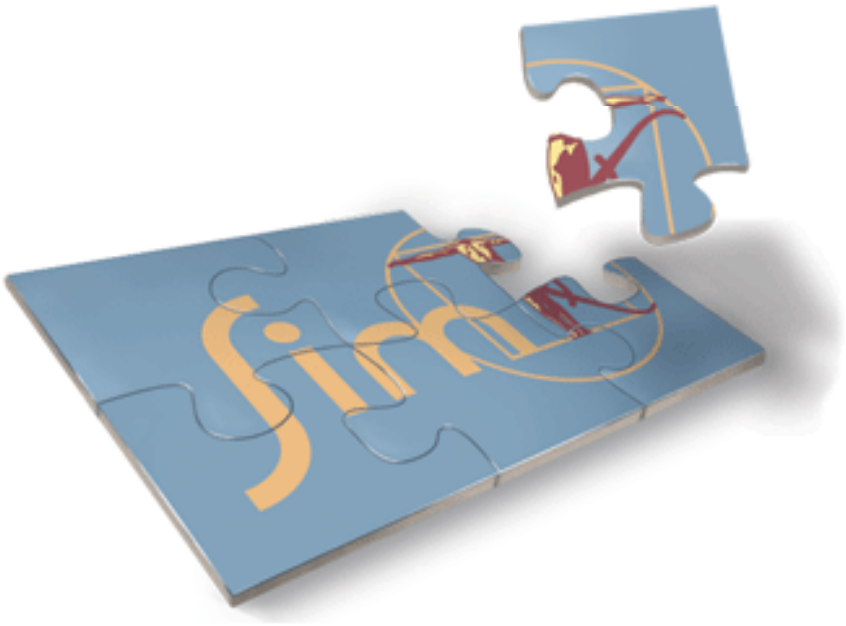
Documents



Simbody and Molmodel User's Guide

Release 3.5

December, 2014

Copyright and Permission Notice

Portions copyright (c) 2008-14 Stanford University, Peter Eastman, 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

SimTK software and related activities are funded by the Simbios National Center for Biomedical Computing at Stanford University (http://simbios/stanford.edu) through the National Institutes of Health Roadmap for Medical Research, Grant U54 GM072970. Information on the National Centers can be found at http://nihroadmap.nih.gov/bioinformatics.

Table of Contents

1 Introduction

1.1 What are SimTK, Simbody, and Molmodel?

SimTK is the Simbios Biosimulation Toolkit, a collection of open source software tools supporting high-performance physics based simulation of biological structures. SimTK includes a variety of application programs and a software development kit (SDK). The SDK includes a family of application programming interfaces (APIs) for convenient and efficient handling of the various tasks that arise in physics based simulation, including vector and matrix arithmetic, linear algebra, numerical integration, optimization, and so on. Most importantly, the SDK contains

- **Simbody**, an API for performing internal coordinate simulations of multibody systems, and
- **OpenMM**, an API for GPU-accelerated computation of molecular force fields, and
- **Molmodel**, an API that uses Simbody to build internal coordinate molecule models that can use OpenMM-accelerated force fields, and
- **OpenSim API,** which uses Simbody to build internal coordinate models of biomechanical systems, such as human musculoskeletal models for studying gait disorders.

All SimTK software is hosted and distributed via simtk.org. But note that simtk.org also serves as a biosimulation community hosting site so there are also many projects hosted there that are not part of the SimTK toolkit.

OpenMM is used independently for particle-based molecular simulations and is packaged and released separately from Simbody and Molmodel. We won't present the OpenMM API here; see https://simtk.org/home/openmm for more information. OpenSim API is also packaged and released separately from Simbody and Molmodel, as part of the OpenSim project that includes a complete GUI for musculoskeletal simulation. We won't present the OpenSim API here either; see https://simtk.org/home/opensim for more information.

Simbody is developed and distributed via GitHub at https://github.com/simbody, with description and documentation at https://simtk.org/home/simbody. It is used for biomechanical simulations (see OpenSim), as well as related areas such as robotics, avatars, and virtual world simulations.

Molmodel is developed and distributed via the project https://simtk.org/home/molmodel. It extends the Simbody API to add molecule modeling and analysis features, and is thus dependent on Simbody. It can optionally use OpenMM too, so installation of those packages is a prerequisite for installation of Molmodel.

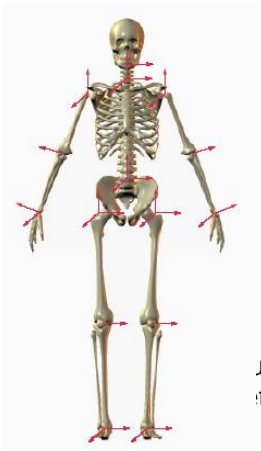
1.2 What is a multibody system?

In this guide we will discuss construction of internal coordinate multibody systems with Simbody, and include some examples of using the Molmodel API to make internal coordinate multibody models of molecules. Let's take a moment to consider what that means.

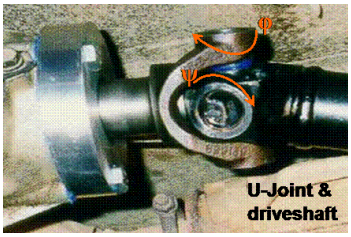
A *multibody system* is a model of a physical system composed of mass-carrying objects, each of which is rigid or nearly rigid, but which can move significantly relative to each other. Here are some examples of multibody systems:

- A human skeleton consists of rigid bones that move relative to each other by bending at joints.
- A protein may be viewed as a collection of atoms or groups of atoms that are internally rigid, but move relative to each other.
- An automobile engine consists of gears, pistons, and other rigid parts that move relative to each other to produce overall motion.

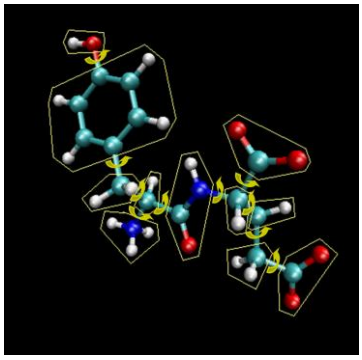
Figure 1 shows some physical systems that can be modeled as multibody systems.



ılated
ton



Mechanical U-joint



Protein backbone

Figure 1. Some multibody systems.

Now let's consider what is meant by an *internal coordinate* multibody model. Suppose you are creating a computer model of a human skeleton. One possible approach would be to independently specify the position and orientation of every bone. Although that could work, it isn't a very natural way to describe a skeleton. It omits all information about connectivity: the fact that bones are attached to each other, that they can only move in very limited ways, and that moving an arm should automatically cause the hand to move as well. You therefore would need to add a very large number of *constraints* to the system to make it move in a physically realistic way.

An alternative approach is to describe the system in terms of its internal coordinates. Rather than specifying six degrees of freedom (three translations and three rotations) for each bone, you specify only the ways in which the skeleton can actually move: for example, the angle by which the right elbow is bent. The result is a simpler, more concise description of the state of the system at any point in time. It also is computationally much more efficient, since it requires many fewer constraints. However, the underlying implementation is considerably more difficult in internal coordinates than in Cartesian coordinates.

This is what Simbody, SimTK's multibody dynamics engine, does. It allows you to describe a multibody system in whatever way is most natural. And it takes care of all the hard parts for you: transforming between internal and Cartesian coordinates, calculating the inertial forces that arise from rigid body motion, determining the effect of forces on internal coordinates, imposing constraints, integrating equations of motion in terms of internal coordinates, and many other details that you really don't want to have to worry about. And it does all of these things in ways that are efficient, robust, and numerically accurate.

1.3 Using this manual

This manual is intended as a starting place for new Simbody and Molmodel users. An overview of the theory and architecture of Simbody is given in chapter 1. Detailed installation instructions are provided in chapter 3, for Linux, Mac, or Windows installation. Then a series of tutorial examples are given in chapter 4 for a simple Simbody example. Chapter 5 steps back to take a look at some of the basic features of the Simbody API. Then we return to examples in chapter 1 which presents a more complex example using Molmodel to construct a Simbody model of a protein molecule, and then exercises a variety of Simbody features to perform studies of the molecule.

1.4 Getting more information

The Simbody project home page https://simtk.org/home/simbody is the place to start for more information. On the Documents page are this document and others including an Advanced User's Guide, the Simbody Theory Manual, technical papers, and the complete Doxygen-generated reference documentation for the Simbody API: https://simtk.org/api_docs/simbody/api_docs30/Simbody/html/index.html. (These are also available in the doc subdirectory of your Simbody installation.)

Under Public Forums, you can find the Simbody help forum, which is a great way to get personalized assistance with any problems or questions you have. If you want to report a bug or request a new feature, the feature and bug trackers are under the Advanced tab.

There is also a Wiki on which you can find, and contribute, additional practical information about using Simbody.

Molmodel-specific documentation, forums, wiki, etc. are handled analogously via the Molmodel project home page at https://simtk.org/home/molmodel. If you are a Molmodel user and aren't sure whether you are seeing something Molmodel specific, go ahead and use the Molmodel project anyway and we'll move it to Simbody if need be.

1.5 License

Simbody and Molmodel are licensed under the extremely permissive MIT license, making them freely available for any purpose. For details, see http://wiki.simtk.org/simbody/LicenseAndCopyright.

1.6 How to acknowledge us

Our license does not *require* that you acknowledge us, but we and our sponsors would be grateful if you did anyway! If our hard work has helped you with yours, please throw us a bone and mention on your "About" page and in your documentation that you are using Simbody from https://simtk.org/home/simbody. Where appropriate, please cite this paper:

> Michael A. Sherman, Ajay Seth, Scott L. Delp, "Simbody: multibody dynamics for biomedical research," *Procedia IUTAM* 2:241-261 (2011)
> doi:10.1016/j.piutam.2011.04.023

We would be particularly grateful if you mention that Simbody is primarily funded by NIH Roadmap grant U54 GM072970. We appreciate that support very much, and the NIH appreciates knowing that its funds are having an impact, particularly on medical research and human health.

2 Simbody theory and architecture overview

This chapter provides brief background material on the mathematical background for Simbody, the architecture as seen through the API, and the component modules that together constitute the installed Simbody package. For a much more detailed discussion, see the Simbody Theory Manual (available on the Simbody project's Documents page) and the following paper:

> Michael A. Sherman, Ajay Seth, Scott L. Delp, "Simbody: multibody dynamics for biomedical research," *Procedia IUTAM* 2:241-261 (2011) doi:10.1016/j.piutam.2011.04.023

You can find this and other papers on the Simbody project's Documents and Publications pages.

2.1 Mathematical Overview

In the previous section, we defined a multibody system as a model of a "physical system." That is one way to think of it, but it is not the only way. After all, simulations can only work with virtual objects, not physical ones. From the simulation's perspective, a multibody system is a *system of equations*. If you have designed it well, the behavior of those equations will in some way reflect the behavior of the physical system you are trying to simulate.

More specifically, the state of the system at any moment in time is described by a vector of *state variables*. For example, a human skeleton could be described by the current angle and angular velocity of every joint, plus the overall motion of the torso. We refer to this state vector as **y**. The job of a simulation is to numerically integrate the equations of motion

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) \tag{1}$$

to produce a complete time history $\mathbf{y}(t)$, called a *trajectory*. Here the function $\mathbf{f}(t, \mathbf{y})$ reflects the forces acting on the bodies and the laws of physics.

The vector of state variables can be subdivided into *generalized coordinates*, which we refer to as **q**, *generalized speeds*, which we refer to as **u**, and *auxiliary variables*, which we refer to as **z**. For example, the generalized coordinates of a human skeleton would be the set of angles for all the joints, and the orientation and position of the torso, while the generalized speeds would be the corresponding angular and linear velocities. If you were simulating a person walking and wanted to keep track of the total energy used as they walked, that would be an auxiliary state variable *z*; it does not describe the configuration of the skeleton, but is defined by a differential equation so would need to be integrated along with the other state variables. The full state vector **y** is the concatenation of these subvectors: **y** = [**q**, **u**, **z**].

The use of generalized coordinates avoids the need for most constraints because they are embedded in the coordinate basis. However, it sometimes is necessary to impose additional constraints on the behavior of a system. For example, some proteins contain disulfide bonds that connect two distant parts of the chain. This is typically modeled as a constraint requiring the distance between the two bonded atoms to remain fixed. Mathematically, a constraint is an algebraic equation which must be satisfied at all times during the simulation:

$$c(t, \mathbf{q}, \mathbf{u}) = 0 \tag{2}$$

There is one such equation for every constraint imposed on the system. Together, they represent a manifold on which the state vector is required to lie at all times.

The above discussion assumes the system can be modeled as evolving continuously according to a set of differential and algebraic equations. In many cases, that is not enough. A system may change discontinuously at discrete times. For example, suppose you are modeling a person walking. As long as a foot is in the air, it has no interaction with the ground. But for some methods of modeling contact, you must monitor its height, and when it touches the ground, you must turn on a constraint to prevent it from passing down into the ground or sliding along the ground. You then monitor the net force acting on the foot, and when you see that it is directed upward, you release the constraint so the foot can rise back off the ground again.

Mathematically, this is modeled with *event trigger functions*. These are arbitrary functions of time and the state variables which are monitored continuously during the simulation. An *event* is said to occur when a trigger function crosses through 0:

$$e(t, \mathbf{y}) = 0 \tag{3}$$

When an event occurs, the corresponding *event handler* is invoked, which can modify the state in arbitrary, discontinuous ways.

