openmmtools Documentation

The Chodera Lab, MSKCC

Jan 24, 2019

Contents:

Caution: This is module is undergoing heavy development. None of the API calls are final. This software is provided without any guarantees of correctness, you will likely encounter bugs.

If you are interested in this code, please wait for the official release to use it. In the mean time, to stay informed of development progress you are encouraged to:

- Follow this feed for updates on releases.
- Check out the github repository .

A batteries-included toolkit for the GPU-accelerated OpenMM molecular simulation engine.

`openmmtools` is a Python library layer that sits on top of OpenMM to provide access to a variety of useful tools for building full-featured molecular simulation packages.

Features include:

- high-quality Langevin integrators, including g-BAOAB, VVVR, and other splittings
- integrators that support nonequilibrium switching for free energy calculations or nonequilibrium candidate Monte Carlo (NCMC)
- an extensible Markov chain Monte Carlo framework for mixing Monte Carlo and molecular dynamics-based methods
- factories for generating alchemically-modified systems for absolute and relative free energy calculations
- a suite of test systems for benchmarking, validation, and debugging

You can go through the *getting started tutorial* for an overview of the library or the *developer's guide* for information on how to extend the existing features.

CHAPTER 1

Installation

1.1 Installing via *conda*

The simplest way to install `openmmtools` is via the conda package manager. Packages are provided on the omnia Anaconda Cloud channel for Linux, OS X, and Win platforms. The openmmtools Anaconda Cloud page has useful instructions and download statistics.

If you are using the anaconda scientific Python distribution, you already have the `conda` package manager installed. If not, the quickest way to get started is to install the miniconda distribution, a lightweight minimal installation of Anaconda Python.

On `linux`, you can install the Python 3 version into `$HOME/miniconda3` with (on `bash` systems):

```
$ wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ bash ./Miniconda3-latest-Linux-x86_64.sh -b -p $HOME/miniconda3
$ export PATH="$HOME/miniconda3/bin:$PATH"
```

On `osx`, you want to use the `osx` binary

```
$ wget https://repo.continuum.io/miniconda/Miniconda2-latest-MacOSX-x86_64.sh
$ bash ./Miniconda3-latest-Linux-x86_64.sh -b -p $HOME/miniconda3
$ export PATH="$HOME/miniconda3/bin:$PATH"
```

You may want to add the new `$PATH extension to your `~/.bashrc` file to ensure Anaconda Python is used by default. Note that openmmtools will be installed into this local Python installation, so that you will not need to worry about disrupting existing Python installations.

Note: `conda` installation is the preferred method since all dependencies are automatically fetched and installed for you.

1.1.1 Release build

You can install the latest stable release build of openmmtools via the `conda` package with

```
$ conda config --add channels omnia --add channels conda-forge
$ conda install openmmtools
```

This version is recommended for all users not actively developing new algorithms for alchemical free energy calculations.

Note: `conda` will automatically dependencies from binary packages automatically, including difficult-to-install packages such as OpenMM, numpy, and scipy. This is really the easiest way to get started.

1.1.2 Development build

The bleeding-edge, absolute latest, very likely unstable development build of openmmtools is pushed to binstar with each GitHub commit, and can be obtained by

```
$ conda config --add channels omnia --add channels conda-forge
$ conda install openmmtools-dev
```

Warning: Development builds may be unstable and are generally subjected to less testing than releases. Use at your own risk!

1.1.3 Upgrading your installation

To update an earlier `conda` installation of openmmtools to the latest release version, you can use `conda update`:

```
$ conda update openmmtools
```

CHAPTER 2

Getting started tutorial

This tutorial will give you an overview of what you can find in OpenMMTools and how you can use the library.

Contents

2.1 Test systems, integrators, and forces

In its basic usage, OpenMMTools extends OpenMM by providing pre-packaged systems, integrators of force objects that are not natively implemented in OpenMM.

2.1.1 Test systems

The *testsystems* module comes with many simulation-ready molecular systems (from analytically-solvable systems to a kinase in explicit solvent) that can be useful for prototyping, validation, testing, and benchmarking. The code below creates a TIP3P water cubic box of 2nm side using PME.

```
from simtk import openmm, unit
from openmmtools import testsystems

water_box = testsystems.WaterBox(box_edge=2.0*unit.nanometer)
system = water_box.system  # An OpenMM System object.
positions = water_box.positions  # Initial coordinates for the system with associated␣
↪units.
```

You can use select a subset of the system atoms using the atom selection domain-specific language (DSL) implemented in MDTraj. For example, the following snippet create a T4-Lysozyme system in implicit OBC GBSA solvent bound to a p-xylene molecule, and finds the atom indices corresponding to the heavy atoms of p-xylene and few residues surrounding the binding site of T4-Lysozyme.

```
lysozyme_pxylene = testsystems.LysozymeImplicit()
t4_system = lysozyme_pxylene.system
pxylene_dsl = '(resname TMP) and (mass > 1.5)'  # Select heavy atoms of p-xylene.
binding_site_dsl = ('(resi 77 or resi 86 or resi 101 or resi 110 or '
                    ' resi 117 or resi 120) and (mass > 1.5)')
pxylene_atom_indices = lysozyme_pxylene.mdtraj_topology.select(pxylene_dsl).tolist()
binding_site_atom_indices = lysozyme_pxylene.mdtraj_topology.select(binding_site_dsl).
↪tolist()
```

2.1.2 Integrators

The systems created by `testsystems` can then be propagated in the usual way with OpenMM. The *integrators* module provide several high-quality integrators for equilibrium and non-equilibrium simulations in OpenMM.

```
from openmmtools import integrators

integrator = integrators.LangevinIntegrator(temperature=298.0*unit.kelvin,
                                            collision_rate=1.0/unit.picoseconds,
                                            timestep=1.0*unit.femtoseconds)
context = openmm.Context(t4_system, integrator)
context.setPositions(lysozyme_pxylene.positions)
integrator.step(n_steps)
```

Our `LangevinIntegrator` allows you to specify the splitting used to carry out the numerical integration. By default, OpenMMTools will construct a BAOAB integrator (i.e. with *V R O R V* splitting), which was shown empirically to add a very small integration error in configurational space, but other solutions are possible.

```
integrator = integrators.LangevinIntegrator(splitting="V0 V1 R R O R R V1 R R O R R␣
↪V1 V0",
                                            measure_shadow_work=True, measure_
↪heat=True)
context = openmm.Context(t4_system, integrator)
context.setPositions(lysozyme_pxylene.positions)
integrator.step(n_steps)

# Obtain the dissipated heat accumulated during Langevin dynamics in molar energy␣
↪units.
heat = integrator.get_heat()
```

The integrator above, for example, implements the geodesic-BAOAB Langevin integrator with solute-solvent splitting, and it collects statistics on the dissipated heat and the shadow work during the propagation (at the cost of a computational overhead).

2.1.3 Forces

The *forces* module is still under construction, but it already provides a few convenient utility functions and force objects. Let's create a T4-Lysozyme system in implicit OBC GBSA solvent bound to a p-xylene and add a harmonic restraint between the two molecules.

```
from openmmtools import forces

harmonic_restraint = forces.HarmonicRestraintForce(spring_constant=0.2*unit.
↪kilocalories_per_mole/unit.angstrom**2,
                                                   restrained_atom_indices1=binding_
↪site_atom_indices,
                                                   restrained_atom_indices2=pxylene_
↪atom_indices)
t4_system.addForce(harmonic_restraint)
```

The restraint force above will place a single harmonic potential between the centers of mass of the heavy atoms of the p-xylene molecule and the binding site of T4-Lysozyme.

The function `forces.find_forces()` provides a convenient way to search for particular force objects in the OpenMM `System`.

```
# Retrieve our harmonic restraint force.
forces.find_forces(t4_system, force_type=forces.HarmonicRestraintForce)

# Find all forces that inherit from an OpenMM CustomBondForce object.
forces.find_forces(t4_system, force_type=openmm.CustomBondForce, include_
↪subclasses=True)
```

(continues on next page)

(continued from previous page)

```
# Search for force names using regular expressions.
# Return all openmm.HarmonicBondForce, openmm.HarmonicAngleForce,
# and forces.HarmonicRestraintForce force objects.
forces.find_forces(t4_system, '.*Harmonic.*')
```

2.2 Alchemical transformations

The *alchemy* module provides helper classes to perform alchemical transformations with OpenMM.

2.2.1 AbsoluteAlchemicalFactory and AlchemicalState

The `AbsoluteAlchemicalFactory` class prepare OpenMM `System` objects for alchemical manipulation. Let's create an alchemical system that we can use to alchemically decouple p-xylene from T4-lysozyme's binding pocket.

```
>>> from openmmtools import alchemy

>>> # Create the reference OpenMM System that will be alchemically modified.
>>> lysozyme_pxylene = testsystems.LysozymeImplicit()
>>> t4_system = lysozyme_pxylene.system

>>> # Define the region of the System to be alchemically modified.
>>> pxylene_atoms = lysozyme_pxylene.mdtraj_topology.select('resname TMP')
>>> alchemical_region = alchemy.AlchemicalRegion(alchemical_atoms=pxylene_atoms)

>>> factory = alchemy.AbsoluteAlchemicalFactory()
>>> alchemical_system = factory.create_alchemical_system(t4_system, alchemical_region)
```

At this point, the p-xylene in alchemical `System` is in its interacting state and it can be then simulated normally

```
>>> integrator = integrators.LangevinIntegrator()
>>> context = openmm.Context(alchemical_system, integrator)
>>> context.setPositions(lysozyme_pxylene.positions)
>>> integrator.step(n_steps)
```

The alchemical degrees of freedom of the Hamiltonian can be controlled during the simulation through the `AlchemicalState` class.

```
>>> alchemical_state = alchemy.AlchemicalState.from_system(alchemical_system)
>>> alchemical_state.lambda_electrostatics = 0.0
>>> alchemical_state.lambda_sterics = 0.5
>>> alchemical_state.apply_to_context(context)
```

The snippet above modifies the simulated `System` to completely turn off the electrostatics interaction and halve the Lennard-Jones potential between p-xylene and its environment.

Note: In OpenMMTools, the convention is to have the interacting state at lambda=1 and the non-interacting state at lambda=0. Some packages adopt the opposite convention.

Note: The `AbsoluteAlchemicalFactory` class is currently specialized for absolute calculations in the sense that it cannot prepare an OpenMM `System` to have an atom changing its element or turn on part of a molecule while decoupling another set of atoms. We're planning to provide these capabilities in the near future.

