GromacsWrapper Documentation

Release 0.7.0

Oliver Beckstein

August 09, 2018

Contents

1 Contents **3**
- 1.1 Installation . . . 3
- 1.2 Configuration . . . 5
- 1.3 GromacsWrapper package . . . 7
- 1.4 Alternatives to GromacsWrapper . . . 77

2 Indices and tables **79**

Bibliography **81**

Python Module Index **83**

Release 0.7.0

Date August 09, 2018

GromacsWrapper is a Python package (Python 2.7.x and Python > 3.4) 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.

Gromacs versions 4.6.x, 2016.x, and 2018.x are all supported. GromacsWrapper detects your Gromacs tools and provides them as `gromacs.grompp()`, `gromacs.mdrun()`, etc, regardless of your Gromacs version, which allows one to write scripts that are broadly Gromacs-version agnostic. Source your `GMXRC` file or make the **`gmx`** binary (for versions 2016) or all the gromacs tools available on your `PATH` for GromacsWrapper to find the Gromacs installation.

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).

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 yet 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: GromacsWrapper* for contact details.

See also:

Other approaches to interfacing Python and Gromacs are listed under *Alternatives to GromacsWrapper*.

CHAPTER 1

Contents

1.1 Installation

The package contains two files, README and INSTALL, which should get you started quickly.

If everything works perfectly then you might be able to install a working version of *GromacsWrapper* with a simple

```
python setup.py install
```

from the unpacked sources.

1.1.1 README: GromacsWrapper

A primitive wrapper around the Gromacs tools until we have proper python bindings. It also provides a small library (cook book) of often-used recipes and an optional analysis module with plugins for more complicated analysis tasks.

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.

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.

Licence

Citing

GromacsWrapper was written by Oliver Beckstein with contributions from many other people. Please see the file AUTHORS for all the names.

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 et al, https://github.com/Becksteinlab/GromacsWrapper doi: 10.5281/zenodo.17901)"

or in the Acknowledgements section.

Thank you.

Download and Availability

The GromacsWrapper home page is https://github.com/Becksteinlab/GromacsWrapper. The latest release of the package is being made available from https://github.com/Becksteinlab/GromacsWrapper/releases

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

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

Contact

Please use the Issue Tracker to report bugs, installation problems, and feature requests (mention `@orbeckst` in the issue report).

1.1.2 INSTALL

This document should help you to install the **GromacsWrapper** package. Please raise and issue in the Issue Tracker if problems occur or if you have suggestions on how to improve the package or these instructions.

Quick installation instructions

The latest release can be directly installed from the internet:

```
pip install GromacsWrapper
```

This will automatically download and install the latest version of GromacsWrapper from PyPi.

Manual Download

If your prefer to download manually, get the latest stable release from https://github.com/Becksteinlab/GromacsWrapper/releases and either

```
pip install GromacsWrapper-0.7.0.tar.gz
```

or install from the unpacked source:

```
tar -zxvf GromacsWrapper-0.7.0.tar.gz
cd GromacsWrapper-0.7.0
python setup.py install
```

Source code access

The tar archive from https://github.com/Becksteinlab/GromacsWrapper/releases contains a full source code distribution.

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

git://github.com/Becksteinlab/GromacsWrapper.git

and checkout the *master& branch:

```
git clone https://github.com/Becksteinlab/GromacsWrapper.git
cd GromacsWrapper
```

Requirements

Python 2.7.x or Python >= 3.4 and Gromacs (4.6.x, 2016, 2018) 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 2.7.x and Python 3.5/3.6 on Linux and Mac OS X. Earlier Python versions are not supported.

Note: Python 3 support is currently in alpha state; in principle it is fully supported but if you find bugs please report them through the Issue Tracker.

Required Python modules

The basic package makes use of numpy and numkit (which uses scipy); all dependencies are installed during a normal installation process.

1.2 Configuration

This section documents how to configure the **GromacsWrapper** package. There are options to configure where log files and templates directories are located and options to tell exactly which commands to load into this package. Any configuration is optional and all options has sane defaults. Further documentation can be found at *gromacs.config*.

Changed in version 0.6.0: The format of the `tools` variable in the `[Gromacs]` section of the config file was changed for Gromacs 5 commands.

1.2.1 Basic options

Place an INI file named `~/.gromacswrapper.cfg` in your home directory, it may look like the following document:

```
[Gromacs]
GMXRC = /usr/local/gromacs/bin/GMXRC
```

The Gromacs software suite needs some environment variables that are set up sourcing the `GMXRC` file. You may source it yourself or set an option like the above one. If this option isn't provided, **GromacsWrapper** will guess that Gromacs was globally installed like if it was installed by the `apt-get` program.

As there isn't yet any way to know which Gromacs version to use, **GromacsWrapper** will first try to use Gromacs 5 if available, then to use Gromacs 4. If you have both versions and want to use version 4 or just want to document it, you may specify the which release version will be used:

```
[Gromacs]
GMXRC = /usr/local/gromacs/bin/GMXRC
release = 4.6.7
```

For now **GromacsWrapper** will guess which tools are available to put it into *gromacs.tools*, but you can always configure it manually. Gromacs 5 has up to 4 commands usually named:

```
[Gromacs]
tools = gmx gmx_d gmx_mpi gmx_mpi_d
```

This option will instruct which commands to load. For Gromacs 4 you'll need to specify more tools:

```
[Gromacs]
GMXRC = /usr/local/gromacs/bin/GMXRC
release = 4
tools =
                g_cluster     g_dyndom       g_mdmat      g_principal  g_select    g_
↪wham    mdrun
     do_dssp    g_clustsize   g_enemat       g_membed     g_protonate  g_sgangle   g_
↪wheel   mdrun_d
     editconf   g_confrms     g_energy       g_mindist    g_rama       g_sham      g_
↪x2top   mk_angndx
     eneconv    g_covar       g_filter       g_morph      g_rdf        g_sigeps    ␣
↪genbox    pdb2gmx
     g_anadock  g_current     g_gyrate       g_msd                     g_sorient   ␣
↪genconf
     g_anaeig   g_density     g_h2order      g_nmeig      g_rms        g_spatial   ␣
↪genion    tpbconv
     g_analyze  g_densmap     g_hbond        g_nmens      g_rmsdist    g_spol      ␣
↪genrestr  trjcat
     g_angle    g_dielectric  g_helix        g_nmtraj     g_rmsf       g_tcaf      ␣
↪gmxcheck  trjconv
     g_bar      g_dih         g_helixorient  g_order      g_rotacf     g_traj      ␣
↪gmxdump   trjorder
     g_bond     g_dipoles     g_kinetics     g_pme_error  g_rotmat     g_tune_pme  ␣
↪grompp
     g_bundle   g_disre       g_lie          g_polystat   g_saltbr     g_vanhove   ␣
↪make_edi  xpm2ps
     g_chi      g_dist        g_luck         g_potential  g_sas        g_velacc    ␣
↪make_ndx
```

Commands will be available directly from the *gromacs*:

```
import gromacs
gromacs.mdrun_d # either v5 `gmx_d mdrun` or v4 `mdrun_d`
gromacs.mdrun   # either v5 `gmx mdrun`   or v4 `mdrun`
```

1.2.2 More options

Other options are to set where template for job submission systems and.mdp files are located:

```
[DEFAULT]
# Directory to store user templates and rc files.
configdir = ~/.gromacswrapper

# Directory to store user supplied queuing system scripts.
qscriptdir = %(configdir)s/qscripts

# Directory to store user supplied template files such as mdp files.
templatesdir = %(configdir)s/templates
```

And there are yet options for how to handle logging:

```
[Logging]
# name of the logfile that is written to the current directory
logfilename = gromacs.log

# loglevels (see Python's logging module for details)
#   ERROR   only fatal errors
#   WARN    only warnings
#   INFO    interesting messages
#   DEBUG   everything

# console messages written to screen
loglevel_console = INFO

# file messages written to logfilename
loglevel_file = DEBUG
```

If needed you may set up basic configuration files and directories using *gromacs.config.setup()*:

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

1.3 GromacsWrapper 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.fileformats 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.

Examples

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

Getting help

In python:

```
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 (*GromacsFailureWarning*, *GromacsImportWarning*, *GromacsValueWarning*, *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 gromacs.**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 gromacs.**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 gromacs.**ParseError**
: Error raised when parsing of a file failed.

The following *warnings* are defined:

exception gromacs.**GromacsFailureWarning**
: Warning about failure of a Gromacs tool.

exception gromacs.**GromacsImportWarning**
: Warns about problems with using a gromacs tool.

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

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

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

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

exception gromacs.**UsageWarning**
Warns if usage is unexpected/documentation ambiguous.

exception gromacs.**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.

It is also possible to capture output from Gromacs commands in a file instead of displaying it on screen, as described under *Input and Output*.

Normally, one starts logging with the `start_logging()` function but in order to obtain logging messages (typically at level *debug*) right from the start one may set the environment variable `GW_START_LOGGING` to any value that evaluates to `True` (e.g., "True" or "1").

Version

The package version is recorded in the `gromacs.__version__` variable.

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. The documentation of *GromacsCommand* applies to all wrapped Gromacs commands and should be read by anyone using this package.

Input and Output

Each command wrapped by either *GromacsCommand* or *Command* takes three additional keyword arguments: *stdout*, *stderr*, and *input*. *stdout* and *stderr* determine how the command returns its own output.

The *input* keyword is a string that is fed to the standard input of the command (actually, `subprocess.Popen.stdin`). Or, if it is not string-like then we assume it's actually a file-like object that we can read from, e.g. a `subprocess.Popen.stdout` or a `File`.

By setting the *stdout* and *stderr* keywords appropriately, one can have the output simply printed to the screen (use `True`; this is the default, although see below for the use of the `capture_output` *gromacs.environment* flag), capture in a python variable as a string for further processing (use `False`), write to a file (use a `File` instance) or as input for another command (e.g. use the `subprocess.Popen.stdin`).

When writing setup- and analysis pipelines it can be rather cumbersome to have the gromacs output on the screen. For these cases GromacsWrapper allows you to change its behaviour globally. By setting the value of the *gromacs.environment Flag* `capture_output` to `True` (in the GromacsWrapper *gromacs.environment.flags* registry)

```
import gromacs.environment
gromacs.environment.flags['capture_output'] = True
```

all commands will capture their output (like *stderr* = `False` and *stdout* = `False`). Explicitly setting these keywords overrides the global default. The default value for `flags['capture_output']` is `False`, i.e. output is directed through STDOUT and STDERR.

Warning: One downside of `flags['capture_output'] = True` is that it becomes much harder to debug scripts unless the script is written in such a way to show the output when the command fails. Therefore, it is advisable to only capture output on well-tested scripts.

A third value of `capture_output` is the value `"file"`:

```
gromacs.environment.flags['capture_output'] = "file"
```

This writes the captured output to a file. The file name is specified in `flags['capture_output_filename']` and defaults to *"gromacs_captured_output.txt"*. This file is *over-written* for each command. In this way one can investigate the output from the last command (presumably because it failed). STDOUT and STDERR are captured into this file by default. STDERR is printed first and then STDOUT, which does not necessarily reflect the order of output one would see on the screen. If your code captures STDOUT for further processing then an uncaptured STDERR is written to the capture file.

Note: There are some commands for which capturing output (`flags['capture_output'] = True`) might be problematic. If the command produces a large or inifinite amount of data then a memory error will occur because Python nevertheless stores the output internally first. Thus one should avoid capturing progress output from e.g. `Mdrun` unless the output has been throttled appropriately.

Classes

class gromacs.core.**GromacsCommand**(**args*, ***kwargs*)

Base class for wrapping a Gromacs tool.

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 the first time the doc string is requested. If the tool is not available at the time (i.e., cannot be found on **:env:'PATH'**) then the generic doc string is shown and an `OSError` exception is only raised when the user is actually trying to the execute the command.

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*, i.e. they determine how this tool behaves during all future invocations although it can be changed by setting *`failuremode`*. 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 (*ignored*) []

Changed in version 0.6.0: The *doc* keyword is now ignored (because it was not worth the effort to make it work with the lazy-loading of docs).

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).

failuremode
mode determines how the GromacsCommand behaves during failure

It can be one of the following:

'raise' raises GromacsError if command fails

'warn' issue a `GromacsFailureWarning`

None just continue silently

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 gromacs.core.**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.

Example

Create a `Ls` class whose instances execute the `ls` command:

