version 2.8.0: Use RNA.params_save() instead!

Parameters

fname (const char) – A filename (path) for the file where the current energy parameters will be written to

RNA.xrna_plot(string, structure, ssfile)

Produce a secondary structure plot for further editing in XRNA.

Parameters

- **string** (string) – The RNA sequence
- **structure** (string) – The secondary structure in dot-bracket notation
- **ssfile** (string) – The filename of the xrna output

Returns

1 on success, 0 otherwise

Return type

int

RNA.**zukersubopt**(*string*)

Compute Zuker type suboptimal structures.

Compute Suboptimal structures according to M. Zuker, i.e. for every possible base pair the minimum energy structure containing the resp. base pair. Returns a list of these structures and their energies.

Deprecated since version 2.8.0: use RNA.zukersubopt() instead

Parameters
string (string) – RNA sequence

Returns
List of zuker suboptimal structures

Return type
SOLUTION *

4.4 Other

4.4.1 Callbacks in RNAlib

Below, you find an enumeration of the available callback functions.

Global *vrna_auxdata_free_f* (void *data)

This callback is supposed to free memory occupied by an auxiliary data structure. It will be called when the *vrna_fold_compound_t* is erased from memory through a call to *vrna_fold_compound_free()* and will be passed the address of memory previously bound to the *vrna_fold_compound_t* via *vrna_fold_compound_add_auxdata()*.

Global *vrna_bs_result_f* (const char *structure, void *data)

This function will be called for each secondary structure that has been successfully backtraced from the partition function DP matrices.

Global *vrna_gr_inside_exp_f* (*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, void *data)

This callback allows for extending the partition function secondary structure decomposition with additional rules.

Global *vrna_gr_inside_f* (*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, void *data)

This callback allows for extending the MFE secondary structure decomposition with additional rules.

Global *vrna_gr_outside_f* (*vrna_fold_compound_t* *fc, unsigned int i, unsigned int j, int e, *vrna_bps_t* bp_stack, *vrna_bts_t* bt_stack, void *data)

This callback allows for backtracking (sub)structures obtained from extending the MFE secondary structure decomposition with additional rules.

Global *vrna_gr_serialize_bp_f* (*vrna_fold_compound_t* *fc, *vrna_bps_t* bp_stack, void *data)

This callback allows for changing the way how base pairs (and other types of data) obtained from the default and extended grammar are converted back into a dot-bracket string.

Global *vrna_hc_eval_f* (int i, int j, int k, int l, unsigned char d, void *data)

This callback enables one to over-rule default hard constraints in secondary structure decompositions.

Global *vrna_heat_capacity_f* (float temp, float heat_capacity, void *data)

This function will be called for each evaluated temperature in the heat capacity prediction.

Global *vrna_math_fun_dbl_f* (double value, *vrna_math_fun_dbl_opt_t* options)

This callback handles mathematical functions working on scalar *double* values.

Global *vrna_mfe_window_f* (unsigned int start, unsigned int end, const char *structure, float en, void *data)

This function will be called for each hit in a sliding window MFE prediction.

Global *vrna_probs_window_f* (*FLT_OR_DBL* *pr, int pr_size, int i, int max, unsigned int type, void *data)

This function will be called for each probability data set in the sliding window probability computation implementation of *vrna_probs_window()*. The argument *type* specifies the type of probability that is passed to this function.

Global *vrna_recursion_status_f* (*vrna_fold_compound_t* *fc, unsigned char status, void *data)

This function will be called to notify a third-party implementation about the status of a currently ongoing recursion. The purpose of this callback mechanism is to provide users with a simple way to ensure pre- and post conditions for auxiliary mechanisms attached to our implementations.

Global *vrna_sc_bt_f* (int i, int j, int k, int l, unsigned char d, void *data)

This callback enables one to add auxiliary base pairs in the backtracking steps of hairpin- and internal loops.

Global *vrna_sc_exp_f* (int i, int j, int k, int l, unsigned char d, void *data)

This callback enables one to add (pseudo-)energy contributions to individual decompositions of the secondary structure (Partition function variant, i.e. contributions must be returned as Boltzmann factors).

Global *vrna_sc_f* (int i, int j, int k, int l, unsigned char d, void *data)

This callback enables one to add (pseudo-)energy contributions to individual decompositions of the secondary structure.

Global *vrna_subopt_result_f* (const char *structure, float energy, void *data)

This function will be called for each suboptimal secondary structure that is successfully backtraced.

Global *vrna_ud_add_probs_f* (*vrna_fold_compound_t* *fc, int i, int j, unsigned int loop_type, *FLT_OR_DBL* exp_energy, void *data)

A callback function to store equilibrium probabilities for the unstructured domain feature

Global *vrna_ud_exp_f* (*vrna_fold_compound_t* *fc, int i, int j, unsigned int loop_type, void *data)

This function will be called to determine the additional energy contribution of a specific unstructured domain, e.g. the binding free energy of some ligand (Partition function variant, i.e. the Boltzmann factors instead of actual free energies).

Global *vrna_ud_exp_production_f* (*vrna_fold_compound_t* *fc, void *data)

The production rule for the unstructured domain grammar extension (Partition function variant)

Global *vrna_ud_f* (*vrna_fold_compound_t* *fc, int i, int j, unsigned int loop_type, void *data)

This function will be called to determine the additional energy contribution of a specific unstructured domain, e.g. the binding free energy of some ligand.

Global *vrna_ud_get_probs_f* (*vrna_fold_compound_t* *fc, int i, int j, unsigned int loop_type, int motif, void *data)

A callback function to retrieve equilibrium probabilities for the unstructured domain feature

Global *vrna_ud_production_f* (*vrna_fold_compound_t* *fc, void *data)

The production rule for the unstructured domain grammar extension

4.4.2 Deprecated List

Global *alifold* (const char **strings, char *structure)

Usage of this function is discouraged! Use *vrna_alifold()*, or *vrna_mfe()* instead!

File alignments.h

Use ViennaRNA/sequences/alignments.h instead

Global alimake_pair_table (const char *structure)

Use *vrna_pt_ali_get()* instead!

Global *alipbacktrack* (double *prob)

Use *vrna_pbacktrack()* instead!

Global *alipf_circ_fold* (const char **sequences, char *structure, *vrna_ep_t* **pl)

Use *vrna_pf()* instead

Global *alipf_fold* (const char **sequences, char *structure, *vrna_ep_t* **pl)

Use *vrna_pf()* instead

Global *alipf_fold_par* (const char **sequences, char *structure, *vrna_ep_t* **pl, *vrna_exp_param_t* *parameters, int calculate_bppm, int is_constrained, int is_circular)

Use *vrna_pf()* instead

Global aliPS_color_aln (const char *structure, const char *filename, const char *seqs[], const char *names[])

Use *vrna_file_PS_aln()* instead!

File aln_util.h

Use ViennaRNA/sequences/alignments.h instead

File alphabet.h

Use ViennaRNA/sequences/alphabet.h instead

Global *assign_plist_from_db* (*vrna_ep_t* **pl, const char *struc, float pr)

Use *vrna_plist()* instead

Global *assign_plist_from_pr* (*vrna_ep_t* **pl, *FLT_OR_DBL* *probs, int length, double cutoff)

Use *vrna_plist_from_probs()* instead!

Global b2C (const char *structure)

See *vrna_db_to_tree_string()* and *VRNA_STRUCTURE_TREE_SHAPIRO_SHORT* for a replacement

Global b2HIT (const char *structure)

See *vrna_db_to_tree_string()* and *VRNA_STRUCTURE_TREE_HIT* for a replacement

Global b2Shapiro (const char *structure)

See *vrna_db_to_tree_string()* and *VRNA_STRUCTURE_TREE_SHAPIRO_WEIGHT* for a replacement

Global base_pair

Do not use this variable anymore!

File boltzmann_sampling.h

Use ViennaRNA/sampling/basic.h instead

Global *bondT*

Use *vrna_bp_stack_t* instead!

Global bp_distance (const char *str1, const char *str2)

Use vrna_bp_distance instead

Global bppm_symbol (const float *x)

Use *vrna_bpp_symbol()* instead!

Global bppm_to_structure (char *structure, *FLT_OR_DBL* *pr, unsigned int length)

Use *vrna_db_from_probs()* instead!

Global centroid (int length, double *dist)

This function is deprecated and should not be used anymore as it is not threadsafe!

File centroid.h

Use ViennaRNA/structures/centroid.h instead

File char_stream.h

Use ViennaRNA/datastructures/char_stream.h instead

Global *circalifold* (const char **strings, char *structure)

Usage of this function is discouraged! Use vrna_alicircfold(), and *vrna_mfe()* instead!

Global *circfold* (const char *sequence, char *structure)

Use *vrna_circfold()*, or *vrna_mfe()* instead!

Global *co_pf_fold* (char *sequence, char *structure)

Global *co_pf_fold_par* (char *sequence, char *structure, *vrna_exp_param_t* *parameters, int calculate_bppm, int is_constrained)

Use *vrna_pf_dimer()* instead!

Global *cofold* (const char *sequence, char *structure)

use *vrna_mfe_dimer()* instead

Global *cofold_par* (const char *string, char *structure, *vrna_param_t* *parameters, int is_constrained)

use *vrna_mfe_dimer()* instead

File combinatorics.h

Use ViennaRNA/combinatorics/basic.h instead

File commands.h

Use ViennaRNA/io/commands.h instead

Global compute_BPdifferences (short *pt1, short *pt2, unsigned int turn)

Use *vrna_refBPdist_matrix()* instead

Global *compute_probabilities* (double FAB, double FEA, double FEB, *vrna_ep_t* *prAB, *vrna_ep_t* *prA, *vrna_ep_t* *prB, int Alength)

Global constrain_ptypes (const char *constraint, unsigned int length, char *ptype, int *BP, int min_loop_size, unsigned int idx_type)

Do not use this function anymore! Structure constraints are now handled through *vrna_hc_t* and related functions.

File constraints.h

Use ViennaRNA/constraints/basic.h instead

File constraints_hard.h

Use ViennaRNA/constraints/hard.h instead

File constraints_ligand.h

Use ViennaRNA/constraints/ligand.h instead

File constraints_SHAPE.h

Use ViennaRNA/probing/SHAPE.h instead

File constraints_soft.h

Use ViennaRNA/constraints/soft.h instead

File convert_epars.h

Use ViennaRNA/params/convert.h instead

Global copy_pair_table (const short *pt)

Use *vrna_ptable_copy()* instead

Global *cpair*

Use *vrna_cpair_t* instead!

Global *cv_fact*

See *vrna_md_t.cv_fact*, and *vrna_mfe()* to avoid using global variables

File data_structures.h

Use ViennaRNA/datastructures/basic.h instead

Global *destroy_TwoDfold_variables* (*TwoDfold_vars* *our_variables)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, and *vrna_fold_compound_free()* instead!

Global destroy_TwoDpfold_variables (TwoDpfold_vars *vars)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_pf_TwoD()*, and *vrna_fold_compound_free()* instead!

File dp_matrices.h

Use ViennaRNA/datastructures/dp_matrices.h instead

Global *E_Stem* (int type, int si1, int sj1, int extLoop, *vrna_param_t* *P)

Please use one of the functions *vrna_E_exterior_stem()* and *vrna_E_multibranch_stem()* instead! Use the former for cases where `extLoop` != 0 and the latter otherwise.

File energy_const.h

Use ViennaRNA/params/constants.h instead

Global *energy_of_alistruct* (const char **sequences, const char *structure, int n_seq, float *energy)

Usage of this function is discouraged! Use *vrna_eval_structure()*, and *vrna_eval_covar_structure()* instead!

Global *energy_of_circ_struct* (const char *string, const char *structure)

This function is deprecated and should not be used in future programs Use *energy_of_circ_structure()* instead!

Global *energy_of_circ_struct_par* (const char *string, const char *structure, *vrna_param_t* *parameters, int verbosity_level)

Use *vrna_eval_structure()* or *vrna_eval_structure_verbose()* instead!

Global *energy_of_circ_structure* (const char *string, const char *structure, int verbosity_level)

Use *vrna_eval_structure()* or *vrna_eval_structure_verbose()* instead!

Global *energy_of_move* (const char *string, const char *structure, int m1, int m2)

Use *vrna_eval_move()* instead!

Global *energy_of_move_pt* (short *pt, short *s, short *s1, int m1, int m2)

Use *vrna_eval_move_pt()* instead!

Global *energy_of_struct* (const char *string, const char *structure)

This function is deprecated and should not be used in future programs! Use *energy_of_structure()* instead!

Global *energy_of_struct_par* (const char *string, const char *structure, *vrna_param_t* *parameters, int verbosity_level)

Use *vrna_eval_structure()* or *vrna_eval_structure_verbose()* instead!

Global *energy_of_struct_pt* (const char *string, short *ptable, short *s, short *s1)

This function is deprecated and should not be used in future programs! Use *energy_of_structure_pt()* instead!

Global *energy_of_struct_pt_par* (const char *string, short *ptable, short *s, short *s1, *vrna_param_t* *parameters, int verbosity_level)

Use *vrna_eval_structure_pt()* or *vrna_eval_structure_pt_verbose()* instead!

Global *energy_of_structure* (const char *string, const char *structure, int verbosity_level)

Use *vrna_eval_structure()* or *vrna_eval_structure_verbose()* instead!

Global *energy_of_structure_pt* (const char *string, short *ptable, short *s, short *s1, int verbosity_level)

Use *vrna_eval_structure_pt()* or *vrna_eval_structure_pt_verbose()* instead!

File energy_par.h

Use ViennaRNA/params/default.h instead

File equilibrium_probs.h

Use ViennaRNA/probabilities/basepairs.h and ViennaRNA/probabilities/structures.h instead

File eval.h

Use ViennaRNA/eval/structures.h instead

Global *exp_E_ExtLoop* (int type, int si1, int sj1, *vrna_exp_param_t* *P)

Use vrna_exp_E_ext_stem() instead!

Global expHairpinEnergy (int u, int type, short si1, short sj1, const char *string)

Use *vrna_exp_E_hairpin()* from loop_energies.h instead

Global expLoopEnergy (int u1, int u2, int type, int type2, short si1, short sj1, short sp1, short sq1)

Use vrna_exp_E_internal() from loop_energies.h instead

Global *export_ali_bppm* (void)

Usage of this function is discouraged! The new *vrna_fold_compound_t* allows direct access to the folding matrices, including the pair probabilities! The pair probability array returned here reflects the one of the latest call to *vrna_pf()*, or any of the old API calls for consensus structure partition function folding.

Global *export_circfold_arrays* (int *Fc_p, int *FcH_p, int *FcI_p, int *FcM_p, int **fM2_p, int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **indx_p, char **ptype_p)

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

Global *export_circfold_arrays_par* (int *Fc_p, int *FcH_p, int *FcI_p, int *FcM_p, int **fM2_p, int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **indx_p, char **ptype_p, *vrna_param_t* **P_p)

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

Global *export_co_bppm* (void)

This function is deprecated and will be removed soon! The base pair probability array is available through the *vrna_fold_compound_t* data structure, and its associated *vrna_mx_pf_t* member.

Global *export_cofold_arrays* (int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **fc_p, int **indx_p, char **ptype_p)

folding matrices now reside within the *vrna_fold_compound_t*. Thus, this function will only work in conjunction with a prior call to the deprecated functions *cofold()* or *cofold_par()*

Global *export_cofold_arrays_gq* (int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **fc_p, int **ggg_p, int **indx_p, char **ptype_p)

folding matrices now reside within the fold compound. Thus, this function will only work in conjunction with a prior call to *cofold()* or *cofold_par()*

Global *export_fold_arrays* (int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **indx_p, char **ptype_p)

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

Global *export_fold_arrays_par* (int **f5_p, int **c_p, int **fML_p, int **fM1_p, int **indx_p, char **ptype_p, *vrna_param_t* **P_p)

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

File exterior_loops.h

Use ViennaRNA/eval/external.h or ViennaRNA/eval/external.h instead

File external.h

Use ViennaRNA/eval/exterior.h, ViennaRNA/mfe/exterior.h, ViennaRNA/backtrack/exterior.h, and ViennaRNA/partfunc/exterior.h instead

File file_formats.h

Use ViennaRNA/io/file_formats.h instead

File file_formats_msa.h

Use ViennaRNA/io/file_formats_msa.h instead

File file_utils.h

Use ViennaRNA/io/utils.h instead

Global filecopy (FILE *from, FILE *to)

Use *vrna_file_copy()* instead!

Global find_saddle (const char *seq, const char *s1, const char *s2, int width)

Use *vrna_path_findpath_saddle()* instead!

File findpath.h

Use ViennaRNA/landscape/findpath.h instead

Global *fold* (const char *sequence, char *structure)

use *vrna_fold()*, or *vrna_mfe()* instead!

Global *fold_par* (const char *sequence, char *structure, *vrna_param_t* *parameters, int is_constrained, int is_circular)

use *vrna_mfe()* instead!

Global *free_alifold_arrays* (void)

Usage of this function is discouraged! It only affects memory being free'd that was allocated by an old API function before. Release of memory occupied by the newly introduced *vrna_fold_compound_t* is handled by *vrna_fold_compound_free()*

Global *free_alipf_arrays* (void)

Usage of this function is discouraged! This function only free's memory allocated by old API function calls. Memory allocated by any of the new API calls (starting with vrna_) will be not affected!

Global *free_arrays* (void)

See *vrna_fold()*, *vrna_circfold()*, or *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

Global *free_co_arrays* (void)

This function will only free memory allocated by a prior call of *cofold()* or *cofold_par()*. See *vrna_mfe_dimer()* for how to use the new API

Global *free_co_pf_arrays* (void)

This function will be removed for the new API soon! See *vrna_pf_dimer()*, *vrna_fold_compound()*, and *vrna_fold_compound_free()* for an alternative

Global free_path (*vrna_path_t* *path)

Use *vrna_path_free()* instead!

Global *free_pf_arrays* (void)

See *vrna_fold_compound_t* and its related functions for how to free memory occupied by the dynamic programming matrices

Global *get_alipf_arrays* (short ***S_p, short ***S5_p, short ***S3_p, unsigned short ***a2s_p, char ***Ss_p, *FLT_OR_DBL* **qb_p, *FLT_OR_DBL* **qm_p, *FLT_OR_DBL* **q1k_p, *FLT_OR_DBL* **qln_p, short **pscore)

It is discouraged to use this function! The new *vrna_fold_compound_t* allows direct access to all necessary consensus structure prediction related variables!

Global *get_boltzmann_factor_copy* (*vrna_exp_param_t* *parameters)

Use *vrna_exp_params_copy()* instead!

Global *get_boltzmann_factors* (double temperature, double betaScale, *vrna_md_t* md, double pf_scale)

Use *vrna_exp_params()* instead!

Global *get_boltzmann_factors_ali* (unsigned int n_seq, double temperature, double betaScale, *vrna_md_t* md, double pf_scale)

Use *vrna_exp_params_comparative()* instead!

Global get_centroid_struct_gquad_pr (int length, double *dist)

This function is deprecated and should not be used anymore as it is not threadsafe!

Global get_centroid_struct_pl (int length, double *dist, *vrna_ep_t* *pl)

This function was renamed to *vrna_centroid_from_plist()*

Global get_centroid_struct_pr (int length, double *dist, *FLT_OR_DBL* *pr)

This function was renamed to *vrna_centroid_from_probs()*

Global get_concentrations (double FEAB, double FEAA, double FEBB, double FEA, double FEB, double *startconc)

Global get_line (FILE *fp)

Use *vrna_read_line()* as a substitute!

Global get_mpi (char *Alseq[], int n_seq, int length, int *mini)

Use *vrna_aln_mpi()* as a replacement

Global get_path (const char *seq, const char *s1, const char *s2, int width)

Use *vrna_path_findpath()* instead!

Global get_plist (*vrna_ep_t* *pl, int length, double cut_off)

use *assign_plist_from_pr()* instead!

Global *get_scaled_alipf_parameters* (unsigned int n_seq)

Use *vrna_exp_params_comparative()* instead!

Global *get_scaled_parameters* (double temperature, *vrna_md_t* md)

Use *vrna_params()* instead!

Global *get_scaled_pf_parameters* (void)

Use *vrna_exp_params()* instead!

Global *get_TwoDfold_variables* (const char *seq, const char *structure1, const char *structure2, int circ)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, and *vrna_fold_compound_free()* instead!

Global get_TwoDpfold_variables (const char *seq, const char *structure1, char *structure2, int circ)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_pf_TwoD()*, and *vrna_fold_compound_free()* instead!

File gquad.h

Use ViennaRNA/eval/gquad.h and other header files instead

File grammar.h

Use ViennaRNA/grammar/basic.h, ViennaRNA/grammar/mfe.h or ViennaRNA/grammar/partfunc.h instead

File hairpin.h

Use ViennaRNA/eval/hairpin.h and ViennaRNA/backtrack/hairpin.h instead

File hairpin_loops.h

Use ViennaRNA/loops/hairpin.h instead

Global *HairpinE* (int size, int type, int si1, int sj1, const char *string)

{This function is deprecated and will be removed soon. Use *vrna_E_hairpin()* instead!}

Global hamming (const char *s1, const char *s2)

Use *vrna_hamming_distance()* instead!

Global hamming_bound (const char *s1, const char *s2, int n)

Use *vrna_hamming_distance_bound()* instead!

Global iindx

Do not use this variable anymore!

Global *init_co_pf_fold* (int length)

Global *init_pf_fold* (int length)

This function is obsolete and will be removed soon!

Global init_rand (void)

Use *vrna_init_rand()* instead!

Global *initialize_cofold* (int length)

Global *initialize_fold* (int length)

See *vrna_mfe()* and *vrna_fold_compound_t* for the usage of the new API!

Global int_urn (int from, int to)

Use *vrna_int_urn()* instead!

File interior_loops.h

Use ViennaRNA/loops/internal.h instead

File internal.h

Use ViennaRNA/eval/internal.h, ViennaRNA/mfe/internal.h, ViennaRNA/backtrack/internal.h, and ViennaRNA/partfunc/internal.h instead

File inverse.h

Use ViennaRNA/inverse/basic.h instead

Global *Lfold* (const char *string, const char *structure, int maxdist)

Use *vrna_mfe_window()* instead!

File loop_energies.h

Use ViennaRNA/loops/all.h instead

Global *loop_energy* (short *ptable, short *s, short *s1, int i)

Use *vrna_eval_loop_pt()* instead!

Global *LoopEnergy* (int n1, int n2, int type, int type_2, int si1, int sj1, int sp1, int sq1)

{This function is deprecated and will be removed soon. Use *vrna_E_internal()* instead!}

Global Make_bp_profile (int length)

This function is deprecated and will be removed soon! See Make_bp_profile_bppm() for a replacement

Global make_pair_table (const char *structure)

Use *vrna_ptable()* instead

Global make_pair_table_snoop (const char *structure)

Use *vrna_pt_snoop_get()* instead!

Global make_referenceBP_array (short *reference_pt, unsigned int turn)

Use *vrna_refBPcnt_matrix()* instead

Global *MEA* (plist *p, char *structure, double gamma)

Use *vrna_MEA()* or *vrna_MEA_from_plist()* instead!

File MEA.h

Use ViennaRNA/structures/mea.h instead

Global mean_bp_dist (int length)

This function is not threadsafe and should not be used anymore. Use *mean_bp_distance()* instead!

Global *mean_bp_distance* (int length)

Use *vrna_mean_bp_distance()* or *vrna_mean_bp_distance_pr()* instead!

Global *mean_bp_distance_pr* (int length, *FLT_OR_DBL* *pr)

Use *vrna_mean_bp_distance()* or *vrna_mean_bp_distance_pr()* instead!

File mfe.h

Use ViennaRNA/mfe/global.h and ViennaRNA/backtrack/global.h instead

File mfe_window.h

Use ViennaRNA/mfe/local.h instead

File multibranch.h

Use ViennaRNA/eval/multibranch.h, ViennaRNA/mfe/multibranch.h, ViennaRNA/backtrack/multibranch.h, and ViennaRNA/partfunc/multibranch.h instead

File multibranch_loops.h

Use ViennaRNA/loops/multibranch.h instead

File naview.h

Use ViennaRNA/plotting/naview/naview.h instead

Global *nc_fact*

See *vrna_md_t.nc_fact*, and *vrna_mfe()* to avoid using global variables

File neighbor.h

Use ViennaRNA/landscape/neighbor.h instead

Global nrerror (const char message[])

Use *vrna_log_error()* instead! (since v2.7.0)

Global pack_structure (const char *struc)

Use *vrna_db_pack()* as a replacement

Global *PAIR*

Use *vrna_basepair_t* instead!

Global pair_info

Use *vrna_pinfo_t* instead!

File params.h

Use ViennaRNA/params/basic.h instead

Global *paramT*

Use *vrna_param_t* instead!

Global parenthesis_structure (char *structure, *vrna_bp_stack_t* *bp, int length)

use vrna_parenthesis_structure() instead

Global parenthesis_zuker (char *structure, *vrna_bp_stack_t* *bp, int length)

use vrna_parenthesis_zuker instead

File part_func.h

Use ViennaRNA/partfunc/global.h instead

File part_func_window.h

Use ViennaRNA/partfunc/local.h instead

Global path_t

Use *vrna_path_t* instead!

Global *pbacktrack_circ* (char *sequence)

Use *vrna_pbacktrack()* instead.

Global *pf_circ_fold* (const char *sequence, char *structure)

Use *vrna_pf()* instead!

Global *pf_fold_par* (const char *sequence, char *structure, *vrna_exp_param_t* *parameters, int calculate_bppm, int is_constrained, int is_circular)

Use *vrna_pf()* instead

File pf_multifold.h

Use ViennaRNA/partfunc/multifold.h instead

Global *pf_paramT*

Use *vrna_exp_param_t* instead!

Global *plist*

Use *vrna_ep_t* or *vrna_elem_prob_s* instead!

File plot_aln.h

Use ViennaRNA/plotting/alignments.h instead

File plot_layouts.h

Use ViennaRNA/plotting/layouts.h instead

File plot_structure.h

Use ViennaRNA/plotting/structures.h instead

File plot_utils.h

Use ViennaRNA/plotting/utils.h instead

Global pr

Do not use this variable anymore!

Global print_tty_constraint (unsigned int option)

Use vrna_message_constraints() instead!

Global print_tty_constraint_full (void)

Use *vrna_message_constraint_options_all()* instead!

Global print_tty_input_seq (void)

Use *vrna_message_input_seq_simple()* instead!

Global print_tty_input_seq_str (const char *s)

Use *vrna_message_input_seq()* instead!

Global PS_color_aln (const char *structure, const char *filename, const char *seqs[], const char *names[])

Use *vrna_file_PS_aln()* instead!

File PS_dot.h

Use ViennaRNA/plotting/probabilities.h instead

Global PS_dot_plot (char *string, char *file)

This function is deprecated and will be removed soon! Use PS_dot_plot_list() instead!

Global *PS_rna_plot* (char *string, char *structure, char *file)

Use *vrna_file_PS_rnaplot()* instead!

Global *PS_rna_plot_a* (char *string, char *structure, char *file, char *pre, char *post)

Use *vrna_file_PS_rnaplot_a()* instead!

Global *PS_rna_plot_a_gquad* (char *string, char *structure, char *ssfile, char *pre, char *post)

Use *vrna_file_PS_rnaplot_a()* instead!

Global random_string (int l, const char symbols[])

Use *vrna_random_string()* instead!

File read_epars.h

Use ViennaRNA/params/io.h instead

Global *read_parameter_file* (const char fname[])

Use *vrna_params_load()* instead!

Global *read_record* (char **header, char **sequence, char ***rest, unsigned int options)

This function is deprecated! Use *vrna_file_fasta_read_record()* as a replacment.

File ribo.h

Use ViennaRNA/params/ribosum.h instead

Global *scale_parameters* (void)

Use *vrna_params()* instead!

Global *sect*

Use *vrna_sect_t* instead!

File sequence.h

Use ViennaRNA/sequences/sequence.h instead

Global *set_model_details* (*vrna_md_t* *md)

This function will vanish as soon as backward compatibility of RNAlib is dropped (expected in version 3). Use *vrna_md_set_default()* instead!

File SHAPE.h

Use ViennaRNA/probing/SHAPE.h instead

Global simple_circplot_coordinates (short *pair_table, float *x, float *y)

Consider switching to *vrna_plot_coords_circular_pt()* instead!

Global simple_xy_coordinates (short *pair_table, float *X, float *Y)

Consider switching to *vrna_plot_coords_simple_pt()* instead!

Global SOLUTION

Use vrna_subopt_solution_t instead!

Global space (unsigned size)

Use *vrna_alloc()* instead! (since v2.2.0)

Global *st_back*

set the *uniq_ML* flag in *vrna_md_t* before passing it to *vrna_fold_compound()*.

Global *stackProb* (double cutoff)

Use *vrna_stack_prob()* instead!

Global str_DNA2RNA (char *sequence)

Use *vrna_seq_toRNA()* instead!

Global str_uppercase (char *sequence)

Use *vrna_seq_toupper()* instead!

File stream_output.h

Use ViennaRNA/datastructures/stream_output.h instead

File string_utils.h

Use ViennaRNA/utils/strings.h instead

File structure_utils.h

Use ViennaRNA/utils/structures.h instead

File structures.h

Use an appropriate header from ViennaRNA/structures/ instead

File subopt.h

Use ViennaRNA/subopt/basic.h and ViennaRNA/subopt/wuchty.h instead

File subopt_zuker.h

Use ViennaRNA/subopt/zuker.h instead

File svm_utils.h

Use ViennaRNA/utils/svm.h instead

Global *temperature*

Use *vrna_md_defaults_temperature()*, and *vrna_md_defaults_temperature_get()* to change, and read the global default temperature settings

Global time_stamp (void)

Use *vrna_time_stamp()* instead!

Global *TwoDfold_backtrack_f5* (unsigned int j, int k, int l, *TwoDfold_vars* *vars)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, *vrna_backtrack5_TwoD()*, and *vrna_fold_compound_free()* instead!

Struct *TwoDfold_vars*

This data structure will be removed from the library soon! Use *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, and *vrna_fold_compound_free()* instead!

Global *TwoDfoldList* (*TwoDfold_vars* *vars, int distance1, int distance2)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_mfe_TwoD()*, and *vrna_fold_compound_free()* instead!

Global TwoDpfold_pbacktrack (TwoDpfold_vars *vars, int d1, int d2)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_pf_TwoD()*, *vrna_pbacktrack_TwoD()*, and *vrna_fold_compound_free()* instead!

Global TwoDpfold_pbacktrack5 (TwoDpfold_vars *vars, int d1, int d2, unsigned int length)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_pf_TwoD()*, *vrna_pbacktrack5_TwoD()*, and *vrna_fold_compound_free()* instead!

Struct TwoDpfold_vars

This data structure will be removed from the library soon! Use *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_pf_TwoD()*, and *vrna_fold_compound_free()* instead!

Global TwoDpfoldList (TwoDpfold_vars *vars, int maxDistance1, int maxDistance2)

Use the new API that relies on *vrna_fold_compound_t* and the corresponding functions vrna_fold_compound_TwoD(), *vrna_pf_TwoD()*, and *vrna_fold_compound_free()* instead!

File units.h

Use ViennaRNA/utils/units.h instead

Global unpack_structure (const char *packed)

Use *vrna_db_unpack()* as a replacement

Global *update_alifold_params* (void)

Usage of this function is discouraged! The new API uses *vrna_fold_compound_t* to lump all folding related necessities together, including the energy parameters. Use vrna_update_fold_params() to update the energy parameters within a *vrna_fold_compound_t*.

Global *update_co_pf_params* (int length)

Use *vrna_exp_params_subst()* instead!

Global *update_co_pf_params_par* (int length, *vrna_exp_param_t* *parameters)

Use *vrna_exp_params_subst()* instead!

Global *update_cofold_params* (void)

See *vrna_params_subst()* for an alternative using the new API

Global *update_cofold_params_par* (*vrna_param_t* *parameters)

See *vrna_params_subst()* for an alternative using the new API

Global *update_fold_params* (void)

For non-default model settings use the new API with *vrna_params_subst()* and *vrna_mfe()* instead!

Global *update_fold_params_par* (*vrna_param_t* *parameters)

For non-default model settings use the new API with *vrna_params_subst()* and *vrna_mfe()* instead!

Global *update_pf_params* (int length)

Use *vrna_exp_params_subst()* instead

Global *update_pf_params_par* (int length, *vrna_exp_param_t* *parameters)

Use *vrna_exp_params_subst()* instead

Global urn (void)

Use *vrna_urn()* instead!

File utils.h

Use ViennaRNA/utils/basic.h instead

Use ViennaRNA/utils/basic.h instead

Struct *vrna_basepair_t*

Use *vrna_bp_t* instead!

Global *vrna_callback_free_auxdata* (void *data)

Use *vrna_auxdata_free_f(void *data)* instead!

Global *vrna_cofold* (const char *sequence, char *structure)

This function is obsolete since *vrna_mfe()*/vrna_fold() can handle complexes multiple sequences since v2.5.0. Use *vrna_mfe()*/vrna_fold() for connected component MFE instead and compute MFEs of unconnected states separately.

Global *VRNA_CONSTRAINT_FILE*

Use 0 instead!

Global *VRNA_CONSTRAINT_MULTILINE*

see *vrna_extract_record_rest_structure()*

Global VRNA_CONSTRAINT_NO_HEADER

This mode is not supported anymore!

Global *VRNA_CONSTRAINT_SOFT_MFE*

This flag has no meaning anymore, since constraints are now always stored! (since v2.2.6)

Global *VRNA_CONSTRAINT_SOFT_PF*

Use *VRNA_OPTION_PF* instead!

Global vrna_exp_param_s::id

This attribute will be removed in version 3

Global *vrna_extract_record_rest_constraint* (char **cstruc, const char **lines, unsigned int option)

Use *vrna_extract_record_rest_structure()* instead!

Global *vrna_message_error* (const char *format,...)

Use *vrna_log_error()* instead! (since v2.7.0)

Global *vrna_message_info* (FILE *fp, const char *format,...)

Use *vrna_log_info()* instead! (since v2.7.0)

Global *vrna_message_verror* (const char *format, va_list args)

Use *vrna_log_error()* instead! (since v2.7.0)

Global *vrna_message_vinfo* (FILE *fp, const char *format, va_list args)

Use *vrna_log_info()* instead! (since v2.7.0)

Global *vrna_message_vwarning* (const char *format, va_list args)

Use *vrna_log_warning()* instead! (since v2.7.0)

Global *vrna_message_warning* (const char *format,...)

Use *vrna_log_warning()* instead! (since v2.7.0)

Global *vrna_mfe_dimer* (*vrna_fold_compound_t* *fc, char *structure)

This function is obsolete since *vrna_mfe()* can handle complexes multiple sequences since v2.5.0. Use *vrna_mfe()* for connected component MFE instead and compute MFEs of unconnected states separately.

File walk.h

Use ViennaRNA/landscape/walk.h instead

Global warn_user (const char message[])

Use *vrna_log_warning()* instead! (since v2.7.0)

Global *write_parameter_file* (const char fname[])

Use *vrna_params_save()* instead!

Global xrealloc (void *p, unsigned size)

Use *vrna_realloc()* instead! (since v2.2.0)

Global *zukersubopt* (const char *string)

use vrna_zukersubopt() instead

Global *zukersubopt_par* (const char *string, *vrna_param_t* *parameters)

use vrna_zukersubopt() instead

Functions

char ***pbacktrack**(char *sequence)

Sample a secondary structure from the Boltzmann ensemble according its probability.

```
#include <ViennaRNA/partfunc/global.h>
```

Parameters

- **sequence** – The RNA sequence

Pre

st_back has to be set to 1 before calling *pf_fold()* or *pf_fold_par()pf_fold_par()* or *pf_fold()* have to be called first to fill the partition function matrices

Returns

A sampled secondary structure in dot-bracket notation

char ***pbacktrack5**(char *sequence, int length)

Sample a sub-structure from the Boltzmann ensemble according its probability.

```
#include <ViennaRNA/partfunc/global.h>
```

char ***pbacktrack_circ**(char *sequence)

Sample a secondary structure of a circular RNA from the Boltzmann ensemble according its probability.

```
#include <ViennaRNA/partfunc/global.h>
```

This function does the same as *pbacktrack()* but assumes the RNA molecule to be circular

Deprecated:

Use *vrna_pbacktrack()* instead.

Pre

st_back has to be set to 1 before calling *pf_fold()* or *pf_fold_par()pf_fold_par()* or *pf_circ_fold()* have to be called first to fill the partition function matrices

Parameters

- **sequence** – The RNA sequence

Returns

A sampled secondary structure in dot-bracket notation

Variables

int **st_back**

Flag indicating that auxilary arrays are needed throughout the computations. This is essential for stochastic backtracking.

```
#include <ViennaRNA/partfunc/global.h>
```

Set this variable to 1 prior to a call of *pf_fold()* to ensure that all matrices needed for stochastic backtracking are filled in the forward recursions

Deprecated:

set the *uniq_ML* flag in *vrna_md_t* before passing it to *vrna_fold_compound()*.

See also

pbacktrack(), *pbacktrack_circ*

Part IV

Miscellaneous

CHAPTER

FIVE

NEWS

5.1 Version 2.8.0

11 - 06 - 2026

Version 2.8.0 introduces interacting interval pair (IIP) soft constraints, enabling the use of experimental structure probing data derived from RNA crosslinking experiments to guide structure prediction!

With our new release we finally provide an *exact algorithmic implementation for integrating RNA crosslinking data* to guide the structure prediction. In contrast to the heuristic approach presented in Skibinski et al. 2025, the new IIP soft constraints feature allows for addressing all candidate structures that support a particular pair of sequence intervals exactly once. Consequently, guiding potentials (soft constraints) derived from interval pairs are independent of the length of the intervals.

Another major change since version 2.7.2 is related to **energy parameters for G-Quadruplexes and modified bases**. For the former, we fix a wrong unit conversion that existed since the introduction of G-Quadruplex support in 2012, that rendered quadruplexes about four times too stable. For the latter, we fix the discrepancy between results obtained from `RNAfold` compared to using the API directly. In particular, our executable programs with modified base support pre-process the input sequences to substitute any modified base by its fallback base, as specified in the modified base parameter file, before handing it over to the API. Unfortunately, using the v2.7.x API directly without this preprocessing step would yield different (wrong) results. The related API functions now internally handle this preprocessing automagically, so no user intervention is necessary.

In addition, the new release fixes many computational bugs, compilation problems with more recent versions of maintainer tools, such as required for the Python and Perl interfaces, and adds many new features. Most notably, we added the LinearBPDesign Scheme to `RNAinverse` and include new and updated energy parameter files for pseudouridine, N1-methyl-pseudouridine, and dihydrouridine. Moreover, the `RNAconsensus` tool finally implements the SoftConsensus Scheme. As of this version we also **removed the Python 2 support!**

Last but not least, we introduce a new look for the HTML documentation.

The full list of changes, bug fixes and additional features is available in the Changelog for version 2.8.0.

5.2 Version 2.7.0

11 - 10 - 2024

Finally, version 2.7 brings G-quadruplex support for circular RNAs and new methods for integrating experimental RNA structure probing data!

This new release brings the source tree closer to the long-awaited version 3. More code within RNAlib and the executable programs has been transformed to use the v3 API. The next major release will then turn-on deprecation warnings by default to warn any third-party programs that rely on our library to take action and adapt their code accordingly. While several bugs have been fixed, tons of additional features have been implemented. Note, that the decomposition scheme for global RNA structure prediction changed from using an M1 DP matrix (exactly one branch within a multiloop) to M2 (at least two branches within a multiloop). This change not only slightly reduces the overall runtime but also unifies the implementations for circular RNAs, global and local RNA structure prediction.

Moreover, the API underwent a major clean-up to unify the naming of functions and the data types of their input and output parameters. RNAlib header include files have now been grouped into separate subdirectories to obtain a better structured interface. As usual, the full list of changes, bug fixes and additional features can be found at the Changelog for version 2.7.0.

5.3 Version 2.6.0

06 - 09 - 2023

With version 2.6 we introduce modifed base support and mono-valent salt corrections!

Several of our executable tools received support for adding additional energy parameters for modified bases. At this time we include publically available energy parameters for inosine, pseudouridine, m6A, 7DA, and purine (a.k.a. nebularine). In addition, we add stacking parameters for dihydrouridine as predicted by Rosetta/RECESS. The implementations allow for easily adding more parameters if they become available. Additionally, we finally integrate a physics-based model to correct energy contributions based on the concentrations of mono-valent ions in solution, such as sodium. This correction is available for virtually all executable programs and easily accessible within our C-library.

See the Changelog for version 2.6.0 for a complete list of new features and bugfixes.

5.4 Version 2.5.0

11 - 08 - 2021

The all new release of version 2.5 brings multi-strand interaction prediction!

The new executable tool RNAmultifold is the successor of the RNA-RNA dimer interaction prediction tool RNAcofold and effectively lifts the restriction to just two interacting strands. It follows the same principle of concatenating the RNA strands that shall form a complex and then predicts MFE and partition function. Along with that, it can also compute equilibrium concentrations of the complexes formed.

See the Changelog for version 2.5.0 for a complete list of additions, novel features and fixed bugs.

5.5 Version 2.4.0

08 - 01 - 2017

With version 2.4 sliding-window structure prediction receives the constraint framework!

Starting with this version, the sliding-window secondary structure prediction implementations as available through RNALfold, RNAplfold, and RNALalifold are constraints-aware. Thus, they can readily incorporate RNA structure probing data, such as from SHAPE experiments, etc.

See the Changelog for version 2.4.0 for a complete list of new features and bugfixes.

5.6 Version 2.3.0

11 - 01 - 2016

Version 2.3 introduces the unstructured domain extension of the RNA folding grammar!

This extension adds RNA-ligand interactions, e.g. RNA-protein, for unpaired stretches in RNA secondary structures. The feature is easy to use through the command file interface in RNAfold.

See the Changelog for version 2.3.0 for a complete list of new features and bugfixes.

5.7 Version 2.2.0

01 - 25 - 2016

After almost a year without a new release, we are happy to announce many new features. This version officially introduces (generic) hard- and soft-constraints for many of the folding algorithms. Thus, chemical probing constraints, such as derived from SHAPE experiments, can be easily incorporated into RNAfold, RNAalifold, and RNAsubopt. Furthermore, RNAfold and the RNAlib interface allow for a simple way to incorporate ligand binding to specific hairpin- or interior-loop motifs. This version also introduces the new v3.0 API of the RNAlib C-library, that will eventually replace the current interface in the future.

See the Changelog for version 2.2.0 for a complete list of new features and bugfixes.

5.8 Version 2.1.9

02 - 05 - 2015

This is a major bugfix release

This release changes the way how the ViennaRNA Package handles dangling end and terminal mismatch contributions for exterior-, and multibranch loops. We strongly recommend upgrading your installation to this or a newer version to obtain predictions that are better comparable to RNAstructure or UNAFold.

Please see the Changelog for version 2.1.9 for further details on the actual changes to the underlying energy parameters.

5.9 Version 2.1.7

04 - 13 - 2014

For a long time, Mac OS X users were not able to correctly build the Perl/Python interface of the ViennaRNA Package. Starting with v2.1.7, this limitation has been removed, and the interface should compile and work as expected.

Please see the Install Notes for Mac OS X users for further details.

5.10 Version 2.1.0

01 - 25 - 2013

Since ViennaRNA Package Version 2.1.0 we have enabled G-Quadruplex prediction support into RNAfold, RNAcofold, RNALfold, RNAalifold, RNAeval and RNAplot.

See the changelog for details.

5.11 Older news

5.12 Version 2.0

- Meanwhile, a lot of changes in the RNAlib have accumulated. See the Reference Manual and the Changelog for further details
- All algorithms use the Turner'04 nearest neighbor model
- The RNAlib provides (OpenMP) threadsafe folding routines per default. This is enables concurrent calls to the folding routines in parallel. The feature can be disabled by passing '–disable-openmp' to the configure script
- serious changes in command line parameters. Everything complies with GNU standard from now on (short options with preceding '-', long options with preceding '–'.

- FASTA file support for RNAfold. RNA sequences do not need to be passed on a single line anymore when a FASTA header is provided.
- The new program RNA2Dfold computes MFE, partition function and stochastically sampled secondary structures in a partitioning of the secondary structure space according to the base pair distance to two reference structures
- The new program PKplex computes…
- The new program RNALfoldz computes locally stable secondary structures together with a z-score
- The new program RNALalifold computes locally stable consensus structures for alignments
- The new program RNAparconv enables the conversion of 'old' energy parameter files (v1.4-v1.8) to the new format used in version 2.x

5.13 Version 1.8

- new RNAalifold has better treatment of gaps and ribosum based covariance scores. Use the -old switch for compatibility with older RNAalifold versions.
- RNAplfold -u now computes all accessibilities up to a maximum length (much faster than computing each individually)
- ATTENTION: output formats of RNAplfold -u and or RNAup have been changed
  Programs parsing RNAplfold and RNAup output will have to be modified.
- RNAfold and RNAalifold compute centroid structures when run with -p use the -MEA option to compute Maximum Expected Accuracy structures.

5.14 Version 1.7

- RNAplfold can now be used to compute accessibilities, i.e. the probability that a stretch of the RNA remains unpaired (and thus available for intermolecular interactions).
- A new version of RNAup predicts RNA-RNA interactions takeing into account the competition between inter- and intramolecular structure in both molcules
- Circular RNAs can be treated by RNAfold, RNAalifold, RNAsubopt, and RNAcofold
- RNAaliduplex predicts RNA-RNA interactions between two sets of aligned sequences (inter-molecular structure only)

5.15 Version 1.6

- The RNAforester program for tree-alignments of RNA structures is now distributed with the Vienna RNA package, see the RNAforester subdirectory for more information. RNAforester was written by Matthias Hoechsmann mhoechsm@techfak.uni-bielefeld.de
- The Kinfold program for stochastic simulation of folding trajectories is now included in the package, see the Kinfold subdirectory.
- cofolding of two structures now supports suboptimal folding and partition function folding. ATTENTION: Energies of hybrid structures now include the Duplex-initiation energy, which was neglected in previous version.
- RNAplfold is a partition function variant of RNALfold. It computes the mean probability of a (local) base pair averaged over all sequence windows that contain the pair.
- new utilities to color alignments and consensus structures
- RNAfold -p now computes the centroid structure
- ATTENTION: ensemble diversities in version <1.6.5 are off by a factor 2

5.16 Version 1.5pre

- ViennaRNA now uses autoconfig generated configure scripts for even better portability (should compile on any UNIX, Linux, MacOS X, Windows with Cygwin).
- The new RNAalifold program predicts consensus structures for a set of aligned sequences.
- Complete suboptimal folding is now integrated in the library.
- Beginning support for co-folding of two strands: energy_of_struct() and RNAeval can now compute energies of duplex structures.
- RNAcofold predicts hybrid structures of two RNA strands
- RNAduplex predicts hybrid structures, while allowing only inter-molecular base pairs (useful for finding potential binding sites)
- RNALfold predicts locally stable structures in long sequences.
- Major changes to Perl module. See the pod documentation (perldoc RNA).
- RNAsubopt can do stochastic backtracking to produce samples of suboptimal structures with Boltzmann statistics.
- New utilities to rotate secondary structure plots and annotate them with reliability data.
- Various small bug fixes

5.17 Version 1.4

- New Turner parameters as described in Mathews et.al. JMB v288, 1999. Small changes to format of parameter files (old param files won't work!)
- mfe and suboptimal folding will produce only structures without isolated pairs if noLonelyPairs=1 (-noLP option), for partition function folding pairs that can only occur as isolated pairs are not formed.
- setting dangles=3 (-d3 option) will allow co-axial stacking of adjacent helices in mfe folding and energy_of_struct().

5.18 Version 1.3.1

- RNAheat would produce spikes in the specific heat because dangling end energies did not go smoothly to 0.
- PS dot plots now have an option to use a log scale (edit _dp.ps file and set logscale to true).

5.19 Version 1.3

- Secondary structure plots now use E. Bruccoleri's naview routines for layout by default. New utility RNAplot produces secondary structure plots from structures in bracket notation with several options.
- New -d2 option in RNAfold and RNAeval sets dangles=2, which makes energy_of_struct() and fold() treat dangling ends as in pf_fold(). -noLP option in RNAfold etc sets noLonelyPairs=1, which avoids most structures containing lonely base pairs (helices of length 1).
- new utility functions pack_structure() unpack_structure() make_pair_table() and bp_distance(). RNAdistance adds bp_distance() via -DP switch.
- First release of RNAsubopt for complete suboptimal folding.
- fixed bug in asymmetry penalty for interior loops.
- Default compilation now uses doubles for partition function folding.

5.20 Version 1.2.1

- Fixed bug in version 1.2 of the RNAheat program causing overflow errors for most input sequences.
- The PS_dot_plot() and PS_rna_plot() routines now return an int. The return value is 0 if the file could not be written, 1 otherwise.
- This version contains the alpha version of a perl5 module, which let's you access all the capabilities of the Vienna RNA library from perl scripts.

5.21 Version 1.2

- New energy parameters from (Walter et.al 1994).
- Energy parameters can be read from file.
- RNAeval and energy_of_struct() support logarithmic energy function for multi-loops.
- gmlRNA() produces secondary structure drawing in gml (Graph Meta Language).
- Many bug fixes.

CHAPTER

SIX

CHANGELOG

Below, you'll find a list of notable changes for each version of the ViennaRNA Package.

