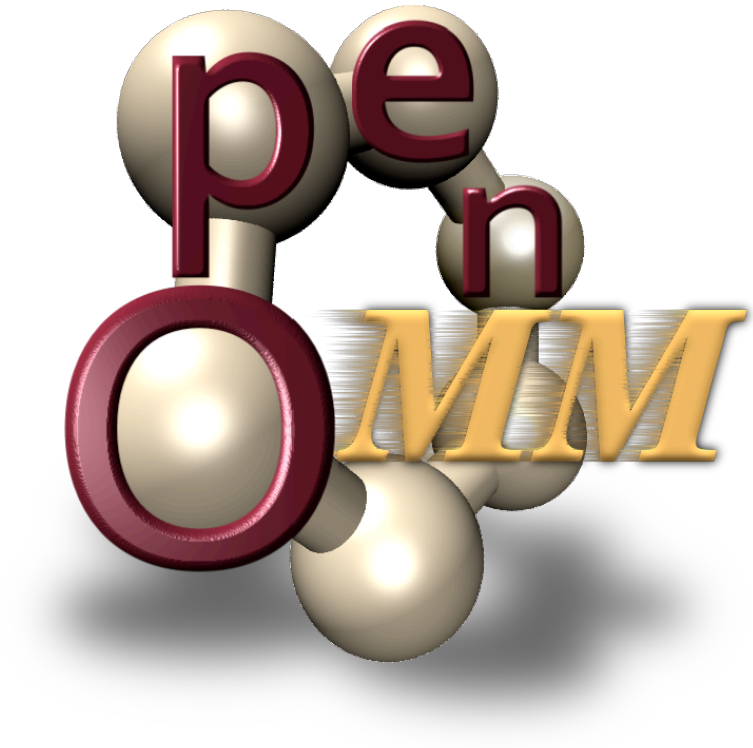

Application Guide

Release 4.0

January 5, 2012

Website: simtk.org/home/openmm

OpenMM Application Guide

Authors

Peter Eastman

Copyright and Permission Notice

Acknowledgments

OpenMM software and all related activities, such as this manual, are funded by the Simbios National Center for Biomedical Computing through the National Institutes of Health Roadmap for Medical Research, Grant U54 GM072970. Information on the National Centers can be found at http://nihroadmap.nih.gov/bioinformatics.

Table of Contents

1 Introduction

OpenMM consists of two parts:

1. A set of libraries that lets programmers easily add molecular simulation features to their programs
2. An "application layer" that exposes those features to end users who just want to run simulations

This guide describes the application layer. If you are not a programmer, but you want to run simulations with OpenMM, this guide is for you. If you *are* a programmer, this guide may still be for you; keep reading to see what OpenMM can do for you.

The first thing to understand is that the OpenMM "application layer" is not exactly an application in the traditional sense: there is no program called "OpenMM" that you run. Rather, it is a collection of libraries written in the Python programming language. Those libraries can easily be chained together to create Python programs that run simulations. But don't worry! You don't need to know anything about Python programming (or programming at all) to use it. Nearly all molecular simulation applications ask you to write some sort of "script" that specifies the details of the simulation to run. With OpenMM, that script happens to be written in Python. But it is no harder to write than those for most other applications, and this guide will teach you everything you need to know.

On the other hand, if you don't mind doing a little programming, this approach gives you enormous power and flexibility. Your script has complete access to the entire OpenMM application programming interface (API), as well as the full power of the Python language and libraries. You have complete control over every detail of the simulation, from defining the molecular system to analyzing the results.

1.1 Online Resources

You can find more documentation and other material at our website http://simtk.org/home/openmm. Among other things there is a discussion forum and several mailing lists with archives.

1.2 Referencing OpenMM

Any work that uses OpenMM should cite the following publication:

M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. LeGrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, V. S. Pande. "Accelerating Molecular Dynamic Simulation on Graphics Processing Units." J. Comp. Chem., 30(6):864-872 (2009).

We depend on academic research grants to fund the OpenMM development efforts; citations of our publication will help demonstrate the value of OpenMM.

2 Installing OpenMM

Detailed installation instructions are found in chapter 3 of the OpenMM Users Guide. What follows is a simplified version of those instructions. They should be sufficient for most cases, but if you run into problems, consult the Users Guide. There also is an online troubleshooting guide that describes common problems and how to fix them (http://wiki.simtk.org/openmm/FAQApp).

2.1 Installing on Mac OS X

OpenMM works on Mac OS X 10.6 or later. It may also work on 10.5, but it has not been tested, and the OpenCL platform (which enables you to run accelerated calculations using OpenCL-supporting GPUs) will not be available.

1. Download the pre-compiled binary of OpenMM for Mac OS X, then double click the .zip file to expand it.

2. If you have not already done so, install Apple's Xcode developer tools from http://developer.apple.com/technologies/tools/. They are required to use OpenMM.

3. Launch the Terminal application. Change to the OpenMM directory by typing

```
cd <openmm_directory>
```

where `<openmm_directory>` is the path to the OpenMM folder. Then run the install script by typing

```
sudo ./install.sh
```

It will prompt you for an install location and the path to the python executable. Unless you are certain you know what you are doing, accept the defaults for both options.

4. (Optional) If you have an Nvidia GPU and want to use the CUDA platform, download CUDA 4.0 from http://developer.nvidia.com/cuda-toolkit-40. Be sure to install both the drivers and toolkit. (This is only needed if you want to use the CUDA platform. Even without it, you can still use the OpenCL platform to run GPU accelerated simulations.)

5. Before running OpenMM, you must add the OpenMM libraries (and CUDA libraries, if you installed those) to your library path so your computer knows where to find them. You can do this by typing

```
export DYLD_LIBRARY_PATH=/usr/local/openmm/lib:/usr/local/cuda/lib
```

This will affect only the particular Terminal window you type it into. If you want to run OpenMM in another Terminal window, you must type the above command in the new window.

6. Verify your installation by running the "testInstallation.py" script found in the "examples" folder of your OpenMM installation. To run it, cd to the examples folder and type

```
python testInstallation.py
```

This script confirms that OpenMM is installed, checks whether GPU acceleration is available (via that OpenCL and/or CUDA platforms), and verifies that all platforms produce consistent results.

Important Note: Some Mac laptops have two GPUs, only one of which is capable of running OpenMM. If you have a laptop, open the System Preferences and go to the Energy Saver panel. On OS X 10.6, look for two radio buttons at the top labeled "Better battery life" and "Higher performance". Make sure that "Higher performance" is selected. On OS X 10.7, there will be a single checkbox labeled "Automatic graphics switching", which should be disabled. Otherwise, trying to run OpenMM may produce an error. You will only see these options if your laptop has two GPUs

2.2 Installing on Linux

1. Download the pre-compiled binary of OpenMM for Linux, then double click the .zip file to expand it.

2. Make sure you have Python 2.6 or 2.7 (other versions will not work) and gcc (we have tested various versions between 4.0 and 4.4) installed on your computer. You can check what versions are installed by typing `python --version` and `gcc --version` into a console window.

3. In a console window, change to the OpenMM directory by typing

```
cd <openmm_directory>
```

where `<openmm_directory>` is the path to the OpenMM folder. Then run the install script by typing

```
sudo ./install.sh
```

It will prompt you for an install location and the path to the python executable. Unless you are certain you know what you are doing, accept the defaults for both options.

4. (Recommended) Install CUDA and/or OpenCL. You can run OpenMM without installing either CUDA and/or OpenCL, but will not be able to take advantage of the accelerated computational power of OpenMM without one or the other.

- If you have an Nvidia GPU, download CUDA 4.0 from http://developer.nvidia.com/cuda-toolkit-40. Be sure to install both the drivers and toolkit. OpenCL is included with the CUDA drivers.
- If you have an AMD GPU, or you want to use the OpenCL platform to run accelerated simulations on the CPU, download the AMD APP SDK from http://developer.amd.com/sdks/amdappsdk/downloads/pages/default.aspx. OpenMM requires version 2.4 or later of the SDK and (if you want to use an AMD GPU) version 11.7 or later of the Catalyst driver.

5. Before running OpenMM, you must add the OpenMM libraries (and CUDA/OpenCL libraries, if you installed those) to your library path. You can do this by typing

```
export LD_LIBRARY_PATH=/usr/local/openmm/lib:/usr/local/cuda/lib
```

This will affect only the particular console window you type it into. If you want to run OpenMM in another console window, you must type the above command in the new window.

6. Verify your installation by running the "testInstallation.py" script found in the "examples" folder of your OpenMM installation. To run it, cd to the examples folder and type

```
python testInstallation.py
```

This script confirms that OpenMM is installed, checks whether GPU acceleration is available (via that OpenCL and/or CUDA platforms), and verifies that all platforms produce consistent results.

2.3 Installing on Windows

1. Download the pre-compiled binary of OpenMM for Windows, then double click the .zip file to expand it. Move the files to C:\Program Files\OpenMM. (On 64 bit Windows, use C:\Program Files (x86)\OpenMM).