```
LS = type("LS", (gromacs.core.Command,), {'command_name': 'ls'})
ls = LS()
ls()         # lists directory like ls
ls(l=True)   # lists directory like ls -l
```

Now create an instance that performs a long directory listing by default:

```
lslong = LS(l=True)
lslong()     # like ls -l
```

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.

__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 [None]

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)

Depending on the value of the GromacsWrapper flag gromacs.environment.flags```['capture_output']` the above default behaviour can be different.

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 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)

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*)
Transform arguments and return them as a list suitable for Popen.

class gromacs.core.**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; It serves to define how to handle log files, set where template files are located and which gromacs tools are exposed in the *gromacs* package.

In order to set up a basic configuration file and the directories a user can execute *gromacs.config.setup()*.

If the configuration file is edited then one can force a rereading of the new config file with *gromacs.config.get_configuration()*:

```
gromacs.config.get_configuration()
```

However, this will not update the available command classes (e.g. when new executables were added to a tool group). In this case one either has to `reload()` a number of modules (*gromacs*, *gromacs.config*, *gromacs.tools*) although it is by far easier simply to quit python and freshly `import gromacs`.

Almost all aspects of *GromacsWrapper* (paths, names, what is loaded) can be changed from within the configuration file. The only exception is the name of the configuration file itself: This is hard-coded as `~/.gromacswrapper.cfg` although it is possible to read other configuration files with the *filename* argument to *get_configuration()*.

Configuration management

Important configuration variables are

gromacs.config.**configdir = '/home/docs/.gromacswrapper'**
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

gromacs.config.**path = ['.', '/home/docs/.gromacswrapper/qscripts', '/home/docs/.gromacswrap**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

When GromacsWrapper starts up it runs *check_setup()*. This notifies the user if any config files or directories are missing and suggests to run *setup()*. The check if the default set up exists can be suppressed by setting the environment variable GROMACSWRAPPER_SUPPRESS_SETUP_CHECK to 'true' ('yes' and '1' also work).

Users

Users will likely only need to run *gromacs.config.setup()* once and perhaps occasionally execute *gromacs.config.get_configuration()*. Mainly the user is expected to configure *GromacsWrapper* by editing the configuration file ~/.gromacswrapper.cfg (which has ini-file syntax as described in ConfigParser).

gromacs.config.**setup**(*filename='/home/docs/.gromacswrapper.cfg'*)
Prepare a default GromacsWrapper global environment.

1. Create the global config file.
2. Create the directories in which the user can store template and config files.

This function can be run repeatedly without harm.

gromacs.config.**get_configuration**(*filename='/home/docs/.gromacswrapper.cfg'*)
Reads and parses the configuration file.

Default values are loaded and then replaced with the values from ~/.gromacswrapper.cfg if that file exists. The global configuration instance gromacswrapper.config.cfg is updated as are a number of global variables such as *configdir*, *qscriptdir*, *templatesdir*, *logfilename*, ...

Normally, the configuration is only loaded when the *gromacs* package is imported but a re-reading of the configuration can be forced anytime by calling *get_configuration()*.

Returns a dict with all updated global configuration variables

gromacs.config.**check_setup**()
Check if templates directories are setup and issue a warning and help.

Set the environment variable GROMACSWRAPPER_SUPPRESS_SETUP_CHECK skip the check and make it always return True

:return True if directories were found and False otherwise

Changed in version 0.3.1: Uses GROMACSWRAPPER_SUPPRESS_SETUP_CHECK to suppress check (useful for scripts run on a server)

Developers

Developers are able to access all configuration data through *gromacs.config.cfg*, which represents the merger of the package default values and the user configuration file values.

gromacs.config.**cfg = <gromacs.config.GMXConfigParser instance>**
Customized ConfigParser.SafeConfigParser.

class gromacs.config.**GMXConfigParser**(**args*, ***kwargs*)
Customized ConfigParser.SafeConfigParser.

Reads and parses the configuration file.

Default values are loaded and then replaced with the values from ~/.gromacswrapper.cfg if that file exists. The global configuration instance gromacswrapper.config.cfg is updated as are a number of global variables such as *configdir*, *qscriptdir*, *templatesdir*, *logfilename*, ...

Normally, the configuration is only loaded when the gromacswrapper package is imported but a re-reading of the configuration can be forced anytime by calling *get_configuration()*.

configuration
Dict of variables that we make available as globals in the module.

Can be used as

```
globals().update(GMXConfigParser.configuration)        # update configdir,␣
↪templatesdir ...
```

getLogLevel(*section*, *option*)
Return the textual representation of logging level 'option' or the number.

Note that option is always interpreted as an UPPERCASE string and hence integer log levels will not be recognized.

getpath(*section*, *option*)
Return option as an expanded path.

A subset of important data is also made available as top-level package variables as described under *Location of template files* (for historical reasons); the same variable are also available in the dict *gromacs.config.configuration*.

gromacs.config.**configuration = {'configdir': '/home/docs/.gromacswrapper', 'configfilename**
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

Default values are hard-coded in

gromacs.config.**CONFIGNAME = '/home/docs/.gromacswrapper.cfg'**
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

gromacs.config.**defaults = {'configdir': '/home/docs/.gromacswrapper', 'logfilename': 'gro**
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

Accessing configuration and template files

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.

gromacs.config.**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.

gromacs.config.**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.

Logging

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

gromacs.config.**logfilename = 'gromacs.log'**
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

gromacs.config.**loglevel_console = 20**
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

gromacs.config.**loglevel_file = 10**
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

Gromacs tools and scripts

Fundamentally, GromacsWrapper makes existing Gromacs tools (executables) available as functions. In order for this to work, these executables must be found in the environment of the Python process that runs GromacsWrapper, and the user must list all the tools that are to be made available.

Setting up the environment

The standard way to set up the Gromacs environment is to source `GMXRC` in the shell before running the Python process. `GMXRC` adjusts a number of environment variables (such as `PATH` and `LD_LIBRARY_PATH`) but also sets Gromacs-specific environment variables such as `GMXBIN`, `GMXDATA`, and many others:

```
source /usr/local/bin/GMXRC
```

(where the path to `GMXRC` is often set differently to disntinguish different installed versions of Gromacs).

Alternatively, GromacsWrapper can itself source a `GMXRC` file and set the environment with the *`set_gmxrc_environment()`* function. The path to a `GMXRC` file can be set in the config file in the `[Gromacs]` section as

[Gromacs]

GMXRC = /usr/local/bin/GMXRC

When GromacsWrapper starts up, it tries to set the environment using the `GMXRC` defined in the config file. If this is left empty or is not in the file, nothing is being done.

gromacs.config.**set_gmxrc_environment**(*gmxrc*)
Set the environment from `GMXRC` provided in *gmxrc*.

Runs `GMXRC` in a subprocess and puts environment variables loaded by it into this Python environment.

If *gmxrc* evaluates to `False` then nothing is done. If errors occur then only a warning will be logged. Thus, it should be safe to just call this function.

List of tools

The list of Gromacs tools can be specified in the config file in the `[Gromacs]` section with the `tools` variable.

The tool groups are a list of names that determines which tools are made available as classes in *gromacs.tools*. If not provided GromacsWrapper will first try to load Gromacs 5.x then Gromacs 4.x tools.

If you choose to provide a list, the Gromacs tools section of the config file can be like this:

```
[Gromacs]
# Release of the Gromacs package to which information in this sections applies.
release = 4.5.3

# 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.
# (Note that this example has a much shorter list than the actual default.)
tools =
      editconf make_ndx grompp genion genbox
      grompp pdb2gmx mdrun mdrun_d

# which tool groups to make available
groups = tools extra
```

For Gromacs 5.x use a section like the following, where driver commands are supplied:

```
[Gromacs]
# Release of the Gromacs package to which information in this sections applies.
release = 5.0.5

# GMXRC contains the path for GMXRC file which will be loaded. If not
provided is expected that it was sourced as usual before importing this
library.
GMXRC = /usr/local/gromacs/bin/GMXRC

# tools contains the command names of all Gromacs tools for which classes are␣
↪generated.
# Editing this list has only an effect when the package is reloaded.
# (Note that this example has a much shorter list than the actual default.)
tools = gmx gmx_d
```

For example, on the commandline you would run

```
gmx grompp -f md.mdp -c system.gro -p topol.top -o md.tpr
```

and within GromacsWrapper this would become

```
gromacs.grompp(f="md.mdp", c="system.gro", p="topol.top", o="md.tpr")
```

Note: Because of changes in the Gromacs tool in 5.x, GromacsWrapper scripts might break, even if the tool names are still the same.

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.

gromacs.config.**qscriptdir = '/home/docs/.gromacswrapper/qscripts'**
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

gromacs.config.**templatesdir = '/home/docs/.gromacswrapper/templates'**
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

gromacs.config.**templates = {'darwin.sh': '/home/docs/.cache/Python-Eggs/GromacsWrapper-0.**
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

gromacs.config.**qscript_template = '/home/docs/.cache/Python-Eggs/GromacsWrapper-0.7.0-py2.**
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

gromacs.environment – Run time modification of behaviour

Some aspects of GromacsWrapper can be determined globally. The corresponding flags *Flag* are set in the environment (think of them like environment variables). They are accessible through the pseudo-dictionary *gromacs.environment.flags*.

The entries appear as 'name'-'value' pairs. Flags check values and illegal ones raise a ValueError. Documentation on all flags can be obtained with

```
print gromacs.environment.flags.doc()
```

List of GromacsWrapper flags with default values

class gromacs.environment.**flagsDocs**
capture_output = *False*

Select if Gromacs command output is *always* captured.