6.1 Version 2.8.x

6.1.1 Unreleased

6.1.2 Version 2.8.0

Release date: *2026-11-06*

Fixed in

- Fix Wuchty subopt with G-quadruplexes
- Fix Wuchty subopt with circular RNAs
- Fix heap-use-after-free in `RNAsubopt` constraint handling
- Remove debug output from `subopt/subopt_zuker.c`
- Fix base pair hard constraints for modified bases
- Fix compiler warning with `glibc-2.43`
- Fix energy evaluation split for internal loops into stacks, bulges, real int loops
- Fix application order bug in hard constraints
- Fix multibranch loop soft constraints in Wuchty subopt
- Fix wrong units for G-Quadruplex energy parameters
- Fix dangling end and mismatch corrections for modified bases
- Fix correction for bulges of length 1 with modified nucleotides
- Fix MSA input processing in `RNAplot`
- SWIG: Fix building with most recent `swig=4.5.0`
- DOC: Fix Python docstring issues with unnamed struct/union since `doxygen=1.18`
- Fix possible resource leak in `ssv` and `xrna` plot outputs
- API: Fix multiple `extern` declarations for deprecated global variables `cv_fact` and `nc_fact`
- Fix Wuchty-style `subopt*()` for `dangles==1|3`
- Fix retrieval of wrong parameter data structures after reloading parameters from file
- Fix discrepancy for modified base support in `RNAfold` vs. API calls due to unmodified input sequence
- DOC: Fix expansion of function-like macros in `doxygen`
- DOC: Fix many `doxygen` related warnings

- Fix problems with enforced base pair hard constraints in combination with unpaired hard constraints
- Do not attempt to perform stochastic backtracking for interacting RNAs
- Fix wrong brackets in `probing/probing.c`
- Fix size of encoding5/encoding3 in `vrna_seq_t`
- Fix a few format strings in `printf()` and `sscanf()` usage
- Fix return value of `vrna_math_fun_dbl_kde()` in `math/fun_kde.c`

Changed since

- API: Preserve fractional unpaired and base-pair soft constraints for single-sequence partition functions, and round accumulated contributions for MFE calculations
- API: Change raw soft-constraint storage fields from `int` to `double`
- Remove redundant check in `backtrack/bt_exterior_f5.c`
- Add 'G' to the list of pairing partners for pseudo-uridine
- Use `vrna_log_info()`` to print input sequence length in interactive mode of executable programs
- Changed output format for structure benchmark in `RNAfold` and `RNAalifold`
- DOC: Major rewrite/restructure of documentation to allow for stand-alone HTML website
- DOC: Move shipped sphinx extensions into subdirectory `doc/sphinx-extensions`
- DOC: Update `breathe` sphinx extension to version `5.1`
- DOC: Use `sphinx-design`, `sphinx-contributors`, and `pydata-sphinx-theme` for documentation
- DOC: Changed `NEWS` file structure
- DOC: Changed `CHANGELOG.md` file structure
- DOC: Update contributors section in manual
- Remove redundant MFE computation in `RNAup`
- Make nucleotide encoding model details `const`
- Always start with default hard constraints before any user-supplied ones in `vrna_hc_prepare()`
- AUTOCONF: Increase minimum `autoconf` version to 2.70
- Preserve fractional soft constraints for partition functions
- WORKFLOWS: Use `pixi` environment for github worker builds
- Require `STD:c++17` for MSVC builds
- Allow for IIP soft constraints in command files
- Allow for more flexibility in how to provide parameters for structure probing data strategies in executable programs
- Update `libsvm` to `libsvm-3.37`
- Update `dlib` to `dlib-20.0.1`

Added in

- Support hi-motifs in Wuchty-style subopt
- Add `vrna_en_internal()`, `vrna_en_bulge()`, `vrna_en_stack()` functions
- Add Dihydrouridine parameters for pairs with `G`
- Add (new) parameter files for Pseudouridine and N1-Methyl-Pseudouridine
- Include all RNA modification parameter files into `RNAlib`

- Add *LinearBPDesign* sequence seeds for `RNAinverse` available via the new `--init-strategy` option
- Finally add softconsensus feature in `RNAconsensus.py`
- Added relaxed metrics to structure benchmark
- Add `vrna_db_compat_remove()` and `vrna_ptable_compat_remove()`
- Add `vrna_ptable_count_bp()` and `vrna_ptable_count_up()`
- Allow for ignoring compatible base pairs for structure benchmark in `RNAfold` and `RNAalifold`
- Add `vrna_ribosum_mx()`
- Add accessibility computations in global scope
- Add callback-based API for global equilibrium probability computations (base pairs, helices, accessibility)
- Add `vrna_eqm_probs_bp()`, `vrna_eqm_probs_up()`, and `vrna_eqm_probs_hx()`
- DOC: Add sphinx extension for release file downloads
- Add `vrna_sequence_mutate()` for in-place mutation of RNA sequence in `vrna_fold_compound_t` data structure
- Add `VRNA_ARRAY_REGISTER*` macros to automatically create typedefs for `vrna_array()` types
- Add `vrna_ep_array_t` type and use it as return type for `vrna_eqm_probs_*()` functions
- Add `.all-contributorsrc` file to list contributors according the the *All Contributors* guidelines
- Add name of lexically-enclosing function of a log call to the log message
- Add log-message calling function name output flag for executable programs
- Add `pixi.toml` with a few pre-set tasks to allow for reproducible builds
- Add **interacting interval pair (IIP) soft constraints** feature
- Add **RNA crosslinking data support** as a source for structure probing input (single sequence MFE and partition function)
- Add Logit (log-odds) function to math functions

Removed in

- SWIG: Remove Python 2.x support

Deprecated since

- Deprecated use of all global variables

6.2 Version 2.7.x

6.2.1 Version 2.7.2

Release date: *2025-12-29*

Fixed in v2.7.2

- Fix structure output in `RNAeval` for more than 2 interacting RNAs
- Fix wrong application of soft constraints to single sequences in `RNAcofold`
- API: Fix wrong application of soft constraints to single sequences in `vrna_mfe_dimer()`
- API: Restore backward compatibility issues after early removal of `vrna_constraints_add_SHAPE()` and others
- Fix misplaced `i` in `misc/rna_turner2004.par`

Changed since v2.7.2

- API: Re-use energy parameters for successive `vrna_params()` / `vrna_exp_params()` calls if possible
- API: Change structure benchmark numbers to `double` type

Added in v2.7.2

- Add `--benchmark` option for `RNAfold`
- Add `--benchmark` option for `RNAalifold`
- API: Add negative log probability method (information content) as generic probing data strategy
- API: Add `vrna_math_dbl_chain()` to allow for chaining `vrna_math_dbl*` functions
- API: Add function to allow for chaining probing data strategies

6.2.2 Version 2.7.1

Release date: *2025-12-06*

Fixed in v2.7.1

- Fix constraints parsing in `RNAsubopt`
- API: Fix backtracking issues in exterior loop
- API: Fix backtracking issues with odd dangles
- API: Fix segmentation fault in `vrna_pt_pk_remove()`
- API: Fix `vrna_plist()` and `vrna_ptable()` to deal with malformed input
- API: Fix soft constraints access in backtracking of QM2 (Boltzmann sampling)
- SWIG: Fix `RNA.fold_compound.bpp()` output for multistrand base pair probabilities
- SWIG: Fix Python object ownership in callbacks for unstructured domains, direct soft constraints and landscape moves
- Fix energy parameter file markup
- Fix compilation with compilers that default to C23
- Fix compilation without `libsvm`

Changed since v2.7.1

- API: Refactor structure probing data API to support multiple data at once, position-wise weighting, user-defined reactivity-to-energy strategies and more
- API: Refactor hard constraint wrappers
- SWIG: Do not return symmetric pair probability matrix by default
- Update `dlib` to version 20.0
- Overall improvements to the documentation
- Update `swig` requirement to 4.2.0
- Update `breathe` to version v5.0.0a5

Added in v2.7.1

- Allow for FASTA header changes in `ct2db`
- Add new command line options for structure probing data to `RNAfold` and `RNAsubopt`
- API: Add `ViennaRNA/math/` to provide several useful mathematical functions

- API: Add `vrna_str_to_dbl_array()`
- API: Add `vrna_file_doubles_read()`
- SWIG: Add wrappers for concentration dependency computations
- Add probing data priors (paired and unpaired) for 1M7 and DMS

6.2.3 Version 2.7.0

Release date: *2024-10-11*

Fixed in v2.7.0

- API: Fix `vrna_neighbor_diff*()` insertion moves
- API: Fix MFE inside recursion for multiloop unpaired positions
- API: Fix `vrna_hx_from_ptable()` when provided hairpins of length 0
- API: Fix `vrna_hx_merge()` to support pseudoknotted helices
- API: Fix re-use of previous energy parameters upon model changes in `duplex.c`
- API: Fix re-use of previous energy parameters upon model changes in `c_plex.c`
- API: Fix re-use of previous energy parameters upon model changes in `plex.c`
- API: Fix re-use of previous energy parameters upon model changes in `ali_plex.c`
- API: Fix re-use of previous energy parameters upon model changes in `snofold.c`
- API: Fix re-use of previous energy parameters upon model changes in `snoop.c`
- API: Fix mismatch and dangling end energies in `sc_cb_mod.c` for modified base support
- API: Fix Zuker-style subopt backtracking
- API: Fix bug in `vrna_strdup_vprintf()` that resulted in losing parmeters upon consecutive calls
- API: Fix default exterior loop MFE soft constraints for comparative structure prediction
- SWIG: Use `av_len()` instad of `av_top_index()` for Perl 5 to support perl5 < v5.18.0
- SWIG: Fix compilation issues for Python 3.12 due to use of `SWIG_Python_str_AsChar()`
- AUTOCONF: Fix several `autoconf/automake` related issues

Changed since v2.7.0

- Adapt structure conservation coloring and add legend in alignment output of `RNAplot`, `RNAalifold`, and `RNALalifold`
- Change `RNAplot` command line argument `-o` to `-f`
- API: Group related API symbols and header files into specific subdirectories
- API: Allow base pair hard constraints via commands file where `j = i + 1`
- API: Refactor `vrna_pk_plex()` accessibility computations
- API: Refactor backtracking implementations and API, now located under `ViennaRNA/backtrack/`
- API: Refactor experimental probing data (SHAPE) implementations, now located under `ViennaRNA/probing/`
- API: Refactor auxiliary grammar extension API, now located under `ViennaRNA/grammar/`
- API: Refactor G-Quadruplex implementation and fix existing bugs and numerical issues in corresponding energy evaluation
- API: Refactor verbose `vrna_eval*()` implementations

- API: Refactor structure plotting API and add unified structure plotting function `vrna_plot_structure()`
- API: Remove `exit()` calls from `RNAlib`
- API: Change behavior and parameter order for `vrna_hc_add_bp_strand()`
- API: Change behavior and parameters of `vrna_hc_add_up_strand()`
- API: Unify backtracking matrix flags
- API: Use `vrna_array()` based base pair and backtrack stacks for MFE implementations
- SWIG: Return `int` instead of float for `eval_structure_pt_simple()`
- DOC: Improve document structure
- Bump `libsvm` to version 3.35

Added in v2.7.0

- Add hard limit for number of input structures in `RNAdistance`
- Add counter example settings to `RNAalifold`
- Add covariance annotation legend to `RNAplot` layout plots for MSA input
- Add covariance annotation legend to `RNALalifold` layout plots
- Add `RNAconsensus` Python program that will eventually replace `refold.pl`
- Add `--log-file`, `--log-level`, `--log-call` and `--log-time` command line options to executable programs
- Add `--betaScale` and `--pfScale` options to and rescale Boltzmann factors in `RNAPKplex`
- Add support for G-Quadruplexes in circular RNAs for `RNAfold`, `RNAalifold`, and `RNAeval`
- Add `--random-seed` option to `RNAsubopt` and `RNAalifold` to specify seed for random number generator
- API: Add circular RNA G-Quadruplex support
- API: Add structure prediction benchmark functions `vrna_compare_structure()` and `vrna_compare_structure_pt()`
- API: Add `vrna_annotate_covar_pt()` that allows for specifying number of counter examples
- API: Add structure conservation legend to EPS consensus structure layout plots
- API: Add `vrna_string_make_space_for()` and `vrna_string_available_space()` functions in `ViennaRNA/datastructures/string.h`
- API: Add `vrna_file_PS_aln_opt()` to allow for changing conservation coloring
- API: Add flexible log message system to avoid spam on `stderr` and `stdout`
- API: Add (generic) compressed sparse row (CSR) matrix implementation
- API: Add Eddy 2014 approach to incorporate experimental probing data (using Gaussian KDE)
- API: Add generic support for experimental probing data via new API in `ViennaRNA/probing/basic.h`
- API: Add full probing data support for consensus structure prediction
- API: Add `vrna_sc_multi_cb_add_comparative()` to allow for multi callback soft constraints in comparative structure predictions
- API: Add `vrna_fold_compound_t` to parameters passed to recursion status callback
- API: Add M2 matrices in favor of M1 for global MFE and partition function computations
- API: Add safeguard to `vrna_array_free()`
- API: Add `vrna_pairing_tendency()` as replacement for `vrna_db_from_probs()`

- API: Introduce entropic penalty for unpaired circular RNAs (may be switched off by model settings `circ_penalty` flag)
- SWIG: Add wrapper for `vrna_hx_merge()`
- SWIG: Add wrapper for `vrna_sc_multi_cb_add()`
- SWIG: Add wrappers for `vrna_hc_add_bp_strand()` and `vrna_hc_add_up_strand()`
- SWIG: Add file I/O constants
- SWIG: Add wrappers for structure prediction benchmark functions
- SWIG: Add wrappers for log message system
- SWIG: Add wrapper for `vrna_stack_prob()`
- SWIG: Add wrappers for new probing data API
- SWIG: Add wrapper for `vrna_n_multichoose_k()`
- AUTOCONF: Add `./configure --enable-debug` option that prevents removal of `vrna_log_debug()` message from `RNAlib`
- Add m5C JSON energy parameter file
- Add N1-methylpseudouridine JSON energy parameter file
- Add OpenMP library flags to `RNAlib2.pc`

Deprecated since v2.7.0

- API: Deprecate `vrna_message_info()`, `vrna_message_warning()`, and `vrna_message_error()` in favor of new logging system

Removed in v2.7.0

- Remove RNA-Tutorial since it is now included as part of the reference manual
- Remove Python 2 builds from MacOSX installer

6.3 Version 2.6.x

6.3.1 Version 2.6.4

Release date: *2023-09-25*

Fixed in v2.6.4

- Fix C++17 compilation issue with `kinwalker`
- Fix potential compilation issues with C++20 in `RNAforester` frontend
- API: Fix char array initialization in `snoop.c`
- API: Fix potentially leaking file pointer in `vrna_file_msa_read()`
- API: Fix potentially leaking memory in `rnaplot_EPS()`
- API: Fix potential use of uninitialized variable in `vrna_rotational_symmetry_db_pos()`
- API: Fix soft constraints issue in external loop of `vrna_subopt*()`
- SWIG: Fix `var_array` Python slices and associated memory leak
- SWIG: Fix bogus `delete/free()` calls in swig interface

Changed since v2.6.4

- Refactor and correct spelling issues in man pages for several executable programs
- SWIG: Return `var_array<vrna_move_t>` objects in Python wrapped `vrna_neighbors()` and `vrna_move_neighbor_diff()`
- SWIG: Refactor file handle wrapping between Python 3 and C
- Remove unused code in `RNApuzzler`
- DOC: Transition reference manual from `doxygen` to `sphinx` via `breathe` bridge
- DOC: Merge documentation of C-API and Python API
- DOC: Merge parts of tutorial into reference manual
- AUTOCONF: Refactor autoconf checks for capability to build reference manual
- AUTOCONF: Deactivate build of `RNAxplorer` if `lapack` requirements are not met

Added in v2.6.4

- API: Add shift move support to `vrna_move_neighbor_diff*()` functions
- SWIG: Add swig class output parameter typemap for Python
- SWIG: Add `__hash__()` and `__eq__()` methods for wrapped `_vrna_move_t` in Python
- Add requirements to build `RNAlib` with MSVC for Windows

6.3.2 Version 2.6.3

Release date: *2023-07-20*

Fixed in v2.6.3

- Fix preparation of input sequences for modified base support in `RNAcofold`

Changed since v2.6.3

- Make JSON parser integral part of ViennaRNA library
- API: Move modified energy parameters into 'modified_base' object in JSON file(s)

Added in v2.6.3

- SWIG: Enable stand-alone build of Python interface (for PyPI)
- Add enthalpy and terminal end values for predicted stacks with dihydrouridine
- TESTS: Allow for using `pytest` to test the Python 3 interface

6.3.3 Version 2.6.2

Release date: *2023-06-21*

Fixed in v2.6.2

- Fix energy corrections for modified base support when unmodified base is not the same as fallback base, e.g. in the case of inosine
- SWIG: Fix `fc.sc_add_bp()` propagation of constraint values

Added in v2.6.2

- Add soft constraints to multifold external loop decomposition
- Add soft constraints preparation stage callback
- SWIG: Wrap energy parameter file strings
- TESTS: Add modified base tests on duplex data with I-C and A-Psi pairs from publications

6.3.4 Version 2.6.1

Release date: *2023-06-12*

Fixed in v2.6.1

- Fix double free corruption in `RNAdos`
- Fix compilation issues due to use of `uint` instead of `unsigned int` for `RNAxplorer`
- Fix compilation issues for `RNAxplorer` when OpenMP is unavailable

Changed in v2.6.1

- AUTOCONF: Update autoconf macros
- Update Debian-based packaging rules

6.3.5 Version 2.6.0

Release date: *2023-06-09*

Fixed in v2.6.0

- Fix missing strand separators in `RNAsubopt` when applied to multiple interacting sequences
- Fix sorted output in `RNAsubopt` with `--gquad` option
- Fix default value of `RNAinverse -R` option in manpage
- Fix alignment input parsing in `refold.pl`
- API: Fix potential problems in free_dp_matrices() of LPfold.c
- API: Fix z-score initialization in `vrna_Lfoldz()` amd `vrna_mfe_window_zscore_cb()`
- API: Fix file close issue in `vrna_file_commands_read()`
- API: Fix backtracking issue in Zuker subopt
- API: Fix missing soft constraints callback execution in Zuker subopt
- API: Fix enumeration of G-quadruplexes in `vrna_subopt()` and `vrna_subopt_cb()`
- API: Fix constraints bug for exterior loop in boltzmann sampling
- API: Fix discrepancy between global and local folding in how hard constraints for unpaired bases and non-specific pairing are applied
- SWIG: Fix Python 3 wrapper suffix issue
- SWIG: Fix Perl 5 wrapper for `vrna_ud_prob_get()`
- Fix compilation issues with RNAforester
- AUTOCONF: Fix requirement checks when SVM support is deactivated and swig is missing

Changed since v2.6.0

- Allow for only `-Fp` in `RNAinverse` instead of always activating `-Fm`
- Restructure `--*help` output and man pages for most executable programs
- Allow for at least as many threads as CPUs are configured if maximum thread number detection fails
- API: Extend `model_details` to allow for salt concentration changes
- API: Refactor function typdefs to make them actual function pointer typedefs
- SWIG: Only accept upper triangular part of matrix input in fc.sc_bp_add()
- SWIG: Use var_array instead of tuples for Python RNA.ptable()
- Update `libsvm` to version 3.31
- Update `dlib` to version 19.24
- Adapt Debian dependencies
- AUTOMAKE: Add `auto` parameters for `-flto` compile/link flags
- AUTOCONF: Require C++17 due to dependencies to compile `DLIB`
- AUTOCONF: Deactivate Python 2 bindings by default

Added in v2.6.0

- Add modified base input support to `RNAfold`
- Add modified base input support to `RNAplfold`
- Add modified base input support to `RNALfold`
- Add modified base input support to `RNAcofold`
- Add modified base input support to `RNAsubopt`
- Allow for cation concentration (Na+) changes in most executable programs (default 1.021M)
- Add `RNAxplorer` program to the distribution
- API: Add functions for salt concentration change derived energy corrections in `ViennaRNA/params/salt.h`
- API: Add arbitrary modified base support (`vrna_sc_mod()`) via soft constraints mechanism and JSON input data
- API: Add Pseuoduridine-A parameters via soft constraints callback
- API: Add Dihydrouridine parameters via soft constraints callback
- API: Add inosine-U and inosine-C parameters via soft constraints callback
- API: Add m6A parameters via soft constraints callback mechanism
- API: Add 7DA modification support via soft constraints
- API: Add Purine (nebularine) modification support
- API: Add new soft constraints multi-callback dispatcher
- API: Add dynamic array data structure utilities
- API: Add string data structure utilities
- API: Add `vrna_strchr()` function
- API: Allow for enforcing 'must pair' constraint (`|`) in dot-bracket constraints strings
- SWIG: Add Python wrapper for `vrna_move_neighbor_diff()`
- SWIG: Add Python docstrings generated from doxygen documentation of C-library

6.4 Version 2.5.x

6.4.1 Version 2.5.1

Release date: *2022-06-02*

Fixed in v2.5.1

- API: Fix MEA computation for G-quadruplex predictions
- API: Fix memory leak in hard constraints container
- API: Fix RNApuzzler edge-case that resulted in segmentation faults
- API: Fix invalid memory access in `vrna_strjoin()`
- SWIG: Fix RNG wrapping and initialize RNG upon module load and update associated functions
- SWIG: Fix memory leak in wrapper for `vrna_db_from_ptable()`
- API: Deprecate cutpoint member of vrna_fold_compound_t and prepare for 5'/3' encoding
- AUTOCONF: Check whether we can build the swig interface when SVM support is deactivated
- AUTOCONF: Fix condition check for CLA build

Changed since v2.5.1

- Refactor `ct2db` program to allow for pseudoknots in output structure
- API: Revisit generic soft constraints for sliding-window base pair probability computations
- API: Enable to overwrite automatic unpaired probability determination in MEA computation
- API: Update sanity checks for input in `vrna_pbacktrack_sub*()`
- API: Do not use `min_loop_size = 0` for multi strand interaction prediction
- API: Remove unnecessary uses of `min_loop_size` at multiple locations
- API: Refactor sequence addition/preparation for `vrna_fold_compound_t`
- DOC: Update documentation
- SWIG: Shadow pair table data structure for efficient interactions between C and target languages
- SWIG: Expose hard constraints members in `fold_compound` objects
- SWIG: Expose DP matrices within `fold_compound` objects
- Update dlib to version 19.23
- DOC: Update doxygen.conf for version 1.9.2
- AUTOCONF: Factor-out Naview layout algorithm to allow for deactivating the Naview layout algorithm at configure-time
- AUTOCONF: Make LaTeX checks more portable and update LaTeX package checks

Added in v2.5.1

- API: Add `#VRNA_PLIST_TYPE_UNPAIRED` and `#VRNA_PLIST_TYPE_TRIPLE` identifiers for `vrna_ep_t`
- API: Add `vrna_init_rand_seed()` to initialize RNG with seed
- API: Add `vrna_zsc_compute_raw()` to obtain mean and sd for Z-score computation
- API: Add `vrna_file_connect_read_record()` function to parse connectivity table (`*.ct`) files
- API: Add `vrna_strtrim()` function
- API: Allow for pseudo-knots in `vrna_db_from_ptable()`

- SWIG: Add simple dot-plot file wrapper `plot_dp_EPS()`
- SWIG: Add `sequence`, `sequence_encoding` and `sequence_encoding2` attributes to `fold_compound` objects
- SWIG: Add more access to member variable arrays for various objects used throughout the library
- SWIG: Add memory efficient wrapper for dynamically allocated arrays and matrices
- SWIG: Add `exp_E_ext_stem()` method (`vrna_exp_E_ext_stem()`) to `fold_compound` objects

6.4.2 Version 2.5.0

Release date: *2021-11-08*

Fixed in v2.5.0

- Fix computation of BB homodimer base pair probabilities in `RNAcofold`
- API: Fix use of undefined values in deprecated function `PS_dot_plot()`
- API: Fix probability computations for unstructured domains within multibranch loops
- API: Fix index error in ensemble defect computations
- API: Fix hard constraints behavior on non-specific base pairing
- API: Fix segmentation fault for short input sequences in `vrna_hx_from_ptable()`
- API: Fix memory leak in static `rna_layout()` function
- API: Fix corner-case in covariance score computation on sequence alignments that determines which alignment columns may pair and which don't
- SWIG: Fix Python 3 file handle as optional argument in `eval*` functions and methods

Changed since v2.5.0

- Replace `RNAcofold` with a wrapper to `RNAmultifold`
- API: Rewrite Zuker suboptimals implementation
- API: Make `type` attribute of `vrna_mx_mfe_t` and `vrna_mx_pf_t` a constant
- API: Guard more functions in `utils/structure_utils.c` against `NULL` input
- API: Rename `vrna_E_ext_loop()` to `vrna_eval_ext_stem()`
- API: Use v3 typedefs in dot-plot function declarations
- Update `libsvm` to version 3.25
- Make Python 3.x the default Python for the scripting languange interfaces

Added in v2.5.0

- Add `RNAmultifold` program to compute secondary structures for multiple interacting RNAs
- Add multistrand capabilities to `RNAeval`
- Add multistrand capabilities to `RNAsubopt`
- API: Add MFE computations for multiple interacting strands
- API: Add partition function computations for multiple interacting strands
- API: Add base pair probability computations for multiple interacting strands
- API: Add suboptimal structure prediction for multiple interacting strands
- API: Add multistrand capabilities to `vrna_eval*()` functions

- API: Add new function `vrna_equilibrium_conc()` fir concentration dependency computations of multiple interacting strands with `dlib` backend
- API: Add `vrna_equilibrium_constants()` function to obtain equilibrium constants for different complexes of multiple interacting strands
- API: Add function `vrna_pf_add()` to add ensemble free energies of two ensembles
- API: Add function `vrna_pf_substrands()` to get ensemble free energies for complexes up to a specific number of interacting strands
- API: Add function `vrna_n_multichoose_k()` to obtain a list of k-combinations with repetition
- API: Add `vrna_cstr_discard()` function to allow for discarding char streams prior to flushing
- API: Add `vrna_bp_distance_pt()` function to allow for base pair distance computation with pseudo-knots
- API: Add functions `vrna_pbacktrack_sub*()` to allow for stochastic backtracing within arbitrary sequence intervals
- API: Add functions `vrna_boustrophedon()` and `vrna_boustrophedon_pos()` to generate lists of or obtain values from sequences of Boustrophedon distributed integer numbers
- API: Add `vrna_pscore()` and `vrna_pscore_freq()` functions to obtain covariance score for particular alignment columns
- SWIG: Add wrapper for `vrna_pf_add()`
- SWIG: Add wrapper for `vrna_hx_from_ptable()`
- SWIG: Add wrapper for `vrna_db_from_probs()`
- Add Python3 capability for Mac OS X installer builds
- TESTS: Create TAP driver output for all unit tests (library, executables, SWIG interfaces)
- Add Contributors License Agreement (CLA) to the Package in `doc/CLA/`

Removed in v2.5.0

- API: Remove old cofold implementations
- Remove compile-time switch to deactivate Boustrophedon backtracing scheme (this is the status-quo now)

6.5 Version 2.4.x

6.5.1 Version 2.4.18

Release date: *2021-04-22*

Fixed in v2.4.18

- Fix and refactor `RNApkplex` program
- Fix occasional backtracing errors in `RNALalifold`
- Prevent segmentation faults upon bogus input data in `RNAfold`, `RNAalifold`, `RNAcofold`, `RNAheat`, and `RNAeval`
- API: Fix energy contributions used in `RNAPKplex` implementations
- API: Fix energy evaluation for cofolding with dangle model 1
- API: Fix wrong arithmetic usage for PF variant of combined generic and simple soft constraints applied to external loops
- API: Fix memory size in #vrna_fold_compound_t initialization
- API: Fix bogus memory access for comparative prediction when preparing hard constraints

- API: Fix wrong index usage in hard constraints for comparative base pair probability computations of internal loops
- API: Fix G-Quadruplex contributions as part of multibranch loops in single sequence base pair probability computations
- API: Fix multibranch loop MFE decomposition step for multiple strand cases
- API: Fix external loop generic hard constraint index updating for partition function computations
- API: Fix memory allocation for auxiliary grammar data structure
- API: Fix incorporation of auxiliary grammar contrib for closing pairs in sliding-window MFE computation
- API: Fix DP matrix intitialization in sliding window MFE computations (fixes occasional backtracing issues in comparative sliding-window MFE computations)
- SWIG: Fix memory leak in wrapper for `vrna_probs_window()`
- SWIG: Refactor and therefore fix soft constraint binding functions for use in comparative structure predictions
- SWIG: Fix typo that prevented properly wrapping `vrna_params_load_RNA_Andronescu2007()`

Changed since v2.4.18

- Restrict available dangling end models in `RNALalifold` to 0 and 2
- Free MFE DP matrices in `RNAsubopt` Boltzmann sampling when not required anymore
- API: Refactor `RNAPKplex` implementation by better using constraints framework and moving out many parts from `RNAPKplex.c` into `RNAlib` as separate re-usable functions
- API: Make `vrna_sc_t.type` attribute a constant
- API: Remove upper-triangular hard constraint matrix in favor of full matrix
- API: Always ensure sane base pair span settings after `vrna_fold_compound_prepare()`
- API: Return INF on predictions of `vrna_mfe_dimer()` that fail due to unsatisfiable constraints
- API: Rename internally used hard and soft constraints API symbols
- API: Fix header file inclusions to prevent #include cycles
- SWIG: Unify wrappers for `vrna_ptable()` and `vrna_ptable_from_string()`
- REFMAN: Refactored structure annotation documentation
- REFMAN: Update Mac OS X install section
- Replace `DEF` placeholders in energy parameter files with their value of -50
- Update `RNAlocmin` subpackage to properly compile with more stringent C++ compilers
- Update `RNAforester` subpackage to properly compile with more stringent C++ compilers
- Update autotools framework, e.g. checks for pthreads
- Update universal binary build instructions for Mac OS X builds to enable ARM compilation for M1 CPUs

Added in v2.4.18

- API: Add `vrna_abstract_shapes()` and `vrna_abstract_shapes_pt()` functions to convert secondary structures into their respective abstract shape notation ala Giegerich et al. 2004
- API: Add functions `vrna_seq_reverse()` and `vrna_DNA_complement()` to create reverse complements of a sequence
- API: Add more soft constraint handling to comparative structure prediction
- API: Add generic soft constraints for sliding window comparative MFE backtracing

- API: Add `vrna_ensemble_defect_pt()` that accepts pair table input instead of dot-bracket string to allow for non-nested reference structures
- API: Add failure/success return values to generic soft constraints application functions
- SWIG: Add wrapper for `vrna_file_fasta_read_record()`

6.5.2 Version 2.4.17

Release date: *2020-11-25*

Fixed in v2.4.17

- Fix `RNAup -b` mode with shorter sequence first
- API: Fix multiloop backtracing with soft constraints for unpaired positions in `vrna_subopt()` and `vrna_subopt_cb()`
- API: Fix parameter parse in `vrna_params_load_from_string()`
- SWIG: Fix typo in interface functions to load DNA parameters
- Fix Debian and Windows installer files

Changed since v2.4.17

- API: Make `vrna_md_t` argument to vrna_fold_compound() a constant pointer
- API: Remove missing symbols from header file `ViennaRNA/params/default.h`
- API: Refactor z-score threshold filter handling for sliding-window MFE prediction

Added in v2.4.17

- Add `--backtrack-global` option to `RNALfold` (currently only available for dangles == 2 | 0)
- Add `--zscore-pre-filter` and `--zscore-report-subsumed` options to `RNALfold`
- API: Add `vrna_heat_capacity()` and `vrna_head_capacity_cb()` functions to RNAlib
- API: Add backtracing function `vrna_backtrack_window()` for global MFE structure to sliding-window predictions
- API: Add SVG support for `RNApuzzler` structure layouts
- SWIG: Add python-3.9 autoconf checks
- SWIG: Add `vrna_head_capacity*()` wrappers
- SWIG: Add access to raw energy parameters
- SWIG: Add `alias` and `pair` attribute to objects of type `md`
- SWIG: Add out/varout typemaps for 2-dimensional int-like arrays
- SWIG: Add all data fields to objects of type 'param' and 'exp_param'

6.5.3 Version 2.4.16

Release date: *2020-10-09*

Fixed in v2.4.16

- Fix backtracing errors in `RNALalifold` for alignments with more than `32768` columns
- Fix backtracing errors in `RNAalifold` and `RNALalifold` for rare cases when two alignment columns may pair due to covariance score threshold but still yield infinite energies due to energy model

- API: Fix undefined behavior due to short int overflows when accessing alignment lengths with alignments larger than 32768 columns. This fixes occasional backtracing errors in `RNALalifold` and `vrna_mfe_window()`
- API: Fix adding pscore to base pairs that yield INF energy in comparative global and local MFE prediction

Changed since v2.4.16

- Refactored manpages/help options for `RNAplfold`, `RNAplot`, `RNApvmin`, `RNAsubopt`, and `RNAup`
- Move `units.h` include file to `ViennaRNA/utils/units.h`

Added in v2.4.16

- API: Add `vrna_convert_kcal_to_dcal()` and vice-versa function for safely converting integer to float representations of energy values
- SWIG: Add a reasonable `Python` interface for objects of type `vrna_path_t`
- SWIG: Add a wrapper for `vrna_seq_encode()`

6.5.4 Version 2.4.15

Release date: *2020-08-18*

Fixed in v2.4.15

- Fix compilation of `Kinfold` with GCC 10
- Prevent `RNAcofold` to process input with more than two strands
- API: Fix removal of strand delimiter in `vrna_plot_dp_PS_list()`
- API: Fix `vrna_enumerate_necklaces()`
- API: Fix bogus backtracing for co-folded structures in `vrna_subopt()` and `vrna_subopt_cb()`
- API: Fix storing co-folded structures for sorted output in `vrna_subopt()`
- API: Fix multibranch loop component hard constraints for multi-strand cases
- API: Prevent adding internal loop energy contributions to enclosed parts with `energy=INF`
- SWIG: Fix invalid memory access when using `MEA_from_plist()` in `Perl 5` or `Python`
- TESTS: Fix results dir path for out-of-tree builds

Changed since v2.4.15

- Update `Kinfold` to version `1.4`
- API: Adapt `vrna_db_pack()`/`vrna_db_unpack()` functions to produce comparable strings
- API: Make hard constrains strand-aware
- SWIG: Make attributes of objects of type `param` and `exp_param` read-only
- Add array of strand nicks to EPS dot plot files instead of single cutpoint
- Draw separator line for each strand nick in EPS dot-plots
- Update `libsvm` to version 3.24
- Disable Link-Time-Optimization (`LTO`) for third-party programs linking against `RNAlib` using `pkg-config`
- TESTS: Set default timeout for library tests to 20s

Added in v2.4.15

- Add `--en-only` flag to `RNAsubopt` to allow for sorting by energy only
- Add cutpoint marker to dot-plots created with `RNAcofold -a`
- API: Add sorting modes `VRNA_UNSORTED`, `VRNA_SORT_BY_ENERGY_LEXICOGRAPHIC_ASC`, and `VRNA_SORT_BY_ENERGY_ASC` to `vrna_subopt()`
- API: Add `vrna_strjoin()` function
- API: Add missing case to external loop hard constraints
- SWIG: Enable keyword argument features in `Python` interface of constructors for `fold_compound`, `md`, `move`, `param`, and `exp_param` objects
- SWIG: Enable autodoc feature for `Python` interface of constructors for `fold_compound`, `md`, and `move` objects
- SWIG: Enable `toString` conversion for `Python` interface for objects of type `fold_compound`, `md`, `move`, `params`, `exp_params`, and `subopt_solution`
- SWIG: Add (read-only) attributes `type`, `length`, `strands`, `params`, and `exp_params` to objects of type `fold_compound`

6.5.5 Version 2.4.14

Release date: *2019-08-13*

Fixed in v2.4.14

- Fix `RNApvmin` pertubation vector computation
- API: Fix memory leak in `vrna_path_gradient()`
- API: Fix release of memory fir `vrna_sequence_remove_all()`
- API: Fix soft-constraints application in `vrna_sc_minimize_pertubation()` that prevented proper computation of the pertubation vector
- REFMAN: Fix LaTeX code in `units.h` that prevented proper compilation with `pdflatex`

Changed since v2.4.14

- API: Prepare sequence data in `vrna_fold_compound_t` with `vrna_sequence_add()`
- API: Use `vrna_pbacktrack_num()` instead of `vrna_pbacktrack()` in `vrna_sc_minimize_pertubation()` to speed-up sample generation
- Reduce use of global variable `cut_point` in `RNAlib`
- SWIG: Use `importlib` in favor of `imp` to determine Python 3 tag extension
- SWIG: Update various wrapper functions
- Update RNA Tutorial
- Colorize final configure script message
- REFMAN: Always use `pdflatex` to compile reference manual and tutorial

Added in v2.4.14

- Add non-redundant sampling option to `RNApvmin`
- Add `RNAdos` program to compute density of states
- Add `-P DNA` convenience command line parameter to most programs to quickly load DNA parameters without any input file
- MAN: Add example section to man-page of `RNAalifold`

- API: Add 5' and 3' neighbor nucleotide encoding arrays and name string to `vrna_seq_t`
- API: Add new data structure for multiple sequence alignments
- API: Add `vrna_sequence_order_update()` function
- API: Add non-redundant sampling mode to `vrna_sc_minimize_pertubation()` through passing negative sample-sizes
- API: Add v3.0 API functions for maximum expected accuracy (MEA) computation
- API: Include energy parameter sets into `RNAlib` and provide functions to load them at runtime
- SWIG: Add wrappers for MEA computation with `vrna_MEA()` and `vrna_MEA_from_plist`
- SWIG: Add wrappers for `vrna_pr_structure()` and `vrna_pr_energy()`
- Add an R script to create 2D landscape plots from `RNA2Dfold` output
- Add `gengetopt` to configure-time requirements to build man-pages
- Add new energy parameter file `rna_misc_special_hairpins.par` with additional UV-melting derived parameters for Tri- and Tetra-loops
- EXAMPLES: Add Python script that performs computations equivalent to `RNAfold -p --MEA`

6.5.6 Version 2.4.13

Release date: *2019-05-30*

Fixed in v2.4.13

- Fix centroid structure prediction for `RNAcofold`
- Fix `--noLP` option for `RNALalifold`
- API: Refactor and fix collision handling in `vrna_hash_table_t`
- API: Fix one access using wrong index for odd dangles in `loops/external.c`
- API: Add two missing `MLbase` contributions for MFE prediction in `loops/multibranch.c`
- API: Fix invalid memory access for lonely pair mode (`--noLP`) in comparative sliding-window MFE prediction
- SWIG: Fix access to global variable `pf_smooth` and `pf_smooth` attribute in `model_details` object
- SWIG: Fix Python reference counting for `Py_None` in `interfaces/findpath.i` wrapper
- Fix `RNApuzzler` header file installation
- SWIG: Fix wrapper building for `swig >= 4.0.0`

Changed since v2.4.13

- API: Refactor multiloop MFE backtracking for odd dangles
- API: Reduce usage of global macro `TURN` by replacing it with `min_loop_size` field of `vrna_md_t`
- SWIG: Refactor reference counting for all Python2 and Python3 wrappers
- REFMAN: Larger updates and restructuring of reference manual
- Install example scripts and source code files, e.g. to $prefix/share/ViennaRNA/examples
- Properly pass GSL, PTHREADS, and MPFR flags to sub-projects
- SWIG: Include Python 3.7 and 3.8 in list of autoconf-probed python interpreters

Added in v2.4.13

- API: Add function `vrna_backtrack5()` to allow for MFE backtracking of sub-sequences starting at the 5'-end
- API: Add functions `vrna_path_direct()` and `vrna_path_direct_ub()` that may also return move lists instead of dot-bracket lists
- API: Add functions `vrna_pt_pk_remove()` and `vrna_db_pk_remove()` that remove pseudoknots from an input structure

6.5.7 Version 2.4.12

Release date: *2019-04-16*

Fixed in v2.4.12

- API: Fix declaration of `vrna_move_apply_db()`
- API: Fix `vrna_path()` lexicographical ordering in gradient walks
- Ensure proper rescaling of energy parameters upon temperature changes

Changed since v2.4.12

- Update Ubuntu/Debian and OpenSUSE build instructions
- Reduce intra-package dependency on non-v3.0 API
- Update help/man page of `RNALfold`
- API: Refactor and speed-up gradient walk implementation available as `vrna_path_gradient()`
- API: Substitute `vrna_file_PS_aln_sub()` alignment plot function by `vrna_file_PS_aln_slice()` that actually slices out a sub-alignment
- API: Rename `vrna_annotate_covar_struct()` to `vrna_annotate_covar_db()` and add new function `vrna_annotate_covar_db_extended()` to support more bracket types
- API: Calling `vrna_params_reset()` now implies a call to `vrna_exp_params_reset()` as well
- API: Move landscape implementations into separate directory, thus headers should be included as <ViennaRNA/landscape/move.h>, <ViennaRNA/landscape/neighbor.h>, etc.
- Refactor soft constraints implementation in stochastic backtracing
- SWIG: Refactor stochastic backtracing interface(s)
- SWIG: Sanitize alignment plot function interface(s)

Added in v2.4.12

- Add non-redundant stochastic backtracing option for `RNAalifold`
- Add `--noDP` option to suppress dot-plot output in `RNAfold` and `RNAalifold`
- Add `RNApuzzler` (4) and `RNAturtle` (3) secondary structure layout algorithm options to `RNAfold` and `RNAplot`
- Allow for multiple input files and parallel input processing in `RNAheat`
- API: Enable non-redundant stochastic backtracing for comparative structure prediction
- API: Enable stochastic backtracing for circular comparative structure prediction
- API: Enable stochastic backtracing of subsequences (5' prefixes) for comparative structure prediction
- API: Add `pf_smooth` attribute to `vrna_md_t` data stucture to allow for disabling Boltzmann factor energy smoothing
- API: Add functions to allow for resuming non-redundant stochastic backtracing

- API: Add functions to retrieve multiple stochastically backtraced structures (list and callback variants)
- API: Add `vrna_positional_entropy` to compute vector of positional entropies
- API: Add `RNApuzzler` and `RNAturtle` secondary structure layout algorithm (Wiegreffe et al. 2018)
- API: Add v3.0 API for secondary structure layout/coordinate algorithms
- API: Add more helper/utility functions for `vrna_move_t` data structures
- API: Add callback-based neighborhood update function for (subsequent) `vrna_move_t` application
- API: Add abstract heap data structure available as <ViennaRNA/datastructures/heap.h>
- SWIG: Wrap all non-redundant stochastic backtracing functions to scripting language interface(s)
- SWIG: Add proper constructor for objects of type `vrna_ep_t`

6.5.8 Version 2.4.11

Release date: *2018-12-17*

Fixed in v2.4.11

- API: Fix wrong access to base pair soft constraints in equilibrium probability computations
- API: Fix behavior of `vrna_nucleotide_encode()` with lowercase characters in sequence
- API: Fix behavior of `encode_char()` with lowercase characters in sequence
- API: Fix forbidden `GU` pairs behavior in pscore computation for comparative folding
- API: Fix potential errors due to uninitialized `next` pointers in `vrna_move_t` of `vrna_eval_move_shift_pt`

Changed since v2.4.11

- API: Change behavior of `vrna_cstr_free()` and `vrna_cstr_close()` to always flush output before unregistering the stream
- Activate compilation for compile-time supported SIMD optimized implementations by default
- Replace `--enable-sse` configure script option with `--disable-simd`

Added in v2.4.11

- Add `--commands` option to `RNAsubopt`
- Add non-redundant Boltzmann sampling mode for `RNAsubopt`
- API: Add `AVX 512` optimized version of MFE multibranch loop decomposition
- API: Add functions for CPU SIMD feature detection
- API: Add dispatcher to automatically delegate exterior-/multibranch loop MFE decomposition to supported SIMD optimized implementation
- API: Add function `vrna_dist_mountain()` to compute mountain distance between two structures
- API: Add function `vrna_ensemble_defect()` to compute ensemble defect given a target structure
- API: Add non-redundant Boltzmann sampling
- SWIG: Add interface for `vrna_loopidx_from_ptable()`

6.5.9 Version 2.4.10

Release date: *2018-09-26*

Fixed in v2.4.10

- Fix wrong output filename for binary opening energies in `RNAplfold`
- Fix broken SSE4.1 support for multibranch loop MFE computation that resulted in increased run times
- Fix redundant output issue in subopt backtracking with unusually high delta energies (>=INF)
- Restore default behavior of '|' symbol in dot-bracket hard constraint strings that got lost with version 2.2.0
- SWIG: Fix invalid memory access in `subopt()` method of fold_compound object when writing to file

Changed since v2.4.10

- Change default linker- and loop length computations for G-Quadruplex predictions in comparative prediction modes
- Add hard constraints warning for base pairs that violate the `min_loop_size` of the model
- Update `libsvm` to version 3.23
- API: Replace upper-triangular hard constraints matrix with full matrix for cache-optimized access
- Add `-ftree-vectorize` compile flag by default if supported

Added in v2.4.10

- Enable G-Quadruplex support for partition function computation in `RNAalifold`
- Add faster (cache-optimized) version of Nussinov Maximum Matching algorithm
- API: Add functions to set auxiliary grammar extension rules
- API: Add G-Quadruplex prediction support for comparative partition function
- SWIG: Add wrapper for Nussinov Maximum Matching algorithm

Removed in v2.4.10

- API: Remove `VRNA_GQUAD_MISMATCH_PENALTY` and `VRNA_GQUAD_MISMATCH_NUM_ALI` macros

6.5.10 Version 2.4.9

Release date: *2018-07-11*

Fixed in v2.4.9

- Fix interactive mode behavior for multiple sequence alignment input in `RNAalifold`, `RNALalifold`
- Fix parameter order bug in `vrna_path_findpath*` functions that resulted in too large search widths
- Fix wrong application of base pair soft constraints in partition function computations
- Fix position ruler string in EPS alignment output files
- Fix MFE backtracking errors that might appear under specific hard constrained base pair patterns

Changed since v2.4.9

- Warn on unsatisfiable hard constraints from dot-bracket string input in `RNAfold`, `RNAcofold`, and `RNAalifold`
- Refrain from reading anything other than `#=GC SS_cons` to retrieve structures when parsing Stockholm 1.0 format
- Complete soft constraints additions to Boltzmann sampling implementation for single sequences
- Allow for disabling alignment wrapping in `vrna_file_PS_aln*` functions

- Do not remove G-Quadruplex annotation from WUSS formatted structure strings upon calls to `vrna_db_from_WUSS`
- Speed-up backward compatibility layer for energy evaluation functions that unnecessarily slowed down third-party tools using the old API
- Remove implicit `exit()` calls from global MFE backtracking implementation.

Added in v2.4.9

- Allow for Stockholm formatted multiple sequence alignment input in `RNAeval` and `RNAplot`
- Allow for multiple input files in `RNAeval` and `RNAplot`
- Allow for parallel processing of input batch jobs in `RNAeval` and `RNAplot`
- Add `-g` option to activate G-Quadruplex support in `RNAheat`
- Allow for passing dot-bracket strings with `'&'` strand-end identifier to simple `vrna_eval_*` functions
- Enable G-Quadruplex related average loop energy correction terms in verbose output of `vrna_eval_*` functions

6.5.11 Version 2.4.8

Release date: *2018-06-23*

Fixed in v2.4.8

- Fix compilation of RNAforester with C++17 standard
- Fix tty input detection in RNAcofold
- Fix bad memory access with RNAcofold -p
- API: Fix incorrect unpaired probability computations in vrna_probs_window()
- API: Fix potential out-of-bounds access situations (for circular RNA folding) in eval.c
- API: Fix comparative exterior internal loop partition function computation for circfold
- SWIG: Fix false-positive use of uninitialized value in Python3/file_py3.i

Added in v2.4.8

- TESTS: Add tests for special features in RNAalifold
- TESTS: Add test case for RNAcofold -p

6.5.12 Version 2.4.7

Release date: *2018-06-13*

Fixed in v2.4.7

- API: Fix discrepancy between comparative and single sequence –noLP predictions
- SWIG: Fix configure-time check for Python 3 interface build
- SWIG: Fix Python 3 IO file stream to C FILE * conversion

Changed since v2.4.7

- Include ligand binding energies in centroid and MEA structure output of RNAfold
- Refactor ct2db program to process multiple structures from single .ct file
- API: Enable processing of comparative fold_compound with vrna_pr_*() functions

- API: Refactor vrna_ostream_t to enable NULL input in vrna_ostream_provide()
- API: Major refactoring in loop energy evaluations (MFE and PF)
- API: Make vrna_mx_pf_aux_el_t and vrna_mx_pf_aux_ml_s opaque pointers
- API: Make fold_compound field `type` a const attribute
- API: Refactor MFE post-processing for circular RNAs
- API: Remove major part of implicit exit() calls in RNAlib
- API: Make vrna_cmd_t an opaque pointer
- API: Move headers for constraints, datastructures, io, loop energy evaluation, energy parameters, plotting, search, and utilities into separate subdirectories (backward compatibility is maintained)
- SWIG: Refactor callback execution for Python 2 / 3 interface to reduce overhead
- Cosmetic changes in final configure notice
- Major changes in source tree structure of the library
- Add autoconf checks for maintainer tools
- Generate C strings from static PostScript files at configure time (for structure- and dot plots)
- REFMAN: Large updates in API documentation and structure of reference manual

Added in v2.4.7

- Allow for parallel processing across multiple input files in RNAfold
- Allow for arbitrary number of input files in RNAalifold
- Allow for parallel processing of input data in RNAalifold
- Allow for arbitrary number of input files in RNAcofold
- Allow for parallel processing of input data in RNAcofold
- Enable parallel processing in RNAfold, RNAcofold, RNAalifold for MS Windows build
- Add centroid and MEA structure computation to RNAcofold
- Add configure time check for LTO capabilities of the linker
- API: Add motif name/id support for unstructured domains
- API: Add implementations of Boyer-Moore-Horspool search algorithm
- API: Add implementations to determine number of rotational symmetry for strings (of objects)
- API: Add hash table data structure
- API: Add functions to replace 'old API' interface of RNAstruct.h
- API: Add functions to replace 'old API' interface of aln_util.h
- API: Add generic soft constraints support to suboptimal structure prediction sensu Wuchty et al.