2.2.2 Decoupling vs annihilating and softcore nonbonded interactions

By default, the alchemical `System` is prepared to annihilate electrostatics (i.e. turn off the alchemical atoms' charges) and decouple the sterics (i.e. preserve the intra-molecular Lennard-Jones interactions), but you can maintain the intra-molecular charges, for example, by configuring the alchemical region.

```
alchemical_region = alchemy.AlchemicalRegion(alchemical_atoms=pxylene_atoms,
                                             annihilate_electrostatics=True)
alchemical_system = factory.create_alchemical_system(t4_system, alchemical_region)
```

Similarly, you can set specific softcore parameters for the sterics and electrostatics interactions (see the API documentation for a detailed explanation of the parameters).

```
alchemical_region = alchemy.AlchemicalRegion(alchemical_atoms=pxylene_atoms,
                                             softcore_alpha=0.5, softcore_c=6)
```

2.2.3 Softening torsions, angles, and bonds

Beside nonbonded interactions, it is possible to modify other terms of the potentials. The following alchemical region is configured to modify the OpenMM `System` to enable torsion softening of all the p-xylene dihedrals. The Hamiltonian parameter controlling the torsion, angles, and bond potential terms can be controlled with `AlchemicalState` in the same way as with nonbonded interactions.

```
alchemical_region = alchemy.AlchemicalRegion(alchemical_atoms=pxylene_atoms,
                                             alchemical_torsions=True)
alchemical_system = factory.create_alchemical_system(t4_system, alchemical_region)
context = openmm.Context(alchemical_system, integrators.LangevinIntegrator())

alchemical_state = alchemy.AlchemicalState.from_system(alchemical_system)
alchemical_state.lambda_torsions = 0.8
alchemical_state.apply_to_context(context)
```

2.2.4 Alchemical functions

Finally you can enslave the degrees of freedom of the Hamiltonian to a variable through a custom function. The code below configure the `AlchemicalState` to turn off first electrostatic and the steric interactions one after the other as a generic variable called `lambda` goes from `1.0` to `0.0`.

```
# Enslave lambda_sterics and lambda_electrostatics to a generic lambda variable.
alchemical_state.set_function_variable('lambda', 1.0)

# The functions here turn off first electrostatic and the steric interactions
# in sequence as lambda goes from 1.0 to 0.0.
f_electrostatics = '2*(lambda-0.5)*step(lambda-0.5)'
f_sterics = '2*lambda*step_hm(0.5-lambda) + step_hm(lambda-0.5)'
alchemical_state.lambda_electrostatics = alchemy.AlchemicalFunction(f_electrostatics)
```

(continues on next page)

(continued from previous page)

```
alchemical_state.lambda_sterics = alchemy.AlchemicalFunction(f_sterics)

alchemical_state.set_function_variable('lambda', 0.75)
assert alchemical_state.lambda_electrostatics == 0.5
assert alchemical_state.lambda_sterics == 1.0

alchemical_state.set_function_variable('lambda', 0.25)
assert alchemical_state.lambda_electrostatics == 0.0
assert alchemical_state.lambda_sterics == 0.5

# Set the alchemical state of the simulated system.
alchemical_state.apply_to_context(context)
```

In the example above, `step_hm` is the Heaviside step function with half-maximum convention (i.e. `step_hm(0.0) == 0.5`), while `step(0.0) == 0.0`. All the functions in the Python standard module `math` can be specified in the string.

2.3 Manipulating the thermodynamic state of your simulation

The classes in the *states* module provide a framework to decouple the degrees of freedom (or parameters) of the simulated thermodynamic state from their implementation details in OpenMM.

2.3.1 Defining temperature and pressure

The fundamental class in the `states` module is `ThermodynamicState`. This class hold a `System` object and controls the ensemble parameters of temperature and pressure. For example, the code below creates a water box in NVT ensemble at 298 K.

```
>>> from openmmtools import states

>>> waterbox = testsystems.WaterBox(box_edge=2*unit.nanometers)
>>> thermo_state = states.ThermodynamicState(system=waterbox.system,
...                                          temperature=298.0*unit.kelvin)
>>> thermo_state.volume.format('%.1f')
'8.0 nm**3'
>>> assert thermo_state.pressure is None
```

The volume is computed from the box vectors associated to the `System` object. To convert the system to an NPT state at 298 K and 1 atm pressure, you can set the `pressure` attribute.

```
thermo_state.pressure = 1.0*unit.atmosphere
assert thermo_state.volume is None
```

Note that the operation of specifying a constant pressure result in a null volume, as the volume will fluctuate during the simulation. You can then create an OpenMM `Context` object that is guaranteed to be in the specified thermodynamic state.

```
>>> integrator = integrators.LangevinIntegrator(temperature=298.0*unit.kelvin)
>>> context = thermo_state.create_context(integrator)
```

(continues on next page)

(continued from previous page)

```
>>> context.setPositions(waterbox.positions)
>>> integrator.step(n_steps)

>>> # ThermodynamicState takes care of adding and configuring a␣
↪MonteCarloBarostatForce
>>> # to keep the pressure at 1atm.
>>> force_index, barostat = forces.find_forces(context.getSystem(),
...                                            openmm.MonteCarloBarostat,
...                                            only_one=True)
>>> barostat.getDefaultTemperature().format('%.1f')
'298.0 K'
>>> print(barostat.getDefaultPressure())
1.01325 bar
```

2.3.2 Consistency checks for free

Using the `ThermodynamicState` class means to take advantage of several consistency checks that can avoid bugs in your application that can be very hard to detect in the first place and then to track down (we speak from personal experience).

For example, trying to create a `Context` using Langevin integrator set to the incorrect temperature or trying to add a barostat to a system in vacuum raises an error.

```
>>> thermo_state.temperature = 298.0*unit.kelvin
>>> integrator = integrators.LangevinIntegrator(temperature=310.0*unit.kelvin)
>>> thermo_state.create_context(integrator)
Traceback (most recent call last):
...
ThermodynamicsError: Integrator is coupled to a heat bath at a different temperature.
```

```
>>> vacuum_system = testsystems.TolueneVacuum().system
>>> thermo_state = states.ThermodynamicState(system=vacuum_system,
...                                          temperature=298.15*unit.kelvin,
...                                          pressure=1.0*unit.atmosphere)
Traceback (most recent call last):
...
ThermodynamicsError: Non-periodic systems cannot have a barostat.
```

While, if you create a `Context` with an integrator that is not coupled to a heat bath, `ThermodynamicState` will take care of adding an `AndersenThermostat`.

```
>>> # Use a non-thermostated integrator.
>>> thermo_state_nvt = states.ThermodynamicState(system=vacuum_system,
...                                              temperature=298.15*unit.kelvin)
>>> integrator = openmm.VerletIntegrator(2.0*unit.femtoseconds)
>>> context_nvt = thermo_state_nvt.create_context(integrator)
>>> len(forces.find_forces(context_nvt.getSystem(), openmm.AndersenThermostat))
1
```

2.3.3 Manipulating the thermodynamic state: Compatible thermodynamic states

Once a `Context` has been created, is is possible to change the simulation thermodynamic state through the method `ThermodynamicState.apply_to_context()`. The method will mask the implementation details and take

care of modifying all the OpenMM forces and integrators that depend on the temperature and pressure parameters. In this sense, the `ThermodynamicState` class decouples the representation of the thermodynamic parameters from their implementation details.

```
>>> # Modify temperature and pressure of a system employing a Langevin
>>> # thermostat and a Monte Carlo barostat.
>>> thermo_state.temperature = 400.0*unit.kelvin
>>> thermo_state.pressure = 1.2*unit.bar
>>> thermo_state.apply_to_context(context)
>>> context.getIntegrator().getTemperature().format('%.1f')
'400.0 K'
>>> context.getParameter(openmm.MonteCarloBarostat.Pressure())
1.2
>>> # The MonteCarloBarostat requires also a temperature parameter for the acceptance␣
↪probability.
>>> context.getParameter(openmm.MonteCarloBarostat.Temperature())
400.0
```

```
>>> # Modify the temperature of a system using an Andersen thermostat.
>>> thermo_state_nvt.temperature = 400.0*unit.kelvin
>>> thermo_state_nvt.apply_to_context(context_nvt)
>>> context_nvt.getParameter(openmm.AndersenThermostat.Temperature())
400.0
```

A `ThermodynamicState` can be applied to any `Context` that was created from a **compatible thermodynamic state**.

Important: Two `ThermodynamicState` objects `x, y` are compatible if a `context` created by `x` can be modified to be in the `y` thermodynamic state through `y.apply_to_context(context)` and viceversa.

This is not always possible in OpenMM because of some implementation details related to optimizations. In short, two `ThermodynamicState``s are compatible if they have the same ``System` and they are in the same ensemble (i.e. NVT and NPT thermodynamic states are incompatible).

```
>>> alanine = testsystems.AlanineDipeptideExplicit()
>>> state1 = states.ThermodynamicState(alanine.system, 273*unit.kelvin)
>>> state2 = states.ThermodynamicState(alanine.system, 310*unit.kelvin)
>>> state1.is_state_compatible(state2)
True

# Switch state1 from NVT to NPT ensemble.
>>> state1.pressure = 1.0*unit.atmosphere
>>> state1.is_state_compatible(state2)
False
```

Luckily, the class *openmmtools.cache.ContextCache* takes care of checking for compatibility and decide whether it's possible to modifying a previously created `Context` object or if it is necessary to create a separate one.

2.3.4 Using the ContextCache

Important: Using `ContextCache` is the recommended way of creating `Context` objects within the OpenMMTools framework.

The `openmmtools.cache.ContextCache` class has the role of maintaining the *minimum number of compatible Contexts allocated on the GPU*, allowing virtually an infinite number of thermodynamic states to be simulated on finite-memory hardware, and minimizing the number of expensive `Context` creation/destruction.

To obtain a `Context` simply use the `ContextCache.get_context()` method.

```
from openmmtools import cache

alanine = testsystems.AlanineDipeptideExplicit()
thermo_state = states.ThermodynamicState(alanine.system, 310*unit.kelvin)
integrator = integrators.LangevinIntegrator(temperature=310*unit.kelvin)

context_cache = cache.ContextCache()
context, context_integrator = context_cache.get_context(thermo_state,
                                                        integrator)
context.setPositions(alanine.positions)
context_integrator.step(n_steps)
```

Note that `get_context()` returns also an `Integrator` that may be a different instance of the `integrator` passed as a parameter. This is because an OpenMM `Context` can be associated with a single integrator instance, thus reusing a previously instantiated `Context` requires using the previously instantiated integrator as well. Nevertheless, `context_integrator` is guaranteed to be identical to `integrator`.

Requesting a context in a compatible `ThermodynamicState` returns the same `Context` object correctly configured to simulate the requested thermodynamic state.

```
>>> compatible_state = states.ThermodynamicState(alanine.system, 400*unit.kelvin)
>>> compatible_integrator = integrators.LangevinIntegrator(temperature=400*unit.
↪kelvin)
>>> compatible_context, compatible_integrator = context_cache.get_context(compatible_
↪state,
...                                                                    compatible_
↪integrator)
>>> id(context) == id(compatible_context)
True
>>> len(context_cache)  # The number of Contexts maintained in memory.
1
>>> compatible_integrator.getTemperature().format('%.1f')
'400.0 K'
```

Requesting a context in a different ensemble causes the creation of another `Context`.

```
>>> thermo_state_npt = copy.deepcopy(thermo_state)
>>> thermo_state_npt.pressure = 1.0*unit.atmosphere
>>> integrator = integrators.LangevinIntegrator(temperature=thermo_state_npt.
↪temperature)
>>> context_npt, integrator_npt = context_cache.get_context(thermo_state_npt,␣
↪integrator)
>>> id(context) == id(context_npt)
False
>>> len(context_cache)
2
```

You can set a capacity and a time to live for contexts. The time to live is currently measured in number of accesses to the `ContextCache`.

```
>>> context_cache = cache.ContextCache(capacity=1, time_to_live=5)
>>> integrator = openmm.VerletIntegrator(1.0*unit.femtosecond)
```

(continues on next page)

(continued from previous page)

```
>>> context1, integrator1 = context_cache.get_context(thermo_state,
...                                                   copy.deepcopy(integrator))
>>> context2, integrator2 = context_cache.get_context(thermo_state_npt,
...                                                   copy.deepcopy(integrator))
>>> len(context_cache)
1
```

In the example above, the maximum capacity of the cache is 1, so the first context is deallocated to make space for the second `Context` created with the incompatible thermodynamic state.