```
>>> flags['capture_output'] = False
```

By default a *GromacsCommand* will direct STDOUT and STDERR output from the command itself to the screen (through /dev/stdout and /dev/stderr). When running the command, this can be changed with the keywords *stdout* and *stderr* as described in *gromacs.core* and *Command*.

If this flag is set to True then by default STDOUT and STDERR are captured as if one had set

```
stdout=False, stderr=False
```

Explicitly setting *stdout* and/or *stderr* overrides the behaviour described above.

If set to the special keyword `"file"` then the command writes to the file whose name is given by ``flags['capture_output_filename']``. This file is *over-written* for each command. In this way one can investigate the output from the last command (presumably because it failed). STDOUT and STDERR are captured into this file by default. STDERR is printed first and then STDOUT, which does not necessarily reflect the order of output one would see on the screen.

The default is False.

capture_output_filename = *'gromacs_captured_output.txt'*

Name of the file that captures output if ''flags['capture_output'] = "file"

```
>>> flags['capture_output_filename'] = 'gromacs_captured_output.txt'
```

This is an *experimental* feature. The default is 'gromacs_captured_output.txt'.

Classes

gromacs.environment.**flags**

class gromacs.environment.**Flags**(**args*)

Global registry of flags. Acts like a dict for item access.

There are a number flags defined that influence how MDAnalysis behaves. They are accessible through the pseudo-dictionary

gromacs.environment.flags

The entries appear as 'name'-'value' pairs. Flags check values and illegal ones raise a `ValueError`. Documentation on all flags can be obtained with

```
print gromacs.environment.flags.__doc__
```

New flags are added with the *Flags.register()* method which takes a new *Flag* instance as an argument.

For **developers**: Initialize Flags registry with a *list* of *Flag* instances.

doc()

Shows doc strings for all flags.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

itervalues() → an iterator over the values of D

register(*flag*)

Register a new *Flag* instance with the Flags registry.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update(**flags*)

Update Flags registry with a list of *Flag* instances.

values() → list of D's values

class gromacs.environment.**Flag**(*name*, *default*, *mapping=None*, *doc=None*)
A Flag, essentially a variable that knows its default and legal values.

Create a new flag which will be registered with FLags.

newflag = Flag(name,default,mapping,doc)

Arguments

name name of the flag, must be a legal python name

default default value

mapping dict that maps allowed input values to canonical values; if `None` then no argument checking will be performed and all values are directly set.

doc doc string; may contain string interpolation mappings for:

```
%%(name)s        name of the flag
%%(default)r     default value
%%(value)r       current value
%%(mapping)r     mapping
```

Doc strings are generated dynamically and reflect the current state.

prop()
Use this for property(**flag.prop())

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.

Contents

Simple xmgrace XVG file format

Gromacs produces graphs in the xmgrace ("xvg") format. These are simple multi-column data files. The class *XVG* encapsulates access to such files and adds a number of methods to access the data (as NumPy arrays), compute aggregates, or quickly plot it.

The *XVG* class is useful beyond reading xvg files. With the *array* keyword or the *XVG.set()* method one can load data from an array instead of a file. The array should be simple "NXY" data (typically: first column time or position, further columns scalar observables). The data should be a NumPy `numpy.ndarray` array `a` with `shape (M, N)` where *M*-1 is the number of observables and *N* the number of observations, e.g.the number of time points in a time series. `a[0]` is the time or position and `a[1:]` the *M*-1 data columns.

Errors

The *XVG.error* attribute contains the statistical error for each timeseries. It is computed from the standard deviation of the fluctuations from the mean and their correlation time. The parameters for the calculations of the correlation time are set with *XVG.set_correlparameters()*.

See also:

`numkit.timeseries.tcorrel()`

Plotting

The *XVG.plot()* and *XVG.errorbar()* methods are set up to produce graphs of multiple columns simultaneously. It is typically assumed that the first column in the selected (sub)array contains the abscissa ("x-axis") of the graph and all further columns are plotted against the first one.

Data selection

Plotting from *XVG* is fairly flexible as one can always pass the *columns* keyword to select which columns are to be plotted. Assuming that the data contains `[t, X1, X2, X3]`, then one can

1. plot all observable columns (X1 to X3) against t:

   ```
   xvg.plot()
   ```

2. plot only X2 against t:

   ```
   xvg.plot(columns=[0,2])
   ```

3. plot X2 and X3 against t:

   ```
   xvg.plot(columns=[0,2,3])
   ```

4. plot X1 against X3:

   ```
   xvg.plot(columns=[2,3])
   ```

Coarse grainining of data

It is also possible to *coarse grain the data* for plotting (which typically results in visually smoothing the graph because noise is averaged out).

Currently, two alternative algorithms to produce "coarse grained" (decimated) graphs are implemented and can be selected with the *method* keyword for the plotting functions in conjuction with *maxpoints* (the number of points to be plotted):

1. **mean** histogram (default) — bin the data (using `numkit.timeseries.regularized_function()` and compute the mean for each bin. Gives the exact number of desired points but the time data are whatever the middle of the bin is.
2. **smooth** subsampled — smooth the data with a running average (other windows like Hamming are also possible) and then pick data points at a stepsize compatible with the number of data points required. Will give exact times but not the exact number of data points.

For simple test data, both approaches give very similar output.

For the special case of periodic data such as angles, one can use the circular mean ("circmean") to coarse grain. In this case, jumps across the -180º/+180º boundary are added as masked datapoints and no line is drawn across the jump in the plot. (Only works with the simple *XVG.plot()* method at the moment, errorbars or range plots are not implemented yet.)

See also:

XVG.decimate()

Examples

In this example we generate a noisy time series of a sine wave. We store the time, the value, and an error. (In a real example, the value might be the mean over multiple observations and the error might be the estimated error of the mean.)

```
>>> import numpy as np
>>> import gromacs.formats
>>> X = np.linspace(-10,10,50000)
>>> yerr = np.random.randn(len(X))*0.05
>>> data = np.vstack((X, np.sin(X) + yerr, np.random.randn(len(X))*0.05))
>>> xvg = gromacs.formats.XVG(array=data)
```

Plot value for *all* time points:

```
>>> xvg.plot(columns=[0,1], maxpoints=None, color="black", alpha=0.2)
```

Plot bin-averaged (decimated) data with the errors, over 1000 points:

```
>>> xvg.errorbar(maxpoints=1000, color="red")
```

(see output in Figure *Plot of Raw vs Decimated data*)

In principle it is possible to use other functions to decimate the data. For *XVG.plot()*, the *method* keyword can be changed (see *XVG.decimate()* for allowed *method* values). For *XVG.errorbar()*, the method to reduce the data values (typically column 1) is fixed to "mean" but the errors (typically columns 2 and 3) can also be reduced with *error_method* = "rms".

If one wants to show the variation of the raw data together with the decimated and smoothed data then one can plot the percentiles of the deviation from the mean in each bin:

```
>>> xvg.errorbar(columns=[0,
↪1,1], maxpoints=1000, color="blue", demean=True)
```

The *demean* keyword indicates if fluctuations from the mean are regularised[1]. The method *XVG.plot_coarsened()* automates this approach and can plot coarsened data selected by the *columns* keyword.

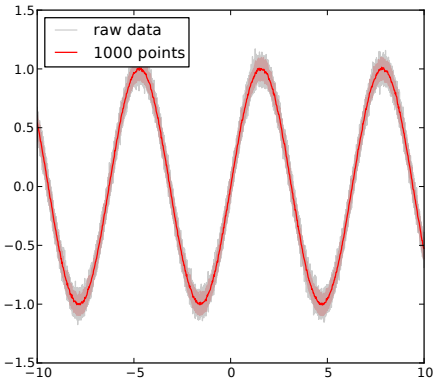

Fig. 1: **Plot of Raw vs Decimated data.** Example of plotting raw data (sine on 50,000 points, gray) versus the decimated graph (reduced to 1000 points, red line). The errors were also decimated and reduced to the errors within the 5% and the 95% percentile. The decimation is carried out by histogramming the data in the desired number of bins and then the data in each bin is reduced by either *numpy.mean()* (for the value) or *scipy.stats.scoreatpercentile()* (for errors).

[1] When *error_method* = "percentile" is selected for *XVG.errorbar()* then *demean* does not actually force a regularisation of the fluctuations from the mean. Instead, the (symmetric) percentiles are computed on the full data and the error ranges for plotting are directly set to the percentiles. In this way one can easily plot the e.g. 10th-percentile to 90th-percentile band (using keyword *percentile* = 10).

Classes and functions

class gromacs.fileformats.xvg.**XVG**(*filename=None*, *names=None*, *array=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

array read data from *array* (see `XVG.set()`)

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`]

stride Only read every *stride* line of data [1].

savedata `True` includes the data (`XVG.array` and associated caches) when the instance is pickled (see `pickle`); this is oftens not desirable because the data are already on disk (the xvg file *filename*) and the resulting pickle file can become very big. `False` omits those data from a pickle. [`False`]

metadata dictionary of metadata, which is not touched by the class

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, ...

decimate(*method*, *a*, *maxpoints=10000*, ***kwargs*)

Decimate data *a* to *maxpoints* using *method*.

If *a* is a 1D array then it is promoted to a (2, N) array where the first column simply contains the index.

If the array contains fewer than *maxpoints* points or if *maxpoints* is `None` then it is returned as it is. The default for *maxpoints* is 10000.

Valid values for the reduction *method*:

- "mean", uses `XVG.decimate_mean()` to coarse grain by averaging the data in bins along the time axis
- "circmean", uses `XVG.decimate_circmean()` to coarse grain by calculating the circular mean of the data in bins along the time axis. Use additional keywords *low* and *high* to set the limits. Assumes that the data are in degrees.
- "min" and "max* select the extremum in each bin
- "rms", uses `XVG.decimate_rms()` to coarse grain by computing the root mean square sum of the data in bins along the time axis (for averaging standard deviations and errors)
- "percentile" with keyword *per*: `XVG.decimate_percentile()` reduces data in each bin to the percentile *per*
- "smooth", uses `XVG.decimate_smooth()` to subsample from a smoothed function (generated with a running average of the coarse graining step size derived from the original number of data points and *maxpoints*)

Returns numpy array `(M', N')` from a `(M', N)` array with `M' == M` (except when `M == 1`, see above) and `N' <= N` (`N'` is *maxpoints*).

decimate_circmean(*a*, *maxpoints*, ***kwargs*)

Return data *a* circmean-decimated on *maxpoints*.

Histograms each column into *maxpoints* bins and calculates the weighted circular mean in each bin as the decimated data, using `numkit.timeseries.circmean_histogrammed_function()`. The coarse grained time in the first column contains the centers of the histogram time.

If *a* contains <= *maxpoints* then *a* is simply returned; otherwise a new array of the same dimensions but with a reduced number of *maxpoints* points is returned.

Keywords *low* and *high* can be used to set the boundaries. By default they are [-pi, +pi].

This method returns a **masked** array where jumps are flagged by an insertion of a masked point.

Note: Assumes that the first column is time and that the data are in **degrees**.

Warning: Breaking of arrays only works properly with a two-column array because breaks are only inserted in the x-column (a[0]) where y1 = a[1] has a break.

decimate_error(*a*, *maxpoints*, ***kwargs*)

Return data *a* error-decimated on *maxpoints*.

Histograms each column into *maxpoints* bins and calculates an error estimate in each bin as the decimated data, using `numkit.timeseries.error_histogrammed_function()`. The coarse grained time in the first column contains the centers of the histogram time.

If *a* contains <= *maxpoints* then *a* is simply returned; otherwise a new array of the same dimensions but with a reduced number of *maxpoints* points is returned.

See also:

`numkit.timeseries.tcorrel()`

Note: Assumes that the first column is time.

Does not work very well because often there are too few datapoints to compute a good autocorrelation function.

---

decimate_max(*a*, *maxpoints*, ***kwargs*)

Return data *a* max-decimated on *maxpoints*.

Histograms each column into *maxpoints* bins and calculates the maximum in each bin as the decimated data, using `numkit.timeseries.max_histogrammed_function()`. The coarse grained time in the first column contains the centers of the histogram time.

If *a* contains <= *maxpoints* then *a* is simply returned; otherwise a new array of the same dimensions but with a reduced number of *maxpoints* points is returned.

Note: Assumes that the first column is time.

---

decimate_mean(*a*, *maxpoints*, ***kwargs*)

Return data *a* mean-decimated on *maxpoints*.

Histograms each column into *maxpoints* bins and calculates the weighted average in each bin as the decimated data, using `numkit.timeseries.mean_histogrammed_function()`. The coarse grained time in the first column contains the centers of the histogram time.

If *a* contains <= *maxpoints* then *a* is simply returned; otherwise a new array of the same dimensions but with a reduced number of *maxpoints* points is returned.

Note: Assumes that the first column is time.

decimate_min(*a*, *maxpoints*, ***kwargs*)
Return data *a* min-decimated on *maxpoints*.

Histograms each column into *maxpoints* bins and calculates the minimum in each bin as the decimated data, using `numkit.timeseries.min_histogrammed_function()`. The coarse grained time in the first column contains the centers of the histogram time.

If *a* contains <= *maxpoints* then *a* is simply returned; otherwise a new array of the same dimensions but with a reduced number of *maxpoints* points is returned.

Note: Assumes that the first column is time.

decimate_percentile(*a*, *maxpoints*, ***kwargs*)
Return data *a* percentile-decimated on *maxpoints*.

Histograms each column into *maxpoints* bins and calculates the percentile *per* in each bin as the decimated data, using `numkit.timeseries.percentile_histogrammed_function()`. The coarse grained time in the first column contains the centers of the histogram time.

If *a* contains <= *maxpoints* then *a* is simply returned; otherwise a new array of the same dimensions but with a reduced number of *maxpoints* points is returned.

Note: Assumes that the first column is time.

Keywords

per percentile as a percentage, e.g. 75 is the value that splits the data into the lower 75% and upper 25%; 50 is the median [50.0]

See also:

`numkit.timeseries.regularized_function()` with `scipy.stats.scoreatpercentile()`

decimate_rms(*a*, *maxpoints*, ***kwargs*)
Return data *a* rms-decimated on *maxpoints*.

Histograms each column into *maxpoints* bins and calculates the root mean square sum in each bin as the decimated data, using `numkit.timeseries.rms_histogrammed_function()`. The coarse grained time in the first column contains the centers of the histogram time.

If *a* contains <= *maxpoints* then *a* is simply returned; otherwise a new array of the same dimensions but with a reduced number of *maxpoints* points is returned.

Note: Assumes that the first column is time.

decimate_smooth(*a*, *maxpoints*, *window='flat'*)
Return smoothed data *a* decimated on approximately *maxpoints* points.

1. Produces a smoothed graph using the smoothing window *window*; "flat" is a running average.
2. select points at a step size approximatelt producing maxpoints

If *a* contains <= *maxpoints* then *a* is simply returned; otherwise a new array of the same dimensions but with a reduced number of points (close to *maxpoints*) is returned.

Note: Assumes that the first column is time (which will *never* be smoothed/averaged), except when the input array *a* is 1D and therefore to be assumed to be data at equidistance timepoints.

TODO: - Allow treating the 1st column as data

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

See *[FrenkelSmit2002]* 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*)
errorbar plot for a single time series with errors.

Set *columns* keyword to select [x, y, dy] or [x, y, dx, dy], e.g. `columns=[0,1,2]`. See *XVG.plot()* for details. Only a single timeseries can be plotted and the user needs to select the appropriate columns with the *columns* keyword.

By default, the data are decimated (see *XVG.plot()*) for the default of *maxpoints* = 10000 by averaging data in *maxpoints* bins.

x,y,dx,dy data can plotted with error bars in the x- and y-dimension (use *filled* = `False`).

For x,y,dy use *filled* = `True` to fill the region between y±dy. *fill_alpha* determines the transparency of the fill color. *filled* = `False` will draw lines for the error bars. Additional keywords are passed to `pylab.errorbar()`.

By default, the errors are decimated by plotting the 5% and 95% percentile of the data in each bin. The percentile can be changed with the *percentile* keyword; e.g. *percentile* = 1 will plot the 1% and 99% perentile (as will *percentile* = 99).

The *error_method* keyword can be used to compute errors as the root mean square sum (*error_method* = "rms") across each bin instead of percentiles ("percentile"). The value of the keyword *demean* is applied to the decimation of error data alone.

See also:

XVG.plot() lists keywords common to both methods.

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(*stride=None*)
Read and cache the file as a numpy array.

Store every *stride* line of data; if `None` then the class default is used.

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 decimate the data at regular intervals. [10000]

method method to decimate the data to *maxpoints*, see *XVG.decimate()* for details

color single color (used for all plots); sequence of colors (will be repeated as necessary); or a matplotlib colormap (e.g. "jet", see `matplotlib.cm`). The default is to use the `XVG.default_color_cycle`.

ax plot into given axes or create new one if `None` [None]

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

Returns

ax axes instance

plot_coarsened(***kwargs*)
Plot data like *XVG.plot()* with the range of **all** data shown.

Data are reduced to *maxpoints* (good results are obtained with low values such as 100) and the actual range of observed data is plotted as a translucent error band around the mean.

Each column in *columns* (except the abscissa, i.e. the first column) is decimated (with *XVG.decimate()*) and the range of data is plotted alongside the mean using *XVG.errorbar()* (see for arguments). Additional arguments:

Kewords

maxpoints number of points (bins) to coarsen over

color single color (used for all plots); sequence of colors (will be repeated as necessary); or a matplotlib colormap (e.g. "jet", see `matplotlib.cm`). The default is to use the `XVG.default_color_cycle`.

method Method to coarsen the data. See *XVG.decimate()*

The *demean* keyword has no effect as it is required to be `True`.

See also:

XVG.plot(), *XVG.errorbar()* and *XVG.decimate()*

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 with correlation functions.

The parameters persist until explicitly changed.

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 *ncorrel*)

ncorrel If no *nstep* is supplied, aim at using *ncorrel* data points for the FFT; sets `XVG.ncorrel` [25000]

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.

---

gromacs.fileformats.xvg.**break_array**(*a*, *threshold=3.141592653589793*, *other=None*)
Create a array which masks jumps >= threshold.

Extra points are inserted between two subsequent values whose absolute difference differs by more than threshold (default is pi).

Other can be a secondary array which is also masked according to *a*.

Returns (*a_masked*, *other_masked*) (where *other_masked* can be `None`)

Gromacs XPM file format

Gromacs stores matrix data in the xpm file format. This implementation of a Python reader is based on Tsjerk Wassenaar's post to gmx-users numerical matrix from xpm file (Mon Oct 4 13:05:26 CEST 2010). This version returns a NumPy array and can guess an appropriate dtype for the array.

Classes

class gromacs.fileformats.xpm.**XPM**(*filename=None*, ***kwargs*)

Class to make a Gromacs XPM matrix available as a NumPy `numpy.ndarray`.

The data is available in the attribute *XPM.array*.

Note: By default, the rows (2nd dimension) in the *XPM.array* are re-ordered so that row 0 (i.e. `array[:, 0]` corresponds to the first residue/hydrogen bond/etc. The original xpm matrix is obtained for *reverse* = `False`. The *XPM* reader always reorders the *XPM.yvalues* (obtained from the xpm file) to match the order of the rows.

Initialize xpm structure.

Arguments

filename read from mdp file

autoconvert try to guess the type of the output array from the colour legend [`True`]

reverse reverse rows (2nd dimension): re-orders the rows so that the first row corresponds e.g. to the first residue or first H-bonds and not the last) [`True`]

xvalues
Values of on the x-axis, extracted from the xpm file.

yvalues
Values of on the y-axis, extracted from the xpm file. These are in the same order as the rows in the xpm matrix. If *reverse* = `False` then this is typically a *descending* list of numbers (highest to lowest residue number, index number, etc). For *reverse* = `True` it is resorted accordingly.

array
XPM matrix as a `numpy.ndarray`.

The attribute itself cannot be assigned a different array but the contents of the array can be modified.

col(*c*)
Parse colour specification

parse()
Parse the xpm file and populate *XPM.array*.

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

static uncomment(*s*)
Return string *s* with C-style comments `/* ... */` removed.

static unquote(*s*)
Return string *s* with quotes `"` removed.

Example: Analysing H-bonds

Run `gromacs.g_hbond()` to produce the existence map (and the log file for the atoms involved in the bonds; the ndx file is also useful):

```
gromacs.g_hbond(s=TPR, f=XTC, g="hbond.log", hbm="hb.xpm", hbn="hb.ndx")
```

Load the XPM:

```
hb = XPM("hb.xpm", reverse=True)
```

Calculate the fraction of time that each H-bond existed:

```
hb_fraction = hb.array.mean(axis=0)
```

Get the descriptions of the bonds:

```
desc = [line.strip() for line in open("hbond.log") if not line.startswith('#')]
```

Note: It is important that `reverse=True` is set so that the rows in the xpm matrix are brought in the same order as the H-bond labels.

Show the results:

```
print "\n".join(["%-40s %4.1f%%" % p for p in zip(desc, 100*hb_fraction)])
```

See also:

`gromacs.analysis.plugins.hbonds`

Gromacs parameter MDP file format

The .mdp file contains a list of keywords that are used to set up a simulation with `Grompp`. The class *MDP* parses this file and provides access to the keys and values as ordered dictionary.

class gromacs.fileformats.mdp.**MDP**(*filename=None*, *autoconvert=True*, ***kwargs*)

Class that represents a Gromacs mdp run input file.

The MDP instance is an ordered dictionary.

- *Parameter names* are keys in the dictionary.
- *Comments* are sequentially numbered with keys Comment0001, Comment0002, ...
- *Empty lines* are similarly preserved as Blank0001, ...

When writing, the dictionary is dumped in the recorded order to a file. Inserting keys at a specific position is not possible.

Currently, comments after a parameter on the same line are discarded. Leading and trailing spaces are always stripped.

See also:

For editing a mdp file one can also use *gromacs.cbook.edit_mdp()* (which works like a poor replacement for sed).

Initialize mdp structure.

Arguments

filename read from mdp file

autoconvert [boolean] `True` converts numerical values to python numerical types; `False` keeps everything as strings [`True`]

kwargs Populate the MDP with key=value pairs. (NO SANITY CHECKS; and also does not work for keys that are not legal python variable names such as anything that includes a minus '-' sign or starts with a number).

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

write (*filename=None*, *skipempty=False*)
Write mdp file to *filename*.

Keywords

filename output mdp file; default is the filename the mdp was read from

skipempty [boolean] `True` removes any parameter lines from output that contain empty values [`False`]

Note: Overwrites the file that the mdp was read from if no *filename* supplied.

Gromacs NDX index file format

The .ndx file contains lists of atom indices that are grouped in sections by *group names*. The classes *NDX* and *uniqueNDX* can parse such ndx files and provide convenient access to the individual groups.

class gromacs.fileformats.ndx.**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()
```

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

groups
Return a list of all groups.

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.

setdefault (*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

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 gromacs.fileformats.ndx.**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 gromacs.fileformats.ndx.**IndexSet**
set which defines '+' as union (OR) and '-' as intersection (AND).

Gromacs Preprocessed Topology (top) Parser

New in version 0.5.0.

Gromacs can produce *preprocessed topology files* that contain *all* topology information (generated using `grompp -pp processed.top`). Reading the regular topol.top *is not supported*, for now, since the `#include` statements are not handled. The *TOP* parser can read an write processed.top files. The *TOP* also provides an interface to modify the force-field terms and parameters in a programmatic way. Example applications involve system preparation for Hamiltonian-replica exchange (REST2 with lambda scaling), and automated force-field parametrization.

Gromacs TOP file format

Classes

class gromacs.fileformats.top.**TOP**(*fname*)
Class to make a TOP object from a GROMACS processed.top file

The force-field and molecules data is exposed as python object.

Note: Only processed.top files generated by GROMACS 'grompp -pp' are supported - the usual topol.top files are not supported (yet!)

Initialize the TOP structure.

Arguments

fname name of the processed.top file

write(*filename*)
Write the TOP object to a file

class gromacs.fileformats.top.**SystemToGroTop**(*system*, *outfile='output.top'*, *multiple_output=False*)
Converter class - represent TOP objects as GROMACS topology file.