We also extend the state description to include a set of *discrete variables*, which we refer to as **d**. In the example above, you would use discrete variables to keep track of which constraints were currently turned on. Discrete variables are not modified by equation (1). They are changed only by event handlers, which modify them at discrete times. The forces, constraints, and event functions in equations (1)-(3) may all depend on the current values of discrete variables. Discrete variables can hold values of any type.

With discrete variables included, and showing the full set of constraints and event trigger functions, our three equations are now:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{d}; t, \mathbf{y}) \tag{4}$$

$$0 = \mathbf{c}(\mathbf{d}; t, \mathbf{q}, \mathbf{u}) \tag{5}$$

$$0 = \mathbf{e}(\mathbf{d}; t, \mathbf{y}) \tag{6}$$

We show **d** set off by a semicolon as a reminder that discrete variables are held constant during continuous intervals when t and **y** are changing.

2.2 Systems and States

Simbody uses two classes to represent the information associated with a multibody system: System and State. You can think of these as the "constant" and "mutable" (non-constant) parts of the system, respectively. A System object contains everything that is expected to remain constant over the course of a simulation (including the code), while a State object stores anything that is expected to change.

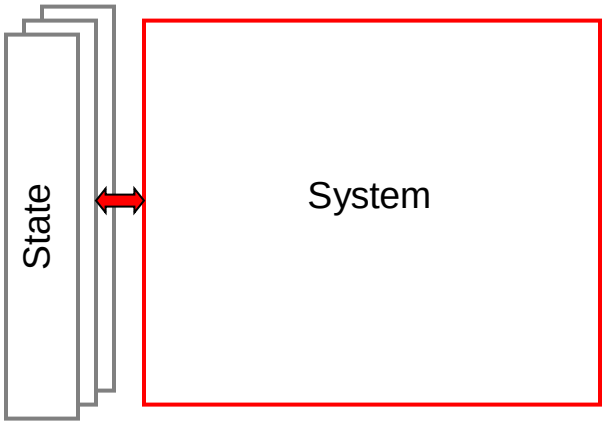


Figure 2. A System is constant once constructed; everything changeable about it is stored in separate State objects.

More specifically, a State object stores the following values:

- The time t.
- The continuous state variables $\mathbf{y}$.
- The discrete state variables $\mathbf{d}$.

It also provides spacing for storing other values that are calculated based on the state variables listed above. This will be discussed shortly.

A State object is purely a place for storing data. In contrast, a System provides much of the logic for the simulation. System functions include the following:

- It defines what information will be stored in a State (how many generalized coordinates, how many generalized speeds, etc.).
- It provides routines to calculate the force function $\mathbf{f}(\mathbf{d}; t, \mathbf{y})$, the vector of constraint functions $\mathbf{c}(\mathbf{d}; t, \mathbf{q}, \mathbf{u})$, and the vector of event trigger functions $\mathbf{e}(\mathbf{d}; t, \mathbf{y})$.
- It provides a routine which takes a state that does not satisfy the constraints, and projects it onto the constraint manifold.
- It provides routines to handle events when they occur.

This division has several important advantages. You can easily save a copy of the State at any point in the simulation, and be confident that you have not missed any other information hidden away in some other object. It also ensures that derived quantities remain synchronized with the state variables they were calculated based on, which eliminates a large class of potential bugs.

2.3 Systems and Subsystems

Now let's look a bit more closely at how a System is put together. Each System is composed of one or more *subsystems*, each represented by a Subsystem object. Most of the functions described above are actually performed by the Subsystems, not by the System itself. For example, the set of state variables for a System is simply the union of the state variables defined by all its Subsystems. The force calculated by the System is simply the sum of the forces calculated by all of its Subsystems. And so on.

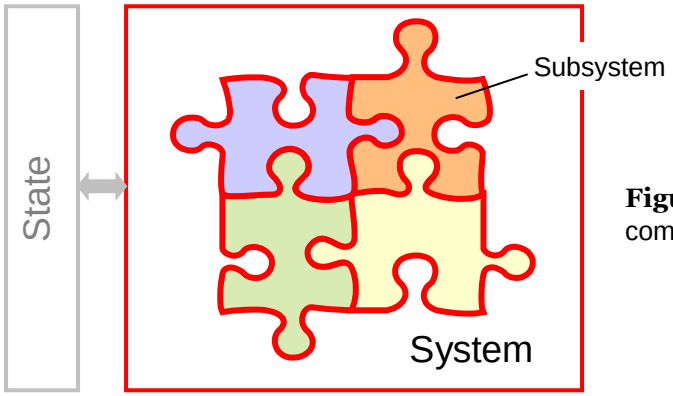


Figure 3. Systems are composed of Subsystems.

This allows you to create a System in a modular way. Subsystems can interact with each other, so you can split up your System in whatever way seems most convenient. For example, one Subsystem might define a subset of the bodies in the system, and all the forces, constraints, and events related to them. Alternatively, one Subsystem might define all the state variables, a different Subsystem define the forces acting on them, and a third Subsystem define a set of events.

Simbody provides a number of standard Subsystems ready for you to use. For example, SimbodyMatterSubsystem allows you to build up arbitrary multibody systems out of a large collection of joint types, while DuMMForceFieldSubsystem (in the Molmodel API) provides a standard force field for use in molecular dynamics simulations. You will rarely need to actually write a new Subsystem yourself.

In addition to whatever other Subsystems it has, every System has a *default Subsystem*. It implements some standard functionality that is required for all Systems. We will see some examples of what it can be used for later.

2.4 The Realization Cache

The state variables t, $\mathbf{y}$, and $\mathbf{d}$ collectively represent a complete description of the state of the system at a given time. On the other hand, there are lots of other numbers you might want to know. Some examples include

- The position of each body in Cartesian coordinates
- The force acting on each body
- The resulting acceleration of each internal coordinate
- The values of event trigger functions

These are not really independent information. Given the state variables, you can calculate them whenever you want. On the other hand, some of them may be expensive to calculate, so you want to avoid recalculating them more often than necessary. The State object therefore provides space for storing these derived values. This space is called the *realization cache*, and the process of calculating the values stored in it is known as *realizing the state*.

If you look at the list of examples above, you will see that they need to be calculated in a particular order. The Cartesian coordinates of each body generally need to be known before the forces can be calculated, and the forces need to be known before the internal coordinate accelerations can be calculated. It also is clear that not all of these pieces of information will be needed in every situation. If you only care about the positions of bodies, you don't want to waste time on an expensive force calculation.

The realization cache is therefore divided into a series of *stages*. Each piece of information in the cache belongs to a particular stage. When you want to realize part of the cache, you specify what stage to realize it up to. This causes the information belonging to that stage and all previous stages to be calculated. In other words, whenever you want to get some information from the cache, you must first make sure the state has been realized up to the stage that information belongs to. Figure 4 shows the complete list of stages, ten in all.

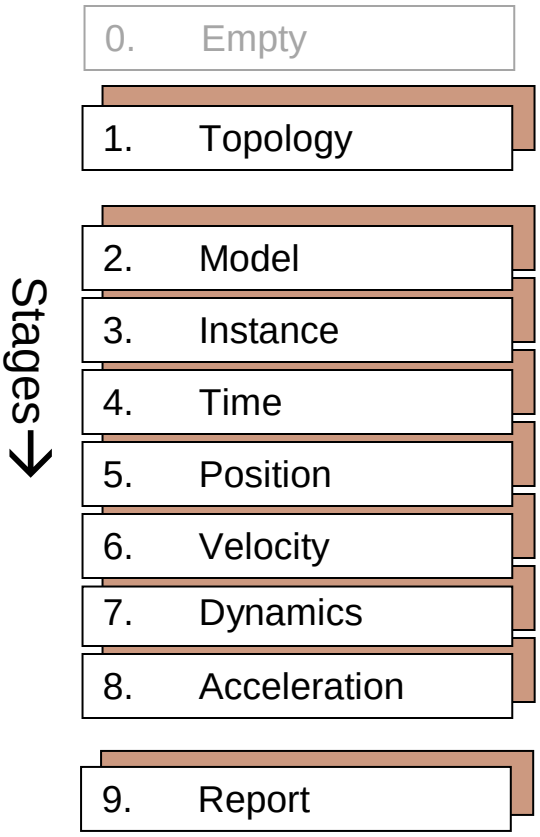


Figure 4. The organization of a computation into ordered stages. When a State is realized to a given stage, the cache information for that stage (shown in brown) is calculated and filled in. A change to a state variable at a given stage invalidates cache information at that stage and higher.

The first four stages (Empty through Instance) are involved in the initial construction and initialization of the system. Don't worry about them right now. All of the information you will want to access during a simulation is associated with one of the later stages. Here is the information associated with each of the later stages:

Time: At this stage, no derived information has yet been calculated. You can query the State for any of the state variables (*t*, **y**, and **d**), but nothing else.

Position: At this stage, the positions of all bodies in Cartesian coordinates are known.

Velocity: At this stage, the velocities of all bodies in Cartesian coordinates are known, along with the amount by which the constraints are violated.

Dynamics: At this stage, the force acting on each body is known, along with the total kinetic and potential energy of the system.

Acceleration: At this stage, the time derivatives of all continuous state variables are known, along with the values of all event trigger functions.

Report: A State is not normally realized to this stage during a simulation. It is available in case a System can calculate values that are not required for time integration, but might be needed by an event handler or for later analysis. That way, these values will only be calculated when they are actually needed.

The State makes sure that all values in the realization cache are consistent with the current state variables. If you modify any state variable, the current stage will automatically "back up" to an earlier stage, invalidating cache entries from later stages so they can no longer be accessed. In particular:

- Modifying t will bring the State back to Instance stage.
- Modifying $\mathbf{q}$ will bring the State back to Time stage.
- Modifying $\mathbf{u}$ will bring the State back to Position stage.
- Modifying $\mathbf{z}$ will bring the State back to Velocity stage.
- When a System defines a discrete state variable d, it specifies what stage the State should be reverted to when that variable is modified. This is chosen to ensure that modifying the variable will invalidate any cache entry that may depend on it.

2.5 Events

Now let's look at event handling in more detail. As described earlier, an event is signaled by an event trigger function, $\mathbf{e}(\mathbf{d}; t, \mathbf{y})$. When that function crosses through 0, an event is said to occur, and a handler function is invoked.

In practice, this means that the value of the event trigger function is calculated at each time step. When it changes sign from the previous time step, the integrator knows that an event occurred at some point during the step. It then tries to identify exactly when the event occurred by generating interpolated States at various points in the middle of the step and evaluating the trigger function at each one.

The result is a time window $(t_{low}, t_{high}]$ within which the event is known to occur. We don't know exactly when it occurred, only that it was somewhere within the window. You can control how large a window is acceptable. Asking for a smaller window will produce more accurate event localization, but also is slower.

The event handler is then invoked and given a State, which reflects what the state of the system *would* have been at t_{high} if the event had not occurred. The event handler is free to modify this State in whatever way it wants. Once it returns, the time integration continues on, using the modified State as its starting point.

An important special case is events that are defined to occur at a particular time that is known in advance. Of course, we can handle this case with the above mechanism, simply defining the event trigger function as $e(\mathbf{d};t,\mathbf{y})=e(t)=t-t_{event}$. But that is unnecessarily inefficient. We know in advance exactly when the event occurs, so there is no need to figure it out by repeated evaluations of the trigger function on a series of interpolated States! Simbody therefore provides a special purpose mechanism for events of this sort. They are known as *scheduled events*, in contrast to *triggered events* which are determined based on an event trigger function.

Another important special case is events that do not actually modify the State. For example, perhaps you want to save a copy of the current State to disk at regular intervals for later analysis. Or you might want to monitor a particular quantity and record the largest value it ever takes on over the course of the simulation. In some cases, an integrator can save work if it knows in advance that an event handler is only going to examine the State but not modify it. Simbody therefore provides a separate mechanism for implementing event handlers of this sort. They are known as *event reporters*.

Like most other aspects of a System, events are defined by Subsystems, but you don't want to have to write an entire Subsystem just to create a single event handler or event reporter. The default Subsystem therefore provides an extensible mechanism to define new events. You simply create a subclass of one of the following classes: TriggeredEventHandler, ScheduledEventHandler, TriggeredEventReporter, or ScheduledEventReporter. You then add it to the default Subsystem by calling addEventHandler() or addEventReporter() on it, and it takes care of everything else for you. We will see examples of doing this in a later chapter.

2.6 The Simbody library stack

Simbody consists of a set of modules that form a stack. Each module depends on those that come before (above) it, but not the ones after (below) it.

Module	Function
LAPACK	Provides routines for high performance linear algebra
SimTKcommon	Contains many of the basic classes for representing systems, states, vectors, matrices, event handlers, etc.
SimTKmath	Provides a variety of high level numerical tools for integration, differentiation, and optimization
Simbody	Provides algorithms and data structures for modeling multibody systems in internal coordinates
Molmodel (optional)	Provides tools for modeling biological macromolecules such as proteins and nucleic acids as Simbody internal coordinate models; can use OpenMM for acceleration.

(If we were including OpenMM here, it would be at the same level as LAPACK; both are hardware-accelerated computational tools independent of any other SimTK software. If we were including the OpenSim API, it would be at the same level as Molmodel.)

In addition to these libraries, Simbody comes with a visualization and user-interaction tool called the Simbody Visualizer. That is a separate executable program (`simbody-visualizer`) installed in the bin subdirectory of the installation. When you use Simbody's Visualizer class, an attempt is made to launch that executable. However, Simbody can be used without the Visualizer if you have your own methods for looking at the results.