6.5.13 Version 2.4.6

Release date: *2018-04-19*

Fixed in v2.4.6

- Stabilize rounding of free energy output in RNAalifold
- API: Fix potential rounding errors for comparative free energies in eval.c and mfe.c
- API: Fix regression in exterior loop dangling end contributions for comparative base pair probabilities and Boltzmann sampling (introduced with v2.4.4)

- API: Fix regression with hard constrained base pairs for comparative structure prediction (introduced with v2.4.4)

Changed since v2.4.6

- TESTS: Ignore 'frequency of MFE structure' in RNAcofold partition function checks

Added in v2.4.6

- TESTS: Add basic tests for RNAalifold executable

6.5.14 Version 2.4.5

Release date: *2018-04-17*

Fixed in v2.4.5

- Fix unstructured domain motif detection in MFE, centroid, and MEA structures computed by RNAfold
- API: Fix bug in sliding-window partition function computation with SHAPE reactivity and Deigan et al. conversion method
- API: Fix application of '<' and '>' constraint symbols in dot-bracket provided constraints (was broken since v2.4.2)
- API: Fix MEA structure computation in the presence of unstructured domains
- API: Stabilize order of probability entries in EPS dot-plot files
- Fix compiler warnings on wrong type of printf() in naview.c
- Stabilize parallel make of Mac OS X installer
- SWIG: Fix 'next' is a perl keyword warnings for Perl5 wrapper

Changed since v2.4.5

- Limit allowed set of commands in command file for RNAcofold to hard and soft constraints
- Define VRNA_VERSION macro as string literal and add macros for major, minor, and patch numbers
- SWIG: Catch errors and throw execptions whenever scripting language provided callback functions are not applicable or fail

Added in v2.4.5

- Allow for arbitrary number of input files in RNAfold
- Allow for parallel processing of input data in RNAfold (UNIX only, no Windows support yet)
- Add SHAPE reactivity support through commandline options for RNAplfold
- API: Add functions to compute equilibrium probability of particular secondary structures
- API: Add dynamic string stream data type and associated functions
- API: Add priority-queue like data structure with unordered fill capability and ordered output callback execution
- API: Add functions to detect unstructured domain motifs in MFE, centroid, and MEA structures
- Add energy parameter set from Langdon et al. 2018
- Add autoconf checks for POSIX threads compiler/linker support
- SWIG: Add keyword arguments and autodoc feature for Python/Python3 wrappers

6.5.15 Version 2.4.4

Release date: *2018-03-06*

Fixed in v2.4.4

- Fix bug in interior-like G-Quadruplex MFE computation for single sequences
- Fix autoconf switch to enable deprecation warnings
- Fix bug in eval_int_loop() that prevented propagation of energy evaluation for loops with nick in strands
- Fix several bugs for SHAPE reactivity related comparative partition function computations
- Fix annotation of PostScript output for soft-constraint derived ligand binding motifs in RNAfold
- Fix constraint indices for multibranch loops in unpaired probability computations of LPfold.c
- Fix dangling end contributions in comparative partition function for exterior loops
- Minor fixes in Boltzmann sampling implementation
- SWIG: Fix wrappers for findpath() implementation
- SWIG: Fix configure-time check of Perl5 interface build capabilities

Changed since v2.4.4

- Change verbose output for soft-constraints derived ligand binding motifs in RNAfold
- Allow for lowercase letters in ct2db input
- API: Move concentraton dependent implementation for co-folding to separate compile unit
- Major restucturing and constraints feature additions in loop type dependent energy evaluation functions
- Major restructuring in MFE implementations
- Major restructuring in PF implementations
- REFMAN: Heavy re-ordering of the RNAlib reference manual

Added in v2.4.4

- API: Add simplified interface for vrna_pf_dimer()
- API: Add new API functions for exterior loop evaluations
- API: Add simplified interfaces for energy evaluation with G-Quadruplexes and circular RNAs
- API: Add findpath functions that allow for specification of an upper bound for the saddle point
- Add configure-time linker check for Python3 interface
- Add automatic CPP suggestions for deprecated function substitutes
- SWIG: Add tons of energy evaluation wrappers
- SWIG: Wrap functions from walk.c and neighbor.c
- DOC: Add some missing references to manpages of executable programs

6.5.16 Version 2.4.3

Release date: *2017-11-14*

Fixed in v2.4.3

- Fix handling of dangling end contribution at sequence boundaries for sliding window base pair probability computations
- Fix handling of base pair hard constraints in sliding-window implementations
- Fix sliding-window pair probability computations with multibranch-loop unpaired constraints
- Fix sliding-window non-specific base pair hard constraint implementation
- Fix probability computation for stochastic backtracking in RNAsubopt –stochBT_en output
- Fix regression in comparative structure prediction for circular RNAs
- Fix LDFLAGS for scripting language interfaces in corresponding Makefiles

Changed since v2.4.3

- Stabilize partition function scaling by always using sfact scaling factor from model details
- Prevent RNAplfold from creating inf/-inf output when solution set is empty with particular hard constraints

Added in v2.4.3

- Add –pf_scale commandling parameter to RNAplfold
- Add constraint framework for single sequence circular RNA structure prediction
- Add RNAfold test suite to check for working implementation of constraints for circular RNAs
- Add a brief contribution guideline CONTRIBUTING.md
- Include RNAforester v2.0.1

6.5.17 Version 2.4.2

Release date: *2017-10-13*

Fixed in v2.4.2

- Fix G-Quadruplex energy corrections in comparative structure energy evaluations
- Fix discrepancy in comparative exterior loop dangling end contribution of eval vs. MFE predictions
- Fix regression in RNAup unstructuredness and interaction energy computations
- Fix sequence length confusions when FASTA input contains carriage returns
- Fix build problems of RNAlocmin with older compilers
- Fix sliding-window hard constraints where single nucleotides are prohibited from pairing
- Fix dot-bracket output string length in sliding-window MFE with G-Quadruplexes
- Fix unpaired probability computations for separate individual loop types in LPfold.c
- Fix bad memory access in RNAsubopt with dot-bracket constraint
- REFMAN: Fix formula to image conversion in HTML output

Changed since v2.4.2

- Use the same model details for SCI computations in RNAalifold
- Abort computations in vrna_eval_structure_v() if structure has unexpected length
- Use original MSA in all output generated by RNAalifold and RNALalifold
- API: Do not convert sequences of input MSA to uppercase letters in vrna_file_msa_read_record()
- API: Rename vrna_annotate_bp_covar() and vrna_annotate_pr_covar()

Added in v2.4.2

- Add full WUSS support for –SS_cons constraint option in RNAalifold
- Add commandline option to RNALalifold that enables splitting of energy contributions into separate parts
- Add missing hard constraint cases to sliding-window partition function implementation
- Add CSV output option to RNAcofold
- API: Add new functions to convert dot-bracket like structure annotations
- API: Add various new utility functions for alignment handling and comparative structure predictions
- API: Add function vrna_strsplit() to split string into tokens
- API: Add new noLP neighbor generation
- SWIG: Add wrapper for functions in file_utils_msa.h
- SWIG: Add wrappers for vrna_pbacktrack() and vrna_pbacktrack5()
- SWIG: Add vrna_db_to_element_string() to scripting language interface

6.5.18 Version 2.4.1

Release date: *2017-08-23*

Fixed in v2.4.1

- Fix memory leak in fold_compound methods of SWIG interface
- Fix memory leaks in double ** returning functions of SWIG Perl5 interface
- Fix memory leak in vrna_ep_t to-string() function of SWIG interface
- Regression: Fix reverting pf_scale to defaults after vrna_exp_params_rescale()
- Regression: Fix homo-dimer partition function computation in RNAcofold

Added in v2.4.1

- Add unit tests for RNAcofold executable
- Add SHAPE reactivity support to RNAcofold
- Add SHAPE reactivity support to RNALalifold

6.5.19 Version 2.4.0

Release date: *2017-08-01*

Fixed in v2.4.0

- Several fixes with respect to G-Quadruplex prediction in sliding-window MFE recursions (single sequence and comparative implementation)
- Fix building PDF Reference manual with non-standard executable paths
- Fix redefinition of macro ON_SAME_STRAND() in subopt.c
- Fix dangling end issues in sliding-window MFE implementations
- Fix regression for –canonicalBPonly switch in RNAfold/RNAcofold/RNAsubopt
- Fix building sliding-window MFE implementation without SVM support
- Fix parsing of STOCKHOLM 1.0 MSA files that contain MSA spanning multiple blocks
- Fix Alidot link in RNAalifold manpage
- Fix wrong pre-processor flags when enabling single-precision PF computations

- Fix unit testing perl5 interface by including builddir/tests in PERL5LIB path
- Fix buffer overflow in hairpin loop sequence motif extraction for circular RNAs
- Fix out-of-bounds memory access in neighbor.c
- Restore capability to compile stand-alone findpath utility
- Restore capability to use non-standard alphabets for structure prediction
- Restore old-API random number functions in SWIG interface

Changed since v2.4.0

- Bump libsvm to version 3.22
- Print G-Quadruplex corrections in verbose mode of RNAeval
- Change behavior of RNAfold –outfile option to something more predictable
- Unify max_bp_span usage among sliding window prediction algorithms: RNAplfold, RNALfold, and RNALalifold now consider any base pair (i,j) with (j - i + 1) <= max_bp_span
- Change default behavior of adding soft constraints to a vrna_fold_compound_t (store only)
- Replace comparative sliding-window MFE recursions (All hits are reported to callback and can be filtered in a post-processing step)
- API: change data type of all constraint bit-flags from `char` to `unsigned char`
- API: change data type of a2s array in comparative structure prediction from `unsigned short` to `unsigned int`
- API: Change function parameter order in vrna_probs_window() to follow the style of other callback-aware functions in RNAlib
- Move sliding-window MFE implementations to new file mfe_window.c
- Allow additional control characters in MAF MSA input that do not end a block
- Improve reference manual
- Make functions in pair_mat.h static inline
- Prevent users from adding out-of-range base pair soft constraints
- Inline print functions in color_output.inc
- Re-write large portions of sliding-window PF implementation
- Clean-up SWIG unit test framework

Added in v2.4.0

- Add SHAPE reactivity data support to RNALfold
- Add commands-file support for RNALfold, RNAplfold (hard/soft constraints)
- Add RNAlocmin - Calculate local minima from structures via gradient walks
- Add RNA Bioinformatics tutorial (PDF version)
- Add hard constraints to sliding-window MFE implementations (RNALfold, RNALalifold)
- Add hard constraints to sliding-window PF implementations (RNAplfold)
- Add soft constraints to sliding-window MFE implementation for single sequences (RNALfold)
- Add soft constraints to sliding-window PF implementations (RNAplfold)
- Add SWIG interfaces for sliding-window MFE/PF implementations
- Add proper SWIG interface for alignment and structure plotting functions

- Add proper SWIG interface for duplexfold, duplex_subopt, and its comparative variants
- Add SWIG wrapper for vrna_exp_params_rescale()
- Add explicit destructor for SWIG generated vrna_md_t objects
- Add SWIG perl5 typemap for simple nested STL vectors
- Add dummy field in vrna_structured_domains_s
- Add note about SSE optimized code in reference manual
- Add SWIG interface for findpath implementation
- Add prepare() functions for ptypes-arrays and vrna_(exp_)param_t
- Add warnings for ignored commands in function vrna_commands_apply()
- Add callback featured functions for sliding window MFE and PF implementations
- Start documenting callback features in reference manual
- Introduce soft-constraint state flag

Removed in v2.4.0

- API: Remove E_mb_loop_stack() and introduce new function vrna_E_mb_loop_stack() as a replacement
- Remove obsolete scripts ct2b.pl and colorrna.pl from src/Utils directory
- Remove old RNAfold tutorial

6.6 Version 2.3.x

6.6.1 Version 2.3.5

Release date: *2017-04-14*

Fixed in v2.3.5

- Fix duplication of output filename prefix in RNAfold

Changed since v2.3.5

- Enable RNAfold unit tests to run in paralllel
- Enable users to turn-off base pair probability computations in RNAcofold with -a option
- Split move set in neighbor.c

Added in v2.3.5

- Add V3.0 API for sliding window partition function (a.k.a. RNAPLfold)
- Add G-Quadruplex prediction to RNALalifold
- Add SWIG wrappers for callback-based sliding window comparative MFE prediction
- Add SSE4.1 multiloop decomposition for single sequence MFE prediction

6.6.2 Version 2.3.4

Release date: *2017-03-10*

Fixed in v2.3.4

- Fix G-Quadruplex probability computation for single sequences
- Fix double-free when using SHAPE reactivity data in RNAalifold
- Fix out-of-bounds access in strand_number array
- Fix weighting of SHAPE reactivity data in consensus structure prediction when fewer data than sequences are present
- Fix z-score output in RNALfold

Changed since v2.3.4

- Substitute field name 'A0'/'B0' in data structure vrna_dimer_conc_s by 'Ac_start'/'Bc_start' to avoid clashes with termios.h (Mac OSX Python wrapper bug)
- Minimize usage of 'unsafe' sprintf() calls
- Enhance auto-id feature in executable programs
- Always sanitize output file names to avoid problems due to strange FASTA headers
- Lift restrictions of FASTA header length in RNAfold, RNAcofold, and RNAeval

Added in v2.3.4

- Add ViennaRNA/config.h with pre-processor definitions of configure time choices
- Add test-suite for RNAfold
- Add functions to procude colored EPS structure alignments
- Add function to write Stockholm 1.0 formatted alignments
- Add function to sanitize file names
- Add callback based implementation for sliding-window MFE prediction (single sequences, comparative structure prediction)
- Add fast API 3.0 implementations to generate structural neighbors and perform steepest descent / random walks (Thanks to Gregor!)
- Add parameter option to RNALalifold for colored EPS structure alignment and structure plot output
- Add parameter option to RNALalifold to write hits into Stockholm file
- Add parameter option to RNAalifold to write Stockholm 1.0 formatted output
- Add parameter option to RNAalifold to suppress stderr spam
- Add auto-id feature to RNAplot, RNALfold, RNAsubopt, RNAplfold, RNAheat
- Add SHAPE reactivity derived pseudo-energies as separate output in RNAalifold
- Add colored output to RNA2Dfold, RNALalifold, RNALfold, RNAduplex, RNAheat, RNAinverse, RNAplfold, and RNAsubopt
- Add command line parameters to RNAsubopt to allow for specification of input/output files

6.6.3 Version 2.3.3

Release date: *2017-01-24*

Fixed in v2.3.3

- Fix multiloop contributions for comparative partition function
- Fix building python2 extension module for OSX

6.6.4 Version 2.3.2

Release date: *2017-01-18*

Fixed in v2.3.2

- Fix pair probability plist creation with G-Quadruplexes
- Fix init of vrna_md_t data structure after call to set_model_details()
- Fix bug in consensus partition function with hard constraints that force nucleotides to be paired
- Fix compilation of functions that use ellipsis/va_list
- Fix init of partition function DP matrices for unusually short RNAs
- Fix behavior of RNAplfold for unusually short RNAs
- Avoid multiple includes of pair_mat.h

Changed since v2.3.2

- Allow for specification of python2/3-config at configure time
- Enable generic hard constraints by default
- Report SCI of 0 in RNAalifold when sum of single sequence MFEs is 0

Added in v2.3.2

- Add configure flag to build entirely static executables

6.6.5 Version 2.3.1

Release date: *2016-11-15*

Fixed in v2.3.1

- Fix compilation issue for Windows platforms with MingW
- Fix regression in vrna_md_update() that resulted in incomplete init of reverse-basepair type array
- Fix scaling of secondary structure in EPS plot such that it always fits into bounding box
- Several fixes and improvements for SWIG generated scripting language interface(s)

Changed since v2.3.1

- Extend coverage of generic hard constraints for partition function computations

Added in v2.3.1

- Add description for how to use unstructured domains through command files to reference manual and RNAfold manpage
- Add missing newline in non-TTY-color output of vrna_message_info()

6.6.6 Version 2.3.0

Release date: *2016-11-01*

Fixed in v2.3.0

- Fix configure script to deal with situations where Perl module can't be build
- Fix bug in doc/Makefile.am that prevented HTML installation due to long argument list
- Bugfix in SWIG wrapped generic soft constraint feature
- Bugfix multiloop decomposition in MFE for circular RNAs
- Bugfix regression that prevented programs to fail on too long input sequences
- Fix bug in interior loop computations when hard constraints result in non-canonical base pairs

Changed since v2.3.0

- Replace explicit calls to asprintf() with portable equivalent functions in the library
- Renamed VRNA_VC_TYPE_* macros to VRNA_FC_TYPE_*

Added in v2.3.0

- Add grammar extension with structured and unstructured domains
- Add default implementation for unstructured domains to allow for ligand/protein binding to unpaired structure segments (MFE and PF for single sequences)
- Introduced command files that subsume constraint definition files (currently used in RNAfold and RNAcofold)
- Added utility functions that deal with conversion between different units
- Add subopt() and subopt_zuker() methods to SWIG wrapped fold_compound objects
- Add separate function to compute pscore for alignments
- Extend EPS dot-plot in RNAfold to include motif/binding probabilities from unstructured domains
- Add variadic functions for error/warning/info message
- Add ID manipulation feature to RNAeval
- Extend API for soft constraint feature for more fine-grained control
- Add section on SWIG wrapped functions in reference manual

6.7 Version 2.2.x

6.7.1 Version 2.2.10

Release date: *2016-09-06*

Fixed in v2.2.10

- Do not 'forget' subopt results when output is not written to file handle and sorting is switched off
- Fix bad memory access in vrna_subopt() with sorted output
- Fix autoconf checks to allow for cross compilation again

Changed since v2.2.10

- Correctly show if C11 features are activated in configure status

Added in v2.2.10

- Add SWIG wrappers for vrna_subopt_cb()

6.7.2 Version 2.2.9

Release date: *2016-09-01*

Fixed in v2.2.9

- Fix bug in partition function scaling for backward compatibility of ali_pf_fold()
- Fix RNAlib2.pc
- Fix bug for temperature setting in RNAplfold
- Fix interpretation of 'P' hard constraint for single nucleotides in constraint definition files
- Fix several hard constraint corner-cases in MFE and partition function computation when nucleotides must not be unpaired
- Fix order of hard constraints when read from input file
- Fix behavior of –without-swig configure script option
- Fix bug in hard constraints usage of exterior loop MFE prediction with odd dangles

Changed since v2.2.9

- Stabilize v3.0 API when building RNAlib and third party program linking against it with compilers that use different C/C++ standards
- Allow for non-canonical base pairs in MFE and partition function computations if hard constraints demand it
- Lift restriction of sequence number in alignments for RNAalifold

Added in v2.2.9

- Add details on how to link against RNAlib to the reference manual
- Add 'A' command for hard constraints
- Use -fflat-lto-objects for static RNAlib library to allow linking without LTO
- Add parsers for Clustal, Stockholm, FASTA, and MAF formatted alignment files
- Enable RNAalifold to use Clustal, Stockholm, FASTA, or MAF alignments as input
- Enable ANSI colors for TTY output in RNAfold, RNAcofold, RNAalifold, RNAsubopt, and warnings/errors issued by RNAlib
- Add various new commandline options to manipulate sequence/alignment IDs in RNAfold, RNAcofold and RNAalifold

6.7.3 Version 2.2.8

Release date: *2016-08-01*

Fixed in v2.2.8

- Fix bad memory access in RNAalifold
- Fix regression in RNAalifold to restore covariance contribution ratio determination for circular RNA alignments
- Fix syntactic incompatibilities that potentially prevented compilation with compilers other than gcc
- Fix regression in energy evaluation for circular RNA sequences

- Fix regression in suboptimal structure enumeration for circular RNAs
- Fix incompatibility with swig >= 3.0.9
- Fix memory leak in swig-generated scripting language interface(s) for user-provided target language soft-constraint callbacks
- Fix linking of python2/python3 interfaces when libpython is in non-standard directory
- Fix –with-*/–without*- and –enable-*/–disable*- configure script behavior

Changed since v2.2.8

- Changed output of RNAsubopt in energy-band enumeration mode to print MFE and energy range in kcal/mol instead of 10cal/mol
- Include latest Kinfold sources that make use of v3.0 API, therefore speeding up runtime substantially
- Re-activate warnings in RNAeval when non-canonical base pairs are encountered
- Secure functions in alphabet.c against NULL pointer arguments
- Build Python3 interface by default
- Restructured viennarna.spec for RPM based distributions
- Several syntactic changes in the implementation to minimize compiler warnings

Added in v2.2.8

- Add function to compare nucleotides encoded in IUPAC format
- Allow for P i-j k-l commands in constraint definition files
- Make free energy evaluation functions polymorphic
- Add free energy evaluation functions that allow for specifying verbosity level
- Expose additional functions to swig-generated scripting language interface(s)
- Start of more comprehensive scripting language interface documentation

6.7.4 Version 2.2.7

Release date: *2016-06-30*

Fixed in v2.2.7

- Fix partition function scaling for long sequences in RNAfold, RNAalifold, and RNAup
- Fix backtracking issue in RNAcofold when –noLP option is activated
- Fix hard constraints issue for circular RNAs in generating suboptimal structures

Changed since v2.2.7

- Rebuild reference manual only when actually required

6.7.5 Version 2.2.6

Release date: *2016-06-19*

Fixed in v2.2.6

- Plugged memory leak in RNAcofold
- Fixed partition function rescaling bug in RNAup
- Fixed bug in RNALfold with window sizes larger than sequence length

- Re-added SCI parameter for RNAalifold
- Fixed backtracking issue for large G-quadruplexes in RNAalifold
- Fixed missing FASTA id in RNAeval output
- Several fixes and additional functions/methods in scripting language interface(s)

Changed since v2.2.6

- Some changes to allow for compilation with newer compilers, such as gcc 6.1

Added in v2.2.6

- Added option to RNAalifold that allows to specify prefix for output files
- Added version information for scripting language interface(s)

6.7.6 Version 2.2.5

Release date: *2016-04-09*

Fixed in v2.2.5

- Fixed regression in RNAcofold that prohibited output of concentration computations
- Fixed behavior of RNAfold and RNAcofold when hard constraints create empty solution set (programs now abort with error message)
- Plugged some memory leaks in SWIG generated scripting language interfaces

Changed since v2.2.5

- Update RNA::Design Perl 5 sub-package
- Simplified usage of v3.0 API with default options
- Changed parameters of recursion status callback in vrna_fold_compound_t

Added in v2.2.5

- Added optional Python 3 interface
- Added RNA::Params Perl 5 sub-package
- Wrap more functions of v3.0 API in SWIG generated scripting language interfaces
- Enable definition and binding of callback functions from within SWIG target language
- Added optional subpackage Kinwalker
- Added several configure options to ease building and packaging under MacOS X
- Added new utility script RNAdesign.pl

6.7.7 Version 2.2.4

Release date: *2016-02-19*

Fixed in v2.2.4

- Fixed bug in RNAsubopt that occasionally produced cofolded structures twice

Removed in v2.2.4

- Removed debugging output in preparations of consensus structure prediction datastructures

6.7.8 Version 2.2.3

Release date: *2016-02-13*

Fixed in v2.2.3

- Restore backward compatibility of get_alipf_arrays()

Added in v2.2.3

- Added postscipt annotations for found ligand motifs in RNAfold
- Added more documentation for the constraints features in RNAfold and RNAalifold

6.7.9 Version 2.2.2

Release date: *2016-02-08*

Fixed in v2.2.2

- Fix regression bug that occasionally prevented backtracking with RNAcofold –noLP

6.7.10 Version 2.2.1

Release date: *2016-02-06*

Fixed in v2.2.1

- Fix regression bug that made RNAcofold -a unusable
- Fix regression bug that prohibited RNAfold to compute the MEA structure when G-Quadruplex support was switched on
- Fix bug in Kinfold to enable loading energy parameters from file
- Fix potential use of uninitialized value in RNApdist
- Fix MEA computation when G-Quadruplex support is activated

Changed since v2.2.1

- Install architecture dependent and independent files of the perl and python interface to their correct file system locations

Added in v2.2.1

- Add manpage for ct2db
- Allow for vendor installation of the perl interface using INSTALLDIRS=vendor at configure time

6.7.11 Version 2.2.0

Release date: *2016-01-25*

Fixed in v2.2.0

- Bugfix in RNAheat to remove numerical instabilities
- Bugfix in RNAplex to allow for computation of interactions without length limitation
- Bugfix in RNAplot for simple layouts and hairpins of size 0

Changed since v2.2.0

- RNAforester is now of version 2.0
- libsvm version as required for z-scoring in RNALfold is now 3.20
- Stochastic backtracking for single sequences is faster due to usage of Boustrophedon scheme
- The FLT_OR_DBL macro is now a typedef
- Libraries are generated using libtool
- Linking of libraries and executables defaults to use Link Time Optimization (LTO)
- Large changes in directory structure of the source code files

Added in v2.2.0

- New program RNApvmin to compute pseudo-energy pertubation vector that minimizes discrepancy between observed and predicted pairing probabilities
- SHAPE reactivity support for RNAfold, RNAsubopt, and RNAalifold
- Ligand binding to hairpin- and interior-loop motif support in RNAfold
- New commandline option to limit maximum base pair span for RNAfold, RNAsubopt, RNAcofold, and RNAalifold
- (generic) hard- and soft-constraints for MFE, partition function, base pair probabilities, stochastic backtracking, and suboptimal secondary structures of single sequences, sequence alignments, and sequence dimers
- First polymorphic functions vrna_mfe(), vrna_pf(), and vrna_pbacktrack().
- New functions to convert between different secondary structure representations, such as helix lists, and RNAshapes abstractions
- First object-oriented interface for new API functions in the scripting language interfaces
- new ViennaRNA-perl submodule that augments the Perl interface to RNAlib
- Ligand binding to hairpin- and interior-loop motif support in C-library and scripting language interfaces.

6.8 Version 2.1.x

6.8.1 Version 2.1.9

- Fixed integer underflow bug in RNALfold
- Added Sequence Conservation index (SCI) option to RNAalifold
- Fixed bug in energy evaluation of dangling ends / terminal mismatches of exterior loops and multibranch loops
- Fixed bug in alifold partition function for circular RNAs
- Fixed bug in alifold that scrambled backtracing with activated G-Quadruplex support
- Fixed bug in alifold backtracking for larger G-Quadruplexes

6.8.2 Version 2.1.8

- Repaired incorporation of RNAinverse user provided alphabet
- Fix missing FASTA ID in RNAeval output
- prevent race condition in parallel calls of Lfold()
- Fixed memory bug in Lfold() that occured using long sequences and activated G-Quad support
- Added latest version of switch.pl

6.8.3 Version 2.1.7

- Fixed bug in RNALfold -z
- Python and Perl interface are compiling again under MacOSX
- Fixed handling of C arrays in Python interface
- Added latest version of switch.pl
- Make relplot.pl work with RNAcofold output

6.8.4 Version 2.1.6

- New commandline switches allow for elimination of non-canonical base pairs from constraint structures in RNAfold, RNAalifold and RNAsubopt
- updated moveset functions
- final fix for discrepancy of tri-loop evaluation between partition function and mfe
- pkg-config file now includes the OpenMP linker flag if necessary
- New program ct2db allows for conversion of .ct files into dot-bracket notation (incl. pseudo-knot removal)

6.8.5 Version 2.1.5

- Fix for discrepancy between special hairpin loop evaluation in partition functions and MFE

6.8.6 Version 2.1.4

- Fix of G-quadruplex support in subopt()
- Fix for discrepancy between special hairpin loop evaluation in partition functions and MFE

6.8.7 Version 2.1.3

- RNAfold: Bugfix for ignoring user specified energy parameter files
- RNAcofold: Bugfix for crashing upon constrained folding without specifying a constraint structure
- RNAsubopt: Added G-quadruplex support
- RNAalifold: Added parameter option to specify base pair probability threshold in dotplot
- Fix of several G-quadruplex related bugs
- Added G-quadruplex support in subopt()

6.8.8 Version 2.1.2

- RNAfold: Bugfix for randomly missing probabilities in dot-plot during batch job execution
- RNAeval: Bugfix for misinterpreted G-quadruplex containing sequences where the quadruplex starts at nucleotide 1
- RNAsubopt: Slight changes to the output of stochastic backtracking and zuker subopt
- Fix of some memory leaks
- Bugfixes in zukersubopt(), assign_plist_from_pr()
- New threadsafe variants of putoutpU_prob*() for LPfold()
- Provision of python2 interface support.

6.8.9 Version 2.1.1

- Bugfix to restore backward compatibility with ViennaRNA Package 1.8.x API (this bug also affected proper usage of the the perl interface)

6.8.10 Version 2.1.0

- G-Quadruplex support in RNAfold, RNAcofold, RNALfold, RNAalifold, RNAeval and RNAplot
- LPfold got a new option to output its computations in split-mode
- several G-Quadruplex related functions were introduced with this release
- several functions for moves in an RNA landscape were introduced
- new function in alipfold.c now enables access to the partition function matrices of alipf_fold()
- different numeric approach was implement for concentration dependend co-folding to avoid instabilities which occured under certain circumstances

6.9 Version 2.0.x

6.9.1 Version 2.0.7

- Bugfix for RNAplfold where segfault happened upon usage of -O option
- Corrected misbehavior of RNAeval and RNAplot in tty mode

6.9.2 Version 2.0.6

- Bugfix for bad type casting with gcc under MacOSX (resulted in accidental "sequence too long" errors)
- Bugfix for disappearing tri-/hexaloop contributions when read in from certain parameter files
- Bugfix for RNALfold that segfaulted on short strange sequences like AT+ repeats
- Change of RNA2Dfold output format for stochastic backtracking

6.9.3 Version 2.0.5

- Restored z-score computation capabilities in RNALfold

6.9.4 Version 2.0.4

- Bugfix for RNAcofold partition function
- Perl wrapper compatibility to changed RNAlib has been restored
- Backward compatibility for partition function calls has been restored

6.9.5 Version 2.0.3

- Bugfix for RNAalifold partition function and base pair probabilities in v2.0.3b
- Added Boltzmann factor scaling in RNAsubopt, RNAalifold, RNAplfold and RNAcofold
- Bugfix for alipfold() in v2.0.3b
- Restored threadsafety of folding matrix access in LPfold.c, alipfold.c, part_func.c, part_func_co.c and part_func_up.c
- Added several new functions regarding threadsafe function calls in terms of concurrently changing the model details
- Added pkg-config file in the distribution to allow easy checks for certain RNAlib2 versions, compiler flags and linker flags.

6.9.6 Version 2.0.2

- added support for Boltzmann factor scaling in RNAfold
- fixed fastaheader to filename bug
- plugged some memory leaks

6.9.7 Version 2.0.1

- First official release of version 2.0
- included latest bugfixes

6.10 History

2011-03-10 Ronny Lorenz ronny@tbi.univie.ac.at

- new naming scheme for all shipped energy parameter files
- fixed bugs that appear while compiling with gcc under MacOS X
- fixed bug in RNAup –interaction-first where the longer of the first two sequences was taken as target
- added full FASTA input support to RNAfold, RNAcofold, RNAheat, RNAplfold RNALfoldz, RNAsubopt and RNALfold

2010-11-24 Ronny Lorenz ronny@tbi.univie.ac.at

- first full pre-release of version 2.0

2009-11-03 Ivo Hofacker ivo@tbi.univie.ac.at

- Fix memory corruption in PS_color_aln()

2009-09-09 Ivo Hofacker ivo@tbi.univie.ac.at

- Fix bug in RNAplfold when -u and -L parameters are equal
- Fix double call to free_arrays() in RNAfold.c
- Improve drawing of cofolded structures

2009-05-14 Ivo Hofacker ivo@tbi.univie.ac.at

- Fix occasional segfault in RNAalifold's print_aliout()

2009-02-24 Ivo Hofacker ivo@tbi.univie.ac.at

- Add -MEA options to RNAfold and RNAalifold
- change energy_of_alistruct to return float not void

2009-02-24 Ivo Hofacker ivo@tbi.univie.ac.at

- RNAfold will draw structures unless -noPS is used (no more "structure too long" messages)
- Restore the "alifold.out" output from RNAalifold -p
- RNAalifold -circ did not work due to wrong return type
- Accessibility calculation with RNAplfold would give wrong results for u<=30

2008-12-03 Ivo Hofacker ivo@tbi.univie.ac.at

- Add zuker style suboptimals to RNAsubopt (-z)
- get_line() should be much faster when reading huge sequences (e.g. whole chromosomes for RNALfold)

2008-08-12 Ivo Hofacker ivo@tbi.univie.ac.at

- Add Ribosum matrices for covariance scoring in RNAalifold

2008-06-27 Ivo Hofacker ivo@tbi.univie.ac.at

- Change RNAalifold to used berni's new energy evaluation w/o gaps
- Add stochastic backtracking in RNAalifold

2008-07-04 Ivo Hofacker ivo@tbi.univie.ac.at

- modify output of RNAup (again). Program reading RNAup output will have to updated!

2008-07-02 Ivo Hofacker ivo@tbi.univie.ac.at

- RNAplfold now computes accessibilities for all regions up to a max length simultaneously. Slightly slower when only 1 value is needed, but much faster if all of them are wanted. This entails a new output format. Programs reading accessibility output from RNAplfold need to be updated!

2008-03-31 Stephan Bernhart berni@tbi.univie.ac.at

- add cofolding to RNAsubopt

2008-01-08 Ivo Hofacker ivo@tbi.univie.ac.at

- ensure circfold works even for open chain

2007-12-13 Ulli Mueckstein ulli@tbi.univie.ac.at

- upate RNAup related files RNAup can now include the intramolecular structure of both molecules and handles constraints.

2007-12-05 Ronny Lorenz ronny@tbi.univie.ac.at

- add circfold variants in part_func.c alipfold.c subopt.c

2007-09-19 Ivo Hofacker ivo@tbi.univie.ac.at

- compute the controid structure of the ensemble in RNAfold -p
- fix a missing factor 2 in mean_bp_dist(). CAUTION ensemble diversities returned by RNAfold -p are now twice as large as in earlier versions.

2007-09-04 Ivo Hofacker ivo@blini.tbi.univie.ac.at

- fix a bug in Lfold() where base number n-max-4 would never pair

2007-08-26 Ivo Hofacker ivo@tbi.univie.ac.at

- add RNAaliduplex the alignment version of RNAduplex
- introduce a minimal distance between hits produced by duplex_subopt()

2007-07-03 Ivo Hofacker ivo@tbi.univie.ac.at

- add a loop_energy() function to compute energy of a single loop

2007-06-23 Ivo Hofacker ivo@tbi.univie.ac.at

- add aliLfold() and RNALalifold, alignment variant of Lfold()

2007-04-30 Ivo Hofacker ivo@tbi.univie.ac.at

- add RNAup to distribution

2007-04-15 Ivo Hofacker ivo@tbi.univie.ac.at

- fix segfault in colorps output (thanks to Andres Varon)

2007-03-03 Ivo Hofacker ivo@tbi.univie.ac.at

- avoid unnormalized doubles in scale[], big speedup for pf_fold() on very long sequences

2007-02-03 Ivo Hofacker ivo@tbi.univie.ac.at

- RNAalifold can now produce colored structure plots and alignment plots

2007-02-01 Ivo Hofacker ivo@tbi.univie.ac.at

- Fix segfault in RNAplfold because of missing prototype

2006-12-01 Ivo Hofacker ivo@tbi.univie.ac.at

- RNAduplex would segfault when no structure base pairs are possible

2006-08-22 Ivo Hofacker ivo@tbi.univie.ac.at

- add computation stacking probabilities using RNAfold -p2
- add -noPS option for NRAfold to supress drawing structures

2006-08-09 Stephan Bernhart berni@tbi.univie.ac.at

- RNAplfold can now compute probabilites of unpaired regions (scanning version of RNAup)

2006-06-14 Ivo Hofacker ivo@tbi.univie.ac.at

- compile library with -fpic (if available) for use as shared library in the Perl module.
- fix another bug when calling Lfold() repeatedly
- fix switch cmdline parsing in RNAalifold (-mis implied -4)
- fix bug in cofold() with dangles=0

2006-05-08 Ivo Hofacker ivo@tbi.univie.ac.at

- fix segfault in Lfold() when calling repeatedly
- fix structure parsing in RNAstruct.c (thanks to Michael Pheasant for reporting both bugs)
- add duplexfold() and alifold() to Perl module
- distinguish window size and max pair span in LPfold

2006-04-05 Ivo Hofacker ivo@tbi.univie.ac.at

- fix performance bug in co_pf_fold()
- use relative error for termination of Newton iteration

2006-03-02 Ivo Hofacker ivo@tbi.univie.ac.at

- add circular folding in alifold()

2006-01-18 Ivo Hofacker ivo@tbi.univie.ac.at

- cleanup berni partition cofold code, including several bug fixes

2006-01-16 Ivo Hofacker ivo@tbi.univie.ac.at

- update RNAplfold to working version
- add PS_dot_plot_turn() in PS_dot.c

2005-11-07 Ivo Hofacker ivo@tbi.univie.ac.at

- add new utilities colorna and coloraln

2005-10-11 Christoph Flamm xtof@tbi.univie.ac.at

- adapt PS_rna_plot() for drawing co-folded structures

2005-07-24 Ivo Hofacker ivo@tbi.univie.ac.at

- fix a few memory problems in structure comparison routines

2005-04-30 Ivo Hofacker ivo@blini.tbi.univie.ac.at

- add folding of circular RNAs

2005-03-11 Ivo Hofacker ivo@blini.tbi.univie.ac.at

- add -mis option to RNAalifold to give “most informative sequence” as consensus

2005-02-10 Ivo Hofacker ivo@tbi.univie.ac.at

- move alifold() into the library

2004-12-22 Stephan Bernhart berni@tbi.univie.ac.at

- add partition function version of RNAcofold

2004-12-23 Ivo Hofacker ivo@tbi.univie.ac.at

- add RNApaln for fast structural alignments (RNApdist improvement)

2004-08-12 Ivo Hofacker ivo@tbi.univie.ac.at

- fix constrained folding in stochastic backtracking

2004-07-21 Ivo Hofacker ivo@tbi.univie.ac.at

- add RNAduplex, to compute hybrid structures without intra-molecular pairs

2004-02-09 Ivo Hofacker ivo@tbi.univie.ac.at

- fix bug in fold that caused segfaults when using Intel compiler
- add computation of ensemble diversity to RNAfold

2003-09-10 Ivo Hofacker ivo@tbi.univie.ac.at

- add annotation options to RNAplot

2003-08-04 Ivo Hofacker ivo@tbi.univie.ac.at

- stochastic backtracking finally works. Try e.g. RNAsubopt -p 10

2003-07-18 Ivo Hofacker ivo@tbi.univie.ac.at

- add relplot.pl and rotate_ss.pl utilities for reliability annotation and rotation of rna structure plots

2003-01-29 Ivo Hofacker ivo@tbi.univie.ac.at

- add RNALfold program to compute locally optimal structures with maximum pair span.
- add RNAcofold for computing hybrid structure

2002-11-07 Ivo Hofacker ivo@tbi.univie.ac.at

- change Make_bp_profile() and profile_edit_distance() to use simple (float *) arrays; makes Perl access much easier. RNApdist -B now works again

2002-10-28 Ivo Hofacker ivo@tbi.univie.ac.at

- Improved Perl module with pod documentation; allow to write things like ($structure, $energy) = RNA::fold($seq); Compatibility warning: the ptrvalue() and related functions are gone, see the pod documentation for alternatives.

2002-10-29 Ivo Hofacker ivo@tbi.univie.ac.at

- added svg structure plots in PS_dot.c and RNAplot

2002-08-15 Ivo Hofacker ivo@tbi.univie.ac.at

- Improve reading of clustal files (alifold)
- add a sample alifold.cgi script

2001-09-18 Ivo Hofacker ivo@tbi.univie.ac.at

- moved suboptimal folding into the library, thus it's now accessible from the Perl module

2001-08-31 Ivo Hofacker ivo@tbi.univie.ac.at

- added co-folding support in energy_of_struct(), and thus RNAeval

2001-04-30 Ivo Hofacker ivo@tbi.univie.ac.at

- switch from handcrafted makefiles to automake and autoconf

2001-04-05 Ivo Hofacker ivo@tbi.univie.ac.at

- added PS_rna_plot_a to produce structure plots with annotation

2001-03-03 Ivo Hofacker ivo@tbi.univie.ac.at

- add alifold; predict consensus structures from alignment

2000-09-28 Ivo Hofacker ivo@tbi.univie.ac.at

- add -d3 option to RNAfold for co-axial stacking

CHAPTER

SEVEN

BIBLIOGRAPHY

CHAPTER

EIGHT

HOW TO CITE THE VIENNARNA PACKAGE

If you use our software and implemented algorithms for your scientific work, you might want to cite the corresponding publications.

8.1 Main References

The *overarching* main publications for the ViennaRNA Package are:

- Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(1):26, 2011. doi:10.1186/1748-7188-6-26.
- I.L. Hofacker, W. Fontana, P.F. Stadler, L.S. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatshefte für Chemie/Chemical Monthly*, 125(2):167–188, 1994. URL: https://www.academia.edu/download/48689421/Fast_Folding_and_Comparison_of_RNA_Secon20160908-13624-1yg70az.pdf.

8.2 Particular Algorithms and Features

To reference particular algorithms used in our sofware, you might want to consider citing the following publications:

8.2.1 Consensus structure prediction

- I.L. Hofacker, M. Fekete, and P.F. Stadler. Secondary structure prediction for aligned RNA sequences. *Journal of molecular biology*, 319(5):1059–1066, 2002. doi:10.1016/S0022-2836(02)00308-X.
- S.H. Bernhart, I.L. Hofacker, S. Will, A.R. Gruber, and P.F. Stadler. RNAalifold: improved consensus structure prediction for RNA alignments. *BMC bioinformatics*, 9(1):474, 2008. doi:10.1186/1471-2105-9-474.

8.2.2 Local pair probabilitiy and accessibility

- Stephan H Bernhart, Ivo L Hofacker, and Peter F Stadler. Local RNA base pairing probabilities in large sequences. *Bioinformatics*, 22(5):614–615, 2005. doi:10.1093/bioinformatics/btk014.
- Stephan H Bernhart, Ullrike Mückstein, and Ivo L Hofacker. RNA accessibility in cubic time. *Algorithms for Molecular Biology*, 6(1):3, 2011. doi:10.1186/1748-7188-6-3.

8.2.3 Suboptimal secondary structures

- S. Wuchty, W. Fontana, I. L. Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49(2):145–165, February 1999. doi:10.1002/(SICI)1097-0282(199902)49:2<145::AID-BIP4>3.0.CO;2-G.

8.2.4 k,l - Neighborhood

- Ronny Lorenz, Christoph Flamm, and Ivo L. Hofacker. 2d projections of RNA folding landscapes. In Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, and Peter F. Stadler, editors, *German Conference on Bioinformatics 2009*, volume 157 of Lecture Notes in Informatics, 11–20. Bonn, September 2009. Gesellschaft f. Informatik. URL: https://dl.gi.de/items/8f88acfe-c389-4dfe-b975-84a638900683.

8.2.5 Experimental RNA structure probing

- Ronny Lorenz, Dominik Luntzer, Ivo L. Hofacker, Peter F. Stadler, and Michael T. Wolfinger. Shape directed rna folding. *Bioinformatics*, 32(1):145–147, 2016. doi:10.1093/bioinformatics/btv523.

8.2.6 Structure Constraints

- Ronny Lorenz, Ivo L. Hofacker, and Peter F. Stadler. RNA folding with hard and soft constraints. *Algorithms for Molecular Biology*, 11(1):1–13, 2016. doi:10.1186/s13015-016-0070-z.

8.2.7 Non-redundant Boltzmann sampling

- Juraj Michálik, Hélène Touzet, and Yann Ponty. Efficient approximations of RNA kinetics landscape using non-redundant sampling. *Bioinformatics*, 33(14):i283–i292, 2017. doi:10.1093/bioinformatics/btx269.

CHAPTER

NINE

FREQUENTLY ASKED QUESTIONS

9.1 Missing EXTERN.h

When compiling from source at the Mac OS X platform, users often encounter an error message stating that the file `EXTERN.h` is missing for compilation of the `Perl 5` wrapper. This is a known problem and due to the fact that users are discouraged to use the `Perl 5` interpreter that is shipped with Mac OS X.

Instead, one should install a more recent version from another source, e.g. `homebrew`. If, however, for any reason you do not want to install your own `Perl 5` interpreter but use the one from Apple, you need to specify its include path to enable building the ViennaRNA Perl interface. Otherwise, the file `EXTERN.h` will be missing at compile time. To fix this problem, you first need to find out where `EXTERN.h` is located:

```
sudo find /Library -type f -name EXTERN.h
```

Then choose the one that corresponds to your default perl interpreter (find out the version number with `perl -v | grep version`), simply execute the following before running the `./configure` script, e.g.:

```
export CPATH=/Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/System/Library/
↪Perl/5.18/darwin-thread-multi-2level/CORE
```

if your default perl is v5.18 running on MacOSX10.15. Change the paths according to your current setup. After that, running `./configure` and compilation should run fine.

See also…

Related question at stackoverflow: https://stackoverflow.com/q/52682304/18609162

9.2 Linking fails with LTO error

By default, *RNAlib* is compiled with configuration:link time optimization. This may introduce problems upon linking a third-party program that was either compiled with a different compiler or compiler version. As a work-around solution, we include the `-fno-lto` linker flag in the output of the pkg-config. This tells the linker to not perform link time optimization even though LTO code is included in the library. Usually, this should not affect the runtime of the algorithms too much.

However, some linkers may not support the `-fno-lto` flag and fail at the linker stage. In addition, if *RNAlib* has been compiled with `clang`, it may not include the non-LTO code required for linking without LTO. To resolve this issue, you may need to deactivate link time optimization while building *RNAlib*.

See also…

configuration:link time optimization

CHAPTER

TEN

CONTRIBUTING TO VIENNARNA

10.1 Reporting Bugs

The ViennaRNA Package is developed by humans and consequently may contain bugs that prevent proper operation of the implemented algorithms. If you believe you have found any of those nasty animals, please help us to improve our software by sending comments and bug reports either as issue at GitHub or by email to rna@tbi.univie.ac.at.

> If in doubt our program is right, nature is at fault.

Adhere to the following rules while reporting any issues:

- In your report, **include as much information as possible, such that we are able to reproduce it**. If possible, find a minimal example that triggers the bug.
- **Include the version number** for the ViennaRNA Package you experience the bug with.
- Include at least some minimal information regarding your operating system (Linux, Mac OS X, Windows, etc.)

10.2 Pull Request Process

The ViennaRNA Package is open-source software, which means that everybody can have a closer look into our implementations to understand and potentially extend it's functionality. If you implemented any novel feature into the ViennaRNA Package that might be of interest to a larger community, please don't hesitate to ask for merging of your code into our official source tree.

Note

If you wish to contribute to this project, please first discuss any proposed changes with the owners and main developers. You may do that either through making an issue at our official GitHub presence, by email, or any other personal communication with the core developer team.

Important

More importantly, if you wish to contribute any files or software, you need to agree to our ViennaRNA Package Contributors License Agreement (CLA)! Otherwise, your contributions can't be merged into our source tree. *See below* for further information and the full CLA details.

Adhere to the following rules, otherwise your pull request can't be considered:

- Ensure that you **do not check-in any files that are automatically build**!
- **Use descriptive commit messages**!
- Make **atomic commits**, i.e. one feature at a time! Do not commit too many changes at once but split them into individual commits.

- **Do not commit cosmetic changes!** Commits that merely beautify the code or fix spelling in the documentation will not be considered for a merge!
- When contributing C source code, **follow our code formatting style**. Have a look into our source code to get an idea how we format everything. Also, you may use the tool `uncrustify` together with our config located in `misc/uncrustify.cfg` to enforce our code formatting to your implementation.
- **Only expose symbols** (functions, variables, etc.) to the libraries interface **that are absolutely necessary**! Hide all other symbols in the corresponding object file(s) by declaring them as `static`.
- **Use the prefix `vrna_`** for any symbol you add to the API of our library! Preprocessor macros in header files require the prefix in capital letters, i.e. `VRNA_`.
- **Use C-style comments** at any place necessary to make sure your implementation can still be understood and followed in the future.
- **Add test cases** for any new implementation! The test suite is located in the `tests` directory and is split into tests for the C-library, executable programs, and the individual scripting language interfaces.
- Run `make check` to **ensure that all other test suites still run properly** with your applied changes!
- When contributing via GitHub, make a personal fork of our project and create a separate branch for your changes. Then make a **pull request to our `user-contrib` branch**. Pull requests to the `master` branch will be rejected to keep its history clean.
- Pull requests that have been successfully merged into the `user-contrib` branch usually find their way into the next release of the ViennaRNA Package. However, please note that the core developers may decide to include your changes in a later version.

10.3 Contributors License Agreement

Thank you for your interest in contributing to the ViennaRNA Package ("We" or "Us").

Before contributing, please note that we adopted a standard Contributors License Agreement (CLA) agreement provided by Project Harmony, a community-centered group focused on contributor agreements for free and open source software (FOSS).

This contributor agreement ("Agreement") documents the rights granted by contributors to Us. To make this document effective, please sign it and send it to Us by email to rna@tbi.univie.ac.at.

The respective CLA PDF documents are available in the `doc/CLA` directory of the distribution tarball, and online at our official ViennaRNA Website:

ViennaRNA Package - Individual Contributor License Agreement

ViennaRNA Package - Entity Contributor License Agreement

CHAPTER

ELEVEN

ABOUT THE VIENNARNA PACKAGE

The core of the **ViennaRNA Package** (Lorenz *et al.* [2011], Hofacker *et al.* [1994]) is formed by a collection of algorithms for the **prediction and comparison** of RNA secondary structures.

Physics based RNA secondary structure prediction is the main functionality in the package. We provide variants of three different kinds of dynamic programming algorithms for structure prediction, (i) the **minimum free energy** algorithm of Zuker and Stiegler [1981] which yields a single optimal structure, (ii) the **partition function** algorithm of McCaskill [1990] that allows for **base pair probability** computation or stochastic backtracking of structure samples, and (iii) the **suboptimal folding** algorithm of Wuchty *et al.* [1999] which generates all suboptimal structures within a given energy range of the optimal energy. For secondary structure comparison, the package contains several measures of distance (dissimilarities) using either **string alignment** or **tree-editing** Shapiro and Zhang [1990]. Finally, we provide an algorithm to design sequences with a predefined structure, i.e. **inverse folding**.

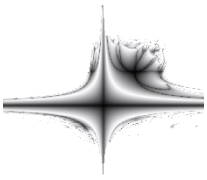
11.1 History

Over the past decades since the ViennaRNA Package first saw the light of day as part of Ivo L. Hofackers PhD project, several authors contributed more and more algorithm implementations. In 2008, Ronny Lorenz took over the extensive task to harmonize and simplify the already existing implementations for the sake of easier feature addition. This eventually lead to version 2.0 of the ViennaRNA Package. Since then, he (re-)implemented a large portion of the currently existing library features, such as the new, generalized constraints framework, RNA folding grammar domain extensions, and the major part of the scripting language interface.

11.2 Contributors

This is an open source project that has only been made possible with the help of many researchers and users that contributed code and documentation.

Here we want to thank everyone who spend time and effort to help thrive of the ViennaRNA Package through discussions, code contributions, testing or bug finding, in particular:



- Ronny Lorenz
  - **Research**
  - **Maintenance**
  - **Code:** ViennaRNA 2.0, (most of) circular RNA support, G-quadruplex support, constraints framework, structure probing data support *(commits: 4792)*
  - **Ideas, Planning, & Feedback**

  - **Bug reports**
  - **Tests**
  - **Reviewed Pull Requests**
  - **Answering Questions**
  - **Packaging/porting to new platform**
  - **Mentoring**

- 

  Ivo L. Hofacker

  - **Research**
  - **Code:** Most of the implementations up to version 1.8.5 *(commits: 694)*
  - **Ideas, Planning, & Feedback**
  - **Answering Questions**
  - **Mentoring**
  - **Infrastructure (Hosting, Build-Tools, etc)**

- 

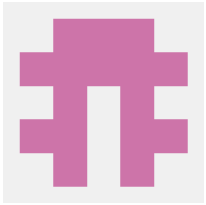
  Christoph Flamm

  - **Research**
  - **Code:** findpath, switch.pl *(commits: 4)*
  - **Ideas, Planning, & Feedback**
  - **Mentoring**

- 

  Peter F. Stadler

  - **Research**
  - **Ideas, Planning, & Feedback**
  - **Answering Questions**
  - **Mentoring**
  - **Infrastructure (Hosting, Build-Tools, etc)**

- 

  Hua-Ting Yao

  - **Research**
  - **Code:** Salt concentration dependency, structure probing data support *(commits: 40)*
  - **Ideas, Planning, & Feedback**
  - **Bug reports**
  - **Tests**
  - **Answering Questions**

- 

  Stefan Badelt

  - **Research**
  - **Code:** cofold evaluation, RNAdesign.pl, cofold findpath extensions *(commits: 17)*
  - **Bug reports**
  - **Ideas, Planning, & Feedback**

- 

  Stephan Bernhart

  - **Research**
  - **Code:** RNAcofold, RNAplfold, RNAalifold, unpaired probabilities, and so much more *(commits: 14)*
  - **Ideas, Planning, & Feedback**
  - **Mentoring**

- 

  Wolfgang Beyer

  - **Research**
  - **Code:** first version of RNApkplex *(commits: 2)*

- 

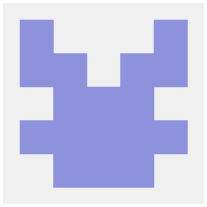
  Sebastian Bonhöffer

  - **Research**
  - **Code:** implementation of partition function folding served as a precursor to our part_func.c

- 

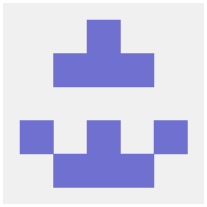
  Theo Boury

  - **Research**
  - **Code:** LinearBP inverse folding seed *(commits: 3)*
  - **Ideas, Planning, & Feedback**

- 

  Giovanni Bussi

  - **Code:** *(commits: 2)*
  - **Ideas, Planning, & Feedback**

- 

  Gregor Entzian

  - **Research**
  - **Code:** RNA landscape neighborhood and moves *(commits: 30)*

- 

  Martin Fekete

  - **Research**

  - **Code:** first attempts at alignment folding

- 

  Andreas Gruber

  - **Research**
  - **Code:** first approach on RNALfold Z-score filtering

- 

  Stefan Hammer

  - **Research**
  - **Code:** Parts of SWIG interface and corresponding unit tests *(commits: 1)*
  - **Bug reports**
  - **Tests**
  - **Ideas, Planning, & Feedback**

- 

  Martijn Huynen

  - **Bug reports**

- 

  Mario Köstl

  - **Tests**
  - **Code:** worked on SWIG interface and related unit testing *(commits: 25)*

- 

Bärbel Krakhofer

- **Bug reports**

- 

William B. Langdon

- **Research**
- **Code:** Initial SSE4.2 support *(commits: 1)*
- **Ideas, Planning, & Feedback**
- **Data:** GGGP derived RNA parameters

- 

Dominik Luntzer

- **Research**
- **Code:** structure probing data support, perturbation fold *(commits: 131)*

- 

Juraj Michalik

- **Research**
- **Code:** non-redundant Boltzmann sampling *(commits: 7)*

- 

Ulrike Mückstein

- **Research**
- **Code:** RNAup *(commits: 2)*

- 

  Yann Ponty
  - **Research**
  - **Code**
  - **Ideas, Planning, & Feedback**
  - **Mentoring**

- 

  Martin Raden
  - **Bug reports**
  - **Code:** *(commits: 10)*

- 

  Thomas Spicher
  - **Research**
  - **Data:** Dihydrouridine parameters
  - **Bug reports**
  - **Ideas, Planning, & Feedback**
  - **Code:** *(commits: 3)*

- 

  Manfred Tacker
  - **Research**
  - **Code:** hacked constrained folding into MFE folding for the first time

- 

  Hakim Tafer

  - **Research**
  - **Code:** RNAplex, RNAsnoop *(commits: 1)*

- 

  Andrea Tanzer

  - **Bug reports**
  - **Code:** *(commits: 1)*

- 

  Daniel Wiegreffe

  - **Research**
  - **Code:** RNAturtle and RNApuzzler secondary structure layouts *(commits: 13)*

- 

  Sebastian Will

  - **Research**
  - **Code:** *(commits: 2)*
  - **Bug reports**
  - **Ideas, Planning, & Feedback**

- 

  Michael T. Wolfinger

  - **Research**
  - **Code:** *(commits: 3)*
  - **Bug reports**
  - **Promotion**

- 

  Stefan Wuchty
  - **Research**
  - **Code:** Suboptimal structures

- 

  fncnt
  - **Code:** *(commits: 14)*

- 

  walter
  - **Code:** *(commits: 6)*

- 

  sl0wl1
  - **Code:** *(commits: 5)*

- 

  Domen Vaupotič
  - **Code:** *(commits: 4)*

- 

  Felix Kuehnl

  - **Code:** *(commits: 4)*

- 

  Marco Matthies

  - **Code:** *(commits: 3)*

- 

  Ansa89

  - **Code:** *(commits: 2)*

- 

  Maxim Koltsov

  - **Code:** *(commits: 2)*

- 

  Yahya Al-Shamali

  - **Code:** *(commits: 2)*

- 

  Anmol Seth

  - **Code:** *(commits: 1)*

- 

  Bobby R. Bruce

  - **Code:** *(commits: 1)*

- 

  Dominik Rabsch

  - **Code:** *(commits: 1)*

- 

  Michael Souza

  - **Code:** *(commits: 1)*

- 

  Nick Booher

  - **Code:** *(commits: 1)*

- 

  joh16

  - **Code:** *(commits: 1)*

CHAPTER

TWELVE

LICENSE

```
Disclaimer and Copyright
```

The programs, library and source code of the Vienna RNA Package are free software. They are distributed in the hope that they will be useful but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Permission is granted for research, educational, and commercial use and modification so long as 1) the package and any derived works are not redistributed for any fee, other than media costs, 2) proper credit is given to the authors and the Institute for Theoretical Chemistry of the University of Vienna.

