Coarse-grained OpenMM Documentation

Release 0.0.1

Garrett A. Meek

Michael R. Shirts

**Dept. of Chemical and Biological Engineering
University of Colorado Boulder**

Sep 20, 2019

CONTENTS

This documentation is generated automatically using Sphinx, which reads all docstring-formatted comments from Python functions in the 'cg_openmm' repository. (See cg_openmm/doc for Sphinx source files.)

CHAPTER

ONE

INSTALLATION NOTES

The cg_openmm package will eventually be available for installation via Anaconda. This will make resolution of software conflicts much easier. However, at present, because the package has not been made public, Anaconda installation is not yet possible, and software conflicts must be resolved by the individual.

Here we provide installation instructions which have been tested on multiple platforms.

1.1 Dependencies for the cg_openmm package

The 'cg_openmm' package was written in order to interface with a separate software package called foldamers. Due to conflicts among dependencies for the foldamers package with other Python versions, **we recommend using Python version 3.6 when installing/using cg_openmm**.

The following is a list of software dependencies for the 'cg_openmm' package, with recommended version numbers in parentheses:

1) foldamers (version 0.0)

    Dependencies for the 'foldamers' software package:

    2) pymbar (version 3.0.3)

    3) MDTraj (version 1.9.3)

    4) MSMBuilder (version 3.8)

    5) scikit-learn (version 0.18.1)

    6) sklearn-genetic (version 0.2)

    7) kHelios

8) OpenMM (version 7.3.1)

9) Yank (version 0.24.1)

1.2 Recommended installation steps

We recommend installation of Anaconda prior to installation of the 'cg_openmm' package, as this makes resolution of conflicts between dependencies much easier.

We direct users that have not installed Anaconda to the Download page in order to select the appropriate version for your platform (Windows, Linux, Mac). (It shouldn't matter which version of Anaconda is installed.)

The following installation steps are recommended for users that have already installed Anaconda on their system:

1) Install 'sklearn-genetic' (used for genetic algorithm optimization of model parameters):

```
pip install sklearn-genetic
```

2) Create an Anaconda environment for Python version 3.6 (the most stable Python version for 'cg_openmm'):

```
conda create -n cg_openmm python=3.6 mdtraj=1.9.3 openmm=7.3.1␣
↪pymbar=3.0.3 msmbuilder=3.8 scikit-learn=0.18.1 yank=0.21.2
```

3) Activate this new environment:

```
conda activate cg_openmm
```

4) Clone and install a fresh copy of 'foldamers':

```
git clone https://github.com/shirtsgroup/foldamers.git
cd foldamers
python setup.py install
```

5) Clone and install a fresh copy of 'cg_openmm' to your system:

```
git clone https://github.com/shirtsgroup/cg_openmm.git
cd cg_openmm
python setup.py install
```

CHAPTER

TWO

BUILDING OPENMM OBJECTS FOR COARSE GRAINED MODELING

All OpenMM simulations require a System() and a Topology(). This chapter details procedures for building those objects for coarse grained models with user-defined properties.

2.1 Building an OpenMM System() for a coarse grained model.

An OpenMM System() object contains force definitions for a molecular model. The 'cg_openmm' repository applies default definitions for all forces (see OpenMM user guide for detailed definitions).

Shown below are tools and functions needed to build and verify an OpenMM System() .

2.2 Building an OpenMM Topology() for a coarse grained model.

An OpenMM Topology() object contains structural definitions for a molecular model (bond assignments, residue assignments, etc.).

Shown below are tools and functions needed to build and verify an OpenMM Topology() .

`build.cg_build.`**`build_topology`**(*cgmodel*, *use_pdbfile=False*, *pdbfile=None*)

Construct an OpenMM Topology() class object for our coarse grained model,

Parameters

- **cgmodel** (*class*) – CGModel() class object
- **use_pdbfile** (*Logical*) – Determines whether or not to use a PDB file in order to generate the Topology().
- **pdbfile** (*str*) – Name of a PDB file to use when building the topology.

