foldamers Documentation

Release 0.0

Garrett A. Meek
Theodore L. Fobe
Connor M. Vogel

Research group of Professor Michael R. Shirts

Dept. of Chemical and Biological Engineering
University of Colorado Boulder

Sep 08, 2019

CONTENTS

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

CHAPTER

ONE

COARSE GRAINED MODEL UTILITIES

The foldamers package uses “CGModel()” objects to define and store information about the properties of coarse grained models.

1.1 ‘basic_cgmodel’: a simple function to build coarse grained homopolymers

Shown below is the ‘basic_cgmodel’ function, which requires a minimal set of input arguments to build a coarse grained holopolymer model.

`cg_model.cgmodel.`**`basic_cgmodel`**(*polymer_length=12, backbone_length=1, sidechain_length=1, sidechain_positions=[0], mass=Quantity(value=100.0, unit=dalton), bond_length=Quantity(value=0.75, unit=nanometer), sigma=Quantity(value=1.85, unit=nanometer), epsilon=Quantity(value=0.5, unit=kilocalorie/mole), positions=None*)

Parameters

- **`polymer_length`** (*`int`*) – Number of monomer units, default = 8
- **`backbone_length`** (*`int`*) – Number of beads in the backbone for individual monomers within a coarse grained model, default = 1
- **`sidechain_length`** (*`int`*) – Number of beads in the sidechain for individual monomers within a coarse grained model, default = 1
- **`sidechain_positions`** (*`List( int )`*) – Designates the indices of backbone beads upon which we will place sidechains, default = [0] (add a sidechain to the first backbone bead in each monomer)

- **mass** (Quantity()) – Mass for all coarse grained beads, default = 100.0 * unit.amu
- **bond_length** – Defines the length for all bond types, default = 7.5 * unit.angstrom
- **sigma** – Lennard-Jones equilibrium interaction distance (by default, calculated for particles that are separated by 3 or more bonds), default = 18.5 * bond_length (for all interaction types)
- **epsilon** – Lennard-Jones equilibrium interaction energy (by default, calculated for particles that are separated by 3 or more bonds), default = 0.5 * unit.kilocalorie_per_mole
- **positions** – Positions for coarse grained particles in the model, default = None

Returns cgmodel: CGModel() class object

Return type class

..warning:: this function has significant limitations, in comparison with building a coarse grained model with the CGModel() class. In particular, this function makes it more difficult to build heteropolymers, and is best-suited for the simulation of homopolymers.

Example

```
>>> from simtk import unit
>>> polymer_length = 20
>>> backbone_length = 1
>>> sidechain_length = 1
>>> sidechain_positions = [0]
>>> mass = 100.0 * unit.amu
>>> bond_length=0.75 * unit.nanometer
>>> sigma=1.85*unit.nanometer
>>> epsilon=0.5 * unit.kilocalorie_per_mole
>>> cgmodel = basic_cgmodel(polymer_length=polymer_length,backbone_
↪length=backbone_length,sidechain_length=sidechain_length,
↪sidechain_positions=sidechain_positions,mass=mass,bond_
↪length=bond_length,sigma=sigma,epsilon=epsilon)
```

1.2 Using the ‘CGModel()’ class to build coarse grained heteropolymers

Shown below is a detailed description of the ‘CGModel()’ class object, as well as some of examples demonstrating how to use its functions and attributes.