If you want to include this software in a commercial product, please contact the authors.

BIBLIOGRAPHY

[1] Robert Giegerich, Björn Voß, and Marc Rehmsmeier. Abstract shapes of RNA. *Nucleic Acids Research*, 32(16):4843–4851, 2004. doi:10.1093/nar/gkh779.

[2] W. Fontana, P.F. Stadler, E.G. Bornberg-Bauer, T. Griesmacher, I.L. Hofacker, M. Tacker, P. Tarazona, E.D. Weinberger, and P. Schuster. RNA folding and combinatory landscapes. *Physical review E*, 47(3):2083, 1993. doi:10.1103/PhysRevE.47.2083.

[3] B.A. Shapiro. An algorithm for comparing multiple RNA secondary structures. *Computer applications in the biosciences: CABIOS*, 4(3):387–393, 1988. doi:10.1093/bioinformatics/4.3.387.

[4] I.L. Hofacker, M. Fekete, and P.F. Stadler. Secondary structure prediction for aligned RNA sequences. *Journal of molecular biology*, 319(5):1059–1066, 2002. doi:10.1016/S0022-2836(02)00308-X.

[5] S.H. Bernhart, I.L. Hofacker, S. Will, A.R. Gruber, and P.F. Stadler. RNAalifold: improved consensus structure prediction for RNA alignments. *BMC bioinformatics*, 9(1):474, 2008. doi:10.1186/1471-2105-9-474.

[6] Christine E Hajdin, Stanislav Bellaousov, Wayne Huggins, Christopher W Leonard, David H Mathews, and Kevin M Weeks. Accurate SHAPE-directed RNA secondary structure modeling, including pseudoknots. *Proceedings of the National Academy of Sciences*, 110(14):5498–5503, 2013. doi:10.1073/pnas.1219988110.

[7] Robert D Jenison, Stanley C Gill, Arthur Pardi, and Barry Polisky. High-resolution molecular discrimination by RNA. *Science*, 263(5152):1425–1429, 1994. doi:10.1126/science.7510417.

[8] Amy YQ Zhang, Anthony Bugaut, and Shankar Balasubramanian. A sequence-independent analysis of the loop length dependence of intramolecular RNA G-quadruplex stability and topology. *Biochemistry*, 50(33):7251–7258, 2011. doi:10.1021/bi200805j.

[9] Robert A Forties and Ralf Bundschuh. Modeling the interplay of single-stranded binding proteins and nucleic acid secondary structure. *Bioinformatics*, 26(1):61–67, 2010. doi:10.1093/bioinformatics/btp627.

[10] Stefan Washietl, Ivo L. Hofacker, Peter F. Stadler, and Manolis Kellis. RNA folding with soft constraints: reconciliation of probing data and thermodynamics secondary structure prediction. *Nucleic Acids Research*, 40(10):4261–4272, 2012. doi:10.1093/nar/gks009.

[11] S. Wuchty, W. Fontana, I. L. Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49(2):145–165, February 1999. doi:10.1002/(SICI)1097-0282(199902)49:2<145::AID-BIP4>3.0.CO;2-G.

[12] Ye Ding and Charles E. Lawrence. A statistical sampling algorithm for RNA secondary structure prediction. *Nucleic Acids Research*, 31(24):7280–7301, 12 2003. doi:10.1093/nar/gkg938.

[13] Christoph Flamm, Ivo L Hofacker, Sebastian Maurer-Stroh, Peter F Stadler, and Martin Zehl. Design of multistable RNA molecules. *RNA*, 7(02):254–265, 2001. doi:10.1017/s1355838201000863.

[14] Ronny Lorenz, Christoph Flamm, and Ivo L. Hofacker. 2d projections of RNA folding landscapes. In Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, and Peter F. Stadler, editors, *German Conference on Bioinformatics 2009*, volume 157 of Lecture Notes in Informatics, 11–20. Bonn, September 2009. Gesellschaft f. Informatik. URL: https://dl.gi.de/items/8f88acfe-c389-4dfe-b975-84a638900683.

[15] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic acids research*, 9(1):133–148, 1981. doi:10.1093/nar/9.1.133.

[16] I.L. Hofacker and P.F. Stadler. Memory efficient folding algorithms for circular RNA secondary structures. *Bioinformatics*, 22(10):1172–1176, 2006. doi:10.1093/bioinformatics/btl023.

[17] J.S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29(6-7):1105–1119, 1990. doi:10.1002/bip.360290621.

[18] M. Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244(4900):48–52, April 1989. doi:10.1126/science.2468181.

[19] G Steger, H Hofmann, J Förtsch, HJ Gross, JW Randies, HL Sänger, and D Riesner. Conformational transitions in viroids and virusoids: comparison of results from energy minimization algorithm and from experimental data. *Journal of Biomolecular Structure and Dynamics*, 2(3):543–571, 1984. doi:10.1080/07391102.1984.10507591.

[20] Thomas R Einert and Roland R Netz. Theory for RNA folding, stretching, and melting including loops and salt. *Biophysical journal*, 100(11):2745–2753, 2011. doi:10.1016/j.bpj.2011.04.038.

[21] S.H. Bernhart, H. Tafer, U. Mückstein, C. Flamm, P.F. Stadler, and I.L. Hofacker. Partition function and base pairing probabilities of RNA heterodimers. *Algorithms for Molecular Biology*, 1(1):3, 2006. doi:10.1186/1748-7188-1-3.

[22] Katherine E. Deigan, Tian W. Li, David H. Mathews, and Kevin M. Weeks. Accurate SHAPE-directed RNA structure determination. *PNAS*, 106:97–102, 2009. doi:10.1073/pnas.080692910.

[23] Ronny Lorenz, Dominik Luntzer, Ivo L. Hofacker, Peter F. Stadler, and Michael T. Wolfinger. Shape directed rna folding. *Bioinformatics*, 32(1):145–147, 2016. doi:10.1093/bioinformatics/btv523.

[24] Pietro Boccaletto, Filip Stefaniak, Angana Ray, Andrea Cappannini, Sunandan Mukherjee, Elżbieta Purta, Małgorzata Kurkowska, Niloofar Shirvanizadeh, Eliana Destefanis, Paula Groza, and others. MODOMICS: a database of RNA modification pathways. 2021 update. *Nucleic Acids Research*, 50(D1):D231–D235, 2022. doi:10.1093/nar/gkab1083.

[25] Ronny Lorenz, Ivo L. Hofacker, and Peter F. Stadler. RNA folding with hard and soft constraints. *Algorithms for Molecular Biology*, 11(1):1–13, 2016. doi:10.1186/s13015-016-0070-z.

[26] Joe Sawada. A fast algorithm to generate necklaces with fixed content. *Theoretical Computer Science*, 301(1):477–489, 2003. doi:10.1016/S0304-3975(03)00049-5.

[27] Yann Ponty. Efficient sampling of RNA secondary structures from the boltzmann ensemble of low-energy: the boustrophedon method. *Journal of mathematical biology*, 56(1):107–127, 2008. doi:10.1007/s00285-007-0137-z.

[28] Stephan H Bernhart, Ivo L Hofacker, and Peter F Stadler. Local RNA base pairing probabilities in large sequences. *Bioinformatics*, 22(5):614–615, 2005. doi:10.1093/bioinformatics/btk014.

[29] Stephan H Bernhart, Ullrike Mückstein, and Ivo L Hofacker. RNA accessibility in cubic time. *Algorithms for Molecular Biology*, 6(1):3, 2011. doi:10.1186/1748-7188-6-3.

[30] Juraj Michálik, Hélène Touzet, and Yann Ponty. Efficient approximations of RNA kinetics landscape using non-redundant sampling. *Bioinformatics*, 33(14):i283–i292, 2017. doi:10.1093/bioinformatics/btx269.

[31] Kourosh Zarringhalam, Michelle M. Meyer, Ivan Dotu, Jeffrey H. Chuang, and Peter Clote. Integrating chemical footprinting data into RNA secondary structure prediction. *PLOS ONE*, 2012. doi:10.1371/journal.pone.0045160.

[32] Sean R Eddy. Computational analysis of conserved rna secondary structure in transcriptomes and genomes. *Annual review of biophysics*, 43:433–456, 2014. doi:10.1146/annurev-biophys-051013-022950.

[33] Fei Deng, Mirko Ledda, Sana Vaziri, and Sharon Aviran. Data-directed RNA secondary structure prediction using probabilistic modeling. *RNA*, 22(8):1109–1119, 2016. doi:10.1261/rna.055756.115.

[34] Elzbieta Kierzek, Xiaoju Zhang, Richard M Watson, Scott D Kennedy, Marta Szabat, Ryszard Kierzek, and David H Mathews. Secondary structure prediction for RNA sequences including N6-methyladenosine. *Nature communications*, 13(1):1–10, 2022. doi:10.1038/s41467-022-28817-4.

[35] Graham A Hudson, Richard J Bloomingdale, and Brent M Znosko. Thermodynamic contribution and nearest-neighbor parameters of pseudouridine-adenosine base pairs in oligoribonucleotides. *RNA*, 19(11):1474–1482, 2013. doi:10.1261/rna.039610.113.

[36] Daniel J Wright, Jamie L Rice, Dawn M Yanker, and Brent M Znosko. Nearest Neighbor Parameters for Inosine· Uridine Pairs in RNA Duplexes. *Biochemistry*, 46(15):4625–4634, 2007. doi:10.1021/bi0616910.

[37] Daniel J Wright, Christopher R Force, and Brent M Znosko. Stability of RNA duplexes containing inosine· cytosine pairs. *Nucleic Acids Research*, 46(22):12099–12108, 2018. doi:10.1093/nar/gky907.

[38] Katherine E Richardson and Brent M Znosko. Nearest-neighbor parameters for 7-deaza-adenosine· uridine base pairs in RNA duplexes. *RNA*, 22(6):934–942, 2016. doi:10.1261/rna.055277.115.

[39] Elizabeth A Jolley and Brent M Znosko. The loss of a hydrogen bond: Thermodynamic contributions of a non-standard nucleotide. *Nucleic acids research*, 45(3):1479–1487, 2017. doi:10.1093/nar/gkw830.

[40] Eva Freyhult, Vincent Moulton, and Paul Gardner. Predicting RNA structure using mutual information. *Applied bioinformatics*, 4(1):53–59, 2005. doi:10.2165/00822942-200504010-00006.

[41] Daniel Wiegreffe, Daniel Alexander, Peter F Stadler, and Dirk Zeckzer. RNApuzzler: efficient outerplanar drawing of RNA-secondary structures. *Bioinformatics*, 35(8):1342–1349, 2019. doi:10.1093/bioinformatics/bty817.

[42] R.E. Bruccoleri and G. Heinrich. An improved algorithm for nucleic acid secondary structure display. *Computer applications in the biosciences: CABIOS*, 4(1):167–173, 1988. doi:10.1093/bioinformatics/4.1.167.

[43] Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(1):26, 2011. doi:10.1186/1748-7188-6-26.

[44] I.L. Hofacker, W. Fontana, P.F. Stadler, L.S. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatshefte für Chemie/Chemical Monthly*, 125(2):167–188, 1994. URL: https://www.academia.edu/download/48689421/Fast_Folding_and_Comparison_of_RNA_Secon20160908-13624-1yg70az.pdf.

[45] B.A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Computer applications in the biosciences: CABIOS*, 6(4):309–318, 1990. doi:10.1093/bioinformatics/6.4.309.

PYTHON MODULE INDEX

r
RNA, 908

INDEX

Symbols

(
    command line option, 106
()
    command line option, 113
(possible
    command line option, 137
,
    command line option, 106, 113
.
    command line option, 106, 113
__E_IntLoop_Co (*C function*), 482
-3
    command line option, 157
-4
    command line option, 22, 25, 34, 45, 54, 60, 70, 80, 84, 89, 97, 103, 107, 114, 118, 122, 128, 138, 152, 158
-5
    command line option, 157
-A
    command line option, 121, 142
-B
    command line option, 49, 106, 113
-C
    command line option, 32, 44, 67, 122, 144, 149, 157
-D
    command line option, 49, 149
-E
    command line option, 34
-F
    command line option, 83
-I
    command line option, 123, 145
-K
    command line option, 22, 121
-L
    command line option, 22, 88, 95, 122, 127, 144
-M
    command line option, 121
-N
    command line option, 32, 121, 137, 145, 149
-O
    command line option, 125, 141
-P
    command line option, 22, 25, 33, 45, 51, 54, 60, 70, 80, 83, 89, 97, 103, 107, 114, 117, 122, 128, 138, 141, 152, 158
-Q
    command line option, 121
-R
    command line option, 34, 61, 83, 90
-S
    command line option, 22, 31, 43, 49, 66, 103, 117, 127, 138, 141, 149, 157
-T
    command line option, 22, 25, 33, 45, 51, 54, 60, 70, 83, 89, 97, 103, 107, 114, 117, 122, 128, 138, 152, 158
-U
    command line option, 121, 141
-V
    command line option, 23, 27, 37, 47, 50, 52, 56, 61, 72, 81, 85, 92, 98, 105, 109, 111, 115, 119, 121, 130, 135, 139, 145, 154, 159
-W
    command line option, 127
-X
    command line option, 49, 107, 113
--ImFeelingLucky
    command line option, 67
--MEA
    command line option, 31, 43, 66
--SS_cons
    command line option, 33
--Tmax
    command line option, 79
--Tmin
    command line option, 79
--WindowLength
    command line option, 122
--absolute-concentrations
    command line option, 102
--accessibility-dir
    command line option, 120
--alignmentLength
    command line option, 144
--alignment-mode
    command line option, 121, 142
--all_pf
    command line option, 42, 102

--allowFlipping
    command line option, 134
--aln
    command line option, 36, 86, 134
--aln-EPS
    command line option, 91
--aln-EPS-cols
    command line option, 36, 91, 134
--aln-EPS-ss
    command line option, 91
--aln-stk
    command line option, 36, 86
--alphabet
    command line option, 83
--auto-id
    command line option, 29, 41, 57, 63, 78, 87, 93, 100, 125, 132, 146
--backbone-length
    command line option, 23, 26, 35, 47, 52, 55, 61, 71, 81, 84, 91, 98, 104, 108, 115, 118, 123, 130, 139, 154, 159
--backtrack
    command line option, 49, 113
--backtrack-global
    command line option, 95
--batch
    command line option, 33, 44, 67, 150
--benchmark
    command line option, 30, 65
--benchmark-fuzzy
    command line option, 31, 65
--betaScale
    command line option, 31, 43, 66, 102, 117, 127, 149
--binaries
    command line option, 125
--binary
    command line option, 120
--bm-ignore-compat
    command line option, 31, 65
--bm-output
    command line option, 30, 65
--bm-output-append
    command line option, 30, 65
--bm-rm-nc
    command line option, 65
--bm-rm-pk
    command line option, 30, 65
--bppmThreshold
    command line option, 32, 43, 67, 103
--canonicalBPonly
    command line option, 44, 67, 150
--centroid
    command line option, 43
--cfactor
    command line option, 34, 60, 90
--circ
    command line option, 22, 32, 59, 66, 79, 149
--color
    command line option, 35
--color-min-sat
    command line option, 35, 91
--color-threshold
    command line option, 35, 91
--commands
    command line option, 45, 68, 96, 103, 128, 150
--compare
    command line option, 49, 113
--concentrations
    command line option, 43, 102
--concfile
    command line option, 43, 102
--constraint
    command line option, 32, 44, 67, 122, 144, 149, 157
--continuous-ids
    command line option, 29
--contributions
    command line option, 156
--convert-to-bin
    command line option, 121
--covar
    command line option, 133
--covar-min-sat
    command line option, 133
--covar-threshold
    command line option, 133
--csv
    command line option, 86
--csv-delim
    command line option, 40
--csv-noheader
    command line option, 40
--cutoff
    command line option, 117, 125
--dangles
    command line option, 23, 25, 34, 46, 52, 55, 60, 71, 80, 84, 89, 97, 104, 107, 114, 129, 138, 153, 158
--deltaEnergy
    command line option, 25, 54, 148
--deltaEnergyPost
    command line option, 148
--detailed-help
    command line option, 23, 27, 37, 47, 50, 52, 55, 61, 72, 81, 85, 91, 98, 105, 109, 111, 115, 119, 123, 130, 135, 139, 145, 154, 159
--direct-redraw
    command line option, 145
--distance
    command line option, 49
--dos
    command line option, 149
--dump
    command line option, 110
--duplex-distance

command line option, 122, 143
--en-only
command line option, 148
--endgaps
command line option, 34, 107
--energyCutoff
command line option, 117
--energyModel
command line option, 35, 47, 60, 71, 81, 84, 90, 98, 104, 108, 114, 129, 139, 154, 159
--energy-threshold
command line option, 122, 142
--enforceConstraint
command line option, 33, 44, 68, 150
--extend3
command line option, 157
--extend5
command line option, 157
--extension-cost
command line option, 121, 142
--fast-folding
command line option, 121, 142
--filename-delim
command line option, 30, 41, 64, 88, 94, 101, 126, 133, 147
--filename-full
command line option, 41, 64, 94, 101, 126, 133, 147
--final
command line option, 83
--from-RNAplfold
command line option, 141
--from-RNAup
command line option, 141
--full-help
command line option, 23, 27, 36, 47, 49, 52, 55, 61, 72, 81, 85, 91, 98, 105, 108, 111, 115, 118, 123, 130, 134, 139, 145, 154, 159
--function
command line option, 83
--gape
command line option, 107
--gapo
command line option, 107
--gquad
command line option, 32, 44, 59, 67, 80, 88, 96, 103, 149
--hashtable-bits
command line option, 51
--helical-rise
command line option, 23, 26, 35, 47, 52, 55, 61, 71, 81, 84, 90, 98, 104, 108, 114, 118, 123, 129, 139, 154, 159
--help
command line option, 23, 27, 36, 47, 49, 52, 55, 61, 72, 81, 85, 91, 98, 104, 108, 111, 115, 118, 123, 130, 134, 139, 145, 154, 159
--id-delim
command line option, 29, 41, 58, 64, 78, 87, 93, 100, 125, 132, 147
--id-digits
command line option, 29, 41, 58, 64, 78, 87, 93, 101, 126, 132, 147
--id-prefix
command line option, 29, 41, 58, 63, 78, 87, 93, 100, 125, 132, 147
--id-start
command line option, 29, 41, 58, 64, 79, 88, 94, 101, 126, 132, 147
--ignoreAncestorIntersections
command line option, 134
--ignoreExteriorIntersections
command line option, 134
--ignoreSiblingIntersections
command line option, 134
--include_both
command line option, 157
--infile
command line option, 57, 63, 77, 93, 131, 146
--init-strategy
command line option, 83
--initialStepSize
command line option, 137
--initialVector
command line option, 137
--input
command line option, 110
--input-format
command line option, 28, 86
--interaction_first
command line option, 157
--interaction_pairwise
command line option, 157
--interaction-length
command line option, 120
--intermediatePath
command line option, 137
--ipoints
command line option, 79
--jobs
command line option, 28, 40, 57, 63, 77, 100, 131
--k-concentration
command line option, 121
--layout-type
command line option, 36, 72, 134
--logML
command line option, 60, 153
--log-call
command line option, 21, 26, 30, 42, 49, 51, 54, 58, 65, 79, 82, 88, 94, 101, 106, 111, 113, 116, 120, 126, 133, 136, 142, 148, 156
--log-file
command line option, 21, 26, 30, 42, 49, 51, 54, 58, 65, 79, 82, 88, 94, 101, 106, 111, 113, 116, 120, 126, 133, 136, 142, 148, 156

--log-func
    command line option, 21, 26, 30, 42, 49, 51, 54, 58, 65, 79, 82, 88, 94, 101, 106, 111, 113, 117, 120, 126, 133, 136, 142, 148, 156
--log-level
    command line option, 21, 26, 30, 42, 49, 51, 54, 58, 64, 79, 82, 88, 94, 101, 106, 111, 113, 116, 120, 126, 133, 136, 141, 147, 156
--log-time
    command line option, 21, 26, 30, 42, 49, 51, 54, 58, 65, 79, 82, 88, 94, 101, 106, 111, 113, 116, 120, 126, 133, 136, 142, 148, 156
--maxBPspan
    command line option, 32, 44, 67, 80, 88, 103, 138, 149
--maxDist1
    command line option, 22
--maxDist2
    command line option, 22
--max-energy
    command line option, 51
--maximal-duplex-box-length
    command line option, 143
--maximal-snoRNA-duplex-length
    command line option, 143
--maximal-snoRNA-stem-loop-length
    command line option, 143
--maximal-stem-asymmetry
    command line option, 144
--mg-concentration
    command line option, 121
--minImprovement
    command line option, 137
--minStepSize
    command line option, 137
--minimal-duplex
    command line option, 143
--minimal-duplex-box-length
    command line option, 143
--minimal-duplex-stem-energy
    command line option, 144
--minimal-loop-energy
    command line option, 142
--minimal-lower-stem-energy
    command line option, 144
--minimal-right-duplex
    command line option, 142
--minimal-snoRNA-duplex-length
    command line option, 143
--minimal-snoRNA-stem-loop-length
    command line option, 143
--minimal-total-energy
    command line option, 144
--minimizer
    command line option, 137
--minimizerTolerance
    command line option, 137
--mis
    command line option, 28, 57, 87, 131
--mod-file
    command line option, 46, 70, 97, 129, 153
--mode
    command line option, 107
--modifications
    command line option, 46, 70, 97, 128, 152
--motif
    command line option, 68, 150
--msa
    command line option, 57, 131
--na-concentration
    command line option, 121
--neighborhood
    command line option, 22
--nfactor
    command line option, 34, 60, 90
--noBT
    command line option, 22
--noClosingGU
    command line option, 23, 26, 34, 47, 55, 71, 80, 84, 90, 98, 104, 108, 114, 118, 129, 139, 153, 158
--noDP
    command line option, 36, 72
--noGU
    command line option, 23, 26, 34, 46, 55, 71, 80, 84, 90, 98, 104, 108, 114, 118, 129, 139, 153, 158
--noLP
    command line option, 26, 34, 46, 55, 71, 80, 90, 98, 104, 108, 114, 118, 129, 139, 153, 158
--noOptimization
    command line option, 134
--noPS
    command line option, 36, 47, 72
--noTetra
    command line option, 22, 25, 34, 45, 54, 60, 70, 80, 84, 89, 97, 103, 107, 114, 118, 122, 128, 138, 152, 158
--no_header
    command line option, 156
--no_output_file
    command line option, 156
--noconv
    command line option, 21, 29, 40, 53, 57, 63, 78, 87, 93, 100, 106, 112, 116, 125, 146, 156
--nonRedundant
    command line option, 32, 137, 149
--nsp
    command line option, 26, 35, 47, 55, 60, 71, 81, 84, 90, 98, 104, 108, 114, 118, 129, 139, 153, 158
--numThreads
    command line option, 21, 51, 136
--objectiveFunction
    command line option, 137
--old

command line option, 35, 61
--opening_energies
command line option, 125
--outfile
command line option, 63, 93, 146
--output
command line option, 110
--output_directory
command line option, 141
--output-format
command line option, 40, 132
--paramFile
command line option, 22, 25, 33, 45, 51, 54, 60, 70, 80, 83, 89, 97, 103, 107, 114, 117, 122, 128, 138, 152, 158
--partfunc
command line option, 22, 31, 42, 66, 102
--pfScale
command line option, 22, 31, 43, 66, 103, 117, 127, 138, 149, 157
--plex_output
command line option, 125
--post
command line option, 132
--pre
command line option, 132
--printAlignment
command line option, 106
--print_onthefly
command line option, 125
--probe-concentration
command line option, 121
--probe-mode
command line option, 121
--produce-ps
command line option, 123, 145
--query
command line option, 120, 141
--quiet
command line option, 28, 86
--random-seed
command line option, 32, 149
--repeat
command line option, 83
--ribosum_file
command line option, 34, 61, 90
--ribosum_scoring
command line option, 35, 61, 90
--salt
command line option, 23, 25, 34, 46, 52, 54, 60, 70, 80, 84, 89, 97, 103, 107, 114, 118, 123, 128, 138, 152, 158
--saltInit
command line option, 25, 46, 54, 123, 158
--sampleSize
command line option, 137
--scale-accessibility
command line option, 121
--sci
command line option, 32
--sequence
command line option, 51
--seqw
command line option, 107
--shape
command line option, 33, 44, 59, 69, 89, 96, 127, 151
--shapeConversion
command line option, 45, 59, 69, 96, 128, 136, 152
--shapeMethod
command line option, 33, 44, 59, 69, 89, 96, 127, 151
--shapiro
command line option, 49
--silent
command line option, 110
--sorted
command line option, 25, 53, 148
--sp-data
command line option, 68, 150
--sp-preprocess
command line option, 69, 151
--sp-strategy
command line option, 68, 150
--span
command line option, 95, 127
--split-contributions
command line option, 87
--stepsize
command line option, 79
--stochBT
command line option, 22, 32, 148
--stochBT_en
command line option, 32, 148
--subopts
command line option, 117
--suffix
command line option, 141
--target
command line option, 120, 141
--tauSigmaRatio
command line option, 136
--temp
command line option, 22, 25, 33, 45, 51, 54, 60, 70, 83, 89, 97, 103, 107, 114, 117, 122, 128, 138, 152, 158
--threshold
command line option, 88
--tris-concentration
command line option, 121
--ulength
command line option, 127, 156
--unordered
command line option, 28, 40, 57, 63, 78, 100
--vanilla

command line option, 110
--verbose
command line option, 21, 25, 28, 40, 49, 50, 53, 56, 62, 77, 82, 86, 92, 99, 105, 109, 112, 116, 119, 124, 131, 135, 141, 146, 155
--version
command line option, 23, 27, 37, 47, 50, 52, 56, 61, 72, 81, 85, 92, 98, 105, 109, 111, 115, 119, 123, 130, 135, 139, 145, 154, 159
--window
command line option, 157
--winsize
command line option, 127
--without-BulgeE
command line option, 110
--without-Dangle3
command line option, 111
--without-Dangle5
command line option, 111
--without-HairpinE
command line option, 110
--without-IntE
command line option, 110
--without-Misc
command line option, 111
--without-MismatchE
command line option, 110
--without-MismatchH
command line option, 110
--without-MismatchI
command line option, 110
--without-MismatchM
command line option, 111
--without-MultiE
command line option, 110
--without-StackE
command line option, 110
--zscore
command line option, 95
--zscore-pre-filter
command line option, 95
--zscore-report-subsumed
command line option, 95
--zuker
command line option, 149
-a
command line option, 42, 57, 83, 102, 120, 131, 144
-b
command line option, 51, 95, 120, 125, 144, 157
-c
command line option, 22, 32, 43, 59, 66, 79, 102, 117, 121, 125, 142, 149, 156
-d
command line option, 23, 25, 34, 46, 52, 55, 60, 71, 80, 84, 89, 97, 104, 107, 114, 129, 138, 143, 153, 158
-e
command line option, 25, 51, 54, 117, 122, 142, 148
-f
command line option, 28, 43, 83, 86, 102, 121, 132, 142
-g
command line option, 32, 44, 59, 67, 80, 88, 96, 103, 149
-h
command line option, 23, 27, 36, 47, 49, 52, 55, 61, 72, 81, 85, 91, 98, 104, 108, 111, 115, 118, 123, 130, 134, 139, 154, 159
-i
command line option, 57, 63, 77, 93, 110, 131, 146
-j
command line option, 21, 28, 40, 51, 57, 63, 77, 100, 131, 136, 143
-k
command line option, 121, 143
-l
command line option, 120, 142
-m
command line option, 46, 70, 79, 97, 128, 143, 152
-n
command line option, 29, 143
-o
command line option, 63, 93, 110, 125, 142, 146, 156
-p
command line option, 22, 31, 42, 66, 102, 121, 148
-q
command line option, 28, 86, 120, 143
-r
command line option, 35, 61, 90
-s
command line option, 25, 32, 51, 53, 117, 141, 148
-t
command line option, 36, 72, 120, 134, 141
-u
command line option, 127, 156
-v
command line option, 21, 25, 28, 40, 49, 50, 53, 56, 62, 77, 82, 86, 92, 99, 105, 109, 112, 116, 119, 124, 131, 135, 143, 146, 155
-w
command line option, 143, 157
-x
command line option, 144
-y
command line option, 144
-z
command line option, 95, 122, 149
{

command line option, 106
{}
command line option, 113
``m``
command line option, 107
|
command line option, 106, 113

A

abstract_shapes() (*in module RNA*), 929
add_auxdata() (*RNA.fold_compound method*), 981
add_callback() (*RNA.fold_compound method*), 982
add_root() (*in module RNA*), 930
advance() (*RNA.SwigPyIterator method*), 927
alias (*RNA.md attribute*), 1047
alias (*RNA.md property*), 1050
aliduplex_subopt() (*in module RNA*), 930
aliduplexfold() (*in module RNA*), 930
alifold (*C function*), 297
alifold() (*in module RNA*), 930
alignment (*RNA.fold_compound attribute*), 978
aliLfold (*C function*), 305
aliLfold() (*in module RNA*), 930
aliLfold_cb (*C function*), 305
aliLfold_cb() (*in module RNA*), 930
alipbacktrack (*C function*), 333
alipf_circ_fold (*C function*), 332
alipf_fold (*C function*), 331
alipf_fold_par (*C function*), 331
allowFlipping (*RNA.plot_options_puzzler attribute*), 1077, 1078
allowFlipping (*RNA.plot_options_puzzler property*), 1078
aln_consensus_mis() (*in module RNA*), 931
aln_consensus_sequence() (*in module RNA*), 931
aln_conservation_col() (*in module RNA*), 932
aln_conservation_struct() (*in module RNA*), 932
aln_mpi() (*in module RNA*), 933
aln_pscore() (*in module RNA*), 933
alpha (*RNA.exp_param attribute*), 960, 964
alpha (*RNA.exp_param property*), 965
append() (*RNA.ConstCharVector method*), 909
append() (*RNA.CoordinateVector method*), 910
append() (*RNA.DoubleDoubleVector method*), 910
append() (*RNA.DoubleVector method*), 911
append() (*RNA.DuplexVector method*), 912
append() (*RNA.ElemProbVector method*), 916
append() (*RNA.HeatCapacityVector method*), 917
append() (*RNA.HelixVector method*), 918
append() (*RNA.IntIntVector method*), 919
append() (*RNA.IntVector method*), 919
append() (*RNA.MoveVector method*), 922
append() (*RNA.PathVector method*), 924
append() (*RNA.SOLUTIONVector method*), 925
append() (*RNA.StringVector method*), 926
append() (*RNA.SuboptVector method*), 927
append() (*RNA.UIntUIntVector method*), 928
append() (*RNA.UIntVector method*), 929
assign() (*RNA.ConstCharVector method*), 909
assign() (*RNA.CoordinateVector method*), 910
assign() (*RNA.DoubleDoubleVector method*), 910
assign() (*RNA.DoubleVector method*), 911
assign() (*RNA.DuplexVector method*), 912
assign() (*RNA.ElemProbVector method*), 916
assign() (*RNA.HeatCapacityVector method*), 917
assign() (*RNA.HelixVector method*), 918
assign() (*RNA.IntIntVector method*), 919
assign() (*RNA.IntVector method*), 919
assign() (*RNA.MoveVector method*), 922
assign() (*RNA.PathVector method*), 924
assign() (*RNA.SOLUTIONVector method*), 925
assign() (*RNA.StringVector method*), 926
assign() (*RNA.SuboptVector method*), 927
assign() (*RNA.UIntUIntVector method*), 928
assign() (*RNA.UIntVector method*), 929
assign_plist_from_db (*C function*), 342
assign_plist_from_pr (*C function*), 343
assign_plist_gquad_from_pr() (*in module RNA*), 933
aux_grammar (*RNA.fold_compound attribute*), 979
auxdata (*RNA.fold_compound attribute*), 979

B

b2C() (*in module RNA*), 933
b2HIT() (*in module RNA*), 934
b2Shapiro() (*in module RNA*), 934
back() (*RNA.ConstCharVector method*), 909
back() (*RNA.CoordinateVector method*), 910
back() (*RNA.DoubleDoubleVector method*), 911
back() (*RNA.DoubleVector method*), 911
back() (*RNA.DuplexVector method*), 912
back() (*RNA.ElemProbVector method*), 916
back() (*RNA.HeatCapacityVector method*), 917
back() (*RNA.HelixVector method*), 918
back() (*RNA.IntIntVector method*), 919
back() (*RNA.IntVector method*), 919
back() (*RNA.MoveVector method*), 922
back() (*RNA.PathVector method*), 924
back() (*RNA.SOLUTIONVector method*), 925
back() (*RNA.StringVector method*), 926
back() (*RNA.SuboptVector method*), 927
back() (*RNA.UIntUIntVector method*), 928
back() (*RNA.UIntVector method*), 929
backbone_length (*C var*), 435
backbone_length (*RNA.md attribute*), 1048
backbone_length (*RNA.md property*), 1050
backtrack (*RNA.md attribute*), 1046
backtrack (*RNA.md property*), 1050
backtrack() (*RNA.fold_compound method*), 982
backtrack_fold_from_pair (*C function*), 305
backtrack_GQuad_IntLoop_L() (*in module RNA*), 934
backtrack_GQuad_IntLoop_L_comparative() (*in module RNA*), 934
backtrack_type (*C var*), 433
backtrack_type (*RNA.md attribute*), 1046

backtrack_type (*RNA.md property*), 1050
basepair (*class in RNA*), 934
begin() (*RNA.ConstCharVector method*), 909
begin() (*RNA.CoordinateVector method*), 910
begin() (*RNA.DoubleDoubleVector method*), 911
begin() (*RNA.DoubleVector method*), 911
begin() (*RNA.DuplexVector method*), 912
begin() (*RNA.ElemProbVector method*), 916
begin() (*RNA.HeatCapacityVector method*), 917
begin() (*RNA.HelixVector method*), 918
begin() (*RNA.IntIntVector method*), 919
begin() (*RNA.IntVector method*), 920
begin() (*RNA.MoveVector method*), 922
begin() (*RNA.PathVector method*), 924
begin() (*RNA.SOLUTIONVector method*), 925
begin() (*RNA.StringVector method*), 926
begin() (*RNA.SuboptVector method*), 927
begin() (*RNA.UIntUIntVector method*), 928
begin() (*RNA.UIntVector method*), 929
beginning
    command line option, 143
benchmark() (*RNA.fold_compound method*), 983
betaScale (*RNA.md attribute*), 1045
betaScale (*RNA.md property*), 1050
bondT (*C type*), 749
boustrophedon() (*in module RNA*), 935
bp_distance() (*in module RNA*), 936
bpdist (*RNA.fold_compound attribute*), 980
bpp() (*RNA.fold_compound method*), 983
bulge (*RNA.param attribute*), 1056, 1060
bulge (*RNA.param property*), 1064

C

c (*RNA.mx_mfe property*), 1055
capacity() (*RNA.ConstCharVector method*), 909
capacity() (*RNA.CoordinateVector method*), 910
capacity() (*RNA.DoubleDoubleVector method*), 911
capacity() (*RNA.DoubleVector method*), 911
capacity() (*RNA.DuplexVector method*), 912
capacity() (*RNA.ElemProbVector method*), 917
capacity() (*RNA.HeatCapacityVector method*), 917
capacity() (*RNA.HelixVector method*), 918
capacity() (*RNA.IntIntVector method*), 919
capacity() (*RNA.IntVector method*), 920
capacity() (*RNA.MoveVector method*), 922
capacity() (*RNA.PathVector method*), 924
capacity() (*RNA.SOLUTIONVector method*), 925
capacity() (*RNA.StringVector method*), 926
capacity() (*RNA.SuboptVector method*), 927
capacity() (*RNA.UIntUIntVector method*), 928
capacity() (*RNA.UIntVector method*), 929
cast() (*RNA.doubleArray method*), 943
cast() (*RNA.floatArray method*), 975
cast() (*RNA.intArray method*), 1040
cdata() (*in module RNA*), 936
centroid() (*in module RNA*), 936
centroid() (*RNA.fold_compound method*), 983
checkAncestorIntersections (*RNA.plot_options_puzzler attribute*), 1077, 1078
checkAncestorIntersections (*RNA.plot_options_puzzler property*), 1078
checkExteriorIntersections (*RNA.plot_options_puzzler attribute*), 1077, 1078
checkExteriorIntersections (*RNA.plot_options_puzzler property*), 1078
checkSiblingIntersections (*RNA.plot_options_puzzler attribute*), 1077, 1078
checkSiblingIntersections (*RNA.plot_options_puzzler property*), 1078
circ (*C var*), 433
circ (*RNA.md attribute*), 1046
circ (*RNA.md property*), 1050
circ_alpha0 (*RNA.md attribute*), 1048
circ_alpha0 (*RNA.md property*), 1050
circ_penalty (*RNA.md attribute*), 1046
circ_penalty (*RNA.md property*), 1050
circalifold (*C function*), 298
circalifold() (*in module RNA*), 936
circfold (*C function*), 303
circfold() (*in module RNA*), 937
clear() (*RNA.ConstCharVector method*), 909
clear() (*RNA.CoordinateVector method*), 910
clear() (*RNA.DoubleDoubleVector method*), 911
clear() (*RNA.DoubleVector method*), 912
clear() (*RNA.DuplexVector method*), 912
clear() (*RNA.ElemProbVector method*), 917
clear() (*RNA.HeatCapacityVector method*), 917
clear() (*RNA.HelixVector method*), 918
clear() (*RNA.IntIntVector method*), 919
clear() (*RNA.IntVector method*), 920
clear() (*RNA.MoveVector method*), 922
clear() (*RNA.PathVector method*), 924
clear() (*RNA.SOLUTIONVector method*), 925
clear() (*RNA.StringVector method*), 926
clear() (*RNA.SuboptVector method*), 927
clear() (*RNA.UIntUIntVector method*), 928
clear() (*RNA.UIntVector method*), 929
cmd (*class in RNA*), 938
co_pf_fold (*C function*), 334
co_pf_fold() (*in module RNA*), 938
co_pf_fold_par (*C function*), 335
cofold (*C function*), 299
cofold() (*in module RNA*), 938
cofold_par (*C function*), 299
cofoldF (*C type*), 493
command line option
    (, 106
    (), 113
    (possible, 137
    ,, 106, 113
    ., 106, 113
    -3, 157

-4, 22, 25, 34, 45, 54, 60, 70, 80, 84, 89, 97, 103, 107, 114, 118, 122, 128, 138, 152, 158
-5, 157
-A, 121, 142
-B, 49, 106, 113
-C, 32, 44, 67, 122, 144, 149, 157
-D, 49, 149
-E, 34
-F, 83
-I, 123, 145
-K, 22, 121
-L, 22, 88, 95, 122, 127, 144
-M, 121
-N, 32, 121, 137, 145, 149
-O, 125, 141
-P, 22, 25, 33, 45, 51, 54, 60, 70, 80, 83, 89, 97, 103, 107, 114, 117, 122, 128, 138, 141, 152, 158
-Q, 121
-R, 34, 61, 83, 90
-S, 22, 31, 43, 49, 66, 103, 117, 127, 138, 141, 149, 157
-T, 22, 25, 33, 45, 51, 54, 60, 70, 83, 89, 97, 103, 107, 114, 117, 122, 128, 138, 152, 158
-U, 121, 141
-V, 23, 27, 37, 47, 50, 52, 56, 61, 72, 81, 85, 92, 98, 105, 109, 111, 115, 119, 121, 130, 135, 139, 145, 154, 159
-W, 127
-X, 49, 107, 113
--ImFeelingLucky, 67
--MEA, 31, 43, 66
--SS_cons, 33
--Tmax, 79
--Tmin, 79
--WindowLength, 122
--absolute-concentrations, 102
--accessibility-dir, 120
--alignmentLength, 144
--alignment-mode, 121, 142
--all_pf, 42, 102
--allowFlipping, 134
--aln, 36, 86, 134
--aln-EPS, 91
--aln-EPS-cols, 36, 91, 134
--aln-EPS-ss, 91
--aln-stk, 36, 86
--alphabet, 83
--auto-id, 29, 41, 57, 63, 78, 87, 93, 100, 125, 132, 146
--backbone-length, 23, 26, 35, 47, 52, 55, 61, 71, 81, 84, 91, 98, 104, 108, 115, 118, 123, 130, 139, 154, 159
--backtrack, 49, 113
--backtrack-global, 95
--batch, 33, 44, 67, 150
--benchmark, 30, 65
--benchmark-fuzzy, 31, 65
--betaScale, 31, 43, 66, 102, 117, 127, 149
--binaries, 125
--binary, 120
--bm-ignore-compat, 31, 65
--bm-output, 30, 65
--bm-output-append, 30, 65
--bm-rm-nc, 65
--bm-rm-pk, 30, 65
--bppmThreshold, 32, 43, 67, 103
--canonicalBPonly, 44, 67, 150
--centroid, 43
--cfactor, 34, 60, 90
--circ, 22, 32, 59, 66, 79, 149
--color, 35
--color-min-sat, 35, 91
--color-threshold, 35, 91
--commands, 45, 68, 96, 103, 128, 150
--compare, 49, 113
--concentrations, 43, 102
--concfile, 43, 102
--constraint, 32, 44, 67, 122, 144, 149, 157
--continuous-ids, 29
--contributions, 156
--convert-to-bin, 121
--covar, 133
--covar-min-sat, 133
--covar-threshold, 133
--csv, 86
--csv-delim, 40
--csv-noheader, 40
--cutoff, 117, 125
--dangles, 23, 25, 34, 46, 52, 55, 60, 71, 80, 84, 89, 97, 104, 107, 114, 129, 138, 153, 158
--deltaEnergy, 25, 54, 148
--deltaEnergyPost, 148
--detailed-help, 23, 27, 37, 47, 50, 52, 55, 61, 72, 81, 85, 91, 98, 105, 109, 111, 115, 119, 123, 130, 135, 139, 145, 154, 159
--direct-redraw, 145
--distance, 49
--dos, 149
--dump, 110
--duplex-distance, 122, 143
--en-only, 148
--endgaps, 34, 107
--energyCutoff, 117
--energyModel, 35, 47, 60, 71, 81, 84, 90, 98, 104, 108, 114, 129, 139, 154, 159
--energy-threshold, 122, 142
--enforceConstraint, 33, 44, 68, 150
--extend3, 157
--extend5, 157
--extension-cost, 121, 142
--fast-folding, 121, 142
--filename-delim, 30, 41, 64, 88, 94, 101, 126, 133, 147
--filename-full, 41, 64, 94, 101, 126, 133, 147
--final, 83

--from-RNAplfold, 141
--from-RNAup, 141
--full-help, 23, 27, 36, 47, 49, 52, 55, 61, 72, 81, 85, 91, 98, 105, 108, 111, 115, 118, 123, 130, 134, 139, 145, 154, 159
--function, 83
--gape, 107
--gapo, 107
--gquad, 32, 44, 59, 67, 80, 88, 96, 103, 149
--hashtable-bits, 51
--helical-rise, 23, 26, 35, 47, 52, 55, 61, 71, 81, 84, 90, 98, 104, 108, 114, 118, 123, 129, 139, 154, 159
--help, 23, 27, 36, 47, 49, 52, 55, 61, 72, 81, 85, 91, 98, 104, 108, 111, 115, 118, 123, 130, 134, 139, 145, 154, 159
--id-delim, 29, 41, 58, 64, 78, 87, 93, 100, 125, 132, 147
--id-digits, 29, 41, 58, 64, 78, 87, 93, 101, 126, 132, 147
--id-prefix, 29, 41, 58, 63, 78, 87, 93, 100, 125, 132, 147
--id-start, 29, 41, 58, 64, 79, 88, 94, 101, 126, 132, 147
--ignoreAncestorIntersections, 134
--ignoreExteriorIntersections, 134
--ignoreSiblingIntersections, 134
--include_both, 157
--infile, 57, 63, 77, 93, 131, 146
--init-strategy, 83
--initialStepSize, 137
--initialVector, 137
--input, 110
--input-format, 28, 86
--interaction_first, 157
--interaction_pairwise, 157
--interaction-length, 120
--intermediatePath, 137
--ipoints, 79
--jobs, 28, 40, 57, 63, 77, 100, 131
--k-concentration, 121
--layout-type, 36, 72, 134
--logML, 60, 153
--log-call, 21, 26, 30, 42, 49, 51, 54, 58, 65, 79, 82, 88, 94, 101, 106, 111, 113, 116, 120, 126, 133, 136, 142, 148, 156
--log-file, 21, 26, 30, 42, 49, 51, 54, 58, 65, 79, 82, 88, 94, 101, 106, 111, 113, 116, 120, 126, 133, 136, 142, 148, 156
--log-func, 21, 26, 30, 42, 49, 51, 54, 58, 65, 79, 82, 88, 94, 101, 106, 111, 113, 117, 120, 126, 133, 136, 142, 148, 156
--log-level, 21, 26, 30, 42, 49, 51, 54, 58, 64, 79, 82, 88, 94, 101, 106, 111, 113, 116, 120, 126, 133, 136, 141, 147, 156
--log-time, 21, 26, 30, 42, 49, 51, 54, 58, 65, 79, 82, 88, 94, 101, 106, 111, 113, 116, 120, 126, 133, 136, 142, 148, 156
--maxBPspan, 32, 44, 67, 80, 88, 103, 138, 149
--maxDist1, 22
--maxDist2, 22
--max-energy, 51
--maximal-duplex-box-length, 143
--maximal-snoRNA-duplex-length, 143
--maximal-snoRNA-stem-loop-length, 143
--maximal-stem-asymmetry, 144
--mg-concentration, 121
--minImprovement, 137
--minStepSize, 137
--minimal-duplex, 143
--minimal-duplex-box-length, 143
--minimal-duplex-stem-energy, 144
--minimal-loop-energy, 142
--minimal-lower-stem-energy, 144
--minimal-right-duplex, 142
--minimal-snoRNA-duplex-length, 143
--minimal-snoRNA-stem-loop-length, 143
--minimal-total-energy, 144
--minimizer, 137
--minimizerTolerance, 137
--mis, 28, 57, 87, 131
--mod-file, 46, 70, 97, 129, 153
--mode, 107
--modifications, 46, 70, 97, 128, 152
--motif, 68, 150
--msa, 57, 131
--na-concentration, 121
--neighborhood, 22
--nfactor, 34, 60, 90
--noBT, 22
--noClosingGU, 23, 26, 34, 47, 55, 71, 80, 84, 90, 98, 104, 108, 114, 118, 129, 139, 153, 158
--noDP, 36, 72
--noGU, 23, 26, 34, 46, 55, 71, 80, 84, 90, 98, 104, 108, 114, 118, 129, 139, 153, 158
--noLP, 26, 34, 46, 55, 71, 80, 90, 98, 104, 108, 114, 118, 129, 139, 153, 158
--noOptimization, 134
--noPS, 36, 47, 72
--noTetra, 22, 25, 34, 45, 54, 60, 70, 80, 84, 89, 97, 103, 107, 114, 118, 122, 128, 138, 152, 158
--no_header, 156
--no_output_file, 156
--noconv, 21, 29, 40, 53, 57, 63, 78, 87, 93, 100, 106, 112, 116, 125, 146, 156
--nonRedundant, 32, 137, 149
--nsp, 26, 35, 47, 55, 60, 71, 81, 84, 90, 98, 104, 108, 114, 118, 129, 139, 153, 158
--numThreads, 21, 51, 136
--objectiveFunction, 137
--old, 35, 61
--opening_energies, 125
--outfile, 63, 93, 146
--output, 110
--output_directory, 141

--output-format, 40, 132
--paramFile, 22, 25, 33, 45, 51, 54, 60, 70, 80, 83, 89, 97, 103, 107, 114, 117, 122, 128, 138, 152, 158
--partfunc, 22, 31, 42, 66, 102
--pfScale, 22, 31, 43, 66, 103, 117, 127, 138, 149, 157
--plex_output, 125
--post, 132
--pre, 132
--printAlignment, 106
--print_onthefly, 125
--probe-concentration, 121
--probe-mode, 121
--produce-ps, 123, 145
--query, 120, 141
--quiet, 28, 86
--random-seed, 32, 149
--repeat, 83
--ribosum_file, 34, 61, 90
--ribosum_scoring, 35, 61, 90
--salt, 23, 25, 34, 46, 52, 54, 60, 70, 80, 84, 89, 97, 103, 107, 114, 118, 123, 128, 138, 152, 158
--saltInit, 25, 46, 54, 123, 158
--sampleSize, 137
--scale-accessibility, 121
--sci, 32
--sequence, 51
--seqw, 107
--shape, 33, 44, 59, 69, 89, 96, 127, 151
--shapeConversion, 45, 59, 69, 96, 128, 136, 152
--shapeMethod, 33, 44, 59, 69, 89, 96, 127, 151
--shapiro, 49
--silent, 110
--sorted, 25, 53, 148
--sp-data, 68, 150
--sp-preprocess, 69, 151
--sp-strategy, 68, 150
--span, 95, 127
--split-contributions, 87
--stepsize, 79
--stochBT, 22, 32, 148
--stochBT_en, 32, 148
--subopts, 117
--suffix, 141
--target, 120, 141
--tauSigmaRatio, 136
--temp, 22, 25, 33, 45, 51, 54, 60, 70, 83, 89, 97, 103, 107, 114, 117, 122, 128, 138, 152, 158
--threshold, 88
--tris-concentration, 121
--ulength, 127, 156
--unordered, 28, 40, 57, 63, 78, 100
--vanilla, 110
--verbose, 21, 25, 28, 40, 49, 50, 53, 56, 62, 77, 82, 86, 92, 99, 105, 109, 112, 116, 119, 124, 131, 135, 141, 146, 155
--version, 23, 27, 37, 47, 50, 52, 56, 61, 72, 81, 85, 92, 98, 105, 109, 111, 115, 119, 123, 130, 135, 139, 145, 154, 159
--window, 157
--winsize, 127
--without-BulgeE, 110
--without-Dangle3, 111
--without-Dangle5, 111
--without-HairpinE, 110
--without-IntE, 110
--without-Misc, 111
--without-MismatchE, 110
--without-MismatchH, 110
--without-MismatchI, 110
--without-MismatchM, 111
--without-MultiE, 110
--without-StackE, 110
--zscore, 95
--zscore-pre-filter, 95
--zscore-report-subsumed, 95
--zuker, 149
-a, 42, 57, 83, 102, 120, 131, 144
-b, 51, 95, 120, 125, 144, 157
-c, 22, 32, 43, 59, 66, 79, 102, 117, 121, 125, 142, 149, 156
-d, 23, 25, 34, 46, 52, 55, 60, 71, 80, 84, 89, 97, 104, 107, 114, 129, 138, 143, 153, 158
-e, 25, 51, 54, 117, 122, 142, 148
-f, 28, 43, 83, 86, 102, 121, 132, 142
-g, 32, 44, 59, 67, 80, 88, 96, 103, 149
-h, 23, 27, 36, 47, 49, 52, 55, 61, 72, 81, 85, 91, 98, 104, 108, 111, 115, 118, 123, 130, 134, 139, 154, 159
-i, 57, 63, 77, 93, 110, 131, 146
-j, 21, 28, 40, 51, 57, 63, 77, 100, 131, 136, 143
-k, 121, 143
-l, 120, 142
-m, 46, 70, 79, 97, 128, 143, 152
-n, 29, 143
-o, 63, 93, 110, 125, 142, 146, 156
-p, 22, 31, 42, 66, 102, 121, 148
-q, 28, 86, 120, 143
-r, 35, 61, 90
-s, 25, 32, 51, 53, 117, 141, 148
-t, 36, 72, 120, 134, 141
-u, 127, 156
-v, 21, 25, 28, 40, 49, 50, 53, 56, 62, 77, 82, 86, 92, 99, 105, 109, 112, 116, 119, 124, 131, 135, 143, 146, 155
-w, 143, 157
-x, 144
-y, 144
-z, 95, 122, 149
{, 106
{}, 113
``m``, 107
|, 106, 113

beginning, 143
profiles, 106
commands_apply() (*RNA.fold_compound method*), 983
compare() (*RNA.move method*), 1054
compare_structure() (*in module RNA*), 939
compute_bpp (*RNA.md attribute*), 1046
compute_bpp (*RNA.md property*), 1050
compute_probabilities (*C function*), 335
ConcEnt (*C type*), 491
config (*RNA.plot_options_puzzler attribute*), 1077, 1078
consens_mis() (*in module RNA*), 939
ConstCharVector (*class in RNA*), 909
constrain (*C struct*), 753
constrain (*C type*), 749
constrain.indx (*C var*), 753
constrain.ptype (*C var*), 753
constraints_add() (*RNA.fold_compound method*), 983
convert_parameter_file (*C function*), 406
COORDINATE (*class in RNA*), 908
CoordinateVector (*class in RNA*), 910
copy() (*RNA.SwigPyIterator method*), 927
copy_parameters (*C function*), 384
copy_pf_param (*C function*), 384
cpair (*C type*), 748
cut_point (*C var*), 490
cutpoint (*RNA.fold_compound attribute*), 977
cv_fact (*C var*), 434
cv_fact (*RNA.md attribute*), 1047
cv_fact (*RNA.md property*), 1050

