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 13, 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 (class) - CGModel() class object

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
>>> 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(Quantity())) - List of unique particle masses

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 (Quantity()) - The assigned bond angle force constant for the provided particles

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

- bond_angles (List(List(int, int, int))) - A list of indices for all of the bond angles in the coarse grained model

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 (Quantity()) - The assigned bond force constant for the provided particles

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 (Quantity()) - The assigned bond length for the provided particles

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 (Quantity()) - The assigned bond length for the provided particles

get_bond_list()

Construct a bond list for the coarse grained model

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

Returns

- bond_list (List(List(int, int))) - A list of the bonds in the coarse grained model.

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 (Quantity()) - The assigned Lennard-Jones epsilon value for the provided particle index

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 (float) - The assigned equilibrium bond angle for the provided particles

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 (float) - The assigned equilibrium torsion angle for the provided particles

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

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

Returns

- 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())))) - A list of unique monomer types in the coarse grained model

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 (List(List(int, int))) - A list of the nonbonded particle interaction exclusions for the coarse grained model

get_nonbonded_interaction_list()

Construct a nonbonded interaction list for the coarse grained model

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

Returns

- interaction_list (List(List(int, int))) - A list of the nonbonded interactions (which don't violate exclusion rules) in the coarse grained model

get_num_beads()

Calculate the number of beads in a coarse grained model class object

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

Returns

- num_beads (int) - The total number of beads in the coarse grained model

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 (Quantity()) - The charge for the provided particle index

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 (List(str)) - A list of unique particles in the coarse grained model

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 (Quantity()) - The mass for the provided particle index

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 (str) - The name of the particle

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 (str) - 'backbone' or 'sidechain'

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 (Quantity()) - The assigned Lennard-Jones sigma value for the provided particle index

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 (Quantity()) - The assigned torsion force constant for the provided particles

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 (List(List(int, int, int, int))) - A list of the particle indices for the torsions in the coarse grained model

CHAPTER

TWO

THERMODYNAMIC ANALYSIS TOOLS FOR COARSE GRAINED MODELING

The 'foldamers' package contains wide-ranging tools to analyze simulation data in order to estimate the thermodynamic properties of a system. In particular, the package leverages tools from pymbar to enable analysis of and estimate expectation values for sampled and unsampled thermodynamic states.

2.1 Using MBAR to compute expectation values for structural & thermodynamic properties

Shown below is the main 'foldamers' function used to re-weight simulation results with 'pymbar'.

`parameters.reweight.`**`get_mbar_expectation`**(*E_kln*, *temperature_list*, *NumIntermediates*, *output=None*, *mbar=None*)

Given a properly-formatted matrix of energies with associated temperatures this function reweights with MBAR (if 'mbar'=None), and can also compute the expectation value for any property of interest.

> **Warning:** This function accepts an input matrix thtat has either 'E_kln' or 'E_kn' format, but always provides an 'E_kn'-formatted matrix in return.

Parameters

- **E_kln** (*List(List(float * simtk.unit.energy for simulation_steps) for num_replicas) OR List(List(List(float * simtk.unit.energy for simulation_steps) for num_replicas) for num_replicas)*) – A matrix of energies or samples for a property that we would like to use to make predictions with MBAR.

- **temperature_list** (*List(float * simtk.unit.temperature)*) – List of temperatures for the simulation data.
- **NumIntermediates** (*int*) – The number of states to insert between existing states in 'T_from_file'
- **output** (*str*) – The 'output' option to use when calling MBAR, default = 'differences'
- **mbar** (MBAR class object) – An MBAR() class object (from the 'pymbar' package), default = None

Returns

- mbar (MBAR) - An MBAR() class object (from the 'pymbar' package)
- E_kn (List(List(float * simtk.unit.energy for num_samples) for num_replicas)) - A matrix of energies or samples for a property that we would like to use to make predictions with MBAR.
- result (List(List(float for num_samples) for num_replicas) - The MBAR expectation value for the energies and/or other samples that were provided.
- dresult (List(List(float for num_samples) for num_replicas) - The MBAR expectation value for the energies and/or other samples that were provided.
- Temp_k (List(float * simtk.unit.temperature)) - A new list of temperatures that includes the inserted intermediates.

2.2 Evaluating thermodynamic properties with 'pymbar'

Shown below 'pymbar'-based tools/functions to evaluate thermodynamic properties within the 'foldamers' package.

parameters.reweight.**get_free_energy_differences**(*mbar*)

Given an MBAR class object, this function computes the free energy differences for the states defined within.

Parameters **mbar** – An MBAR() class object (from the 'pymbar' package)

Returns

- df_ij (np.array(n_mbar_states x n_thermo_states) - Free energy differences for the thermodynamic states in 'mbar'
- ddf_ij (np.array(n_mbar_states x n_thermo_states) - Uncertainty in the free energy differences for the thermodynamic states in 'mbar'

parameters.reweight.**get_entropy_differences**(*mbar*)

Given an MBAR class object, this function computes the entropy differences for the states defined within.

Parameters **mbar** – An MBAR() class object (from the 'pymbar' package)

Returns

- Delta_s (np.array(n_mbar_states x n_thermo_states) - Entropy differences for the thermodynamic states in 'mbar'
- dDelta_s (np.array(n_mbar_states x n_thermo_states) - Uncertainty in the entropy differences for the thermodynamic states in 'mbar'

parameters.reweight.**get_enthalpy_differences**(*mbar*)

Given an MBAR class object, this function computes the enthalpy differences for the states defined within.

Parameters **mbar** – An MBAR() class object (from the 'pymbar' package)

Returns

- Delta_u (np.array(n_mbar_states x n_thermo_states) - Enthalpy differences for the thermodynamic states in 'mbar'
- dDelta_u (np.array(n_mbar_states x n_thermo_states) - Uncertainty in the enthalpy differences for the thermodynamic states in 'mbar'

2.3 Tools to evaluate thermodynamic properties with 'pymbar'

Shown below are other 'pymbar'-based tools/functions that aid evaluation of thermodynamic properties within the 'foldamers' package.

parameters.reweight.**get_temperature_list**(*min_temp*, *max_temp*, *num_replicas*)

Given the parameters to define a temperature range as input, this function uses logarithmic spacing to generate a list of intermediate temperatures.

Parameters

- **min_temp** (Quantity()) – The minimum temperature in the temperature list.
- **max_temp** – The maximum temperature in the temperature list.
- **num_replicas** (*int*) – The number of temperatures in the list.

Returns

- temperature_list (List(float * simtk.unit.temperature)) - List of temperatures

parameters.reweight.**get_intermediate_temperatures**(*T_from_file*, *NumIntermediates*)

Given a list of temperatures and a number of intermediate states as input, this function calculates the values for temperatures intermediate between those in this list, as the mean between values in the list.

Parameters

- **T_from_file** (*List(float * simtk.unit.temperature)*) – List of temperatures
- **NumIntermediates** (*int*) – The number of states to insert between existing states in 'T_from_file'

Returns

- Temp_k (List(float * simtk.unit.temperature)) - A new list of temperatures that includes the inserted intermediates.

parameters.reweight.**calc_temperature_spacing**(*min_temp*, *max_temp*, *num_replicas*, *replica_index*)

Given the parameters to define a temperature range as input, as well as the current temperature index, this function uses logarithmic scaling to calculate the temperature spacing.

Parameters

- **min_temp** – The minimum temperature in the temperature list.
- **max_temp** – The maximum temperature in the temperature list.
- **num_replicas** (*int*) – The number of temperatures in the list.
- **replica_index** (*int*) – The index for the current temperature of interest.

Returns

- delta (Quantity()) - The spacing to use when assigning the next temperature value.

parameters.reweight.**get_decorrelated_samples**(*replica_positions*, *replica_energies*, *temperature_list*)

Given a set of replica exchange trajectories, energies, and associated temperatures, this function returns decorrelated samples, as obtained from pymbar with timeseries.subsampleCorrelatedData.

Parameters

- **replica_positions** – Positions array for the replica exchange data for which we will write PDB files
- **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(float * simtk.unit.temperature)*) – List of temperatures for the simulation data.

Returns

- configurations (List(Quantity() (np.array([n_decorrelated_samples,cgmodel.num_beads,3]), simtk.unit)) - A list of decorrelated samples
- energies (List(Quantity())) - The energies for the decorrelated samples (configurations)

2.4 Calculating the heat capacity with pymbar

Shown below is the primary function used to evaluate the heat capacity with pymbar:

thermo.calc.**get_heat_capacity**(*replica_energies*, *temperature_list*, *num_intermediate_states=None*)

Given a set of trajectories, a temperature list, and a number of intermediate states to insert for the temperature list, this function calculates and plots the heat capacity profile.

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 = None
- **num_intermediate_states** (*int*) – The number of states to insert between existing states in 'temperature_list'

Returns

- C_v (List(float)) - The heat capacity values for all (including inserted intermediates) states
- dC_v (List(float)) - The uncertainty in the heat capacity values for intermediate states
- new_temp_list (List(float * unit.simtk.temperature)) - The temperature list corresponding to the heat capacity values in 'C_v'

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

thermo.calc.**calculate_heat_capacity**(*E_expect*, *E2_expect*, *dE_expect*, *DeltaE_expect*, *dDeltaE_expect*, *df_ij*, *ddf_ij*, *Temp_k*, *originalK*, *numIntermediates*, *ntypes=3*, *dertype='temperature'*)

Given numerous pieces of thermodynamic data this function calculates the heat capacity by following the 'pymbar' example .

thermo.calc.**plot_heat_capacity**(*C_v*, *dC_v*, *temperature_list*, *file_name='heat_capacity.png'*)

Given an array of temperature-dependent heat capacity values and the uncertainties in their estimates, this function plots the heat capacity curve.

Parameters

- **C_v** (*List(float)*) – The heat capacity data to plot.

- **dC_v** (*List(float)*) – The uncertainties in the heat capacity data
- **file_name** (*str*) – The name/path of the file where plotting output will be written, default = "heat_capacity.png"

CHAPTER

THREE

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.

3.1 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).

ensembles.cluster.**concatenate_trajectories**(*pdb_file_list*, *combined_pdb_file='combined.pdb'*)

Given a list of PDB files, this function reads their coordinates, concatenates them, and saves the combined coordinates to a new file (useful for clustering with MSMBuilder).

Parameters

- **pdb_file_list** (*List(str)*) – A list of PDB files to read and concatenate
- **combined_pdb_file** (*str*) – The name of file/path in which to save a combined version of the PDB files, default = "combined.pdb"

Returns

- combined_pdb_file (str) - The name/path for a file within which the concatenated coordinates will be written.

ensembles.cluster.**align_structures**(*reference_traj*, *target_traj*)

Given a reference trajectory, this function performs a structural alignment for a second input trajectory, with respect to the reference.

Parameters

- **reference_traj** (MDTraj() trajectory) – The trajectory to use as a reference for alignment.
- **target_traj** – The trajectory to align with the reference.

Returns

- aligned_target_traj (MDTraj() trajectory) - The coordinates for the aligned trajectory.

ensembles.cluster.**get_cluster_centroid_positions**(*pdb_file*, *cgmodel*, *n_clusters=None*)

Given a PDB file and coarse grained model as input, this function performs K-means clustering on the poses in the PDB file, and returns a list of the coordinates for the "centroid" pose of each cluster.

Parameters

- **pdb_file** (*str*) – The path/name of a file from which to read trajectory data
- **cgmodel** (*class*) – A CGModel() class object
- **n_clusters** (*int*) – The number of "bins" within which to cluster the poses from the input trajectory.

Returns

- centroid_positions (List (np.array(float * unit.angstrom (num_particles x 3)))) - A list of the poses corresponding to the centroids of all trajectory clusters.

3.2 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*)

Given a native structure as input, this function builds both native and nonnative ensembles.

Parameters

- **cgmodel** (*class*) – CGModel() class object
- **native_structure** (Quantity() (np.array([cgmodel.num_beads,3]), simtk.unit)) – The positions for the model's native structure
- **ensemble_size** (*int*) – The number of poses to generate for the nonnative ensemble, default = None

Returns

- nonnative_ensemble (List(positions(np.array(float*simtk.unit (shape = num_beads x 3))))) - A list of the positions for all members in the nonnative ensemble
- nonnative_ensemble_energies (List(Quantity())) - A list of the energies for all members of the nonnative ensemble
- native_ensemble (List(positions(np.array(float*simtk.unit (shape = num_beads x 3))))) - A list of the positions for all members in the native ensemble
- native_ensemble_energies (List(Quantity())) - A list of the energies for the native ensemble

ensembles.ens_build.**get_native_ensemble**(*cgmodel*, *native_structure*, *ensemble_size=10*, *native_fraction_cutoff=0.9*, *rmsd_cutoff=10.0*, *ensemble_build_method='mbar'*)

Given a native structure as input, this function builds a "native" ensemble of structures.

Parameters

- **cgmodel** (*class*) – CGModel() class object
- **native_structure** – The positions for the model's native structure
- **ensemble_size** (*int*) – The number of poses to generate for the nonnative ensemble, default = 10

- **native_fraction_cutoff** (*float*) – The fraction of native contacts above which a pose will be considered "native", default = 0.9
- **rmsd_cutoff** (*float*) – The distance beyond which non-bonded interactions will be ignored, default = 10.0 x bond_length
- **ensemble_build_method** (*str*) – The method that will be used to generate a nonnative ensemble. Valid options include "mbar" and "native_contacts". If the "mbar" approach is chosen, decorrelated replica exchange simulation data is used to generate the nonnative ensemble. If the "native_contacts" approach is chosen, individual NVT simulations are used to generate the nonnative ensemble, default = "mbar"

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.

ensembles.ens_build.**get_nonnative_ensemble**(*cgmodel*, *native_structure*, *ensemble_size=100*, *native_fraction_cutoff=0.75*, *rmsd_cutoff=10.0*, *ensemble_build_method='mbar'*)

Given a native structure as input, this function builds a "nonnative" ensemble of structures.

Parameters

- **cgmodel** (*class*) – CGModel() class object
- **native_structure** – The positions for the model's native structure
- **ensemble_size** (*int*) – The number of poses to generate for the nonnative ensemble, default = 100
- **native_fraction_cutoff** (*float*) – The fraction of native contacts below which a pose will be considered "nonnative", default = 0.75
- **rmsd_cutoff** (*float*) – The distance beyond which non-bonded interactions will be ignored, default = 10.0 x bond_length
- **ensemble_build_method** (*str*) – The method that will be used to generate a nonnative ensemble. Valid options include "mbar" and "native_contacts". If the "mbar" approach is chosen, decorrelated replica exchange simulation data is used to generate the nonnative ensemble. If the "native_contacts" approach is chosen, individual NVT simulations are used to generate the nonnative ensemble, default = "mbar"

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.

ensembles.ens_build.**z_score**(*nonnative_ensemble_energies*, *native_ensemble_energies*)

Given a set of nonnative and native ensemble energies, this function computes the Z-score (for a set of model parameters).

Parameters

- **nonnative_ensemble_energies** – A list of the energies for all members of the nonnative ensemble
- **native_ensemble_energies** – A list of the energies for the native ensemble

Returns

- z_score (float) - The Z-score for the input ensembles.

3.3 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** – 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.

3.4 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

FOUR

UTILITIES FOR THE 'FOLDAMERS' PACKAGE

The 'foldamers' package contains a number of utilities for reading and writing data, as well as random structure generation.

4.1 Random structure generation in 'foldamers'

Random structures are often needed as a starting point for simulation of a new coarse grained model. Shown below are the main tools that allows the user to build a random structure:

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.

Parameters

- **max_attempts** (*int*) – The maximum number of attempts to generate random positions a coarse grained model with the current parameters, default = 1000
- **use_library** – A logical variable determining if a new random structure will be generated, or if an ensemble will be read from the 'foldamers' database, default = False
- **use_library** – Logical
- **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
- **generate_library** – If set to 'True', this function will save the poses that are generated to the 'foldamers' ensemble database.

Returns

- positions (np.array(float * unit.angstrom (num_particles x 3))) - A set of coarse grained model positions.

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 'foldamers' 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 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

Returns

- positions (np.array(float * unit.angstrom (num_particles x 3))) - A set of coarse grained model positions.

Shown below are other tools that support the task of random structure generation:

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

Assign random position for a bead

param positions Positions for the particles in a coarse grained model.

type positions np.array(float * unit.angstrom (num_particles x 3))

param bond_length The distance to step when placing new particles.

type bond_length Quantity()

param distance_cutoff The distance below which particles will be considered to have "collisions".

type distance_cutoff Quantity()

param parent_bead_index The index of the particle from which we will bond a new particle, when assigning pos

itions.

type parent_bead_index int

param bead_index The index of the particle for which the function will assign positions.

type bead_index int

returns

- positions (np.array(float * unit.angstrom (num_particles x 3))) - A set of positions for the updated model, including the particle that was just added.
- success (Logical) - Indicates whether or not a particle was placed successfully.

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

Assign random position for a particle

Parameters

- **cgmodel** (*class*) – CGModel() class object.
- **positions** (*np.array(float * unit.angstrom (num_particles x 3))*) – Positions for the particles in a coarse grained model.

- **distance_cutoff** – The distance below which particles will be considered to have "collisions".
- **parent_bead_index** (*int*) – The index of the particle from which we will bond a new particle, when assigning positions.
- **bead_index** (*int*) – The index of the particle for which the function will assign positions.

Returns

- test_positions (np.array(float * unit.angstrom (num_particles x 3))) - A set of positions for the updated model, including the particle that was just added.
- success (Logical) - Indicates whether or not a particle was placed successfully.

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

Given a set of cartesian coordinates this function positions a new particle a distance of 'bond_length' away in a random direction.

Parameters

- **parent_coordinates** (*np.array(float * unit.angstrom (length = 3))*) – Positions for a single particle, away from which we will place a new particle a distance of 'bond_length' away.
- **bond_length** – Bond length for all beads that are bonded.
- **move_direction_list** (*List(int)*) – A list of cartesian directions (denoted by integers) that tracks the directions in which a particle placement has been attempted.

Returns

- trial_coordinates (np.array(float * unit.angstrom (length = 3))) - The coordinates for a new, trial particle.
- move_direction_list (List(int)) - A list of cartesian directions (denoted by integers) that tracks the directions in which a particle placement has been attempted.

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.

param parent_coordinates Positions for a single particle, away from which we will place a new particle a distance of 'bond_length' away.

type parent_coordinates np.array(float * unit.angstrom (length = 3))

param bond_length Bond length for all beads that are bonded.

type bond_length Quantity()

returns

- trial_coordinates (np.array(float * unit.angstrom (length = 3))) - The coordinates for a new, trial

particle.

utilities.util.**collisions**(*positions*, *distance_list*, *distance_cutoff*)

Determine if there are any collisions between non-bonded particles, where a "collision" is defined as a distance shorter than 'distance_cutoff'.

Parameters

- **positions** (*np.array(float * unit.angstrom (num_particles x 3))*) – Positions for the particles in a coarse grained model.
- **distance_list** – A list of distances.
- **distance_cutoff** – The distance below which particles will be considered to have "collisions".

Returns

- collision (Logical) - A variable indicating whether or not the model contains particle collisions.

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

Calculate the distance between two particles.

Parameters

- **positions_1** – Positions for a particle
- **positions_2** – Positions for a particle

Returns

- distance (Quantity()) - The distance between the provided particles.

utilities.util.**distance_matrix**(*positions*)

Construct a matrix of the distances between an input array of particles.

Parameters **positions** (*np.array(float * unit.angstrom (num_particles x 3))*) – Positions for an array of particles.

Returns

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

utilities.util.**distances**(*interaction_list*, *positions*)

Calculate the distances between all non-bonded particles in a model, given a list of particle interactions and particle positions.

Parameters

- **interaction_list** (*List([int, int])*) – A list of non-bonded particle interactions
- **positions** (*np.array(float * unit.angstrom (num_particles x 3))*) – Positions for the particles in a coarse grained model.

Returns

- distance_list (List(Quantity())) - A list of distances for the non-bonded interactions in the coarse grained model.

utilities.util.**first_bead**(*positions*)

Determine if the provided 'positions' contain any particles (are the coordinates non-zero).

Parameters **positions** (*np.array(float * unit (shape = num_beads x 3))*) – Positions for all beads in the coarse-grained model.

Returns

- first_bead (Logical) - Variable stating if the positions are all non-zero.

utilities.util.**get_move**(*trial_coordinates*, *move_direction*, *distance*, *bond_length*, *finish_bond=False*)

Used to build random structures. Given a set of input coordinates, this function attempts to add a new particle.

Parameters

- **trial_coordinates** (*np.array(float * unit.angstrom (length = 3))*) – Positions for a particle
- **move_direction** (*int*) – Cartesian direction in which we will attempt a particle placement, where: x=0, y=1, z=2.
- **distance** – Current distance between the trial coordinates for the particle this function is positioning and the particle that it is branched from (bonded to).
- **bond_length** – The distance to step before placing a new particle.
- **finish_bond** – Logical variable determining how we will update the coordinates for this particle, default = False. If set to "True", the "move"

length will be the difference between “distance” and “bond_length”.

Returns

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

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

Returns the provided ‘number’ with a random sign.

Parameters **number** (*float*) – The number to add a random sign (positive or negative) to

Returns

- number (float) - The provided number with a random sign added

4.2 Reading and writing to output within the 'foldamers' package

Shown below are 'foldamers' tools for writing data and structures to output files.

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

CHAPTER

FIVE

PARAMETER ANALYSIS TOOLS FOR COARSE GRAINED MODELING

The 'foldamers' package allows wide-ranging parameter analyses for a coarse grained model. In particular, the package contains tools to analyze quantities that reflect secondary structure, including: 1) the fraction of native contacts, 2) the orientational ordering parameter 'P2', and 3) using kHelios, helical order parameters such as the pitch.

parameters.secondary_structure.**fraction_native_contacts**(*cgmodel*, *positions*, *native_structure*, *cutoff_distance=None*)

Given a coarse grained model, positions for that model, and positions for the native structure, this function calculates the fraction of native contacts for the model.

Parameters

- **cgmodel** (*class*) – CGModel() class object
- **positions** (*np.array(float * unit.angstrom (num_particles x 3))*) – Positions for the particles in a coarse grained model.
- **native_structure** (*np.array(float * unit.angstrom (num_particles x 3))*) – Positions for the native structure.
- **cutoff_distance** (Quantity()) – The maximum distance for two nonbonded particles that are defined as "native",default=None

Returns

- Q (float) - The fraction of native contacts for the model with the current structure.

parameters.secondary_structure.**get_helical_data**(*cgmodel*)

parameters.secondary_structure.**get_helical_parameters**(*cgmodel*)

Given a coarse grained model as input, this function uses the kHelios software package to analyze the helical properties of the model.

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

Returns

- pitch (float) - The distance between monomers in adjacent turns of a helix
- radius (float) - The radius of the helix
- monomers_per_turn (float) - The number of monomrs per turn of the helix
- residual (float) - The average distance of all backbone particles from a circle projected onto the x-y plane. Used to determine the accuracy of the helical axis, as fit to the input data. Units are in Angstroms.

Warning: This function requires a pre-installed version of kHelios . Because kHelios is formatted to accept input job scripts, this function writes and executes a job script for kHelios. In order to function properly, the user must redefine the 'helios_path' variable for their system.

parameters.secondary_structure.**orient_along_z_axis**(*cgmodel*, *plot_projections=False*)

Given a coarse grained model as input, this function orients the model along the z-axis.

Parameters

- **cgmodel** (*class*) – CGModel() class object
- **plot_projections** – Variable indicating whether or not to plot intermediate projections/operations during identification of a helical axis.

Returns

- cgmodel (class) - CGModel() class object, with positions oriented so that the helical axis is along the z-axis

parameters.optimize.**calculate_C_v_fitness**(*C_v*, *T_list*)

parameters.optimize.**get_fwhm_symmetry**(*C_v*, *T_list*)

parameters.optimize.**get_num_maxima**(*C_v*)

parameters.optimize.**optimize_heat_capacity**(*cgmodel*)

parameters.optimize.**optimize_lj**(*cgmodel*, *base_epsilon=0.0*, *sigma_attempts=3*, *epsilon_attempts=3*)

Optimize the Lennard-Jones interaction potential parameters (sigma and epsilon, for all in-

teraction types) in the model defined by a cgmodel() class object, using a combination of replica exchange simulations and re-weighting techniques.

Parameters

- **cgmodel** (*class.*) – CGModel() class object, default = None
- **sigma_attempts** –

cgmodel: CGModel() class object.

parameters.optimize.**optimize_model_parameter**(*optimization_target*)

parameters.optimize.**optimize_parameter**(*cgmodel*, *optimization_parameter*, *optimization_range_min*, *optimization_range_max*, *steps=None*)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

I

O

P

R

T

U

W

Z