class cg_model.cgmodel.**CGModel**(*positions=None, polymer_length=12, backbone_lengths=[1], sidechain_lengths=[1], sidechain_positions=[0], masses={'backbone_bead_masses': Quantity(value=100.0, unit=dalton), 'sidechain_bead_masses': Quantity(value=100.0, unit=dalton)}, sigmas={'bb_bb_sigma': Quantity(value=1.875, unit=nanometer), 'bb_sc_sigma': Quantity(value=1.875, unit=nanometer), 'sc_sc_sigma': Quantity(value=1.875, unit=nanometer)}, epsilons={'bb_bb_eps': Quantity(value=0.05, unit=kilocalorie/mole), 'sc_sc_eps': Quantity(value=0.05, unit=kilocalorie/mole)}, bond_lengths={'bb_bb_bond_length': Quantity(value=0.75, unit=nanometer), 'bb_sc_bond_length': Quantity(value=0.75, unit=nanometer), 'sc_sc_bond_length': Quantity(value=0.75, unit=nanometer)}, bond_force_constants=None, bond_angle_force_constants=None, torsion_force_constants=None, equil_bond_angles=None, equil_torsion_angles=None, charges=None, constrain_bonds=True, include_bond_forces=False, include_nonbonded_forces=True, include_bond_angle_forces=True, include_torsion_forces=True, check_energy_conservation=True, use_structure_library=False, heteropolymer=False, monomer_types=None, sequence=None, random_positions=False*)

Build a coarse grained model class object.

Example

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

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
```

(continues on next page)

(continued from previous page)

```
>>> from simtk import unit
>>> bond_length = 7.5 * unit.angstrom
>>> bond_lengths = {'bb_bb_bond_length': bond_length,'bb_sc_bond_
↪length': bond_length,'sc_sc_bond_length': bond_length}
>>> constrain_bonds = False
>>> cgmodel = CGModel(bond_lengths=bond_lengths,constrain_
↪bonds=constrain_bonds)
```

Example