D

dangle3 (*RNA.param attribute*), 1057, 1060
dangle3 (*RNA.param property*), 1064
dangle5 (*RNA.param attribute*), 1057, 1060
dangle5 (*RNA.param property*), 1064
dangles (*C var*), 432
dangles (*RNA.md attribute*), 1045
dangles (*RNA.md property*), 1050
data (*RNA.hc attribute*), 1036, 1038
db_compat_remove() (*in module RNA*), 939
db_flatten() (*in module RNA*), 939
db_from_plist() (*in module RNA*), 940
db_from_probs() (*RNA.fold_compound method*), 984
db_from_ptable() (*in module RNA*), 940
db_from_WUSS() (*in module RNA*), 939
db_pack() (*in module RNA*), 940
db_pk_remove() (*in module RNA*), 941
db_to_element_string() (*in module RNA*), 941
db_to_tree_string() (*in module RNA*), 942
db_unpack() (*in module RNA*), 942
ddG (*RNA.duplexT attribute*), 944
ddG (*RNA.duplexT property*), 944
decr() (*RNA.SwigPyIterator method*), 928
delete_doubleP() (*in module RNA*), 943
delete_floatP() (*in module RNA*), 943
delete_intP() (*in module RNA*), 943
delete_shortP() (*in module RNA*), 943
delete_ushortP() (*in module RNA*), 943
density_of_states (*C var*), 512
depot (*RNA.hc attribute*), 1036, 1038
deref_any() (*in module RNA*), 943
destroy_TwoDfold_variables (*C function*), 500
dG1 (*RNA.duplexT attribute*), 944
dG1 (*RNA.duplexT property*), 944
dG2 (*RNA.duplexT attribute*), 944
dG2 (*RNA.duplexT property*), 944
dist_mountain() (*in module RNA*), 943
distance() (*RNA.SwigPyIterator method*), 928
do_backtrack (*C var*), 433
domains_struc (*RNA.fold_compound attribute*), 979
domains_up (*RNA.fold_compound attribute*), 979
doubleArray (*class in RNA*), 943
DoubleDoubleVector (*class in RNA*), 910
doubleP_getitem() (*in module RNA*), 943
doubleP_setitem() (*in module RNA*), 943
DoublePair (*class in RNA*), 911
DoubleVector (*class in RNA*), 911
drawArcs (*RNA.plot_options_puzzler attribute*), 1077, 1078
duplex_list_t (*class in RNA*), 945
duplex_subopt() (*in module RNA*), 945
duplexfold() (*in module RNA*), 945
DuplexInit (*RNA.param attribute*), 1058, 1061
DuplexInit (*RNA.param property*), 1063
duplexT (*C struct*), 753
duplexT (*class in RNA*), 943
duplexT.ddG (*C var*), 754
duplexT.dG1 (*C var*), 754
duplexT.dG2 (*C var*), 754
duplexT.end (*C var*), 754
duplexT.energy (*C var*), 754
duplexT.energy_backtrack (*C var*), 754
duplexT.i (*C var*), 754
duplexT.j (*C var*), 754
duplexT.offset (*C var*), 754
duplexT.opening_backtrack_x (*C var*), 754
duplexT.opening_backtrack_y (*C var*), 754
duplexT.qb (*C var*), 755
duplexT.qe (*C var*), 755
duplexT.structure (*C var*), 754
duplexT.tb (*C var*), 755
duplexT.te (*C var*), 755
DuplexVector (*class in RNA*), 912
dupVar (*C struct*), 757
dupVar (*C type*), 749
dupVar.ddG (*C var*), 758
dupVar.dG1 (*C var*), 757
dupVar.dG2 (*C var*), 757
dupVar.end (*C var*), 757
dupVar.energy (*C var*), 757
dupVar.i (*C var*), 757
dupVar.inactive (*C var*), 758
dupVar.j (*C var*), 757

dupVar.offset (*C var*), 757
dupVar.pk_helix (*C var*), 757
dupVar.processed (*C var*), 758
dupVar.qb (*C var*), 758
dupVar.qe (*C var*), 758
dupVar.structure (*C var*), 757
dupVar.tb (*C var*), 758
dupVar.te (*C var*), 758

E

E_ext_hp_loop() (*RNA.fold_compound method*), 980
E_ext_int_loop() (*RNA.fold_compound method*), 980
E_ExtLoop (*C function*), 477
E_ExtLoop() (*in module RNA*), 913
E_gquad() (*in module RNA*), 916
E_gquad_ali_en() (*in module RNA*), 916
E_GQuad_IntLoop_L() (*in module RNA*), 913
E_GQuad_IntLoop_L_comparative() (*in module RNA*), 913
E_Hairpin (*C function*), 478
E_Hairpin() (*in module RNA*), 913
E_hp_loop() (*RNA.fold_compound method*), 980
E_int_loop() (*RNA.fold_compound method*), 980
E_IntLoop (*C function*), 480
E_IntLoop() (*in module RNA*), 914
E_IntLoop_Co (*C function*), 482
E_IntLoop_Co() (*in module RNA*), 915
E_ml_rightmost_stem() (*in module RNA*), 916
E_MLstem (*C function*), 483
E_MLstem() (*in module RNA*), 915
E_stack() (*RNA.fold_compound method*), 980
E_Stem (*C function*), 476
E_Stem() (*in module RNA*), 915
ElemProbVector (*class in RNA*), 916
empty() (*RNA.ConstCharVector method*), 909
empty() (*RNA.CoordinateVector method*), 910
empty() (*RNA.DoubleDoubleVector method*), 911
empty() (*RNA.DoubleVector method*), 912
empty() (*RNA.DuplexVector method*), 912
empty() (*RNA.ElemProbVector method*), 917
empty() (*RNA.HeatCapacityVector method*), 917
empty() (*RNA.HelixVector method*), 918
empty() (*RNA.IntIntVector method*), 919
empty() (*RNA.IntVector method*), 920
empty() (*RNA.MoveVector method*), 923
empty() (*RNA.PathVector method*), 924
empty() (*RNA.SOLUTIONVector method*), 925
empty() (*RNA.StringVector method*), 926
empty() (*RNA.SuboptVector method*), 927
empty() (*RNA.UIntUIntVector method*), 928
empty() (*RNA.UIntVector method*), 929
en (*RNA.path property*), 1070
encode_seq() (*in module RNA*), 945
end (*RNA.duplexT attribute*), 943
end (*RNA.duplexT property*), 944
end (*RNA.hx property*), 1039
end() (*RNA.ConstCharVector method*), 909
end() (*RNA.CoordinateVector method*), 910
end() (*RNA.DoubleDoubleVector method*), 911
end() (*RNA.DoubleVector method*), 912
end() (*RNA.DuplexVector method*), 912
end() (*RNA.ElemProbVector method*), 917
end() (*RNA.HeatCapacityVector method*), 917
end() (*RNA.HelixVector method*), 918
end() (*RNA.IntIntVector method*), 919
end() (*RNA.IntVector method*), 920
end() (*RNA.MoveVector method*), 923
end() (*RNA.PathVector method*), 924
end() (*RNA.SOLUTIONVector method*), 925
end() (*RNA.StringVector method*), 926
end() (*RNA.SuboptVector method*), 927
end() (*RNA.UIntUIntVector method*), 928
end() (*RNA.UIntVector method*), 929
energy (*RNA.duplex_list_t property*), 945
energy (*RNA.duplexT attribute*), 943
energy (*RNA.duplexT property*), 944
energy (*RNA.SOLUTION property*), 925
energy (*RNA.struct_en attribute*), 1089, 1090
energy (*RNA.struct_en property*), 1090
energy (*RNA.subopt_solution property*), 1090
energy_backtrack (*RNA.duplexT attribute*), 943
energy_backtrack (*RNA.duplexT property*), 944
energy_of_alistruct (*C function*), 476
energy_of_circ_struct (*C function*), 490
energy_of_circ_struct() (*in module RNA*), 945
energy_of_circ_struct_par (*C function*), 485
energy_of_circ_structure (*C function*), 484
energy_of_circ_structure() (*in module RNA*), 946
energy_of_gquad_struct_par (*C function*), 485
energy_of_gquad_structure (*C function*), 485
energy_of_gquad_structure() (*in module RNA*), 946
energy_of_move (*C function*), 487
energy_of_move() (*in module RNA*), 946
energy_of_move_pt (*C function*), 487
energy_of_move_pt() (*in module RNA*), 947
energy_of_struct (*C function*), 488
energy_of_struct() (*in module RNA*), 947
energy_of_struct_par (*C function*), 484
energy_of_struct_pt (*C function*), 489
energy_of_struct_pt() (*in module RNA*), 947
energy_of_struct_pt_par (*C function*), 486
energy_of_structure (*C function*), 483
energy_of_structure() (*in module RNA*), 948
energy_of_structure_pt (*C function*), 486
energy_of_structure_pt() (*in module RNA*), 949
energy_set (*C var*), 433
energy_set (*RNA.md attribute*), 1046
energy_set (*RNA.md property*), 1050
ensemble_defect() (*RNA.fold_compound method*), 984
enumerate_necklaces() (*in module RNA*), 949
eos_debug (*C var*), 490
ep (*class in RNA*), 950

eqm_probs_bp() (*in module RNA*), 951
eqm_probs_bp() (*RNA.fold_compound method*), 985
eqm_probs_hx() (*in module RNA*), 951
eqm_probs_hx() (*RNA.fold_compound method*), 985
eqm_probs_opt (*class in RNA*), 951
eqm_probs_up() (*in module RNA*), 952
eqm_probs_up() (*RNA.fold_compound method*), 985
equal() (*RNA.SwigPyIterator method*), 928
equilibrium_conc() (*in module RNA*), 952
equilibrium_constants() (*in module RNA*), 952
erase() (*RNA.ConstCharVector method*), 909
erase() (*RNA.CoordinateVector method*), 910
erase() (*RNA.DoubleDoubleVector method*), 911
erase() (*RNA.DoubleVector method*), 912
erase() (*RNA.DuplexVector method*), 912
erase() (*RNA.ElemProbVector method*), 917
erase() (*RNA.HeatCapacityVector method*), 917
erase() (*RNA.HelixVector method*), 918
erase() (*RNA.IntIntVector method*), 919
erase() (*RNA.IntVector method*), 920
erase() (*RNA.MoveVector method*), 923
erase() (*RNA.PathVector method*), 924
erase() (*RNA.SOLUTIONVector method*), 925
erase() (*RNA.StringVector method*), 926
erase() (*RNA.SuboptVector method*), 927
erase() (*RNA.UIntUIntVector method*), 928
erase() (*RNA.UIntVector method*), 929
eval_circ_gquad_structure() (*in module RNA*), 952
eval_circ_structure() (*in module RNA*), 952
eval_covar_structure() (*RNA.fold_compound method*), 985
eval_ext (*RNA.hc attribute*), 1036, 1038
eval_ext_hp_loop() (*RNA.fold_compound method*), 985
eval_ext_stem() (*RNA.fold_compound method*), 985
eval_gquad_structure() (*in module RNA*), 953
eval_hp (*RNA.hc attribute*), 1036, 1038
eval_hp_loop() (*RNA.fold_compound method*), 985
eval_int (*RNA.hc attribute*), 1036, 1038
eval_int_loop() (*RNA.fold_compound method*), 985
eval_loop_pt() (*RNA.fold_compound method*), 986
eval_mb (*RNA.hc attribute*), 1036, 1038
eval_move() (*RNA.fold_compound method*), 986
eval_move_pt() (*RNA.fold_compound method*), 986
eval_structure() (*RNA.fold_compound method*), 987
eval_structure_pt() (*RNA.fold_compound method*), 988
eval_structure_pt_simple() (*in module RNA*), 954
eval_structure_pt_verbose() (*RNA.fold_compound method*), 988
eval_structure_simple() (*in module RNA*), 954
eval_structure_verbose() (*RNA.fold_compound method*), 989
exp_E_ext_stem() (*RNA.fold_compound method*), 989
exp_E_ExtLoop (*C function*), 478
exp_E_ExtLoop() (*in module RNA*), 955
exp_E_gquad() (*in module RNA*), 957
exp_E_gquad_ali() (*in module RNA*), 957
exp_E_Hairpin (*C function*), 480
exp_E_Hairpin() (*in module RNA*), 955
exp_E_hp_loop() (*RNA.fold_compound method*), 989
exp_E_int_loop() (*RNA.fold_compound method*), 989
exp_E_interior_loop() (*RNA.fold_compound method*), 989
exp_E_IntLoop (*C function*), 482
exp_E_IntLoop() (*in module RNA*), 956
exp_E_MLstem (*C function*), 483
exp_E_MLstem() (*in module RNA*), 957
exp_E_Stem (*C function*), 478
exp_E_Stem() (*in module RNA*), 957
exp_matrices (*RNA.fold_compound attribute*), 978
exp_matrices (*RNA.fold_compound property*), 989
exp_param (*class in RNA*), 957
exp_params (*RNA.fold_compound attribute*), 978
exp_params (*RNA.fold_compound property*), 989
exp_params_rescale() (*RNA.fold_compound method*), 989
exp_params_reset() (*RNA.fold_compound method*), 990
exp_params_subst() (*RNA.fold_compound method*), 991
expand_Full() (*in module RNA*), 967
expand_Shapiro() (*in module RNA*), 967
expbulge (*RNA.exp_param attribute*), 958, 962
expbulge (*RNA.exp_param property*), 966
expdangle3 (*RNA.exp_param attribute*), 958, 962
expdangle3 (*RNA.exp_param property*), 966
expdangle5 (*RNA.exp_param attribute*), 958, 962
expdangle5 (*RNA.exp_param property*), 966
expDuplexInit (*RNA.exp_param attribute*), 959, 963
expDuplexInit (*RNA.exp_param property*), 965
expgquad (*RNA.exp_param attribute*), 960, 964
expgquad (*RNA.exp_param property*), 966
expgquadLayerMismatch (*RNA.exp_param attribute*), 960, 964
expgquadLayerMismatch (*RNA.exp_param property*), 966
exphairpin (*RNA.exp_param attribute*), 958, 961
exphairpin (*RNA.exp_param property*), 966
exphex (*RNA.exp_param attribute*), 959, 963
exphex (*RNA.exp_param property*), 966
expint11 (*RNA.exp_param attribute*), 958, 962
expint11 (*RNA.exp_param property*), 966
expint21 (*RNA.exp_param attribute*), 958, 962
expint21 (*RNA.exp_param property*), 966
expint22 (*RNA.exp_param attribute*), 959, 962
expint22 (*RNA.exp_param property*), 966
expinternal (*RNA.exp_param attribute*), 958, 962
expinternal (*RNA.exp_param property*), 966
expmismatch1nI (*RNA.exp_param attribute*), 958, 962

expmismatch1nI (*RNA.exp_param property*), 966
expmismatch23I (*RNA.exp_param attribute*), 958, 962
expmismatch23I (*RNA.exp_param property*), 966
expmismatchExt (*RNA.exp_param attribute*), 958, 962
expmismatchExt (*RNA.exp_param property*), 966
expmismatchH (*RNA.exp_param attribute*), 958, 962
expmismatchH (*RNA.exp_param property*), 966
expmismatchI (*RNA.exp_param attribute*), 958, 962
expmismatchI (*RNA.exp_param property*), 966
expmismatchM (*RNA.exp_param attribute*), 958, 962
expmismatchM (*RNA.exp_param property*), 966
expMLbase (*RNA.exp_param attribute*), 959, 963
expMLbase (*RNA.exp_param property*), 965
expMLbase (*RNA.mx_pf property*), 1055
expMLclosing (*RNA.exp_param attribute*), 959, 963
expMLclosing (*RNA.exp_param property*), 966
expMLintern (*RNA.exp_param attribute*), 959, 963
expMLintern (*RNA.exp_param property*), 966
expMultipleCA (*RNA.exp_param attribute*), 960, 964
expMultipleCA (*RNA.exp_param property*), 966
expMultipleCB (*RNA.exp_param attribute*), 960, 964
expMultipleCB (*RNA.exp_param property*), 966
expninio (*RNA.exp_param attribute*), 959, 963
expninio (*RNA.exp_param property*), 966
export_ali_bppm (*C function*), 332
export_bppm (*C function*), 340
export_bppm() (*in module RNA*), 967
export_circfold_arrays (*C function*), 304
export_circfold_arrays_par (*C function*), 304
export_co_bppm (*C function*), 336
export_cofold_arrays (*C function*), 300
export_cofold_arrays_gq (*C function*), 300
export_fold_arrays (*C function*), 304
export_fold_arrays_par (*C function*), 304
expSaltLoop (*RNA.exp_param attribute*), 961, 965
expSaltLoop (*RNA.exp_param property*), 966
expSaltStack (*RNA.exp_param attribute*), 961, 965
expSaltStack (*RNA.exp_param property*), 966
expstack (*RNA.exp_param attribute*), 958, 961
expstack (*RNA.exp_param property*), 966
expTermAU (*RNA.exp_param attribute*), 959, 963
expTermAU (*RNA.exp_param property*), 966
exptetra (*RNA.exp_param attribute*), 959, 963
exptetra (*RNA.exp_param property*), 966
exptri (*RNA.exp_param attribute*), 959, 963
exptri (*RNA.exp_param property*), 966
expTriloop (*RNA.exp_param attribute*), 959, 963
expTriloop (*RNA.exp_param property*), 966
expTripleC (*RNA.exp_param attribute*), 960, 964
expTripleC (*RNA.exp_param property*), 966
extract_record_rest_structure (*C function*), 768
extract_record_rest_structure() (*in module RNA*), 967

F

f (*RNA.hc attribute*), 1035, 1037
F1 (*RNA.score property*), 1086
f3 (*RNA.mx_mfe property*), 1055
f5 (*RNA.mx_mfe property*), 1055
F_monomer (*C var*), 493
Fc (*RNA.mx_mfe property*), 1055
fc_add_pycallback() (*in module RNA*), 967
fc_add_pydata() (*in module RNA*), 967
FcH (*RNA.mx_mfe property*), 1055
FcI (*RNA.mx_mfe property*), 1055
FcM (*RNA.mx_mfe property*), 1055
FDR (*RNA.score property*), 1087
file_commands_apply() (*RNA.fold_compound method*), 991
file_commands_read() (*in module RNA*), 968
file_connect_read_record() (*in module RNA*), 969
file_fasta_read() (*in module RNA*), 969
file_msa_detect_format() (*in module RNA*), 970
file_msa_read() (*in module RNA*), 971
file_msa_read_record() (*in module RNA*), 972
file_msa_write() (*in module RNA*), 973
file_PS_aln() (*in module RNA*), 967
file_PS_rnaplot() (*in module RNA*), 968
file_PS_rnaplot_a() (*in module RNA*), 968
file_RNAstrand_db_read_record() (*in module RNA*), 968
file_SHAPE_read() (*in module RNA*), 968
filename (*RNA.plot_options_puzzler attribute*), 1077, 1079
FILENAME_ID_LENGTH (*C macro*), 797
FILENAME_MAX_LENGTH (*C macro*), 797
filename_sanitize() (*in module RNA*), 974
final_cost (*C var*), 496
find_saddle() (*in module RNA*), 975
first (*RNA.DoublePair property*), 911
floatArray (*class in RNA*), 975
floatP_getitem() (*in module RNA*), 975
floatP_setitem() (*in module RNA*), 975
FLT_OR_DBL (*C type*), 748
fM1 (*RNA.mx_mfe property*), 1055
fM2 (*RNA.mx_mfe property*), 1055
fML (*RNA.mx_mfe property*), 1055
FN (*RNA.score property*), 1087
FNR (*RNA.score property*), 1087
fold (*C function*), 302
fold() (*in module RNA*), 975
fold_compound (*class in RNA*), 976
fold_par (*C function*), 301
folden (*C type*), 749
FOR (*RNA.score property*), 1087
FP (*RNA.score property*), 1087
FPR (*RNA.score property*), 1087
free_alifold_arrays (*C function*), 298
free_alifold_arrays() (*in module RNA*), 1030
free_alipf_arrays (*C function*), 333
free_arrays (*C function*), 303
free_arrays() (*in module RNA*), 1030
free_auxdata (*RNA.fold_compound attribute*), 979

free_co_arrays (*C function*), 299
free_co_arrays() (*in module RNA*), 1030
free_co_pf_arrays (*C function*), 336
free_co_pf_arrays() (*in module RNA*), 1031
free_data (*RNA.hc attribute*), 1036, 1038
free_interact (*C function*), 495
free_path() (*in module RNA*), 1031
free_pf_arrays (*C function*), 339
free_pf_arrays() (*in module RNA*), 1031
free_profile() (*in module RNA*), 1031
free_pu_contrib (*C function*), 495
free_pu_contrib_struct (*C function*), 495
free_tree() (*in module RNA*), 1031
frompointer() (*RNA.doubleArray static method*), 943
frompointer() (*RNA.floatArray static method*), 975
frompointer() (*RNA.intArray static method*), 1040
front() (*RNA.ConstCharVector method*), 909
front() (*RNA.CoordinateVector method*), 910
front() (*RNA.DoubleDoubleVector method*), 911
front() (*RNA.DoubleVector method*), 912
front() (*RNA.DuplexVector method*), 912
front() (*RNA.ElemProbVector method*), 917
front() (*RNA.HeatCapacityVector method*), 917
front() (*RNA.HelixVector method*), 918
front() (*RNA.IntIntVector method*), 919
front() (*RNA.IntVector method*), 920
front() (*RNA.MoveVector method*), 923
front() (*RNA.PathVector method*), 924
front() (*RNA.SOLUTIONVector method*), 925
front() (*RNA.StringVector method*), 926
front() (*RNA.SuboptVector method*), 927
front() (*RNA.UIntUIntVector method*), 928
front() (*RNA.UIntVector method*), 929

G

get() (*RNA.COORDINATE method*), 909
get() (*RNA.SOLUTION method*), 925
get() (*RNA.varArrayChar method*), 1093
get() (*RNA.varArrayFLTorDBL method*), 1093
get() (*RNA.varArrayInt method*), 1093
get() (*RNA.varArrayMove method*), 1093
get() (*RNA.varArrayShort method*), 1094
get() (*RNA.varArrayUChar method*), 1094
get() (*RNA.varArrayUInt method*), 1094
get_aligned_line() (*in module RNA*), 1031
get_alipf_arrays (*C function*), 333
get_allocator() (*RNA.ConstCharVector method*), 909
get_allocator() (*RNA.CoordinateVector method*), 910
get_allocator() (*RNA.DoubleDoubleVector method*), 911
get_allocator() (*RNA.DoubleVector method*), 912
get_allocator() (*RNA.DuplexVector method*), 912
get_allocator() (*RNA.ElemProbVector method*), 917
get_allocator() (*RNA.HeatCapacityVector method*), 917
get_allocator() (*RNA.HelixVector method*), 918
get_allocator() (*RNA.IntIntVector method*), 919
get_allocator() (*RNA.IntVector method*), 920
get_allocator() (*RNA.MoveVector method*), 923
get_allocator() (*RNA.PathVector method*), 925
get_allocator() (*RNA.SOLUTIONVector method*), 925
get_allocator() (*RNA.StringVector method*), 926
get_allocator() (*RNA.SuboptVector method*), 927
get_allocator() (*RNA.UIntUIntVector method*), 928
get_allocator() (*RNA.UIntVector method*), 929
get_boltzmann_factor_copy (*C function*), 383
get_boltzmann_factors (*C function*), 382
get_boltzmann_factors_ali (*C function*), 383
get_centroid_struct_pl() (*in module RNA*), 1031
get_centroid_struct_pr() (*in module RNA*), 1031
get_concentrations() (*in module RNA*), 1032
get_gquad_ali_matrix() (*in module RNA*), 1032
get_gquad_count (*C function*), 574
get_gquad_L_matrix (*C function*), 574
get_gquad_L_matrix() (*in module RNA*), 1032
get_gquad_matrix() (*in module RNA*), 1032
get_gquad_pattern_exhaustive (*C function*), 574
get_gquad_pattern_pf (*C function*), 575
get_gquad_pattern_pf() (*in module RNA*), 1032
get_gquad_pf_matrix() (*in module RNA*), 1032
get_gquad_pf_matrix_comparative() (*in module RNA*), 1032
get_input_line (*C function*), 645
get_multi_input_line (*C function*), 768
get_multi_input_line() (*in module RNA*), 1032
get_parameter_copy (*C function*), 382
get_path() (*in module RNA*), 1032
get_pf_arrays (*C function*), 341
get_plist_gquad_from_db (*C function*), 576
get_plist_gquad_from_db() (*in module RNA*), 1032
get_plist_gquad_from_pr (*C function*), 576
get_plist_gquad_from_pr() (*in module RNA*), 1032
get_plist_gquad_from_pr_max (*C function*), 576
get_plist_gquad_from_pr_max() (*in module RNA*), 1032
get_pr() (*in module RNA*), 1032
get_pu_contrib_struct (*C function*), 495
get_ribosum (*C function*), 384
get_scaled_alipf_parameters (*C function*), 383
get_scaled_parameters (*C function*), 384
get_scaled_pf_parameters (*C function*), 382
get_subseq_F (*C function*), 341
get_TwoDfold_variables (*C function*), 499
get_xy_coordinates() (*in module RNA*), 1032
gettype (*C function*), 403
gettype() (*in module RNA*), 1033
give_up (*C var*), 496

gmlRNA (*C function*), 889
gmlRNA() (*in module RNA*), 1033
gq_parse() (*in module RNA*), 1033
gquad (*C var*), 433
gquad (*RNA.md attribute*), 1046
gquad (*RNA.md property*), 1050
gquad (*RNA.param attribute*), 1059, 1062
gquad (*RNA.param property*), 1064
gquadLayerMismatch (*RNA.param attribute*), 1059, 1062
gquadLayerMismatch (*RNA.param property*), 1064
gquadLayerMismatchMax (*RNA.exp_param attribute*), 960, 964
gquadLayerMismatchMax (*RNA.exp_param property*), 966
gquadLayerMismatchMax (*RNA.param attribute*), 1059, 1062
gquadLayerMismatchMax (*RNA.param property*), 1064

H

hairpin (*RNA.param attribute*), 1056, 1060
hairpin (*RNA.param property*), 1064
HairpinE (*C function*), 305
hamming() (*in module RNA*), 1033
hamming_bound() (*in module RNA*), 1034
hamming_distance() (*in module RNA*), 1034
hamming_distance_bound() (*in module RNA*), 1034
hc (*class in RNA*), 1034
hc (*RNA.fold_compound attribute*), 978
hc (*RNA.fold_compound property*), 991
hc_add_bp() (*RNA.fold_compound method*), 991
hc_add_bp_nonspecific() (*RNA.fold_compound method*), 991
hc_add_bp_strand() (*RNA.fold_compound method*), 992
hc_add_from_db() (*RNA.fold_compound method*), 992
hc_add_up() (*RNA.fold_compound method*), 992
hc_add_up_strand() (*RNA.fold_compound method*), 993
hc_init() (*RNA.fold_compound method*), 993
heat_capacity (*RNA.heat_capacity_result property*), 1039
heat_capacity() (*in module RNA*), 1038
heat_capacity() (*RNA.fold_compound method*), 993
heat_capacity_cb() (*RNA.fold_compound method*), 994
heat_capacity_result (*class in RNA*), 1039
HeatCapacityVector (*class in RNA*), 917
helical_rise (*C var*), 435
helical_rise (*RNA.md attribute*), 1048
helical_rise (*RNA.md property*), 1050
HelixVector (*class in RNA*), 918
Hexaloop_E (*RNA.param attribute*), 1058, 1062
Hexaloop_E (*RNA.param property*), 1063
Hexaloops (*RNA.exp_param attribute*), 960, 964
Hexaloops (*RNA.exp_param property*), 965
Hexaloops (*RNA.param attribute*), 1058, 1062
Hexaloops (*RNA.param property*), 1063
hx (*class in RNA*), 1039
hx_from_ptable() (*in module RNA*), 1040
hx_length (*RNA.eqm_probs_opt attribute*), 951, 952
hx_length (*RNA.eqm_probs_opt property*), 952
hx_merge() (*in module RNA*), 1040

I

i (*RNA.basepair attribute*), 934, 935
i (*RNA.basepair property*), 935
i (*RNA.duplex_list_t property*), 945
i (*RNA.duplexT attribute*), 943
i (*RNA.duplexT property*), 944
i (*RNA.ep attribute*), 950
i (*RNA.ep property*), 951
id (*RNA.exp_param attribute*), 957, 961
id (*RNA.exp_param property*), 966
id (*RNA.param attribute*), 1056, 1060
id (*RNA.param property*), 1064
iindx (*RNA.fold_compound attribute*), 978
iindx (*RNA.fold_compound property*), 995
incr() (*RNA.SwigPyIterator method*), 928
init_co_pf_fold (*C function*), 336
init_pf_circ_fold() (*in module RNA*), 1040
init_pf_fold (*C function*), 342
init_pf_fold() (*in module RNA*), 1040
init_rand() (*in module RNA*), 1040
initialize_cofold (*C function*), 301
initialize_cofold() (*in module RNA*), 1040
initialize_fold (*C function*), 305
insert() (*RNA.ConstCharVector method*), 909
insert() (*RNA.CoordinateVector method*), 910
insert() (*RNA.DoubleDoubleVector method*), 911
insert() (*RNA.DoubleVector method*), 912
insert() (*RNA.DuplexVector method*), 912
insert() (*RNA.ElemProbVector method*), 917
insert() (*RNA.HeatCapacityVector method*), 917
insert() (*RNA.HelixVector method*), 918
insert() (*RNA.IntIntVector method*), 919
insert() (*RNA.IntVector method*), 920
insert() (*RNA.MoveVector method*), 923
insert() (*RNA.PathVector method*), 925
insert() (*RNA.SOLUTIONVector method*), 926
insert() (*RNA.StringVector method*), 926
insert() (*RNA.SuboptVector method*), 927
insert() (*RNA.UIntUIntVector method*), 928
insert() (*RNA.UIntVector method*), 929
int11 (*RNA.param attribute*), 1057, 1061
int11 (*RNA.param property*), 1064
int21 (*RNA.param attribute*), 1057, 1061
int21 (*RNA.param property*), 1064
int22 (*RNA.param attribute*), 1057, 1061
int22 (*RNA.param property*), 1064
int_urn() (*in module RNA*), 1041
intArray (*class in RNA*), 1040
interact (*C struct*), 752
interact (*C type*), 749

interact.Gi (*C var*), 752
interact.Gikjl (*C var*), 752
interact.Gikjl_wo (*C var*), 752
interact.i (*C var*), 752
interact.j (*C var*), 752
interact.k (*C var*), 752
interact.l (*C var*), 752
interact.length (*C var*), 752
interact.Pi (*C var*), 752
internal_loop (*RNA.param attribute*), 1056, 1060
internal_loop (*RNA.param property*), 1064
IntIntVector (*class in RNA*), 919
intP_getitem() (*in module RNA*), 1041
intP_setitem() (*in module RNA*), 1041
IntVector (*class in RNA*), 919
inv_verbose (*C var*), 496
inverse_fold (*C function*), 495
inverse_fold() (*in module RNA*), 1041
inverse_pf_fold (*C function*), 496
inverse_pf_fold() (*in module RNA*), 1041
is_insertion() (*RNA.move method*), 1054
is_removal() (*RNA.move method*), 1054
is_shift() (*RNA.move method*), 1054
iterator() (*RNA.ConstCharVector method*), 909
iterator() (*RNA.CoordinateVector method*), 910
iterator() (*RNA.DoubleDoubleVector method*), 911
iterator() (*RNA.DoubleVector method*), 912
iterator() (*RNA.DuplexVector method*), 912
iterator() (*RNA.ElemProbVector method*), 917
iterator() (*RNA.HeatCapacityVector method*), 918
iterator() (*RNA.HelixVector method*), 918
iterator() (*RNA.IntIntVector method*), 919
iterator() (*RNA.IntVector method*), 920
iterator() (*RNA.MoveVector method*), 923
iterator() (*RNA.PathVector method*), 925
iterator() (*RNA.SOLUTIONVector method*), 926
iterator() (*RNA.StringVector method*), 926
iterator() (*RNA.SuboptVector method*), 927
iterator() (*RNA.UIntUIntVector method*), 928
iterator() (*RNA.UIntVector method*), 929

J

j (*RNA.basepair attribute*), 935
j (*RNA.basepair property*), 935
j (*RNA.duplex_list_t property*), 945
j (*RNA.duplexT attribute*), 943
j (*RNA.duplexT property*), 944
j (*RNA.ep attribute*), 950
j (*RNA.ep property*), 951
jindx (*RNA.fold_compound attribute*), 978
jindx (*RNA.fold_compound property*), 995

K

kT (*RNA.exp_param attribute*), 960, 964
kT (*RNA.exp_param property*), 966

L

last_parameter_file (*C function*), 403
last_parameter_file() (*in module RNA*), 1041
length (*RNA.fold_compound attribute*), 977
length (*RNA.fold_compound property*), 995
length (*RNA.hx property*), 1039
length (*RNA.mx_mfe property*), 1055
length (*RNA.mx_pf property*), 1055
Lfold (*C function*), 305
Lfold() (*in module RNA*), 920
Lfold_cb() (*in module RNA*), 921
Lfoldz() (*in module RNA*), 921
Lfoldz_cb() (*in module RNA*), 921
log_cb_add() (*in module RNA*), 1042
log_cb_add_pycallback() (*in module RNA*), 1042
log_cb_num() (*in module RNA*), 1042
log_fp() (*in module RNA*), 1042
log_fp_set() (*in module RNA*), 1042
log_level() (*in module RNA*), 1042
log_level_set() (*in module RNA*), 1042
log_options() (*in module RNA*), 1043
log_options_set() (*in module RNA*), 1043
log_reset() (*in module RNA*), 1043
logML (*C var*), 434
logML (*RNA.md attribute*), 1046
logML (*RNA.md property*), 1051
loop_energy (*C function*), 488
loop_energy() (*in module RNA*), 1043
LoopEnergy (*C function*), 304
loopidx_from_ptable() (*in module RNA*), 1044
lxc (*RNA.exp_param attribute*), 959, 963
lxc (*RNA.exp_param property*), 967
lxc (*RNA.param attribute*), 1057, 1061
lxc (*RNA.param property*), 1064

M

Make_bp_profile() (*in module RNA*), 922
Make_bp_profile_bppm() (*in module RNA*), 922
make_loop_index() (*in module RNA*), 1044
Make_swString() (*in module RNA*), 922
make_tree() (*in module RNA*), 1044
matrices (*RNA.fold_compound attribute*), 978
matrices (*RNA.fold_compound property*), 995
matrix_local (*RNA.hc attribute*), 1035, 1037
MAX2 (*C macro*), 643
MAX3 (*C macro*), 643
max_bp_span (*C var*), 434
max_bp_span (*RNA.md attribute*), 1047
max_bp_span (*RNA.md property*), 1051
MAXALPHA (*C macro*), 413
maxD1 (*RNA.fold_compound attribute*), 979
maxD2 (*RNA.fold_compound attribute*), 979
maximum_matching() (*in module RNA*), 1044
maximumNumberOfConfigChangesAllowed (*RNA.plot_options_puzzler attribute*), 1077, 1078
maxmimum_matching() (*RNA.fold_compound method*), 995
MCC (*RNA.score property*), 1087
md (*class in RNA*), 1044

md (*RNA.plot_data property*), 1074
MEA (*C function*), 373
MEA() (*RNA.fold_compound method*), 981
MEA_from_plist() (*in module RNA*), 921
mean_bp_dist() (*in module RNA*), 1052
mean_bp_distance (*C function*), 341
mean_bp_distance() (*in module RNA*), 1052
mean_bp_distance() (*RNA.fold_compound method*), 995
mean_bp_distance_pr (*C function*), 342
mean_bp_distance_pr() (*in module RNA*), 1052
memmove() (*in module RNA*), 1052
mfe() (*RNA.fold_compound method*), 995
mfe_dimer() (*RNA.fold_compound method*), 996
mfe_window() (*RNA.fold_compound method*), 996
mfe_window_cb() (*RNA.fold_compound method*), 997
mfe_window_zscore() (*RNA.fold_compound method*), 997
mfe_window_zscore_cb() (*RNA.fold_compound method*), 997
MIN2 (*C macro*), 643
MIN3 (*C macro*), 643
min_loop_size (*RNA.md attribute*), 1047
min_loop_size (*RNA.md property*), 1051
mirnatog (*C var*), 493
mismatch1nI (*RNA.param attribute*), 1057, 1060
mismatch1nI (*RNA.param property*), 1064
mismatch23I (*RNA.param attribute*), 1057, 1060
mismatch23I (*RNA.param property*), 1064
mismatchExt (*RNA.param attribute*), 1056, 1060
mismatchExt (*RNA.param property*), 1064
mismatchH (*RNA.param attribute*), 1057, 1060
mismatchH (*RNA.param property*), 1064
mismatchI (*RNA.param attribute*), 1057, 1060
mismatchI (*RNA.param property*), 1064
mismatchM (*RNA.param attribute*), 1057, 1060
mismatchM (*RNA.param property*), 1064
MLbase (*RNA.param attribute*), 1057, 1061
MLbase (*RNA.param property*), 1063
MLclosing (*RNA.param attribute*), 1058, 1061
MLclosing (*RNA.param property*), 1063
MLintern (*RNA.param attribute*), 1058, 1061
MLintern (*RNA.param property*), 1063
mm1 (*RNA.fold_compound attribute*), 980
mm2 (*RNA.fold_compound attribute*), 980
mode (*RNA.eqm_probs_opt attribute*), 951
mode (*RNA.eqm_probs_opt property*), 952
model_details (*RNA.exp_param attribute*), 961, 964
model_details (*RNA.exp_param property*), 967
model_details (*RNA.param attribute*), 1059, 1062
model_details (*RNA.param property*), 1064
model_detailsT (*C macro*), 413
module
    RNA, 908
move (*class in RNA*), 1052
move (*RNA.path property*), 1070
move_neighbor_diff() (*RNA.fold_compound method*), 997
move_standard() (*in module RNA*), 1055
MoveVector (*class in RNA*), 922
MultipleCA (*RNA.param attribute*), 1058, 1062
MultipleCA (*RNA.param property*), 1063
MultipleCB (*RNA.param attribute*), 1058, 1062
MultipleCB (*RNA.param property*), 1063
mx (*RNA.hc attribute*), 1035, 1037
mx (*RNA.hc property*), 1038
mx_mfe (*class in RNA*), 1055
mx_pf (*class in RNA*), 1055
my_aln_consensus_sequence2() (*in module RNA*), 1056
my_PS_rna_plot_snoop_a() (*in module RNA*), 1056

N

n (*RNA.hc attribute*), 1035, 1037
n (*RNA.hc property*), 1038
n_multichoose_k() (*in module RNA*), 1056
naview_xy_coordinates() (*in module RNA*), 1056
NBASES (*C macro*), 407
nc_fact (*C var*), 434
nc_fact (*RNA.md attribute*), 1047
nc_fact (*RNA.md property*), 1051
neighbors() (*RNA.fold_compound method*), 997
new_doubleP() (*in module RNA*), 1056
new_floatP() (*in module RNA*), 1056
new_intP() (*in module RNA*), 1056
new_shortP() (*in module RNA*), 1056
new_ushortP() (*in module RNA*), 1056
next (*RNA.move attribute*), 1053
next() (*RNA.var_array_Iterator method*), 1094
ninio (*RNA.param attribute*), 1057, 1061
ninio (*RNA.param property*), 1064
no_closingGU (*C var*), 433
node (*C struct*), 755
node.energy (*C var*), 755
node.k (*C var*), 755
node.next (*C var*), 755
noGU (*C var*), 433
noGU (*RNA.md attribute*), 1045
noGU (*RNA.md property*), 1051
noGUclosure (*RNA.md attribute*), 1046
noGUclosure (*RNA.md property*), 1051
noLonelyPairs (*C var*), 432
noLP (*RNA.md attribute*), 1045
noLP (*RNA.md property*), 1051
nonstandards (*C var*), 434
nonstandards (*RNA.md attribute*), 1046
nonstandards (*RNA.md property*), 1051
NPV (*RNA.score property*), 1087
nucleotides (*RNA.fold_compound attribute*), 978
numberOfChangesAppliedToConfig (*RNA.plot_options_puzzler attribute*), 1078, 1079

O

offset (*RNA.duplexT attribute*), 944
offset (*RNA.duplexT property*), 944

oldAliEn (*C var*), 434
oldAliEn (*RNA.md attribute*), 1047
oldAliEn (*RNA.md property*), 1051
ON_SAME_STRAND (*C macro*), 476
opening_backtrack_x (*RNA.duplexT attribute*), 943
opening_backtrack_x (*RNA.duplexT property*), 944
opening_backtrack_y (*RNA.duplexT attribute*), 944
opening_backtrack_y (*RNA.duplexT property*), 945
optimize (*RNA.plot_options_puzzler attribute*), 1077, 1078
optimize (*RNA.plot_options_puzzler property*), 1078
option_string (*C function*), 431
option_string() (*RNA.md method*), 1051
options (*RNA.plot_data property*), 1074

P

p (*RNA.ep attribute*), 950, 951
p (*RNA.ep property*), 951
pack_structure() (*in module RNA*), 1056
PAIR (*C type*), 748
pair (*RNA.md attribute*), 1047
pair (*RNA.md property*), 1051
pair_dist (*RNA.md attribute*), 1048
paired (*RNA.plot_options_puzzler attribute*), 1077, 1078
param (*class in RNA*), 1056
param_file (*RNA.exp_param attribute*), 961, 965
param_file (*RNA.exp_param property*), 967
param_file (*RNA.param attribute*), 1059, 1062
param_file (*RNA.param property*), 1064
params (*RNA.fold_compound attribute*), 978
params (*RNA.fold_compound property*), 998
params_load() (*in module RNA*), 1064
params_load_DNA_Mathews1999() (*in module RNA*), 1065
params_load_DNA_Mathews2004() (*in module RNA*), 1066
params_load_from_string() (*in module RNA*), 1069
params_load_RNA_Andronescu2007() (*in module RNA*), 1066
params_load_RNA_Langdon2018() (*in module RNA*), 1067
params_load_RNA_misc_special_hairpins() (*in module RNA*), 1068
params_load_RNA_Turner1999() (*in module RNA*), 1067
params_load_RNA_Turner2004() (*in module RNA*), 1068
params_reset() (*RNA.fold_compound method*), 998
params_save() (*in module RNA*), 1069
params_subst() (*RNA.fold_compound method*), 999
paramT (*C type*), 376
parse_gquad() (*in module RNA*), 1070
parse_structure() (*in module RNA*), 1070
parset (*C enum*), 395
parset.B (*C enumerator*), 396
parset.B_H (*C enumerator*), 396
parset.D3 (*C enumerator*), 396
parset.D3_H (*C enumerator*), 397
parset.D5 (*C enumerator*), 396
parset.D5_H (*C enumerator*), 396
parset.HEX (*C enumerator*), 397
parset.HP (*C enumerator*), 396
parset.HP_H (*C enumerator*), 396
parset.IL (*C enumerator*), 396
parset.IL_H (*C enumerator*), 396
parset.INT11 (*C enumerator*), 397
parset.INT11_H (*C enumerator*), 397
parset.INT21 (*C enumerator*), 397
parset.INT21_H (*C enumerator*), 397
parset.INT22 (*C enumerator*), 397
parset.INT22_H (*C enumerator*), 397
parset.MISC (*C enumerator*), 397
parset.ML (*C enumerator*), 397
parset.MME (*C enumerator*), 396
parset.MME_H (*C enumerator*), 396
parset.MMH (*C enumerator*), 396
parset.MMH_H (*C enumerator*), 396
parset.MMI (*C enumerator*), 396
parset.MMI1N (*C enumerator*), 396
parset.MMI1N_H (*C enumerator*), 396
parset.MMI23 (*C enumerator*), 396
parset.MMI23_H (*C enumerator*), 396
parset.MMI_H (*C enumerator*), 396
parset.MMM (*C enumerator*), 396
parset.MMM_H (*C enumerator*), 396
parset.NIN (*C enumerator*), 397
parset.QUIT (*C enumerator*), 395
parset.S (*C enumerator*), 396
parset.S_H (*C enumerator*), 396
parset.TL (*C enumerator*), 397
parset.TRI (*C enumerator*), 397
parset.UNKNOWN (*C enumerator*), 395
path (*class in RNA*), 1070
path() (*RNA.fold_compound method*), 999
path_direct() (*RNA.fold_compound method*), 1000
path_findpath() (*RNA.fold_compound method*), 1001
path_findpath_saddle() (*RNA.fold_compound method*), 1001
path_gradient() (*RNA.fold_compound method*), 1001
path_options (*class in RNA*), 1070
path_options_findpath() (*in module RNA*), 1070
path_random() (*RNA.fold_compound method*), 1002
PathVector (*class in RNA*), 924
pbacktrack (*C function*), 1110
pbacktrack() (*in module RNA*), 1071
pbacktrack() (*RNA.fold_compound method*), 1003
pbacktrack5 (*C function*), 1110
pbacktrack5() (*in module RNA*), 1071
pbacktrack5() (*RNA.fold_compound method*), 1004
pbacktrack_circ (*C function*), 1110
pbacktrack_circ() (*in module RNA*), 1071
pbacktrack_mem (*class in RNA*), 1072

pbacktrack_sub() (*RNA.fold_compound method*), 1005
pf() (*RNA.fold_compound method*), 1006
pf_add() (*in module RNA*), 1072
pf_circ_fold (*C function*), 339
pf_circ_fold() (*in module RNA*), 1072
pf_dimer() (*RNA.fold_compound method*), 1007
pf_float_precision() (*in module RNA*), 1072
pf_fold (*C function*), 338
pf_fold() (*in module RNA*), 1072
pf_fold_par (*C function*), 337
pf_interact (*C function*), 494
pf_paramT (*C type*), 376
pf_scale (*C var*), 432
pf_scale (*RNA.exp_param attribute*), 960, 964
pf_scale (*RNA.exp_param property*), 967
pf_smooth (*RNA.md attribute*), 1045
pf_smooth (*RNA.md property*), 1051
pf_unstru (*C function*), 494
pfl_fold (*C function*), 344
pfl_fold() (*in module RNA*), 1072
pfl_fold_cb() (*in module RNA*), 1073
pfl_fold_par (*C function*), 344
pfl_fold_up() (*in module RNA*), 1073
pfl_fold_up_cb() (*in module RNA*), 1073
PhonyNameDueToError.a2s (*C var*), 674
PhonyNameDueToError.c (*C var*), 696
PhonyNameDueToError.c_gq (*C var*), 697
PhonyNameDueToError.c_local (*C var*), 697
PhonyNameDueToError.cons_seq (*C var*), 673
PhonyNameDueToError.E_C (*C var*), 699
PhonyNameDueToError.E_C_rem (*C var*), 702
PhonyNameDueToError.E_F3 (*C var*), 699
PhonyNameDueToError.E_F3_rem (*C var*), 702
PhonyNameDueToError.E_F5 (*C var*), 698
PhonyNameDueToError.E_F5_rem (*C var*), 702
PhonyNameDueToError.E_Fc (*C var*), 701
PhonyNameDueToError.E_Fc_rem (*C var*), 703
PhonyNameDueToError.E_FcH (*C var*), 701
PhonyNameDueToError.E_FcH_rem (*C var*), 703
PhonyNameDueToError.E_FcI (*C var*), 701
PhonyNameDueToError.E_FcI_rem (*C var*), 703
PhonyNameDueToError.E_FcM (*C var*), 702
PhonyNameDueToError.E_FcM_rem (*C var*), 703
PhonyNameDueToError.E_M (*C var*), 699
PhonyNameDueToError.E_M1 (*C var*), 700
PhonyNameDueToError.E_M1_rem (*C var*), 703
PhonyNameDueToError.E_M2 (*C var*), 700
PhonyNameDueToError.E_M2_rem (*C var*), 703
PhonyNameDueToError.E_M_rem (*C var*), 703
PhonyNameDueToError.encoding3 (*C var*), 671
PhonyNameDueToError.encoding5 (*C var*), 671
PhonyNameDueToError.energy_bp (*C var*), 602
PhonyNameDueToError.energy_bp_local (*C var*), 602
PhonyNameDueToError.exp_energy_bp (*C var*), 602
PhonyNameDueToError.exp_energy_bp_local (*C var*), 602
PhonyNameDueToError.f3 (*C var*), 696
PhonyNameDueToError.f3_local (*C var*), 698
PhonyNameDueToError.f5 (*C var*), 696
PhonyNameDueToError.Fc (*C var*), 697
PhonyNameDueToError.FcH (*C var*), 697
PhonyNameDueToError.FcI (*C var*), 697
PhonyNameDueToError.FcM (*C var*), 697
PhonyNameDueToError.fM1 (*C var*), 696
PhonyNameDueToError.fM1_new (*C var*), 696
PhonyNameDueToError.fM2 (*C var*), 696
PhonyNameDueToError.fM2_real (*C var*), 697
PhonyNameDueToError.fML (*C var*), 696
PhonyNameDueToError.fML_local (*C var*), 698
PhonyNameDueToError.fms3 (*C var*), 696
PhonyNameDueToError.fms5 (*C var*), 696
PhonyNameDueToError.G_local (*C var*), 722
PhonyNameDueToError.ggg_local (*C var*), 698
PhonyNameDueToError.ggg_local_shift (*C var*), 698
PhonyNameDueToError.k_max_C (*C var*), 699
PhonyNameDueToError.k_max_F3 (*C var*), 699
PhonyNameDueToError.k_max_F5 (*C var*), 698
PhonyNameDueToError.k_max_Fc (*C var*), 701
PhonyNameDueToError.k_max_FcH (*C var*), 701
PhonyNameDueToError.k_max_FcI (*C var*), 702
PhonyNameDueToError.k_max_FcM (*C var*), 702
PhonyNameDueToError.k_max_M (*C var*), 700
PhonyNameDueToError.k_max_M1 (*C var*), 700
PhonyNameDueToError.k_max_M2 (*C var*), 701
PhonyNameDueToError.k_max_Q (*C var*), 723
PhonyNameDueToError.k_max_Q_B (*C var*), 723
PhonyNameDueToError.k_max_Q_c (*C var*), 725
PhonyNameDueToError.k_max_Q_cH (*C var*), 725
PhonyNameDueToError.k_max_Q_cI (*C var*), 725
PhonyNameDueToError.k_max_Q_cM (*C var*), 726
PhonyNameDueToError.k_max_Q_M (*C var*), 723
PhonyNameDueToError.k_max_Q_M1 (*C var*), 724
PhonyNameDueToError.k_max_Q_M2 (*C var*), 724
PhonyNameDueToError.k_min_C (*C var*), 699
PhonyNameDueToError.k_min_F3 (*C var*), 699
PhonyNameDueToError.k_min_F5 (*C var*), 698
PhonyNameDueToError.k_min_Fc (*C var*), 701
PhonyNameDueToError.k_min_FcH (*C var*), 701
PhonyNameDueToError.k_min_FcI (*C var*), 702
PhonyNameDueToError.k_min_FcM (*C var*), 702
PhonyNameDueToError.k_min_M (*C var*), 700
PhonyNameDueToError.k_min_M1 (*C var*), 700
PhonyNameDueToError.k_min_M2 (*C var*), 700
PhonyNameDueToError.k_min_Q (*C var*), 722
PhonyNameDueToError.k_min_Q_B (*C var*), 723
PhonyNameDueToError.k_min_Q_c (*C var*), 725
PhonyNameDueToError.k_min_Q_cH (*C var*), 725
PhonyNameDueToError.k_min_Q_cI (*C var*), 725
PhonyNameDueToError.k_min_Q_cM (*C var*), 726
PhonyNameDueToError.k_min_Q_M (*C var*), 723
PhonyNameDueToError.k_min_Q_M1 (*C var*), 724