Initialize GROMACS topology writer.

Arguments

system `blocks.System` object, containing the topology

outfile name of the file to write to

multiple_output if True, write moleculetypes to separate files, named mol_MOLNAME.itp (default: False)

assemble_topology()
Call the various member self._make_* functions to convert the topology object into a string

History

Sources adapted from code by Reza Salari https://github.com/resal81/PyTopol

Example: Read a processed.top file and scale charges

Run `grompp -pp` to produce a processed.top from conf.gro, grompp.mdp and topol.top files:

```
$ grompp -pp
```

This file now containts all the force-field information:

```
from gromacs.fileformats import TOP
top = TOP("processed.top")
```

Scale the LJ epsilon by an arbitrary number, here 0.9

```
scaling = 0.9
for at in top.atomtypes:
  at.gromacs['param']['lje'] *= scaling
```

Write out the scaled down topology:

```
top.write("output.top")
```

Note: You can use this to prepare a series of top files for Hamiltonian Replica Exchange (HREX) simulations. See `scripts/gw-partial_tempering.py` for an example.

Gromacs TOP - BLOCKS boiler-plate code

Classes

class gromacs.fileformats.blocks.**System**
Top-level class containing molecule topology.

Contains all the parameter types (AtomTypes, BondTypes, ...) and molecules.

class gromacs.fileformats.blocks.**Molecule**
Class that represents a Molecule

Contains all the molecule attributes: atoms, bonds, angles dihedrals. Also contains settle, cmap and exclusion sections, if present.

anumb_to_atom(*anumb*)
Returns the atom object corresponding to an atom number

renumber_atoms()
Reset the molecule's atoms `number` to be 1-indexed

class gromacs.fileformats.blocks.**Atom**
Class that represents an Atom

Contains only the simplest atom attributes, that are contained like in section example below.

Molecule cantains an `atoms` that's a list-container for *Atom* instances.

class gromacs.fileformats.blocks.**Param**(*format*)
Class that represents an abstract Parameter.

This class is the parent to AtomType, BondType and all the other parameter types.

The class understands a parameter line and that a `comment` that may follow. CMapType is an exception (it's a multi-line parameter).

`convert()` provides a rudimentary support for parameter unit conversion between GROMACS and CHARMM notation: change kJ/mol into kcal/mol and nm into Angstrom.

`disabled` for supressing output when writing-out to a file.

class gromacs.fileformats.blocks.**AtomType**(*format*)

class gromacs.fileformats.blocks.**BondType**(*format*)

class gromacs.fileformats.blocks.**AngleType**(*format*)

class gromacs.fileformats.blocks.**DihedralType**(*format*)

class gromacs.fileformats.blocks.**ImproperType**(*format*)

class gromacs.fileformats.blocks.**CMapType**(*format*)

class gromacs.fileformats.blocks.**InteractionType**(*format*)

class gromacs.fileformats.blocks.**SettleType**(*format*)

class gromacs.fileformats.blocks.**ConstraintType**(*format*)

class gromacs.fileformats.blocks.**NonbondedParamType**(*format*)

class gromacs.fileformats.blocks.**VirtualSites3Type**(*format*)

class gromacs.fileformats.blocks.**Exclusion**

Class to define non-interacting pairs of atoms, or "exclusions".

Note: Does not inherit from *Param* unlike other classes in `blocks`

History

Sources adapted from code by Reza Salari https://github.com/resal81/PyTopol

gromacs.fileformats.convert — converting entries of tables

The *Autoconverter* converts input values to appropriate Python types.

It is mainly used by *gromacs.fileformats.xpm.XPM* to automagically generate useful NumPy arrays from xpm files. Custom conversions beyond the default ones in *Autoconverter* can be provided with the constructor keyword *mapping*.

See also:

The *Autoconverter* class was taken and slightly adapted from *recsql.converter* in RecSQL.

class gromacs.fileformats.convert.**Autoconverter**(*mode='fancy'*, *mapping=None*, *active=True*, *sep=False*, ***kwargs*)

Automatically convert an input value to a special python object.

The *Autoconverter.convert()* method turns the value into a special python value and casts strings to the "best" type (see *besttype()*).

The defaults for the conversion of a input field value to a special python value are:

value	python
'---'	`None`
''	`None`
'True'	`True`
'x'	`True`
'X'	`True`
'yes'	`True`
'Present'	`True`
'False'	`False`
'-'	`False`
'no'	`False`
'None'	`False`
'none'	`False`

If the *sep* keyword is set to a string instead of `False` then values are split into tuples. Probably the most convenient way to use this is to set *sep* = `True` (or `None`) because this splits on all white space whereas *sep* = ' ' would split multiple spaces.

Example

- With *sep* = `True`: 'foo bar 22 boing ---' –> ('foo', 'bar', 22, 'boing', None)
- With *sep* = ',': 1,2,3,4 –> (1,2,3,4)

Initialize the converter.

Arguments

mode defines what the converter does

"simple" convert entries with *besttype()*

"singlet" convert entries with *besttype()* and apply mappings

"fancy" first splits fields into lists, tries mappings, and does the stuff that "singlet" does

"unicode" convert all entries with *to_unicode()*

mapping any dict-like mapping that supports lookup. If''None'' then the hard-coded defaults are used

active* or *autoconvert initial state of the *Autoconverter.active* toggle. `False` deactivates any conversion. [`True`]

sep character to split on (produces lists); use `True` or `None` (!) to split on all white space.

Changed in version 0.7.0: removed 'encoding keyword argument

convert(*x*)
Convert *x* (if in the active state)

active
If set to `True` then conversion takes place; `False` just returns *besttype()* applid to the value.

active
Toggle the state of the Autoconverter. `True` uses the mode, `False` does nothing

gromacs.fileformats.convert.**besttype**(*x*)
Convert string x to the most useful type, i.e. int, float or unicode string.

If x is a quoted string (single or double quotes) then the quotes are stripped and the enclosed string returned.

Note: Strings will be returned as Unicode strings (using *to_unicode()*).

Changed in version 0.7.0: removed 'encoding keyword argument

gromacs.fileformats.convert.**to_unicode**(*obj*)
Convert obj to unicode (if it can be be converted).

Conversion is only attempted if *obj* is a string type (as determined by `six.string_types`).

Changed in version 0.7.0: removed 'encoding keyword argument

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 gromacs.utilities.**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

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.

Changed in version 0.3.1: An empty string as *ext* = "" will suppress appending an extension.

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

class gromacs.utilities.**AttributeDict**
A dictionary with pythonic access to keys as attributes — useful for interactive work.

class gromacs.utilities.**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:

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

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

Arguments

datasource a stream or a filename

mode `'r'` opens for reading, `'w'` for writing ['r']

kwargs additional keyword arguments that are passed through to the actual handler; if these are not appropriate then an exception will be raised by the handler

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

Expands ~ and environment variables such as `$HOME`.

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

gromacs.utilities.**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.

gromacs.utilities.**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`.

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

gromacs.utilities.**which**(*program*)
Determine full path of executable *program* on `PATH`.

(Jay at http://stackoverflow.com/questions/377017/test-if-executable-exists-in-python)

New in version 0.5.1.

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

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

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

gromacs.utilities.**firstof**(*obj*)
Returns the first entry of a sequence or the obj.

Treats strings as single objects.

Functions that help handling Gromacs files:

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

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

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

gromacs.utilities.**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:

gromacs.utilities.**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.

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

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

Miscellaneous functions:

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

gromacs.utilities.**autoconvert**(*s*)
Convert input to a numerical type if possible.

1. A non-string object is returned as it is
2. Try conversion to int, float, str.

Data

gromacs.utilities.**amino_acid_codes = {'A': 'ALA', 'C': 'CYS', 'D': 'ASP', 'E': 'GLU', 'F':**
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

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 gromacs.collections.**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']
```

gromacs.tools – Gromacs commands classes

A Gromacs command class produces an instance of a Gromacs tool command (*gromacs.core.GromacsCommand*), any argument or keyword argument supplied will be used as default values for when the command is run.

Classes has the same name of the corresponding Gromacs tool with the first letter capitalized and dot and dashes replaced by underscores to make it a valid python identifier. Gromacs 5 tools are also aliased to their Gromacs 4 tool names (e.g, *sasa* to *g_sas*) for backwards compatibility.

The list of tools to be loaded is configured with the tools and groups options of the ~/.gromacswrapper.cfg file. Guesses are made if these options are not provided.

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