```
>>> from foldamers.cg_model.cgmodel import CGModel
>>> from simtk import unit
>>> backbone_length=1
>>> sidechain_length=1
>>> sidechain_positions=0
>>> bond_length = 7.5 * unit.angstrom
>>> sigma = 2.0 * bond_length
>>> epsilon = 0.2 * unit.kilocalorie_per_mole
>>> sigmas = {'bb_bb_sigma': sigma,'sc_sc_sigma': sigma}
>>> epsilons = {'bb_bb_eps': epsilon,'bb_sc_eps': epsilon,'sc_sc_
↪eps': epsilon}
>>> A = {'monomer_name': "A", 'backbone_length': backbone_length,
↪'sidechain_length': sidechain_length, 'sidechain_positions':␣
↪sidechain_positions, 'num_beads': num_beads, 'bond_lengths':␣
↪bond_lengths, 'epsilons': epsilons, 'sigmas': sigmas}
>>> B = {'monomer_name': "B", 'backbone_length': backbone_length,
↪'sidechain_length': sidechain_length, 'sidechain_positions':␣
↪sidechain_positions, 'num_beads': num_beads, 'bond_lengths':␣
↪bond_lengths, 'epsilons': epsilons, 'sigmas': sigmas}
>>> monomer_types = [A,B]
>>> sequence = [A,A,A,B,A,A,A,B,A,A,A,B]
>>> cgmodel = CGModel(heteropolymer=True,monomer_types=monomer_
↪types,sequence=sequence)
```

`get_all_particle_masses`()

Returns a list of all unique particle masses

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

Returns list_of_masses: List of unique particle masses

Return type

List(Quantity())

`get_bond_angle_force_constant`(*particle_1_index*, *particle_2_index*, *particle_3_index*)

Determines the correct bond angle force constant for a bond angle between three particles, given their indices within the coarse grained model

Parameters

- **CGModel** (*class*) – CGModel() class object
- **particle_1_index** (*int*) – Index for the first particle
- **particle_2_index** (*int*) – Index for the second particle
- **particle_3_index** (*int*) – Index for the third particle

Returns bond_angle_force_constant: The assigned bond angle force constant for the provided particles

Return type

bond_angle_force_constant: Quantity()

get_bond_angle_list()

Construct a list of bond angles, which can be used to build bond angle potentials for the coarse grained model

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

Returns A list of indices for all of the bond angles in the coarse grained model

Return type List(List(int, int, int))

get_bond_force_constant(*particle_1_index*, *particle_2_index*)

Determines the correct bond force constant for two particles, given their indices

Parameters

- **CGModel** (*class*) – CGModel() class object
- **particle_1_index** (*int*) – Index for the first particle
- **particle_2_index** (*int*) – Index for the second particle

Returns bond_force_constant: The assigned bond force constant for the provided particles

Return type

bond_length: Quantity()

get_bond_length(*particle_1_index*, *particle_2_index*)

Determines the correct bond length for two particles, given their indices.

Parameters

- **CGModel** (*class*) – CGModel() class object
- **particle_1_index** (*int*) – Index for the first particle

- **`particle_2_index`** (*int*) – Index for the second particle

Returns bond_length: The assigned bond length for the provided particles

Return type

bond_length: Quantity()

`get_bond_length_from_names`(*particle_1_name*, *particle_2_name*)
Determines the correct bond length for two particles, given their symbols.

Parameters

- **`CGModel`** (*class*) – CGModel() class object
- **`particle_1_name`** (*str*) – Name for the first particle
- **`particle_2_name`** (*str*) – Name for the second particle

Returns bond_length: The assigned bond length for the provided particles

Return type

bond_length: Quantity()

`get_bond_list`()
Construct a bond list for the coarse grained model

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

Returns bond_list: A list of the bonds in the coarse grained model.

Return type bond_list: List(List(int, int))

`get_epsilon`(*particle_index*, *particle_type=None*)
Returns the Lennard-Jones potential epsilon value for a particle, given its index within the coarse grained model.

Parameters

- **`CGModel`** (*class*) – CGModel() class object
- **`particle_index`** (*int*) – Index of the particle for which we would like to determine the type
- **`particle_type`** (*str*) – Designates a particle as "backbone" or "sidechain"

Returns epsilon: The assigned Lennard-Jones epsilon value for the provided particle index

Return type

Quantity()

get_equil_bond_angle(*particle_1_index*, *particle_2_index*, *particle_3_index*)
Determines the correct equilibrium bond angle between three particles, given their indices within the coarse grained model

Parameters

- **CGModel** (*class*) – CGModel() class object