For now, we will not worry too much about the boundaries between the different modules. The Simbody installation includes all the lower levels; only Molmodel is installed separately. You can access all of them from your code by including a single header file: Molmodel.h if you are doing a biomolecular simulation, Simbody.h otherwise.

3 Installing Simbody and Molmodel

Please see the `README.md` file displayed at https://github.com/simbody/simbody for up-to-date instructions for Simbody.

Most users will not need Molmodel. If you are interested in it, please post to the user forum at https://simtk.org/home/molmodel or https://simtk.org/home/rnatoolbox.

4 Simple Example: A Double Pendulum

It's now time to look at our first example. Here we'll introduce features as we go. In the next chapter we'll step back and talk more about the Simbody API in general.

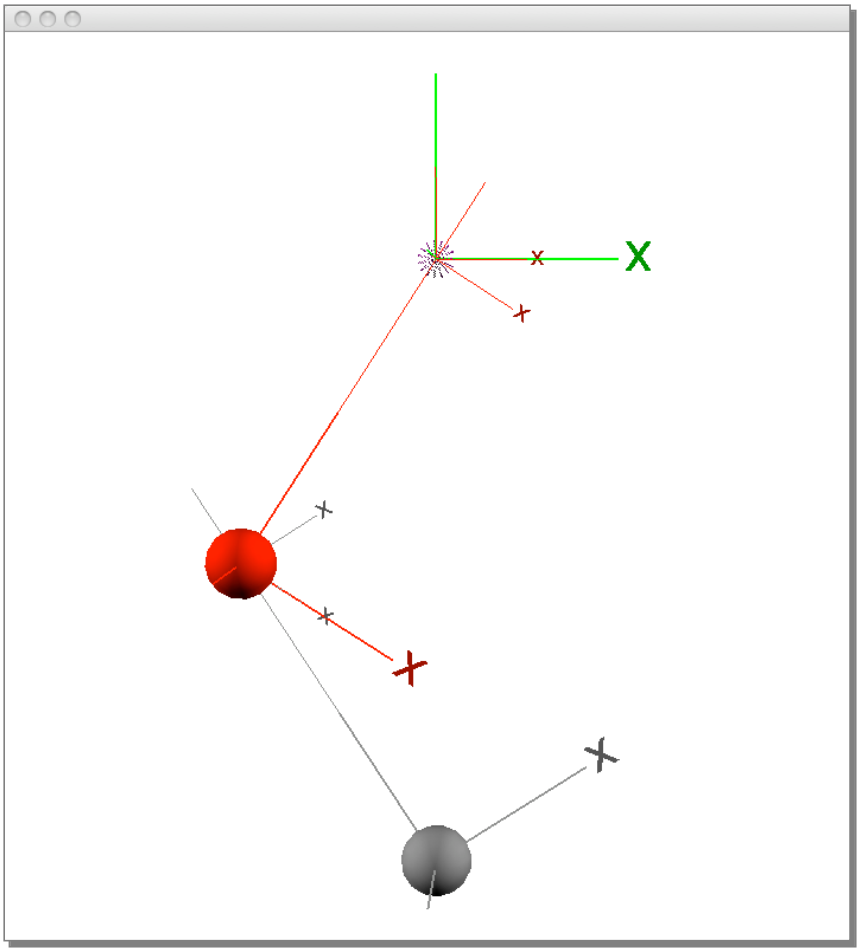
4.1 A First Example

The following program creates a system representing a *double pendulum*: one pendulum attached to the end of a second one. It simulates the behavior of this system over a period of 50 seconds, and displays a movie of it.

```
#include "Simbody.h"
using namespace SimTK;
int main() {
  try {

    // Create the system.
    MultibodySystem system;
    SimbodyMatterSubsystem matter(system);
    GeneralForceSubsystem forces(system);
    Force::UniformGravity gravity(forces, matter, Vec3(0, -9.8, 0));
    Body::Rigid pendulumBody(MassProperties(1.0, Vec3(0), Inertia(1)));
    pendulumBody.addDecoration(Transform(), DecorativeSphere(0.1));
    MobilizedBody::Pin pendulum1(matter.Ground(), Transform(Vec3(0)),
            pendulumBody, Transform(Vec3(0, 1, 0)));
    MobilizedBody::Pin pendulum2(pendulum1, Transform(Vec3(0)),
            pendulumBody, Transform(Vec3(0, 1, 0)));

    // Set up visualization.
    Visualizer viz(system);
    system.addEventReporter(new Visualizer::Reporter(viz, 0.01));

    // Initialize the system and state.
    system.realizeTopology();
    State state = system.getDefaultState();
    pendulum2.setRate(state, 5.0);

    // Simulate it.
    RungeKuttaMersonIntegrator integ(system);
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(50.0);

  } catch (const std::exception& e) {
    std::cout << "Error: " << e.what() << std::endl;
    return 1;
  }
}
```

Before you can compile and run this program, you need to have Simbody installed (see Chapter 3). The installation directory has subdirectories “include”, “lib”, and “bin”. Make sure the “include” directory is part of your compiler’s include path, and the “lib” directory is available to the linker. At runtime the shared library directory (lib for Mac and Linux, bin for Windows) must be on the appropriate “path” environment variable. Exactly how you do this will depend on the compiler and operating system you are using.

If everything is working correctly, you should see a window that looks something like this, showing an animation of the pendulum swinging:



Let's go through the program line by line and see how it works. It begins with an include statement:

```
#include "Simbody.h"
```

That gets us all the declarations we need to write a Simbody-using program.

Next we import the SimTK namespace, which includes nearly all of the symbols used by SimTK:

```
using namespace SimTK;
```

That's optional, but it saves having to preface every symbol with "SimTK::".

The `try`/`catch` block should always be included because most Simbody error conditions are returned by throwing exceptions. We don't expect any errors here but this is a good habit to get into and can save a lot of trouble.

Now we create our System and a pair of Subsystems:

```
MultibodySystem system;
SimbodyMatterSubsystem matter(system);
GeneralForceSubsystem forces(system);
```

MultibodySystem is a subclass of System. It defines the functionality for working with general multibody systems. In most cases, you will use a MultibodySystem or one of its subclasses in your simulations.

SimbodyMatterSubsystem is the Subsystem that will define all the bodies in the system. It is a modular Subsystem, to which you can add whatever bodies you want. A MultibodySystem must always have a SimbodyMatterSubsystem. Notice that we do not explicitly add the Subsystem to the System. Instead, the constructor takes a reference to the System, and it adds itself.

GeneralForceSubsystem is a Subsystem that can be used to add a variety of forces to a system. Much like SimbodyMatterSubsystem, it is designed to be modular; you can add whatever forces you want to it. In the next line, we add a uniform gravitational force of 9.8 m/s^2 in the negative y direction:

```
Force::UniformGravity gravity(forces, matter, Vec3(0, -9.8, 0));
```

The Force class has many subclasses representing a variety of common forces: springs, dampers, constant forces, etc. It also has a subclass called Force::Custom, which you can use to define completely new forces.

To understand the next few lines, we need to consider two important classes: Body and MobilizedBody. The Body class represents the physical properties of a body, such as its mass and moment of inertia. Body::Rigid is a subclass that represents a generic rigid body. The MobilizedBody class combines the body's physical properties (represented by a Body object) with a set of *mobilities*—that is, the set of state variables describing how the body is allowed to move. It has many different subclasses defining a wide variety of types of joints.

We begin by creating a Body to describe the physical properties of our pendulum:

```
Body::Rigid pendulumBody(MassProperties(1.0, Vec3(0), Inertia(1)));
```

We specify that it has a mass of 1 kg (the first argument), the center of mass is at the body's origin (the second argument), and a moment of inertia of 1 kg·m² around all three axes (the third argument).

A Body object can also define how the body should be drawn in graphical displays. This has no effect on the simulation, so it is completely optional. Since we want to show an animation of the pendulum, we add a *decoration* to the Body: a sphere of radius 0.1.

```
pendulumBody.addDecoration(Transform(), DecorativeSphere(0.1));
```

So far we have not actually added any bodies to our System. We have simply created a Body instance that defines a certain set of physical and display properties. The next two lines actually add bodies to the System:

```
MobilizedBody::Pin pendulum1(matter.Ground(), Transform(Vec3(0)),
            pendulumBody, Transform(Vec3(0, 1, 0)));
MobilizedBody::Pin pendulum2(pendulum1, Transform(Vec3(0)),
            pendulumBody, Transform(Vec3(0, 1, 0)));
```

The MobilizedBody::Pin constructor adds a new body to the System by creating a pin joint connecting it to a body that is already in the System. Simbody uses the term *mobilizer* for any joint that connects a body to its parent in a multibody tree, so a MobilizedBody always contains a Body as described above, and its mobilizer, in this case a Pin. A Pin mobilizer has one generalized coordinate and one generalized speed, which allow it to rotate around a single axis. Our pendulum consists of two of these linked to each other.

The first constructor argument is the MobilizedBody's parent; that is, the already-in-System MobilizedBody relative to which its position is defined. Every SimbodyMatterSubsystem has a *Ground* body which is fixed at the origin, and forms the root of the multibody tree. We specify Ground as the parent for pendulum1, and then specify pendulum1 as the parent for pendulum2. So the parent body is always closer to ground than is the child body we're adding, and every body in the System has a unique parent body.

The third argument is a Body object. Notice that we specify the same Body for both MobilizedBodies. Remember, a Body is simply a description of a set of physical properties. If several MobilizedBodies have identical properties, it is fine to use the same Body for all of them but they aren't shared; each MobilizedBody gets its own copy of the Body.

Now let's look at the second and fourth arguments, which are both Transform objects. These are used to specify the location and orientation of a coordinate frame fixed on each of the

bodies. The mobilizer we're using will connect those two frames, permitting some motion between them that depends on the kind of joint. Let's look at this more precisely so we can say exactly how the Transforms define the pin joint mobilizer.

When defining a MobilizedBody, think of the parent as though it were temporarily fixed to Ground and the new body as though it does all the moving. With that in mind, we'll call the frame on the parent the "fixed" frame F, and the frame on the child the "moving" frame M. We'll call the parent's own body frame P, and the new (child) body's frame B. So the second argument above locates the F frame with respect to the parent body's frame P, and the fourth locates M with respect to the child's body frame B. You can see these frames in Figure 5, with F in blue and M in red.

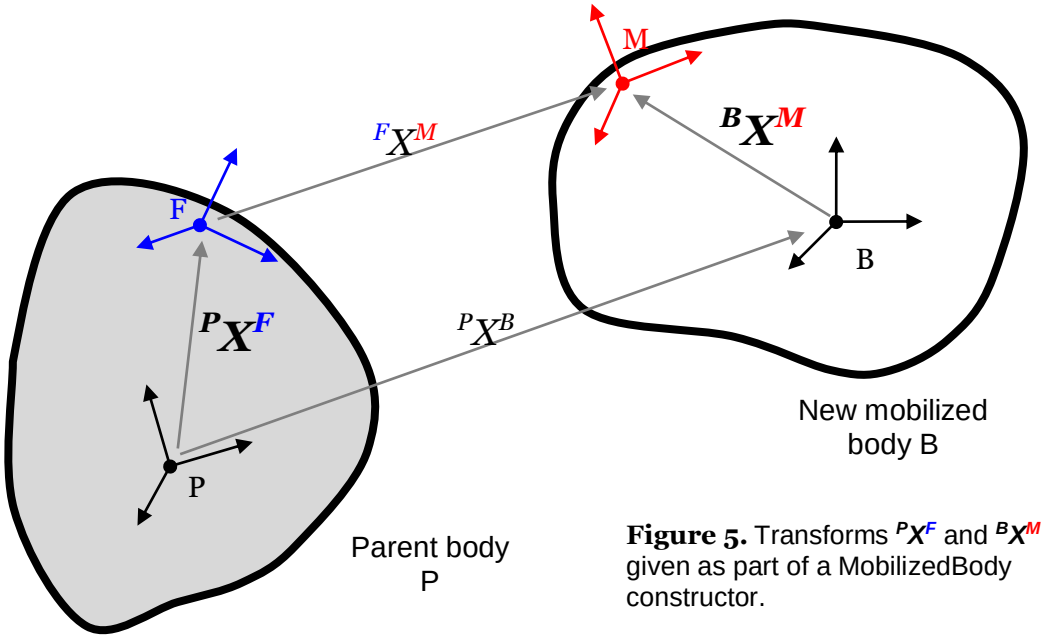


Figure 5. Transforms $^{P}X^{F}$ and $^{B}X^{M}$ are given as part of a MobilizedBody constructor.

Now we can describe precisely what the Pin mobilizer does: it permits body B to move only in a manner that keeps the F and M frame origins in the same place, and the **z** axes of those frames lined up. The two Transforms tell us where F and M are positioned on their bodies. In our case, we specified a vector of length 0 for both the fixed frames, so we use the parent bodies' origins as the fixed point. For both the moving frames, we specified a vector of length 1 pointing in the y (up) direction, so when both joints are in their neutral positions (that is, when **q**=[0, 0]), the pendulum hangs straight down.

By convention, we denote a transform with the letter X (not T as you might expect). Then we use two superscripts to denote the "reference" and "target" frames, with the reference frame on the left: ${}^{R}X^{T}$. A transform tells us how the target frame is positioned with respect to the reference frame. Figure 5 shows the four transforms involved in a MobilizedBody. The "fixed" transform ${}^{P}X^{F}$ and the "moving" transform ${}^{B}X^{M}$ are given in the constructor and don't change with time. The *mobilizer transform* ${}^{F}X^{M}$ depends on this joint's generalized coordinates and tells us where the M frame is currently in relation to the F frame. These are combined to create the *body transform* ${}^{P}X^{B}$ which relates the position and orientation of the body to that of its parent:

$$ {}^{P}X^{B} = {}^{P}X^{F} \cdot {}^{F}X^{M} \cdot {}^{M}X^{B}, \qquad \text{where } {}^{M}X^{B} = ({}^{B}X^{M})^{-1}. $$

Note that with this notation composition of transforms always requires that consecutive superscripts match; if they don't, you are doing something wrong!