PhonyNameDueToError.k_min_Q_M2 (*C var*), 724
PhonyNameDueToError.l_max_C (*C var*), 699
PhonyNameDueToError.l_max_F3 (*C var*), 699
PhonyNameDueToError.l_max_F5 (*C var*), 698
PhonyNameDueToError.l_max_Fc (*C var*), 701
PhonyNameDueToError.l_max_FcH (*C var*), 701
PhonyNameDueToError.l_max_FcI (*C var*), 702
PhonyNameDueToError.l_max_FcM (*C var*), 702
PhonyNameDueToError.l_max_M (*C var*), 700
PhonyNameDueToError.l_max_M1 (*C var*), 700
PhonyNameDueToError.l_max_M2 (*C var*), 700
PhonyNameDueToError.l_max_Q (*C var*), 722
PhonyNameDueToError.l_max_Q_B (*C var*), 723
PhonyNameDueToError.l_max_Q_c (*C var*), 724
PhonyNameDueToError.l_max_Q_cH (*C var*), 725
PhonyNameDueToError.l_max_Q_cI (*C var*), 725
PhonyNameDueToError.l_max_Q_cM (*C var*), 726
PhonyNameDueToError.l_max_Q_M (*C var*), 723
PhonyNameDueToError.l_max_Q_M1 (*C var*), 724
PhonyNameDueToError.l_max_Q_M2 (*C var*), 724
PhonyNameDueToError.l_min_C (*C var*), 699
PhonyNameDueToError.l_min_F3 (*C var*), 699
PhonyNameDueToError.l_min_F5 (*C var*), 698
PhonyNameDueToError.l_min_Fc (*C var*), 701
PhonyNameDueToError.l_min_FcH (*C var*), 701
PhonyNameDueToError.l_min_FcI (*C var*), 702
PhonyNameDueToError.l_min_FcM (*C var*), 702
PhonyNameDueToError.l_min_M (*C var*), 699
PhonyNameDueToError.l_min_M1 (*C var*), 700
PhonyNameDueToError.l_min_M2 (*C var*), 700
PhonyNameDueToError.l_min_Q (*C var*), 722
PhonyNameDueToError.l_min_Q_B (*C var*), 723
PhonyNameDueToError.l_min_Q_c (*C var*), 724
PhonyNameDueToError.l_min_Q_cH (*C var*), 725
PhonyNameDueToError.l_min_Q_cI (*C var*), 725
PhonyNameDueToError.l_min_Q_cM (*C var*), 726
PhonyNameDueToError.l_min_Q_M (*C var*), 723
PhonyNameDueToError.l_min_Q_M1 (*C var*), 724
PhonyNameDueToError.l_min_Q_M2 (*C var*), 724
PhonyNameDueToError.n_seq (*C var*), 673
PhonyNameDueToError.oldAliEn (*C var*), 675
PhonyNameDueToError.p_gq (*C var*), 720
PhonyNameDueToError.PhonyNameDueToError.a2s (*C var*), 669
PhonyNameDueToError.PhonyNameDueToError.c (*C var*), 688
PhonyNameDueToError.PhonyNameDueToError.c_gq (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.c_local (*C var*), 690
PhonyNameDueToError.PhonyNameDueToError.cons_seq (*C var*), 668
PhonyNameDueToError.PhonyNameDueToError.E_C (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.E_C_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_F3 (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.E_F3_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_F5 (*C var*), 690
PhonyNameDueToError.PhonyNameDueToError.E_F5_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_Fc (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.E_Fc_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_FcH (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.E_FcH_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_FcI (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.E_FcI_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_FcM (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.E_FcM_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_M (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.E_M1 (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.E_M1_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_M2 (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.E_M2_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.E_M_rem (*C var*), 695
PhonyNameDueToError.PhonyNameDueToError.encoding3 (*C var*), 666
PhonyNameDueToError.PhonyNameDueToError.encoding5 (*C var*), 666
PhonyNameDueToError.PhonyNameDueToError.energy_bp (*C var*), 601
PhonyNameDueToError.PhonyNameDueToError.energy_bp_local (*C var*), 602
PhonyNameDueToError.PhonyNameDueToError.exp_energy_bp (*C var*), 602
PhonyNameDueToError.PhonyNameDueToError.exp_energy_bp_lo (*C var*), 602
PhonyNameDueToError.PhonyNameDueToError.f3 (*C var*), 688
PhonyNameDueToError.PhonyNameDueToError.f3_local (*C var*), 690
PhonyNameDueToError.PhonyNameDueToError.f5 (*C var*), 688
PhonyNameDueToError.PhonyNameDueToError.Fc (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.FcH (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.FcI (*C var*), 689

PhonyNameDueToError.PhonyNameDueToError.FcM (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.fM1 (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.fM1_rev (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.fM2 (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.fM2_rev (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.fML (*C var*), 689
PhonyNameDueToError.PhonyNameDueToError.fML_rev (*C var*), 690
PhonyNameDueToError.PhonyNameDueToError.fms3 (*C var*), 688
PhonyNameDueToError.PhonyNameDueToError.fms5 (*C var*), 688
PhonyNameDueToError.PhonyNameDueToError.G_lonely (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.ggg_lonely (*C var*), 690
PhonyNameDueToError.PhonyNameDueToError.ggg_lonely_shift (*C var*), 690
PhonyNameDueToError.PhonyNameDueToError.k_max_C (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.k_max_F3 (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.k_max_F5 (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.k_max_Fc (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.k_max_FcH (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.k_max_FcI (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.k_max_FcM (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.k_max_M (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.k_max_M1 (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.k_max_M2 (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.k_max_Q (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.k_max_Q_B (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.k_max_Q_c (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.k_max_Q_cH (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.k_max_Q_cI (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.k_max_Q_cM (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.k_max_Q_M (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.k_max_Q_M1 (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.k_max_Q_M2 (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.k_min_C (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.k_min_F3 (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.k_min_F5 (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.k_min_Fc (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.k_min_FcH (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.k_min_FcI (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.k_min_FcM (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.k_min_M (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.k_min_M1 (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.k_min_M2 (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.k_min_Q (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.k_min_Q_B (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.k_min_Q_c (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.k_min_Q_cH (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.k_min_Q_cI (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.k_min_Q_cM (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.k_min_Q_M (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.k_min_Q_M1 (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.k_min_Q_M2 (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.l_max_C (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.l_max_F3 (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.l_max_F5 (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.l_max_Fc (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.l_max_FcH (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.l_max_FcI (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.l_max_FcM (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.l_max_M (*C var*), 692

PhonyNameDueToError.PhonyNameDueToError.l_max_M1 (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.l_max_M2 (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.l_max_Q (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.l_max_Q_B (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.l_max_Q_c (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.l_max_Q_cH (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.l_max_Q_cI (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.l_max_Q_cM (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.l_max_q_gq (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.l_max_q_local (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.l_max_Q_M (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.l_max_Q_M1 (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.l_min_Q (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.l_min_q (*C var*), 691
PhonyNameDueToError.PhonyNameDueToError.l_min_q1k (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.l_min_q2l (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_B (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_B_rem (*C var*), 694
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_c (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_c_rem (*C var*), 692
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_cH (*C var*), 693
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_cH_rem (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_cI (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_cI_rem (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_cM (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_cM_rem (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.l_min_q_gq (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.l_min_q_local (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_M (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_M1 (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.l_min_Q_M2 (*C var*), 668
PhonyNameDueToError.PhonyNameDueToError.n_seq (*C var*), 670
PhonyNameDueToError.PhonyNameDueToError.oldAliEn (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.p_gq (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.pR (*C var*), 712
PhonyNameDueToError.PhonyNameDueToError.probs (*C var*), 669
PhonyNameDueToError.PhonyNameDueToError.pscore (*C var*), 670
PhonyNameDueToError.PhonyNameDueToError.pscore_local (*C var*), 670
PhonyNameDueToError.PhonyNameDueToError.pscore_pf_compat (*C var*), 666
PhonyNameDueToError.PhonyNameDueToError.ptype (*C var*), 667
PhonyNameDueToError.PhonyNameDueToError.ptype_pf_compat (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.Q (*C var*), 712
PhonyNameDueToError.PhonyNameDueToError.q (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.q1k (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.q2l (*C var*), 715
PhonyNameDueToError.PhonyNameDueToError.Q_B (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.Q_B_rem (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.Q_c (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.Q_c_rem (*C var*), 717
PhonyNameDueToError.PhonyNameDueToError.Q_cH (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.Q_cH_rem (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.Q_cI (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.Q_cI_rem (*C var*), 718
PhonyNameDueToError.PhonyNameDueToError.Q_cM (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.Q_cM_rem (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.q_gq (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.q_local (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.Q_M (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.Q_M1 (*C var*), 716

PhonyNameDueToError.PhonyNameDueToError.Q_M1 (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.Q_M2 (*C var*), 716
PhonyNameDueToError.PhonyNameDueToError.Q_M2_rem (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.Q_M_rem (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.Q_rem (*C var*), 719
PhonyNameDueToError.PhonyNameDueToError.qb (*C var*), 712
PhonyNameDueToError.PhonyNameDueToError.qb_local (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.qho (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.QI5 (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.qio (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.qln (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.qm (*C var*), 712
PhonyNameDueToError.PhonyNameDueToError.qm1 (*C var*), 712
PhonyNameDueToError.PhonyNameDueToError.qm1_new (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.qm2 (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.qm2_local (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.qm2_real (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.qm_local (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.qmb (*C var*), 714
PhonyNameDueToError.PhonyNameDueToError.qmo (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.qo (*C var*), 713
PhonyNameDueToError.PhonyNameDueToError.S (*C var*), 668
PhonyNameDueToError.PhonyNameDueToError.S3 (*C var*), 669
PhonyNameDueToError.PhonyNameDueToError.S5 (*C var*), 669
PhonyNameDueToError.PhonyNameDueToError.S_cons (*C var*), 668
PhonyNameDueToError.PhonyNameDueToError.sc (*C var*), 667
PhonyNameDueToError.PhonyNameDueToError.scs (*C var*), 670
PhonyNameDueToError.PhonyNameDueToError.sequence (*C var*), 666
PhonyNameDueToError.PhonyNameDueToError.sequence_encoding (*C var*), 666
PhonyNameDueToError.PhonyNameDueToError.sequence_encoding2 (*C var*), 666
PhonyNameDueToError.PhonyNameDueToError.sequences (*C var*), 667
PhonyNameDueToError.PhonyNameDueToError.Ss (*C var*), 669
PhonyNameDueToError.pR (*C var*), 721
PhonyNameDueToError.probs (*C var*), 720
PhonyNameDueToError.pscore (*C var*), 674
PhonyNameDueToError.pscore_local (*C var*), 675
PhonyNameDueToError.pscore_pf_compat (*C var*), 675
PhonyNameDueToError.ptype (*C var*), 671
PhonyNameDueToError.ptype_pf_compat (*C var*), 672
PhonyNameDueToError.Q (*C var*), 722
PhonyNameDueToError.q (*C var*), 720
PhonyNameDueToError.q1k (*C var*), 720
PhonyNameDueToError.q2l (*C var*), 722
PhonyNameDueToError.Q_B (*C var*), 723
PhonyNameDueToError.Q_B_rem (*C var*), 726
PhonyNameDueToError.Q_c (*C var*), 724
PhonyNameDueToError.Q_c_rem (*C var*), 726
PhonyNameDueToError.Q_cH (*C var*), 725
PhonyNameDueToError.Q_cH_rem (*C var*), 726
PhonyNameDueToError.Q_cI (*C var*), 725
PhonyNameDueToError.Q_cI_rem (*C var*), 727
PhonyNameDueToError.Q_cM (*C var*), 726
PhonyNameDueToError.Q_cM_rem (*C var*), 727
PhonyNameDueToError.q_gq (*C var*), 720
PhonyNameDueToError.q_local (*C var*), 721
PhonyNameDueToError.Q_M (*C var*), 723
PhonyNameDueToError.Q_M1 (*C var*), 723
PhonyNameDueToError.Q_M1_rem (*C var*), 726
PhonyNameDueToError.Q_M2 (*C var*), 724
PhonyNameDueToError.Q_M2_rem (*C var*), 726
PhonyNameDueToError.Q_M_rem (*C var*), 726
PhonyNameDueToError.Q_rem (*C var*), 726
PhonyNameDueToError.qb (*C var*), 720
PhonyNameDueToError.qb_local (*C var*), 721
PhonyNameDueToError.qho (*C var*), 721
PhonyNameDueToError.QI5 (*C var*), 722
PhonyNameDueToError.qio (*C var*), 721
PhonyNameDueToError.qln (*C var*), 720
PhonyNameDueToError.qm (*C var*), 720
PhonyNameDueToError.qm1 (*C var*), 720
PhonyNameDueToError.qm1_new (*C var*), 721
PhonyNameDueToError.qm2 (*C var*), 720
PhonyNameDueToError.qm2_local (*C var*), 722
PhonyNameDueToError.qm2_real (*C var*), 721
PhonyNameDueToError.qm_local (*C var*), 721
PhonyNameDueToError.qmb (*C var*), 722
PhonyNameDueToError.qmo (*C var*), 721
PhonyNameDueToError.qo (*C var*), 720
PhonyNameDueToError.S (*C var*), 673
PhonyNameDueToError.S3 (*C var*), 674
PhonyNameDueToError.S5 (*C var*), 674
PhonyNameDueToError.S_cons (*C var*), 673

PhonyNameDueToError.sc (*C var*), 672
PhonyNameDueToError.scs (*C var*), 675
PhonyNameDueToError.sequence (*C var*), 670
PhonyNameDueToError.sequence_encoding (*C var*), 671
PhonyNameDueToError.sequence_encoding2 (*C var*), 671
PhonyNameDueToError.sequences (*C var*), 672
PhonyNameDueToError.Ss (*C var*), 674
PhonyNameDueToError.[anonymous] (*C var*), 601, 665, 688, 712
plist (*C type*), 748
plist() (*in module RNA*), 1073
plist_from_probs() (*RNA.fold_compound method*), 1008
plot_data (*class in RNA*), 1074
plot_dp_EPS() (*in module RNA*), 1074
plot_layout (*class in RNA*), 1075
plot_layout_circular() (*in module RNA*), 1075
plot_layout_naview() (*in module RNA*), 1075
plot_layout_puzzler() (*in module RNA*), 1075
plot_layout_simple() (*in module RNA*), 1076
plot_layout_turtle() (*in module RNA*), 1076
plot_options_puzzler (*class in RNA*), 1077
plot_options_puzzler() (*RNA.plot_options_puzzler method*), 1078
plot_structure() (*in module RNA*), 1079
plot_structure_eps() (*in module RNA*), 1079
plot_structure_gml() (*in module RNA*), 1079
plot_structure_ssv() (*in module RNA*), 1079
plot_structure_svg() (*in module RNA*), 1079
plot_structure_xrna() (*in module RNA*), 1079
pop() (*RNA.ConstCharVector method*), 909
pop() (*RNA.CoordinateVector method*), 910
pop() (*RNA.DoubleDoubleVector method*), 911
pop() (*RNA.DoubleVector method*), 912
pop() (*RNA.DuplexVector method*), 913
pop() (*RNA.ElemProbVector method*), 917
pop() (*RNA.HeatCapacityVector method*), 918
pop() (*RNA.HelixVector method*), 918
pop() (*RNA.IntIntVector method*), 919
pop() (*RNA.IntVector method*), 920
pop() (*RNA.MoveVector method*), 923
pop() (*RNA.PathVector method*), 925
pop() (*RNA.SOLUTIONVector method*), 926
pop() (*RNA.StringVector method*), 926
pop() (*RNA.SuboptVector method*), 927
pop() (*RNA.UIntUIntVector method*), 928
pop() (*RNA.UIntVector method*), 929
pop_back() (*RNA.ConstCharVector method*), 909
pop_back() (*RNA.CoordinateVector method*), 910
pop_back() (*RNA.DoubleDoubleVector method*), 911
pop_back() (*RNA.DoubleVector method*), 912
pop_back() (*RNA.DuplexVector method*), 913
pop_back() (*RNA.ElemProbVector method*), 917
pop_back() (*RNA.HeatCapacityVector method*), 918
pop_back() (*RNA.HelixVector method*), 918
pop_back() (*RNA.IntIntVector method*), 919
pop_back() (*RNA.IntVector method*), 920
pop_back() (*RNA.MoveVector method*), 923
pop_back() (*RNA.PathVector method*), 925
pop_back() (*RNA.SOLUTIONVector method*), 926
pop_back() (*RNA.StringVector method*), 926
pop_back() (*RNA.SuboptVector method*), 927
pop_back() (*RNA.UIntUIntVector method*), 928
pop_back() (*RNA.UIntVector method*), 929
pos_3 (*RNA.move attribute*), 1053
pos_3 (*RNA.move property*), 1054
pos_5 (*RNA.move attribute*), 1053
pos_5 (*RNA.move property*), 1055
positional_entropy() (*RNA.fold_compound method*), 1008
post (*RNA.plot_data property*), 1074
PPV (*RNA.score property*), 1087
pr_energy() (*RNA.fold_compound method*), 1008
pr_structure() (*RNA.fold_compound method*), 1008
pre (*RNA.plot_data property*), 1074
previous() (*RNA.SwigPyIterator method*), 928
print_bppm() (*in module RNA*), 1079
print_energy (*C var*), 349
print_tree() (*in module RNA*), 1079
PRIVATE (*C macro*), 641
probing_data (*class in RNA*), 1079
probing_data_deigan() (*in module RNA*), 1080
probing_data_deigan_comparative() (*in module RNA*), 1080
probing_data_eddy() (*in module RNA*), 1080
probing_data_eddy_comparative() (*in module RNA*), 1080
probing_data_free() (*in module RNA*), 1080
probing_data_linear() (*in module RNA*), 1081
probing_data_linear_energies() (*RNA.probing_data method*), 1080
probing_data_linear_num() (*RNA.probing_data method*), 1080
probing_data_linear_raw() (*RNA.probing_data method*), 1080
probing_data_linear_weight() (*RNA.probing_data method*), 1080
probing_data_zarringhalam() (*in module RNA*), 1081
probing_data_zarringhalam_comparative() (*in module RNA*), 1081
probs (*RNA.mx_pf property*), 1055
probs_window() (*RNA.fold_compound method*), 1009
profile_edit_distance() (*in module RNA*), 1081
profiles
    command line option, 106
progress_callback (*C type*), 556
PS_color_dot_plot() (*in module RNA*), 923
PS_color_dot_plot_turn() (*in module RNA*), 923
PS_dot_plot() (*in module RNA*), 923
PS_dot_plot_list() (*in module RNA*), 923
PS_dot_plot_turn() (*in module RNA*), 924
PS_rna_plot (*C function*), 890
PS_rna_plot() (*in module RNA*), 924

PS_rna_plot_a (*C function*), 890
PS_rna_plot_a() (*in module RNA*), 924
PS_rna_plot_a_gquad (*C function*), 890
PS_rna_plot_a_gquad() (*in module RNA*), 924
PS_rna_plot_snoop_a (*C function*), 889
psNumber (*RNA.plot_options_puzzler attribute*), 1078, 1079
pt_pk_remove() (*in module RNA*), 1081
ptable() (*in module RNA*), 1081
ptable_compat_remove() (*in module RNA*), 1082
ptable_pk() (*in module RNA*), 1082
ptype_local (*RNA.fold_compound attribute*), 980
pu_contrib (*C struct*), 751
pu_contrib (*C type*), 749
pu_contrib.E (*C var*), 751
pu_contrib.H (*C var*), 751
pu_contrib.I (*C var*), 751
pu_contrib.length (*C var*), 751
pu_contrib.M (*C var*), 751
pu_contrib.w (*C var*), 751
pu_out (*C struct*), 753
pu_out (*C type*), 749
pu_out.contribs (*C var*), 753
pu_out.header (*C var*), 753
pu_out.len (*C var*), 753
pu_out.u_vals (*C var*), 753
pu_out.u_values (*C var*), 753
PUBLIC (*C macro*), 641
push_back() (*RNA.ConstCharVector method*), 909
push_back() (*RNA.CoordinateVector method*), 910
push_back() (*RNA.DoubleDoubleVector method*), 911
push_back() (*RNA.DoubleVector method*), 912
push_back() (*RNA.DuplexVector method*), 913
push_back() (*RNA.ElemProbVector method*), 917
push_back() (*RNA.HeatCapacityVector method*), 918
push_back() (*RNA.HelixVector method*), 918
push_back() (*RNA.IntIntVector method*), 919
push_back() (*RNA.IntVector method*), 920
push_back() (*RNA.MoveVector method*), 923
push_back() (*RNA.PathVector method*), 925
push_back() (*RNA.SOLUTIONVector method*), 926
push_back() (*RNA.StringVector method*), 926
push_back() (*RNA.SuboptVector method*), 927
push_back() (*RNA.UIntUIntVector method*), 928
push_back() (*RNA.UIntVector method*), 929
putoutpU_prob (*C function*), 344
putoutpU_prob_bin (*C function*), 345

Q

q (*RNA.mx_pf property*), 1055
q1k (*RNA.mx_pf property*), 1055
qb (*RNA.duplexT attribute*), 944
qb (*RNA.duplexT property*), 945
qb (*RNA.mx_pf property*), 1055
qe (*RNA.duplexT attribute*), 944
qe (*RNA.duplexT property*), 945
qho (*RNA.mx_pf property*), 1055
qio (*RNA.mx_pf property*), 1055
qln (*RNA.mx_pf property*), 1055
qm (*RNA.mx_pf property*), 1055
qm1 (*RNA.mx_pf property*), 1055
qm2 (*RNA.mx_pf property*), 1056
qmo (*RNA.mx_pf property*), 1056
qo (*RNA.mx_pf property*), 1056

R

random_string() (*in module RNA*), 1082
rbegin() (*RNA.ConstCharVector method*), 909
rbegin() (*RNA.CoordinateVector method*), 910
rbegin() (*RNA.DoubleDoubleVector method*), 911
rbegin() (*RNA.DoubleVector method*), 912
rbegin() (*RNA.DuplexVector method*), 913
rbegin() (*RNA.ElemProbVector method*), 917
rbegin() (*RNA.HeatCapacityVector method*), 918
rbegin() (*RNA.HelixVector method*), 918
rbegin() (*RNA.IntIntVector method*), 919
rbegin() (*RNA.IntVector method*), 920
rbegin() (*RNA.MoveVector method*), 923
rbegin() (*RNA.PathVector method*), 925
rbegin() (*RNA.SOLUTIONVector method*), 926
rbegin() (*RNA.StringVector method*), 926
rbegin() (*RNA.SuboptVector method*), 927
rbegin() (*RNA.UIntUIntVector method*), 928
rbegin() (*RNA.UIntVector method*), 929
read_parameter_file (*C function*), 403
read_parameter_file() (*in module RNA*), 1083
read_record (*C function*), 768
read_record() (*in module RNA*), 1083
readribosum (*C function*), 779
reference_pt1 (*RNA.fold_compound attribute*), 979
reference_pt2 (*RNA.fold_compound attribute*), 980
referenceBPs1 (*RNA.fold_compound attribute*), 980
referenceBPs2 (*RNA.fold_compound attribute*), 980
rend() (*RNA.ConstCharVector method*), 909
rend() (*RNA.CoordinateVector method*), 910
rend() (*RNA.DoubleDoubleVector method*), 911
rend() (*RNA.DoubleVector method*), 912
rend() (*RNA.DuplexVector method*), 913
rend() (*RNA.ElemProbVector method*), 917
rend() (*RNA.HeatCapacityVector method*), 918
rend() (*RNA.HelixVector method*), 918
rend() (*RNA.IntIntVector method*), 919
rend() (*RNA.IntVector method*), 920
rend() (*RNA.MoveVector method*), 923
rend() (*RNA.PathVector method*), 925
rend() (*RNA.SOLUTIONVector method*), 926
rend() (*RNA.StringVector method*), 926
rend() (*RNA.SuboptVector method*), 927
rend() (*RNA.UIntUIntVector method*), 928
rend() (*RNA.UIntVector method*), 929
reserve() (*RNA.ConstCharVector method*), 909
reserve() (*RNA.CoordinateVector method*), 910
reserve() (*RNA.DoubleDoubleVector method*), 911
reserve() (*RNA.DoubleVector method*), 912
reserve() (*RNA.DuplexVector method*), 913

reserve() (*RNA.ElemProbVector method*), 917
reserve() (*RNA.HeatCapacityVector method*), 918
reserve() (*RNA.HelixVector method*), 918
reserve() (*RNA.IntIntVector method*), 919
reserve() (*RNA.IntVector method*), 920
reserve() (*RNA.MoveVector method*), 923
reserve() (*RNA.PathVector method*), 925
reserve() (*RNA.SOLUTIONVector method*), 926
reserve() (*RNA.StringVector method*), 926
reserve() (*RNA.SuboptVector method*), 927
reserve() (*RNA.UIntUIntVector method*), 928
reserve() (*RNA.UIntVector method*), 929
reset() (*RNA.md method*), 1051
resize() (*RNA.ConstCharVector method*), 910
resize() (*RNA.CoordinateVector method*), 910
resize() (*RNA.DoubleDoubleVector method*), 911
resize() (*RNA.DoubleVector method*), 912
resize() (*RNA.DuplexVector method*), 913
resize() (*RNA.ElemProbVector method*), 917
resize() (*RNA.HeatCapacityVector method*), 918
resize() (*RNA.HelixVector method*), 918
resize() (*RNA.IntIntVector method*), 919
resize() (*RNA.IntVector method*), 920
resize() (*RNA.MoveVector method*), 923
resize() (*RNA.PathVector method*), 925
resize() (*RNA.SOLUTIONVector method*), 926
resize() (*RNA.StringVector method*), 927
resize() (*RNA.SuboptVector method*), 927
resize() (*RNA.UIntUIntVector method*), 928
resize() (*RNA.UIntVector method*), 929
ribo (*C var*), 434
ribo (*RNA.md attribute*), 1047
ribo (*RNA.md property*), 1051
ribosum_mx() (*in module RNA*), 1083
RNA
    module, 908
rotational_symmetry() (*in module RNA*), 1083
rotational_symmetry_db() (*RNA.fold_compound method*), 1010
rtype (*RNA.md attribute*), 1047
rtype (*RNA.md property*), 1051

S

s (*RNA.path property*), 1070
salt (*C var*), 435
salt (*RNA.md attribute*), 1048
salt (*RNA.md property*), 1051
salt_duplex_init() (*in module RNA*), 1084
salt_loop() (*in module RNA*), 1084
salt_loop_int() (*in module RNA*), 1084
salt_ml() (*in module RNA*), 1085
salt_stack() (*in module RNA*), 1085
saltDPXInit (*C var*), 435
SaltDPXInit (*RNA.exp_param attribute*), 961, 965
SaltDPXInit (*RNA.exp_param property*), 965
saltDPXInit (*RNA.md attribute*), 1048
saltDPXInit (*RNA.md property*), 1051
SaltDPXInit (*RNA.param attribute*), 1059, 1063
SaltDPXInit (*RNA.param property*), 1063
saltDPXInitFact (*RNA.md attribute*), 1048
saltDPXInitFact (*RNA.md property*), 1051
SaltLoop (*RNA.param attribute*), 1059, 1063
SaltLoop (*RNA.param property*), 1063
SaltLoopDbl (*RNA.exp_param attribute*), 961, 965
SaltLoopDbl (*RNA.exp_param property*), 965
SaltLoopDbl (*RNA.param attribute*), 1059, 1063
SaltLoopDbl (*RNA.param property*), 1063
SaltMLbase (*RNA.exp_param attribute*), 961, 965
SaltMLbase (*RNA.exp_param property*), 965
SaltMLbase (*RNA.param attribute*), 1059, 1063
SaltMLbase (*RNA.param property*), 1063
SaltMLclosing (*RNA.exp_param attribute*), 961, 965
SaltMLclosing (*RNA.exp_param property*), 965
SaltMLclosing (*RNA.param attribute*), 1059, 1063
SaltMLclosing (*RNA.param property*), 1063
SaltMLintern (*RNA.exp_param attribute*), 961, 965
SaltMLintern (*RNA.exp_param property*), 965
SaltMLintern (*RNA.param attribute*), 1059, 1063
SaltMLintern (*RNA.param property*), 1063
saltMLLower (*RNA.md attribute*), 1048
saltMLLower (*RNA.md property*), 1051
saltMLUpper (*RNA.md attribute*), 1048
saltMLUpper (*RNA.md property*), 1051
SaltStack (*RNA.param attribute*), 1059, 1062
SaltStack (*RNA.param property*), 1063
sc_add_bp() (*RNA.fold_compound method*), 1013
sc_add_bt() (*RNA.fold_compound method*), 1014
sc_add_bt_pycallback() (*in module RNA*), 1085
sc_add_data() (*RNA.fold_compound method*), 1014
sc_add_exp_f() (*RNA.fold_compound method*), 1015
sc_add_exp_f_pycallback() (*in module RNA*), 1085
sc_add_f() (*RNA.fold_compound method*), 1015
sc_add_f_pycallback() (*in module RNA*), 1085
sc_add_hi_motif() (*RNA.fold_compound method*), 1016
sc_add_pydata() (*in module RNA*), 1085
sc_add_SHAPE_deigan() (*RNA.fold_compound method*), 1011
sc_add_SHAPE_deigan_ali() (*RNA.fold_compound method*), 1011
sc_add_SHAPE_eddy_2() (*RNA.fold_compound method*), 1012
sc_add_SHAPE_zarringhalam() (*RNA.fold_compound method*), 1012
sc_add_stack() (*RNA.fold_compound method*), 1016
sc_add_up() (*RNA.fold_compound method*), 1016
sc_init() (*RNA.fold_compound method*), 1017
sc_mod() (*RNA.fold_compound method*), 1017
sc_mod_7DA() (*RNA.fold_compound method*), 1018
sc_mod_dihydrouridine() (*RNA.fold_compound method*), 1019
sc_mod_inosine() (*RNA.fold_compound method*), 1019
sc_mod_json() (*RNA.fold_compound method*), 1020

sc_mod_jsonfile() (*RNA.fold_compound method*), 1020
sc_mod_m6A() (*RNA.fold_compound method*), 1021
sc_mod_param (*class in RNA*), 1085
sc_mod_parameters_free() (*in module RNA*), 1085
sc_mod_pseudouridine() (*RNA.fold_compound method*), 1022
sc_mod_purine() (*RNA.fold_compound method*), 1022
sc_mod_read_from_json() (*in module RNA*), 1085
sc_mod_read_from_jsonfile() (*in module RNA*), 1086
sc_multi_cb_add() (*RNA.fold_compound method*), 1023
sc_multi_cb_add_pycallback() (*in module RNA*), 1086
sc_probing() (*RNA.fold_compound method*), 1023
sc_remove() (*RNA.fold_compound method*), 1023
sc_set_bp() (*RNA.fold_compound method*), 1024
sc_set_stack() (*RNA.fold_compound method*), 1024
sc_set_up() (*RNA.fold_compound method*), 1024
scale (*RNA.mx_pf property*), 1056
scale_parameters (*C function*), 383
scale_pf_parameters (*C function*), 384
score (*class in RNA*), 1086
second (*RNA.DoublePair property*), 911
sect (*C type*), 749
seq_encode() (*in module RNA*), 1087
sequence (*RNA.fold_compound property*), 1025
sequence_add() (*RNA.fold_compound method*), 1025
sequence_encoding (*RNA.fold_compound property*), 1025
sequence_encoding2 (*RNA.fold_compound property*), 1025
sequence_prepare() (*RNA.fold_compound method*), 1025
sequence_remove() (*RNA.fold_compound method*), 1025
sequence_remove_all() (*RNA.fold_compound method*), 1025
set_from_globals() (*RNA.md method*), 1051
set_model_details (*C function*), 431
set_parameters (*C function*), 384
set_pf_param (*C function*), 384
settype (*C function*), 403
settype() (*in module RNA*), 1087
sfact (*RNA.md attribute*), 1047
sfact (*RNA.md property*), 1051
shortP_getitem() (*in module RNA*), 1087
shortP_setitem() (*in module RNA*), 1087
simple_circplot_coordinates() (*in module RNA*), 1087
simple_xy_coordinates() (*in module RNA*), 1087
size (*RNA.ep attribute*), 950, 951
size (*RNA.ep property*), 951
size() (*RNA.ConstCharVector method*), 910
size() (*RNA.CoordinateVector method*), 910
size() (*RNA.DoubleDoubleVector method*), 911
size() (*RNA.DoubleVector method*), 912
size() (*RNA.DuplexVector method*), 913
size() (*RNA.ElemProbVector method*), 917
size() (*RNA.HeatCapacityVector method*), 918
size() (*RNA.HelixVector method*), 919
size() (*RNA.IntIntVector method*), 919
size() (*RNA.IntVector method*), 920
size() (*RNA.MoveVector method*), 923
size() (*RNA.PathVector method*), 925
size() (*RNA.SOLUTION method*), 925
size() (*RNA.SOLUTIONVector method*), 926
size() (*RNA.StringVector method*), 927
size() (*RNA.SuboptVector method*), 927
size() (*RNA.UIntUIntVector method*), 928
size() (*RNA.UIntVector method*), 929
size() (*RNA.varArrayChar method*), 1093
size() (*RNA.varArrayFLTorDBL method*), 1093
size() (*RNA.varArrayInt method*), 1093
size() (*RNA.varArrayMove method*), 1093
size() (*RNA.varArrayShort method*), 1094
size() (*RNA.varArrayUChar method*), 1094
size() (*RNA.varArrayUInt method*), 1094
sn (*RNA.hc attribute*), 1035, 1037
snoopT (*C struct*), 755
snoopT.Duplex_El (*C var*), 756
snoopT.Duplex_Er (*C var*), 756
snoopT.Duplex_Ol (*C var*), 756
snoopT.Duplex_Or (*C var*), 756
snoopT.Duplex_Ot (*C var*), 756
snoopT.energy (*C var*), 756
snoopT.fullStemEnergy (*C var*), 757
snoopT.i (*C var*), 755
snoopT.j (*C var*), 755
snoopT.Loop_D (*C var*), 756
snoopT.Loop_E (*C var*), 756
snoopT.pscd (*C var*), 756
snoopT.pscg (*C var*), 756
snoopT.psct (*C var*), 756
snoopT.structure (*C var*), 756
snoopT.u (*C var*), 755
SOLUTION (*class in RNA*), 925
SOLUTIONVector (*class in RNA*), 925
special_hp (*RNA.md attribute*), 1045
special_hp (*RNA.md property*), 1051
ssv_rna_plot (*C function*), 889
ssv_rna_plot() (*in module RNA*), 1088
st_back (*C var*), 1111
stack (*RNA.param attribute*), 1056, 1060
stack (*RNA.param property*), 1064
stack_prob() (*RNA.fold_compound method*), 1025
stackProb (*C function*), 342
stackProb() (*in module RNA*), 1088
start (*RNA.hx property*), 1040
stat_cb (*RNA.fold_compound attribute*), 978
state (*RNA.hc attribute*), 1035, 1037
STR (*C macro*), 797
str_to_dbl_array() (*in module RNA*), 1088
strand_end (*RNA.fold_compound attribute*), 977

strand_end (*RNA.fold_compound property*), 1025
strand_number (*RNA.fold_compound attribute*), 977
strand_number (*RNA.fold_compound property*), 1025
strand_order (*RNA.fold_compound attribute*), 977
strand_order (*RNA.fold_compound property*), 1026
strand_order_uniq (*RNA.fold_compound attribute*), 977
strand_start (*RNA.fold_compound attribute*), 977
strand_start (*RNA.fold_compound property*), 1026
strands (*RNA.fold_compound attribute*), 977
strands (*RNA.fold_compound property*), 1026
strands (*RNA.hc attribute*), 1035, 1037
strands (*RNA.mx_mfe property*), 1055
string_edit_distance() (*in module RNA*), 1088
StringVector (*class in RNA*), 926
strtrim() (*in module RNA*), 1088
struct_en (*class in RNA*), 1089
structure (*RNA.duplex_list_t property*), 945
structure (*RNA.duplexT attribute*), 943
structure (*RNA.duplexT property*), 945
structure (*RNA.SOLUTION property*), 925
structure (*RNA.struct_en attribute*), 1089, 1090
structure (*RNA.struct_en property*), 1090
structure (*RNA.subopt_solution property*), 1090
subopt (*C function*), 348
subopt() (*in module RNA*), 1090
subopt() (*RNA.fold_compound method*), 1026
subopt_cb() (*RNA.fold_compound method*), 1026
subopt_circ (*C function*), 349
subopt_par (*C function*), 349
subopt_solution (*class in RNA*), 1090
subopt_sorted (*C var*), 349
subopt_zuker() (*RNA.fold_compound method*), 1027
SuboptVector (*class in RNA*), 927
svg_rna_plot (*C function*), 889
svg_rna_plot() (*in module RNA*), 1090
swap() (*RNA.ConstCharVector method*), 910
swap() (*RNA.CoordinateVector method*), 910
swap() (*RNA.DoubleDoubleVector method*), 911
swap() (*RNA.DoubleVector method*), 912
swap() (*RNA.DuplexVector method*), 913
swap() (*RNA.ElemProbVector method*), 917
swap() (*RNA.HeatCapacityVector method*), 918
swap() (*RNA.HelixVector method*), 919
swap() (*RNA.IntIntVector method*), 919
swap() (*RNA.IntVector method*), 920
swap() (*RNA.MoveVector method*), 923
swap() (*RNA.PathVector method*), 925
swap() (*RNA.SOLUTIONVector method*), 926
swap() (*RNA.StringVector method*), 927
swap() (*RNA.SuboptVector method*), 927
swap() (*RNA.UIntUIntVector method*), 928
swap() (*RNA.UIntVector method*), 929
SwigPyIterator (*class in RNA*), 927
symbolset (*C var*), 496

T

tb (*RNA.duplexT attribute*), 944
tb (*RNA.duplexT property*), 945
te (*RNA.duplexT attribute*), 944
te (*RNA.duplexT property*), 945
temperature (*C var*), 431
temperature (*RNA.exp_param attribute*), 960, 964
temperature (*RNA.exp_param property*), 967
temperature (*RNA.heat_capacity_result property*), 1039
temperature (*RNA.md attribute*), 1045
temperature (*RNA.md property*), 1051
temperature (*RNA.param attribute*), 1059, 1062
temperature (*RNA.param property*), 1064
TerminalAU (*RNA.param attribute*), 1058, 1061
TerminalAU (*RNA.param property*), 1063
tetra_loop (*C var*), 432
Tetraloop_E (*RNA.param attribute*), 1058, 1061
Tetraloop_E (*RNA.param property*), 1063
Tetraloops (*RNA.exp_param attribute*), 959, 963
Tetraloops (*RNA.exp_param property*), 965
Tetraloops (*RNA.param attribute*), 1058, 1061
Tetraloops (*RNA.param property*), 1064
thisown (*RNA.basepair property*), 935
thisown (*RNA.cmd property*), 938
thisown (*RNA.ConstCharVector property*), 910
thisown (*RNA.COORDINATE property*), 909
thisown (*RNA.CoordinateVector property*), 910
thisown (*RNA.doubleArray property*), 943
thisown (*RNA.DoubleDoubleVector property*), 911
thisown (*RNA.DoublePair property*), 911
thisown (*RNA.DoubleVector property*), 912
thisown (*RNA.duplex_list_t property*), 945
thisown (*RNA.duplexT property*), 945
thisown (*RNA.DuplexVector property*), 913
thisown (*RNA.ElemProbVector property*), 917
thisown (*RNA.ep property*), 951
thisown (*RNA.eqm_probs_opt property*), 952
thisown (*RNA.exp_param property*), 967
thisown (*RNA.floatArray property*), 975
thisown (*RNA.fold_compound property*), 1027
thisown (*RNA.hc property*), 1038
thisown (*RNA.heat_capacity_result property*), 1039
thisown (*RNA.HeatCapacityVector property*), 918
thisown (*RNA.HelixVector property*), 919
thisown (*RNA.hx property*), 1040
thisown (*RNA.intArray property*), 1040
thisown (*RNA.IntIntVector property*), 919
thisown (*RNA.IntVector property*), 920
thisown (*RNA.md property*), 1051
thisown (*RNA.move property*), 1055
thisown (*RNA.MoveVector property*), 923
thisown (*RNA.mx_mfe property*), 1055
thisown (*RNA.mx_pf property*), 1056
thisown (*RNA.param property*), 1064
thisown (*RNA.path property*), 1070
thisown (*RNA.path_options property*), 1070
thisown (*RNA.PathVector property*), 925
thisown (*RNA.pbacktrack_mem property*), 1072
thisown (*RNA.plot_data property*), 1074

thisown (*RNA.plot_layout property*), 1075
thisown (*RNA.plot_options_puzzler property*), 1079
thisown (*RNA.probing_data property*), 1080
thisown (*RNA.sc_mod_param property*), 1085
thisown (*RNA.score property*), 1087
thisown (*RNA.SOLUTION property*), 925
thisown (*RNA.SOLUTIONVector property*), 926
thisown (*RNA.StringVector property*), 927
thisown (*RNA.struct_en property*), 1090
thisown (*RNA.subopt_solution property*), 1090
thisown (*RNA.SuboptVector property*), 927
thisown (*RNA.SwigPyIterator property*), 928
thisown (*RNA.UIntUIntVector property*), 928
thisown (*RNA.UIntVector property*), 929
thisown (*RNA.varArrayChar property*), 1093
thisown (*RNA.varArrayFLTorDBL property*), 1093
thisown (*RNA.varArrayInt property*), 1093
thisown (*RNA.varArrayMove property*), 1093
thisown (*RNA.varArrayShort property*), 1094
thisown (*RNA.varArrayUChar property*), 1094
thisown (*RNA.varArrayUInt property*), 1094
TN (*RNA.score property*), 1087
TNR (*RNA.score property*), 1087
TP (*RNA.score property*), 1087
TPR (*RNA.score property*), 1087
tree_edit_distance() (*in module RNA*), 1090
tree_string_to_db() (*in module RNA*), 1090
tree_string_unweight() (*in module RNA*), 1091
Triloop_E (*RNA.param attribute*), 1058, 1061
Triloop_E (*RNA.param property*), 1064
Triloops (*RNA.exp_param attribute*), 960, 963
Triloops (*RNA.exp_param property*), 965
Triloops (*RNA.param attribute*), 1058, 1062
Triloops (*RNA.param property*), 1064
TripleC (*RNA.param attribute*), 1058, 1062
TripleC (*RNA.param property*), 1064
TwoDfold (*C function*), 501
TwoDfold_backtrack_f5 (*C function*), 500
TwoDfold_solution (*C macro*), 498
TwoDfold_vars (*C struct*), 502
TwoDfold_vars.bpdist (*C var*), 503
TwoDfold_vars.circ (*C var*), 503
TwoDfold_vars.compatibility (*C var*), 509
TwoDfold_vars.dangles (*C var*), 504
TwoDfold_vars.do_backtrack (*C var*), 502
TwoDfold_vars.E_C (*C var*), 504
TwoDfold_vars.E_C_rem (*C var*), 508
TwoDfold_vars.E_F3 (*C var*), 504
TwoDfold_vars.E_F3_rem (*C var*), 508
TwoDfold_vars.E_F5 (*C var*), 504
TwoDfold_vars.E_F5_rem (*C var*), 508
TwoDfold_vars.E_Fc (*C var*), 504
TwoDfold_vars.E_Fc_rem (*C var*), 508
TwoDfold_vars.E_FcH (*C var*), 504
TwoDfold_vars.E_FcH_rem (*C var*), 508
TwoDfold_vars.E_FcI (*C var*), 504
TwoDfold_vars.E_FcI_rem (*C var*), 509
TwoDfold_vars.E_FcM (*C var*), 504
TwoDfold_vars.E_FcM_rem (*C var*), 509
TwoDfold_vars.E_M (*C var*), 504
TwoDfold_vars.E_M1 (*C var*), 504
TwoDfold_vars.E_M1_rem (*C var*), 508
TwoDfold_vars.E_M2 (*C var*), 504
TwoDfold_vars.E_M2_rem (*C var*), 508
TwoDfold_vars.E_M_rem (*C var*), 508
TwoDfold_vars.k_max_values (*C var*), 505
TwoDfold_vars.k_max_values_f (*C var*), 506
TwoDfold_vars.k_max_values_f3 (*C var*), 506
TwoDfold_vars.k_max_values_fc (*C var*), 507
TwoDfold_vars.k_max_values_fcH (*C var*), 507
TwoDfold_vars.k_max_values_fcI (*C var*), 507
TwoDfold_vars.k_max_values_fcM (*C var*), 508
TwoDfold_vars.k_max_values_m (*C var*), 505
TwoDfold_vars.k_max_values_m1 (*C var*), 505
TwoDfold_vars.k_max_values_m2 (*C var*), 506
TwoDfold_vars.k_min_values (*C var*), 505
TwoDfold_vars.k_min_values_f (*C var*), 506
TwoDfold_vars.k_min_values_f3 (*C var*), 506
TwoDfold_vars.k_min_values_fc (*C var*), 507
TwoDfold_vars.k_min_values_fcH (*C var*), 507
TwoDfold_vars.k_min_values_fcI (*C var*), 507
TwoDfold_vars.k_min_values_fcM (*C var*), 508
TwoDfold_vars.k_min_values_m (*C var*), 505
TwoDfold_vars.k_min_values_m1 (*C var*), 505
TwoDfold_vars.k_min_values_m2 (*C var*), 506
TwoDfold_vars.l_max_values (*C var*), 505
TwoDfold_vars.l_max_values_f (*C var*), 506
TwoDfold_vars.l_max_values_f3 (*C var*), 506
TwoDfold_vars.l_max_values_fc (*C var*), 507
TwoDfold_vars.l_max_values_fcH (*C var*), 507
TwoDfold_vars.l_max_values_fcI (*C var*), 507
TwoDfold_vars.l_max_values_fcM (*C var*), 508
TwoDfold_vars.l_max_values_m (*C var*), 505
TwoDfold_vars.l_max_values_m1 (*C var*), 505
TwoDfold_vars.l_max_values_m2 (*C var*), 506
TwoDfold_vars.l_min_values (*C var*), 505
TwoDfold_vars.l_min_values_f (*C var*), 506
TwoDfold_vars.l_min_values_f3 (*C var*), 506
TwoDfold_vars.l_min_values_fc (*C var*), 507
TwoDfold_vars.l_min_values_fcH (*C var*), 507
TwoDfold_vars.l_min_values_fcI (*C var*), 507
TwoDfold_vars.l_min_values_fcM (*C var*), 508
TwoDfold_vars.l_min_values_m (*C var*), 505
TwoDfold_vars.l_min_values_m1 (*C var*), 505
TwoDfold_vars.l_min_values_m2 (*C var*), 506
TwoDfold_vars.maxD1 (*C var*), 502
TwoDfold_vars.maxD2 (*C var*), 503
TwoDfold_vars.mm1 (*C var*), 503
TwoDfold_vars.mm2 (*C var*), 503
TwoDfold_vars.my_iindx (*C var*), 503
TwoDfold_vars.P (*C var*), 502
TwoDfold_vars.ptype (*C var*), 502
TwoDfold_vars.reference_pt1 (*C var*), 503
TwoDfold_vars.reference_pt2 (*C var*), 503
TwoDfold_vars.referenceBPs1 (*C var*), 503
TwoDfold_vars.referenceBPs2 (*C var*), 503

TwoDfold_vars.S (*C var*), 502
TwoDfold_vars.S1 (*C var*), 502
TwoDfold_vars.seq_length (*C var*), 504
TwoDfold_vars.sequence (*C var*), 502
TwoDfold_vars.temperature (*C var*), 503
TwoDfoldList (*C function*), 500
type (*RNA.ep attribute*), 950, 951
type (*RNA.ep property*), 951
type (*RNA.fold_compound attribute*), 977
type (*RNA.fold_compound property*), 1027
type (*RNA.hc attribute*), 1035, 1037
type (*RNA.hc property*), 1038
type (*RNA.mx_mfe property*), 1055
type (*RNA.mx_pf property*), 1056
type (*RNA.path property*), 1070
type() (*RNA.varArrayChar method*), 1093
type() (*RNA.varArrayFLTorDBL method*), 1093
type() (*RNA.varArrayInt method*), 1093
type() (*RNA.varArrayMove method*), 1093
type() (*RNA.varArrayShort method*), 1094
type() (*RNA.varArrayUChar method*), 1094
type() (*RNA.varArrayUInt method*), 1094

U

ubf_eval_ext_int_loop (*C function*), 483
ubf_eval_ext_int_loop() (*in module RNA*), 1091
ubf_eval_int_loop (*C function*), 482
ubf_eval_int_loop() (*in module RNA*), 1091
ubf_eval_int_loop2 (*C function*), 483
ubf_eval_int_loop2() (*in module RNA*), 1091
ud_add_motif() (*RNA.fold_compound method*), 1027
ud_remove() (*RNA.fold_compound method*), 1028
ud_set_data() (*RNA.fold_compound method*), 1028
ud_set_exp_prod_cb() (*in module RNA*), 1091
ud_set_exp_prod_rule_cb() (*RNA.fold_compound method*), 1029
ud_set_prob_cb() (*in module RNA*), 1091
ud_set_prob_cb() (*RNA.fold_compound method*), 1029
ud_set_prod_cb() (*in module RNA*), 1091
ud_set_prod_rule_cb() (*RNA.fold_compound method*), 1029
ud_set_pydata() (*in module RNA*), 1091
UIntUIntVector (*class in RNA*), 928
UIntVector (*class in RNA*), 928
unexpand_aligned_F() (*in module RNA*), 1092
unexpand_Full() (*in module RNA*), 1091
uniq_ML (*C var*), 433
uniq_ML (*RNA.md attribute*), 1046
uniq_ML (*RNA.md property*), 1051
unpack_structure() (*in module RNA*), 1092
unpaired (*RNA.plot_options_puzzler attribute*), 1077, 1078
unweight() (*in module RNA*), 1092
up3 (*RNA.hx property*), 1040
up5 (*RNA.hx property*), 1040
up_ext (*RNA.hc attribute*), 1035, 1037
up_ext (*RNA.hc property*), 1038
up_hp (*RNA.hc attribute*), 1035, 1037
up_hp (*RNA.hc property*), 1038
up_int (*RNA.hc attribute*), 1035, 1037
up_int (*RNA.hc property*), 1038
up_length (*RNA.eqm_probs_opt attribute*), 951
up_length (*RNA.eqm_probs_opt property*), 952
up_ml (*RNA.hc attribute*), 1035, 1037
up_ml (*RNA.hc property*), 1038
Up_plot (*C function*), 495
update_alifold_params (*C function*), 298
update_co_pf_params (*C function*), 336
update_co_pf_params() (*in module RNA*), 1092
update_co_pf_params_par (*C function*), 337
update_cofold_params (*C function*), 299
update_cofold_params() (*in module RNA*), 1092
update_cofold_params_par (*C function*), 300
update_fold_params (*C function*), 303
update_fold_params() (*in module RNA*), 1092
update_fold_params_par (*C function*), 304
update_pf_params (*C function*), 340
update_pf_params() (*in module RNA*), 1092
update_pf_params_par (*C function*), 340
update_pf_paramsLP (*C function*), 343
update_pf_paramsLP_par (*C function*), 344
urn() (*in module RNA*), 1092
ushortP_getitem() (*in module RNA*), 1093
ushortP_setitem() (*in module RNA*), 1093

