PyCGTOOL Documentation

Release 0.1

James Graham

Jun 15, 2017
# Contents

1 PyCGTOOLS Tutorial 1
   1.1 Atomistic Simulation 1
   1.2 Mapping Design 1
   1.3 Model Generation 2
   1.4 Running the CG Simulation 3
   1.5 Model Validation 3

2 Mapping Only Mode 5

3 pycgtools 7
   3.1 pycgtools package 7

4 Features 19

5 Requirements 21

6 Basic Usage 23

7 Mapping / Bond Definition Files 25
   7.1 Mapping Definition 25
   7.2 Bond Definition 26

8 Advanced Usage 27
   8.1 Modes 27
   8.2 Advanced Options 27

9 Indexes 29

Python Module Index 31
This tutorial follows the complete process of parametrising a new molecule within the MARTINI forcefield, covering aspects of mapping design, model generation and model validation. PyCGTOOL is used at multiple stages, showing its use in several different situations. All files required to follow this tutorial are present in the doc/tutorial_files directory.

The molecule chosen as a target for this parametrisation is the $\beta_1$ antagonist atenolol.

**Atomistic Simulation**

The reference simulation for the parametrisation of atenolol was performed using the GROMOS 54A7 united atom forcefield with a topology from the ATB database. A single molecule of atenolol was solvated and equilibrated, before collecting a 50 ns trajectory using the GROMACS molecular dynamics simulator. A reduced copy of this trajectory is provided in the tutorial files (as ref.xtc, ref.gro contains the initial structure) since the original is prohibitively large.

**Mapping Design**

Designing a suitable mapping from the atomistic to the coarse-grained representation requires some experience and a degree of chemical intuition, but the ease with which the mapping may be modified using PyCGTOOL allows the mapping to be iterated much more quickly than if the construction of the CG model were to be performed manually.

The process of designing a mapping involves splitting the molecule into fragments, each of which contains approximately four heavy atoms. Start by finding functional groups such as amides or carboxylates, each of which may become a single bead. Next, replace phenyl rings with a triangle of three small type beads, each of which contains two heavy atoms and has reduced Lennard-Jones size and mass, as compared to the normal four-atom beads. Finally, divide any remaining parts of the molecule into beads of four heavy atoms as required. The ideal bead will contain four heavy atoms and be nearly spherical, but this is not always possible. If any atoms remain after clusters of four have been allocated, it may be required to use a mapping of three-to-one for some beads.
After the atoms have been allocated to beads, determine which beads should be bonded by inspection of the atomistic structure; bonds between functional groups in the atomistic structure become bonds between beads in the CG structure. It will probably be the case that there is no obviously best mapping and bond topology, which is not at this point a major issue as multiple mappings can be assessed easily.

Once the mapping and bond topology have been determined, they must be put into a format readable by PyCGTOOL. This format is as described in the introduction to PyCGTOOL. The mapping file must contain a molecule header which is the residue name enclosed in square brackets. Each subsequent line represents a single CG bead and is of the structure (where items in square brackets are optional):

```
<bead name> <MARTINI type> [<bead charge>] <component atom 1> [<component atom 2> ...
```

The bonding file has a similar structure with a molecule header followed by lines defining each bond in the format:

```
<bead 1> <bead 2>
```

It is not necessary to define angles since PyCGTOOL is able to infer these from the bond network. If only a subset of the possible angles require parameters then they may be listed, preventing automatic enumeration of all possible angles.

The mapping used in the paper to describe atenolol (just one of the several possible mappings) is described by the following input files.

atenolol.map:

```
[36KB]
N1 P1 N1 C12 C13 C14
O1 P1 O3 C1 C2 C3
O2 SP3 O2 C4
C1 SC3 C5 C6
C2 SC3 C8 C9
C3 SC3 C7 C10
N2 P5 C11 O1 N2
```

atenolol.bnd:

```
[36KB]
N1 O1
O1 O2
O2 C1
O2 C2
C1 C2
C1 C3
C2 C3
C3 N2
```

## Model Generation

The process of model generation after having created the mapping and bond definition files is automated by PyCGTOOL. In the simplest case, a parameter set may be generated simply by passing the four input files to PyCGTOOL:

```
pycgtool.py -g ref.gro -x ref.xtc -m atenolol.map -b atenolol.bnd
```

This will create two output files `out.gro`, the mapped CG coordinates, and `out.itp`, the calculated CG model parameters.
**Running the CG Simulation**

Note: These instructions assume the use of GROMACS 5.0 or newer.