2. Make sure you have the 32-bit version of Python 2.6 or 2.7 (other versions will not work) installed on your computer. To do this, launch the Python program (either the command line version or the GUI version). The first line in the Python window will indicate the version you have, e.g., Python 2.6.6, as well as whether you have a 32-bit or 64-bit version.

3. Double click the Python API Installer for your version of Python (2.6 or 2.7) to install the Python components.

4. (Recommended) Install CUDA and/or OpenCL. You can run OpenMM without installing either CUDA and/or OpenCL, but will not be able to take advantage of the accelerated computational power of OpenMM without one or the other.

- If you have an Nvidia GPU, download CUDA 4.0 from http://developer.nvidia.com/cuda-toolkit-40. Be sure to install both the drivers and toolkit. For 64-bit machines, you should install the 64-bit driver, but download the 32-bit version of the toolkit since the OpenMM binary is 32-bit. OpenCL is included with the CUDA drivers.
- If you have an AMD GPU, or you want to use the OpenCL platform to run accelerated simulations on the CPU, download the AMD APP SDK from http://developer.amd.com/sdks/amdappsdk/downloads/pages/default.aspx. OpenMM requires version 2.4 or later of the SDK and (if you want to use an AMD GPU) version 11.7 or later of the Catalyst driver.

5. Before running OpenMM, you must add the OpenMM libraries to your PATH environment variable. You may also need to add the Python executable to your PATH.

a. To find out if the Python executable is already in your PATH, open a command prompt window by clicking on Start -> Programs -> Accessories -> Command Prompt. (On Windows 7, select Start -> All Programs -> Accessories -> Command Prompt). Type

```
python
```

If you get an error message, such as `'python' is not recognized as an internal or external command, operable program or batch file,` then you need to add Python to your PATH. To do so, locate it by typing

```
dir C:\py*
```

The files are typically located in a directory like C:\Python26 or C:\Python27. Remember this location. You will need to enter it, along with the location of the OpenMM libraries, later in this process.

b. Click on Start -> Control Panel -> System (On Windows 7, select Start -> Control Panel -> System and Security -> System)
c. Click on the "Advanced" tab or the "Advanced system settings" link
d. Click "Environment Variables"
e. Under "System variables," select the line for "Path" and click "Edit…"
f. Add C:\Program Files\OpenMM\lib to the "Variable value". If you also need to add Python to your PATH, enter that directory location here. Directory locations need to be separated by semi-colons (;).

6. Verify your installation by running the "testInstallation.py" script found in the "examples" folder of your OpenMM installation. To run it, open a command window, cd to the examples folder, and type

```
python testInstallation.py
```

This script confirms that OpenMM is installed, checks whether GPU acceleration is available (via that OpenCL and/or CUDA platforms), and verifies that all platforms produce consistent results.

3 Running Simulations

3.1 A First Example

Let's begin with our first example of an OpenMM script. It loads a PDB file called "input.pdb", models it using the AMBER99SB force field and TIP3P water model, energy minimizes it, simulates it for 10,000 steps with a Langevin integrator, and saves a frame to a PDB file called "output.pdb" every 1000 time steps.

```
from simtk.openmm.app import *
from simtk.openmm import *
from simtk.unit import *

pdb = PDBFile('input.pdb')
forcefield = ForceField('amber99sb.xml', 'tip3p.xml')
system = forcefield.createSystem(pdb.topology, nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer, constraints=HBonds)
integrator = LangevinIntegrator(300*kelvin, 1/picosecond,
      0.002*picoseconds)
simulation = Simulation(pdb.topology, system, integrator)
simulation.context.setPositions(pdb.positions)
simulation.minimizeEnergy()
simulation.reporters.append(PDBReporter('output.pdb', 1000))
simulation.step(10000)
```

Example 3.1

You can find this script in the "examples" folder of your OpenMM installation. It is called "simulatePdb.py". To execute it from a command line, go to your terminal/console/command prompt window (see Chapter 2 on setting up the window to use OpenMM). Navigate to the "examples" folder by typing

```
cd <examples_directory>
```

where the typical directory is `/usr/local/openmm/examples` on Linux and Mac machines and "`C:\Program Files\OpenMM\examples`" on Windows machines.

Then type

```
python simulatePdb.py
```

You can name your own scripts whatever you want, but their names should end with ".py". Let's go through the script line by line and see how it works.

```
from simtk.openmm.app import *
from simtk.openmm import *
from simtk.unit import *
```

These lines are just telling the Python interpreter about some libraries we will be using. Don't worry about exactly what they mean. Just include them at the start of all your scripts.

```
pdb = PDBFile('input.pdb')
```

This line loads the PDB file from disk. (The input.pdb file in the examples directory contains the villin headpiece in explicit solvent.) More precisely, it creates a PDBFile object, passes the file name input.pdb to it as an argument, and assigns the object to a variable called `pdb`. The PDBFile object contains the information that was read from the file: the molecular topology and atom positions. Your file need not be called "input.pdb". Feel free to change this line to specify any file you want. Make sure you include the single quotes around the file name.

```
forcefield = ForceField('amber99sb.xml', 'tip3p.xml')
```

This line specifies the force field to use for the simulation. Force fields are defined by XML files. Chapter 5 describes how to write these files, if you are interested in that sort of thing, but you probably won't need to. OpenMM includes XML files defining lots of standard force fields (see section 3.3.1). In this case we load two of those files: amber99sb.xml, which contains the AMBER99SB force field, and tip3p.xml, which contains the TIP3P water model. The ForceField object is assigned to a variable called `forcefield`.

```
system = forcefield.createSystem(pdb.topology, nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer, constraints=HBonds)
```

This line combines the force field with the molecular topology loaded from the PDB file to create a complete mathematical description of the system we want to simulate. (More precisely, we invoke the ForceField object's "createSystem" function. It creates a System object, which we assign to the variable `system`.) It specifies some additional options about how to do that: use particle mesh Ewald for the long range electrostatic interactions (`nonbondedMethod=PME`), use a 1 nm cutoff for the direct space interactions (`nonbondedCutoff=1*nanometer`), and constrain the length of all bonds that involve a hydrogen atom (`constraints=HBonds`).

```
integrator = LangevinIntegrator(300*kelvin, 1/picosecond,
      0.002*picoseconds)
```

This line creates the integrator to use for advancing the equations of motion. It specifies a LangevinIntegrator, which (surprise!) performs Langevin dynamics, and assigns it to a variable called `integrator`. It also specifies the values of three parameters that are specific to Langevin dynamics: the simulation temperature (300K), the friction coefficient (1 ps^{-1}), and the step size (0.002 ps).

```
simulation = Simulation(pdb.topology, system, integrator)
```

This line combines the molecular topology, system, and integrator to begin a new simulation. It creates a Simulation object and assigns it to a variable called `simulation`. A Simulation object coordinates all the processes involved in running a simulation, such as advancing time and writing output.

```
simulation.context.setPositions(pdb.positions)
```

This line specifies the initial atom positions for the simulation: in this case, the positions that were loaded from the PDB file.

```
simulation.minimizeEnergy()
```

This line tells OpenMM to perform a local energy minimization. It is usually a good idea to do this at the start of a simulation, since the coordinates in the PDB file might produce very large forces.

```
simulation.reporters.append(PDBReporter('output.pdb', 1000))
```

This line creates a "reporter" to generate output during the simulation, and adds it to the Simulation object's list of reporters. A PDBReporter writes structures to a PDB file. We specify that the output file should be called "output.pdb", and that a structure should be written every 1000 time steps.

```
simulation.step(10000)
```

Finally, we run the simulation, integrating the equations of motion for 10,000 time steps. Once it is finished, you can load the PDB file into any program you want for analysis and visualization (VMD, PyMol, AmberTools, etc.).

3.2 Using AMBER Files

OpenMM can build a system in two different ways. One option, as shown above, is to start with a PDB file and then select a force field with which to model it. Alternatively, you can use AmberTools to model your system. In that case, you provide a prmtop file and an inpcrd file. OpenMM loads the files and creates a system from them. This is shown in the following script. It can be found in OpenMM's "examples" folder with the name "simulateAmber.py".

```
from simtk.openmm.app import *
from simtk.openmm import *
from simtk.unit import *

prmtop = AmberPrmtopFile('input.prmtop')
inpcrd = AmberInpcrdFile('input.inpcrd')
system = prmtop.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer, constraints=HBonds)
integrator = LangevinIntegrator(300*kelvin, 1/picosecond,
      0.002*picoseconds)
simulation = Simulation(prmtop.topology, system, integrator)
simulation.context.setPositions(inpcrd.positions)
simulation.minimizeEnergy()
simulation.reporters.append(PDBReporter('output.pdb', 1000))
simulation.step(10000)
```

Example 3.2

This script is very similar to the previous one. There are just a few significant differences:

```
prmtop = AmberPrmtopFile('input.prmtop')
inpcrd = AmberInpcrdFile('input.inpcrd')
```

In these lines, we load the prmtop file and inpcrd file. More precisely, we create AmberPrmtopFile and AmberInpcrdFile objects and assign them to the variables `prmtop` and `inpcrd`, respectively. As before, you can change these lines to specify any files you want. Be sure to include the single quotes around the file names.

```
system = prmtop.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer, constraints=HBonds)
```

This line creates the system. In the previous section, we loaded the topology from a PDB file and then had the force field create a system based on it. In this case, we don't need a force field; the prmtop file already contains the force field parameters, so it can create the system directly.

```
simulation = Simulation(prmtop.topology, system, integrator)
simulation.context.setPositions(inpcrd.positions)
```

Notice that we now get the topology from the prmtop file and the atom positions from the inpcrd file. In the previous section, both of these came from a PDB file, but AMBER puts the topology and positions in separate files.

