GromacsWrapper Documentation

Release 0.1.12

Oliver Beckstein

June 11, 2010

CONTENTS

GromacsWrapper is a python package that wraps system calls to Gromacs tools into thin classes. This allows for fairly seamless integration of the gromacs tools into python scripts. This is generally superior to shell scripts because of python's better error handling and superior data structures. It also allows for modularization and code re-use. In addition, commands, warnings and errors are logged to a file so that there exists a complete history of what has been done.

See *INSTALL* for download and installation instructions. Documentation is primarily provided through the python doc strings (from which most of the online documentation is generated).

There is also auto-generated online source code documentation available and the source code itself is available in the GromacsWrapper git repository.

Warning: Please be aware that this is **alpha** software that most definitely contains bugs. The API is not stable and can change between releases.
It is *your* responsibility to ensure that you are running simulations with sensible parameters.

The package and the documentation are still in flux and any feedback, bug reports, suggestions and contributions are very welcome. See the package *README* for contact details.

For other approaches to interfacing python and Gromacs see *Alternatives to GromacsWrapper*.

CHAPTER ONE

CONTENTS

1.1 README

See *INSTALL* for installation instructions. Documentation is mostly provided through the python doc strings. See Download and Availability for download instructions if the instructions in *INSTALL* are not sufficient.

There is also auto-generated online source code documentation available. The source code is also available in the GromacsWrapper git repository.

Please be aware that this is **alpha** software that most definitely contains bugs. It is *your* responsibility to ensure that you are running simulations with sensible parameters.

1.1.1 License

The **GromacsWrapper** package is made available under the terms of the GNU Public License v3 (or any higher version at your choice).

See the file COPYING for the licensing terms for all modules except the **vmd** module, which is made available under the LGPL v3 (see COPYING and COPYING.LESSER).

1.1.2 Included Software

The distribution contains third party software that is copyrighted by the authors but distributed under licences compatible with this package license. Where permitted and necessary, software/files were modified to integrate with GromacsWrapper.

In case of problems please direct error reports to Oliver Beckstein in the first instance as these bugs might not have been present in the original software or files.

Included third party content:

GridMat-MD

- Grid-based Membrane Analysis Tool for use with Molecular Dynamics [Allen2009]
- version: 1.0.2
- license: GPL 3.0
- W. J. Allen, J. A. Lemkul, and D. R. Bevan. (2009) “GridMAT-MD: A Grid-based Membrane Analysis Tool for Use With Molecular Dynamics.” J. Comput. Chem. 30 (12): 1952-1958.
- http://bevanlab.biochem.vt.edu/GridMAT-MD/

odict.py

- a simple implementation of an ordered dictionary as proposed in **PEP 0372**
- copyright: (c) 2008 by Armin Ronacher and PEP 273 authors.
- license: modified BSD license (compatible with GPL)
- http://dev.pocoo.org/hg/sandbox/raw-file/tip/odict.py

1.1.3 Citing

If you find this package useful and use it in published work I'd be grateful if it was acknowledged in text as

> "... used GromacsWrapper (Oliver Beckstein, http://sbcb.bioch.ox.ac.uk/oliver/software/GromacsWrapper/)"

or in the Acknowledgements section.

If you use the `gridmatmd` plugin also cite [Allen2009].

Thank you.

References

1.1.4 Download and Availability

The GromacsWrapper home page is http://sbcb.bioch.ox.ac.uk/oliver/software/GromacsWrapper/ . The latest version of the package is being made available via the internet-thingy at the direct download URI

> http://sbcb.bioch.ox.ac.uk/oliver/download/Python/

You can use this URI if you want to install from the network using `easy_install` as described in *INSTALL*.

You can also clone the GromacsWrapper git repository or fork for your own development:

```
git clone git://github.com/orbeckst/GromacsWrapper.git
```

1.1.5 Contact

Please use the Issue Tracker to report bugs and feature requests; general feedback and inquiries can be sent to Oliver Beckstein by e-mail.

1.2 INSTALL

This document should help you to install the **GromacsWrapper** package. The installation uses setuptools (also known as `easy_install` or "egg install"); if this is not available on your system you can either let the installer download it automatically from the internet (so just go to Quick installation instructions) or install it using your package manager, eg:

```
aptitude install python-setuptools
```

or similar.

Please do not hesitate to contact Oliver Beckstein if problems occur or if you have suggestions on how to improve the package or these instructions.

1.2.1 Quick installation instructions

If you have `easy_install` on your system you can directly install from the interweb:

```
easy_install -f http://sbcb.bioch.ox.ac.uk/oliver/download/Python GromacsWrapper
```

This will automatically download and install the latest version.

1.2.2 Manual Download

If your prefer to download manually, get the latest version from

> http://sbcb.bioch.ox.ac.uk/oliver/download/Python

and use any of the following methods (in increasing order of complexity):

- From an egg install file, eg GromacsWrapper-0.1-py2.5.egg:

  ```
  easy_install GromacsWrapper-0.1-py2.5.egg
  ```

- From a tar ball, eg GromacsWrapper-0.1.tar.gz:

  ```
  easy_install GromacsWrapper-0.1.tar.gz
  ```

- From the unpacked source:

  ```
  tar -zxvf GromacsWrapper-0.1.tar.gz
  cd GromacsWrapper-0.1
  python setup.py install
  ```

See the easy_install instructions for explanation of the options that allow you to install into non-standard places.

1.2.3 Source code access

The tar archive from http://sbcb.bioch.ox.ac.uk/oliver/download/Python contains a full source code distribution.

In order to follow code development you can also browse the code **git** repository at http://github.com/orbeckst/GromacsWrapper or clone the git repository from

> git://github.com/orbeckst/GromacsWrapper.git

1.2.4 Requirements

Python and Gromacs must be installed. ipython is very much recommended. These packages might already be available through your local package manager such as `aptitude/apt`, `yum`, `yast`, `fink` or `macports`.

System requirements

Tested with python 2.5, 2.6 on Linux and Mac OS X. Earlier python versions will likely fail.

Required python modules

The basic package makes use of numpy and can use matplotlib (in the form of the `pylab` package). Only numpy is immediately required (and automatically installed with `easy_install`).

For the `gromacs.analysis` library additional packages are required:

package	version	source
matplotlib	>=0.91.3	http://matplotlib.sourceforge.net/
RecSQL	>=0.3	http://sbcb.bioch.ox.ac.uk/oliver/software/RecSQL/

See Installing all packages and requirements for hints on how to install these package.

1.2.5 Additional instructions

Installing *all* packages and requirements

If you want to make sure that `easy_install` also installs requirements for optional modules then you will have to add the additional requirement `[analysis]` to the command line. For a web install this would look like

```
easy_install -f http://sbcb.bioch.ox.ac.uk/oliver/download/Python GromacsWrapper[analysis]
```

For installation from a downloaded source distribution

```
easy_install GromacsWrapper-0.1.tar.gz[analysis]
```

or from within the unpacked source

```
cd GromacsWrapper-0.1
easy_install . GromacsWrapper[analysis]
```

In each case this will try to download additional packages for the extra *analysis* module.

A common problem appears to be the error Could not find matplotlib as discussed below.

1.2.6 Troubleshooting

For problems with `easy_install` please read the User setuptools instructions.

Installing in non-standard locations

Inform yourself about how to use `easy_install` to install packages in Custom Installation Locations.

For code hacking and development a *developer installation* is often useful. In the unpacked source:

```
python setup.py develop --install-dir python-lib-dir
```

where `python-lib-dir` must be on the `PYTHONPATH`.

easy_install import error

Online installation can run into issues where the installation dies with the error:

```
ImportError: No module named ez_setup
```

If EasyInstall Troubleshooting does not help then try downloading the source distribution package manually, unpack, and install from inside with something like:

```
python setup.py install
```

If this is still not working contact the author and complain.

Could not find matplotlib

Automatic downloading of `matplotlib` often fails:

```
Searching for matplotlib>=0.91.3
Reading http://pypi.python.org/simple/matplotlib/
Reading http://matplotlib.sourceforge.net
Reading https://sourceforge.net/project/showfiles.php?group_id=80706&package_id=278194
Reading https://sourceforge.net/project/showfiles.php?group_id=80706&package_id=82474
Reading http://sourceforge.net/project/showfiles.php?group_id=80706
No local packages or download links found for matplotlib>=0.91.3
error: Could not find suitable distribution for Requirement.parse('matplotlib>=0.91.3')
```

If automatic downloading of `matplotlib` fails then the best approach is to install it through your package management system. Search for "matplotlib" or "pylab" in the list of available packages.

If this is not an option then download matplotlib manually and install matplotlib manually first. For example,

```
wget http://kent.dl.sourceforge.net/sourceforge/matplotlib/matplotlib-0.98.5.3-py2.5-macosx-10.3-fat
     -O matplotlib-0.98.5.3-py2.5.egg

easy_install matplotlib-0.98.5.3-py2.5.egg
```

Note that you should look at the download matplotlib page to get the latest distribution. As highlighted in the matplotlib installation FAQ it is important to rename the `egg` file (as done in the example above).

Possibly the following installation from the source distribution works, too:

```
easy_install matplotlib-0.98.5.3.tar.gz
```

Once this has been accomplished, try the above installation instructions again; `easy_install` should now pick up the newly installed matplotlib.

1.3 Gromacs package

The `gromacs` package makes Gromacs tools available via thin python wrappers. In addition, it provides little building blocks to solve commonly encountered tasks.

Contents:

1.3.1 `gromacs` – GromacsWrapper Package Overview

GromacsWrapper (package `gromacs`) is a thin shell around the Gromacs tools for light-weight integration into python scripts or interactive use in ipython.

Modules

`gromacs` The top level module contains all gromacs tools; each tool can be run directly or queried for its documentation. It also defines the root logger class (name *gromacs* by default).

`gromacs.config` Configuration options. Not really used much at the moment.

`gromacs.cbook` The Gromacs cook book contains typical applications of the tools. In many cases this not more than just an often-used combination of parameters for a tool.

`gromacs.tools` Contains classes that wrap the gromacs tools. They are automatically generated from the list of tools in `gromacs.tools.gmx_tools`.

`gromacs.formats` Classes to represent data files in various formats such as xmgrace graphs. The classes allow reading and writing and for graphs, also plotting of the data.

`gromacs.utilities` Convenience functions and mixin-classes that are used as helpers in other modules.

`gromacs.setup` Functions to set up a MD simulation, containing tasks such as solvation and adding ions, energy minimizqtion, MD with position-restraints, and equilibrium MD.

`gromacs.qsub` Functions to handle batch submission queuing systems.

`gromacs.run` Classes to run **mdrun** in various way, including on multiprocessor systems.

`gromacs.analysis` A package that collects whole analysis tasks. It uses the gromacs but is otherwise only loosely coupled with the rest. At the moment it only contains the infrastructure and an example application. See the package documentation.

Examples

The following examples should simply convey the flavour of using the package. See the individual modules for more examples.

Getting help

In python:

```
help(gromacs.g_dist)
gromacs.g_dist.help()
gromacs.g_dist.help(long=True)
```

In `ipython`:

```
gromacs.g_dist ?
```

Simple usage

Gromacs flags are given as python keyword arguments:

```
gromacs.g_dist(v=True, s='topol.tpr', f='md.xtc', o='dist.xvg', dist=1.2)
```

Input to stdin of the command can be supplied:

```
gromacs.make_ndx(f='topol.tpr', o='md.ndx',
                 input=('keep "SOL"', '"SOL" | r NA | r CL', 'name 2 solvent', 'q'))
```

Output of the command can be caught in a variable and analyzed:

```
rc, output, junk = gromacs.grompp(..., stdout=False)        # collects command output
for line in output.split('\n'):
    line = line.strip()
    if line.startswith('System has non-zero total charge:'):
          qtot = float(line[34:])
          break
```

(See `gromacs.cbook.grompp_qtot()` for a more robust implementation of this application.)

Warnings and Exceptions

A number of package-specific exceptions (`GromacsError`) and warnings (`Gromacs*Warning`, `AutoCorrectionWarning`, `BadParameterWarning`) can be raised.

If you want to stop execution at, for instance, a `AutoCorrectionWarning` or `BadParameterWarning` then use the python `warnings` filter:

```
import warnings
warnings.simplefilter('error', gromacs.AutoCorrectionWarning)
warnings.simplefilter('error', gromacs.BadParameterWarning)
```

This will make python raise an exception instead of moving on. The default is to always report, eg:

```
warnings.simplefilter('always', gromacs.BadParameterWarning)
```

The following *exceptions* are defined:

exception `GromacsError`

Error raised when a gromacs tool fails.

Returns error code in the errno attribute and a string in strerror. # TODO: return status code and possibly error message

exception `MissingDataError`

Error raised when prerequisite data are not available.

For analysis with `gromacs.analysis.core.Simulation` this typically means that the `analyze()` method has to be run first.

exception `ParseError`

Error raised when parsing of a file failed.

The following *warnings* are defined:

exception GromacsFailureWarning
Warning about failure of a Gromacs tool.

exception GromacsImportWarning
Warns about problems with using a gromacs tool.

exception GromacsValueWarning
Warns about problems with the value of an option or variable.

exception AutoCorrectionWarning
Warns about cases when the code is choosing new values automatically.

exception BadParameterWarning
Warns if some parameters or variables are unlikely to be appropriate or correct.

exception MissingDataWarning
Warns when prerequisite data/files are not available.

exception UsageWarning
Warns if usage is unexpected/documentation ambiguous.

exception LowAccuracyWarning
Warns that results may possibly have low accuracy.

Logging

The library uses python's logging module to keep a history of what it has been doing. In particular, every wrapped Gromacs command logs its command line (including piped input) to the log file (configured in `gromacs.config.logfilename`). This facilitates debugging or simple re-use of command lines for very quick and dirty work. The logging facilty appends to the log file and time-stamps every entry. See `gromacs.config` for more details on configuration.

Version

The package version can be queried with the `gromacs.get_version()` function.

get_version()
Return current package version as a string.

get_version_tuple()
Return current package version as a tuple (*MAJOR*, *MINOR*, *PATCHLEVEL*).

1.3.2 Gromacs core modules

This section documents the modules, classes, and functions on which the other parts of the package rely. The information is probably mostly relevant to anyone who wants to extend the package.

`gromacs.core` – Core functionality

Here the basic command class `GromacsCommand` is defined. All Gromacs command classes in `gromacs.tools` are automatically generated from it.

class Command(**args, **kwargs*)
Wrap simple script or command.

Set up the command class.

The arguments can always be provided as standard positional arguments such as

```
"-c", "config.conf", "-o", "output.dat", "--repeats=3", "-v",
"input.dat"
```

In addition one can also use keyword arguments such as

```
c="config.conf", o="output.dat", repeats=3, v=True
```

These are automatically transformed appropriately according to simple rules:

- Any single-character keywords are assumed to be POSIX-style options and will be prefixed with a single dash and the value separated by a space.
- Any other keyword is assumed to be a GNU-style long option and thus will be prefixed with two dashes and the value will be joined directly with an equals sign and no space.

If this does not work (as for instance for the options of the UNIX `find` command) then provide options and values in the sequence of positional arguments.

`__call__`(**args, **kwargs*)

Run command with the given arguments:

```
rc,stdout,stderr = command(*args, input=None, **kwargs)
```

All positional parameters *args and all gromacs **kwargs are passed on to the Gromacs command. input and output keywords allow communication with the process via the python subprocess module.

Arguments

input [string, sequence] to be fed to the process' standard input; elements of a sequence are concatenated with newlines, including a trailing one [`None`]

stdin `None` or automatically set to `PIPE` if input given [`None`]

stdout how to handle the program's stdout stream [`None`]

filehandle anything that behaves like a file object

`None` or `True` to see output on screen

`False` or `PIPE` returns the output as a string in the stdout parameter

stderr how to handle the stderr stream [`STDOUT`]

`STDOUT` merges standard error with the standard out stream

`False` or `PIPE` returns the output as a string in the stderr return parameter

`None` or `True` keeps it on stderr (and presumably on screen)

All other kwargs are passed on to the Gromacs tool.

Returns The shell return code rc of the command is always returned. Depending on the value of output, various strings are filled with output from the command.

Notes By default, the process stdout and stderr are merged.

In order to chain different commands via pipes one must use the special `PopenWithInput` object (see `GromacsCommand.Popen()` method) instead of the simple call described here and first construct the pipeline explicitly and then call the `PopenWithInput.communicate()` method.

`STDOUT` and `PIPE` are objects provided by the `subprocess` module. Any python stream can be provided and manipulated. This allows for chaining of commands. Use

```
from subprocess import PIPE, STDOUT
```

when requiring these special streams (and the special boolean switches `True`/`False` cannot do what you need.)

(TODO: example for chaining commands)

run (**args, **kwargs*)
Run the command; args/kwargs are added or replace the ones given to the constructor.

transform_args (**args, **kwargs*)
Transform arguments and return them as a list suitable for Popen.

Popen (**args, **kwargs*)
Returns a special Popen instance (`PopenWithInput`).

The instance has its input pre-set so that calls to `communicate()` will not need to supply input. This is necessary if one wants to chain the output from one command to an input from another.

Todo Write example.

help (*long=False*)
Print help; same as using `?` in `ipython`. long=True also gives call signature.

command_name
Derive a class from command; typically one only has to set *command_name* to the name of the script or executable. The full path is required if it cannot be found by searching **PATH**.

class **GromacsCommand** (**args, **kwargs*)
Base class for wrapping a g_* command.

Limitations: User must have sourced `GMXRC` so that the python script can inherit the environment and find the gromacs programs.

The class doc string is dynamically replaced by the documentation of the gromacs command when an instance is created.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

`Popen`(**args, **kwargs*)
Returns a special Popen instance (`PopenWithInput`).

The instance has its input pre-set so that calls to `communicate()` will not need to supply input. This is necessary if one wants to chain the output from one command to an input from another.

Todo Write example.

`commandline`(**args, **kwargs*)
Returns the commandline that run() uses (without pipes).

`failuremodes`
Available failure modes.

`gmxdoc`
Usage for the underlying Gromacs tool (cached).

`help`(*long=False*)
Print help; same as using `?` in `ipython`. long=True also gives call signature.

run (**args, **kwargs*)
Run the command; args/kwargs are added or replace the ones given to the constructor.

transform_args (**args, **kwargs*)
Combine arguments and turn them into gromacs tool arguments.

class **PopenWithInput** (**args, **kwargs*)
Popen class that knows its input.

1.Set up the instance, including all the input it shoould receive.

2.Call `PopenWithInput.communicate()` later.

Note: Some versions of python have a bug in the subprocess module (issue 5179) which does not clean up open file descriptors. Eventually code (such as this one) fails with the error:

OSError: [Errno 24] Too many open files

A weak workaround is to increase the available number of open file descriptors with `ulimit -n 2048` and run analysis in different scripts.

Initialize with the standard `subprocess.Popen` arguments.

Keywords

input string that is piped into the command

communicate (*use_input=True*)
Run the command, using the input that was set up on __init__ (for *use_input* = `True`)

`gromacs.config` – Configuration for GromacsWrapper

The config module provides configurable options for the whole package; eventually it might grow into a sophisticated configuration system such as matplotlib's rc system but right now it mostly serves to define which gromacs tools and other scripts are offered in the package and where template files are located. If the user wants to change anything they will still have to do it here in source until a better mechanism with rc files has been implemented.

User-supplied templates are stored under `gromacs.config.configdir`. Eventually this will also contain the configuration options currently hard-coded in `gromacs.config`.

configdir
Directory to store user templates and rc files. The default value is `~/.gromacswrapper`.

path
Search path for user queuing scripts and templates. The internal package-supplied templates are always searched last via `gromacs.config.get_templates()`. Modify `gromacs.config.path` directly in order to customize the template and qscript searching. By default it has the value `['.', qscriptdir, templatesdir]`. (Note that it is not a good idea to have template files and qscripts with the same name as they are both searched on the same path.)

The user should execute `gromacs.config.setup()` at least once to prepare the user configurable area in their home directory:

```
import gromacs
gromacs.config.setup()
```

Logging

Gromacs commands log their invocation to a log file; typically at loglevel *INFO* (see the python logging module for details).

`logfilename`
File name for the log file; all gromacs command and many utility functions (e.g. in `gromacs.cbook` and `gromacs.setup`) append messages there. Warnings and errors are also recorded here. The default is *gromacs.log*.

`loglevel_console`
The default loglevel that is still printed to the console.

`loglevel_file`
The default loglevel that is still written to the `logfilename`.

Gromacs tools and scripts

`load_*` variables are lists that contain instructions to other parts of the code which packages and scripts should be wrapped.

`load_tools`
Python list of all tool file names. Automatically filled from `gmx_tools` and `gmx_extra_tools`, depending on the values in `gmx_tool_groups`.

`load_scripts`
3rd party analysis scripts and tools; this is a list of triplets of

(*script name/path*, *command name*, *doc string*)

(See the source code for examples.)

`load_tools` is populated by listing `gmx_*` tool group variables in `gmx_tool_groups`.

`gmx_tool_groups`
List of the variables in gromacs.tools that should be loaded. Possible values: *gmx_tools*, *gmx_extra_tools*. Right now these are variable names in `gromacs.config`, referencing `gromacs.config.gmx_tools` and `gromacs.config.gmx_extra_tools`.

The tool groups variables are strings that contain white-space separated file names of Gromacs tools. These lists determine which tools are made available as classes in `gromacs.tools`.

`gmx_tools`
Contains the file names of all Gromacs tools for which classes are generated. Editing this list has only an effect when the package is reloaded. If you want additional tools then add the, to the source (`config.py`) or derive new classes manually from `gromacs.core.GromacsCommand`. (Eventually, this functionality will be in a per-user configurable file.) The current list was generated from Gromacs 4.0.99 (git). Removed (because of various issues)

- g_kinetics

`gmx_extra_tools`
Additional gromacs tools (add *gmx_extra_tools* to `gromacs.config.gmx_tool_groups` to enable them, provided the binaries have been provided on the **PATH**).

Location of template files

Template variables list files in the package that can be used as templates such as run input files. Because the package can be a zipped egg we actually have to unwrap these files at this stage but this is completely transparent to the user.