Finally, you can force the `ContextCache` to create contexts on a specific platform.

```
platform = openmm.Platform.getPlatformByName('Reference')
context_cache = cache.ContextCache(platform=platform)
```

2.3.5 The global ContextCache

The *openmmtools.cache* module exposes a global variable that provides a shared `ContextCache` for all the classes in the framework.

```
cache.global_context_cache.platform = openmm.Platform.getPlatformByName('CPU')
cache.global_context_cache.capacity = 2
cache.global_context_cache.time_to_live = 10
verlet_integrator = openmm.VerletIntegrator(1.0*unit.femtosecond)
context, integrator = cache.global_context_cache.get_context(thermo_state,
                                                             verlet_integrator)
```

Usually, you'll want to create a `Context` using the `global_context_cache` to minimize the number of created contexts overall. This is, for example, the context cache used by default by all the `MCMCMove` objects internally, which we'll touch shortly.

2.3.6 Extending ThermodynamicState to control arbitrary parameters

It is possible to extend the `ThermodynamicState` to manipulate other thermodynamic parameters of the `System` through the `states.CompoundThermodynamicState` class and one or more *composable states*. An example may clarify this. Remember the `alchemy.AlchemicalState` class we discussed above? `AlchemicalState` is a composable state.

```
# Prepare T4-Lysozyme + p-xylene system for alchemical perturbation.
factory = alchemy.AbsoluteAlchemicalFactory()
alchemical_region = alchemy.AlchemicalRegion(alchemical_atoms=pxylene_atoms)
alchemical_system = factory.create_alchemical_system(t4_system, alchemical_region)

# Define the basic thermodynamic state of the system.
thermo_state = states.ThermodynamicState(alchemical_system, temperature=298*unit.
↪kelvin)

# Extend the definition of thermodynamic state to consider alchemical parameters as␣
↪well.
alchemical_state = alchemy.AlchemicalState.from_system(alchemical_system)
compound_state = states.CompoundThermodynamicState(thermodynamic_state=thermo_state,
                                                   composable_states=[alchemical_
↪state])
```

At this point, `compound_state` is *both* a `ThermodynamicState` and an `AlchemicalState` in the sense that it exposes the interface to modify the thermodynamic parameters controlled by both objects.

```
context = compound_state.create_context(integrators.LangevinIntegrator())
compound_state.temperature = 350*unit.kelvin  # Increase temperature of simulation.
compound_state.lambda_sterics = 0.2  # Soften torsions.
compound_state.apply_to_context(context)
```

Obviously, `CompoundThermodynamicState` is not compatible exclusively with `AlchemicalState` but with any object implementing the `states.IComposableState` interface. A quick way to define your own composable state is described in the *developer's tutorial*.

The power of this abstraction will become evident when we'll implement a simple replica-exchange algorithm at the end of the tutorial.

2.4 MCMC framework

The Markov chain Monte Carlo (MCMC) framework implemented in the *mcmc* module take advantage of the thermodynamic state objects described above to provide an easy way to experiment with different propagation schemes mixing Monte Carlo moves and dynamics.

2.4.1 Basic usage

The basic object in the module is the `mcmc.MCMCMove` abstract class that provides a common interface for both integrators and Monte Carlo to propagate the state of the system.

```
from openmmtools import mcmc

# Define the thermodynamic state of the T4-Lysozyme + p-xylene system
thermo_state = states.ThermodynamicState(t4_system, temperature=300*unit.kelvin)

# Create a SamplerState system holding the coordinates of the system.
sampler_state = states.SamplerState(positions=lysozyme_pxylene.positions)

# Propagate the system with a GHMC integrator.
ghmc_move = mcmc.GHMCMove(timestep=1.0*unit.femtosecond, n_steps=n_steps)
ghmc_move.apply(thermo_state, sampler_state)
```

The `SamplerState` object in the snippet above holds the configurational degrees of freedom of the `System` (e.g., positions, velocities, and eventually box vectors). The sampler state is updated by `MCMCMove.apply` to hold the coordinates and velocities after 1000 steps of GHMC integration. Note however that, in princple, the framework allows an `MCMCMove` to change also the thermodynamic degrees of freedom in `thermo_state`.

2.4.2 OpenMM integrators as MCMCMoves

The *mcmc* module provides a few integrators in the form of an `MCMCMove`, including `openmmtools.integrators.LangevinIntegrator`. Casting integrators in the form of an `MCMCMove` object makes it easy to combine them with Monte Carlo techniques. Moreover, integrator ‘‘MCMCMove‘‘s provide a few extra features such as automatic recovery after a NaN.

```
langevin_move = mcmc.LangevinSplittingDynamicsMove(splitting='V R O R V',
                                                   n_steps=n_steps,
                                                   n_restart_attempts=5)
langevin_move.apply(thermo_state, sampler_state)
```

Propagating your system through Langevin dynamics has always a non-zero probability of incurring into a NaN error. When this happens, instead of crashing, the Langevin move above will restore the state of the `System` before integrating and try again, relying on the stochastic component of the propagation to obtain a different solution. This is repeated to a maximum of 5 times before giving up and throwing an error. The raised exception exposes a method to serialize the simulation objects automatically for further debugging.

```
try:
    langevin_move.apply(thermo_state, sampler_state)
except mcmc.IntegratorMoveError as e:
    # This saves to disk the OpenMM System, Integrator, and State objects.
    e.serialize_error(path_files_prefix='debug/langevin')
```

When a NaN occurr, the code above serializes the OpenMM `System`, `Integrator`, and `State` objects on disk at `debug/langevin-system.xml`, `debug/langevin-integrator.xml`, and `debug/langevin-state.xml` respectively.

This feature can easily be extended to other integrators that are not explicitly provided in the *mcmc* module.

```
integrator = integrators.HMCIntegrator(timestep=1.0*unit.femtosecond)
HMC_move = mcmc.IntegratorMove(integrator, n_steps=n_steps, n_restart_attempts=4)
```

2.4.3 Combining Monte Carlo and dynamics

Combining and mixing multiple `MCMCMove` is usually performed through the `mcmc.SequenceMove` object

```
sequence_move = mcmc.SequenceMove(move_list=[
    mcmc.MCDisplacementMove(atom_subset=pxylene_atoms),
    mcmc.MCRotationMove(atom_subset=pxylene_atoms),
    mcmc.LangevinSplittingDynamicsMove(timestep=2.0*unit.femtoseconds, n_steps=n_
↪steps,
                                       reassign_velocities=True, n_restart_attempts=6)
])

sequence_move.apply(thermo_state, sampler_state)
```

The `MCMCMove` above performs in sequence a Metropolized Monte Carlo rigid translation and rotation of the p-xylene molecule followed by Langevin dynamics after randomizing the velocities according to the Boltzmann distribution at the temperature of `thermo_state`.

2.4.4 ContextCache and Platform with MCMCMoves

All `MCMCMove` objects implemented in OpenMMTools accept a `context_cache` in the constructor. This parameter defaults to `mmtools.cache.global_context_cache`, but you can pass a local cache to trigger other behaviors.

```
local_cache = cache.ContextCache(platform=openmm.Platform.getPlatformByName('CPU'))
dummy_cache = cache.DummyContextCache()  # Disable caching.
move = mcmc.SequenceMove(move_list=[
    mcmc.MCDisplacementMove(atom_subset=pxylene_atoms, context_cache=local_cache),
```

(continues on next page)

(continued from previous page)

```
    mcmc.MCRotationMove(atom_subset=pxylene_atoms, context_cache=dummy_cache),
    mcmc.LangevinSplittingDynamicsMove(n_steps=n_steps)  # Uses global_context_cache.
])
```

In the example above, applying the `move` will perform an MC translation of the ligands atom using a local `ContextCache` that runs on the CPU, then an MC rotation using the `DummyContextCache`, which recreates context every time effectively deactivating caching, and finally propagates the system with Langevin dynamics using the global cache on the fastest platform available.

2.5 Example: A minimal implementation of a general replica-exchange simulation class

Our most recent enhanced-sampling facilities are currently hosted in YANK, and they are still waiting to be moved to OpenMMTools. However, the following minimal implementation of a replica exchange simulation class should give you an idea of what is possible to do when taking advantage of the full framework.

```
import math
from random import random, randint

class ReplicaExchange:

    def __init__(self, thermodynamic_states, sampler_states, mcmc_move):
        self._thermodynamic_states = thermodynamic_states
        self._replicas_sampler_states = sampler_states
        self._mcmc_move = mcmc_move

    def run(self, n_iterations=1):
        for iteration in range(n_iterations):
            self._mix_replicas()
            self._propagate_replicas()

    def _propagate_replicas(self):
        # _thermodynamic_state[i] is associated to the replica configuration in _
↪replicas_sampler_states[i].
        for thermo_state, sampler_state in zip(self._thermodynamic_states, self._
↪replicas_sampler_states):
            self._mcmc_move.apply(thermo_state, sampler_state)

    def _mix_replicas(self, n_attempts=1):
        # Attempt to switch two replicas at random. Obviously, this scheme can be␣
↪improved.
        for attempt in range(n_attempts):
            # Select two replicas at random.
            i = randint(0, len(self._thermodynamic_states)-1)
            j = randint(0, len(self._thermodynamic_states)-1)
            sampler_state_i, sampler_state_j = (self._replicas_sampler_states[k] for␣
↪k in [i, j])
            thermo_state_i, thermo_state_j = (self._thermodynamic_states[k] for k in␣
↪[i, j])
```

(continues on next page)

(continued from previous page)

```
            # Compute the energies.
            energy_ii = self._compute_reduced_potential(sampler_state_i, thermo_state_
↪i)
            energy_jj = self._compute_reduced_potential(sampler_state_j, thermo_state_
↪j)
            energy_ij = self._compute_reduced_potential(sampler_state_i, thermo_state_
↪j)
            energy_ji = self._compute_reduced_potential(sampler_state_j, thermo_state_
↪i)

            # Accept or reject the swap.
            log_p_accept = - (energy_ij + energy_ji) + energy_ii + energy_jj
            if log_p_accept >= 0.0 or random() < math.exp(log_p_accept):
                # Swap states in replica slots i and j.
                self._thermodynamic_states[i] = thermo_state_j
                self._thermodynamic_states[j] = thermo_state_i

    def _compute_reduced_potential(self, sampler_state, thermo_state):
        # Obtain a Context to compute the energy with OpenMM. Any integrator will do.
        context, integrator = cache.global_context_cache.get_context(thermo_state)
        # Compute the reduced potential of the sampler_state configuration
        # in the given thermodynamic state.
        sampler_state.apply_to_context(context)
        return thermo_state.reduced_potential(context)
```

The first observation is that the bulk of the code complexity lies in the replica swapping code, while most of the other details are handled by the specialized classes of the framework. From a software engineering perspective, this is a good sign as it is compatible with the single responsibility principle.