From this stage the usual preparation of MARTINI/GROMACS simulations applies. The output coordinates `out.gro` must be solvated using the GROMACS tool `gmx solvate` with the options:

```
gmx solvate -cp out.gro -cs ../../data/water.gro -o solv.gro -radius 0.21
```

Since MARTINI water cannot be automatically added to the `.top` file, this must be done manually. A template file, `template.top`, is provided. Copy this to `topol.top` and add the line “W 251” to the bottom, since 251 should be the number of water molecules added by `gmx solvate`.

The three stages of simulation: minimisation, equilibration, and production may then be run:

```
gmx grompp -f em.mdp -c solv.gro -p topol.top -o em.tpr
gmx mdrun -deffnm em

gmx grompp -f npt.mdp -c em.gro -p topol.top -o npt.tpr
gmx mdrun -deffnm npt -v

gmx grompp -f md.mdp -c em.gro -p topol.top -o md.tpr
gmx mdrun -deffnm md -v
```

These simulations should take a few minutes on a modern desktop.

**Model Validation**

It is recommended to perform validation before using a generated CG model for production simulations, so that we may have confidence in its ability to replicate the behaviour of the molecule being studied. The methods of validation applied in the PyCGTOOL paper are a comparison of the distribution of bonded term measurements between the CG test simulation and the atomistic reference simulation, and a comparison of the radius of gyration between these two simulations. Additionally, other methods of validation should be applied relevant to the class of molecule being studied; for instance, validation of membrane lipids should compare the membrane thickness and surface area per lipid to the atomistic reference.

To compare the distribution of bonded terms, we must first rerun PyCGTOOL to generate samples of the bonded measurements. For the atomistic reference simulation, this can be done by running:

```
pycgtool.py -g ref.gro -x ref.xtc -m atenolol.map -b atenolol.bnd --advanced
```

In the menu, set the advanced option `dump_measurements` to `True` by selecting it with the arrow keys and toggling with the enter key. Once this option has been set, continue by pressing the q key. PyCGTOOL will now output a sample of each measured bond length and angle (but since the reference trajectory is short, the target sample size is not met and all values are collected), in the files `36KB_length.dat` and `36KB_angle.dat`.

Since we will be collecting samples of the same measurements from the CG simulation, these files should be renamed to, for instance, `ref_length.dat` and `ref_angle.dat`. Collect the same samples for the CG simulation using:

```
pycgtool.py -g md.gro -x md.xtc -b atenolol.bnd
```

Since we provide a bond file, but not a mapping file, PyCGTOOL will know that this is intended to simply collect bond measurements and will automatically set the `dump_measurements` option to `True`. Again, the files created will be called `36KB_length.dat` and `36KB_angle.dat`.

1.4. Running the CG Simulation 3
These samples were compared in the paper using an R script to generate a series of boxplots, but a simpler Python script is provided which may be used to compare the mean and standard deviations of the samples:

```
./average_columns.py ref_length.dat 36KB_length.dat
./average_columns.py ref_angle.dat 36KB_angle.dat
```

If the automatically generated parameters provide an accurate representation of the reference structure, the percentage error between the two samples will be small.

Validation of the more general molecular conformation may be performed by comparison of the radius of gyration of the reference and CG models. This may be performed using the standard GROMACS tool `gmx gyrate`:

```
gmx gyrate -f ref.xtc -s ref.tpr -o ref-gyr.xvg
```

In both cases select the 36KB group as the one on which to perform the calculation. These commands will calculate the radius of gyration for each trajectory frame for both the reference and CG simulations. The resulting `.xvg` files may be visualised using a graphing program such as `xmgrace` or compared in the same way as the bonded samples, using:

```
./average_columns.py ref-gyr.xvg cg-gyr.xvg
```

As before, a small percentage difference in each of the columns suggests good replication of gross conformation.

In addition to these simple forms of validation, it is recommended that further validation, relevant to the class of molecule, is performed. In the case of membrane lipids, for instance, this may take the form of an assessment of membrane thickness and surface area per lipid.
CHAPTER 2

Mapping Only Mode

PyCGTOOL may be used in mapping only mode to simply convert an all-atom (AA) coordinate file into its coarse-grained (CG) representation. In this respect it functions similarly to the MARTINI tool Martinize. The main uses of the functionality are:

- Enable direct comparison between AA and CG simulations
- Use a pre-equilibrated AA coordinate file as the starting point for a CG simulation

In order to perform the AA to CG mapping, a PyCGTOOL mapping file is required. For guidance on the creation of this file see the full PyCGTOOL Tutorial.