`qscriptdir`
Directory to store user supplied queuing system scripts. The default value is `~/.gromacswrapper/qscripts`.

`templatesdir`
Directory to store user supplied template files such as mdp files. The default value is `~/.gromacswrapper/templates`.

`templates`
GromacsWrapper comes with a number of templates for run input files and queuing system scripts. They are provided as a convenience and examples but **WITHOUT ANY GUARANTEE FOR CORRECTNESS OR SUITABILITY FOR ANY PURPOSE**.

All template filenames are stored in `gromacs.config.templates`. Templates have to be extracted from the GromacsWrapper python egg file because they are used by external code: find the actual file locations from this variable.

Gromacs mdp templates

These are supplied as examples and there is **NO GUARANTEE THAT THEY PRODUCE SENSIBLE OUTPUT** — check for yourself! Note that only existing parameter names can be modified with `gromacs.cbook.edit_mdp()` at the moment; if in doubt add the parameter with its gromacs default value (or empty values) and modify later with `edit_mdp()`.

The safest bet is to use one of the `mdout.mdp` files produced by `gromacs.grompp()` as a template as this mdp contains all parameters that are legal in the current version of Gromacs.

Queuing system templates

The queing system scripts are highly specific and you will need to add your own into `gromacs.config.qscriptdir`. See `gromacs.qsub` for the format and how these files are processed.

`qscript_template`
The default template for SGE/PBS run scripts.

`setup``()`
Create the directories in which the user can store template and config files.

This function can be run repeatedly without harm.

Accessing configuration data

The following functions can be used to access configuration data. Note that files are searched first with their full filename, then in all directories listed in `gromacs.config.path`, and finally within the package itself.

`get_template``(`*t*`)`
Find template file *t* and return its real path.

t can be a single string or a list of strings. A string should be one of

1.a relative or absolute path,

2.a file in one of the directories listed in `gromacs.config.path`, 3.a filename in the package template directory (defined in the template dictionary `gromacs.config.templates`) or

4.a key into `templates`.

The first match (in this order) is returned. If the argument is a single string then a single string is returned, otherwise a list of strings.

Arguments *t* : template file or key (string or list of strings)

Returns os.path.realpath(*t*) (or a list thereof)

Raises `ValueError` if no file can be located.

`get_templates`(*t*)

Find template file(s) *t* and return their real paths.

t can be a single string or a list of strings. A string should be one of

1.a relative or absolute path,

2.a file in one of the directories listed in `gromacs.config.path`,

3.a filename in the package template directory (defined in the template dictionary `gromacs.config.templates`) or

4.a key into `templates`.

The first match (in this order) is returned for each input argument.

Arguments *t* : template file or key (string or list of strings)

Returns list of os.path.realpath(*t*)

Raises `ValueError` if no file can be located.

`gromacs.formats` – Accessing various files

This module contains classes that represent data files on disk. Typically one creates an instance and

- reads from a file using a `read()` method, or
- populates the instance (in the simplest case with a `set()` method) and the uses the `write()` method to write the data to disk in the appropriate format.

For function data there typically also exists a `plot()` method which produces a graph (using matplotlib).

The module defines some classes that are used in other modules; they do *not* make use of `gromacs.tools` or `gromacs.cbook` and can be safely imported at any time.

Classes

class XVG(*filename=None, names=None, permissive=False, **kwargs*)

Class that represents the numerical data in a grace xvg file.

All data must be numerical. `NAN` and `INF` values are supported via python's `float()` builtin function.

The `array` attribute can be used to access the the array once it has been read and parsed. The `ma` attribute is a numpy masked array (good for plotting).

Conceptually, the file on disk and the XVG instance are considered the same data. Whenever the filename for I/O (`XVG.read()` and `XVG.write()`) is changed then the filename associated with the instance is also changed to reflect the association between file and instance.

With the *permissive* = `True` flag one can instruct the file reader to skip unparseable lines. In this case the line numbers of the skipped lines are stored in `XVG.corrupted_lineno`.

A number of attributes are defined to give quick access to simple statistics such as

- `mean`: mean of all data columns
- `std`: standard deviation
- `min`: minimum of data
- `max`: maximum of data
- `error`: error on the mean, taking correlation times into account (see also `XVG.set_correlparameters()`)
- `tc`: correlation time of the data (assuming a simple exponential decay of the fluctuations around the mean)

These attributes are numpy arrays that correspond to the data columns, i.e. :attr:‘XVG.array‘[1:].

Note:

- Only simple XY or NXY files are currently supported, *not* Grace files that contain multiple data sets separated by ‘&’.
- Any kind of formatting (i.e. **xmgrace** commands) is discarded.

Initialize the class from a xvg file.

Arguments

filename is the xvg file; it can only be of type XY or NXY. If it is supplied then it is read and parsed when `XVG.array` is accessed.

names optional labels for the columns (currently only written as comments to file); string with columns separated by commas or a list of strings

permissive `False` raises a `ValueError` and logs and errior when encountering data lines that it cannot parse. `True` ignores those lines and logs a warning—this is a risk because it might read a corrupted input file [`False`]

`array`

Represent xvg data as a (cached) numpy array.

The array is returned with column-first indexing, i.e. for a data file with columns X Y1 Y2 Y3 ... the array a will be a[0] = X, a[1] = Y1, ...

`error`

Error on the mean of the data, taking the correlation time into account.

See Frenkel and Smit, Academic Press, San Diego 2002, p526:

error = sqrt(2*tc*acf[0]/T)

where acf() is the autocorrelation function of the fluctuations around the mean y-<y>, tc is the correlation time, and T the total length of the simulation.

`errorbar`(***kwargs*)

Quick hack: errorbar plot.

Set columns to select [x, y, dy].

`ma`

Represent data as a masked array.

The array is returned with column-first indexing, i.e. for a data file with columns X Y1 Y2 Y3 ... the array a will be a[0] = X, a[1] = Y1, ...

inf and nan are filtered via `numpy.isfinite()`.

max
Maximum of the data columns.

mean
Mean value of all data columns.

min
Minimum of the data columns.

parse()
Read and cache the file as a numpy array.

The array is returned with column-first indexing, i.e. for a data file with columns X Y1 Y2 Y3 ... the array a will be a[0] = X, a[1] = Y1, ...

plot(***kwargs*)
Plot xvg file data.

The first column of the data is always taken as the abscissa X. Additional columns are plotted as ordinates Y1, Y2, ...

In the special case that there is only a single column then this column is plotted against the index, i.e. (N, Y).

Keywords

columns [list] Select the columns of the data to be plotted; the list is used as a numpy.array extended slice. The default is to use all columns. Columns are selected *after* a transform.

transform [function] function `transform(array) -> array` which transforms the original array; must return a 2D numpy array of shape [X, Y1, Y2, ...] where X, Y1, ... are column vectors. By default the transformation is the identity [`lambda x: x`].

maxpoints [int] limit the total number of data points; matplotlib has issues processing png files with >100,000 points and pdfs take forever to display. Set to `None` if really all data should be displayed. At the moment we simply subsample the data at regular intervals. [10000]

kwargs All other keyword arguments are passed on to `pylab.plot()`.

read(*filename=None*)
Read and parse xvg file *filename*.

set(*a*)
Set the *array* data from *a* (i.e. completely replace).

No sanity checks at the moment...

set_correlparameters(***kwargs*)
Set and change the parameters for calculations involving correlation functions.

Keywords

nstep only process every *nstep* data point to speed up the FFT; if left empty a default is chosen that produces roughly 25,000 data points (or whatever is set in `XVG.ncorrel`).

force force recalculating correlation data even if cached values are available

kwargs see `numkit.timeseries.tcorrel()` for other options

std
Standard deviation from the mean of all data columns.

tc
Correlation time of the data.

See `XVG.error()` for details.

write (*filename=None*)
Write array to xvg file *filename* in NXY format.

Note: Only plain files working at the moment, not compressed.

class NDX (*filename=None, **kwargs*)
Gromacs index file.

Represented as a ordered dict where the keys are index group names and values are numpy arrays of atom numbers.

Use the `NDX.read()` and `NDX.write()` methods for I/O. Access groups by name via the `NDX.get()` and `NDX.set()` methods.

Alternatively, simply treat the `NDX` instance as a dictionary. Setting a key automatically transforms the new value into a integer 1D numpy array (*not* a set, as would be the **make_ndx** behaviour).

Note: The index entries themselves are ordered and can contain duplicates so that output from NDX can be easily used for **g_dih** and friends. If you need set-like behaviour you will have do use `gromacs.formats.uniqueNDX` or `gromacs.cbook.IndexBuilder` (which uses **make_ndx** throughout).

Example

Read index file, make new group and write to disk:

```
ndx = NDX()
ndx.read('system.ndx')
print ndx['Protein']
ndx['my_group'] = [2, 4, 1, 5]    # add new group
ndx.write('new.ndx')
```

Or quicker (replacing the input file `system.ndx`):

```
ndx = NDX('system')           # suffix .ndx is automatically added
ndx['chi1'] = [2, 7, 8, 10]
ndx.write()
```

format
standard ndx file format: '%6d'

get (*name*)
Return index array for index group *name*.

groups
Return a list of all groups.

ncol
standard ndx file format: 15 columns

ndxlist
Return a list of groups in the same format as `gromacs.cbook.get_ndx_groups()`.

Format: [{'name': group_name, 'natoms': number_atoms, 'nr': # group_number}, ...]

read (*filename=None*)
Read and parse index file *filename*.

set (*name, value*)
Set or add group *name* as a 1D numpy array.

size (*name*)
Return number of entries for group *name*.

sizes
Return a dict with group names and number of entries,

write (*filename=None, ncol=15, format='%6d'*)
Write index file to *filename* (or overwrite the file that the index was read from)

class **uniqueNDX** (*filename=None, **kwargs*)
Index that behaves like make_ndx, i.e. entries behaves as sets, not lists.

The index lists behave like sets: - adding sets with '+' is equivalent to a logical OR: x + y == "x | y" - subtraction '-' is AND: x - y == "x & y" - see `join()` for ORing multiple groups (x+y+z+...)

Example :: I = uniqueNDX('system.ndx') I['SOLVENT'] = I['SOL'] + I['NA+'] + I['CL-']

join (**groupnames*)
Return an index group that contains atoms from all *groupnames*.

The method will silently ignore any groups that are not in the index.

Example

Always make a solvent group from water and ions, even if not all ions are present in all simulations:

```
I['SOLVENT'] = I.join('SOL', 'NA+', 'K+', 'CL-')
```

class **GRO** (***kwargs*)
Class that represents a GROMOS (gro) structure file.

File format:

(Not implemented yet)

read (*filename=None*)
Read and parse index file *filename*.

`gromacs.utilities` – Helper functions and classes

The module defines some convenience functions and classes that are used in other modules; they do *not* make use of `gromacs.tools` or `gromacs.cbook` and can be safely imported at any time.

Classes

`FileUtils` provides functions related to filename handling. It can be used as a base or mixin class. The `gromacs.analysis.Simulation` class is derived from it.

class **FileUtils** ()
Mixin class to provide additional file-related capabilities.

check_file_exists (*filename, resolve='exception', force=None*)
If a file exists then continue with the action specified in `resolve`.

`resolve` must be one of

"ignore" always return `False`

"indicate" return `True` if it exists

"warn" indicate and issue a `UserWarning`

"exception" raise `IOError` if it exists

Alternatively, set *force* for the following behaviour (which ignores *resolve*):

`True` same as *resolve* = "ignore" (will allow overwriting of files)

`False` same as *resolve* = "exception" (will prevent overwriting of files)

`None` ignored, do whatever *resolve* says

`default_extension`
Default extension for files read/written by this class.

`filename` (*filename=None, ext=None, set_default=False, use_my_ext=False*)
Supply a file name for the class object.

Typical uses:

```
fn = filename()             ---> <default_filename>
fn = filename('name.ext')   ---> 'name'
fn = filename(ext='pickle') ---> <default_filename>'.pickle'
fn = filename('name.inp','pdf') --> 'name.pdf'
fn = filename('foo.pdf',ext='png',use_my_ext=True) --> 'foo.pdf'
```

The returned filename is stripped of the extension (`use_my_ext=False`) and if provided, another extension is appended. Chooses a default if no filename is given.

Raises a `ValueError` exception if no default file name is known.

If `set_default=True` then the default filename is also set.

`use_my_ext=True` lets the suffix of a provided filename take priority over a default `ext` tension.

`infix_filename` (*name, default, infix, ext=None*)
Unless *name* is provided, insert *infix* before the extension *ext* of *default*.

class **`AttributeDict`** ()
A dictionary with pythonic access to keys as attributes — useful for interactive work.

class **`Timedelta`** ()
Extension of `datetime.timedelta`.

Provides attributes ddays, dhours, dminutes, dseconds to measure the delta in normal time units.

ashours gives the total time in fractional hours.

Functions

Some additional convenience functions that deal with files and directories:

`openany` (*directory, [mode='r']*)
Context manager to open a compressed (bzip2, gzip) or plain file (uses `anyopen()`).

`anyopen` (*datasource, mode='r'*)
Open datasource (gzipped, bzipped, uncompressed) and return a stream.

Arguments

- *datasource*: a file or a stream

•*mode*: 'r' or 'w'

realpath (**args*)
Join all args and return the real path, rooted at /.

Returns `None` if any of the args is none.

in_dir (*directory, [create=True]*)
Context manager to execute a code block in a directory.

•The *directory* is created if it does not exist (unless *create* = `False` is set)

•At the end or after an exception code always returns to the directory that was the current directory before entering the block.

find_first (*filename, suffices=None*)
Find first *filename* with a suffix from *suffices*.

Arguments

filename base filename; this file name is checked first

suffices list of suffices that are tried in turn on the root of *filename*; can contain the ext separator (`os.path.extsep`) or not

Returns The first match or `None`.

withextsep (*extensions*)
Return list in which each element is guaranteed to start with `os.path.extsep`.

Functions that improve list processing and which do *not* treat strings as lists:

iterable (*obj*)
Returns `True` if *obj* can be iterated over and is *not* a string.

asiterable (*obj*)
Returns obj so that it can be iterated over; a string is *not* treated as iterable

Functions that help handling Gromacs files:

unlink_f (*path*)
Unlink path but do not complain if file does not exist.

unlink_gmx (**args*)
Unlink (remove) Gromacs file(s) and all corresponding backups.

unlink_gmx_backups (**args*)
Unlink (rm) all backup files corresponding to the listed files.

number_pdbs (**args, **kwargs*)
Rename pdbs x1.pdb ... x345.pdb –> x0001.pdb ... x0345.pdb

Arguments

- *args*: filenames or glob patterns (such as "pdb/md*.pdb")
- *format*: format string including keyword *num* ["%(num)04d"]

Functions that make working with matplotlib easier:

activate_subplot (*numPlot*)
Make subplot *numPlot* active on the canvas.

Use this if a simple `subplot(numRows, numCols, numPlot)` overwrites the subplot instead of activating it.

`remove_legend`(*ax=None*)
Remove legend for axes or gca.

See http://osdir.com/ml/python.matplotlib.general/2005-07/msg00285.html

Miscellaneous functions:

`convert_aa_code`(*x*)
Converts between 3-letter and 1-letter amino acid codes.

Data

`amino_acid_codes`
translation table for 1-letter codes –> 3-letter codes .. Note: This does not work for HISB and non-default charge state aa!

`gromacs.tools` – Gromacs commands classes

A Gromacs command class can be thought of as a factory function that produces an instance of a gromacs command (`gromacs.core.GromacsCommand`) with initial default values.

By convention, a class has the capitalized name of the corresponding Gromacs tool; dots are replaced by underscores to make it a valid python identifier.

The list of Gromacs tools to be loaded is configured in `gromacs.config.gmx_tool_groups`.

It is also possible to extend the basic commands and patch in additional functionality. For example, the `GromacsCommandMultiIndex` class makes a command accept multiple index files and concatenates them on the fly; the behaviour mimics Gromacs' "multi-file" input that has not yet been enabled for all tools.

class `GromacsCommandMultiIndex`(***kwargs*)
Initialize instance.

1. Sets up the combined index file.
2. Inititialize `GromacsCommand` with the new index file.

See the documentation for `gromacs.core.GromacsCommand` for details.

`run`(**args, **kwargs*)
Run the command; make a combined multi-index file if necessary.

`_fake_multi_ndx`(***kwargs*)
Combine multiple index file into a single one and return appropriate kwargs.

Calling the method combines multiple index files into a a single temporary one so that Gromacs tools that do not (yet) support multi file input for index files can be used transparently as if they did.

If a temporary index file is required then it is deleted once the object is destroyed.

Returns The method returns the input keyword arguments with the necessary changes to use the temporary index files.

Keywords Only the listed keywords have meaning for the method:

n [filename or list of filenames] possibly multiple index files; *n* is replaced by the name of the temporary index file.

s [filename] structure file (tpr, pdb, ...) or `None`; if a structure file is supplied then the Gromacs default index groups are automatically added to the temporary indexs file.

Example Used in derived classes that replace the standard `run()` (or `__init__()`) methods with something like:

```
def run(self,*args,**kwargs):
    kwargs = self._fake_multi_ndx(**kwargs)
    return super(G_mindist, self).run(*args, **kwargs)
```

`__del__`()
Clean up temporary multi-index files if they were used.

Example

In this example we create two instances of the `gromacs.tools.Trjconv` command (which runs the Gromacs `trjconv` command):

```
import gromacs.tools as tools

trjconv = tools.Trjconv()
trjconv_compact = tools.Trjconv(ur='compact', center=True, boxcenter='tric', pbc='mol',
                                input=('protein','system'),
                                doc="Returns a compact representation of the system centered on the p
```

The first one, `trjconv`, behaves as the standard commandline tool but the second one, `trjconv_compact`, will by default create a compact representation of the input data by taking into account the shape of the unit cell. Of course, the same effect can be obtained by providing the corresponding arguments to `trjconv` but by naming the more specific command differently one can easily build up a library of small tools that will solve a specifi, repeatedly encountered problem reliably. This is particularly helpful when doing interactive work.

Gromacs tools

The documentation of all wrapped gromacs tools is auto-generated and can be found in *Wrapped Gromacs Tools*. Here only two examples (`Mdrun` and `GridMAT_MD`) are shown.

class `Mdrun`(**args, **kwargs*)
Gromacs tool 'mdrun'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `GridMAT_MD`(**args, **kwargs*)

External tool 'GridMAT-MD.pl'

GridMAT-MD: A Grid-based Membrane Analysis Tool for use with Molecular Dynamics.

This `GridMAT-MD` is a patched version of the original `GridMAT-MD.pl` v1.0.2, written by WJ Allen, JA Lemkul and DR Bevan. The original version is available from the GridMAT-MD home page,

Please cite

W. J. Allen, J. A. Lemkul, and D. R. Bevan. (2009) "GridMAT-MD: A Grid-based Membrane Analysis Tool for Use With Molecular Dynamics." J. Comput. Chem. 30 (12): 1952-1958.

when using this programme.

Usage:

class `GridMAT_MD`(*config, [structure]*)

Arguments

- *config* : See the original documentation for a description for the configuration file.
- *structure* : A gro or pdb file that overrides the value for *bilayer* in the configuration file.

.

Set up the command class.

The arguments can always be provided as standard positional arguments such as

```
"-c", "config.conf", "-o", "output.dat", "--repeats=3", "-v",
"input.dat"
```

In addition one can also use keyword arguments such as

```
c="config.conf", o="output.dat", repeats=3, v=True
```

These are automatically transformed appropriately according to simple rules:

- Any single-character keywords are assumed to be POSIX-style options and will be prefixed with a single dash and the value separated by a space.
- Any other keyword is assumed to be a GNU-style long option and thus will be prefixed with two dashes and the value will be joined directly with an equals sign and no space.

If this does not work (as for instance for the options of the UNIX `find` command) then provide options and values in the sequence of positional arguments.

Wrapped Gromacs tools

This is the auto-generated list of all Gromacs tools that were available when this documentation was built. They are part of the *Gromacs core modules*.

`gromacs.tools` – Gromacs commands classes

A Gromacs command class can be thought of as a factory function that produces an instance of a gromacs command (`gromacs.core.GromacsCommand`) with initial default values.

By convention, a class has the capitalized name of the corresponding Gromacs tool; dots are replaced by underscores to make it a valid python identifier.

The list of Gromacs tools to be loaded is configured in `gromacs.config.gmx_tool_groups`.

It is also possible to extend the basic commands and patch in additional functionality. For example, the `GromacsCommandMultiIndex` class makes a command accept multiple index files and concatenates them on the fly; the behaviour mimics Gromacs' "multi-file" input that has not yet been enabled for all tools.

class `GromacsCommandMultiIndex`(***kwargs*)

Initialize instance.

1. Sets up the combined index file.
2. Inititialize `GromacsCommand` with the new index file.

See the documentation for `gromacs.core.GromacsCommand` for details.

`run`(**args, **kwargs*)

Run the command; make a combined multi-index file if necessary.

_fake_multi_ndx(***kwargs*)

Combine multiple index file into a single one and return appropriate kwargs.

Calling the method combines multiple index files into a a single temporary one so that Gromacs tools that do not (yet) support multi file input for index files can be used transparently as if they did.

If a temporary index file is required then it is deleted once the object is destroyed.

Returns The method returns the input keyword arguments with the necessary changes to use the temporary index files.

Keywords Only the listed keywords have meaning for the method:

n [filename or list of filenames] possibly multiple index files; *n* is replaced by the name of the temporary index file.

s [filename] structure file (tpr, pdb, ...) or `None`; if a structure file is supplied then the Gromacs default index groups are automatically added to the temporary indexs file.

Example Used in derived classes that replace the standard `run()` (or `__init__()`) methods with something like:

```
def run(self,*args,**kwargs):
    kwargs = self._fake_multi_ndx(**kwargs)
    return super(G_mindist, self).run(*args, **kwargs)
```

__del__()

Clean up temporary multi-index files if they were used.

