blues Documentation

Release 0.2.5

Samuel C. Gill, Nathan M. Lim, Kalistyn Burley, David L. Mobley

Jan 22, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

Fig. 1: BLUES is a python package that takes advantage of non-equilibrium candidate Monte Carlo moves (NCMC) to help sample between different ligand binding modes.

1.1 Github

Check out our Github repository.

If you have any problems or suggestions through our issue tracker.

To contribute to our code, please fork our repository and open a Pull Request.

1.2 Publication

Binding Modes of Ligands Using Enhanced Sampling (BLUES): Rapid Decorrelation of Ligand Binding Modes via Nonequilibrium Candidate Monte Carl

Samuel C. Gill, Nathan M. Lim, Patrick B. Grinaway, Ariën S. Rustenburg, Josh Fass, Gregory A. Ross, John D. Chodera , and David L. Mobley

The Journal of Physical Chemistry B **2018** *122* (21), 5579-5598

DOI: 10.1021/acs.jpcb.7b11820

Publication Date (Web): February 27, 2018

Abstract

Accurately predicting protein–ligand binding affinities and binding modes is a major goal in computational chemistry, but even the prediction of ligand binding modes in proteins poses major challenges. Here, we focus on solving the binding mode prediction problem for rigid fragments. That is, we focus on computing the dominant placement, conformation, and orientations of a relatively rigid, fragment-like ligand in a receptor, and the populations of the multiple binding modes which may be relevant. This problem is important in its own right, but is even more timely given the recent success of alchemical free energy calculations. Alchemical calculations are increasingly used to predict binding free energies of ligands to receptors. However, the accuracy of these calculations is dependent on proper sampling of the relevant ligand binding modes. Unfortunately, ligand binding modes may often be uncertain, hard to predict, and/or slow to interconvert on simulation time scales, so proper sampling with current techniques can require prohibitively long simulations. We need new methods which dramatically improve sampling of ligand binding modes. Here, we develop and apply a nonequilibrium candidate Monte Carlo (NCMC) method to improve sampling of ligand binding modes. In this technique, the ligand is rotated and subsequently allowed to relax in its new position through alchemical perturbation before accepting or rejecting the rotation and relaxation as a nonequilibrium Monte Carlo move. When applied to a T4 lysozyme model binding system, this NCMC method shows over 2 orders of magnitude improvement in binding mode sampling efficiency compared to a brute force molecular dynamics simulation. This is a first step toward applying this methodology to pharmaceutically relevant binding of fragments and, eventually, drug-like molecules. We are making this approach available via our new Binding modes of ligands using enhanced sampling (BLUES) package which is freely available on GitHub.

1.3 Theory

Suggested readings:

CHAPTER

TWO

INSTALLATION

BLUES is compatible with MacOSX/Linux with Python=3.6.

This is a python tool kit with a few dependencies. We recommend installing miniconda. Then you can create an environment with the following commands:

```
conda create -n blues python=3.6
conda activate blues
```

2.1 Stable Releases

The recommended way to install BLUES would be to install from conda.

```
# Install OpenEye toolkits and related tools first
conda install -c openeye/label/Orion -c omnia oeommtools
conda install -c openeye openeye-toolkits

# Install necessary dependencies
conda install -c omnia -c conda-forge openmmtools=0.15.0 openmm=7.4.2 numpy cython

conda install -c mobleylab blues
```

2.2 Source Installation

Although we do NOT recommend it, you can also install directly from the source code.

```
git clone https://github.com/MobleyLab/blues.git ./blues
conda install -c omnia -c conda-forge openmmtools=0.15.0 openmm=7.4.2 numpy cython
conda install -c openeye/label/Orion -c omnia oeommtools
conda install -c openeye openeye-toolkits
pip install -e .
```

To validate your BLUES installation run the tests.

```
cd blues/tests
pytest -v -s
```

CHAPTER

THREE

MODULES

3.1 Moves

Provides the main Move class which allows definition of moves which alter the positions of subsets of atoms in a context during a BLUES simulation, in order to increase sampling. Also provides functionality for CombinationMove definitions which consist of a combination of other pre-defined moves such as via instances of Move.

Authors: Samuel C. Gill

Contributors: Nathan M. Lim, Kalistyn Burley, David L. Mobley

3.1.1 Move

class blues.moves.**Move**

This is the base Move class. Move provides methods for calculating properties and applying the move on the set of atoms being perturbed in the NCMC simulation.

initializeSystem(*system*, *integrator*)

If the system or integrator needs to be modified to perform the move (ex. adding a force) this method is called during the start of the simulation to change the system or integrator to accomodate that.

Parameters

- **system** (*openmm.System*) – System to be modified.
- **integrator** (*openmm.Integrator*) – Integrator to be modified.

Returns

- **system** (*openmm.System*) – The modified System object.
- **integrator** (*openmm.Integrator*) – The modified Integrator object.

beforeMove(*context*)

This method is called at the start of the NCMC portion if the context needs to be checked or modified before performing the move at the halfway point.

Parameters **context** (*openmm.Context*) – Context containing the positions to be moved.

Returns **context** (*openmm.Context*) – The same input context, but whose context were changed by this function.

afterMove(*context*)

This method is called at the end of the NCMC portion if the context needs to be checked or modified before performing the move at the halfway point.

Parameters **context** (*openmm.Context*) – Context containing the positions to be moved.

Returns **context** (*openmm.Context*) – The same input context, but whose context were changed by this function.

move(*context*)

This method is called at the end of the NCMC portion if the context needs to be checked or modified before performing the move at the halfway point.

Parameters **context** (*openmm.Context*) – Context containing the positions to be moved.

Returns **context** (*openmm.Context*) – The same input context, but whose context were changed by this function.

3.1.2 RandomLigandRotationMove

class blues.moves.**RandomLigandRotationMove**(*structure*, *resname='LIG'*, *random_state=None*)

RandomLightRotationMove that provides methods for calculating properties on the object 'model' (i.e ligand) being perturbed in the NCMC simulation. Current methods calculate the object's atomic masses and center of masss. Calculating the object's center of mass will get the positions and total mass.

Parameters

- **resname** (*str*) – String specifying the residue name of the ligand.
- **structure** (*parmed.Structure*) – ParmEd Structure object of the relevant system to be moved.
- **random_state** (*integer or numpy.RandomState, optional*) – The generator used for random numbers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

structure

The structure of the ligand or selected atoms to be rotated.

Type parmed.Structure

resname

The residue name of the ligand or selected atoms to be rotated.

Type str, default='LIG'

topology

The topology of the ligand or selected atoms to be rotated.

Type openmm.Topology

atom_indices

Atom indicies of the ligand.

Type list

masses

Particle masses of the ligand with units.

Type list

totalmass

Total mass of the ligand.

Type int

center_of_mass

Calculated center of mass of the ligand in XYZ coordinates. This should be updated every iteration.

Type numpy.array

positions
Ligands positions in XYZ coordinates. This should be updated every iteration.

Type numpy.array

Examples

```
>>> from blues.move import RandomLigandRotationMove
>>> ligand = RandomLigandRotationMove(structure, 'LIG')
>>> ligand.resname
    'LIG'
```

getAtomIndices(*structure*, *resname*)
Get atom indices of a ligand from ParmEd Structure.

Parameters

- **resname** (*str*) – String specifying the residue name of the ligand.
- **structure** (*parmed.Structure*) – ParmEd Structure object of the atoms to be moved.

Returns **atom_indices** (*list of ints*) – list of atoms in the coordinate file matching lig_resname

getMasses(*topology*)
Returns a list of masses of the specified ligand atoms.

Parameters **topology** (*parmed.Topology*) – ParmEd topology object containing atoms of the system.

Returns

- **masses** (*1xn numpy.array * simtk.unit.dalton*) – array of masses of len(self.atom_indices), denoting the masses of the atoms in self.atom_indices
- **totalmass** (*float * simtk.unit.dalton*) – The sum of the mass found in masses

getCenterOfMass(*positions*, *masses*)
Returns the calculated center of mass of the ligand as a numpy.array

Parameters

- **positions** (*nx3 numpy array * simtk.unit compatible with simtk.unit.nanometers*) – ParmEd positions of the atoms to be moved.
- **masses** (*numpy.array*) – numpy.array of particle masses

Returns **center_of_mass** (*numpy array * simtk.unit compatible with simtk.unit.nanometers*) – 1x3 numpy.array of the center of mass of the given positions

move(*context*)
Function that performs a random rotation about the center of mass of the ligand.

Parameters **context** (*simtk.openmm.Context object*) – Context containing the positions to be moved.

Returns **context** (*simtk.openmm.Context object*) – The same input context, but whose positions were changed by this function.

3.1.3 MoveEngine

class blues.moves.**MoveEngine**(*moves*, *probabilities=None*)

MoveEngine provides perturbation functions for the context during the NCMC simulation.

Parameters

- **moves** (*blues.move object or list of N blues.move objects*) – Specifies the possible moves to be performed.
- **probabilities** (*list of floats, optional, default=None*) – A list of N probabilities, where probabilities[i] corresponds to the probaility of moves[i] being selected to perform its associated move() method. If None, uniform probabilities are assigned.

moves

Possible moves to be performed.

Type blues.move or list of N blues.move objects

probabilities

Normalized probabilities for each move.

Type list of floats

selected_move

Selected move to be performed.

Type blues.move

move_name

Name of the selected move to be performed

Type str

Examples

Load a parmed.Structure, list of moves with probabilities, initialize the MoveEngine class, and select a move from our list.

```
>>> import parmed
>>> from blues.moves import *
>>> structure = parmed.load_file('tests/data/eqToluene.prmtop', xyz='tests/data/
↪eqToluene.inpcrd')
>>> probabilities = [0.25, 0.75]
>>> moves = [SideChainMove(structure, [111]),RandomLigandRotationMove(structure,
↪'LIG')]
>>> mover = MoveEngine(moves, probabilities)
>>> mover.moves
[<blues.moves.SideChainMove at 0x7f2eaa168470>,
 <blues.moves.RandomLigandRotationMove at 0x7f2eaaaa51d0>]
>>> mover.selectMove()
>>> mover.selected_move
<blues.moves.RandomLigandRotationMove at 0x7f2eaaaa51d0>
```

selectMove()

Chooses the move which will be selected for a given NCMC iteration

runEngine(*context*)

Selects a random Move object based on its assigned probability and and performs its move() function on a context.

Parameters **context** (*openmm.Context*) – OpenMM context whose positions should be moved.

Returns **context** (*openmm.Context*) – OpenMM context whose positions have been moved.

3.1.4 Under Development

WARNING: The following move classes have not been tested. Use at your own risk.

class blues.moves.**SideChainMove**(*structure*, *residue_list*, *verbose=False*, *write_move=False*)

NOTE: Usage of this class requires a valid OpenEye license.

SideChainMove provides methods for calculating properties needed to rotate a sidechain residue given a parmed.Structure. Calculated properties include: backbone atom indicies, atom pointers and indicies of the residue sidechain, bond pointers and indices for rotatable heavy bonds in the sidechain, and atom indices upstream of selected bond.

The class contains functions to randomly select a bond and angle to be rotated and applies a rotation matrix to the target atoms to update their coordinates on the object 'model' (i.e sidechain) being perturbed in the NCMC simulation.

Parameters

- **structure** (*parmed.Structure*) – The structure of the entire system to be simulated.
- **residue_list** (*list of int*) – List of the residue numbers of the sidechains to be rotated.
- **verbose** (*bool, default=False*) – Enable verbosity to print out detailed information of the rotation.
- **write_move** (*bool, default=False*) – If True, writes a PDB of the system after rotation.

structure
The structure of the entire system to be simulated.

Type parmed.Structure

molecule
The OEMolecule containing the sidechain(s) to be rotated.

Type oechem.OEMolecule

residue_list
List containing the residue numbers of the sidechains to be rotated.

Type list of int

all_atoms
List containing the atom indicies of the sidechains to be rotated.

Type list of int

rot_atoms
Dictionary of residues, bonds and atoms to be rotated

Type dict

rot_bonds
Dictionary containing the bond pointers of the rotatable bonds.

Type dict of oechem.OEBondBase

qry_atoms
Dictionary containing all the atom pointers (as OpenEye objects) that make up the given residues.

Type dict of oechem.OEAtomBase

Examples

```
>>> from blues.move import SideChainMove
>>> sidechain = SideChainMove(structure, [1])
```

getBackboneAtoms (*molecule*)

Takes an OpenEye Molecule, finds the backbone atoms and returns the indicies of the backbone atoms.

Parameters **molecule** (*oechem.OEMolecule*) – The OEmolecule of the simulated system.

Returns **backbone_atoms** (*list of int*) – List containing the atom indices of the backbone atoms.

getTargetAtoms (*molecule*, *backbone_atoms*, *residue_list*)

Takes an OpenEye molecule and a list of residue numbers then generates a dictionary containing all the atom pointers and indicies for the non-backbone, atoms of those target residues, as well as a list of backbone atoms. Note: The atom indicies start at 0 and are thus -1 from the PDB file indicies

Parameters

- **molecule** (*oechem.OEMolecule*) – The OEmolecule of the simulated system.
- **backbone_atoms** (*list of int*) – List containing the atom indices of the backbone atoms.
- **residue_list** (*list of int*) – List containing the residue numbers of the sidechains to be rotated.