3.3 Simulation Parameters

In almost all cases, you can simply use one of the two scripts given above (the one in section 3.1 if you are starting from a PDB file, or the one in section 3.2 if you are starting from AMBER prmtop and inpcrd files), then customize it to suit your needs. Now let's consider lots of ways you might want to customize it.

3.3.1 Force Fields

When you create a force field, you specify one or more XML files from which to load the force field definition. Most often, there will be one file to define the main force field, and possibly a second file to define the water model (either implicit or explicit). For example:

```
forcefield = ForceField('amber99sb.xml', 'tip3p.xml')
```

For the main force field, OpenMM provides the following options:

File	Force Field
amber96.xml	AMBER96[1]
amber99sb.xml	AMBER99[2] with modified backbone torsions[3]
amber99sbildn.xml	AMBER99SB plus improved side chain torsions[4]
amber99sbnmr.xml	AMBER99SB with modifications to fit NMR data[5]
amber03.xml	AMBER03[6]
amber10.xml	AMBER10
amoeba2009.xml	AMOEBA[7]

The AMBER files do not include parameters for water molecules. This allows you to separately select which water model you want to use. For simulations that include explicit water molecules, you should also specify one of the following files:

File	Water Model
tip3p.xml	TIP3P water model[8]
spce.xml	SPC/E water model[9]

For the AMOEBA force field, only one explicit water model is currently available and the water parameters are included in the file amoeba2009.xml. Also the AMOEBA force field file only includes the parameters for amino acids and ions; nucleic acids will be included in the next release.

If you want to include an implicit solvation model, you can also specify one of the following files:

File	Implicit Solvation Model
amber96_obc.xml	GBSA-OBC solvation model[11] for use with AMBER96 force field
amber99_obc.xml	GBSA-OBC solvation model for use with AMBER99 force fields
amber03_obc.xml	GBSA-OBC solvation model for use with AMBER03 force field
amber10_obc.xml	GBSA-OBC solvation model for use with AMBER10 force field
amoeba2009_gk.xml	Generalized Kirkwood solvation model[12] for use with AMOEBA force field

For example, to use the GBSA-OBC solvation model with the Amber99SB force field, you would type:

```
forcefield = ForceField('amber99sb.xml', 'amber99_obc.xml')
```

If you are running a vacuum simulation, you do not need to specify a water model. The following line specifies the AMBER10 force field and no water model. If you try to use it with a PDB file that contains explicit water, it will produce an error since no water parameters are defined:

```
forcefield = ForceField('amber10.xml')
```

3.3.2 Nonbonded Interactions

When creating the system (either from a force field or a prmtop file), you can specify options about how nonbonded interactions should be treated:

```
system = prmtop.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer)
```

The `nonbondedMethod` parameter can have any of the following values:

Value	Meaning
`NoCutoff`	No cutoff is applied.
`CutoffNonPeriodic`	The reaction field method is used to eliminate all interactions beyond a cutoff distance. Not valid for AMOEBA.
`CutoffPeriodic`	The reaction field method is used to eliminate all interactions beyond a cutoff distance. Periodic boundary conditions are applied, so each atom interacts only with the nearest periodic copy of every other atom. Not valid for AMOEBA.
`Ewald`	Periodic boundary conditions are applied. Ewald summation is used to compute long range interactions. (This option is rarely used, since PME is much faster for all but the smallest systems.) Not valid for AMOEBA.
`PME`	Periodic boundary conditions are applied. The Particle Mesh Ewald method is used to compute long range interactions.

When using any method other than `NoCutoff`, you should also specify a cutoff distance. Be sure to specify units, as shown in the examples above. For example, `nonbondedCutoff=1.5*nanometers` or `nonbondedCutoff=12*angstroms` are legal values.

When using `Ewald` or `PME`, you can optionally specify an error tolerance for the force computation. For example:

```
system = prmtop.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer, ewaldErrorTolerance=0.00001)
```

The error tolerance is roughly equal to the fractional error in the forces due to truncating the Ewald summation. If you do not specify it, a default value of 0.0005 is used.

3.3.2.1 Nonbonded Forces for AMOEBA

For the AMOEBA force field, the valid values for the `nonbondedMethod` are `NoCutoff` and `PME`. The other nonbonded methods, `CutoffNonPeriodic`, `CutoffPeriodic`, and `Ewald` are unavailable for this force field.

For implicit solvent runs using AMOEBA, only the `nonbondedMethod` option `NoCutoff` is available.

3.3.2.1.1 Lennard-Jones Interaction Cutoff Value

In addition, for the AMOEBA force field a cutoff for the Lennard-Jones interaction independent of the value used for the electrostatic interactions may be specified using the keyword `vdwCutoff`.

```
system = forcefield.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer, ewaldErrorTolerance=0.00001,
      vdwCutoff=1.2*nanometer)
```

If `vdwCutoff` is not specified, then the value of `nonbondedCutoff` is used for the Lennard-Jones interactions.

3.3.2.1.2 Specifying the Polarization Method

OpenMM allows the setting of several other parameters particular to the AMOEBA force field. The `mutualInducedTargetEpsilon` option allows you to specify the accuracy to which the induced dipoles are calculated at each time step; the default value is 0.00001. The `polarization` setting determines whether the calculation of the induced dipoles is continued until the dipoles are self-consistent to within the tolerance specified by `mutualInducedTargetEpsilon` or whether a quick estimate of the induced dipoles is used instead. The first option corresponds to the `polarization='mutual'` setting and is the default; the quick estimate option is given by `polarization='direct'` and in this case, `mutualInducedTargetEpsilon` is ignored, if provided. Simulations using `polarization='direct'` will be significantly faster than those with `polarization='mutual'`, but less accurate. Examples using the two options are given below:

```
system = forcefield.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer,ewaldErrorTolerance=0.00001,
      vdwCutoff=1.2*nanometer, mutualInducedTargetEpsilon=0.01)

system = forcefield.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer,ewaldErrorTolerance=0.00001,
      vdwCutoff=1.2*nanometer, polarization ='direct')
```

3.3.2.1.3 Implicit Solvent and Solute Dielectrics

For implicit solvent simulations using the AMOEBA force field, the 'amoeba2009_gk.xml' file should be included in the initialization of the force field:

```
forcefield = ForceField('amoeba2009.xml', 'amoeba2009_gk.xml')
```

Only the `nonbondedMethod` option `NoCutoff` is available for implicit solvent runs using AMOEBA. In addition, the solvent and solute dielectric values can be specified for implicit solvent simulations:

```
system=forcefield.createSystem(nonbondedMethod=NoCutoff,
      soluteDielectric=2.0, solventDielectric=80.0)
```

The default values are 1.0 for the solute dielectric and 78.3 for the solvent dielectric.

3.3.3 Constraints

When creating the system (either from a force field or a prmtop file), you can optionally tell OpenMM to constrain certain bond lengths and angles. For example,

```
system = prmtop.createSystem(nonbondedMethod=NoCutoff, constraints=HBonds)
```

The `constraints` parameter can have any of the following values:

Value	Meaning
`None`	No constraints are applied. This is the default value.
`HBonds`	The lengths of all bonds that involve a hydrogen atom are constrained.
`AllBonds`	The lengths of all bonds are constrained.
`HAngles`	The lengths of all bonds are constrained. In addition, all angles of the form H-X-H or H-O-X (where X is an arbitrary atom) are constrained.

The main reason to use constraints is that it allows one to use a larger integration time step. With no constraints, one is typically limited to a time step of about 1 fs. With `HBonds` constraints, this can be increased to about 2 fs. With `HAngles`, it can be further increased to 3.5 or 4 fs.

Regardless of the value of this parameter, OpenMM makes water molecules completely rigid, constraining both their bond lengths and angles. You can disable this behavior with the `rigidWater` parameter:

```
system = prmtop.createSystem(nonbondedMethod=NoCutoff, constraints=None,
      rigidWater=False)
```

Be aware that flexible water may require you to further reduce the integration step size, typically to about 0.5 fs.