Example In this example we create two instances of the `gromacs.tools.Trjconv` command (which runs the Gromacs `trjconv` command):

```
import gromacs.tools as tools

trjconv = tools.Trjconv()
trjconv_compact = tools.Trjconv(ur='compact', center=True, boxcenter='tric', pbc='mol',
                                input=('protein','system'),
                                doc="Returns a compact representation of the system centered on the p
```

The first one, `trjconv`, behaves as the standard commandline tool but the second one, `trjconv_compact`, will by default create a compact representation of the input data by taking into account the shape of the unit cell. Of course, the same effect can be obtained by providing the corresponding arguments to `trjconv` but by naming the more specific command differently one can easily build up a library of small tools that will solve a specifi, repeatedly encountered problem reliably. This is particularly helpful when doing interactive work.

Gromacs tools

class G_spatial(**args*, ***kwargs*)

Gromacs tool 'g_spatial'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Sigeps`(**args, **kwargs*)

Gromacs tool 'sigeps'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class A_gridcalc(**args, **kwargs*)
Gromacs tool 'a_gridcalc'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_filter`(**args, **kwargs*)

Gromacs tool 'g_filter'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

None just continue silently

doc [string] additional documentation []

class **`Protonate`**(**args, **kwargs*)

Gromacs tool 'protonate'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Genrestr`(**args, **kwargs*)

Gromacs tool 'genrestr'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Trjcat`(**args, **kwargs*)

Gromacs tool 'trjcat'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`G_helixorient`**(**args, **kwargs*)

Gromacs tool 'g_helixorient'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`G_gyrate`**(**args, **kwargs*)

Gromacs tool 'g_gyrate'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class G_densmap(**args*, ***kwargs*)

Gromacs tool 'g_densmap'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_zcoord`(**args, **kwargs*)

Gromacs tool 'g_zcoord'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_nmens` (**args, **kwargs*)

Gromacs tool 'g_nmens'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Grompp`(**args, **kwargs*)

Gromacs tool 'grompp'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_angle`(**args, **kwargs*)

Gromacs tool 'g_angle'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Trjconv`(**args, **kwargs*)

Gromacs tool 'trjconv'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`G_rama`**(**args, **kwargs*)

Gromacs tool 'g_rama'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **G_sgangle**(**args*, ***kwargs*)
Gromacs tool 'g_sgangle'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`G_anaeig`**(**args*, ***kwargs*)

Gromacs tool 'g_anaeig'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Genion`(**args, **kwargs*)

Gromacs tool 'genion'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_sham`(**args, **kwargs*)

Gromacs tool 'g_sham'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_enemat`(**args, **kwargs*)

Gromacs tool 'g_enemat'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_density`(**args*, ***kwargs*)

Gromacs tool 'g_density'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_nmeig`(**args, **kwargs*)

Gromacs tool 'g_nmeig'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class Tpbconv (**args, **kwargs*)

Gromacs tool 'tpbconv'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_tcaf`(**args, **kwargs*)

Gromacs tool 'g_tcaf'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Mdrun_d`(**args, **kwargs*)

Gromacs tool 'mdrun_d'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Genbox`(**args*, ***kwargs*)

Gromacs tool 'genbox'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_rms`(**args, **kwargs*)

Gromacs tool 'g_rms'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_current`(**args, **kwargs*)

Gromacs tool 'g_current'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_flux`(**args, **kwargs*)

Gromacs tool 'g_flux'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class G_dielectric(**args, **kwargs*)

Gromacs tool 'g_dielectric'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_msd`(**args, **kwargs*)

Gromacs tool 'g_msd'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_disre`(**args, **kwargs*)

Gromacs tool 'g_disre'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_analyze`(**args, **kwargs*)

Gromacs tool 'g_analyze'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Mdrun`(**args, **kwargs*)

Gromacs tool 'mdrun'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_confrms`(**args, **kwargs*)

Gromacs tool 'g_confrms'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`Trjorder`**(**args, **kwargs*)

Gromacs tool 'trjorder'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class G_principal(**args, **kwargs*)

Gromacs tool 'g_principal'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`G_hbond`**(**args, **kwargs*)

Gromacs tool 'g_hbond'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Anadock` (**args, **kwargs*)

Gromacs tool 'anadock'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_rdf`(**args, **kwargs*)

Gromacs tool 'g_rdf'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_sdf`(**args, **kwargs*)

Gromacs tool 'g_sdf'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Gmxdump`(**args*, ***kwargs*)

Gromacs tool 'gmxdump'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`G_h2order`**(**args, **kwargs*)

Gromacs tool 'g_h2order'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class G_traj(**args, **kwargs*)

Gromacs tool 'g_traj'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

None just continue silently

doc [string] additional documentation []

class G_count (**args, **kwargs*)

Gromacs tool 'g_count'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Gmxcheck`(**args, **kwargs*)

Gromacs tool 'gmxcheck'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_mindist`(***kwargs*)

Gromacs tool 'g_mindist' (with patch to handle multiple ndx files).

Initialize instance.

1. Sets up the combined index file.
2. Inititialize `GromacsCommand` with the new index file.

See the documentation for `gromacs.core.GromacsCommand` for details.

class `G_sas`(**args, **kwargs*)

Gromacs tool 'g_sas'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value None then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a SyntaxError but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, -or translates to the illegal keyword or so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the run() method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a GromacsFailureWarning

None just continue silently

doc [string] additional documentation []

class G_nmtraj(**args, **kwargs*)

Gromacs tool 'g_nmtraj'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_bond`(**args, **kwargs*)

Gromacs tool 'g_bond'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class A_ri3dc (**args*, ***kwargs*)

Gromacs tool 'a_ri3Dc'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_covar`(**args*, ***kwargs*)

Gromacs tool 'g_covar'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Editconf`(**args, **kwargs*)

Gromacs tool 'editconf'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Pdb2gmx`(**args, **kwargs*)

Gromacs tool 'pdb2gmx'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_helix`(**args, **kwargs*)

Gromacs tool 'g_helix'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Luck`(**args, **kwargs*)

Gromacs tool 'luck'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Mk_angndx`(**args, **kwargs*)

Gromacs tool 'mk_angndx'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class G_mdmat (**args, **kwargs*)

Gromacs tool 'g_mdmat'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

None just continue silently

doc [string] additional documentation []

class Eneconv (**args, **kwargs*)

Gromacs tool 'eneconv'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_energy`(**args, **kwargs*)

Gromacs tool 'g_energy'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_clustsize`(**args, **kwargs*)

Gromacs tool 'g_clustsize'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `GridMAT_MD`(**args, **kwargs*)

External tool 'GridMAT-MD.pl'

GridMAT-MD: A Grid-based Membrane Analysis Tool for use with Molecular Dynamics.

This `GridMAT-MD` is a patched version of the original `GridMAT-MD.pl` v1.0.2, written by WJ Allen, JA Lemkul and DR Bevan. The original version is available from the GridMAT-MD home page,

Please cite

W. J. Allen, J. A. Lemkul, and D. R. Bevan. (2009) "GridMAT-MD: A Grid-based Membrane Analysis Tool for Use With Molecular Dynamics." J. Comput. Chem. 30 (12): 1952-1958.

when using this programme.

Usage:

class `GridMAT_MD`(*config, [structure]*)

Arguments

- *config* : See the original documentation for a description for the configuration file.
- *structure* : A gro or pdb file that overrides the value for *bilayer* in the configuration file.

.

Set up the command class.

The arguments can always be provided as standard positional arguments such as

```
"-c", "config.conf", "-o", "output.dat", "--repeats=3", "-v",
"input.dat"
```

In addition one can also use keyword arguments such as

```
c="config.conf", o="output.dat", repeats=3, v=True
```

These are automatically transformed appropriately according to simple rules:

- Any single-character keywords are assumed to be POSIX-style options and will be prefixed with a single dash and the value separated by a space.
- Any other keyword is assumed to be a GNU-style long option and thus will be prefixed with two dashes and the value will be joined directly with an equals sign and no space.

If this does not work (as for instance for the options of the UNIX `find` command) then provide options and values in the sequence of positional arguments.

class G_dipoles(**args, **kwargs*)

Gromacs tool 'g_dipoles'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_lie`(**args, **kwargs*)

Gromacs tool 'g_lie'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Xpm2ps` (**args, **kwargs*)

Gromacs tool 'xpm2ps'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_cluster`(**args, **kwargs*)

Gromacs tool 'g_cluster'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`G_wham`**(**args, **kwargs*)

Gromacs tool 'g_wham'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class G_vanhove (**args*, ***kwargs*)

Gromacs tool 'g_vanhove'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_rotacf`(**args*, ***kwargs*)

Gromacs tool 'g_rotacf'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_spol`(**args, **kwargs*)

Gromacs tool 'g_spol'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`Make_edi`**(**args, **kwargs*)

Gromacs tool 'make_edi'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_dist`(***kwargs*)

Gromacs tool 'g_dist' (with patch to handle multiple ndx files).

Initialize instance.

1. Sets up the combined index file.
2. Inititialize `GromacsCommand` with the new index file.

See the documentation for `gromacs.core.GromacsCommand` for details.

class `G_potential`(**args, **kwargs*)

Gromacs tool 'g_potential'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class **`G_ri3dc`**(**args, **kwargs*)

Gromacs tool 'g_ri3Dc'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class G_velacc (**args, **kwargs*)

Gromacs tool 'g_velacc'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `X2top`(**args, **kwargs*)

Gromacs tool 'x2top'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

None just continue silently

doc [string] additional documentation []

class G_polystat (**args, **kwargs*)

Gromacs tool 'g_polystat'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Wheel`(**args, **kwargs*)

Gromacs tool 'wheel'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_bundle`(**args, **kwargs*)

Gromacs tool 'g_bundle'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_order`(**args, **kwargs*)

Gromacs tool 'g_order'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_saltbr`(**args, **kwargs*)

Gromacs tool 'g_saltbr'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

- **'raise'** raises GromacsError if command fails
- **'warn'** issue a `GromacsFailureWarning`
- **`None`** just continue silently

doc [string] additional documentation []

class Do_dssp(**args, **kwargs*)

Gromacs tool 'do_dssp'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

None just continue silently

doc [string] additional documentation []

class G_dih (**args, **kwargs*)

Gromacs tool 'g_dih'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Make_ndx` (**args, **kwargs*)

Gromacs tool 'make_ndx'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_dyndom`(**args, **kwargs*)

Gromacs tool 'g_dyndom'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `Genconf`(**args, **kwargs*)

Gromacs tool 'genconf'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_morph`(**args, **kwargs*)

Gromacs tool 'g_morph'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class `G_chi`(**args, **kwargs*)

Gromacs tool 'g_chi'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

class G_sorient (**args, **kwargs*)

Gromacs tool 'g_sorient'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

None just continue silently

doc [string] additional documentation []

class G_rmsf (**args, **kwargs*)

Gromacs tool 'g_rmsf'.

Set up the command with gromacs flags as keyword arguments.

The following are generic instructions; refer to the Gromacs command usage information that should have appeared before this generic documentation.

As an example, a generic Gromacs command could use the following flags:

```
cmd = GromacsCommand('v', f=['md1.xtc','md2.xtc'], o='processed.xtc', t=200, ...)
```

which would correspond to running the command in the shell as

```
GromacsCommand -v -f md1.xtc md2.xtc -o processed.xtc -t 200
```

Gromacs command line arguments

Gromacs boolean switches (such as `-v`) are given as python positional arguments (`'v'`) or as keyword argument (`v=True`); note the quotes in the first case. Negating a boolean switch can be done with `'nov'`, `nov=True` or `v=False` (and even `nov=False` works as expected: it is the same as `v=True`).

Any Gromacs options that take parameters are handled as keyword arguments. If an option takes multiple arguments (such as the multi-file input `-f file1 file2 ...`) then the list of files must be supplied as a python list.

If a keyword has the python value `None` then it will *not* be added to the Gromacs command line; this allows for flexible scripting if it is not known in advance if an input file is needed. In this case the default value of the gromacs tool is used.

Keywords must be legal python keywords or the interpreter raises a `SyntaxError` but of course Gromacs commandline arguments are not required to be legal python. In this case "quote" the option with an underscore (_) and the underscore will be silently stripped. For instance, `-or` translates to the illegal keyword `or` so it must be underscore-quoted:

```
cmd(...., _or='mindistres.xvg')
```

Command execution

The command is executed with the `run()` method or by calling it as a function. The two next lines are equivalent:

```
cmd(...)
cmd.run(...)
```

When the command is run one can override options that were given at initialization or one can add additional ones. The same rules for supplying Gromacs flags apply as described above.

Non-Gromacs keyword arguments

The other keyword arguments (listed below) are not passed on to the Gromacs tool but determine how the command class behaves. They are only useful when instantiating a class. This is mostly of interest to developers.

Keywords

failure determines how a failure of the gromacs command is treated; it can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

`None` just continue silently

doc [string] additional documentation []

1.3.3 Gromacs building blocks

Building blocks are small classes or functions that can be freely combined in setup or analysis scripts or used interactively. These modules act as "library" for common tasks.

`gromacs.cbook` – Gromacs Cook Book

The `cbook` (cook book) module contains short recipes for tasks that are often repeated. In the simplest case this is just one of the gromacs tools with a certain set of default command line options.

By abstracting and collecting these invocations here, errors can be reduced and the code snippets can also serve as canonical examples for how to do simple things.

Miscellaneous canned Gromacs commands

Simple commands with new default options so that they solve a specific problem (see also Manipulating trajectories and structures):

`rmsd_backbone` (*[s="md.tpr", f="md.xtc", [...]]*)
Computes the RMSD of the "Backbone" atoms after fitting to the "Backbone" (including both translation and rotation).

Manipulating trajectories and structures

Standard invocations for manipulating trajectories.

`trj_compact` (*[s="md.tpr", f="md.xtc", o="compact.xtc", [...]]*)
Writes an output trajectory or frame with a compact representation of the system centered on the protein. It centers on the group "Protein" and outputs the whole "System" group.

`trj_xyfitted` (*[s="md.tpr", f="md.xtc", [...]]*)
Writes a trajectory centered and fitted to the protein in the XY-plane only.

This is useful for membrane proteins. The system *must* be oriented so that the membrane is in the XY plane. The protein backbone is used for the least square fit, centering is done for the whole protein., but this can be changed with the *input* = `('backbone', 'protein','system')` keyword.

Note: Gromacs 4.x only

trj_fitandcenter (*xy=False, **kwargs*)
Center everything and make a compact representation (pass 1) and fit the system to a reference (pass 2).

Keywords

s input structure file (tpr file required to make molecule whole)

f input trajectory

o output trajectory

input

A list with three groups. The default is ['backbone', 'protein','system']

The fit command uses all three (1st for least square fit, 2nd for centering, 3rd for output), the centered/make-whole stage use 2nd for centering and 3rd for output.

input1 If *input1* is supplied then *input* is used exclusively for the fitting stage (pass 2) and *input1* for the centering (pass 1).

n Index file used for pass 1 and pass 2.

n1 If *n1* is supplied then index *n1* is only used for pass 1 (centering) and *n* for pass 2 (fitting).

xy [boolean] If `True` then only do a rot+trans fit in the xy plane (good for membrane simulations); default is `False`.

kwargs All other arguments are passed to `Trjconv`.

Note that here we first center the protein and create a compact box, using `-pbc mol -ur compact -center -boxcenter tric` and write an intermediate xtc. Then in a second pass we perform a rotation+translation fit (or restricted to the xy plane if *xy* = `True` is set) on the intermediate xtc to produce the final trajectory. Doing it in this order has the disadvantage that the solvent box is rotating around the protein but the opposite order (with center/compact second) produces strange artifacts where columns of solvent appear cut out from the box—it probably means that after rotation the information for the periodic boundaries is not correct any more.

Most kwargs are passed to both invocations of `gromacs.tools.Trjconv` so it does not really make sense to use eg *skip*; in this case do things manually.

By default the *input* to the fit command is ('backbone', 'protein','system'); the compact command always uses the second and third group for its purposes or if this fails, prompts the user.

Both steps cannot performed in one pass; this is a known limitation of `trjconv`. An intermediate temporary XTC files is generated which should be automatically cleaned up unless bad things happened.

The function tries to honour the input/output formats. For instance, if you want trr output you need to supply a trr file as input and explicitly give the output file also a trr suffix.

Note: For big trajectories it can **take a very long time** and consume a **large amount of temporary diskspace**.

We follow the g_spatial documentation in preparing the trajectories:

```
trjconv -s a.tpr -f a.xtc -o b.xtc -center tric -ur compact -pbc none
trjconv -s a.tpr -f b.xtc -o c.xtc -fit rot+trans
```

cat (*prefix='md', dirname='.', partsdir='parts', fulldir='full', resolve_multi='pass'*)
Concatenate all parts of a simulation.

The xtc, trr, and edr files in *dirname* such as prefix.xtc, prefix.part0002.xtc, prefix.part0003.xtc, ... are

1.moved to the *partsdir* (under *dirname*)

2.concatenated with the Gromacs tools to yield prefix.xtc, prefix.trr, prefix.edr, prefix.gro (or prefix.md) in *dirname*

3.Store these trajectories in *fulldir*

Note: Trajectory files are *never* deleted by this function to avoid data loss in case of bugs. You will have to clean up yourself by deleting *dirname/partsdir*.

Symlinks for the trajectories are *not* handled well and break the function. Use hard links instead.

Warning: If an exception occurs when running this function then make doubly and triply sure where your files are before running this function again; otherwise you might **overwrite data**. Possibly you will need to manually move the files from *partsdir* back into the working directory *dirname*; this should onlu overwrite generated files so far but *check carefully*!

Keywords

prefix deffnm of the trajectories [md]

***resolve_multi”** how to deal with multiple “final” gro or pdb files: normally there should only be one but in case of restarting from the checkpoint of a finished simulation one can end up with multiple identical ones.

- “pass” : do nothing and log a warning
- **“guess”** [take prefix.pdb or prefix.gro if it exists, otherwise the one of] prefix.partNNNN.gro|pdb with the highes NNNN

dirname change to *dirname* and assume all tarjectories are located there [.]

partsdir directory where to store the input files (they are moved out of the way); *partsdir* must be manually deleted [parts]

fulldir directory where to store the final results [full]

class `Frames` (*structure, trj, maxframes=None, format='pdb', **kwargs*)

A iterator that transparently provides frames from a trajectory.

The iterator chops a trajectory into individual frames for analysis tools that only work on separate structures such as `gro` or `pdb` files. Instead of turning the whole trajectory immediately into pdb files (and potentially filling the disk), the iterator can be instructed to only provide a fixed number of frames and compute more frames when needed.

Note: Setting a limit on the number of frames on disk can lead to longish waiting times because `trjconv` must re-seek to the middle of the trajectory and the only way it can do this at the moment is by reading frames sequentially. This might still be preferrable to filling up a disk, though.

Warning: The *maxframes* option is not implemented yet; use the *dt* option or similar to keep the number of frames manageable.

Set up the Frames iterator.

Arguments

structure name of a structure file (tpr, pdb, ...)

trj name of the trajectory (xtc, trr, ...)

format output format for the frames, eg “pdb” or “gro” [pdb]

maxframes [int] maximum number of frames that are extracted to disk at one time; set to `None` to extract the whole trajectory at once. [`None`]

kwargs All other arguments are passed to *class:~gromacs.tools.Trjconv*; the only options that cannot be changed are *sep* and the output file name *o*.

all_frames
Unordered list of all frames currently held on disk.

cleanup()
Clean up all temporary frames (which can be HUGE).

delete_frames()
Delete all frames.

extract()
Extract frames from the trajectory to the temporary directory.

class **Transformer**(*s='topol.tpr', f='traj.xtc', n=None, force=None, dirname='.'*)
Class to handle transformations of trajectories.

1.Center, compact, and fit to reference structure in tpr (optionally, only center in the xy plane): `center_fit()`

2.Write compact xtc and tpr with water removed: `strip_water()`

3.Write compact xtc and tpr only with protein: `keep_protein_only()`

Set up Transformer with structure and trajectory.

Supply *n* = tpr, *f* = xtc (and *n* = ndx) relative to dirname.

Keywords

s tpr file (or similar); note that this should not contain position restraints if it is to be used with a reduced system (see `strip_water()`)

f trajectory (xtc, trr, ...)

n index file (it is typically safe to leave this as `None`; in cases where a trajectory needs to be centered on non-standard groups this should contain those groups)

force

Set the default behaviour for handling existing files:

- `True`: overwrite existing trajectories
- `False`: throw a IOError exception
- `None`: skip existing and log a warning [default]

center_fit(***kwargs*)
Write compact xtc that is fitted to the tpr reference structure.

See :func:gromacs.cbook.trj_fitandcenter‘ for details and description of *kwargs*. The most important ones are listed here but in most cases the defaults should work.

Keywords

s Input structure (typically the default tpr file but can be set to some other file with a different conformation for fitting)

n Alternative index file.

o Name of the output trajectory.

xy [Boolean] If `True` then only fit in xy-plane (useful for a membrane normal to z). The default is `False`.

force

- `True`: overwrite existing trajectories
- `False`: throw a IOError exception
- `None`: skip existing and log a warning [default]

Returns dictionary with keys *tpr*, *xtc*, which are the names of the the new files

`fit`(*xy=False, **kwargs*)

Write xtc that is fitted to the tpr reference structure.

See `gromacs.cbook.trj_xyfitted()` for details and description of *kwargs*. The most important ones are listed here but in most cases the defaults should work.

Keywords

s Input structure (typically the default tpr file but can be set to some other file with a different conformation for fitting)

n Alternative index file.

o Name of the output trajectory. A default name is created. If e.g. *dt* = 100 is one of the *kwargs* then the default name includes "_dt100ps".

xy [boolean] If `True` then only do a rot+trans fit in the xy plane (good for membrane simulations); default is `False`.

force `True`: overwrite existing trajectories `False`: throw a IOError exception `None`: skip existing and log a warning [default]

kwargs kwargs are passed to `trj_xyfitted()`

Returns dictionary with keys *tpr*, *xtc*, which are the names of the the new files

`keep_protein_only`(*os=None, o=None, on=None, compact=False, groupname='proteinonly', **kwargs*)

Write xtc and tpr only containing the protein.