Our System is almost complete at this point, but there's one more thing we want to add, a way to visualize the simulation:

```
Visualizer viz(system);
system.addEventReporter(new Visualizer::Reporter(viz, 0.01));
```

The first line creates a Visualizer for our System. The second adds an event reporter that will invoke the Visualizer at regular intervals during the simulation. You will recall from the previous section that an event reporter is a special type of event handler that does not actually modify the state of the system when it is called. It is there just to observe and report. `Visualizer::Reporter` uses a Visualizer to display a movie of the simulation. We ask it to draw a new frame every 0.01 seconds. That's measured in simulation time, not real time. Depending on how demanding your simulation, how fast your computer is, and whether you are running in debug or optimized mode, one second of simulation time may take more or less than one second to calculate and display. The Visualizer will take care of controlling the real time rate at which the frames are displayed; by default it displays 30 frames per second. In this case our simulation will display in slow motion since 30 frames at 0.01 seconds per is 0.3 seconds of simulated time in a second of real time. A Simbody Visualizer has many options for controlling what gets displayed and when, and can also be used to interact with the user during the simulation.

You can find out a lot more about any of these classes by looking in the API reference documentation, posted on the Simbody project's Documentation page and also available in

the doc subdirectory of your Simbody installation (SimbodyAPI.html). Type the name of a class into the search box or browse the class list.

We're all done building our System. Now we need to prepare it for simulation:

```
system.realizeTopology();
```

This function informs Simbody that you are done constructing the System, and performs lots of initialization. Don't worry about the details right now; they usually only matter if you are writing a new Subsystem class. For the moment, you just need to remember that after you build a System, you need to call realizeTopology() on it. If you then make any changes to the System (such as adding another event handler), you need to call realizeTopology() again, and any State objects you previously created will no longer be valid.

Now we need to get a State for the System. Every System provides a *default State* which has been initialized to have the right set of state variables and cache entries for the System. The default State is created when realizeTopology() is called. The easiest way to create a new State is simply to make a copy of the default State. That is what we do:

```
State state = system.getDefaultState();
```

The default State has all state variables initialized to 0. We could use that as the starting point for our simulation, but it would make for a very dull simulation. The pendulum would simply hang there and not do anything. We need to give it some energy to make it move. We do this by modifying the State to give pendulum2 an initial angular velocity of 5 radians/sec:

```
pendulum2.setRate(state, 5.0);
```

The setRate() method is specific to the Pin mobilizer, for which "the rate" is unambiguous since there is only one degree of freedom and it is angular. (Recall that pendulum2 is a MobilizedBody::Pin.) We now have a System to simulate, and an initial State to begin the simulation from. It's time to do some simulating!

```
RungeKuttaMersonIntegrator integ(system);
TimeStepper ts(system, integ);
ts.initialize(state);
```

To understand these lines, we need to discuss two important classes: Integrator and TimeStepper. An Integrator is an object that knows how to advance the continuous state variables **y** by integrating the equations of motion. There are, of course, lots of different

algorithms for doing that. The best algorithm in a particular case depends on lots of factors: how smooth the system's energy landscape is, what level of accuracy you want, whether the equations are numerically stiff, etc. SimTK therefore provides a choice of different Integrator subclasses, each implementing a different algorithm. In this case, we have chosen a RungeKuttaMersonIntegrator, which is a good choice for most mechanical systems. We could also, for example, have used VerletIntegrator instead to use the velocity Verlet algorithm; you would use it identically to the RungeKuttaMersonIntegrator shown here.

A TimeStepper takes care of the discrete part of a simulation. It repeatedly invokes the Integrator to evolve the equations of motion, calls event handlers when events occur, and notifies the Integrator of any changes made by the event handlers. It takes care of most of the details of running a simulation for you. All you need to do is tell it what Integrator to use, and it does the rest.

Now we are all ready to run the simulation. Here is how we do it:

```
ts.stepTo(50.0);
```

That's it! Just tell the TimeStepper what time to advance the simulation to, and it does it.

Before we finish with this example, there is one point worth remarking on. Throughout the discussion, we have assumed that all quantities were measured in SI units: seconds for time, kg for mass, etc. You may have wondered how Simbody knew that. The answer is simple: it didn't. You are free to use whatever units you want. All that matters is that you use a single, consistent set of units for all quantities. For this example, we assumed SI units. In a later chapter, we will see an example that uses a different set of units. The only absolute requirement is that Newton's 2nd law force=mass*acceleration must hold in whatever unit system you choose.

4.2 A Scheduled Event Reporter

Now let's expand on the previous example. We are going to create a new event reporter. Its job will be to print out the position of the end of the pendulum at regular intervals during the simulation. Since the reporting times are fixed in advance, not determined by the behavior of the system, this is a *scheduled event reporter*.

Here is the code which implements the event reporter:

```
class PositionReporter : public PeriodicEventReporter {
```

```
public:
    PositionReporter(const MultibodySystem& system, const MobilizedBody& mobod,
                     Real reportInterval)
    :   PeriodicEventReporter(reportInterval), system(system), mobod(mobod) {}
    void handleEvent(const State& state) const {
        system.realize(state, Stage::Position);
        Vec3 pos = mobod.getBodyOriginLocation(state);
        std::cout<<state.getTime()<<"\t"<<pos[0]<<"\t"<<pos[1]<<std::endl;
    }
private:
    const MultibodySystem& system;
    const MobilizedBody&   mobod;
};
```

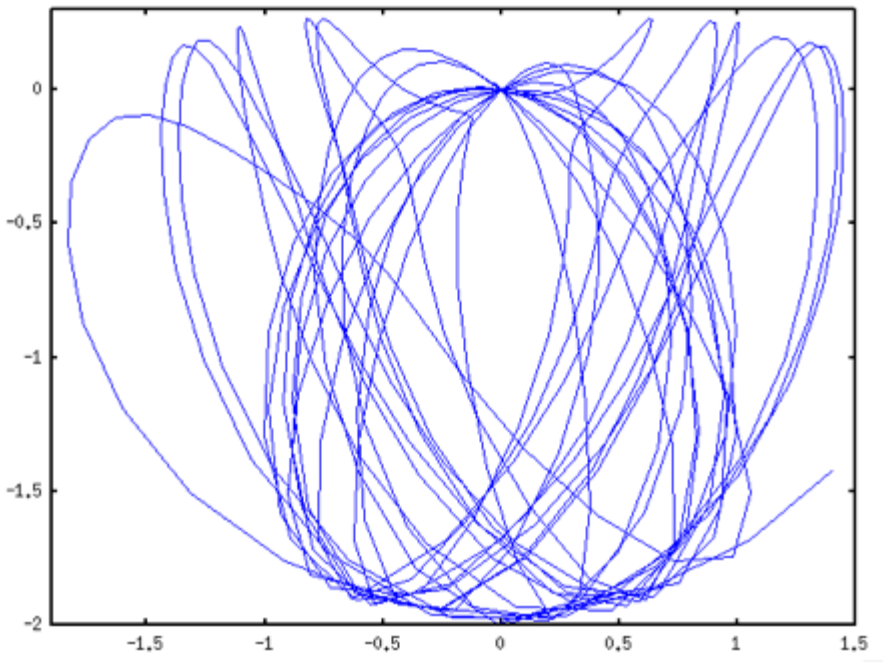
We can add this event reporter to our System with the following line:

```
system.addEventReporter(new PositionReporter(system, pendulum2, 0.1));
```

Now when you run the program, it prints out the current time and the X and Y coordinates of the pendulum every 0.1 seconds. Here are the first few lines of the output:

```
0       0           -2
0.1     0.481219    -1.88155
0.2     0.870604    -1.58657
0.3     1.13856     -1.22146
```

We output the values as tab delimited text, which makes it easy to load into other programs for analysis. For example, here is an XY plot which traces out the pendulum's trajectory over the course of the simulation:



Let's go through the code and see exactly what it is doing. First, we declare our event reporter to be a subclass of PeriodicEventReporter:

```
class PositionReporter : public PeriodicEventReporter {
```

PeriodicEventReporter is a subclass of ScheduledEventReporter which is used for the (very common) case of a reporter that should be called at fixed intervals throughout the simulation. We could have subclassed ScheduledEventReporter directly instead; we just would have needed to implement one additional method.

Now look at the constructor:

```
PositionReporter(const MultibodySystem& system, const MobilizedBody& mobod,
                 Real reportInterval)
:   PeriodicEventReporter(reportInterval), system(system), mobod(mobod) {}
```

This is all very simple. We just store references to the System and MobilizedBody we will be reporting on, and pass the reporting interval along to the superclass.

Notice that the reporting interval is of type "Real." This is the type used by SimTK for nearly all floating point values. The Simbody release always defines Real as double.

All the really interesting things happen in the handleEvent() method. It is called every time an event occurs (that is, at the time intervals specified to the constructor), and receives a reference to the current State.

We want to print out the position of the pendulum in Cartesian coordinates. That information is only available if the State has been realized to at least Position stage, so the very first thing we do is ask the System to realize it:

```
system.realize(state, Stage::Position);
```

Now we can ask for the location of the MobilizedBody:

```
Vec3 pos = mobod.getBodyOriginLocation(state);
```

Notice that we do not ask the State for this information directly. Instead, we pass the State to the MobilizedBody and ask it for the location. This is a common pattern. All the State object knows about is state variables. It does not understand that those variables represent the rotation angles of a pendulum, and that the pendulum is made up of bodies with positions in Cartesian space. It is the System that knows these things, so you need to get the System (or in this case, the MobilizedBody which is a part of the System) to interpret the State and extract the desired information.

Finally, we print out the desired information:

```
std::cout<<state.getTime()<<"\t"<<pos[0]<<"\t"<<pos[1]<<std::endl;
```

We do not bother to print the z component of the position because we have constructed the pendulum to move entirely in the xy plane.

4.3 A Triggered Event Reporter

Suppose we are especially interested in knowing how high up the pendulum gets on each swing. We can't rely on a scheduled event reporter to tell us this: the highest point on the swing probably won't exactly correspond to a reporting time, so we will often miss it. We really want to be guaranteed that a report will occur at *exactly* the moment when the pendulum's height reaches its local maximum. This calls for a *triggered event reporter*. The event times are determined by the behavior of the System, not scheduled in advance.

The first thing we need to do is choose an event trigger function. This must be a continuous function that crosses through zero at the moment when an event should occur. Fortunately, there is an easy choice we can use: the y component of the body's velocity. This has exactly the properties we need. There is a catch, though. We only care about the highest point on the

swing, not the lowest point. The velocity becomes zero at local minima as well as maxima, but we don't want the event reporter to be called at those points.

One option would be to have handleEvent() filter out the unwanted events. Each time it was called, it could calculate the derivative of the event function (in this case, the body's acceleration), and use that to decide whether to ignore the event. But there's a better solution: Simbody can do this filtering for you. You simply tell it which zero crossings you are interested in: rising transitions, falling transitions, or both.

Here is the code for the event reporter.

```
class PositionReporter : public TriggeredEventReporter {
public:
    PositionReporter(const MultibodySystem& system, const MobilizedBody& mobod)
    :   TriggeredEventReporter(Stage::Velocity), system(system), mobod(mobod) {
        getTriggerInfo().setTriggerOnRisingSignTransition(false);
    }
    Real getValue(const State& state) const {
        Vec3 vel = mobod.getBodyOriginVelocity(state);
        return vel[1]; // y coordinate
    }
    void handleEvent(const State& state) const {
        system.realize(state, Stage::Position);
        Vec3 pos = mobod.getBodyOriginLocation(state);
        std::cout<<state.getTime()<<"\t"<<pos[0]<<"\t"<<pos[1]<<std::endl;
    }
private:
    const MultibodySystem& system;
    const MobilizedBody&   mobod;
};
```

This time, we subclass TriggeredEventReporter, which should come as no surprise. Take a look at the constructor:

```
PositionReporter(const MultibodySystem& system, const MobilizedBody& mobod)
:   TriggeredEventReporter(Stage::Velocity), system(system), mobod(mobod) {
```

Notice the value we pass to the superclass constructor: Stage::Velocity. Event trigger functions are calculated as part of the process of realizing a State. Each event handler must specify what stage its trigger function should be calculated after. Since our function depends on information that is available at Velocity stage, that is the stage we specify.

The constructor also contains the following line:

```
getTriggerInfo().setTriggerOnRisingSignTransition(false);
```

This tells Simbody that events should not occur on rising sign transitions (when the trigger function increases from negative to positive). The default is to trigger on any sign transition; we only care about falling transitions.

Now look at the getValue() method, which returns the value of the event trigger function:

```
Real getValue(const State& state) const {
    Vec3 vel = mobod.getBodyOriginVelocity(state);
    return vel[1]; // y coordinate
}
```

Although this looks straightforward, you might notice something odd about it. We access the body's velocity, which is only available once the State has been realized to Velocity stage. Shouldn't we therefore call system.realize()?

The answer is no. Remember, this function is called as *part of* realizing the State. We are already in the middle of a call to realize(), and all of the Velocity stage cache entries have already been calculated. In fact, if we inserted a call to realize() here asking to realize the state to Velocity stage, the result would be an infinite recursion!