To perform a mapping from a single snapshot .gro file use the command:

```
pycgtool.py -g <GRO file> -m <MAP file>
```

To perform a mapping of a complete .xtc trajectory use the command:

```
pycgtool.py -g <GRO file> -x <XTC file> -m <MAP file>
```

But note that mapping an entire trajectory requires that the optional dependency MDTraj is installed.
pycgtool package

Subpackages

pycgtool.parsers package

Submodules

pycgtool.parsers.cfg module

Module containing classes used to parse custom CFG file format.
Format is based upon GROMACS .itp files but does not support nesting of sections.

class pycgtool.parsers.cfg.CFG (filename=None)
   Bases: collections.OrderedDict
   Class representing a CFG file.
   Contains a dictionary of Sections.
   filename

exception pycgtool.parsers.cfg.DuplicateSectionError (section, filename)
   Bases: KeyError
   Exception used to indicate that a section has appeared twice in a file.

exception pycgtool.parsers.cfg.NoSectionError (filename)
   Bases: KeyError
   Exception used to indicate that a file contains no sections.
Module contents

Submodules

pycgtool.bondset module

Module containing classes to calculate bonded properties from a Frame.
BondSet contains a dictionary of lists of Bonds. Each list corresponds to a single molecule.

```python
class pycgtool.bondset.Bond(atoms, atom_numbers=None, func_form=None)
    Bases: object
    Class holding the properties of a single bonded term.
    Bond lengths, angles and dihedrals are all equivalent, distinguished by the number of atoms present.

    atom_numbers
    atoms

    boltzmann_invert(temp=310)
        Perform Boltzmann Inversion using measured values of bond to calculate equilibrium value and force constant.

        Parameters
        temp -- Temperature at which the simulation was performed

    eqm
    fconst

    gromacs_type_id

    values

class pycgtool.bondset.BondSet(filename, options)
    Bases: object
    Class used to perform bond measurements in a Frame.
    BondSet contains a dictionary of lists of Bonds. Each list corresponds to a single molecule.

    apply(frame)
        Calculate bond lengths/angles for a given Frame and store into Bonds.

        Parameters
        frame -- Frame from which to calculate values

    boltzmann_invert(progress=False)
        Perform Boltzmann Inversion of all bonds to calculate equilibrium value and force constant.

        Parameters
        progress -- Display a progress bar using tqdm if available

    dump_values(target_number=10000)
        Output measured bond values to files for length, angles and dihedrals.

        Parameters
        target_number -- Approx number of sample measurements to output. If None, all samples will be output

    get_bond_angles(mol, exclude_triangle=True)
        Return list of all bond angles in molecule.

        Parameters

    • mol -- Molecule name to return bonds for

    • exclude_triangle -- Exclude angles that are part of a triangle?
Returns List of bonds

get_bond_dihedrals (mol)
Return list of all bond dihedrals in molecule.

Parameters mol – Molecule name to return bonds for

Returns List of bonds

get_bond_length_constraints (mol)
Return list of all bond length constraints in molecule.

Parameters mol – Molecule name to return bonds for

Returns List of bonds

get_bond_lengths (mol, with_constr=False)
Return list of all bond lengths in molecule. May include constraints.

Parameters

• mol – Molecule name to return bonds for

• with_constr – Include constraints?

Returns List of bonds

write_itp (filename, mapping)
Output a GROMACS .itp file containing atoms/beads and bonded terms.

Parameters

• filename – Name of output file

• mapping – AA->CG Mapping from which to collect bead properties

pycgtool.forcefield module

This module contains a single class ForceField used to output a GROMACS .ff forcefield.

class pycgtool.forcefield.ForceField (name)
Bases: object

Class used to output a GROMACS .ff forcefield

write (filename, mapping, bonds)

pycgtool.frame module

This module contains classes for storing atomic data.

The Frame class may contain multiple Residues which may each contain multiple Atoms. Both Frame and Residue
are iterable. Residue is indexable with either atom numbers or names.

class pycgtool.frame.Atom (name, num, type=None, mass=None, charge=None, coords=None)
Bases: object

Hold data for a single atom

add_missing_data (other)

charge

coords
mass
name
num
type
class pycgtool.frame.Frame (gro=None, xtc=None, itp=None, frame_start=0, xtc_reader='simpletraj')
Bases: object
Hold Atom data separated into Residues

add_residue(residue)
Add a Residue to this Frame

    Parameters residue -- Residue to add

classmethod instance_from_reader(reader)
Return Frame instance initialised from existing FrameReader object

    Parameters reader (FrameReader) -- FrameReader object

    Returns Frame instance

next_frame()
Read next frame from input XTC.

    Returns True if successful else False

output(filename, format='gro')
Write coordinates from Frame to file.

    Parameters

    * filename -- Name of file to write to
    * format -- Format to write e.g. 'gro', 'lammps'

write_xtc(filename)
Write frame to output XTC file.

    Parameters filename -- XTC filename to write to

yield_resname_in(container)
class pycgtool.frame.Residue (name=None, num=None)
Bases: object
Hold data for a residue - list of atoms

add_atom(atom)
Add an Atom to this Residue and store location in index

    Parameters atom -- Atom to add to Residue

    Returns None

atoms
name
name_to_num
num
pycgtool.functionalforms module

class pycgtool.functionalforms.CosHarmonic
    Bases: pycgtool.functionalforms.FunctionalForm
        static fconst(values, temp)
        gromacs_type_ids = (None, 2, None)