Returns

- topology (Topology()) - OpenMM Topology() object

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> from foldamers.util.iotools import write_pdbfile_without_
↪topology
>>> input_pdb = "top.pdb"
>>> cgmodel = CGModel()
>>> write_pdbfile_without_topology(cgmodel,input_pdb)
>>> topology = build_topology(cgmodel,use_pdbfile=True,
↪pdbfile=input_pdb)
>>> cgmodel.topology = topology
```

> **Warning:** When 'use_pdbfile'=True, this function will use the PDBFile() class object from OpenMM to build the Topology(). In order for this approach to function correctly, the particle names in the PDB file must match the particle names in the coarse grained model.

`build.cg_build.`**`verify_topology`**(*cgmodel*)

Given a coarse grained model that contains a Topology() (cgmodel.topology), this function verifies the validity of the topology.

Parameters **cgmodel** (*class*) – CGModel() class object.

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> verify_topology(cgmodel)
```

Warning: The function will force an error exit if the topology is invalid, and will proceed as normal if the topology is valid.

2.3 Configuring OpenMM Forces() for a coarse grained model.

The 'cg_openmm' package contains multiple tools for verifying the validity of forces that are added to an OpenMM System(). These tools are shown below:

`build.cg_build.add_force(cgmodel, force_type=None)`

Given a 'cgmodel' and 'force_type' as input, this function adds the OpenMM force corresponding to 'force_type' to 'cgmodel.system'.

Parameters

- **cgmodel** – CGModel() class object.
- **type** – class
- **force_type** (*str*) – Designates the kind of 'force' provided. (Valid options include: "Bond", "Nonbonded", "Angle", and "Torsion")

Returns

- cgmodel (class) - 'foldamers' CGModel() class object
- force (class) - An OpenMM Force() object.

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> force_type = "Bond"
>>> cgmodel,force = add_force(cgmodel,force_type=force_type)
```

`build.cg_build.test_force(cgmodel, force, force_type=None)`

Given an OpenMM Force(), this function determines if there are any problems with its configuration.

Parameters

- **cgmodel** (*class*) – CGModel() class object.
- **force** – An OpenMM Force() object.
- **force_type** (*str*) – Designates the kind of 'force' provided. (Valid options include: "Nonbonded")

Returns

- 'success' (Logical) - a variable indicating if the force test passed.

Example

```
>>> from simtk.openmm.openmm import NonbondedForce
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> force = NonbondedForce()
>>> force_type = "Nonbonded"
>>> test_result = test_force(cgmodel,force,force_type="Nonbonded")
```

build.cg_build.**test_forces**(*cgmodel*)

Given a cgmodel that contains positions and an an OpenMM System() object, this function tests the forces for cgmodel.system.

More specifically, this function confirms that the model does not have any "NaN" or unphysically large forces.

Parameters

- **cgmodel** – CGModel() class object.
- **type** – class

Returns

- success (Logical) - Indicates if this cgmodel has unphysical forces.

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> pass_forces_test = test_forces(cgmodel)
```

build.cg_build.**get_num_forces**(*cgmodel*)

Given a CGModel() class object, this function determines how many forces we are including when evaluating the energy.

Parameters **cgmodel** (*class*) – CGModel() class object

Returns

- total_forces (int) - Number of forces in the coarse grained model

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> total_number_forces = get_num_forces(cgmodel)
```

2.4 Other tools for building and verifying the OpenMM System() and Topology()

Shown below are other utilities to build and verify a coarse grained model System()/Topology() for OpenMM:

build.cg_build.**add_new_elements**(*cgmodel*)

Add coarse grained particle types to OpenMM.

Parameters **cgmodel** (*class*) – CGModel object (contains all attributes for a coarse grained model).

Returns

- particle_list (list) - a list of the particles that were added to OpenMM's 'Element' List.

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> particle_types = add_new_elements(cgmodel)
```

> **Warning:** If the particle names were user defined, and any of the names conflict with existing element names in OpenMM, OpenMM will issue an error exit.

build.cg_build.**write_xml_file**(*cgmodel*, *xml_file_name*)

Write an XML-formatted forcefield file for a coarse grained model.

Parameters

- **cgmodel** (*class*) – CGModel() class object.
- **xml_file_name** (*str*) – Path to XML output file.