Keywords

os Name of the output tpr file; by default use the original but insert "proteinonly" before suffix.

o Name of the output trajectory; by default use the original name but insert "proteinonly" before suffix.

on Name of a new index file.

compact `True`: write a compact and centered trajectory `False`: use trajectory as it is [`False`]

groupname Name of the protein-only group.

keepalso List of literal make_ndx selections of additional groups that should be kept, e.g. ['resname DRUG', 'atom 6789'].

force [Boolean]

- `True`: overwrite existing trajectories
- `False`: throw a IOError exception
- `None`: skip existing and log a warning [default]

kwargs are passed on to `gromacs.cbook.trj_compact()` (unless the values have to be set to certain values such as s, f, n, o keywords). The *input* keyword is always mangled: Only the first entry (the group to centre the trajectory on) is kept, and as a second group (the output group) *groupname* is used.

Returns dictionary with keys *tpr*, *xtc*, *ndx* which are the names of the the new files

> **Warning:** The input tpr file should *not* have *any position restraints*; otherwise Gromacs will throw a hissy-fit and say
> *Software inconsistency error: Position restraint coordinates are missing*
> (This appears to be a bug in Gromacs 4.x.)

`rp`(**args*)
Return canonical path to file under *dirname* with components *args*

If *args* form an absolute path then just return it as the absolute path.

`strip_water`(*os=None, o=None, on=None, compact=False, resn='SOL', groupname='notwater', **kwargs*)
Write xtc and tpr with water (by resname) removed.

Keywords

os Name of the output tpr file; by default use the original but insert "nowater" before suffix.

o Name of the output trajectory; by default use the original name but insert "nowater" before suffix.

on Name of a new index file (without water).

compact `True`: write a compact and centered trajectory `False`: use trajectory as it is [`False`]

resn Residue name of the water molecules; all these residues are excluded.

groupname Name of the group that is generated by subtracting all waters from the system.

force [Boolean]

- `True`: overwrite existing trajectories
- `False`: throw a IOError exception
- `None`: skip existing and log a warning [default]

kwargs are passed on to `gromacs.cbook.trj_compact()` (unless the values have to be set to certain values such as s, f, n, o keywords). The *input* keyword is always mangled: Only the first entry (the group to centre the trajectory on) is kept, and as a second group (the output group) *groupname* is used.

Returns dictionary with keys *tpr*, *xtc*, *ndx* which are the names of the the new files

> **Warning:** The input tpr file should *not* have *any position restraints*; otherwise Gromacs will throw a hissy-fit and say
> *Software inconsistency error: Position restraint coordinates are missing*
> (This appears to be a bug in Gromacs 4.x.)

`get_volume`(*f*)
Return the volume in nm^3 of structure file *f*.

(Uses `gromacs.editconf()`; error handling is not good)

Processing output

There are cases when a script has to to do different things depending on the output from a Gromacs tool.

For instance, a common case is to check the total charge after grompping a tpr file. The `grompp_qtot` function does just that.

`grompp_qtot`(**args, **kwargs*)

Run `gromacs.grompp` and return the total charge of the system.

Arguments The arguments are the ones one would pass to `gromacs.grompp()`.

Returns The total charge as reported

Some things to keep in mind:

- The stdout output of grompp is not shown. This can make debugging pretty hard. Try running the normal `gromacs.grompp()` command and analyze the output if the debugging messages are not sufficient.
- Check that `qtot` is correct; because the function is based on pattern matching of the output it can break when the output format changes.

`get_volume`(*f*)

Return the volume in nm^3 of structure file *f*.

(Uses `gromacs.editconf()`; error handling is not good)

`parse_ndxlist`(*output*)

Parse output from make_ndx to build list of index groups:

```
groups = parse_ndxlist(output)
```

output should be the standard output from `make_ndx`, e.g.:

```
rc,output,junk = gromacs.make_ndx(..., input=('', 'q'), stdout=False, stderr=True)
```

(or simply use

rc,output,junk = cbook.make_ndx_captured(...)

which presets input, stdout and stderr; of course input can be overriden.)

Returns The function returns a list of dicts (`groups`) with fields

name name of the groups

nr number of the group (starts at 0)

natoms number of atoms in the group

Working with topologies and mdp files

`create_portable_topology`(*topol, struct, **kwargs*)

Create a processed topology.

The processed (or portable) topology file does not contain any `#include` statements and hence can be easily copied around. It also makes it possible to re-grompp without having any special itp files available.

Arguments

topol topology file

struct coordinat (structure) file

Keywords

processed name of the new topology file; if not set then it is named like *topol* but with `pp_` prepended

includes path or list of paths of directories in which itp files are searched for

Returns full path to the processed trajectory

`edit_mdp`(*mdp, new_mdp=None, extend_parameters=None, **substitutions*)
Change values in a Gromacs mdp file.

Parameters and values are supplied as substitutions, eg `nsteps=1000`.

By default the template mdp file is **overwritten in place**.

If a parameter does not exist in the template then it cannot be substituted and the parameter/value pair is returned. The user has to check the returned list in order to make sure that everything worked as expected. At the moment it is not possible to automatically append the new values to the mdp file because of ambiguities when having to replace dashes in parameter names with underscores (see the notes below on dashes/underscores).

If a parameter is set to the value `None` then it will be ignored.

Arguments

mdp [filename] filename of input (and output filename of `new_mdp=None`)

new_mdp [filename] filename of alternative output mdp file [None]

extend_parameters [string or list of strings] single parameter or list of parameters for which the new values should be appended to the existing value in the mdp file. This makes mostly sense for a single parameter, namely 'include', which is set as the default. Set to `[]` to disable. ['include']

substitutions parameter=value pairs, where parameter is defined by the Gromacs mdp file; dashes in parameter names have to be replaced by underscores.

Returns Dict of parameters that have *not* been substituted.

Example

```
edit_mdp('md.mdp', new_mdp='long_md.mdp', nsteps=100000, nstxtcout=1000, lincs_iter=2)
```

Note:

- Dashes in Gromacs mdp parameters have to be replaced by an underscore when supplied as python keyword arguments (a limitation of python). For example the MDP syntax is `lincs-iter = 4` but the corresponding keyword would be `lincs_iter = 4`.
- If the keyword is set as a dict key, eg `mdp_params['lincs-iter']=4` then one does not have to substitute.
- Parameters *aa_bb* and *aa-bb* are considered the same (although this should not be a problem in practice because there are no mdp parameters that only differ by a underscore).
- This code is more compact in `Perl` as one can use `s///` operators: `s/^(\s*${key}\s*=\s*).*/$1${val}/`

See Also:

One can also load the mdp file with `gromacs.formats.MDP`, edit the object (a dict), and save it again.

`add_mdp_includes`(*topology=None, kwargs=None*)
Set the mdp *include* key in the *kwargs* dict.

1. Add the directory containing *topology*.
2. Add all directories appearing under the key *includes*
3. Generate a string of the form "-Idir1 -Idir2 ..." that is stored under the key *include* (the corresponding mdp parameter)

By default, the directories `.` and `..` are also added to the *include* string for the mdp; when fed into `gromacs.cbook.edit_mdp()` it will result in a line such as

```
include = -I. -I.. -I../topology_dir ....
```

Note that the user can always override the behaviour by setting the *include* keyword herself; in this case this function does nothing.

If no *kwargs* were supplied then a dict is generated with the single *include* entry.

Arguments

topology [top filename] Topology file; the name of the enclosing directory is added to the include path (if supplied) [`None`]

kwargs [dict] Optional dictionary of mdp keywords; will be modified in place. If it contains the *includes* keyword with either a single string or a list of strings then these paths will be added to the include statement.

Returns *kwargs* with the *include* keyword added if it did not exist previously; if the keyword already existed, nothing happens.

Note: The *kwargs* dict is **modified in place**. This function is a bit of a hack. It might be removed once all setup functions become methods in a nice class.

`grompp_qtot`(**args, **kwargs*)

Run `gromacs.grompp` and return the total charge of the system.

Arguments The arguments are the ones one would pass to `gromacs.grompp()`.

Returns The total charge as reported

Some things to keep in mind:

- The stdout output of grompp is not shown. This can make debugging pretty hard. Try running the normal `gromacs.grompp()` command and analyze the output if the debugging messages are not sufficient.
- Check that `qtot` is correct; because the function is based on pattern matching of the output it can break when the output format changes.

Working with index files

Manipulation of index files (`ndx`) can be cumbersome because the `make_ndx` program is not very sophisticated (yet) compared to full-fledged atom selection expression as available in Charmm, VMD, or MDAnalysis. Some tools help in building and interpreting index files.

See Also:

The `gromacs.formats.NDX` class can solve a number of index problems in a cleaner way than the classes and functions here.

class **`IndexBuilder`**(*struct=None, selections=None, names=None, name_all=None, ndx=None, out_ndx='selection.ndx', offset=0*)

Build an index file with specified groups and the combined group.

This is *not* a full blown selection parser a la Charmm, VMD or MDAnalysis but a very quick hack.

Example

How to use the IndexBuilder:

```
G = gromacs.cbook.IndexBuilder('md_posres.pdb',
              ['S312:OG','T313:OG1','A38:O','A309:O','@a62549 & r NA'],
              offset=-9, out_ndx='selection.ndx')
groupname, ndx = G.combine()
del G
```

The residue numbers are given with their canonical resids from the sequence or pdb. *offset=-9* says that one calculates Gromacs topology resids by subtracting 9 from the canonical resid.

The combined selection is `OR` ed by default and written to *selection.ndx*. One can also add all the groups in the initial *ndx* file (or the **make_ndx** default groups) to the output (see the *defaultgroups* keyword for `IndexBuilder.combine()`).

Generating an index file always requires calling `combine()` even if there is only a single group.

Deleting the class removes all temporary files associated with it (see `IndexBuilder.indexfiles`).

Raises If an empty group is detected (which does not always work) then a `gromacs.BadParameterWarning` is issued.

Bugs If `make_ndx` crashes with an unexpected error then this is fairly hard to diagnose. For instance, in certain cases it segmentation faults when a tpr is provided as a *struct* file and the resulting error messages becomes

```
GromacsError: [Errno -11] Gromacs tool failed
Command invocation: make_ndx -o /tmp/tmp_Na1__NK7cT3.ndx -f md_posres.tpr
```

In this case run the command invocation manually to see what the problem could be.

See Also:

In some cases it might be more straightforward to use `gromacs.formats.NDX`.

Build a index group from the selection arguments.

If selections and a structure file are supplied then the individual selections are constructed with separate calls to `gromacs.make_ndx()`. Use `IndexBuilder.combine()` to combine them into a joint selection.

Arguments

struct [filename] Structure file (tpr, pdb, ...)

selections [list] The list must contain strings or tuples, which must be be one of the following constructs:

"<1-letter aa code><resid>[:<atom name]"

Selects the CA of the residue or the specified atom name.

example: `"S312:OA"` or `"A22"` (equivalent to `"A22:CA"`)

("<1-letter aa code><resid>", "<1-letter aa code><resid>, ["<atom name>"])

Selects a *range* of residues. If only two residue identifiers are provided then all atoms are selected. With an optional third atom identifier, only this atom anme is selected for each residue in the range. [EXPERIMENTAL]

"@<make_ndx selection>"

The `@` letter introduces a verbatim `make_ndx` command. It will apply the given selection without any further processing or checks.

example: `"@a 6234 - 6238"` or `'@"SOL"'` (note the quoting) or `"@r SER & r 312 & t OA"`.

names [list] Strings to name the selections; if not supplied or if individuals are `None` then a default name is created.

offset [int, dict] This number is added to the resids in the first selection scheme; this allows names to be the same as in a crystal structure. If offset is a dict then it is used to directly look up the resids.

ndx [filename or list of filenames] Optional input index file(s).

out_ndx [filename] Output index file.

combine(*name_all=None, out_ndx=None, operation='|', defaultgroups=False*)

Combine individual groups into a single one and write output.

Keywords

name_all [string] Name of the combined group, `None` generates a name. [`None`]

out_ndx [filename] Name of the output file that will contain the individual groups and the combined group. If `None` then default from the class constructor is used. [`None`]

operation [character] Logical operation that is used to generate the combined group from the individual groups: "|" (OR) or "&" (AND) ["|"]

defaultgroups [bool] `True`: append everything to the default groups produced by **make_ndx** (or rather, the groups provided in the ndx file on initialization — if this was `None` then these are truly default groups); `False`: only use the generated groups

Returns `(combinedgroup_name, output_ndx)`, a tuple showing the actual group name and the name of the file; useful when all names are autogenerated.

Warning: The order of the atom numbers in the combined group is *not* guaranteed to be the same as the selections on input because `make_ndx` sorts them ascending. Thus you should be careful when using these index files for calculations of angles and dihedrals. Use `gromacs.formats.NDX` in these cases.

gmx_resid(*resid*)

Returns resid in the Gromacs index by transforming with offset.

parse_ndxlist(*output*)

Parse output from make_ndx to build list of index groups:

```
groups = parse_ndxlist(output)
```

output should be the standard output from `make_ndx`, e.g.:

```
rc,output,junk = gromacs.make_ndx(..., input=('', 'q'), stdout=False, stderr=True)
```

(or simply use

rc,output,junk = cbook.make_ndx_captured(...)

which presets input, stdout and stderr; of course input can be overriden.)

Returns The function returns a list of dicts (`groups`) with fields

name name of the groups

nr number of the group (starts at 0)

natoms number of atoms in the group

`get_ndx_groups`(*ndx*, ***kwargs*)

Return a list of index groups in the index file *ndx*.

Arguments

- *ndx* is a Gromacs index file.
- kwargs are passed to `make_ndx_captured()`.

Returns list of groups as supplied by `parse_ndxlist()`

Alternatively, load the index file with `gromacs.formats.NDX` for full control.

`make_ndx_captured`(***kwargs*)

make_ndx that captures all output

Standard `make_ndx()` command with the input and output pre-set in such a way that it can be conveniently used for `parse_ndxlist()`.

Example:: ndx_groups = parse_ndxlist(make_ndx_captured(n=ndx)[0])

Note that the convenient `get_ndx_groups()` function does exactly that and can probably used in most cases.

Arguments keywords are passed on to `make_ndx()`

Returns (*returncode*, *output*, `None`)

File editing functions

It is often rather useful to be able to change parts of a template file. For specialized cases the two following functions are useful:

`edit_mdp`(*mdp*, *new_mdp=None*, *extend_parameters=None*, ***substitutions*)

Change values in a Gromacs mdp file.

Parameters and values are supplied as substitutions, eg `nsteps=1000`.

By default the template mdp file is **overwritten in place**.

If a parameter does not exist in the template then it cannot be substituted and the parameter/value pair is returned. The user has to check the returned list in order to make sure that everything worked as expected. At the moment it is not possible to automatically append the new values to the mdp file because of ambiguities when having to replace dashes in parameter names with underscores (see the notes below on dashes/underscores).

If a parameter is set to the value `None` then it will be ignored.

Arguments

mdp [filename] filename of input (and output filename of `new_mdp=None`)

new_mdp [filename] filename of alternative output mdp file [None]

extend_parameters [string or list of strings] single parameter or list of parameters for which the new values should be appended to the existing value in the mdp file. This makes mostly sense for a single parameter, namely 'include', which is set as the default. Set to `[]` to disable. ['include']

substitutions parameter=value pairs, where parameter is defined by the Gromacs mdp file; dashes in parameter names have to be replaced by underscores.

Returns Dict of parameters that have *not* been substituted.

Example

```
edit_mdp('md.mdp', new_mdp='long_md.mdp', nsteps=100000, nstxtcout=1000, lincs_iter=2)
```

Note:

- Dashes in Gromacs mdp parameters have to be replaced by an underscore when supplied as python keyword arguments (a limitation of python). For example the MDP syntax is `lincs-iter = 4` but the corresponding keyword would be `lincs_iter = 4`.
- If the keyword is set as a dict key, eg `mdp_params['lincs-iter']=4` then one does not have to substitute.
- Parameters *aa_bb* and *aa-bb* are considered the same (although this should not be a problem in practice because there are no mdp parameters that only differ by a underscore).
- This code is more compact in `Perl` as one can use `s///` operators: `s/^(\s*${key}\s*=\s*).*/$1${val}/`

See Also:

One can also load the mdp file with `gromacs.formats.MDP`, edit the object (a dict), and save it again.

`edit_txt` (*filename, substitutions, newname=None*)

Primitive text file stream editor.

This function can be used to edit free-form text files such as the topology file. By default it does an **in-place edit** of *filename*. If *newname* is supplied then the edited file is written to *newname*.

Arguments

filename input text file

substitutions substitution commands (see below for format)

newname output filename; if `None` then *filename* is changed in place [`None`]

substitutions is a list of triplets; the first two elements are regular expression strings, the last is the substitution value. It mimics `sed` search and replace. The rules for *substitutions*:

```
substitutions        ::=  "[" search_replace_tuple, ...  "]"
search_replace_tuple ::=  "(" line_match_RE "," search_RE "," replacement ")"
line_match_RE        ::=  regular expression that selects the line (uses match)
search_RE            ::=  regular expression that is searched in the line
replacement          ::=  replacement string for search_RE
```

Running `edit_txt()` does pretty much what a simple

```
sed /line_match_RE/s/search_RE/replacement/
```

with repeated substitution commands does.

Special replacement values: - `None`: the rule is ignored - `False`: the line is deleted (even if other rules match)

Note:

- No sanity checks are performed and the substitutions must be supplied exactly as shown.
- All substitutions are applied to a line; thus the order of the substitution commands may matter when one substitution generates a match for a subsequent rule.

- If replacement is set to `None` then the whole expression is ignored and whatever is in the template is used. To unset values you must provided an empty string or similar.
- Delete a matching line if replacement=``False``.

`gromacs.setup` – Setting up a Gromacs MD run

Individual steps such as solvating a structure or energy minimization are set up in individual directories. For energy minimization one should supply appropriate mdp run input files; otherwise example templates are used.

Warning: You **must** check all simulation parameters for yourself. Do not rely on any defaults provided here. The scripts provided here are provided under the assumption that you know what you are doing and you just want to automate the boring parts of the process.

User functions

The individual steps of setting up a simple MD simulation are broken down in a sequence of functions that depend on the previous step(s):

`topology()` generate initial topology file (limited functionality, might require manual setup)

`solvate()` solvate globular protein and add ions to neutralize

`energy_minimize()` set up energy minimization and run it (using `mdrun_d`)

`em_schedule()` set up and run multiple energy minimizations one after another (as an alternative to the simple single energy minimization provided by `energy_minimize()`)

`MD_restrained()` set up restrained MD

`MD()` set up equilibrium MD

Each function uses its own working directory (set with the `dirname` keyword argument, but it should be safe and convenient to use the defaults). Other arguments assume the default locations so typically not much should have to be set manually.

One can supply non-standard itp files in the topology directory. In some cases one does not use the `topology()` function at all but sets up the topology manually. In this case it is safest to call the topology directory `top` and make sure that it contains all relevant top, itp, and pdb files.

Example

Run a single protein in a dodecahedral box of SPC water molecules and use the GROMOS96 G43a1 force field. We start with the structure in `protein.pdb`:

```
from gromacs.setup import *
f1 = topology(protein='MyProtein', struct='protein.pdb', ff='G43a1', water='spc', force=True, ignh=Tr
```

Each function returns "interesting" new files in a dictionary in such a away that it can often be used as input for the next function in the chain (although in most cases one can get away with the defaults of the keyword arguments):

```
f2 = solvate(**f1)
f3 = energy_minimize(**f2)
```

Now prepare input for a MD run with restraints on the protein:

```
MD_restrained(**f3)
```

Use the files in the directory to run the simulation locally or on a cluster. You can provide your own template for a queuing system submission script; see the source code for details.

Once the restraint run has completed, use the last frame as input for the equilibrium MD:

```
MD(struct='MD_POSRES/md.gro', runtime=1e5)
```

Run the resulting tpr file on a cluster.

User functions

The following functions are provided for the user:

topology (*struct=None, protein='protein', top='system.top', dirname='top', **pdb2gmx_args*)
Build Gromacs topology files from pdb.

Keywords

struct input structure (**required**)

protein name of the output files

top name of the topology file

dirname directory in which the new topology will be stored

pdb2gmxargs arguments for `pdb2gmx` such as `ff`, `water`, ...

Note: At the moment this function simply runs `pdb2gmx` and uses the resulting topology file directly. If you want to create more complicated topologies and maybe also use additional itp files or make a protein itp file then you will have to do this manually.

solvate (*struct='top/protein.pdb', top='top/system.top', distance=0.90000000000000002, boxtype='dodecahedron', concentration=0, cation='NA+', anion='CL-', water='spc', with_membrane=False, ndx='main.ndx', mainselection='"Protein"', dirname='solvate', **kwargs*)
Put protein into box, add water, add counter-ions.

Currently this really only supports solutes in water. If you need to embedd a protein in a membrane then you will require more sophisticated approaches.

However, you *can* supply a protein already inserted in a bilayer. In this case you will probably want to set *distance* = `None` and also enable *with_membrane* = `True` (using extra big vdw radii for typical lipids).

Arguments

struct [filename] pdb or gro input structure

top [filename] Gromacs topology

distance [float] When solvating with water, make the box big enough so that at least *distance* nm water are between the solute *struct* and the box boundary. Set this to `None` in order to use a box size in the input file (gro or pdb).

boxtype [string] Any of the box types supported by `Genbox`. If set to `None` it will also ignore *distance* and use the box inside the *struct* file.

concentration [float] Concentration of the free ions in mol/l. Note that counter ions are added in excess of this concentration.

cation* and *anion [string] Molecule names of the ions. This depends on the chosen force field.

water [string] Name of the water model; one of "spc", "spce", "tip3p", "tip4p". This should be appropriate for the chosen force field. If no water is requird, simply supply the path to a box with solvent molecules (used by `gromacs.genbox()`'s *cs* argument).

with_membrane [bool] `True`: use special `vdwradii.dat` with 0.1 nm-increased radii on lipids. Default is `False`.

ndx [filename] The name of the custom index file that is produced here.

mainselection [string] A string that is fed to `Make_ndx` and which should select the solute.

dirname [directory name] Name of the directory in which all files for the solvation stage are stored.

includes : list of additional directories to add to the mdp include path

Note: non-water solvents only work if the molecules are named SOL.

`energy_minimize`(*dirname='em', mdp='/sansom/gfio/oliver/Library/python/GromacsWrapper/gromacs/templates/em.mdp', struct='solvate/ionized.gro', top='top/system.top', output='em.pdb', deffnm='em', mdrunner=None, **kwargs*)

Energy minimize the system.