Secondly, the class can be used to implement a variety of algorithm. A few examples follow.

2.5.1 Parallel tempering

To run a parallel tempering simulation, we just have initialize the `ReplicaExchange` object with a list of thermodynamic states that vary in temperature. You can make use of the utility function `states.create_thermodynamic_state_protocol` to initialize efficiently a list of `ThermodynamicState` or `CompoundThermodynamicState`.

```
>>> # Initialize thermodynamic states at different temperatures.
>>> host_guest = testsystems.HostGuestVacuum()
>>> protocol = {'temperature': [300, 310, 330, 370, 450] * unit.kelvin}
>>> thermo_states = states.create_thermodynamic_state_protocol(host_guest.system,␣
↪protocol)

>>> # Initialize replica initial configurations.
>>> sampler_states = [states.SamplerState(positions=host_guest.positions)
...                   for _ in thermo_states]

>>> # Propagate the replicas with Langevin dynamics.
>>> langevin_move = mcmc.LangevinSplittingDynamicsMove(timestep=2.0*unit.femtosecond,
...                                                    n_steps=n_steps)

>>> # Run the parallel tempering simulation.
>>> parallel_tempering = ReplicaExchange(thermo_states, sampler_states, langevin_move)
>>> parallel_tempering.run()
```

This example creates 5 replicas starting from the same configurations but at the temperatures of 300, 310, . . . , 450 K, and propagates the system with Langevin dynamics.

2.5.2 Hamiltonian replica exchange + parallel tempering

Let's say we want to implement an enhanced sampling scheme that increases the temperature while alchemically softening part of a system.

```
>>> # Prepare the T4 Lysozyme + p-xylene system for alchemical modification.
>>> guest_atoms = host_guest.mdtraj_topology.select('resname B2')
>>> alchemical_region = alchemy.AlchemicalRegion(alchemical_atoms=guest_atoms)
>>> factory = alchemy.AbsoluteAlchemicalFactory()
>>> alchemical_system = factory.create_alchemical_system(host_guest.system,␣
↪alchemical_region)

>>> # Initialize compound thermodynamic states at different temperatures and␣
↪alchemical states.
>>> protocol = {'temperature': [300, 310, 330, 370, 450] * unit.kelvin,
...             'lambda_electrostatics': [1.0, 0.5, 0.0, 0.0, 0.0],
...             'lambda_sterics': [1.0, 1.0, 1.0, 0.5, 0.0]}
>>> alchemical_state = alchemy.AlchemicalState.from_system(alchemical_system)
>>> compound_states = states.create_thermodynamic_state_protocol(
...     alchemical_system, protocol=protocol, composable_states=[alchemical_state])

>>> # Run the combined Hamiltonian replica exchange + parallel tempering simulation.
>>> hrex_tempering = ReplicaExchange(compound_states, sampler_states, langevin_move)
```

2.5.3 Hamiltonian replica exchange + parallel tempering mixing Monte Carlo and dynamics

Finally, let's mix Monte Carlo and dynamics for propagation.

```
>>> sequence_move = mcmc.SequenceMove(move_list=[
...     mcmc.MCDisplacementMove(atom_subset=pxylene_atoms),
...     mcmc.MCRotationMove(atom_subset=pxylene_atoms),
...     mcmc.LangevinSplittingDynamicsMove(timestep=2.0*unit.femtoseconds, n_steps=n_
↪steps,
...                                        reassign_velocities=True, n_restart_
↪attempts=6)
... ])

>>> # Run the combined Hamiltonian replica exchange + parallel tempering simulation
>>> # using a combination of Monte Carlo moves and Langevin dynamics.
>>> hrex_tempering = ReplicaExchange(compound_states, sampler_states, sequence_move)
```

CHAPTER 3

Advanced features and developer's tutorial

This tutorial describes more advanced features that can be useful for people developing their software using or extending OpenMMTools.

Contents

3.1 Using and implementing integrator and force objects

3.1.1 Copy and serialization utilities

The *integrators* and *forces* in OpenMMTools are usually implemented by extending custom classes in OpenMM. For example, the declaration of our `LangevinIntegrator` and the `HarmonicRestraintForce` goes boils down to

```
class LangevinIntegrator(mmtools.utils.RestorableOpenMMObject, openmm.
↪CustomIntegrator):
    pass

class HarmonicRestraintForce(mmtools.utils.RestorableOpenMMObject, openmm.
↪CustomCentroidBondForce):
    pass
```

The purpose of inheriting from *openmmtools.utils.RestorableOpenMMObject* class has to do with copies and serialization. Without `RestorableOpenMMObject`, these objects can still be copied and go through the standard serialization and deserialization in OpenMM without errors

```
from simtk import openmm, unit

class VelocityVerlet(openmm.CustomIntegrator):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.addComputePerDof("x", "x+dt*v")
        self.addComputePerDof("v", "v+0.5*dt*f/m")

    def my_method(self):
        return 0.0

integrator = VelocityVerlet(1*unit.femtosecond)
copied_integrator = copy.deepcopy(integrator)
integrator_serialization = openmm.XmlSerializer.serialize(integrator)
deserialized_integrator = openmm.XmlSerializer.deserialize(integrator_serialization)
```

However, copies and serializations in OpenMM are performed at the C++ level, and thus they don't keep track of the Python class and methods.

```
>>> print(type(copied_integrator))
<class 'simtk.openmm.openmm.CustomIntegrator'>

>>> deserialized_integrator.my_method()
Traceback (most recent call last):
...
AttributeError: type object 'object' has no attribute '__getattr__'
```

Inheriting from *openmmtools.utils.RestorableOpenMMObject*, allows you to easily recover the original interface after copying or deserializing. This happens automatically for copies, but you'll have to use `RestorableOpenMMObject.restore_interface()` after deserialization.

```
from openmmtools import utils

class VelocityVerlet(utils.RestorableOpenMMObject, openmm.CustomIntegrator):

```

(continues on next page)

(continued from previous page)

```
    def __init(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.addComputePerDof("x", "x+dt*v")
        self.addComputePerDof("v", "v+0.5*dt*f/m")

    def my_method(self):
        return 0.0

integrator = VelocityVerlet(1*unit.femtosecond)
```

```
>>> copied_integrator = copy.deepcopy(integrator)
>>> isinstance(copied_integrator, VelocityVerlet)
True
```

```
>>> integrator_serialization = openmm.XmlSerializer.serialize(integrator)
>>> deserialized_integrator = openmm.XmlSerializer.deserialize(integrator_
↪serialization)
>>> utils.RestorableOpenMMObject.restore_interface(deserialized_integrator)
True
>>> deserialized_integrator.my_method()
0.0
```

For forces, the function *openmmtools.forces.find_forces(system)* automatically calls `RestorableOpenMMObject.restore_interface()` on all `system` forces so there's usually no need to perform that call after deserialization.

3.1.2 Integrators coupled to a heat bath

If you implement an integrator coupled to a heat bath, you have to expose *getTemperature* and *setTemperature* methods or `ThermodynamicState` won't have any way to recognize it, and it will add an `AndersenThermostat` force when initializing the OpenMM `Context` object.

The base class *openmmtools.integrators.ThermostatedIntegrator* is a convenience class implemented for this purpose. Inheriting from `ThermostatedIntegrator` will implicitly add the `RestorableOpenMMObject` functionalities as well.

```
>>> from openmmtools import integrators

>>> class MyIntegrator(integrators.ThermostatedIntegrator):
...     def __init__(self, temperature=298.0*unit.kelvin, timestep=1.0*unit.
↪femtoseconds):
...         super().__init__(temperature, timestep)
...
>>> integrator = MyIntegrator(temperature=350*unit.kelvin)
>>> integrator.getTemperature()
Quantity(value=350.0, unit=kelvin)
>>> integrator.setTemperature(380.0*unit.kelvin)
```

3.1.3 Using standard Python attribute in custom integrators and forces

You should avoid having pure Python attributes when inheriting from custom OpenMM integrators and forces and instead favor using properties that read that attribute from the underlying OpenMM object as, for example, a global variable.

For example, an integrator exposing the temperature should **not** hold a simple `temperature` Python attribute internally such as

```
class INCORRECTIntegrator(openmm.CustomIntegrator):

    def __init__(self, *args, temperature=298.15*unit.kelvin, **kwargs):
        super().__init__(*args, **kwargs)
        self.temperature = temperature
```

but it expose it as a getter or a property similarly to the follow.

```
class CorrectIntegrator(openmm.CustomIntegrator):

    def __init__(self, *args, temperature=298.15*unit.kelvin, **kwargs):
        super().__init__(*args, **kwargs)
        self.addGlobalVariable('temperature', temperature)

    @property
    def temperature(self):
        return self.getGlobalVariableByName('temperature') * unit.kelvin
```

This is because:

1. If the parameter doesn't affect serialization `ContextCache` won't be able to distinguish between two integrators that differ by that parameter, and it may return an incorrect integrator.
2. Python attribute cannot be restored by `RestorableOpenMMObject` since there's no information about them in the XML string, and thus they will be lost with serialization.

3.2 Handling (compound) thermodynamic states

In the examples that follow, we'll use a simple `ThermodynamicState`, but everything applies to `CompoundThermodynamicState` as well as `CompoundThermodynamicState` is a subclass of `ThermodynamicState`.

3.2.1 Modifying a System object buried in a ThermodynamicState

Setting a thermodynamic parameter in `ThermodynamicState` is practically instantaneous, but modifying anything else involves the copy of the internal `System` object so it can be very slow.

```
from openmmtools import states
from openmmtools import testsystems

system = testsystems.TolueneVacuum().system
thermo_state = states.ThermodynamicState(system, temperature=300*unit.kelvin)

# This is very fast.
thermo_state.temperature = 400.0*unit.kelvin

system = thermo_state.system  # This is a copy! Changes to this System won't affect␣
↪thermo_state.
```

(continues on next page)

(continued from previous page)

```
# Make your changes to system.
thermo_state.system = system  # This involves another System copy.
```

The copies are there to ensure the consistency of `ThermodynamicState` internal state. If you need to consistently modifying part of the systems during the simulation consider implementing a composable state that handle those degrees of freedom (see section *Implementing a new ComposableState*).