class pycgtool.functionalforms.FunctionalForm
    Bases: object
    Parent class of any functional form used in Boltzmann Inversion to convert variance to a force constant.
    New functional forms must define a static __call__ method.

    static eqm(values, temp)
        Calculate equilibrium value. May be overridden by functional forms.
        Parameters
            • values – Measured internal coordinate values from which to calculate equilibrium value
            • temp – Temperature of simulation
        Returns Calculated equilibrium value

    static fconst(values, temp)
        Calculate force constant. Abstract static method to be defined by all functional forms.
        Parameters
            • values – Measured internal coordinate values from which to calculate force constant
            • temp – Temperature of simulation
        Returns Calculated force constant

    classmethod gromacs_type_id_by_natoms(natoms)
        Return the GROMACS potential type id for this functional form when used with natoms.
        Parameters natoms (int) –
        Return int GROMACS potential type id

    gromacs_type_ids
        Return tuple of GROMACS potential type ids when used as length, angle, dihedral.
        Return tuple[int] Tuple of GROMACS potential type ids

class pycgtool.functionalforms.FunctionalForms(**kwargs)
    Bases: object
    Class holding list of all defined functional forms for Boltzmann Inversion.
    Creating an instance causes the Enum of functional forms to be updated with all new subclasses of FunctionalForm. These may then be accessed by name, either as attributes or using square brackets.

    FormsEnum = <pycgtool.util.FormsEnum object>

class pycgtool.functionalforms.Harmonic
    Bases: pycgtool.functionalforms.FunctionalForm
        static fconst(values, temp)
        gromacs_type_ids = (1, 1, 1)
class pycgtool.functionalforms.MartiniDefaultAngle
    Bases: pycgtool.functionalforms.FunctionalForm
    static fconst (values, temp)
gromacs_type_ids = (None, 2, None)
class pycgtool.functionalforms.MartiniDefaultDihedral
    Bases: pycgtool.functionalforms.FunctionalForm
    static fconst (values, temp)
gromacs_type_ids = (None, None, 1)
class pycgtool.functionalforms.MartiniDefaultLength
    Bases: pycgtool.functionalforms.FunctionalForm
    static fconst (values, temp)
gromacs_type_ids = (1, None, None)

pycgtool.interface module

This module contains classes for interaction at the terminal.
class pycgtool.interface.Options (default, args=None)
    Bases: object
    Class to hold program options not specified at the initial command line.
    Values can be queried by indexing as a dictionary or by attribute. Iterable.
    interactive ()
        Read options in interactive terminal mode using curses.
    set (opt, val)
        Set an argument by name.
    Parameters
        • opt – Option to set
        • val – Value to set option to
toggle_boolean (opt)
        Toggle a boolean argument by name.
    Parameters opt – Option to toggle
class pycgtool.interface.Progress (maxits, length=20, dowhile=None, quiet=False)
    Bases: object
    Display a progress bar during the main loop of a program.
    run ()
        Iterate through self until stopped by maximum iterations or False condition.
        Use the tqdm library if it is present.

pycgtool.mapping module

This module contains classes required to perform an atomistic to coarse-grain mapping.
The Mapping class contains a dictionary of lists of BeadMaps. Each list corresponds to a single molecule.
class pycgtool.mapping.BeadMap (name, num, type=None, atoms=None, charge=0, mass=0)
    Bases: pycgtool.frame.Atom

POD class holding values relating to the AA->CG transformation for a single bead.

atoms
charge
mass
name
type
weights
weights_dict

exception pycgtool.mapping.EmptyBeadError
    Bases: Exception

Exception used to indicate that none of the required atoms are present.

class pycgtool.mapping.Mapping (filename, options, itp=None)
    Bases: object

Class used to perform the AA->CG mapping.

Contains a dictionary of lists of BeadMaps. Each list corresponds to a single molecule.

apply (frame, cgframe=None)
    Apply the AA->CG mapping to an atomistic Frame.

    Parameters
    • frame – Frame to which mapping will be applied
    • cgframe – CG Frame to remap - optional

    Returns Frame instance containing the CG frame

pycgtool.mapping.calc_coords_weight (ref_coors, coords, box, weights)
    Calculate the coordinates of a single CG bead from weighted component atom coordinates.

    Parameters
    • ref_coors – Coordinates of reference atom, usually first atom in bead
    • coords – Array of coordinates of component atoms
    • box – PBC box vectors
    • weights – Array of atom weights, must sum to 1

    Returns Coordinates of CG bead

pycgtool.mapping.calc_coords_weight_nobox (ref_coors, coords, box, weights)
    Calculate the coordinates of a single CG bead from weighted component atom coordinates.

    Parameters
    • ref_coors – Coordinates of reference atom, usually first atom in bead
    • coords – Array of coordinates of component atoms
    • box – PBC box vectors
    • weights – Array of atom weights, must sum to 1
Returns Coordinates of CG bead

**pycgtool.pycgtool module**

`pycgtool.pycgtool.main(args, config)`  
Main function of the program PyCGTOOL.  
Performs the complete AA->CG mapping and outputs a files dependent on given input.  

**Parameters**  
- `args` – Arguments from argparse  
- `config` – Configuration dictionary

`pycgtool.pycgtool.map_only(args, config)`  
Perform AA->CG mapping and output coordinate file.  

**Parameters**  
- `args` – Program arguments  
- `config` – Object containing run options

**pycgtool.util module**

This module contains some general purpose utility functions used in PyCGTOOL.