This sets up the system (creates run input files) and also runs `mdrun_d`. Thus it can take a while.

Additional itp files should be in the same directory as the top file.

Many of the keyword arguments below already have sensible values.

Keywords

dirname set up under directory dirname [em]

struct input structure (gro, pdb, ...) [solvate/ionized.gro]

output output structure (will be put under dirname) [em.pdb]

deffnm default name for mdrun-related files [em]

top topology file [top/system.top]

mdp mdp file (or use the template) [templates/em.mdp]

includes additional directories to search for itp files

mdrunner `gromacs.run.MDrunner` class; by defauly we just try `gromacs.mdrun_d()` and `gromacs.mdrun()` but a MDrunner class gives the user the ability to run mpi jobs etc. [None]

kwargs remaining key/value pairs that should be changed in the template mdp file, eg `nstxtcout=250, nstfout=250`.

Note: If `mdrun_d()` is not found, the function falls back to `mdrun()` instead.

`em_schedule`(***kwargs*)

Run multiple energy minimizations one after each other.

Keywords

integrators list of integrators (from 'l-bfgs', 'cg', 'steep') [['bfgs', 'steep']]

nsteps list of maximum number of steps; one for each integrator in in the *integrators* list [[100,1000]]

kwargs mostly passed to `gromacs.setup.energy_minimize()`

Returns dictionary with paths to final structure ('struct') and other files

Example

Conduct three minimizations:

1. low memory Broyden-Goldfarb-Fletcher-Shannon (BFGS) for 30 steps
2. steepest descent for 200 steps
3. finish with BFGS for another 30 steps

We also do a multi-processor minimization when possible (i.e. for steep (and conjugate gradient) by using a `gromacs.run.MDrunner` class for a **mdrun** executable compiled for OpenMP in 64 bit (see `gromacs.run` for details):

```
import gromacs.run
gromacs.setup.em_schedule(struct='solvate/ionized.gro',
          mdrunner=gromacs.run.MDrunnerOpenMP64,
          integrators=['l-bfgs', 'steep', 'l-bfgs'],
          nsteps=[50,200, 50])
```

Note: You might have to prepare the mdp file carefully because at the moment one can only modify the *nsteps* parameter on a per-minimizer basis.

`MD_restrained`(*dirname='MD_POSRES'*, ***kwargs*)

Set up MD with position restraints.

Additional itp files should be in the same directory as the top file.

Many of the keyword arguments below already have sensible values. Note that setting *mainselection* = `None` will disable many of the automated choices and is often recommended when using your own mdp file.

Keywords

dirname set up under directory dirname [MD_POSRES]

struct input structure (gro, pdb, ...) [em/em.pdb]

top topology file [top/system.top]

mdp mdp file (or use the template) [templates/md.mdp]

ndx index file (supply when using a custom mdp)

includes additional directories to search for itp files

mainselection **make_ndx** selection to select main group ["Protein"] (If `None` then no canonical index file is generated and it is the user's responsibility to set *tc_grps*, *tau_t*, and *ref_t* as keyword arguments, or provide the mdp template with all parameter pre-set in *mdp* and probably also your own *ndx* index file.)

deffnm default filename for Gromacs run [md]

runtime total length of the simulation in ps [1000]

dt integration time step in ps [0.002]

qscript script to submit to the queuing system; by default uses the template `gromacs.config.qscript_template`, which can be manually set to another template from `gromacs.config.templates`; can also be a list of template names.

qname name to be used for the job in the queuing system [PR_GMX]

mdrun_opts option flags for the **mdrun** command in the queuing system scripts such as "-stepout 100". [""]

kwargs remaining key/value pairs that should be changed in the template mdp file, eg `nstxtcout=250, nstfout=250` or command line options for `grompp` such as ``maxwarn=1`.

In particular one can also set **define** and activate whichever position restraints have been coded into the itp and top file. For instance one could have

define = "-DPOSRES_MainChain -DPOSRES_LIGAND"

if these preprocessor constructs exist. Note that there **must not be any space between "-D" and the value.**

By default *define* is set to "-DPOSRES".

Returns a dict that can be fed into `gromacs.setup.MD()` (but check, just in case, especially if you want to change the `define` parameter in the mdp file)

Note: The output frequency is drastically reduced for position restraint runs by default. Set the corresponding `nst*` variables if you require more output.

MD(*dirname='MD'*, ***kwargs*)
Set up equilibrium MD.

Additional itp files should be in the same directory as the top file.

Many of the keyword arguments below already have sensible values. Note that setting *mainselection* = `None` will disable many of the automated choices and is often recommended when using your own mdp file.

Keywords

dirname set up under directory dirname [MD]

struct input structure (gro, pdb, ...) [MD_POSRES/md_posres.pdb]

top topology file [top/system.top]

mdp mdp file (or use the template) [templates/md.mdp]

ndx index file (supply when using a custom mdp)

includes additional directories to search for itp files

mainselection `make_ndx` selection to select main group ["Protein"] (If `None` then no canonical index file is generated and it is the user's responsibility to set *tc_grps*, *tau_t*, and *ref_t* as keyword arguments, or provide the mdp template with all parameter pre-set in *mdp* and probably also your own *ndx* index file.)

deffnm default filename for Gromacs run [md]

runtime total length of the simulation in ps [1000]

dt integration time step in ps [0.002]

qscript script to submit to the queuing system; by default uses the template `gromacs.config.qscript_template`, which can be manually set to another template from `gromacs.config.templates`; can also be a list of template names.

qname name to be used for the job in the queuing system [MD_GMX]

mdrun_opts option flags for the **mdrun** command in the queuing system scripts such as "-stepout 100 -dgdl". [""]

kwargs remaining key/value pairs that should be changed in the template mdp file, e.g. `nstxtcout=250, nstfout=250` or command line options for :program`grompp` such as `maxwarn=1`.

Returns a dict that can be fed into `gromacs.setup.MD()` (but check, just in case, especially if you want to change the *define* parameter in the mdp file)

Helper functions

The following functions are used under the hood and are mainly useful when writing extensions to the module.

`make_main_index`(*struct, selection='"Protein"', ndx='main.ndx', oldndx=None*)

Make index file with the special groups.

This routine adds the group __main__ and the group __environment__ to the end of the index file. __main__ contains what the user defines as the *central* and *most important* parts of the system. __environment__ is everything else.

The template mdp file, for instance, uses these two groups for T-coupling.

These groups are mainly useful if the default groups "Protein" and "Non-Protein" are not appropriate. By using symbolic names such as __main__ one can keep scripts more general.

Returns *groups* is a list of dictionaries that describe the index groups. See `gromacs.cbook.parse_ndxlist()` for details.

Arguments

struct [filename] structure (tpr, pdb, gro)

selection [string] is a `make_ndx` command such as `"Protein"` or `r DRG` which determines what is considered the main group for centering etc. It is passed directly to `make_ndx`.

ndx [string] name of the final index file

oldndx [string] name of index file that should be used as a basis; if None then the `make_ndx` default groups are used.

This routine is very dumb at the moment; maybe some heuristics will be added later as could be other symbolic groups such as __membrane__.

`check_mdpargs`(*d*)

Check if any arguments remain in dict *d*.

`get_lipid_vdwradii`(*outdir='.', libdir=None*)

Find vdwradii.dat and add special entries for lipids.

See `gromacs.setup.vdw_lipid_resnames` for lipid resnames. Add more if necessary.

`_setup_MD`(*dirname, deffnm='md', mdp='/sansom/gfio/oliver/Library/python/GromacsWrapper/gromacs/templates/md_OPLSAA.m struct=None, top='top/system.top', ndx=None, mainselection='"Protein"', qscript='/sansom/gfio/oliver/Library/python/GromacsWrapper/gromacs/templates/local.sh', qname=None, mdrun_opts='', budget=None, walltime=0.33333333333333331, dt=0.002, runtime=1000.0, **mdp_kwargs*)

Generic function to set up a `mdrun` MD simulation.

See the user functions for usage.

Defined constants:

`CONC_WATER`

Concentration of water at standard conditions in mol/L. Density at 25 degrees C and 1 atmosphere pressure: rho = 997.0480 g/L. Molecular weight: M = 18.015 g/mol. c = n/V = m/(V*M) = rho/M = 55.345 mol/L.

vdw_lipid_resnames
Hard-coded lipid residue names for a `vdwradii.dat` file. Use together with `vdw_lipid_atom_radii` in `get_lipid_vdwradii()`.

vdw_lipid_atom_radii
Increased atom radii for lipid atoms; these are simply the standard values from `GMXLIB/vdwradii.dat` increased by 0.1 nm (C) or 0.05 nm (N, O, H).

`gromacs.qsub` – utilities for batch submission systems

The module helps writing submission scripts for various batch submission queuing systems. The known ones are listed stored as `QueuingSystem` instances in `queuing_systems`; append new ones to this list.

The working paradigm is that template scripts are provided (see `gromacs.config.templates`) and only a few place holders are substituted (using `gromacs.cbook.edit_txt()`).

User-supplied template scripts can be stored in `gromacs.config.qscriptdir` (by default `~/.gromacswrapper/qscripts`) and they will be picked up before the package-supplied ones.

The `Manager` handles setup and control of jobs in a queuing system on a remote system via **ssh**.

At the moment, some of the functions in `gromacs.setup` use this module but it is fairly independent and could conceivably be used for a wider range of projects.

Queuing system templates

The queuing system scripts are highly specific and you will need to add your own. Templates should be shell scripts. Some parts of the templates are modified by the `generate_submit_scripts()` function. The "place holders" that can be replaced are shown in the table below. Typically, the place holders are either shell variable assignments or batch submission system commands. The table shows SGE commands but PBS and LoadLeveler have similar constructs; e.g. PBS commands start with `#PBS` and LoadLeveller uses `#@` with its own command keywords).

Table 1.1: Substitutions in queuing system templates.

place holder	default	replacement	description	regex
#$ -N	GMX_MD	*sgename*	job name	/^#.*(-N|job_name)/
#$ -l walltime=	00:20:00	*walltime*	max run time	/^#.*(-l walltime|wall_clock_limit)/
#$ -A	BUDGET	*budget*	account	/^#.*(-A|account_no)/
DEFFNM=	md	*deffnm*	default gmx name	/^DEFFNM=/
WALL_HOURS=	0.33	*walltime* h	mdrun's -maxh	/^WALL_HOURS=/
MDRUN_OPTS=	""	*mdrun_opts*	more options	/^MDRUN_OPTS=/

Lines with place holders should not have any white space at the beginning. The regular expression pattern ("regex") is used to find the lines for the replacement and the literal default values ("default") are replaced. Not all place holders have to occur in a template; for instance, if a queue has no run time limitation then one would probably not include *walltime* and *WALL_HOURS* place holders.

The line `# JOB_ARRAY_PLACEHOLDER` can be replaced by `generate_submit_array()` to produce a "job array" (also known as a "task array") script that runs a large number of related simulations under the control of a single queuing system job. The individual array tasks are run from different sub directories. Only queuing system scripts that are using the **bash** shell are supported for job arrays at the moment.

A queuing system script *must* have the appropriate suffix to be properly recognized, as shown in the table below.

Table 1.2: Suffices for queuing system templates. Pure shell-scripts are only used to run locally.

Queuing system	suffix	notes
Sun Gridengine	.sge	Sun's Sun Gridengine
Portable Batch queuing system	.pbs	OpenPBS and PBS Pro
LoadLeveler	.ll	IBM's LoadLeveler
bash script	.bash, .sh	Advanced bash scripting
csh script	.csh	avoid csh

Example queuing system script template for PBS The following script is a usable PBS script for a super computer. It contains almost all of the replacement tokens listed in the table (indicated by ++++++; these values should be kept in the template as they are or they will not be subject to replacement).

```
#!/bin/bash
# File name: ~/.gromacswrapper/qscripts/supercomputer.somewhere.fr_64core.pbs
#PBS -N GMX_MD
#       ++++++
#PBS -j oe
#PBS -l select=8:ncpus=8:mpiprocs=8
#PBS -l walltime=00:20:00
#                ++++++++

# host: supercomputer.somewhere.fr
# queuing system: PBS

# set this to the same value as walltime; mdrun will stop cleanly
# at 0.99 * WALL_HOURS
WALL_HOURS=0.33
#          ++++

# deffnm line is possibly modified by gromacs.setup
# (leave it as it is in the template)
DEFFNM=md
#      ++

TPR=${DEFFNM}.tpr
OUTPUT=${DEFFNM}.out
PDB=${DEFFNM}.pdb

MDRUN_OPTS=""
#          ++

# If you always want to add additional MDRUN options in this script then
# you can either do this directly in the mdrun commandline below or by
# constructs such as the following:
## MDRUN_OPTS="-npme 24 $MDRUN_OPTS"

# JOB_ARRAY_PLACEHOLDER
#+++++++++++++++++++++++   leave the full commented line intact!

# avoids some failures
export MPI_GROUP_MAX=1024
# use hard coded path for time being
GMXBIN="/opt/software/SGI/gromacs/4.0.3/bin"
MPIRUN=/usr/pbs/bin/mpiexec
```

```
APPLICATION=$GMXBIN/mdrun_mpi

$MPIRUN $APPLICATION -stepout 1000 -deffnm ${DEFFNM} -s ${TPR} -c ${PDB} -cpi
rc=$?

# dependent jobs will only start if rc == 0
exit $rc
```

Save the above script in `~/.gromacswrapper/qscripts` under the name `supercomputer.somewhere.fr_64core.pbs`. This will make the script immediately usable. For example, in order to set up a production MD run with `gromacs.setup.MD()` for this super computer one would use

```
gromacs.setup.MD(..., qscripts=['supercomputer.somewhere.fr_64core.pbs', 'local.sh'])
```

This will generate submission scripts based on `supercomputer.somewhere.fr_64core.pbs` and also the default `local.sh` that is provided with *GromacsWrapper*.

In order to modify `MDRUN_OPTS` one would use the additonal *mdrun_opts* argument, for instance:

```
gromacs.setup.MD(..., qscripts=['supercomputer.somewhere.fr_64core.pbs', 'local.sh'],
                 mdrun_opts="-v -npme 20 -dlb yes -nosum")
```

Currently there is no good way to specify the number of processors when creating run scripts. You will need to provided scripts with different numbers of cores hard coded or set them when submitting the scripts with command line options to **qsub**.

Classes and functions

class `QueuingSystem`(*name, suffix, qsub_prefix, array_variable=None, array_option=None*)

Class that represents minimum information about a batch submission system.

Define a queuing system's functionality

Arguments

name name of the queuing system, e.g. 'Sun Gridengine'

suffix suffix of input files, e.g. 'sge'

qsub_prefix prefix string that starts a qsub flag in a script, e.g. '#$'

Keywords

array_variable environment variable exported for array jobs, e.g. 'SGE_TASK_ID'

array_option qsub option format string to launch an array (e.g. '-t %d-%d')

`array`(*directories*)

Return multiline string for simple array jobs over *directories*.

> **Warning:** The string is in `bash` and hence the template must also be `bash` (and *not* `csh` or `sh`).

`array_flag`(*directories*)

Return string to embed the array launching option in the script.

`flag`(**args*)

Return string for qsub flag *args* prefixed with appropriate inscript prefix.

has_arrays()
True if known how to do job arrays.

isMine(*scriptname*)
Primitive queuing system detection; only looks at suffix at the moment.

generate_submit_scripts(*templates, prefix=None, deffnm='md', jobname='MD', budget=None, mdrun_opts=None, walltime=1.0, jobarray_string=None, **kwargs*)
Write scripts for queuing systems.

This sets up queuing system run scripts with a simple search and replace in templates. See `gromacs.cbook.edit_txt()` for details. Shell scripts are made executable.

Arguments

templates Template file or list of template files. The "files" can also be names or symbolic names for templates in the templates directory. See `gromacs.config` for details and rules for writing templates.

prefix Prefix for the final run script filename; by default the filename will be the same as the template. [None]

dirname Directory in which to place the submit scripts. [.]

deffnm Default filename prefix for **mdrun** `-deffnm` [md]

jobname Name of the job in the queuing system. [MD]

budget Which budget to book the runtime on [None]

mdrun_opts String of additional options for **mdrun**.

walltime Maximum runtime of the job in hours. [1]

jobarray_string Multi-line string that is spliced in for job array functionality (see `gromacs.qsub.generate_submit_array()`; do not use manually)

kwargs all other kwargs are ignored

Returns list of generated run scripts

generate_submit_array(*templates, directories, **kwargs*)
Generate a array job.

For each `work_dir` in *directories*, the array job will

1. cd into `work_dir`
2. run the job as detailed in the template

It will use all the queuing system directives found in the template. If more complicated set ups are required, then this function cannot be used.

Arguments

templates Basic template for a single job; the job array logic is spliced into the position of the line

```
# JOB_ARRAY_PLACEHOLDER
```

The appropriate commands for common queuing systems (Sun Gridengine, PBS) are hard coded here. The queuing system is detected from the suffix of the template.

directories List of directories under *dirname*. One task is set up for each directory.

dirname The array script will be placed in this directory. The *directories* **must** be located under *dirname*.

kwargs See `gromacs.setup.generate_submit_script()` for details.

`detect_queuing_system`(*scriptfile*)
Return the queuing system for which *scriptfile* was written.

`queuing_systems`
Pre-defined queuing systems (SGE, PBS). Add your own here.

Queuing system Manager

The `Manager` class must be customized for each system such as a cluster or a super computer. It then allows submission and control of jobs remotely (using ssh).

class `Manager`(*dirname='.'*, ***kwargs*)
Base class to launch simulations remotely on computers with queuing systems.

Basically, ssh into machine and run job.

Derive a class from `Manager` and override the attributes

- `Manager._hostname` (hostname of the machine)
- `Manager._scratchdir` (all files and directories will be created under this scratch directory; it must be a path on the remote host)
- `Manager._qscript` (the default queuing system script template)
- `Manager._walltime` (if there is a limit to the run time of a job; in hours)

and implement a specialized `Manager.qsub()` method if needed.

ssh must be set up (via ~/.ssh/config) to allow access via a commandline such as

```
ssh <hostname> <command> ...
```

Typically you want something such as

```
host <hostname>
     hostname <hostname>.fqdn.org
     user     <remote_user>
```

in `~/.ssh/config` and also set up public-key authentication in order to avoid typing your password all the time.

Set up the manager.

Arguments

statedir directory component under the remote scratch dir (should be different for different jobs) [basename(CWD)]

prefix identifier for job names [MD]

`_hostname`
hostname of the super computer (**required**)

`_scratchdir`
scratch dir on hostname (**required**)

`_qscript`
name of the template submission script appropriate for the queuing system on `Manager._hostname`; can be a path to a local file or a template stored in `gromacs.config.qscriptdir` or a key for `gromacs.config.templates` (**required**)

`_walltime`
maximum run time of script in hours; the queuing system script `Manager._qscript` is supposed to stop **mdrun** after 99% of this time via the `-maxh` option. A value of `None` or `inf` indicates no limit.

`job_done``()`
alias for `get_status()`

`qstat``()`
alias for `get_status()`

`cat`(*dirname, prefix='md', cleanup=True*)
Concatenate parts of a run in *dirname*.

Always uses `gromacs.cbook.cat()` with *resolve_multi* = 'guess'.

Note: The default is to immediately delete the original files (*cleanup* = `True`).

Keywords

dirname directory to work in

prefix prefix (deffnm) of the files [md]

cleanup [boolean] if `True`, remove all used files [`True`]

`get`(*dirname, checkfile=None, targetdir='.'*)
`scp -r` *dirname* from host into *targetdir*

Arguments

- *dirname*: dir to download
- *checkfile*: raise OSError/ENOENT if *targetdir/dirname/checkfile* was not found
- *targetdir*: put *dirname* into this directory

Returns return code from scp

`get_dir`(**args*)
Directory on the remote machine.

`get_status`(*dirname, logfilename='md*.log', silent=False*)
Check status of remote job by looking into the logfile.

Report on the status of the job and extracts the performance in ns/d if available (which is saved in `Manager.performance`).

Arguments

- *dirname*
- *logfilename* can be a shell glob pattern [md*.log]
- *silent* = True/False; True suppresses log.info messages

Returns `True` is job is done, `False` if still running `None` if no log file found to look at

Note: Also returns `False` if the connection failed.

Warning: This is an important but somewhat **fragile** method. It needs to be improved to be more robust.

local_get (*dirname, checkfile, cattrajectories=True, cleanup=False*)
Find *checkfile* locally if possible.

If *checkfile* is not found in *dirname* then it is transferred from the remote host.

If needed, the trajectories are concatenated using `Manager.cat()`.

Returns local path of *checkfile*

log_RE
Regular expression used by `Manager.get_status()` to parse the logfile from **mdrun**.

ndependent (*runtime, performance=None, walltime=None*)
Calculate how many dependent (chained) jobs are required.

Uses *performance* in ns/d (gathered from `get_status()`) and job max *walltime* (in hours) from the class unless provided as keywords.

n = ceil(runtime/(performance*0.99*walltime)

Keywords

runtime length of run in ns

performance ns/d with the given setup

walltime maximum run length of the script (using 99% of it), in h

Returns *n* or 1 if walltime is unlimited

put (*dirname*)
scp dirname to host.

Arguments dirname to be transferred

Returns return code from scp

putfile (*filename, dirname*)
scp *filename* to host in *dirname*.

Arguments filename and dirname to be transferred to

Returns return code from scp

qsub (*dirname, **kwargs*)
Submit job remotely on host.

This is the most primitive implementation: it just runs the commands

```
cd remotedir && qsub qscript
```

on `Manager._hostname`. *remotedir* is *dirname* under `Manager._scratchdir` and *qscript* defaults to the queuing system script hat was produced from the template `Manager._qscript`.

remotepath (**args*)
Directory on the remote machine.

remoteuri (**args*)
URI of the directory on the remote machine.

setup_MD (*jobnumber, struct='MD_POSRES/md.pdb', **kwargs*)
Set up production and transfer to host.