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> xml_file_name = "openmm_cgmodel.xml"
>>> write_xml_file(cgmodel,xml_file_name)
```

CHAPTER

THREE

OPENMM SIMULATION TOOLS FOR COARSE GRAINED MODELING

3.1 Building OpenMM Simulation() objects

OpenMM simulations are propagated using a Simulation() object.

Shown below are the main tools needed to build OpenMM Simulaton() objects for coarse grained modeling.

simulation.tools.**build_mm_simulation**(*topology*, *system*, *positions*, *temperature=Quantity(value=300.0, unit=kelvin)*, *simulation_time_step=None*, *total_simulation_time=Quantity(value=1.0, unit=picosecond)*, *output_pdb=None*, *output_data=None*, *print_frequency=100*, *test_time_step=False*)

Build an OpenMM Simulation()

Parameters

- **topology** (Topology()) – OpenMM Topology()
- **system** (System()) – OpenMM System()
- **positions** (Quantity() (np.array([cgmodel.num_beads,3]), simtk.unit)) – Positions array for the model we would like to test
- **temperature** (SIMTK Unit()) – Simulation temperature, default = 300.0 K
- **simulation_time_step** – Simulation integration time step
- **total_simulation_time** – Total run time for individual simulations

- **output_pdb** (*str*) – Output destination for PDB coordinates, Default = None
- **output_data** (*str*) – Output destination for non-coordinate simulation data, Default = None
- **print_frequency** – Number of simulation steps to skip when writing to output, Default = 100
- **test_time_step** (*Logical*) – Logical variable determining if a test of the time step will be performed, Default = False

Returns

- simulation (Simulation()) - OpenMM Simulation() object

Example

```
>>> from simtk import unit
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> topology = cgmodel.topology
>>> system = cgmodel.system
>>> positions = cgmodel.positions
>>> temperature = 300.0 * unit.kelvin
>>> simulation_time_step = 5.0 * unit.femtosecond
>>> total_simulation_time= 1.0 * unit.picosecond
>>> output_pdb = "output.pdb"
>>> output_data = "output.dat"
>>> print_frequency = 20
>>> openmm_simulation = build_mm_simulation(topology,system,
↪positions,temperature=temperature,simulation_time_
↪step=simulation_time_step,total_simulation_time=total_simulation_
↪time,output_pdb=output_pdb,output_data=output_data,print_
↪frequency=print_frequency,test_time_step=False)
```

simulation.tools.**run_simulation**(*cgmodel*, *output_directory*, *total_simulation_time*, *simulation_time_step*, *temperature*, *print_frequency*, *output_pdb=None*, *output_data=None*)

Run OpenMM() simulation

Parameters

- **cgmodel** (*class*) – CGModel() object
- **output_directory** (*str*) – Output directory for simulation data
- **total_simulation_time** – Total run time for individual simulations
- **simulation_time_step** – Simulation integration time step

- **temperature** – Simulation temperature, default = 300.0 K
- **print_frequency** – Number of simulation steps to skip when writing to output, Default = 100

Example

```
>>> import os
>>> from simtk import unit
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> topology = cgmodel.topology
>>> system = cgmodel.system
>>> positions = cgmodel.positions
>>> temperature = 300.0 * unit.kelvin
>>> simulation_time_step = 5.0 * unit.femtosecond
>>> total_simulation_time= 1.0 * unit.picosecond
>>> output_directory = os.getcwd()
>>> output_pdb = "output.pdb"
>>> output_data = "output.dat"
>>> print_frequency = 20
>>> run_simulation(cgmodel,output_directory,total_simulation_time,
↪simulation_time_step,temperature,print_frequency,output_
↪pdb=output_pdb,output_data=output_data)
```

Warning: When run with default options this subroutine is capable of producing a large number of output files. For example, by default this subroutine will plot the simulation data that is written to an output file.

3.2 Running Yank replica exchange simulations

The Yank python package is used to perform replica exchange sampling with OpenMM simulations.

Shown below are the main functions and tools necessary to conduct Yank replica exchange simulations with a coarse grained model in OpenMM.

`simulation.rep_exch.`**`run_replica_exchange`**(*topology*, *system*, *positions*, *temperature_list=None*, *simulation_time_step=None*, *total_simulation_time=Quantity(value=1.0, unit=picosecond)*, *output_data='output.nc'*, *print_frequency=100*, *verbose_simulation=False*, *exchange_attempts=None*, *test_time_step=False*, *output_directory=None*)

Run a Yank replica exchange simulation using an OpenMM coarse grained model.

Parameters

- **`topology`** – OpenMM Topology
- **`system`** – OpenMM System()
- **`positions`** – Positions array for the model we would like to test
- **`temperature_list`** – List of temperatures for which to perform replica exchange simulations, default = None
- **`simulation_time_step`** – Simulation integration time step
- **`total_simulation_time`** – Total run time for individual simulations
- **`output_data`** (*`string`*) – Name of NETCDF file where we will write simulation data
- **`print_frequency`** – Number of simulation steps to skip when writing to output, Default = 100
- **`verbose_simulation`** (*`Logical`*) – Determines how much output is printed during a simulation run. Default = False
- **`exchange_attempts`** (*`int`*) – Number of exchange attempts to make during a replica exchange simulation run, Default = None
- **`test_time_step`** (*`Logical`*) – Logical variable determining if a test of the time step will be performed, Default = False

- **output_directory** (*str*) – Path to which we will write the output from simulation runs.

Returns

- replica_energies (Quantity() (np.float([number_replicas,number_simulation_steps]), simtk.unit)) - The potential energies for all replicas at all (printed) time steps
- replica_positions (Quantity() (np.float([number_replicas,number_simulation_steps,cgmodel.num_beads,3]), simtk.unit)) - The positions for all replicas at all (printed) time steps
- replica_state_indices (np.int64([number_replicas,number_simulation_steps]), simtk.unit) - The thermodynamic state assignments for all replicas at all (printed) time steps

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> from cg_openmm.simulation.rep_exch import *
>>> cgmodel = CGModel()
>>> replica_energies,replica_positions,replica_state_indices = run_
↪replica_exchange(cgmodel.topology,cgmodel.system,cgmodel.
↪positions)
```