```
from gromacs.tools import Trjconv

trjconv = tools.Trjconv()
trjconv_compact = tools.Trjconv(ur='compact', center=True, boxcenter='tric', pbc='mol
↪',
                                input=('protein','system'))
```

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 specific, repeatedly encountered problem reliably. This is particularly helpful when doing interactive work.

Multi index

It is possible to extend the tool 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 gromacs.tools.**GromacsCommandMultiIndex**(***kwargs*)
Command class that accept multiple index files.

It works combining multiple index files into a single temporary one so that tools that do not (yet) support multi index files as input can be used as if they did.

It creates a new file only if multiple index files are supplied.

gromacs.tools.**merge_ndx**(**args*)
Takes one or more index files and optionally one structure file and returns a path for a new merged index file.

Parameters **args** – index files and zero or one structure file

Returns path for the new merged index file

Helpers

gromacs.tools.**tool_factory**(*clsname*, *name*, *driver*, *base=<class 'gromacs.core.GromacsCommand'>*)
Factory for GromacsCommand derived types.

gromacs.tools.**load_v4_tools**()
Load Gromacs 4.x tools automatically using some heuristic.

Tries to load tools (1) in configured tool groups (2) and fails back to automatic detection from `GMXBIN` (3) then to a prefilled list.

Also load any extra tool configured in `~/.gromacswrapper.cfg`

Returns dict mapping tool names to GromacsCommand classes

gromacs.tools.**load_v5_tools**()
Load Gromacs 2018/2016/5.x tools automatically using some heuristic.

Tries to load tools (1) using the driver from configured groups (2) and falls back to automatic detection from `GMXBIN` (3) then to rough guesses.

In all cases the command `gmx help` is ran to get all tools available.

Returns dict mapping tool names to GromacsCommand classes

gromacs.tools.**find_executables**(*path*)
Find executables in a path.

Searches executables in a directory excluding some know commands unusable with GromacsWrapper.

Parameters **path** – dirname to search for

Returns list of executables

gromacs.tools.**make_valid_identifier**(*name*)
Turns tool names into valid identifiers.

Parameters **name** – tool name

Returns valid identifier

exception gromacs.tools.**GromacsToolLoadingError**
Raised when no Gromacs tool could be found.

Gromacs tools

class gromacs.tools.**Dyecoupl**

class gromacs.tools.**G_spatial**

class gromacs.tools.**Sigeps**

class gromacs.tools.**Density**

class gromacs.tools.**Chi**

class gromacs.tools.**G_filter**

class gromacs.tools.**Genrestr**

class gromacs.tools.**Nmtraj**

class gromacs.tools.**Analyze**

class gromacs.tools.**Helixorient**

class gromacs.tools.**G_sans**

class gromacs.tools.**Trjcat**

class gromacs.tools.**G_densorder**

class gromacs.tools.**G_helixorient**

class gromacs.tools.**Velacc**

class gromacs.tools.**G_principal**

class gromacs.tools.**Spol**

class gromacs.tools.**G_densmap**

class gromacs.tools.**Confrms**

class gromacs.tools.**G_order**

class gromacs.tools.**G_nmens**

class gromacs.tools.**Grompp**

class gromacs.tools.**G_angle**

class gromacs.tools.**Editconf**

class gromacs.tools.**Bar**

class gromacs.tools.**Trjconv**

class gromacs.tools.**Clustsize**

class gromacs.tools.**G_sgangle**

class gromacs.tools.**G_pairdist**

class gromacs.tools.**G_dyecoupl**

class gromacs.tools.**G_hydorder**

class gromacs.tools.**Sasa**

class gromacs.tools.**Vanhove**

class gromacs.tools.**G_help**

class gromacs.tools.**Help**

class gromacs.tools.**Solvate**

class gromacs.tools.**Hydorder**

class gromacs.tools.**Enemat**

class gromacs.tools.**Genion**

class gromacs.tools.**G_sham**

class gromacs.tools.**Polystat**

class gromacs.tools.**G_enemat**

class gromacs.tools.**G_density**

class gromacs.tools.**G_sigeps**

class gromacs.tools.**Check**

class gromacs.tools.**Select**

class gromacs.tools.**Tpbconv**

class gromacs.tools.**G_tcaf**

class gromacs.tools.**Genbox**

class gromacs.tools.**Nmens**

class gromacs.tools.**G_rms**

class gromacs.tools.**Pairdist**

class gromacs.tools.**G_dielectric**

class gromacs.tools.**Spatial**

class gromacs.tools.**G_anadock**

class gromacs.tools.**H2order**

class gromacs.tools.**G_disre**

class gromacs.tools.**Wham**

class gromacs.tools.**Nmr**

class gromacs.tools.**Mdrun**

class gromacs.tools.**Densorder**

class gromacs.tools.**G_confrms**

class gromacs.tools.**Trjorder**

class gromacs.tools.**G_view**

class gromacs.tools.**Awh**

class gromacs.tools.**G_dos**

class gromacs.tools.**G_hbond**

class gromacs.tools.**Tune_pme**

class gromacs.tools.**Anadock**

class gromacs.tools.**G_rdf**

class gromacs.tools.**Rmsf**

class gromacs.tools.**Sans**

class gromacs.tools.**Rmsdist**

class gromacs.tools.**Saltbr**

class gromacs.tools.**G_rama**

class gromacs.tools.**Disre**

class gromacs.tools.**Rdf**

class gromacs.tools.**Gmxdump**

class gromacs.tools.**Gangle**

class gromacs.tools.**G_h2order**

class gromacs.tools.**G_traj**

class gromacs.tools.**Anaeig**

class gromacs.tools.**Dump**

class gromacs.tools.**G_rotacf**

class gromacs.tools.**Energy**

class gromacs.tools.**G_lie**

class gromacs.tools.**G_x2top**

class gromacs.tools.**G_nmeig**

class gromacs.tools.**Rotmat**

class gromacs.tools.**G_dist**

class gromacs.tools.**G_gyrate**

class gromacs.tools.**Gmxcheck**

class gromacs.tools.**G_mindist**

class gromacs.tools.**G_sas**

class gromacs.tools.**Convert_tpr**

class gromacs.tools.**G_nmtraj**

class gromacs.tools.**Helix**

class gromacs.tools.**Densmap**

class gromacs.tools.**Msd**

class gromacs.tools.**Sham**

class gromacs.tools.**G_covar**

class gromacs.tools.**Saxs**

class gromacs.tools.**Bundle**

class gromacs.tools.**Pdb2gmx**

class gromacs.tools.**Mindist**

class gromacs.tools.**G_helix**

class gromacs.tools.**Mk_angndx**

class gromacs.tools.**G_mdmat**

class gromacs.tools.**Sorient**

class gromacs.tools.**G_nmr**

class gromacs.tools.**Insert_molecules**

class gromacs.tools.**Morph**

class gromacs.tools.**G_energy**

class gromacs.tools.**Filter**

class gromacs.tools.**G_rmsdist**

class gromacs.tools.**Gyrate**

class gromacs.tools.**G_clustsize**

class gromacs.tools.**Mdmat**

class gromacs.tools.**G_dipoles**

class gromacs.tools.**Freevolume**

class gromacs.tools.**Rama**

class gromacs.tools.**Xpm2ps**

class gromacs.tools.**Rms**

class gromacs.tools.**G_wham**

class gromacs.tools.**G_vanhove**

class gromacs.tools.**G_anaeig**

class gromacs.tools.**Cluster**

class gromacs.tools.**G_spol**

class gromacs.tools.**Dyndom**

class gromacs.tools.**G_pme_error**

class gromacs.tools.**G_current**

class gromacs.tools.**Eneconv**

class gromacs.tools.**Make_edi**

class gromacs.tools.**Lie**

class gromacs.tools.**G_potential**

class gromacs.tools.**Angle**

class gromacs.tools.**X2top**

class gromacs.tools.**Pme_error**

class gromacs.tools.**Trajectory**

class gromacs.tools.**G_select**

class gromacs.tools.**G_insert_molecules**

class gromacs.tools.**Current**

class gromacs.tools.**G_velacc**

class gromacs.tools.**Potential**

class gromacs.tools.**Dipoles**

class gromacs.tools.**Tcaf**

class gromacs.tools.**G_rotmat**

class gromacs.tools.**G_polystat**

class gromacs.tools.**Wheel**

class gromacs.tools.**G_bundle**

class gromacs.tools.**G_dyndom**

class gromacs.tools.**Covar**

class gromacs.tools.**G_msd**

class gromacs.tools.**Dielectric**

class gromacs.tools.**G_cluster**

class gromacs.tools.**Distance**

class gromacs.tools.**G_saltbr**

class gromacs.tools.**Rotacf**

class gromacs.tools.**G_freevolume**

class gromacs.tools.**Do_dssp**

class gromacs.tools.**G_bar**

class gromacs.tools.**Nmeig**

class gromacs.tools.**Hbond**

class gromacs.tools.**Traj**

class gromacs.tools.**G_wheel**

class gromacs.tools.**G_analyze**

class gromacs.tools.**Make_ndx**

class gromacs.tools.**Dos**

class gromacs.tools.**Order**

class gromacs.tools.**View**

class gromacs.tools.**Genconf**

class gromacs.tools.**G_tune_pme**

class gromacs.tools.**G_saxs**

class gromacs.tools.**G_mk_angndx**

class gromacs.tools.**G_trajectory**

class gromacs.tools.**G_morph**

class gromacs.tools.**G_chi**

class gromacs.tools.**G_awh**

class gromacs.tools.**G_sorient**

class gromacs.tools.**G_rmsf**

class gromacs.tools.**Principal**

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*):

gromacs.cbook.**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.

gromacs.cbook.**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.

gromacs.cbook.**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

---

gromacs.cbook.**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); if a list or tuple is provided then s[0] is used for pass 1 (should be a tpr) and s[1] is used for the fitting step (can be a pdb of the whole system)

If a second structure is supplied then it is assumed that the fitted trajectory should *not* be centered.

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 -boxcenter tric -ur compact -pbc mol
trjconv -s a.tpr -f b.xtc -o c.xtc -fit rot+trans
```

gromacs.cbook.**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 gromacs.cbook.**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 gromacs.cbook.**Transformer**(*s='topol.tpr'*, *f='traj.xtc'*, *n=None*, *force=None*, *dirname='.'*, *outdir=None*)
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]

dirname directory in which all operations are performed, relative paths are interpreted relative to *dirname* [.]

outdir directory under which output files are placed; by default the same directory where the input files live

center_fit(***kwargs*)

Write compact xtc that is fitted to the tpr reference structure.

See *gromacs.cbook.trj_fitandcenter()* for details and description of *kwargs* (including *input*, *input1*, *n* and *n1* for how to supply custom index groups). 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.

Runs `gromacs.tools.trjconv` with appropriate arguments for fitting. The most important *kwargs* are listed here but in most cases the defaults should work.

Note that the default settings do *not* include centering or periodic boundary treatment as this often does not work well with fitting. It is better to do this as a separate step (see *center_fit()* or *gromacs.cbook.trj_fitandcenter()*)

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]

fitgroup index group to fit on ["backbone"]

Note: If keyword *input* is supplied then it will override *fitgroup*; *input* = `[fitgroup, outgroup]`

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.)

outfile(*p*)

Path for an output file.

If `outdir` is set then the path is `outdir/basename(p)` else just `p`

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_fit(***kwargs*)
Strip water and fit to the remaining system.

First runs *strip_water()* and then *fit()*; see there for arguments.

- *strip_input* is used for *strip_water()* (but is only useful in special cases, e.g. when there is no Protein group defined. Then set *strip_input* = `['Other']`.
- *input* is passed on to *fit()* and can contain the `[center_group, fit_group, output_group]`
- *fitgroup* is only passed to *fit()* and just contains the group to fit to ("backbone" by default)

> **Warning:** *fitgroup* can only be a Gromacs default group and not a custom group (because the indices change after stripping)

- By default *fit* = "rot+trans" (and *fit* is passed to *fit()*, together with the *xy* = `False` keyword)

Note: The call signature of *strip_water()* is somewhat different from this one.

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`]

centergroup Index group used for centering ["Protein"]

Note: If *input* is provided (see below under *kwargs*) then *centergroup* is ignored and the group for centering is taken as the first entry in *input*.

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.)

gromacs.cbook.**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.

gromacs.cbook.**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 only shown when an error occurs. For debugging, look at the log file or screen output and 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 informative output of **grompp** it can break when the output format changes. This version recognizes lines like

```
'  System has non-zero total charge: -4.000001e+00'
```

using the regular expression `System has non-zero total charge: *(?P<qtot>[-+]?d*.d+([eE][-+]d+)?)`.

gromacs.cbook.**get_volume**(*f*)

Return the volume in nm^3 of structure file *f*.

(Uses `gromacs.editconf()`; error handling is not good)

gromacs.cbook.**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

`gromacs.cbook.`**`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

grompp_kwargs* other options for **`grompp`** such as `maxwarn=2` can also be supplied

Returns full path to the processed topology

`gromacs.cbook.`**`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. If a value is a list-like object then the items are written as a sequence, joined with spaces, e.g.

```
ref_t=[310,310,310] --->  ref_t = 310 310 310
```

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.

`gromacs.cbook.`**`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.

---

gromacs.cbook.**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 only shown when an error occurs. For debugging, look at the log file or screen output and 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 informative output of **grompp** it can break when the output format changes. This version recognizes lines like

```
'  System has non-zero total charge: -4.000001e+00'
```

using the regular expression System has non-zero total charge: *(?P<qtot>[-+]?d*.d+([eE][-+]d+)?).

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 gromacs.cbook.**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 or `IndexBuilder.write()` to simply write out the individual named selections (useful with *names*).

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. When simply using `IndexBuilder.write()` then these should be supplied.

name_all [string] Name of the group that is generated by *IndexBuilder.combine()*.

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); if set to `False` then no combined group is created and only the individual groups are written. ["|"]

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.

See also:

`IndexBuilder.write()`.

gmx_resid(*resid*)
Returns resid in the Gromacs index by transforming with offset.

`gromacs.cbook.`**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

`gromacs.cbook.`**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.

`gromacs.cbook.`**`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:

`gromacs.cbook.`**`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. If a value is a list-like object then the items are written as a sequence, joined with spaces, e.g.

```
ref_t=[310,310,310] --->  ref_t = 310 310 310
```

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.