3.3.4 Integrators

OpenMM offers a choice of several different integration methods. You select which one to use by creating an integrator object of the appropriate type.

3.3.4.1 Langevin Integrator

In the examples of the previous sections, we used Langevin integration:

```
integrator = LangevinIntegrator(300*kelvin, 1/picosecond,
      0.002*picoseconds)
```

The three parameter values in this line are the simulation temperature (300K), the friction coefficient (1 ps^{-1}), and the step size (0.002 ps). You are free to change these to whatever values you want. Be sure to specify units on all values. For example, the step size could be written either as `0.002*picoseconds` or `2*femtoseconds`. They are exactly equivalent.

3.3.4.2 Leapfrog Verlet Integrator

A leapfrog Verlet integrator can be used for running constant energy dynamics. The command for this is:

```
integrator = VerletIntegrator(0.002*picoseconds)
```

The only option is the step size.

3.3.4.3 Brownian Integrator

Brownian (diffusive) dynamics can be used by specifying the following:

```
integrator = BrownianIntegrator(300*kelvin, 1/picosecond,
      0.002*picoseconds)
```

The parameters are the same as for Langevin dynamics: temperature (300K), friction coefficient (1 ps^{-1}), and step size (0.002 ps).

3.3.4.4 Variable Time Step Langevin Integrator

A variable time step Langevin integrator continuously adjusts its step size to keep the integration error below a specified tolerance. In some cases, this can allow you to use a larger average step size than would be possible with a fixed step size integrator. It also is very useful in cases where you do not know in advance what step size will be stable, such as when first equilibrating a system. You create this integrator with the following command:

```
integrator = VariableLangevinIntegrator(300*kelvin, 1/picosecond, 0.001)
```

In place of a step size, you specify an integration error tolerance (0.001 in this example). It is best not to think of this value as having any absolute meaning. Just think of it as an adjustable parameter that affects the step size and integration accuracy. Smaller values will produce a smaller average step size. You should try different values to find the largest one that produces a trajectory sufficiently accurate for your purposes.

3.3.4.5 Variable Time Step Leapfrog Verlet Integrator

A variable time step leapfrog Verlet integrator works similarly to the variable time step Langevin integrator in that it continuously adjusts its step size to keep the integration error below a specified tolerance. The command for this integrator is:

```
integrator = VariableVerletIntegrator(0.001)
```

The parameter is the integration error tolerance (0.001), whose meaning is the same as for the Langevin integrator.

3.3.5 Temperature Coupling

If you want to run a simulation at constant temperature, using a Langevin integrator (as shown in the examples above) is usually the best way to do it. OpenMM does provide an alternative, however: you can use a Verlet integrator, then add an Andersen thermostat to your system to provide temperature coupling.

To do this, add a single line to the script as shown below. (The lines in grey are just for context.)

```
...
system = prmtop.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer, constraints=HBonds)
system.addForce(AndersenThermostat(300*kelvin, 1/picosecond))
integrator = VerletIntegrator(0.002*picoseconds)
...
```

The two parameters of the Andersen thermostat are the temperature (300K) and collision frequency (1 ps^{-1}).

3.3.6 Pressure Coupling

All the examples so far have been constant volume simulations. If you want to run at constant pressure instead, add a Monte Carlo barostat to your system. You do this exactly the same way you added the Andersen thermostat in the previous section:

```
...
system = prmtop.createSystem(nonbondedMethod=PME,
      nonbondedCutoff=1*nanometer, constraints=HBonds)
system.addForce(MonteCarloBarostat(1*bar, 300*kelvin))
integrator = LangevinIntegrator(300*kelvin, 1/picosecond,
      0.002*picoseconds)
...
```

The parameters of the Monte Carlo barostat are the pressure (1 bar) and temperature (300K). The barostat assumes the simulation is being run at constant temperature, but it does not itself do anything to regulate the temperature. It is therefore critical that you always use it along with a Langevin integrator or Andersen thermostat, and that you specify the same temperature for both the barostat and the integrator or thermostat. Otherwise, you will get incorrect results.

3.3.7 Energy Minimization

As seen in the examples, performing a local energy minimization takes a single line in the script:

```
simulation.minimizeEnergy()
```

In most cases, that is all you need. There are two optional parameters you can specify if you want further control over the minimization. First, you can specify a tolerance for when the energy should be considered to have converged:

```
simulation.minimizeEnergy(tolerance=10*kilojoule/mole)
```

If you do not specify this parameter, a default tolerance of 1 kJ/mole is used.

Second, you can specify a maximum number of iterations:

```
simulation.minimizeEnergy(maxIterations=100)
```

The minimizer will exit once the specified number of iterations is reached, even if the energy has not yet converged. If you do not specify this parameter, the minimizer will continue until convergence is reached, no matter how many iterations it takes.

These options are independent. You can specify both if you want:

```
simulation.minimizeEnergy(tolerance=0.1*kilojoule/mole, maxIterations=500)
```

3.3.8 Removing Center of Mass Motion

By default, OpenMM removes all center of mass motion at every time step so the system as a whole does not drift with time. This is almost always what you want. In rare situations, you may want to allow the system to drift with time. You can do this by specifying the `removeCMMotion` parameter when you create the System:

```
system = forcefield.createSystem(pdb.topology, nonbondedMethod=NoCutoff,
      removeCMMotion=False)
```

3.3.9 Writing Trajectories

OpenMM can save simulation trajectories to disk in two formats: PDB and DCD. Both of these are widely supported formats, so you should be able to read them into most analysis and visualization programs.

To save a trajectory, just add a "reporter" to the simulation, as shown in the example scripts above:

```
simulation.reporters.append(PDBReporter('output.pdb', 1000))
```

The two parameters of the PDBReporter are the output filename and how often (in number of time steps) output structures should be written. To use DCD format, just replace "PDBReporter" with "DCDReporter". The parameters represent the same values:

```
simulation.reporters.append(DCDReporter('output.dcd', 1000))
```

4 Advanced Examples

In the previous chapter, we looked at some basic scripts for running simulations and saw lots of ways to customize them. If that is all you want to do—run straightforward molecular simulations—you already know everything you need to know. Just use the example scripts and customize them in the ways described in section 3.3.

OpenMM can do far more than that. Your script has the full OpenMM API at its disposal, along with all the power of the Python language and libraries. In this chapter, we will consider some examples that illustrate more advanced techniques. Remember that these are still only examples; it would be impossible to give an exhaustive list of everything OpenMM can do. Hopefully they will give you a sense of what is possible, and inspire you to experiment further on your own.

Starting in this section, we will assume some knowledge of programming, as well as familiarity with the OpenMM API. Consult the OpenMM Users Guide and API documentation if you are uncertain about how something works. You can also use the Python "help" command. For example,

```
help(Simulation)
```

will print detailed documentation on the Simulation class.

4.1 Simulated Annealing

Here is a very simple example of how to do simulated annealing. The following lines linearly reduce the temperature from 300K to 0K in 100 increments, executing 1000 time steps at each temperature:

```
...
simulation.context.setPositions(pdb.positions)
simulation.minimizeEnergy()
for i in range(100):
    integrator.setTemperature(3*(100-i)*kelvin)
    simulation.step(1000)
```

Example 4.1

This code needs very little explanation. The loop is executed 100 times. Each time through, it adjusts the temperature of the LangevinIntegrator and then calls `step(1000)` to take 1000 time steps.

4.2 Applying an External Force to Particles: a Spherical Container

In this example, we will simulate a non-periodic system contained inside a spherical container with radius 2 nm. We implement the container by applying a harmonic potential to every particle:

$$E(r) = \begin{cases} 0 & r \le 2 \\ 100(r-2)^2 & r > 2 \end{cases}$$

where r is the distance of the particle from the origin, measured in nm. We can easily do this using OpenMM's CustomExternalForce class. This class applies a force to some or all of the particles in the system, where the energy is an arbitrary function of each particle's (x, y, z) coordinates. Here is the code to do it:

```
...
system = forcefield.createSystem(pdb.topology,
      nonbondedMethod=CutoffNonPeriodic, nonbondedCutoff=1*nanometer,
      constraints=None)
force = CustomExternalForce('100*max(0, r-2)^2; r=sqrt(x*x+y*y+z*z)')
system.addForce(force)
for i in range(system.getNumParticles()):
    force.addParticle(i, [])
integrator = LangevinIntegrator(300*kelvin, 91/picosecond,
0.002*picoseconds)
...
```

Example 4.2

The first thing it does is create a CustomExternalForce object and add it to the System. The argument to CustomExternalForce is a mathematical expression specifying the energy of each particle. This can be any function of *x*, *y*, and *z* you want. It also can depend on global or per-particle parameters. A wide variety of restraints, steering forces, shearing forces, etc. can be implemented with this method.