simulation.rep_exch.**read_replica_exchange_data**(*system=None*, *topology=None*, *temperature_list=None*, *output_data='output.nc'*, *print_frequency=None*)

Read replica exchange simulation data.

Parameters

- **system** – OpenMM system object, default = None
- **topology** – OpenMM topology object, default = None
- **temperature_list** – List of temperatures that will be used to define different replicas (thermodynamics states), default = None
- **output_data** (*str*) – Path to the output data for a Yank, NetCDF-formatted file containing replica exchange simulation data, default = None
- **print_frequency** (*int*) – Number of simulation steps to skip when writing data, default = None

Returns

- replica_energies (Quantity() (np.float([number_replicas,number_simulation_steps]), simtk.unit)) - The potential energies for all replicas at all (printed) time steps
- replica_positions (Quantity() (np.float([number_replicas,number_simulation_steps,cgmodel.num_beads,3]), simtk.unit)) - The positions for all replicas at all (printed) time steps
- replica_state_indices (np.int64([number_replicas,number_simulation_steps]), simtk.unit) - The thermodynamic state assignments for all replicas at all (printed) time steps

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> from cg_openmm.simulation.rep_exch import *
>>> cgmodel = CGModel()
>>> replica_energies,replica_positions,replica_state_indices = run_
↪replica_exchange(cgmodel.topology,cgmodel.system,cgmodel.
↪positions)
>>> replica_energies,replica_positions,replica_state_indices =␣
↪read_replica_exchange_data(system=cgmodel.system,
↪topology=cgmodel.topology,temperature_list=,output_data="output.
↪nc",print_frequency=None)
```

simulation.rep_exch.**make_replica_pdb_files**(*topology*, *replica_positions*)

Make PDB files from replica exchange simulation trajectory data

Parameters

- **topology** – OpenMM Topology
- **replica_positions** – Positions array for the replica exchange data for which we will write PDB files

Returns

- file_list (List(str)) - A list of names for the files that were written

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> from cg_openmm.simulation.rep_exch import *
>>> cgmodel = CGModel()
>>> replica_energies,replica_positions,replica_state_indices = run_
↪replica_exchange(cgmodel.topology,cgmodel.system,cgmodel.
↪positions)
>>> pdb_file_list = make_replica_pdb_files(cgmodel.topology,
↪replica_positions)
```

(continues on next page)

