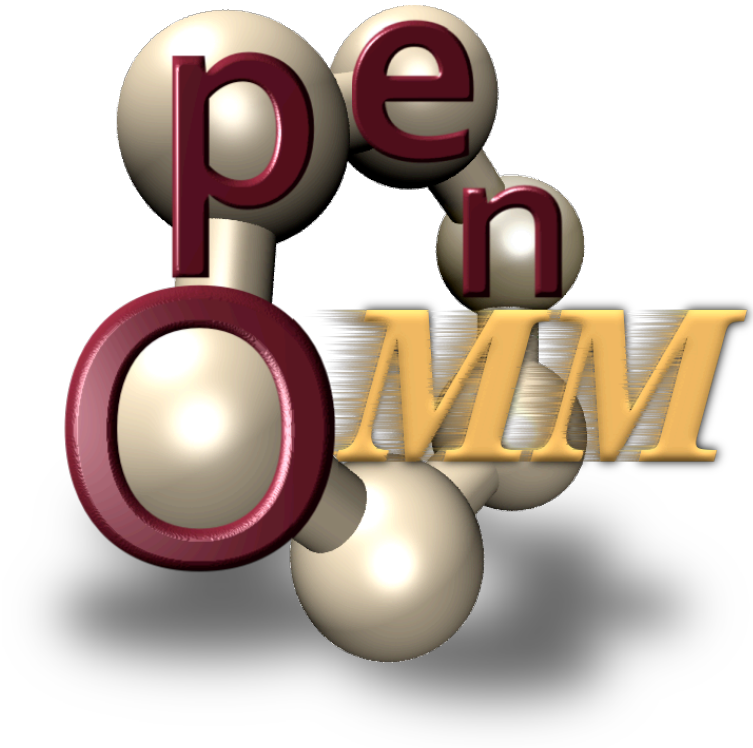




# Users Manual and Theory Guide

Release 4.0

November 28, 2011

# OpenMM Users Manual and Theory Guide


**Authors**

Kyle Beauchamp

Christopher Bruns

Peter Eastman

Mark Friedrichs

Joy P. Ku

Tom Markland

Vijay Pande

Randy Radmer

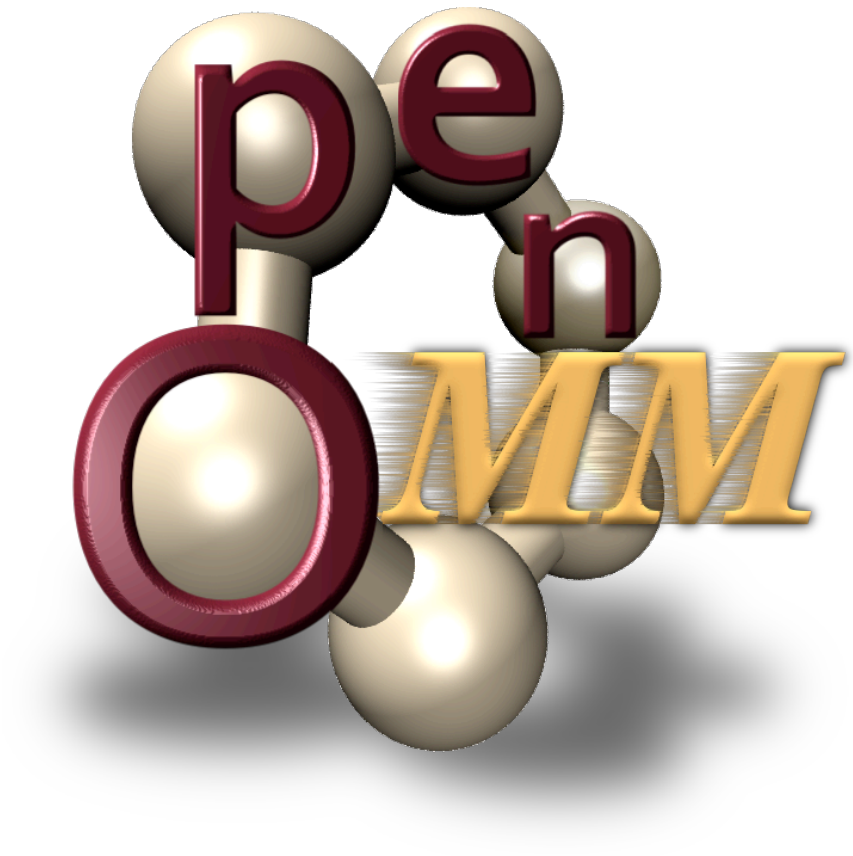
Michael Sherman

## 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




# Part I

# Users Manual

# 1 Introduction

## 1.1 What Is OpenMM?

OpenMM is an API for executing molecular dynamics simulations on high performance computer architectures. Examples of the sorts of architectures it is intended to support include:

- Highly parallel systems with large numbers of CPU cores
- Graphics processing units (GPUs)
- Clusters of computers communicating over a network

OpenMM consists of two parts. First, there is a set of libraries for performing many types of computations needed for molecular simulations: force evaluation, numerical integration, energy minimization, etc. These libraries provide an interface targeted at developers of simulation software, allowing them to easily add simulation features to their programs.

Second, there is an "application layer", a set of Python libraries providing a high level interface for running simulations. This layer is targeted at computational biologists or other people who want to run simulations, and who may or may not be programmers.

This guide describes the computational libraries. If you are only interested in running simulations, not in writing software, much of it will not be relevant to you. The application layer is described in a separate OpenMM Application Guide.

## 1.2 OpenMM Version 4.0

Most parts of the current release are stable and suitable for production use. There are a few exceptions to that. The GBVIForce class should be considered beta quality, since it has not yet been extensively tested. Also, the CustomGBForce and CustomHbondForce classes are

still under development. They work correctly, but they have only been implemented on the Reference and OpenCL Platforms, and their APIs might change in the future.

OpenMM is being actively developed, and although we expect the API to be relatively stable for the foreseeable future, it is possible that some small changes will occur. Users should expect that programs written to use this release may require modifications to work with future versions.

We also are open to other possible changes. All comments and suggestions for ways to make OpenMM a better, more useful toolkit are welcome. Email us at openmm-team@simtk.org.

## 1.3 Using this Manual

### 1.3.1 Organization of this document

This manual is divided into two distinct sections:

- **Users Manual** – The goal of this section is to present a high-level overview of OpenMM and provide instructions for using the OpenMM API and creating plug-ins to add functionality to OpenMM.
- **Theory Manual** – This section describes the mathematical theory behind the functions available in OpenMM. As appropriate, specific tips are given on how to use the function to produce accurate, fast results.

### 1.3.2 How to get started

We have provided a number of files that make it easy to get started with OpenMM. Pre-compiled binaries are provided for quickly getting OpenMM onto your computer (See Chapter 3 for set-up instructions). We recommend that you then compile and run some of the tutorial examples, described in Chapter 5. These highlight key functions within OpenMM and teach you the basic programming concepts for using OpenMM. Once you are ready to begin integrating OpenMM into a specific software package, read through Chapter 8 to see how other software developers have done this.

## 1.4 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, several mailing lists with archives and tutorial slides and videos.

## 1.5 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.

## 1.6 Acknowledgements and License

OpenMM was developed by Simbios, the NIH National Center for Physics-Based Simulation of Biological Structures at Stanford, funded under the NIH Roadmap for Medical Research, grant U54 GM072970. See https://simtk.org.

OpenMM also uses several pieces of code that were written by other people and are covered by other licenses. All of these licenses are similar in their terms to the MIT license, and do not significantly restrict how OpenMM can be used.

All of these licenses may be found in the "licenses" directory included with OpenMM.

# 2 OpenMM Design and API Overview

## 2.1 Design Principles

The design of the OpenMM API is guided by the following principles.

1. The API must support efficient implementations on a variety of architectures.

The most important consequence of this goal is that the API cannot provide direct access to state information (particle positions, velocities, etc.) at all times. On some architectures, accessing this information is expensive. With a GPU, for example, it will be stored in video memory, and must be transferred to main memory before outside code can access it. On a distributed architecture, it might not even be present on the local computer. OpenMM therefore only allows state information to be accessed in bulk, with the understanding that doing so may be a slow operation.

2. The API should be easy to understand and easy to use.

This seems obvious, but it is worth stating as an explicit goal. We are creating OpenMM with the hope that many other people will use it. To achieve that goal, it should be possible for someone to learn it without an enormous amount of effort. An equally important aspect of being "easy to use" is being easy to use *correctly*. A well designed API should minimize the opportunities for a programmer to make mistakes. For both of these reasons, clarity and simplicity are essential.

3. It should be modular and extensible.

We cannot hope to provide every feature any user will ever want. For that reason, it is important that OpenMM be easy to extend. If a user wants to add a new molecular force field, a new thermostat algorithm, or a new hardware platform, the API should make that easy to do.

4. The API should be hardware independent.

Computer architectures are changing rapidly, and it is impossible to predict what hardware platforms might be important to support in the future. One of the goals of OpenMM is to separate the API from the hardware. The developers of a simulation application should be able to write their code once, and have it automatically take advantage of any architecture that OpenMM supports, even architectures that do not yet exist when they write it.

## 2.2 Choice of Language

Molecular modeling and simulation tools are written in a variety of languages: C, C++, Fortran, Python, TCL, etc. It is important that any of these tools be able to use OpenMM. There are two possible approaches to achieving this goal.

One option is to provide a separate version of the API for each language. These could be created by hand, or generated automatically with a wrapper generator such as SWIG. This would require the API to use only "lowest common denominator" features that can be reasonably supported in all languages. For example, an object oriented API would not be an option, since it could not be cleanly expressed in C or Fortran.

The other option is to provide a single version of the API written in a single language. This would permit a cleaner, simpler API, but also restrict the languages it could be directly called from. For example, a C++ API could not be invoked directly from Fortran or Python.

We have chosen to use a hybrid of these two approaches. OpenMM is based on an object oriented C++ API. This is the primary way to invoke OpenMM, and is the only API that fully exposes all features of the library. We believe this will ultimately produce the best, easiest to use API and create the least work for developers who use it. It does require that any code

which directly invokes this API must itself be written in C++, but this should not be a significant burden. Regardless of what language we had chosen, developers would need to write a thin layer for translating between their own application's data model and OpenMM. That layer is the only part which needs to be written in C++.

In addition, we have created wrapper APIs that allow OpenMM to be invoked from other languages. The current release includes wrappers for C, Fortran, and Python. These wrappers support as many features as reasonably possible given the constraints of the particular languages, but some features cannot be fully supported. In particular, writing plug-ins to extend the OpenMM API can only be done in C++.

We are also aware that some features of C++ can easily lead to compatibility and portability problems, and we have tried to avoid those features. In particular, we make minimal use of templates and avoid multiple inheritance altogether. Our goal is to eventually support OpenMM on all major compilers and operating systems.

## 2.3 Architectural Overview

OpenMM is based on a layered architecture, as shown in the following diagram:

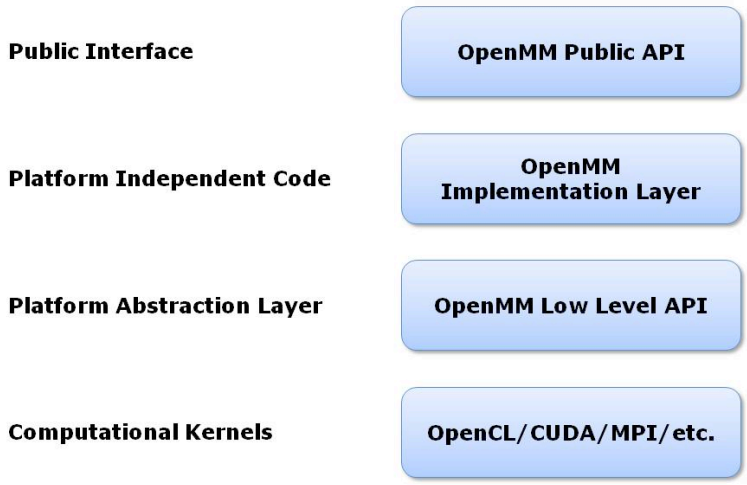



**Figure 2.1: OpenMM architecture**

At the highest level is the OpenMM public API. This is the API developers program against when using OpenMM within their own applications. It is designed to be simple, easy to understand, and completely platform independent. This is the only layer that many users will ever need to look at.

The public API is implemented by a layer of platform independent code. It serves as the interface to the lower level, platform specific code. Most users will never need to look at it.

The next level down is the OpenMM Low Level API (OLLA). This acts as an abstraction layer to hide the details of each hardware platform. It consists of a set of C++ interfaces that each platform must implement. Users who want to extend OpenMM will need to write classes at the OLLA level. Note the different roles played by the public API and the low level API: the public API defines an interface for users to invoke in their own code, while OLLA defines an interface that users must implement, and that is invoked by the OpenMM implementation layer.

At the lowest level is hardware specific code that actually performs computations. This code may be written in any language and use any technologies that are appropriate. For example, code for GPUs will be written in stream processing languages such as OpenCL or CUDA, code written to run on clusters will use MPI or other distributed computing tools, code written for multicore processors will use threading tools such as Pthreads or OpenMP, etc. OpenMM sets no restrictions on how these computational kernels are written. As long as they are wrapped in the appropriate OLLA interfaces, OpenMM can use them.

## 2.4 The OpenMM Public API

The public API is based on a small number of classes:

**System**: A System specifies generic properties of the system to be simulated: the number of particles it contains, the mass of each one, the size of the periodic box, etc. The interactions between the particles are specified through a set of Force objects (see below) that are added to the System. Force field specific parameters, such as particle charges, are not direct

properties of the System. They are properties of the Force objects contained within the System.

**Force**: The Force objects added to a System define the behavior of the particles. Force is an abstract class; subclasses implement specific behaviors. The Force class is actually slightly more general than its name suggests. A Force can, indeed, apply forces to particles, but it can also directly modify particle positions and velocities in arbitrary ways. Some thermostats and barostats, for example, can be implemented as Force classes. Examples of Force subclasses in OpenMM 4.0 include HarmonicBondForce, NonbondedForce, and MonteCarloBarostat.

**Context**: This stores all of the state information for a simulation: particle positions and velocities, as well as arbitrary parameters defined by the Forces in the System. It is possible to create multiple Contexts for a single System, and thus have multiple simulations of that System in progress at the same time.

**Integrator**: This implements an algorithm for advancing the simulation through time. It is an abstract class; subclasses implement specific algorithms. Examples of Integrator subclasses in OpenMM 4.0 include LangevinIntegrator, VerletIntegrator, and BrownianIntegrator.

**State**: A State stores a snapshot of the simulation at a particular point in time. It is created by calling a method on a Context. As discussed earlier, this is a potentially expensive operation. This is the only way to query the values of state variables, such as particle positions and velocities; Context does not provide methods for accessing them directly.

Here is an example of what the source code to create a System and run a simulation might look like:

```
System system;
for (int i = 0; i < numParticles; ++i)
    system.addParticle(particle[i].mass);
HarmonicBondForce* bonds = new HarmonicBondForce();
system.addForce(bonds);
```

```
for (int i = 0; i < numBonds; ++i)
    bonds->addBond(bond[i].particle1, bond[i].particle2,
        bond[i].length, bond[i].k);
HarmonicAngleForce* angles = new HarmonicAngleForce();
system.addForce(angles);
for (int i = 0; i < numAngles; ++i)
    angles->addAngle(angle[i].particle1, angle[i].particle2,
        angle[i].particle3, angle[i].angle, angle[i].k);
// ...create and initialize other force field terms in the same way
LangevinIntegrator integrator(temperature, friction, stepSize);
Context context(system, integrator);
context.setPositions(initialPositions);
context.setVelocities(initialVelocities);
integrator.step(10000);
```

We create a System, add various Forces to it, and set parameters on both the System and the Forces. We then create a LangevinIntegrator, initialize a Context in which to run a simulation, and instruct the Integrator to advance the simulation for 10,000 time steps.

## 2.5 The OpenMM Low Level API

The OpenMM Low Level API (OLLA) defines a set of interfaces that users must implement in their own code if they want to extend OpenMM, such as to create a new Force subclass or support a new hardware platform. It is based on the concept of "kernels" that define particular computations to be performed.

More specifically, there is an abstract class called **KernelImpl**. Instances of this class (or rather, of its subclasses) are created by **KernelFactory** objects. These classes provide the concrete implementations of kernels for a particular platform. For example, to perform calculations on a GPU, one would create one or more KernelImpl subclasses that implemented the computations with GPU kernels, and one or more KernelFactory subclasses to instantiate the KernelImpl objects.

All of these objects are encapsulated in a single object that extends **Platform**. KernelFactory objects are registered with the Platform to be used for creating specific named kernels. The

choice of what implementation to use (a GPU implementation, a multithreaded CPU implementation, an MPI-based distributed implementation, etc.) consists entirely of choosing what Platform to use.

As discussed so far, the low level API is not in any way specific to molecular simulation; it is a fairly generic computational API. In addition to defining the generic classes, OpenMM also defines abstract subclasses of KernelImpl corresponding to specific calculations. For example, there is a class called CalcHarmonicBondForceKernel to implement HarmonicBondForce and a class called IntegrateLangevinStepKernel to implement LangevinIntegrator. It is these classes for which each Platform must provide a concrete subclass.

This architecture is designed to allow easy extensibility. To support a new hardware platform, for example, you create concrete subclasses of all the abstract kernel classes, then create appropriate factories and a Platform subclass to bind everything together. Any program that uses OpenMM can then use your implementation simply by specifying your Platform subclass as the platform to use.

Alternatively, you might want to create a new Force subclass to implement a new type of interaction. To do this, define an abstract KernelImpl subclass corresponding to the new force, then write the Force class to use it. Any Platform can support the new Force by providing a concrete implementation of your KernelImpl subclass. Furthermore, you can easily provide that implementation yourself, even for existing Platforms created by other people. Simply create a new KernelFactory subclass for your kernel and register it with the Platform object. The goal is to have a completely modular system. Each module, which might be distributed as an independent library, can either add new features to existing platforms or support existing features on new platforms.

In fact, there is nothing “special” about the kernel classes defined by OpenMM. They are simply KernelImpl subclasses that happen to be used by Forces and Integrators that happen to be bundled with OpenMM. They are treated exactly like any other KernelImpl, including the ones you define yourself.

It is important to understand that OLLA defines an interface, not an implementation. It would be easy to assume a one-to-one correspondence between KernelImpl objects and the pieces of code that actually perform calculations, but that need not be the case. For a GPU implementation, for example, a single KernelImpl might invoke several GPU kernels. Alternatively, a single GPU kernel might perform the calculations of several KernelImpl subclasses.

## 2.6 Platforms

This release of OpenMM contains the following Platform subclasses:

**ReferencePlatform.** This is designed to serve as reference code for writing other platforms. It is written with simplicity and clarity in mind, not performance.
**CudaPlatform.** This platform is implemented using the CUDA language, and performs calculations on Nvidia GPUs.
**OpenCLPlatform.** This platform is implemented using the OpenCL language, and performs calculations on a variety of types of GPUs and CPUs.

The choice of which platform to use for a simulation depends on various factors:

1. The Reference platform is much slower than the others, and therefore is rarely used for production simulations. It does have one advantage, however: it is the only platform that does force and energy computations in double precision. For most applications, single precision is entirely sufficient, but for the rare situations when higher accuracy is required, the Reference platform is the only option.
2. The CUDA platform can only be used with NVIDIA GPUs. For using an AMD GPU or for running on a CPU, use the OpenCL platform.
3. When running on an NVIDIA GPU, many factors affect which platform is faster, CUDA or OpenCL. Examples include the model of GPU (OpenCL tends to be faster on recent GPUs, while CUDA tends to be faster on older ones), whether the simulation includes water (OpenCL tends to be faster for explicit solvent, while CUDA tends to be faster for implicit solvent), whether any custom forces are used

(these are much faster with OpenCL than with CUDA), etc. You should try both platforms and see which is faster for your simulation.

4. The OpenCL platform uses memory more efficiently than the CUDA platform when simulating large systems. For this reason, the OpenCL platform can simulate larger systems than the CUDA platform can.
5. CustomGBForce and CustomHbondForce only work with the OpenCL platform, not with the CUDA platform.
6. The AMOEBA force field only works with the CUDA platform, not with the OpenCL platform.
7. GBVIForce only works with the CUDA platform, not with the OpenCL platform.

# 3 Instructions for Pre-Compiled OpenMM Binaries and GPU Software

OpenMM provides pre-compiled binaries for a number of platforms:

- Windows (Visual Studio 9 and 10)
- Linux (32 and 64 bit)
- Mac OS X (10.6 or later; it may also work on 10.5, but it has not been tested and the OpenCL platform will not be available)

Source code is also available. Instructions for compiling OpenMM from source code are provided in Chapter 4.

## 3.1 Prerequisites

To run OpenMM and the provided test examples, you will need:

- A C++ compiler
  - gcc on Mac/Linux - We have tested with various gcc versions between 4.0 and 4.4. If you are using Mac OS X, gcc is included with Apple's Xcode developer tools.
  - Visual Studio 9 or 10 on Windows - You can download a free version of Visual C++ 2010 Express Edition (similar to Visual Studio 10) from http://www.microsoft.com/express/vc/. (This is not required if you only plan to use Python.)

- Python 2.6 or 2.7.
- OpenMM pre-compiled binaries for your platform (see Section 3.3 below)

To take advantage of the GPU-accelerated molecular dynamics, you must have a supported GPU. You will also need to have the special programming language(s) used for your particular GPU (see Section 3.4).

## 3.2 Quick Instructions

Below is a quick-start guide to getting OpenMM and running the provided test examples. More details follow in the subsequent sections.

1. Download OpenMM binaries from http://simtk.org/home/openmm. Extract the files from the zip archive.

2. Install OpenMM.
    a. On Windows, extract the files and save them to C:\ProgramFiles\OpenMM. Double click the Python API Installer to install the Python components.
    b. On Mac or Linux, execute the install.sh script (e.g. "`sudo ./install.sh`"). It will prompt you to select an install location and to locate your Python executable.

3. Set path variables for the lib directory within the openmm or OpenMM folder – See Section 3.3 for more detailed instructions.

4. Install GPU software, if applicable – See Section 3.4 for more detailed instructions.

5. (Optional) Build and run the HelloArgon program to test the installation – see Section 5.1 for more detailed instructions.
    - On Linux/Mac OS X, type `make`. Then, run the HelloArgon program.
    - On Windows, double-click on HelloArgon.sln, located in the VisualStudio2005 folder. Make sure the "Solution Configuration" in Visual Studio is set to "Release"; due to incompatibilities among Visual Studio

versions, we do not provide pre-compiled debug binaries. Build the program (Select Debug -> Start Without Debugging).

## 3.3 Installing OpenMM

The pre-compiled OpenMM libraries can be obtained from http://simtk.org/home/openmm. Click on "Downloads." Under the list of "Pre-compiled binaries," select the file that corresponds to your platform.

### 3.3.1 Windows

Extract all files from the zip file and place them in C:\Program Files\OpenMM. Programs that use OpenMM should include C:\Program Files\OpenMM\lib in the PATH. To set the PATH permanently:

1. Click on Start -> Control Panel -> System (On Windows 7, select Start -> Control Panel -> System and Security -> System)
2. Click on the "Advanced" tab or the "Advanced system settings" link
3. Click "Environment Variables"
4. Under "System variables," select the line for "Path" and click "Edit..."
5. Add C:\Program Files\OpenMM\lib to the "Variable value"
6. If you install OpenMM to a location other than C:\Program Files, you will also need to set the variable OPENMM_PLUGIN_DIR. Under "System variables," click the "New" button. Set the "Variable name" to OPENMM_PLUGIN_DIR. Set the "Variable value" to the path for the plug-ins directory (default: C:\Program Files\OpenMM\lib\plugins). Click "OK."
7. Click "OK"

### 3.3.2 Linux

The install script copies the files to the directory you select, which is /usr/local/openmm by default. Programs that use OpenMM should include /usr/local/openmm/lib in the LD_LIBRARY_PATH. To set the LD_LIBRARY_PATH, type:

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

This sets the LD_LIBRARY_PATH only for the terminal you are in. To set it permanently, you will need to add it to, for example, your .bash_profile if you use the BASH shell.

If you choose to install OpenMM some place other than the default location (/usr/local/openmm), you will need to also set the OPENMM_PLUGIN_DIR to the openmm/lib/plugins directory. For example:

```
export OPENMM_PLUGIN_DIR=/home/<user_name>/openmm/lib/plugins
```

Again, to set the variable permanently, you will need to add it to, for example, your .bash_profile if you use the BASH shell.

### 3.3.3 Mac OS X

The install script copies the files to the directory you select, which is /usr/local/openmm by default. Programs that use OpenMM should include /usr/local/openmm/lib in the DYLD_LIBRARY_PATH. To set the DYLD_LIBRARY_PATH, type:

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

This sets the DYLD_LIBRARY_PATH only for the terminal you are in. To set it permanently, you will need to add it to your .bash_profile.

If you choose to install OpenMM some place other than the default location (/usr/local/openmm), you will need to also set the OPENMM_PLUGIN_DIR to the openmm/lib/plugins directory. For example:

```
export OPENMM_PLUGIN_DIR=/Users/<user_name>/openmm/lib/plugins
```

Again, to set the variable permanently, you will need to add it to, for example, your .bash_profile if you use the BASH shell.

## 3.4 Installing GPU Software

To take advantage of the GPU acceleration provided via OpenMM, your computer needs to be equipped with one of the supported GPU cards:

Supported NVIDIA GPUs (CUDA or OpenCL):
http://www.nvidia.com/object/cuda_learn_products.html

Supported AMD GPUs (OpenCL):
http://developer.amd.com/sdks/AMDAPPSDK/pages/DriverCompatibility.aspx

You also need to install CUDA (for NVIDIA GPUs), or OpenCL (for AMD GPUs) and test it before running OpenMM and the provided examples.

### 3.4.1 Installing CUDA for NVIDIA GPUs

For NVIDIA GPUs, you need to have CUDA version 4.0 or later installed to get the GPU acceleration. It is recommended that you test your installation before trying to run OpenMM and the provided examples.

#### *3.4.1.1 Windows*

1. Go to http://www.nvidia.com/object/cuda_get.html

2. Download and install the CUDA Driver, the CUDA Toolkit, and (optionally) the CUDA SDK code samples. The driver and toolkit are needed to get the GPU acceleration. The code samples are required for testing purposes. 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.

3. (Optional) To verify that you've installed things correctly, run a sample program available with the SDK code samples.

   Go to Start -> All Programs -> NVIDIA Corporation -> NVIDIA GPU Computing SDK ... -> NVIDIA GPU Computing SDK ... Browser

A window appears showing all the different sample programs you can try running (Figure 3.1).

Select the "CUDA C Samples" tab. Locate the program "Device Query" on this page and click on the associated "Run" link on the right-hand side. If things are running correctly, a window will appear stating how many devices are running CUDA (there should be at least 1) and that it/they passed the test.

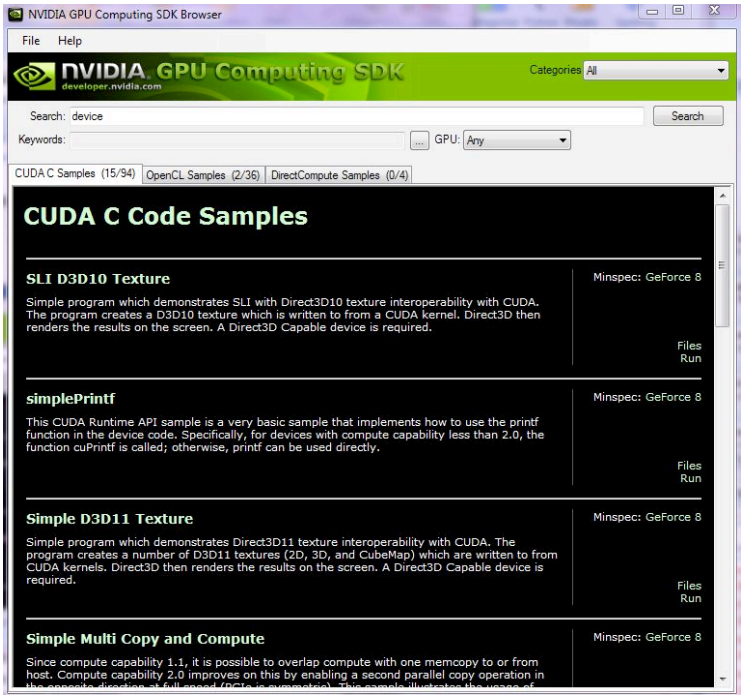



**Figure 3.1: Window for browsing the NVIDIA code samples**

### *3.4.1.2 Mac OS X*

1. Go to http://www.nvidia.com/object/cuda_get.html

2. Download and install the CUDA Toolkit, CUDA Driver, and (optionally) CUDA SDK code samples, version 4.0. The toolkit and driver are needed to get the GPU acceleration. The code samples are required for testing purposes.

3. (Optional) To verify that you've installed things correctly, run a sample program available with the SDK code samples.

    a. Open a terminal window. Go to Macintosh HD -> Applications -> Utilities. Click on Terminal.
    b. Set your environment variables so that your computer can locate the CUDA programs by typing the following two lines:

```
export PATH=/usr/local/cuda/bin:$PATH

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

    This sets the environment variables only for the terminal you are in. To set them permanently, you can add it to your bash_profile.

    Within the terminal window, navigate to the location of the code samples. If you installed everything in the default directories, then you would type:

```
cd /Developer/GPU Computing/C
```

    c. Compile the test programs by typing:

```
make
```

    d. Navigate to the location of the compiled programs by typing:

```
cd /Developer/GPU Computing/C/bin/darwin/release
```

    e. Run the deviceQuery program:

```
./deviceQuery
```

If things are running correctly, you will see how many devices are running CUDA (there should be at least 1) and a printout saying that it/they passed the test.

*Troubleshooting:*

If no devices are found, verify that you have a supported GPU card. If you do, re-run the installer and make sure to select a custom installation verifying that all boxes, including the kernel extension, are checked.

If you have multiple GPUs and only one is activated, this may be because of the energy-saving options. (This is the case for some MacBook Pros, which ship with a deactivated 9600M GPU). To change the energy-saving options, click System Preferences -> Energy Saver and set the graphics option to "Higher Performance." You will need to log out and then log back in for the new options to take effect.

Additional instructions and troubleshooting tips are provided in the "Getting Started" manual on the CUDA download site.

#### *3.4.1.3 Linux*

1. To compile the GPU code on a Linux machine, you will need gcc version 4.0 through 4.4. You can verify the version gcc installed on your system by typing:

```
gcc --version
```

2. Go to http://www.nvidia.com/object/cuda_get.html

3. Download and install the CUDA Driver, the CUDA Toolkit and (optionally) the CUDA SDK code samples, version 4.0. The toolkit and driver are needed to get the GPU acceleration. The code samples are required for testing purposes. We have tested this for the Redhat Enterprise Linux 5.x version (64-bit). Please refer to the CUDA website and "Getting Started" manual for a list of all supported Linux distributions and additional instructions.

a. Open a terminal window.

b. If you are running X Windows, you will need to turn it off to install the driver. You can do this by typing in the following as a **superuser**:

```
/sbin/init 3
```

c. Run the CUDA driver installation script as a **superuser**. If you turned off X Windows, you can turn it on again after the installation is complete (try the commands `startx` or `init 5`).

d. Run the CUDA toolkit installation script as a **superuser**.

e. Set your environment variables so that your computer can locate the CUDA programs.

**For the BASH shell (for your individual account)**

1. Set your PATH by typing:

```
export PATH=/usr/local/cuda/bin:$PATH
```

2. Set your library path. Depending on whether you use 32-bit or 64-bit Linux, type one of the following:

*For 32-bit, type (all on one line):*

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

*For 64-bit, type (all on one line):*

```
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:
$LD_LIBRARY_PATH
```

**These commands set the environment variables **only** for the terminal you are in and **only** for your account. To set them permanently, you will need to add it to ~/.bash_profile or ~/.bashrc

**For csh or tcsh shells (for your individual account)**

1. Set your PATH by typing:

```
setenv PATH ".:/usr/local/cuda/bin:$PATH"
```

2. Set your library path. Depending on whether you use 32-bit or 64-bit Linux, type one of the following:

*For 32-bit, type (all on one line):*

```
setenv LD_LIBRARY_PATH "/usr/local/cuda/lib:
$LD_LIBRARY_PATH"
```

*For 64-bit, type (all on one line):*

```
setenv LD_LIBRARY_PATH "/usr/local/cuda/lib64:
$LD_LIBRARY_PATH"
```

**These commands set the environment variables **only** for the terminal you are in and **only** for your account. To set them permanently, you will need to add it to ~/.cshrc (or similar) file.

**To set library path system-wide**

**CONSULT YOUR SYSTEM ADMINISTRATOR BEFORE CONTINUING**

1. You will still need to set your PATH as above.

2. Depending on whether you use 32-bit or 64-bit Linux, have your system administrator include one of the following paths in /etc/ld.so.conf (or equivalent type file) in the list of directories:

   *For 32-bit:* `/usr/local/cuda/lib`

   *For 64-bit:* `/usr/local/cuda/lib64`

3. Then, type as superuser/root:

```
ldconfig
```

f. Run the CUDA SDK installation script as a **regular user**.

4. (Optional) To verify that you've installed things correctly, run a sample program available with the SDK code samples.

   a. Within the terminal window, navigate to the location to compile the code samples. If you installed everything in the default directories, then you would type (default directory is only valid for version 2.3):

```
cd $HOME/NVIDIA_GPU_Computing_SDK/C
```

b. Compile the test programs by typing:

```
make
```

c. Navigate to the location of the compiled programs by typing (directory is only valid for version 2.3):

```
cd $HOME/NVIDIA_GPU_Computing_SDK/C/bin/linux/release
```

d. Run the deviceQuery program:

```
./deviceQuery
```

If things are running correctly, you will see how many devices are running CUDA (there should be at least 1) and a printout saying that it/they passed the test.

Additional instructions and troubleshooting tips are provided in the "Getting Started" manual on the CUDA download site.

### 3.4.2 Installing OpenCL

#### *3.4.2.1 Mac OS X*

Mac computers with Mac OS X 10.6 (Snow Leopard) or later come with OpenCL installed. OpenCL is not available for earlier versions of Mac OS. If you have Snow Leopard or later, you already have OpenCL installed. If you don't have Snow Leopard or later, you must upgrade your operating system to get OpenCL.

#### *3.4.2.2 Linux or Windows with NVIDIA GPUs*

OpenCL is installed with CUDA on Linux and Windows with NVIDIA GPUs. Follow the instructions in section 3.4.1 to install CUDA if you have an NVIDIA GPU.

#### *3.4.2.3 Linux or Windows with AMD GPUs (or to run on CPU)*

If you have an AMD GPU, install OpenCL by following the instructions at the AMD website http://developer.amd.com/sdks/amdappsdk/downloads/pages/default.aspx. The AMD APP SDK also supports running OpenCL on CPUs and is an efficient way to use OpenMM on machines without GPUs (such as headless servers).

OpenMM requires version 2.4 or later of the AMD APP SDK. If you want to use an AMD GPU, you also need version 11.7 or later of the Catalyst driver.

*Do not use OpenMM with earlier versions of the driver or SDK.* They contain serious bugs which may cause your simulations to produce incorrect results.

# 4 Compiling OpenMM from Source Code

This chapter describes the procedure for building and installing OpenMM libraries from source code. It is recommended that you use binary OpenMM libraries, if possible. If there are not suitable binary libraries for your system, consider building OpenMM from source code by following these instructions.

## 4.1 Prerequisites

Before building OpenMM from source, you will need the following:

- A C++ compiler
- CMake
- OpenMM source code

See the sections below for specific instructions for the different platforms.

### 4.1.1 Get a C++ compiler

You must have a C++ compiler installed before attempting to build OpenMM from source.

#### *4.1.1.1 Mac and Linux: gcc*

Use gcc on Mac/Linux. We have tested the examples on Fedora 10 with gcc 4.3.2 and on Mac OS X 10.6.8 with gcc 4.2.1.

To find out whether you have gcc installed, type:

```
which gcc
```

To find out what version of gcc you have, type:

```
gcc -version
```

If you do not already have gcc installed, you will need to download and install it. On the Mac, this means downloading the Xcode Tools from http://developer.apple.com/tools/Xcode/.

#### *4.1.1.2 Windows: Visual Studio*

On Windows systems, use the C++ compiler in Visual Studio version 10 (2010) or 9 (2008). You can download a free version of Visual C++ 10 2010 (Express Edition) from http://www.microsoft.com/express/vc/.

### 4.1.2 Install CMake

CMake is the build system used for OpenMM. You must install CMake version 2.8 or higher before attempting to build OpenMM from source. You can get CMake from http://www.cmake.org/. If you choose to build CMake from source on Linux, make sure you have the curses library installed beforehand, so that you will be able to build the CCMake visual CMake tool.

### 4.1.3 Get the OpenMM source code

You will also need the OpenMM source code before building OpenMM from source. To download and unpack OpenMM source code:

1. Browse to https://simtk.org/home/openmm/.
2. Click the "Downloads" link in the navigation bar on the left side.
3. Download OpenMM<Version>-Source.zip, choosing the latest version.
4. Unpack the zip file. Note the location where you unpacked the OpenMM source code.

### 4.1.4 Other Required Software

There are several other pieces of software you must install to compile certain parts of OpenMM. Which of these you need depends on the options you select in CMake.

- For compiling the CUDA Platform, you need:
  - CUDA (See Chapter 3 for installation instructions.)
- For compiling the OpenCL Platform, you need:
  - OpenCL (See Chapter 3 for installation instructions.)
- For compiling C and Fortran API wrappers, you need:
  - A Fortran compiler
  - gccxml (http://www.gccxml.org) - Download the 'Development Version from CVS' on the site's download page. The 'Latest Release (0.6.0)' files also available on the site's download page have been reported to fail.
- For compiling the Python API wrappers, you need:
  - Python 2.6 or 2.7 (http://www.python.org)
  - SWIG (http://www.swig.org)
  - py-dom-xpath (http://code.google.com/p/py-dom-xpath)
  - Doxygen (http://www.doxygen.org)
- To generate API documentation, you need:
  - Doxygen (http://www.doxygen.org)

## 4.2 Step 1: Configure with CMake

### 4.2.1 Build and source directories

First, create a directory in which to build OpenMM. A good name for this directory is build_openmm. We will refer to this as the "build_openmm directory" in the instructions below. This directory will contain the temporary files used by the OpenMM CMake build system. Do not create this build directory within the OpenMM source code directory. This is what is called an "out of source" build, because the build files will not be mixed with the source files.

Also note the location of the OpenMM source directory (i.e., where you unpacked the source code zip file). There should be a subdirectory called src, which contains a file called CMakeLists.txt. Note the location of this src directory. This directory is what we will call the "OpenMM source directory" in the following instructions.

### 4.2.2 Starting CMake

Configuration is the first step of the CMake build process. In the configuration step, the values of important build variables will be established.

#### *4.2.2.1 Mac and Linux*

On Mac and Linux machines, type the following two lines:

```
cd build_openmm
ccmake -i <path to OpenMM src directory>
```

That is not a typo. `ccmake` has two c's. CCMake is the visual CMake configuration tool. Press "`c`" within the CCMake interface to configure CMake. Follow the instructions in the "All Platforms" section below.

#### *4.2.2.2 Windows*

On Windows, perform the following steps:

- Click Start->All Programs->CMake 2.8->CMake
- In the box labeled "Where is the source code:" browse to OpenMM src directory (containing top CMakeLists.txt)
- In the box labeled "Where to build the binaries" browse to your build_openmm directory.
- Click the "Configure" button at the bottom of the CMake screen.
- Select "Visual Studio 9 2008" from the list of Generators. (or Visual Studio 10, if that is what you have installed)
- Follow the instructions in the "All Platforms" section below.

#### *4.2.2.3 All platforms*

There are several variables that can be adjusted in the CMake interface:

- If you intend to use CUDA (NVIDIA) or OpenCL acceleration, set the variable OPENMM_BUILD_CUDA_LIB or OPENMM_BUILD_OPENCL_LIB, respectively, to ON. Before doing so, be certain that you have installed and tested the drivers for the platform you have selected (see Section 3.4 on installing GPU software).

- There are lots of other options starting with OPENMM_BUILD that control whether to build particular features of OpenMM, such as plugins, API wrappers, and documentation.

- Do not worry about the SVNVERSION_EXE variable with value SVNVERSION_EXE_NOT_FOUND. That is unimportant.

- Set the variable CMAKE_INSTALL_PREFIX to the location where you want to install OpenMM. If you choose to change the CMAKE_INSTALL_PREFIX, you might also need to change the variable OPENMM_INSTALL_PREFIX, which is found in the advanced parameters. Press "t" or "Show Advanced Values" to expose the OPENMM_INSTALL_PREFIX variable in the CMake interface.

Configure (press "c") again. Adjust any variables that cause an error or are set to NOTFOUND (except for SVNVERSION_EXE).

Continue to configure (press "c") until no starred/red CMake variables are displayed. Congratulations, you have completed the configuration step.

## 4.3 Step 2: Generate Build Files with CMake

Once the configuration is done, the next step is generation. The generate "g" or "OK" or "Generate" option will not be available until configuration has completely converged.

### 4.3.1 Windows

- Press the "OK" or "Generate" button to generate Visual Studio project files.
- Ignore any warnings about "Policy CMP003" (Press "OK")
- If CMake does not exit automatically, press the close button in the upper-right corner of the CMake title bar to exit.

### 4.3.2 Mac and Linux

- Press g to generate the Makefile.
- Ignore any warnings about "Policy CMP003" (Press "e")

- If CMake does not exit automatically, press "`q`" to exit.

That's it! Generation is the easy part. Now it's time to build.

## 4.4 Step 3: Build OpenMM

### 4.4.1 Windows

- Open the file OpenMM.sln in your openmm_build directory in Visual Studio.
- Set the configuration type to "Release" (not "Debug") in the toolbar.
- From the Build menu, click Build->Build Solution
- The OpenMM libraries and test programs will be created. This takes some time.
- The test program TestCudaRandom might not build on Windows. This is OK.

### 4.4.2 Mac and Linux

- Type `make` in the openmm_build directory.
- The OpenMM libraries and test programs will be created. This takes some time.

## 4.5 Step 4: Test your build

After OpenMM has been built, test the build before installing.

### 4.5.1 Windows

In Visual Studio, far-click/right-click RUN_TESTS in the Solution Explorer Panel. Select RUN_TESTS->build to begin testing. Ignore any failures for TestCudaRandom.

### 4.5.2 Mac and Linux

Type:

```
make test
```

You should see a series of test results like this:

```
1/ 38 Testing TestReferenceAndersenThermosta   Passed
2/ 38 Testing TestReferenceBrownianIntegrato   Passed
3/ 38 Testing TestReferenceCMMotionRemover     Passed
4/ 38 Testing TestReferenceCustomNonbondedFo   Passed
... <many other tests> ...
```

`Passed` is good. `FAILED` is bad. If any tests fail, you can run them individually to get more detailed error information. Note that some tests are stochastic, and therefore are expected to fail a small fraction of the time. These tests will say so in the error message:

```
./TestReferenceLangevinIntegrator

exception: Assertion failure at
TestReferenceLangevinIntegrator.cpp:129.  Expected 9.97741,
found 10.7884 (This test is stochastic and may occasionally
fail)
```

## 4.6 Step 5: Install OpenMM

If all of the tests pass, you are ready to install OpenMM.

### 4.6.1 Windows

In the Solution Explorer Panel, far-click/right-click INSTALL->build.

### 4.6.2 Mac and Linux

Type:

```
make install
```

If you are installing to a system area, such as /usr/local/openmm/, you will need to type:

```
sudo make install
```

## 4.7 Step 6: Set Your Library Path

Refer to Section 3.3 for instructions on setting your library path environment variable (PATH, LD_LIBRARY_PATH, or DYLD_LIBRARY_PATH) to point to your new OpenMM installation.

Congratulations! You successfully have built and installed OpenMM from source!

# 5 OpenMM Tutorials

## 5.1 Example Files Overview

Four example files are provided in the examples folder, each designed with a specific objective.

- **HelloArgon:** A very simple example intended for verifying that you have installed OpenMM correctly. It also introduces you to the basic classes within OpenMM.
- **HelloSodiumChloride:** This example shows you our recommended strategy for integrating OpenMM into an existing molecular dynamics code.
- **HelloEthane:** The main purpose of this example is to demonstrate how to tell OpenMM about bonded forces (bond stretch, bond angle bend, dihedral torsion).
- **HelloWaterBox:** This example shows you how to use OpenMM to model explicit solvation, including setting up periodic boundary conditions. It runs extremely fast on a GPU but very, very slowly on a CPU, so it is an excellent example to use to compare performance on the GPU versus the CPU. The other examples provided use systems where the performance difference would be too small to notice.

The two fundamental examples—HelloArgon and HelloSodiumChloride—are provided in C++, C, and Fortran, as indicated in the table below. The other two examples—HelloEthane and HelloWaterBox—follow the same structure as HelloSodiumChloride but demonstrate more calls within the OpenMM API. They are only provided in C++ but can be adapted to run in C and Fortran by following the mappings described in Chapter 7. HelloArgon and HelloSodiumChloride also serve as examples of how to do these mappings. The sections below describe the HelloArgon, HelloSodiumChloride, and HelloEthane programs in more detail.

| Example | Solvent | Thermostat | Boundary | Forces & Constraints | API |
|---|---|---|---|---|---|
| **Argon** | Vacuum | None | None | Non-bonded* | C++, C, Fortran |
| **Sodium Chloride** | Implicit water | Langevin | None | Non-bonded* | C++, C, Fortran |
| **Ethane** | Vacuum | None | None | Non-bonded,* stretch, bend, torsion | C++ |
| **Water Box** | Explicit water | Andersen | Periodic | Non-bonded,* stretch, bend, constraints | C++ |

*van der Waals and Coulomb forces

## 5.2 Running Example Files

The instructions below are for running the HelloArgon program. A similar process would be used to run the other examples.

### 5.2.1 Visual Studio

Navigate to wherever you saved the example files. Descend into the directory folder VisualStudio2005. Double-click the file HelloArgon.sln (a Microsoft Visual Studio Solution file). Visual Studio will launch.

Note: these files were created using Visual Studio 8. If you are using Visual Studio 9 (2008 Express Edition), the program will ask if you want to convert the files to the new version. Agree and continue through the conversion process.

In Visual Studio, make sure the "Solution Configuration" is set to "Release" and not "Debug". The “Solution Configuration” can be set using the drop-down menu in the top toolbar, next

to the green arrow (see Figure 5.1 below). Due to incompatibilities among Visual Studio versions, we do not provide pre-compiled debug binaries.

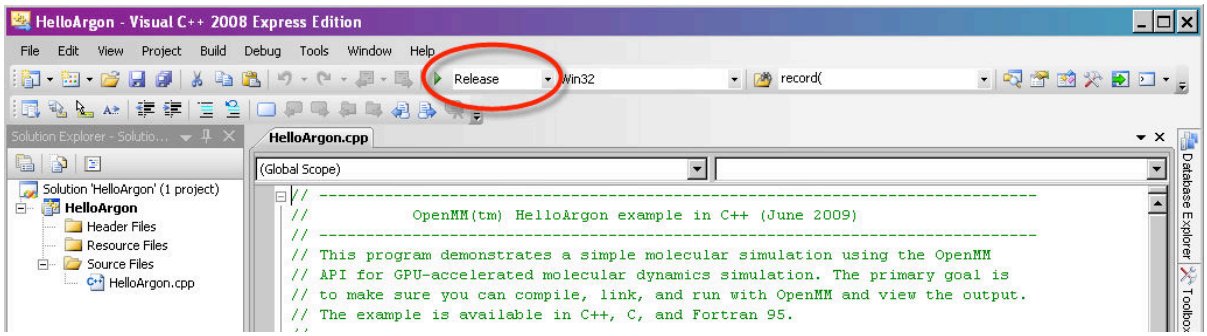



**Figure 5.1: Setting "Solution Configuration" to "Release" mode in Visual Studio**

From the command options select Debug -> Start Without Debugging (or CTRL-F5). See Figure 5.2. This will also compile the program, if it has not previously been compiled.

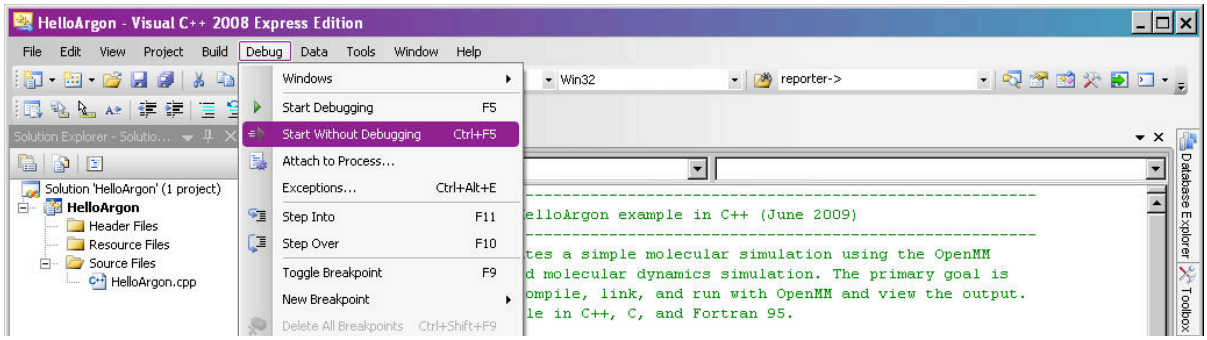



**Figure 5.2: Run a program in Visual Studio**

You should see a series of lines like the following output on your screen:

```
REMARK  Using OpenMM platform Reference
MODEL     1
ATOM      1  AR   AR     1       0.000   0.000   0.000  1.00  0.00
ATOM      2  AR   AR     1       5.000   0.000   0.000  1.00  0.00
ATOM      3  AR   AR     1       10.000  0.000   0.000  1.00  0.00
ENDMDL
```

…

```
MODEL     250
ATOM      1  AR   AR     1       0.233   0.000   0.000  1.00  0.00
ATOM      2  AR   AR     1       5.068   0.000   0.000  1.00  0.00
ATOM      3  AR   AR     1       9.678   0.000   0.000  1.00  0.00
ENDMDL
MODEL     251
ATOM      1  AR   AR     1       0.198   0.000   0.000  1.00  0.00
ATOM      2  AR   AR     1       5.082   0.000   0.000  1.00  0.00
ATOM      3  AR   AR     1       9.698   0.000   0.000  1.00  0.00
ENDMDL
MODEL     252
ATOM      1  AR   AR     1       0.165   0.000   0.000  1.00  0.00
ATOM      2  AR   AR     1       5.097   0.000   0.000  1.00  0.00
ATOM      3  AR   AR     1       9.717   0.000   0.000  1.00  0.00
ENDMDL
```

#### *5.2.1.1 Determining the platform being used*

The very first line of the output will indicate whether you are running on the CPU (Reference platform) or a GPU (CUDA or OpenCL platform). It will say one of the following:

```
REMARK  Using OpenMM platform Reference
REMARK  Using OpenMM platform Cuda
REMARK  Using OpenMM platform OpenCL
```

If you have a supported GPU, the program should, by default, run on the GPU.

#### *5.2.1.2 Visualizing the results*

You can output the results to a PDB file that could be visualized using programs like VMD (http://www.ks.uiuc.edu/Research/vmd/) or PyMol (http://pymol.sourceforge.net/). To do this within Visual Studios:

1. Right-click on the project name HelloArgon (not one of the files) and select the "Properties" option.

2. On the "Property Pages" form, select "Debugging" under the "Configuration Properties" node.
3. In the "Command Arguments" field, type:

```
> argon.pdb
```

   This will save the output to a file called argon.pdb in the current working directory (default is the VisualStudio2005 directory). If you want to save it to another directory, you will need to specify the full path.
4. Select "OK"

Now, when you run the program in Visual Studio, no text will appear. After a short time, you should see the message "`Press any key to continue…`," indicating that the program is complete and that the PDB file has been completely written.

### 5.2.2 Mac OS X/Linux

Navigate to wherever you saved the example files.

Verify your makefile by consulting the MakefileNotes file in this directory, if necessary.

Type:

```
make
```

Then run the program by typing:

```
./HelloArgon
```

You should see a series of lines like the following output on your screen:

```
REMARK  Using OpenMM platform Reference
MODEL     1
ATOM      1  AR   AR     1       0.000   0.000   0.000  1.00  0.00
ATOM      2  AR   AR     1       5.000   0.000   0.000  1.00  0.00
```

```
ATOM      3  AR   AR     1       10.000  0.000   0.000  1.00  0.00
ENDMDL

...

MODEL     250
ATOM      1  AR   AR     1       0.233   0.000   0.000  1.00  0.00
ATOM      2  AR   AR     1       5.068   0.000   0.000  1.00  0.00
ATOM      3  AR   AR     1       9.678   0.000   0.000  1.00  0.00
ENDMDL
MODEL     251
ATOM      1  AR   AR     1       0.198   0.000   0.000  1.00  0.00
ATOM      2  AR   AR     1       5.082   0.000   0.000  1.00  0.00
ATOM      3  AR   AR     1       9.698   0.000   0.000  1.00  0.00
ENDMDL
MODEL     252
ATOM      1  AR   AR     1       0.165   0.000   0.000  1.00  0.00
ATOM      2  AR   AR     1       5.097   0.000   0.000  1.00  0.00
ATOM      3  AR   AR     1       9.717   0.000   0.000  1.00  0.00
ENDMDL
```

#### *5.2.2.1 Determining the platform being used*

The very first line of the output will indicate whether you are running on the CPU (Reference platform) or a GPU (CUDA or OpenCL platform). It will say one of the following:

```
REMARK  Using OpenMM platform Reference
REMARK  Using OpenMM platform Cuda
REMARK  Using OpenMM platform OpenCL
```

If you have a supported GPU, the program should, by default, run on the GPU.

#### *5.2.2.2 Visualizing the results*

You can output the results to a PDB file that could be visualized using programs like VMD (http://www.ks.uiuc.edu/Research/vmd/) or PyMol (http://pymol.sourceforge.net/) by typing:

```
./HelloArgon > argon.pdb
```

#### *5.2.2.3 Compiling Fortran and C examples*

The Makefile provided with the examples can also be used to compile the Fortran and C examples.

The Fortran compiler needs to load a version of the libstdc++.dylib library that is compatible with the version of gcc used to build OpenMM; OpenMM for Mac is compiled using gcc 4.2. If you are compiling with a different version, edit the Makefile and add the following flag to FCPPLIBS: `-L/usr/lib/gcc/i686-apple-darwin10/4.2.1`.

When the Makefile has been updated, type:

```
make all
```

## 5.3 HelloArgon Program

The HelloArgon program simulates three argon atoms in a vacuum. It is a simple program primarily intended for you to verify that you are able to compile, link, and run with OpenMM. It also demonstrates the basic calls needed to run a simulation using OpenMM.

### 5.3.1 Including OpenMM-defined functions

The OpenMM header file *OpenMM.h* instructs the program to include everything defined by the OpenMM libraries. Include the header file by adding the following line at the top of your program:

```
#include "OpenMM.h"
```

### 5.3.2 Running a program on GPU platforms

By default, a program will run on the Reference platform. In order to run a program on another platform (e.g., an NVIDIA or AMD GPU), you need to load the required shared libraries for that other platform (e.g., Cuda, OpenCL). The easy way to do this is to call:

```
OpenMM::Platform::loadPluginsFromDirectory(
      OpenMM::Platform::getDefaultPluginsDirectory());
```

This will load all the shared libraries (plug-ins) that can be found, so you do not need to explicitly know which libraries are available on a given machine. In this way, the program will be able to run on another platform, if it is available.

### 5.3.3 Running a simulation using the OpenMM public API

The OpenMM public API was described in Section 2.4. Here you will see how to use those classes to create a simple system of three argon atoms and run a short simulation. The main components of the simulation are within the function `simulateArgon()`:

1. **System** – We first establish a system and add a non-bonded force to it. At this point, there are no particles in the system.

```
// Create a system with nonbonded forces.
OpenMM::System system;
OpenMM::NonbondedForce* nonbond =
            new OpenMM::NonbondedForce();
system.addForce(nonbond);
```

   We then add the three argon atoms to the system. For this system, all the data for the particles are hard-coded into the program. While not a realistic scenario, it makes the example simpler and clearer. The `std::vector<OpenMM::Vec3>` is an array of vectors of 3.

```
// Create three atoms.
std::vector<OpenMM::Vec3> initPosInNm(3);
for (int a = 0; a < 3; ++a)
{
  initPosInNm[a] = OpenMM::Vec3(0.5*a,0,0); // location, nm

  system.addParticle(39.95); // mass of Ar, grams per mole

  // charge, L-J sigma (nm), well depth (kJ)
  nonbond->addParticle(0.0, 0.3350, 0.996); // vdWRad(Ar)=
      .188 nm
}
```

***Units:*** Be very careful with the units in your program. It is very easy to make mistakes with the units, so we recommend including them in your variable names, as we have done here `initPosInNm` (position in nanometers). OpenMM provides conversion constants that should be used whenever there are conversions to be done; for simplicity, we did not do that in HelloArgon, but all the other examples show the use of these constants.

It is hard to overemphasize the importance of careful units handling—it is very easy to make a mistake despite, or perhaps because of, the trivial nature of units conversion. For more information about the units used in OpenMM, see Section 13.2.

***Adding Particle Information:*** Both the system and the non-bonded force require information about the particles. The system just needs to know the mass of the particle. The non-bonded force requires information about the charge (in this case, argon is uncharged), and the Lennard-Jones parameters sigma (zero-energy separation distance) and well depth (see Section 14.6.1 for more details).

Note that the van der Waals radius for argon is 0.188 nm and that it has already been converted to sigma (0.335 nm) in the example above where it is added to the non-bonded force; in your code, you should make use of the appropriate conversion factor supplied with OpenMM as discussed in Section 13.2.

2. **Integrator** – We next specify the integrator to use to perform the calculations. In this case, we choose a Verlet integrator to run a constant energy simulation. The only argument required is the step size in picoseconds.

```
OpenMM::VerletIntegrator integrator(0.004); // step size in ps
```

We have chosen to use 0.004 picoseconds, or 4 femtoseconds, which is larger than that used in a typical molecular dynamics simulation. However, since this example does not have any bonds with higher frequency components, like most molecular dynamics simulations do, this is an acceptable value.

3. **Context** – The context is an object that consists of an integrator and a system. It manages the state of the simulation. The code below initializes the context. We then let the context select the best platform available to run on, since this is not specifically specified, and print out the chosen platform. This is useful information, especially when debugging.

```
// Let OpenMM Context choose best platform.
OpenMM::Context context(system, integrator);
printf( "REMARK  Using OpenMM platform %s\n",
              context.getPlatform().getName().c_str() );
```

We then initialize the system, setting the initial time, as well as the initial positions and velocities of the atoms. In this example, we leave time and velocity at their default values of zero.

```
// Set starting positions of the atoms. Leave time and velocity
zero.
          context.setPositions(initPosInNm);
```

4. **Initialize and run the simulation** – The next block of code runs the simulation and saves its output. For each frame of the simulation (in this example, a frame is defined by the advancement interval of the integrator; see below), the current state of the simulation is obtained and written out to a PDB-formatted file.

```
// Simulate.
for (int frameNum=1; ;++frameNum) {
  // Output current state information.
  OpenMM::State state    =
            context.getState(OpenMM::State::Positions);
  const double  timeInPs = state.getTime();
  writePdbFrame(frameNum, state); // output coordinates
```

*Getting state information has to be done in bulk, asking for information for all the particles at once.* This is computationally expensive since this information can reside on the GPUs and requires communication overhead to retrieve, so you do not want to do it very often. In the above code, we only request the positions, since that is all that is needed, and time from the state.

The simulation stops after 10 ps; otherwise we ask the integrator to take 10 steps (so one frame is equivalent to 10 time steps). Normally, we would want to take more than 10 steps at a time, but to get a reasonable-looking animation, we use 10.

```
if (timeInPs >= 10.)
            break;

// Advance state many steps at a time, for efficient use of OpenMM.
integrator.step(10); // (use a lot more than this normally)
```

### 5.3.4 Error handling for OpenMM

Error handling for OpenMM is explicitly designed so you do not have to check the status after every call. If anything goes wrong, OpenMM throws an exception. It uses standard exceptions, so on many platforms, you will get the exception message automatically. However, we recommend using `try-catch` blocks to ensure you do catch the exception.

```
int main()
{
    try {
        simulateArgon();
        return 0; // success!
    }
    // Catch and report usage and runtime errors detected by OpenMM and
fail.
    catch(const std::exception& e) {
        printf("EXCEPTION: %s\n", e.what());
        return 1; // failure!
    }
}
```

### 5.3.5 Writing out PDB files

For the HelloArgon program, we provide a simple PDB file writing function `writePdbFrame` that *only* writes out argon atoms. The function has nothing to do with OpenMM except for using the OpenMM State. The function extracts the positions from the State in nanometers ($10^{-9}$ m) and converts them to Angstroms ($10^{-10}$ m) to be compatible with the PDB format. Again, we emphasize how important it is to track the units being used!

```
void writePdbFrame(int frameNum, const OpenMM::State& state)
{
    // Reference atomic positions in the OpenMM State.
```

```
    const std::vector<OpenMM::Vec3>& posInNm = state.getPositions();

    // Use PDB MODEL cards to number trajectory frames
    printf("MODEL     %d\n", frameNum); // start of frame
    for (int a = 0; a < (int)posInNm.size(); ++a)
    {
        printf("ATOM  %5d  AR   AR     1    ", a+1); // atom number
        printf("%8.3f%8.3f%8.3f  1.00  0.00\n",      // coordinates
            // "*10" converts nanometers to Angstroms
            posInNm[a][0]*10, posInNm[a][1]*10, posInNm[a][2]*10);
    }
    printf("ENDMDL\n"); // end of frame
}
```

`MODEL` and `ENDMDL` are used to mark the beginning and end of a frame, respectively. By including multiple frames in a PDB file, you can visualize the simulation trajectory.

### 5.3.6 HelloArgon output

The output of the HelloArgon program can be saved to a *.pdb* file and visualized using programs like VMD or PyMol (see Section 5.2). You should see three atoms moving linearly away and towards one another:

You may need to adjust the van der Waals radius in your visualization program to see the atoms colliding.

## 5.4 HelloSodiumChloride Program

The HelloSodiumChloride models several sodium ($Na^+$) and chloride ($Cl^-$) ions in implicit solvent (using a Generalized Born/Surface Area, or GBSA, OBC model). As with the HelloArgon program, only non-bonded forces are simulated.

The main purpose of this example is to illustrate our recommended strategy for integrating OpenMM into an existing molecular dynamics (MD) code:

1. **Write a few, high-level interface routines containing all your OpenMM calls**: Rather than make OpenMM calls throughout your program, we recommend writing a handful of interface routines that understand both your MD code's data structures and OpenMM. Organize these routines into a separate compilation unit so you do not have to make huge changes to your existing MD code. These routines could be written in any language that is callable from the existing MD code. We recommend writing them in C++ since that is what OpenMM is written in, but you can also write them in C or Fortran; see Chapter 7.

2. **Call only these high-level interface routines from your existing MD code:** This provides a clean separation between the existing MD code and OpenMM, so that changes to OpenMM will not directly impact the existing MD code. One way to implement this is to use opaque handles, a standard trick used (for example) for opening files in Linux. An existing MD code can communicate with OpenMM via the handle, but knows none of the details of the handle. It only has to hold on to the handle and give it back to OpenMM.

In the example described below, you will see how this strategy can be implemented for a very simple MD code. Chapter 8 describes the strategies used in integrating OpenMM into real MD codes.

### 5.4.1 Simple molecular dynamics system

The initial sections of HelloSodiumChloride.cpp represent a very simple molecular dynamics system. The system includes modeling and simulation parameters and the atom and force field data. It also provides a data structure `posInAng[3]` for storing the current state. These sections represent (in highly simplified form) information that would be available from an existing MD code, and will be used to demonstrate how to integrate OpenMM with an existing MD program.

```
// ----------------------------------------------------------------
//                   MODELING AND SIMULATION PARAMETERS
// ----------------------------------------------------------------
static const double Temperature         = 300;      // Kelvins
static const double FrictionInPerPs     = 91.;      // collisions per
picosecond
```

```
static const double SolventDielectric   = 80.;     // typical for water
static const double SoluteDielectric    = 2.;      // typical for protein

static const double StepSizeInFs        = 2;       // integration step
size (fs)
static const double ReportIntervalInFs  = 50;      // how often to issue
PDB frame (fs)
static const double SimulationTimeInPs  = 100;     // total simulation
time (ps)

// Decide whether to request energy calculations.
static const bool   WantEnergy          = true;


// ----------------------------------------------------------------
//                          ATOM AND FORCE FIELD DATA
// ----------------------------------------------------------------
// This is not part of OpenMM; just a struct we can use to collect atom
// parameters for this example. Normally atom parameters would come from
the
// force field's parameterization file. We're going to use data in
Angstrom and
// Kilocalorie units and show how to safely convert to OpenMM's internal
unit
// system which uses nanometers and kilojoules.
static struct MyAtomInfo {
    const char* pdb;
    double      mass, charge, vdwRadiusInAng, vdwEnergyInKcal,
                gbsaRadiusInAng, gbsaScaleFactor;
    double      initPosInAng[3];
    double      posInAng[3]; // leave room for runtime state info
} atoms[] = {
// pdb   mass  charge  vdwRad vdwEnergy   gbsaRad gbsaScale  initPos
{" NA ", 22.99,  1,    1.8680, 0.00277,    1.992,   0.8,     8, 0,  0},
{" CL ", 35.45, -1,    2.4700, 0.1000,     1.735,   0.8,    -8, 0,  0},
{" NA ", 22.99,  1,    1.8680, 0.00277,    1.992,   0.8,     0, 9,  0},
{" CL ", 35.45, -1,    2.4700, 0.1000,     1.735,   0.8,     0,-9,  0},
{" NA ", 22.99,  1,    1.8680, 0.00277,    1.992,   0.8,     0, 0,-10},
{" CL ", 35.45, -1,    2.4700, 0.1000,     1.735,   0.8,     0, 0, 10},
{""} // end of list
};
```

### 5.4.2 Interface routines

The key to our recommended integration strategy is the interface routines. You will need to decide what interface routines are required for effective communication between your existing MD program and OpenMM, but typically there will only be six or seven. In our example, the following four routines suffice:

- **Initialize:** Data structures that already exist in your MD program (i.e., force fields, constraints, atoms in the system) are passed to the `Initialize` routine, which makes appropriate calls to OpenMM and then returns a handle to the OpenMM object that can be used by the existing MD program.
- **Terminate:** Clean up the heap space allocated by `Initialize` by passing the handle to the `Terminate` routine.
- **Advance State:** The `AdvanceState` routine advances the simulation. It requires that the calling function, the existing MD code, gives it a handle.
- **Retrieve State:** When you want to do an analysis or generate some kind of report, you call the `RetrieveState` routine. You have to give it a handle. It then fills in a data structure that is defined in the existing MD code, allowing the MD program to use it in its existing routines without further modification.

Note that these are just descriptions of the routines' functions—you can call them anything you like and implement them in whatever way makes sense for your MD code.

In the example code, the four routines performing these functions, plus an opaque data structure (the handle), would be declared, as shown below. Then, the main program, which sets up, runs, and reports on the simulation, accesses these routines and the opaque data structure (in this case, the variable `omm`). As you can see, it does not have access to any OpenMM declarations, only to the interface routines that you write so there is no need to change the build environment.

```
struct MyOpenMMData;
static MyOpenMMData* myInitializeOpenMM(const MyAtomInfo atoms[],
                                        double temperature,
                                        double frictionInPs,
                                        double solventDielectric,
                                        double soluteDielectric,
                                        double stepSizeInFs,
                                        std::string& platformName);
static void          myStepWithOpenMM(MyOpenMMData*, int numSteps);
static void          myGetOpenMMState(MyOpenMMData*, bool
                                 wantEnergy,double& time, double& energy,
                                 MyAtomInfo atoms[]);
static void          myTerminateOpenMM(MyOpenMMData*);
```

```
// ----------------------------------------------------------------------
//                                MAIN PROGRAM
// ----------------------------------------------------------------------
int main() {
    const int NumReports     = (int)(SimulationTimeInPs*1000 /
            ReportIntervalInFs + 0.5);
    const int NumSilentSteps = (int)(ReportIntervalInFs / StepSizeInFs +
            0.5);

    // ALWAYS enclose all OpenMM calls with a try/catch block to make sure
that
    // usage and runtime errors are caught and reported.
    try {
        double        time, energy;
        std::string   platformName;

        // Set up OpenMM data structures; returns OpenMM Platform name.
        MyOpenMMData* omm = myInitializeOpenMM(atoms, Temperature,
              FrictionInPerPs,SolventDielectric, SoluteDielectric,
             StepSizeInFs, platformName);

        // Run the simulation:
        //  (1) Write the first line of the PDB file and the initial
                configuration.
        //  (2) Run silently entirely within OpenMM between reporting
                intervals.
        //  (3) Write a PDB frame when the time comes.
        printf("REMARK  Using OpenMM platform %s\n",
                platformName.c_str());
        myGetOpenMMState(omm, WantEnergy, time, energy, atoms);
        myWritePDBFrame(1, time, energy, atoms);

        for (int frame=2; frame <= NumReports; ++frame) {
            myStepWithOpenMM(omm, NumSilentSteps);
            myGetOpenMMState(omm, WantEnergy, time, energy, atoms);
            myWritePDBFrame(frame, time, energy, atoms);
        }

        // Clean up OpenMM data structures.
        myTerminateOpenMM(omm);

        return 0; // Normal return from main.
    }

    // Catch and report usage and runtime errors detected by OpenMM and
fail.
    catch(const std::exception& e) {
        printf("EXCEPTION: %s\n", e.what());
        return 1;
    }
}
```

We will examine the implementation of each of the four interface routines and the opaque data structure (handle) in the sections below.

#### 5.4.2.1 *Units*

The simple molecular dynamics system described in Section 5.4.1 employs the commonly used units of angstroms and kcals. These differ from the units and parameters used within OpenMM (see Section 13.2): nanometers and kilojoules. These differences may be small but they are critical and must be carefully accounted for in the interface routines.

#### 5.4.2.2 *Lennard-Jones potential*

The Lennard-Jones potential describes the energy between two identical atoms as the distance between them varies.

The van der Waals "size" parameter is used to identify the distance at which the energy between these two atoms is at a minimum (that is, where the van der Waals force is most attractive). There are several ways to specify this parameter, typically, either as the van der Waals radius $r_{vdw}$ or as the actual distance between the two atoms $d_{min}$ (also called $r_{min}$), which is twice the van der Waals radius $r_{vdw}$. A third way to describe the potential is through sigma $\sigma$, which identifies the distance at which the energy function crosses zero as the atoms move closer together than $d_{min}$. (See Section 14.6.1 for more details about the relationship between these).

$\sigma$ turns out to be about 0.89 * $d_{min}$, which is close enough to $d_{min}$ that it makes it hard to distinguish the two. Be very careful that you use the correct value. In the example below, we will show you how to use the built-in OpenMM conversion constants to avoid errors.

Lennard-Jones parameters are defined for pairs of identical atoms, but must also be applied to pairs of dissimilar atoms. That is done by "combining rules" that differ among popular MD codes. Two of the most common are:

- Lorentz-Berthelot (used by AMBER, CHARMM): $r = \frac{r_i + r_j}{2}, \quad \varepsilon = \sqrt{\varepsilon_i \varepsilon_j}$
- Jorgensen (used by OPLS): $r = \sqrt{r_i r_j}, \quad \varepsilon = \sqrt{\varepsilon_i \varepsilon_j}$

where $r$ = the effective van der Waals "size" parameter (minimum radius, minimum distance, or zero crossing (sigma)), and $\varepsilon$ = the effective van der Waals energy well depth parameter, for the dissimilar pair of atoms $i$ and $j$.

OpenMM only implements Lorentz-Berthelot directly, but others can be implemented using the CustomNonbondedForce class. (See Section 15.1 for details.)

#### 5.4.2.3 *Opaque handle MyOpenMMData*

In this example, the handle used by the interface to OpenMM is a pointer to a struct called `MyOpenMMData`. The pointer itself is opaque, meaning the calling program has no knowledge of what the layout of the object it points to is, or how to use it to directly interface with OpenMM. The calling program will simply pass this opaque handle from one interface routine to another.

There are many different ways to implement the handle. The code below shows just one example. A simulation requires three OpenMM objects (a System, a Context, and an Integrator) and so these must exist within the handle. If other objects were required for a simulation, you would just add them to your handle; there would be no change in the main program using the handle.

```
struct MyOpenMMData {
    MyOpenMMData() : system(0), context(0), integrator(0) {}
    ~MyOpenMMData() {delete system; delete context; delete integrator;}
    OpenMM::System*         system;
    OpenMM::Context*        context;
    OpenMM::Integrator*     integrator;
};
```

In addition to establishing pointers to the required three OpenMM objects, `MyOpenMMData` has a constructor `MyOpenMMData()` that sets the pointers for the three OpenMM objects to zero and a destructor `~MyOpenMMData()` that (in C++) gives the heap space back. This was done in-line in the HelloArgon program, but we recommend you use something like the method here instead.

#### 5.4.2.4 *myInitializeOpenMM*

The `myInitializeOpenMM` function takes the data structures and simulation parameters from the existing MD code and returns a new handle that can be used to do efficient computations with OpenMM. It also returns the `platformName` so the calling program knows what platform (e.g., CUDA, OpenCL, Reference) was used.

```
static MyOpenMMData*
myInitializeOpenMM( const MyAtomInfo    atoms[],
                    double              temperature,
                    double              frictionInPs,
                    double              solventDielectric,
                    double              soluteDielectric,
                    double              stepSizeInFs,
                    std::string&        platformName)
```

This initialization routine is very similar to the HelloArgon example program, except that objects are created and put in the handle. For instance, just as in the HelloArgon program, the first step is to load the OpenMM plug-ins, so that the program will run on the best performing platform that is available. Then, a System is created **and** assigned to the handle `omm`. Similarly, forces are added to the System which is already in the handle.

```
// Load all available OpenMM plugins from their default location.
OpenMM::Platform::loadPluginsFromDirectory
       (OpenMM::Platform::getDefaultPluginsDirectory());

// Allocate space to hold OpenMM objects while we're using them.
MyOpenMMData* omm = new MyOpenMMData();

// Create a System and Force objects within the System. Retain a reference
// to each force object so we can fill in the forces. Note: the OpenMM
// System takes ownership of the force objects;don't delete them yourself.
omm->system = new OpenMM::System();
OpenMM::NonbondedForce* nonbond = new OpenMM::NonbondedForce();
OpenMM::GBSAOBCForce*   gbsa    = new OpenMM::GBSAOBCForce();
omm->system->addForce(nonbond);
omm->system->addForce(gbsa);

// Specify dielectrics for GBSA implicit solvation.
gbsa->setSolventDielectric(solventDielectric);
gbsa->setSoluteDielectric(soluteDielectric);
```

In the next step, atoms are added to the System within the handle, with information about each atom coming from the data structure that was passed into the initialization function from the existing MD code. As shown in the HelloArgon program, both the System and the

forces need information about the atoms. For those unfamiliar with the C++ Standard Template Library, the `push_back` function called at the end of this code snippet just adds the given argument to the end of a C++ "vector" container.

```
// Specify the atoms and their properties:
//  (1) System needs to know the masses.
//  (2) NonbondedForce needs charges,van der Waals properties(in MD
units!).
//  (3) GBSA needs charge, radius, and scale factor.
//  (4) Collect default positions for initializing the simulation later.
std::vector<Vec3> initialPosInNm;
for (int n=0; *atoms[n].pdb; ++n) {
     const MyAtomInfo& atom = atoms[n];

     omm->system->addParticle(atom.mass);

     nonbond->addParticle(atom.charge,
                          atom.vdwRadiusInAng * OpenMM::NmPerAngstrom
                                              * OpenMM::SigmaPerVdwRadius,
                          atom.vdwEnergyInKcal * OpenMM::KJPerKcal);

     gbsa->addParticle(atom.charge,
                       atom.gbsaRadiusInAng * OpenMM::NmPerAngstrom,
                       atom.gbsaScaleFactor);

     // Convert the initial position to nm and append to the array.
     const Vec3 posInNm(atom.initPosInAng[0] * OpenMM::NmPerAngstrom,
                  atom.initPosInAng[1] * OpenMM::NmPerAngstrom,
                  atom.initPosInAng[2] * OpenMM::NmPerAngstrom);
     initialPosInNm.push_back(posInNm);
```

***Units:*** Here we emphasize the need to pay special attention to the units. As mentioned earlier, the existing MD code in this example uses units of angstroms and kcals, but OpenMM uses nanometers and kilojoules. So the initialization routine will need to convert the values from the existing MD code into the OpenMM units before assigning them to the OpenMM objects.

In the code above, we have used the unit conversion constants that come with OpenMM (e.g., `OpenMM::NmPerAngstrom`) to perform these conversions. Combined with the naming convention of including the units in the variable name (e.g., `initPosInAng`), the unit conversion constants are useful reminders to pay attention to units and minimize errors.

Finally, the initialization routine creates the Integrator and Context for the simulation. Again, note the change in units for the arguments! The routine then gets the platform that will be used to run the simulation and returns that, along with the handle `omm`, back to the calling function.

```
// Choose an Integrator for advancing time, and a Context connecting the
// System with the Integrator for simulation. Let the Context choose the
// best available Platform. Initialize the configuration from the default
// positions we collected above. Initial velocities will be zero but could
// have been set here.
omm->integrator = new OpenMM::LangevinIntegrator(temperature,
                                                 frictionInPs,
                                                 stepSizeInFs *
                                                 OpenMM::PsPerFs);
omm->context    = new OpenMM::Context(*omm->system, *omm->integrator);
omm->context->setPositions(initialPosInNm);

platformName = omm->context->getPlatform().getName();
return omm;
```

#### 5.4.2.5 *myGetOpenMMState*

The `myGetOpenMMState` function takes the handle and returns the time, energy, and data structure for the atoms in a way that the existing MD code can use them without modification.

```
static void
myGetOpenMMState(MyOpenMMData* omm, bool wantEnergy,
                 double& timeInPs, double& energyInKcal,
                 MyAtomInfo atoms[])
```

Again, this is another interface routine in which you need to be very careful of your units! Note the conversion from the OpenMM units back to the units used in the existing MD code.

```
int infoMask = 0;
infoMask = OpenMM::State::Positions;
if (wantEnergy) {
   infoMask += OpenMM::State::Velocities; // for kinetic energy (cheap)
   infoMask += OpenMM::State::Energy;     // for pot. energy (more
expensive)
}
// Forces are also available (and cheap).

const OpenMM::State state = omm->context->getState(infoMask);
```

```
timeInPs = state.getTime(); // OpenMM time is in ps already

// Copy OpenMM positions into atoms array and change units from nm to
Angstroms.
const std::vector<Vec3>& positionsInNm = state.getPositions();
for (int i=0; i < (int)positionsInNm.size(); ++i)
    for (int j=0; j < 3; ++j)
         atoms[i].posInAng[j] = positionsInNm[i][j] *
OpenMM::AngstromsPerNm;

// If energy has been requested, obtain it and convert from kJ to kcal.
energyInKcal = 0;
if (wantEnergy)
   energyInKcal = (state.getPotentialEnergy() + state.getKineticEnergy())
                  * OpenMM::KcalPerKJ;
```

#### 5.4.2.6 *myStepWithOpenMM*

The `myStepWithOpenMM` routine takes the handle, uses it to find the Integrator, and then sets the number of steps for the Integrator to take. It does not return any values.

```
static void
myStepWithOpenMM(MyOpenMMData* omm, int numSteps) {
    omm->integrator->step(numSteps);
}
```

#### 5.4.2.7 *myTerminateOpenMM*

The `myTerminateOpenMM` routine takes the handle and deletes all the components, e.g., the Context and System, cleaning up the heap space.

```
static void
myTerminateOpenMM(MyOpenMMData* omm) {
    delete omm;
}
```

## 5.5 HelloEthane Program

The HelloEthane program simulates ethane (H3-C-C-H3) in a vacuum. It is structured similarly to the HelloSodiumChloride example, but includes bonded forces (bond stretch, bond angle bend, dihedral torsion). In setting up these bonded forces, the program illustrates some of the other inconsistencies in definitions and units that you should watch out for.

The bonded forces are added to the system within the initialization interface routine, similar to how the non-bonded forces were added in the HelloSodiumChloride example:

```
// Create a System and Force objects within the System. Retain a reference
// to each force object so we can fill in the forces. Note: the System
owns
// the force objects and will take care of deleting them; don't do it
yourself!
OpenMM::System&                 system      = *(omm->system = new
OpenMM::System());
OpenMM::NonbondedForce&         nonbond     = *new
OpenMM::NonbondedForce();
OpenMM::HarmonicBondForce&      bondStretch = *new
OpenMM::HarmonicBondForce();
OpenMM::HarmonicAngleForce&     bondBend    = *new
OpenMM::HarmonicAngleForce();
OpenMM::PeriodicTorsionForce&   bondTorsion = *new
OpenMM::PeriodicTorsionForce();
    system.addForce(&nonbond);
    system.addForce(&bondStretch);
    system.addForce(&bondBend);
    system.addForce(&bondTorsion);
```

***Constrainable and non-constrainable bonds:*** In the initialization routine, we also set up the bonds. If constraints are being used, then we tell the System about the constrainable bonds:

```
    std::vector< std::pair<int,int> > bondPairs;
    for (int i=0; bonds[i].type != EndOfList; ++i) {
        const int*      atom = bonds[i].atoms;
        const BondType& bond = bondType[bonds[i].type];

        if (UseConstraints && bond.canConstrain) {
            system.addConstraint(atom[0], atom[1],
                                 bond.nominalLengthInAngstroms
                                 * OpenMM::NmPerAngstrom);
        }
```

Otherwise, we need to give the HarmonicBondForce the bond stretch parameters.

***Warning****:* The constant used to specify the stiffness may be defined differently between the existing MD code and OpenMM. For instance, AMBER uses the constant, as given in the harmonic *energy* term $kx^2$, where the force is $2kx$ (k = constant and x = distance). OpenMM wants the constant, as used in the *force* term kx (with energy $0.5 * kx^2$). So a factor of 2

must be introduced when setting the bond stretch parameters in an OpenMM system using data from an AMBER system.

```
bondStretch.addBond(atom[0], atom[1],
                    bond.nominalLengthInAngstroms
                        * OpenMM::NmPerAngstrom,
                    bond.stiffnessInKcalPerAngstrom2
                        * 2 * OpenMM::KJPerKcal
                        * OpenMM::AngstromsPerNm *
                        OpenMM::AngstromsPerNm);
```

***Non-bond exclusions:*** Next, we deal with non-bond exclusions. These are used for pairs of atoms that appear close to one another in the network of bonds in a molecule. For atoms that close, normal non-bonded forces do not apply or are reduced in magnitude. First, we create a list of bonds to generate the non-bond exclusions:

```
bondPairs.push_back(std::make_pair(atom[0], atom[1]));
```

OpenMM's non-bonded force provides a convenient routine for creating the common exceptions. These are: (1) for atoms connected by one bond (1-2) or connected by just one additional bond (1-3), Coulomb and van der Waals terms do not apply; and (2) for atoms connected by three bonds (1-4), Coulomb and van der Waals terms apply but are reduced by a force-field dependent scale factor. In general, you may introduce additional exceptions, but the standard ones suffice here and in many other circumstances.

```
// Exclude 1-2, 1-3 bonded atoms from nonbonded forces, and scale down 1-4
bonded atoms.
nonbond.createExceptionsFromBonds(bondPairs, Coulomb14Scale,
LennardJones14Scale);

// Create the 1-2-3 bond angle harmonic terms.
for (int i=0; angles[i].type != EndOfList; ++i) {
     const int*       atom  = angles[i].atoms;
     const AngleType& angle = angleType[angles[i].type];

// See note under bond stretch above regarding the factor of 2 here.
bondBend.addAngle(atom[0],atom[1],atom[2],
                  angle.nominalAngleInDegrees     *
                  OpenMM::RadiansPerDegree,
                  angle.stiffnessInKcalPerRadian2 * 2 *
                  OpenMM::KJPerKcal);
}

// Create the 1-2-3-4 bond torsion (dihedral) terms.
for (int i=0; torsions[i].type != EndOfList; ++i) {
     const int*         atom = torsions[i].atoms;
```

```
    const TorsionType& torsion = torsionType[torsions[i].type];
    bondTorsion.addTorsion(atom[0],atom[1],atom[2],atom[3],
            torsion.periodicity,
            torsion.phaseInDegrees  * OpenMM::RadiansPerDegree,
            torsion.amplitudeInKcal * OpenMM::KJPerKcal);
}
```

The rest of the code is similar to the HelloSodiumChloride example and will not be covered in detail here. Please refer to the program HelloEthane.cpp itself, which is well-commented, for additional details.

# 6 Platform-Specific Properties

When creating a Context, you can specify values for properties specific to a particular Platform. This is used to control how calculations are done in ways that are outside the scope of the generic OpenMM API.

To do this, pass both the Platform object and a map of property values to the Context constructor:

```
Platform& platform = Platform::getPlatformByName("OpenCL");
map<string, string> properties;
properties["OpenCLDeviceIndex"] = "1";
Context context(system, integrator, platform, properties);
```

After a Context is created, you can use the Platform's `getPropertyValue()` method to query the values of properties.

## 6.1 OpenCL Platform

The OpenCL Platform recognizes the following Platform-specific properties:

- OpenCLPlatformIndex: When multiple OpenCL implementations are installed on your computer, this is used to select which one to use. The value is the zero-based index of the platform (in the OpenCL sense, not the OpenMM sense) to use, in the order they are returned by the OpenCL platform API. This is useful, for example, in selecting whether to use a GPU or CPU based OpenCL implementation.

- OpenCLDeviceIndex: When multiple OpenCL devices are available on your computer, this is used to select which one to use. The value is the zero-based index of the device to use, in the order they are returned by the OpenCL device API.

The OpenCL Platform also supports parallelizing a simulation across multiple GPUs. To do that, set the OpenCLDeviceIndex property to a comma separated list of values. For example,

```
properties["OpenCLDeviceIndex"] = "0,1";
```

This tells it use both devices 0 and 1, splitting the work between them.

## 6.2 CUDA Platform

The CUDA Platform recognizes the following Platform-specific properties:

- CudaDevice: When multiple CUDAdevices are available on your computer, this is used to select which one to use. The value is the zero-based index of the device to use, in the order they are returned by the CUDA API.
- CudaUseBlockingSync: This is used to control how the CUDA runtime synchronizes between the CPU and GPU. If this is set to "true" (the default), CUDA will allow the calling thread to sleep while the GPU is performing a computation, allowing the CPU to do other work. If it is set to "false", CUDA will spin-lock while the GPU is working. This can improve performance slightly, but also prevents the CPU from doing anything else while the GPU is working.

# 7 Using OpenMM with Software Written in Languages Other than C++

Although the native OpenMM API is object-oriented C++ code, it is possible to directly translate the interface so that it is callable from C, Fortran 95, and Python with no substantial conceptual changes. We have developed a straightforward mapping for these languages that, while perhaps not the most elegant possible, has several advantages:

- Almost all documentation, training, forum discussions, and so on are equally useful to users of all these languages. There are syntactic differences of course, but all the important concepts remain unchanged.
- We are able to generate the C, Fortran, and Python APIs from the C++ API. Obviously, this reduces development effort, but more importantly it means that the APIs are likely to be error-free and are always available immediately when the native API is updated.
- Because OpenMM performs expensive operations "in bulk" there is no noticeable overhead in accessing these operations through the C, Fortran, or Python APIs.
- All symbols introduced to a C or Fortran program begin with the prefix "`OpenMM_`" so will not interfere with symbols already in use.

*Availability of APIs in other languages:* All necessary C and Fortran bindings are built in to the main OpenMM library; no separate library is required. The Python wrappers are contained in a module that is distributed with OpenMM and that can be installed by executing its setup.py script in the standard way.

(This doesn't apply to most users: if you are building your own OpenMM from source using CMake and want the API bindings generated, be sure to enable the `OPENMM_BUILD_C_AND_FORTRAN_WRAPPERS` option for C and Fortran, or `OPENMM_BUILD_PYTHON_WRAPPERS` option for Python. The Python module will be placed in a subdirectory of your main build directory called "python")

*Documentation for APIs in other languages:* While there is extensive Doxygen documentation available for the C++ API, there is no separate on-line documentation for the C and Fortran API. Instead, you should use the C++ documentation, employing the mappings described here to figure out the equivalent syntax in C or Fortran. Documentation for the Python API is included in the module through the standard help and pydoc interfaces.

## 7.1 C API

Before you start writing your own C program that calls OpenMM, be sure you can build and run the two C examples that are supplied with OpenMM (see Chapter 5). These can be built from the supplied `Makefile` on Linux and Mac, or supplied `NMakefile` and Visual Studio solution files on Windows.

The example programs are `HelloArgonInC` and `HelloSodiumChlorideInC`. The argon example serves as a quick check that your installation is set up properly and you know how to build a C program that is linked with OpenMM. It will also tell you whether OpenMM is executing on the GPU or is running (slowly) on the Reference platform. However, the argon example is not a good template to follow for your own programs. The sodium chloride example, though necessarily simplified, is structured roughly in the way we recommended you set up your own programs to call OpenMM. Please be sure you have both of these programs executing successfully on your machine before continuing.

### 7.1.1 Mechanics of using the C API

The C API is generated automatically from the C++ API when OpenMM is built. There are two resulting components: C bindings (functions to call), and C declarations (in a header file). The C bindings are small `extern` (global) interface functions, one for every method of

every OpenMM class, whose signatures (name and arguments) are predictable from the class name and method signatures. There are also "helper" types and functions provided for the few cases in which the C++ behavior cannot be directly mapped into C. These interface and helper functions are compiled in to the main OpenMM library so there is nothing special you have to do to get access to them.

In the `/include` subdirectory of your OpenMM installation directory, there is a machine-generated header file `OpenMMCWrapper.h` that should be #included in any C program that is to make calls to OpenMM functions. That header contains declarations for all the OpenMM C interface functions and related types. Note that if you follow our suggested structure, you will not need to include this file in your `main()` compilation unit but can instead use it only in a local file that you write to provide a simple interface to your existing code (see Chapter 5).

### 7.1.2 Mapping from the C++ API to the C API

The automated generator of the C "wrappers" follows the translation strategy shown in Table 7.1. The idea is that if you see the construct on the left in the C++ API documentation, you should interpret it as the corresponding construct on the right in C. Please look at the supplied example programs to see how this is done in practice.

| C++ API declaration | | Equivalent in C API |
|---|---|---|
| namespace | `OpenMM::` | `OpenMM_` (prefix) |
| class | `class OpenMM::ClassName` | `typedef OpenMM_ClassName` |
| constant | `OpenMM::RadiansPerDeg` | `OpenMM_RadiansPerDeg` (static constant) |
| class enum | `OpenMM::State::Positions` | `OpenMM_State_Positions` |
| constructor | `new OpenMM::ClassName()` | `OpenMM_ClassName* OpenMM_ClassName_create()`<br>(addl. constructors are `_create_2()`, etc.) |
| destructor | `OpenMM::ClassName* thing;`<br>`delete thing;` | `OpenMM_ClassName* thing;`<br>`OpenMM_ClassName_destroy(thing);` |
| class method | `OpenMM::ClassName* thing;`<br>`thing->someName(`*args*`)` | `OpenMM_ClassName* thing;`<br>`OpenMM_ClassName_someName (thing, `*args*`)` |
| Boolean type & constants | `bool`<br>`true, false` | `OpenMM_Boolean`<br>`OpenMM_True` (1), `OpenMM_False` (0) |
| string | `std::string` | `char*` |
| 3-vector | `OpenMM::Vec3` | `typedef OpenMM_Vec3` |
| arrays | `std::vector<std::string>`<br>`std::vector<double>`<br>`std::vector<Vec3>`<br>`std::vector<std::pair<int,int>>`<br>`std::map<std::string,double>` | `typedef OpenMM_StringArray`<br>`typedef OpenMM_DoubleArray`<br>`typedef OpenMM_Vec3Array`<br>`typedef OpenMM_BondArray`<br>`typedef OpenMM_ParameterArray` |

**Table 7.1: Default mapping of objects from the C++ API to the C API**

There are some exceptions to the generic translation rules shown in the table; they are enumerated in the next section. And because there are no C++ API equivalents to the array types, they are described in detail below.

### 7.1.3 Exceptions

These two methods are handled somewhat differently in the C API than in the C++ API:

- **`OpenMM::Context::getState()`**

  The C version, `OpenMM_Context_getState()`, returns a pointer to a heap allocated `OpenMM_State` object. You must then explicitly destroy this `State` object when you are done with it, by calling `OpenMM_State_destroy()`.

- **`OpenMM::Platform::loadPluginsFromDirectory()`**

  The C version `OpenMM_Platform_loadPluginsFromDirectory()` returns a heap-allocated `OpenMM_StringArray` object containing a list of all the file names that were successfully loaded. You must then explicitly destroy this `StringArray` object when you are done with it. Do not ignore the return value; if you do you'll have a memory leak since the `StringArray` will still be allocated.

(In the C++ API, the equivalent methods return references into existing memory rather than new heap-allocated memory, so the returned objects do not need to be destroyed.)

### 7.1.4 OpenMM_Vec3 helper type

Unlike the other OpenMM objects which are opaque and manipulated via pointers, the C API provides an explicit definition for the C `OpenMM_Vec3` type that is compatible with the `OpenMM::Vec3` type. The definition of `OpenMM_Vec3` is:

```
typedef struct {double x, y, z;} OpenMM_Vec3;
```

You can work directly with the individual fields of this type from your C program if you want. For convenience, a scale() function is provided that creates a new OpenMM_Vec3 from an old one and a scale factor:

```
OpenMM_Vec3 OpenMM_Vec3_scale(const OpenMM_Vec3 vec, double scale);
```

### 7.1.5 Array helper types

C++ has built-in container types `std::vector` and `std::map` which OpenMM uses to manipulate arrays of objects. These don't have direct equivalents in C, so we supply special array types for each kind of object for which OpenMM creates containers. These are: string, double, Vec3, bond, and parameter map. See Table 7.1 for the names of the C types for each of these object arrays. Each of the array types provides these functions (prefixed by `OpenMM_` and the actual *Thing* name), with the syntax shown conceptually since it differs slightly for each kind of object.

| | |
|---|---|
| *Thing*`Array* create(int size)` | Create a heap-allocated array of *Things*, with space pre-allocated to hold `size` of them. You can start at `size==0` if you want since these arrays are dynamically resizeable. |
| `void destroy(`*Thing*`Array*)` | Free the heap space that is currently in use for the passed-in array of *Things*. |
| `int getSize(`*Thing*`Array*)` | Return the current number of *Things* in this array. This means you can `get()` and `set()` elements up to `getSize()-1`. |
| `void resize(`*Thing*`Array*,int size)` | Change the size of this array to the indicated value which may be smaller or larger than the current size. Existing elements remain in their same locations as long as they still fit. |
| `void append(`*Thing*`Array*,`*Thing*`)` | Add a *Thing* to the end of the array, increasing the array size by one. The precise syntax depends on the actual type of *Thing*; see below. |
| `void set(`*Thing*`Array*, int index,`*Thing*`)` | Store a copy of *Thing* in the indicated element of the array (indexed from 0). The array must be of length at least `index+1`; you can't grow the array with this function. |
| *Thing* `get(`*Thing*`Array*, int index)` | Retrieve a particular element from the array (indexed from 0). (For some Things the value is returned in arguments rather than as the function return.) |

**Table 7.2: Generic description of array helper types**

Here are the exact declarations with deviations from the generic description noted, for each of the array types.

#### *7.1.5.1 OpenMM_DoubleArray*

```
OpenMM_DoubleArray*
           OpenMM_DoubleArray_create(int size);
```

```
void        OpenMM_DoubleArray_destroy(OpenMM_DoubleArray*);
int         OpenMM_DoubleArray_getSize(const OpenMM_DoubleArray*);
void        OpenMM_DoubleArray_resize(OpenMM_DoubleArray*, int size);
void        OpenMM_DoubleArray_append(OpenMM_DoubleArray*, double value);
void        OpenMM_DoubleArray_set(OpenMM_DoubleArray*, int index, double value);
double      OpenMM_DoubleArray_get(const OpenMM_DoubleArray*, int index);
```

#### *7.1.5.2 OpenMM_StringArray*

```
OpenMM_StringArray*
            OpenMM_StringArray_create(int size);
void        OpenMM_StringArray_destroy(OpenMM_StringArray*);
int         OpenMM_StringArray_getSize(const OpenMM_StringArray*);
void        OpenMM_StringArray_resize(OpenMM_StringArray*, int size);
void        OpenMM_StringArray_append(OpenMM_StringArray*, const char* string);
void        OpenMM_StringArray_set(OpenMM_StringArray*, int index, const char* string);
const char* OpenMM_StringArray_get(const OpenMM_StringArray*, int index);
```

#### *7.1.5.3 OpenMM_Vec3Array*

```
OpenMM_Vec3Array*
            OpenMM_Vec3Array_create(int size);
void        OpenMM_Vec3Array_destroy(OpenMM_Vec3Array*);
int         OpenMM_Vec3Array_getSize(const OpenMM_Vec3Array*);
void        OpenMM_Vec3Array_resize(OpenMM_Vec3Array*, int size);
void        OpenMM_Vec3Array_append(OpenMM_Vec3Array*, const OpenMM_Vec3 vec);
void        OpenMM_Vec3Array_set(OpenMM_Vec3Array*, int index, const OpenMM_Vec3 vec);
const OpenMM_Vec3*
            OpenMM_Vec3Array_get(const OpenMM_Vec3Array*, int index);
```

#### *7.1.5.4 OpenMM_BondArray*

Note that bonds are specified by pairs of integers (the atom indices). The `get()` method returns those in a pair of final arguments rather than as its functional return.

```
OpenMM_BondArray*
            OpenMM_BondArray_create(int size);
void        OpenMM_BondArray_destroy(OpenMM_BondArray*);
int         OpenMM_BondArray_getSize(const OpenMM_BondArray*);
void        OpenMM_BondArray_resize(OpenMM_BondArray*, int size);
void        OpenMM_BondArray_append(OpenMM_BondArray*, int particle1, int particle2);
void        OpenMM_BondArray_set(OpenMM_BondArray*, int index, int particle1, int particle2);
void        OpenMM_BondArray_get(const OpenMM_BondArray*, int index,
                                 int* particle1, int* particle2);
```

#### *7.1.5.5 OpenMM_ParameterArray*

OpenMM returns references to internal `ParameterArrays` but does not support user-created `ParameterArrays`, so only the `get()` and `getSize()` functions are available. Also, note that since this is actually a map rather than an array, the "index" is the *name* of the parameter rather than its ordinal.

```
int         OpenMM_ParameterArray_getSize(const OpenMM_ParameterArray*);
double      OpenMM_ParameterArray_get(const OpenMM_ParameterArray*, const char* name);
```

## 7.2 Fortran 95 API

Before you start writing your own Fortran program that calls OpenMM, be sure you can build and run the two Fortran examples that are supplied with OpenMM (see Chapter 5). These can be built from the supplied `Makefile` on Linux and Mac, or supplied `NMakefile` and Visual Studio solution files on Windows.

The example programs are `HelloArgonInFortran` and `HelloSodiumChlorideInFortran`. The argon example serves as a quick check that your installation is set up properly and you know how to build a Fortran program that is linked with OpenMM. It will also tell you whether OpenMM is executing on the GPU or is running (slowly) on the Reference platform. However, the argon example is not a good template to follow for your own programs. The sodium chloride example, though necessarily simplified, is structured roughly in the way we recommended you set up your own programs to call OpenMM. Please be sure you have both of these programs executing successfully on your machine before continuing.

### 7.2.1 Mechanics of using the Fortran API

The Fortran API is generated automatically from the C++ API when OpenMM is built. There are two resulting components: Fortran bindings (subroutines to call), and Fortran declarations of types and subroutines (in the form of a Fortran 95 module file). The Fortran bindings are small interface subroutines, one for every method of every OpenMM class, whose signatures (name and arguments) are predictable from the class name and method signatures. There are also "helper" types and subroutines provided for the few cases in which the C++ behavior cannot be directly mapped into Fortran. These interface and helper subroutines are compiled in to the main OpenMM library so there is nothing special you have to do to get access to them.

Because Fortran is case-insensitive, calls to Fortran subroutines (however capitalized) are mapped by the compiler into all-lowercase or all-uppercase names, and different compilers use different conventions. The automatically-generated OpenMM Fortran "wrapper" subroutines, which are generated in C and thus case-sensitive, are provided in two forms for compatibility with the majority of Fortran compilers, including Intel Fortran and gfortran.

The two forms are: (1) all-lowercase with a trailing underscore, and (2) all-uppercase without a trailing underscore. So regardless of the Fortran compiler you are using, it should find a suitable subroutine to call in the main OpenMM library.

In the `/include` subdirectory of your OpenMM installation directory, there is a machine-generated module file `OpenMMFortranModule.f90` that must be compiled along with any Fortran program that is to make calls to OpenMM functions. (You can look at the `Makefile` or Visual Studio solution file provided with the OpenMM examples to see how to build a program that uses this module file.) This module file contains definitions for two modules: `MODULE OpenMM_Types` and `MODULE OpenMM`; however, only the `OpenMM` module will appear in user programs (it references the other module internally). The modules contain declarations for all the OpenMM Fortran interface subroutines, related types, and parameters (constants). Note that if you follow our suggested structure, you will not need to `use` the `OpenMM` module in your `main()` compilation unit but can instead use it only in a local file that you write to provide a simple interface to your existing code (see Chapter 5).

### 7.2.2 Mapping from the C++ API to the Fortran API

The automated generator of the Fortran "wrappers" follows the translation strategy shown in Table 7.3. The idea is that if you see the construct on the left in the C++ API documentation, you should interpret it as the corresponding construct on the right in Fortran. Please look at the supplied example programs to see how this is done in practice. Note that all subroutines and modules are declared with "`implicit none`", meaning that the type of every symbol is declared explicitly and should not be inferred from the first letter of the symbol name.

| C++ API declaration | | Equivalent in Fortran API |
|---|---|---|
| namespace | `OpenMM::` | `OpenMM_` (prefix) |
| class | `class OpenMM::ClassName` | `type (OpenMM_ClassName)` |
| constant | `OpenMM::RadiansPerDeg` | `parameter (OpenMM_RadiansPerDeg)` |
| class enum | `OpenMM::State::Positions` | `parameter (OpenMM_State_Positions)` |
| constructor | `new OpenMM::ClassName()` | `type (OpenMM_ClassName) thing`<br>`call OpenMM_ClassName_create(thing)`<br>(addl. constructors are `_create_2()`, etc.) |
| destructor | `OpenMM::ClassName* thing;`<br>`delete thing;` | `type (OpenMM_ClassName) thing`<br>`call OpenMM_ClassName_destroy(thing)` |
| class method | `OpenMM::ClassName* thing;`<br>`thing->someName(`*args*`)` | `type (OpenMM_ClassName) thing`<br>`call OpenMM_ClassName_someName (thing, `*args*`)` |
| Boolean type & constants | `bool`<br>`true, false` | `integer*4`<br>`parameter (OpenMM_True=1)`<br>`parameter (OpenMM_False=0)` |
| string | `std::string` | `character(*)` |
| 3-vector | `OpenMM::Vec3` | `real*8 vec(3)` |
| arrays | `std::vector<std::string>`<br>`std::vector<double>`<br>`std::vector<Vec3>`<br>`std::vector<std::pair<int,int>>`<br>`std::map<std::string,double>` | `type (OpenMM_StringArray)`<br>`type (OpenMM_DoubleArray)`<br>`type (OpenMM_Vec3Array)`<br>`type (OpenMM_BondArray)`<br>`type (OpenMM_ParameterArray)` |

**Table 7.3: Default mapping of objects from the C++ API to the Fortran API**

Because there are no C++ API equivalents to the array types, they are described in detail below.

### 7.2.3 OpenMM_Vec3 helper type

Unlike the other OpenMM objects which are opaque and manipulated via pointers, the Fortran API uses an ordinary `real*8(3)` array in place of the `OpenMM::Vec3` type. The You can work directly with the individual elements of this type from your Fortran program if you want. For convenience, a `scale()` function is provided that creates a new Vec3 from an old one and a scale factor:

```
subroutine OpenMM_Vec3_scale(vec, scale, result)
real*8 vec(3), scale, result(3)
```

No explicit `type(OpenMM_Vec3)` is provided in the Fortran API since it is not needed.

### 7.2.4 Array helper types

C++ has built-in container types `std::vector` and `std::map` which OpenMM uses to manipulate arrays of objects. These don't have direct equivalents in Fortran, so we supply

special array types for each kind of object for which OpenMM creates containers. These are: string, double, Vec3, bond, and parameter map. See Table 7.3 for the names of the Fortran types for each of these object arrays. Each of the array types provides these functions (prefixed by `OpenMM_` and the actual *Thing* name), with the syntax shown conceptually since it differs slightly for each kind of object.

| | |
|---|---|
| `subroutine create(array,size)`<br>`type (OpenMM_`*Thing*`Array) array`<br>`integer*4 size` | Create a heap-allocated array of *Things*, with space pre-allocated to hold `size` of them. You can start at `size==0` if you want since these arrays are dynamically resizeable. |
| `subroutine destroy(array)`<br>`type (OpenMM_`*Thing*`Array) array` | Free the heap space that is currently in use for the passed-in array of *Things*. |
| `function getSize(array)`<br>`type (OpenMM_`*Thing*`Array) array`<br>`integer*4 getSize` | Return the current number of *Things* in this array. This means you can `get()` and `set()` elements up to `getSize()`. |
| `subroutine resize(array,size)`<br>`type (OpenMM_`*Thing*`Array) array`<br>`integer*4 size` | Change the size of this array to the indicated value which may be smaller or larger than the current size. Existing elements remain in their same locations as long as they still fit. |
| `subroutine append(array,elt)`<br>`type (OpenMM_`*Thing*`Array) array`<br>*Thing* `elt` | Add a *Thing* to the end of the array, increasing the array size by one. The precise syntax depends on the actual type of *Thing*; see below. |
| `subroutine set(array,index,elt)`<br>`type (OpenMM_`*Thing*`Array) array`<br>`integer*4 index`<br>*Thing* `elt` | Store a copy of `elt` in the indicated element of the array (indexed from 1). The array must be of length at least `index`; you can't grow the array with this function. |
| `subroutine get(array,index,elt)`<br>`type (OpenMM_`*Thing*`Array) array`<br>`integer*4 index`<br>*Thing* `elt` | Retrieve a particular element from the array (indexed from 1). Some *Things* require more than one argument to return. |

**Table 7.4: Generic description of array helper types**

Here are the exact declarations with deviations from the generic description noted, for each of the array types.

#### *7.2.4.1 OpenMM_DoubleArray*

```
subroutine OpenMM_DoubleArray_create(array, size)
    integer*4 size
    type (OpenMM_DoubleArray) array
subroutine OpenMM_DoubleArray_destroy(array)
    type (OpenMM_DoubleArray) array
```

```
function OpenMM_DoubleArray_getSize(array)
    type (OpenMM_DoubleArray) array
    integer*4 OpenMM_DoubleArray_getSize
subroutine OpenMM_DoubleArray_resize(array, size)
    type (OpenMM_DoubleArray) array
    integer*4 size
subroutine OpenMM_DoubleArray_append(array, value)
    type (OpenMM_DoubleArray) array
    real*8 value
subroutine OpenMM_DoubleArray_set(array, index, value)
    type (OpenMM_DoubleArray) array
    integer*4 index
    real*8 value
subroutine OpenMM_DoubleArray_get(array, index, value)
    type (OpenMM_DoubleArray) array
    integer*4 index
    real*8 value
```

#### 7.2.4.2 OpenMM_StringArray

```
subroutine OpenMM_StringArray_create(array, size)
    integer*4 size
    type (OpenMM_StringArray) array
subroutine OpenMM_StringArray_destroy(array)
    type (OpenMM_StringArray) array
function OpenMM_StringArray_getSize(array)
    type (OpenMM_StringArray) array
    integer*4 OpenMM_StringArray_getSize
subroutine OpenMM_StringArray_resize(array, size)
    type (OpenMM_StringArray) array
    integer*4 size
subroutine OpenMM_StringArray_append(array, str)
    type (OpenMM_StringArray) array
    character(*) str
subroutine OpenMM_StringArray_set(array, index, str)
    type (OpenMM_StringArray) array
    integer*4 index
    character(*) str
subroutine OpenMM_StringArray_get(array, index, str)
    type (OpenMM_StringArray) array
    integer*4 index
    character(*)str
```

#### 7.2.4.3 OpenMM_Vec3Array

```
subroutine OpenMM_Vec3Array_create(array, size)
    integer*4 size
    type (OpenMM_Vec3Array) array
subroutine OpenMM_Vec3Array_destroy(array)
    type (OpenMM_Vec3Array) array
function OpenMM_Vec3Array_getSize(array)
    type (OpenMM_Vec3Array) array
    integer*4 OpenMM_Vec3Array_getSize
subroutine OpenMM_Vec3Array_resize(array, size)
    type (OpenMM_Vec3Array) array
    integer*4 size
subroutine OpenMM_Vec3Array_append(array, vec)
    type (OpenMM_Vec3Array) array
    real*8 vec(3)
subroutine OpenMM_Vec3Array_set(array, index, vec)
    type (OpenMM_Vec3Array) array
    integer*4 index
    real*8 vec(3)
subroutine OpenMM_Vec3Array_get(array, index, vec)
    type (OpenMM_Vec3Array) array
    integer*4 index
    real*8 vec (3)
```

#### *7.2.4.4 OpenMM_BondArray*

Note that bonds are specified by pairs of integers (the atom indices). The `get()` method returns those in a pair of final arguments rather than as its functional return.

```
subroutine OpenMM_BondArray_create(array, size)
    integer*4 size
    type (OpenMM_BondArray) array
subroutine OpenMM_BondArray_destroy(array)
    type (OpenMM_BondArray) array
function OpenMM_BondArray_getSize(array)
    type (OpenMM_BondArray) array
    integer*4 OpenMM_BondArray_getSize
subroutine OpenMM_BondArray_resize(array, size)
    type (OpenMM_BondArray) array
    integer*4 size
subroutine OpenMM_BondArray_append(array, particle1, particle2)
    type (OpenMM_BondArray) array
    integer*4 particle1, particle2
subroutine OpenMM_BondArray_set(array, index, particle1, particle2)
    type (OpenMM_BondArray) array
    integer*4 index, particle1, particle2
subroutine OpenMM_BondArray_get(array, index, particle1, particle2)
    type (OpenMM_BondArray) array
    integer*4 index, particle1, particle2
```

#### *7.2.4.5 OpenMM_ParameterArray*

OpenMM returns references to internal `ParameterArrays` but does not support user-created `ParameterArrays`, so only the `get()` and `getSize()` functions are available. Also, note that since this is actually a map rather than an array, the "index" is the *name* of the parameter rather than its ordinal.

```
function OpenMM_ParameterArray_getSize(array)
    type (OpenMM_ParameterArray) array
    integer*4 OpenMM_ParameterArray_getSize
subroutine OpenMM_ParameterArray_get(array, name, param)
    type (OpenMM_ParameterArray) array
    character(*) name
    character(*) param
```

## 7.3 Python API

### 7.3.1 Installing the Python API

There are currently two types of packages for installing the Python API. One contains wrapper source code for Unix-type machines (including Linux and Mac operating systems). You will need a C++ compiler to install it using this type of package. The other type of

installation package is a binary package which contains compiled wrapper code for Windows machines (no compilers are needed to install binary packages).

#### *7.3.1.1 Installing on Windows*

OpenMM only works with Python 2.6 or 2.7, so make sure that one of those versions is installed before you try installing. For Python installation packages and instructions, go to http://python.org. Note that if you have a 64-bit machine, you should still install the 32-bit version of Python since the OpenMM Python API binary is 32-bit. We suggest that you install Python using the default options.

Double click on the Python API Installer icon, located in the top level directory for the OpenMM installation (by default, this is C:\Program Files\OpenMM). This will install the OpenMM package into the Python installation area. If you have more than one Python installation, you will be asked which Python to use—make sure to select Python 2.6 or 2.7.

#### *7.3.1.2 Installing on Linux and Mac*

Make sure you have Python 2.6 or 2.7 installed. For Python installation packages and instructions, go to http://python.org. If you do not have the correct Python version, install a valid version using the default options. Most versions of Linux and Mac OS X have a suitable Python preinstalled. You can check by typing "`python --version`" in a terminal window.

You must have a C++ compiler to install the OpenMM Python API. If you are using a Mac, install Apple's Xcode development tools (http://developer.apple.com/TOOLS/Xcode) to get the needed compiler. On other Unix-type systems, install gcc-c++ (version 4.0 or later).

The install.sh script installs the Python API automatically as part of the installation process, so you probably already have it installed. If for some reason you need to install it manually, you can do that with the `setup.py` script included with OpenMM. Before executing this script, you must set two environment variables: `OPENMM_INCLUDE_PATH` must point to the directory containing OpenMM header files, and `OPENMM_LIB_PATH` must point to the directory containing OpenMM library files. Assuming OpenMM is installed in the default location (`/usr/local/openmm`), you would type the following commands. Note that if you

are using the system Python (as opposed to a locally installed version), you may need to run the Python installation process as a superuser using the `sudo` command:

```
export OPENMM_INCLUDE_PATH=/usr/local/openmm/include
export OPENMM_LIB_PATH=/usr/local/openmm/lib
python setup.py build
python setup.py install   OR   sudo python setup.py install
```

If you are compiling OpenMM from source, you can also install by building the "PythonInstall" target:

```
make PythonInstall   OR   sudo make PythonInstall
```

### 7.3.2 Mapping from the C++ API to the Python API

The Python API follows the C++ API as closely as possible. There are three notable differences:

1) The `getState()` method in the `Context` class takes Pythonic-type arguments to indicate which state variables should be made available. For example:

   ```
   myContext.getState(getEnergy=True, getForce=False, …)
   ```

2) Wherever the C++ API uses references to return multiple values from a method, the Python API returns a tuple. For example, in C++ you would query a HarmonicBondForce for a bond's parameters as follows:

   ```
   int particle1, particle2;
   double length, k;
   f.getBondParameters(i, particle1, particle2, length, k);
   ```

   In Python, the equivalent code is:

   ```
   [particle1, particle2, length, k] = f.getBondParameters(i)
   ```

3) Unlike C++, the Python API accepts and returns quantities with units attached to most values (see the "Units and dimensional analysis" section below for details). In

short, this means that while values in C++ have *implicit* units, the Python API returns objects that have values and *explicit* units.

### 7.3.3 Mechanics of using the Python API

When using the Python API, be sure to include the OpenMM and GPU support libraries in your library path, just as you would for a C++ application. This is set with the `LD_LIBRARY_PATH` environment variable on Linux, `DYLD_LIBRARY_PATH` on Mac, or `PATH` on Windows. See sections 3.3 and 3.4 for details.

The Python API is contained in the simtk.openmm package, while the units code is contained in the simtk.units package. (The application layer, described in the Application Guide, is contained in the simtk.openmm.app package.) A program using it will therefore typically begin

```
import simtk.openmm as mm
import simtk.unit as unit
```

Creating and using OpenMM objects is then done exactly as in C++:

```
system = mm.System()
nb = mm.NonbondedForce()
nb.setNonbondedMethod(mm.NonbondedForce.CutoffNonPeriodic)
nb.setCutoffDistance(1.2*unit.nanometer)
system.addForce(nb)
```

Note that when setting the cutoff distance, we explicitly specify that it is in nanometers. We could just as easily specify it in different units:

```
nb.setCutoffDistance(12*unit.angstrom)
```

The use of units in OpenMM is discussed in the next section.

### 7.3.4 Units and dimensional analysis

#### *7.3.4.1 Why does the Python API include units?*

The C++ API for OpenMM uses an *implicit* set of units for physical quantities such as lengths, masses, energies, etc. These units are based on daltons, nanometers, and picoseconds for the mass, length, and time dimensions, respectively. When using the C++ API, it is very important to ensure that quantities being manipulated are always expressed in terms of these units. For example, if you read in a distance in Angstroms, you must multiply that distance by a conversion factor to turn it into nanometers before using it in the C++ API. Such conversions can be a source of tedium and errors. This is true in many areas of scientific programming. Units confusion was blamed for the loss of the Mars Climate Orbiter spacecraft in 1999, at a cost of more than $100 million. Units were introduced in the Python API to minimize the chance of such errors.

The Python API addresses the potential problem of conversion errors by using quantities with explicit units. If a particular distance is expressed in Angstroms, the Python API will know that it is in Angstroms. When the time comes to call the C++ API, it will understand that the quantity must be converted to nanometers. You, the programmer, must declare upfront that the quantity is in Angstrom units, and the API will take care of the details from then on. Using explicit units is a bit like brushing your teeth: it requires some effort upfront, but it probably saves you trouble in the long run.

#### *7.3.4.2 Quantities, units, and dimensions*

The explicit unit system is based on three concepts: Dimensions, Units, and Quantities.

Dimensions are measurable physical concepts such as mass, length, time, and energy. Energy is actually a composite dimension based on mass, length, and time.

A Unit defines a linear scale used to measure amounts of a particular physical Dimension. Examples of units include meters, seconds, joules, inches, and grams.

A Quantity is a specific amount of a physical Dimension. An example of a quantity is “0.63 kilograms”. A Quantity is expressed as a combination of a value (e.g., 0.63), and a Unit (e.g., kilogram). The same Quantity can be expressed in different Units.

The set of BaseDimensions defined in the simtk.unit module includes:

- mass
- length
- time
- temperature
- amount
- charge
- luminous intensity
- angle

These are not precisely the same list of base dimensions used in the SI unit system. SI defines "current" (charge per time) as a base unit, while simtk.unit uses "charge". And simtk.unit treats angle as a dimension, even though angle quantities are often considered dimensionless. In this case, we choose to err on the side of explicitness, particularly because interconversion of degrees and radians is a frequent source of unit headaches.

#### *7.3.4.3 Units examples*

Many common units are defined in the simtk.unit module.

```
from simtk.unit import nanometer, angstrom, dalton
```

Sometimes you don't want to type the full unit name every time, so you can assign it a shorter name using the `as` functionality:

```
from simtk.unit import nanometer as nm
```

New quantities can be created from a value and a unit. You can use either the multiply operator ('*') or the explicit Quantity constructor:

```
from simk.unit import nanometer, Quantity
# construct a Quantity using the multiply operator
bond_length = 1.53 * nanometer
```

```
# equivalently using the explicit Quantity constructor
bond_length = Quantity(1.53, nanometer)
# or more verbosely
bond_length = Quantity(value=1.53, unit=nanometer)
```

#### *7.3.4.4 Arithmetic with units*

Addition and subtraction of quantities is only permitted between quantities that share the same dimension. It makes no sense to add a mass to a distance. If you attempt to add or subtract two quantities with different dimensions, an exception will be raised. This is a good thing; it helps you avoid errors.

```
x = 5.0*dalton + 4.3*nanometer; # error
```

Addition or subtraction of quantities with the same dimension, but different units, is fine, and results in a new quantity created using the correct conversion factor between the units used.

```
x = 1.3*nanometer + 5.6*angstrom; # OK, result in nanometers
```

Quantities can be added and subtracted. Naked Units cannot.

Multiplying or dividing two quantities creates a new quantity with a composite dimension. For example, dividing a distance by a time results in a velocity.

```
from simtk.unit import kilogram, meter, second
a = 9.8 * meter / second**2; # acceleration
m = 0.36 * kilogram; # mass
F = m * a; # force in kg*m/s**2
```

Multiplication or division of two Units results in a composite Unit.

```
mps = meter / second
```

Unlike amount (moles), angle (radians) is arguably dimensionless. But simtk.unit treats angle as another dimension. Use the trigonometric functions from the simtk.unit module (not those from the Python math module!) when dealing with Units and Quantities.

```
from simtk.unit import sin, cos, acos
x = sin(90.0*degrees)
angle = acos(0.68); # returns an angle quantity (in radians)
```

The method `pow()` is a built-in Python method that works with Quantities and Units.

```
area = pow(3.0*meter, 2)
# or, equivalently
area = (3.0*meter)**2
# or
area = 9.0*(meter**2)
```

The method `sqrt()` is not as built-in as `pow()`. Do not use the Python `math.sqrt()` method with Units and Quantities. Use the `simtk.unit.sqrt()` method instead:

```
from simtk.unit import sqrt
side_length = sqrt(4.0*meter**2)
```

#### *7.3.4.5 Atomic scale mass and energy units are "per amount"*

Mass and energy units at the atomic scale are specified "per amount" in the simtk.unit module. Amount (mole) is one of the seven fundamental dimensions in the SI unit system. The atomic scale mass unit, dalton, is defined as grams per mole. The dimension of dalton is therefore mass/amount, instead of simply mass. Similarly, the atomic scale energy unit, kilojoule_per_mole (and kilocalorie_per_mole) has "per amount" in its dimension. Be careful to always use "per amount" mass and energy types at the atomic scale, and your dimensional analysis should work out properly.

The energy unit kilocalories_per_mole does not have the same Dimension as the macroscopic energy unit kilocalories. Molecular scientists sometimes use the word "kilocalories" when they mean "kilocalories per mole". Use "kilocalories per mole" or "kilojoules per mole" for molecular energies. Use "kilocalories" for the metabolic energy content of your lunch. The energy unit kilojoule_per_mole happens to go naturally with the units nanometer, picoseconds, and dalton. This is because 1 kilojoule/mole happens to be equal to 1 gram-nanometer$^2$/mole-picosecond$^2$, and is therefore consistent with the molecular dynamics unit system used in the C++ OpenMM API.

These "per mole" units are what you should be using for molecular calculations, as long as you are using SI / cgs / calorie sorts of units.

#### *7.3.4.6 SI prefixes*

Many units with SI prefixes such as “milligram” (milli) and “kilometer” (kilo) are provided in the simtk.unit module. Others can be created by multiplying a prefix symbol by a non-prefixed unit:

```
from simtk.unit import mega, kelvin
megakelvin = mega * kelvin
t = 8.3 * megakelvin
```

Only grams and meters get all of the SI prefixes (from yotto-($10^{-24}$) to yotta-($10^{24}$)) automatically.

#### *7.3.4.7 Converting to different units*

Use the `Quantity.in_units_of()` method to create a new Quantity with different units.

```
from simtk.unit import nanosecond, fortnight
x = (175000*nanosecond).in_units_of(fortnight)
```

When you want a plain number out of a Quantity, use the `value_in_unit()` method:

```
from simtk.unit import femtosecond, picosecond
t = 5.0*femtosecond
t_just_a_number = t.value_in_unit(picoseconds)
```

Using `value_in_unit()` puts the responsibility for unit analysis back into your hands, and it should be avoided. It is sometimes necessary, however, when you are called upon to use a non-units-aware Python API.

#### *7.3.4.8 Lists, tuples, numpy arrays, and Units*

Units can be attached to containers of numbers to create a vector quantity. The simtk.unit module overloads the `__setitem__` and `__getitem__` methods for these containers to ensure that Quantities go in and out.

```
>>> s2 = [[1,2,3],[4,5,6]] * centimeter
>>> print s2
[[1, 2, 3], [4, 5, 6]] cm
>>> print s2 / millimeter
[[10.0, 20.0, 30.0], [40.0, 50.0, 60.0]]

>>> import numpy
>>> a = Quantity(numpy.array([1,2,3]), centimeter)
>>> print a
[1 2 3] cm
>>> print a / millimeter
[ 10.  20.  30.]
```

Converting a whole list to different units at once is much faster than converting each element individually. For example, consider the following code that prints out the position of every particle in a State, as measured in Angstroms:

```
for v in state.getPositions():
    print v.value_in_unit(angstrom)
```

This can be rewritten as follows:

```
for v in state.getPositions().value_in_unit(angstrom):
    print v
```

The two versions produce identical results, but the second one will run faster, and therefore is preferred.

# 8 Examples of OpenMM Integration

## 8.1 GROMACS

GROMACS is a large, complex application written primarily in C. The considerations involved in adapting it to use OpenMM are likely to be similar to those faced by developers of other existing applications. The GROMACS version with OpenMM integrated can be downloaded from http://simtk.org/home/openmm (click on the "Downloads" link).

The first principle we followed in adapting GROMACS was to keep all OpenMM-related code isolated to just a few files, while modifying as little of the existing GROMACS code as possible. This minimized the risk of breaking existing parts of the code, while making the OpenMM-related parts as easy to work with as possible. It also minimized the need for C code to invoke the C++ API. (This would not be an issue if we used the OpenMM C API wrapper, but that is less convenient than the C++ API, and placing all of the OpenMM calls into separate C++ files solves the problem equally well.)

In fact, only a single existing source file (md.c) was modified, while two new files (md_openmm.h and md_openmm.cpp) were added. md_openmm.h defines just four functions which encapsulate all of the interaction between OpenMM and the rest of GROMACS:

`openmm_init()`: As arguments, this function takes pointers to lots of internal GROMACS data structures that describe the simulation to be run. It creates a System, Integrator, and Context based on them, then returns an opaque reference to an object containing them. That reference is an input argument to all of the other functions defined in md_openmm.h. This allows information to be passed between those functions without exposing it to the rest of GROMACS.

`openmm_take_one_step()`: This calls `step(1)` on the Integrator that was created by `openmm_init()`.

`openmm_copy_state()`: This calls `getState()` on the Context that was created by `openmm_init()`, and then copies information from the resulting State into various GROMACS data structures. This function is how state data generated by OpenMM is passed back to GROMACS for output, analysis, etc.

`openmm_cleanup()`: This is called at the end of the simulation. It deletes all the objects that were created by `openmm_init()`.

This set of functions defines the interactions between GROMACS and OpenMM: copying information from the application to OpenMM, performing integration, copying information from OpenMM back to the application, and freeing resources at the end of the simulation. While the details of their implementations are specific to GROMACS, this overall pattern is fairly generic. A similar set of functions can be used for many other applications as well.

## 8.2 PyMD

PyMD is a lightweight Python library for molecular dynamics (MD) simulation and analysis. It was created with several goals in mind.

First and foremost, doing a molecular dynamics simulation should be straightforward—simple tasks should have simple solutions. The simplicity of PyMD arises by pairing the OpenMM library with Python, Numpy, and PyTables. Python-Numpy-Scipy provides easy and efficient array data types and numerical algorithms. PyTables allows high performance file input-output using an underlying HDF5 format.

Second, the infrastructure for doing and analyzing molecular dynamics should be written as a high-quality, well-documented, and user-extensible library. In particular, the barrier to adding new features must be small. PyMD is a companion library for OpenMM, providing many features necessary for running molecular dynamics but which are beyond the scope of

OpenMM, for example, ForceField, Conformation, Trajectory, and Topology classes for facilitating the MD pipeline.

Third, MD must be integrated with software for interactive data analysis. The Python-Numpy tool chain provides an obvious answer to this requirement. PyMD stores numerical data as Numpy arrays. Thus, no special data containers are necessary, and you can directly interact with both the input and output of your molecular dynamics simulations. PyMD is particularly well-suited for ipython-matploblib, an interactive environment for numerical computation and plotting, but you can easily use the analysis environment of your choosing.

Finally, MD must be fast. OpenMM provides excellent performance for all MD calculations. Similarly, the high-performance Numpy library ensures that analysis code runs efficiently; often, Numpy functions are comparable in speed to functions written in C or Fortran.

### 8.2.1 OpenMM integration

The majority of the PyMD code has no awareness of OpenMM. For instance, the Conformation, Trajectory, ForceField, and Topology objects are all OpenMM-agnostic classes used to store the information necessary for biomolecular modeling. All the interactions that PyMD has with OpenMM occur via the PyMD Simulation class.

By pushing all the OpenMM calls to one class, two design goals were attained. First, changes to OpenMM calls can be easily made as new OpenMM features become available. Second, the Conformation, ForceField, and Topology classes are independent of OpenMM and can be useful for tasks that do not involve OpenMM—for instance, PDB renaming, RMSD calculation, and sequence mutation.

Below are the Python calls in a typical PyMD example, where a user wants to perform molecular dynamics of a protein:

```
import FF
import Simulation
Amber03=FF.ForceField.LoadFromHDF("Amber03.h5")
C1=FF.Conformation.LoadFromPDB("Protein.pdb")
T1=FF.Topology.CreateTopologyFromConformation(Amber03,C1)
P1=Simulation.SimulationParameters.Langevin()
S1=Simulation.Simulation.CreateSimulation(T1,C1,P1)
S1.Step(1000)
```

As mentioned above, OpenMM calls only occur within member functions of the Simulation class (in this example, `S1`). In this case, there are two functions that interact with OpenMM: `CreateSimulation()` and `Step()`.

The call `Step(1000)` does 1000 steps of integration. The `CreateSimulation` function, as the name implies, sets up the simulation. It requires three inputs: a topology, conformation, and simulation parameters. The topology (`T1`) contains all the information about the bonds and forces in a system. The conformation (`C1`) contains much the same information as a PDB file, providing OpenMM with the three-dimensional coordinates of the atoms in the system. Finally, the simulation parameters (`P1`), obtained with a call to `SimulationParameters.Langevin()`, provide values needed by OpenMM to simulate the system of interest (e.g., temperature, friction, timestep).

To call the OpenMM functions, PyMD uses the PyOpenMM wrappers. We can examine the implementation of the `CreateSimulation` function using these wrappers in more detail below.

When `CreateSimulation` is called to create a Simulation (S1), PyMD uses the PyOpenMM wrappers to find the appropriate platform, create the desired forces, and initialize an OpenMM System, similar to the steps in the HelloArgon tutorial example (see Section 5.3). The majority of the code in this function involves iterating over the entries in the topology and creating the appropriate OpenMM Forces. For example, the section of `CreateSimulation` that adds the Periodic Torsion forces is included below. The effect of the code is to add the appropriate force for each 4-tuple of atoms (a0, a1, a2, a3) involved in a Periodic Torsion Force.

```
        if Parameters["AddPeriodic"]==True:
            Sim.system.addForce(Sim.PeriodicTorsionForce)
            for i in range(len(Topology["Impropers"])):
                a0=int(Topology["Impropers"][i][0])
                a1=int(Topology["Impropers"][i][1])
                a2=int(Topology["Impropers"][i][2])
                a3=int(Topology["Impropers"][i][3])
                period=int(Topology["ImproperPn"][i])
                phase=float(Topology["ImproperPhase"][i])
                kd=float(Topology["ImproperKd"][i])

Sim.PeriodicTorsionForce.addTorsion(a0,a1,a2,a3,period,phase*pi/180.,kd)
```

The member variables `Sim.system` and `Sim.PeriodicTorsionForce` are both OpenMM objects. As seen in the code, the actual interactions with OpenMM are simple and few.

# 9 Testing and Validation of OpenMM

Three types of validation of OpenMM have been performed:

- **Unit tests:** Unit tests are provided with each major force and integrator class and other auxiliary functions (e.g., the random number generator). The unit tests exercise the basic functionality of each class to probe for problems; a separate unit test is available for each of the different platforms. Typically, but not exclusively, these tests use simple model systems comprised of a small number of particles.

- **System tests:** In contrast to the unit tests, the system tests are performed on a collection of biomolecules. The types of tests included for these systems are checks for consistency between the forces for the different platforms (CPU vs. GPU), energy-force consistency (outlined below), and tests for energy conservation for Verlet integrators, thermostability for stochastic integrators, and checks that constraints are satisfied within the prescribed tolerance.

- **Direct comparison between GROMACS and OpenMM forces:** The third type of validation performed was a direct comparison, when possible, of the individual forces computed in GROMACS with those in OpenMM for a collection of biomolecules.

Each type of test is outlined in greater detail below; a discussion of the current status of the tests is then given.

## 9.1 Description of Tests

### 9.1.1 Unit tests

The unit tests are available in the source code and can be run by the user. See Section 4 (Compiling OpenMM from Source Code) for details on compiling the tests.

If a test is run and no problems are detected, the program will return 'Done'. If an error is detected, an exception is thrown, and an appropriate message is printed. The error message should be examined carefully since the discrepancy is often close to the allowed tolerance, and hence may be acceptable.

### 9.1.2 System tests

Systems tests were performed to validate: 1) the consistency of the calculated forces across platforms, including non-Fermi and Fermi boards for CUDA and OpenCL, 2) the consistency of energy and force for each force class on each platform and board type, and 3) energy conservation and thermostability on the CUDA and OpenCL platforms. The tests were run on 20 different systems employing an implicit solvent model, including a DNA and RNA system; the system sizes ranged from 75-6450 particles; for explicit solvent tests, 5 systems were tested with sizes ranging from 910-173181 particles. Tests involving the OpenCL platform were only done for NVIDIA boards; tests using AMD boards will be performed once the required software is available.

*Force consistency between platforms:* The first set of system tests was a comparison of the forces between the Reference (CPU) platform and the CUDA (GPU) platform and the Reference platform and the OpenCL (GPU) platform. These tests consist of building an OpenMM System with a single force class or multiple force classes and then checking that the calculated force components agree to a specified tolerance. Multiple force classes are used for testing the GBSA force since for the GPU platforms the nonbonded and GBSA terms are folded into a single kernel and as a result the individual forces cannot be separated.

*Energy-force consistency on a platform:* The second set of tests was a check that the energy and force are consistent for each force class on each platform. The test protocol is as follows:

- Compute the force ($F_o = -\nabla V|_{r = r_o}$) and potential energy ($V_o$) for a given configuration
- Perturb the coordinates in the direction of the force $F_o$ by an amount $\varepsilon$:

  $\Delta r = -F_o * \varepsilon / |F_o|$, where $\varepsilon \sim 10^{-2} - 10^{-6}$ nm
- Calculate the potential energy V at the perturbed configuration

  $V = V_o + \nabla V \cdot \Delta r + \ldots$

  $V - V_o \cong -F_o \cdot \Delta r = -F_o \cdot (-\varepsilon * F_o / |F_o|)$

  $[V - V_o] / \varepsilon \cong |F_o|$

Here $\Delta r$ is the perturbation in the coordinates of the system. The relative difference between $[V - V_o] / \varepsilon$ and $|F_o|$ should be within a specified tolerance.

*Energy conservation and thermostability:* The focus of the third set of tests is on the integrators. The systems are first equilibrated for 30 ps. A simulation is then run for 1 ns, accumulating the total energy for Verlet integrators and the kinetic energy for the Langevin (LangevinIntegrator and VariableLangevinIntegrator) every ps. Each time the energies are calculated a check is made that any constraints are satisfied to within the desired tolerance. When the runs have completed, the energy drift is computed in units of $k_B T$/degrees-of-freedom/ns for the Verlet integrators. For the stochastic integrators, the deviation of the average temperature from the user-specified temperature is monitored. Note that this third set of tests was only carried out on the CUDA and OpenCL platforms, since running the tests on the Reference platform would require substantial computational effort.

### 9.1.3 Direct comparisons between GROMACS and OpenMM forces

A direct comparison between the forces computed in GROMACS 4.5 and OpenMM 4.0 for a variety of biomolecules was made. The comparisons include the following forces and conditions:

- HarmonicBond
- HarmonicAngle
- PeriodicTorsion
- GBSA OBC implicit solvent

Because of the usage of charge groups in GROMACS and their absence in OpenMM, the neighbor lists are in general different for the nonbonded interactions (including Ewald and PME) for the two applications. Hence detailed comparisons of the calculated forces are not possible. For reference, the nonbonded forces for one system were compared using a cutoff much greater than the system size and in a second comparison with a cutoff of 1nm.

## 9.2 Test Results

### 9.2.1 Unit tests

The unit tests should pass, although some of the tests applied to the LangevinIntegrator, VariableLangevinIntegrator, and BrownianIntegrator test are stochastic and may occasionally fail. In some cases, a test may fail, but only marginally since the calculated value is just outside the specified range of acceptable values. For these cases, you must decide if the difference is significant.

### 9.2.2 System tests

A concise summary of the results is presented below to provide estimates of the order of magnitude of differences observed. The full results for the system tests are too numerous to delineate.

*Force consistency across Reference, CUDA and OpenCL platforms:* The summary of these system tests are given in Table 9.1 (*Reference, CUDA)* and Table 9.2 (*Reference and OpenCL)* below. The Average Relative Difference column is the weighted average over all systems with the weight set to the number of particles in the system. The maximum relative difference reported is $2.0*|F_{Reference} - F_{Cuda}|/(|F_{Reference}| + |F_{Cuda}|)$, where $F_i$ is the force on platform i, and the difference is the maximum observed over all systems. The norm of the force entry with the maximum observed relative difference is also provided.

| Force | Average Relative Difference | Max Relative Difference | Force Norm w/ Max Relative Difference |
|---|---|---|---|
| HarmonicBond[1] | 1.982e-05 | 1.414e+00 | 1.361e-02 |
| HarmonicAngle[1] | 1.153e-05 | 4.366e-02 | 1.662e-01 |
| PeriodicTorsion[1] | 1.506e-05 | 2.548e-01 | 2.089e-03 |
| RB Torsion[1] | 3.878e-06 | 5.811e-02 | 6.048e-03 |
| CustomBond[1] | 1.982e-05 | 1.414e+00 | 1.361e-02 |
| CustomAngle[1] | 1.154e-05 | 4.364e-02 | 1.662e-01 |
| CustomTorsion[1] | 1.506e-05 | 2.548e-01 | 2.089e-03 |
| Nonbonded no cutoff[1] | 7.375e-07 | 1.291e-04 | 1.110e+01 |
| Nonbonded/cutoff/non periodic[2] | 4.753e-07 | 4.163e-05 | 1.440e+02 |
| Nonbonded/cutoff/periodic[2] | 4.801e-07 | 4.289e-05 | 1.383e+02 |
| Ewald[2] | 2.174e-06 | 2.633e-04 | 2.774e+01 |
| PME[2] | 3.681e-06 | 7.006e-04 | 2.249e+01 |
| CustomNonbonded no cutoff[2] | 9.593e-07 | 4.823e-05 | 7.821e+01 |
| CustomNonbonded/cutoff/non periodic[2] | 9.945e-07 | 1.393e-03 | 5.171e+02 |
| CustomNonbonded/cutoff/periodic[2] | 1.168e-06 | 2.010e-03 | 2.968e+02 |
| OBC/no cutoff[1] | 3.041e-06 | 2.468e-04 | 4.810e+01 |
| OBC/cutoff/non periodic[1] | 1.995e-06 | 9.733e-03 | 7.641e+01 |
| OBC/cutoff/periodic[1] | 1.447e-06 | 4.038e-03 | 2.265e+02 |

**Table 9.1: Summary of force consistency across CUDA and Reference platforms. The Max Relative Difference reported is $2.0*|F_{Reference} - F_{Cuda}|/(|F_{Reference}| + |F_{Cuda}|)$, where $F_i$ is the force on platform i, and is the maximum observed over all systems. The average relative difference is that over all systems weighted by the number of particles in the system. CustomGbsaForce is not available for the CUDA platform.**

**[1]Tested on 1PLX(75),* ala10(112), bench10(247), villin(582), bench80(598), bench2(616), bench0(627), bench3(629), bench4(629), bench1(634), bench81(634), dna(758), bpti(854), lambda(1254), lyso(1961), bench102(2150), tna(2444), proteinAmber(3120), spectrin(5078), bench101(6450)**

**[2]Tested on ala10(910),* bpti(8018), lambda(8133), villin(10461), dna(19046), tna(90968). The 173,181 particle system (a box of water) did not run on CUDA due to memory constraints.**

***The number in parentheses following the system names is the number of particles in the system.**

| Force | Average Relative Difference | Max Relative Difference | Force Norm w/ Max Relative Difference |
|---|---|---|---|
| HarmonicBond[1] | 1.982e-05 | 1.414e+00 | 1.361e-02 |
| HarmonicAngle[1] | 1.154e-05 | 4.364e-02 | 1.662e-01 |
| PeriodicTorsion[1] | 1.504e-05 | 2.548e-01 | 2.089e-03 |
| RB Torsion[1] | 3.868e-06 | 5.588e-02 | 6.055e-03 |
| CustomBond[1] | 1.982e-05 | 1.414e+00 | 1.361e-02 |
| CustomAngle[1] | 1.154e-05 | 4.364e-02 | 1.662e-01 |
| CustomTorsion[1] | 1.504e-05 | 2.548e-01 | 2.089e-03 |
| Nonbonded no cutoff[1] | 7.273e-07 | 1.284e-04 | 1.110e+01 |
| Nonbonded/cutoff/non periodic[2] | 4.758e-07 | 4.312e-05 | 1.440e+02 |
| Nonbonded/cutoff/periodic[2] | 4.801e-07 | 4.312e-05 | 1.383e+02 |
| Ewald[2] | 1.925e-06 | 1.673e-04 | 2.253e+01 |
| PME[2] | 1.341e-05 | 1.724e-03 | 5.311e+01 |
| CustomNonbonded no cutoff[2] | 8.968e-07 | 9.853e-05 | 3.747e+02 |
| CustomNonbonded/cutoff/non periodic[2] | 7.662e-07 | 5.196e-04 | 7.784e+02 |
| CustomNonbonded/cutoff/periodic[2] | 8.279e-07 | 2.010e-03 | 2.968e+02 |
| OBC/no cutoff[1] | 3.085e-06 | 2.467e-04 | 4.885e+01 |
| OBC/cutoff/non periodic[1] | 2.034e-06 | 9.924e-03 | 7.642e+01 |
| OBC/cutoff/periodic[1] | 1.479e-06 | 4.038e-03 | 2.265e+02 |
| CustomGbsa/no cutoff[1] | 7.763e-06 | 9.392e-04 | 5.660e-01 |
| CustomGbsa/nonperiodic/cutoff[1] | 2.077e-06 | 1.148e-04 | 1.577e+00 |
| CustomGbsa/periodic/cutoff[1] | 2.157e-06 | 1.226e-04 | 3.448e-01 |

**Table 9.2: Summary of force consistency across OpenCL and Reference platforms. The Max Relative Difference reported is $2.0*|F_{Reference} - F_{Cuda}|/(|F_{Reference}| + |F_{Cuda}|)$, where $F_i$ is the force on platform i, and is the maximum observed over all systems. The average relative difference is that over all systems weighted by the number of particles in the system.**

**[1]Tested on 1PLX(75),* ala10(112), bench10(247), villin(582), bench80(598), bench2(616), bench0(627), bench3(629), bench4(629), bench1(634), bench81(634), dna(758), bpti(854), lambda(1254), lyso(1961), bench102(2150), tna(2444), proteinAmber(3120), spectrin(5078), bench101(6450)**

**[2]Tested on ala10(910),* bpti(8018), lambda(8133), villin(10461), dna(19046), tna(90968), water(173181)**

***The number in parentheses following the system names is the number of particles in the system.**

*Energy-force consistency*: The summary of these system tests are given in Table 9.3 (*CUDA)* and Table 9.4 (*OpenCL)* below. The reported maximum relative difference was calculated as $\left| [V-V_0]/\varepsilon - |F_0| \right| / |F_0|$, where V is the potential energy of the system, F is the force, and $\varepsilon$ is the perturbation in the coordinates of the system (see test description in Section 9.1.2 above). The $\varepsilon$ used to perturb the coordinates in the calculations was chosen to minimize the relative difference; the magnitude of the optimal $\varepsilon$ ranged from 1.0e-05 to 1.0e-03. The values shown in Table 9.3 and Table 9.4 are the maximum observed over all systems running on the CUDA and OpenCL platforms, respectively. The results shown here were obtained on an NVIDIA GTX 480; similar results were observed on a Tesla C1060. The results for OBC/cutoffs/non periodic and OBC/cutoffs/periodic are high since no tapering of the OBC contribution to the energy is applied.

| Force | Max Relative Difference | Log Average Relative Difference |
|---|---|---|
| HarmonicBond[1] | 7.512e-03 | 6.875e-04 |
| HarmonicAngle[1] | 4.170e-03 | 3.842e-04 |
| PeriodicTorsion[1] | 1.434e-02 | 1.261e-03 |
| RB Torsion[1] | 3.540e-03 | 6.062e-04 |
| Nonbonded no cutoff[1] | 6.831e-04 | 3.299e-04 |
| Nonbonded/cutoffs/non periodic[2] | 3.916e-03 | 3.688e-04 |
| Nonbonded/cutoff/periodic[2] | 3.913e-03 | 6.487e-04 |
| Ewald[2] | 3.918e-03 | 4.441e-04 |
| PME[2] | 3.918e-03 | 1.439e-04 |
| OBC/no cutoffs[1] | 1.263e-03 | 2.110e-04 |
| OBC/cutoffs/non periodic[1] | 1.630e-01 | 8.766e-04 |
| OB/cutoffs/periodic[1] | 8.054e-02 | 8.423e-04 |

**Table 9.3: Summary of energy-force consistency for the CUDA platform on an NVIDIA GeForce GTX 480 using CUDA 3.2. The reported maximum relative difference was calculated as $\left| [V-V_0]/\varepsilon - |F_0| \right| / |F_0|$, where V is the potential energy of the system, F is the force, and $\varepsilon$ is the perturbation in the coordinates of the system.**

**[1]Tested on 1PLX(75),* ala10(112), bench10(247), villin(582), bench80(598), bench2(616), bench0(627), bench3(629), bench4(629), bench1(634), bench81(634), dna(758), bpti(854), lambda(1254), lyso(1961), bench102(2150), tna(2444), proteinAmber(3120), spectrin(5078), bench101(6450)**

**[2]Tested on ala10(910),* bpti(8018), lambda(8133), villin(10461), dna(19046), tna(90968), water(173181)**

***The number in parentheses following the system names is the number of particles in the system.**

| Force | Max Relative Difference | Log Average Relative Difference |
|---|---|---|
| HarmonicBond[1] | 7.512e-03 | 6.875e-04 |
| HarmonicAngle[1] | 4.191e-03 | 3.933e-04 |
| PeriodicTorsion[1] | 1.435e-02 | 1.102e-03 |
| RB Torsion[1] | 2.890e-03 | 7.914e-04 |
| Nonbonded no cutoff[1] | 1.101e-03 | 3.778e-04 |
| Nonbonded/cutoffs/non periodic[2] | 3.834e-03 | 3.461e-04 |
| Nonbonded/cutoff/periodic[2] | 3.822e-03 | 4.037e-04 |
| Ewald[2] | 3.867e-03 | 7.929e-04 |
| PME[2] | 3.886e-03 | 5.984e-04 |
| OBC/no cutoffs[1] | 6.176e-04 | 2.031e-04 |
| OBC/cutoffs/non periodic[1] | 1.515e-01 | 8.700e-04 |
| OB/cutoffs/periodic[1] | NA | 9.354e-04 |

**Table 9.4: Summary of energy-force consistency for the OpenCL platform on an NVIDIA GeForce GTX 480 using CUDA 3.2. The reported maximum relative difference was calculated as $\left|[V-V_0]/\varepsilon - |F_0|\right|/|F_0|$, where V is the potential energy of the system, F is the force, and ε is the perturbation in the coordinates of the system.**

**[1]Tested on 1PLX(75),* ala10(112), bench10(247), villin(582), bench80(598), bench2(616), bench0(627), bench3(629), bench4(629), bench1(634), bench81(634), dna(758), bpti(854), lambda(1254), lyso(1961), bench102(2150), tna(2444), proteinAmber(3120), spectrin(5078), bench101(6450)**

**[2]Tested on ala10(910), bpti(8018), lambda(8133), villin(10461), dna(19046), tna(90968), water(173181)**

***The number in parentheses following the system names is the number of particles in the system.**

*Energy conservation and thermostability:* For the VerletIntegrator(step size=0.001 ps) and VariableVerletIntegrator (error tolerance=2.0e-06), the energy drift ranged from 2.0e-03 to 2.0e-02 kT/degrees-of-freedom/ns for both the CUDA and OpenCL platforms for over 50 different runs; in general the smaller systems had a higher drift value. For a specified temperature of 300 K, the average temperature ranged from [299-312]K for the LangevinIntegrator (step size=0.001 ps) and VariableLangevinIntegrator (error tolerance=1.0e-05). Test simulations using the BrownianIntegrator were not performed. No systemic constraint violations were observed.

### 9.2.3 GROMACS-Reference platform differences

The summary of comparisons between GROMACS and the OpenMM Reference platform are given in Table 9.5 below. The value reported is $2.0*|F_{OpenMMReference} - F_{GROMACS}|/(|F_{OpenMMReference}| + |F_{GROMACS}|)$, where $F_i$ is the force computed with software i, and is the maximum observed over all systems.

| Force | Average Relative Difference | Max Relative Difference | Force Norm w/ Max Relative Difference |
|---|---|---|---|
| HarmonicBond | 1.658e-04 | 1.764e-01 | 1.166e-01 |
| HarmonicAngle | 6.347e-05 | 9.755e-05 | 1.801e+01 |
| PeriodicTorsion | 3.701e-05 | 1.590e-02 | 2.087e-01 |
| Nonbonded/cutoff=100 nm/non periodic[1] | 6.125e-07 | 6.953e-05 | 1.415e+02 |
| Nonbonded/cutoff=1 nm/non periodic[1] | 2.468e-02 | 3.677e-01 | 1.125e+02 |
| OBC/nonbonded/no cutoffs | 3.821e-06 | 9.755e-05 | 1.801e+01 |

**Table 9.5: Comparison of forces computed by GROMACS versus the OpenMM Reference platform. The value reported is $2*|F_{OpenMMReference} - F_{GROMACS}|/(|F_{OpenMMReference}| + |F_{GROMACS}|)$, where $F_i$ is the force computed with software i, and is the maximum observed over all systems. The test systems used were ala10(112), bpti(854), lyso(1961), bench101(6450); the number in parentheses is the number of particles in the system. [1]Tests were only performed for ala10(4000) to give order of magnitude estimates of the differences; as noted above, nonbonded forces calculated with cutoffs are not directly comparable between GROMACS and OpenMM due to the usage of charge groups in building neighbor lists in GROMACS.**

## 9.3 Validation Software

Users have reported instances where all the OpenMM unit tests pass for a given hardware, software, and operating system setup, but the OpenMM program was clearly giving incorrect results for simulations of their larger systems. The same molecular system was reported to run properly for a different hardware/software/operating system combination. As a first step to help identify these types of situations, we have added a unit test

(TestCudaUsingParameterFile) that reads an ASCII file containing parameters for a 1254-atom protein, compares the forces computed with the Reference platform with those using the CUDA platform, and reports any significant discrepancies. See Chapter 4 (Compiling OpenMM from Source Code) for details on compiling and running the tests.

A library of routines has also been added to allow users to more easily compare calculations of the forces on the Reference and CUDA platforms. An example snippet of code using the library is given below:

```
#include "libraries/validate/include/ValidateOpenMMForces.h
…
ValidateOpenMMForces validateForce;
validateForce.setLog( stderr );       // direct any logging info to stderr
std::string summary;                  // output string
int misses=validateForce.compareWithReferencePlatform(*context,&summary);
(void)fprintf(stderr,"Misses=%d Summary\n\n%s\n",misses,summary.c_str() );
```

The input to the method `validateForce.compareWithReferencePlatform()` is an instance of the OpenMM Context class that is to be tested, and the output is a std::string containing a summary of the comparisons. The method's return value is nonzero, if errors were detected, and otherwise is zero. The comparison method will calculate the forces that have been registered with the System object associated with the Context object (HarmonicBond , HarmonicAngle , …) individually and collectively on each platform and compare the results; the particle coordinates used in the calculations are those specified in the context via `context->setPositions()`. The individual forces are compared instead of just the sum of all forces since problems can sometimes be masked if the magnitude of one force is significantly larger than other forces. The exceptions to performing the calculations for individual forces are the implicit solvent forces (GBSAOBCForce and GBVIForce). The calculations of these forces on the CUDA platform are combined with the nonbonded forces to reduce the computational time (one less $O(N^2)$ loop). As a consequence, only the combination of the implicit solvent force and the nonbonded forces can be directly compared. If implicit solvent forces are present, the comparison method will make two comparisons: the nonbonded alone and nonbonded + implicit solvent forces. In addition to comparing the forces, the method also checks the energies.

An example of the output contained in the summary string is given below. The first block gives the result for the NonbondedForce (Nb), the second block for the combined NonbondedForce and GBSAOBCForce, …. The last block is a comparison for all the registered forces (HarmonicAngle, HarmonicBond, Nb, Obc, PeriodicTorsion). An error is reported for the HarmonicAngle force. Errors are registered if nans or infinities are detected or if the average of the norm of the two forces and the relative difference between the forces are greater than a specified tolerance. The default tolerance is 1.0e-02; the tolerance value may be set via the call `validateForce.setForceTolerance(userSpecifiedValue)`.

The logic used in reporting problems is that significant relative differences in the force values may be ignored, if the magnitude of the force is small. The primary goal of the library is to identify cases where the GPU board is giving incorrect values; in general, these will not be small discrepancies.

```
Misses=1 Summary

   Platforms                                    Reference        Cuda

   Force                                 Nb
   Tolerance                             1.000e-02
   Max Delta                             4.671e-03 at index    458
   Max Relative Delta                    4.443e-05 at index   1571
   Potential energies relative delta     6.3725e-06 PE[ -2.535492e+04  -2.535476e+04]


   Force                                 Nb::Obc
   Tolerance                             1.000e-02
   Max Delta                             2.798e-02 at index    218
   Max Relative Delta                    1.993e-04 at index   1340
   Potential energies relative delta     1.0321e-05 PE[ -3.367265e+04  -3.367230e+04]


   Force                                 HarmonicBond
   Tolerance                             1.000e-02
   Max Delta                             1.141e-02 at index   2104
   Max Relative Delta                    4.474e-03 at index    103
   Potential energies relative delta     4.3744e-07 PE[  2.094316e+03   2.094315e+03]
```

```
    Force                                   HarmonicAngle
    Tolerance                               1.000e-02
    Max Delta                               4.137e-03 at index   2230
    Max Relative Delta                      1.818e-01 at index    317
    Potential energies relative delta       3.6725e-06 PE[  3.239476e+03   3.239464e+03]
    Error                                   3.17460e-02 at index 408
                                      norms: [1.16380e-02 1.12743e-02]
                                     forces: [-8.98444e-03 -3.92734e-04 -7.38707e-03]
                                             [-8.70368e-03 -3.80461e-04 -7.15622e-03]


    Total errors                            1

    Force                                   PeriodicTorsion
    Tolerance                               3.000e-01
    Max Delta                               6.044e-03 at index   1250
    Max Relative Delta                      2.000e+00 at index   1958
    Potential energies relative delta       1.7841e-06 PE[  4.226045e+03   4.226052e+03]


    Force
HarmonicAngle::HarmonicBond::Nb::Obc::PeriodicTorsion
    Tolerance                               1.000e-02
    Max Delta                               2.841e-02 at index    218
    Max Relative Delta                      1.420e-04 at index    907
    Potential energies relative delta       1.4185e-05 PE[ -2.411281e+04  -2.411247e+04]
```

# 10 AMOEBA Plugin

OpenMM 4.0 provides a CUDA platform plugin that implements the AMOEBA polarizable atomic multipole force field[1-2] from Jay Ponder's lab. A Reference platform plugin is also provided, but only a subset of the forces has been currently implemented for this platform. The plugin may be accessed using a modified version of TINKER (referred to as TINKER-OpenMM here). TINKER-OpenMM can be created from a TINKER package using three files made available through the OpenMM home page. The plugin can also be accessed through pyMD. OpenMM AMOEBA force and System objects containing AMOEBA forces can be serialized. In the following sections, the individual forces and options available in the plugin are listed, and the steps required to build and use the plugin and Tinker-OpenMM are outlined. Validation and benchmark results are also reported

## 10.1 OpenMM AMOEBA Supported Forces and Options

The AMOEBA force terms implemented in the Cuda platform are listed in the table below along with the supported and unsupported options. Tinker options that are not supported for any OpenMM force include the grouping of atoms (e.g., protein chains), the infinite polymer check, and no exclusion of particles from energy/force calculations ('active'/'inactive' particles). Switching is not applied to any of the long-range interactions. The virial is not calculated for any force.

All rotation axis types are supported: 'Z-then-X', 'Bisector', 'Z-Bisect', '3-Fold', 'Z-Only'.

Table 9.1 **Mapping between Tinker and OpenMM AMOEBA Forces**

| Tinker force(key file parameter) | OpenMM force | Option/Note |
|---|---|---|
| ebond1(bondterm) | AmoebaHarmonicBondForce | bndtyp='HARMONIC' supported, 'MORSE' not implemented |
| Eangle71(angleterm) | AmoebaHarmonicAngleForce | angtyp='HARMONIC' and 'IN-PLANE' supported; 'LINEAR' and 'FOURIER' not implemented |
| etors1a(torsionterm) | AmoebaTorsionForce | All options implemented; smoothing version(etors1b) not supported |
| etortor1(tortorterm) | AmoebaTorsionTorsionForce | All options implemented |
| eopbend1(opbendterm) | AmoebaOutOfPlaneBendForce | opbtyp = 'ALLINGER' implemented; 'W-D-C' not implemented |
| epitors1(pitorsterm) | AmoebaPiTorsionForce | All options implemented |
| estrbnd1(strbndterm) | AmoebaStretchBendForce | All options implemented |
| ehal1a(vdwterm) | AmoebaVdwForce | ehal1b(LIGHTS) not supported; long-range van der Waals energy correction unsupported |
| empole1a(mpoleterm) | AmoebaMultipoleForce | poltyp = 'MUTUAL', 'DIRECT' supported |
| empole1c(mpoleterm) PME | AmoebaMultipoleForce | poltyp = 'MUTUAL', 'DIRECT' supported; boundary= 'VACUUM' unsupported |
| esolv1 (solvateterm) | AmoebaWcaDispersionForce, AmoebaGeneralizedKirkwoodForce | Only born-radius='grycuk' and solvate='GK' supported; unsupported solvate settings: 'ASP', 'SASA', 'ONION', 'pb', 'GB-HPMF', 'Gk-HPMF'; SASA computation is based on ACE approximation |
| eurey1(ureyterm) | AmoebaUreyBradleyForce | All options implemented |

Notes:

Forces available in Tinker but not implemented in the OpenMM plugin include the following: angle-angle, out-of-plane distance, improper dihedral, improper torsion, stretch-

torsion, charge-charge, atomwise charge-dipole, dipole-dipole, reaction field, ligand field, restraint, scf molecular orbital calculation; strictly speaking these are not part of the AMOEBA force field.

In Tinker the nonpolar contribution to the solvation term is calculated using an algorithm that does not map well to GPUs. Instead the OpenMM plugin uses the Tinker version of the ACE approximation to estimate the SASA.

Calculations using the Cuda platform are done in single precision; for the Reference platform double precision is used.

The Tinker parameter files for the AMOEBA force-field parameters are based on units of kilocalorie/Å, whereas OpenMM uses units of kilojoules/nanometer; both Tinker and OpenMM used picoseconds time units. Hence in mapping the force-field parameters from Tinker files to OpenMM many of the parameter values must be converted to the OpenMM units. The setup methods in the Tinker-OpenMM application perform the required conversions.

## 10.2 Tinker-OpenMM

### 10.2.1 Building Tinker-OpenMM (Linux)

(1) Build and install the OpenMM plugin libraries following the steps outlined in Chapter 4. When running CMake, set the following options to 'ON':

OPENMM_BUILD_AMOEBA_PLUGIN,
OPENMM_BUILD_AMOEBA_CUDA_LIB,
OPENMM_BUILD_CUDA_LIB ,
OPENMM_BUILD_C_AND_FORTRAN_WRAPPERS

(2) Obtain the modified Tinker file dynamic.f, the interface file dynamic_openmm.c and the Makefile from the 'Downloads & Source Code' section of OpenMM Home page (https://simtk.org/home/openmm) and place in the Tinker source directory.

(3) In the Makefile edit the fields 'TINKERDIR', ' LINKDIR', 'CC' and 'OpenMM_INSTALL_DIR' as needed. The added field 'CC' field should point to the C compiler and 'OpenMM_INSTALL_DIR' should be the directory where the OpenMM files were installed, i.e., the OPENMM_INSTALL_PREFIX setting when CMake was run in step (1).
(4) Type 'make dynamic_openmm.x' at the command line to create the executable
(5) Check that the environment variable 'OPENMM_PLUGIN_DIR' is set to the installed plugins directory and that the environment variable 'LD_LIBRARY_PATH' includes both the installed lib and plugins directory; for example:

OPENMM_PLUGIN_DIR=/home/usr/install/openmm/lib/plugins
LD_LIBRARY_PATH=/usr/local/cuda/lib64:/home/usr/install/openmm/lib:/home/usr/install/openmm/lib/plugins

(6) Run dynamic_openmm.x with the same command-line options as you would dynamic.x

Notes:
To use the Tinker routines as opposed to the OpenMM plugin to run the simulation, set 'useOpenMM' to .false. in dynamic.f (around line 300).

To specify a GPU board other than the default, set the environment variable 'CUDA_DEVICE' to the desired board id. A line like the following will be printed to stderr for the setting CUDA_DEVICE=2:

'Platform Cuda: setting device id to 2 based on env variable CUDA_DEVICE.'

To turn on testing (comparison of forces and potential energy for the initial conformation calculated using Tinker routines and OpenMM routines), set 'applyOpenMMTest' to a nonzero value in dynamic.f (also around line 300). Testing of individual forces can be done by adding lines like 'bondterm only' to the Tinker key file. For implicit solvent ('solvate GK' runs) the forces and energies will differ due to the different treatments of the SASA term (see Notes below Table 9.1 above). To test the implicit calculations use the following Tinker key file entries:

solvate GK
born-radius grycuk
solvateterm only
polarizeterm
mpoleterm

With these options, the test routine will remove the cavity contribution from the Tinker and OpenMM forces/energy when performing the comparison.

The program exits after the force/energy comparisons; it does not execute the main md loop.

### 10.2.2 Overview of Tinker-OpenMM

Tinker-OpenMM allows the OpenMM plugin to be used to calculate the forces and energies and perform the integration in the main molecular dynamics loop. The only significant changes to the Tinker source code are made in the file dynamic.f for the setup and running of a molecular dynamics simulation. An added file, dynamic_openmm.c, contains the interface C-code between Tinker and the OpenMM plugin.
In addition to the limitations to the forces outlined above, Tinker-OpenMM can only use either the 'Verlet' or 'Stochastic' integrators when the OpenMM plugin is used; a 'Beeman' integrator is unavailable in OpenMM.

The flow of the molecular dynamics simulation using the OpenMM plugin is as follows:

(1) the Tinker code is used to read the AMOEBA parameter file, the '*.xyz' and '*.key' files and parse the command-line options

(2) the routine openmm_validate() is called from dynamic.f before the main loop. This routine checks that all required options and settings obtained from the input in step (1) are available in the plugin. If an option or setting is unsupported, the program exits with an appropriate message. This routine also maps data (parameters, coordinates, …) from the Tinker Fortran common blocks to C data structures. The routine openmm_validate() and the other OpenMM interface methods are in the file dynamic_openmm.c

(3) openmm_init() is called to create the OpenMM System, Integrator and Context objects. The System object contains the particle masses, the constraints, and the AMOEBA

force objects, the Integrator is either the VerletIntegrator or the LangevinIntegrator and will contain the step size and for the Langevin the temperature and collision frequency. Note that pressure coupling is implemented via the OpenMM MonteCarloBarostat class. When initializing the AMOEBA force objects, the parameters obtained from the Tinker common blocks are converted to the OpenMM units (kJ/nm).

(4) the routine openmm_take_steps() is called to take a specified number of time steps.

(5) openmm_update() is then called to retrieve the state (energies/positions/velocities) and populate the appropriate Tinker data structures; these values are converted from the OpenMM units of kJ/nm to kcal/Å when populating the Tinker arrays. A breakdown of the energy and force into individual terms (bond, angle, ...) as is done in Tinker is unavailable through the plugin; only the total force and potential energy are returned. Also the pressure cannot be calculated since the virial is not calculated in the plugin. Frequent retrieval of the state information from the GPU board can use up a substantial portion of the total wall clock time. This is due to the fact that the forces and energies are recalculated for each retrieval. Hence if the state information is obtained after every timestep, the wall clock time will approximately double.

Two options are provided for implementing steps (4) & (5):

(i) If the logical value of 'oneTimeStepPerUpdate' in dynamic.f is true, then a single step is taken and the Tinker data structures are populated at each step. This option is conceptually simpler and is consistent with the Tinker md loops; for example, the output from the Tinker subroutine mdstat() will be accurate for this choice. However, the performance will be degraded since the forces and energy are recalculated with each call, doubling the required time. This is the default option.

(ii) If 'oneTimeStepPerUpdate' is false, then depending on the values of iprint (Tinker keyword 'PRINTOUT') and iwrite (=dump time/dt), multiple time steps are taken on the GPU before data is transferred from the GPU to the CPU. This will lead to better performance than option (i). One downside to this approach is that the fluctuation values printed by mdstat() will be incorrect.

(6) once the main loop has completed, the routine openmm_cleanup() is called to delete the OpenMM objects and release resources being used on the GPU board.

## 10.3 OpenMM AMOEBA Validation

OpenMM AMOEBA has been validated for 12 systems. The validation focused on comparisons between the individual forces and energies using Tinker's routines and the plugin. The results for calmodulin are shown in Table 9.2. Comparisons for villin (596 particles), ubiquitin (1228 particles), crambin(642 particles), lysozyme(1961 particles), dhfr (2489(implicit),23558 (explicit)), 1not (176) and boxes of water yielded similar results. Smaller molecule system were also tested (alanine tetrapeptide, dialaline-D, dialaline-L, ammonia). The relative difference for the potential energies was comparable or less than the average relative difference for the forces.

Table 9.2 **Force Comparisons for Calmodulin (2498 particles)**

| | Average Relative Difference | Maximum Relative Difference | Norm of Force w/ Maximum Relative Difference |
|---|---|---|---|
| AmoebaHarmonicBond | 9.532e-05 | 6.470e-02 | 1.792e-02 |
| AmoebaHarmonicAngle | 4.353e-05 | 4.887e-03 | 7.499e-02 |
| AmoebaStretchBend | 1.854e-05 | 3.294e-04 | 3.188e-02 |
| AmoebaOutOfPlaneBend | 5.536e-05 | 7.270e-03 | 2.846e-02 |
| AmoebaPiTorsion | 6.743e-05 | 5.362e-03 | 5.971e-03 |
| AmoebaTorsion | 4.441e-05 | 1.083e-02 | 1.252e-02 |
| AmoebaTorsionTorsion | 1.281e-06 | 3.048e-04 | 2.751e-01 |
| AmoebaUreyBradley | 2.919e-06 | 2.577e-04 | 1.090e+00 |
| AmoebaVdw (no cutoff) | 4.262e-06 | 1.801e-04 | 1.146e-02 |
| AmoebaVdw (9 Å cutoff) | 2.679e-03 | 1.325e-02 | 8.931e-02 |
| AmoebaWcaDispersion | 7.533e-07 | 1.064e-05 | 2.589e-02 |
| AmoebaMultipole | 1.541e-06 | 3.937e-05 | 1.105e+00 |
| AmoebaKirkwood | 5.086e-06 | 6.271e-05 | 3.308e-01 |

The drift for villin in implicit solvent was below 1.0e-02 kT/degrees-of-freedom/ns

# 11 Free Energy Plugin

The free energy plugin modifies the Lennard-Jones interaction and the implicit solvent models to allow free energy calculations based on thermodynamic integration. In particular, simulation-derived estimates of solvation free energies, ΔG, can be made by using thermodynamic integration:

$$\Delta G = \int_0^1 d\lambda \left[ dH(\lambda) / d\lambda \right]$$

Here H is a parameterized Hamiltonian of a system comprised of solute and solvent, and λ is a coupling parameter modulating the interactions between the solute and solvent: λ =1 corresponds to a fully coupled system, whereas λ=0 corresponds to fully decoupled interactions between the solute and solvent. As described in (Pande, 2005) for small distances between solute and solvent particles and λ ≈ 0, near singularities in dH/dλ lead to instabilities in the estimation of ΔG. The instabilities can be removed by first decoupling the solute-solvent Coulomb interactions and then decoupling the Lennard-Jones interaction using a 'softcore' parameterization to avoid the sampling issues and singularities for values of λ near zero and solute-solvent interactions with r ≈ 0. The Lennard-Jones softcore parameterization is given by equation 4 in (Pande, 2005):

$$E = 4\varepsilon\lambda \left( [1.0/[0.5*(1-\lambda)^2 + \left(\frac{r}{\sigma}\right)^6]^2 - 1.0/[0.5*(1-\lambda)^2 + \left(\frac{r}{\sigma}\right)^6] \right)$$

The free energy plugin also includes modifications to the OBC and GB/VI implicit solvent models to allow thermodynamic integration. For the OBC model, the single-atom contributions to the cavity energy term associated with solvating a neutral molecule are scaled by the atom's λ value. In addition for both implicit solvent models, the contribution of particle j to the calculation of the Born radius of particle i is scaled by the lambda value of particle j.

The nonbonded methods available for the plugin include no cutoff, cutoffs and cutoffs with periodic boundary conditions; the Ewald summation and particle-mesh Ewald methods are unavailable. The electrostatic interactions are modulated using the reaction-field method for the cutoff and cutoff with periodic boundary condition methods. The distance cutoff used for calculating the Born radii is a hard cutoff with no tapering of the contribution for separation distances near the cutoff. For the GB/VI model, a quintic spline scaling method is the default option for modulating the Born radii near a user-specified cutoff; a no-scaling option is also available. Without the scaling, cases where the pairwise overlap approximation used in calculating the Born volumetric integral breaks down and can lead to very large or infinite values of the Born radii; the scaling method sets an upper limit for the Born radii and the application of the quintic spline for Born radii near the cutoff insures the radii and first derivatives are continuous. The free energy plugin is available on the Reference and Cuda platforms; it has not been implemented on the OpenCL platform.

# 12 Ring Polymer Molecular Dynamics (RPMD) Plugin

Ring Polymer Molecular Dynamics (RPMD) provides an efficient approach to include nuclear quantum effects in molecular simulations.[3] When used to calculate static equilibrium properties, RPMD reduces to path integral molecular dynamics and gives an exact description of the effect of quantum fluctuations for a given potential energy model.[4] For dynamical properties RPMD is no longer exact but has shown to be a good approximation in many cases.

For a system with a classical potential energy $E(q)$, the RPMD Hamiltonian is given by

$$H = \sum_{k=1}^{n}\left(\frac{{p_k}^2}{2m} + E(q_k) + \frac{m(k_B T n)^2}{2\hbar^2}(q_k - q_{k-1})^2\right)$$

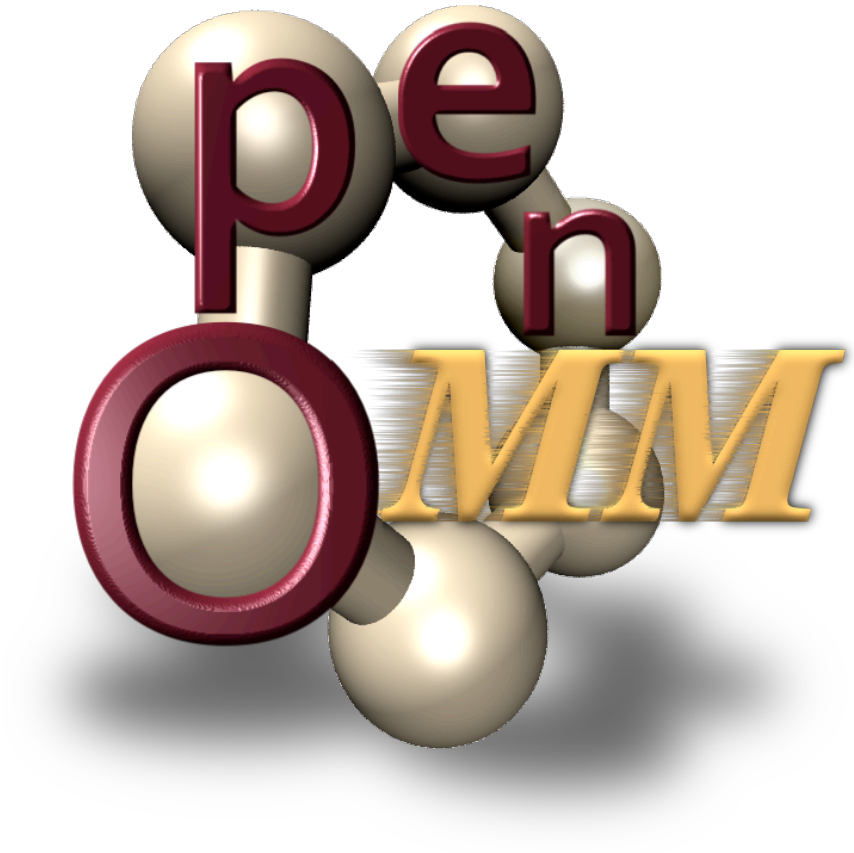
This Hamiltonian resembles that of a system of classical ring polymers where different copies of the system are connected by harmonic springs. Hence each copy of the classical system is commonly referred to as a "bead". The spread of the ring polymer representing each particle is directly related to its De Broglie thermal wavelength (uncertainty in its position).

RPMD calculations must be converged with respect to the number $n$ of beads used. Each bead is evolved at the effective temperature $nT$, where $T$ is the temperature for which properties are required. The number of beads needed to converge a calculation can be estimated using[5]

$$n > \frac{\hbar\omega_{max}}{k_B T}$$

where $\omega_{max}$ is the highest frequency in the problem. For example, for flexible liquid water the highest frequency is the OH stretch at around 3000 $cm^{-1}$, so around 24 to 32 beads are needed depending on the accuracy required. For rigid water where the highest frequency is only around 1000 $cm^{-1}$, only 6 beads are typically needed. Due to the replication needed of the classical system, the extra cost of the calculation compared to a classical simulation increases linearly with the number of beads used.

Due to the stiff spring terms between the beads, NVE RPMD trajectories can suffer from ergodicity problems and hence thermostatting is highly recommended, especially when dynamical properties are not required.[6] The thermostat implemented here is the path integral Langevin equation (PILE) approach.[7] This method couples an optimal white noise Langevin thermostat to the normal modes of each polymer, leaving only one parameter to be chosen by the user which controls the friction applied to the center of mass of each ring polymer. A good choice for this is to use a value similar to that used in a classical calculation of the same system.




# Part II

# Theory Guide

# 13 The Theory Behind OpenMM: an Introduction

## 13.1 Overview

This guide describes the mathematical theory behind OpenMM. For each computational class, it describes what computations the class performs and how it should be used. This serves two purposes. If you are using OpenMM within an application, this guide teaches you how to use it correctly. If you are implementing the OpenMM API for a new Platform, it teaches you how to correctly implement the required kernels.

On the other hand, many details are intentionally left unspecified. Any behavior that is not specified either in this guide or in the API documentation is left up to the Platform, and may be implemented in different ways by different Platforms. For example, an Integrator is required to produce a trajectory that satisfies constraints to within the user specified tolerance, but the algorithm used to enforce those constraints is left up to the Platform. Similarly, this guide provides the functional form of each Force, but does not specify what level of numerical precision it must be calculated to.

This is an essential feature of the design of OpenMM, because it allows the API to be implemented efficiently on a wide variety of hardware and software platforms, using whatever methods are most appropriate for each platform. On the other hand, it means that a single program may produce meaningfully different results depending on which Platform it uses. For example, different constraint algorithms may have different regions of convergence, and thus a time step that is stable on one platform may be unstable on a different one. It is essential that you validate your simulation methodology on each Platform you intend to use, and do not assume that good results on one Platform will guarantee good results on another Platform when using identical parameters.

## 13.2 Units

There are several different sets of units widely used in molecular simulations. For example, energies may be measured in kcal/mol or kJ/mol, distances may be in Angstroms or nm, and angles may be in degrees or radians. OpenMM uses the following units everywhere.

| Quantity | Units |
|---|---|
| distance | nm |
| time | ps |
| mass | atomic mass units |
| charge | proton charge |
| temperature | Kelvin |
| angle | radians |
| energy | kJ/mol |

**Table 13.1: Units used within OpenMM**

These units have the important feature that they form an internally consistent set. For example, a force always has the same units (kJ/mol/nm) whether it is calculated as the gradient of an energy or as the product of a mass and an acceleration. This is not true in some other widely used unit systems, such as those that express energy in kcal/mol.

The header file Units.h contains predefined constants for converting between the OpenMM units and some other common units. For example, if your application expresses distances in Angstroms, you should multiply them by OpenMM::NmPerAngstrom before passing them to OpenMM, and positions calculated by OpenMM should be multiplied by OpenMM::AngstromsPerNm before passing them back to your application.

# 14 Standard Forces

The following classes implement standard force field terms that are widely used in molecular simulations.

## 14.1 HarmonicBondForce

Each harmonic bond is represented by an energy term of the form

$$E = \frac{1}{2}k\left(x - x_0\right)^2$$

where $x$ is the distance between the two particles, $x_0$ is the equilibrium distance, and $k$ is the force constant. This produces a force of magnitude $k(x-x_0)$.

Be aware that some force fields define their harmonic bond parameters in a slightly different way: $E = k'(x-x_0)^2$, leading to a force of magnitude $2k'(x-x_0)$. Comparing these two forms, you can see that $k = 2k'$. Be sure to check which form a particular force field uses, and if necessary multiply the force constant by 2.

## 14.2 HarmonicAngleForce

Each harmonic angle is represented by an energy term of the form

$$E = \frac{1}{2}k\left(\theta - \theta_0\right)^2$$

where $\theta$ is the angle formed by the three particles, $\theta_0$ is the equilibrium angle, and $k$ is the force constant.

As with HarmonicBondForce, be aware that some force fields define their harmonic angle parameters as $E = k'(\theta-\theta_0)^2$. Be sure to check which form a particular force field uses, and if necessary multiply the force constant by 2.

## 14.3 PeriodicTorsionForce

Each torsion is represented by an energy term of the form

$$E = k\left(1 + \cos\left(n\theta - \theta_0\right)\right)$$

where $\theta$ is the dihedral angle formed by the four particles, $\theta_0$ is the equilibrium angle, $n$ is the periodicity, and $k$ is the force constant.

## 14.4 RBTorsionForce

Each torsion is represented by an energy term of the form

$$E = \sum_{i=0}^{5} C_i \left(\cos\phi\right)^i$$

where $\phi$ is the dihedral angle formed by the four particles and $C_0$ through $C_5$ are constant coefficients.

For reason of convention, PeriodicTorsionForce and RBTorsonForce define the torsion angle differently. $\theta$ is zero when the first and last particles are on the *same* side of the bond formed by the middle two particles (the *cis* configuration), whereas $\phi$ is zero when they are on *opposite* sides (the *trans* configuration). This means that $\theta = \phi - \pi$.

## 14.5 CMAPTorsionForce

Each torsion pair is represented by an energy term of the form

$$E = f(\theta_1,\theta_2)$$

where $\theta_1$ and $\theta_2$ are the two dihedral angles coupled by the term, and $f(x,y)$ is defined by a user supplied grid of tabulated values. A natural cubic spline surface is fit through the tabulated values, then evaluated to determine the energy for arbitrary ($\theta_1$, $\theta_2$) pairs.

## 14.6 NonbondedForce

### 14.6.1 Lennard-Jones Interaction

The Lennard-Jones interaction between each pair of particles is represented by an energy term of the form

$$E = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right)$$

where $r$ is the distance between the two particles, $\sigma$ is the distance at which the energy equals zero, and $\varepsilon$ sets the strength of the interaction. If the NonbondedMethod in use is anything other than NoCutoff and $r$ is greater than the cutoff distance, the energy and force are both set to zero. Because the interaction decreases very quickly with distance, the cutoff usually has little effect on the accuracy of simulations.

When an exception has been added for a pair of particles, $\sigma$ and $\varepsilon$ are the parameters specified by the exception. Otherwise they are determined from the parameters of the individual particles using the Lorentz-Bertelot combining rule:

$$\sigma = \frac{\sigma_1 + \sigma_2}{2}$$
$$\varepsilon = \sqrt{\varepsilon_1\varepsilon_2}$$

When using periodic boundary conditions, NonbondedForce can optionally add a term (known as a *long range dispersion correction*) to the energy that approximately represents the contribution from all interactions beyond the cutoff distance:[8]

$$E = 8\pi N^2 V \left( \frac{\left\langle \varepsilon_{ij} \sigma_{ij}^{12} \right\rangle}{9 r_c^9} - \frac{\left\langle \varepsilon_{ij} \sigma_{ij}^{6} \right\rangle}{6 r_c^3} \right)$$

where $N$ is the number of particles in the system, $V$ is the volume of the periodic box, $r_c$ is the cutoff distance, $\sigma_{ij}$ and $\varepsilon_{ij}$ are the interaction parameters between particle $i$ and particle $j$, and $\langle ... \rangle$ represents an average over all pairs of particles in the system. The long range dispersion correction is primarily useful when running simulations at constant pressure, since it produces a more accurate variation in system energy with respect to volume.

The Lennard-Jones interaction is often parameterized in two other equivalent ways. One is

$$E = \varepsilon \left( \left( \frac{r_{min}}{r} \right)^{12} - 2 \left( \frac{r_{min}}{r} \right)^{6} \right)$$

where $r_{min}$ (sometimes known as $d_{min}$; this is not a radius) is the center-to-center distance at which the energy is minimum. It is related to $\sigma$ by

$$\sigma = \frac{r_{min}}{2^{1/6}}$$

In turn, $r_{min}$ is related to the van der Waals radius by $r_{min} = 2\, r_{vdw}$.

Another common form is

$$E = \frac{A}{r^{12}} - \frac{B}{r^6}$$

The coefficients A and B are related to $\sigma$ and $\varepsilon$ by

$$\sigma = \left( \frac{A}{B} \right)^{1/6}$$

$$\varepsilon = \frac{B^2}{4A}$$

### 14.6.2 Coulomb Interaction Without Cutoff

The form of the Coulomb interaction between each pair of particles depends on the NonbondedMethod in use. For NoCutoff, it is given by

$$E = \frac{1}{4\pi\varepsilon_0}\frac{q_1 q_2}{r}$$

where $q_1$ and $q_2$ are the charges of the two particles, and $r$ is the distance between them.

### 14.6.3 Coulomb Interaction With Cutoff

For CutoffNonPeriodic or CutoffPeriodic, it is modified using the reaction field approximation. This is derived by assuming everything beyond the cutoff distance is a solvent with a uniform dielectric constant.[9]

$$E = \frac{q_1 q_2}{4\pi\varepsilon_0}\left(\frac{1}{r} + k_{rf} r^2 - c_{rf}\right)$$

$$k_{rf} = \left(\frac{1}{r_{cutoff}^3}\right)\left(\frac{\varepsilon_{solvent} - 1}{2\varepsilon_{solvent} + 1}\right)$$

$$c_{rf} = \left(\frac{1}{r_{cutoff}}\right)\left(\frac{3\varepsilon_{solvent}}{2\varepsilon_{solvent} + 1}\right)$$

where $r_{cutoff}$ is the cutoff distance and $\varepsilon_{solvent}$ is the dielectric constant of the solvent. In the limit $\varepsilon_{solvent} >> 1$, this causes the force to go to zero at the cutoff.

### 14.6.4 Coulomb Interaction With Ewald Summation

For Ewald, the total Coulomb energy is the sum of three terms: the *direct space sum*, the *reciprocal space sum*, and the *self-energy term*.[10]

$$E = E_{dir} + E_{rec} + E_{self}$$

$$E_{dir} = \frac{1}{2}\sum_{i,j}\sum_{\mathbf{n}} q_i q_j \frac{\mathrm{erfc}(\alpha r_{ij,\mathbf{n}})}{r_{ij,\mathbf{n}}}$$

$$E_{rec} = \frac{1}{2\pi V}\sum_{i,j} q_i q_j \sum_{\mathbf{k}\neq 0} \frac{\exp(-(\pi\mathbf{k}/\alpha)^2 + 2\pi i\mathbf{k}\cdot(\mathbf{r}_i - \mathbf{r}_j))}{\mathbf{m}^2}$$

$$E_{self} = -\frac{\alpha}{\sqrt{\pi}}\sum_i q_i^2$$

In the above expressions, the indices *i* and *j* run over all particles, **n** = ($n_1$, $n_2$, $n_3$) runs over all copies of the periodic cell, and **k** = ($k_1$, $k_2$, $k_3$) runs over all integer wave vectors from (-$k_{max}$, -$k_{max}$, -$k_{max}$) to ($k_{max}$, $k_{max}$, $k_{max}$) excluding (0, 0, 0). $\mathbf{r}_i$ is the position of particle *i*, while $r_{ij}$ is the distance between particles *i* and *j*. *V* is the volume of the periodic cell, and α is an internal parameter.

In the direct space sum, all pairs that are further apart than the cutoff distance are ignored. Because the cutoff is required to be less than half the width of the periodic cell, the number of terms in this sum is never greater than the square of the number of particles.

The error made by applying the direct space cutoff depends on the magnitude of erfc($\alpha r_{cutoff}$). Similarly, the error made in the reciprocal space sum by ignoring wave numbers beyond $k_{max}$ depends on the magnitude of $\exp(-(\pi k_{max}/\alpha)^2)$. By changing α, one can decrease the error in either term while increasing the error in the other one.

Instead of having the user specify α and $k_{max}$, NonbondedForce instead asks the user to choose an error tolerance δ. It then calculates α as

$$\alpha = \sqrt{-\log(2\delta)/r_{cutoff}}$$

Finally, it estimates the error in the reciprocal space sum as

$$error = \frac{k_{max}\sqrt{d\alpha}}{20}\exp(-(\pi k_{max}/d\alpha)^2)$$

where $d$ is the width of the periodic box, and selects the smallest value for $k_{max}$ which gives $error < \delta$. (If the box is not square, $k_{max}$ will have a different value along each axis.)

This means that the accuracy of the calculation is determined by $\delta$. $r_{cutoff}$ does not affect the accuracy of the result, but does affect the speed of the calculation by changing the relative costs of the direct space and reciprocal space sums. You therefore should test different cutoffs to find the value that gives best performance; this will in general vary both with the size of the system and with the Platform being used for the calculation. When the optimal cutoff is used for every simulation, the overall cost of evaluating the nonbonded forces scales as $O(N^{3/2})$ in the number of particles.

Be aware that the error tolerance $\delta$ is not a rigorous upper bound on the errors. The formulas given above are empirically found to produce average relative errors in the forces that are less than or similar to $\delta$ across a variety of systems and parameter values, but no guarantees are made. It is important to validate your own simulations, and identify parameter values that produce acceptable accuracy for each system.

### 14.6.5 Coulomb Interaction With Particle Mesh Ewald

The Particle Mesh Ewald (PME) algorithm[11] is similar to Ewald summation, but instead of calculating the reciprocal space sum directly, it first distributes the particle charges onto nodes of a rectangular mesh using 5th order B-splines. By using a Fast Fourier Transform, the sum can then be computed very quickly, giving performance that scales as O(N log N) in the number of particles (assuming the volume of the periodic box is proportional to the number of particles).

As with Ewald summation, the user specifies the direct space cutoff $r_{cutoff}$ and error tolerance $\delta$. NonbondedForce then selects $\alpha$ as

$$\alpha = \sqrt{-\log(2\delta)/r_{cutoff}}$$

and the number of nodes in the mesh along each dimension as

$$n_{mesh} = \frac{2\alpha d}{3\delta^{1/5}}$$

where $d$ is the width of the periodic box along that dimension. (Note that some Platforms may choose to use a larger value of $n_{mesh}$ than that given by this equation. For example, some FFT implementations require the mesh size to be a multiple of certain small prime numbers, so a Platform might round it up to the nearest permitted value. It is guaranteed that $n_{mesh}$ will never be smaller than the value given above.)

The comments in the previous section regarding the interpretation of δ for Ewald summation also apply to PME, but even more so. The behavior of the error for PME is more complicated than for simple Ewald summation, and while the above formulas will usually produce an average relative error in the forces less than or similar to δ, this is not a rigorous guarantee. PME is also more sensitive to numerical round-off error than Ewald summation. For Platforms that do calculations in single precision, making δ too small (typically below about $5 \cdot 10^{-5}$) can actually cause the error to increase.

## 14.7 GBSAOBCForce

### 14.7.1 Generalized Born Term

GBSAOBCForce consists of two energy terms: a Generalized Born Approximation term to represent the electrostatic interaction between the solute and solvent, and a surface area term to represent the free energy cost of solvating a neutral molecule. The Generalized Born energy is given by[12]

$$E = -\frac{1}{2}\left(\frac{1}{\varepsilon_{solute}} - \frac{1}{\varepsilon_{solvent}}\right)\sum_{i,j}\frac{q_i q_j}{f^{GB}(d_{ij}, R_i, R_j)}$$

where the indices $i$ and $j$ run over all particles, $\varepsilon_{solute}$ and $\varepsilon_{solvent}$ are the dielectric constants of the solute and solvent respectively, $q_i$ is the charge of particle $i$, and $d_{ij}$ is the distance between particles $i$ and $j$. $f^{GB}(d_{ij}, R_i, R_j)$ is defined as

$$f^{GB}(d_{ij}, R_i, R_j) = \left[d_{ij}^{\,2} + R_i R_j \exp\left(\frac{-d_{ij}}{4R_i R_j}\right)\right]^{1/2}$$

$R_i$ is the Born radius of particle *i*, which calculated as

$$R_i = \frac{1}{\rho_i^{-1} - \rho_i^{-1}\tanh(\alpha\Psi_i - \beta\Psi_i^2 + \gamma\Psi_i^3)}$$

where α, β, and γ are the GB$^{OBC}$II parameters α = 1, β = 0.8, and γ = 4.85. $\rho_i$ is the adjusted atomic radius of particle *i*, which is calculated from the atomic radius $r_i$ as $\rho_i = r_i$-0.009 nm. $\Psi_i$ is calculated as an integral over the van der Waals spheres of all particles outside particle *i*:

$$\Psi_i = \frac{\rho_i}{4\pi}\int_{VDW}\theta(|\mathbf{r}| - \rho_i)\frac{1}{|\mathbf{r}|^4}d^3\mathbf{r}$$

where $\theta(r)$ is a step function that excludes the interior of particle *i* from the integral.

### 14.7.2 Surface Area Term

The surface area term is given by[13-14]

$$E = 4\pi \cdot 2.26\sum_i (r_i + r_{solvent})^2\left(\frac{r_i}{R_i}\right)^6$$

where $r_i$ is the atomic radius of particle *i*, $R_i$ is its Born radius, and $r_{solvent}$ is the solvent radius, which is taken to be 0.14 nm.

## 14.8 GBVIForce

The GBVI force is an implicit solvent force based on an algorithm developed by Paul Labute.[15] The GBVI force is currently undergoing testing to validate that it is correctly implementing the algorithm. The GBVI energy is given by Equation 2 of the referenced paper:

$$E = -\frac{1}{2}\left(\frac{1}{\varepsilon_{solute}} - \frac{1}{\varepsilon_{solvent}}\right)\sum_{i,j}\frac{q_i q_j}{f^{GB}(d_{ij}, R_i, R_j)} + \sum_i^n \gamma_i \left(\frac{r_i}{R_i}\right)^3$$

where the indices $i$ and $j$ run over all n particles, $\varepsilon_{solute}$ and $\varepsilon_{solvent}$ are the dielectric constants of the solute and solvent respectively, $q_i$ is the charge of particle $i$, $d_{ij}$ is the distance between particles $i$ and $j$, $r_i$ are the input particle radii, and the $\gamma_i$ are adjustable parameters. $f^{GB}(d_{ij}, R_i, R_j)$ is defined as above (Section 10.6) for the GBSAOBCForce. The Born radii, $R_i$, are defined by the equation

$$R_i = \left[r_i^{-3} - \sum_j^n V(d_{ij}, r_i, S_j)\right]^{-\frac{1}{3}}$$

where V(d,r,S) is given by

$$V(d,r,S) = \begin{Bmatrix} L(d,x,S)\big|_{x=\max(r,d-S)}^{x=d+S} & |r-S|<d \\ 0 & 0 \le d \le r-S \\ L(d,x,S)\big|_{x=d-S}^{x=d+S} & 0 \le d \le S-r \end{Bmatrix}$$

and

$$L(d,x,S) = \frac{3}{2}\left[\frac{1}{4dx^2} - \frac{1}{3x^3} + \frac{d^2 - S^2}{8dx^4}\right]$$

The $S_i$ are derived from the covalent topology of the solute:

$$S_i = 0.95 * \max\{0, \nu_i^{1/3}\}$$

$$\nu_i = r_i^3 - \frac{1}{8}\sum_j a_{ij}^2(3r_i - a_{ij}) + a_{ji}^2(3r_j - a_{ji})$$

and

$$a_{ij} = \frac{r_j^2 - (r_i - d_{ij})^2}{2d_{ij}}$$

where $d_{ij}$ is the fixed covalent bond length between particles i and j, and the sum in the calculation of the $\nu_i$ is over the particles j covalently bonded to particle i.

## 14.9 AndersenThermostat

AndersenThermostat couples the system to a heat bath by randomly selecting a subset of particles at the start of each time step, then setting their velocities to new values chosen from a Boltzmann distribution. This represents the effect of random collisions between particles in the system and particles in the heat bath.[16]

The probability that a given particle will experience a collision in a given time step is

$$P = 1 - e^{-f\Delta t}$$

where $f$ is the collision frequency and $\Delta t$ is the step size. Each component of its velocity is then set to

$$\mathbf{v}_i = \sqrt{\frac{k_B T}{m}} R$$

where $T$ is the thermostat temperature, $m$ is the particle mass, and $R$ is a random number chosen from a normal distribution with mean of zero and variance of one.

## 14.10 MonteCarloBarostat

MonteCarloBarostat models the effect of constant pressure by allowing the size of the periodic box to vary with time.[17-18] At regular intervals, it attempts a Monte Carlo step by

scaling the box vectors and the coordinates of each molecule's center by a factor *s*. The scale factor *s* is chosen to change the volume of the periodic box from *V* to *V*+$\delta V$:

$$s = \left( \frac{V + \delta V}{V} \right)^{1/3}$$

The change in volume is chosen randomly as

$$\delta V = A \cdot r$$

where *A* is a scale factor and *r* is a random number uniformly distributed between -1 and 1. The step is accepted or rejected based on the weight function

$$\Delta W = \Delta E + P\delta V - Nk_B T \ln\left( \frac{V + \delta V}{V} \right)$$

where $\Delta E$ is the change in potential energy resulting from the step, *P* is the system pressure, *N* is the number of molecules in the system, $k_B$ is Boltzmann's constant, and *T* is the system temperature. In particular, if $\Delta W \leq 0$ the step is always accepted. If $\Delta W > 0$, the step is accepted with probability $\exp(-\Delta W/k_B T)$.

This algorithm tends to be more efficient than deterministic barostats such as the Berendsen or Parrinello-Rahman algorithms, since it does not require an expensive virial calculation at every time step. Each Monte Carlo step involves two energy evaluations, but this can be done much less often than every time step. It also does not require you to specify the compressibility of the system, which usually is not known in advance.

The scale factor *A* that determines the size of the steps is chosen automatically to produce an acceptance rate of approximately 50%. It is initially set to 1% of the periodic box volume. The acceptance rate is then monitored, and if it varies too much from 50% then *A* is modified accordingly.

Each Monte Carlo step modifies particle positions by scaling the centroid of each molecule, then applying the resulting displacement to each particle in the molecule. This ensures that each molecule is translated as a unit, so bond lengths and constrained distances are unaffected.

MonteCarloBarostat assumes the simulation is being run at constant temperature as well as pressure, and the simulation temperature affects the step acceptance probability. It does not itself perform temperature regulation, however. You must use another mechanism along with it to maintain the temperature, such as LangevinIntegrator or AndersenThermostat.

## 14.11 CMMotionRemover

CMMotionRemover prevents the system from drifting in space by periodically removing all center of mass motion. At the start of every *n*'th time step (where *n* is set by the user), it calculates the total center of mass velocity of the system:

$$\mathbf{v}_{CM} = \frac{\sum_i m_i \mathbf{v}_i}{\sum_i m_i}$$

where $m_i$ and $\mathbf{v}_i$ are the mass and velocity of particle *i*. It then subtracts $\mathbf{v}_{CM}$ from the velocity of every particle.

# 15 Custom Forces

In addition to the standard forces described in the previous chapter, OpenMM provides a number of "custom" force classes. These classes provide detailed control over the mathematical form of the force by allowing the user to specify one or more arbitrary algebraic expressions. The details of how to write these custom expressions are described in section 15.8.

## 15.1 CustomBondForce

CustomBondForce is similar to HarmonicBondForce in that it represents an interaction between certain pairs of particles as a function of the distance between them, but it allows the precise form of the interaction to be specified by the user. That is, the interaction energy of each bond is given by

$$E = f(r)$$

where $f(r)$ is a user defined mathematical expression.

In addition to depending on the inter-particle distance $r$, the energy may also depend on an arbitrary set of user defined parameters. Parameters may be specified in two ways:

Global parameters have a single, fixed value. The value is stored in the Context, and may be changed in the middle of a simulation.

Per-bond parameters are defined by specifying a value for each bond. The values are part of the force definition, and therefore cannot change during a simulation.

## 15.2 CustomAngleForce

CustomAngleForce is similar to HarmonicAngleForce in that it represents an interaction between sets of three particles as a function of the angle between them, but it allows the precise form of the interaction to be specified by the user. That is, the interaction energy of each angle is given by

$$E = f(\theta)$$

where $f(\theta)$ is a user defined mathematical expression.

In addition to depending on the angle $\theta$, the energy may also depend on an arbitrary set of user defined parameters. Parameters may be specified in two ways:

Global parameters have a single, fixed value. The value is stored in the Context, and may be changed in the middle of a simulation.

Per-angle parameters are defined by specifying a value for each angle. The values are part of the force definition, and therefore cannot change during a simulation.

## 15.3 CustomTorsionForce

CustomTorsionForce is similar to PeriodicTorsionForce in that it represents an interaction between sets of four particles as a function of the dihedral angle between them, but it allows the precise form of the interaction to be specified by the user. That is, the interaction energy of each angle is given by

$$E = f(\theta)$$

where $f(\theta)$ is a user defined mathematical expression. The angle $\theta$ is guaranteed to be in the range $[-\pi, \pi]$. Like PeriodicTorsionForce, it is defined to be zero when the first and last particles are on the same side of the bond formed by the middle two particles (the *cis* configuration).

In addition to depending on the angle $\theta$, the energy may also depend on an arbitrary set of user defined parameters. Parameters may be specified in two ways:

Global parameters have a single, fixed value. The value is stored in the Context, and may be changed in the middle of a simulation.

Per-torsion parameters are defined by specifying a value for each torsion. The values are part of the force definition, and therefore cannot change during a simulation.

## 15.4 CustomNonbondedForce

CustomNonbondedForce is similar to NonbondedForce in that it represents a pairwise interaction between all particles in the System, but it allows the precise form of the interaction to be specified by the user. That is, the interaction energy between each pair of particles is given by

$$E = f(r)$$

where $f(r)$ is a user defined mathematical expression.

In addition to depending on the inter-particle distance $r$, the energy may also depend on an arbitrary set of user defined parameters. Parameters may be specified in two ways:

Global parameters have a single, fixed value. The value is stored in the Context, and may be changed in the middle of a simulation.

Per-particle parameters are defined by specifying a value for each particle. The values are part of the force definition, and therefore cannot change during a simulation.

## 15.5 CustomExternalForce

CustomExternalForce represents a force that is applied independently to each particle as a function of its position. That is, the energy of each particle is given by

$$E = f(x,y,z)$$

where $f(x, y, z)$ is a user defined mathematical expression.

In addition to depending on the particle's $(x, y, z)$ coordinates, the energy may also depend on an arbitrary set of user defined parameters. Parameters may be specified in two ways:

Global parameters have a single, fixed value. The value is stored in the Context, and may be changed in the middle of a simulation.

Per-particle parameters are defined by specifying a value for each particle. The values are part of the force definition, and therefore cannot change during a simulation.

## 15.6 CustomGBForce

CustomGBForce implements complex, multiple stage nonbonded interactions between particles. It is designed primarily for implementing Generalized Born implicit solvation models, although it is not strictly limited to that purpose.

The interaction is specified as a series of computations, each defined by an arbitrary algebraic expression. These computations consist of some number of per-particle *computed values*, followed by one or more *energy terms*. A computed value is a scalar value that is computed for each particle in the system. It may depend on an arbitrary set of global and per-particle parameters, and well as on other computed values that have been calculated before it. Once all computed values have been calculated, the energy terms and their derivatives are evaluated to determine the system energy and particle forces. The energy terms may depend on global parameters, per-particle parameters, and per-particle computed values.

Computed values can be calculated in two different ways:

- *Single particle* values are calculated by evaluating a user defined expression for each particle:

  $$value_i = f(...)$$

  where *f*(...) may depend only on properties of particle *i* (its coordinates and parameters, as well as other computed values that have already been calculated).
- *Particle pair* values are calculated as a sum over pairs of particles:

  $$value_i = \sum_{j \neq i} f(r,...)$$

  where the sum is over all other particles in the System, and *f*(*r*, ...) is a function of the distance *r* between particles *i* and *j*, as well as their parameters and computed values.

Energy terms may similarly be calculated per-particle or per-particle-pair.

- *Single particle* energy terms are calculated by evaluating a user defined expression for each particle:

  $$E = f(...)$$

  where *f*(...) may depend only on properties of that particle (its coordinates, parameters, and computed values).
- *Particle pair* energy terms are calculated by evaluating a user defined expression once for every pair of particles in the System:

  $$E = \sum_{i,j} f(r,...)$$

  where the sum is over all particle pairs $i < j$, and *f*(*r*, ...) is a function of the distance *r* between particles *i* and *j*, as well as their parameters and computed values.

Note that energy terms are assumed to be symmetric with respect to the two interacting particles, and therefore are evaluated only once per pair. In contrast, expressions for computed values need not be symmetric and therefore are calculated twice for each pair: once when calculating the value for the first particle, and again when calculating the value for the second particle.

Be aware that, although this class is extremely general in the computations it can define, particular Platforms may only support more restricted types of computations. In particular, all currently existing Platforms require that the first computed value *must* be a particle pair computation, and all computed values after the first *must* be single particle computations. This is sufficient for most Generalized Born models, but might not permit some other types of calculations to be implemented.

## 15.7 CustomHbondForce

CustomHbondForce supports a wide variety of energy functions used to represent hydrogen bonding. It computes interactions between "donor" particle groups and "acceptor" particle groups, where each group may include up to three particles. Typically a donor group consists of a hydrogen atom and the atoms it is bonded to, and an acceptor group consists of a negatively charged atom and the atoms it is bonded to. The interaction energy between each donor group and each acceptor group is given by

$$E = f(\{r_i\},\{\theta_i\},\{\phi_i\})$$

where $f(...)$ is a user defined mathematical expression. It may depend on an arbitrary set of distances $\{r_i\}$, angles $\{\theta_i\}$, and dihedral angles $\{\phi_i\}$.

Each distance, angle, or dihedral is defined by specifying a sequence of particles chosen from the interacting donor and acceptor groups (up to six atoms to choose from, since each group may contain up to three atoms). A distance variable is defined by two particles, and equals the distance between them. An angle variable is defined by three particles, and equals the angle between them. A dihedral variable is defined by four particles, and equals the angle

between the first and last particles about the axis formed by the middle two particles. It is equal to zero when the first and last particles are on the same side of the axis.

In addition to depending on distances, angles, and dihedrals, the energy may also depend on an arbitrary set of user defined parameters. Parameters may be specified in three ways:

Global parameters have a single, fixed value. The value is stored in the Context, and may be changed in the middle of a simulation.

Per-donor parameters are defined by specifying a value for each donor group. The values are part of the force definition, and therefore cannot change during a simulation.

Per-acceptor parameters are defined by specifying a value for each acceptor group. The values are part of the force definition, and therefore cannot change during a simulation.

## 15.8 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.

# 16 Integrators

## 16.1 VerletIntegrator

VerletIntegrator implements the leap-frog Verlet integration method. The positions and velocities stored in the context are offset from each other by half a time step. In each step, they are updated as follows:

$$\mathbf{v}_i(t+\Delta t/2) = \mathbf{v}_i(t-\Delta t/2) + \mathbf{f}_i(t)\Delta t/m_i$$
$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t+\Delta t/2)\Delta t$$

where $\mathbf{v}_i$ is the velocity of particle $i$, $\mathbf{r}_i$ is its position, $\mathbf{f}_i$ is the force acting on it, $m_i$ is its mass, and $\Delta t$ is the time step.

Because the positions are always half a time step later than the velocities, care must be used when calculating the energy of the system. In particular, the potential energy and kinetic energy in a State correspond to different times, and you cannot simply add them to get the total energy of the system. Instead, it is better to retrieve States after two successive time steps, calculate the on-step velocities as

$$\mathbf{v}_i(t) = \frac{\mathbf{v}_i(t-\Delta t/2) + \mathbf{v}_i(t+\Delta t/2)}{2}$$

then use those velocities to calculate the kinetic energy at time $t$.

## 16.2 LangevinIntegator

LangevinIntegator simulates a system in contact with a heat bath by integrating the Langevin equation of motion:

$$m_i \frac{d\mathbf{v}_i}{dt} = \mathbf{f}_i - \gamma m_i \mathbf{v}_i + \mathbf{R}_i$$

where $\mathbf{v}_i$ is the velocity of particle $i$, $\mathbf{f}_i$ is the force acting on it, $m_i$ is its mass, $\gamma$ is the friction coefficient, and $\mathbf{R}_i$ is an uncorrelated random force whose components are chosen from a normal distribution with mean zero and variance $2m_i\gamma k_B T$, where $T$ is the temperature of the heat bath.

The integration is done using a leap-frog method similar to VerletIntegrator.[19] The same comments about the offset between positions and velocities apply to this integrator as to that one.

## 16.3 BrownianIntegrator

BrownianIntegrator simulates a system in contact with a heat bath by integrating the Brownian equation of motion:

$$\frac{d\mathbf{r}_i}{dt} = \frac{1}{\gamma m_i}\mathbf{f}_i + \mathbf{R}_i$$

where $\mathbf{r}_i$ is the position of particle $i$, $\mathbf{f}_i$ is the force acting on it, $\gamma$ is the friction coefficient, and $\mathbf{R}_i$ is an uncorrelated random force whose components are chosen from a normal distribution with mean zero and variance $2k_B T/m_i\gamma$, where $T$ is the temperature of the heat bath.

The Brownian equation of motion is derived from the Langevin equation of motion in the limit of large $\gamma$. In that case, the velocity of a particle is determined entirely by the instantaneous force acting on it, and kinetic energy ceases to have much meaning, since it disappears as soon as the applied force is removed.

## 16.4 VariableVerletIntegrator

This is very similar to VerletIntegrator, but instead of using the same step size for every time step, it continuously adjusts the step size to keep the integration error below a user specified tolerance. It compares the positions generated by Verlet integration with those that would be generated by an explicit Euler integrator, and takes the difference between them as an estimate of the integration error:

$$error = (\Delta t)^2 \sum_i \frac{|\mathbf{f}_i|}{m_i}$$

where $\mathbf{f}_i$ is the force acting on particle $i$ and $m_i$ is its mass. (In practice, the error made by the Euler integrator is usually larger than that made by the Verlet integrator, so this tends to overestimate the true error. Even so, it can provide a useful mechanism for step size control.)

It then selects the value of $\Delta t$ that makes the error exactly equal the specified error tolerance:

$$\Delta t = \sqrt{\frac{\delta}{\sum_i \frac{|\mathbf{f}_i|}{m_i}}}$$

where $\delta$ is the error tolerance. This is the largest step that may be taken consistent with the user specified accuracy requirement.

(Note that the integrator may sometimes choose to use a smaller value for $\Delta t$ than given above. For example, it might restrict how much the step size can grow from one step to the next, or keep the step size constant rather than increasing it by a very small amount. This behavior is not specified and may vary between Platforms. It is required, however, that $\Delta t$ never be larger than the value given above.)

A variable time step integrator is generally superior to a fixed time step one in both stability and efficiency. It can take larger steps on average, but will automatically reduce the step size to preserve accuracy and avoid instability when unusually large forces occur. Conversely, when each uses the same step size on average, the variable time step one will usually be

more accurate since the time steps are concentrated in the most difficult areas of the trajectory.

Unlike a fixed step size Verlet integrator, variable step size Verlet is not symplectic. This means that for a given average step size, it will not conserve energy as precisely over long time periods, even though each local region of the trajectory is more accurate. For this reason, it is most appropriate when precise energy conservation is not important, such as when simulating a system at constant temperature. For constant energy simulations that must maintain the energy accurately over long time periods, the fixed step size Verlet may be more appropriate.

### 16.5 VariableLangevinIntegrator

This is similar to LangevinIntegrator, but it continuously adjusts the step size using the same method as VariableVerletIntegrator. It is usually preferred over the fixed step size Langevin integrator for the reasons given above. Furthermore, because Langevin dynamics involves a random force, it can never be symplectic and therefore the fixed step size Verlet integrator's advantages do not apply to the Langevin integrator.

# 17 Other Tools

### 17.1 LocalEnergyMinimizer

This provides an implementation of the L-BFGS optimization algorithm.[20] Given a Context specifying initial particle positions, it searches for a nearby set of positions that represent a local minimum of the potential energy. Distance constraints are enforced during minimization by adding a harmonic restraining force to the potential function. The strength of the restraining force is steadily increased until the minimum energy configuration satisfies all constraints to within the tolerance specified by the Context's Integrator.

### 17.2 XMLSerializer

This provides the ability to "serialize" a System object to a portable XML format, then reconstruct it again later. The XML data contains a complete copy of the entire system definition, including all Forces that have been added to it.

Here are some examples of uses for this class:

1. A model building utility could generate a System in memory, then serialize it to a file on disk. Other programs that perform simulation or analysis could then reconstruct the model by simply loading the XML file.
2. When running simulations on a cluster, all model construction could be done on a single node. The Systems could then be encoded as XML, allowing them to be easily transmitted to other nodes.

XMLSerializer is a templatized class that, in principle, can be used to serialize any type of object. At present, however, System is the only class that is supported.

# 18 Bibliography


1. 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.
2. Ren, P.; Ponder, J. W., Polarizable Atomic Multipole Water Model for Molecular Mechanics Simulation. *Journal of Physical Chemistry B* **2003,** *107*, 5933-5947.
3. Craig, I. R.; Manolopoulos, D. E., Quantum statistics and classical mechanics: Real time correlation functions from ring polymer molecular dynamics. *Journal of Chemical Physics* **2004,** *121*, 3368-3373.
4. Parrinello, M.; Rahman, A., Study of an F center in molten KCl. *Journal of Chemical Physics* **1984,** *80* (2), 860-867.
5. Markland, T. E.; Manolopoulos, D. E., An efficient ring polymer contraction scheme for imaginary time path integral simulations. *Journal of Chemical Physics* **2008,** *129* (2).
6. Hall, R. W.; Berne, B. J., Nonergodicity in path integral molecular dynamics. *Journal of Chemical Physics* **1984,** *81* (8).
7. Ceriotti, M.; Parrinello, M.; Markland, T. E.; Manolopoulos, D. E., Efficient stochastic thermostatting of path integral molecular dynamics. *Journal of Chemical Physics* **2010,** *133* (12).
8. Shirts, M. R.; Mobley, D. L.; Chodera, J. D.; Pande, V. S., Accurate and Efficient Corrections for Missing Dispersion Interactions in Molecular Simulations. *Journal of Physical Chemistry B* **2007,** *111*, 13052-13063.
9. Tironi, I. G.; Sperb, R.; Smith, P. E.; van Gunsteren, W. F., A generalized reaction field method for molecular dynamics simulations. *Journal of Chemical Physics* **1995,** *102* (13), 5451-5459.
10. Toukmaji, A. Y.; Board Jr, J. A., Ewald summation techniques in perspective: a survey. *Computer Physics Communications* **1996,** *95*, 73-92.
11. Essmann, U.; Perera, L.; Berkowitz, M. L.; Darden, T.; Lee, H.; Pedersen, L. G., A smooth particle mesh Ewald method. *Journal of Chemical Physics* **1995,** *103* (19), 8577-8593.
12. 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.
13. Schaefer, M.; Bartels, C.; Karplus, M., Solution conformations and thermodynamics of structured peptides: molecular dynamics simulation with an implicit solvation model. *Journal of Molecular Biology* **1998,** *284* (3), 835-848.
14. Ponder, J., Personal communication. This expression differs slightly from that given by Schaefer et al. This form was found to give a better correlation with surface area. ed.
15. Labute, P., The generalized Born/volume integral implicit solvent model: Estimation of the free energy of hydration using London dispersion instead of atomic surface area. *Journal of Computational Chemistry* **2008,** *29* (10), 1693-1698.
16. Andersen, H. C., Molecular dynamics simulations at constant pressure and/or temperature. *Journal of Chemical Physics* **1980,** *72* (4), 2384-2393.

17. Chow, K.-H.; Ferguson, D. M., Isothermal-isobaric molecular dynamics simulations with Monte Carlo volume sampling. *Computer Physics Communications* **1995,** *91*, 283-289.
18. Åqvist, J.; Wennerström, P.; Nervall, M.; Bjelic, S.; Brandsdal, B. O., Molecular dynamics simulations of water and biomolecules with a Monte Carlo constant pressure algorithm. *Chemical Physics Letters* **2004,** *384*, 288-294.
19. Izaguirre, J. A.; Sweet, C. R.; Pande, V. S., Multiscale dynamics of macromolecules using Normal Mode Langevin. *Pacific Symposium on Biocomputing* **2010,** *15*, 240-251.
20. Liu, D. C.; Nocedal, J., On the Limited Memory BFGS Method For Large Scale Optimization. *Mathematical Programming* **1989,** *45*, 503-528.