Next it must specify which particles to apply the force to. In this case, we want it to affect every particle in the system, so we loop over them and call `addParticle()` once for each one. The two arguments are the index of the particle to affect, and the list of per-particle parameter values (an empty list in this case). If we had per-particle parameters, such as to make the force stronger for some particles than for others, this is where we would specify them.

Notice that we do all of this immediately after creating the System. That is not an arbitrary choice. If you add new forces to a System, you must do so *before* creating the Simulation. Once you create a Simulation, modifying the System will have no effect on that Simulation.

4.3 Extracting and Reporting Forces (and other data)

OpenMM provides reporters for two output formats: PDB and DCD. Both of those formats store only positions, not velocities, forces, or other data. In this section, we create a new reporter that outputs forces. This illustrates two important things: how to write a reporter, and how to query the simulation for forces or other data.

Here is the definition of the ForceReporter class:

```
class ForceReporter(object):
    def __init__(self, file, reportInterval):
        self._out = open(file, 'w')
        self._reportInterval = reportInterval

    def __del__(self):
        self._out.close()

    def describeNextReport(self, simulation):
        steps = self._reportInterval -
            simulation.currentStep%self._reportInterval
        return (steps, False, False, True, False)

    def report(self, simulation, state):
        forces =
            state.getForces().value_in_unit(kilojoules/mole/nanometer)
        for f in forces:
            print >>self._out, f[0], f[1], f[2]
```

Example 4.3

The constructor and destructor are straightforward. The arguments to the constructor are the output filename and the interval (in time steps) at which it should generate reports. It opens the output file for writing and records the reporting interval. The destructor closes the file.

We then have two methods that every reporter must implement: `describeNextReport()` and `report()`. A Simulation object periodically calls `describeNextReport()` on each of its reporters to find out when that reporter will next generate a report, and what information will be needed to generate it. The return value should be a five element tuple, whose elements are as follows:

- The number of time steps until the next report. We calculate this as *(report interval)-(current step)%(report interval)*. For example, if we want a report every 100 steps and the simulation is currently on step 530, we will return 100-(530%100) = 70.
- Whether the next report will need particle positions.
- Whether the next report will need particle velocities.
- Whether the next report will need forces.

- Whether the next report will need energies.

When the time comes for the next scheduled report, the Simulation calls `report()` to generate the report. The arguments are the Simulation object, and a State that is guaranteed to contain all the information that was requested by `describeNextReport()`. A State object contains a snapshot of information about the simulation, such as forces or particle positions. We call `getForces()` to retrieve the forces and convert them to the units we want to output (kJ/mole/nm). Then we loop over each value and write it to the file. To keep the example simple, we just print the values in text format, one line per particle. In a real program, you might choose a different output format.

Now that we have defined this class, we can use it exactly like any other reporter. For example,

```
simulation.reporters.append(ForceReporter('forces.txt', 100))
```

will output forces to a file called "forces.txt" every 100 time steps.

4.4 Computing Energies

This example illustrates a different sort of analysis. Instead of running a simulation, assume we have already identified a set of structures we are interested in. These structures are saved in a set of PDB files. We want to loop over all the files in a directory, load them in one at a time, and compute the potential energy of each one. Assume we have already created our System and Simulation. The following lines perform the analysis:

```
import os
for file in os.listdir('structures'):
    pdb = PDBFile(os.path.join('structures', file))
    simulation.context.setPositions(pdb.positions)
    state = simulation.context.getState(getEnergy=True)
    print file, state.getPotentialEnergy()
```

Example 4.4

We use Python's `listdir()` function to list all the files in the directory. We create a PDBFile object for each one and call `setPositions()` on the Context to specify the particle positions loaded from the PDB file. We then compute the energy by calling `getState()` with the option `getEnergy=True`, and print it to the console along with the name of the file.

5 Creating Force Fields

OpenMM uses a simple XML file format to describe force fields. It includes many common force fields, but you can also create your own. A force field can use all the standard OpenMM force classes, as well as the very flexible custom force classes. You can even extend the ForceField class to add support for completely new forces, such as ones defined in plugins. This makes it a powerful tool for force field development.

5.1 Basic Concepts

Let's start by considering how OpenMM defines a force field. There are a small number of basic concepts to understand.

5.1.1 Atom Types and Atom Classes

Force field parameters are assigned to atoms based on their "atom types". Atom types should be the most specific identification of an atom that will ever be needed. Two atoms should have the same type only if the force field will always treat them identically in every way.

Multiple atom types can be grouped together into "atom classes". In general, two types should be in the same class if the force field usually (but not necessarily always) treats them identically. For example, the α-carbon of an alanine residue will probably have a different atom type than the α-carbon of a leucine residue, but both of them will probably have the same atom class.

All force field parameters can be specified either by atom type or atom class. Classes exist as a convenience to make force field definitions more compact. If necessary, you could define everything in terms of atom types, but when many types all share the same parameters, it is convenient to only have to specify them once.

5.1.2 Residue Templates

Types are assigned to atoms by matching residues to templates. A template specifies a list of atoms, the type of each one, and the bonds between them. For each residue in the PDB file, the force field searches its list of templates for one that has an identical set of atoms with identical bonds between them. When matching templates, neither the order of the atoms nor their names matter; it only cares about their elements and the set of bonds between them. (The PDB file reader does care about names, of course, since it needs to figure out which atom each line of the file corresponds to.)

5.1.3 Forces

Once a force field has defined its atom types and residue templates, it must define its force field parameters. This generally involves one block of XML for each Force object that will be added to the System. The details are different for each Force, but it generally consists of a set of rules for adding interactions based on bonds and atom types or classes. For example, when adding a HarmonicBondForce, the force field will loop over every pair of bonded atoms, check their types and classes, and see if they match any of its rules. If so, it will call `addBond()` on the HarmonicBondForce. If none of them match, it simply ignores that pair and continues.

5.2 Writing the XML File

The root element of the XML file must be a `<ForceField>` tag:

```
<ForceField>
...
</ForceField>
```

The `<ForceField>` tag contains the following children:

- An `<AtomTypes>` tag containing the atom type definitions
- A `<Residues>` tag containing the residue template definitions
- Zero or more tags defining specific forces

The order of these tags does not matter. They are described in details below.

5.2.1 <AtomTypes>

The atom type definitions look like this:

```
<AtomTypes>
 <Type name="0" class="N" element="N" mass="14.00672"/>
 <Type name="1" class="H" element="H" mass="1.007947"/>
 <Type name="2" class="CT" element="C" mass="12.01078"/>
 ...
</AtomTypes>
```

There is one `<Type>` tag for each atom type. It specifies the name of the type, the name of the class it belongs to, the symbol for its element, and its mass in amu. The names are arbitrary strings: they need not be numbers, as in this example. The only requirement is that all types have unique names. The classes are also arbitrary strings, and in general will not be unique. Two types belong to the same class if they list the same value for the `class` attribute.

5.2.2 <Residues>

The residue template definitions look like this:

```
<Residues>
 <Residue name="ACE">
  <Atom name="HH31" type="710"/>
  <Atom name="CH3" type="711"/>
  <Atom name="HH32" type="710"/>
  <Atom name="HH33" type="710"/>
  <Atom name="C" type="712"/>
  <Atom name="O" type="713"/>
  <Bond from="0" to="1"/>
  <Bond from="1" to="2"/>
  <Bond from="1" to="3"/>
  <Bond from="1" to="4"/>
  <Bond from="4" to="5"/>
  <ExternalBond from="4"/>
 </Residue>
 <Residue name="ALA">
  ...
 </Residue>
 ...
</Residues>
```

There is one <Residue> tag for each residue template. That in turn contains the following tags:

- An <Atom> tag for each atom in the residue. This specifies the name of the atom and its atom type.
- A <Bond> tag for each pair of atoms that are bonded to each other. The to and from attributes are the indices of the two bonded atoms (starting from 0) in the order they were listed. For example, <Bond from="1" to="3"/> describes a bond between atom CH3 and atom HH33.
- An <ExternalBond> tag for each atom that will be bonded to an atom of a different residue.

5.2.3 <HarmonicBondForce>

To add a HarmonicBondForce to the System, include a tag that looks like this:

```
<HarmonicBondForce>
 <Bond class1="C" class2="C" length="0.1525" k="259408.0"/>
 <Bond class1="C" class2="CA" length="0.1409" k="392459.2"/>
 <Bond class1="C" class2="CB" length="0.1419" k="374049.6"/>
 ...
</HarmonicBondForce>
```

Every <Bond> tag defines a rule for creating harmonic bond interactions between atoms. Each tag may identify the atoms either by type (using the attributes type1 and type2) or by class (using the attributes class1 and class2). For every pair of bonded atoms, the force field searches for a rule whose atom types or atom classes match the two atoms. If it finds one, it calls addBond() on the HarmonicBondForce with the specified parameters. Otherwise, it ignores that pair and continues. length is the equilibrium bond length in nm, and k is the spring constant in kJ/mol/nm^2.