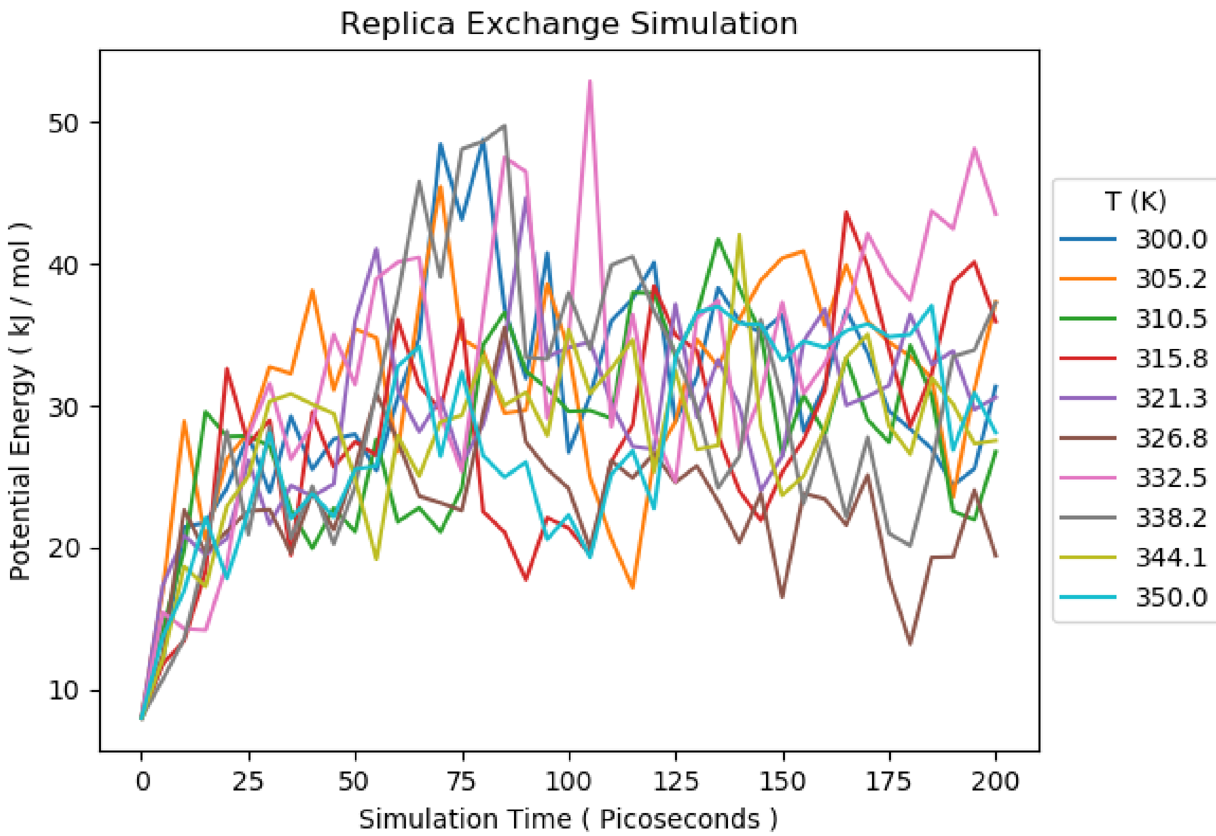
(continued from previous page)