V

value() (*RNA.SwigPyIterator method*), 928
var_array_Iterator (*class in RNA*), 1094
varArrayChar (*class in RNA*), 1093
varArrayFLTorDBL (*class in RNA*), 1093
varArrayInt (*class in RNA*), 1093
varArrayMove (*class in RNA*), 1093
varArrayShort (*class in RNA*), 1093
varArrayUChar (*class in RNA*), 1094
varArrayUInt (*class in RNA*), 1094
vrna__array_set_capacity (*C function*), 740
vrna_abstract_shapes (*C function*), 824
vrna_abstract_shapes_pt (*C function*), 825
vrna_alifold (*C function*), 290
vrna_alignment_s (*C struct*), 796
vrna_alignment_s.a2s (*C var*), 797
vrna_alignment_s.gapfree_seq (*C var*), 796
vrna_alignment_s.gapfree_size (*C var*), 796
vrna_alignment_s.genome_size (*C var*), 797
vrna_alignment_s.n_seq (*C var*), 796
vrna_alignment_s.orientation (*C var*), 797
vrna_alignment_s.sequences (*C var*), 796
vrna_alignment_s.start (*C var*), 797
vrna_aliLfold (*C function*), 294
vrna_aliLfold_cb (*C function*), 294
vrna_alloc (*C function*), 643
vrna_aln_consensus_mis (*C function*), 811
vrna_aln_consensus_sequence (*C function*), 811
vrna_aln_conservation_col (*C function*), 810
vrna_aln_conservation_struct (*C function*), 809

vrna_aln_copy (*C function*), 809
VRNA_ALN_DEFAULT (*C macro*), 806
VRNA_ALN_DNA (*C macro*), 806
vrna_aln_encode (*C function*), 794
vrna_aln_free (*C function*), 808
VRNA_ALN_LOWERCASE (*C macro*), 806
vrna_aln_mpi (*C function*), 807
vrna_aln_opt_t (*C struct*), 886
vrna_aln_opt_t.color_min_sat (*C var*), 886
vrna_aln_opt_t.color_threshold (*C var*), 886
vrna_aln_opt_t.columns (*C var*), 886
vrna_aln_opt_t.end (*C var*), 886
vrna_aln_opt_t.offset (*C var*), 886
vrna_aln_opt_t.start (*C var*), 886
vrna_aln_pinfo (*C function*), 807
vrna_aln_pscore (*C function*), 807
VRNA_ALN_RNA (*C macro*), 806
vrna_aln_slice (*C function*), 808
vrna_aln_toRNA (*C function*), 809
vrna_aln_uppercase (*C function*), 808
VRNA_ALN_UPPERCASE (*C macro*), 806
vrna_annotate_covar_db (*C function*), 880
vrna_annotate_covar_db_extended (*C function*), 881
vrna_annotate_covar_pairs (*C function*), 881
vrna_annotate_covar_pt (*C function*), 881
VRNA_ANY_LOOP (*C macro*), 316
vrna_array (*C macro*), 738
vrna_array_append (*C macro*), 739
vrna_array_capacity (*C macro*), 738
vrna_array_free (*C macro*), 739
vrna_array_grow (*C macro*), 739
VRNA_ARRAY_GROW_FORMULA (*C macro*), 738
VRNA_ARRAY_HEADER (*C macro*), 738
vrna_array_header_s (*C struct*), 740
vrna_array_header_s.num (*C var*), 740
vrna_array_header_s.size (*C var*), 740
vrna_array_header_t (*C type*), 739
vrna_array_init (*C macro*), 739
vrna_array_init_size (*C macro*), 739
vrna_array_make (*C macro*), 738
VRNA_ARRAY_REGISTER (*C macro*), 739
VRNA_ARRAY_REGISTER_NAME (*C macro*), 739
vrna_array_set_capacity (*C macro*), 739
vrna_array_size (*C macro*), 738
vrna_auxdata_free_f (*C type*), 649
vrna_auxdata_prepare_f (*C type*), 650
vrna_backtrack5 (*C function*), 295
vrna_backtrack5_TwoD (*C function*), 498
vrna_backtrack_from_intervals (*C function*), 295
vrna_backtrack_window (*C function*), 293
vrna_basename (*C function*), 778
vrna_basepair_s (*C struct*), 763
vrna_basepair_s.i (*C var*), 763
vrna_basepair_s.j (*C var*), 763
vrna_basepair_s.L (*C var*), 763
vrna_basepair_s.l (*C var*), 763
vrna_basepair_t (*C struct*), 750
vrna_basepair_t.i (*C var*), 750
vrna_basepair_t.j (*C var*), 750
vrna_boltzmann_sampling_callback (*C type*), 350
vrna_boustrophedon (*C function*), 861
vrna_boustrophedon_pos (*C function*), 861
vrna_bp_distance (*C function*), 830
vrna_bp_distance_pt (*C function*), 829
vrna_bp_stack_t (*C struct*), 751
vrna_bp_stack_t.i (*C var*), 751
vrna_bp_stack_t.j (*C var*), 751
vrna_bp_t (*C type*), 759
vrna_bpp_symbol (*C function*), 834
vrna_bps_at (*C function*), 762
vrna_bps_free (*C function*), 761
vrna_bps_init (*C function*), 760
vrna_bps_pop (*C function*), 761
vrna_bps_push (*C function*), 761
vrna_bps_size (*C function*), 762
vrna_bps_t (*C type*), 758
vrna_bps_top (*C function*), 761
VRNA_BRACKETS_ALPHA (*C macro*), 812
VRNA_BRACKETS_ANG (*C macro*), 813
VRNA_BRACKETS_ANY (*C macro*), 814
VRNA_BRACKETS_CLY (*C macro*), 813
VRNA_BRACKETS_DEFAULT (*C macro*), 813
VRNA_BRACKETS_RND (*C macro*), 813
VRNA_BRACKETS_SQR (*C macro*), 813
vrna_bs_result_f (*C type*), 350
vrna_bt_exterior_f3 (*C function*), 295
vrna_bt_exterior_f3_pp (*C function*), 295
vrna_bt_exterior_f5 (*C function*), 295
vrna_bt_f (*C function*), 295
vrna_bt_gquad (*C function*), 575
vrna_bt_gquad_internal (*C function*), 575
vrna_bt_gquad_mfe (*C function*), 575
vrna_bt_hairpin (*C function*), 296
vrna_bt_internal_loop (*C function*), 296
vrna_bt_m (*C function*), 296
vrna_bt_multibranch_loop (*C function*), 296
vrna_bt_multibranch_split (*C function*), 297
vrna_bt_stacked_pairs (*C function*), 296
vrna_bts_free (*C function*), 759
vrna_bts_init (*C function*), 759
vrna_bts_pop (*C function*), 760
vrna_bts_push (*C function*), 760
vrna_bts_size (*C function*), 760
vrna_bts_t (*C type*), 758
vrna_bts_top (*C function*), 760
vrna_C11_features (*C function*), 750
vrna_callback_free_auxdata (*C type*), 650
vrna_callback_heap_cmp (*C type*), 734
vrna_callback_heap_get_pos (*C type*), 734
vrna_callback_heap_set_pos (*C type*), 734
vrna_callback_ht_compare_entries (*C type*), 727
vrna_callback_ht_free_entry (*C type*), 728
vrna_callback_ht_hash_function (*C type*), 728
vrna_callback_move_update (*C type*), 515
vrna_callback_pk_plex_score (*C type*), 569

vrna_callback_recursion_status (*C type*), 651
vrna_callback_sc_backtrack (*C type*), 615
vrna_callback_sc_energy (*C type*), 613
vrna_callback_sc_exp_energy (*C type*), 614
vrna_callback_stream_output (*C type*), 742
vrna_centroid (*C function*), 374
vrna_centroid_from_plist (*C function*), 374
vrna_centroid_from_probs (*C function*), 375
vrna_circalifold (*C function*), 290
vrna_circfold (*C function*), 289
VRNA_CMD_PARSE_DEFAULTS (*C macro*), 776
VRNA_CMD_PARSE_HC (*C macro*), 775
VRNA_CMD_PARSE_SC (*C macro*), 775
VRNA_CMD_PARSE_SD (*C macro*), 776
VRNA_CMD_PARSE_SILENT (*C macro*), 776
VRNA_CMD_PARSE_UD (*C macro*), 775
vrna_cmd_t (*C type*), 776
vrna_cofold (*C function*), 291
vrna_color_t (*C type*), 748
vrna_commands_apply (*C function*), 777
vrna_commands_free (*C function*), 778
vrna_compare_structure (*C function*), 832
vrna_compare_structure_pt (*C function*), 831
VRNA_CONSTRAINT_CONTEXT_ALL_LOOPS (*C macro*), 587
VRNA_CONSTRAINT_CONTEXT_EXT_LOOP (*C macro*), 586
VRNA_CONSTRAINT_CONTEXT_HP_LOOP (*C macro*), 586
VRNA_CONSTRAINT_CONTEXT_INT_LOOP (*C macro*), 586
VRNA_CONSTRAINT_CONTEXT_INT_LOOP_ENC (*C macro*), 586
VRNA_CONSTRAINT_CONTEXT_MB_LOOP (*C macro*), 586
VRNA_CONSTRAINT_CONTEXT_MB_LOOP_ENC (*C macro*), 586
VRNA_CONSTRAINT_DB (*C macro*), 583
VRNA_CONSTRAINT_DB_DEFAULT (*C macro*), 586
VRNA_CONSTRAINT_DB_DOT (*C macro*), 584
VRNA_CONSTRAINT_DB_ENFORCE_BP (*C macro*), 583
VRNA_CONSTRAINT_DB_GQUAD (*C macro*), 585
VRNA_CONSTRAINT_DB_INTERMOL (*C macro*), 585
VRNA_CONSTRAINT_DB_INTRAMOL (*C macro*), 585
VRNA_CONSTRAINT_DB_PIPE (*C macro*), 584
VRNA_CONSTRAINT_DB_RND_BRACK (*C macro*), 584
VRNA_CONSTRAINT_DB_WUSS (*C macro*), 585
VRNA_CONSTRAINT_DB_X (*C macro*), 584
VRNA_CONSTRAINT_FILE (*C macro*), 577
VRNA_CONSTRAINT_MULTILINE (*C macro*), 763
VRNA_CONSTRAINT_SOFT_MFE (*C macro*), 577
VRNA_CONSTRAINT_SOFT_PF (*C macro*), 577
vrna_constraints_add (*C function*), 588
vrna_constraints_add_SHAPE (*C function*), 555
vrna_constraints_add_SHAPE_ali (*C function*), 555
vrna_convert_dcal_to_kcal (*C function*), 865
vrna_convert_energy (*C function*), 864
vrna_convert_kcal_to_dcal (*C function*), 865
VRNA_CONVERT_OUTPUT_ALL (*C macro*), 404
VRNA_CONVERT_OUTPUT_BULGE (*C macro*), 405
VRNA_CONVERT_OUTPUT_DANGLE3 (*C macro*), 405
VRNA_CONVERT_OUTPUT_DANGLE5 (*C macro*), 404
VRNA_CONVERT_OUTPUT_DUMP (*C macro*), 406
VRNA_CONVERT_OUTPUT_HP (*C macro*), 404
VRNA_CONVERT_OUTPUT_INT (*C macro*), 405
VRNA_CONVERT_OUTPUT_INT_11 (*C macro*), 405
VRNA_CONVERT_OUTPUT_INT_21 (*C macro*), 405
VRNA_CONVERT_OUTPUT_INT_22 (*C macro*), 405
VRNA_CONVERT_OUTPUT_MISC (*C macro*), 405
VRNA_CONVERT_OUTPUT_ML (*C macro*), 405
VRNA_CONVERT_OUTPUT_MM_EXT (*C macro*), 404
VRNA_CONVERT_OUTPUT_MM_HP (*C macro*), 404
VRNA_CONVERT_OUTPUT_MM_INT (*C macro*), 404
VRNA_CONVERT_OUTPUT_MM_INT_1N (*C macro*), 404
VRNA_CONVERT_OUTPUT_MM_INT_23 (*C macro*), 404
VRNA_CONVERT_OUTPUT_MM_MULTI (*C macro*), 404
VRNA_CONVERT_OUTPUT_NINIO (*C macro*), 406
VRNA_CONVERT_OUTPUT_SPECIAL_HP (*C macro*), 405
VRNA_CONVERT_OUTPUT_STACK (*C macro*), 404
VRNA_CONVERT_OUTPUT_VANILLA (*C macro*), 406
vrna_convert_temperature (*C function*), 864
vrna_cpair_t (*C type*), 748
vrna_cstr (*C function*), 742
vrna_cstr_close (*C function*), 743
vrna_cstr_discard (*C function*), 743
vrna_cstr_fflush (*C function*), 744
vrna_cstr_free (*C function*), 743
vrna_cstr_message_info (*C function*), 744
vrna_cstr_message_vinfo (*C function*), 744
vrna_cstr_message_vwarning (*C function*), 744
vrna_cstr_message_warning (*C function*), 744
vrna_cstr_print_eval_ext_loop (*C function*), 745
vrna_cstr_print_eval_ext_loop_revert (*C function*), 745
vrna_cstr_print_eval_gquad (*C function*), 746
vrna_cstr_print_eval_hp_loop (*C function*), 745
vrna_cstr_print_eval_hp_loop_revert (*C function*), 745
vrna_cstr_print_eval_int_loop (*C function*), 746
vrna_cstr_print_eval_int_loop_revert (*C function*), 746
vrna_cstr_print_eval_mb_loop (*C function*), 746
vrna_cstr_print_eval_mb_loop_revert (*C function*), 746
vrna_cstr_print_eval_sd_corr (*C function*), 745
vrna_cstr_print_fasta_header (*C function*), 744
vrna_cstr_printf (*C function*), 744
vrna_cstr_printf_comment (*C function*), 745
vrna_cstr_printf_structure (*C function*), 745
vrna_cstr_printf_tbody (*C function*), 745
vrna_cstr_printf_thead (*C function*), 745
vrna_cstr_string (*C function*), 744
vrna_cstr_t (*C type*), 742
vrna_cstr_vprintf (*C function*), 744
vrna_cstr_vprintf_comment (*C function*), 745

vrna_cstr_vprintf_structure (*C function*), 745
vrna_cstr_vprintf_tbody (*C function*), 745
vrna_cstr_vprintf_thead (*C function*), 745
vrna_cut_point_insert (*C function*), 805
vrna_cut_point_remove (*C function*), 805
vrna_data_lin_t (*C type*), 748
vrna_data_lin_transform (*C function*), 852
vrna_db_compat_remove (*C function*), 834
vrna_db_flatten (*C function*), 815
vrna_db_flatten_to (*C function*), 816
vrna_db_from_bp_stack (*C function*), 835
vrna_db_from_bps (*C function*), 834
vrna_db_from_plist (*C function*), 816
vrna_db_from_probs (*C function*), 834
vrna_db_from_ptable (*C function*), 816
vrna_db_from_WUSS (*C function*), 818
vrna_db_insert_gq (*C function*), 576
vrna_db_pack (*C function*), 814
vrna_db_pk_remove (*C function*), 817
vrna_db_to_element_string (*C function*), 817
vrna_db_to_tree_string (*C function*), 828
vrna_db_unpack (*C function*), 815
VRNA_DECOMP_EXT_EXT (*C macro*), 580
VRNA_DECOMP_EXT_EXT_EXT (*C macro*), 581
VRNA_DECOMP_EXT_EXT_STEM (*C macro*), 581
VRNA_DECOMP_EXT_EXT_STEM1 (*C macro*), 582
VRNA_DECOMP_EXT_STEM (*C macro*), 581
VRNA_DECOMP_EXT_STEM_EXT (*C macro*), 581
VRNA_DECOMP_EXT_STEM_OUTSIDE (*C macro*), 581
VRNA_DECOMP_EXT_UP (*C macro*), 580
VRNA_DECOMP_ML_COAXIAL (*C macro*), 579
VRNA_DECOMP_ML_COAXIAL_ENC (*C macro*), 580
VRNA_DECOMP_ML_ML (*C macro*), 579
VRNA_DECOMP_ML_ML_ML (*C macro*), 578
VRNA_DECOMP_ML_ML_STEM (*C macro*), 579
VRNA_DECOMP_ML_STEM (*C macro*), 578
VRNA_DECOMP_ML_UP (*C macro*), 579
VRNA_DECOMP_PAIR_HP (*C macro*), 577
VRNA_DECOMP_PAIR_IL (*C macro*), 578
VRNA_DECOMP_PAIR_ML (*C macro*), 578
vrna_dimer_conc_t (*C type*), 491
vrna_dimer_pf_s (*C struct*), 311
vrna_dimer_pf_s.F0AB (*C var*), 311
vrna_dimer_pf_s.FA (*C var*), 311
vrna_dimer_pf_s.FAB (*C var*), 311
vrna_dimer_pf_s.FB (*C var*), 311
vrna_dimer_pf_s.FcAB (*C var*), 311
vrna_dimer_pf_t (*C type*), 492
vrna_dirname (*C function*), 778
vrna_dist_mountain (*C function*), 830
vrna_DNA_complement (*C function*), 799
vrna_dotplot_auxdata_t (*C struct*), 883
vrna_dotplot_auxdata_t.bottom (*C var*), 884
vrna_dotplot_auxdata_t.bottom_title (*C var*), 884
vrna_dotplot_auxdata_t.comment (*C var*), 884
vrna_dotplot_auxdata_t.left (*C var*), 884
vrna_dotplot_auxdata_t.left_title (*C var*), 884
vrna_dotplot_auxdata_t.right (*C var*), 884
vrna_dotplot_auxdata_t.right_title (*C var*), 884
vrna_dotplot_auxdata_t.title (*C var*), 884
vrna_dotplot_auxdata_t.top (*C var*), 884
vrna_dotplot_auxdata_t.top_title (*C var*), 884
vrna_E_consensus_gquad (*C function*), 574
vrna_E_ext_hp_loop (*C function*), 479
vrna_E_ext_int_loop (*C function*), 451
vrna_E_ext_loop_3 (*C function*), 444
vrna_E_ext_loop_5 (*C function*), 444
vrna_E_ext_stem (*C function*), 477
vrna_E_exterior_loop (*C function*), 442
vrna_E_exterior_stem (*C function*), 441
vrna_E_gquad (*C function*), 573
vrna_E_hairpin (*C function*), 444
vrna_E_hp_loop (*C function*), 479
vrna_E_int_loop (*C function*), 451
vrna_E_internal (*C function*), 448
vrna_E_multibranch_stem (*C function*), 453
vrna_E_stack (*C function*), 483
vrna_elem_prob_s (*C struct*), 823
vrna_elem_prob_s.i (*C var*), 824
vrna_elem_prob_s.j (*C var*), 824
vrna_elem_prob_s.p (*C var*), 824
vrna_elem_prob_s.size (*C var*), 824
vrna_elem_prob_s.type (*C var*), 824
vrna_en_bulge (*C function*), 450
vrna_en_internal (*C function*), 449
vrna_en_stack (*C function*), 450
vrna_ensemble_defect (*C function*), 325
vrna_ensemble_defect_pt (*C function*), 324
vrna_enumerate_necklaces (*C function*), 856
vrna_ep_array_t (*C type*), 823
vrna_ep_t (*C type*), 823
vrna_eqm_probabilities (*C function*), 330
vrna_eqm_probs_bp (*C function*), 322
VRNA_EQM_PROBS_BP (*C macro*), 327
vrna_eqm_probs_component_t (*C type*), 330
vrna_eqm_probs_data_t (*C type*), 330
vrna_eqm_probs_f (*C type*), 330
vrna_eqm_probs_hx (*C function*), 323
VRNA_EQM_PROBS_HX (*C macro*), 328
VRNA_EQM_PROBS_MASK (*C macro*), 329
vrna_eqm_probs_opt_t (*C struct*), 331
vrna_eqm_probs_opt_t.hx_length (*C var*), 331
vrna_eqm_probs_opt_t.mode (*C var*), 331
vrna_eqm_probs_opt_t.up_length (*C var*), 331
VRNA_EQM_PROBS_OUTSIDE_PF (*C macro*), 329
VRNA_EQM_PROBS_SPLIT (*C macro*), 328
VRNA_EQM_PROBS_STORE_OUTSIDE (*C macro*), 329
vrna_eqm_probs_up (*C function*), 322
VRNA_EQM_PROBS_UP (*C macro*), 328
vrna_equilibrium_conc (*C function*), 493
vrna_equilibrium_constants (*C function*), 491

vrna_eval_circ_consensus_structure (*C function*), 466
vrna_eval_circ_consensus_structure_v (*C function*), 470
vrna_eval_circ_gquad_consensus_structure (*C function*), 468
vrna_eval_circ_gquad_consensus_structure_v (*C function*), 471
vrna_eval_circ_gquad_structure (*C function*), 462
vrna_eval_circ_gquad_structure_v (*C function*), 465
vrna_eval_circ_structure (*C function*), 461
vrna_eval_circ_structure_v (*C function*), 464
vrna_eval_consensus_structure_pt_simple (*C function*), 474
vrna_eval_consensus_structure_pt_simple_v (*C function*), 475
vrna_eval_consensus_structure_pt_simple_verbose (*C function*), 475
vrna_eval_consensus_structure_simple (*C function*), 466
vrna_eval_consensus_structure_simple_v (*C function*), 469
vrna_eval_consensus_structure_simple_verbose (*C function*), 468
vrna_eval_covar_structure (*C function*), 457
vrna_eval_ext_hp_loop (*C function*), 480
vrna_eval_ext_stem (*C function*), 476
vrna_eval_exterior_stem (*C function*), 442
vrna_eval_gquad_consensus_structure (*C function*), 467
vrna_eval_gquad_consensus_structure_v (*C function*), 471
vrna_eval_gquad_structure (*C function*), 461
vrna_eval_gquad_structure_v (*C function*), 464
vrna_eval_hairpin (*C function*), 445
vrna_eval_hp_loop (*C function*), 479
vrna_eval_int_loop (*C function*), 483
vrna_eval_internal (*C function*), 450
VRNA_EVAL_LOOP_DEFAULT (*C macro*), 475
VRNA_EVAL_LOOP_NO_CONSTRAINTS (*C macro*), 475
VRNA_EVAL_LOOP_NO_HC (*C macro*), 475
VRNA_EVAL_LOOP_NO_SC (*C macro*), 475
vrna_eval_loop_pt (*C function*), 440
vrna_eval_loop_pt_v (*C function*), 440
vrna_eval_move (*C function*), 455
vrna_eval_move_pt (*C function*), 455
vrna_eval_move_pt_simple (*C function*), 456
vrna_eval_move_shift_pt (*C function*), 456
vrna_eval_stack (*C function*), 451
vrna_eval_structure (*C function*), 456
vrna_eval_structure_cstr (*C function*), 459
vrna_eval_structure_pt (*C function*), 459
vrna_eval_structure_pt_simple (*C function*), 472
vrna_eval_structure_pt_simple_v (*C function*), 473
vrna_eval_structure_pt_simple_verbose (*C function*), 473
vrna_eval_structure_pt_v (*C function*), 460
vrna_eval_structure_pt_verbose (*C function*), 459
vrna_eval_structure_simple (*C function*), 461
vrna_eval_structure_simple_v (*C function*), 463
vrna_eval_structure_simple_verbose (*C function*), 463
vrna_eval_structure_v (*C function*), 458
vrna_eval_structure_verbose (*C function*), 458
vrna_exp_E_consensus_gquad (*C function*), 574
vrna_exp_E_ext_fast (*C function*), 441
vrna_exp_E_ext_fast_free (*C function*), 441
vrna_exp_E_ext_fast_init (*C function*), 441
vrna_exp_E_ext_fast_rotate (*C function*), 441
vrna_exp_E_ext_fast_update (*C function*), 441
vrna_exp_E_ext_stem (*C function*), 477
vrna_exp_E_exterior_loop (*C function*), 443
vrna_exp_E_exterior_stem (*C function*), 443
vrna_exp_E_gquad (*C function*), 573
vrna_exp_E_hairpin (*C function*), 446
vrna_exp_E_hp_loop (*C function*), 480
vrna_exp_E_int_loop (*C function*), 451
vrna_exp_E_interior_loop (*C function*), 483
vrna_exp_E_internal (*C function*), 451
vrna_exp_E_m2_fast (*C function*), 453
vrna_exp_E_mb_loop_fast (*C function*), 453
vrna_exp_E_ml_fast (*C function*), 453
vrna_exp_E_ml_fast_free (*C function*), 453
vrna_exp_E_ml_fast_init (*C function*), 453
vrna_exp_E_ml_fast_qqm (*C function*), 453
vrna_exp_E_ml_fast_qqm1 (*C function*), 453
vrna_exp_E_ml_fast_rotate (*C function*), 453
vrna_exp_E_multibranch_stem (*C function*), 454
vrna_exp_eval_hairpin (*C function*), 447
vrna_exp_eval_internal (*C function*), 451
vrna_exp_eval_stack (*C function*), 451
vrna_exp_param_s (*C struct*), 389
vrna_exp_param_s.alpha (*C var*), 392
vrna_exp_param_s.expbulge (*C var*), 389
vrna_exp_param_s.expdangle3 (*C var*), 390
vrna_exp_param_s.expdangle5 (*C var*), 390
vrna_exp_param_s.expDuplexInit (*C var*), 391
vrna_exp_param_s.expgquad (*C var*), 392
vrna_exp_param_s.expgquadLayerMismatch (*C var*), 392
vrna_exp_param_s.exphairpin (*C var*), 389
vrna_exp_param_s.exphex (*C var*), 391
vrna_exp_param_s.expint11 (*C var*), 390
vrna_exp_param_s.expint21 (*C var*), 390
vrna_exp_param_s.expint22 (*C var*), 390
vrna_exp_param_s.expinternal (*C var*), 389
vrna_exp_param_s.expmismatch1nI (*C var*), 389
vrna_exp_param_s.expmismatch23I (*C var*), 389
vrna_exp_param_s.expmismatchExt (*C var*), 389
vrna_exp_param_s.expmismatchH (*C var*), 390
vrna_exp_param_s.expmismatchI (*C var*), 389
vrna_exp_param_s.expmismatchM (*C var*), 390

vrna_exp_param_s.expMLbase (*C var*), 390
vrna_exp_param_s.expMLclosing (*C var*), 390
vrna_exp_param_s.expMLintern (*C var*), 390
vrna_exp_param_s.expMultipleCA (*C var*), 391
vrna_exp_param_s.expMultipleCB (*C var*), 391
vrna_exp_param_s.expninio (*C var*), 390
vrna_exp_param_s.expSaltLoop (*C var*), 393
vrna_exp_param_s.expSaltStack (*C var*), 392
vrna_exp_param_s.expstack (*C var*), 389
vrna_exp_param_s.expTermAU (*C var*), 391
vrna_exp_param_s.exptetra (*C var*), 391
vrna_exp_param_s.exptri (*C var*), 391
vrna_exp_param_s.expTriloop (*C var*), 391
vrna_exp_param_s.expTripleC (*C var*), 391
vrna_exp_param_s.gquadLayerMismatchMax (*C var*), 392
vrna_exp_param_s.Hexaloops (*C var*), 391
vrna_exp_param_s.id (*C var*), 389
vrna_exp_param_s.kT (*C var*), 392
vrna_exp_param_s.lxc (*C var*), 390
vrna_exp_param_s.model_details (*C var*), 392
vrna_exp_param_s.param_file (*C var*), 392
vrna_exp_param_s.pf_scale (*C var*), 392
vrna_exp_param_s.SaltDPXInit (*C var*), 393
vrna_exp_param_s.SaltLoopDbl (*C var*), 393
vrna_exp_param_s.SaltMLbase (*C var*), 393
vrna_exp_param_s.SaltMLclosing (*C var*), 393
vrna_exp_param_s.SaltMLintern (*C var*), 393
vrna_exp_param_s.temperature (*C var*), 392
vrna_exp_param_s.Tetraloops (*C var*), 391
vrna_exp_param_s.Triloops (*C var*), 391
vrna_exp_param_t (*C type*), 376
vrna_exp_params (*C function*), 377
vrna_exp_params_comparative (*C function*), 378
vrna_exp_params_copy (*C function*), 378
vrna_exp_params_rescale (*C function*), 379
vrna_exp_params_reset (*C function*), 381
vrna_exp_params_subst (*C function*), 379
VRNA_EXT_LOOP (*C macro*), 315
vrna_extract_record_rest_constraint (*C function*), 768
vrna_extract_record_rest_structure (*C function*), 766
vrna_fc_s (*C struct*), 655
vrna_fc_s.alignment (*C var*), 657
vrna_fc_s.aux_grammar (*C var*), 658
vrna_fc_s.auxdata (*C var*), 658
vrna_fc_s.bpdist (*C var*), 659
vrna_fc_s.cutpoint (*C var*), 656
vrna_fc_s.domains_struc (*C var*), 658
vrna_fc_s.domains_up (*C var*), 658
vrna_fc_s.exp_matrices (*C var*), 657
vrna_fc_s.exp_params (*C var*), 657
vrna_fc_s.free_auxdata (*C var*), 658
vrna_fc_s.hc (*C var*), 657
vrna_fc_s.iindx (*C var*), 657
vrna_fc_s.jindx (*C var*), 657
vrna_fc_s.length (*C var*), 656
vrna_fc_s.matrices (*C var*), 657
vrna_fc_s.maxD1 (*C var*), 659
vrna_fc_s.maxD2 (*C var*), 659
vrna_fc_s.mm1 (*C var*), 659
vrna_fc_s.mm2 (*C var*), 659
vrna_fc_s.nucleotides (*C var*), 657
vrna_fc_s.params (*C var*), 657
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.a2s (*C var*), 664
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.cons_s (*C var*), 663
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.encodi (*C var*), 661
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.encodi (*C var*), 661
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.n_seq (*C var*), 663
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.oldAli (*C var*), 665
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.pscore (*C var*), 664
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.pscore (*C var*), 664
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.pscore (*C var*), 665
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.ptype (*C var*), 661
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.ptype_ (*C var*), 662
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.S (*C var*), 663
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.S3 (*C var*), 664
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.S5 (*C var*), 663
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.S_cons (*C var*), 663
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.sc (*C var*), 662
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.scs (*C var*), 665
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.sequen (*C var*), 660
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.sequen (*C var*), 661
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.sequen (*C var*), 661
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.sequen (*C var*), 662
vrna_fc_s.PhonyNameDueToError.PhonyNameDueToError.Ss (*C var*), 664
vrna_fc_s.PhonyNameDueToError.[anonymous] (*C var*), 660
vrna_fc_s.ptype_local (*C var*), 660
vrna_fc_s.reference_pt1 (*C var*), 659
vrna_fc_s.reference_pt2 (*C var*), 659
vrna_fc_s.referenceBPs1 (*C var*), 659
vrna_fc_s.referenceBPs2 (*C var*), 659

vrna_fc_s.stat_cb (*C var*), 658
vrna_fc_s.strand_end (*C var*), 656
vrna_fc_s.strand_number (*C var*), 656
vrna_fc_s.strand_order (*C var*), 656
vrna_fc_s.strand_order_uniq (*C var*), 656
vrna_fc_s.strand_start (*C var*), 656
vrna_fc_s.strands (*C var*), 657
vrna_fc_s.type (*C var*), 656
vrna_fc_s.window_size (*C var*), 660
vrna_fc_s.[anonymous] (*C var*), 660
vrna_fc_type_e (*C enum*), 651
vrna_fc_type_e.VRNA_FC_TYPE_COMPARATIVE (*C enumerator*), 651
vrna_fc_type_e.VRNA_FC_TYPE_SINGLE (*C enumerator*), 651
vrna_file_bpseq (*C function*), 764
vrna_file_commands_apply (*C function*), 777
vrna_file_commands_read (*C function*), 776
vrna_file_connect (*C function*), 764
vrna_file_connect_read_record (*C function*), 768
vrna_file_copy (*C function*), 778
vrna_file_doubles_read (*C function*), 767
vrna_file_exists (*C function*), 779
vrna_file_fasta_read_record (*C function*), 765
VRNA_FILE_FORMAT_EPS (*C macro*), 887
VRNA_FILE_FORMAT_GML (*C macro*), 887
VRNA_FILE_FORMAT_MSA_APPEND (*C macro*), 770
VRNA_FILE_FORMAT_MSA_CLUSTAL (*C macro*), 769
VRNA_FILE_FORMAT_MSA_DEFAULT (*C macro*), 770
VRNA_FILE_FORMAT_MSA_FASTA (*C macro*), 769
VRNA_FILE_FORMAT_MSA_MAF (*C macro*), 769
VRNA_FILE_FORMAT_MSA_MIS (*C macro*), 770
VRNA_FILE_FORMAT_MSA_NOCHECK (*C macro*), 770
VRNA_FILE_FORMAT_MSA_QUIET (*C macro*), 771
VRNA_FILE_FORMAT_MSA_SILENT (*C macro*), 771
VRNA_FILE_FORMAT_MSA_STOCKHOLM (*C macro*), 769
VRNA_FILE_FORMAT_MSA_UNKNOWN (*C macro*), 770
VRNA_FILE_FORMAT_PLOT_DEFAULT (*C macro*), 887
VRNA_FILE_FORMAT_SSV (*C macro*), 887
VRNA_FILE_FORMAT_SVG (*C macro*), 887
VRNA_FILE_FORMAT_XRNA (*C macro*), 887
vrna_file_helixlist (*C function*), 764
vrna_file_json (*C function*), 765
vrna_file_msa_detect_format (*C function*), 773
vrna_file_msa_read (*C function*), 771
vrna_file_msa_read_record (*C function*), 772
vrna_file_msa_write (*C function*), 774
vrna_file_PS_aln (*C function*), 885
vrna_file_PS_aln_opt (*C function*), 886
vrna_file_PS_aln_slice (*C function*), 885
vrna_file_PS_rnaplot (*C function*), 888
vrna_file_PS_rnaplot_a (*C function*), 888
vrna_file_PS_rnaplot_layout (*C function*), 888
vrna_file_RNAstrand_db_read_record (*C function*), 768
vrna_file_SHAPE_read (*C function*), 767
vrna_filename_sanitize (*C function*), 778
vrna_fold (*C function*), 288
vrna_fold_compound (*C function*), 651
vrna_fold_compound_add_auxdata (*C function*), 654
vrna_fold_compound_add_callback (*C function*), 654
vrna_fold_compound_comparative (*C function*), 652
vrna_fold_compound_comparative2 (*C function*), 653
vrna_fold_compound_free (*C function*), 653
vrna_fold_compound_prepare (*C function*), 653
vrna_fold_compound_t (*C type*), 649
vrna_fold_compound_TwoD (*C function*), 653
vrna_get_gquad_pattern_pf (*C function*), 575
vrna_get_ptype (*C function*), 794
vrna_get_ptype_md (*C function*), 794
vrna_get_ptype_window (*C function*), 795
vrna_gq_int_loop_pf (*C function*), 574
vrna_gq_int_loop_subopt (*C function*), 574
vrna_gq_parse (*C function*), 575
vrna_gq_pos_pf (*C function*), 575
VRNA_GQUAD_DB_SYMBOL (*C macro*), 814
VRNA_GQUAD_DB_SYMBOL_END (*C macro*), 814
VRNA_GQUAD_MAX_BOX_SIZE (*C macro*), 376
VRNA_GQUAD_MAX_LINKER_LENGTH (*C macro*), 375
VRNA_GQUAD_MAX_STACK_SIZE (*C macro*), 375
VRNA_GQUAD_MIN_BOX_SIZE (*C macro*), 375
VRNA_GQUAD_MIN_LINKER_LENGTH (*C macro*), 375
VRNA_GQUAD_MIN_STACK_SIZE (*C macro*), 375
vrna_gquad_mx_local_update (*C function*), 575
vrna_gr_add_aux (*C function*), 635
vrna_gr_add_aux_c (*C function*), 632
vrna_gr_add_aux_exp (*C function*), 640
vrna_gr_add_aux_exp_c (*C function*), 637
vrna_gr_add_aux_exp_f (*C function*), 636
vrna_gr_add_aux_exp_m (*C function*), 638
vrna_gr_add_aux_exp_m1 (*C function*), 638
vrna_gr_add_aux_exp_m2 (*C function*), 639
vrna_gr_add_aux_f (*C function*), 631
vrna_gr_add_aux_m (*C function*), 633
vrna_gr_add_aux_m1 (*C function*), 634
vrna_gr_add_aux_m2 (*C function*), 634
vrna_gr_add_status (*C function*), 630
vrna_gr_aux_t (*C type*), 627
vrna_gr_inside_exp_f (*C type*), 629
vrna_gr_inside_f (*C type*), 628
vrna_gr_outside_exp_f (*C type*), 630
vrna_gr_outside_f (*C type*), 628
vrna_gr_prepare (*C function*), 630
vrna_gr_reset (*C function*), 631
vrna_gr_serialize_bp_f (*C type*), 627
vrna_gr_set_serialize_bp (*C function*), 630
vrna_hamming_distance (*C function*), 804
vrna_hamming_distance_bound (*C function*), 804
vrna_hash_table_t (*C type*), 727
vrna_hc_add_bp (*C function*), 590
vrna_hc_add_bp_nonspecific (*C function*), 591
vrna_hc_add_bp_strand (*C function*), 591

vrna_hc_add_data (*C function*), 592
vrna_hc_add_from_db (*C function*), 592
vrna_hc_add_up (*C function*), 589
vrna_hc_add_up_batch (*C function*), 590
vrna_hc_add_up_strand (*C function*), 589
vrna_hc_eval_f (*C type*), 587
vrna_hc_free (*C function*), 592
vrna_hc_init (*C function*), 589
vrna_hc_s (*C struct*), 593
vrna_hc_s.data (*C var*), 594
vrna_hc_s.depot (*C var*), 595
vrna_hc_s.eval_ext (*C var*), 595
vrna_hc_s.eval_hp (*C var*), 595
vrna_hc_s.eval_int (*C var*), 595
vrna_hc_s.eval_mb (*C var*), 595
vrna_hc_s.f (*C var*), 594
vrna_hc_s.free_data (*C var*), 595
vrna_hc_s.matrix_local (*C var*), 594
vrna_hc_s.mx (*C var*), 594
vrna_hc_s.n (*C var*), 593
vrna_hc_s.sn (*C var*), 594
vrna_hc_s.state (*C var*), 594
vrna_hc_s.strands (*C var*), 594
vrna_hc_s.type (*C var*), 593
vrna_hc_s.up_ext (*C var*), 594
vrna_hc_s.up_hp (*C var*), 594
vrna_hc_s.up_int (*C var*), 594
vrna_hc_s.up_ml (*C var*), 594
vrna_hc_t (*C type*), 587
vrna_hc_up_s (*C struct*), 595
vrna_hc_up_s.options (*C var*), 595
vrna_hc_up_s.position (*C var*), 595
vrna_hc_up_s.strand (*C var*), 595
vrna_hc_up_t (*C type*), 587
vrna_heap_cmp_f (*C type*), 733
vrna_heap_free (*C function*), 735
vrna_heap_get_pos_f (*C type*), 734
vrna_heap_init (*C function*), 734
vrna_heap_insert (*C function*), 736
vrna_heap_pop (*C function*), 736
vrna_heap_remove (*C function*), 737
vrna_heap_set_pos_f (*C type*), 734
vrna_heap_size (*C function*), 735
vrna_heap_t (*C type*), 733
vrna_heap_top (*C function*), 736
vrna_heap_update (*C function*), 737
vrna_heat_capacity (*C function*), 319
vrna_heat_capacity_callback (*C type*), 330
vrna_heat_capacity_cb (*C function*), 320
vrna_heat_capacity_f (*C type*), 329
vrna_heat_capacity_s (*C struct*), 330
vrna_heat_capacity_s.heat_capacity (*C var*), 331
vrna_heat_capacity_s.temperature (*C var*), 330
vrna_heat_capacity_simple (*C function*), 321
vrna_heat_capacity_t (*C type*), 330
VRNA_HP_LOOP (*C macro*), 316
vrna_ht_clear (*C function*), 731
vrna_ht_cmp_f (*C type*), 727
vrna_ht_collisions (*C function*), 729
vrna_ht_db_comp (*C function*), 731
vrna_ht_db_free_entry (*C function*), 732
vrna_ht_db_hash_func (*C function*), 732
vrna_ht_entry_db_t (*C struct*), 732
vrna_ht_entry_db_t.energy (*C var*), 733
vrna_ht_entry_db_t.structure (*C var*), 733
vrna_ht_free (*C function*), 731
vrna_ht_free_f (*C type*), 728
vrna_ht_get (*C function*), 729
vrna_ht_hashfunc_f (*C type*), 728
vrna_ht_init (*C function*), 728
vrna_ht_insert (*C function*), 730
vrna_ht_remove (*C function*), 730
vrna_ht_size (*C function*), 729
vrna_hx_from_ptable (*C function*), 825
vrna_hx_merge (*C function*), 826
vrna_hx_s (*C struct*), 826
vrna_hx_s.end (*C var*), 826
vrna_hx_s.length (*C var*), 826
vrna_hx_s.start (*C var*), 826
vrna_hx_s.up3 (*C var*), 826
vrna_hx_s.up5 (*C var*), 826
vrna_hx_t (*C type*), 825
vrna_idx_col_wise (*C function*), 646
vrna_idx_row_wise (*C function*), 645
vrna_init_rand (*C function*), 643
vrna_init_rand_seed (*C function*), 644
VRNA_INPUT_BLANK_LINE (*C macro*), 642
VRNA_INPUT_COMMENT (*C macro*), 643
VRNA_INPUT_CONSTRAINT (*C macro*), 642
VRNA_INPUT_ERROR (*C macro*), 641
VRNA_INPUT_FASTA_HEADER (*C macro*), 642
VRNA_INPUT_MISC (*C macro*), 641
VRNA_INPUT_NO_REST (*C macro*), 642
VRNA_INPUT_NO_SPAN (*C macro*), 642
VRNA_INPUT_NO_TRUNCATION (*C macro*), 642
VRNA_INPUT_NOSKIP_BLANK_LINES (*C macro*), 642
VRNA_INPUT_NOSKIP_COMMENTS (*C macro*), 642
VRNA_INPUT_QUIT (*C macro*), 641
VRNA_INPUT_SEQUENCE (*C macro*), 642
VRNA_INPUT_VERBOSE (*C macro*), 764
VRNA_INT_LOOP (*C macro*), 316
vrna_int_urn (*C function*), 644
vrna_inverse_linearbp_opt_t (*C struct*), 497
vrna_inverse_linearbp_opt_t.biseparable (*C var*), 497
vrna_inverse_linearbp_opt_t.GCweight (*C var*), 497
vrna_inverse_linearbp_opt_t.maxmodulo (*C var*), 497
vrna_inverse_seed_linearbp (*C function*), 496
vrna_letter_structure (*C function*), 835
vrna_Lfold (*C function*), 293
vrna_Lfold_cb (*C function*), 293
vrna_log (*C function*), 787
vrna_log_cb_add (*C function*), 789

vrna_log_cb_f (*C type*), 786
vrna_log_cb_num (*C function*), 790
vrna_log_cb_remove (*C function*), 790
vrna_log_critical (*C macro*), 785
vrna_log_debug (*C macro*), 784
vrna_log_error (*C macro*), 785
vrna_log_event_s (*C struct*), 791
vrna_log_event_s.file_name (*C var*), 792
vrna_log_event_s.format_string (*C var*), 791
vrna_log_event_s.function (*C var*), 792
vrna_log_event_s.level (*C var*), 791
vrna_log_event_s.line_number (*C var*), 791
vrna_log_event_s.params (*C var*), 791
vrna_log_event_t (*C type*), 786
vrna_log_fp (*C function*), 789
vrna_log_fp_set (*C function*), 789
vrna_log_info (*C macro*), 785
vrna_log_level (*C function*), 788
VRNA_LOG_LEVEL_DEFAULT (*C macro*), 783
vrna_log_level_set (*C function*), 788
vrna_log_levels_e (*C enum*), 787
vrna_log_levels_e.VRNA_LOG_LEVEL_CRITICAL (*C enumerator*), 787
vrna_log_levels_e.VRNA_LOG_LEVEL_DEBUG (*C enumerator*), 787
vrna_log_levels_e.VRNA_LOG_LEVEL_ERROR (*C enumerator*), 787
vrna_log_levels_e.VRNA_LOG_LEVEL_INFO (*C enumerator*), 787
vrna_log_levels_e.VRNA_LOG_LEVEL_SILENT (*C enumerator*), 787
vrna_log_levels_e.VRNA_LOG_LEVEL_UNKNOWN (*C enumerator*), 787
vrna_log_levels_e.VRNA_LOG_LEVEL_WARNING (*C enumerator*), 787
vrna_log_lock_f (*C type*), 786
vrna_log_lock_set (*C function*), 791
VRNA_LOG_OPTION_DEFAULT (*C macro*), 784
VRNA_LOG_OPTION_QUIET (*C macro*), 783
VRNA_LOG_OPTION_TRACE_CALL (*C macro*), 783
VRNA_LOG_OPTION_TRACE_FUNC (*C macro*), 783
VRNA_LOG_OPTION_TRACE_TIME (*C macro*), 784
vrna_log_options (*C function*), 789
vrna_log_options_set (*C function*), 789
vrna_log_reset (*C function*), 791
vrna_log_warning (*C macro*), 785
vrna_logdata_free_f (*C type*), 786
VRNA_LOOP_TYPE_ANY (*C macro*), 327
VRNA_LOOP_TYPE_EXT (*C macro*), 327
VRNA_LOOP_TYPE_HP (*C macro*), 327
VRNA_LOOP_TYPE_INT (*C macro*), 327
VRNA_LOOP_TYPE_MB (*C macro*), 327
vrna_loopidx_from_ptable (*C function*), 835
vrna_loopidx_update (*C function*), 517
VRNA_MATH_FUN_BIN_OPTION_DEFAULT (*C macro*), 836
VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_LOWERBOUND (*C macro*), 836
VRNA_MATH_FUN_BIN_OPTION_MAP_OUTOF_UPPERBOUND (*C macro*), 836
VRNA_MATH_FUN_BIN_OPTION_PROJECT (*C macro*), 836
vrna_math_fun_dbl_bin (*C function*), 840
vrna_math_fun_dbl_bin_opt (*C function*), 840
vrna_math_fun_dbl_chain (*C function*), 849
vrna_math_fun_dbl_chain_append (*C function*), 849
vrna_math_fun_dbl_chain_at (*C function*), 849
vrna_math_fun_dbl_chain_size (*C function*), 849
vrna_math_fun_dbl_f (*C type*), 839
vrna_math_fun_dbl_gaussian (*C function*), 848
vrna_math_fun_dbl_gaussian_opt (*C function*), 848
vrna_math_fun_dbl_kde (*C function*), 846
vrna_math_fun_dbl_kde_opt (*C function*), 847
vrna_math_fun_dbl_linear (*C function*), 841
vrna_math_fun_dbl_linear_opt (*C function*), 842
vrna_math_fun_dbl_log (*C function*), 843
vrna_math_fun_dbl_log_opt (*C function*), 843
vrna_math_fun_dbl_logistic (*C function*), 844
vrna_math_fun_dbl_logistic_opt (*C function*), 845
vrna_math_fun_dbl_logit (*C function*), 846
vrna_math_fun_dbl_logit_opt (*C function*), 845
vrna_math_fun_dbl_opt_free_f (*C type*), 839
vrna_math_fun_dbl_opt_t (*C type*), 839
VRNA_MATH_FUN_GAUSSIAN_OPTION_DEFAULT (*C macro*), 839
VRNA_MATH_FUN_KDE_OPTION_DEFAULT (*C macro*), 838
VRNA_MATH_FUN_LINEAR_OPTION_DEFAULT (*C macro*), 837
VRNA_MATH_FUN_LINEAR_OPTION_LOG (*C macro*), 837
VRNA_MATH_FUN_LOG_OPTION_DEFAULT (*C macro*), 837
VRNA_MATH_FUN_LOG_OPTION_NONDEFAULT_BASE (*C macro*), 837
VRNA_MATH_FUN_LOGISTIC_OPTION_DEFAULT (*C macro*), 837
VRNA_MATH_FUN_LOGIT_BOUNDS_DEFAULT (*C macro*), 838
VRNA_MATH_FUN_LOGIT_OOB_VALUES_DEFAULT (*C macro*), 838
VRNA_MATH_FUN_LOGIT_OPTION_BOUNDS_INCLUSIVE (*C macro*), 838
VRNA_MATH_FUN_LOGIT_OPTION_BOUNDS_INCLUSIVE_LOWER (*C macro*), 838
VRNA_MATH_FUN_LOGIT_OPTION_BOUNDS_INCLUSIVE_UPPER (*C macro*), 838
VRNA_MATH_FUN_LOGIT_OPTION_BOUNDS_OPEN (*C macro*), 838
VRNA_MATH_FUN_LOGIT_OPTION_DEFAULT (*C macro*), 838
VRNA_MB_LOOP (*C macro*), 316
vrna_md_copy (*C function*), 414

vrna_md_defaults_backbone_length (*C function*), 431
vrna_md_defaults_backbone_length_get (*C function*), 431
vrna_md_defaults_backtrack (*C function*), 423
vrna_md_defaults_backtrack_get (*C function*), 423
vrna_md_defaults_backtrack_type (*C function*), 423
vrna_md_defaults_backtrack_type_get (*C function*), 424
vrna_md_defaults_betaScale (*C function*), 416
vrna_md_defaults_betaScale_get (*C function*), 416
vrna_md_defaults_circ (*C function*), 420
vrna_md_defaults_circ_alpha0 (*C function*), 431
vrna_md_defaults_circ_alpha0_get (*C function*), 431
vrna_md_defaults_circ_get (*C function*), 420
vrna_md_defaults_circ_penalty (*C function*), 420
vrna_md_defaults_circ_penalty_get (*C function*), 421
vrna_md_defaults_compute_bpp (*C function*), 424
vrna_md_defaults_compute_bpp_get (*C function*), 424
vrna_md_defaults_cv_fact (*C function*), 428
vrna_md_defaults_cv_fact_get (*C function*), 428
vrna_md_defaults_dangles (*C function*), 416
vrna_md_defaults_dangles_get (*C function*), 417
vrna_md_defaults_energy_set (*C function*), 422
vrna_md_defaults_energy_set_get (*C function*), 422
vrna_md_defaults_gquad (*C function*), 421
vrna_md_defaults_gquad_get (*C function*), 421
vrna_md_defaults_helical_rise (*C function*), 431
vrna_md_defaults_helical_rise_get (*C function*), 431
vrna_md_defaults_logML (*C function*), 419
vrna_md_defaults_logML_get (*C function*), 419
vrna_md_defaults_max_bp_span (*C function*), 425
vrna_md_defaults_max_bp_span_get (*C function*), 425
vrna_md_defaults_min_loop_size (*C function*), 425
vrna_md_defaults_min_loop_size_get (*C function*), 425
vrna_md_defaults_nc_fact (*C function*), 428
vrna_md_defaults_nc_fact_get (*C function*), 428
vrna_md_defaults_noGU (*C function*), 418
vrna_md_defaults_noGU_get (*C function*), 418
vrna_md_defaults_noGUclosure (*C function*), 419
vrna_md_defaults_noGUclosure_get (*C function*), 419
vrna_md_defaults_noLP (*C function*), 418
vrna_md_defaults_noLP_get (*C function*), 418
vrna_md_defaults_oldAliEn (*C function*), 426
vrna_md_defaults_oldAliEn_get (*C function*), 427
vrna_md_defaults_pf_smooth (*C function*), 416
vrna_md_defaults_pf_smooth_get (*C function*), 416
vrna_md_defaults_reset (*C function*), 414
vrna_md_defaults_ribo (*C function*), 427
vrna_md_defaults_ribo_get (*C function*), 427
vrna_md_defaults_salt (*C function*), 429
vrna_md_defaults_salt_get (*C function*), 429
vrna_md_defaults_saltDPXInit (*C function*), 430
vrna_md_defaults_saltDPXInit_get (*C function*), 430
vrna_md_defaults_saltDPXInitFact (*C function*), 430
vrna_md_defaults_saltDPXInitFact_get (*C function*), 431
vrna_md_defaults_saltMLLower (*C function*), 429
vrna_md_defaults_saltMLLower_get (*C function*), 430
vrna_md_defaults_saltMLUpper (*C function*), 430
vrna_md_defaults_saltMLUpper_get (*C function*), 430
vrna_md_defaults_sfact (*C function*), 429
vrna_md_defaults_sfact_get (*C function*), 429
vrna_md_defaults_special_hp (*C function*), 417
vrna_md_defaults_special_hp_get (*C function*), 417
vrna_md_defaults_temperature (*C function*), 415
vrna_md_defaults_temperature_get (*C function*), 415
vrna_md_defaults_uniq_ML (*C function*), 421
vrna_md_defaults_uniq_ML_get (*C function*), 422
vrna_md_defaults_window_size (*C function*), 426
vrna_md_defaults_window_size_get (*C function*), 426
vrna_md_option_string (*C function*), 414
vrna_md_s (*C struct*), 435
vrna_md_s.alias (*C var*), 439
vrna_md_s.backbone_length (*C var*), 440
vrna_md_s.backtrack (*C var*), 437
vrna_md_s.backtrack_type (*C var*), 437
vrna_md_s.betaScale (*C var*), 436
vrna_md_s.circ (*C var*), 437
vrna_md_s.circ_alpha0 (*C var*), 440
vrna_md_s.circ_penalty (*C var*), 437
vrna_md_s.compute_bpp (*C var*), 437
vrna_md_s.cv_fact (*C var*), 438
vrna_md_s.dangles (*C var*), 436
vrna_md_s.energy_set (*C var*), 437
vrna_md_s.gquad (*C var*), 437
vrna_md_s.helical_rise (*C var*), 439
vrna_md_s.logML (*C var*), 437
vrna_md_s.max_bp_span (*C var*), 438
vrna_md_s.min_loop_size (*C var*), 438
vrna_md_s.nc_fact (*C var*), 438
vrna_md_s.noGU (*C var*), 436
vrna_md_s.noGUclosure (*C var*), 437
vrna_md_s.noLP (*C var*), 436
vrna_md_s.nonstandards (*C var*), 438
vrna_md_s.oldAliEn (*C var*), 438

vrna_md_s.pair (*C var*), 439
vrna_md_s.pair_dist (*C var*), 439
vrna_md_s.pf_smooth (*C var*), 436
vrna_md_s.ribo (*C var*), 438
vrna_md_s.rtype (*C var*), 439
vrna_md_s.salt (*C var*), 439
vrna_md_s.saltDPXInit (*C var*), 439
vrna_md_s.saltDPXInitFact (*C var*), 439
vrna_md_s.saltMLLower (*C var*), 439
vrna_md_s.saltMLUpper (*C var*), 439
vrna_md_s.sfact (*C var*), 438
vrna_md_s.special_hp (*C var*), 436
vrna_md_s.temperature (*C var*), 436
vrna_md_s.uniq_ML (*C var*), 437
vrna_md_s.window_size (*C var*), 438
vrna_md_set_default (*C function*), 414
vrna_md_set_nonstandards (*C function*), 414
vrna_md_t (*C type*), 413
vrna_md_update (*C function*), 414
vrna_MEA (*C function*), 372
vrna_MEA_from_plist (*C function*), 373
vrna_mean_bp_distance (*C function*), 324
vrna_mean_bp_distance_pr (*C function*), 323
VRNA_MEASURE_SHANNON_ENTROPY (*C macro*), 806
vrna_message_constraint_options (*C function*), 582
vrna_message_constraint_options_all (*C function*), 583
vrna_message_error (*C function*), 780
vrna_message_info (*C function*), 781
vrna_message_input_msa (*C function*), 782
vrna_message_input_seq (*C function*), 782
vrna_message_input_seq_simple (*C function*), 782
vrna_message_verror (*C function*), 780
vrna_message_vinfo (*C function*), 782
vrna_message_vwarning (*C function*), 781
vrna_message_warning (*C function*), 781
vrna_mfe (*C function*), 287
vrna_mfe_dimer (*C function*), 288
vrna_mfe_exterior_f3 (*C function*), 444
vrna_mfe_exterior_f5 (*C function*), 444
vrna_mfe_gquad_internal_loop (*C function*), 574
vrna_mfe_gquad_mx (*C function*), 574
vrna_mfe_internal (*C function*), 451
vrna_mfe_internal_ext (*C function*), 451
vrna_mfe_multibranch_fast_free (*C function*), 452
vrna_mfe_multibranch_fast_init (*C function*), 452
vrna_mfe_multibranch_fast_rotate (*C function*), 452
vrna_mfe_multibranch_loop_fast (*C function*), 452
vrna_mfe_multibranch_loop_stack (*C function*), 452
vrna_mfe_multibranch_m1 (*C function*), 452
vrna_mfe_multibranch_m2_fast (*C function*), 452
vrna_mfe_multibranch_stems_fast (*C function*), 452
vrna_mfe_TwoD (*C function*), 498
vrna_mfe_window (*C function*), 292
vrna_mfe_window_callback (*C type*), 295
vrna_mfe_window_cb (*C function*), 292
vrna_mfe_window_f (*C type*), 294
VRNA_MINIMIZER_CONJUGATE_FR (*C macro*), 555
VRNA_MINIMIZER_CONJUGATE_PR (*C macro*), 556
VRNA_MINIMIZER_DEFAULT (*C macro*), 555
VRNA_MINIMIZER_STEEPEST_DESCENT (*C macro*), 556
VRNA_MINIMIZER_VECTOR_BFGS (*C macro*), 556
VRNA_MINIMIZER_VECTOR_BFGS2 (*C macro*), 556
vrna_mkdir_p (*C function*), 778
VRNA_MODEL_BACKBONE_LENGTH_DNA (*C macro*), 413
VRNA_MODEL_BACKBONE_LENGTH_RNA (*C macro*), 413
VRNA_MODEL_DEFAULT_ALI_CV_FACT (*C macro*), 411
VRNA_MODEL_DEFAULT_ALI_NC_FACT (*C macro*), 411
VRNA_MODEL_DEFAULT_ALI_OLD_EN (*C macro*), 411
VRNA_MODEL_DEFAULT_ALI_RIBO (*C macro*), 411
VRNA_MODEL_DEFAULT_BACKBONE_LENGTH (*C macro*), 413
VRNA_MODEL_DEFAULT_BACKTRACK (*C macro*), 409
VRNA_MODEL_DEFAULT_BACKTRACK_TYPE (*C macro*), 410
VRNA_MODEL_DEFAULT_BETA_SCALE (*C macro*), 407
VRNA_MODEL_DEFAULT_CIRC (*C macro*), 409
VRNA_MODEL_DEFAULT_CIRC_ALPHA0 (*C macro*), 413
VRNA_MODEL_DEFAULT_CIRC_PENALTY (*C macro*), 413
VRNA_MODEL_DEFAULT_COMPUTE_BPP (*C macro*), 410
VRNA_MODEL_DEFAULT_DANGLES (*C macro*), 407
VRNA_MODEL_DEFAULT_ENERGY_SET (*C macro*), 409
VRNA_MODEL_DEFAULT_GQUAD (*C macro*), 409
VRNA_MODEL_DEFAULT_HELICAL_RISE (*C macro*), 413
VRNA_MODEL_DEFAULT_LOG_ML (*C macro*), 411
VRNA_MODEL_DEFAULT_MAX_BP_SPAN (*C macro*), 410
VRNA_MODEL_DEFAULT_NO_GU (*C macro*), 408
VRNA_MODEL_DEFAULT_NO_GU_CLOSURE (*C macro*), 408
VRNA_MODEL_DEFAULT_NO_LP (*C macro*), 408
VRNA_MODEL_DEFAULT_PF_SCALE (*C macro*), 407
VRNA_MODEL_DEFAULT_PF_SMOOTH (*C macro*), 412
VRNA_MODEL_DEFAULT_SALT (*C macro*), 412
VRNA_MODEL_DEFAULT_SALT_DPXINIT (*C macro*), 412
VRNA_MODEL_DEFAULT_SALT_DPXINIT_FACT (*C macro*), 412
VRNA_MODEL_DEFAULT_SALT_MLLOWER (*C macro*), 412
VRNA_MODEL_DEFAULT_SALT_MLUPPER (*C macro*), 412
VRNA_MODEL_DEFAULT_SPECIAL_HP (*C macro*), 408
VRNA_MODEL_DEFAULT_TEMPERATURE (*C macro*), 407
VRNA_MODEL_DEFAULT_UNIQ_ML (*C macro*), 409
VRNA_MODEL_DEFAULT_WINDOW_SIZE (*C macro*), 410