Another thing to keep in mind is that by default the property `ThermodynamicState.system` will return a `System` containing an `AndersenThermostat` force. If you only use `ThermodynaicState.create_context()` or the `ContextCache` class to create OpenMM `Context` objects, this shouldn't cause issues, but if for any reason you don't want that thermostat you can use the getter instead of the property.

```
system = thermo_state.get_system(remove_thermostat=True)
```

3.2.2 Implementation details of compatibility checks

Internally, `ThermodynamicState` associates a unique hash to a `System` in a particular ensemble, and it compares this hash to check for compatibility. The function that performs this task looks like this:

```
@classmethod
def _standardize_and_hash(cls, system):
    """Standardize the system and return its hash."""
    cls._standardize_system(system)
    system_serialization = openmm.XmlSerializer.serialize(system)
    return system_serialization.__hash__()
```

The `_standardize_system()` functions sets the thermodynamic parameters controlled by the `ThermodynamicState` to a standard value so that `System` that differ by only those parameters will have identical XML serialized strings, and thus identical hashes.

The section *Implementing a new ComposableState* has information on how the composable states expand the concept of compatibility to thermodynamic parameters other than temperature and pressure.

Note: As a consequence of how the compatibility hash is computed, two ThermodynamicStates to be compatible must have Systems with the same particles and forces in the same order, or the XML serialization will be different.

3.2.3 Copying and initialization of multiple thermodynamic states

Because of some memory optimizations, copying a `ThermodynamicState` or a `CompoundThermodynamicState` does not copy the internal `System` so it is practically instantaneous. On the other hand, initializing a new `ThermodynamicState` or a `CompoundThermodynamicState` object does involve a `System` copy.

```
thermo_state1 = states.ThermodynamicState(system, temperature=300*unit.kelvin)

# Very fast.
thermo_state2 = copy.deepcopy(thermo_state)
thermo_state2.temperature = 350*unit.kelvin

# Slow.
thermo_state2 = states.ThermodynamicState(system, temperature=350*unit.kelvin)
```

The function *openmmtools.states.create_thermodynamic_state_protocol* takes advantage of this to make it easy to instantiate a list of `ThermodynamicState` or `CompoundThermodynamicState` objects that differ only by the controlled parameters.

3.3 Implementing a new ComposableState

3.3.1 The IComposableInterface

Composable states allow to control thermodynamic parameters of the simulation while masking their implementation details. There are no restrictions on the implementation details, but the class must implement the *openmmtools.states.IComposableState* interface. You can see the API docs for contract details, but here is a list of the methods.

```
class IComposableState:

    def apply_to_system(self, system):
        """Modify an OpenMM System to be in this thermodynamic state."""

    def check_system_consistency(self, system):
        """Raise AlchemicalStateError if system has different parameters."""

    def apply_to_context(self, context):
        """Modify an OpenMM Context to be in this thermodynamic state."""

    def _standardize_system(cls, system):
        """Modify the System to be in the standard thermodynamic state."""

    def _on_setattr(self, standard_system, attribute_name, old_attribute_value):
        """Callback that checks if standard system needs to be updated after a state␣
↪attribute is set."""

    def _find_force_groups_to_update(self, context, current_context_state, memo)
        """Find the force groups whose energy must be recomputed after apply_to_
↪context."""
        # Optional. This is used only for optimizations.
```

The `_standardize_system` method effectively determines which other states will be compatible (see also section *Implementation details of compatibility checks*). The purpose of `_standardize_system` is to set the parameters of the `System` that can be manipulated in the `Context` to the same value so that their XML serialization string and their hash will be identical. Systems that after standardization are identical are assigned to the same `Context` by `ContextCache.get_context()`.

Relatedly, the callback `_on_setattr()` is called by `CompoundThermodynamicState` after a thermodynamic parameter has been set. The method must return `True` if the change in the thermodynamic parameter has caused the standard system to have a different hash. For example, in the basic `ThermodynamicState` class this happens when the `pressure` parameter goes from `None` to any valid value because states in NVT and NPT are not compatible.

The method `_find_force_groups_to_update` is optional and related to the optimization described in *Computing the reduced potential of one configuration at multiple thermodynamic states*.

3.3.2 ComposableStates that control a Force global parameter

Often, a thermodynamic parameter can be implemented with OpenMM as a global parameter added to a custom force. For example, to alchemically soften torsions, `alchemy.AbsoluteAlchemicalFactory` substitute some of the torsion potential terms using a `openmm.CustomTorsionForce` whose energy is multiplied by a global parameter called `lambda_torsions`.

```
energy_function = "lambda_torsions * k*(1+cos(periodicity*theta-phase))"
custom_force = openmm.CustomTorsionForce(energy_function)
custom_force.addGlobalParameter('lambda_torsions', 1.0)
# Other force configurations.
system.addForce(custom_force)
```

When this is the case, the base class `openmmtools.states.GlobalParameterState` can be used to create a composable state very quickly.

```
from openmmtools.states import GlobalParameterState

class MyComposableState(GlobalParameterState):

    lambda_torsions = GlobalParameterState.GlobalParameter('lambda_torsions',␣
↪standard_value=1.0)
```

It is possible to perform checks on the assigned value by adding a validator.

```
class MyComposableState(GlobalParameterState):

    lambda_torsions = GlobalParameterState.GlobalParameter('lambda_torsions',␣
↪standard_value=1.0)

    @lambda_torsions.validator
        def lambda_torsions(self, instance, new_value):
            if new_value is not None and not (0.0 <= new_value <= 1.0):
                raise ValueError('lambda_torsions must be between 0.0 and 1.0')
            return new_value
```

The example above allows only values between 0.0 and 1.0 for `lambda_torsions`.

3.3.3 Computing the reduced potential of one configuration at multiple thermodynamic states

When computing the potential energy of a single configuration at multiple thermodynamic states, it is often unnecessary to compute the whole Hamiltonian multiple times but just the terms of the Hamiltonian that change from one state to another. OpenMM makes this possible to compute only the energy of a subset of forces through the force groups mechanism.

```
force = openmm.CustomBondForce('(K/2)*(r-r0)^2;')
force.setForceGroup(5)
```

The utility function `openmmtools.states.reduced_potential_at_states()` takes advantage of forces separated in different groups to efficiently compute the reduced potentials at the thermodynamic states.

```
from openmmtools import alchemy
from openmmtools import cache

```

(continues on next page)

(continued from previous page)

```
alanine = testsystems.AlchemicalAlanineDipeptide()
protocol = {'lambda_sterics': [1.0, 0.5, 0.0],
            'lambda_electrostatics': [1.0, 0.5, 0.0]}
constants = {'temperature': 300*unit.kelvin}
composable_states = [alchemy.AlchemicalState.from_system(alanine.system)]
compound_states = states.create_thermodynamic_state_protocol(alanine.system, protocol,
                                                             constants, composable_
↪states)

sampler_state = states.SamplerState(positions=alanine.positions)
reduced_potentials = states.reduced_potential_at_states(sampler_state, compound_
↪states,
                                                        cache.global_context_cache)
```

In order for the optimization to take effect, the composable states must implement the method `_find_force_groups_to_update(self, context, current_context_state, memo)`. This method inspects the `System` associated to the `context` and return the force groups that will have an updated energy after the state will be changed from `current_context_state` to `self`. The `memo` dictionary can be use to store the force groups to inspect in subsequent calls of the method within a `reduced_potential_at_states` execution so that the `System` must be parsed only the first time.

3.4 Implementing a new MCMCMove

3.4.1 The MCMCMove interface

An `MCMCMove` requires exclusively the implementation of an `apply` method with the following signature (see the *API documentation* for more details.

```
class MCMCMove(SubhookedABCMeta):

    def apply(self, thermodynamic_state, sampler_state):
        pass
```

Anything can happen inside `apply` as long as `thermodynamic_state` and `sampler_state` are updated correctly. It is usually a good idea to include in the constructor a `context_cache` argument to let the user specify how the `Context` should be created and on which platform.

3.4.2 OpenMM integrators that modify the thermodnamic state

Custom OpenMM integrators can modify global variables that effectively change the thermodynamic state of the `Context`.

Important: Remember to update the `thermodynamic_state` object correctly at the end of `apply` if the integrator changes the thermodynamic state of the simulation.

When this is the case, it's not possible to cast your integrator into an `MCMCMove` with `IntegratorMove`. Nevertheless, it's still possible to take advantage of the extra features already offered by `IntegratorMove` by subclassing

the *openmmtools.mcmc.BaseIntegratorMove <mcmc>* class. `IntegratorMove` inherits from this base class. An implementation would look more or less like this (see the API documentation for the details).

```
class MyMove(BaseIntegratorMove):
    def __init__(self, timestep, n_steps, **kwargs):
        super(MyMove, self).__init__(n_steps, **kwargs)
        self.timestep = timestep

    def _get_integrator(self, thermodynamic_state):
        return MyIntegrator(self.timestep, thermodynamic_state.temperature)

    def _before_integration(self, context, thermodynamic_state):
        # Optional: Any operation performed after the context
        # was created but before integration.

    def _after_integration(self, context, thermodynamic_state):
        # Update thermodynamic_state from context parameters.
        # Optional: Read statistics from context.getIntegrator() parameters.
```

3.4.3 Metropolized MCMCMoves

The *mcmc* module contains a base class for Metropolized moves as well. The following class implement an example that simply adds the unit vector to the initial coordinates.

```
from openmmtools import mcmc

class AddOneVector(mcmc.MetropolizedMove):
    def _propose_positions(self, initial_positions):
        print('Propose new positions')
        displacement = numpy.array([1.0, 1.0, 1.0]) * unit.angstrom
        return initial_positions + displacement
```

The parent class will take care of implementing the Metropolis acceptance criteria, collecting acceptance statistics, and updating the `SamplerState` correctly. The constructor accepts an optional `atom_subset` to limit the move to certain atoms. In this case, the `initial_positions` will be the positions of the atom subset only.

```
>>> alanine = testsystems.AlanineDipeptideVacuum()
>>> sampler_state = states.SamplerState(alanine.positions)
>>> thermodynamic_state = states.ThermodynamicState(alanine.system, 300*unit.kelvin)
>>> move = AddOneVector(atom_subset=list(range(sampler_state.n_particles)))
>>> move.apply(thermodynamic_state, sampler_state)
Propose new positions
>>> move.n_accepted
1
>>> move.n_proposed
1
```

CHAPTER 4

Release History

4.1 0.17.0 - Removed Py2 support, faster exact PME treatment

4.1.1 New features

- Add `GlobalParameterFunction` that allows to enslave a `GlobalParameter` to an arbitrary function of controlling

variables (#380). - Allow to ignore velocities when building the dict representation of a `SamplerState`. This can be useful for example to save bandwidth when sending a `SamplerState` over the network and velocities are not required (#386). - Add `DoubleWellDimer_WCAFluid` and `DoubleWellChain_WCAFluid` test

systems (#389).

4.1.2 Enhancements

- New implementation of the exact PME handling that uses the parameter offset feature in OpenMM 7.3. This comes with a

considerable speed improvement over the previous implementation (#380). - Exact PME is now the default for the `alchemical_pme_treatment` parameter in the constructor of `AbsoluteAchemicalFactory` (#386). - It is now possible to have multiple composable states exposing the same attributes/getter/setter in a `CompoundThermodynamicState` (#380).