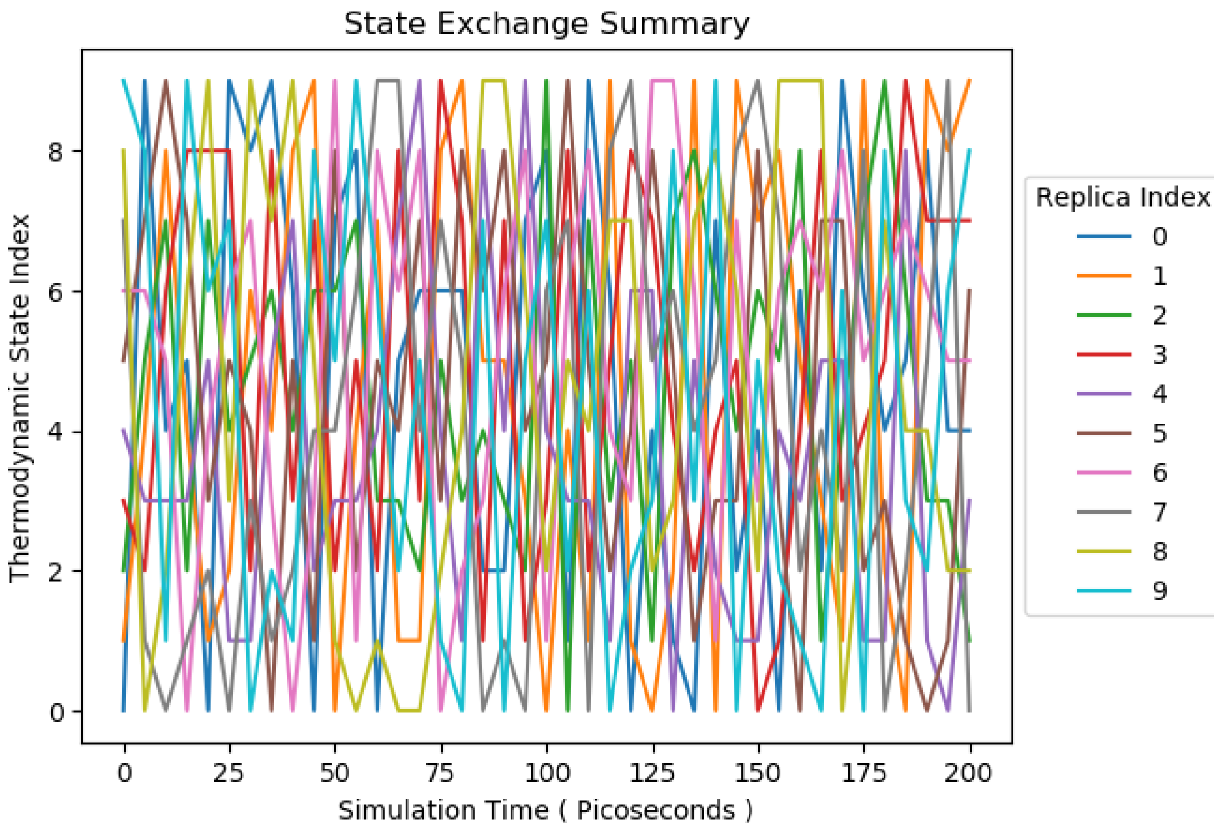
```

```

3.3 Tools to plot coarse grained model simulation results

The 'cg_openmm' package contains multiple functions to plot OpenMM and Yank simulation results using matplotlib .

Shown below is example output from a Yank replica exchange simulation run:





Shown below are functions which allow plotting of simulation results.

`simulation.rep_exch.`**`plot_replica_exchange_energies`**(*replica_energies*, *temperature_list*, *simulation_time_step*, *steps_per_stage=1*, *file_name='rep_ex_ener.png'*, *legend=True*, *output_directory=None*)

Plot the potential energies for a batch of replica exchange trajectories

Parameters

- **`replica_energies`** (*List(List(float * simtk.unit.energy for simulation_steps) for num_replicas)*) – List of dimension num_replicas X simulation_steps, which gives the energies for all replicas at all simulation steps
- **`temperature_list`** – List of temperatures for which to perform replica exchange simulations, default = [(300.0 * unit.kelvin).__add__(i * unit.kelvin) for i in range(-20,100,10)]

- **simulation_time_step** – Simulation integration time step
- **steps_per_stage** (*int*) – The number of simulation steps for individual replica "stages" (period of time between state exchanges), default = 1
- **file_name** (*str*) – The pathname of the output file for plotting results, default = "replica_exchange_energies.png"
- **output_directory** (*str*) – Path to which we will write the output from simulation runs, Default = None
- **legend** (*Logical*) – Controls whether a legend is added to the plot

..warning:: If more than 10 replica exchange trajectories are provided as input data, by default, this function will only plot the first 10 thermodynamic states. These thermodynamic states are chosen based upon their indices, not their instantaneous temperature (ensemble) assignment.

simulation.rep_exch.**plot_replica_exchange_summary**(*replica_states, temperature_list, simulation_time_step, steps_per_stage=1, file_name='rep_ex_states.png', legend=True, output_directory=None*)

Plot the thermodynamic state assignments for individual temperature replicas as a function of the simulation time, in order to obtain a visual summary of the replica exchanges from a Yank simulation.

Parameters

- **replica_states** (*List(List(float * simtk.unit.energy for simulation_steps) for num_replicas)*) – List of dimension num_replicas X simulation_steps, which gives the thermodynamic state indices for all replicas at all simulation steps
- **temperature_list** – List of temperatures for which to perform replica exchange simulations, default = [(300.0 * unit.kelvin).__add__(i * unit.kelvin) for i in range(-20,100,10)]
- **simulation_time_step** – Simulation integration time step
- **steps_per_stage** (*int*) – The number of simulation steps for individual replica "stages" (period of time between state exchanges), default = 1

- **file_name** (*str*) – The pathname of the output file for plotting results, default = "replica_exchange_state_transitions.png"
- **legend** (*Logical*) – Controls whether a legend is added to the plot
- **output_directory** (*str*) – Path to which we will write the output from simulation runs, default = None

..warning:: If more than 10 replica exchange trajectories are provided as input data, by default, this function will only plot the first 10 thermodynamic states. These thermodynamic states are chosen based upon their indices, not their instantaneous temperature (ensemble) assignment.

simulation.tools.**plot_simulation_data**(*simulation_times*, *y_data*, *plot_type=None*, *output_directory=None*)

Plot simulation data.