- **particle_1_index** (*int*) – Index for the first particle
- **particle_2_index** (*int*) – Index for the second particle
- **particle_3_index** (*int*) – Index for the third particle

Returns equil_bond_angle: The assigned equilibrium bond angle for the provided particles

Return type equil_bond_angle: float

get_equil_torsion_angle(*torsion*)
Determines the correct equilibrium angle for a torsion (bond angle involving four particles), given their indices within the coarse grained model

Parameters

- **CGModel** (*class*) – CGModel() class object
- **torsion** (*List(int)*) – A list of the indices for the particles in a torsion

Returns equil_torsion_angle: The assigned equilibrium torsion angle for the provided particles

Return type equil_torsion_angle: float

get_monomer_types()
Get a list of 'monomer_types' for all unique monomers.

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

Returns monomer_types: A list of unique monomer types in the coarse grained model

Return type

monomer_types: List(dict('monomer_name': str, 'backbone_length': int, 'sidechain_length': int, 'sidechain_positions': List(int), 'num_beads': int, 'bond_lengths': List(Quantity()), 'epsilons': List(Quantity()), 'sigmas': List(Quantity())))

get_nonbonded_exclusion_list()
Get a list of the nonbonded interaction exclusions, which are assigned if two particles are separated by less than three bonds

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

Returns exclusion_list: A list of the nonbonded particle interaction exclusions for the coarse grained model

Return type List(List(int, int))

`get_nonbonded_interaction_list`()
Construct a nonbonded interaction list for the coarse grained model

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

Returns interaction_list: A list of the nonbonded interactions (which don't violate exclusion rules) in the coarse grained model

Return type interaction_list: List(List(int, int))

`get_num_beads`()
Calculate the number of beads in a coarse grained model class object

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

Returns num_beads: The total number of beads in the coarse grained model

Return type num_beads: int

`get_particle_charge`(*particle_index*)
Returns the charge for a particle, given its index within the coarse grained model

Parameters

- **`CGModel`** (*`class`*) – CGModel() class object
- **`particle_index`** (*`int`*) – Index of the particle for which we would like to determine the type

Returns particle_charge: The charge for the provided particle index

Return type

Quantity()

`get_particle_list`()
Get a list of particles, where the indices correspond to those in the system/topology.

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

Returns particle_list: A list of unique particles in the coarse grained model

Return type particle_list: List(str)

`get_particle_mass`(*particle_index*)
Get the mass for a particle, given its index within the coarse grained model

Parameters

- **`CGModel`** (*`class`*) – CGModel() class object

- **particle_index** (*int*) – Index of the particle for which we would like to determine the type

Returns particle_mass: The mass for the provided particle index

Return type

Quantity()

get_particle_name(*particle_index*)

Returns the name of a particle, given its index within the model

Parameters

- **CGModel** (*class*) – CGModel() class object
- **particle_index** (*int*) – Index of the particle for which we would like to determine the type

Returns particle_name: The name of the particle

Return type particle_name: str

get_particle_type(*particle_index*, *particle_name=None*)

Indicates if a particle is a backbone bead or a sidechain bead

Parameters

- **CGModel** (*class*) – CGModel() class object
- **particle_index** (*int*) – Index of the particle for which we would like to determine the type
- **particle_name** (*str*) – Name of the particle that we would like to "type".

Returns particle_type: 'backbone' or 'sidechain'

Return type particle_type: str

get_sigma(*particle_index*, *particle_type=None*)

Returns the Lennard-Jones potential sigma value for a particle, given its index within the coarse grained model.

Parameters

- **CGModel** (*class*) – CGModel() class object
- **particle_index** (*int*) – Index of the particle for which we would like to determine the type
- **particle_type** (*str*) – Designates a particle as "backbone" or "sidechain"

Returns sigma: The assigned Lennard-Jones sigma value for the provided particle index

Return type

Quantity()

`get_torsion_force_constant`(*torsion*)