5.2.4 <HarmonicAngleForce>

To add a HarmonicAngleForce to the System, include a tag that looks like this:

```
<HarmonicAngleForce>
 <Angle class1="C" class2="C" class3="O" angle="2.094" k="669.44"/>
 <Angle class1="C" class2="C" class3="OH" angle="2.094" k="669.44"/>
```

```
 <Angle class1="CA" class2="C" class3="CA" angle="2.094" k="527.184"/>
 ...
</HarmonicAngleForce>
```

Every `<Angle>` tag defines a rule for creating harmonic angle interactions between triplets of atoms. Each tag may identify the atoms either by type (using the attributes `type1`, `type2`, ...) or by class (using the attributes `class1`, `class2`, ...). The force field identifies every set of three atoms in the system where the first is bonded to the second, and the second to the third. For each one, it searches for a rule whose atom types or atom classes match the three atoms. If it finds one, it calls `addAngle()` on the HarmonicAngleForce with the specified parameters. Otherwise, it ignores that set and continues. `angle` is the equilibrium angle in radians, and `k` is the spring constant in kJ/mol/radian^2.

5.2.5 <PeriodicTorsionForce>

To add a PeriodicTorsionForce to the System, include a tag that looks like this:

```
<PeriodicTorsionForce>
 <Proper class1="HC" class2="CT" class3="CT" class4="CT" periodicity1="3"
      phase1="0.0" k1="0.66944"/>
 <Proper class1="HC" class2="CT" class3="CT" class4="HC" periodicity1="3"
      phase1="0.0" k1="0.6276"/>
 ...
 <Improper class1="N" class2="C" class3="CT" class4="O" periodicity1="2"
      phase1="3.14159265359" k1="4.6024"/>
 <Improper class1="N" class2="C" class3="CT" class4="H" periodicity1="2"
      phase1="3.14159265359" k1="4.6024"/>
 ...
</PeriodicTorsionForce>
```

Every child tag defines a rule for creating periodic torsion interactions between sets of four atoms. Each tag may identify the atoms either by type (using the attributes `type1`, `type2`, ...) or by class (using the attributes `class1`, `class2`, ...).

The force field recognizes two different types of torsions: proper and improper. A proper torsion involves four atoms that are bonded in sequence: 1 to 2, 2 to 3, and 3 to 4. An improper torsion involves a central atom and three others that are bonded to it: atoms 2, 3, and 4 are all bonded to atom 1. The force field begins by identifying every set of atoms in the system of each of these types. For each one, it searches for a rule whose atom types or atom classes match the four atoms. If it finds one, it calls `addTorsion()` on the

PeriodicTorsionForce with the specified parameters. Otherwise, it ignores that set and continues. `periodicity1` is the periodicity of the torsion, `phase1` is the phase offset in radians, and `k1` is the force constant in kJ/mol.

Each torsion definition can specify multiple periodic torsion terms to add to its atoms. To add a second one, just add three more attributes: `periodicity2`, `phase2`, and `k2`. You can have as many terms as you want. Here is an example of a rule that adds three torsion terms to its atoms:

```
<Proper class1="CT" class2="CT" class3="CT" class4="CT" periodicity1="3"
      phase1="0.0" k1="0.75312" periodicity2="2" phase2="3.14159265359"
      k2="1.046" periodicity3="1" phase3="3.14159265359" k3="0.8368"/>
```

You can also use wildcards when defining torsions. To do this, simply leave the type or class name for an atom empty. That will cause it to match any atom. For example, the following definition will match any sequence of atoms where the second atom has class OS and the third has class P:

```
<Proper class1="" class2="OS" class3="P" class4="" periodicity1="3"
      phase1="0.0" k1="1.046"/>
```

5.2.6 <RBTorsionForce>

To add an RBTorsionForce to the System, include a tag that looks like this:

```
<RBTorsionForce>
 <Proper class1="CT" class2="CT" class3="OS" class4="CT" c0="2.439272"
      c1="4.807416" c2="-0.8368" c3="-6.409888" c4="0" c5="0" />
 <Proper class1="C" class2="N" class3="CT" class4="C" c0="10.46" c1="-
      3.34720" c2="-7.1128" c3="0" c4="0" c5="0" />
 ...
 <Improper class1="N" class2="C" class3="CT" class4="O" c0="0.8368" c1="0"
      c2="-2.76144" c3="0" c4="3.3472" c5="0" />
 <Improper class1="N" class2="C" class3="CT" class4="H" c0="29.288" c1="-
      8.368" c2="-20.92" c3="0" c4="0" c5="0" />
 ...
</RBTorsionForce>
```

Every child tag defines a rule for creating Ryckaert-Bellemans torsion interactions between sets of four atoms. Each tag may identify the atoms either by type (using the attributes `type1`, `type2`, ...) or by class (using the attributes `class1`, `class2`, ...).

The force field recognizes two different types of torsions: proper and improper. A proper torsion involves four atoms that are bonded in sequence: 1 to 2, 2 to 3, and 3 to 4. An improper torsion involves a central atom and three others that are bonded to it: atoms 2, 3, and 4 are all bonded to atom 1. The force field begins by identifying every set of atoms in the system of each of these types. For each one, it searches for a rule whose atom types or atom classes match the four atoms. If it finds one, it calls `addTorsion()` on the RBTorsionForce with the specified parameters. Otherwise, it ignores that set and continues. The attributes `c0` through `c5` are the coefficients of the terms in the Ryckaert-Bellemans force expression.

You can also use wildcards when defining torsions. To do this, simply leave the type or class name for an atom empty. That will cause it to match any atom. For example, the following definition will match any sequence of atoms where the second atom has class OS and the third has class P:

```
<Proper class1="" class2="OS" class3="P" class4="" c0="2.439272"
      c1="4.807416" c2="-0.8368" c3="-6.409888" c4="0" c5="0" />
```

5.2.7 <CMAPTorsionForce>

To add a CMAPTorsionForce to the System, include a tag that looks like this:

```
<CMAPTorsionForce>
 <Map>
  0.0 0.809 0.951 0.309
  -0.587 -1.0 -0.587 0.309
  0.951 0.809 0.0 -0.809
  -0.951 -0.309 0.587 1.0
 </Map>
 <Torsion map="0" class1="CT" class2="CT" class3="C" class4="N"
      class5="CT"/>
 <Torsion map="0" class1="N" class2="CT" class3="C" class4="N"
      class5="CT"/>
 ...
</CMAPTorsionForce>
```

Each `<Map>` tag defines an energy correction map. Its content is the list of energy values in kJ/mole, listed in the correct order for CMAPTorsionForce's `addMap()` method and separated by white space. See the API documentation for details. The size of the map is determined from the number of energy values.

Each <Torsion> tag defines a rule for creating CMAP torsion interactions between sets of five atoms. The tag may identify the atoms either by type (using the attributes type1, type2, ...) or by class (using the attributes class1, class2, ...). The force field identifies every set of five atoms that are bonded in sequence: 1 to 2, 2 to 3, 3 to 4, and 4 to 5. For each one, it searches for a rule whose atom types or atom classes match the five atoms. If it finds one, it calls addTorsion() on the CMAPTorsionForce with the specified parameters. Otherwise, it ignores that set and continues. The first torsion is defined by the sequence of atoms 1-2-3-4, and the second one by atoms 2-3-4-5. map is the index of the map to use, starting from 0, in the order they are listed in the file.

You can also use wildcards when defining torsions. To do this, simply leave the type or class name for an atom empty. That will cause it to match any atom. For example, the following definition will match any sequence of five atoms where the middle three have classes CT, C, and N respectively:

```
<Torsion map="0" class1="" class2="CT" class3="C" class4="N" class5=""/>
```

5.2.8 <NonbondedForce>

To add a NonbondedForce to the System, include a tag that looks like this:

```
<NonbondedForce coulomb14scale="0.833333" lj14scale="0.5">
 <Atom type="0" charge="-0.4157" sigma="0.32499" epsilon="0.71128"/>
 <Atom type="1" charge="0.2719" sigma="0.10690" epsilon="0.06568"/>
 <Atom type="2" charge="0.0337" sigma="0.33996" epsilon="0.45772"/>
 ...
</NonbondedForce>
```

The <NonbondedForce> tag has two attributes coulomb14scale and lj14scale that specify the scale factors between pairs of atoms separated by three bonds. After setting the nonbonded parameters for all atoms, the force field calls createExceptionsFromBonds() on the NonbondedForce, passing in these scale factors as arguments.

Each <Atom> tag specifies the nonbonded parameters for one atom type (specified with the type attribute) or atom class (specified with the class attribute). It is fine to mix these two methods, having some tags specify a type and others specify a class. However you do it, you

must make sure that a unique set of parameters is defined for every atom type. `charge` is measured in units of the proton charge, `sigma` is in nm, and `epsilon` is in kJ/mole.