Parameters

- **simulation_times** (*List*) – List of simulation times (x data)
- **y_data** (*List*) – List of simulation data
- **plot_type** (*str*) – Form of data to plot, Default = None, Valid options include: "Potential Energy", "Kinetic Energy", "Total Energy", "Temperature"

Example

```
>>> import os
>>> from simtk import unit
>>> simulation_data_file = "output.pdb"
>>> simulation_time_step = 5.0 * unit.femtosecond
>>> simulation_data = read_simulation_data(simulation_data_file,
↪simulation_time_step)
>>> simulation_times = simulation_data["Simulation Time"]
>>> y_data = simulation_data["Potential Energy"]
>>> plot_type = "Potential Energy"
>>> plot_simulation_data(simulation_times,y_data,plot_type=
↪"Potential Energy")
```

simulation.tools.**plot_simulation_results**(*simulation_data_file*, *plot_output_directory*, *simulation_time_step*)

Plot all data from an OpenMM output file

Parameters

- **simulation_data_file** (*str*) – Path to file containing simulation data

- **plot_output_directory** (*str*) – Path to folder where plotting results will be written.
- **simulation_time_step** – Simulation integration time step

Example

```
>>> import os
>>> from simtk import unit
>>> simulation_data_file = "output.pdb"
>>> plot_output_directory = os.getcwd()
>>> simulation_time_step = 5.0 * unit.femtosecond
>>> plot_simulation_results(simulation_data_file,plot_output_
↪directory,simulation_time_step)
```

3.4 Other simulation tools

Shown below are other tools which aid the building and verification of OpenMM simulation objects.

simulation.tools.**get_mm_energy**(*topology*, *system*, *positions*)

Get the OpenMM potential energy for a system, given a topology and set of positions.

Parameters

- **topology** – OpenMM Topology()
- **system** – OpenMM System()
- **positions** – Positions array for the model we would like to test

Returns

- potential_energy (Quantity()) - The potential energy for the model with the provided positions.

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> topology = cgmodel.topology
>>> system = cgmodel.system
>>> positions = cgmodel.positions
>>> openmm_potential_energy = get_mm_energy(topology,system,
↪positions)
```

simulation.tools.**get_simulation_time_step**(*topology*, *system*, *positions*, *temperature*, *total_simulation_time*, *time_step_list=None*)

Determine a suitable simulation time step.

Parameters

- **topology** – OpenMM Topology
- **system** – OpenMM System()
- **positions** – Positions array for the model we would like to test
- **temperature** – Simulation temperature
- **total_simulation_time** – Total run time for individual simulations
- **time_step_list** (*List, default = None*) – List of time steps for which to attempt a simulation in OpenMM.

Returns

- time_step (SIMTK Unit()) - A successfully-tested simulation time-step for the provided coarse grained model
- tolerance (SIMTK Unit()) - The maximum change in forces that will be tolerated when testing the time step.

Example

```
>>> from simtk import unit
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> topology = cgmodel.topology
>>> system = cgmodel.system
>>> positions = cgmodel.positions
>>> temperature = 300.0 * unit.kelvin
>>> total_simulation_time = 1.0 * unit.picosecond
>>> time_step_list = [1.0 * unit.femtosecond, 2.0 * unit.
↪femtosecond, 5.0 * unit.femtosecond]
>>> best_time_step,max_force_tolerance = get_simulation_time_
↪step(topology,system,positions,temperature,total_simulation_time,
↪time_step_list=time_step_list)
```

simulation.tools.**minimize_structure**(*topology, system, positions, temperature=Quantity(value=0.0, unit=kelvin), simulation_time_step=None, total_simulation_time=Quantity(value=1.0, unit=picosecond), output_pdb=None, output_data=None, print_frequency=1*)

Minimize the potential energy

Parameters

- **topology** (*Topology()*) – OpenMM topology
- **system** (*System()*) – OpenMM system
- **positions** – Positions array for the model we would like to test
- **temperature** – Simulation temperature
- **total_simulation_time** – Total run time for individual simulations
- **output_pdb** (*str*) – Output destinaton for PDB-formatted coordinates during the simulation
- **output_data** (*str*) – Output destination for simulation data

- **print_frequency** (*int*) – Number of simulation steps to skip when writing data, default = 1

Returns

- positions (Quantity() (np.array([cgmodel.num_beads,3]), simtk.unit)) - Minimized positions
- potential_energy (Quantity() - Potential energy for the minimized structure.

Example