Arguments

- *jobnumber*: 1,2 ...
- *struct* is the starting structure (default from POSRES run but that is just a guess);
- kwargs are passed to `gromacs.setup.MD()`

`setup_posres`(***kwargs*)
Set up position restraints run and transfer to host.

kwargs are passed to `gromacs.setup.MD_restrained()`

`waitfor`(*dirname, **kwargs*)
Wait until the job associated with *dirname* is done.

Super-primitive, uses a simple while ... sleep for *seconds* delay

Arguments

dirname look for log files under the remote dir corresponding to *dirname*

seconds delay in *seconds* during re-polling

`gromacs.run` – Running simulations

Helper functions and classes around `gromacs.tools.Mdrun`.

class **`MDrunner`**(*dirname='.', **kwargs*)
A class to manage running **mdrun** in various ways.

In order to do complicated multiprocessor runs with mpiexec or similar you need to derive from this class and override

- `MDrunner.mdrun` with the path to the `mdrun` executable
- `MDrunner.mpiexec` with the path to the MPI launcher
- `MDrunner.mpicommand()` with a function that returns the mpi command as a list

In addition there are two methods named `prehook()` and `posthook()` that are called right before and after the process is started. If they are overriden appropriately then they can be used to set up a mpi environment.

Set up a simple run with `mdrun`.

Keywords

dirname Change to this directory before launching the job. Input files must be supplied relative to this directory.

keywords All other keword arguments are used to construct the `mdrun` commandline. Note that only keyword arguments are allowed.

`check_success`()
Check if **mdrun** finished successfully.

(See `check_mdrun_success()` for details)

`commandline`(***mpiargs*)
Returns simple command line to invoke mdrun.

If `mpiexec` is set then `mpicommand()` provides the mpi launcher command that prefixes the actual `mdrun` invocation:

`mpiexec` [*mpiargs*] `mdrun` [*mdrun-args*]

The *mdrun-args* are set on initializing the class. Override `mpicommand()` to fit your system if the simple default OpenMP launcher is not appropriate.

mdrun
path to the **mdrun** executable (or the name if it can be found on **PATH**)

mpicommand (**args, **kwargs*)
Return a list of the mpi command portion of the commandline.

Only allows primitive mpi at the moment: *mpiexec* -n *ncores mdrun mdrun-args*

(This is a primitive example for OpenMP. Override it for more complicated cases.)

mpiexec
path to the MPI launcher (e.g. **mpiexec**)

posthook (***kwargs*)
Called directly after the process terminated (also if it failed).

prehook (***kwargs*)
Called directly before launching the process.

run (*pre=None, post=None, **mpiargs*)
Execute the mdrun command (possibly as a MPI command) and run the simulation.

Keywords

pre a dictionary containing keyword arguments for the `prehook()`

post a dictionary containing keyword arguments for the `posthook()`

mpiargs keyword arguments that are processed by `mpicommand()`

run_check (***kwargs*)
Run **mdrun** and check if run completed when it finishes.

This works by looking at the mdrun log file for 'Finished mdrun on node'. It is useful to implement robust simulation techniques.

Arguments *kwargs* are keyword arguments that are passed on to `run()` (typically used for mpi things)

Returns

- `True` if run completed successfully
- `False` otherwise

class **MDrunnerOpenMP** (*dirname='.', **kwargs*)
Manage running **mdrun** as an OpenMP multiprocessor job.

Set up a simple run with `mdrun`.

Keywords

dirname Change to this directory before launching the job. Input files must be supplied relative to this directory.

keywords All other keword arguments are used to construct the `mdrun` commandline. Note that only keyword arguments are allowed.

class **MDrunnerOpenMP64** (*dirname='.', **kwargs*)
Manage running **mdrun** as an OpenMP multiprocessor job (64-bit executable).

Set up a simple run with `mdrun`.

Keywords

dirname Change to this directory before launching the job. Input files must be supplied relative to this directory.

keywords All other keword arguments are used to construct the `mdrun` commandline. Note that only keyword arguments are allowed.

class `MDrunnerMpich2Smpd`(*dirname='.'*, ***kwargs*)

Manage running **mdrun** as mpich2 multiprocessor job with the SMPD mechanism.

Set up a simple run with `mdrun`.

Keywords

dirname Change to this directory before launching the job. Input files must be supplied relative to this directory.

keywords All other keword arguments are used to construct the `mdrun` commandline. Note that only keyword arguments are allowed.

`check_mdrun_success`(*logfile*)

Check if `mdrun` finished successfully.

Analyses the output from `mdrun` in *logfile*. Right now we are simply looking for the line "Finished mdrun on node" in the last 1kb of the file. (The file must be seeakable.)

Arguments

logfile [filename] Logfile produced by `mdrun`.

Returns boolean (`True` if all ok, `False` otherwise)

1.4 Analysis

The analysis package uses the `gromacs` package and various other third party ones such as numpy and pylab. It provides a frame work to analyze Gromacs MD simulations.

1.4.1 Analysis core modules

The core modules contain the important classes `Simulation` and `Plugin`. An analysis class is derived from `gromacs.analysis.Simulation` and additional plugins from `gromacs.analysis.plugins` are added to the instance; these plugin classes must be derived from `Plugin`.

`gromacs.analysis` – Analysis Package Overview

The `gromacs.analysis` package is a framework for analyzing Gromacs MD trajectories. The basic object is the `Simulation` class. For a particular project one has to derive a class from `Simulation` and add analysis plugin classes (from `gromacs.analysis.plugins`) for specific analysis tasks. This is slightly cumbersome but flexible.

New analysis plugins should follow the API sketched out in `gromacs.analysis.core`; see an example for use there.

Right now the number of plugins is limited and simply demonstrates how to use the framework in principle. If you would like to contribute your own plugins feel free to send then to the package author. If they have been written according to the API they will be added to the distribution and of course you will be acknowledged in the list of plugin authors in `gromacs.analysis.plugins`.

Simulation class

The `Simulation` class is central for doing analysis. The user can derive a custom analysis class that pre-defines values for plugins as seen in the Example.

class `Simulation`(***kwargs*)

Class that represents one simulation.

Analysis capabilities are added via plugins.

1. Set the *active plugin* with the `Simulation.set_plugin()` method.
2. Analyze the trajectory with the active plugin by calling the `Simulation.run()` method.
3. Analyze the output from `run()` with `Simulation.analyze()`; results are stored in the plugin's `results` dictionary.
4. Plot results with `Simulation.plot()`.

Set up a Simulation object.

Keywords

sim Any object that contains the attributes *tpr*, *xtc*, and optionally *ndx* (e.g. `gromacs.cbook.Transformer`). The individual keywrods such as *xtc* override the values in *sim*.

tpr Gromacs tpr file (**required**)

xtc Gromacs trajectory, can also be a trr (**required**)

edr Gromacs energy file (only required for some plugins)

ndx Gromacs index file

analysisdir directory under which derived data are stored; defaults to the directory containing the tpr [None]

plugins [list] plugin instances or tuples (*plugin class*, *kwarg dict*) or tuples (*plugin_class_name*, *kwarg dict*) to be used; more can be added later with `Simulation.add_plugin()`.

`add_plugin`(*plugin, **kwargs*)

Add a plugin to the registry.

- If *plugin* is a `Plugin` instance then the instance is directly registered and any keyword arguments are ignored.
- If *plugin* is a `Plugin` class object or a string that can be found in `gromacs.analysis.plugins` then first an instance is created with the given keyword arguments and then registered.

Arguments

plugin [class or string, or instance] If the parameter is a class then it should have been derived from `Plugin`. If it is a string then it is taken as a plugin name in `gromacs.analysis.plugins` and the corresponding class is added. In both cases any parameters for initizlization should be provided.

If *plugin* is already a `Plugin` instance then the kwargs will be ignored.

kwargs The kwargs are specific for the plugin and should be described in its documentation.

`set_plugin`(*plugin_name*)

Set the plugin that should be used by default.

If no *plugin_name* is supplied to `run()`, `analyze()` etc. then this will be used.

run (*plugin_name=None, **kwargs*)
Generate data files as prerequisite to analysis.

analyze (*plugin_name=None, **kwargs*)
Run analysis for the plugin.

plot (*plugin_name=None, figure=False, **kwargs*)
Plot all data for the selected plugin:

```
plot(plugin_name, **kwargs)
```

Arguments

plugin_name name of the plugin to plot data from

figure

- `True`: plot to file with default name.
- string: use this filename (+extension for format)
- `False`: only display

kwargs arguments for plugin plot function (in many cases provided by `gromacs.formats.XVG.plot()` and ultimately by `pylab.plot()`)

Example

Here we analyze a protein, which has three Cysteines (C96, C243, C372). We will use the `plugins.CysAccessibility` and the `plugins.Distances` plugin (arguments for `Distances` omitted):

```
from gromacs.analysis import Simulation
from gromacs.analysis.plugins import CysAccessibility, Distances

S = Simulation(tpr=..., xtc=..., analysisdir=...,
               plugins=[('CysAccessibility', {'cysteines': [96, 243, 372]}),
                        Distances(...),
                        ])
S.set_plugin('CysAccessibility')          # do CysAccessibility analysis
S.run()                                   # analyze trajectory and write files
S.analyze()                               # analyze output files
S.plot(figure=True)                       # plot and save the figure
```

The plugins can be supplied when the `Simulation` object is constructed, or they can be later added, e.g.

```
S.add_plugin(Distances(name='Dist2', ...))
```

This second `Distances` analysis would be available with

```
S.set_plugin('Dist2')
```

Other plugins might require no or a very different initialization. See the plugin documentation for what is required.

`analysis.core` – Core classes for analysis of Gromacs trajectories

This documentation is mostly of interest to programmers who want to write analysis plugins.

Programming API for plugins

Additional analysis capabilities are added to a `gromacs.analysis.Simulation` class with *plugin* classes. For an example see `gromacs.analysis.plugins`.

API description Analysis capabilities can be added with plugins to the simulation class. Each plugin is registered with the simulation class and provides at a minimum `run()`, `analyze()`, and `plot()` methods.

A plugin consists of a subclass of `Plugin` and an associated `Worker` instance. The former is responsible for administrative tasks and documentation, the latter implements the analysis code.

A plugin class must be derived from `Plugin` and typically bears the name that is used to access it. A plugin instance must be *registered* with a `Simulation` object. This can be done implicitly by passing the `Simulation` instance in the `simulation` keyword argument to the constructor or by explicitly calling the `Plugin.register()` method with the simulation instance. Alternatively, one can also register a plugin via the `Simulation.add_plugin()` method.

Registering the plugin means that the actual worker class is added to the `Simulation.plugins` dictionary.

A plugin must eventually obtain a pointer to the `Simulation` class in order to be able to access simulation-global parameters such as top directories or input files.

See `analysis.plugins.CysAccessibility` and `analysis.plugins._CysAccessibility` in `analysis/plugins/CysAccessibility.py` as examples.

API requirements

- Each plugin is contained in a separate module in the `gromacs.analysis.plugins` package. The name of the module *must* be the name of the plugin class in all lower case.
- The plugin name is registered in `gromacs.analysis.plugins.__plugins__`. (Together with the file naming convention this allows for automatic and consistent loading.)
- The plugin itself is derived from `Plugin`; the only changes are the doc strings and setting the `Plugin.worker_class` class attribute to a `Worker` class.
- The corresponding worker class is derived from `Worker` and must implement
  - `Worker.__init__()` which can only use keyword arguments to initialize the plugin. It must ensure that init methods of super classes are also called. See the existing plugins for details.
  - `Worker.run()` which typically generates the data by analyzing a trajectory, possibly multiple times. It should store results in files.
  - `Worker.analyze()` analyzes the data generated by `Worker.run()`.
  - `Worker.plot()` plots the analyzed data.
  - `Worker._register_hook()` (see below)
- The worker class can access parameters of the simulation via the `Worker.simulation` attribute that is automatically set when the plugin registers itself with `Simulations`. However, the plugin should *not* rely on `simulation` being present during initialization (__init__) because registration of the plugin might occur *after* init.

  This also means that one cannot use the directory methods such as `Worker.plugindir()` because they depend on `Simulation.topdir()` and `Simulation.plugindir()`.

  Any initialization that requires access to the `Simulation` instance should be moved into the `Worker._register_hook()` method. It is called when the plugin is actually being registered. Note that the hook *must* also call the hook of the super class before setting any values. The hook should pop any arguments that it requires and ignore everything else.

- Parameters of the plugin are stored in `Worker.parameters` (either as attributes or as key/value pairs, see the container class `gromacs.utilities.AttributeDict`).
- Results are stored in `Worker.results` (also a `gromacs.utilities.AttributeDict`).

Classes

class `Simulation`(***kwargs*)

Bases: `object`

Class that represents one simulation.

Analysis capabilities are added via plugins.

1. Set the *active plugin* with the `Simulation.set_plugin()` method.
2. Analyze the trajectory with the active plugin by calling the `Simulation.run()` method.
3. Analyze the output from `run()` with `Simulation.analyze()`; results are stored in the plugin's `results` dictionary.
4. Plot results with `Simulation.plot()`.

Set up a Simulation object.

Keywords

sim Any object that contains the attributes *tpr*, *xtc*, and optionally *ndx* (e.g. `gromacs.cbook.Transformer`). The individual keywrods such as *xtc* override the values in *sim*.

tpr Gromacs tpr file (**required**)

xtc Gromacs trajectory, can also be a trr (**required**)

edr Gromacs energy file (only required for some plugins)

ndx Gromacs index file

analysisdir directory under which derived data are stored; defaults to the directory containing the tpr [None]

plugins [list] plugin instances or tuples (*plugin class*, *kwarg dict*) or tuples (*plugin_class_name*, *kwarg dict*) to be used; more can be added later with `Simulation.add_plugin()`.

`add_plugin`(*plugin, **kwargs*)

Add a plugin to the registry.

- If *plugin* is a `Plugin` instance then the instance is directly registered and any keyword arguments are ignored.
- If *plugin* is a `Plugin` class object or a string that can be found in `gromacs.analysis.plugins` then first an instance is created with the given keyword arguments and then registered.

Arguments

plugin [class or string, or instance] If the parameter is a class then it should have been derived from `Plugin`. If it is a string then it is taken as a plugin name in `gromacs.analysis.plugins` and the corresponding class is added. In both cases any parameters for initizlization should be provided.

If *plugin* is already a `Plugin` instance then the kwargs will be ignored.

kwargs The kwargs are specific for the plugin and should be described in its documentation.

`set_plugin`(*plugin_name*)
Set the plugin that should be used by default.

If no *plugin_name* is supplied to `run()`, `analyze()` etc. then this will be used.

`get_plugin`(*plugin_name=None*)
Return valid plugin or the default for *plugin_name*=``None``.

`run`(*plugin_name=None, **kwargs*)
Generate data files as prerequisite to analysis.

`analyze`(*plugin_name=None, **kwargs*)
Run analysis for the plugin.

`plot`(*plugin_name=None, figure=False, **kwargs*)
Plot all data for the selected plugin:

```
plot(plugin_name, **kwargs)
```

Arguments

plugin_name name of the plugin to plot data from

figure

- `True`: plot to file with default name.
- string: use this filename (+extension for format)
- `False`: only display

kwargs arguments for plugin plot function (in many cases provided by `gromacs.formats.XVG.plot()` and ultimately by `pylab.plot()`)

`run_all`(***kwargs*)
Execute the run() method for all registered plugins.

`analyze_all`(***kwargs*)
Execute the analyze() method for all registered plugins.

`_apply_all`(*func, **kwargs*)
Execute *func* for all plugins.

`topdir`(**args*)
Returns a path under self.analysis_dir, which is guaranteed to exist.

Note: Parent dirs are created if necessary.

`plugindir`(*plugin_name, *args*)
Directory where the plugin creates and looks for files.

`check_file`(*filetype, path*)
Raise `ValueError` if path does not exist. Uses *filetype* in message.

`has_plugin`(*plugin_name*)
Returns True if *plugin_name* is registered.

`check_plugin_name`(*plugin_name*)
Raises a exc:*ValueError* if *plugin_name* is not registered.

current_plugin
The currently active plugin (set with `Simulation.set_plugin()`).

class **Plugin**(*name=None, simulation=None, **kwargs*)
Plugin class that can be added to a `Simulation` instance.

All analysis plugins must be derived from this base class.

If a `Simulation` instance is provided to the constructore in the *simulation* keyword argument then the plugin instantiates and registers a worker class in `Simulation.plugins` and adds the `Simulation` instance to the worker.

Otherwise the `Plugin.register()` method must be called explicitly with a `Simulation` instance.

The plugin class handles the administrative tasks of interfacing with the `Simulation` class. The worker runs the analysis.

Note: If multiple Plugin instances are added to a Simulation one *must* set the *name* keyword argument to distinguish the instances. Plugins are referred to by this name in all further interactions with the user. There are no sanity checks: A newer plugin with the same *name* simply replaces the previous one.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

plugin_name
Name of the plugin; this must be a *unique* identifier across all plugins of a `Simulation` object. It should also be human understandable and must be a valid python identifier as it is used as a dict key.

simulation
The `Simulation` instance who owns the plugin. Can be `None` until a successful call to `register()`.

worker
The `Worker` instance of the plugin.

worker_class
actual plugin `gromacs.analysis.core.Worker` class (name with leading underscore)

register(*simulation*)
Register the plugin with the `Simulation` instance.

This method also ensures that the worker class knows the simulation instance. This is typically required for its `run()`, `analyze()`, and `plot()` methods.

class **Worker**(***kwargs*)
Bases: `gromacs.utilities.FileUtils`

Base class for a plugin worker.

Set up Worker class.

Keywords

plugin [instance] The `Plugin` instance that owns this worker. **Must be supplied.**

simulation A :class:Simulation‘ object, required for registration, but can be supplied later.

kwargs All other keyword arguments are passed to the super class.

topdir(**args*)
Returns a directory located under the simulation top directory.

plugindir(**args*)
Returns a directory located under the plugin top directory.

savefig(*filename=None, ext='png'*)
Save the current figure under the default name or *filename*.

Uses the supplied format and extension *ext*.

_register_hook(***kwargs*)
Things to initialize once the `Simulation` instance is known.

The hook is called from `Plugin.register()`.

Note: Subclasses should do all their `Simulation` - dependent initialization in their own `_register_hook()` which **must** call the super class hook via the `super` mechanism.

1.4.2 Support modules

Support modules contain code that simplifies working with `Simulation` instances. It also uses routines from `numkit`.

`analysis.collections` – Handling of groups of simulation instances

This module contains classes and functions that combine multiple `gromacs.analysis.core.Simulation` objects. In this way the same kind of analysis or plotting task can be carried out simultaneously for all simulations in the collection.

class Collection()
Multiple objects (organized as a list).

Methods are applied to all objects in the Collection and returned as new Collection:

```
>>> from gromacs.analysis.collections import Collection
>>> animals = Collection(['ant', 'boar', 'ape', 'gnu'])
>>> animals.startswith('a')
Collection([True, False, True, False])
```

Similarly, attributes are returned as a Collection.

Using `Collection.save()` one can save the whole collection to disk and restore it later with the `Collection.load()` method

```
>>> animals.save('zoo')
>>> arc = Collection()
>>> arc.load('zoo')
>>> arc.load('zoo', append=True)
>>> arc
['ant', 'boar', 'ape', 'gnu', 'ant', 'boar', 'ape', 'gnu']
```

1.4.3 Analysis plugins

Analysis plugins consist of a book-keeping class derived from `gromacs.analysis.core.Plugin` and a "worker" class (a child of `gromacs.analysis.core.Worker`), which contains the actual analysis code.

Plugins

Plugin modules are named like the plugin class but the filename is all lower case. All plugin classes are available in the `gromacs.analysis.plugins` name space.

`analysis.plugins` – Plugin Modules

Classes for `gromacs.analysis.core.Simulation` that provide code to analyze trajectory data.

New analysis plugins should follow the API sketched out in `gromacs.analysis.core`; see an example for use there.

List of plugins Right now the number of plugins is limited. Feel free to contribute your own by sending it to the package author. You will be acknowledged in the list below.

Table 1.3: Plugins for analysis.

plugin	author	description
`CysAccessibility`	[1]	estimate accessibility of Cys residues by water
`HelixBundle`	[1]	g_bundle analysis of helices
`Distances`	[1]	time series of distances
`MinDistances`	[1]	time series of shortest distances
`COM`	[1]	time series of centres of mass
`Dihedrals`	[1]	analysis of dihedral angles
`RMSF`	[1]	calculate root mean square fluctuations
`RMSD`	[1]	calculate root mean square distance
`Energy`	[1]	terms from the energy file

Table 1.4: Plugins for trajectory manipulation and status queries.

plugin	author	description
`Trajectories`	[1]	write xy-fitted trajectories
`StripWater`	[1]	remove solvent (and optionally fit to reference)
:class:‘ProteinOnly	[1]	remove all atoms except the Protein (and optionally fit to reference)
`Ls`	[1]	simple **ls** (for testing)

Plugin classes

class **`CysAccessibility`**(*name=None, simulation=None, **kwargs*)

CysAccessibility plugin.

For each frame of a trajectory, the shortest distance of all water oxygens to all cysteine sulphur atoms is computed. For computational efficiency, only distances smaller than a cutoff are taken into account. A histogram of the distances shows how close water molecules can get to cysteines. The closest approach distance should be indicative of the reactivity of the SH group with crosslinking agents.

class **`CysAccessibility`**(*cysteines, [cys_cutoff, [name, [simulation]]]*)

Arguments

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

cysteines [list] list of *all* resids (eg from the sequence) that are used as labels or in the form 'Cys<resid>'. (**required**)

cys_cutoff [number] cutoff in nm for the minimum S-OW distance [1.0]

Note that *all* Cys residues in the protein are analyzed. Therefore, the list of cysteine labels *must* contain as many entries as there are cysteines in the protein. There are no sanity checks.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

`worker_class`
alias of `_CysAccessibility`

class **`HelixBundle`**(*name=None, simulation=None, **kwargs*)
HelixBundle plugin.