```python
class pycgtool.util.FixedFormatUnpacker(format_string, format_style=<FormatStyle: C>)
    Bases: object

    Unpack strings printed in fixed format.

class FormatItem(type, width)
    Bases: tuple

    type
        Alias for field number 0

    width
        Alias for field number 1

FixedFormatUnpacker.FormatStyle = <pycgtool.util.FormatStyle object>
FixedFormatUnpacker.unpack(string)
```

```python
class pycgtool.util.NumbaDummy
    Bases: object

    Dummy Numba module

    static jit(*args, **kwargs)
        Dummy version of numba.jit decorator, does nothing
```

```python
class pycgtool.util.SimpleEnum
    Bases: object

    class Enum
        Bases: object

        as_dict()

    class SimpleEnum.EnumItem(enum_name, key, value=None)
        Bases: object
```

14 Chapter 3. pycgtool
**compare_value** *(other)*

Compare enums by value rather than by key.

**Parameters**
- **other** – RHS enum

**Returns**
Whether EnumValues are equal by value

**classmethod** `SimpleEnum.enum` *(name, keys=None, values=None)*

**classmethod** `SimpleEnum.enum_from_dict` *(name, key_val_dict)*

`pycgtool.util.backup_file` *(name)*

Backup a file using the GROMACS backup naming scheme. name -> #name.x#

**Parameters**
- **name** – File to backup

**Returns**
New name of file after backup

`pycgtool.util.cmp_whitespace_float` *(ref_filename, test_filename, float_rel_error=0.01)*

Compare two files ignoring spacing on a line and using a tolerance on floats

**Parameters**
- **ref_filename** – Name of reference file
- **test_filename** – Name of test file
- **float_rel_error** – Acceptable relative error on floating point numbers

**Returns**
True if files are the same, else False

`pycgtool.util.dir_up` *(name, n=1)*

Return the directory path n levels above a specified file/directory.

**Parameters**
- **name** – Name of file/directory to start from
- **n** – Number of directory levels to go up

**Returns**
Directory n directories above name

`pycgtool.util.dist_with_pbc` *(pos1, pos2, box)*

Calculate the distance between two points accounting for periodicity.

**Parameters**
- **pos1** – 3d vector position 1
- **pos2** – 3d vector position 2
- **box** – Cubic box vectors

**Returns**
Vector between two points

`pycgtool.util.extend_graph_chain` *(extend, pairs)*

Take list of tuples representing chained links in an undirected graph and extend the chain length.

**Parameters**
- **extend** – List of link tuples to extend
- **pairs** – Graph edges as list of tuples

**Returns**
List of link tuples for chain length one greater than input

`pycgtool.util.numba`

Dummy Numba module
pycgtool.util.once_wrapper(func)
Wrap a function such that it runs only once, subsequent calls are ignored.

Parameters
- **func** – Function to wrap

Returns
- Wrapped function which will only run once.

pycgtool.util.r_squared(ref, fit)
Calculate residual R squared of fitted data against reference data by y values.

Parameters
- **ref** – Reference y values
- **fit** – Fitted y values

Returns
- R squared

pycgtool.util.sliding(vals)
Yield three values in a sliding window along an iterable.

Parameters
- **vals** – Iterable to iterate over

Returns
- Generator of tuples

pycgtool.util.stat_moments(vals, ignore_nan=True)
Return statistical (population) moments of data provided.

Parameters
- **vals** – The data for which to calculate moments
- **ignore_nan** – Whether to exclude np.nan and infinity from calculation

Returns
- Tuple of moments - population mean and variance, both zero if input is empty list

pycgtool.util.tqdm_dummy(iterable, **kwargs)

pycgtool.util.transpose_and_sample(sequence, n=None)
Transpose a sequence of lists and sample to provide target number of rows.

Parameters
- **sequence** – 2d sequence object to transpose
- **n** – Number of samples to take

pycgtool.util.tuple_equivalent(tuple1, tuple2)
Check if two node tuples are equivalent. Assumes undirected edges.

Parameters
- **tuple1** – First tuple to compare
- **tuple2** – Second tuple to compare

Returns
- True if tuples are equivalent, else False

pycgtool.util.vector_angle(a, b)
Calculate the angle between two vectors.

Parameters
- **a** – First vector
- **b** – Second vector

Returns
- Angle in radians

pycgtool.util.vector_angle_signed(a, b, ref=array([ 0., 0., 1.]))
Calculate the signed angle between two vectors.

Parameters
- **a** – First vector
• \( \mathbf{b} \) – Second vector
• \( \mathbf{ref} \) – Optional reference vector, will use global z-axis if not provided

**Returns**  Signed angle in radians