Determines the correct torsion force constant for a torsion (bond angle involving four particles), given their indices within the coarse grained model

Parameters

- **`CGModel`** (*`class`*) – CGModel() class object
- **`torsion`** (*`List( int )`*) – A list of the indices for the particles in a torsion

Returns torsion_force_constant: The assigned torsion force constant for the provided particles

Return type

torsion_force_constant: Quantity()

`get_torsion_list`()

Construct a list of particle indices from which to define torsions for the coarse grained model

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

Returns torsions: A list of the particle indices for the torsions in the coarse grained model

Return type torsions: List(List(int, int, int, int))

`nonbonded_interaction_list = None`

Initialize new (coarse grained) particle types:

CHAPTER TWO

ENSEMBLE BUILDING TOOLS

The foldamers package contains several tools for building conformational ensembles. The MDTraj and MSMBuilder packages are leveraged to perform structural analyses in order to identify poses that are structurally similar.

2.1 Using MSMBuilder to generate conformational ensembles

The foldamers package allows the user to apply K-means clustering tools from MSMBuilder in order to search for ensembles of poses that are structurally similar. The centroid configurations for individual clusters are used as a reference, and ensembles are defined by including all structures that fall below an RMSD positions threshold (<2 Angstroms).

2.2 Native structure-based ensemble generation tools

The foldamers package allows the user to build "native" and "nonnative" structural ensembles, and to evaluate their energetic differences with the Z-score. These tools require identification of a "native" structure.

`ensembles.ens_build.`**`get_ensembles`**`(`*cgmodel, native_structure, ensemble_size=None*`)`

`ensembles.ens_build.`**`get_native_ensemble`**`(`*cgmodel, native_structure, ensemble_size=10, native_fraction_cutoff=0.9, rmsd_cutoff=10.0, ensemble_build_method='native_contacts'*`)` `ensembles.ens_build.`**`get_nonnative_ensemble`**(*cgmodel*, *native_structure*, *ensemble_size=100*, *native_fraction_cutoff=0.75*, *rmsd_cutoff=10.0*, *ensemble_build_method='native_contacts'*)

`ensembles.ens_build.`**`z_score`**(*topology*, *system*, *nonnative_ensemble_energies*, *native_ensemble_energies*)

Given an ensemble of nonnative structures, and a low-energy ("native") structure, this subroutine will calculate the Z-score.

nonnative_ensemble: List(positions(np.array(float * simtk.unit (shape = num_beads x 3)))
A list of the positions for all members in the high_energy ensemble.

native_structure: positions(np.array(float * simtk.unit (shape = num_beads x 3))
The positions for a low energy structure.

2.3 Energy-based ensemble generation tools

The foldamers package allows the user to build structural ensembles that exhibit similar energies. Shown below are tools that enable energy-based ensemble generation.

`ensembles.ens_build.`**`get_ensemble`**(*cgmodel*, *ensemble_size=100*, *high_energy=False*, *low_energy=False*)

Given a coarse grained model, this function generates an ensemble of high energy configurations and, by default, saves this ensemble to the foldamers/ensembles database for future reference/use, if a high-energy ensemble with these settings does not already exist.

Parameters

- **`cgmodel`** (*`class`*) – CGModel() class object.
- **`ensemble_size`** (*`integer`*) – Number of structures to generate for this ensemble, default = 100
- **`high_energy`** (*`Logical`*) – If set to 'True', this function will generate an ensemble of high-energy structures, default = False
- **`low_energy`** (*`Logical`*) – If set to 'True', this function will generate an ensemble of low-energy structures, default = False

Returns

- ensemble (List(positions(np.array(float*simtk.unit (shape = num_beads x 3))))) - A list of the positions for all members in the ensemble.

`ensembles.ens_build.`**`test_energy`**(*energy*)

Given an energy, this function determines if that energy is too large to be "physical". This function is used to determine if the user-defined input parameters for a coarse grained model give a reasonable potential function.

Parameters **energy** (Quantity() or float) – The energy to test.

Returns

- pass_energy_test (Logical) - A variable indicating if the energy passed ("True") or failed ("False") a "sanity" test for the model's energy.

`ensembles.ens_build.`**`improve_ensemble`**(*energy*, *positions*, *ensemble*, *ensemble_energies*, *unchanged_iterations*)