Returns

- **backbone_atoms** (*list of int*) – List containing the atom indices of the backbone atoms to be rotated.
- **qry_atoms** (*dict of oechem.OEAtomBase*) – Dictionary containing all the atom pointers (as OpenEye objects) that make up the given residues.

findHeavyRotBonds (*pdb_OEMol*, *qry_atoms*)

Takes in an OpenEye molecule as well as a dictionary of atom locations (keys) and atom indicies. It loops over the query atoms and identifies any heavy bonds associated with each atom. It stores and returns the bond indicies (keys) and the two atom indicies for each bond in a dictionary Note: atom indicies start at 0, so are offset by 1 compared to pdb)

Parameters

- **pdb_OEMol** (*oechem.OEMolecule*) – The OEmolecule of the simulated system generated from a PDB file.
- **qry_atoms** (*dict of oechem.OEAtomBase*) – Dictionary containing all the atom pointers (as OpenEye objects) that make up the given residues.

Returns **rot_bonds** (*dict of oechem.OEBondBase*) – Dictionary containing the bond pointers of the rotatable bonds.

getRotAtoms (*rotbonds*, *molecule*, *backbone_atoms*)

Function identifies and stores neighboring, upstream atoms for a given sidechain bond.

Parameters

- **rot_bonds** (*dict of oechem.OEBondBase*) – Dictionary containing the bond pointers of the rotatable bonds.
- **molecule** (*oechem.OEMolecule*) – The OEmolecule of the simulated system.
- **backbone_atoms** (*list of int*) – List containing the atom indices of the backbone atoms.

Returns **rot_atom_dict** (*dict of oechem.OEAtomBase*) – Dictionary containing the atom pointers for a given sidechain bond.

getRotBondAtoms()
This function is called on class initialization.

Takes in a PDB filename (as a string) and list of residue numbers. Returns a nested dictionary of rotatable bonds (containing only heavy atoms), that are keyed by residue number, then keyed by bond pointer, containing values of atom indicies [axis1, axis2, atoms to be rotated] Note: The atom indicies start at 0, and are offset by -1 from the PDB file indicies

Returns

- **rot_atoms** (*dict*) – Dictionary of residues, bonds and atoms to be rotated
- **rot_bonds** (*dict of oechem.OEBondBase*) – Dictionary containing the bond pointers of the rotatable bonds.
- **qry_atoms** (*dict of oechem.OEAtomBase*) – Dictionary containing all the atom pointers (as OpenEye objects) that make up the given residues.

chooseBondandTheta()
This function is called on class initialization.

Takes a dictionary containing nested dictionary, keyed by res#, then keyed by bond_ptrs, containing a list of atoms to move, randomly selects a bond, and generates a random angle (radians). It returns the atoms associated with the the selected bond, the pointer for the selected bond and the randomly generated angle

Returns

- *theta_ran*
- *targetatoms*
- *res_choice*
- *bond_choice*

rotation_matrix(*axis*, *theta*)
Function returns the rotation matrix associated with counterclockwise rotation about the given axis by theta radians.

Parameters

- **axis**
- **theta** (*float*) – The angle of rotation in radians.

move(*context*, *verbose=False*)
Rotates the target atoms around a selected bond by angle theta and updates the atom coordinates in the parmed structure as well as the ncmc context object

Parameters

- **context** (*simtk.openmm.Context object*) – Context containing the positions to be moved.
- **verbose** (*bool, default=False*) – Enable verbosity to print out detailed information of the rotation.

Returns **context** (*simtk.openmm.Context object*) – The same input context, but whose positions were changed by this function.

class blues.moves.**SmartDartMove**(*structure*, *basis_particles*, *coord_files*, *topology=None*, *dart_radius=Quantity(value=0.2, unit=nanometer)*, *self_dart=False*, *resname='LIG'*)
WARNING: This class has not been completely tested. Use at your own risk.

Move object that allows center of mass smart darting moves to be performed on a ligand, allowing translations of a ligand between pre-defined regions in space. The *SmartDartMove.move()* method translates the ligand to the locations of the ligand found in the coord_files. These locations are defined in terms of the basis_particles. These locations are picked with a uniform probability. Based on Smart Darting Monte Carlo [smart-dart]

Parameters

- **structure** (*parmed.Structure*) – ParmEd Structure object of the relevant system to be moved.
- **basis_particles** (*list of 3 ints*) – Specifies the 3 indices of the protein whose coordinates will be used to define a new set of basis vectors.
- **coord_files** (*list of str*) – List containing paths to coordinate files of the whole system for smart darting.
- **topology** (*str, optional, default=None*) – A path specifying a topology file matching the files in coord_files. Not necessary if the coord_files already contain topologies (ex. PDBs).
- **dart_radius** (*simtk.unit float object compatible with simtk.unit.nanometers unit,*) – optional, default=0.2*simtk.unit.nanometers The radius of the darting region around each dart.
- **self_dart** (*boolean, optional, default='False'*) – When performing the center of mass darting in *SmartDartMove.move()*,this specifies whether or not to include the darting region where the center of mass currently resides as an option to dart to.
- **resname** (*str, optional, default='LIG'*) – String specifying the residue name of the ligand.

References

dartsFromParmEd(*coord_files*, *topology=None*)

Used to setup darts from a generic coordinate file, through MDtraj using the basis_particles to define new basis vectors, which allows dart centers to remain consistant through a simulation. This adds to the self.n_dartboard, which defines the centers used for smart darting.

Parameters

- **coord_files** (*list of str*) – List containing coordinate files of the whole system for smart darting.
- **topology** (*str, optional, default=None*) – A path specifying a topology file matching the files in coord_files. Not necessary if the coord_files already contain topologies.

move(*context*)

Function for performing smart darting move with darts that depend on particle positions in the system.

Parameters **context** (*simtk.openmm.Context object*) – Context containing the positions to be moved.

Returns **context** (*simtk.openmm.Context object*) – The same input context, but whose positions were changed by this function.

class blues.moves.**CombinationMove**(*moves*)

WARNING: This class has not been completely tested. Use at your own risk.

Move object that allows Move object moves to be performed according to the order in move_list. To ensure detailed balance, the moves have an equal chance to be performed in listed or reverse order.

Parameters **moves** (*list of blues.move.Move*)

move(*context*)

Performs the move() functions of the Moves in move_list on a context.

Parameters **context** (*simtk.openmm.Context object*) – Context containing the positions to be moved.

Returns **context** (*simtk.openmm.Context object*) – The same input context, but whose positions were changed by this function.

3.2 Simulation

Provides classes for setting up and running the BLUES simulation.

- *SystemFactory* : setup and modifying the OpenMM System prior to the simulation.
- *SimulationFactory* : generates the OpenMM Simulations from the System.
- *BLUESSimulation* : runs the NCMC+MD hybrid simulation.
- *MonteCarloSimulation* : runs a pure Monte Carlo simulation.

Authors: Samuel C. Gill Contributors: Nathan M. Lim, Meghan Osato, David L. Mobley

3.2.1 SystemFactory

Methods

---

class blues.simulation.**SystemFactory**(*structure*, *atom_indices*, *config=None*)

SystemFactory contains methods to generate/modify the OpenMM System object required for generating the openmm.Simulation using a given parmed.Structure()

Examples

Load Parmed Structure, select move type, initialize *MoveEngine*, and generate the openmm.Systems

```
>>> structure = parmed.load_file('eqToluene.prmtop', xyz='eqToluene.inpcrd')
>>> ligand = RandomLigandRotationMove(structure, 'LIG')
>>> ligand_mover = MoveEngine(ligand)
>>> systems = SystemFactory(structure, ligand.atom_indices, config['system'])
```

The MD and alchemical Systems are generated and stored as an attribute

```
>>> systems.md
>>> systems.alch
```

Freeze atoms in the alchemical system

```
>>> systems.alch = SystemFactory.freeze_atoms(systems.alch,
                                              freeze_distance=5.0,
                                              freeze_center='LIG'
                                              freeze_solvent='HOH,NA,CL')
```

Parameters

- **structure** (*parmed.Structure*) – A chemical structure composed of atoms, bonds, angles, torsions, and other topological features.

- **atom_indices** (*list of int*) – Atom indicies of the move or designated for which the nonbonded forces (both sterics and electrostatics components) have to be alchemically modified.
- **config** (dict, parameters for generating the *openmm.System* for the MD) – and NCMC simulation. For complete parameters, see docs for *generateSystem* and *generateAlchSystem*

static amber_selection_to_atomidx(*structure*, *selection*)
Converts AmberMask selection [amber-syntax] to list of atom indices.

Parameters

- **structure** (*parmed.Structure()*) – Structure of the system, used for atom selection.
- **selection** (*str*) – AmberMask selection that gets converted to a list of atom indices.

Returns **mask_idx** (*list of int*) – List of atom indices.

References

static atomidx_to_atomlist(*structure*, *mask_idx*)
Goes through the structure and matches the previously selected atom indices to the atom type.

Parameters

- **structure** (*parmed.Structure()*) – Structure of the system, used for atom selection.
- **mask_idx** (*list of int*) – List of atom indices.

Returns **atom_list** (*list of atoms*) – The atoms that were previously selected in mask_idx.

classmethod generateSystem(*structure*, ***kwargs*)
Construct an OpenMM System representing the topology described by the prmtop file. This function is just a wrapper for parmed Structure.createSystem().

Parameters

- **structure** (*parmed.Structure()*) – The parmed.Structure of the molecular system to be simulated
- **nonbondedMethod** (*cutoff method*) – This is the cutoff method. It can be either the NoCutoff, CutoffNonPeriodic, CutoffPeriodic, PME, or Ewald objects from the simtk.openmm.app namespace
- **nonbondedCutoff** (*float or distance Quantity*) – The nonbonded cutoff must be either a floating point number (interpreted as nanometers) or a Quantity with attached units. This is ignored if nonbondedMethod is NoCutoff.
- **switchDistance** (*float or distance Quantity*) – The distance at which the switching function is turned on for van der Waals interactions. This is ignored when no cutoff is used, and no switch is used if switchDistance is 0, negative, or greater than the cutoff
- **constraints** (*None, app.HBonds, app.HAngles, or app.AllBonds*) – Which type of constraints to add to the system (e.g., SHAKE). None means no bonds are constrained. HBonds means bonds with hydrogen are constrained
- **rigidWater** (*bool=True*) – If True, water is kept rigid regardless of the value of constraints. A value of False is ignored if constraints is not None.
- **implicitSolvent** (*None, app.HCT, app.OBC1, app.OBC2, app.GBn, app.GBn2*) – The Generalized Born implicit solvent model to use.

- **implicitSolventKappa** (*float or 1/distance Quantity = None*) – This is the Debye kappa property related to modeling saltwater conditions in GB. It should have units of 1/distance (1/nanometers is assumed if no units present). A value of None means that kappa will be calculated from implicitSolventSaltConc (below)
- **implicitSolventSaltConc** (*float or amount/volume Quantity=0 moles/liter*) – If implicitSolventKappa is None, the kappa will be computed from the salt concentration. It should have units compatible with mol/L
- **temperature** (*float or temperature Quantity = 298.15 kelvin*) – This is only used to compute kappa from implicitSolventSaltConc
- **soluteDielectric** (*float=1.0*) – The dielectric constant of the protein interior used in GB
- **solventDielectric** (*float=78.5*) – The dielectric constant of the water used in GB
- **useSASA** (*bool=False*) – If True, use the ACE non-polar solvation model. Otherwise, use no SASA-based nonpolar solvation model.
- **removeCMMotion** (*bool=True*) – If True, the center-of-mass motion will be removed periodically during the simulation. If False, it will not.
- **hydrogenMass** (*float or mass quantity = None*) – If not None, hydrogen masses will be changed to this mass and the difference subtracted from the attached heavy atom (hydrogen mass repartitioning)
- **ewaldErrorTolerance** (*float=0.0005*) – When using PME or Ewald, the Ewald parameters will be calculated from this value
- **flexibleConstraints** (*bool=True*) – If False, the energies and forces from the constrained degrees of freedom will NOT be computed. If True, they will (but those degrees of freedom will *still* be constrained).
- **verbose** (*bool=False*) – If True, the progress of this subroutine will be printed to stdout
- **splitDihedrals** (*bool=False*) – If True, the dihedrals will be split into two forces – proper and impropers. This is primarily useful for debugging torsion parameter assignments.

Returns *openmm.System* – System formatted according to the prmtop file.

Notes

This function calls prune_empty_terms if any Topology lists have changed.

classmethod generateAlchSystem(*system*, *atom_indices*, *softcore_alpha=0.5*, *softcore_a=1*, *softcore_b=1*, *softcore_c=6*, *softcore_beta=0.0*, *softcore_d=1*, *softcore_e=1*, *softcore_f=2*, *annihilate_electrostatics=True*, *annihilate_sterics=False*, *disable_alchemical_dispersion_correction=True*, *alchemical_pme_treatment='direct-space'*, *suppress_warnings=True*, ***kwargs*)

Returns the OpenMM System for alchemical perturbations. This function calls *openmmtools.alchemy.AbsoluteAlchemicalFactory* and *openmmtools.alchemy.AlchemicalRegion* to generate the System for the NCMC simulation.

Parameters

- **system** (*openmm.System*) – The OpenMM System object corresponding to the reference system.

- **atom_indices** (*list of int*) – Atom indicies of the move or designated for which the non-bonded forces (both sterics and electrostatics components) have to be alchemically modified.
- **annihilate_electrostatics** (*bool, optional*) – If True, electrostatics should be annihilated, rather than decoupled (default is True).
- **annihilate_sterics** (*bool, optional*) – If True, sterics (Lennard-Jones or Halgren potential) will be annihilated, rather than decoupled (default is False).
- **softcore_alpha** (*float, optional*) – Alchemical softcore parameter for Lennard-Jones (default is 0.5).
- **softcore_a, softcore_b, softcore_c** (*float, optional*) – Parameters modifying softcore Lennard-Jones form. Introduced in Eq. 13 of Ref. [TTPham-JChemPhys135-2011] (default is 1).
- **softcore_beta** (*float, optional*) – Alchemical softcore parameter for electrostatics. Set this to zero to recover standard electrostatic scaling (default is 0.0).
- **softcore_d, softcore_e, softcore_f** (*float, optional*) – Parameters modifying softcore electrostatics form (default is 1).
- **disable_alchemical_dispersion_correction** (*bool, optional, default=True*) – If True, the long-range dispersion correction will not be included for the alchemical region to avoid the need to recompute the correction (a CPU operation that takes ~ 0.5 s) every time 'lambda_sterics' is changed. If using nonequilibrium protocols, it is recommended that this be set to True since this can lead to enormous (100x) slowdowns if the correction must be recomputed every time step.
- **alchemical_pme_treatment** (*str, optional, default = 'direct-space'*) – Controls how alchemical region electrostatics are treated when PME is used. Options are 'direct-space', 'coulomb', 'exact'. - 'direct-space' only models the direct space contribution - 'coulomb' includes switched Coulomb interaction - 'exact' includes also the reciprocal space contribution, but it's only possible to annihilate the charges and the softcore parameters controlling the electrostatics are deactivated. Also, with this method, modifying the global variable *lambda_electrostatics* is not sufficient to control the charges. The recommended way to change them is through the *AlchemicalState* class.

Returns **alch_system** (*alchemical_system*) – System to be used for the NCMC simulation.

References

classmethod restrain_positions(*structure*, *system*, *selection='(@CA,C,N)'*, *weight=5.0*, ***kwargs*)

Applies positional restraints to atoms in the openmm.System by the given parmed selection [amber-syntax].

Parameters

- **system** (*openmm.System*) – The OpenMM System object to be modified.
- **structure** (*parmed.Structure()*) – Structure of the system, used for atom selection.
- **selection** (*str, Default = "(@CA,C,N)"*) – AmberMask selection to apply positional restraints to
- **weight** (*float, Default = 5.0*) – Restraint weight for xyz atom restraints in kcal/(mol A^2)

Returns **system** (*openmm.System*) – Modified with positional restraints applied.

classmethod freeze_atoms(*structure*, *system*, *freeze_selection=':LIG'*, ***kwargs*)

Zeroes the masses of atoms from the given parmed selection [amber-syntax]. Massless atoms will be ignored by the integrator and will not change positions.

Parameters

- **system** (*openmm.System*) – The OpenMM System object to be modified.
- **structure** (*parmed.Structure()*) – Structure of the system, used for atom selection.
- **freeze_selection** (*str, Default = ":LIG"*) – AmberMask selection for the center in which to select atoms for zeroing their masses. Defaults to freezing protein backbone atoms.

Returns **system** (*openmm.System*) – The modified system with the selected atoms

classmethod freeze_radius(*structure*, *system*, *freeze_distance=Quantity(value=5.0, unit=angstrom)*, *freeze_center=':LIG'*, *freeze_solvent=':HOH,NA,CL'*, ***kwargs*)

Zero the masses of atoms outside the given raidus of the *freeze_center* parmed selection [amber-syntax]. Massless atoms will be ignored by the integrator and will not change positions.This is intended to freeze the solvent and protein atoms around the ligand binding site.

Parameters

- **system** (*openmm.System*) – The OpenMM System object to be modified.
- **structure** (*parmed.Structure()*) – Structure of the system, used for atom selection.
- **freeze_distance** (*float, Default = 5.0*) – Distance (angstroms) to select atoms for retaining their masses. Atoms outside the set distance will have their masses set to 0.0.
- **freeze_center** (*str, Default = ":LIG"*) – AmberMask selection for the center in which to select atoms for zeroing their masses. Default: LIG
- **freeze_solvent** (*str, Default = ":HOH,NA,CL"*) – AmberMask selection in which to select solvent atoms for zeroing their masses.

Returns **system** (*openmm.System*) – Modified system with masses outside the *freeze center* zeroed.

3.2.2 SimulationFactory

Methods

—

class blues.simulation.**SimulationFactory**(*systems*, *move_engine*, *config=None*, *md_reporters=None*, *ncmc_reporters=None*)

SimulationFactory is used to generate the 3 required OpenMM Simulation objects (MD, NCMC, ALCH) required for the BLUES run. This class can take a list of reporters for the MD or NCMC simulation in the arguments *md_reporters* or *ncmc_reporters*.

Parameters

- **systems** (*blues.simulation.SystemFactory object*) – The object containing the MD and alchemical openmm.Systems
- **move_engine** (*blues.moves.MoveEngine object*) – MoveEngine object which contains the list of moves performed in the NCMC simulation.
- **config** (*dict*) – Simulation parameters which include: nIter, nstepsNC, nstepsMD, nprop, propLambda, temperature, dt, propSteps, write_move

- **md_reporters** (*(optional) list of Reporter objects for the MD openmm.Simulation*)
- **ncmc_reporters** (*(optional) list of Reporter objects for the NCMC openmm.Simulation*)

Examples

Load Parmed Structure from our input files, select the move type, initialize the MoveEngine, and generate the openmm systems.

```
>>> structure = parmed.load_file('eqToluene.prmtop', xyz='eqToluene.inpcrd')
>>> ligand = RandomLigandRotationMove(structure, 'LIG')
>>> ligand_mover = MoveEngine(ligand)
>>> systems = SystemFactory(structure, ligand.atom_indices, config['system'])
```

Now, we can generate the Simulations from our openmm Systems using the SimulationFactory class. If a configuration is provided at on initialization, it will call *generateSimulationSet()* for convenience. Otherwise, the class can be instantiated like a normal python class.

Below is an example of initializing the class like a normal python object.

```
>>> simulations = SimulationFactory(systems, ligand_mover)
>>> hasattr(simulations, 'md')
False
>>> hasattr(simulations, 'ncmc')
False
```

Below, we provide a dict for configuring the Simulations and then generate them by calling *simulations.generateSimulationSet()*. The MD/NCMC simulation objects can be accessed separately as class attributes.

```
>>> sim_cfg = { 'platform': 'OpenCL',
            'properties' : { 'OpenCLPrecision': 'single',
                             'OpenCLDeviceIndex' : 2},
            'nprop' : 1,
            'propLambda' : 0.3,
            'dt' : 0.001 * unit.picoseconds,
            'friction' : 1 * 1/unit.picoseconds,
            'temperature' : 100 * unit.kelvin,
            'nIter': 1,
            'nstepsMD': 10,
            'nstepsNC': 10,}
>>> simulations.generateSimulationSet(sim_cfg)
>>> hasattr(simulations, 'md')
True
>>> hasattr(simulations, 'ncmc')
True
```

After generating the Simulations, attach your own reporters by providing the reporters in a list. Be sure to attach to either the MD or NCMC simulation.

```
>>> from simtk.openmm.app import StateDataReporter
>>> md_reporters = [ StateDataReporter('test.log', 5) ]
>>> ncmc_reporters = [ StateDataReporter('test-ncmc.log', 5) ]
>>> simulations.md = simulations.attachReporters( simulations.md, md_reporters)
>>> simulations.ncmc = simulations.attachReporters( simulations.ncmc, ncmc_
↪reporters)
```

Alternatively, you can pass the configuration dict and list of reporters upon class initialization. This will do all of the above for convenience.

```
>>> simulations = SimulationFactory(systems, ligand_mover, sim_cfg,
                                    md_reporters, ncmc_reporters)
>>> print(simulations)
>>> print(simulations.md)
>>> print(simulations.ncmc)
<blues.simulation.SimulationFactory object at 0x7f461b7a8b00>
<simtk.openmm.app.simulation.Simulation object at 0x7f461b7a8780>
<simtk.openmm.app.simulation.Simulation object at 0x7f461b7a87b8>
>>> print(simulations.md.reporters)
>>> print(simulations.ncmc.reporters)
[<simtk.openmm.app.statedatareporter.StateDataReporter object at 0x7f1b4d24cac8>]
[<simtk.openmm.app.statedatareporter.StateDataReporter object at 0x7f1b4d24cb70>]
```

classmethod addBarostat(*system*, *temperature=Quantity(value=300, unit=kelvin)*, *pressure=Quantity(value=1, unit=atmosphere)*, *frequency=25*, ***kwargs*)

Adds a MonteCarloBarostat to the MD system.

Parameters

- **system** (*openmm.System*) – The OpenMM System object corresponding to the reference system.
- **temperature** (*float, default=300*) – temperature (Kelvin) to be simulated at.
- **pressure** (*int, configional, default=None*) – Pressure (atm) for Barostat for NPT simulations.
- **frequency** (*int, default=25*) – Frequency at which Monte Carlo pressure changes should be attempted (in time steps)

Returns **system** (*openmm.System*) – The OpenMM System with the MonteCarloBarostat attached.

classmethod generateIntegrator(*temperature=Quantity(value=300, unit=kelvin)*, *dt=Quantity(value=0.002, unit=picosecond)*, *friction=1*, ***kwargs*)

Generates a LangevinIntegrator for the Simulations.

Parameters

- **temperature** (*float, default=300*) – temperature (Kelvin) to be simulated at.
- **friction** (*float, default=1*) – friction coefficient which couples to the heat bath, measured in 1/ps
- **dt** (*int, configional, default=0.002*) – The timestep of the integrator to use (in ps).

Returns **integrator** (*openmm.LangevinIntegrator*) – The LangevinIntegrator object intended for the System.

classmethod generateNCMCIntegrator(*nstepsNC=None*, *alchemical_functions={'lambda_electrostatics': 'step(0.2-lambda) - 1/0.2*lambda*step(0.2-lambda) + 1/0.2*(lambda-0.8)*step(lambda-0.8)', 'lambda_sterics': 'min(1, (1/0.3)*abs(lambda-0.5))'}*, *splitting='H V R O R V H'*, *temperature=Quantity(value=300, unit=kelvin)*, *dt=Quantity(value=0.002, unit=picosecond)*, *nprop=1*, *propLambda=0.3*, ***kwargs*)

Generates the AlchemicalExternalLangevinIntegrator using openmmtools.

Parameters

- **nstepsNC** (*int*) – The number of NCMC relaxation steps to use.
- **alchemical_functions** (*dict*) – default = *{'lambda_sterics' : 'min(1, (1/0.3)*abs(lambda-0.5))', lambda_electrostatics' : 'step(0.2-lambda) - 1/0.2*lambda*step(0.2-lambda) + 1/0.2*(lambda-0.8)*step(lambda-0.8)'}* key : value pairs such as "global_parameter" : function_of_lambda where function_of_lambda is a Lepton-compatible string that depends on the variable "lambda".
- **splitting** (*string, default: "H V R O R V H"*) – Sequence of R, V, O (and optionally V{i}), and { }substeps to be executed each timestep. There is also an H option, which increments the global parameter *lambda* by 1/nsteps_neq for each step. Forces are only used in V-step. Handle multiple force groups by appending the force group index to V-steps, e.g. "V0" will only use forces from force group 0. "V" will perform a step using all forces. (will cause metropolization, and must be followed later by a).
- **temperature** (*float, default=300*) – temperature (Kelvin) to be simulated at.
- **dt** (*int, optional, default=0.002*) – The timestep of the integrator to use (in ps).
- **nprop** (*int (Default: 1)*) – Controls the number of propagation steps to add in the lambda region defined by *propLambda*
- **propLambda** (*float, optional, default=0.3*) – The range which additional propogation steps are added, defined by [0.5-propLambda, 0.5+propLambda].

Returns ncmc_integrator (*blues.integrator.AlchemicalExternalLangevinIntegrator*) – The NCMC integrator for the alchemical process in the NCMC simulation.

classmethod generateSimFromStruct(*structure*, *system*, *integrator*, *platform=None*, *properties={}*, ***kwargs*)

Generate the OpenMM Simulation objects from a given parmed.Structure()

Parameters

- **structure** (*parmed.Structure*) – ParmEd Structure object of the entire system to be simulated.
- **system** (*openmm.System*) – The OpenMM System object corresponding to the reference system.
- **integrator** (*openmm.Integrator*) – The OpenMM Integrator object for the simulation.
- **platform** (*str, default = None*) – Valid choices: 'Auto', 'OpenCL', 'CUDA' If None is specified, the fastest available platform will be used.

Returns simulation (*openmm.Simulation*) – The generated OpenMM Simulation from the parmed.Structure, openmm.System, amd the integrator.

static attachReporters(*simulation*, *reporter_list*)

Attach the list of reporters to the Simulation object

Parameters

- **simulation** (*openmm.Simulation*) – The Simulation object to attach reporters to.
- **reporter_list** (*list of openmm Reporeters*) – The list of reporters to attach to the OpenMM Simulation.

Returns simulation (*openmm.Simulation*) – The Simulation object with the reporters attached.

generateSimulationSet(*config=None*)

Generates the 3 OpenMM Simulation objects.

Parameters **config** (*dict*) – Dictionary of parameters for configuring the OpenMM Simulations

3.2.3 BLUESSimulation

class blues.simulation.**BLUESSimulation**(*simulations*, *config=None*)

BLUESSimulation class provides methods to execute the NCMC+MD simulation.

Parameters

- **simulations** (*blues.simulation.SimulationFactory object*) – SimulationFactory Object which carries the 3 required OpenMM Simulation objects (MD, NCMC, ALCH) required to run BLUES.
- **config** (*dict*) – Dictionary of parameters for configuring the OpenMM Simulations If None, will search for configuration parameters on the *simulations* object.

Examples

Create our SimulationFactory object and run *BLUESSimulation*

```
>>> sim_cfg = { 'platform': 'OpenCL',
                'properties' : { 'OpenCLPrecision': 'single',
                                 'OpenCLDeviceIndex' : 2},
                'nprop' : 1,
                'propLambda' : 0.3,
                'dt' : 0.001 * unit.picoseconds,
                'friction' : 1 * 1/unit.picoseconds,
                'temperature' : 100 * unit.kelvin,
                'nIter': 1,
                'nstepsMD': 10,
                'nstepsNC': 10,}
>>> simulations = SimulationFactory(systems, ligand_mover, sim_cfg)
>>> blues = BLUESSimulation(simulations)
>>> blues.run()
```

classmethod getStateFromContext(*context*, *state_keys*)

Gets the State information from the given context and list of state_keys to query it with.

Returns the state data as a dict.

Parameters

- **context** (*openmm.Context*) – Context of the OpenMM Simulation to query.
- **state_keys** (*list*) – Default: [positions, velocities, potential_energy, kinetic_energy] A list that defines what information to get from the context State.

Returns **stateinfo** (*dict*) – Current positions, velocities, energies and box vectors of the context.

classmethod getIntegratorInfo(*ncmc_integrator*, *integrator_keys=['lambda', 'shadow_work', 'protocol_work', 'Eold', 'Enew']*)

Returns a dict of alchemical/ncmc-swtiching data from querying the the NCMC integrator.

Parameters

- **ncmc_integrator** (*openmm.Context.Integrator*) – The integrator from the NCMC Context
- **integrator_keys** (*list*) – list containing strings of the values to get from the integrator. Default = [‘lambda’, ‘shadow_work’, ‘protocol_work’, ‘Eold’, ‘Enew’,’Epert’]

Returns **integrator_info** (*dict*) – Work values and energies from the NCMC integrator.

classmethod setContextFromState(*context*, *state*, *box=True*, *positions=True*, *velocities=True*)

Update a given Context from the given State.

Parameters

- **context** (*openmm.Context*) – The Context to be updated from the given State.
- **state** (*openmm.State*) – The current state (box_vectors, positions, velocities) of the Simulation to update the given context.

Returns **context** (*openmm.Context*) – The updated Context whose box_vectors, positions, and velocities have been updated.

_printSimulationTiming()

Prints the simulation timing and related information.

_setStateTable(*simkey*, *stateidx*, *stateinfo*)

Updates *stateTable* (dict) containing: Positions, Velocities, Potential/Kinetic energies of the state before and after a NCMC step or iteration.

Parameters

- **simkey** (*str (key: 'md', 'ncmc', 'alch')*) – Key corresponding to the simulation.
- **stateidx** (*str (key: 'state0' or 'state1')*) – Key corresponding to the state information being stored.
- **stateinfo** (*dict*) – Dictionary containing the State information.

_syncStatesMDtoNCMC()

Retrieves data on the current State of the MD context to replace the box vectors, positions, and velocties in the NCMC context.

_stepNCMC(*nstepsNC*, *moveStep*, *move_engine=None*)

Advance the NCMC simulation.

Parameters

- **nstepsNC** (*int*) – The number of NCMC switching steps to advance by.
- **moveStep** (*int*) – The step number to perform the chosen move, which should be half the number of nstepsNC.
- **move_engine** (*blues.moves.MoveEngine*) – The object that executes the chosen move.

_computeAlchemicalCorrection()

Computes the alchemical correction term from switching between the NCMC and MD potentials.

_acceptRejectMove(*write_move=False*)

Choose to accept or reject the proposed move based on the acceptance criterion.

Parameters **write_move** (*bool, default=False*) – If True, writes the proposed NCMC move to a PDB file.

_resetSimulations(*temperature=None*)

At the end of each iteration:

1. Reset the step number in the NCMC context/integrator
2. Set the velocities to random values chosen from a Boltzmann distribution at a given *temperature*.

Parameters **temperature** (*float*) – The target temperature for the simulation.

_stepMD (*nstepsMD*)
Advance the MD simulation.

Parameters **nstepsMD** (*int*) – The number of steps to advance the MD simulation.

run (*nIter=0*, *nstepsNC=0*, *moveStep=0*, *nstepsMD=0*, *temperature=300*, *write_move=False*, ***config*)
Executes the BLUES engine to iterate over the actions: Perform NCMC simulation, perform proposed move, accepts/rejects move, then performs the MD simulation from the NCMC state, niter number of times. **Note:** If the parameters are not given explicitly, will look for the parameters in the provided configuration on the *SimulationFactory* object.

Parameters

- **nIter** (*int, default = None*) – Number of iterations of NCMC+MD to perform.
- **nstepsNC** (*int*) – The number of NCMC switching steps to advance by.
- **moveStep** (*int*) – The step number to perform the chosen move, which should be half the number of nstepsNC.
- **nstepsMD** (*int*) – The number of steps to advance the MD simulation.
- **temperature** (*float*) – The target temperature for the simulation.
- **write_move** (*bool, default=False*) – If True, writes the proposed NCMC move to a PDB file.

3.3 Integrators

class blues.integrators.**AlchemicalExternalLangevinIntegrator**(*alchemical_functions*, *splitting='R V O H O V R'*, *temperature=Quantity(value=298.0, unit=kelvin)*, *collision_rate=Quantity(value=1.0, unit=/picosecond)*, *timestep=Quantity(value=1.0, unit=femtosecond)*, *constraint_tolerance=1e-08*, *measure_shadow_work=False*, *measure_heat=True*, *nsteps_neq=100*, *nprop=1*, *prop_lambda=0.3*, **args*, ***kwargs*)
Allows nonequilibrium switching based on force parameters specified in alchemical_functions. A variable named lambda is switched from 0 to 1 linearly throughout the nsteps of the protocol. The functions can use this to create more complex protocols for other global parameters.

As opposed to *openmmtools.integrators.AlchemicalNonequilibriumLangevinIntegrator*, which this inherits from, the AlchemicalExternalLangevinIntegrator integrator also takes into account work done outside the nonequilibrium switching portion(between integration steps). For example if a molecule is rotated between integration steps, this integrator would correctly account for the work caused by that rotation.

Propagator is based on Langevin splitting, as described below. One way to divide the Langevin system is into three parts which can each be solved "exactly:"

- **R: Linear "drift" / Constrained "drift"** Deterministic update of *positions*, using current velocities `x <- x + v dt`
- **V: Linear "kick" / Constrained "kick"** Deterministic update of *velocities*, using current forces `v <- v + (f/m) dt`; where f = force, m = mass
- **O: Ornstein-Uhlenbeck** Stochastic update of velocities, simulating interaction with a heat bath `v <- av + b sqrt(kT/m) R` where:
  - a = e^(-gamma dt)
  - b = sqrt(1 - e^(-2gamma dt))
  - R is i.i.d. standard normal

We can then construct integrators by solving each part for a certain timestep in sequence. (We can further split up the V step by force group, evaluating cheap but fast-fluctuating forces more frequently than expensive but slow-fluctuating forces. Since forces are only evaluated in the V step, we represent this by including in our "alphabet" V0, V1, ...) When the system contains holonomic constraints, these steps are confined to the constraint manifold.

Parameters

- **alchemical_functions** (*dict of strings*) – key: value pairs such as "global_parameter" : function_of_lambda where function_of_lambda is a Lepton-compatible string that depends on the variable "lambda"
- **splitting** (*string, default: "H V R O V R H"*) – Sequence of R, V, O (and optionally V{i}), and { }substeps to be executed each timestep. There is also an H option, which increments the global parameter *lambda* by 1/nsteps_neq for each step. Forces are only used in V-step. Handle multiple force groups by appending the force group index to V-steps, e.g. "V0" will only use forces from force group 0. "V" will perform a step using all forces.(will cause metropolization, and must be followed later by a).
- **temperature** (*numpy.unit.Quantity compatible with kelvin, default: 298.0*simtk.unit.kelvin*) – Fictitious "bath" temperature
- **collision_rate** (*numpy.unit.Quantity compatible with 1/picoseconds, default: 91.0/simtk.unit.picoseconds*) – Collision rate
- **timestep** (*numpy.unit.Quantity compatible with femtoseconds, default: 1.0*simtk.unit.femtoseconds*) – Integration timestep
- **constraint_tolerance** (*float, default: 1.0e-8*) – Tolerance for constraint solver
- **measure_shadow_work** (*boolean, default: False*) – Accumulate the shadow work performed by the symplectic substeps, in the global *shadow_work*
- **measure_heat** (*boolean, default: True*) – Accumulate the heat exchanged with the bath in each step, in the global *heat*
- **nsteps_neq** (*int, default: 100*) – Number of steps in nonequilibrium protocol. Default 100
- **prop_lambda** (*float (Default = 0.3)*) – Defines the region in which to add extra propagation steps during the NCMC simulation from the midpoint 0.5. i.e. A value of 0.3 will add extra steps from lambda 0.2 to 0.8.
- **nprop** (*int (Default: 1)*) – Controls the number of propagation steps to add in the lambda region defined by *prop_lambda*.

_kinetic_energy

This is 0.5*m*v*v by default, and is the expression used for the kinetic energy

Type str

Examples

- **g-BAOAB:** splitting="R V O H O V R"
- **VVVR** splitting="O V R H R V O"
- **VV** splitting="V R H R V"
- **An NCMC algorithm with Metropolized integrator:** splitting="O { V R H R V } O"

References

[Nilmeier, et al. 2011] Nonequilibrium candidate Monte Carlo is an efficient tool for equilibrium simulation

[Leimkuhler and Matthews, 2015] Molecular dynamics: with deterministic and stochastic numerical methods, Chapter 7

reset()
Reset all statistics, alchemical parameters, and work.

3.4 Utilities

Provides a host of utility functions for the BLUES engine.

Authors: Samuel C. Gill Contributors: Nathan M. Lim, David L. Mobley

blues.utils.**saveSimulationFrame**(*simulation*, *outfname*)
Extracts a ParmEd structure and writes the frame given an OpenMM Simulation object.

Parameters

- **simulation** (*openmm.Simulation*) – The OpenMM Simulation to write a frame from.
- **outfname** (*str*) – The output file name to save the simulation frame from. Supported extensions:
  - PDB (.pdb, pdb)
  - PDBx/mmCIF (.cif, cif)
  - PQR (.pqr, pqr)
  - Amber topology file (.prmtop/.parm7, amber)
  - CHARMM PSF file (.psf, psf)
  - CHARMM coordinate file (.crd, charmmcrd)
  - Gromacs topology file (.top, gromacs)
  - Gromacs GRO file (.gro, gro)
  - Mol2 file (.mol2, mol2)
  - Mol3 file (.mol3, mol3)
  - Amber ASCII restart (.rst7/.inpcrd/.restrt, rst7)
  - Amber NetCDF restart (.ncrst, ncrst)

blues.utils.**print_host_info**(*simulation*)
Prints hardware related information for the openmm.Simulation

Parameters **simulation** (*openmm.Simulation*) – The OpenMM Simulation to write a frame from.

blues.utils.**calculateNCMCSteps**(*nstepsNC=0*, *nprop=1*, *propLambda=0.3*, ***kwargs*)
Calculates the number of NCMC switching steps.

Parameters

- **nstepsNC** (*int*) – The number of NCMC switching steps
- **nprop** (*int, default=1*) – The number of propagation steps per NCMC switching steps
- **propLambda** (*float, default=0.3*) – The lambda values in which additional propagation steps will be added or 0.5 +/- propLambda. If 0.3, this will add propgation steps at lambda values 0.2 to 0.8.

blues.utils.**check_amber_selection**(*structure*, *selection*)
Given a AmberMask selection (str) for selecting atoms to freeze or restrain, check if it will actually select atoms. If the selection produces None, suggest valid residues or atoms.

Parameters

- **structure** (*parmed.Structure*) – The structure of the simulated system
- **selection** (*str*) – The selection string uses Amber selection syntax to select atoms to be restrained/frozen during simulation.
- **logger** (*logging.Logger*) – Records information or streams to terminal.

blues.utils.**parse_unit_quantity**(*unit_quantity_str*)
Utility for parsing parameters from the YAML file that require units.

Parameters **unit_quantity_str** (*str*) – A string specifying a quantity and it's units. i.e. '3.024 * daltons'

Returns **unit_quantity** (*simtk.unit.Quantity*) – i.e *unit.Quantity(3.024, unit=dalton)*

blues.utils.**zero_masses**(*system*, *atomList=None*)
Zeroes the masses of specified atoms to constrain certain degrees of freedom.

Parameters

- **system** (*penmm.System*) – system to zero masses
- **atomList** (*list of ints*) – atom indicies to zero masses

Returns **system** (*openmm.System*) – The modified system with massless atoms.

blues.utils.**atomIndexfromTop**(*resname*, *topology*)
Get atom indices of a ligand from OpenMM Topology.

Parameters

- **resname** (*str*) – resname that you want to get the atom indicies for (ex. 'LIG')
- **topology** (*str, optional, default=None*) – path of topology file. Include if the topology is not included in the coord_file

Returns **lig_atoms** (*list of ints*) – list of atoms in the coordinate file matching lig_resname

blues.utils.**get_data_filename**(*package_root*, *relative_path*)
Get the full path to one of the reference files in testsystems. In the source distribution, these files are in `blues/data/`, but on installation, they're moved to somewhere in the user's python site-packages directory. Adapted from: https://github.com/open-forcefield-group/smarty/blob/master/smarty/utils.py

Parameters

- **package_root** (*str*) – Name of the included/installed python package

- **relative_path** (*str*) – Path to the file within the python package

Returns **fn** (*str*) – Full path to file

blues.utils.**spreadLambdaProtocol**(*switching_values*, *steps*, *switching_types='auto'*, *kind='cubic'*, *return_tab_function=True*)

Takes a list of lambda values (either for sterics or electrostatics) and transforms that list to be spread out over a given *steps* range to be easily compatible with the OpenMM Discrete1DFunction tabulated function.

Parameters

- **switching_values** (*list*) – A list of lambda values decreasing from 1 to 0.
- **steps** (*int*) – The number of steps wanted for the tabulated function.
- **switching_types** (*str, optional, default='auto'*) – The type of lambda switching the *switching_values* corresponds to, either 'auto', 'electrostatics', or 'sterics'. If 'electrostatics' this assumes the inital value immediately decreases from 1. If 'sterics' this assumes the inital values stay at 1 for some period. If 'auto' this function tries to guess the switching_types based on this, based on typical lambda protocols turning off the electrostatics completely, before turning off sterics.
- **kind** (*str, optional, default='cubic'*) – The kind of interpolation that should be performed (using scipy.interpolate.interp1d) to define the lines between the points of switching_values.

Returns **tab_steps** (*list or simtk.openmm.openmm.Discrete1DFunction*) – List of length *steps* that corresponds to the tabulated-friendly version of the input switching_values. If return-tab_function=True

Examples

```
>>> from simtk.openmm.openmm import Continuous1DFunction, Discrete1DFunction
>>> sterics = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.95, 0.8848447462380346,
               0.8428373352131427, 0.7928373352131427, 0.7490146003095886, 0.
↪6934088361682191,
               0.6515123083157823, 0.6088924298371354, 0.5588924298371354, 0.
↪5088924298371353,
               0.4649556683144045, 0.4298606804827029, 0.3798606804827029, 0.
↪35019373288005945,
               0.31648339779024653, 0.2780498882483276, 0.2521302239477468, 0.
↪23139484523965026,
               0.18729812232625365, 0.15427643961733822, 0.12153116162972155,
               0.09632462702545555, 0.06463743549588846, 0.01463743549588846,
               0.0]
```

```
>>> statics = [1.0, 0.8519493439593149, 0.7142750443470669,
               0.5385929179832776, 0.3891972949356391, 0.18820309596839535, 0.0,
               0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
               0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> statics_tab = spreadLambdaProtocol(statics, opt['nstepsNC'], switching_types=
↪'auto')
>>> sterics_tab = spreadLambdaProtocol(sterics, opt['nstepsNC'], switching_types=
↪'sterics')
```

```
>>> # Assuming some Context already exists:
>>> context._integrator.addTabulatedFunction( 'sterics_tab', sterics_tab)
>>> context._integrator.addTabulatedFunction( 'electrostatics_tab', statics_tab)
```

3.5 Reporters

blues.reporters.**addLoggingLevel**(*levelName*, *levelNum*, *methodName=None*)

Comprehensively adds a new logging level to the *logging* module and the currently configured logging class.

levelName becomes an attribute of the *logging* module with the value *levelNum*. *methodName* becomes a convenience method for both *logging* itself and the class returned by *logging.getLoggerClass()* (usually just *logging.Logger*). If *methodName* is not specified, *levelName.lower()* is used.

To avoid accidental clobberings of existing attributes, this method will raise an *AttributeError* if the level name is already an attribute of the *logging* module or if the method name is already present

Parameters

- **levelName** (*str*) – The new level name to be added to the *logging* module.
- **levelNum** (*int*) – The level number indicated for the logging module.
- **methodName** (*str, default=None*) – The method to call on the logging module for the new level name. For example if provided 'trace', you would call *logging.trace()*.

Example

```
>>> addLoggingLevel('TRACE', logging.DEBUG - 5)
>>> logging.getLogger(__name__).setLevel("TRACE")
>>> logging.getLogger(__name__).trace('that worked')
>>> logging.trace('so did this')
>>> logging.TRACE
5
```

blues.reporters.**init_logger**(*logger*, *level=20*, *stream=True*, *outfname='blues-20210122-023427'*)

Initialize the Logger module with the given logger_level and outfname.

Parameters

- **logger** (*logging.getLogger()*) – The root logger object if it has been created already.
- **level** (*logging.<LEVEL>*) – Valid options for <LEVEL> would be DEBUG, INFO, WARNING, ERROR, CRITICAL.
- **stream** (*bool, default = True*) – If True, the logger will also stream information to sys.stdout as well as the output file.
- **outfname** (*str, default = time.strftime("blues-%Y%m%d-%H%M%S")*) – The output file path prefix to store the logged data. This will always write to a file with the extension *.log*.

Returns **logger** (*logging.getLogger()*) – The logging object with additional Handlers added.

class blues.reporters.**ReporterConfig**(*outfname*, *reporter_config*, *logger=None*)

Generates a set of custom/recommended reporters for BLUES simulations from YAML configuration. It can also be called externally without a YAML configuration file.

Parameters

- **outfname** (*str,*) – Output filename prefix for files generated by the reporters.
- **reporter_config** (*dict*) – Dict of parameters for the md_reporters or ncmc_reporters. Valid keys for reporters are: *state*, *traj_netcdf*, *restart*, *progress*, and *stream*. All reporters except *stream* are extensions of the parmed.openmm.reporters. More below: - *state* : State

data reporter for OpenMM simulations, but it is a little more generalized. Writes to a `.ene` file. For full list of parameters see *parmed.openmm.reporters.StateDataReporter*. - *traj_netcdf* : Customized AMBER NetCDF (`.nc`) format reporter - *restart* : Restart AMBER NetCDF (`.rst7`) format reporter - *progress* : Write to a file (`.prog`), the progress report of how many steps has been done, how fast the simulation is running, and how much time is left (similar to the mdinfo file in Amber). File is overwritten at each reportInterval. For full list of parameters see *parmed.openmm.reporters.ProgressReporter* - *stream* : Customized version of openmm.app.StateDataReporter.This will instead stream/print the information to the terminal as opposed to writing to a file. Takes the same parameters as the openmm.app.StateDataReporter

- **logger** (*logging.Logger object*) – Provide the root logger for printing information.

Examples

This class is intended to be called internally from *blues.config.set_Reporters*. Below is an example to call this externally.

```
>>> from blues.reporters import ReporterConfig
>>> import logging
>>> logger = logging.getLogger(__name__)
>>> md_reporters = { "restart": { "reportInterval": 1000 },
                     "state" : { "reportInterval": 250  },
                     "stream": { "progress": true,
                                 "remainingTime": true,
                                 "reportInterval": 250,
                                 "speed": true,
                                 "step": true,
                                 "title": "md",
                                 "totalSteps": 10000},
                     "traj_netcdf": { "reportInterval": 250 }
                    }
>>> md_reporter_cfg = ReporterConfig(outfname='blues-test', md_reporters, logger)
>>> md_reporters_list = md_reporter_cfg.makeReporters()
```

makeReporters()

Returns a list of openmm Reporters based on the configuration at initialization of the class.

class blues.reporters.**BLUESHDF5Reporter**(*file*, *reportInterval=1*, *title='NCMC Trajectory'*, *coordinates=True*, *frame_indices=[]*, *time=False*, *cell=True*, *temperature=False*, *potentialEnergy=False*, *kineticEnergy=False*, *velocities=False*, *atomSubset=None*, *protocolWork=True*, *alchemicalLambda=True*, *parameters=None*, *environment=True*)

This is a subclass of the HDF5 class from mdtraj that handles reporting of the trajectory.

HDF5Reporter stores a molecular dynamics trajectory in the HDF5 format. This object supports saving all kinds of information from the simulation – more than any other trajectory format. In addition to all of the options, the topology of the system will also (of course) be stored in the file. All of the information is compressed, so the size of the file is not much different than DCD, despite the added flexibility.

Parameters

- **file** (*str, or HDF5TrajectoryFile*) – Either an open HDF5TrajecoryFile object to write to, or a string specifying the filename of a new HDF5 file to save the trajectory to.
- **title** (*str,*) – String to specify the title of the HDF5 tables

- **frame_indices** (*list, frame numbers for writing the trajectory*)
- **reportInterval** (*int*) – The interval (in time steps) at which to write frames.
- **coordinates** (*bool*) – Whether to write the coordinates to the file.
- **time** (*bool*) – Whether to write the current time to the file.
- **cell** (*bool*) – Whether to write the current unit cell dimensions to the file.
- **potentialEnergy** (*bool*) – Whether to write the potential energy to the file.
- **kineticEnergy** (*bool*) – Whether to write the kinetic energy to the file.
- **temperature** (*bool*) – Whether to write the instantaneous temperature to the file.
- **velocities** (*bool*) – Whether to write the velocities to the file.
- **atomSubset** (*array_like, default=None*) – Only write a subset of the atoms, with these (zero based) indices to the file. If None, *all* of the atoms will be written to disk.
- **protocolWork** (*bool=False,*) – Write the protocolWork for the alchemical process in the NCMC simulation
- **alchemicalLambda** (*bool=False,*) – Write the alchemicalLambda step for the alchemical process in the NCMC simulation.
- **parameters** (*dict*) – Dict of the simulation parameters. Useful for record keeping.
- **environment** (*bool*) – True will attempt to export your conda environment to JSON and store the information in the HDF5 file. Useful for record keeping.

Notes

If you use the `atomSubset` option to write only a subset of the atoms to disk, the `kineticEnergy`, `potentialEnergy`, and `temperature` fields will not change. They will still refer to the energy and temperature of the *whole* system, and are not "subsetted" to only include the energy of your subsystem.

`describeNextReport`(*simulation*)
Get information about the next report this object will generate.

Parameters **simulation** (`app.Simulation`) – The simulation to generate a report for

Returns **nsteps, pos, vel, frc, ene** (*int, bool, bool, bool, bool*) – nsteps is the number of steps until the next report pos, vel, frc, and ene are flags indicating whether positions, velocities, forces, and/or energies are needed from the Context

`report`(*simulation*, *state*)
Generate a report.

Parameters

- **simulation** (*simtk.openmm.app.Simulation*) – The Simulation to generate a report for
- **state** (*simtk.openmm.State*) – The current state of the simulation

class blues.reporters.**BLUESStateDataReporter**(*file*, *reportInterval=1*, *frame_indices=[]*, *title=''*, *step=False*, *time=False*, *potentialEnergy=False*, *kineticEnergy=False*, *totalEnergy=False*, *temperature=False*, *volume=False*, *density=False*, *progress=False*, *remainingTime=False*, *speed=False*, *elapsedTime=False*, *separator='\t'*, *systemMass=None*, *totalSteps=None*, *protocolWork=False*, *alchemicalLambda=False*, *currentIter=False*)

StateDataReporter outputs information about a simulation, such as energy and temperature, to a file. To use it, create a StateDataReporter, then add it to the Simulation's list of reporters. The set of data to write is configurable using boolean flags passed to the constructor. By default the data is written in comma-separated-value (CSV) format, but you can specify a different separator to use. Inherited from *openmm.app.StateDataReporter*

Parameters

- **file** (*string or file*) – The file to write to, specified as a file name or file-like object (Logger)
- **reportInterval** (*int*) – The interval (in time steps) at which to write frames
- **frame_indices** (*list, frame numbers for writing the trajectory*)
- **title** (*str,*) – Text prefix for each line of the report. Used to distinguish between the NCMC and MD simulation reports.
- **step** (*bool=False*) – Whether to write the current step index to the file
- **time** (*bool=False*) – Whether to write the current time to the file
- **potentialEnergy** (*bool=False*) – Whether to write the potential energy to the file
- **kineticEnergy** (*bool=False*) – Whether to write the kinetic energy to the file
- **totalEnergy** (*bool=False*) – Whether to write the total energy to the file
- **temperature** (*bool=False*) – Whether to write the instantaneous temperature to the file
- **volume** (*bool=False*) – Whether to write the periodic box volume to the file
- **density** (*bool=False*) – Whether to write the system density to the file
- **progress** (*bool=False*) – Whether to write current progress (percent completion) to the file. If this is True, you must also specify totalSteps.
- **remainingTime** (*bool=False*) – Whether to write an estimate of the remaining clock time until completion to the file. If this is True, you must also specify totalSteps.
- **speed** (*bool=False*) – Whether to write an estimate of the simulation speed in ns/day to the file
- **elapsedTime** (*bool=False*) – Whether to write the elapsed time of the simulation in seconds to the file.
- **separator** (*string=','*) – The separator to use between columns in the file
- **systemMass** (*mass=None*) – The total mass to use for the system when reporting density. If this is None (the default), the system mass is computed by summing the masses of all particles. This parameter is useful when the particle masses do not reflect their actual physical mass, such as when some particles have had their masses set to 0 to immobilize them.
- **totalSteps** (*int=None*) – The total number of steps that will be included in the simulation. This is required if either progress or remainingTime is set to True, and defines how many steps will indicate 100% completion.

- **protocolWork** (*bool=False,*) – Write the protocolWork for the alchemical process in the NCMC simulation
- **alchemicalLambda** (*bool=False,*) – Write the alchemicalLambda step for the alchemical process in the NCMC simulation.

describeNextReport(*simulation*)
Get information about the next report this object will generate.

Parameters **simulation** (app.Simulation) – The simulation to generate a report for

Returns **nsteps, pos, vel, frc, ene** (*int, bool, bool, bool, bool*) – nsteps is the number of steps until the next report pos, vel, frc, and ene are flags indicating whether positions, velocities, forces, and/or energies are needed from the Context

report(*simulation*, *state*)
Generate a report.

Parameters

- **simulation** (*Simulation*) – The Simulation to generate a report for
- **state** (*State*) – The current state of the simulation

class blues.reporters.**NetCDF4Reporter**(*file*, *reportInterval=1*, *frame_indices=[]*, *crds=True*, *vels=False*, *frcs=False*, *protocolWork=False*, *alchemicalLambda=False*)
Class to read or write NetCDF trajectory files Inherited from *parmed.openmm.reporters.NetCDFReporter*

Parameters

- **file** (*str*) – Name of the file to write the trajectory to
- **reportInterval** (*int*) – How frequently to write a frame to the trajectory
- **frame_indices** (*list, frame numbers for writing the trajectory*) – If this reporter is used for the NCMC simulation, 0.5 will report at the moveStep and -1 will record at the last frame.
- **crds** (*bool=True*) – Should we write coordinates to this trajectory? (Default True)
- **vels** (*bool=False*) – Should we write velocities to this trajectory? (Default False)
- **frcs** (*bool=False*) – Should we write forces to this trajectory? (Default False)
- **protocolWork** (*bool=False,*) – Write the protocolWork for the alchemical process in the NCMC simulation
- **alchemicalLambda** (*bool=False,*) – Write the alchemicalLambda step for the alchemical process in the NCMC simulation.

describeNextReport(*simulation*)
Get information about the next report this object will generate.

Parameters **simulation** (app.Simulation) – The simulation to generate a report for

Returns **nsteps, pos, vel, frc, ene** (*int, bool, bool, bool, bool*) – nsteps is the number of steps until the next report pos, vel, frc, and ene are flags indicating whether positions, velocities, forces, and/or energies are needed from the Context

report(*simulation*, *state*)
Generate a report.

Parameters

- **simulation** (app.Simulation) – The Simulation to generate a report for
- **state** (mm.State) – The current state of the simulation

3.6 Formats

class blues.formats.**LoggerFormatter**

Formats the output of the *logger.Logger* object. Allows customization for customized logging levels. This will add a custom level 'REPORT' to all custom BLUES reporters from the *blues.reporters* module.

Examples

Below we add a custom level 'REPORT' and have the logger module stream the message to *sys.stdout* without any additional information to our custom reporters from the *blues.reporters* module

```
>>> from blues import reporters
>>> from blues.formats import LoggerFormatter
>>> import logging, sys
>>> logger = logging.getLogger(__name__)
>>> reporters.addLoggingLevel('REPORT', logging.WARNING - 5)
>>> fmt = LoggerFormatter(fmt="%(message)s")
>>> stdout_handler = logging.StreamHandler(stream=sys.stdout)
>>> stdout_handler.setFormatter(fmt)
>>> logger.addHandler(stdout_handler)
>>> logger.report('This is a REPORT call')
    This is a REPORT call
>>> logger.info('This is an INFO call')
    INFO: This is an INFO call
```

format(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class blues.formats.**BLUESHDF5TrajectoryFile**(*filename*, *mode='r'*, *force_overwrite=True*, *compression='zlib'*)

Extension of the *mdtraj.formats.hdf5.HDF5TrajectoryFile* class which handles the writing of the trajectory data to the HDF5 file format. Additional features include writing NCMC related data to the HDF5 file.

Parameters

- **filename** (*str*) – The filename for the HDF5 file.
- **mode** (*str, default='r'*) – The mode to open the HDF5 file in.
- **force_overwrite** (*bool, default=True*) – If True, overwrite the file if it already exists
- **compression** (*str, default='zlib'*) – Valid choices are ['zlib', 'lzo', 'bzip2', 'blosc']

write(*coordinates*, *parameters=None*, *environment=None*, *time=None*, *cell_lengths=None*, *cell_angles=None*, *velocities=None*, *kineticEnergy=None*, *potentialEnergy=None*, *temperature=None*, *alchemicalLambda=None*, *protocolWork=None*, *title=None*)

Write one or more frames of data to the file This method saves data that is associated with one or more simulation frames. Note that all of the arguments can either be raw numpy arrays or unitted arrays (with simtk.unit.Quantity). If the arrays are unittted, a unit conversion will be automatically done from the supplied units into the proper units for saving on disk. You won't have to worry about it.

Furthermore, if you wish to save a single frame of simulation data, you can do so naturally, for instance by supplying a 2d array for the coordinates and a single float for the time. This "shape deficiency" will be recognized, and handled appropriately.

Parameters

- **coordinates** (*np.ndarray, shape=(n_frames, n_atoms, 3)*) – The cartesian coordinates of the atoms to write. By convention, the lengths should be in units of nanometers.
- **time** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the simulation time, in picoseconds corresponding to each frame.
- **cell_lengths** (*np.ndarray, shape=(n_frames, 3), dtype=float32, optional*) – You may optionally specify the unitcell lengths. The length of the periodic box in each frame, in each direction, *a*, *b*, *c*. By convention the lengths should be in units of angstroms.
- **cell_angles** (*np.ndarray, shape=(n_frames, 3), dtype=float32, optional*) – You may optionally specify the unitcell angles in each frame. Organized analogously to cell_lengths. Gives the alpha, beta and gamma angles respectively. By convention, the angles should be in units of degrees.
- **velocities** (*np.ndarray, shape=(n_frames, n_atoms, 3), optional*) – You may optionally specify the cartesian components of the velocity for each atom in each frame. By convention, the velocities should be in units of nanometers / picosecond.
- **kineticEnergy** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the kinetic energy in each frame. By convention the kinetic energies should b in units of kilojoules per mole.
- **potentialEnergy** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the potential energy in each frame. By convention the kinetic energies should b in units of kilojoules per mole.
- **temperature** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the temperature in each frame. By convention the temperatures should b in units of Kelvin.
- **alchemicalLambda** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the alchemicalLambda in each frame. These have no units, but are generally between zero and one.
- **protocolWork** (*np.ndarray, shape=(n_frames,), optional*) – You may optionally specify the protocolWork in each frame. These are in reduced units of kT but are stored dimensionless
- **title** (*str*) – Title of the HDF5 trajectory file

class blues.formats.**NetCDF4Traj**(*fname*, *mode='r'*)

Extension of *parmed.amber.netcdffiles.NetCDFTraj* to allow proper file flushing. Requires the netcdf4 library (not scipy), install with *conda install -c conda-forge netcdf4* .

Parameters

- **fname** (*str*) – File name for the trajectory file
- **mode** (*str, default='r'*) – The mode to open the file in.

flush()

Flush buffered data to disc.

classmethod open_new(*fname*, *natom*, *box*, *crds=True*, *vels=False*, *frcs=False*, *remd=None*, *remd_dimension=None*, *title=''*, *protocolWork=False*, *alchemicalLambda=False*)

Opens a new NetCDF file and sets the attributes

Parameters

- **fname** (*str*) – Name of the new file to open (overwritten)
- **natom** (*int*) – Number of atoms in the restart
- **box** (*bool*) – Indicates if cell lengths and angles are written to the NetCDF file
- **crds** (*bool, default=True*) – Indicates if coordinates are written to the NetCDF file
- **vels** (*bool, default=False*) – Indicates if velocities are written to the NetCDF file
- **frcs** (*bool, default=False*) – Indicates if forces are written to the NetCDF file
- **remd** (*str, default=None*) – 'T[emperature]' if replica temperature is written 'M[ulti]' if Multi-D REMD information is written None if no REMD information is written
- **remd_dimension** (*int, default=None*) – If remd above is 'M[ulti]', this is how many REMD dimensions exist
- **title** (*str, default=''*) – The title of the NetCDF trajectory file
- **protocolWork** (*bool, default=False*) – Indicates if protocolWork from the NCMC simulation should be written to the NetCDF file
- **alchemicalLambda** (*bool, default=False*) – Indicates if alchemicalLambda from the NCMC simulation should be written to the NetCDF file

property protocolWork
Store the accumulated protocolWork from the NCMC simulation as property.

add_protocolWork (*stuff*)
Adds the time to the current frame of the NetCDF file

Parameters **stuff** (*float or time-dimension Quantity*) – The time to add to the current frame

property alchemicalLambda
Store the current alchemicalLambda (0->1.0) from the NCMC simulation as property.

add_alchemicalLambda (*stuff*)
Adds the time to the current frame of the NetCDF file

Parameters **stuff** (*float or time-dimension Quantity*) – The time to add to the current frame

CHAPTER

FOUR

TUTORIAL

This page provides examples on how to use BLUES. A Jupyter notebook is available in the examples folder on github so you can try them out yourself or you can view it on nbviewer

“nbsphinx-toctree”: { “maxdepth”: 2 }

4.1 Introduction to BLUES

In this Jupyter Notebook, we will cover the following topics:

- YAML configuration
- Setting up a system for BLUES
- Advanced options (HMR, restraints, freezing)
- Configuring reporters
- Running a BLUES simulation

4.2 Background

4.2.1 Coupling MD simulations with random NCMC moves for enhanced sampling of ligand binding modes via BLUES

BLUES is an approach that combines molecular dynamics (MD) simulations and the Non-equilibrium Candidate Monte Carlo (NCMC) framework to enhance ligand binding mode sampling (Github)

During a MD simulation, BLUES will perform a random rotation of the bound ligand and then allow the system to relax through alchemically scaling off/on the ligand-receptor interactions. BLUES enables us to sample alternative ligand binding modes, that would normally take *very* long simulations to capture in a traditional MD simulation because of the large gap in timescales between atomistic motions and biological motions.

4.3 YAML Configuration

BLUES can be configured in either pure python through dictionaries of the appropriate parameters or using a YAML file (which is converted to a dict under the hood). Below we will walk through the keywords for configuring BLUES. An example YAML configuration file can be found in `rotmove_cuda.yaml`.

Note: Code blocks in this notebook denoted with —— indicate a section from a YAML configuration file.

4.3.1 Input/Output

```
------
output_dir: .
outfname: t4-toluene
logger_level: info #critical, error, warning, info, debug
-----
```

Output files

Specify the directory you want all the simulation output files to be saved to with **``output_dir``**. By default, BLUES will save them in the current directory that you're running BLUES in. The parameter **``outfname``** will be used for the filename prefix for all output files (e.g. `t4-toluene.nc`, `t4-toleune.log`). The level of verbosity can be controlled by **``logger_level``**. The default **``logger_level``** is set to `info` and valid choices are `critical`, `error`, `warning`, `info` or `debug`.

Input files to generate the structure of your system

- Input a Parameter/topology file and a Coordinate file, which will be used to generate the ParmEd Structure.
- The ParmEd Structure is a chemical structure composed of atoms, bonds, angles, torsions, and other topological features.

To see a full list of supported file formats: https://parmed.github.io/ParmEd/html/readwrite.html

```
-----------------
structure:
  filename: tests/data/eqToluene.prmtop
  xyz: tests/data/eqToluene.inpcrd
  restart: t4-toluene_2.rst7
-----------------
```

BLUES simulations all begin from a `parmed.Structure`. Keywords nested under **``structure``** are for generating the `parmed.Structure` by calling parmed.load_file() under the hood. The **``filename``** keyword should point to the file containing the parameters for your system. For example, if coming from AMBER you would specify the `.prmtop` file. The **``xyz``** keyword is intended for specifying the coordinates of your system (e.g. `.inpcrd` or `.pdb`).

Restart files

BLUES supports "soft" restarts of simulations from AMBER restart files ``**.rst7**`` via parmed.amber.Rst7. "Soft" restart implies the simulation will begin from the saved positions, velocities, and box vectors but does not store any internal data such as the states of random number generators. It should be noted that velocities are re-initialized at every BLUES iteration, so storing these is not so important.

4.3.2 System configuration

From the YAML file, there is a section dedicated for generating an `openmm.System` from a `parmed.Structure`. For definitions of ``**system**`` keywords and valid options see parmed.Structure.createSystem()

Below we provide an example for generating a system in a cubic box with explicit solvent.

```
--------
system:
  nonbondedMethod: PME
  nonbondedCutoff: 10 * angstroms
  constraints: HBonds
  rigidWater: True
  removeCMMotion: True
  ewaldErrorTolerance: 0.005
  flexibleConstraints: True
  splitDihedrals: False
---------
```

(Optional) Hydrogen mass repartitioning

BLUES has the option to use the hydrogen mass repartitioning scheme HMR to allow use of longer time steps in the simulation. Simply provide the keyword ``**hydrogenMass**`` in the YAML file like below:

```
--------
system:
  nonbondedMethod: PME
  nonbondedCutoff: 10 * angstroms
  constraints: HBonds
  rigidWater: True
  removeCMMotion: True
  ewaldErrorTolerance: 0.005
  flexibleConstraints: True
  splitDihedrals: False
  hydrogenMass: 3.024 * daltons
---------
```

If using HMR, you can set the timestep (``**dt**``) for the simulation to 4fs and ``**constraints**`` should be set to either ``**HBonds**`` or ``**AllBonds**``.

(Optional) Alchemical system configuration

Nested under the system parameters, you can modify parameters for the alchemical system. Below are the default settings and are not required to be specified in the YAML configuration. **Modifications to these parameters are for advanced users.**

```
-------
system:
  nonbondedMethod: PME
  nonbondedCutoff: 10 * angstroms
  constraints: HBonds
  rigidWater: True
  removeCMMotion: True
  ewaldErrorTolerance: 0.005
  flexibleConstraints: True
  splitDihedrals: False
  alchemical:
    # Sterics
    softcore_alpha: 0.5
    softcore_a : 1
    softcore_b : 1
    softcore_c : 6

    # Electrostatics
    softcore_beta : 0.0
    softcore_d : 1
    softcore_e : 1
    softcore_f : 2

    annihilate_electrostatics : True
    annihilate_sterics : False
-------
```

For further details on alchemical parameters see: http://getyank.org/0.16.2/yamlpages/options.html

4.3.3 Simulation Configuration

The keywords for configuring the Simulations for BLUES are explained below:

- **``dt``**: timestep
- **``nIter``**: number of iterations or proposed moves
- **``nstepsMD``**: number of MD steps
- **``nstepsNC``**: number NCMC steps

The configuration below will run BLUES in NVT with 2fs timesteps for 10 iterations. The MD and NCMC simulation will run 10,000 steps per iteration.

```
----------
simulation:
  platform: CUDA
  dt: 0.002 * picoseconds
  friction: 1 * 1/picoseconds
  temperature: 300 * kelvin
  nIter: 10
  nstepsMD: 10000
```

(continues on next page)

(continued from previous page)

```
  nstepsNC: 10000
----------
```

NPT Simulation

To run BLUES in NPT, simply specify a ``**pressure**``:

```
----------
simulation:
  platform: CUDA
  dt: 0.002 * picoseconds
  friction: 1 * 1/picoseconds
  temperature: 300 * kelvin
  nIter: 10
  nstepsMD: 10000
  nstepsNC: 10000
  pressure: 1 * atmospheres
----------
```

(Optional) Additional relaxation steps in the NCMC simulation

Keywords ``**nprop**`` and ``**propLambda**`` allow you to add additional relxation steps between a set range in the lambda schedule for the alchemical process in the NCMC simulation. Setting ``**propLambda**`` to 0.3 will select a lambda range of -/+ 0.3 from the midpoint (0.5), giving [0.2, 0.8]. During the alchemical process, when lambda is between 0.2 to 0.8, ``**nprop**`` controls the number of additional relaxation steps to add at each lambda step (change in lambda). Additional relaxation steps has been show to increase acceptance proposed NCMC moves.

```
----------
simulation:
  platform: CUDA
  dt: 0.002 * picoseconds
  friction: 1 * 1/picoseconds
  temperature: 300 * kelvin
  nIter: 10
  nstepsMD: 10000
  nstepsNC: 10000
  pressure: 1 * atmospheres
  nprop: 3
  propLambda: 0.3
----------
```

(Optional) Platform Properties

If you need to modify platform properties for the simulation, you can set the keyword ``**properties**`` like below:

Example: OpenCL in single precision on GPU device 2

Note: works for running on the GPU on MacBook Pro 2017

```
----------
simulation:
  platform: OpenCL
  properties:
    OpenCLPrecision: single
    OpenCLDeviceIndex: 2
  dt: 0.002 * picoseconds
  friction: 1 * 1/picoseconds
  temperature: 300 * kelvin
  nIter: 10
  nstepsMD: 10000
  nstepsNC: 10000
  pressure: 1 * atmospheres
----------
```

Example: CUDA in double precision on GPU device 0

```
----------
simulation:
  platform: CUDA
  properties:
    CudaPrecision: double
    CudaDeviceIndex: 0
  dt: 0.002 * picoseconds
  friction: 1 * 1/picoseconds
  temperature: 300 * kelvin
  nIter: 10
  nstepsMD: 10000
  nstepsNC: 10000
  pressure: 1 * atmospheres
----------
```

4.3.4 Reporter Configuration

We provide functionality to configure a recommended set of reporters from the YAML file. These are used to record information for either the MD or NCMC simulation. Below are the keywords for each reporter. Each reporter will require the ``**reportInterval**`` keyword to specify the frequency to store the simulation data: - ``**state**`` : State data reporter for OpenMM simulations, but it is a little more generalized. Writes to a `.ene` file. - For full list of parameters see parmed.openmm.reporters.StateDataReporter - ``**traj_netcdf**`` : Customized AMBER NetCDF (`.nc`) format reporter - ``**restart**`` : Restart AMBER NetCDF (`.rst7`) format reporter - ``**progress**`` : Write to a file (`.prog`), the progress report of how many steps has been done, how fast the simulation is running, and how much time is left (similar to the mdinfo file in Amber). File is overwritten at each reportInterval. - For full list of parameters see parmed.openmm.reporters.ProgressReporter - ``**stream**`` : Customized version of openmm.app.StateDataReporter. This will instead stream/print the information to the terminal as opposed to writing to a file. - takes the same parameters as the openmm.app.StateDataReporter

To attach them to the MD simulation. You nest the reporter keywords under the keyword ``**md_reporters**`` like below. To attach the reporters to NCMC simulation, use the ``**ncmc_reporters**`` keyword instead.

```
------
md_reporters:
  state:
    reportInterval: 250
  traj_netcdf:
    reportInterval: 250
  restart:
    reportInterval: 1000
  progress:
    totalSteps: 10000
    reportInterval: 10
  stream:
    title: md
    reportInterval: 250
    totalSteps: 10000
    step: True
    speed: True
    progress: True
    remainingTime: True
----
```

In the above example, we are using the ``**stream**`` reporter to print the speeds on the intergrator at regular intervals. This may be a bit redudant with the ``**progress**`` reporter if are running the job remotely and don't need the information streamed to terminal.

(Optional) Advanced options to the ``traj_netcdf`` reporter

The ``**traj_netcdf**`` reporter can store additional information that may be useful for the NCMC simulation or record at specific frames. In the example below, we will store the first, midpoint (when the move is applied), and last frame of each NCMC iteration, along with the ``**alchemicalLambda**`` step and the ``**protocolWork**``.

```
----
ncmc_reporters:
  traj_netcdf:
    frame_indices: [1, 0.5, -1]
    alchemicalLambda: True
    protocolWork: True
-----
```

To access the numerical data stored in the NetCDF file:

```
from netCDF4 import Dataset

f = Dataset("t4-toluene-ncmc.nc")
print(f.variables['alchemicalLambda'][:])
print(f.variables['protocolWork'][:])

>>> [ 0.001  0.5    1.   ]
>>> [  0.03706791  30.72696877  25.708498  ]
```

4.4 Running a BLUES simulation

Below we will provide an example for running an NPT BLUES simulation which applies random rotational moves to the toluene ligand in T4-lysozyme from a YAML configuration.

[1]:
```
yaml_cfg = """
output_dir: .
outfname: t4-toluene
logger_level: info

structure:
  filename: ../blues/tests/data/eqToluene.prmtop
  xyz: ../blues/tests/data/eqToluene.inpcrd

system:
  nonbondedMethod: PME
  nonbondedCutoff: 10 * angstroms
  constraints: HBonds
  rigidWater: True
  removeCMMotion: True
  hydrogenMass: 3.024 * daltons
  ewaldErrorTolerance: 0.005
  flexibleConstraints: True
  splitDihedrals: False

freeze:
  freeze_center: ':LIG'
  freeze_solvent: ':WAT,Cl-'
  freeze_distance: 5 * angstroms

simulation:
  platform: CUDA
  properties:
    CudaPrecision: single
    CudaDeviceIndex: 0
  dt: 0.004 * picoseconds
  friction: 1 * 1/picoseconds
  pressure: 1 * atmospheres
  temperature: 300 * kelvin
  nIter: 5
  nstepsMD: 1000
  nstepsNC: 1000

md_reporters:
  state:
    reportInterval: 250
  traj_netcdf:
    reportInterval: 250
  restart:
    reportInterval: 1000
  stream:
    title: md
    reportInterval: 250
    totalSteps: 5000 # nIter * nstepsMD
    step: True
    speed: True
    progress: True
```

(continues on next page)

(continued from previous page)

```
    remainingTime: True

ncmc_reporters:
  stream:
    title: ncmc
    reportInterval: 250
    totalSteps: 1000 # Use nstepsNC
    step: True
    speed: True
    progress: True
    remainingTime: True
"""
```

Import the following BLUES modules required for the following steps.

- Make sure to specify the type of move that you want to import from **blues.moves**
- Available moves can be veiwed in **moves.py**, supported/tested moves include:
  - RandomLigandRotationMove
  - SideChainMove

[2]:
```
from blues.moves import RandomLigandRotationMove, MoveEngine
from blues.simulation import *
from blues.settings import *
```

[3]:
```
#Read in the YAML file
cfg = Settings(yaml_cfg).asDict()
#Shortcut to access `parmed.Structure` from dict
structure = cfg['Structure']
```

```
./t4-toluene
```

Below is what the resulting configuration dictionary looks like (formatted into JSON for readability).

[4]:
```
import json
print(json.dumps(cfg, sort_keys=True, indent=2, skipkeys=True, default=str))
```

```
{
  "Logger": "<logging.RootLogger object at 0x7f96b1bff940>",
  "Structure": "../blues/tests/data/eqToluene.prmtop",
  "freeze": {
    "freeze_center": ":LIG",
    "freeze_distance": "5.0 A",
    "freeze_solvent": ":WAT,Cl-"
  },
  "logger_level": "info",
  "md_reporters": [
    "<parmed.openmm.reporters.StateDataReporter object at 0x7f966832c128>",
    "<blues.reporters.NetCDF4Reporter object at 0x7f966574e898>",
    "<parmed.openmm.reporters.RestartReporter object at 0x7f966574e860>",
    "<blues.reporters.BLUESStateDataReporter object at 0x7f966574e828>"
  ],
  "ncmc_reporters": [
    "<blues.reporters.BLUESStateDataReporter object at 0x7f966574e908>"
  ],
  "outfname": "./t4-toluene",
```

(continues on next page)

(continued from previous page)

```
  "output_dir": ".",
  "simulation": {
    "dt": "0.004 ps",
    "friction": "1.0 /ps",
    "md_trajectory_interval": 250,
    "moveStep": 500,
    "nIter": 5,
    "nprop": 1,
    "nstepsMD": 1000,
    "nstepsNC": 1000,
    "outfname": "./t4-toluene",
    "platform": "CUDA",
    "pressure": "1.0 atm",
    "propLambda": 0.3,
    "propSteps": 1000,
    "properties": {
      "CudaDeviceIndex": 0,
      "CudaPrecision": "single"
    },
    "temperature": "300.0 K",
    "verbose": false
  },
  "structure": {
    "filename": "../blues/tests/data/eqToluene.prmtop",
    "xyz": "../blues/tests/data/eqToluene.inpcrd"
  },
  "system": {
    "constraints": "HBonds",
    "ewaldErrorTolerance": 0.005,
    "flexibleConstraints": true,
    "hydrogenMass": "3.024 Da",
    "nonbondedCutoff": "10.0 A",
    "nonbondedMethod": "PME",
    "removeCMMotion": true,
    "rigidWater": true,
    "splitDihedrals": false,
    "verbose": false
  },
  "verbose": false
}
```

4.4.1 Selecting a move and initialize the move engine

Here, we initialize the **``RandomLigandRotationMove``** from the **``blues.moves``** module which proposes random rotations on the toluene ligand. We select the toluene ligand by providing the residue name `LIG` and the `parmed.Structure` to select the atoms from. If we begin BLUES from our YAML configuration, the `parmed.Structure` for our system is generated from the call to `startup()`. We can access it at the top level with `cfg['Structure']`

After initialization of the selected move, we pass the move object to the **``MoveEngine``** from the **``blues.engine``** module. The **``MoveEngine``** controls what types of moves will be performed during the NCMC protocol and with a given probability. This will be more useful when we use multiple move types.

[5]:
```
#Initialize the move class and pass it to the engine
ligand = RandomLigandRotationMove(structure, 'LIG')
ligand_mover = MoveEngine(ligand)
```

4.4.2 Generating the Systems for openMM: ``SystemFactory``

Next, we must generate the `openmm.System` from the `parmed.Structure` by calling the **``SystemFactory``** class from the `blues.simulation` module. The class must be initialized by providing 3 required arguments:

```
structure : parmed.Structure
    A chemical structure composed of atoms, bonds, angles, torsions, and
    other topological features.
atom_indices : list of int
    Atom indicies of the move or designated for which the nonbonded forces
    (both sterics and electrostatics components) have to be alchemically
    modified.
config : dict, parameters for generating the `openmm.System for the MD
    and NCMC simulation. For complete parameters, see docs for `generateSystem`
    and `generateAlchSystem`
```

Upon initialization, this class will create the system for the MD simulation and the NCMC simulation. They can be accessed through the attributes **``systems.md``** or **``systems.alch``**. Any modifications to either of these Systems should be done within the context of this object. Once the systems are passed into an `openmm.Simulation`, you will not be able to modify the system easily.

[6]:
```
systems = SystemFactory(structure, ligand.atom_indices, cfg['system'])
```

```
INFO: Adding bonds...
INFO: Adding angles...
INFO: Adding dihedrals...
INFO: Adding Ryckaert-Bellemans torsions...
INFO: Adding Urey-Bradleys...
INFO: Adding improper torsions...
INFO: Adding CMAP torsions...
INFO: Adding trigonal angle terms...
INFO: Adding out-of-plane bends...
INFO: Adding pi-torsions...
INFO: Adding stretch-bends...
INFO: Adding torsion-torsions...
INFO: Adding Nonbonded force...
```

(Optional) Applying restraints or freezing atoms

The **``SystemFactory``** class also provides functionality for restraining or freezing the atoms. **Use extreme caution when freezing/restraining atoms. You should consider if freezing/restraining should be applied to BOTH the MD and alchemical system.**

Selections for either restraining or freezing atoms in your system use the Amber mask syntax.

Positional restraints: ``SystemFactory.restrain_positions()``

To apply positional restraints, you can call **``SystemFactory.restrain_positions()``**. You can specify the parameters/selection for applying positional restraints in the YAML file.

```
------
restraints:
    selection: '@CA,C,N'
    weight: 5
------
```

restrain keywords: - ``**selection**``: Specify what to apply positional restraints to using Amber mask syntax. Default = '@CA,C,N' - ``**weight**``: Restraint weight for xyz atom restraints in kcal/(mol A^2). Default = 5

From the YAML example above, we would be applying positional restraints to the backbone atoms of the protein. If applying restraints, you most likely will want to apply it to BOTH the MD and alchemical systems like below:

```
systems.md = SystemFactory.restrain_positions(structure, systems.md, **cfg['restraints
↪'])
systems.alch = SystemFactory.restrain_positions(structure, systems.alch, **cfg[
↪'restraints'])
```

Freezing selected atoms: ``SystemFactory.freeze_atoms()``

To freeze a selection, call ``**SystemFactory.freeze_atoms()**``. Atoms that have a mass of zero will be ignored by the integrator and will not change positions during the simulation, effectively they are frozen.

To freeze atoms using a given selection string. Use the keyword ``**freeze_selection**``.

```
-----
freeze:
  freeze_selection: ':LIG'
------
```

From the YAML example above, we would be freezing only the atoms belonging to the residue `LIG`. Although freezing the ligand in this example wouldn't be very useful. It would be applied like

```
systems.md = SystemFactory.freeze_atoms(structure, systems.md, **cfg['freeze'])
```

Freezing atoms around a selection: ``SystemFactory.freeze_radius()``

Alternatively, you can choose to freeze atoms around a given selection. To do so, call ``**SystemFactory.freeze_radius()**``. For example, you may want to freeze atoms that are 5 angstroms away from the ligand and include the solvent.

```
-----
freeze:
  freeze_center: ':LIG'
  freeze_solvent: ':WAT,Cl-'
  freeze_distance: 5 * angstroms
------
```

- ``**freeze_center**``: Specifies the center of the object for freezing, masses will be zeroed. Default = ':LIG'
- ``**freeze_solvent**``: select which solvent atoms should have their masses zeroed. Default = ':HOH,NA,CL'
- ``**freeze_distance**``: Distance (in angstroms) to select atoms for retaining their masses. Atoms outside the set distance will have their masses set to 0.0. Default = 5.0

We often utilize this type of freezing to speed up the alchemical process during the NCMC simulation while leaving them completely free in the MD simulation for proper relaxation.

```
systems.alch = SystemFactory.freeze_radius(structure, systems.alch, **cfg['freeze'])
```

In this notebook example, our YAML config indicates we will be freezing around the ligand (keyword: ``**freeze_center**``). So we will call the ``**freeze_radius**`` function.

[7]:
```
systems.alch = SystemFactory.freeze_radius(structure, systems.alch, **cfg['freeze'])
```

```
INFO: Freezing 22065 atoms 5.0 Angstroms from ':LIG' on <simtk.openmm.openmm.System;␣
↪proxy of <Swig Object of type 'OpenMM::System *' at 0x7f96aa74fab0> >
```

4.4.3 Generating the OpenMM Simulations: ``SimulationFactory``

Now that we have generated our `openmm.System` for the MDand frozen the solvent around the ligand. We are now ready to create the set of simulations for running BLUES. We do this by calling the ``**SimulationFactory**`` class . The expected parameters are:

```
systems : blues.simulation.SystemFactory object
    The object containing the MD and alchemical openmm.Systems
move_engine : blues.engine.MoveEngine object
    MoveProposal object which contains the dict of moves performed
    in the NCMC simulation.
config : dict of parameters for the simulation (i.e timestep, temperature, etc.)
md_reporters : list of Reporter objects for the MD openmm.Simulation
ncmc_reporters : list of Reporter objects for the NCMC openmm.Simulation
```

If you wish to use your own openmm reporters, simply pass them into the arguments as a *list* of Reporter objects. Since we have configured our reporters from the YAML file, we can pass them into the arguments ``**md_reporters**`` and ``**ncmc_reporters**``.

[8]:
```
# List of MD reporters
cfg['md_reporters']
```

[8]:
```
[<parmed.openmm.reporters.StateDataReporter at 0x7f966832c128>,
 <blues.reporters.NetCDF4Reporter at 0x7f966574e898>,
 <parmed.openmm.reporters.RestartReporter at 0x7f966574e860>,
 <blues.reporters.BLUESStateDataReporter at 0x7f966574e828>]
```

[9]:
```
# List of NCMC reporters
cfg['ncmc_reporters']
```

[9]:
```
[<blues.reporters.BLUESStateDataReporter at 0x7f966574e908>]
```

[10]:
```
simulations = SimulationFactory(systems, ligand_mover, cfg['simulation'],
                                cfg['md_reporters'], cfg['ncmc_reporters'])
```

```
INFO: Adding MonteCarloBarostat with 1.0 atm. MD simulation will be 300.0 K NPT.
WARNING: NCMC simulation will NOT have pressure control. NCMC will use pressure from␣
↪last MD state.
INFO: OpenMM(7.1.1.dev-c1a64aa) simulation generated for CUDA platform
system = Linux
node = titanpascal
release = 4.13.0-41-generic
version = #46~16.04.1-Ubuntu SMP Thu May 3 10:06:43 UTC 2018
machine = x86_64
processor = x86_64
DeviceIndex = 0
DeviceName = TITAN Xp
UseBlockingSync = true
Precision = single
UseCpuPme = false
CudaCompiler = /usr/local/cuda-8.0/bin/nvcc
```

(continues on next page)

(continued from previous page)

```
TempDirectory = /tmp
CudaHostCompiler =
DisablePmeStream = false
DeterministicForces = false
```

If you would like to access the MD or NCMC simulation. You can access them as attributes to the ``**SimulationFactory**`` class with ``**simulations.md**`` or ``**simulations.ncmc**``. This will allow you to do things like energy minimize the system or run a few steps of regular dynamics before running the hybrid (MD+NCMC) BLUES approach. The NCMC simulation will automatically be synced to the state of the MD simulation when running the BLUES simulation.

[11]:
```
# Energy minimization
state = simulations.md.context.getState(getPositions=True, getEnergy=True)
print('Pre-Minimized energy = {}'.format(state.getPotentialEnergy().in_units_of(unit.
↪kilocalorie_per_mole)))

simulations.md.minimizeEnergy(maxIterations=0)
state = simulations.md.context.getState(getPositions=True, getEnergy=True)
print('Minimized energy = {}'.format(state.getPotentialEnergy().in_units_of(unit.
↪kilocalorie_per_mole)))
```

```
Pre-Minimized energy = -69057.34671058532 kcal/mol
Minimized energy = -87007.38938198877 kcal/mol
```

[12]:
```
# Running only the MD simulation
simulations.md.step(500)
```

```
#"Progress (%)"       "Step"  "Speed (ns/day)"        "Time Remaining"
md: 5.0%       250     0       --
md: 10.0%      500     271     0:05
```

4.4.4 Run the BLUES Simulation

To run the full BLUES simulation, where we apply NCMC moves and follow-up with the MD simulation, we simply pass the ``**SimulationFactory**`` object to the ``**Simulation**`` class and call the ``**run**()`` function which takes ``**nIter**``, ``**nstepsNC**``, ``**nstepsMD**`` as arguments (or we can pass it the simulation configuration from the YAML on initialization of the class).

[13]:
```
blues = BLUESSimulation(simulations, cfg['simulation'])
blues.run()
```

```
INFO: Total BLUES Simulation Time = 40.0 ps (8.0 ps/Iter)
Total Force Evaluations = 10000
Total NCMC time = 20.0 ps (4.0 ps/iter)
Total MD time = 20.0 ps (4.0 ps/iter)
Trajectory Interval = 2.0 ps/frame (4.0 frames/iter)
INFO: Running 5 BLUES iterations...
INFO: BLUES Iteration: 0
INFO: Advancing 1000 NCMC switching steps...
#"Progress (%)"       "Step"  "Speed (ns/day)"        "Time Remaining"
ncmc: 25.0%    250     0       --
ncmc: 50.0%    500     110     0:01
Performing RandomLigandRotationMove...
ncmc: 75.0%    750     93.3    0:00
```

(continues on next page)

(continued from previous page)

```
ncmc: 100.0%   1000    93      0:00
NCMC MOVE REJECTED: work_ncmc -17.61007128922719 < -3.476065340494845
Advancing 1000 MD steps...
md: 15.0%      750     31.9    0:46
md: 20.0%      1000    43.1    0:32
md: 25.0%      1250    54.8    0:23
md: 30.0%      1500    65.5    0:18
BLUES Iteration: 1
Advancing 1000 NCMC switching steps...
ncmc: 25.0%    250     57.2    --
ncmc: 50.0%    500     62.6    0:13
Performing RandomLigandRotationMove...
ncmc: 75.0%    750     65.4    0:03
ncmc: 100.0%   1000    69.1    0:00
NCMC MOVE REJECTED: work_ncmc -28.930984747001787 < -2.602259467723195
Advancing 1000 MD steps...
md: 35.0%      1750    45.5    0:24
md: 40.0%      2000    50.5    0:20
md: 45.0%      2250    56.3    0:16
md: 50.0%      2500    61.9    0:13
BLUES Iteration: 2
Advancing 1000 NCMC switching steps...
ncmc: 25.0%    250     57.9    --
ncmc: 50.0%    500     60.1    0:25
Performing RandomLigandRotationMove...
ncmc: 75.0%    750     62.4    0:06
ncmc: 100.0%   1000    64.6    0:00
NCMC MOVE REJECTED: work_ncmc -9.178847171172448 < -0.3081492900307722
Advancing 1000 MD steps...
md: 55.0%      2750    49.7    0:15
md: 60.0%      3000    53.1    0:13
md: 65.0%      3250    56.7    0:10
md: 70.0%      3500    60.4    0:08
BLUES Iteration: 3
Advancing 1000 NCMC switching steps...
ncmc: 25.0%    250     58.4    --
ncmc: 50.0%    500     60.4    0:37
Performing RandomLigandRotationMove...
ncmc: 75.0%    750     61.6    0:09
ncmc: 100.0%   1000    63.6    0:00
NCMC MOVE REJECTED: work_ncmc -6.022089748510081 < -0.28447339331708726
Advancing 1000 MD steps...
md: 75.0%      3750    52.6    0:08
md: 80.0%      4000    55      0:06
md: 85.0%      4250    58      0:04
md: 90.0%      4500    60.8    0:02
BLUES Iteration: 4
Advancing 1000 NCMC switching steps...
ncmc: 25.0%    250     58.6    --
ncmc: 50.0%    500     59.8    0:49
Performing RandomLigandRotationMove...
ncmc: 75.0%    750     60.7    0:12
ncmc: 100.0%   1000    61.9    0:00
NCMC MOVE REJECTED: work_ncmc -63.43688730563919 < -0.5417610940298393
Advancing 1000 MD steps...
md: 95.0%      4750    53.3    0:01
md: 100.0%     5000    55.2    0:00
```

(continues on next page)

(continued from previous page)

```
md: 105.0%      5250    57.5    23:59:59
md: 110.0%      5500    59.8    23:59:58
Acceptance Ratio: 0.0
nIter: 5
```

[ ]:

Let us know if you have any problems or suggestions through our issue tracker:

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[smart-dart] I. Andricioaei, J. E. Straub, and A. F. Voter, J. Chem. Phys. 114, 6994 (2001). https://doi.org/10.1063/1.1358861

[amber-syntax] J. Swails, ParmEd Documentation (2015). http://parmed.github.io/ParmEd/html/amber.html#amber-mask-syntax

[TTPham-JChemPhys135-2011] T. T. Pham and M. R. Shirts, J. Chem. Phys 135, 034114 (2011). http://dx.doi.org/10.1063/1.3607597

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

F

G

I

L

M

N

O

P

Q

R

S

T

W

Z