5.2.9 <GBSAOBCForce>

To add a GBSAOBCForce to the System, include a tag that looks like this:

```
<GBSAOBCForce>
 <Atom type="0" charge="-0.4157" radius="0.1706" scale="0.79"/>
 <Atom type="1" charge="0.2719" radius="0.115" scale="0.85"/>
 <Atom type="2" charge="0.0337" radius="0.19" scale="0.72"/>
 ...
</GBSAOBCForce>
```

Each `<Atom>` tag specifies the OBC parameters for one atom type (specified with the `type` attribute) or atom class (specified with the `class` attribute). It is fine to mix these two methods, having some tags specify a type and others specify a class. However you do it, you must make sure that a unique set of parameters is defined for every atom type. `charge` is measured in units of the proton charge, `radius` is the GBSA radius in nm, and `scale` is the OBC scaling factor.

5.2.10 <CustomBondForce>

To add a CustomBondForce to the System, include a tag that looks like this:

```
<CustomBondForce energy="scale*k*(r-r0)^2">
 <GlobalParameter name="scale" defaultValue="0.5"/>
 <PerBondParameter name="k"/>
 <PerBondParameter name="r0"/>
 <Bond class1="OW" class2="HW" r0="0.09572" k="462750.4"/>
 <Bond class1="HW" class2="HW" r0="0.15136" k="462750.4"/>
 <Bond class1="C" class2="C" r0="0.1525" k="259408.0"/>
 ...
</CustomBondForce>
```

The energy expression for the CustomBondForce is specified by the `energy` attribute. This is a mathematical expression that gives the energy of each bond as a function of its length *r*. It also may depend on an arbitrary list of global or per-bond parameters. Use a `<GlobalParameter>` tag to define a global parameter, and a `<PerBondParameter>` tag to define a per-bond parameter.

Every <Bond> tag defines a rule for creating custom bond interactions between atoms. Each tag may identify the atoms either by type (using the attributes type1 and type2) or by class (using the attributes class1 and class2). For every pair of bonded atoms, the force field searches for a rule whose atom types or atom classes match the two atoms. If it finds one, it calls addBond() on the CustomBondForce. Otherwise, it ignores that pair and continues. The remaining attributes are the values to use for the per-bond parameters. All per-bond parameters must be specified for every <Bond> tag, and the attribute name must match the name of the parameter. For instance, if there is a per-bond parameter with the name "k", then every <Bond> tag must include an attribute called k.

5.2.11 <CustomAngleForce>

To add a CustomAngleForce to the System, include a tag that looks like this:

```
<CustomAngleForce energy="scale*k*(theta-theta0)^2">
 <GlobalParameter name="scale" defaultValue="0.5"/>
 <PerAngleParameter name="k"/>
 <PerAngleParameter name=" theta0"/>
 <Angle class1="HW" class2="OW" class3="HW" theta0="1.824218" k="836.8"/>
 <Angle class1="HW" class2="HW" class3="OW" theta0="2.229483" k="0.0"/>
 <Angle class1="C" class2="C" class3="O" theta0="2.094395" k="669.44"/>
 ...
</CustomAngleForce>
```

The energy expression for the CustomAngleForce is specified by the energy attribute. This is a mathematical expression that gives the energy of each angle as a function of the angle *theta*. It also may depend on an arbitrary list of global or per-angle parameters. Use a <GlobalParameter> tag to define a global parameter, and a <PerAngleParameter> tag to define a per-angle parameter.

Every <Angle> tag defines a rule for creating custom angle interactions between triplets of atoms. Each tag may identify the atoms either by type (using the attributes type1, type2, ...) or by class (using the attributes class1, class2, ...). The force field identifies every set of three atoms in the system where the first is bonded to the second, and the second to the third. For each one, it searches for a rule whose atom types or atom classes match the three atoms. If it finds one, it calls addAngle() on the CustomAngleForce. Otherwise, it ignores that set and continues. The remaining attributes are the values to use for the per-angle

parameters. All per-angle parameters must be specified for every `<Angle>` tag, and the attribute name must match the name of the parameter. For instance, if there is a per-angle parameter with the name "k", then every `<Angle>` tag must include an attribute called `k`.

5.2.12 <CustomTorsionForce>

To add a CustomTorsionForce to the System, include a tag that looks like this:

```
<CustomTorsionForce energy="scale*k*(1+cos(per*theta-phase))">
 <GlobalParameter name="scale" defaultValue="1"/>
 <PerTorsionParameter name="k"/>
 <PerTorsionParameter name="per"/>
 <PerTorsionParameter name="phase"/>
 <Proper class1="HC" class2="CT" class3="CT" class4="CT" per="3"
      phase="0.0" k="0.66944"/>
 <Proper class1="HC" class2="CT" class3="CT" class4="HC" per="3"
      phase="0.0" k="0.6276"/>
 ...
 <Improper class1="N" class2="C" class3="CT" class4="O" per="2"
      phase="3.14159265359" k="4.6024"/>
 <Improper class1="N" class2="C" class3="CT" class4="H" per="2"
      phase="3.14159265359" k="4.6024"/>
 ...
</CustomTorsionForce>
```

The energy expression for the CustomTorsionForce is specified by the `energy` attribute. This is a mathematical expression that gives the energy of each torsion as a function of the angle *theta*. It also may depend on an arbitrary list of global or per-torsion parameters. Use a `<GlobalParameter>` tag to define a global parameter, and a `<PerTorsionParameter>` tag to define a per-torsion parameter.

Every child tag defines a rule for creating custom torsion interactions between sets of four atoms. Each tag may identify the atoms either by type (using the attributes `type1`, `type2`, ...) or by class (using the attributes `class1`, `class2`, ...).

The force field recognizes two different types of torsions: proper and improper. A proper torsion involves four atoms that are bonded in sequence: 1 to 2, 2 to 3, and 3 to 4. An improper torsion involves a central atom and three others that are bonded to it: atoms 2, 3, and 4 are all bonded to atom 1. The force field begins by identifying every set of atoms in the system of each of these types. For each one, it searches for a rule whose atom types or atom

classes match the four atoms. If it finds one, it calls `addTorsion()` on the CustomTorsionForce with the specified parameters. Otherwise, it ignores that set and continues. The remaining attributes are the values to use for the per-torsion parameters. Every `<Torsion>` tag must include one attribute for every per-torsion parameter, and the attribute name must match the name of the parameter.

You can also use wildcards when defining torsions. To do this, simply leave the type or class name for an atom empty. That will cause it to match any atom. For example, the following definition will match any sequence of atoms where the second atom has class OS and the third has class P:

```
<Proper class1="" class2="OS" class3="P" class4="" per="3" phase="0.0"
      k="0.66944"/>
```

5.2.13 <CustomGBForce>

To add a CustomGBForce to the System, include a tag that looks like this:

```
<CustomGBForce>
 <GlobalParameter name="solventDielectric" defaultValue="78.3"/>
 <GlobalParameter name="soluteDielectric" defaultValue="1"/>
 <PerParticleParameter name="charge"/>
 <PerParticleParameter name="radius"/>
 <PerParticleParameter name="scale"/>
 <ComputedValue name="I" type="ParticlePairNoExclusions">
  step(r+sr2-or1)*0.5*(1/L-1/U+0.25*(1/U^2-1/L^2)*(r-
      sr2*sr2/r)+0.5*log(L/U)/r+C); U=r+sr2; C=2*(1/or1-1/L)*step(sr2-r-
      or1); L=max(or1, D); D=abs(r-sr2); sr2 = scale2*or2; or1 = radius1-
      0.009; or2 = radius2-0.009
 </ComputedValue>
 <ComputedValue name="B" type="SingleParticle">
  1/(1/or-tanh(1*psi-0.8*psi^2+4.85*psi^3)/radius); psi=I*or; or=radius-
      0.009
 </ComputedValue>
 <EnergyTerm type="SingleParticle">
  28.3919551*(radius+0.14)^2*(radius/B)^6-
      0.5*138.935456*(1/soluteDielectric-1/solventDielectric)*charge^2/B
 </EnergyTerm>
 <EnergyTerm type="ParticlePair">
  -138.935456*(1/soluteDielectric-1/solventDielectric)*charge1*charge2/f;
      f=sqrt(r^2+B1*B2*exp(-r^2/(4*B1*B2)))
 </EnergyTerm>
 <Atom type="0" charge="-0.4157" radius="0.1706" scale="0.79"/>
 <Atom type="1" charge="0.2719" radius="0.115" scale="0.85"/>
 <Atom type="2" charge="0.0337" radius="0.19" scale="0.72"/>
 ...
```

```
</CustomGBForce>
```

The above (rather complicated) example defines a generalized Born model that is equivalent to GBSAOBCForce. The definition consists of a set of computed values (defined by `<ComputedValue>` tags) and energy terms (defined by `<EnergyTerm>` tags), each of which is evaluated according to a mathematical expression. See the API documentation for details.

The expressions may depend on an arbitrary list of global or per-atom parameters. Use a `<GlobalParameter>` tag to define a global parameter, and a `<PerAtomParameter>` tag to define a per-atom parameter.