Given an energy and positions for a single pose, as well as the same data for a reference ensemble, this function "improves" the quality of the ensemble by identifying poses with the lowest potential energy.

Parameters

- **energy** – The energy for a pose.
- **positions** – Positions for coarse grained particles in the model, default = None
- **ensemble** (*List(positions(np.array(float*simtk.unit (shape = num_beads x 3))))*) – A group of similar poses.
- **ensemble_energies** – A list of energies for a conformational ensemble.
- **unchanged_iterations** (*int*) – The number of iterations for which the ensemble has gone unchanged.

Returns

- ensemble (List(positions(np.array(float*simtk.unit (shape = num_beads x 3))))) - A list of the positions for all members in the ensemble.
- ensemble_energies (List(Quantity())) - A list of the energies that were stored in the PDB files for the ensemble, if any.
- unchanged_iterations (int) - The number of iterations for which the ensemble has gone unchanged.

2.4 Writing and reading ensemble data from the 'foldamers' database

The foldamers package is designed to store the low-energy poses from simulation runs of new (previously un-modelled) coarse grained representations. At present, the package does not enable storage of heteropolymers, in order to minimize the size of the database. For homopolymers, the syntax for assigning directory names for coarse grained model data is as follows:

directory_name = str("foldamers/ensembles/" + str(polymer_length) + "_" + str(backbone_length) + "_" + str(sidechain_length) "_" + str(sidechain_positions) + "_" + str(bb_bb_bond_length) + "_" + str(sc_bb_bond_length) + "_" + str(sc_sc_bond_length))

For example, the directory name for a model with 20 monomers, all of which contain one backbone bead and one sidechain bead, and whose bond lengths are all 7.5 Angstroms, would be: "foldamers/ensembles/20_1_1_0_7.5_7.5_7.5".

The following functions are used to read and write ensemble data to the foldamers database (located in 'foldamers/ensembles').

`ensembles.ens_build.`**`get_ensemble_directory`**(*cgmodel*, *ensemble_type=None*)

Given a CGModel() class object, this function uses its attributes to assign an ensemble directory name.

For example, the directory name for a model with 20 monomers, all of which contain one backbone bead and one sidechain bead, and whose bond lengths are all 7.5 Angstroms, would be: "foldamers/ensembles/20_1_1_0_7.5_7.5_7.5".

Parameters

- **`cgmodel`** (*`class`*) – CGModel() class object
- **`ensemble_type`** (*`str`*) – Designates the type of ensemble for which we will assign a directory name. default = None. Valid options include: "native" and "nonnative"

Returns

- ensemble_directory (str) - The path/name for the ensemble directory.

`ensembles.ens_build.`**`write_ensemble_pdb`**(*cgmodel*, *ensemble_directory=None*)

Given a CGModel() class object that contains positions, this function writes a PDB file for the coarse grained model, using those positions.

Parameters

- **`cgmodel`** (*`class`*) – CGModel() class object
- **`ensemble_directory`** (*`str`*) – Path to a folder containing PDB files, default = None

Warning: If no 'ensemble_directory' is provided, the

`ensembles.ens_build.`**`get_pdb_list`**(*ensemble_directory*)

Given an 'ensemble_directory', this function retrieves a list of the PDB files within it.

Parameters **`ensemble_directory`** (*`str`*) – Path to a folder containing PDB files

Returns

- pdb_list (List(str)) - A list of the PDB files in the provided 'ensemble_directory'.

`ensembles.ens_build.`**`get_ensemble_data`**(*cgmodel*, *ensemble_directory*)

Given a CGModel() class object and an 'ensemble_directory', this function reads the PDB files within that directory, as well as any energy data those files contain.

Parameters

- **`cgmodel`** (*`class`*) – CGModel() class object
- **`ensemble_directory`** (*`str`*) – The path/name of the directory where PDB files for this ensemble are stored

Returns

- ensemble (List(positions(np.array(float*simtk.unit (shape = num_beads x 3))))) - A list of the positions for all members in the ensemble.
- ensemble_energies (List(Quantity())) - A list of the energies that were stored in the PDB files for the ensemble, if any.

Warning: When energies are written to a PDB file, only the sigma and epsilon values for the model are written to the file with the positions. Unless the user is confident about the model parameters that were used to generate the energies in the PDB files, it is probably best to re-calculate their energies. This can be done with the 'cg_openmm' package. More specifically, one can compute an updated energy for individual ensemble members, with the current coarse grained model parameters, with 'get_mm_energy', a function in 'cg_openmm/cg_openmm/simulation/tools.py'.