```python
pycgtool.util.vector_cross(u, v)
```
Return vector cross product of two 3d vectors as numpy array.

**Parameters**

• \( u \) – First 3d vector
• \( v \) – Second 3d vector

**Returns**  Cross product of two vectors as numpy.array

```python
pycgtool.util.vector_dot(u, v)
```
Return vector dot product of two 3d vectors.

**Parameters**

• \( u \) – First 3d vector
• \( v \) – Second 3d vector

**Returns**  Dot product of two vectors

```python
pycgtool.util.vector_len(v)
```

**Module contents**

*PyCGTOOL Tutorial*
PyCGTOOL provides a means to quickly and easily generate coarse-grained molecular dynamics models within the MARTINI framework from all-atom or united-atom simulation trajectories.

A user-defined mapping is applied to the input trajectory and bonded terms (lengths, angles and dihedrals) are measured. From these measurements, equilibrium values and force constants are found and a GROMACS topology is created for the target molecules.
PyCGTOOL requires:

- Python 3
- Numpy (http://www.numpy.org/)
- simpletraj (https://github.com/arose/simpletraj)

Optional:

- MDTraj for pseudo-CG XTC output (http://mdtraj.org/1.7.2/) with own dependencies:
  - Scipy (https://www.scipy.org/)
  - Cython (http://cython.org/)
- Python testing framework (e.g. Nose2, py.test)
- Numba for increased performance (http://numba.pydata.org/)
- Sphinx to generate documentation yourself (http://www.sphinx-doc.org/en/stable/)
PyCGTOOL requires four input files to generate a coarse-grained model:

- 
  -g GROMACS gro coordinate file
- 
  -x GROMACS xtc trajectory file
- 
  -m PyCGTOOL mapping definition file
- 
  -b PyCGTOOL bond definition file

The program is called by:

```
pycgtool.py -g <GRO file> -x <XTC file> -m <MAP file> -b <BND file>
```

Example mapping and bond definition files are present in the `test/data` directory. Their format is explained below.

After running PyCGTOOL two files, `out.gro` and `out.itp` will be created. The gro file contains the mapped coarse-grain coordinates with every molecule for which a mapping was provided. The itp file contains the parameters for each molecule type.

It is important to perform validation of any new parameter set. This is typically done by comparing properties between the reference simulation and simulations using the new CG model. In the tutorial we use the radius of gyration, but there are many other suitable properties, depending on the class of molecule being parametrised.
The mapping and bond definition input files use a format similar to the GROMACS itp/top format.

**Mapping Definition**

An example of mapping definition file for the monosaccharide allose taken from test/data/sugar.map is shown below.

Molecule names (as present in the gro coordinate file) are used as section headers inside square brackets. Each of the following lines describes a single coarse-grained bead mapping. The items on a line are: the name of the bead, its MARTINI bead type, optionally the bead charge, and a list of all the atom names it should contain. All items on a line are whitespace separated. Multiple molecules may be specified in their own sections. It is not recommended to provide a mapping for water since MARTINI water combines four molecules into a single bead which is not yet supported by PyCG TOOL. Note that bead charges in the MARTINI framework are by convention integers and are used only for formally charged functional groups. An example of a molecule mapping using charges can be found in test/data/dppc.map.