The question is a good one, though, and this is an unusual case. At any time *other than* in the middle of realizing the State, there is no harm in sticking in an extra call to realize(). If the State has already been realized to the specified stage, it will simply return without doing anything.

Conversely, if you forget to realize the State and then ask for information that is not yet available, it will simply throw an exception. This will usually cause the simulation to terminate, but at least you discover your mistake right away. This scheme ensures that expensive calculations are done only by explicit request rather than as hidden side effects that may have been unintended.

4.4 An Event Handler

So far, we have looked at event reporters: event handlers that can only observe the State, not modify it. Now let's create an event handler that actually modifies the State. Suppose that half way through the simulation, we want to briefly apply a brake to the pendulum that decreases its angular velocity. Here is the code for an event handler that does this.

```
class VelocityReducer : public ScheduledEventHandler {
```

```
public:
    VelocityReducer() {
    }
    Real getNextEventTime(const State&, bool includeCurrentTime) const {
        return 25.0;
    }
    void handleEvent(State& state, Real accuracy, bool& shouldTerminate) const {
        state.updU() *= 0.2;
    }
};
```

We add it to the System with the following line:

```
system.addEventHandler(new VelocityReducer());
```

We subclass ScheduledEventHandler. As you would expect, there are also classes called PeriodicEventHandler and TriggeredEventHandler. All of these classes are nearly identical to the corresponding event reporter classes. The only difference is the signature of handleEvent().

ScheduledEventHandler requires us to provide a method that returns the time of the next event:

```
Real getNextEventTime(const State&, bool includeCurrentTime) const {
    return 25.0;
}
```

Since we only want the event handler to be called once, we just return a hardcoded value. In general, though, the event time could be calculated based on the current State. For example, PeriodicEventHandler looks at the current time and uses that to determine the next event time.

Now look at the signature of handleEvent():

```
void handleEvent(State& state, Real accuracy, bool& shouldTerminate) const {
```

That's a little more complicated than for an event reporter. The first thing to notice is that the State is no longer const. As promised, we are now permitted to modify it.

The next argument conveys the current integration accuracy requirement to the event handler. The event handler is supposed to reflect some physical process. Any calculations it does are an intrinsic part of the simulation, and the State it produces is part of the simulated trajectory. The user may have definite requirements for the accuracy of the trajectory, and

the event handler is expected to honor them if it performs any approximate computations. You can ignore this argument in the common case that your event handler does a (numerically) exact computation as we're going to do here. If the resulting State is calculated in an approximate way, the error in it should be less than this value. If the System includes any constraints, the error in the constraint functions should be less than it.

A State may also provide weights for the individual state variables and constraint functions. Advanced event handlers should respect those weightings; we won't cover that here but you can look in the Doxygen documentation for the State API. See for example the getUWeights() method.

Finally, there is an argument which is used for output: shouldTerminate. If you want the time stepper to stop the simulation as a result of this event, set that argument to true. It will have been set to false before the call so you don't need to set it to anything if you just want the simulation to continue after the event has been handled.

The actual implementation of this method is quite simple:

```
state.updU() *= 0.2;
```

updU() returns a mutable reference to the State's **u** vector, which we multiply by 0.2. There is also a method called getU(), which returns a const reference to it. This naming convention is used frequently throughout SimTK: methods that return a const reference begin with "get", and methods that return a non-const reference begin with "upd".

4.5 Constraints

Working in internal coordinates greatly reduces the number of constraints needed for most systems, but it often cannot eliminate them altogether. Fortunately, Simbody makes it very easy to add constraints to your System. (It is worth emphasizing here that mobilizers—internal coordinate joints—are *not* constraints in Simbody.)

Each constraint is represented by an object of the Constraint class. Subclasses of Constraint represent different kinds of constraints. Simbody provides a collection of Constraint subclasses for various common types of constraints: that the distance between two bodies must remain fixed, that a body can only move in a particular plane, etc.

As an example, let's add a constraint forcing the end of our pendulum to remain on the vertical line through the origin. That is, the body in the middle of the pendulum will be free

to swing back and forth, but the body at the end will only be able to move up and down. Adding a Constraint works exactly like adding a MobilizedBody:

```
Constraint::PointOnLine(matter.Ground(), UnitVec3(0, 1, 0), Vec3(0), pendulum2,
    Vec3(0));
```

Constraint::PointOnLine is a subclass of Constraint that implements constraints of this sort: a point on one body is only allowed to move along a line defined by a different body. The first three arguments specify the line: it is defined in the Ground body's reference frame, it points in the y-axis direction (0, 1, 0), and it passes through the origin. The last two arguments specify the point which must remain on the line: it is at the origin of pendulum2's reference frame.

With this single additional line, the behavior of the pendulum changes dramatically. Instead of moving chaotically, the first mass simply swings back and forth like an ordinary pendulum, while the second one slides up and down in unison with it.

5 Some Simbody Basics

Now that you've seen an example of using Simbody, we'll step back and talk in more detail about some of the basic conventions and low-level features of the Simbody API. Then in the next chapter we'll return to a more complicated example.

Simbody uses basic numerical classes from the Simmatrix package, which provides high-performance, Matlab-like functionality accessible from our C++ API. We will only discuss the basics here; much more information is available in the API Reference documentation. There is also a separate document describing the design and implementation of Simmatrix. See the Simbody home page https://simtk.org/home/simbody, Documents page.

5.1 Naming Conventions

All symbol names generated by Simbody are in the `SimTK` namespace. In most cases you will want to include the following line at the top of your source files so that you do not have to repeat the namespace for every SimTK symbol:

```
using namespace SimTK;
```

Alternatively, you can introduce symbols selectively with statements like

```
using SimTK::Vec3;
```

Or you can prefix the symbols with `SimTK::` when you use them. There are a few symbols, typically pre-processor macros, that cannot be put in a C++ namespace; those symbols begin with the six-character string "`SimTK_`" instead.

Simbody uses mixedUpperAndLowerCase names with a capital letter used to begin a new word. Class names and constants begin with a CapitalLetter. Method (function) names and variable names begin with a lowerCaseLetter. Pre-processor macros, except for the initial `SimTK` prefix, are written in `ALL_CAPS` with underscores separating words.

5.2 Numbers and Constants in SimTK

Simbody supports both float (single) and double precision, and is compiled with one of those as its default, which is then referred to as type `Real`. So `SimTK::Real` is simply a `typedef` to either the built-in `float` or `double` type. The default is double precision and we

recommend you leave it that way, so for our purposes here the `SimTK::Real` type is always defined

```
typedef double Real;
```

Similarly, Simbody has a type `SimTK::Complex` which is interchangeable here with the C++ built-in type `std::complex<double>`. You probably will not need to use complex numbers often with Simbody, but they are available if you need them and supported for all vector and matrix operations.

Simbody pre-defines a number of constants to machine precision; we recommend you use those rather than defining your own. The most useful ones are: `Pi`, `E` (e), `Infinity`, `NaN` (not-a-number), and `I` ($i=\sqrt{-1}$). Note that the first letter of constants are capitalized and in the `SimTK` namespace – these are not preprocessor macros; they have types and addresses.

5.3 Vectors and Matrices

Simbody has two different sets of classes for vector and matrix objects, to deal with small objects where overhead must be minimized, and with large objects where computation strategy and memory access must be optimized. You have seen types `Vec3` (a 3-vector) and `Vector` (a variable-length vector) in the previous example. Here we will discuss those in more detail. If you want to know more about the design goals and implementation of these classes, see the Simmatrix document here: https://simtk.org/home/simbody, Documents tab.

First, there are classes to represent small, fixed size vectors and matrices with zero runtime overhead: `Vec` for column vectors, and `Mat` for matrices[1]. These classes are templatized based on size and element type. Synonyms (`typedefs`) are defined for common combinations; for example, `Vec3` is a synonym for `Vec<3,Real>`, while `Mat22` is a synonym for `Mat<2,2,Real>`. You can also create other combinations, such as `Mat<2,10,Real>` or `Vec<4,std::complex<float>>`. You can also nest these objects – for example, the type `Vec<2,Vec3>`, called a *spatial vector*, is useful for combining rotational and translational quantities into a single object representing a spatial velocity or

[1] There is also a `Row` type that does not normally appear in user programs.

spatial force. However, the size must always be determinable at compile time. The in-memory representation of these small objects is minimal: only the data elements are stored.

Second, there are classes to represent large vectors and matrices whose sizes are determined at runtime: `Vector_` for column vectors and `Matrix_` for matrices[2]. These classes are templatized based on element type. Types `Vector` and `Matrix` are synonyms for `Vector_<Real>` and `Matrix_<Real>`. Again, you can use other element types. In fact, the element type can even be one of the fixed-size vector or matrix objects. For example, `Vector_<Vec3>` is a vector, where each element is itself a three-component vector. However, it is not permissible to use the variable-size `Vector_` or `Matrix_` objects as element types. The in-memory representation of these objects includes, in addition to the data, an opaque descriptor containing the length and information on how the data is laid out; the declared objects actually consist only of a pointer (essentially a void*) to the descriptors. This has many advantages for implementation and binary compatibility, but makes it difficult to look through these objects in a debugger as you can with the small `Vec` and `Mat` classes.

Here are some sample declarations:

```
Vec<3>           v;   // a 3-vector of Reals
Vec3             w;   // same thing, using abbr.
Vector           b,x; // vectors of doubles
Matrix           M;   // mxn matrix of doubles
Matrix_<Complex> C;   // mxn matrix of complex<double>
Vector_<Vec3>    v3;  // big vector of 3-vecs

// This type is a 2-element vector whose elements
// are 3-vectors. Memory layout and computational
// efficiency are identical to Vec<6>.
typedef Vec<2,Vec3> SpatialVec;
```

5.3.1 Operators

All of these classes support standard mathematical operators like +, -, *, / and C-style assignment operators like +=, -=, *=, /=. In addition, Simbody overloads the ~ unary operator to indicate transpose (or more precisely, Hermitian conjugate). That is, for any vector or matrix `x`, Simbody's "`~x`" has the same meaning as Matlab's "`x'`". For `Vec3` there is also a cross product operator % available so that you can write compact expressions like

[2] Again with a normally-hidden `RowVector_` type.

```
Vec3 w, r;              // defined somewhere
Vec3 a = w % (w % r);   // a=w x (w x r)
```

Like Matlab, Simbody requires strict shape conformance for vector and matrix arguments. So for `Vec3 v` and `w`, `~v*w` is a dot product (scalar=row*vector) while `v*~w` is an outer product (matrix=vector*row), while `v*w` fails to compile because the arguments are not conforming. Scalar multiplication acts as expected. Global functions `dot()`, `cross()`, and `outer()` are available for those who prefer them to using the operators.

There also are versions of many standard math functions that operate on vectors and matrices: `sin()`, `exp()`, `sqrt()`, etc. and additional functions `abs()`, `min()`, `max()`, `sort()`, `mean()`, `median()`. These allow many calculations to be written in a very concise way.

5.3.2 Construction and assignment

All vector and matrix types define a default constructor, that is, a constructor with no arguments. In Debug mode, the default constructor initializes all elements to `NaN`. In Release mode, all elements are left uninitialized.

Constructors are also available for initializing data elements from individual element values, or by copying compatible objects. Initialization values can be provided in the constructor or via a pointer (or C array) to values of the appropriate type. Assignment operators are available for copying whole objects, and for setting single elements, subvectors and submatrices.

One convention followed by Simbody, which is different from that of most similar systems, is the treatment of scalar assignment. We follow this convention: (1) when a scalar *s* is assigned to a vector, every element of the vector is set to *s* (this is the typical convention), and (2) when a scalar *s* is assigned to a matrix, the *diagonal* elements of the matrix are set to *s* while the off-diagonals are set to zero. Examples:

```
Vec3   v;
Mat22  m;

Vec3 v(0);  // v=(0,0,0)
v=0;        //     "
v=3;        // v=(3,3,3)
Mat33 m(0); // m=( 0,0 )  zero
m=0         //    ( 0,0 )
```

```
Mat33 m(1); // m=( 1,0 ) identity
m=1;        //   ( 0,1 )

Vector b(10);     // initial size 10 doubles
Matrix M(20,10);  // initial size 20x10
b=0; // b=10 zeroes
M=1; // M=0, except M(i,i)=1, 0<=i<10
```

This convention is especially apt for matrices, because the matrix resulting from such a scalar assignment "acts like" that scalar. That is, if you multiply by this matrix the result is identical to a scalar multiply by the original scalar. Two important cases enabled by this convention are: (1) setting a matrix to the scalar "1" results in the multiplicative identity matrix of that shape, and (2) setting a matrix to the scalar "0" results in the additive identity matrix of that shape. In general, in any operation involving a scalar *s* and a `Matrix` or `Mat`, the scalar is treated as if it were a conforming matrix whose main diagonal consists of all *s*'s with all other elements zero. So `Matrix` *m* += *s* will result in *s* being added to *m*'s diagonal, which is what would happen if *s* were replaced by `diag`(*s*) of the same dimension as *m*. *m*`-=1` thus subtracts an identity matrix from *m*, *without* touching any of the off-diagonal elements. Note that for multiply and divide this convention yields the ordinary scalar multiply and divide operations: *m***s* (=*m**`diag`(*s*)) multiplies every element of *m* by *s*, while *m*/*s* (=*m** (1/*s*)) divides every element of *m* by *s*.