4.1.3 Bug fixes

- Fixed a bug involving the `NoseHooverChainVelocityVerletIntegrator` with `System` with constraints. The constraints

were not taken into account when calculating the number of degrees of freedom resulting in the temperature not converging to the target value. (#384) - Fixed a bug affecting `reduced_potential_at_states` when computing

the reduced potential of systems in different ‘‘AlchemicalState‘‘s when the same alchemical parameter appeared in force objects split in different force groups. (#385)

4.1.4 Deprecated and API breaks

- Python 2 is not supported anymore.
- The `update_alchemical_charges` attribute of ‘‘AlchemicalState‘, which was deprecated in 0.16.0, has now been removed

since it doesn't make sense with the new parameter offset implementation. - The methods `AlchemicalState.get_alchemical_variable` and `AlchemicalState.set_alchemical_variable` have been deprecated. Use `AlchemicalState.get_alchemical_function` and `AlchemicalState.set_alchemical_function` instead.

4.2 0.16.0 - Py2 deprecated, GlobalParameterState class, SamplerState reads CVs

4.2.1 New features

- Add ability for `SamplerState` to access new OpenMM Custom CV Force Variables (#362).
- `SamplerState.update_from_context` now has keywords to support finer grain updating from the Context. This is only recommended for advanced users (#362).
- Added the new class `states.GlobalParameterState` designed to simplify the implementation of composable states that control global variables (#363).
- Allow restraint force classes to be controlled by a parameter other than `lambda_restraints`. This will enable multi-restraints simulations (#363).

4.2.2 Enhancements

- Global variables of integrators are now automatically copied over the integrator returned by `ContextCache.get_context`. It is possible to specify exception through `ContextCache.INCOMPATIBLE_INTEGRATOR_ATTRIBUTES` (#364).

4.2.3 Others

- Integrator `MCMCMove``s now attempt to recover from NaN automatically by default (with ``n_restart_attempts` set to 4) (#364).

4.2.4 Deprecated

- Python2 is officially deprecated. Support will be dropped in future versions.
- Deprecated the signature of `IComposableState._on_setattr` to fix a bug where the objects were temporarily left in an inconsistent state when an exception was raised and caught.
- Deprecated `update_alchemical_charges` in `AlchemicalState` in anticipation of the new implementation of the exact PME that will be based on the `NonbondedForce` offsets rather than `updateParametersInContext()`.

4.3 0.15.0 - Restraint forces

- Add radially-symmetric restraint custom forces (#336).
- Copy Python attributes of integrators on `deepcopy()` (#336).
- Optimization of `states.CompoundThermodynamicState` deserialization (#338).
- Bugfixes (#332, #343).

4.4 0.14.0 - Exact treatment of alchemical PME electrostatics, water cluster test system, optimizations

4.4.1 New features

- Add a `WaterCluster` testsystem (#322)
- Add exact treatment of PME electrostatics in *alchemy.AbsoluteAlchemicalFactory*. (#320)
- Add method in `ThermodynamicState` for the efficient computation of the reduced potential at a list of states. (#320)

4.4.2 Enhancements

- When a `SamplerState` is applied to many ‘‘Context‘‘s, the units are stripped only once for optimization. (#320)

4.4.3 Bug fixes

- Copy thermodynamic state on compound state initialization. (#320)

4.5 0.13.4 - Barostat/External Force Bugfix, Restart Robustness

4.5.1 Bug fixes

- Fixed implementation bug where `CustomExternalForce` restraining atoms to absolute coordinates caused an issue when a Barostat was used (#310)

4.5.2 Enhancements

- MCMC Integrators now attempt to re-initialize the `Context` object on the last restart attempt when NaN's are encountered. This has internally been shown to correct some instances where normally resetting positions does not work around the NaN's. This is a slow step relative to just resetting positions, but better than simulation crashing.

4.6 0.13.3 - Critical Bugfix to SamplerState Context Manipulation

4.6.1 Critical Fixes

- `SamplerState.apply_to_context()` applies box vectors before positions are set to prevent a bug on non-Reference OpenMM Platforms which can re-order system atoms. (#305)

4.6.2 Additional Fixes

- LibYAML is now optional (#304)
- Fix AppVeyor testing against Python 3.4 (now Python 3.5/3.6 and NumPy 1.12) (#307)
- Release History now included in online Docs

4.7 0.13.2 - SamplerState Slicing and BitWise And/Or Ops

Added support for SamplerState slicing (#298) Added bit operators `and` and `or` to `math_eval` (#301)

4.8 0.13.1 - Bugfix release

- Fix pickling of `CompoundThermodynamicState` (#284).
- Add missing term to OBC2 GB alchemical Force (#288).
- Generalize `forcefactories.restrain_atoms()` to non-protein receptors (#290).
- Standardize integrator global variables in ContextCache (#291).

4.9 0.13.0 - Alternative reaction field models, Langevin splitting MCMCMove

4.9.1 New Features

- Storage Interface module with automatic disk IO handling
- Option for shifted or switched Reaction Field
- `LangevinSplittingDynamic` MCMC move with specifiable sub step ordering
- Nose-Hoover Chain Thermostat

4.9.2 Bug Fixes

- Many doc string cleanups
- Tests are based on released versions of OpenMM
- Tests also compare against development OpenMM, but do not fail because of it
- Fixed bug in Harmonic Oscillator tests' error calculation

- Default collision rate in Langevin Integrators now matches docs

4.10 0.12.1 - Add virtual sites support in alchemy

- Fixed AbsoluteAlchemicalFactory treatment of virtual sites that were previously ignored (#259).
- Add possibility to add ions to the WaterBox test system (#259).

4.11 0.12.0 - GB support in alchemy and new forces module

4.11.1 New features

- Add AbsoluteAlchemicalFactory support for all GB models (#250)
- Added `forces` and `forcefactories` modules implementing `UnishiftedReactionFieldForce` and `replace_reaction_field` respectively. The latter has been moved from `AbsoluteAlchemicalFactory` (#253)
- Add `restrain_atoms` to restrain molecule conformation through an harmonic restrain (#255)

4.11.2 Bug fixes

- Bugfix for `testsystems` that use implicit solvent (#250)
- Bugfix for `ContextCache`: two consecutive calls retrieve the same `Context` with same thermodynamic state and no integrator (#252)

4.12 0.11.2 - Bugfix release

- Hotfix in fringe Python2/3 compatibility issue when using old style serialization systems in Python 2

4.13 0.11.1 - Optimizations

- Adds Drew-Dickerson DNA dodecamer test system (#223)
- Bugfix and optimization to `ContextCache` (#235)
- Compress serialized `ThermodynamicState` strings for speed and size (#232)
- Backwards compatible with uncompressed serialized `ThermodynamicStates`

4.14 0.11.0 - Conda forge installation

4.14.1 New Features

- `LangevinIntegrator` now sets `measure_heat=False` by default for increased performance (#211)

- `AbsoluteAlchemicalFactory` now supports `disable_alchemical_dispersion_correction` to prevent 600x slowdowns with nonequilibrium integration (#218)
- We now require conda-forge as a dependency for testing and deployment (#216)
- Conda-forge added as channel to conda packages

4.15 0.10.0 - Optimizations of ThermodynamicState, renamed AlchemicalFactory

- BREAKS API: Renamed AlchemicalFactory to AbsoluteAlchemicalFactory (#206)
- Major optimizations of ThermodynamicState (#200, #205)
  - Keep in memory only a single System object per compatible state
  - Fast copy/deepcopy
  - Enable custom optimized serialization for multiple states
- Added readthedocs documentation (#191)
- Bugfix for serialization of context when NaN encountered (#199)
- Added tests for Python 3.6 (#184)
- Added tests for integrators (#186, #187)

4.16 0.9.4 - Nonequilibrium integrators overhaul

4.16.1 Major changes

- Overhaul of `LangevinIntegrator` and subclasses to better support nonequilibrium integrators
- Add true reaction-field support to `AlchemicalFactory`
- Add some alchemical test systems

4.16.2 Updates to `openmmtools.integrators.LangevinIntegrator` and friends

API-breaking changes

- The nonequilibrium integrators are now called `AlchemicalNonequilibriumLangevinIntegrator` and `ExternalPerturbationLangevinIntegrator`, and both are subclasses of a common `NonequilibriumLangevinIntegrator` that provides a consistent interface to setting and getting `protocol_work`
- `AlchemicalNonequilibriumLangevinIntegrator` now has a default `alchemical_functions` to eliminate need for every test to treat it as a special case (#180)
- The `get_protocol_work()` method allows you to retrieve the protocol work from any `NonequilibriumLangevinIntegrator` subclass and returns a unit-bearing work. The optional `dimensionless=True` argument returns a dimensionless float in units of kT.

- Integrator global variables now store all energies in natural OpenMM units (kJ/mol) but the new accessor methods (see below) should b used instead of getting integrator global variables for work and heat. (#181)
- Any private methods for adding steps to the integrator have been prepended with _ to hide them from the public API.

New features

- Order of arguments for all `LangevinIntegrator` derivatives matches `openmm.LangevinIntegrator` so it can act as a drop-in replacement. (#176)
- The `get_shadow_work()` and `get_heat()` methods are now available for any `LangevinIntegrator` subclass, as well as the corresponding properties `shadow_work` and heat. The functions also support `dimensionless=True`. (#163)
- The `shadow_work` and `heat` properties were added to all LangevinIntegrator subclasses, returning the values of these properties (if the integrator was constructed with the appropriate `measure_shadow_work=True` or `measure_heat=True` flags) as unit-bearing quantities
- The `get_protocol_work()` and `get_total_work()` methods are now available for any `NonequilibriumLangevinIntegrator`, returning unit-bearing quantities unless `dimensionless=True` is provided in which case they return the work in implicit units of kT. `get_total_work()` requires the integrator to have been constructed with `measure_shadow_work=True`.
- The `protocol_work` and `total_work` properties were added to all `NonequilibriumLangevinIntegrator` subclasses, and return the unit-bearing work quantities. `total_work` requires the integrator to have been constructed with `measure_shadow_work=True`.
- The subclasses have been reworked to support any kwargs that the base classes support, and defaults have all been made consistent.
- Various reset() methods have been added to reset statistics for all `LangevinIntegrator` subclasses.
- All custom integrators support `.pretty_format()` and `.pretty_print()` with optional highlighting of specific step types.

Bugfixes

- Zero-step perturbations now work correctly (#177)
- `AlchemicalNonequilibriumLangevinIntegrator` now correctly supports multiple `H` steps.

Internal changes

- Adding new LangevinIntegrator step methods now uses a `self._register_step_method(step_string, callback_function, supports_force_groups=False)` call to simplify this process.
- Code duplication has been reduced through the use of calling base class methods whenever possible.
- `run_nonequilibrium_switching()` test now uses BAR to test dragging a harmonic oscillator and tests a variety of integrator splittings (`["O { V R H R V } O", "O V R H R V O", "R V O H O V R", "H R V O V R H"]`).
- Integrator tests use deterministic PME and mixed precision when able.