```
>>> from simtk import unit
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> topology = cgmodel.topology
>>> system = cgmodel.system
>>> positions = cgmodel.positions
>>> temperature = 300.0 * unit.kelvin
>>> total_simulation_time = 1.0 * unit.picosecond
>>> simulation_time_step = 1.0 * unit.femtosecond
>>> output_pdb = "output.pdb"
>>> output_data = "output.dat"
>>> print_frequency = 20
>>> minimum_energy_structure,potential_energy,openmm_simulation_
↪object = minimize_structure(topology,system,positions,
↪temperature=temperature,simulation_time_step=simulation_time_
↪step,total_simulation_time=total_simulation_time,output_
↪pdb=output_pdb,output_data=output_data,print_frequency=print_
↪frequency)
```

simulation.tools.**read_simulation_data**(*simulation_data_file*, *simulation_time_step*)

Read OpenMM simulation data

Parameters

- **simulation_data_file** (*str*) – Path to file that will be read
- **simulation_time_step** – Time step to apply for the simulation data

Returns

- data (dict("Simulation Time": list,"Potential Energy": list,"Kinetic Energy": list,"Total Energy": list,"Temperature": list)) - A dictionary containing the simulation times, potential energies, kinetic energies, and total energies from an OpenMM simulation trajectory.

Example

```
>>> from simtk import unit
>>> simulation_data_file = "output.dat"
>>> simulation_time_step = 5.0 * unit.femtosecond
>>> data = read_simulation_data(simulation_data_file,simulation_
↪time_step)
```

simulation.rep_exch.**get_minimum_energy_ensemble**(*topology*, *replica_energies*, *replica_positions*, *ensemble_size=5*, *file_name=None*)

Get an ensemble of low (potential) energy poses, and write the lowest energy structure to a PDB file if a file_name is provided.

Parameters

- **topology** – OpenMM Topology()
- **replica_energies** (*List(List(float * simtk.unit.energy for simulation_steps) for num_replicas)*) – List of dimension num_replicas X simulation_steps, which gives the energies for all replicas at all simulation steps
- **replica_positions** (*np.array((float * simtk.unit.positions for num_beads) for simulation_steps)*) – List of positions for all output frames for all replicas
- **file_name** – Output destination for PDB coordinates of minimum energy pose, Default = None

Returns

- ensemble (List()) - A list of poses that are in the minimum energy ensemble.

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> from cg_openmm.simulation.rep_exch import *
>>> cgmodel = CGModel()
>>> replica_energies,replica_positions,replica_state_indices = run_
↪replica_exchange(cgmodel.topology,cgmodel.system,cgmodel.
↪positions)
>>> ensemble_size = 5
>>> file_name = "minimum.pdb"
>>> minimum_energy_ensemble = get_minimum_energy_ensemble(cgmodel.
↪topology,replica_energies,replica_positions,ensemble_
↪size=ensemble_size,file_name=file_name)
```

CHAPTER FOUR

UTILITIES FOR COARSE GRAINED MODELING IN OPENMM

This page details the functionality of utilities in cg_openmm/src/utilities/util.py.

`utilities.util.distance(positions_1, positions_2)`

Calculate the distance between two particles, given their positions.

Parameters

- **positions_1** (Quantity() (np.array([3]), simtk.unit)) – Positions for the first particle
- **positions_2** – Positions for the first particle

Returns

- distance (Quantity()) - Distance between two particles

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> cgmodel = CGModel()
>>> particle_1_coordinates = cgmodel.positions[0]
>>> particle_2_coordinates = cgmodel.positions[1]
>>> particle_distance = distance(particle_1_coordinates,particle_2_
↪coordinates)
```

`utilities.util.get_box_vectors(box_size)`

Given a simulation box length, construct a vector.

Parameters **box_size** – Length of individual sides of a simulation box

Returns

- box_vectors (List(Quantity())) - Vectors to use when defining an OpenMM simulation box.

`utilities.util.lj_v(positions_1, positions_2, sigma, epsilon)`

Calculate the Lennard-Jones interaction energy between two particles, given their positions and definitions for their equilbrium interaction distance (sigma) and strength (epsilon).

Parameters

- **positions_1** – Positions for the first particle
- **positions_2** – Positions for the first particle
- **sigma** – Lennard-Jones equilibrium interaction distance for two non-bonded particles
- **epsilon** – Lennard-Jones equilibrium interaction energy for two non-bonded particles.

Returns

- v (Quantity()) - Lennard-Jones interaction energy

utilities.util.**set_box_vectors**(*system*, *box_size*)

Impose a set of simulation box vectors on an OpenMM simulation object.

Parameters

- **system** (System()) – OpenMM System()
- **box_size** – Length of individual sides of a simulation box

Returns

- system (System()) - OpenMM system object

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

s

u

INDEX

V

W