VRNA_MODEL_HELICAL_RISE_DNA (*C macro*), 413
VRNA_MODEL_HELICAL_RISE_RNA (*C macro*), 412
VRNA_MODEL_SALT_DPXINIT_FACT_DNA (*C macro*), 412
VRNA_MODEL_SALT_DPXINIT_FACT_RNA (*C macro*), 412
vrna_move_apply (*C function*), 515
vrna_move_apply_db (*C function*), 515
vrna_move_compare (*C function*), 516
vrna_move_init (*C function*), 515
vrna_move_is_insertion (*C function*), 516
vrna_move_is_removal (*C function*), 516
vrna_move_is_shift (*C function*), 516
vrna_move_list_free (*C function*), 515
vrna_move_neighbor_diff (*C function*), 519
vrna_move_neighbor_diff_cb (*C function*), 518
VRNA_MOVE_NO_APPLY (*C macro*), 513
vrna_move_s (*C struct*), 519
vrna_move_s.next (*C var*), 520
vrna_move_s.pos_3 (*C var*), 520
vrna_move_s.pos_5 (*C var*), 520
vrna_move_t (*C type*), 514
vrna_move_update_f (*C type*), 514
VRNA_MOVESET_DEFAULT (*C macro*), 513
VRNA_MOVESET_DELETION (*C macro*), 512
VRNA_MOVESET_INSERTION (*C macro*), 512
VRNA_MOVESET_NO_LP (*C macro*), 513
VRNA_MOVESET_SHIFT (*C macro*), 513
vrna_msa_add (*C function*), 795
vrna_msa_t (*C type*), 792
vrna_multimer_pf_s (*C struct*), 311
vrna_multimer_pf_s.F_connected (*C var*), 312
vrna_multimer_pf_s.F_monomers (*C var*), 312
vrna_multimer_pf_s.num_monomers (*C var*), 312
vrna_mx_add (*C function*), 677
VRNA_MX_FLAG_C (*C macro*), 676
VRNA_MX_FLAG_F3 (*C macro*), 676
VRNA_MX_FLAG_F5 (*C macro*), 676
VRNA_MX_FLAG_G (*C macro*), 676
VRNA_MX_FLAG_M (*C macro*), 676
VRNA_MX_FLAG_M1 (*C macro*), 676
VRNA_MX_FLAG_M2 (*C macro*), 676
VRNA_MX_FLAG_MAX (*C macro*), 676
VRNA_MX_FLAG_MS3 (*C macro*), 676
VRNA_MX_FLAG_MS5 (*C macro*), 676
vrna_mx_mfe_add (*C function*), 678
vrna_mx_mfe_aux_ml_t (*C type*), 452
vrna_mx_mfe_free (*C function*), 678
vrna_mx_mfe_s (*C struct*), 679
vrna_mx_mfe_s.length (*C var*), 679
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError (*C var*), 680
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError_sq (*C var*), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError_local (*C var*), 682
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError_6 (*C var*), 683
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 683
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 682
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 685
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 685
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 686
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 686
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 685
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.E_ (*C var*), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.f3 (*C var*), 680
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.f3 (*C var*), 682
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.f5 (*C var*), 680
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.Fc (*C var*), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.Fc (*C var*), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.Fc (*C var*), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.Fc (*C var*), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.fM (*C var*), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.fM (*C var*), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.fM (*C var*), 681

vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 681
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 682
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 680
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 680
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 682
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 682
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 683
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 683
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 685
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 686
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 686
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 687
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 685
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 683
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 683
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 685
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 686
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 683
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 683
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 685
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 686
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 684
vrna_mx_mfe_s.PhonyNameDueToError.PhonyNameDueToError.l_ (C var), 685

(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.l_m
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.l_m
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.l_m
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.p_g
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.pR
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.pro
(*C var*), 707
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q1k
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q2l
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_B
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_B
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 707
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q_g
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q_l
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 707
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_r
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.l_m
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.l_m
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.l_m
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.p_g
(*C var*), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.pR
(*C var*), 706
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.pro
(*C var*), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q
(*C var*), 707
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q
(*C var*), 704
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q1k
(*C var*), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q2l
(*C var*), 707
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_B
(*C var*), 707
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_B
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 710
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_c
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q_g
(*C var*), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.q_l
(*C var*), 706
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 708
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 709
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_M
(*C var*), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError.Q_r

(C var), 711
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 704
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 706
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 706
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 706
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 706
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 705
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 706
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 707
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 706
vrna_mx_pf_s.PhonyNameDueToError.PhonyNameDueToError (C var), 705
vrna_mx_pf_s.PhonyNameDueToError.[anonymous] (C var), 704
vrna_mx_pf_s.scale (C var), 703
vrna_mx_pf_s.type (C var), 703
vrna_mx_pf_s.[anonymous] (C var), 704
vrna_mx_pf_t (C type), 676
vrna_mx_prepare (C function), 678
vrna_mx_type_e (C enum), 677
vrna_mx_type_e.VRNA_MX_2DFOLD (C enumerator), 677
vrna_mx_type_e.VRNA_MX_DEFAULT (C enumerator), 677
vrna_mx_type_e.VRNA_MX_WINDOW (C enumerator), 677
vrna_n_multichoose_k (C function), 860
VRNA_NEIGHBOR_CHANGE (C macro), 513
VRNA_NEIGHBOR_INVALID (C macro), 514
VRNA_NEIGHBOR_NEW (C macro), 514
vrna_neighbors (C function), 517
vrna_neighbors_successive (C function), 518
vrna_nucleotide_decode (C function), 794
vrna_nucleotide_encode (C function), 794
vrna_nucleotide_IUPAC_identity (C function), 793
VRNA_OBJECTIVE_FUNCTION_ABSOLUTE (C macro), 555
VRNA_OBJECTIVE_FUNCTION_QUADRATIC (C macro), 555
VRNA_OPTION_DEFAULT (C macro), 648
VRNA_OPTION_EVAL_ONLY (C macro), 648
VRNA_OPTION_F3 (C macro), 649
VRNA_OPTION_F5 (C macro), 649
VRNA_OPTION_HYBRID (C macro), 648
VRNA_OPTION_MFE (C macro), 648
VRNA_OPTION_MULTILINE (C macro), 763
VRNA_OPTION_PF (C macro), 648
VRNA_OPTION_WINDOW (C macro), 649
VRNA_OPTION_WINDOW_F3 (C macro), 649
VRNA_OPTION_WINDOW_F5 (C macro), 649
vrna_ostream_free (C function), 746
vrna_ostream_init (C function), 746
vrna_ostream_provide (C function), 747
vrna_ostream_request (C function), 747
vrna_ostream_t (C type), 742
vrna_ostream_threadsafe (C function), 747
vrna_pairing_probs (C function), 322
vrna_pairing_tendency (C function), 834
vrna_param_s (C struct), 385
vrna_param_s.dangle3 (C var), 385
vrna_param_s.dangle3 (C var), 386
vrna_param_s.dangle5 (C var), 386
vrna_param_s.DuplexInit (C var), 387
vrna_param_s.gquad (C var), 387
vrna_param_s.gquadLayerMismatch (C var), 387
vrna_param_s.gquadLayerMismatchMax (C var), 388
vrna_param_s.hairpin (C var), 385
vrna_param_s.Hexaloop_E (C var), 387
vrna_param_s.Hexaloops (C var), 387
vrna_param_s.id (C var), 385
vrna_param_s.int11 (C var), 386
vrna_param_s.int21 (C var), 386
vrna_param_s.int22 (C var), 386
vrna_param_s.internal_loop (C var), 385
vrna_param_s.lxc (C var), 386
vrna_param_s.mismatch1nI (C var), 385
vrna_param_s.mismatch23I (C var), 385
vrna_param_s.mismatchExt (C var), 385
vrna_param_s.mismatchH (C var), 385
vrna_param_s.mismatchI (C var), 385
vrna_param_s.mismatchM (C var), 386
vrna_param_s.MLbase (C var), 386
vrna_param_s.MLclosing (C var), 386
vrna_param_s.MLintern (C var), 386
vrna_param_s.model_details (C var), 388
vrna_param_s.MultipleCA (C var), 387
vrna_param_s.MultipleCB (C var), 387
vrna_param_s.ninio (C var), 386
vrna_param_s.param_file (C var), 388
vrna_param_s.SaltDPXInit (C var), 388
vrna_param_s.SaltLoop (C var), 388
vrna_param_s.SaltLoopDbl (C var), 388
vrna_param_s.SaltMLbase (C var), 388

vrna_param_s.SaltMLclosing (*C var*), 388
vrna_param_s.SaltMLintern (*C var*), 388
vrna_param_s.SaltStack (*C var*), 388
vrna_param_s.stack (*C var*), 385
vrna_param_s.temperature (*C var*), 388
vrna_param_s.TerminalAU (*C var*), 386
vrna_param_s.Tetraloop_E (*C var*), 387
vrna_param_s.Tetraloops (*C var*), 387
vrna_param_s.Triloop_E (*C var*), 387
vrna_param_s.Triloops (*C var*), 387
vrna_param_s.TripleC (*C var*), 387
vrna_param_t (*C type*), 376
VRNA_PARAMETER_FORMAT_DEFAULT (*C macro*), 395
vrna_params (*C function*), 376
vrna_params_copy (*C function*), 377
vrna_params_dirty (*C function*), 376
vrna_params_load (*C function*), 397
vrna_params_load_defaults (*C function*), 399
vrna_params_load_DNA_Mathews1999 (*C function*), 402
vrna_params_load_DNA_Mathews2004 (*C function*), 402
vrna_params_load_from_string (*C function*), 398
vrna_params_load_RNA_Andronescu2007 (*C function*), 400
vrna_params_load_RNA_Langdon2018 (*C function*), 401
vrna_params_load_RNA_misc_special_hairpins (*C function*), 401
vrna_params_load_RNA_Turner1999 (*C function*), 400
vrna_params_load_RNA_Turner2004 (*C function*), 399
vrna_params_prepare (*C function*), 382
vrna_params_reset (*C function*), 380
vrna_params_save (*C function*), 398
vrna_params_subst (*C function*), 378
vrna_path (*C function*), 529
VRNA_PATH_DEFAULT (*C macro*), 529
vrna_path_direct (*C function*), 527
vrna_path_direct_ub (*C function*), 527
vrna_path_findpath (*C function*), 524
vrna_path_findpath_saddle (*C function*), 523
vrna_path_findpath_saddle_ub (*C function*), 523
vrna_path_findpath_ub (*C function*), 525
vrna_path_free (*C function*), 521
vrna_path_gradient (*C function*), 530
VRNA_PATH_NO_TRANSITION_OUTPUT (*C macro*), 529
vrna_path_options_findpath (*C function*), 526
vrna_path_options_free (*C function*), 521
vrna_path_options_t (*C type*), 521
vrna_path_random (*C function*), 531
VRNA_PATH_RANDOM (*C macro*), 528
vrna_path_s (*C struct*), 521
vrna_path_s.en (*C var*), 522
vrna_path_s.move (*C var*), 523
vrna_path_s.s (*C var*), 522
vrna_path_s.type (*C var*), 522
VRNA_PATH_STEEPEST_DESCENT (*C macro*), 528
vrna_path_t (*C type*), 521
VRNA_PATH_TYPE_DOT_BRACKET (*C macro*), 520
VRNA_PATH_TYPE_MOVES (*C macro*), 520
vrna_pbacktrack (*C function*), 358
vrna_pbacktrack5 (*C function*), 351
vrna_pbacktrack5_cb (*C function*), 353
vrna_pbacktrack5_num (*C function*), 352
vrna_pbacktrack5_resume (*C function*), 354
vrna_pbacktrack5_resume_cb (*C function*), 356
vrna_pbacktrack5_TwoD (*C function*), 511
vrna_pbacktrack_cb (*C function*), 360
VRNA_PBACKTRACK_DEFAULT (*C macro*), 349
vrna_pbacktrack_mem_free (*C function*), 372
vrna_pbacktrack_mem_t (*C type*), 351
VRNA_PBACKTRACK_NON_REDUNDANT (*C macro*), 350
vrna_pbacktrack_num (*C function*), 359
vrna_pbacktrack_resume (*C function*), 361
vrna_pbacktrack_resume_cb (*C function*), 363
vrna_pbacktrack_sub (*C function*), 364
vrna_pbacktrack_sub_cb (*C function*), 367
vrna_pbacktrack_sub_num (*C function*), 365
vrna_pbacktrack_sub_resume (*C function*), 368
vrna_pbacktrack_sub_resume_cb (*C function*), 370
vrna_pbacktrack_TwoD (*C function*), 510
vrna_pf (*C function*), 306
vrna_pf_add (*C function*), 307
vrna_pf_alifold (*C function*), 309
vrna_pf_circalifold (*C function*), 309
vrna_pf_circfold (*C function*), 308
vrna_pf_co_fold (*C function*), 310, 492
vrna_pf_dimer (*C function*), 307
vrna_pf_dimer_concentrations (*C function*), 491
vrna_pf_dimer_probs (*C function*), 322
vrna_pf_float_precision (*C function*), 306
vrna_pf_fold (*C function*), 307
vrna_pf_substrands (*C function*), 307
vrna_pf_TwoD (*C function*), 509
vrna_pfl_fold (*C function*), 313
vrna_pfl_fold_cb (*C function*), 314
vrna_pfl_fold_up (*C function*), 314
vrna_pfl_fold_up_cb (*C function*), 315
vrna_pinfo_s (*C struct*), 811
vrna_pinfo_s.bp (*C var*), 812
vrna_pinfo_s.comp (*C var*), 812
vrna_pinfo_s.ent (*C var*), 812
vrna_pinfo_s.i (*C var*), 812
vrna_pinfo_s.j (*C var*), 812
vrna_pinfo_s.p (*C var*), 812
vrna_pinfo_t (*C type*), 807
vrna_pk_plex (*C function*), 570
vrna_pk_plex_accessibility (*C function*), 571
vrna_pk_plex_opt (*C function*), 571
vrna_pk_plex_opt_defaults (*C function*), 571
vrna_pk_plex_opt_fun (*C function*), 572
vrna_pk_plex_opt_t (*C type*), 569
vrna_pk_plex_result_s (*C struct*), 572
vrna_pk_plex_result_s.dG1 (*C var*), 573

vrna_pk_plex_result_s.dG2 (*C var*), 573
vrna_pk_plex_result_s.dGint (*C var*), 573
vrna_pk_plex_result_s.dGpk (*C var*), 572
vrna_pk_plex_result_s.end_3 (*C var*), 573
vrna_pk_plex_result_s.end_5 (*C var*), 573
vrna_pk_plex_result_s.energy (*C var*), 572
vrna_pk_plex_result_s.start_3 (*C var*), 573
vrna_pk_plex_result_s.start_5 (*C var*), 573
vrna_pk_plex_result_s.structure (*C var*), 572
vrna_pk_plex_score_f (*C type*), 569
vrna_pk_plex_t (*C type*), 570
vrna_plist (*C function*), 823
vrna_plist_append (*C function*), 823
vrna_plist_from_probs (*C function*), 311
vrna_plist_gquad_from_pr (*C function*), 576
vrna_plist_gquad_from_pr_max (*C function*), 576
vrna_plist_t (*C type*), 748
VRNA_PLIST_TYPE_BASEPAIR (*C macro*), 822
VRNA_PLIST_TYPE_GQUAD (*C macro*), 822
VRNA_PLIST_TYPE_H_MOTIF (*C macro*), 822
VRNA_PLIST_TYPE_HELIX (*C macro*), 822
VRNA_PLIST_TYPE_I_MOTIF (*C macro*), 822
VRNA_PLIST_TYPE_STACK (*C macro*), 822
VRNA_PLIST_TYPE_TRIPLE (*C macro*), 822
VRNA_PLIST_TYPE_UD_MOTIF (*C macro*), 822
VRNA_PLIST_TYPE_UNPAIRED (*C macro*), 822
vrna_plot_coords (*C function*), 870
vrna_plot_coords_circular (*C function*), 873
vrna_plot_coords_circular_pt (*C function*), 874
vrna_plot_coords_pt (*C function*), 871
vrna_plot_coords_puzzler (*C function*), 875
vrna_plot_coords_puzzler_pt (*C function*), 876
vrna_plot_coords_simple (*C function*), 872
vrna_plot_coords_simple_pt (*C function*), 873
vrna_plot_coords_turtle (*C function*), 877
vrna_plot_coords_turtle_pt (*C function*), 878
vrna_plot_data_s (*C struct*), 890
vrna_plot_data_s.md (*C var*), 891
vrna_plot_data_s.options (*C var*), 891
vrna_plot_data_s.post (*C var*), 890
vrna_plot_data_s.pre (*C var*), 890
vrna_plot_data_t (*C type*), 887
vrna_plot_dp_EPS (*C function*), 882
vrna_plot_dp_PS_list (*C function*), 883
vrna_plot_layout (*C function*), 867
vrna_plot_layout_circular (*C function*), 868
vrna_plot_layout_free (*C function*), 870
vrna_plot_layout_puzzler (*C function*), 869
vrna_plot_layout_s (*C struct*), 878
vrna_plot_layout_s.arcs (*C var*), 879
vrna_plot_layout_s.bbox (*C var*), 879
vrna_plot_layout_s.length (*C var*), 879
vrna_plot_layout_s.type (*C var*), 879
vrna_plot_layout_s.x (*C var*), 879
vrna_plot_layout_s.y (*C var*), 879
vrna_plot_layout_simple (*C function*), 868
vrna_plot_layout_t (*C type*), 867
vrna_plot_layout_turtle (*C function*), 869
vrna_plot_options_puzzler (*C function*), 876
vrna_plot_options_puzzler_free (*C function*), 877
vrna_plot_options_puzzler_t (*C struct*), 879
vrna_plot_options_puzzler_t.allowFlipping (*C var*), 880
vrna_plot_options_puzzler_t.checkAncestorIntersections (*C var*), 879
vrna_plot_options_puzzler_t.checkExteriorIntersections (*C var*), 880
vrna_plot_options_puzzler_t.checkSiblingIntersections (*C var*), 880
vrna_plot_options_puzzler_t.config (*C var*), 880
vrna_plot_options_puzzler_t.drawArcs (*C var*), 879
vrna_plot_options_puzzler_t.filename (*C var*), 880
vrna_plot_options_puzzler_t.maximumNumberOfConfigChanges (*C var*), 880
vrna_plot_options_puzzler_t.numberOfChangesAppliedToConf (*C var*), 880
vrna_plot_options_puzzler_t.optimize (*C var*), 880
vrna_plot_options_puzzler_t.paired (*C var*), 879
vrna_plot_options_puzzler_t.psNumber (*C var*), 880
vrna_plot_options_puzzler_t.unpaired (*C var*), 879
VRNA_PLOT_PROBABILITIES_ACC (*C macro*), 881
VRNA_PLOT_PROBABILITIES_BP (*C macro*), 881
VRNA_PLOT_PROBABILITIES_DEFAULT (*C macro*), 882
VRNA_PLOT_PROBABILITIES_SC_BP (*C macro*), 882
VRNA_PLOT_PROBABILITIES_SC_MOTIF (*C macro*), 882
VRNA_PLOT_PROBABILITIES_SC_UP (*C macro*), 882
VRNA_PLOT_PROBABILITIES_SD (*C macro*), 881
VRNA_PLOT_PROBABILITIES_UD (*C macro*), 881
VRNA_PLOT_PROBABILITIES_UD_LIN (*C macro*), 881
vrna_plot_structure (*C function*), 887
vrna_plot_structure_eps (*C function*), 887
vrna_plot_structure_gml (*C function*), 887
vrna_plot_structure_ssv (*C function*), 888
vrna_plot_structure_svg (*C function*), 887
vrna_plot_structure_xrna (*C function*), 888
VRNA_PLOT_TYPE_CIRCULAR (*C macro*), 866
VRNA_PLOT_TYPE_DEFAULT (*C macro*), 867
VRNA_PLOT_TYPE_NAVIEW (*C macro*), 866
VRNA_PLOT_TYPE_PUZZLER (*C macro*), 866
VRNA_PLOT_TYPE_SIMPLE (*C macro*), 866
VRNA_PLOT_TYPE_TURTLE (*C macro*), 866
vrna_positional_entropy (*C function*), 326
vrna_pr_energy (*C function*), 327
vrna_pr_structure (*C function*), 326
VRNA_PROBING_DATA_CHECK_SEQUENCE (*C macro*), 551

VRNA_PROBING_DATA_DEFAULT (*C macro*), 537
vrna_probing_data_deigan (*C function*), 541
vrna_probing_data_Deigan2009 (*C function*), 535
vrna_probing_data_Deigan2009_comparative (*C function*), 535
vrna_probing_data_deigan_comparative (*C function*), 542
vrna_probing_data_deigan_trans (*C function*), 542
vrna_probing_data_deigan_trans_comparative (*C function*), 543
vrna_probing_data_eddy (*C function*), 548
vrna_probing_data_Eddy2014_2 (*C function*), 536
vrna_probing_data_Eddy2014_2_comparative (*C function*), 536
vrna_probing_data_eddy_comparative (*C function*), 549
vrna_probing_data_eddy_trans (*C function*), 549
vrna_probing_data_eddy_trans_comparative (*C function*), 551
vrna_probing_data_free (*C function*), 533
vrna_probing_data_linear (*C function*), 533
vrna_probing_data_linear_energies (*C function*), 534
vrna_probing_data_linear_multi (*C function*), 533
vrna_probing_data_linear_num (*C function*), 533
vrna_probing_data_linear_raw (*C function*), 533
VRNA_PROBING_DATA_LINEAR_TARGET_BP (*C macro*), 537
VRNA_PROBING_DATA_LINEAR_TARGET_STACK (*C macro*), 537
VRNA_PROBING_DATA_LINEAR_TARGET_UP (*C macro*), 537
vrna_probing_data_linear_weight (*C function*), 534
vrna_probing_data_load_n_distribute (*C function*), 535
VRNA_PROBING_DATA_SINGLE_STRATEGY (*C macro*), 537
VRNA_PROBING_DATA_SINGLE_WEIGHT (*C macro*), 537
vrna_probing_data_t (*C type*), 532
VRNA_PROBING_DATA_WEIGHT_POSITION_WISE (*C macro*), 537
vrna_probing_data_zarringhalam (*C function*), 544
vrna_probing_data_Zarringhalam2012 (*C function*), 535
vrna_probing_data_Zarringhalam2012_comparative (*C function*), 536
vrna_probing_data_zarringhalam_comparative (*C function*), 546
vrna_probing_data_zarringhalam_trans (*C function*), 545
vrna_probing_data_zarringhalam_trans_comparative (*C function*), 547
VRNA_PROBING_METHOD_DEIGAN2009_DEFAULT_b (*C macro*), 540
VRNA_PROBING_METHOD_DEIGAN2009_DEFAULT_m (*C macro*), 540
VRNA_PROBING_METHOD_MULTI_PARAMS_0 (*C macro*), 537
VRNA_PROBING_METHOD_MULTI_PARAMS_1 (*C macro*), 538
VRNA_PROBING_METHOD_MULTI_PARAMS_2 (*C macro*), 538
VRNA_PROBING_METHOD_MULTI_PARAMS_3 (*C macro*), 538
VRNA_PROBING_METHOD_MULTI_PARAMS_DEFAULT (*C macro*), 538
VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_beta (*C macro*), 543
VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_conversion (*C macro*), 544
VRNA_PROBING_METHOD_ZARRINGHALAM2012_DEFAULT_probability (*C macro*), 544
vrna_probing_strategy_chain (*C function*), 539
vrna_probing_strategy_chain_data (*C function*), 539
vrna_probing_strategy_chain_data_append (*C function*), 539
vrna_probing_strategy_chain_data_at (*C function*), 540
vrna_probing_strategy_chain_data_free (*C function*), 540
vrna_probing_strategy_chain_data_size (*C function*), 540
vrna_probing_strategy_deigan (*C function*), 541
vrna_probing_strategy_deigan_options (*C function*), 541
vrna_probing_strategy_deigan_options_free (*C function*), 541
vrna_probing_strategy_eddy (*C function*), 548
VRNA_PROBING_STRATEGY_EDDY_NO_TEMPERATURE_RESCALING (*C macro*), 547
vrna_probing_strategy_eddy_options (*C function*), 548
VRNA_PROBING_STRATEGY_EDDY_OPTIONS_DEFAULT (*C macro*), 547
vrna_probing_strategy_eddy_options_free (*C function*), 548
vrna_probing_strategy_f (*C type*), 539
vrna_probing_strategy_opt_t (*C type*), 539
vrna_probing_strategy_zarringhalam (*C function*), 544
vrna_probing_strategy_zarringhalam_options (*C function*), 544
vrna_probing_strategy_zarringhalam_options_free (*C function*), 544
vrna_probs_window (*C function*), 312
VRNA_PROBS_WINDOW_BPP (*C macro*), 316
vrna_probs_window_callback (*C type*), 319
vrna_probs_window_f (*C type*), 318
VRNA_PROBS_WINDOW_PF (*C macro*), 317
VRNA_PROBS_WINDOW_STACKP (*C macro*), 317

VRNA_PROBS_WINDOW_UP (*C macro*), 316
VRNA_PROBS_WINDOW_UP_SPLIT (*C macro*), 317
vrna_pscore (*C function*), 808
vrna_pscore_freq (*C function*), 808
vrna_pt_ali_get (*C function*), 820
vrna_pt_pk_get (*C function*), 820
vrna_pt_pk_remove (*C function*), 820
vrna_pt_snoop_get (*C function*), 820
vrna_ptable (*C function*), 819
vrna_ptable_compat_remove (*C function*), 821
vrna_ptable_copy (*C function*), 820
vrna_ptable_count_bp (*C function*), 821
vrna_ptable_count_up (*C function*), 821
vrna_ptable_from_string (*C function*), 819
vrna_ptypes (*C function*), 793
vrna_ptypes_prepare (*C function*), 793
vrna_random_string (*C function*), 804
VRNA_REACTIVITY_MISSING (*C macro*), 532
vrna_read_line (*C function*), 778
vrna_realloc (*C function*), 643
vrna_recursion_status_f (*C type*), 650
vrna_refBPcnt_matrix (*C function*), 835
vrna_refBPdist_matrix (*C function*), 835
vrna_rotational_symmetry (*C function*), 858
vrna_rotational_symmetry_db (*C function*), 859
vrna_rotational_symmetry_db_pos (*C function*), 859
vrna_rotational_symmetry_num (*C function*), 856
vrna_rotational_symmetry_pos (*C function*), 858
vrna_rotational_symmetry_pos_num (*C function*), 857
vrna_salt_duplex_init (*C function*), 395
vrna_salt_loop (*C function*), 393
vrna_salt_loop_int (*C function*), 394
vrna_salt_ml (*C function*), 394
vrna_salt_stack (*C function*), 394
vrna_sc_add_auxdata (*C function*), 616
vrna_sc_add_bp (*C function*), 609
vrna_sc_add_bp_comparative (*C function*), 610
vrna_sc_add_bp_comparative_seq (*C function*), 611
vrna_sc_add_bt (*C function*), 617
vrna_sc_add_data (*C function*), 615
vrna_sc_add_exp_f (*C function*), 617
vrna_sc_add_f (*C function*), 616
vrna_sc_add_hi_motif (*C function*), 567
vrna_sc_add_iip (*C function*), 598
vrna_sc_add_SHAPE_deigan (*C function*), 552
vrna_sc_add_SHAPE_deigan_ali (*C function*), 553
vrna_sc_add_SHAPE_eddy_2 (*C function*), 554
vrna_sc_add_SHAPE_zarringhalam (*C function*), 553
vrna_sc_add_stack (*C function*), 612
vrna_sc_add_stack_comparative (*C function*), 612
vrna_sc_add_stack_comparative_seq (*C function*), 612
vrna_sc_add_up (*C function*), 605
vrna_sc_add_up_comparative (*C function*), 605
vrna_sc_add_up_comparative_seq (*C function*), 606
vrna_sc_bp_storage_t (*C struct*), 598
vrna_sc_bp_storage_t.e (*C var*), 598
vrna_sc_bp_storage_t.interval_end (*C var*), 598
vrna_sc_bp_storage_t.interval_start (*C var*), 598
vrna_sc_bt_f (*C type*), 614
vrna_sc_direct_f (*C type*), 613
vrna_sc_exp_direct_f (*C type*), 614
vrna_sc_exp_f (*C type*), 613
vrna_sc_f (*C type*), 612
vrna_sc_free (*C function*), 597
vrna_sc_init (*C function*), 596
vrna_sc_ligand_detect_motifs (*C function*), 568
vrna_sc_ligand_get_all_motifs (*C function*), 568
vrna_sc_minimize_pertubation (*C function*), 557
vrna_sc_mod (*C function*), 562
vrna_sc_mod_7DA (*C function*), 564
VRNA_SC_MOD_CHECK_FALLBACK (*C macro*), 558
VRNA_SC_MOD_CHECK_UNMOD (*C macro*), 558
VRNA_SC_MOD_DEFAULT (*C macro*), 559
vrna_sc_mod_dihydrouridine (*C function*), 566
vrna_sc_mod_inosine (*C function*), 564
vrna_sc_mod_json (*C function*), 560
vrna_sc_mod_jsonfile (*C function*), 561
vrna_sc_mod_m6A (*C function*), 563
vrna_sc_mod_param_t (*C type*), 559
vrna_sc_mod_parameters_free (*C function*), 560
vrna_sc_mod_pseudouridine (*C function*), 563
vrna_sc_mod_purine (*C function*), 565
vrna_sc_mod_read_from_json (*C function*), 560
vrna_sc_mod_read_from_jsonfile (*C function*), 559
VRNA_SC_MOD_SILENT (*C macro*), 559
vrna_sc_motif_s (*C struct*), 568
vrna_sc_motif_s.i (*C var*), 568
vrna_sc_motif_s.j (*C var*), 568
vrna_sc_motif_s.k (*C var*), 568
vrna_sc_motif_s.l (*C var*), 568
vrna_sc_motif_s.number (*C var*), 568
vrna_sc_motif_t (*C type*), 567
vrna_sc_multi_cb_add (*C function*), 616
vrna_sc_multi_cb_add_comparative (*C function*), 617
vrna_sc_prepare (*C function*), 597
vrna_sc_probing (*C function*), 532
vrna_sc_remove (*C function*), 597
vrna_sc_s (*C struct*), 598
vrna_sc_s.bp_storage (*C var*), 599
vrna_sc_s.bt (*C var*), 599
vrna_sc_s.data (*C var*), 600
vrna_sc_s.energy_stack (*C var*), 599
vrna_sc_s.energy_up (*C var*), 598
vrna_sc_s.exp_energy_stack (*C var*), 599
vrna_sc_s.exp_energy_up (*C var*), 599
vrna_sc_s.exp_f (*C var*), 600

vrna_sc_s.f (*C var*), 599
vrna_sc_s.free_data (*C var*), 600
vrna_sc_s.iip (*C var*), 599
vrna_sc_s.n (*C var*), 598
vrna_sc_s.PhonyNameDueToError.PhonyNameDueToError.energy_bp (*C var*), 601
vrna_sc_s.PhonyNameDueToError.PhonyNameDueToError.energy_bp_local (*C var*), 601
vrna_sc_s.PhonyNameDueToError.PhonyNameDueToError.exp_energy_bp (*C var*), 601
vrna_sc_s.PhonyNameDueToError.PhonyNameDueToError.exp_energy_bp_local (*C var*), 601
vrna_sc_s.PhonyNameDueToError.[anonymous] (*C var*), 600
vrna_sc_s.prepare_data (*C var*), 600
vrna_sc_s.state (*C var*), 598
vrna_sc_s.type (*C var*), 598
vrna_sc_s.up_storage (*C var*), 599
vrna_sc_s.[anonymous] (*C var*), 600
vrna_sc_set_auxdata_comparative (*C function*), 616
vrna_sc_set_auxdata_comparative_seq (*C function*), 616
vrna_sc_set_bp (*C function*), 607
vrna_sc_set_bp_comparative (*C function*), 607
vrna_sc_set_bp_comparative_seq (*C function*), 608
vrna_sc_set_data_comparative (*C function*), 616
vrna_sc_set_data_comparative_seq (*C function*), 616
vrna_sc_set_exp_f_comparative (*C function*), 618
vrna_sc_set_f_comparative (*C function*), 617
vrna_sc_set_stack (*C function*), 612
vrna_sc_set_stack_comparative (*C function*), 612
vrna_sc_set_stack_comparative_seq (*C function*), 612
vrna_sc_set_up (*C function*), 603
vrna_sc_set_up_comparative (*C function*), 603
vrna_sc_set_up_comparative_seq (*C function*), 604
vrna_sc_SHAPE_parse_method (*C function*), 536
vrna_sc_SHAPE_to_pr (*C function*), 552
vrna_sc_t (*C type*), 596
vrna_sc_type_e (*C enum*), 596
vrna_sc_type_e.VRNA_SC_DEFAULT (*C enumerator*), 596
vrna_sc_type_e.VRNA_SC_WINDOW (*C enumerator*), 596
vrna_sc_update (*C function*), 597
vrna_score_from_confusion_matrix (*C function*), 831
vrna_score_s (*C struct*), 832
vrna_score_s.F1 (*C var*), 833
vrna_score_s.FDR (*C var*), 833
vrna_score_s.FN (*C var*), 832
vrna_score_s.FNR (*C var*), 833
vrna_score_s.FOR (*C var*), 833
vrna_score_s.FP (*C var*), 832
vrna_score_s.FPR (*C var*), 833
vrna_score_s.MCC (*C var*), 833
vrna_score_s.NPV (*C var*), 833
vrna_score_s.PPV (*C var*), 833
vrna_score_s.TN (*C var*), 832
vrna_score_s.TNR (*C var*), 833
vrna_score_s.TP (*C var*), 832
vrna_score_s.TPR (*C var*), 832
vrna_search_BM_BCT (*C function*), 855
vrna_search_BM_BCT_num (*C function*), 855
vrna_search_BMH (*C function*), 854
vrna_search_BMH_num (*C function*), 854
vrna_sect_s (*C struct*), 762
vrna_sect_s.i (*C var*), 762
vrna_sect_s.j (*C var*), 762
vrna_sect_s.ml (*C var*), 762
vrna_sect_t (*C type*), 759
vrna_seq_encode (*C function*), 793
vrna_seq_encode_simple (*C function*), 793
vrna_seq_reverse (*C function*), 805
vrna_seq_t (*C type*), 792
vrna_seq_toRNA (*C function*), 799
vrna_seq_toupper (*C function*), 804
vrna_seq_type_e (*C enum*), 792
vrna_seq_type_e.VRNA_SEQ_DNA (*C enumerator*), 793
vrna_seq_type_e.VRNA_SEQ_RNA (*C enumerator*), 793
vrna_seq_type_e.VRNA_SEQ_UNKNOWN (*C enumerator*), 792
vrna_seq_ungapped (*C function*), 799
vrna_sequence (*C function*), 795
vrna_sequence_add (*C function*), 795
VRNA_SEQUENCE_DNA (*C macro*), 792
vrna_sequence_length_max (*C function*), 793
vrna_sequence_mutate (*C function*), 795
vrna_sequence_mutate_strand (*C function*), 795
vrna_sequence_order_update (*C function*), 795
vrna_sequence_prepare (*C function*), 795
vrna_sequence_remove (*C function*), 795
vrna_sequence_remove_all (*C function*), 795
VRNA_SEQUENCE_RNA (*C macro*), 792
vrna_sequence_s (*C struct*), 795
vrna_sequence_s.encoding (*C var*), 796
vrna_sequence_s.encoding3 (*C var*), 796
vrna_sequence_s.encoding5 (*C var*), 796
vrna_sequence_s.length (*C var*), 796
vrna_sequence_s.name (*C var*), 796
vrna_sequence_s.string (*C var*), 796
vrna_sequence_s.type (*C var*), 795
vrna_sol_TwoD_pf_t (*C struct*), 509
vrna_sol_TwoD_pf_t.k (*C var*), 510
vrna_sol_TwoD_pf_t.l (*C var*), 510
vrna_sol_TwoD_pf_t.q (*C var*), 510
vrna_sol_TwoD_t (*C struct*), 501
vrna_sol_TwoD_t.en (*C var*), 501
vrna_sol_TwoD_t.k (*C var*), 501

vrna_sol_TwoD_t.l (*C var*), 501
vrna_sol_TwoD_t.s (*C var*), 502
vrna_stack_prob (*C function*), 323
VRNA_STATUS_MFE_POST (*C macro*), 647
VRNA_STATUS_MFE_PRE (*C macro*), 647
VRNA_STATUS_PF_POST (*C macro*), 647
VRNA_STATUS_PF_PRE (*C macro*), 647
vrna_str_to_dbl_array (*C function*), 804
vrna_strcat_printf (*C function*), 801
vrna_strcat_vprintf (*C function*), 801
vrna_strchr (*C function*), 806
vrna_strdup_printf (*C function*), 800
vrna_strdup_vprintf (*C function*), 800
vrna_stream_output_f (*C type*), 742
vrna_string_append (*C function*), 741
vrna_string_append_cstring (*C function*), 741
vrna_string_available_space (*C function*), 741
vrna_string_free (*C function*), 741
VRNA_STRING_HEADER (*C macro*), 740
vrna_string_header_s (*C struct*), 741
vrna_string_header_s.backup (*C var*), 741
vrna_string_header_s.len (*C var*), 741
vrna_string_header_s.shift_post (*C var*), 741
vrna_string_header_s.size (*C var*), 741
vrna_string_header_t (*C type*), 740
vrna_string_length (*C function*), 741
vrna_string_make (*C function*), 741
vrna_string_make_space_for (*C function*), 741
vrna_string_size (*C function*), 741
vrna_string_t (*C type*), 740
vrna_strjoin (*C function*), 804
vrna_strsplit (*C function*), 803
vrna_strtrim (*C function*), 802
VRNA_STRUCTURE_TREE_EXPANDED (*C macro*), 827
VRNA_STRUCTURE_TREE_HIT (*C macro*), 826
VRNA_STRUCTURE_TREE_SHAPIRO (*C macro*), 827
VRNA_STRUCTURE_TREE_SHAPIRO_EXT (*C macro*), 827
VRNA_STRUCTURE_TREE_SHAPIRO_SHORT (*C macro*), 827
VRNA_STRUCTURE_TREE_SHAPIRO_WEIGHT (*C macro*), 827
vrna_subopt (*C function*), 347
vrna_subopt_cb (*C function*), 348
vrna_subopt_result_f (*C type*), 346
vrna_subopt_zuker (*C function*), 346
vrna_time_stamp (*C function*), 645
VRNA_TRANSFORM_DEFAULT (*C macro*), 852
VRNA_TRANSFORM_ENFORCE_DOMAIN_SOURCE (*C macro*), 849
VRNA_TRANSFORM_ENFORCE_DOMAIN_TARGET (*C macro*), 850
VRNA_TRANSFORM_ENFORCE_DOMAINS (*C macro*), 850
VRNA_TRANSFORM_MAP (*C macro*), 852
VRNA_TRANSFORM_MAP_SOURCE (*C macro*), 851
VRNA_TRANSFORM_MAP_SOURCE_HIGH (*C macro*), 850
VRNA_TRANSFORM_MAP_SOURCE_LOW (*C macro*), 850
VRNA_TRANSFORM_MAP_TARGET (*C macro*), 851
VRNA_TRANSFORM_MAP_TARGET_HIGH (*C macro*), 851
VRNA_TRANSFORM_MAP_TARGET_LOW (*C macro*), 851
vrna_tree_string_to_db (*C function*), 829
vrna_tree_string_unweight (*C function*), 828
VRNA_TRIM_ALL (*C macro*), 799
VRNA_TRIM_DEFAULT (*C macro*), 798
VRNA_TRIM_IN_BETWEEN (*C macro*), 798
VRNA_TRIM_LEADING (*C macro*), 797
VRNA_TRIM_SUBST_BY_FIRST (*C macro*), 798
VRNA_TRIM_TRAILING (*C macro*), 798
vrna_ud_add_motif (*C function*), 622
vrna_ud_add_probs_f (*C type*), 620
vrna_ud_exp_f (*C type*), 619
vrna_ud_exp_production_f (*C type*), 620
vrna_ud_f (*C type*), 619
vrna_ud_get_probs_f (*C type*), 620
vrna_ud_motifs_centroid (*C function*), 621
vrna_ud_motifs_MEA (*C function*), 621
vrna_ud_motifs_MFE (*C function*), 622
vrna_ud_production_f (*C type*), 620
vrna_ud_remove (*C function*), 623
vrna_ud_set_data (*C function*), 623
vrna_ud_set_exp_prod_rule_cb (*C function*), 624
vrna_ud_set_prod_rule_cb (*C function*), 624
vrna_ud_t (*C type*), 619
vrna_unit_energy_e (*C enum*), 862
vrna_unit_energy_e.VRNA_UNIT_CAL (*C enumerator*), 862
vrna_unit_energy_e.VRNA_UNIT_CAL_IT (*C enumerator*), 862
vrna_unit_energy_e.VRNA_UNIT_DACAL (*C enumerator*), 863
vrna_unit_energy_e.VRNA_UNIT_DACAL_IT (*C enumerator*), 862
vrna_unit_energy_e.VRNA_UNIT_EV (*C enumerator*), 863
vrna_unit_energy_e.VRNA_UNIT_G_TNT (*C enumerator*), 863
vrna_unit_energy_e.VRNA_UNIT_J (*C enumerator*), 862
vrna_unit_energy_e.VRNA_UNIT_KCAL (*C enumerator*), 863
vrna_unit_energy_e.VRNA_UNIT_KCAL_IT (*C enumerator*), 862
vrna_unit_energy_e.VRNA_UNIT_KG_TNT (*C enumerator*), 863
vrna_unit_energy_e.VRNA_UNIT_KJ (*C enumerator*), 862
vrna_unit_energy_e.VRNA_UNIT_KWH (*C enumerator*), 863
vrna_unit_energy_e.VRNA_UNIT_T_TNT (*C enumerator*), 863
vrna_unit_energy_e.VRNA_UNIT_WH (*C enumerator*), 863
vrna_unit_temperature_e (*C enum*), 863
vrna_unit_temperature_e.VRNA_UNIT_DEG_C (*C enumerator*), 863
vrna_unit_temperature_e.VRNA_UNIT_DEG_DE

(*C enumerator*), 864
vrna_unit_temperature_e.VRNA_UNIT_DEG_F (*C enumerator*), 863
vrna_unit_temperature_e.VRNA_UNIT_DEG_N (*C enumerator*), 864
vrna_unit_temperature_e.VRNA_UNIT_DEG_R (*C enumerator*), 863
vrna_unit_temperature_e.VRNA_UNIT_DEG_RE (*C enumerator*), 864
vrna_unit_temperature_e.VRNA_UNIT_DEG_RO (*C enumerator*), 864
vrna_unit_temperature_e.VRNA_UNIT_K (*C enumerator*), 863
VRNA_UNSTRUCTURED_DOMAIN_ALL_LOOPS (*C macro*), 619
VRNA_UNSTRUCTURED_DOMAIN_EXT_LOOP (*C macro*), 618
VRNA_UNSTRUCTURED_DOMAIN_HP_LOOP (*C macro*), 618
VRNA_UNSTRUCTURED_DOMAIN_INT_LOOP (*C macro*), 618
VRNA_UNSTRUCTURED_DOMAIN_MB_LOOP (*C macro*), 618
VRNA_UNSTRUCTURED_DOMAIN_MOTIF (*C macro*), 619
vrna_unstructured_domain_s (*C struct*), 625
vrna_unstructured_domain_s.data (*C var*), 626
vrna_unstructured_domain_s.energy_cb (*C var*), 626
vrna_unstructured_domain_s.exp_energy_cb (*C var*), 626
vrna_unstructured_domain_s.exp_prod_cb (*C var*), 626
vrna_unstructured_domain_s.free_data (*C var*), 626
vrna_unstructured_domain_s.motif (*C var*), 625
vrna_unstructured_domain_s.motif_count (*C var*), 625
vrna_unstructured_domain_s.motif_en (*C var*), 626
vrna_unstructured_domain_s.motif_name (*C var*), 625
vrna_unstructured_domain_s.motif_size (*C var*), 626
vrna_unstructured_domain_s.motif_type (*C var*), 626
vrna_unstructured_domain_s.probs_add (*C var*), 627
vrna_unstructured_domain_s.probs_get (*C var*), 627
vrna_unstructured_domain_s.prod_cb (*C var*), 626
vrna_unstructured_domain_s.uniq_motif_count (*C var*), 625
vrna_unstructured_domain_s.uniq_motif_size (*C var*), 625
VRNA_UNUSED (*C macro*), 641
vrna_urn (*C function*), 644
VRNA_VERBOSITY_DEFAULT (*C macro*), 475
VRNA_VERBOSITY_QUIET (*C macro*), 475

W

window_size (*RNA.fold_compound attribute*), 980
window_size (*RNA.md attribute*), 1047
window_size (*RNA.md property*), 1051
write_parameter_file (*C function*), 403
write_parameter_file() (*in module RNA*), 1094

X

X (*RNA.COORDINATE attribute*), 908, 909
X (*RNA.COORDINATE property*), 909
xrna_plot (*C function*), 889
xrna_plot() (*in module RNA*), 1094
XSTR (*C macro*), 797
xsubi (*C var*), 646

Y

Y (*RNA.COORDINATE attribute*), 908, 909
Y (*RNA.COORDINATE property*), 909

Z

zsc_compute() (*RNA.fold_compound method*), 1030
zsc_compute_raw() (*RNA.fold_compound method*), 1030
zsc_filter_free() (*RNA.fold_compound method*), 1030
zsc_filter_init() (*RNA.fold_compound method*), 1030
zsc_filter_on() (*RNA.fold_compound method*), 1030
zsc_filter_threshold() (*RNA.fold_compound method*), 1030
zsc_filter_update() (*RNA.fold_compound method*), 1030
zukersubopt (*C function*), 345
zukersubopt() (*in module RNA*), 1094
zukersubopt_par (*C function*), 346