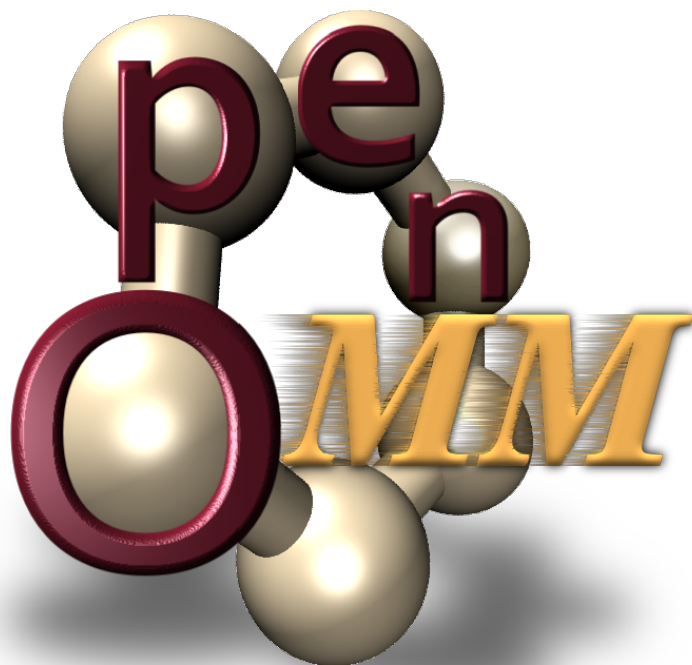




Users Manual and Theory Guide

Release 1.0

January 15, 2010

Website: simtk.org/home/openmm

OpenMM Users Manual and Theory Guide

Authors

Kyle Beauchamp
Christopher Bruns
Peter Eastman
Mark Friedrichs
Joy P. Ku
Vijay Pande
Randy Radmer
Michael Sherman

Copyright and Permission Notice

Portions copyright (c) 2008-2010 Stanford University and the Authors

Contributors: Kyle Beauchamp, Christopher Bruns, Peter Eastman, Mark Friedrichs, Joy P. Ku, Vijay Pande, Randy Radmer, Michael Sherman

Permission is hereby granted, free of charge, to any person obtaining a copy of this document (the "Document"), to deal in the Document without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Document, and to permit persons to whom the Document is furnished to do so, subject to the following conditions:

This copyright and permission notice shall be included in all copies or substantial portions of the Document.

THE DOCUMENT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS, CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENT OR THE USE OR OTHER DEALINGS IN THE DOCUMENT.

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

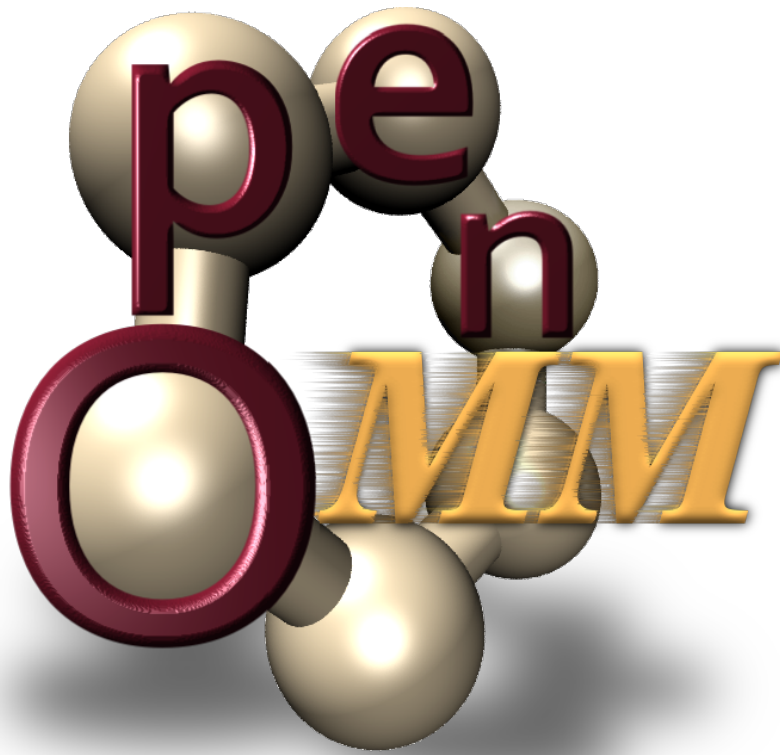
PART I: USERS MANUAL

1	INTRODUCTION	11
1.1	What Is OpenMM?.....	11
1.2	OpenMM Version 1.0.....	11
1.3	Using this Manual.....	12
1.3.1	<i>Organization of this document</i>	<i>12</i>
1.3.2	<i>How to get started.....</i>	<i>12</i>
1.4	Online Resources	12
1.5	Referencing OpenMM	13
1.6	Acknowledgements and License	13
2	OPENMM DESIGN AND API OVERVIEW	15
2.1	Design Principles	15
2.2	Choice of Language.....	16
2.3	Architectural Overview.....	18
2.4	The OpenMM Public API	19
2.5	The OpenMM Low Level API.....	21
2.6	Platforms	23
3	INSTRUCTIONS FOR PRE-COMPILED OPENMM BINARIES AND GPU SOFTWARE	24
3.1	Prerequisites	24
3.2	Quick Instructions	25
3.3	Installing OpenMM	26
3.3.1	<i>Visual Studio 8 version</i>	<i>26</i>
3.3.2	<i>Visual Studio 9 version.....</i>	<i>26</i>
3.3.3	<i>Linux</i>	<i>27</i>
3.3.4	<i>Mac OSX.....</i>	<i>27</i>
3.4	Installing GPU Software	28
3.4.1	<i>Installing CUDA for NVIDIA GPUs</i>	<i>28</i>
4	COMPILING OPENMM FROM SOURCE CODE	37

4.1	Prerequisites.....	37
4.1.1	Get a C++ compiler.....	37
4.1.2	Install CMake.....	38
4.1.3	Get the OpenMM source code.....	38
4.2	Step 1: Configure with CMake	39
4.2.1	Build and source directories.....	39
4.2.2	Starting CMake	39
4.3	Step 2: Generate Build Files with CMake	41
4.3.1	Windows	41
4.3.2	Mac and Linux	41
4.4	Step 3: Build OpenMM	42
4.4.1	Windows	42
4.4.2	Mac and Linux	42
4.5	Step 4: Test your build.....	42
4.5.1	Windows.....	42
4.5.2	Mac and Linux.....	42
4.6	Step 5: Install OpenMM	43
4.6.1	Windows	43
4.6.2	Mac and Linux	43
4.7	Step 6: Set Your PATH Variables	43
5	OPENMM TUTORIALS	44
5.1	Download Example Files	44
	Example Files Overview.....	44
5.3	Running Example Files.....	45
5.3.1	Visual Studio	45
5.3.2	Mac OS X/Linux.....	48
5.4	HelloArgon Program.....	50
5.4.1	Including OpenMM-defined functions.....	50
5.4.2	Running a program on GPU platforms	51
5.4.3	Running a simulation using the OpenMM public API	51
5.4.4	Error handling for OpenMM	54
5.4.5	Writing out PDB files	54
5.4.6	HelloArgon output	55
5.5	HelloSodiumChloride Program.....	55
5.5.1	Simple molecular dynamics system	56
5.5.2	Interface routines.....	58

5.6	HelloEthane Program.....	66
6	USING OPENMM WITH SOFTWARE WRITTEN IN LANGUAGES OTHER THAN C++	69
6.1	C API.....	70
6.1.1	<i>Mechanics of using the C API.....</i>	<i>70</i>
6.1.2	<i>Mapping from the C++ API to the C API.....</i>	<i>71</i>
6.1.3	<i>Exceptions</i>	<i>72</i>
6.1.4	<i>OpenMM_Vec3 helper type.....</i>	<i>72</i>
6.1.5	<i>Array helper types</i>	<i>72</i>
6.2	Fortran 95 API	75
6.2.1	<i>Mechanics of using the Fortran API</i>	<i>75</i>
6.2.2	<i>Mapping from the C++ API to the Fortran API</i>	<i>76</i>
6.2.3	<i>OpenMM_Vec3 helper type</i>	<i>77</i>
6.2.4	<i>Array helper types.....</i>	<i>77</i>
7	EXAMPLES OF OPENMM INTEGRATION	81
7.1	GROMACS.....	81
7.2	PyMD	82
7.2.1	<i>OpenMM integration</i>	<i>83</i>
8	TESTING AND VALIDATION OF OPENMM	86
8.1	Description of Tests	87
8.1.1	<i>Unit tests.....</i>	<i>87</i>
8.1.2	<i>System tests</i>	<i>87</i>
8.1.3	<i>Direct comparisons between GROMACS and OpenMM forces.....</i>	<i>88</i>
8.2	Test Results	89
8.2.1	<i>Unit tests.....</i>	<i>89</i>
8.2.2	<i>System tests.....</i>	<i>89</i>
8.2.3	<i>GROMACS-Reference platform differences</i>	<i>92</i>
8.3	Validation Software	93
9	THE THEORY BEHIND OPENMM: AN INTRODUCTION	98
9.1	Overview.....	98
9.2	Units	99
10	STANDARD FORCES	100
10.1	HarmonicBondForce	100

10.2	HarmonicAngleForce.....	100
10.3	PeriodicTorsionForce	101
10.4	RBTorsionForce	101
10.5	NonbondedForce	101
10.5.1	<i>Lennard-Jones Interaction</i>	102
10.5.2	<i>Coulomb Interaction Without Cutoff</i>	103
10.5.3	<i>Coulomb Interaction With Cutoff</i>	103
10.5.4	<i>Coulomb Interaction With Ewald Summation</i>	104
10.5.5	<i>Coulomb Interaction With Particle Mesh Ewald</i>	105
10.6	GBSAOBCForce.....	106
10.6.1	<i>Generalized Born Term</i>	106
10.6.2	<i>Surface Area Term</i>	107
10.7	GBVIForce	108
10.8	AndersenThermostat	109
10.9	CMMotionRemover	110
11	CUSTOM FORCES.....	111
11.1	CustomBondForce	111
11.2	CustomNonbondedForce.....	111
11.3	CustomExternalForce	112
11.4	CustomGBForce	113
11.5	Writing Custom Expressions.....	115
12	INTEGRATORS.....	115
12.1	VerletIntegrator	116
12.2	LangevinIntegrator.....	116
12.3	BrownianIntegrator	117
12.4	VariableVerletIntegrator	117
12.5	VariableLangevinIntegrator	119
13	BIBLIOGRAPHY	120



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

The target audience for the API is developers of simulation software. It is explicitly *not* targeted at computational biologists or other people who want to run simulations. They will continue to use the same software packages they currently do. OpenMM is targeted at the developers of those packages, and offers them a way to easily take advantage of a variety of high performance architectures.

1.2 OpenMM Version 1.0

Most parts of the current release are stable and suitable for production use. There are a few exceptions to that. The OpenCL Platform should still be considered beta quality. It is complete and works correctly on some OpenCL implementations, but it is not guaranteed to work or produce correct results on all implementations. The GBVIForce class should also be considered beta quality, since it has not yet been extensively tested. Finally, the CustomGBForce class is still under development. It works correctly, but it has only been implemented on the Reference and OpenCL Platforms, and its API is likely to 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 7 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>.

Two different licenses are used for different parts of OpenMM. The public API, the low level API, and the reference platform are all distributed under the MIT license. This is a very permissive license which allows them to be used in almost any way, requiring only that you retain the copyright notice and disclaimer when distributing them.

The CUDA and OpenCL platforms are distributed under the GNU Lesser General Public License (LGPL). This also allows you to use, modify, and distribute them in any way you want, but it requires you to also distribute the source code for your modifications. This restriction applies only to modifications to OpenMM itself; you need not distribute the source code to applications that use it.

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 should be narrow in scope.

We have intentionally restricted the API to only those features which directly support the goal stated above: allowing developers of simulation software to support high performance architectures. For example, it does not include any routines for reading or writing files, any model building utilities, or any analysis features. These are outside the scope of OpenMM. Any simulation package will certainly need the ability to read files, build molecular models based on them, etc., but OpenMM does not help with those tasks. It does not try to solve all possible problems, only to solve one particular problem well.

2. 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.

3. 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.

4. 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.

5. 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 and Fortran, and a Python wrapper is under development. 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, avoid multiple inheritance altogether, and use exceptions only to report user errors, never within computational kernels. 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:

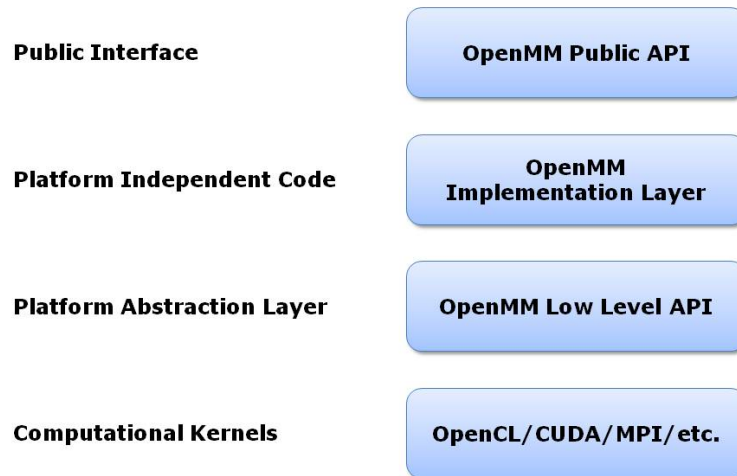


Figure 2.1: OpenMM architecture

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.

OpenMM Release 1.0 provides the following Force subclasses: HarmonicBondForce, CustomBondForce, HarmonicAngleForce, PeriodicTorsionForce, RBTorsionForce, NonbondedForce, CustomNonbondedForce, GBSAOBCForce, GBVIForce, CustomGBForce, CustomExternalForce, AndersenThermostat, and CMMotionRemover.

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. OpenMM Release 1.0 provides the following Integrator subclasses: LangevinIntegrator, VerletIntegrator, BrownianIntegrator, VariableLangevinIntegrator, and VariableVerletIntegrator.

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. It should still be considered beta quality, and is not yet recommended for actual simulations.

3 Instructions for Pre-Compiled OpenMM Binaries and GPU Software

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

- Windows (Visual Studio 8 and 9)
- Linux (32 and 64 bit)
- Mac OS X (10.5 or later)

Source code is also available. Instructions for compiling OpenMM from source code are provided in Chapter 5.

3.1 Prerequisites

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

- A C++ compiler
- gcc on Mac/Linux - We have tested the examples on Centos 5.2 with gcc 4.1.2 and on Mac OS X 10.5.6 and 10.5.7 with gcc 4.0.1
- Visual Studio 8 or 9 on Windows - You can download a free version of Visual C++ 2008 Express Edition (similar to Visual Studio 9) from <http://www.microsoft.com/express/vc/>
- OpenMM pre-compiled binaries for your platform (see Section 4.0 below)
- OpenMM example files (see Chapter 5 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 o).

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 and save them to C:\ProgramFiles\OpenMM (Windows) or /usr/local/openmm (Mac OS X/Linux).
2. Set path variables for the lib directory within the openmm or OpenMM folder – See Section o for more detailed instructions.
3. Install GPU software, if applicable – See Section o for more detailed instructions.
4. Download and unzip the OpenMMExample.zip file (available on <http://simtk.org/home/openmm>). These are also included in the /src/examples folder when you download the OpenMM source code.
5. Build and run the HelloArgon program to test the installation – see Section o for more detailed instructions.
 - o On Linux/Mac OS X, type `make`. Then, run the HelloArgon program.
 - o On Windows, double-click on HelloArgon.sln, located in the HelloArgonVS8 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 Visual Studio 8 version

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
2. Click on the “Advanced” tab or the “Advanced system settings” link
3. Click “Environment Variables”
4. Under “System variables,” double-click the line for “Path”
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 Visual Studio 9 version

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
2. Click on the “Advanced” tab
3. Click “Environment Variables”
4. Under “System variables,” select the line for “Path”
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”

3.3.3 Linux

Extract all files from the zip file and place them in /usr/local/openmm. 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=/usr/<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.

3.3.4 Mac OSX

Extract all files from the zip file and place them in /usr/local/openmm. 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:

http://www.nvidia.com/object/cuda_learn_products.html

You also need to install CUDA (for NVIDIA 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 2.2 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 the CUDA SDK code samples. We recommend using version 2.2, as some of our machines had driver

incompatibility issues with later versions, but later versions might work for your machine. The driver and toolkit are needed to get the GPU acceleration. The code samples are required for testing purposes.

3. 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 CUDA SDK -> NVIDIA CUDA SDK Browser

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

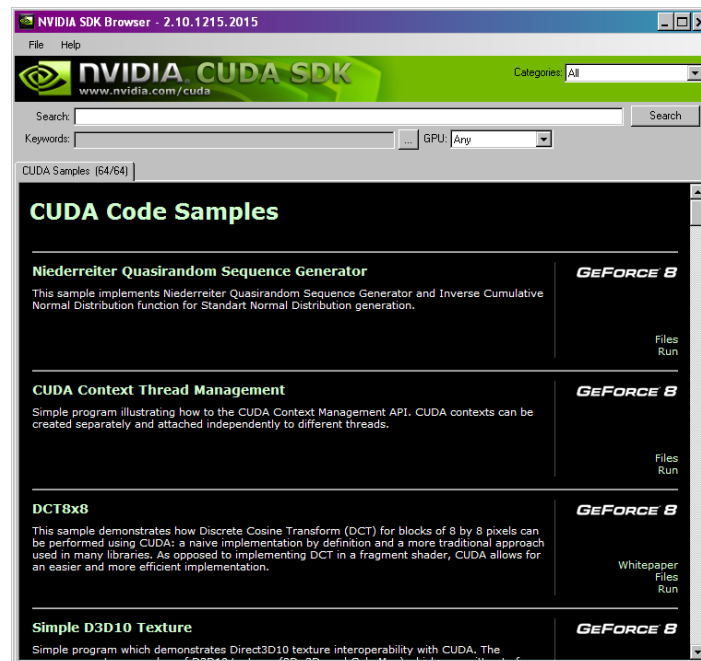


Figure 3.1: Window for browsing the NVIDIA code samples

Locate the program “Device Query” 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.

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 and the CUDA SDK code samples, version 2.2. Install the toolkit using a *custom install* and make sure all boxes, including the kernel extension are checked. With version 2.2, the driver is included; with later versions, which may work depending on your setup, the driver is a separate download, and you need to make sure you download and install that as well. The toolkit and driver are needed to get the GPU acceleration. The code samples are required for testing purposes.
3. 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 will need to add it to your `bash_profile`.

- c. Within the terminal window, navigate to the location of the code samples. If you installed everything in the default directories, then you would type one of the following (depending on the version of SDK that you downloaded):

```
cd /Developer/CUDA
```

OR

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

- d. Compile the test programs by typing:

```
make
```

- e. Navigate to the location of the compiled programs by typing one of the following (depending on the version of SDK that you downloaded):

```
cd /Developer/CUDA/bin/darwin/release
```

OR

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

- f. 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 new 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 3.4 or 4.x through 4.2. 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 the CUDA SDK code samples, version 2.3. 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. 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.

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 Centos 5.2 with gcc 4.1.2 and on Mac OS X 10.5.6 and 10.5.7 with gcc 4.0.1.

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 9 (2008) or 8 (2005). You can download a free version of Visual C++ 9 2008 (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 2.6 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.

4.2 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 Mac OS X 10.6 (Snow Leopard) with gcc 4.2

CUDA 2.3 is not compatible with gcc 4.2 on Snow Leopard. You must use gcc version 4.0 instead if you wish to use the CUDA platform. This situation is further complicated by the

need to specify the name of a directory containing the correct gcc compiler as an argument to the CUDA nvcc compiler.

1. Create a new directory to contain the gcc version 4.0 compiler, such as `/Users/joe/GCC_4.0`
2. Copy the gcc 4.0 compiler to the new directory (e.g. `cp /usr/bin/gcc-4.0 /Users/joe/GCC_4.0/`)
3. In the cmake interface, set the cmake `CMAKE_C_COMPILER` variable to `/usr/bin/gcc-4.0`
4. In the cmake interface, set the cmake `CMAKE_CXX_COMPILER` variable to `/usr/bin/g++-4.0`
5. In the cmake interface, add `“;--compiler-bindir=/Users/joe/GCC_4.0”` to the `CUDA_NVCC_FLAGS` variable. Do not overwrite the other nvcc parameters that are already there.
6. Proceed to the “All Platforms” section below.

4.2.2.3 Windows

On Windows, perform the following steps:

- Click Start->All Programs->CMake 2.6->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 8, if that is what you have installed)
- Follow the instructions in the “All Platforms” section below.

4.2.2.4 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).

- 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.

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 PATH Variables

- Refer to Section 3.3 on

Installing OpenMM to set your PATH variables to point to your new OpenMM installation.

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

5 OpenMM Tutorials

5.1 Download Example Files

Go to <http://simtk.org/home/openmm> and click on “Downloads.” Select the OpenMMExamples.zip file and unzip them to wherever you like. Note: the examples are also included in the examples folder when you download the OpenMM source code.

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 6. HelloArgon and

HelloSodiumChloride also serve as examples of how to do these mappings. The sections below describe the HelloArgon, HelloSodiumChloride, and HelloEthan 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

5.3 Running Example Files

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

5.3.1 Visual Studio

Navigate to wherever you saved the example files. Descend into the directory folder HelloArgonVS8. 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.

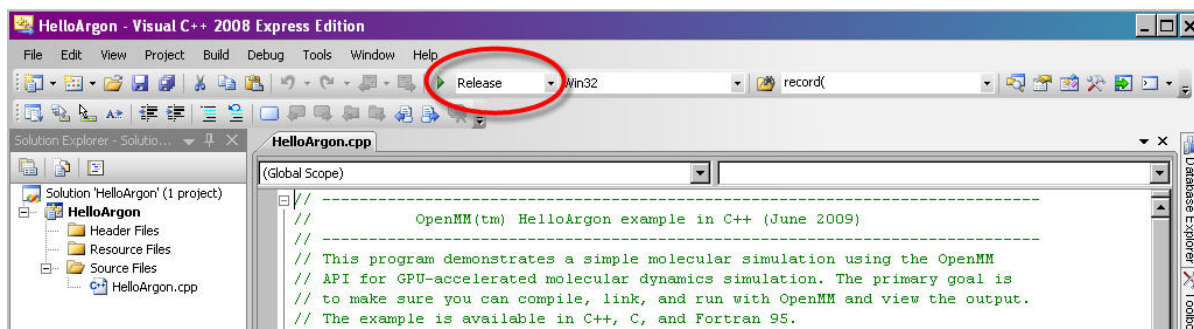


Figure 5.1: Setting "Solution Configuration" to "Release" mode in Visual Studio

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.

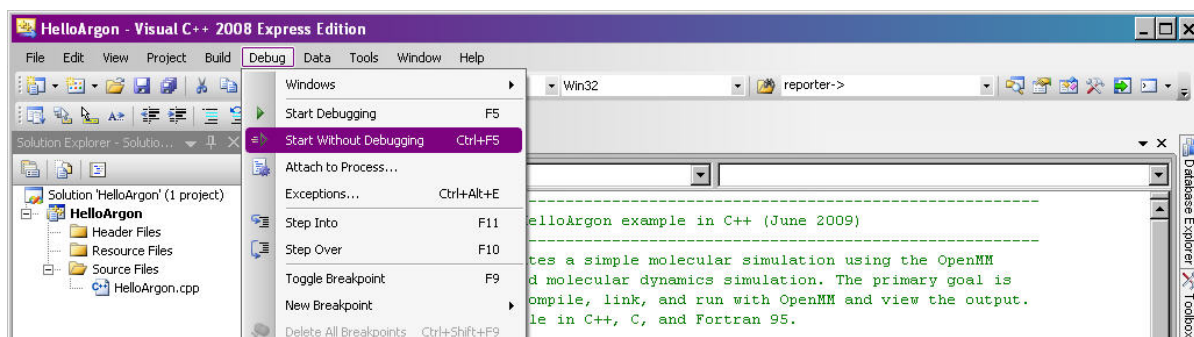


Figure 5.2: Run a program in Visual Studio

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

```

5.3.1.1 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 platform). It will say one of the following:

```

REMARK    Using OpenMM platform Reference
REMARK    Using OpenMM platform Cuda

```

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

5.3.1.2 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 HelloArgonVS8 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.

5.3.2 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. On MacOS SnowLeopard, if you use CUDA libraries that do not support 64-bit, you will need to modify the Makefile CFLAGS variable to include the `-m32` or `-arch i386` flag.

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

```

5.3.2.1 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 platform). It will say one of the following:

```

REMARK    Using OpenMM platform Reference
REMARK    Using OpenMM platform Cuda

```

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

5.3.2.2 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
```

5.3.2.3 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 4.0. Version 4.3 of the library did not work in our test case, but we were able to compile by linking to version 4.2.1. In this case, we added the following flag `-L/usr/lib/gcc/i686-apple-darwin10/4.2.1` to `FCPPPLIBS`.

When the Makefile has been updated, type:

```
make all
```

5.4 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.

5.4.1 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"
```

5.4.2 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 ATI 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.

5.4.3 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 9.2.

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 σ (zero-energy separation distance) and well depth (see Section 10.5.1 for more details).

Note that the van der Waals radius for argon is 0.188 nm and that it has already been converted to σ (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 9.2.

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)
```

5.4.4 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!
    }
}
```

5.4.5 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!

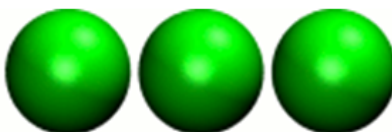
```
void writePdbFrame(int frameNum, const OpenMM::State& state)
{
    // Reference atomic positions in the OpenMM State.
    const std::vector<OpenMM::Vec3>& posInNm = state.getPositions();

    // 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.

5.4.6 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 5.3). 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.

5.5 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 6.
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 7 describes the strategies used in integrating OpenMM into real MD codes.

5.5.1 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
};
```

```

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

// -----
//                               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.
        MyOpenMMDData* 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.

5.5.2.1 *Units*

The simple molecular dynamics system described in Section 5.5.1 employs the commonly used units of angstroms and kcals. These differ from the units and parameters used within OpenMM (see Section 9.2): nanometers and kilojoules. These differences may be small but they are critical and must be carefully accounted for in the interface routines.

5.5.2.2 *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 σ , which identifies the distance at which the energy function crosses zero as the atoms move closer together than d_{min} . (See Section 10.5.1 for more details about the relationship between these).

σ turns out to be about $0.89 * d_{\text{min}}$, 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): $r = \frac{r_i + r_j}{2}$, $\epsilon = \sqrt{\epsilon_i \epsilon_j}$
- Jorgensen (used by OPLS): $r = \sqrt{r_i r_j}$, $\epsilon = \sqrt{\epsilon_i \epsilon_j}$

where r = the effective van der Waals “size” parameter (minimum radius, minimum distance, or zero crossing (sigma)), and ϵ = 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 11.1 for details.)

5.5.2.3 *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.

5.5.2.4 *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;
```

5.5.2.5 *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;

```

5.5.2.6 *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(MyOpenMMDData* omm, int numSteps) {
    omm->integrator->step(numSteps);
}

```

5.5.2.7 *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(MyOpenMMDData* omm) {
    delete omm;
}

```

5.6 HelloEthane Program

The HelloEthane program simulates ethane ($\text{H}_3\text{-C-C-H}_3$) 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 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. Currently, the C and Fortran APIs are automatically generated within the OpenMM build system. 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. The Python API will also eventually be automatically generated within the build system.
- 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 have not yet been incorporated into the OpenMM release packages, but are available as a separate download at <https://simtk.org/home/pyopenmm>.

(This doesn't apply to most users: if you are building your own OpenMM from source using CMake and want the C or Fortran APIs generated, be sure to enable the `OPENMM_BUILD_API_WRAPPERS` option.)

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 available through its own website: <https://simtk.org/home/pyopenmm>.

6.1 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.

6.1.1 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).

6.1.2 Mapping from the C++ API to the C API

The automated generator of the C “wrappers” follows the translation strategy shown in Table 6.1. 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	<code>OpenMM::</code>	<code>OpenMM_</code> (prefix)
class	<code>class OpenMM::ClassName</code>	<code>typedef OpenMM_ClassName</code>
constant	<code>OpenMM::RadiansPerDeg</code>	<code>OpenMM_RadiansPerDeg</code> (static constant)
class enum	<code>OpenMM::State::Positions</code>	<code>OpenMM_State_Positions</code>
constructor	<code>new OpenMM::ClassName()</code>	<code>OpenMM_ClassName*</code> <code>OpenMM_ClassName_create()</code> (addl. constructors are <code>_create_2()</code> , etc.)
destructor	<code>OpenMM::ClassName* thing;</code> <code>delete thing;</code>	<code>OpenMM_ClassName* thing;</code> <code>OpenMM_ClassName_destroy(thing);</code>
class method	<code>OpenMM::ClassName* thing;</code> <code>thing->someName(args)</code>	<code>OpenMM_ClassName* thing;</code> <code>OpenMM_ClassName_someName</code> (thing, args)
Boolean type & constants	<code>bool</code> <code>true</code> , <code>false</code>	<code>OpenMM_Boolean</code> <code>OpenMM_True(1)</code> , <code>OpenMM_False(0)</code>
string	<code>std::string</code>	<code>char*</code>
3-vector	<code>OpenMM::Vec3</code>	<code>typedef OpenMM_Vec3</code>
arrays	<code>std::vector<std::string></code> <code>std::vector<double></code> <code>std::vector<Vec3></code> <code>std::vector<std::pair<int,int>></code> <code>std::map<std::string,double></code>	<code>typedef OpenMM_StringArray</code> <code>typedef OpenMM_DoubleArray</code> <code>typedef OpenMM_Vec3Array</code> <code>typedef OpenMM_BondArray</code> <code>typedef OpenMM_ParameterArray</code>

Table 6.1: 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.

6.1.3 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.)

6.1.4 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);
```

6.1.5 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 6.1 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.

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

Table 6.2: Generic description of array helper types

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

6.1.5.1 *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);

```

6.1.5.2 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);
```

6.1.5.3 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);
```

6.1.5.4 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);
```

6.1.5.5 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);
```

6.2 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.

6.2.1 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).

6.2.2 Mapping from the C++ API to the Fortran API

The automated generator of the Fortran “wrappers” follows the translation strategy shown in Table 6.3. 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	<code>class</code> OpenMM::ClassName	<code>type</code> (OpenMM_ClassName)
constant	OpenMM::RadiansPerDeg	<code>parameter</code> (OpenMM_RadiansPerDeg)
class enum	OpenMM::State::Positions	<code>parameter</code> (OpenMM_State_Positions)
constructor	<code>new</code> OpenMM::ClassName()	<code>type</code> (OpenMM_ClassName) thing <code>call</code> OpenMM_ClassName_create(thing) (addl. constructors are <code>create_2()</code> , etc.)
destructor	OpenMM::ClassName* thing; <code>delete</code> thing;	<code>type</code> (OpenMM_ClassName) thing <code>call</code> OpenMM_ClassName_destroy(thing)
class method	OpenMM::ClassName* thing; thing->someName(<i>args</i>)	<code>type</code> (OpenMM_ClassName) thing <code>call</code> OpenMM_ClassName_someName (thing, <i>args</i>)
Boolean type & constants	<code>bool</code> <code>true</code> , <code>false</code>	<code>integer*4</code> <code>parameter</code> (OpenMM_True=1) <code>parameter</code> (OpenMM_False=0)
string	std::string	<code>character</code> (*)
3-vector	OpenMM::Vec3	<code>real*8</code> vec(3)
arrays	std::vector<std::string> std::vector<double> std::vector<Vec3> std::vector<std::pair<int,int>> std::map<std::string,double>	<code>type</code> (OpenMM_StringArray) <code>type</code> (OpenMM_DoubleArray) <code>type</code> (OpenMM_Vec3Array) <code>type</code> (OpenMM_BondArray) <code>type</code> (OpenMM_ParameterArray)

Table 6.3: 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.

6.2.3 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.

6.2.4 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 6.3 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.

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

Table 6.4: Generic description of array helper types

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

6.2.4.1 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
    
```

6.2.4.2 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
    
```

6.2.4.3 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)
    
```

6.2.4.4 *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
```

6.2.4.5 *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 Examples of OpenMM Integration

7.1 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.

7.2 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-matplotlib, 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.

7.2.1 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 5.4). 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.

8 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 atoms.
- **System tests:** In contrast to the unit tests, the system tests are performed on a collection of more realistic 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 and scenarios (e.g, explicit vs. implicit solvent).

Below, each type of test is outlined in greater detail, followed by a discussion of the current status of the tests.

8.1 Description of Tests

8.1.1 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.

8.1.2 System tests

Systems tests were performed to validate: 1) the consistency of the calculated forces across platforms, 2) the consistency of energy and force for each force class on each platform, and 3) energy conservation and thermostability on the Cuda platform. The tests were run on five proteins (met-enkephalin, a 10-residue poly-alanine chain, villin, lambda, bpti), 1 RNA (6tna) and 1 DNA (Dickerson dodecamer); the systems ranged in size from 75-2444 atoms.

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. 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. Note: comparisons between the Reference and Brook platform were also made for earlier releases. However, these have been discontinued due to the deprecation of Brook and the development of the OpenCL platform that will run on the ATI hardware.

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 ϵ :

$$\Delta r = -F_o^* \epsilon / |F_o|, \text{ where } \epsilon \sim 10^{-2} - 10^{-6} \text{ nm}$$

- Calculate the potential energy V at the perturbed configuration

$$V = V_0 + \nabla V \cdot \Delta r + \dots$$

$$V - V_0 \approx -F_0 \cdot \Delta r = -F_0 \cdot (-\epsilon * F_0 / |F_0|)$$

$$[V - V_0] / \epsilon \approx |F_0|$$

Here Δr is the perturbation in the coordinates of the system. The relative difference between $[V - V_0] / \epsilon$ and $|F_0|$ 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 20 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 platform, since running them on the Reference platform would require substantial computational effort.

8.1.3 Direct comparisons between GROMACS and OpenMM forces

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

- HarmonicBond
- HarmonicAngle
- PeriodicTorsion
- RBTorsion (Ryckaert-Bellemans torsion)
- Nonbonded
 - No cutoffs
 - Reaction field with cutoffs (see below)
- GBSA OBC implicit solvent

Because of the usage of charge groups in GROMACS and their absence in OpenMM, checks were only made with cutoffs greater than the system size. We are looking into whether a more judicious choice of GROMACS mdp parameters will allow us to directly test the non-bonded force with more realistic cutoffs and with and without periodic boundary conditions. Also, Ewald was not tested due to a bug in GROMACS. GROMACS 4.0.5 was used for all comparisons, except for the implicit solvent check which used an early version of GROMACS 4.1.

8.2 Test Results

8.2.1 Unit tests

The unit tests should pass, although the BrownianIntegrator test often fails; we are currently exploring the source of these failures. 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.

8.2.2 System tests

A concise summary of the results is presented below to provide you with 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 and CUDA platforms: The summary of these system tests are given in Table 8.1 below. The maximum relative difference reported is $|F_{\text{Reference}} - F_{\text{Cuda}}| / (|F_{\text{Reference}}| + |F_{\text{Cuda}}|)$, where F_i is the force on platform i , and the difference is the maximum observed over all systems. The average relative delta is the average relative difference over all systems.

Force	Max Relative Difference	Average Relative Difference
HarmonicBond	1.33e-01	4.17e-05
HarmonicAngle	3.84e-03	2.01e-05
PeriodicTorsion	6.83e-03	2.27e-05
RB Torsion	4.64e-03	9.86e-06
Nonbonded no cutoff	7.27e-05	1.71e-06
Nonbonded/cutoffs/non periodic boundary conditions	1.42e-03	1.02e-06
Nonbonded/cutoff/periodic boundary conditions	2.94e-03	9.64e-07
Ewald	9.16e-05	1.51e-06
PME	8.00e-02	4.01e-05
OBC/nonbonded/no cutoffs	4.24e-04	6.08e-06

Table 8.1: Summary of force consistency across CUDA and Reference platforms. The Max Relative Difference reported is $|F_{\text{Reference}} - F_{\text{Cuda}}| / (|F_{\text{Reference}}| + |F_{\text{Cuda}}|)$, where F_i is the force on platform i , and is the maximum observed over all systems. The Average Delta is the average difference over all systems

The large maximum relative difference for the HarmonicBond force was due to an entry from the lambda protein with a small magnitude of 3.19e-02 kJ/nm-mol. In the previous release, large relative differences were seen for the torsion forces with large absolute differences on the order of 1-0.1 kJ/mol-nm. The differences were attributed to the use of single precision arithmetic on the CPU and GPU for configurations with torsion angles very near 180°. However, it was determined the differences arose from a bug in Gromacs that was duplicated in the OpenMM code. Both Gromacs and OpenMM have removed the bug.

Energy-force consistency: The summary of these system tests are given in Table 8.2 below. The reported maximum relative difference was calculated as $|[V - V_0] / \epsilon - |F_0|| / |F_0|$, where V is the potential energy of the system, F is the force, and ϵ is the perturbation in the coordinates of the system (see test description in Section 8.1.2 above). Note: the ϵ used to perturb the coordinates in the calculations was 1.0e-04 nm for the bonded forces and 1.0e-03 nm for the non-bonded calculations, except for the 10-residue

polyalanine chain where ϵ was set to $1.0\text{e-}04$. The values shown in Table 8.2 are the maximum observed over all systems running on the CUDA platform and in Table 8.3 the corresponding quantities on the Reference platform.

Force	Max Relative Difference	Average Relative Difference
HarmonicBond	3.07e-03	1.81e-03
HarmonicAngle	3.32e-03	1.19e-03
PeriodicTorsion	9.24e-03	3.32e-03
RB Torsion	3.00e-03	1.40e-03
Nonbonded no cutoff	4.39e-03	2.03e-03
Nonbonded/cutoffs/no periodic boundary conditions	4.09e-03	1.65e-03
Nonbonded/cutoff/periodic boundary conditions	3.99e-03	1.54e-03
Ewald	3.99e-03	1.60e-03
PME	4.13e-03	1.74e-03
OBC/nonbonded/no cutoffs	6.23e-03	2.60e-03

Table 8.2: Summary of energy-force consistency for the CUDA platform. The reported maximum relative difference was calculated as $|[V-V_0]/\epsilon - |F_0||/|F_0|$, where V is the potential energy of the system, F is the force, and ϵ is the perturbation in the coordinates of the system.

Force	Max Relative Difference	Average Relative Difference
HarmonicBond	9.31e-03	3.03e-03
HarmonicAngle	5.93e-03	2.21e-03
PeriodicTorsion	5.46e-02	1.03e-02
RB Torsion	9.59e-03	5.71e-03
Nonbonded no cutoff	2.68e-02	1.09e-02
Nonbonded/cutoffs/no periodic boundary conditions	2.94e-02	1.01e-02

Nonbonded/cutoff/periodic boundary conditions	6.05e-02	1.52e-02
Ewald	6.78e-02	1.60e-02
PME	2.24e-02	7.72e-03
OBC/nonbonded/no cutoffs	5.87e-02	2.07e-02

Table 8.3: Summary of energy-force consistency for the Reference platform. The reported maximum relative difference was calculated as $|[V-V_0]/\epsilon - |F_0||/|F_0|$, where V is the potential energy of the system, F is the force, and ϵ is the perturbation in the coordinates of the system.

Energy conservation and thermostability: For the VerletIntegrator, the energy drift ranged from 3.89e-03 to 8.32e-03 kT/degrees-of-freedom/ns, and for the VariableVerletIntegrator, the energy drift ranged from 7.30e-04 to 3.87e-02 kT/degrees-of-freedom/ns for the seven biomolecules. For a specified temperature of 300 K, the average temperature ranged from [291-312]K for the LangevinIntegrator for the seven biomolecules, and for the VariableLangevinIntegrator the average temperature spanned the interval [299, 310]K. Test simulations using the BrownianIntegrator have not performed. A few hydrogen-heavy atom constraint violations were observed in simulations of the RNA system; all other systems had no constraint violations.

8.2.3 GROMACS-Reference platform differences

The summary of comparisons between GROMACS and the OpenMM Reference platform are given in

Table 8.2 below. The value reported is $|F_{\text{OpenMMReference}} - F_{\text{GROMACS}}| / (|F_{\text{OpenMMReference}}| + |F_{\text{GROMACS}}|)$, where F_i is the force computed with software i , and is the maximum observed over all systems. The tests were performed on only six of the systems used in the tests in Section 8.2.2; a tpr file was unavailable for one of the systems.

Force	Max Relative Difference	Average Relative Difference
HarmonicBond	7.73e-02	1.14e-05
HarmonicAngle	4.70e-03	5.33e-06
PeriodicTorsion	7.87e-03	3.80e-06

RB Torsion	1.80e-02	4.41e-06
Nonbonded no cutoff	2.92-03	1.71e-06
OBC/nonbonded/no cutoffs	1.01e-01	1.46e-01

Table 8.4: Comparison of forces computed by GROMACS versus the OpenMM Reference platform. The value reported is $|F_{\text{OpenMMReference}} - F_{\text{GROMACS}}| / (|F_{\text{OpenMMReference}}| + |F_{\text{GROMACS}}|)$, where F_i is the force computed with software i , and is the maximum observed over all systems. The test for OBC/nonbonded/no cutoffs was only carried out on the protein BPTI. The relatively large reported value was for a force component of size $2.41\text{e-}01$ kJ/mol-nm. The maximum difference in the norm of the forces was $2.96\text{e-}02$ kJ/mol-nm for this system.

8.3 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]



Part II

Theory Guide

9 The Theory Behind

OpenMM: an Introduction

9.1 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.

9.2 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 9.1: 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.

10 Standard Forces

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

10.1 HarmonicBondForce

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

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

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.

10.2 HarmonicAngleForce

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

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

where θ is the angle formed by the three particles, θ_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.

10.3 PeriodicTorsionForce

Each torsion is represented by an energy term of the form

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

where θ is the dihedral angle formed by the four particles, θ_0 is the equilibrium angle, n is the periodicity, and k is the force constant.

10.4 RBTorsionForce

Each torsion is represented by an energy term of the form

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

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

For reason of convention, PeriodicTorsionForce and RBTorsionForce define the torsion angle differently. θ 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 ϕ is zero when they are on *opposite* sides (the *trans* configuration). This means that $\theta = \phi - \pi$.

10.5 NonbondedForce

10.5.1 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, σ is the distance at which the energy equals zero, and ε 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, σ and ε are the parameters specified by the exception. Otherwise they are determined from the parameters of the individual particles using the Lorentz-Bertelot combining rule:

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

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 σ 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 σ and ε by

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

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

10.5.2 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.

10.5.3 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.¹

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

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

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

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

10.5.4 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*.²

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

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

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

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

In the above expressions, the indices i and j run over all particles, $\mathbf{n} = (n_1, n_2, n_3)$ runs over all copies of the periodic cell, and $\mathbf{k} = (k_1, k_2, k_3)$ 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 $\text{erfc}(\alpha r_{cutoff})$. 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 δ . 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 δ 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 δ 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.

10.5.5 Coulomb Interaction With Particle Mesh Ewald

The Particle Mesh Ewald (PME) algorithm³ 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 δ . NonbondedForce then selects α as

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

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

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

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.

10.6 GBSAOBCForce

10.6.1 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 by⁴

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

where the indices i and j run over all particles, ϵ_{solute} and $\epsilon_{solvent}$ 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 $\alpha = 1$, $\beta = 0.8$, and $\gamma = 4.85$. ρ_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. Ψ_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.

10.6.2 Surface Area Term

The surface area term is given by^{5, 6}

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

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.

10.7 GBVIForce

The GBVI force is an implicit solvent force based on an algorithm developed by Paul Labute.⁷ 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:

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

where the indices i and j run over all n particles, ϵ_{solute} and $\epsilon_{\text{solvent}}$ 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 γ_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

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

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

$$V(d, r, S) = \left\{ \begin{array}{ll} L(d, x, S) \Big|_{x=\max(r, d-S)}^{x=d+S} & |r - S| < d \\ 0 & 0 \leq d \leq r - S \\ L(d, x, S) \Big|_{x=d-S}^{x=d+S} & 0 \leq d \leq S - r \end{array} \right\}$$

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:

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

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

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 v_i is over the particles j covalently bonded to particle i .

10.8 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.⁸

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 Δt 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.

10.9 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.

11 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 11.5.

11.1 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.

11.2 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.

11.3 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.

11.4 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 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 $f(r, \dots)$ 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:

$$E = f(\dots)$$

where $f(\dots)$ may depend only on properties of that particle (its 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:

$$E = \sum_{i,j} f(r, \dots)$$

where the sum is over all particle pairs $i < j$, and $f(r, \dots)$ 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.

11.5 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, step. $\text{step}(x) = 0$ 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.

12 Integrators

12.1 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:

$$\begin{aligned}\mathbf{v}_i(t + \Delta t / 2) &= \mathbf{v}_i(t - \Delta t / 2) + \mathbf{f}_i(t) \Delta t / m_i \\ \mathbf{r}_i(t + \Delta t) &= \mathbf{r}_i(t) + \mathbf{v}_i(t + \Delta t / 2) \Delta t\end{aligned}$$

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 Δt 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 .

12.2 LangevinIntegrator

LangevinIntegrator 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, γ 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`.⁹ The same comments about the offset between positions and velocities apply to this integrator as to that one.

12.3 **BrownianIntegrator**

`BrownianIntegrator` simulates a system in contact with a heat bath by integrating the Brownian equation of motion:

$$\frac{d\mathbf{r}_i}{dt} = \frac{1}{\gamma} \mathbf{f}_i + \mathbf{R}_i$$

where $\mathbf{r}_i$ is the position of particle i , $\mathbf{f}_i$ is the force acting on it, γ 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 $2k_B T/\gamma$, 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 γ . 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.

12.4 **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 Δt that makes the error exactly equal the specified error tolerance:

$$\Delta t = \sqrt{\frac{\delta}{\sum_i \frac{|\mathbf{f}_i|}{m_i}}}$$

where δ 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 Δt 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 Δt 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.

12.5 **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.

13 Bibliography

1. 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.
2. Toukmaji, A. Y.; Board Jr, J. A., Ewald summation techniques in perspective: a survey. *Computer Physics Communications* **1996**, 95, 73-92.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. Andersen, H. C., Molecular dynamics simulations at constant pressure and/or temperature. *Journal of Chemical Physics* **1980**, 72, (4), 2384-2393.
9. van Gunsteren, W. F.; Berendsen, H. J. C., A Leap-Frog Algorithm for Stochastic Dynamics. *Molecular Simulation* **1988**, 1, 173-185.