```plaintext
; comments begin with a semicolon
[ALLA]
C1 P3 C1 O1
C2 P3 C2 O2
C3 P3 C3 O3
C4 P3 C4 O4
C5 P2 C5 C6 O6
O5 P4 O5

[SOL]
W P4 OW HW1 HW2
```
Bond Definition

An example bond definition file for the monosaccharide allose taken from `test/data/sugar.bnd` is shown below. As in the mapping definition file, molecule names are used as section headers inside square brackets. The following lines define bonds lengths, angles and dihedrals between coarse-grained beads. Each line is a list of bead names, using the names defined in the mapping file. Two bead names on a line defines a bond length, three defines an angle, and four defines a dihedral.

If no angles are defined for a molecule, PyCGTOOL will construct all angles from the list of bonds. This may also be enabled for dihedrals via the `--advanced` flag, but is not recommended as in most cases coarse-grained models do not require dihedrals. Additionally, any angles inside a triangle of bond lengths are excluded from the output as they often cause simulation stability issues when used in conjunction with LINCS.

```plaintext
; comments begin with a semicolon
[ALLA]
C1 C2
C2 C3
C3 C4
C4 C5
C5 O5
O5 C1
C1 C2 C3
C2 C3 C4
C3 C4 C5
C4 C5 O5
C5 O5 C1
O5 C1 C2
C1 C2 C3 C4
C2 C3 C4 C5
C3 C4 C5 O5
C4 C5 O5 C1
C5 O5 C1 C2
O5 C1 C2 C3
```
CHAPTER 8

Advanced Usage

Modes

PyCGTOOL performs several other functions which may be useful in the testing and use of coarse-grained models.

Mapping Only

Mapping-only mode simply converts an input atomistic coordinate file into its coarse-grained representation. For full detail see: Mapping Only Mode.

This mode may be invoked by:

```
pycgtool.py -g <GRO file> -m <MAP file>
```

Measure Only

Measure-only mode may be used to aid in the testing of a coarse-grained model by making measurements of bonds from a true coarse-grained simulation trajectory. These bond measurements may be compared directly to those collected from the pseudo-coarse-grained trajectory used to generate the model. This mode may be invoked by:

```
pycgtool.py -g <GRO file> -x <XTC file> -b <BND file>
```

Advanced Options

By passing the flag --advanced to PyCGTOOL several advanced options are accessible. The arrow keys may be used to navigate through the menu. Enter selects an option to be edited, or if the option is boolean toggles it. Once you have edited an option press enter again. When all options are satisfactory, press q to proceed.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_name</td>
<td>Base name of output files</td>
<td>out, any string</td>
</tr>
<tr>
<td>output</td>
<td>Coordinate output format</td>
<td>gro</td>
</tr>
<tr>
<td>output_xtc</td>
<td>Should a pseudo-CG XTC be created</td>
<td>False, True</td>
</tr>
<tr>
<td>map_only</td>
<td>Run in mapping-only mode</td>
<td>False, True</td>
</tr>
<tr>
<td>map_center</td>
<td>Mapping method</td>
<td>geom, mass</td>
</tr>
<tr>
<td>constr_threshold</td>
<td>Convert stiff bonds to constraints</td>
<td>100000, any number</td>
</tr>
<tr>
<td>dump_measurements</td>
<td>Whether to output bond measurements</td>
<td>False, True</td>
</tr>
<tr>
<td>dump_n_values</td>
<td>How many measurements to output</td>
<td>100000, any number</td>
</tr>
<tr>
<td>output_forcefield</td>
<td>Output a GROMACS forcefield directory?</td>
<td>False, True</td>
</tr>
<tr>
<td>temperature</td>
<td>Temperature of reference simulation</td>
<td>310, any number</td>
</tr>
<tr>
<td>default_fc</td>
<td>Use default MARTINI force constants?</td>
<td>False, True</td>
</tr>
<tr>
<td>generate_angles</td>
<td>Generate angles from bonds</td>
<td>True, False</td>
</tr>
<tr>
<td>generate_dihedrals</td>
<td>Generate dihedrals from bonds</td>
<td>False, True</td>
</tr>
</tbody>
</table>
CHAPTER 9

Indexes

- genindex
- modindex
- search
Python Module Index

p
pycgtool, 17
pycgtool.bondset, 8
pycgtool.forcefield, 9
pycgtool.frame, 9
pycgtool.functionalforms, 11
pycgtool.interface, 12
pycgtool.mapping, 12
pycgtool.parsers, 8
pycgtool.parsers.cfg, 7
pycgtool.pycgtool, 14
pycgtool.util, 14
| A | add_atom() (pycgtool.frame.Residue method), 10 |
|   | add_missing_data() (pycgtool.frame.Atom method), 9 |
|   | add_residue() (pycgtool.frame.Frame method), 10 |
|   | apply() (pycgtool.bondset.BondSet method), 8 |
|   | apply() (pycgtool.mapping.Mapping method), 13 |
|   | as_dict() (pycgtool.util.SimpleEnum.Enum method), 14 |
|   | Atom (class in pycgtool.frame), 9 |
|   | atom_numbers (pycgtool.bondset.Bond attribute), 8 |
|   | atoms (pycgtool.bondset.Bond attribute), 8 |
|   | atoms (pycgtool.frame.Residue attribute), 10 |
|   | atoms (pycgtool.mapping.BeadMap attribute), 13 |
| B | backup_file() (in module pycgtool.util), 15 |
|   | BeadMap (class in pycgtool.mapping), 12 |
|   | boltzmann_invert() (pycgtool.bondset.Bond method), 8 |
|   | boltzmann_invert() (pycgtool.bondset.BondSet method), 8 |
|   | Bond (class in pycgtool.bondset), 8 |
|   | BondSet (class in pycgtool.bondset), 8 |
| C | calc_coords_weight() (in module pycgtool.mapping), 13 |
|   | calc_coords_weight_nobox() (in module pycgtool.mapping), 13 |
|   | CFG (class in pycgtool.parsers.cfg), 7 |
|   | charge (pycgtool.frame.Atom attribute), 9 |
|   | charge (pycgtool.mapping.BeadMap attribute), 13 |
|   | cmp_whitespace_float() (in module pycgtool.util), 15 |
|   | compare_value() (pycgtool.util.SimpleEnum.EnumItem method), 14 |
|   | coords (pycgtool.frame.Atom attribute), 9 |
|   | CosHarmonic (class in pycgtool.functionalforms), 11 |
| D | dir_up() (in module pycgtool.util), 15 |
|   | dist_with_pbc() (in module pycgtool.util), 15 |
|   | dump_values() (pycgtool.bondset.BondSet method), 8 |

DuplicateSectionError, 7

EmptyBeadError, 13

enum() (pycgtool.util.SimpleEnum class method), 15

enum_from_dict() (pycgtool.util.SimpleEnum class method), 15

eqm (pycgtool.bondset.Bond attribute), 8
eqm() (pycgtool.functionalforms.FunctionalForm static method), 11
extend_graph_chain() (in module pycgtool.util), 15

fconst (pycgtool.bondset.Bond attribute), 8

fconst() (pycgtool.functionalforms.CosHarmonic static method), 11

fconst() (pycgtool.functionalforms.FunctionalForm static method), 11

fconst() (pycgtool.functionalforms.Harmonic static method), 11

fconst() (pycgtool.functionalforms.MartiniDefaultAngle static method), 12

fconst() (pycgtool.functionalforms.MartiniDefaultDihedral static method), 12

fconst() (pycgtool.functionalforms.MartiniDefaultLength static method), 12

filename (pycgtool.parsers.cfg.CFG attribute), 7

FixedFormatUnpacker (class in pycgtool.util), 14

FixedFormatUnpacker.FormatItem (class in pycgtool.util), 14

ForceField (class in pycgtool.forcefield), 9

FormatStyle (pycgtool.util.FixedFormatUnpacker attribute), 14

FormsEnum (pycgtool.functionalforms.FunctionalForms attribute), 11

Frame (class in pycgtool.frame), 10

FunctionalForm (class in pycgtool.functionalforms), 11

FunctionalForms (class in pycgtool.functionalforms), 11
U
unpack() (pycgtool.util.FixedFormatUnpacker method), 14

V
values (pycgtool.bondset.Bond attribute), 8
vector_angle() (in module pycgtool.util), 16
vector_angle_signed() (in module pycgtool.util), 16
vector_cross() (in module pycgtool.util), 17
vector_dot() (in module pycgtool.util), 17
vector_len() (in module pycgtool.util), 17

W
weights (pycgtool.mapping.BeadMap attribute), 13
weights_dict (pycgtool.mapping.BeadMap attribute), 13
width (pycgtool.util.FixedFormatUnpacker.FormatItem attribute), 14
write() (pycgtool.forcefield.ForceField method), 9
write_itp() (pycgtool.bondset.BondSet method), 9
write_xtc() (pycgtool.frame.Frame method), 10

Y
yield_resname_in() (pycgtool.frame.Frame method), 10