`gromacs.g_bundle()` helix analysis

class **`HelixBundle`**(*[helixtable, offset, with_kinks, [name, [simulation]]]*)

Arguments

helixtable reST table with columns "name", "top", "bottom", "kink"; see `gromacs.analysis.plugins.helixbundle` for details

offset add the *offset* to the residue numbers in *helixtable* [0]

helixndx provide a index file with the appropriate groups instead of the table; also requires *na*

na number of helices

with_kinks take kinks into account [True]

name plugin name [HelixBundle]

simulation The `gromacs.analysis.Simulation` instance that owns the plugin. [None]

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

worker_class
alias of `_HelixBundle`

class **Distances**(*name=None, simulation=None, **kwargs*)
Distances plugin.

The distance between the center of mass of two index groups are calculated for each time step and written to files.

class **Distances**(*groups, ndx, [cutoff, [name, [simulation]]]*)

Arguments

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

groups [list of index group names] The first entry is the *primary group*, the second is the *secondary group.

ndx [index filename or list] All index files that contain the listed groups.

cutoff [float] A contact is recorded if the distance is <cutoff [0.6 nm]

Example:

Generate index files with the groups of interest, for instance with `gromacs.cbook.IndexBuilder`:

```
from gromacs.cbook import IndexBuilder
A_grp, A_ndx = IndexBuilder(tpr, ['@a 62549 & r NA'], names=['Na1_ion'], offset=-9,
                            out_ndx='Na1.ndx', name_all="Na1").combine()
B = IndexBuilder(tpr, ['S312:OG','T313:OG1','A38:O','I41:O','A309:O'], offset=-9,
                      out_ndx='Na1_site.ndx', name_all="Na1_site")
B_grp, B_ndx = B.combine()
all_ndx_files = [A_ndx, B_ndx]
```

To calculate the distance between "Na1" and the "Na1_site", create an instance with the appropriate parameters and add them to a `gromacs.analysis.Simulation` instance:

```
dist_Na1_site = Distances(name='Dsite', groups=['Na1', 'Na1_site'], ndx=all_ndx_files)
S.add_plugin(dist_Na1_site)
```

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

worker_class
alias of `_Distances`

class MinDistances(*name=None, simulation=None, **kwargs*)
MinDistances plugin.

The minimum distances between the members of at least two index groups and the number of contacts are calculated for each time step and written to files.

class Distances(*groups, ndx, [cutoff, [name, [simulation]]]*)

Arguments

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

groups [list of index group names] The first entry is the *primary group*. All other entries are *secondary groups* and the plugin calculates the minimum distance between members of the primary group and the members of each secondary group.

ndx [index filename or list] All index files that contain the listed groups.

cutoff [float] A contact is recorded if the distance is <cutoff [0.6 nm]

Example:

Generate index files with the groups of interest, for instance with `gromacs.cbook.IndexBuilder`:

```
from gromacs.cbook import IndexBuilder
A_grp, A_ndx = IndexBuilder(tpr, ['@a 62549 & r NA'], names=['Na1_ion'], offset=-9,
                            out_ndx='Na1.ndx', name_all="Na1").combine()
B = IndexBuilder(tpr, ['S312:OG','T313:OG1','A38:O','I41:O','A309:O'], offset=-9,
                      out_ndx='Na1_site.ndx', name_all="Na1_site")
B_grp, B_ndx = B.combine()
all_ndx_files = [A_ndx, B_ndx]
```

To calculate the distance between "Na1" and the "Na1_site", create an instance with the appropriate parameters and add them to a `gromacs.analysis.Simulation` instance:

```
dist_Na1_site = Distances(name='Dsite', groups=['Na1', 'Na1_site'], ndx=all_ndx_files)
S.add_plugin(dist_Na1_site)
```

To calculate the individual distances:

```
dist_Na1_res = Distances(name='Dres', groups=['Na1']+B.names, ndx=all_ndx_files)
S.add_plugin(dist_Na1_res)
```

(Keeping the second IndexBuilder instance `B` allows us to directly use all groups without typing them, `B.names = ['A309_O', 'S312_OG', 'I41_O', 'T313_OG1', 'A38_O']`.)

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

`worker_class`
alias of `_MinDistances`

class **`COM`**(*name=None, simulation=None, **kwargs*)
COM plugin.

Calculate the centre of mass (COM) of various index groups.

class **`COM`**(*group_names, [ndx, [offset, [name, [simulation]]]]*)

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

`worker_class`
alias of `_COM`

class **`Dihedrals`**(*name=None, simulation=None, **kwargs*)
Dihedrals plugin.

class **`Dihedrals`**(*dihedrals, [labels, [name, [simulation]]]*)

Keywords

dihedrals list of tuples; each tuple contains atom indices that define the dihedral.

labels optional list of labels for the dihedrals. Must have as many entries as *dihedrals*.

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

worker_class
alias of `_Dihedrals`

class RMSF (*name=None, simulation=None, **kwargs*)
RMSF plugin.

Compute the root mean square fluctuations (RMSF) of the C-alpha atoms. The trajectory is always fitted to the reference structure in the tpr file.

class RMSF (*[name, [simulation]]*)

Arguments

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

worker_class
alias of `_RMSF`

class RMSD (*name=None, simulation=None, **kwargs*)
RMSD plugin.

Calculation of the root mean square distance (RMSD) of a protein structure over the course of a MD simulation.

The trajectory is always fitted to the reference structure in the tpr file.

class RMSD (*[name, [simulation]]*)

Arguments

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

`worker_class`
alias of `_RMSD`

class `Energy`(*name=None, simulation=None, **kwargs*)
Energy plugin.

Analysis of terms in the Gromacs energy (edr) file.

class `Energy`(*[name, [simulation]]*)

Arguments

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

`worker_class`
alias of `_Energy`

class `Trajectories`(*name=None, simulation=None, **kwargs*)
Trajectories plugin.

Write new xy-fitted trajectories (see `gromacs.cbook.trj_fitandcenter()`),

class `Trajectories`(*[name, [simulation]]*)

Arguments

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

worker_class
alias of `_Trajectories`

class StripWater (*name=None, simulation=None, **kwargs*)
StripWater plugin.

Write a new trajectory which has the water index group removed.

class StripWater (*[selection, [name, [simulation]]]*)

Arguments

selection optional selection for the water instead of "SOL"

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

worker_class
alias of `_StripWater`

class ProteinOnly (*name=None, simulation=None, **kwargs*)
ProteinOnly plugin.

Write a new trajectory which has the water index group removed.

class ProteinOnly (*[selection, [name, [simulation, [...]]]]*)

Arguments

selection optional selection for the water instead of "SOL"

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

force `True` will always regenerate trajectories even if they already exist, `False` raises an exception, `None` does the sensible thing in most cases (i.e. notify and then move on).

dt [float or list of floats] only write every dt timestep (in ps); if a list of floats is supplied, write multiple trajectories, one for each dt.

compact [bool] write a compact representation

fit Create an additional trajectory from the stripped one in which the Protein group is rms-fitted to the initial structure. See `gromacs.cbook.Transformer.fit()` for details. Useful values:

- "xy" : perform a rot+trans fit in the x-y plane
- "all": rot+trans
- `None`: no fitting

If *fit* is not supplied then the constructore-default is used (`_ProteinOnly.parameters.fit`).

keepalso List of literal `make_ndx` selections that select additional groups of atoms that should also be kept in addition to the protein. For example *keepalso*=["'POPC"', 'resname DRUG'].

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

`worker_class`
alias of `_ProteinOnly`

class `Ls` (*name=None, simulation=None, **kwargs*)
ls plugin.

This simply lists the files on disk. It is useful for testing the plugin architecture.

class `Ls` (*[name, [simulation]]*)

Arguments

name [string] plugin name (used to access it)

simulation [instance] The `gromacs.analysis.Simulation` instance that owns the plugin.

Registers the plugin with the simulation class.

Specific keyword arguments are listed below, all other kwargs are passed through.

Arguments

name [string] Name of the plugin. Should differ for different instances. Defaults to the class name.

simulation [Simulation instance] The `Simulation` instance that owns this plugin instance. Can be `None` but then the `register()` method has to be called manually with a simulation instance later.

kwargs All other keyword arguments are passed to the Worker.

`worker_class`
alias of `_Ls`

Developer notes In principle all that needs to be done to automatically load plugins is to add their name to `__plugins__`. See the source code for further comments and how the auto loading of plugins is done.

`__plugins__`
All available plugin names are listed here. Because this is used to automatically set up imports a module file *must* be named like the plugin class it contains but in all lower case. For example, the *Distances* plugin class is contained in the module *distances* (the file `plugins/distances.py`).

`__plugin_classes__`
Gives access to all available plugin classes (or use the module __dict__)

Helper modules

Various helper classes that can be used by plugins. Because they are not necessarily restricted to a single plugin they have been moved into separate modules for code reuse.

`analysis.plugins.dist` — Helper Class for `g_dist`

`dist` contains helper classes for other analysis plugins that want to make use of the Gromacs command `g_dist`.

Overview The task we are solving is to analyze output from

```
g_dist -f md.xtc -s md.tpr -n cys_ow.ndx -dist 1.0 | bzip2 -vc > mindist_C60_OW_1nm.dat.bz2
```

and produce a histogram of minimum contact distances. This should provide an estimate for water accessibility of the atom (here: SG of Cys60).

File format `g_dist` with the `-dist CUTOFF` option writes to stdout the identity of all atoms within the cutoff distance and the distance itself:

```
Selected 22: 'CYSH_CYSH_60_&_SG'
Selected 25: 'OW'
....
t: 184  6682 SOL 35993 OW  0.955138 (nm)
t: 184  10028 SOL 46031 OW  0.803889 (nm)
t: 185  6682 SOL 35993 OW  0.879949 (nm)
t: 185  10028 SOL 46031 OW  0.738299 (nm)
t: 186  6682 SOL 35993 OW  0.897016 (nm)
t: 186  10028 SOL 46031 OW  0.788268 (nm)
t: 187  6682 SOL 35993 OW  0.997688 (nm)
....
```

Classes

class `Mindist`(*datasource*, *cutoff=None*)
The Mindist class allows analysis of the output from `g_dist -dist CUTOFF`.

Output is read from a file or stream. The raw data is transformed into a true 'mindist' time series (available in the `Mindist.distances` attribute): for each frame only the shortest distance is stored (whereas `g_dist` provides *all* distances below the cutoff).

Todo

- Save analysis to pickle or data files.

- Export data as simple data files for plotting in other programs.

Note: `gromacs.tools.G_mindist` is apparently providing exactly the service that is required: a time-series of the minimum distance between two groups. Feel free to use that tool instead :-).

Read mindist data from file or stream.

Arguments

datasource a filename (plain, gzip, bzip2) or file object

cutoff the `-dist CUTOFF` that was provided to `g_dist`; if supplied we work around a bug in `g_dist` (at least in Gromacs 4.0.2) in which sometimes numbers >> CUTOFF are printed.

`histogram`(*nbins=None, lo=None, hi=None, midpoints=False, normed=True*)

Returns a distribution or histogram of the minimum distances:

If no values for the bin edges are given then they are set to 0.1 below and 0.1 above the minimum and maximum values seen in the data.

If the number of bins is not provided then it is set so that on average 100 counts come to a bin. Set nbins manually if the histogram only contains a single bin (and then get more data)!

Keywords

nbins [int] number of bins

lo [float] lower edge of histogram

hi [float] upper edge of histogram

midpoints [boolean] False: return edges. True: return midpoints

normed [boolean] True: return probability distribution. False: histogram

`hist`

Histogram of the minimum distances.

`dist`

Distribution of the minimum distances.

`edges`

Edges of the histogram of the minimum distances.

`midpoints`

Midpoints of the histogram of the minimum distances.

`plot`(***kwargs*)

Plot histograms with matplotlib's plot() function:

```
plot(**histogramargs, **plotargs)
```

Arguments for both `Mindist.histogram()` and `pylab.plot()` can be provided (qv).

class `GdistData`(*stream*)

Object that represents the standard output of `g_dist -dist CUTOFF`.

Initialize from a stream (e.g. a pipe) and the iterate over the instance to get the data line by line. Each line consists of a tuple

(*frame*, *distance*)

Initialize with an open stream to the data (eg stdin or file).

Arguments

stream open stream (file or pipe or really any iterator providing the data from `g_dist`); the *stream* is not closed automatically when the iterator completes.

`__iter__`()
Iterator that filters the input stream and returns (frame, distance) tuples.

`gromacs.analysis.plugins.gridmatmd` — Lipid bilayer analysis helper

This helper module contains code to drive `GridMAT-MD.pl`, available from the GridMAT-MD home page and written by WJ Allen et al [Allen2009] . The GromacsWrapper distribution comes with version 1.0.2 of `GridMAT-MD.pl` and includes a small patch so that it can accept filenames on the command line.

References

Module contents

class **`GridMatMD`**(*config, filenames*)
Analysis of lipid bilayers with GridMAT-MD.

It requires a configuration file and a list of structure files (gro or pdb) as input. See the documentation (pdf) for the format of the config file. Note that the *bilayer* keyword will be ignored in the config file.

Set up GridMAT-MD analysis.

Arguments

config [filename] input file for GridMAT-MD (see docs)

filenames [list or glob-pattern] list of gro or pdb files, or a glob pattern that creates such a list

`imshow`(*name, **kwargs*)
Display array *name* with `pylab.imshow`.

`run`()
Run analysis on all files and average results.

`run_frame`(*frame*)
Run GridMAT-MD on a single *frame* and return results.

Arguments *frame* is a filename (gro or pdb)

Returns a dict of `GridMapData` objects; the keys are "top", "bottom", "average"

`save`(*name*)
Save object as pickle.

class **`GridMatData`**(*filename, shape=None, delta=None*)
Represent GridMatMD data file.

The loaded array data is accessible as a numpy array in `GridMatData.array` and bins and midpoints as `GridMatData.bins` and `GridMatData.midpoints` respectively.

Load the data into a numpy array.

The *filename* is an output file from GridMAT-MD. *shape* and *delta* are optional. The *shape* of the array is parsed from the filename if not provided. The spacing is set to (1,1) if not provided.

Arguments

filename 2D grid as written by GridMAT-MD

shape Shape tuple (NX, NY) of the array in filename.

delta Tuple of bin sizes of grid (DX, DY).

`imshow`(***kwargs*)
Display data as a 2D image using `pylab.imshow()`.

`parse_filename`(*filename*)
Get dimensions from filename

class `Grid2D`(*data, bins*)
Represents a 2D array with bin sizes.

Addition and subtraction of grids is defined for the arrays and the bins. Multiplication and division with scalars is also defined. Each operation returns a new `Grid2D` object.

(Actually, it should work for arrays of any dimension, not just 2D.)

Initialize the Grid2D instance.

Arguments

data array data, e.g. a list of array

bins tuple of lists of bin **edges**, one for each dimension

`imshow`(***kwargs*)
Display data as a 2D image using `pylab.imshow()`.

`__add__`(*other*)
Add arrays and bins (really only makes sense when averaging).

`__sub__`(*other*)
Subtract other from self (also subtracts bins... which is odd but consistent).

`__mul__`(*x*)
Multiply arrays (and bins) by a scalar *x*.

`__div__`(*x*)
Divide arrays (and bins) by a scalar *x*.

1.5 Auxiliary modules

A number of small python modules are currently bundled with *GromacsWrapper* as they have proven useful in the context of which the main package is being used. However, *GromacsWrapper* does not depend on them and they could be safely ignored (and might be removed if we ever get to a 1.0 release...).

1.5.1 `numkit` — Numerical analysis toolkit

A collection of functions and classes built on top of NumPy and SciPy to aid the numerical analysis of data. It is geared towards the use of data coming from molecular simulations, namely time series. It is used in `gromacs.analysis`.

See Also:

Core functionality is based on SciPy (`scipy` module).

Contents:

`numkit.fitting` — Fitting data

`Pearson_r` (*x, y*)

Pearson's r (correlation coefficient).

Pearson(x,y) –> correlation coefficient

x and *y* are arrays of same length.

Historical note – Naive implementation of Pearson's r :: Ex = scipy.stats.mean(x) Ey = scipy.stats.mean(y) covxy = numpy.sum((x-Ex)*(y-Ey)) r = covxy/math.sqrt(numpy.sum((x-Ex)**2)*numpy.sum((y-Ey)**2))

`linfit` (*x, y, dy=, []*)

Fit a straight line y = a + bx to the data in *x* and *y*.

Errors on y should be provided in dy in order to assess the goodness of the fit and derive errors on the parameters.

linfit(x,y[,dy]) –> result_dict

Fit y = a + bx to the data in x and y by analytically minimizing chi^2. dy holds the standard deviations of the individual y_i. If dy is not given, they are assumed to be constant (note that in this case Q is set to 1 and it is meaningless and chi2 is normalised to unit standard deviation on all points!).

Returns the parameters a and b, their uncertainties sigma_a and sigma_b, and their correlation coefficient r_ab; it also returns the chi-squared statistic and the goodness-of-fit probability Q (that the fit would have chi^2 this large or larger; Q < 10^-2 indicates that the model is bad — Q is the probability that a value of chi-square as _poor_ as the calculated statistic chi2 should occur by chance.)

Returns result_dict with components

intercept, sigma_intercept a +/- sigma_a

slope, sigma_slope b +/- sigma_b

parameter_correlation correlation coefficient r_ab between a and b

chi_square chi^2 test statistic

Q_fit goodness-of-fit probability

Based on 'Numerical Recipes in C', Ch 15.2.

class `FitFunc` (*x, y*)

Fit a function f to data (x,y) using the method of least squares.

The function is fitted when the object is created, using `scipy.optimize.leastsq()`. One must derive from the base class `FitFunc` and override the `FitFunc.f_factory()` (including the definition of an appropriate local `fitfunc()` function) and `FitFunc.initial_values()` appropriately. See the examples for a linear fit `FitLin`, a 1-parameter exponential fit `FitExp`, or a 3-parameter double exponential fit `FitExp2`.

The object provides two attributes

`FitFunc.parameters` list of parameters of the fit

`FitFunc.message` message from `scipy.optimize.leastsq()`

After a successful fit, the fitted function can be applied to any data (a 1D-numpy array) with `FitFunc.fit()`.

`f_factory` ()

Stub for fit function factory, which returns the fit function. Override for derived classes.

`fit` (*x*)

Applies the fit to all *x* values

initial_values()
List of initital guesses for all parameters p[]

class **FitLin** (*x, y*)
y = f(x) = p[0]*x + p[1]

class **FitExp** (*x, y*)
y = f(x) = exp(-p[0]*x)

class **FitExp2** (*x, y*)
y = f(x) = p[0]*exp(-p[1]*x) + (1-p[0])*exp(-p[2]*x)

`numkit.timeseries` — Time series manipulation and analysis

autocorrelation_fft (*series, remove_mean=True, paddingcorrection=True, normalize=False, **kwargs*)
Calculate the auto correlation function.

autocorrelation_fft(series,remove_mean=False,**kwargs) –> acf

The time series is correlated with itself across its whole length. Only the [0,len(series)[interval is returned.

By default, the mean of the series is subtracted and the correlation of the fluctuations around the mean are investigated.

For the default setting remove_mean=True, acf[0] equals the variance of the series, acf[0] = Var(series) = <(series - <series>)**2>.

Optional:

- The series can be normalized to its 0-th element so that acf[0] == 1.
- For calculating the acf, 0-padding is used. The ACF should be corrected for the 0-padding (the values for larger lags are increased) unless mode='valid' is set (see below).

Note that the series for mode='same'|'full' is inaccurate for long times and should probably be truncated at 1/2*len(series)

Arguments

series (time) series, a 1D numpy array of length N

remove_mean `False`: use series as is; `True`: subtract mean(series) from series [`True`]

paddingcorrection `False`: corrected for 0-padding; `True`: return as is it is. (the latter is appropriate for periodic signals). The correction for element 0=<i<N amounts to a factor N/(N-i). Only applied for modes != "valid" [`True`]

normalize `True` divides by acf[0] so that the first element is 1; `False` leaves un-normalized [`False`]

mode "full" | "same" | "valid": see `scipy.signal.fftconvolve()` ["full"]

kwargs other keyword arguments for `scipy.signal.fftconvolve()`

tcorrel (*x, y, nstep=100, debug=False*)
Calculate the correlation time and an estimate of the error of <y>.

The autocorrelation function f(t) is calculated via FFT on every *nstep* of the **fluctuations** of the data around the mean (y-<y>). The normalized ACF f(t)/f(0) is assumed to decay exponentially, f(t)/f(0) = exp(-t/tc) and the decay constant tc is estimated as the integral of the ACF from the start up to its first root.

See Frenkel and Smit, Academic Press, San Diego 2002, p526.

Note: *nstep* should be set sufficiently large so that there are less than ~50,000 entries in the input.

Arguments

x 1D array of abscissa values (typically time)

y 1D array of the ibservable y(x)

nstep only analyze every *nstep* datapoint to speed up calculation [100]

Returns dictionary with entries *tc* (decay constant in units of *x*), *t0* (value of the first root along x (y(t0) = 0)), *sigma* (error estimate for the mean of y, <y>, corrected for correlations in the data).

`numkit.integration` — Numerical integration of data

See Also:

`scipy.integrate`

`simps_error`(*y, x=None, dx=1, axis=-1, even='avg'*)

Error on integral evaluated with Simpson's rule from errors of points, y.

Evaluate the integral with `scipy.integrate.simps()`. For a given vector *y* of errors on the function values, the error on the integral is calculated via propagation of errors.

Arguments

y errors for the tabulated values of the integrand f

x values of abscissa at which f was tabulated (can be `None` and then *dx* should be provided)

dx constant spacing of the abscissa

axis axis in *y* along which the data lies

even see `scipy.integrate.simps()` ('avg', 'first', 'last')

exception `LowAccuracyWarning`

Warns that results may possibly have low accuracy.

`numkit.observables` — Observables as quantities with errors

Example showing how to use `QuantityWithError`:

```
>>> from numkit.observables import QuantityWithError
>>> a = QuantityWithError(2.0, 1.0)
>>> a2 = QuantityWithError(2.0, 1.0)  # 2nd independent measurement of a
>>> a3 = QuantityWithError(2.0, 1.0)  # 3rd independent measurement of a
>>> b = QuantityWithError(-1, 0.5)
>>> a+a
4 (2)
>>> a+a2
4 (1.41421)
>>> (a+a+a)/3
2 (1)
>>> (a+a2+a3)/3
2 (0.57735)
>>> a/b
-2 (1.41421)
```

Note that each quantity has an identity: it makes a difference to the error of a combined quantity such as a+a if the inputs are independent measurements of the same.

class **QuantityWithError** (*value, error=None, qid=None, **kwargs*)

Number with error and basic error propagation arithmetic.