CHAPTER

THREE

THERMODYNAMIC ANALYSIS TOOLS FOR COARSE GRAINED MODELING

This page details the functions and classes in src/thermo

3.1 Tools to calculate the heat capacity with pymbar

Shown below are functions/tools used in order to calculate the heat capacity with pymbar.

CHAPTER

FOUR

UTILITIES FOR THE ‘FOLDAMERS’ PACKAGE

This page details the functions and classes in src/util.

4.1 Input/Output options (src/utilities/iotools.py)

Shown below is a detailed description of the input/output options for the foldamers package.

`utilities.iotools.`**`write_bonds`**`(`*CGModel*, *pdb_object*`)`

Writes the bonds from an input CGModel class object to the file object ‘pdb_object’, using PDB ‘CONECT’ syntax.

CGModel: Coarse grained model class object

pdb_object: File object to which we will write the bond list

`utilities.iotools.`**`write_cg_pdb`**`(`*cgmodel*, *file_name*`)`

Writes the positions from an input CGModel class object to the file ‘filename’. Used to test the compatibility of coarse grained model parameters with the OpenMM PDBFile() functions, which are needed to write coordinates to a PDB file during MD simulations.

CGModel: Coarse grained model class object

filename: Path to the file where we will write PDB coordinates.

`utilities.iotools.`**`write_pdbfile_without_topology`**`(`*CGModel*, *filename*, *energy=None*`)`

Writes the positions from an input CGModel class object to the file ‘filename’.

CGModel: Coarse grained model class object

filename: Path to the file where we will write PDB coordinates.

energy: Energy to write to the PDB file, default = None

4.2 Utilities and random functions (src/utilities/util.py)

`utilities.util.`**`assign_position`**(*positions*, *bond_length*, *sigma*, *bead_index*, *parent_index*)

Assign random position for a bead

positions: Positions for all beads in the coarse-grained model. (np.array(num_beads x 3))

bond_length: Bond length for all beads that are bonded, (float * simtk.unit.distance) default = 1.0 * unit.angstrom

positions: Positions for all beads in the coarse-grained model. (np.array(num_beads x 3))

`utilities.util.`**`assign_position_lattice_style`**(*cgmodel*, *positions*, *distance_cutoff*, *parent_bead_index*, *bead_index*)

Assign random position for a bead

positions: Positions for all beads in the coarse-grained model. (np.array(num_beads x 3))

bond_length: Bond length for all beads that are bonded, (float * simtk.unit.distance) default = 1.0 * unit.angstrom

positions: Positions for all beads in the coarse-grained model. (np.array(num_beads x 3))

`utilities.util.`**`attempt_lattice_move`**(*parent_coordinates*, *bond_length*, *move_direction_list*)

Given a set of cartesian coordinates, assign a new particle a distance of 'bond_length' away in a random direction.

parent_coordinates: Positions for a single particle, away from which we will place a new particle a distance of 'bond_length' away. (np.array(float * unit.angstrom (length = 3)))

bond_length: Bond length for all beads that are bonded, (float * simtk.unit.distance) default = 1.0 * unit.angstrom

trial_coordinates: Positions for a new trial particle (np.array(float * unit.angstrom (length = 3)))

`utilities.util.`**`attempt_move`**(*parent_coordinates*, *bond_length*)

Given a set of cartesian coordinates, assign a new particle a distance of 'bond_length' away in a random direction.

parent_coordinates: Positions for a single particle, away from which we will place a new particle a distance of 'bond_length' away. (np.array(float * unit.angstrom (length = 3)))

bond_length: Bond length for all beads that are bonded, (float * simtk.unit.distance) default = 1.0 * unit.angstrom

trial_coordinates: Positions for a new trial particle (np.array(float * unit.angstrom (length = 3)))

`utilities.util.`**`collisions`**(*positions*, *distance_list*, *distance_cutoff*)
Determine whether there are any collisions between non-bonded particles, where a "collision" is defined as a distance shorter than the user-provided 'bond_length'.

distances: List of the distances between all nonbonded particles. (list (float * simtk.unit.distance (length = # nonbonded_interactions)))

bond_length: Bond length for all beads that are bonded, (float * simtk.unit.distance) default = 1.0 * unit.angstrom

collision: Logical variable stating whether or not the model has bead collisions. default = False