`gromacs.cbook.`**`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=True)
```

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:

gromacs.setup.**topology**(*struct=None*, *protein='protein'*, *top='system.top'*, *dirname='top'*, *posres='posres.itp'*, *ff='oplsaa'*, *water='tip4p'*, ***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

ff force field (string understood by `pdb2gmx`); default "oplsaa"

water water model (string), default "tip4p"

pdb2gmxargs other arguments for `pdb2gmx`

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.

---

gromacs.setup.**solvate**(*struct='top/protein.pdb'*, *top='top/system.top'*, *distance=0.9*, *boxtype='dodecahedron'*, *concentration=0*, *cation='NA'*, *anion='CL'*, *water='tip4p'*, *solvent_name='SOL'*, *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).

Note: The defaults are suitable for solvating a globular protein in a fairly tight (increase *distance*!) dodecahedral box.

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 *boxtype* to `None` in order to use a box size in the input file (gro or pdb).

***boxtype* or *bt*: string** Any of the box types supported by `Editconf` (triclinic, cubic, dodecahedron, octahedron). Set the box dimensions either with *distance* or the *box* and *angle* keywords.

If set to `None` it will ignore *distance* and use the box inside the *struct* file.

bt overrides the value of *boxtype*.

box List of three box lengths [A,B,C] that are used by `Editconf` in combination with *boxtype* (`bt` in **`editconf`**) and *angles*. Setting *box* overrides *distance*.

angles List of three angles (only necessary for triclinic boxes).

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 an alternative solvent is required, simply supply the path to a box with solvent molecules (used by `genbox()`'s *cs* argument) and also supply the molecule name via *solvent_name*.

solvent_name Name of the molecules that make up the solvent (as set in the itp/top). Typically needs to be changed when using non-standard/non-water solvents. ["SOL"]

with_membrane [bool] `True`: use special `vdwradii.dat` with 0.1 nm-increased radii on lipids. Default is `False`.

ndx [filename] How to name the index file that is produced by this function.

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

kwargs Additional arguments are passed on to `Editconf` or are interpreted as parameters to be changed in the mdp file.

`gromacs.setup.`**`energy_minimize`**(*dirname='em'*, *mdp='/home/docs/.cache/Python-Eggs/GromacsWrapper-0.7.0-py2.7.egg-tmp/gromacs/templates/em.mdp'*, *struct='solvate/ionized.gro'*, *top='top/system.top'*, *output='em.pdb'*, *deffnm='em'*, *mdrunner=None*, *mdrun_args=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` instance; by default we just try `gromacs.mdrun_d()` and `gromacs.mdrun()` but a MDrunner instance gives the user the ability to run mpi jobs etc. [None]

mdrun_args arguments for *mdrunner* (as a dict), e.g. `{'nt': 2}`; empty by default

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.

`gromacs.setup.`**`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.

`gromacs.setup.`**`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. The pressure coupling option *refcoord_scaling* is set to "com" by default (but can be changed via *kwargs*) and the pressure coupling algorithm itself is set to *Pcoupl* = "Berendsen" to run a stable simulation.

`gromacs.setup.`**`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.

`gromacs.setup.`**`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__.

`gromacs.setup.`**`check_mdpargs`**(*d*)
Check if any arguments remain in dict *d*.

`gromacs.setup.`**`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.

`gromacs.setup.`**`_setup_MD`**(*dirname*, *deffnm='md'*, *mdp='/home/docs/.cache/Python-Eggs/GromacsWrapper-0.7.0-py2.7.egg-tmp/gromacs/templates/md_OPLSAA.mdp'*, *struct=None*, *top='top/system.top'*, *ndx=None*, *mainselection='"Protein"'*, *qscript='/home/docs/.cache/Python-Eggs/GromacsWrapper-0.7.0-py2.7.egg-tmp/gromacs/templates/local.sh'*, *qname=None*, *startdir=None*, *mdrun_opts=''*, *budget=None*, *walltime=0.3333333333333333*, *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:

`gromacs.setup.`**`CONC_WATER = 55.345`**
float(x) -> floating point number

Convert a string or number to a floating point number, if possible.

`gromacs.setup.`**`vdw_lipid_resnames = ['POPC', 'POPE', 'POPG', 'DOPC', 'DPPC', 'DLPC', 'DMPC',`**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

`gromacs.setup.`**`vdw_lipid_atom_radii = {'C': 0.25, 'H': 0.09, 'N': 0.16, 'O': 0.155}`**
dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

`gromacs.scaling` – Partial tempering

Author Jan Domanski, @jandom

New in version 0.5.0.

Helper functions for scaling gromacs topologies; useful for setting up simulations with Hamiltonian replicate exchange and partial tempering (REST2).

gromacs.scaling.**scale_dihedrals**(*mol*, *dihedrals*, *scale*, *banned_lines=None*)
Scale dihedral angles

gromacs.scaling.**scale_impropers**(*mol*, *impropers*, *scale*, *banned_lines=None*)
Scale improper dihedrals

gromacs.scaling.**partial_tempering**(*topfile='processed.top'*, *outfile='scaled.top'*, *banned_lines=''*, *scale_lipids=1.0*, *scale_protein=1.0*)
Set up topology for partial tempering (REST2) replica exchange.

Changed in version 0.7.0: Use keyword arguments instead of an *args* Namespace object.

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.

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 2: 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=/*
STARTDIR=	.	*startdir*	remote jobdir	*/^ *STARTDIR=/*
WALL_HOURS=	0.33	*walltime* h	mdrun's -maxh	*/^ *WALL_HOURS=/*
NPME=		*npme*	PME nodes	*/^ *NPME=/*
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. (Exception: any value that follows an equals sign "=" is replaced, regardless of the default value in the table *except* for `MDRUN_OPTS` where *only "" will be replace*.) 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 3: 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 ++++++).

```
#!/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
```

(continues on next page)

(continued from previous page)

```
# 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
↪                $MDRUN_OPTS                          -maxh ${WALL_HOURS} > $OUTPUT
rc=$?

# 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 provide 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 gromacs.qsub.**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.

gromacs.qsub.**generate_submit_scripts**(*templates*, *prefix=None*, *deffnm='md'*, *jobname='MD'*, *budget=None*, *mdrun_opts=None*, *walltime=1.0*, *jobarray_string=None*, *startdir=None*, *npme=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]

startdir Explicit path on the remote system (for run scripts that need to *cd* into this directory at the beginning of execution) [None]

mdrun_opts String of additional options for **mdrun**.

walltime Maximum runtime of the job in hours. [1]

npme number of PME nodes

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

gromacs.qsub.**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.

`gromacs.qsub.`**`detect_queuing_system`**(*scriptfile*)
Return the queuing system for which *scriptfile* was written.

`gromacs.qsub.`**`queuing_systems = [<Sun Gridengine QueuingSystem instance>, <PBS QueuingSyster`**
list() -> new empty list list(iterable) -> new list initialized from iterable's items

1.4 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*.

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.

ParmEd A general tool for working with topology files for all the popular MD codes, including the parmed.gromacs module for ITP and TOP files.

gmxapi (M.E. Irrgang, J.M. Hays, and P.M. Kasson) `gmxapi` provides interfaces for managing and extending molecular dynamics simulation workflows. In this repository, a Python package provides the `gmx` module for high-level interaction with GROMACS. `gmx.core` provides Python bindings to the gmxapi C++ GROMACS external API.

Irrgang, M. E., Hays, J. M., & Kasson, P. M. gmxapi: a high-level interface for advanced control and extension of molecular dynamics simulations. Bioinformatics 2018. DOI: 10.1093/bioinformatics/bty484

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

gmxscript (Pedro Lacerda) **gmxscript** is a framework for Gromacs simulations. Its main goal is make simulation protocols easily reproducible and to define canonical steps to perform and analyze a simulation. The commands are stored in very readable and structured Python file that requires no programming knowledge except syntax.

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 (René Pool, 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 and the latest sources are hosted at https://github.com/GromPy

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.

copernicus Copernicus is a Python-based client-server network that allows running of large and complicated MD simulation workflows. It supports Gromacs (grompp and mdrun).

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).

JGromacs (Márton Münz and Philip C Biggin) JGromacs is a Java library designed to facilitate the development of cross-platform analysis applications for Molecular Dynamics (MD) simulations. The package contains parsers for file formats applied by GROMACS. It provides a multilevel object-oriented representation of simulation data to integrate and interconvert sequence, structure and dynamics information. In addititon, a basic analysis toolkit is included in the package. The programmer is also provided with simple tools (e.g. XML-based configuration) to create applications with a user interface resembling the command-line UI of Gromacs applications.

Please open an issue in the issue tracker to let us know of other efforts so that they can be added here. Thanks.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[FrenkelSmit2002] D. Frenkel and B. Smit, Understanding Molecular Simulation. Academic Press, San Diego 2002

Python Module Index

g

gromacs, 7
gromacs.cbook, 51
gromacs.collections, 44
gromacs.config, 15
gromacs.core, 10
gromacs.environment, 21
gromacs.fileformats.blocks, 38
gromacs.fileformats.convert, 39
gromacs.fileformats.mdp, 34
gromacs.fileformats.ndx, 35
gromacs.fileformats.top, 36
gromacs.fileformats.xpm, 32
gromacs.fileformats.xvg, 23
gromacs.qsub, 73
gromacs.scaling, 72
gromacs.setup, 66
gromacs.tools, 44
gromacs.utilities, 40

Index

Symbols

$HOME, 42
__call__() (gromacs.core.Command method), 14
_setup_MD() (in module gromacs.setup), 72

A

activate_subplot() (in module gromacs.utilities), 43
active (gromacs.fileformats.convert.Autoconverter attribute), 40
add_mdp_includes() (in module gromacs.cbook), 60
all_frames (gromacs.cbook.Frames attribute), 54
amino_acid_codes (in module gromacs.utilities), 43
AngleType (class in gromacs.fileformats.blocks), 38
anumb_to_atom() (gromacs.fileformats.blocks.Molecule method), 38
anyopen() (in module gromacs.utilities), 42
array (gromacs.fileformats.xpm.XPM attribute), 33
array (gromacs.fileformats.xvg.XVG attribute), 27
array() (gromacs.qsub.QueuingSystem method), 75
array_flag() (gromacs.qsub.QueuingSystem method), 75
asiterable() (in module gromacs.utilities), 43
assemble_topology() (gromacs.fileformats.top.SystemToGroTop method), 37
Atom (class in gromacs.fileformats.blocks), 38
AtomType (class in gromacs.fileformats.blocks), 38
AttributeDict (class in gromacs.utilities), 41
autoconvert() (in module gromacs.utilities), 43
Autoconverter (class in gromacs.fileformats.convert), 39
Autoconverter.convert() (in module gromacs.fileformats.convert), 40
AutoCorrectionWarning, 9

B

BadParameterWarning, 9
besttype() (in module gromacs.fileformats.convert), 40
BondType (class in gromacs.fileformats.blocks), 38
break_array() (in module gromacs.fileformats.xvg), 32

C

cat() (in module gromacs.cbook), 52
center_fit() (gromacs.cbook.Transformer method), 55
cfg (in module gromacs.config), 16
check_file_exists() (gromacs.utilities.FileUtils method), 41
check_mdpargs() (in module gromacs.setup), 72
check_setup() (in module gromacs.config), 16
cleanup() (gromacs.cbook.Frames method), 54
CMapType (class in gromacs.fileformats.blocks), 39
col() (gromacs.fileformats.xpm.XPM method), 33
Collection (class in gromacs.collections), 44
combine() (gromacs.cbook.IndexBuilder method), 62
Command (class in gromacs.core), 13
commandline() (gromacs.core.GromacsCommand method), 13
communicate() (gromacs.core.PopenWithInput method), 15
CONC_WATER (in module gromacs.setup), 72
configdir (in module gromacs.config), 16
CONFIGNAME (in module gromacs.config), 17
configuration (gromacs.config.GMXConfigParser attribute), 17
configuration (in module gromacs.config), 17
ConstraintType (class in gromacs.fileformats.blocks), 39
convert_aa_code() (in module gromacs.utilities), 43
create_portable_topology() (in module gromacs.cbook), 59