The quantity is assumed to be a mean of an observable (`value`) with an associated (Gaussian) error `error` (which is the sqrt of the variance `variance` of the data).

The covariance is not taken into account in the error propagation (i.e. all quantities are assumed to be uncorrelated) with the exception of the case of binary relations of the quantity with itself. For instance, a*a is correctly interpreted as a**2). However, this behaviour is not guaranteed to work for any complicated expression.

When combining with pure numbers you have to wrap these numbers if they are to the left of a `QuantityWithError`, e.g.

```
>>> a = QuantityWithError(2.0, 1.0)
>>> 1/a
TypeError
>>> QuantityWithError(1.0)/a
0.5 (0.25)
```

static **asQuantityWithError** (*other*)

Return a `QuantityWithError`.

If the input is already a `QuantityWithError` then it is returned itself. This is important because a new quantity x' would be considered independent from the original one x and thus lead to different error estimates for quantities such as x*x versus x*x'.

copy ()

Create a new quantity with the same value and error.

deepcopy ()

Create an exact copy with the same identity.

isSame (*other*)

True if *other* is the same observable.

Also True if *other* was derived from *self* without using any other independent quantities with errors.

See Also:

Various packages that describe quantities with units.

1.5.2 `vmd` — Remote Tcl commands in VMD

The `vmd` module contains a very simple client that can connect to a locally running server process in VMD. This allows running VMD Tcl commands remotely. See VMD Tcl Text Commands for all available commands.

The `vmd` module is independent from `gromacs` and distribute for convenience. It can be used to implement additional visualization and analysis tasks.

VMD control

Simple client to transmit Tcl commands to a server running in VMD.

VMD and the server run locally and can be started from the module. Once the server is running, one can use `vmd.client` to communicate with the server process via a local socket.

Example

Start a VMD server and connect:

```
from vmd.control import *
VMD = server()
VMD.command('molecule new load 1AKE')
```

or start an interactive Tcl session connected to a running VMD server process:

```
interactive(host)
asyncore.loop()       # necessary
```

See VMD Tcl Text Commands for all available commands.

1.5.3 `edPDB` — Simple processing of PDB files

The `edPDB` module contains extensions to the Biopython `Bio.PDB` module that help with editing PDB files in preparation for molecular dynamics simulations.

This module is completely independent from `gromacs` and distributed with *GromacsWrapper* as a convenience. As all other code, it is in a steady state of development and flux...

The `edPDB` module is distributed under the Biopython licence .

Contents:

`edPDB` – editing PDB files

A collection of python snippets to quickly edit pdb files. This is typically used for setting up system for MD simulations.

Typically, one instantiates a `edPDB.cbook.PDB` object (which can also be accessed as `edPDB.PDB`) and uses the methods defined on it to write new pdb files.

The cook book `edPDB.cbook` contains some more specialized functions that are not integrated into `PDB` yet; study the source and use them as examples.

Built on top of `Bio.PDB` from Biopython.

Modules

`edPDB.cbook` Cook-book with short functions that show how to implement basic functionality.

`edPDB.xpdb` Extensions to the Bio.PDB class.

`edPDB.selections` Selections that can be used to extract parts of a pdb.

Future plans

Eventually using this module should become as intuitive as `grep`, `sed` and `cat` of pdb files.

`edPDB.cbook` – Recipes for editing PDB files

The cook book contains short python functions that demonstrate how to implement basic PDB editing functionality. They do not do exhaustive error checking and might have to be altered for your purpose.

class **`PDB`** (*pdbname*)

Class that represents a PDB file and allows extractions of interesting parts.

The structure itself is never changed. In order to extract sub-parts of a structure one selects these parts and writes them as new pdb file.

The advantage over a simple **grep** is that you will be able to read any odd pdb file and you will also able to do things like `extract_protein()` or `extract_lipids()`.

Load structure from file *pdbname*.

`extract_lipids` (*filename, lipid_resnames='POPC|POPG|POPE|DMPC|DPPE|DOPE', **kwargs*)

Write a pdb file with the lipids extracted.

Note that resnames are also tried truncated to the first three characters, which means that POPE and POPG are identical and cannot be distinguished.

`extract_notprotein` (*filename, **kwargs*)

Write a pdb file without any amino acids extracted.

`extract_protein` (*filename, **kwargs*)

Write a pdb file with the protein (i.e. all amino acids) extracted.

`extract_resnames` (*filename, resnames, **kwargs*)

Write a pdb file with *resnames* extracted.

`residues_by_resname` (*resnames, **kwargs*)

Return a list of BioPDB residues that match *resnames*.

resnames can be a string or a list.

`residues_by_selection` (*selection*)

Return a list of BioPDB residues that are selected by *selection*.

selection must be BioPDB.PDB.PDBIO.Select instance (see for example `edPDB.xpdb.ProteinSelect`).

`write` (*filename, **kwargs*)

Write pdbfile which includes or excludes residues.

Arguments

filename output pdb filename

inclusions list of residues to include

exclusions list of residues to exclude

chain relabel the selection with a new chain identifier

Residues must be BioPDB residues as returned by, for instance, `residues_by_resname()`.

Note: Currently only either *inclusions* or *exclusions* can be supplied, not both.

`align_ligand` (*protein_struct, ligand_struct, ligand_resname, output='ligand_aligned.pdb'*)

Align a ligand to the same ligand in a protein, based on the heavy atoms.

This is useful when a new ligand was generated with hydrogens but the position in space changed.

Arguments

protein_struct protein + ligand pdb

ligand_struct ligand pdb

ligand_resname residue name of the ligand in the file *protein_struct*

ligand_aligned output

Returns RMSD of the fit in Angstroem.

Warning: Assumes only heavy atoms in PDB (I think... check source!)

`remove_overlap_water`(*pdbname, output, ligand_resname, distance=3.0, water='SOL', **kwargs*)

Remove water (SOL) molecules overlapping with ligand.

Arguments

pdbname pdb file that contains the ligand and the water

output pdb output filename

ligand_resname name of the ligand residue(s) in the pdb

distance overlap is defined as a centre-centre distance of any solvent OW atom with any ligand atom of less than *distance*

Note: The residue and atom numbering will be fairly meaningless in the final PDB because it wraps at 100,000 or 10,000.

Also make sure that there are either consistent chain identifiers or none (blank) because otherwise the residue blocks migh become reordered. (This is due to the way the Bio.PDB.PDBIO writes files.)

`edPDB.xpdb` – Extensions to `Bio.PDB`

Extensions to Bio.PDB, such as handling of large pdb files and some useful selections (see `edPDB.selections`).

Partly published on http://biopython.org/wiki/Reading_large_PDB_files

License: like Biopython

Module content

class `SloppyStructureBuilder`(*verbose=False*)

Cope with resSeq < 10,000 limitation by just incrementing internally.

Solves the follwing problem with `Bio.PDB.StructureBuilder.StructureBuilder`: Q: What's wrong here??

> Some atoms or residues will be missing in the data structure. WARNING: Residue (' ', 8954, ' ') redefined at line 74803. PDBConstructionException: Blank altlocs in duplicate residue SOL (' ', 8954, ' ') at line 74803.

A: resSeq only goes to 9999 –> goes back to 0 (PDB format is not really good here)

Warning: H and W records are probably not handled yet (don't have examples to test)

class `SloppyPDBIO`(*use_model_flag=0*)

PDBIO class that can deal with large pdb files as used in MD simulations.

- resSeq simply wrap and are printed modulo 10,000.

- atom numbers wrap at 99,999 and are printed modulo 100,000

@param use_model_flag: if 1, force use of the MODEL record in output. @type use_model_flag: int

class **`AtomGroup`**(*atoms=None*)

`get_structure`(*pdbfile, pdbid='system'*)

`write_pdb`(*structure, filename, **kwargs*)
Write Bio.PDB molecule *structure* to *filename*.

Arguments

structure Bio.PDB structure instance

filename pdb file

selection Bio.PDB.Selection

exclusions list of **residue** instances that will *not* be included

inclusions list of **residue** instances that will be included

chain set the chain identifier for **all** atoms written; this can be useful to simply to erase all chain ids by setting it to ' '

Typical use is to supply a list of water molecules that should not be written or a ligand that should be include.

Note: Currently only one of *selection, *exclusions* or *inclusions* is supported.

`edPDB.selections` — Selections

Extensions to Bio.PDB, some useful selections.

Partly published on http://biopython.org/wiki/Reading_large_PDB_files

License: like Biopython

Selection classes

Provide an instance to PDBIO to select a subset of a structure or use it with `residues_by_selection()` to obtain a list of residues.

class **`ResnameSelect`**(*resnames, complement=False*)
Select all atoms that match *resnames*.

Supply a *resname*, e.g. 'SOL' or 'PHE' or a list.

class **`ResidueSelect`**(*residues, complement=False*)
Select all atoms that are in the *residues* list.

Supply a list of Bio.PDB residues for the search.

class **`NotResidueSelect`**(*residues, complement=False*)
Select all atoms that are *not* in the *residues* list.

(Same as `ResidueSelect(residues, complement=True)`.)

Supply a list of Bio.PDB residues for the search.

class **`ProteinSelect`**(*complement=False*)
Select all amino acid residues.

class `NotProteinSelect`(*complement=False*)
Select all non-aminoacid residues.

Supply a list of Bio.PDB residues for the search.

Selection functions

Functions always act on a structure and return a list of residues.

`residues_by_resname`(*structure, resnames*)
Return a list of residue instances that match *resnames*.

resnames can be a single string or a list of strings.

`residues_by_selection`(*structure, selection*)
General residue selection: supply a Bio.PDB.PDBIO.Select instance.

`find_water`(*structure, ligand, radius=3.0, water='SOL'*)
Find all water (SOL) molecules within radius of ligand.

Arguments

structure Bio.PDB structure of system with water

ligand [list] Bio.PDB list of atoms of the ligand (Bio.PDB.Atom.Atom instances)

radius [float] Find waters for which the ligand-atom - OW distance is < radius [3.0]

water [string] resname of a water molecule [SOL]

Returns list of residue instances

Utility functions

`canonical`(*resname*)
Return canonical representation of resname.

space stripped and upper case

`PROTEIN_RESNAMES`
List of residue names that determine what is recognized as a protein with `ProteinSelect`. Can be extended with non-standard residues.

Helper modules

`edPDB.utilities` – Helper functions and classes

The module defines some convenience functions and classes that are used in other modules

Functions Functions that improve list processing and which do *not* treat strings as lists:

`iterable`(*obj*)
Returns `True` if *obj* can be iterated over and is *not* a string.

`asiterable`(*obj*)
Returns obj so that it can be iterated over; a string is *not* treated as iterable

Configure logging for edPDB analysis.

Import this module if logging is desired in application code.

See Also:

`Bio.PDB` in Biopython `edPDB` is built on top of the great work done in the Bio.PDB module

pdbcat Andrew Dahlke's pdbcat tools might also be a useful alternative if you prefer shell to python.

1.5.4 `staging` — Staging of input/output files on a queuing system

The `staging` module provides a framework to run python scripts easily through a queuing system that requires copying of files to the scratch directory on compute nodes ("staging"). Instead of submitting a shell script to the queuing system, one uses a python script that imports the module and instantiates a `Job` class that contains methods to manage staging.

`staging` — Staging module overview

The 'staging' module provides a framework to run python scripts easily through a queuing system that requires copying of files to the scratch directory on compute nodes.

Load the appropriate submodule at the of the script. Currently available submodules are:

`staging.SunGridEngine` Use the `staging.SunGridEngine.Job` class when running under Sun Grid Engine.

`staging.Local` Runs the job without any staging.

Example

```
from staging.SunGridEngine import Job

job = Job(inputfiles=dict(psf = 'inp/apo.psf',
                          dcd = 'trj/prod.dcd'),
          outputfiles=dict(dx = '*.dx', pickle = '*.pickle'))

job.stage()      # copy files to staging directory, creating dirs

# your python script here...
# access all filenames as job.filenames[<KEY>]

job.unstage()    # copies files backs and creates dirs as needed
job.cleanup()    # removes stage dir, careful!
```

`staging.SunGridEngine` — staging class for SunGridEngine

Primitive framework for staging jobs in Sun Grid Engine via a customized `Job` class.

Example python submission script

Write the SGE script like this:

```
#!/usr/bin/env python
#$ -N bulk
#$ -S /usr/bin/python
#$ -v PYTHONPATH=/home/oliver/Library/python-lib
#$ -v LD_LIBRARY_PATH=/opt/intel/cmkl/8.0/lib/32:/opt/intel/itc60/slib:/opt/intel/ipp41/ia32_itanium
#$ -r n
#$ -j y
# The next line is IMPORTANT when you are using the default for Job(startdir=None)
#$ -cwd

from staging.SunGridEngine import Job

job = Job(inputfiles=dict(psf = 'inp/crbp_apo.psf',
                          dcd = 'trj/rmsfit_1opa_salt_ewald_shake_10ang_prod.dcd'),
          outputfiles=dict(dx = '*.dx', pickle = '*.pickle'),
          variables=dict(normalize = True, ...))

job.stage()
F = job.filenames  # use F[key] to reference filenames from inputfiles or outputfiles
V = job.variables  # and V[key] for the variables

# your python script here...
print "psf: %(psf)s  dcd: %(dcd)" % F
print "normalize = %(normalize)s" % V


job.unstage()
job.cleanup()   # removes stage dir, careful!
```

Description of the `Job` class

class Job (**args*, ***kwargs*)

The Job class encapsulates the SGE job and allows for clean staging and unstaging.

Set up the Job:

```
job = Job(inputfiles=dict(...),outputfiles=dict(...),variables=dict(...),**kwargs)
```

inputfiles and *outputfiles* are dictionaries with arbitrary keys; each item is a path to a file relative to the startdir (which by default is the directory from which the SGE job starts — use the `#$ -cwd` flag!). If the files are not relative to the start dir then new directories are constructed under the stage dir; in this instance it uis important that the user script *only* uses the filenames in `Job.filenames`: These have the proper paths of the local (staged) files for the script to operate on.

With

```
job.stage()
```

inputfiles are copied to the stagedir on the node's scratch dir and sub directories are created as necessary; directories mentioned as part of the outputfiles are created, too.

```
job.unstage()
```

copies back all files mentioned in output files (again, use directories as part of the path as necessary) and create the directories in the startdir if needed. For the outputfiles one can also use shell-style glob patterns, e.g. `outfiles = {'all_dcd': '*.dcd', 'last_data':'*[5-9].dat'}`

Sensible defaults are automatically selected for startdir (cwd) and stagedir (/scratch/USER/JOB_NAME.JOB_ID).

If the script is not run through SGE (i.e. the environment variable **JOB_NAME** is not set) then the script is run without staging; this is pretty much equivalent to using

```
from staging.Local import Job
```

(i.e. using the `staging.Local.Job` class).

Attributes

`input` inputfiles dict (relative to startdir or absolute)

`output` outputfiles dict (relative to startdir or absolute, can contain globs)

`filenames` merged dict of input and output, pointing to *staged* files

`variables` variables dict

Methods

`stage()` setup job on the nodes in stagedir

`unstage()` retrieve results to startdir

`cleanup()` remove all files on the node (rm -rf stagedir)

Set up SGE job.

Arguments

inputfiles dict of input files (with relative path to startdir); globs are not supported.

outputfiles dict of result files or glob patterns (relative to stagedir == relative to startdir)

variables key/value pairs that can be used in the script as Job.variables[key]

startdir path to the directory where the input can be found (must be nfs-mounted on node)

stagedir local scratch directory on node; all input files are copied there. The default should be ok.

JOB_NAME unique identifier (only set this if this NOT submitted through the Gridengine queuing system AND if the files should be copied to a scratch disk (i.e. staging proceeds as it would for a SGE-submitted job).)

SGE_TASK_ID fake a task id (use with JOB_NAME)

cleanup()

Remove stage dir

save(*filename*)

Save the Job() as a pickled file.

Restore with

```
import staging.SunGridengine
import cPickle
job = cPickle.load(open(<filename>,'r'))
```

stage()
Copy all input files to the scratch directory.

unstage()
Copy results back. Shell-style glob patterns are allowed.

Helper functions for building job arrays

getline_from_arraylist(*filename=None, ENVNAME='ARRAYLIST', default='arraylist.txt'*)
Read a list of values from filename and return the line that corresponds to the current SGE_TASK_ID.

line = get_line_from_arraylist(filename=None,ENVNAME='ARRAYLIST',default="arraylist.txt")

fields will be different depending on the value of **SGE_TASK_ID** (set by SunGridengine). The lines are simply numbered consecutively.

Arguments

filename name of the arraylist file

ENVNAME try to get filename from environment variable if filename is not set

default if all fails, try this as a default filename

File format:

```
# comment lines are ignored as are whitespace lines
# only the first column is read; the internal numbering starts at 1
line1 ...   <---- task id 1
line2 ...   <---- task id 2
# more comments, they are NOT counted for the task id
line3 ...   <---- task id 3
...
```

Ignores white space lines and lines starting with `#`. Lines are stripped of left and right white space.

get_fields_from_arraylist(***kwargs*)
Read a list of values from filename and return the line that corresponds to the current SGE_TASK_ID.

get_line_from_arraylist(filename=None,ENVNAME='ARRAYLIST',default="arraylist.txt") -> fields

fields will be different depending on the value of SGE_TASK_ID (set by SunGridengine). The lines are simply numbered consecutively.

See `getline_from_arraylist()` for more details.

get_value_from_arraylist(*index=0, **kwargs*)
Get field[index] of the entry in the array list corresponding to SGE_TASK_ID.

See `get_fields_from_arraylist()` for details.

`staging.Local` — staging class for running local jobs

Ersatz framework for running a staged script without actually doing any staging.

Simply replace

```
from staging.SunGridEngine import Job
```

with

```
from staging.Local import Job
```

in the python run script (see `staging.SunGridEngine` for an example script).

Description of the `Job` class

class **`Job`** (**args, **kwargs*)

Job class that doesn't do anything but provides parameters as the 'real' classes do.

job = Job(inputfiles=<dict>,outputfiles=<dict>,variables=<dict>,startdir=<PWD>)

`save` (*filename*)

Save the Job() as a pickled file.

Restore with

import staging.SunGridengine import cPickle job = cPickle.load(open(<filename>,'r'))

1.6 Alternatives to GromacsWrapper

GromacsWrapper is simplistic; in particular it does not directly link to the Gromacs libraries but relies on python wrappers to call gromacs tools. Some people find this very crude (the author included). Other people have given more thought to the problem and you are encouraged to see if their efforts speed up your work more than does *GromacsWrapper*.

pymacs (Daniel Seeliger) pymacs is a python module for dealing with structure files and trajectory data from the GROMACS molecular dynamics package. It has interfaces to some gromacs functions and uses gromacs routines for command line parsing, reading and writing of structure files (pdb,gro,...) and for reading trajectory data (only xtc at the moment). It is quite useful to write python scripts for simulation setup and analysis that can be combined with other powerful python packages like numpy, scipy or plotting libraries like pylab. It has an intuitive data structure (Model –> Chain –> Molecule –> Atom) and allows modifications at all levels like

- Changing of atom, residue and chain properties (name, coordinate, b-factor,...
- Deleting and inserting atoms, residues, chains
- Straightforward selection of structure subsets
- Structure building from sequence
- Handling gromacs index files

Gromacs XTC Library Version 1.1 of the separate xtc/trr library contains example code to access a Gromacs trajectory from python. It appears to be based on grompy (also see below).

various implementations of python wrappers See the discussion on the gmx-developers mailinglist: check the thread [gmx-developers] Python interface for Gromacs

grompy (Martin Hoefling, Roland Schulz) uses ctypes to wrap **libgmx**:

"Here's a bunch of code I wrote to wrap libgmx with ctypes and make use of parts of gromacs functionality. My application for this was the processing of a trajectories using gromac's pbc removal and fitting routines as well as reading in index groups etc. It's very incomplete atm and also focused on wrapping libgmx with all gromacs types and definitions...

... so python here feels a bit like lightweight c-code glueing together gromacs library functions :-)

> The attached code lacks a bit of documentation, but I included a test.py as an example using it."

Roland Schulz added code:

> "I added a little bit wrapper code to easily access the atom information in tpx. I attached the version. It is backward compatible ..."

A working grompy tar ball (with Roland's enhancements) is cached at gmane.org.

LOOS (Grossfield lab at the University of Rochester) The idea behind *LOOS* (*Lightweight Object-Oriented Structure* library) is to provide a lightweight C++ library for analysis of molecular dynamics simulations. This includes parsing a number of PDB variants, as well as the native system description and trajectory formats for CHARMM, NAMD, and Amber. *LOOS* is not intended to be an all-encompassing library and it is primarily geared towards reading data in and processing rather than manipulating the files and structures and writing them out.

The LOOS documentation is well written and comprehensive and the code is published under the GPL.

VMD (Schulten lab at UIUC) VMD is a great analysis tool; the only downside is that (at the moment) trajectories have to fit into memory. In some cases this can be circumvented by reading a trajectory frame by frame using the bigdcd script (which might also work for Gromacs xtcs).

MDAnalysis (N. Michaud-Agrawal, E. J. Dennning, and O. Beckstein) Reads various trajectories (dcd, xtc, trr) and makes coordinates available as numpy arrays. It also has a fairly sophisticated selection language, similar to Charmm or VMD.

Please let me know of other efforts so that I can add them here. Thanks.

CHAPTER

TWO

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

BIBLIOGRAPHY

[Allen2009] W. J. Allen, J. A. Lemkul, and D. R. Bevan. (2009) “GridMAT-MD: A Grid-based Membrane Analysis Tool for Use With Molecular Dynamics.” J. Comput. Chem. 30, 1952-1958.

[Allen2009] W. J. Allen, J. A. Lemkul, and D. R. Bevan. (2009) “GridMAT-MD: A Grid-based Membrane Analysis Tool for Use With Molecular Dynamics.” J. Comput. Chem. 30 (12): 1952-1958.

MODULE INDEX

INDEX