5.3.3 Indexing

Simbody provides 0-based indexing using the `[]` operator. If a `Matrix` is modifiable (non-const), then the indexed element can be modified and that change affects the contents of the object. The `[]` operator applied to a `Matrix` returns a row (specifically, a `RowVector` object), which may in turn be indexed to obtain an element in C style. Simbody also permits indexing using round brackets `()` yielding identical results to `[]` for `Vector` but selecting a column (`Vector`) rather than a row when applied to a `Matrix`. A two-argument round bracket operator accesses a `Matrix` element.

```
Matrix m; Vector v; …
v[i]     // ref to ith element of v, 0-based
v(i)     //  same
m[i][j]  // ref to i,jth element of m, 0-based
m(i,j)   //  same, but faster
m[i]     // ref to ith row of m, 0-based
m(j)     // ref to jth column of m, 0-based
```

For the variable-size `Matrix` and `Vector` classes only, there are also operators for selecting sub-vectors and sub-matrices. Like the indexing operators, these return references into the *original* object, not copies. Sub-matrices are thus "lvalues" (in C terminology), meaning that they can appear on the left hand side of an assignment.

```
Matrix m; Vector v; …
v(i,m)      // ref to m-element subvector whose 0th
            //   element is v's ith element
m(i,j,m,n)  // ref to mXn submatrix whose (0,0)
            //   element is m's (i,j) element
```

5.3.4 Output

The C++ operator `<<` is overloaded for all the matrix and vector types so you can look at a human-readable version of their contents via statements like

```
Matrix m;
std::cout << "m=" << m;
```

5.4 Basic Geometry and Mechanics

In this section, we provide information on several basic Simbody classes, all based on the small `Vec` and `Mat` classes described above, that are used to deal with geometrical and mechanical concepts.

5.4.1 Stations (points)

Stations are simply points which are fixed in a particular reference frame or body (i.e., they are "stationary" in that frame). They are specified by the position vector which would take the frame's origin to the station. A position is represented by a `Vec3` type. Simbody does not provide an explicit `Station` class; `Vec3`s are adequate whenever a station is to be specified.

5.4.2 Directions (unit vectors)

Directions are unit vectors, which are `Vec3`s with the additional property that their lengths are always 1. Simbody provides a class `UnitVec3` which behaves identically to `Vec3` in most respects but restricts the ways in which values can be assigned to ensure that the length is always 1. This has the practical advantage that you never need to normalize a `UnitVec3`; it is guaranteed already to have been normalized. A `UnitVec3` can be used in any context or operator that would normally take a `Vec3` (that is, it has an implicit conversion to `Vec3`) except where the context would allow the `Vec3` to be modified in a way that would change its

length. In particular, you can apply a `Rotation` to a `UnitVec3` and get a `UnitVec3` back because that operation is known to preserve length.

Note that when you assign a `Vec3` to a `UnitVec3`, normalization will be performed automatically, whereas assigning a `UnitVec3` to a `Vec3` or to another `UnitVec3` requires no computation. If you attempt to set a `UnitVec3` to zero, you will get `NaN`s instead.

Some specialized constants and classes are provided for dealing conveniently with the coordinate axis directions: `XAxis`, `YAxis`, `ZAxis` are the type-safe constants associated with the `CoordinateAxis` class which provides useful methods for working with these. These constants implicitly convert respectively to 0, 1, or 2 when needed so can be used as indices. There is also a `CoordinateDirection` class whose constants are `-XAxis`, `+YAxis`, etc. `UnitVec3` can provide the equivalent unit vectors.

5.4.3 Rotations

There are many ways to express 3D rotations. Examples are: pitch-roll-yaw, azimuth-elevation-twist, axis-angle, and quaternions. Many others are in common use, and Simbody provides extensive support for most of them. However, each way of writing orientation has its own quirks and complexities, and all of them are equivalent to a 3x3 matrix, called a *rotation matrix* (synonyms: orientation matrix, direction cosine matrix). Rotation matrices have a particularly simple definition and straightforward physical interpretation, and are very easy to work with. Simbody uses the rotation matrix as a least common denominator, embodied in a class `Rotation`. `Rotation` provides a set of methods which can be used to construct a rotation matrix from a wide variety of commonly-used rotation schemes. These are represented internally in objects of type `Rotation` as an ordinary `Mat33`, and can be used wherever a `Mat33` is expected, except that construction, assignment, and writable element access are restricted to ensure that certain properties are maintained. Columns (rows) of a `Rotation` have type `UnitVec3` (`UnitRow3`) rather than `Vec3` (`Row3`).

There can be some confusion as to whether to use a rotation matrix or its inverse in a given context. We use a consistent notation to avoid that confusion, and show here how the notation corresponds to the physical layout of the `SimTK::Rotation` object. The symbol R with left and right superscripts ${}^{from}R^{to}$ represents the orientation of the "to" frame (the right superscript) measured with respect to the "from" frame (the left superscript), like this:

$$ {}^{G}R^{B} \equiv \left({}^{G}\left[B_x \right] \quad {}^{G}\left[B_y \right] \quad {}^{G}\left[B_z \right] \right) $$

(B_x is the x-direction unit vector of frame B, with measure numbers expressed in B's frame, while the operator ${}^{F}\left[\bullet \right]$ indicates that the measure numbers of some physical quantity are re-expressed in coordinate frame F.) So the symbol ${}^{G}R^{B}$ should be read "the axes of frame B expressed in frame G," or "the orientation of frame B in G," or just "B in G." We never use "R" alone for a rotation matrix and neither should you; that is a recipe for certain disaster. Instead, always provide the two frames. Using this notation, you can simply match up superscripts to rotate vectors or compose rotations. When under tight typographical restrictions, as in source code, we write ${}^{G}R^{B}$ in "monogram" notation as `R_GB`. Also, since these are orthogonal, the inverse of a rotation matrix is just its transpose, which serves simply to swap the superscripts. Use the Simbody "~" operator to indicate rotation inversion: $\sim {}^{G}R^{B} = {}^{B}R^{G}$. As an example, if you have a rotation ${}^{G}R^{B}$ and a vector ${}^{B}\mathbf{v}$ expressed in B, you can re-express that same vector in G like this: ${}^{G}\mathbf{v} = {}^{G}R^{B} \bullet {}^{B}\mathbf{v}$. To go the other direction, we can write ${}^{B}\mathbf{v} = {}^{B}R^{G} \bullet {}^{G}\mathbf{v} = \sim {}^{G}R^{B} \bullet {}^{G}\mathbf{v}$. As a C++ code fragment, this can be written:

```
Rotation R_GB;  //orientation of frame B in G
Vec3     v_G;   //a vector expressed in G

…

Vec3     v_B = ~R_GB*v_G; //re-express v_G in frame B
```

Composition of rotations is similarly accomplished by lining up superscripts (subject to order reversal with the "~" operator). So given ${}^{G}R^{B}$ and ${}^{G}R^{C}$ we can get ${}^{B}R^{C}$ as ${}^{B}R^{C} = {}^{B}R^{G} \bullet {}^{G}R^{C} = \sim {}^{G}R^{B} \bullet {}^{G}R^{C}$. Note that the "~" operator has a high precedence like unary "–" so $\sim {}^{G}R^{B} \bullet {}^{G}R^{C}$ is $(\sim {}^{G}R^{B}) \bullet {}^{G}R^{C}$, not $\sim ({}^{G}R^{B} \bullet {}^{G}R^{C})$.

As is typical for Simbody operations on small quantities, the transpose operator is actually just a change in point of view and involves no computation or copying of data. That is, the operations ${}^{B}R^{G} \bullet {}^{G}\mathbf{v}$ and $\sim {}^{G}R^{B} \bullet {}^{G}\mathbf{v}$ are exactly equivalent in both meaning and performance: the cost is just three inline dot products, with no wasted data copying or subroutine calls.

There is also a `Quaternion` class you can use; look it up in the API documentation if you are interested.

5.4.4 Transforms

Transforms combine a rotation and a position (translation) and are used to define the configuration (pose) of one frame with respect to another. We represent a frame B's configuration with respect to another frame G by giving the measure numbers in G of each of B's axes, and the measure numbers in G of the vector from G's origin point to B's origin point, for a total of 4 vectors, which can be interpreted as a 3x3 `Rotation` (see above) followed by the origin point location (a `Vec3`). We call this object a *transform* (abbreviated *xform*) and *conceptually* augment the axes and origin point to create a 4x4 linear operator which can be applied to augmented vectors (4th element is 0) or points (4th element is 1), or composed using matrix multiplication. We define a type `Transform` which conceptually represents transforms as follows:

$$ {}^{G}X^{B} \triangleq \begin{pmatrix} {}^{G}[B_x] & {}^{G}[B_y] & {}^{G}[B_z] & {}^{G}p^{B} \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

(The notation ${}^{G}p^{B} \equiv {}^{Go}p^{Bo}$, that is, the vector from the origin of the G frame to the origin of the B frame.) We use the symbol *X* for transforms, with superscripts ${}^{from}X^{to}$ so ${}^{G}X^{B}$ means "the transform from frame *G* to frame *B*," or "frame *B* measured from and expressed in frame *G*." As for rotations, never write a transform as just *X* without indicating frames. When under tight typographical restrictions, as in source code, we write ${}^{G}X^{B}$ in "monogram" notation as `X_GB`.

Another way to interpret ${}^{G}X^{B}$ is that it represents the operations that must be performed on *G* to bring it into alignment with *B* (a rotation and a translation). Then, as for rotation matrices described above, we can interpret ${}^{G}X^{B} \bullet {}^{B}X^{C}$ as a composition of operators yielding ${}^{G}X^{C}$, and $\sim {}^{G}X^{B}$ is defined to yield the inverse transform ${}^{B}X^{G}$.

The above transform matrix can be considered a matrix of four columns as shown: three augmented vectors and an augmented point. An alternate, and entirely equivalent, way to view this is as a rotation matrix, translation vector, and an extra row:

$$ {}^{G}X^{B} \equiv \begin{pmatrix} \begin{pmatrix} & {}^{G}R^{B} & \end{pmatrix} \begin{pmatrix} {}^{G}p^{B} \end{pmatrix} \\ \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix} \end{pmatrix} $$

In our implementation, the physical layout of a Simbody `Transform` is just the three columns of the rotation matrix followed immediately in memory by the translation vector, that is, ${}^{G}X^{B} = \left({}^{G}R^{B} \mid {}^{G}p^{B} \right)_{3\times 4}$. There is no need for the fourth row to be stored in memory since it is always the same.

The multiplication operator * is overloaded to work with transforms on `Vec3` objects with the assumption that these are points (stations) to be shifted as well as rotated. That is, they are treated as though there were a fourth element set to 1. If you only want to apply a rotation, you can extract the `Rotation` matrix from the `Transform` and then apply that. As an example:

```
Transform X_GB;  //orientation and position of frame B in G
Vec3 v_G;   //a vector expressed in G
Vec3 p_G;   //location of point measured from G's origin, expressed
in G
…
Vec3 p_B = ~X_GB*p_G; //point p, now measured from B's origin, exp.
in B
Vec3 v_B = ~X_GB.R()*v_G; //re-express v_G in frame B, without
shifting
```

Given a `Transform`, you can work with it as though it were a 4x4 matrix, or work directly with the rotation matrix *R* and translation vector *p* individually, without having to make copies (methods *x*`.R()` and *x*`.p()` are available to provide references to the contained objects of a transform *x*). Although a transform defined this way is not orthogonal, its inverse is easy to apply with no additional calculation. Simbody overloads the normal matrix transpose operator "~" to recast a `Transform` to its inverse so that either the transform or its inverse can be used conveniently in an expression, for example, ${}^{B}X^{C} = {\sim}{}^{G}X^{B} \bullet {}^{G}X^{C}$.

5.4.5 Inertia and MassProperties

The Simbody `Inertia` class is a 3x3 symmetric matrix. The class provides some convenient constructors and methods for shifting to and from a body's center of mass. Simbody uses

this class for specifying body inertia properties. There is also a `UnitInertia` class which represents inertia per unit mass (this is sometimes called a "gyration" matrix); multiplying a `UnitInertia` by a mass yields an `Inertia`. The `MassProperties` class combines a mass, center of mass location, and inertia to provide the complete mass properties for a rigid body.

If you want to see what you can do with these classes, look them up in the Doxygen-generated API reference documentation available at https://simtk.org/home/simbody, Documents tab.

5.5 Available Simbody Numerical Methods

Simbody uses many advanced numerical methods, and these are available for direct use in your programs if you need them. We call the collection of numerical methods "Simmath". Some of the most commonly used are:

- Linear algebra (various object-oriented factorization and eigenvalue classes, as well as direct access to Lapack and Blas if needed).
- Optimization (constrained and unconstrained), see `Optimizer` and subclasses.
- Numerical integration (see `Integrator` class and subclasses)
- Numerical differentiation (see `Differentiator` class)
- Random number generation (see `Random` class)
- Polynomial root finding (see `PolynomialRootFinder` class)
- Spline fitting (see `Spline_<T>` and `SplineFitter` classes)

For more information, see the Simmath User's Guide at https://simtk.org/home/simbody, Documents page. Detailed information is provided in the Doxygen-generated API Reference that describes individual classes and methods, also on the Documents page.

In the next chapter we'll see an example of using the `Optimizer` classes.

5.6 Some other Simbody classes and utilities

Simbody contains a collection of utility classes that have proven useful when writing programs that use the Simbody API. You can read about them in the API reference documentation mentioned above. Here we'll give a brief description and where to look for

more information (where there are class or function names you can type them into the search box in the API documentation).

- Parallel execution and threading: `ParallelExecutor`, `Parallel2DExecutor`, `ParallelWorkQueue`, `AtomicInteger`, `ThreadLocal<T>`, `threadCpuTime()` function
- Predefined constants at high precision: see "Predefined Constants" under "Modules" in the API documentation
- Contact classes: see contact example programs in the Simbody examples directory. Also, `Contact`, `ContactGeometry`, `ContactSurface`, `ContactTrackerSubsystem`, `CompliantContactSubsystem`
- Geometry: `DecorativeGeometry`, `ContactGeometry`, `PolygonalMesh`
- Some useful solvers: `LocalEnergyMinimizer`, `Assembler` (inverse kinematics, motion tracking), `ObservedPointFitter`, `project()` method of `System` class.
- Platform-independent file name handling: `Pathname`
- Platform-independent handling of plug-in libraries: `Plugin`
- String handling: `String`
- Reading and writing Xml files: `Xml` and related classes
- Timing and miscellaneous inline utilities: `realTime()`, `cpuTime()`, `step()`, `clamp()`, `square()`, `cube()`, `sign()`, `isFinite()`, `isInf()`, `isNaN()`, elliptic integrals, various bit functions: see Global Functions in the SimTK namespace under "Modules" in the API documentation.

6 Complex Example: A Protein Simulation

In chapter 4 we simulated a very simple system with only two degrees of freedom. Now let's make an enormous jump in complexity, and simulate an entire protein. This would be difficult with just Simbody, but the Molmodel API exists specifically for this purpose and makes it very easy to create Simbody models of molecules.

Even if you are not using Molmodel and have no interest in molecules, you should still take a look at this example since most of what it does involves only the Simbody API. Molmodel is just a modeling layer that helps build Simbody models for a specific class of physical system; any use of Simbody will begin with a modeling phase of some sort.

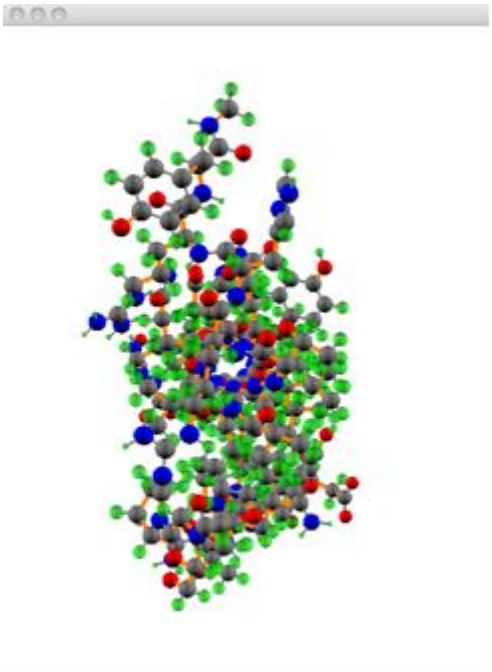
6.1 Simulating a Protein

The following program loads a protein structure from a PDB file, constructs a System representing it, and simulates its behavior for 10 ps.

```
#include "Molmodel.h"
using namespace SimTK;

int main() {
  try {
    // Load the PDB file and construct the system.
    CompoundSystem system;
    SimbodyMatterSubsystem matter(system);
    DecorationSubsystem decoration(system);
    DuMMForceFieldSubsystem forces(system);
    forces.loadAmber99Parameters();
    PDBReader pdb("1PPT.pdb");
    pdb.createCompounds(system);
    system.modelCompounds();
    // Adjust the temperature periodically.
    system.addEventHandler(new VelocityRescalingThermostat(system, 293.15, 0.1));
    // Arrange for visualization.
    system.addEventReporter(new Visualizer::Reporter(system, 0.025));
    // Initialize and get the default state for the simulation.
    State state = system.realizeTopology();
```

```
    pdb.createState(system, state);
    LocalEnergyMinimizer::minimizeEnergy(system, state, 15.0);

    // Simulate it.
    VerletIntegrator integ(system);
    integ.setAccuracy(1e-2);
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(10.0);

  } catch (const std::exception& e) {
    std::cout << "Error: " << e.what() << std::endl;
    return 1;
  }
}
```

When you run this program, a Visualizer window should appear and (after some delay for building the model and doing energy minimization) display the protein:



Before we go through the code in detail, let's take a moment to consider how one goes about modeling a molecule. In previous chapters, we have described multibody systems in terms of rigid bodies connected by joints. In molecular simulations, it's more natural to use a different language. One thinks of molecules, which are made up of atoms connected by covalent bonds. What is the relationship between these two descriptions?

One answer would be to simply equate them. You could say that each atom is a rigid body, and each bond is a joint. That is certainly an option, but it isn't the only one, and it often isn't the best one.

When simulating a macromolecule, one usually constrains the bond lengths to remain fixed, and often some of the bond angles as well. This produces clusters of atoms that are completely immobile with respect to each other. If you treat each of these clusters as a rigid body, the system will have many fewer bodies than atoms. This results in a simpler system that is easier to simulate.

The Molmodel interface to Simbody lets you work with either description, and translate from one to the other. It describes molecules in terms of *Compounds*. A Compound is a set of atoms covalently bonded to each other: a molecule or a piece of a molecule. A Compound may be built up hierarchically out of other compounds. For example, a protein is a single Compound, but each amino acid residue it contains is itself a Compound.

After you create one or more Compounds, you add them to a CompoundSystem, which is a subclass of MultibodySystem. The CompoundSystem examines the Compounds and generates an appropriate set of MobilizedBody objects based on them. You can tell it which bonds and angles should be constrained, and it automatically figures out what sets of atoms form rigid bodies and what degrees of freedom they have.

Now let's look at the code. Notice that we include Molmodel.h instead of Simbody.h. This gets us the molecular modeling extensions we want, and also includes all of Simbody as well. We begin by creating our CompoundSystem and Subsystems:

```
CompoundSystem system;
SimbodyMatterSubsystem matter(system);
DecorationSubsystem decoration(system);
DuMMForceFieldSubsystem forces(system);
```

There are two new Subsystems that we haven't seen before. DecorationSubsystem lets you add arbitrary decorations to a System to control how it is visualized. When CompoundSystem looks through the Compounds and creates MobilizedBody objects from them, it also creates a set of decorations to draw the molecules in a standard way. Decorations never affect the behavior of a simulation. DuMMForceFieldSubsystem provides a standard molecular dynamics force field. It has built in support for the Amber99 force field parameters, which we load by calling

```
forces.loadAmber99Parameters();
```

If you want to use a different set of parameters, you can load them from a file by calling populateFromTinkerParameterFile() instead. It also uses a GBSA implicit solvation model to represent the effect of water. (It doesn't make sense to use explicit solvation for a reduced-coordinate molecular model, except possibly for a few localized waters.)

Now we need to create a set of Compound objects and tell the CompoundSystem to create MobilizedBody objects from them. We could create them directly, but in this example we use a different method: we do it by loading a PDB file:

```
PDBReader pdb("1PPT.pdb");
pdb.createCompounds(system);
system.modelCompounds();
```

The argument to the PDBReader constructor is the name or path of the file to read. In this example, we use the 1PPT structure downloaded from http://www.pdb.org. This is a small, 36 residue protein. When we call createCompounds(), it creates all the necessary Compounds and adds them to the CompoundSystem. Finally, we call modelCompounds(), which causes the CompoundSystem to generate MobilizedBodies based on the Compounds that have been added.

Next we add an event handler to the System:

```
System.addEventHandler(new VelocityRescalingThermostat(system, 293.15, 0.1));
```

Molecular simulations are usually performed at constant temperature, so we need a *thermostat* to maintain the temperature. VelocityRescalingThermostat does this by periodically rescaling all of the velocities in the System. We tell it to maintain a temperature of 293.15 K (20°C), and to rescale the velocities every 0.1 ps. In the previous chapter we used SI units. In this chapter we use a different set of units known as MD units that measure length in nm, mass in Daltons, time in ps, and energy in kJ/mol. Unlike Simbody, which is units-agnostic, the Molmodel interface *requires* MD units. SimTK provides predefined constants that give the values of various physical constants in both systems of units.

We also add a Visualizer::Reporter to show a movie of our simulation, and ask it to show a frame every 0.025 ps. Here we're using a constructor that tells the Reporter to allocate its own Visualizer for the System, with all-default options:

```
system.addEventReporter(new Visualizer::Reporter(system, 0.025));
```

The default background for a CompoundSystem is plain white; that's different from a MultibodySystem where the default background is a ground plane with a blue sky.

After creating the System, we need to create an initial State. We can get an appropriate State object from the return value of the realizeTopology() method; however, that will be some simple, regular configuration (such as fully extended), not the molecule's configuration as found in the pdb file. Once again we turn to the PDBReader:

```
State state = system.realizeTopology();
pdb.createState(system, state);
```

This method figures out the values of all generalized coordinates that produce a best fit to the coordinates read from the PDB file and sets the generalized coordinates (position states **q**) in the State object.

We could use this State as the starting point for our simulation, but that usually is not a good idea. PDB structures have significant uncertainty in them, and may not be at a local minimum of the particular potential function we are using; some atoms may even overlap. If we started the simulation from this State, the protein might experience very large forces that would disrupt its structure. We therefore perform a local energy minimization to get a better starting State:

```
LocalEnergyMinimizer::minimizeEnergy(system, state, 15.0);
```

We now are ready to perform our simulation. This works exactly as it did in the last chapter, although we're using a different Integrator: we create a VerletIntegrator and TimeStepper, initialize it, and tell it to step to 10 ps. There is only one line that is different:

```
integ.setAccuracy(1e-2);
```

Biomolecular systems can usually be integrated at fairly low accuracy. We are simulating a large, chaotic system at constant temperature, so small errors in the integration have very little effect on the long time behavior of the simulation. We therefore tell the Integrator to use lower accuracy than the default. This allows it to take larger time steps, which makes the simulation faster.

6.2 Accelerating Molmodel with OpenMM

If you have OpenMM installed and operating on a hardware-accelerated platform (e.g. CUDA) on your computer, you can easily use it to accelerate the protein simulation we did above. Simply insert the lines:

```
forces.setUseOpenMMAcceleration(true);
forces.setTraceOpenMM(true);
```

before the call to realizeTopology(). Only the first line is required; the second is very useful though to verify that OpenMM is actually being used and to debug the problem if not—it sends helpful trace output to stdout (cout). Note that if you did not install SimTK in its standard location, the environment variable SimTK_INSTALL_DIR must be set so that the OpenMM plugin can be located (it is in $SimTK_INSTALL_DIR/lib/plugins).

For problems with OpenMM installation, go to https://simtk.org/home/openmm. Note that acceleration requires supported hardware, appropriate drivers, and an OpenMM 3.0 installation.

6.3 Vector Arithmetic: Calculating Radius of Gyration

Watching a movie of a protein is fine as far as it goes, but you generally want to analyze your simulations in a more quantitative way. In this section, we will expand the previous example to monitor the radius of gyration of the protein over the course of the simulation. The radius of gyration is defined by

$$R_G = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left|\mathbf{r}_i - \mathbf{r}_{avg}\right|^2}$$

where N is the number of atoms, $\mathbf{r}_i$ is the location of the i'th atom, and $\mathbf{r}_{avg}$ is the average location of all atoms. In this example, we will calculate it based only on the alpha carbon of each residue.

Before looking at the code, this might be a good time to say something about matrix and vector math in SimTK. The Vec3 class has already appeared a few times in the earlier examples, but I didn't comment on it, since it was generally obvious from the context how it was being used. This example will require more significant vector math.

Actually, SimTK has two different sets of classes for vector math. First, there are classes to represent small, fixed size vectors and matrices: Vec for column vectors, Row for row

vectors, and Mat for Matrices. These classes are templatized based on size and element type. Synonyms are defined for common combinations; for example, Vec3 is a synonym for Vec<3,Real>, while Mat22 is a synonym for Mat<2,2,Real>. You can also create other combinations, such as Mat<2,10,Real> or Vec<4,Complex>.

Second, there are classes to represent large vectors and matrices whose sizes are determined at runtime: Vector_ for column vectors, RowVector_ for row vectors, and Matrix_ for matrices. These classes are templatized based on element type only. Vector, RowVector, and Matrix are synonyms for Vector_<Real>, RowVector_<Real>, and Matrix_<Real>, respectively. Again, you can use other element types. In fact, the element type can even be a fixed-size vector or matrix. For example, Vector_<Vec3> is a variable-length vector, where each element is itself a three-component vector.

All of these classes support standard mathematical operators like +, -, and *. The ~ operator performs a transpose (or more precisely, Hermitian conjugate which is the same as transpose for real numbers). There also are versions of many standard math functions that operate on vectors and matrices: sin(), exp(), sqrt(), abs(), sum(), mean(), etc. These allow many calculations to be written in a very concise way.

Here is the code for the event reporter that monitors the radius of gyration.

```
class RadiusReporter : public PeriodicEventReporter {
public:
    RadiusReporter(const CompoundSystem& system, const Compound& compound,
                   Real interval)
    :   PeriodicEventReporter(interval), system(system), compound(compound) {
        for (Compound::AtomIndex i = Compound::AtomIndex(0); i <
                compound.getNAtoms(); ++i) {
            std::string name = compound.getAtomName(i);
            if (name.size() > 3 && name.substr(name.size()-3) == "/CA")
                atoms.push_back(i);
        }
    }
    void handleEvent(const State& state) const {
        system.realize(state, Stage::Position);
        Vector_<Vec3> pos(atoms.size());
        for (int i = 0; i < atoms.size(); ++i)
            pos[i] = compound.calcAtomLocationInGroundFrame(state, atoms[i]);
        pos -= mean(pos);
        Real radius = std::sqrt(~pos*pos/atoms.size());
        std::cout<<state.getTime()<<"\t"<<radius<<std::endl;
    }
```

```
private:
    const CompoundSystem&             system;
    const Compound&                   compound;
    std::vector<Compound::AtomIndex> atoms;
};
```

We add it to the System with the following line:

```
system.addEventReporter(new RadiusReporter(system,
    system.getCompound(Compound::Index(0)), 0.1));
```

The constructor arguments are the CompoundSystem we are working with, the Compound for which to calculate R_G, and the reporting interval.

The constructor builds a list of all the atoms that will be used in the calculation. It does this by looping over all atoms in the Compound and checking the name of each one. If the name ends in "/CA", it is the alpha carbon of some residue, so we add it to the list.

Now look at handleEvent(). Since we will be working with the locations of atoms in Cartesian coordinates, we first realize the State to Position stage. We then look up the location of each atom:

```
Vector_<Vec3> pos(atoms.size());
for (int i = 0; i < atoms.size(); ++i)
    pos[i] = compound.calcAtomLocationInGroundFrame(state, atoms[i]);
```

We store the locations in a Vector_<Vec3>; that is, a vector of which each element is itself a three-component vector.

We now need to calculate R_G. Since it involves a sum over $\mathbf{r}_i$-$\mathbf{r}_{avg}$, we first subtract the average atom position from each one:

```
pos -= mean(pos);
```

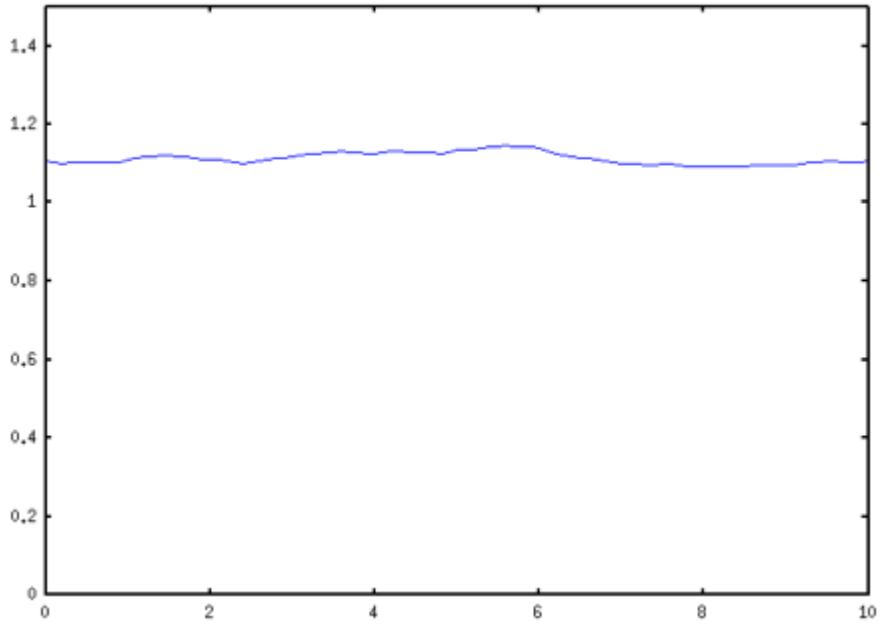
We now can do the rest of the calculation in a single line:

```
Real radius = std::sqrt(~pos*pos/atoms.size());
```

That line may seem a bit confusing at first. How does it represent the entire calculation for radius of gyration? Let's look at it more closely, particularly the expression ~pos*pos. Remember that ~ is the transpose operator, so this expression is simply the dot product of pos with itself (you can also write dot(pos,pos) if you prefer). It multiplies each element by itself, and adds them up. But each element of pos is itself a Vec3. The ~ operator transposes

these sub-vectors along with the parent vector, so we are actually taking the dot product of each Vec3 with itself (yielding the absolute value squared), then adding all of them up. Finally we divide by the number of atoms and take the square root to yield R_G: not bad for a single line of code!

Here is a graph of the radius of gyration over the course of the simulation. As you can see, it has very little variation, indicating that the protein is stable.



Before we leave this example, I should point out that I cheated a bit at one point. I assumed the System contained only one top level Compound, and simply called getCompound(Compound::Index(0)) to look it up. This is not always a good assumption. If the PDB file contained multiple protein chains, each one would be a separate Compound. I really should have asked the System how many Compounds there were, and looped over all of them. But I happened to know that the particular PDB file I was working with only contained one chain, and making the code more general would have resulted in an example that was more complicated but no more educational. For real programs, though, you should handle this more robustly.

6.4 Using an Optimizer: RMS Distance from Native

In this section we will measure a different quantity over the course of our simulation: the root-mean-square distance (RMSD) from the native structure. This is defined as

$$RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left|\mathbf{r}_i - \mathbf{r}_i^0\right|^2}$$

where $\mathbf{r}_i$ is the position of the i'th atom at the current time, and $\mathbf{r}_i^0$ is the position of that atom in the native structure.

At first, this might seem like a trivial modification to the previous example, but there's a catch. Over time, the molecule may drift or rotate away from its starting position. We don't care about that, and we don't want it to cause the RMSD to change. We only care about conformational changes in the molecule. We therefore need to align the two states before calculating the RMSD. That is, out of all possible translations and rotations that could be applied to the molecule, we want to find the one that minimizes the RMSD.

SimTK provides a tool for solving problems of this sort: the Optimizer class. To use it, you define a function of the form

$$y = f(\mathbf{x})$$

where $\mathbf{x}$ is a vector of parameters, and f($\mathbf{x}$) is a real valued function of those parameters. Optimizer searches for a set of values for the parameters $\mathbf{x}$ at which y is a local minimum. You can also impose constraints on the allowed parameter values.

You define the function by creating an object which subclasses OptimizerSystem. Here is an OptimizerSystem that calculates the RMSD between two structures.

```
class RMSDFunction : public OptimizerSystem {
public:
    RMSDFunction(const Vector_<Vec3>& pos1, const Vector_<Vec3>& pos2)
    :   OptimizerSystem(6), pos1(pos1), pos2(pos2) {}

    int objectiveFunc(const Vector& params, bool new_params,
                      Real& f) const {
        Rotation R;
        R.setRotationToBodyFixedXYZ(Vec3(params[0], params[1], params[2]));
        Transform X(R, Vec3(params[3], params[4], params[5]));
        Vector_<Vec3> diff = X*pos1-pos2;
        f = std::sqrt(~diff*diff/pos1.size());
        return 0;
    }
private:
    const Vector_<Vec3>& pos1;
    const Vector_<Vec3>& pos2;
};
```

The arguments to the constructor contain the atom positions in the two States we want to compare. We pass 6 to the superclass constructor to tell it that our function has six parameters to be optimized: three rotation angles and three translation distances.

Now look at objectiveFunc() which calculates the function to be minimized. Each time it is called, it receives a vector of parameter values. We choose to interpret them as a transformation X to be applied to pos1. We therefore use them to create a Transform object, which combines a rotation (represented by a Rotation object R) and a translation (represented by a Vec3):

```
Rotation R;
R.setRotationToBodyFixedXYZ(Vec3(params[0], params[1], params[2]));
Transform X(R, Vec3(params[3], params[4], params[5]));
```

We apply it to pos1, take the difference between that and pos2, and calculate the RMSD exactly as we calculated the radius of gyration in the previous section:

```
Vector_<Vec3> diff = X*pos1-pos2;
f = std::sqrt(~diff*diff/pos1.size());
```

An OptimizerSystem can implement other methods as well. If you know how to calculate the gradient or Jacobian of the function analytically, you can provide methods to calculate them. This can speed up the optimization quite a bit, especially when there are many parameters. If you want to apply constraints to the parameters, you can provide a method to evaluate the constraint errors. But for simple cases like this one, a single method is all we need.

Now we are ready to write our event reporter. Here it is.

```
class RMSDReporter : public PeriodicEventReporter {
public:
    RMSDReporter(const CompoundSystem& system, const Compound& compound,
                 Real interval)
    :   PeriodicEventReporter(interval), system(system), compound(compound) {
        for (Compound::AtomIndex i = Compound::AtomIndex(0);
             i < compound.getNAtoms(); ++i) {
            std::string name = compound.getAtomName(i);
            if (name.size() > 3 && name.substr(name.size()-3) == "/CA")
                atoms.push_back(i);
        }
    }
    void setReferenceState(const State& state) {
        system.realize(state, Stage::Position);
```

```
        refPos.resize(atoms.size());
        for (int i = 0; i < atoms.size(); ++i)
            refPos[i] = compound.calcAtomLocationInGroundFrame(state, atoms[i]);
    }
    void handleEvent(const State& state) const {
        system.realize(state, Stage::Position);
        Vector_<Vec3> pos(atoms.size());
        for (int i = 0; i < atoms.size(); ++i)
            pos[i] = compound.calcAtomLocationInGroundFrame(state, atoms[i]);
        RMSDFunction func(refPos, pos);
        Optimizer opt(func);
        opt.useNumericalGradient(true);
        opt.useNumericalJacobian(true);
        Vector parameters(6, 0.0);
        opt.optimize(parameters);
        Real rmsd;
        func.objectiveFunc(parameters, true, rmsd);
        std::cout<<state.getTime()<<"\t"<<rmsd<<std::endl;
    }
private:
    const CompoundSystem&            system;
    const Compound&                  compound;
    Vector_<Vec3>                    refPos;
    std::vector<Compound::AtomIndex> atoms;
};
```

The constructor is identical to the one in the previous example. After the System has been created and the State representing the native structure has been created, we must call setReferenceState(). This looks up the position of every atom and stores them for later use.

Now look at handleEvent(). The first few lines should look very familiar. Once again, we realize the State and look up the position of every atom. We then create an instance of our OptimizerFunction, passing to it the two vectors of atomic positions we want to compare:

```
RMSDFunction func(refPos, pos);
```

We then create and initialize an Optimizer:

```
Optimizer opt(func);
opt.useNumericalGradient(true);
opt.useNumericalJacobian(true);
```

Those last two lines tell the Optimizer how to calculate the gradient and Jacobian of the objective function. By default, it calls methods on the OptimizerSystem to calculate them.

Since we have not provided any such methods, we tell it to calculate them from numerical differences instead.

We now create a vector of parameter values and invoke the Optimizer:

```
Vector parameters(6, 0.0);
opt.optimize(parameters);
```

The constructor arguments tell it to create a vector of length 6, and to initialize all elements to 0. On entry, this vector contains the initial parameter values from which to search for a local minimum. On exit, it contains the values that optimize the objective function.

We now have the optimal parameter values, but we really want to know the corresponding RMSD. That requires one more call to the RMSDFunction:

```
Real rmsd;
func.objectiveFunc(parameters, true, rmsd);
```

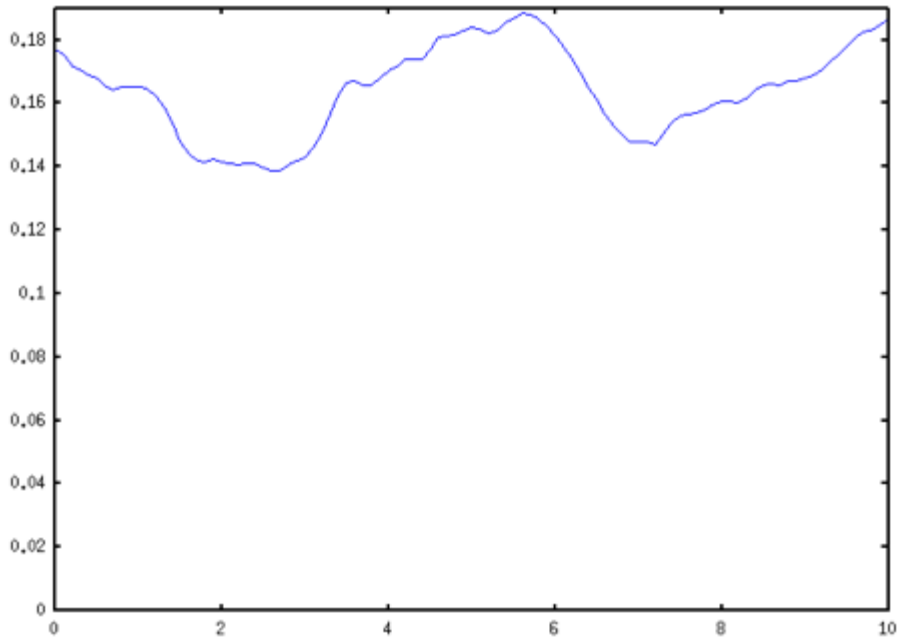
At the time we add the RMSDReporter to our System, we don't yet have a State representing the native structure, so we need to keep a reference to it:

```
RMSDReporter* rmsd = new RMSDReporter(system,
    system.getCompound(Compound::Index(0)), 0.1);
system.addEventReporter(rmsd);
```

Then, after we call createState() on the PDBReader to find the native structure, we can pass it on to the reporter:

```
rmsd->setReferenceState(state);
```

Here is a graph of the RMSD over the course of the simulation:



It fluctuates up and down, but overall is fairly flat. This again shows that the protein is stable and is not drifting away from the native structure.

Before leaving this example, I should mention that using an Optimizer is not actually the most efficient way to solve this particular problem. Finding the optimal alignment between two sets of points can be done analytically, which is faster than doing an iterative optimization. But of course, the real goal of this example was to show how to use the Optimizer class, which is useful for solving a wide variety of problems.