D

decimate() (gromacs.fileformats.xvg.XVG method), 27
decimate_circmean() (gromacs.fileformats.xvg.XVG method), 27
decimate_error() (gromacs.fileformats.xvg.XVG method), 28
decimate_max() (gromacs.fileformats.xvg.XVG method), 28
decimate_mean() (gromacs.fileformats.xvg.XVG method), 28

decimate_min() (gromacs.fileformats.xvg.XVG method), 29
decimate_percentile() (gromacs.fileformats.xvg.XVG method), 29
decimate_rms() (gromacs.fileformats.xvg.XVG method), 29
decimate_smooth() (gromacs.fileformats.xvg.XVG method), 29
defaults (in module gromacs.config), 17
delete_frames() (gromacs.cbook.Frames method), 54
detect_queuing_system() (in module gromacs.qsub), 77
DihedralType (class in gromacs.fileformats.blocks), 38
doc() (gromacs.environment.Flags method), 22

E

edit_mdp() (in module gromacs.cbook), 59, 64
edit_txt() (in module gromacs.cbook), 65
em_schedule() (in module gromacs.setup), 69
energy_minimize() (in module gromacs.setup), 68
environment variable
    $HOME, 42
    GMXBIN, 19
    GMXDATA, 19
    GROMACSWRAPPER_SUPPRESS_SETUP_CHECK, 16
    GW_START_LOGGING, 10
    LD_LIBRARY_PATH, 19
    PATH, 1, 19, 42
error (gromacs.fileformats.xvg.XVG attribute), 30
errorbar() (gromacs.fileformats.xvg.XVG method), 30
Exclusion (class in gromacs.fileformats.blocks), 39
extract() (gromacs.cbook.Frames method), 54

F

failuremode (gromacs.core.GromacsCommand attribute), 13
filename() (gromacs.utilities.FileUtils method), 41
FileUtils (class in gromacs.utilities), 41
find_executables() (in module gromacs.tools), 45
find_first() (in module gromacs.utilities), 42
firstof() (in module gromacs.utilities), 43
fit() (gromacs.cbook.Transformer method), 55
Flag (class in gromacs.environment), 22
flag() (gromacs.qsub.QueuingSystem method), 76
Flags (class in gromacs.environment), 22
flags (in module gromacs.environment), 22
flagsDocs (class in gromacs.environment), 21
Frames (class in gromacs.cbook), 53

G

generate_submit_array() (in module gromacs.qsub), 76
generate_submit_scripts() (in module gromacs.qsub), 76
get() (gromacs.fileformats.ndx.NDX method), 35
get_configuration() (in module gromacs.config), 16
get_lipid_vdwradii() (in module gromacs.setup), 72
get_ndx_groups() (in module gromacs.cbook), 63
get_template() (in module gromacs.config), 18
get_templates() (in module gromacs.config), 18
get_volume() (in module gromacs.cbook), 58
getLogLevel() (gromacs.config.GMXConfigParser method), 17
getpath() (gromacs.config.GMXConfigParser method), 17
gmx_resid() (gromacs.cbook.IndexBuilder method), 63
GMXBIN, 19
GMXConfigParser (class in gromacs.config), 17
GMXDATA, 19
gromacs (module), 7
gromacs.cbook (module), 51
gromacs.collections (module), 44
gromacs.config (module), 15
gromacs.core (module), 10
gromacs.environment (module), 21
gromacs.fileformats.blocks (module), 38
gromacs.fileformats.convert (module), 39
gromacs.fileformats.mdp (module), 34
gromacs.fileformats.ndx (module), 35
gromacs.fileformats.top (module), 36
gromacs.fileformats.xpm (module), 32
gromacs.fileformats.xvg (module), 23
gromacs.qsub (module), 73
gromacs.scaling (module), 72
gromacs.setup (module), 66
gromacs.tools (module), 44
gromacs.utilities (module), 40
GromacsCommand (class in gromacs.core), 11
GromacsCommandMultiIndex (class in gromacs.tools), 45
GromacsError, 9
GromacsFailureWarning, 9
GromacsImportWarning, 9
GromacsToolLoadingError, 46
GromacsValueWarning, 9
GROMACSWRAPPER_SUPPRESS_SETUP_CHECK, 16
grompp_qtot() (in module gromacs.cbook), 58, 60
groups (gromacs.fileformats.ndx.NDX attribute), 36
GW_START_LOGGING, 10

H

has_arrays() (gromacs.qsub.QueuingSystem method), 76
help() (gromacs.core.Command method), 15
help() (gromacs.core.GromacsCommand method), 13

I

ImproperType (class in gromacs.fileformats.blocks), 38
in_dir() (in module gromacs.utilities), 42
IndexBuilder (class in gromacs.cbook), 61

IndexSet (class in gromacs.fileformats.ndx), 36
infix_filename() (gromacs.utilities.FileUtils method), 41
InteractionType (class in gromacs.fileformats.blocks), 39
isMine() (gromacs.qsub.QueuingSystem method), 76
items() (gromacs.environment.Flags method), 22
iterable() (in module gromacs.utilities), 42
iteritems() (gromacs.environment.Flags method), 22
itervalues() (gromacs.environment.Flags method), 22

J

join() (gromacs.fileformats.ndx.uniqueNDX method), 36

K

keep_protein_only() (gromacs.cbook.Transformer method), 56

L

LD_LIBRARY_PATH, 19
load_v4_tools() (in module gromacs.tools), 45
load_v5_tools() (in module gromacs.tools), 45
logfilename (in module gromacs.config), 18
loglevel_console (in module gromacs.config), 18
loglevel_file (in module gromacs.config), 19
LowAccuracyWarning, 10

M

ma (gromacs.fileformats.xvg.XVG attribute), 30
make_main_index() (in module gromacs.setup), 71
make_ndx_captured() (in module gromacs.cbook), 64
make_valid_identifier() (in module gromacs.tools), 45
max (gromacs.fileformats.xvg.XVG attribute), 30
MD() (in module gromacs.setup), 70
MD_restrained() (in module gromacs.setup), 69
MDP (class in gromacs.fileformats.mdp), 34
mean (gromacs.fileformats.xvg.XVG attribute), 30
merge_ndx() (in module gromacs.tools), 45
min (gromacs.fileformats.xvg.XVG attribute), 30
MissingDataError, 9
MissingDataWarning, 9
Molecule (class in gromacs.fileformats.blocks), 38

N

NDX (class in gromacs.fileformats.ndx), 35
ndxlist (gromacs.fileformats.ndx.NDX attribute), 36
NonbondedParamType (class in gromacs.fileformats.blocks), 39
number_pdbs() (in module gromacs.utilities), 43

O

openany() (in module gromacs.utilities), 42
outfile() (gromacs.cbook.Transformer method), 56

P

Param (class in gromacs.fileformats.blocks), 38
parse() (gromacs.fileformats.xpm.XPM method), 33
parse() (gromacs.fileformats.xvg.XVG method), 30
parse_ndxlist() (in module gromacs.cbook), 58, 63
ParseError, 9
partial_tempering() (in module gromacs.scaling), 73
PATH, 1, 19, 42
path (in module gromacs.config), 16
plot() (gromacs.fileformats.xvg.XVG method), 31
plot_coarsened() (gromacs.fileformats.xvg.XVG method), 31
Popen() (gromacs.core.Command method), 14
Popen() (gromacs.core.GromacsCommand method), 13
PopenWithInput (class in gromacs.core), 15
prop() (gromacs.environment.Flag method), 23

Q

qscript_template (in module gromacs.config), 21
qscriptdir (in module gromacs.config), 21
queuing_systems (in module gromacs.qsub), 77
QueuingSystem (class in gromacs.qsub), 75

R

read() (gromacs.fileformats.mdp.MDP method), 35
read() (gromacs.fileformats.ndx.NDX method), 36
read() (gromacs.fileformats.xpm.XPM method), 33
read() (gromacs.fileformats.xvg.XVG method), 32
realpath() (in module gromacs.utilities), 42
register() (gromacs.environment.Flags method), 22
remove_legend() (in module gromacs.utilities), 43
renumber_atoms() (gromacs.fileformats.blocks.Molecule method), 38
rmsd_backbone() (in module gromacs.cbook), 51
rp() (gromacs.cbook.Transformer method), 56
run() (gromacs.core.Command method), 15
run() (gromacs.core.GromacsCommand method), 13

S

scale_dihedrals() (in module gromacs.scaling), 72
scale_impropers() (in module gromacs.scaling), 73
set() (gromacs.fileformats.ndx.NDX method), 36
set() (gromacs.fileformats.xvg.XVG method), 32
set_correlparameters() (gromacs.fileformats.xvg.XVG method), 32
set_gmxrc_environment() (in module gromacs.config), 19
setdefault() (gromacs.environment.Flags method), 22
setdefault() (gromacs.fileformats.ndx.NDX method), 36
SettleType (class in gromacs.fileformats.blocks), 39
setup() (in module gromacs.config), 16
size() (gromacs.fileformats.ndx.NDX method), 36
sizes (gromacs.fileformats.ndx.NDX attribute), 36
solvate() (in module gromacs.setup), 67
std (gromacs.fileformats.xvg.XVG attribute), 32
strip_fit() (gromacs.cbook.Transformer method), 56

strip_water() (gromacs.cbook.Transformer method), 57
System (class in gromacs.fileformats.blocks), 38
SystemToGroTop (class in gromacs.fileformats.top), 37

T

tc (gromacs.fileformats.xvg.XVG attribute), 32
templates (in module gromacs.config), 21
templatesdir (in module gromacs.config), 21
Timedelta (class in gromacs.utilities), 41
to_unicode() (in module gromacs.fileformats.convert), 40
tool_factory() (in module gromacs.tools), 45
TOP (class in gromacs.fileformats.top), 37
topology() (in module gromacs.setup), 67
transform_args() (gromacs.core.Command method), 15
transform_args() (gromacs.core.GromacsCommand method), 13
Transformer (class in gromacs.cbook), 54
trj_compact() (in module gromacs.cbook), 51
trj_fitandcenter() (in module gromacs.cbook), 51
trj_xyfitted() (in module gromacs.cbook), 51

U

uncomment() (gromacs.fileformats.xpm.XPM static method), 33
uniqueNDX (class in gromacs.fileformats.ndx), 36
unlink_f() (in module gromacs.utilities), 43
unlink_gmx() (in module gromacs.utilities), 43
unlink_gmx_backups() (in module gromacs.utilities), 43
unquote() (gromacs.fileformats.xpm.XPM static method), 33
update() (gromacs.environment.Flags method), 22
UsageWarning, 10

V

values() (gromacs.environment.Flags method), 22
vdw_lipid_atom_radii (in module gromacs.setup), 72
vdw_lipid_resnames (in module gromacs.setup), 72
VirtualSites3Type (class in gromacs.fileformats.blocks), 39

W

which() (in module gromacs.utilities), 42
withextsep() (in module gromacs.utilities), 42
write() (gromacs.fileformats.mdp.MDP method), 35
write() (gromacs.fileformats.ndx.NDX method), 36
write() (gromacs.fileformats.top.TOP method), 37
write() (gromacs.fileformats.xvg.XVG method), 32

X

XPM (class in gromacs.fileformats.xpm), 33
xvalues (gromacs.fileformats.xpm.XPM attribute), 33
XVG (class in gromacs.fileformats.xvg), 26

Y

yvalues (gromacs.fileformats.xpm.XPM attribute), 33