Each `<Atom>` tag specifies the parameters for one atom type (specified with the `type` attribute) or atom class (specified with the `class` attribute). It is fine to mix these two methods, having some tags specify a type and others specify a class. However you do it, you must make sure that a unique set of parameters is defined for every atom type. The remaining attributes are the values to use for the per-atom parameters. All per-atom parameters must be specified for every `<Atom>` tag, and the attribute name must match the name of the parameter. For instance, if there is a per-atom parameter with the name "radius", then every `<Atom>` tag must include an attribute called `radius`.

CustomGBForce also allows you to define tabulated functions. To define a function, include a `<Function>` tag inside the `<CustomGBForce>` tag:

```
<Function name="myfn" min="-5" max="5">
   0.983674857694 -0.980096396266 -0.975743130031 -0.970451936613 -
   0.964027580076 -0.956237458128 -0.946806012846 -0.935409070603 -
   0.921668554406 -0.905148253645 -0.885351648202 -0.861723159313 -
   0.833654607012 -0.800499021761 -0.761594155956 -0.716297870199 -
   0.664036770268 -0.604367777117 -0.537049566998 -0.46211715726 -
   0.379948962255 -0.291312612452 -0.197375320225 -0.099667994625 0.0
   0.099667994625 0.197375320225 0.291312612452 0.379948962255
   0.46211715726 0.537049566998 0.604367777117 0.664036770268
   0.716297870199 0.761594155956 0.800499021761 0.833654607012
   0.861723159313 0.885351648202 0.905148253645 0.921668554406
   0.935409070603 0.946806012846 0.956237458128 0.964027580076
   0.970451936613 0.975743130031 0.980096396266 0.983674857694
   0.986614298151 0.989027402201
</Function>
```

The tag's attributes define the name of the function and the range of values for which it is defined. The tabulated values are listed inside the body of the tag, with successive values separated by white space. Again, see the API documentation for more details.

5.2.14 Writing Custom Expressions

The custom forces described in this chapter involve user defined algebraic expressions. These expressions are specified as character strings, and may involve a variety of standard operators and mathematical functions.

The following operators are supported: + (add), - (subtract), * (multiply), / (divide), and ^ (power). Parentheses "(" and ")" may be used for grouping.

The following standard functions are supported: sqrt, exp, log, sin, cos, sec, csc, tan, cot, asin, acos, atan, sinh, cosh, tanh, erf, erfc, min, max, abs, step. step(x) = 0 if x < 0, 1 otherwise. Some custom forces allow additional functions to be defined from tabulated values.

Numbers may be given in either decimal or exponential form. All of the following are valid numbers: 5, -3.1, 1e6, and 3.12e-2.

The variables that may appear in expressions are specified in the API documentation for each force class. In addition, an expression may be followed by definitions for intermediate values that appear in the expression. A semicolon ";" is used as a delimiter between value definitions. For example, the expression

```
a^2+a*b+b^2; a=a1+a2; b=b1+b2
```

is exactly equivalent to

```
(a1+a2)^2+(a1+a2)*(b1+b2)+(b1+b2)^2
```

The definition of an intermediate value may itself involve other intermediate values. All uses of a value must appear *before* that value's definition.

5.3 Using Multiple Files

If multiple XML files are specified when a ForceField is created, their definitions are combined as follows.

- A file may refer to atom types and classes that it defines, as well as those defined in previous files. It may not refer to ones defined in later files. This means that the order in which files are listed when calling the ForceField constructor is potentially significant.
- Forces that involve per-atom parameters (such as NonbondedForce or GBSAOBCForce) require parameter values to be defined for every atom type. It does not matter which file those types are defined in. For example, files that define explicit water models generally define a small number of atom types, as well as nonbonded parameters for those types. In contrast, files that define implicit solvent models do not define any new atom types, but provide parameters for all the atom types that were defined in the main force field file.
- For other forces, the files are effectively independent. For example, if two files each include a `<HarmonicBondForce>` tag, bonds will be created based on the rules in the first file, and then more bonds will be created based on the rules in the second file. This means you could potentially end up with multiple bonds between a single pair of atoms.

5.4 Extending ForceField

The ForceField class is designed to be modular and extensible. This means you can add support for entirely new force types, such as ones implemented with plugins.

For every force class, there is a "generator" class that parses the corresponding XML tag, then creates Force objects and adds them to the System. ForceField maintains a map of tag names to generator classes. When a ForceField is created, it scans through the XML files, looks up the generator class for each tag, and asks that class to create a generator object

based on it. Then, when you call `createSystem()`, it loops over each of its generators and asks each one to create its Force object. Adding a new Force type therefore is simply a matter of creating a new generator class and adding it to ForceField's map.

The generator class must define two methods. First, it needs a static method with the following signature to parse the XML tag and create the generator:

```
@staticmethod
def parseElement(element, forcefield):
```

`element` is the XML tag (an xml.etree.ElementTree.Element object) and `forcefield` is the ForceField being created. This method should create a generator and add it to the ForceField:

```
generator = MyForceGenerator()
forcefield._forces.append(generator)
```

It then should parse the information contained in the XML tag and configure the generator based on it.

Second, it must define a method with the following signature:

```
def createForce(self, system, data, nonbondedMethod, nonbondedCutoff,
      args):
```

When `createSystem()` is called on the ForceField, it first creates the System object, then loops over each of its generators and calls `createForce()` on each one. This method should create the Force object and add it to the System. `data` is a ForceField._SystemData object containing information about the System being created (atom types, bonds, angles, etc.), `system` is the System object, and the remaining arguments are values that were passed to `createSystem()`. To get a better idea of how this works, look at the existing generator classes in forcefield.py.

Finally, you need to register your class by adding it to ForceField's map:

```
forcefield.parsers['MyForce'] = MyForceGenerator.parseElement
```

The key is the XML tag name, and the value is the static method to use for parsing it.

Now you can simply create a ForceField object as usual. If an XML file contains a `<MyForce>` tag, it will be recognized and processed correctly.

6 Bibliography

1. Kollman, P. A.; Dixon, R.; Cornell, W.; Fox, T.; Chipot, C.; Pohorille, A., Computer Simulation of Biomolecular Systems. In Wilkinson, A.; Weiner, P.; van Gunsteren, W. F., Eds. Elsevier: 1997; Vol. 3, pp 83-96.
2. Wang, J.; Cieplak, P.; Kollman, P. A., How well does a restrained electrostatic potential (RESP) model perform in calculating conformational energies of organic and biological molecules? *Journal of Computational Chemistry* **2000,** 21, 1049-1074.
3. Hornak, V.; Abel, R.; Okur, A.; Strockbine, B.; Roitberg, A.; Simmerling, C., Comparison of multiple Amber force fields and development of improved protein backbone parameters. *Proteins* **2006,** 65, 712-725.
4. Lindorff-Larsen, K.; Piana, S.; Palmo, K.; Maragakis, P.; Klepeis, J.; Dror, R. O.; Shaw, D. E., Improved side-chain torsion potentials for the Amber ff99SB protein force field. *Proteins* **2010,** 78, 1950-1958.
5. Li, D. W.; Bru□schweiler, R., NMR-based protein potentials. *Angewandte Chemie International Edition* **2010,** 49, 6778-6780.
6. Duan, Y. W., C.; Chowdhury, S.; Lee, M. C.; Xiong, G.; Zhang, W.; Yang, R.; Cieplak, P.; Luo, R.; Lee, T., A point-charge force field for molecular mechanics simulations of proteins based on condensed-phase quantum mechanical calculations. *Journal of Computational Chemistry* **2003,** 24, 1999-2012.
7. Ren, P.; Ponder, J. W., A Consistent Treatment of Inter- and Intramolecular Polarization in Molecular Mechanics Calculations. *Journal of Computational Chemistry* **2002,** 23, 1497-1506.
8. Jorgensen, W. L.; Chandrasekhar, J.; Madura, J. D.; Impey, R. W.; Klein, M. L., Comparison of simple potential functions for simulating liquid water. *Journal of Chemical Physics* **1983,** 79, 926-935.
9. Berendsen, H. J. C.; Grigera, J. R.; Straatsma, T. P., The missing term in effective pair potentials. *Journal of Physical Chemistry* **1987,** 91, 6269-6271.
10. Ren, P.; Ponder, J. W., Polarizable Atomic Multipole Water Model for Molecular Mechanics Simulation. *Journal of Physical Chemistry B* **2003,** 107, 5933-5947.
11. Onufriev, A.; Bashford, D.; Case, D. A., Exploring protein native states and large-scale conformational changes with a modified generalized born model. *Proteins* **2004,** 55, (22), 383-394.
12. Schnieders, M. J.; Ponder, J. W., Polarizable Atomic Multipole Solutes in a Generalized Kirkwood Continuum. *Journal of Chemical Theory and Computation* **2007,** 3, 2083-2097.