4.16.3 Updates to openmmtools.alchemy.AlchemicalFactory

- Reaction field electrostatics now removes the shift, setting `c_rf = 0`.
- A convenience method AlchemicalFactory.replace_reaction_field() has been added to allow fully-interacting systems to be modified to force `c_rf = 0` by recoding reaction-field electrostatics as a `CustomNonbondedForce`

4.16.4 New `openmmtools.testsystems` classes

- AlchemicalWaterBox was added, which has the first water molecule in the system alchemically modified

CHAPTER 5

Modules

5.1 Test Systems

`openmmtools.testsystems` contains a variety of test systems useful for benchmarking, validation, testing, and debugging.

5.1.1 Analytically tractable systems

These test systems are simple test systems where some properties are analytically tractable.

`HarmonicOscillator`
`PowerOscillator`
`ConstraintCoupledHarmonicOscillator`
`HarmonicOscillatorArray`
`IdealGas`
`MolecularIdealGas`
`Diatom`
`CustomExternalForcesTestSystem`
`CustomGBForceSystem`
`LennardJonesPair`

5.1.2 Clusters and simple fluids

`DiatomicFluid`
`UnconstrainedDiatomicFluid`
`ConstrainedDiatomicFluid`
`DipolarFluid`
`UnconstrainedDipolarFluid`

Continued on next page

Table 2 – continued from previous page

ConstrainedDipolarFluid
LennardJonesCluster
LennardJonesFluid
LennardJonesFluidTruncated
LennardJonesFluidSwitched
LennardJonesGrid
CustomLennardJonesFluidMixture
WCAFluid
DoubleWellDimer_WCAFluid
DoubleWellChain_WCAFluid
TolueneVacuum
TolueneImplicit
TolueneImplicitHCT
TolueneImplicitOBC1
TolueneImplicitOBC2
TolueneImplicitGBn
TolueneImplicitGBn2
MethanolBox
WaterCluster

5.1.3 Solids

SodiumChlorideCrystal

5.1.4 Water boxes

WaterBox
FlexibleWaterBox
FlexibleReactionFieldWaterBox
FlexiblePMEWaterBox
PMEWaterBox
GiantFlexibleWaterBox
FourSiteWaterBox
FiveSiteWaterBox
DischargedWaterBox
FlexibleDischargedWaterBox
GiantFlexibleDischargedWaterBox
DischargedWaterBoxHsites
AlchemicalWaterBox
WaterCluster

5.1.5 Peptide and protein systems

AlanineDipeptideVacuum
AlanineDipeptideImplicit
AlanineDipeptideExplicit
DHFRExplicit

Continued on next page

Table 5 – continued from previous page

LysozymeImplicit
SrcImplicit
SrcExplicit
SrcExplicitReactionField

5.1.6 Complexes

HostGuestVacuum
HostGuestImplicit
HostGuestImplicitHCT
HostGuestImplicitOBC1
HostGuestImplicitOBC2
HostGuestImplicitGBn
HostGuestImplicitGBn2
HostGuestExplicit

5.1.7 Polarizable test systems

AMOEBAIonBox
AMOEBAProteinBox

5.1.8 Test system base classes

These are base classes you can inherit from to develop new test systems.

TestSystem

5.2 Integrators

`openmmtools.integrators` provides a number of high quality integrators implemented using OpenMM's CustomIntegrator facility.

The integrators provided in the `openmmtools.integrators` package subclass the OpenMM CustomIntegrator, providing a more full-featured Pythonic class wrapping the Swig-wrapped CustomIntegrator.

Warning: OpenMM's CompoundIntegrator caches OpenMM `Integrator` objects, but can only return the SWIG-wrapped base integrator object if you call `CompoundIntegrator.getIntegrator()` or `CompoundIntegrator.getCurrentIntegrator()`. If you want to hold onto one of the Python subclasses we make available in `openmmtools.integrators`, you will need to cache the original Python integrator you create. You can still use either `integrator.step()` call, but you *MUST MAKE SURE THAT INTEGRATOR IS SELECTED* currently in `CompoundIntegrator.setCurrentIntegrator(index)` before calling `integrator.step()` or else the behavior is undefined.

5.2.1 Langevin integrators

The entire family of Langevin integrators described by Trotter splittings of the propagator is available. These integrators also support multiple-timestep force splittings and Metropolization. In addition, we provide special subclasses for several popular classes of Langevin integrators.

Note: We highly recommend the excellent geodesic BAOAB (g-BAOAB) integrator of Leimkuhler and Matthews for all equilibrium simulations where only equilibrium configurational properties are of interest. This integrator (g-BAOAB) has extraordinarily good properties for biomolecular simulation.

`LangevinIntegrator`
`VVVRIntegrator`
`BAOABIntegrator`
`GeodesicBAOABIntegrator`
`GHMCIntegrator`

5.2.2 Nonequilibrium integrators

These integrators are available for nonequilibrium switching simulations, and provide additional features for measuring protocol, shadow, and total work.

`NonequilibriumLangevinIntegrator`
`AlchemicalNonequilibriumLangevinIntegrator`
`ExternalPerturbationLangevinIntegrator`

5.2.3 Miscellaneous integrators

Other miscellaneous integrators are available.

`MTSIntegrator`
`DummyIntegrator`
`GradientDescentMinimizationIntegrator`
`VelocityVerletIntegrator`
`AndersenVelocityVerletIntegrator`
`NoseHooverChainVelocityVerletIntegrator`
`MetropolisMonteCarloIntegrator`
`HMCIntegrator`

5.2.4 Mix-ins

A number of useful mix-ins are provided to endow integrators with additional features.

PrettyPrintableIntegrator
ThermostatedIntegrator

5.2.5 Base classes

New integrators can inherit from these base classes to inherit extra features

ThermostatedIntegrator
NonequilibriumLangevinIntegrator

5.3 Thermodynamic and Sampler States

The module `openmmtools.states` contains classes to maintain a consistent state of the simulation.

- `ThermodynamicState`: Represent and manipulate the thermodynamic state of OpenMM System and Context objects.
- `SamplerState`: Represent and cache the state of the simulation that changes when the System is integrated.
- `CompoundThermodynamicState`: Extend the `ThermodynamicState` to handle parameters other than temperature and pressure through the implementations of the `IComposableState` abstract class.
- `GlobalParameterState`: An implementation of `IComposableState` specialized to control OpenMM forces' global parameters.
- `GlobalParameterFunction`: Allow enslaving a global parameter to arbitrary variables and mathematical expressions.

5.3.1 States

ThermodynamicState
SamplerState
CompoundThermodynamicState
IComposableState
GlobalParameterFunction
GlobalParameterState

5.4 Cache

The module `openmmtools.cache` implements a shared LRU cache for OpenMM Context objects that tries to minimize the number of objects in memory at the same time.

5.4.1 Cache objects

LRUCache
ContextCache
global_context_cache

5.5 Markov chain Monte Carlo (MCMC)

`openmmtools` provides an extensible Markov chain Monte Carlo simulation framework.

This module provides a framework for equilibrium sampling from a given thermodynamic state of a biomolecule using a Markov chain Monte Carlo scheme.

It currently offer supports for

- Langevin dynamics (assumed to be free of integration error; use at your own risk]),
- hybrid Monte Carlo,
- generalized hybrid Monte Carlo, and
- Monte Carlo barostat moves,

which can be combined through the `SequenceMove` and `WeightedMove` classes.

By default, the `MCMCMoves` use the fastest OpenMM platform available and a shared global `ContextCache` that minimizes the number of OpenMM `Context` objects that must be maintained at once. The examples below show how to configure these aspects.

Note: To use the `ContextCache` on the CUDA platform, the NVIDIA driver must be set to `shared` mode to allow the process to create multiple GPU contexts.

Using the MCMC framework requires importing `ThermodynamicState` and `SamplerState` from `openmmtools.states`:

```
from simtk import unit
from openmmtools import testsystems, cache
from openmmtools.states import ThermodynamicState, SamplerState
```

Create the initial state (thermodynamic and microscopic) for an alanine dipeptide system in vacuum.

```
test = testsystems.AlanineDipeptideVacuum()
thermodynamic_state = ThermodynamicState(system=test.system, temperature=298*unit.
↪kelvin)
sampler_state = SamplerState(positions=test.positions)
```

Create an MCMC move to perform at every iteration of the simulation, and initialize a sampler instance.

```
ghmc_move = GHMCMove(timestep=1.0*unit.femtosecond, n_steps=50)
langevin_move = LangevinDynamicsMove(n_steps=10)
sampler = MCMCSampler(thermodynamic_state, sampler_state, move=ghmc_move)
```

You can combine them to form a sequence of moves

```
sequence_move = SequenceMove([ghmc_move, langevin_move])
sampler = MCMCSampler(thermodynamic_state, sampler_state, move=sequence_move)
```

or create a move that selects one of them at random with given probability at each iteration.

```
weighted_move = WeightedMove([(ghmc_move, 0.5), (langevin_move, 0.5)])
sampler = MCMCSampler(thermodynamic_state, sampler_state, move=weighted_move)
```

By default the `MCMCMove` use a global ContextCache that creates `Context` on the fastest available OpenMM platform. You can configure the default platform to use before starting the simulation

```
reference_platform = openmm.Platform.getPlatformByName('Reference')
cache.global_context_cache.platform = reference_platform
cache.global_context_cache.time_to_live = 10  # number of read/write operations
```

Minimize and run the simulation for few iterations.

```
sampler.minimize()
sampler.run(n_iterations=2)
```

If you don't want to use a global cache, you can create local ones.

```
local_cache1 = cache.ContextCache(capacity=5, time_to_live=50)
local_cache2 = cache.ContextCache(platform=reference_platform, capacity=1)
sequence_move = SequenceMove([HMCMove(), LangevinDynamicsMove()], context_cache=local_
↪cache1)
ghmc_move = GHMCMove(context_cache=local_cache2)
```

If you don't want to cache `Context` at all but create one every time, you can use the `DummyCache`.

```
dummy_cache = cache.DummyContextCache(platform=reference_platform)
ghmc_move = GHMCMove(context_cache=dummy_cache)
```

This book by Jun Liu is an excellent overview of Markov chain Monte Carlo:

Jun S. Liu. Monte Carlo Strategies in Scientific Computing. Springer, 2008.

5.5.1 MCMC samplers

An MCMC sampler driver is provided that can either utilize a programmed sequence of moves or draw from a weighted set of moves.

`MCMCSampler`
`SequenceMove`
`WeightedMove`

5.5.2 MCMC move types

A number of MCMC component move types that can be arranged into groups or subclassed are provided.

`MCMCMove`
`BaseIntegratorMove`
`MetropolizedMove`
`IntegratorMove`
`LangevinDynamicsMove`
`LangevinSplittingDynamicsMove`
`GHMCMove`
`HMCMove`
`MonteCarloBarostatMove`
`MCDisplacementMove`
`MCRotationMove`

5.6 Sampling from multiple alchemical (or other thermodynamic) states

`openmmtools` provides several schemes for sampling from multiple thermodynamic states within a single calculation:

- `MultistateSampler`: Independent simulations at distinct thermodynamic states
- `ReplicaExchangeSampler`: Replica exchange among thermodynamic states (also called Hamiltonian exchange if only the Hamiltonian is changing)
- `SAMSSampler`: Self-adjusted mixture sampling (also known as optimally-adjusted mixture sampling)

While the thermodynamic states sampled usually differ only in the alchemical parameters, other thermodynamic parameters (such as temperature) can be modulated as well at intermediate alchemical states. This may be useful in, for example, experimenting with ways to reduce correlation times.