`utilities.util.`**`distance`**(*positions_1*, *positions_2*)
Construct a matrix of the distances between all particles.

positions_1: Positions for a particle (np.array(length = 3))

positions_2: Positions for a particle (np.array(length = 3))

distance (float * unit)

`utilities.util.`**`distance_matrix`**(*positions*)
Construct a matrix of the distances between all particles.

positions: Positions for an array of particles. (np.array(num_particles x 3))

distance_matrix: Matrix containing the distances between all beads. (np.array(num_particles x 3))

`utilities.util.`**`distances`**(*interaction_list*, *positions*)
Calculate the distances between a trial particle ('new_coordinates') and all existing particles ('existing_coordinates').

new_coordinates: Positions for a single trial particle (np.array(float * unit.angstrom (length = 3)))

existing_coordinates: Positions for a single trial particle (np.array(float * unit.angstrom (shape = num_particles x 3)))

distances: List of the distances between all nonbonded particles. (list (float * simtk.unit.distance (length = # nonbonded_interactions)))

`utilities.util.`**`first_bead`**(*positions*)
Determine if we have any particles in 'positions' positions: Positions for all beads in the coarse-grained model. (np.array(float * unit (shape = num_beads x 3))) first_bead: Logical variable stating if this is the first particle.

`utilities.util.`**`get_move`**(*trial_coordinates*, *move_direction*, *distance*, *bond_length*, *finish_bond=False*)
Given a 'move_direction', a current distance, and a target 'bond_length' (Index denoting x,y,z Cartesian direction), update the coordinates for the particle.

trial_coordinates: positions for a particle (np.array(float * unit.angstrom (length = 3)))

move_direction: Cartesian direction in which we will attempt a particle placement, where: x=0, y=1, z=2. (integer)

distance: Current distance from parent particle (float * simtk.unit.distance)

bond_length: Target bond_length for particle placement. (float * simtk.unit.distance)

finish_bond: Logical variable determining how we will update the coordinates for this particle.

trial_coordinates: Updated positions for the particle (np.array(float * unit.angstrom (length = 3)))

`utilities.util.`**`get_structure_from_library`**(*cgmodel*, *high_energy=False*, *low_energy=False*)

Given a coarse grained model class object, this function retrieves a set of positions for the model from the ensemble library, in: '../foldamers/ensembles/${backbone_length}_${sidechain_length}_${sidechain_positions}' If this coarse grained model does not have an ensemble library, an error message will be returned and we will attempt to assign positions at random with 'random_positions()'.

cgmodel: CGModel() class object.

Parameters

- **`high_energy`** (*`Logical`*) – If set to 'True', this function will generate an ensemble of high-energy structures, default = False
- **`low_energy`** (*`Logical`*) – If set to 'True', this function will generate an ensemble of low-energy structures, default = False

positions: Positions for all beads in the coarse-grained model. (np.array(num_beads x 3))

`utilities.util.`**`random_positions`**(*cgmodel*, *max_attempts=1000*, *use_library=False*, *high_energy=False*, *low_energy=False*, *generate_library=False*)

Assign random positions for all beads in a coarse-grained polymer.

cgmodel: CGModel() class object.

max_attempts: The maximum number of times that we will attempt to build a coarse grained model with the settings in 'cgmodel'. default = 1000

use_library: A logical variable determining if we will generate a new random structure, or take a random structure from the library in the following path: '../foldamers/ensembles/${backbone_length}_${sidechain_length}_${sidechain_positions}' default = True (NOTE: By default, if use_library = False, new structures will be added to the ensemble library for the relevant coarse grained model. If that model does not have an ensemble library, one will be created.)

Parameters

- **`high_energy`** (*`Logical`*) – If set to 'True', this function will generate an ensemble of high-energy structures, default = False
- **`low_energy`** (*`Logical`*) – If set to 'True', this function will generate an ensemble of low-energy structures, default = False

positions: Positions for all beads in the coarse-grained model. (np.array(num_beads x 3))

`utilities.util.`**`random_sign`**(*number*)

Returns 'number' with a random sign.

number: float

number

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX