blues Documentation

Release 0.2.5

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

Jun 18, 2019

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 Carlo

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

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

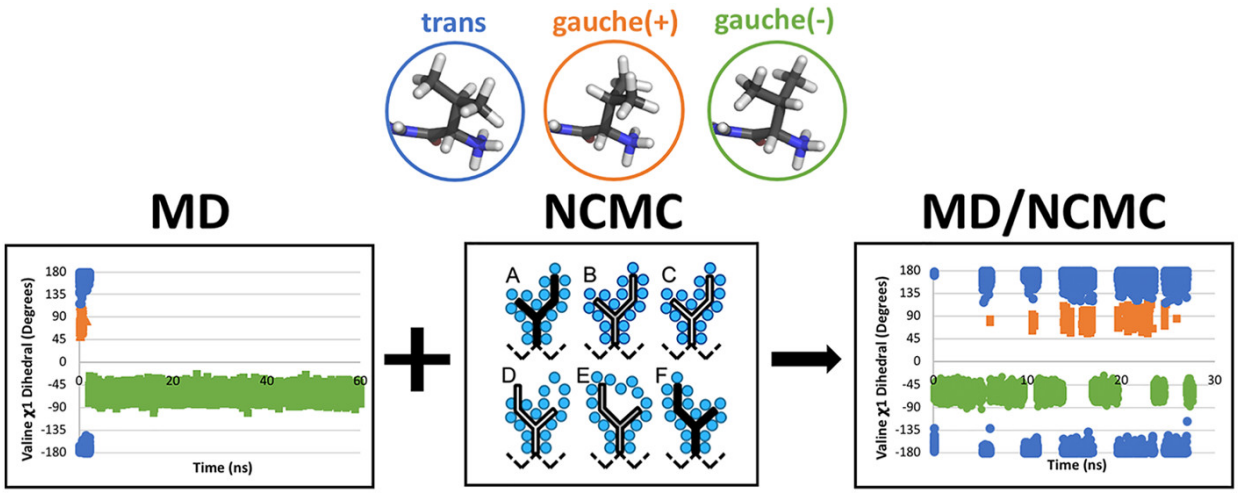
DOI: 10.1021/acs.jpcb.7b11820

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 Citations



Enhancing Side Chain Rotamer Sampling Using Nonequilibrium Candidate Monte Carlo

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

Journal of Chemical Theory and Computation. **2019** *15* (3), 1848-1862

DOI: *10.1021/acs.jctc.8b01018* <https://pubs.acs.org/doi/abs/10.1021/acs.jctc.8b01018>

Publication Date (Web): January 24, 2019

Abstract

Molecular simulations are a valuable tool for studying biomolecular motions and thermodynamics. However, such motions can be slow compared to simulation time scales, yet critical. Specifically, adequate sampling of side chain motions in protein binding pockets is crucial for obtaining accurate estimates of ligand binding free energies from molecular simulations. The time scale of side chain rotamer flips can range from a few ps to several hundred ns or longer, particularly in crowded environments like the interior of proteins. Here, we apply a mixed nonequilibrium candidate Monte Carlo (NCMC)/molecular dynamics (MD) method to enhance sampling of side chain rotamers. The NCMC portion of our method applies a switching protocol wherein the steric and electrostatic interactions between target side chain atoms and the surrounding environment are cycled off and then back on during the course of a move proposal. Between NCMC move proposals, simulation of the system continues via traditional molecular dynamics. Here, we first validate this approach on a simple, solvated valine-alanine dipeptide system and then apply it to a well-studied model ligand binding site in T4 lysozyme L99A. We compute the rate of rotamer transitions for a valine side chain using our approach and compare it to that of traditional molecular dynamics simulations. Here, we show that our NCMC/MD method substantially enhances side chain sampling, especially in systems where the torsional barrier to rotation is high (10 kcal/mol). These barriers can be intrinsic torsional barriers or steric barriers imposed by the environment. Overall, this may provide a promising strategy to selectively improve side chain sampling in molecular simulations.

1.4 Theory

Suggested readings:

CHAPTER

TWO

INSTALLATION

BLUES is compatible with MacOSX/Linux with Python>=3.6 (blues<1.1 still works with Python 2.7)

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
source activate blues
```

2.1 Stable Releases

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

```
conda install -c mobleylab blues
```

2.2 Development Builds

Alternatively, you can install the latest development build. Development builds contain the latest commits/PRs not yet issued in a point release.

```
conda install -c mobleylab/label/dev blues
```

In order to use the *SideChainMove* class you will need OpenEye Toolkits and some related tools.

```
conda install -c openeye/label/Orion -c omnia oeommtools packmol
conda install -c openeye openeye-toolkits
```

2.3 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
conda install -c omnia -c conda-forge openmmtools openmm numpy cython
pip install -e .
```

To validate your BLUES installation run the tests.

```
pip instal -e .[tests]
pytest -v -s
```

CHAPTER

THREE

USAGE

This package takes advantage of non-equilibrium candidate Monte Carlo moves (NCMC) to help sample between different ligand binding modes using the OpenMM simulation package. One goal for this package is to allow for easy additions of other moves of interest, which will be covered below.

The integrator from **BLUES** contains the framework necessary for NCMC. Specifically, the integrator class calculates the work done during a NCMC move. It also controls the lambda scaling of parameters. The integrator that BLUES uses inherits from `openmmtools.integrators.AlchemicalExternalLangevinIntegrator` to keep track of the work done outside integration steps, allowing Monte Carlo (MC) moves to be incorporated together with the NCMC thermodynamic perturbation protocol. Currently, the `openmmtools.alchemy` package is used to generate the lambda parameters for the ligand, allowing alchemical modification of the sterics and electrostatics of the system.

The **BLUESSampler** class in `ncmc.py` serves as a wrapper for running NCMC+MD simulations. To run the hybrid simulation, the **BLUESSampler** class requires defining two moves for running the (1) MD simulation and (2) the NCMC protcol. These moves are defined in the `ncmc.py` module. A simple example is provided below.

3.1 Example

Using the BLUES framework requires the use of a **ThermodynamicState** and **SamplerState** from `openmmtools` which we import from `openmmtools.states`:

```
from openmmtools.states import ThermodynamicState, SamplerState
from openmmtools.testsystems import TolueneVacuum
from blues.ncmc import *
from simtk import unit
```

Create the states for a toluene molecule in vacuum.

```
tol = TolueneVacuum()
thermodynamic_state = ThermodynamicState(tol.system, temperature=300*unit.kelvin)
sampler_state = SamplerState(positions=tol.positions)
```

Define our langevin dynamics move for the MD simulation portion and then our NCMC move which performs a random rotation. Here, we use a customized `openmmtools.mcmc.LangevinDynamicsMove` which allows us to store information from the MD simulation portion.

```
dynamics_move = ReportLangevinDynamicsMove(n_steps=10)
ncmc_move = RandomLigandRotationMove(n_steps=10, atom_subset=list(range(15)))
```

Provide the **BLUESSampler** class with an `openmm.Topology` and these objects to run the NCMC+MD simulation.

```
sampler = BLUESSampler(thermodynamic_state=thermodynamic_state,
                       sampler_state=sampler_state,
                       dynamics_move=dynamics_move,
                       ncmc_move=ncmc_move,
                       topology=tol.topology)
sampler.run(n_iterations=1)
```

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

CHAPTER

FOUR

MODULES

4.1 Nonequilibrium Candidate Monte Carlo (NCMC)

Provides moves and classes for running the BLUES simulation.

4.1.1 ReportLangevinDynamicsMove

class blues.ncmc.**ReportLangevinDynamicsMove**(*n_steps=1000*, *timestep=Quantity(value=2.0, unit=femtosecond)*, *collision_rate=Quantity(value=1.0, unit=/picosecond)*, *reassign_velocities=True*, *context_cache=None*, *reporters=[]*)

Langevin dynamics segment as a (pseudo) Monte Carlo move.

This move class allows the attachment of a reporter for storing the data from running this segment of dynamics. This move assigns a velocity from the Maxwell-Boltzmann distribution and executes a number of Maxwell-Boltzmann steps to propagate dynamics. This is not a *true* Monte Carlo move, in that the generation of the correct distribution is only exact in the limit of infinitely small timestep; in other words, the discretization error is assumed to be negligible. Use HybridMonteCarloMove instead to ensure the exact distribution is generated.

Warning: No Metropolization is used to ensure the correct phase space distribution is sampled. This means that timestep-dependent errors will remain uncorrected, and are amplified with larger timesteps. Use this move at your own risk!

Parameters

- **n_steps** (*int, optional*) – The number of integration timesteps to take each time the move is applied (default is 1000).
- **timestep** (*simtk.unit.Quantity, optional*) – The timestep to use for Langevin integration (time units, default is 1*simtk.unit.femtosecond).
- **collision_rate** (*simtk.unit.Quantity, optional*) – The collision rate with fictitious bath particles (1/time units, default is 10/simtk.unit.picoseconds).
- **reassign_velocities** (*bool, optional*) – If True, the velocities will be reassigned from the Maxwell-Boltzmann distribution at the beginning of the move (default is False).

- **context_cache** (*openmmtools.cache.ContextCache, optional*) – The ContextCache to use for Context creation. If None, the global cache openmmtools.cache.global_context_cache is used (default is None).
- **reporters** (*list*) – A list of the storage classes inteded for reporting the simulation data. This can be either blues.storage.(NetCDF4Storage/BLUESStateDataStorage).

n_steps
The number of integration timesteps to take each time the move is applied.

Type int

timestep
The timestep to use for Langevin integration (time units).

Type simtk.unit.Quantity

collision_rate
The collision rate with fictitious bath particles (1/time units).

Type simtk.unit.Quantity

reassign_velocities
If True, the velocities will be reassigned from the Maxwell-Boltzmann distribution at the beginning of the move.

Type bool

context_cache
The ContextCache to use for Context creation. If None, the global cache openmmtools.cache.global_context_cache is used.

Type openmmtools.cache.ContextCache

reporters
A list of the storage classes inteded for reporting the simulation data. This can be either blues.storage.(NetCDF4Storage/BLUESStateDataStorage).

Type list

Examples

First we need to create the thermodynamic state and the sampler state to propagate. Here we create an alanine dipeptide system in vacuum.

```
>>> from simtk import unit
>>> from openmmtools import testsystems
>>> from openmmtools.states import SamplerState, ThermodynamicState
>>> test = testsystems.AlanineDipeptideVacuum()
>>> sampler_state = SamplerState(positions=test.positions)
>>> thermodynamic_state = ThermodynamicState(system=test.system,␣
↪temperature=298*unit.kelvin)
```

Create reporters for storing our simulation data.

```
>>> from blues.storage import NetCDF4Storage, BLUESStateDataStorage
nc_storage = NetCDF4Storage('test-md.nc',
                            reportInterval=5,
                            crds=True, vels=True, frcs=True)
state_storage = BLUESStateDataStorage('test.log',
```

(continues on next page)

(continued from previous page)

```
                                        reportInterval=5,
                                        step=True, time=True,
                                        potentialEnergy=True,
                                        kineticEnergy=True,
                                        totalEnergy=True,
                                        temperature=True,
                                        volume=True,
                                        density=True,
                                        progress=True,
                                        remainingTime=True,
                                        speed=True,
                                        elapsedTime=True,
                                        systemMass=True,
                                        totalSteps=10)
```

Create a Langevin move with default parameters

```
>>> move = ReportLangevinDynamicsMove()
```

or create a Langevin move with specified parameters.

```
>>> move = ReportLangevinDynamicsMove(timestep=0.5*unit.femtoseconds,
                                      collision_rate=20.0/unit.picoseconds, n_
↪steps=10,
                                      reporters=[nc_storage, state_storage])
```

Perform one update of the sampler state. The sampler state is updated with the new state.

```
>>> move.apply(thermodynamic_state, sampler_state)
>>> np.allclose(sampler_state.positions, test.positions)
False
```

The same move can be applied to a different state, here an ideal gas.

```
>>> test = testsystems.IdealGas()
>>> sampler_state = SamplerState(positions=test.positions)
>>> thermodynamic_state = ThermodynamicState(system=test.system,
...                                          temperature=298*unit.kelvin)
>>> move.apply(thermodynamic_state, sampler_state)
>>> np.allclose(sampler_state.positions, test.positions)
False
```

apply (*thermodynamic_state*, *sampler_state*)

Propagate the state through the integrator.

This updates the SamplerState after the integration.

Parameters

- **thermodynamic_state** (*openmmtools.states.ThermodynamicState*) – The thermodynamic state to use to propagate dynamics.
- **sampler_state** (*openmmtools.states.SamplerState*) – The sampler state to apply the move to. This is modified.

4.1.2 NCMCMove

class blues.ncmc.**NCMCMove**(*n_steps=1000*, *timestep=Quantity(value=2.0, unit=femtosecond)*, *atom_subset=None*, *context_cache=None*, *nprop=1*, *propLambda=0.3*, *reporters=[]*)

A general NCMC move that applies an alchemical integrator.

This class is intended to be inherited by NCMCMoves that need to alchemically modify and perturb part of the system. The child class has to implement the _propose_positions method. Reporters can be attached to report data from the NCMC part of the simulation.

You can decide to override _before_integration() and _after_integration() to execute some code at specific points of the workflow, for example to read data from the Context before the it is destroyed.

Parameters

- **n_steps** (*int, optional*) – The number of integration timesteps to take each time the move is applied (default is 1000).
- **timestep** (*simtk.unit.Quantity, optional*) – The timestep to use for Langevin integration (time units, default is 1*simtk.unit.femtosecond).
- **atom_subset** (*slice or list of int, optional*) – If specified, the move is applied only to those atoms specified by these indices. If None, the move is applied to all atoms (default is None).
- **context_cache** (*openmmtools.cache.ContextCache, optional*) – The ContextCache to use for Context creation. If None, the global cache openmmtools.cache.global_context_cache is used (default is None).
- **reporters** (*list*) – A list of the storage classes inteded for reporting the simulation data. This can be either blues.storage.(NetCDF4Storage/BLUESStateDataStorage).

n_steps
The number of integration timesteps to take each time the move is applied.

Type int

timestep
The timestep to use for Langevin integration (time units).

Type simtk.unit.Quantity

atom_subset
If specified, the move is applied only to those atoms specified by these indices. If None, the move is applied to all atoms (default is None).

Type slice or list of int, optional

context_cache
The ContextCache to use for Context creation. If None, the global cache openmmtools.cache.global_context_cache is used.

Type openmmtools.cache.ContextCache

reporters
A list of the storage classes inteded for reporting the simulation data. This can be either blues.storage.(NetCDF4Storage/BLUESStateDataStorage).

Type list

property statistics
Statistics as a dictionary.

apply(*thermodynamic_state*, *sampler_state*)
Apply a move to the sampler state.

Parameters

- **thermodynamic_state** (*openmmtools.states.ThermodynamicState*) – The thermodynamic state to use to apply the move.
- **sampler_state** (*openmmtools.states.SamplerState*) – The initial sampler state to apply the move to. This is modified.

4.1.3 RandomLigandRotationMove

class blues.ncmc.**RandomLigandRotationMove**(*n_steps=1000*, *timestep=Quantity(value=2.0, unit=femtosecond)*, *atom_subset=None*, *context_cache=None*, *nprop=1*, *propLambda=0.3*, *reporters=[]*)
An NCMC move which proposes random rotations.

This class will propose a random rotation (as a rigid body) using the center of mass of the selected atoms. This class does not metropolize the proposed moves. Reporters can be attached to record the ncmc simulation data, mostly useful for debugging by storing coordinates of the proposed moves or monitoring the ncmc simulation progression by attaching a state reporter.

Parameters

- **n_steps** (*int, optional*) – The number of integration timesteps to take each time the move is applied (default is 1000).
- **timestep** (*simtk.unit.Quantity, optional*) – The timestep to use for Langevin integration (time units, default is 1*simtk.unit.femtosecond).
- **atom_subset** (*slice or list of int, optional*) – If specified, the move is applied only to those atoms specified by these indices. If None, the move is applied to all atoms (default is None).
- **context_cache** (*openmmtools.cache.ContextCache, optional*) – The ContextCache to use for Context creation. If None, the global cache openmmtools.cache.global_context_cache is used (default is None).
- **reporters** (*list*) – A list of the storage classes inteded for reporting the simulation data. This can be either blues.storage.(NetCDF4Storage/BLUESStateDataStorage).

n_steps
The number of integration timesteps to take each time the move is applied.

Type int

timestep
The timestep to use for Langevin integration (time units).

Type simtk.unit.Quantity

atom_subset
If specified, the move is applied only to those atoms specified by these indices. If None, the move is applied to all atoms (default is None).

Type slice or list of int, optional

context_cache
The ContextCache to use for Context creation. If None, the global cache openmmtools.cache.global_context_cache is used.

Type openmmtools.cache.ContextCache

reporters
A list of the storage classes inteded for reporting the simulation data. This can be either blues.storage.(NetCDF4Storage/BLUESStateDataStorage).

Type list

Examples

First we need to create the thermodynamic state, alchemical thermodynamic state, and the sampler state to propagate. Here we create a toy system of a charged ethylene molecule in between two charged particles.

```
>>> from simtk import unit
>>> from openmmtools import testsystems, alchemy
>>> from openmmtools.states import SamplerState, ThermodynamicState
>>> from blues.systemfactories import generateAlchSystem
>>> from blues import utils
```

```
>>> structure_pdb = utils.get_data_filename('blues', 'tests/data/ethylene_
↪structure.pdb')
>>> structure = parmed.load_file(structure_pdb)
>>> system_xml = utils.get_data_filename('blues', 'tests/data/ethylene_system.xml
↪')
    with open(system_xml, 'r') as infile:
        xml = infile.read()
        system = openmm.XmlSerializer.deserialize(xml)
>>> thermodynamic_state = ThermodynamicState(system=system, temperature=200*unit.
↪kelvin)
>>> sampler_state = SamplerState(positions=structure.positions.in_units_of(unit.
↪nanometers))
>>> alchemical_atoms = [2, 3, 4, 5, 6, 7]
>>> alch_system = generateAlchSystem(thermodynamic_state.get_system(), alchemical_
↪atoms)
>>> alch_state = alchemy.AlchemicalState.from_system(alch_system)
>>> alch_thermodynamic_state = ThermodynamicState(
        alch_system, thermodynamic_state.temperature)
>>> alch_thermodynamic_state = CompoundThermodynamicState(
        alch_thermodynamic_state, composable_states=[alch_state])
```

Create reporters for storing our ncmc simulation data.

```
>>> from blues.storage import NetCDF4Storage, BLUESStateDataStorage
nc_storage = NetCDF4Storage('test-ncmc.nc',
                            reportInterval=5,
                            crds=True, vels=True, frcs=True,
                            protocolWork=True, alchemicalLambda=True)
state_storage = BLUESStateDataStorage('test-ncmc.log',
                                      reportInterval=5,
                                      step=True, time=True,
                                      potentialEnergy=True,
                                      kineticEnergy=True,
                                      totalEnergy=True,
                                      temperature=True,
                                      volume=True,
                                      density=True,
                                      progress=True,
```

(continues on next page)

(continued from previous page)

```
                                        remainingTime=True,
                                        speed=True,
                                        elapsedTime=True,
                                        systemMass=True,
                                        totalSteps=10,
                                        protocolWork=True,
                                        alchemicalLambda=True)
```

Create a RandomLigandRotationMove move

```
>>> rot_move = RandomLigandRotationMove(n_steps=5,
                                        timestep=1*unit.femtoseconds,
                                        atom_subset=alchemical_atoms,
                                        reporters=[nc_storage, state_storage])
```

Perform one update of the sampler state. The sampler state is updated with the new state.

```
>>> move.apply(thermodynamic_state, sampler_state)
>>> np.allclose(sampler_state.positions, structure.positions)
False
```

4.1.4 BLUESSampler

class blues.ncmc.**BLUESSampler**(*thermodynamic_state=None*, *alch_thermodynamic_state=None*, *sampler_state=None*, *dynamics_move=None*, *ncmc_move=None*, *topology=None*)

BLUESSampler runs the NCMC+MD hybrid simulation.

This class ties together the two moves classes to execute the NCMC+MD hybrid simulation. One move class is intended to carry out traditional MD and the other is intended carry out the NCMC move proposals which performs the alchemical transformation to given atom subset. This class handles proper metropolization of the NCMC move proposals, while correcting for the switch in integrators.

equil(*n_iterations=1*)

Equilibrate the system for N iterations.

run(*n_iterations=1*)

Run the sampler for the specified number of iterations.

descriptive summary here

Parameters **niterations** (*int, optional, default=1*) – Number of iterations to run the sampler for.

4.2 SystemFactory

SystemFactory contains methods to generate/modify the OpenMM System object.

blues.systemfactory.**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*)

Return 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 nonbonded 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

J. Chem. Phys 135, 034114 (2011). http://dx.doi.org/10.1063/1.3607597

`blues.systemfactory.`**`zero_masses`**(*system*, *atomList=None*)

Zeroes the masses of specified atoms to constrain certain degrees of freedom.

Parameters

- **system** (*openmm.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.systemfactory.**restrain_positions**(*structure*, *system*, *selection='(@CA, C, N)'*, *weight=5.0*, ***kwargs*)

Apply 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.

blues.systemfactory.**freeze_atoms**(*structure*, *system*, *freeze_selection=':LIG'*, ***kwargs*)

Zero 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

blues.systemfactory.**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.

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

Add 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.

4.3 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*, *propLambda=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()
Manually reset protocol work and other statistics.

4.4 Utilities

Provides a host of utility functions for the BLUES engine.

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

blues.utils.**logger = <Logger blues.utils (WARNING)>**

blues.utils.**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

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.**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.

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.**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.**getMasses**(*atom_subset*, *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

blues.utils.**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

blues.utils.**saveContextFrame**(*context*, *topology*, *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**(*context*)
Prints hardware related information for the openmm.Simulation

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

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

4.5 Storage

blues.storage.**setup_logging**(*filename=None*, *yml_path='logging.yml'*, *default_level=20*, *env_key='LOG_CFG'*)
Setup logging configuration

blues.storage.**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.storage.**init_logger**(*logger*, *level=20*, *stream=True*, *outfname='blues-20190618-164120'*)
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, warningING, 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.storage.**NetCDF4Storage**(*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(*context_state*)

Get information about the next report this object will generate.

Parameters **context_state** (`openmm.State`) – The current state of the context

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(*context_state*, *integrator*)

Generate a report.

Parameters

- **context_state** (`openmm.State`) – The current state of the context
- **integrator** (`openmm.Integrator`) – The integrator belonging to the given context

class blues.storage.**BLUESStateDataStorage**(*file=None*, *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`(*context_state*)

Get information about the next report this object will generate.

Parameters **context_state** (`openmm.State`) – The current state of the context

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 (*context_state*, *integrator*)
Generate a report.

Parameters

- **context_state** (`openmm.State`) – The current state of the context
- **integrator** (`openmm.Integrator`) – The integrator belonging to the given context

4.6 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.**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

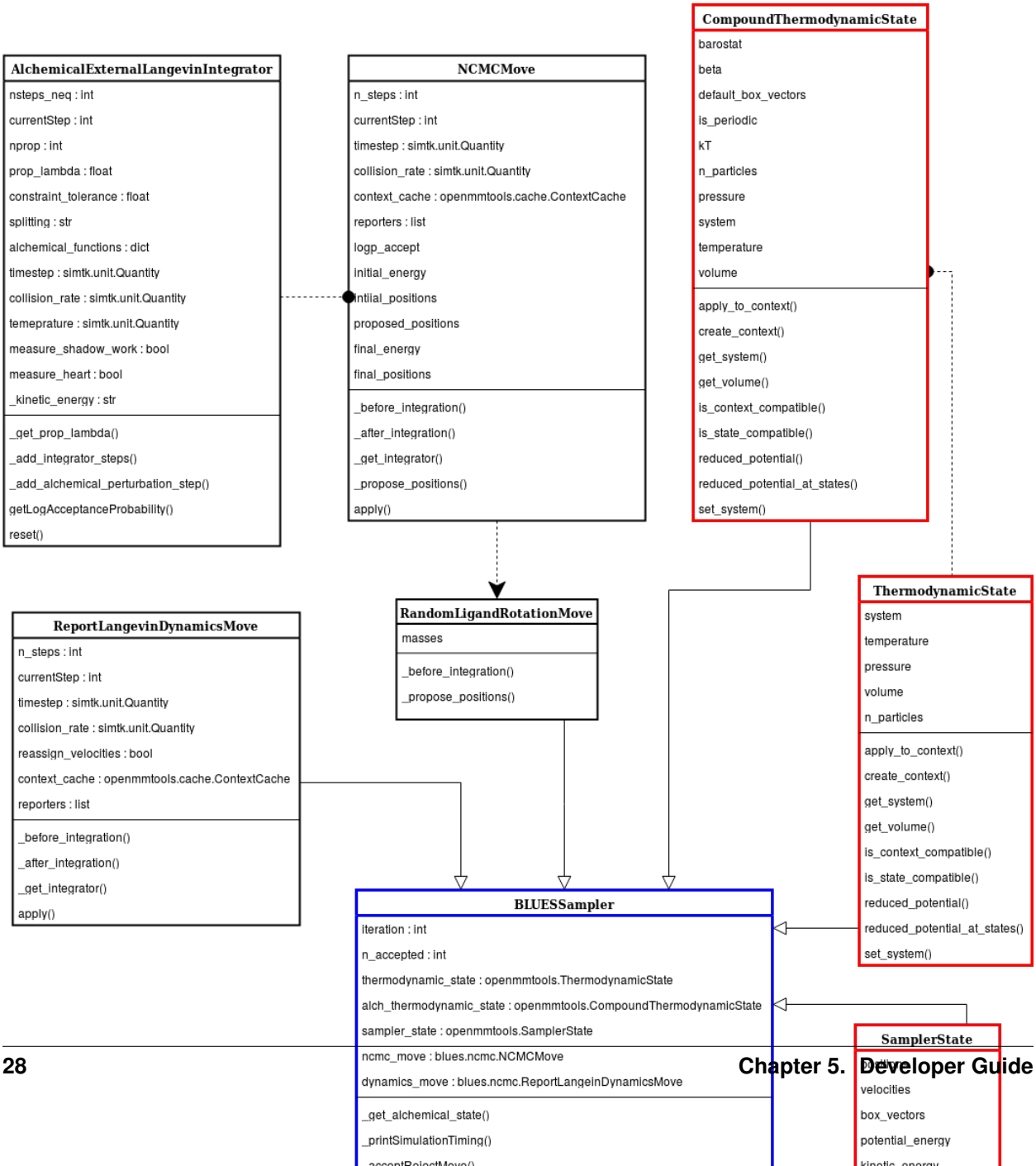
Parameters **stuff** (*float or time-dimension Quantity*) – The time to add to the current frame

CHAPTER

FIVE

DEVELOPER GUIDE

5.1 UML Diagram



5.2 OpenMMTools Objects

Highlighted in red are 3 objects that we use from the `openmmtools` library. They are the **ThermodynamicState**, **CompoundThermodynamicState**, and **SamplerState** objects. For more details of each class, please see the official openmmtools documentation.

Briefly, the **ThermodynamicState** class represents the portion of the state of an `openmm.Context` that does not change with integration (i.e. particles, temperature, or pressure). The **CompoundThermodynamicState** class is essentially the same as the **ThermodynamicState** class except in this package, it is used for the handling the `openmmtools.alchemy.AlchemicalState` object. Thus, in order to create the **CompoundThermodynamicState**, one needs to first create the plain **ThermodynamicState** object first. If a **CompoundThermodynamicState** object is not provided to the `blues.ncmc.BLUESSampler` class, one is created using the default parameters from the given **ThermodynamicState**. Lastly, the **SamplerState** class represents the state of an `openmm.Context` which does change with integration (i.e positions, velocities, and box_vectors). Within the context of this package, the **SamplerState** is used to sync information between the MD and NCMC simulations.

5.3 Integrators and Moves

Integrators Integrators are the lowest level openmm objects this package interacts with, where each intergrator is tied to an `openmm.Context` that it advances. Each integrator is generated by using the embedded function `_get_integrator()` function within each move class. The integrators will control whether we are carrying out the Non-equilibirum Candidate Monte Carlo (NCMC) or Molecular Dynamics (MD) simulation.

Every move class has 3 hidden methods: `_get_integrator()` for generating the integrator of each move class, `_before_integration()` for performing any necessary setup before integration, and `_after_integration()` for performing any cleanup or data collection after integration. Every move class also contains the `apply()` method which carries out calls to the 3 hidden methods and stepping with the integrator.

In this package, we provide the move class `blues.ncmc.ReportLangevinDynamicsMove` to execute the MD simulation. As the name suggests, this will carry forward the MD simulation using Langevin dynamics, by generating an `openmm.LangevinIntegrator`. This class is essentially the same as the `openmmtools.LangevinDynamicsMove` but with modifications to the `apply()` method which allows storing simulation data for the MD simulation.

For running the NCMC simulation, we provide a custom integrator `blues.integrator.AlchemicalExternalLangevinIntegrator`. This integrator is generated in every move which inherits from the base class `blues.ncmc.NCMCMove`. Every class which inherits from the base move class must override the `_propose_positions()` method. If necessary, one can override the `_before_integration()` and `_after_integration()` methods for any necessary setup and cleanup. Again, these hidden methods will be called when a call is made to the `apply()` method from the move class.

Moves In order to implement custom NCMC moves, inherit from the base class and override the `_propose_positions()` method. This method is expected to take in a positions array of the atoms to be modified and returns the proposed positions. In pseudo-code, it would look something like:

```
from blues.ncmc import NCMCMove
class CustomNCMCMove(NCMCMove):
    def _propose_positions(positions):
        """Add 1 nanometer displacement vector."""
        positions_unit = positions.unit
        unitless_displacement = 1.0 / positions_unit
        displacement_vector = unit.Quantity(np.random.randn(3) * unitless_
↪displacement_sigma, positions_unit)
        proposed_positions = positions + displacement_vector
        return proposed_positions
```

In this package, we provide the `blues.ncmc.RandomLigandRotationMove` in order to propose a random ligand rotation about the center of mass. This class overrides the `_before_integration()` method for obtaining the masses of the ligand and overrides the `_propose_positions()` function for generating the rotated coordinates. Updating the context with the rotated coordinates is handled when the `apply()` method is called in the move class. Code snippet of the class is shown below:

```
from blues.ncmc import RandomLigandRotationMove
class RandomLigandRotationMove(NCMCMove):
   def _before_integration(self, context, thermodynamic_state):
       """Obtain the masses of the ligand before integration."""
       super(RandomLigandRotationMove, self)._before_integration(context,␣
↪thermodynamic_state)
       masses, totalmass = utils.getMasses(self.atom_subset, thermodynamic_state.
↪topology)
       self.masses = masses
   def _propose_positions(self, positions):
        # Calculate the center of mass
        center_of_mass = utils.getCenterOfMass(positions, self.masses)
        reduced_pos = positions - center_of_mass
        # Define random rotational move on the ligand
        rand_quat = mdtraj.utils.uniform_quaternion(size=None)
        rand_rotation_matrix = mdtraj.utils.rotation_matrix_from_quaternion(rand_quat)
        # multiply lig coordinates by rot matrix and add back COM translation from␣
↪origin
        proposed_positions = numpy.dot(reduced_pos, rand_rotation_matrix) * positions.
↪unit + center_of_mass

        return proposed_positions
```

Since BLUES (v0.2.5) the API has been re-written to be more compatible with the `openmmtools` API. This means one can turn a regular Markov Chain Monte Carlo (MCMC) move from the `openmmtools` library into an NCMC move to be used in this package. In this case, one simply needs to make use of dual inheritance, using the `blues.ncmc.NCMCMove` that we provide and override the `_get_integrator()` method to generate the NCMC integrator we provide, i.e. `blues.integrator.AlchemicalExternalLangevinIntegrator`. When using dual inheritance, it is important that you first inherit the desired MCMC move and then the `blues.ncmc.NCMCMove` class. For example, if we wanted to take the `openmmtools.mcmc.MCDisplacementMove` class and turn it into an NCMC move, it would look like:

```
from blues.ncmc import NCMCMove
from openmmtools.mcmc import MCDisplacementMove
class NCMCDisplacementMove(MCDisplacementMove, NCMCMove):
    def _get_integrator(self, thermodynamic_state):
        return NCMCMove._get_integrator(self,thermodynamic_state)
```

5.4 BLUESSampler

The `blues.ncmc.BLUESSampler` object ties together all the previously mentioned state objects and the two move classes for running the NCMC+MD simulation. Details of the parameters for this class are listed in the *Modules* documentation. For a more detailed example of it's usage see the *Usage* documentation.

To be explicit, the input parameters refer to the objects below:

- **thermodynamic_state** : `openmmtools.states.ThermodynamicState`
- **alch_thermodynamic_state** : `openmmtools.states.CompoundThermodynamicState`
- **sampler_state** : `openmmtools.states.SamplerState`

- **dynamics_move** : `blues.ncmc.ReportLangevinDynamicsMove`
- **ncmc_move** : `blues.ncmc.RandomLigandRotationMove`
- **topology** : `openmm.Topology`

When the `run()` method in the `blues.ncmc.BLUESSampler` is called the following takes place:

- **Initialization:**
  - `_print_host_info()` : Information print out of host
  - `_printSimulationTiming()` : Calculation of total number of steps
  - `equil()` : Equilibration
- **BLUES iterations:**
  - `ncmc_move.apply()` : NCMC simulation
  - `_acceptRejectMove()` : Metropolization
  - `dynamics_move.apply()` : MD Simulation

A code snippet of the `run()` method is shown below:

```
def run(self, n_iterations=1):
    context, integrator = cache.global_context_cache.get_context(self.thermodynamic_
↪state)
    utils.print_host_info(context)
    self._printSimulationTiming(n_iterations)
    if self.iteration == 0:
        self.equil(1)

    self.iteration = 0
    for iteration in range(n_iterations):
        self.ncmc_move.apply(self.alch_thermodynamic_state, self.sampler_state)

        self._acceptRejectMove()

        self.dynamics_move.apply(self.thermodynamic_state, self.sampler_state)

        self.iteration += 1
```

5.4.1 Initialization

The first thing that occurs when `run()` is called is the initialization stage. During this stage, a call is made to `utils.print_host_info()` and the `_printSimulationTiming()` method which will print out some information about the host machine and the total number of force evaluations and simulation time. The output will look something like below:

In the `blues.ncmc.BLUESSampler` class, there is an `equil()` method which lets you run iterations of just the MD simulation in order to equilibrate your system before running the NCMC+MD hybrid simulation. An equilibration iteration, in this case is controlled by the given attribute *n_steps* from the *dynamics_move* class. For example, if I create a `blues.ncmc.ReportLangevinDynamicsMove` class with *n_steps=20* and call the `blues.ncmc.BLUESSampler.equil(n_iterations=100)`, this will run *(n_steps x n_iterations)* or 2000 steps of MD or 2 picoseconds of MD simulation time. When the `run()` method is called without a prior call to the `equil()` method, the class will always run 1 iteration of equilibration in order to set the initial conditions in the MD simulation. This is required prior to running the NCMC simulation.

5.4.2 BLUES Iterations

NCMC Simulation

After at least 1 iteration of equilibration, the `blues.ncmc.BLUESSampler` class will then proceed forward with running iterations of the NCMC+MD hybrid simulation. It will first run the NCMC simulation by calling the `apply()` method on the **ncmc_move** class or, for sake of this example, the `blues.ncmc.RandomLigandRotationMove` class. The `apply()` method for the **ncmc_move** will take in the **alch_thermodynamic_state** parameter or specifically the `openmmtools.states.CompoundThermodynamicState` object.

A code snippet of the `ncmc_move.apply()` method is shown below:

```
def apply(self, thermodynamic_state, sampler_state):
    if self.context_cache is None:
        context_cache = cache.global_context_cache
    else:
        context_cache = self.context_cache
    integrator = self._get_integrator(thermodynamic_state)
    context, integrator = context_cache.get_context(thermodynamic_state, integrator)
    sampler_state.apply_to_context(context, ignore_velocities=False)
    self._before_integration(context, thermodynamic_state)
    try:
        endStep = self.currentStep + self.n_steps
        while self.currentStep < endStep:
            alch_lambda = integrator.getGlobalVariableByName('lambda')
            if alch_lambda == 0.5:
                sampler_state.update_from_context(context)
                proposed_positions = self._propose_positions(sampler_state.
↪positions[self.atom_subset])
                sampler_state.positions[self.atom_subset] = proposed_positions
                sampler_state.apply_to_context(context, ignore_velocities=True)

            nextSteps = endStep - self.currentStep
            stepsToGo = nextSteps
            while stepsToGo > 10:
                integrator.step(10)
                stepsToGo -= 10
            integrator.step(stepsToGo)
            self.currentStep += nextSteps
    except Exception as e:
        print(e)
    else:
        context_state = context.getState(
            getPositions=True,
            getVelocities=True,
            getEnergy=True,
            enforcePeriodicBox=thermodynamic_state.is_periodic)

        self._after_integration(context, thermodynamic_state)
        sampler_state.update_from_context(
            context_state, ignore_positions=False, ignore_velocities=False, ignore_
↪collective_variables=True)
        sampler_state.update_from_context(
            context, ignore_positions=True, ignore_velocities=True, ignore_collective_
↪variables=False)
```

When the `apply()` method on **ncmc_move** is called, it will first generate the `blues.integrators.AlchemicalExternalLangevinIntegrator` by calling the `_get_integrator()` method inherent to the

move class. Then, it will create (or fetch from the **context_cache**) a corresponding `openmm.Context` given the **alch_thermodynamic_state**. Next, the **sampler_state** which contains the last state of the MD simulation is synced to the newly created context from the corresponding **alch_thermodynamic_state**. Particularly, the context will be updated with the *box_vectors*, *positions*, and *velocities* from the last state of the MD simulation.

Just prior to integration, a call is made to the `_before_integration()` method in order to store the initial *energies*, *positions*, *box_vectors* and the *masses* of the ligand to be rotated. Then, we actually step with the integrator where we perform the ligand rotation when *lambda* has reached the half-way point or *lambda=0.5*, continuing integration until we have completed the *n_steps*. After the integration steps have been completed, a call is made to the `_after_integration()` method to store the final *energies*, *positions*, and *box_vectors*. Lastly, the **sampler_state** is updated from the final state of the context.

Metropolization

After advancing the NCMC simulation, a call is made to the `_acceptRejectMove()` method embedded in the `blues.ncmc.BLUESSampler` class for metropolization of the proposed move.

A code snippet of the `_acceptRejectMove()` is shown below:

```
def _acceptRejectMove(self):
    integrator = self.dynamics_move._get_integrator(self.thermodynamic_state)
    context, integrator = cache.global_context_cache.get_context(self.thermodynamic_
↪state, integrator)
    self.sampler_state.apply_to_context(context, ignore_velocities=True)
    alch_energy = self.thermodynamic_state.reduced_potential(context)

    correction_factor = (self.ncmc_move.initial_energy - self.dynamics_move.final_
↪energy + alch_energy - self.ncmc_move.final_energy)
    logp_accept = self.ncmc_move.logp_accept
    randnum = numpy.log(numpy.random.random())

    logp_accept = logp_accept + correction_factor
    if (not numpy.isnan(logp_accept) and logp_accept > randnum):
        self.n_accepted += 1
    else:
        self.accept = False
        self.sampler_state.positions = self.ncmc_move.initial_positions
        self.sampler_state.box_vectors = self.ncmc_move.initial_box_vectors
```

Here, is we compute a correction term for switching between the MD and NCMC integrators and factor this in with natural log of the acceptance probability (**logp_accept**). Then, a random number is generated in which: the move is accepted if the random number is less than the **logp_accept** or rejected if greater. When the move is rejected, we set the *positions* and *box_vectors* on the **sampler_state** to the initial positions and box_vectors from the NCMC simulation. If the move is accepted, nothing on the **sampler_state** is changed so that the following MD simulation will contain the final state of the NCMC simulation.

MD Simulation

After metropolization of the previously proposed move, a call is made to the `apply()` method on the given **dynamics_move** object. In this example, this would refer to the `blues.ncmc.ReportLangevinDynamicsMove` class to run the MD simulation.

A code snippet of the `dynamics_move.apply()` method is shown below:

```
def apply(self, thermodynamic_state, sampler_state):
    if self.context_cache is None:
        context_cache = cache.global_context_cache
    else:
        context_cache = self.context_cache
```

(continues on next page)

(continued from previous page)

```
    integrator = self._get_integrator(thermodynamic_state)
    context, integrator = context_cache.get_context(thermodynamic_state, integrator)
    thermodynamic_state.apply_to_context(context)

    sampler_state.apply_to_context(context, ignore_velocities=self.reassign_
↪velocities)
    if self.reassign_velocities:
        context.setVelocitiesToTemperature(thermodynamic_state.temperature)

    self._before_integration(context, thermodynamic_state)
    try:
        endStep = self.currentStep + self.n_steps
        while self.currentStep < endStep:
            nextSteps = endStep - self.currentStep
            stepsToGo = nextSteps
            while stepsToGo > 10:
                integrator.step(10)
                stepsToGo -= 10
            integrator.step(stepsToGo)
            self.currentStep += nextSteps

    except Exception as e:
        print(e)

    else:
        context_state = context.getState(
            getPositions=True,
            getVelocities=True,
            getEnergy=True,
            enforcePeriodicBox=thermodynamic_state.is_periodic)
        self._after_integration(context, thermodynamic_state)
        sampler_state.update_from_context(
            context_state, ignore_positions=False, ignore_velocities=False, ignore_
↪collective_variables=True)
        sampler_state.update_from_context(
            context, ignore_positions=True, ignore_velocities=True, ignore_collective_
↪variables=False)
```

When the `apply()` method is called, a very similar procedure to the NCMC simulation occurs. The first thing that happens is to generate the integrator through a call to `_get_integrator()`, where in this given class, it will generate an `openmm.LangevinIntegrator` given the **thermodynamic_state** parameter. Then, it will create (or fetch from the **context_cache**) a corresponding `openmm.Context` given the **thermodynamic_state**. Next, the **sampler_state**, which contains the last state of the NCMC simulation if the previous move was accepted or the initial state of the NCMC simulation if the move was rejected, is used to update *box_vectors* and *positions* in the newly created `openmm.Context`. In this case, we reassign the *velocities* in the MD simulation in order to preserve detailed balance.

Following, a call is made to `_before_integration()` to store the intial *positions*, *box_vectors* and *energies* and then we carry forward with the integration for *n_steps*. After the integration steps have been completed, a call is made to the `_after_integration()` method to store the final *energies* and *positions*. Lastly, the **sampler_state** object is updated from the final state of the MD simulation context.

This completes 1 iteration of the BLUES cycle. Here, the **sampler_state** is then used to sync the final state of the MD simulation (i.e. *box_vectors*, *positions*, and *velocities*) from the previous iteration to the NCMC simulation of the next iteration. Then, we repeat the cycle of **NCMC -> Metropolization -> MD** for the given number of iterations.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[TTPham-JChemPhys135-2011] T. T. Pham and M. R. Shirts,

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

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

I

L

N

O

P

R

S

T

Z