In all of these schemes, one or more **replicas** is simulated. Each iteration includes the following phases: | * Allow replicas to switch thermodynamic states (optional) | * Allow replicas to sample a new configuration using Markov chain Monte Carlo (MCMC) | * Each replica computes the potential energy of the current configuration in multiple thermodynamic states | * Data is written to disk

Below, we describe some of the aspects of these samplers.

5.6.1 `MultiStateSampler`: Independent simulations at multiple thermodynamic states

The `MultiStateSampler` allows independent simulations from multiple thermodynamic states to be sampled. In this case, the MCMC scheme is used to propagate each replica by sampling from a fixed thermodynamic state.

$$s_{k,n+1} = s_{k,n}$$
$$x_{k,n+1} \sim p(x|s_{k,n+1})$$

An inclusive "neighborhood" of thermodynamic states around this specified state can be used to define which thermodynamic states the reduced potential should be computed for after each iteration. If all thermodynamic states are included in this neighborhood (the default), the MBAR scheme **:cite:'Shirts2008statistically'** can be used to optimally estimate free energies and uncertainties. If a restricted neighborhood is used (in order to reduce the amount of time spent in the energy evaluation stage), a variant of the L-WHAM (local weighted histogram analysis method) **:cite:'kumar1992weighted'** is used to extract an estimate from all available information.

5.6.2 `ReplicaExchangeSampler`: Replica exchange among thermodynamic states

The `ReplicaExchangeSampler` implements a Hamiltonian replica exchange scheme with Gibbs sampling **:cite:'Chodera2011'** to sample multiple thermodynamic states in a manner that improves mixing of the overall Markov chain. By allowing replicas to execute a random walk in thermodynamic state space, correlation times may be reduced when sampling certain thermodynamic states (such as those with alchemically-softened potentials or elevated temperatures).

In the basic version of this scheme, a proposed swap of configurations between two alchemical states, *i* and *j*, made by comparing the energy of each configuration in each replica and swapping with a basic Metropolis criteria of

$$\begin{aligned} P_{\text{accept}}(i, x_i, j, x_j) &= \min\left\{1, \frac{e^{-[u_i(x_j)+u_j(x_i)]}}{e^{-[u_i(x_i)+u_j(x_j)]}}\right. \\ &= \min\left\{1, \exp\left[\Delta u_{ji}(x_i) + \Delta u_{ij}(x_j)\right]\right. \end{aligned}$$

where x is the configuration of the subscripted states i or j, and u is the reduced potential energy. While this scheme is typically carried out on neighboring states only, we also implement a much more efficient form of Gibbs sampling in which many swaps are attempted to generate an approximately uncorrelated sample of the state permutation over all K **:cite:'Chodera2011'**. This speeds up mixing and reduces the total number of samples needed to produce uncorrelated samples.

5.6.3 `SAMSSampler`: Self-adjusted mixture sampling

The `SAMSSampler` implements self-adjusted mixture sampling (SAMS; also known as optimally adjusted mixture sampling) **:cite:'Tan2017:SAMS'**. This combines one or more replicas that sample from an expanded ensemble with an asymptotically optimal Wang-Landau-like weight update scheme.

$$s_{k,n+1} = p(s|x_{k,n})$$
$$x_{k,n+1} \sim p(x|s_{k,n+1})$$

SAMS state update schemes

Several state update schemes are available:

- `global-jump` (default): The sampler can jump to any thermodynamic state (RECOMMENDED)
- `restricted-range-jump`: The sampler can jump to any thermodynamic state within the specified local neighborhood (EXPERIMENTAL; DISABLED)
- `local-jump`: Only proposals within the specified neighborhood are considered, but rejection rates may be high (EXPERIMENTAL; DISABLED)

SAMS Locality

The local neighborhood is specified by the `locality` parameter. If this is a positive integer, the neighborhood will be defined by state indices `[k - locality, k + locality]`. Reducing locality will restrict the range of states for which reduced potentials are evaluated, which can speed up the energy evaluation stage of each iteration at the cost of restricting the amount of information available for free energy estimation. By default, the `locality` is global, such that energies at all thermodynamic states are computed; this allows the use of MBAR in data analysis.

SAMS weight adaptation algorithm

SAMS provides two ways of accumulating log weights each iteration:

- `optimal` accumulates weight only in the currently visited state `s`
- `rao-blackwellized` accumulates fractional weight in all states within the energy evaluation neighborhood

SAMS initial weight adaptation stage

Because the asymptotically-optimal weight adaptation scheme works best only when the log weights are close to optimal, a heuristic initial stage is used to more rapidly adapt the log weights before the asymptotically optimal scheme is used. The behavior of this first stage can be controlled by setting two parameters:

- `gamma0` controls the initial rate of weight adaptation. By default, this is 1.0, but can be set larger (e.g., 10.0) if the free energy differences between states are much larger.
- `flatness_threshold` controls the number of (fractional) visits to each thermodynamic state that must be accumulated before the asymptotically optimal weight adaptation scheme is used.

5.7 Alchemical factories

`openmmtools.alchemy` contains factories for generating alchemically-modified versions of OpenMM System objects for use in alchemical free energy calculations.

5.7.1 Absolute alchemical factories

Absolute alchical factories modify the `System` object to allow part of the system to be alchemically annihilated or decoupled. This is useful for computing free energies of transfer, solvation, or binding for small molecules.

`AlchemicalFunction`
`AlchemicalState`
`AbsoluteAlchemicalFactory`

5.7.2 Relative alchemical factories

Coming soon!

5.8 Forces

The module `openmmtools.forces` implements custom forces that are not natively found in OpenMM.

5.8.1 Restraint Force Classes

`RadiallySymmetricRestraintForce`
`RadiallySymmetricCentroidRestraintForce`
`RadiallySymmetricBondRestraintForce`
`HarmonicRestraintForceMixIn`
`HarmonicRestraintForce`
`HarmonicRestraintBondForce`
`FlatBottomRestraintForceMixIn`
`FlatBottomRestraintForce`
`FlatBottomRestraintBondForce`

5.8.2 Useful Custom Forces

`UnshiftedReactionFieldForce`

5.8.3 Utility functions

iterate_forces
find_forces

5.8.4 Exceptions

MultipleForcesError
NoForceFoundError

5.9 Force Factories

The module openmmtools.forcefactories implements utility methods and factories to configure system forces.

5.9.1 Force Factories

replace_reaction_field
restrain_atoms

5.10 Storage

This submodule is a user-friendly storage driver which relies on two major classes from the user perspective: StorageIODriver and StorageInterface

5.10.1 StorageIODriver

The StorageIODriver is the abstract base class which handles IO operations on disk with the real data. Derived classes from this handle the specific storage medium, like NetCDF. This class tracks all the known variables, and where they are on the disk. However, because the abstract class cannot know how the derived class actually interacts with the disk, it is up to the derived class to know how each variable writes to disk.

The NetCDFIODriver is the derived StorageIODriver for NetCDF storage. The NetCDFIODriver handles the top level file operations and keeps track of where each variable and group (equivalent to a directory) is on the disk. Read/Write operations are handed off to the individual NCVariableCodec classes which interpret and write to file.

The NCVariableCodec is an abstract base class which defines how data is passed to and from the disk. Its derived classes handled interpreting the specific types of data we want to store and read from disk, e.g. ints, lists, np.arrays etc. Each derived NCVariableCodec enacts is own codec to know how to format the data type for storage on disk, and how to read that data back from disk, converting it to the correct type.

The StorageIODriver's and the StorageInterface work on the principal of not knowing or caring what is on the disk until the user first attempt to access it, the process of initial interaction with the disk is called "*Binding*." Variables and directories are considered "unbound" if they have not accessed the disk yet, and "bound" if they have. This bound/unbound mechanism is to reduce the amount of IO actions to disk, which is a slow process relative to the main code.

StorageIODriver
NetCDFIODriver
NCVariableCodec

5.10.2 Binding

Unbound variables and directories do not know what type of data they will handle, and only store where on the disk data will be accessed. Upon the first attempt to read/write/append, a binding action occurs. The variables check if there is already data on the disk at the known location, what happens next depends on what operation was called: * If read and on disk:

1. Determine the codec the variable will use.
2. Fetch data, only accept data the codec can interpret.

- If read and NOT on disk: Raise error.
- If write/append and on disk:
    1. Ensure data to write is compatible with codec that was used to store data.
    2. Ensure data to store is of the same shape (for non-scalar data)
    3. Store new data.
- If write/append and not on disk:
    1. Allocate storage on disk
    2. Store new data

The variable is now considered "bound" and there are some checks which ensure new data can now be stored on this variable.

5.10.3 StorageInterface

`StorageInterface` (SI) is a layer which runs on top of a provided `StorageIODriver` to create an way for users to interface with the disk with as minimal effort as possible. Variables and directories are treated as user defined properties of the SI, which then those properties can also be given user defined properties to point to other variables below it. E.g. `SI.mydir.myvar` creates a directory object called `mydir` at the top level of the SI object on disk, then `myvar` is the variable inside `mydir` on disk. The depth of this can be arbitrary. None of the user defined properties are bound until the first read/write/append operation, which is done with `.read()` `.write()` and `.append()` functions respectively.

5.10.4 StorageInterfaceDirVar

`StorageInterfaceDirVar` (SIDV) is the class which is assigned to each of the user defined properties in the SI are attached to. This class is what hooks into the `StorageIODriver` and passes the instructions to create/manage variables and handle any other sub-directories/variables attached to it.

StorageInterface
StorageInterfaceDirVar

5.11 Miscellaneous Utilities

`openmmtools.utils` provides a number of useful utilities for working with OpenMM.

5.11.1 Timing functions

time_it
with_timer
Timer

5.11.2 Temporary directories

temporary_directory

5.11.3 Symbolic mathematics

sanitize_expression
math_eval

5.11.4 Quantity functions

is_quantity_close

5.11.5 OpenMM Platform utilities

get_available_platforms
get_fastest_platform

5.11.6 Serialization utilities

serialize
deserialize

5.11.7 Metaclass utilities

with_metaclass
SubhookedABCMeta
find_all_subclasses
find_subclass

5.11.8 OpenMM custom object utilities

RestorableOpenMMObject

openmmtools.scripts contains scripts that may be useful in testing your OpenMM installation is functioning correctly:

5.12 Command-line scripts

./test-openmm-platforms will test the various platforms available to OpenMM to ensure that all systems in openmmtools.testsystems give consistent potential energies. If differences in energies in excess of ENERGY_TOLERANCE (default: 0.06 kcal/mol) are detected, these systems will be serialized to XML for further debugging.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search