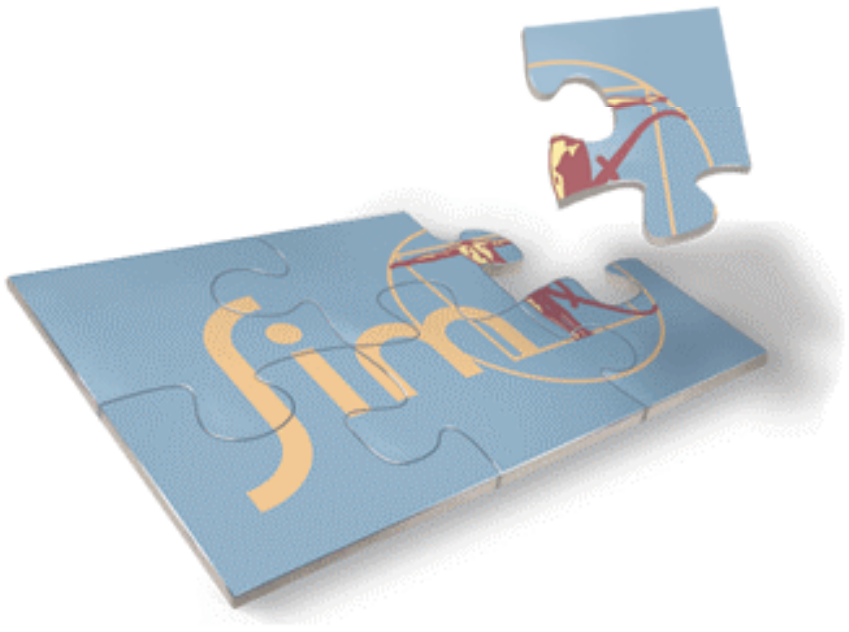
SimTK Documents



SimTK Tutorial

Release 1.0

March 3, 2008

Website: SimTK.org/home/dissemination

Acknowledgments

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

Table of Contents

1 Introduction

1.1 What is SimTK?

SimTK is a toolkit for physics based simulation of biological structures. It contains tools for doing lots of jobs related to this purpose: vector and matrix math, linear algebra, numerical integration, optimization, etc. Most importantly, it includes Simbody, a library for performing internal coordinate simulations of multibody systems. Let's take a moment to consider what that means.

A *multibody system* is a physical system composed of objects, each of which is rigid or nearly rigid, but which can move 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 small 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 motion.

Now let's consider what is meant by an *internal coordinate* simulation. 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.

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 as a result of the transformation, 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.

1.2 Mathematical Overview

In the previous section, I defined a multibody system as 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 would be described by the current angle and angular velocity of every joint. We refer to this vector as **y**. The job of a simulation is to numerically integrate the equation of motion

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

where the function **f**(**y**, t) 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, while the generalized speeds would be the corresponding angular velocities. (It is also possible for a system to have more generalized coordinates than generalized speeds, but don't worry about that right now.) If you were simulating a person walking and wanted to keep track of the total distance they had walked, that would be an auxiliary variable; it does not describe the configuration of the skeleton, but still needs to be integrated along with the other state variables. The full state vector is the union of these subvectors: $\mathbf{y} = [\mathbf{q}, \mathbf{u}, \mathbf{z}]$.

It sometimes is necessary to impose 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:

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

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 single differential equation. 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 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 the state variables which are monitored continuously during the simulation. An *event* is said to occur when a trigger function crosses through 0:

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

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.

2 Architecture

2.1 The SimTK Stack

SimTK consists of a set of modules that form a stack. Each module depends on those that come before it, but not the ones after 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.
CPODES	Contains the CPODES numerical integrator developed at Lawrence Livermore National Laboratory
Simmath	Provides a variety of high level numerical tools for integration, differentiation, and optimization
Simbody	Provides algorithms and data structure for modeling multibody systems in internal coordinates
Molmodel	Provides tools for modeling biological macromolecules such as proteins and nucleic acids

For now, we will not worry too much about the boundaries between the different modules. All of them are included in the SimTK Core installer, and you can access all of them from your code by including a single header file: SimTKmolmodel.h if you are doing a biomolecular simulation, SimTKsimbody.h otherwise.

2.2 Systems and States

SimTK uses two classes to represent the information associated with a multibody system: System and State. You can think of these as the "constant" and "non-constant" parts of the system, respectively. System stores everything that is expected to remain constant over the course of a simulation, while State stores everything that is expected to change.

More specifically, a State object stores the following values:

- The time t.
- The continuous state variables **y**.
- The discrete state variables **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. Its 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 **f**(**y**, **d**, t), the vector of constraint functions **c**(**q**, **u**, **d**, t), and the vector of event trigger functions **e**(**y**, **d**, t).
- 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.

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.

SimTK 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 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 **y**, **d**, and t 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.

Here is the complete list of stages. There are ten in all.

1. Empty

2. Topology
3. Model
4. Instance
5. Time
6. Position
7. Velocity
8. Dynamics
9. Acceleration
10. Report

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. This is the lowest stage at which the System can project the State onto the constraint manifold.

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, it will automatically "back itself up" to an earlier stage, invalidating cache entries from later stages so they can no longer be accessed. In particular:

- Modifying t or **q** will bring the State back to Time stage.
- Modifying **u** will bring the State back to Position stage.
- Modifying **z** will bring the State back to Velocity stage.
- When a System defines a discrete state variable, it specifies what stage the State should be reverted to when that variable is modified. This should be 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, **e**(**y**, **d**, t). 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 $\mathbf{e}(\mathbf{y}, \mathbf{d}, t) = t - t_{event}$. 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! SimTK 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. SimTK 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 ScheduledEventHandler. 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.

3 Example: A Double Pendulum

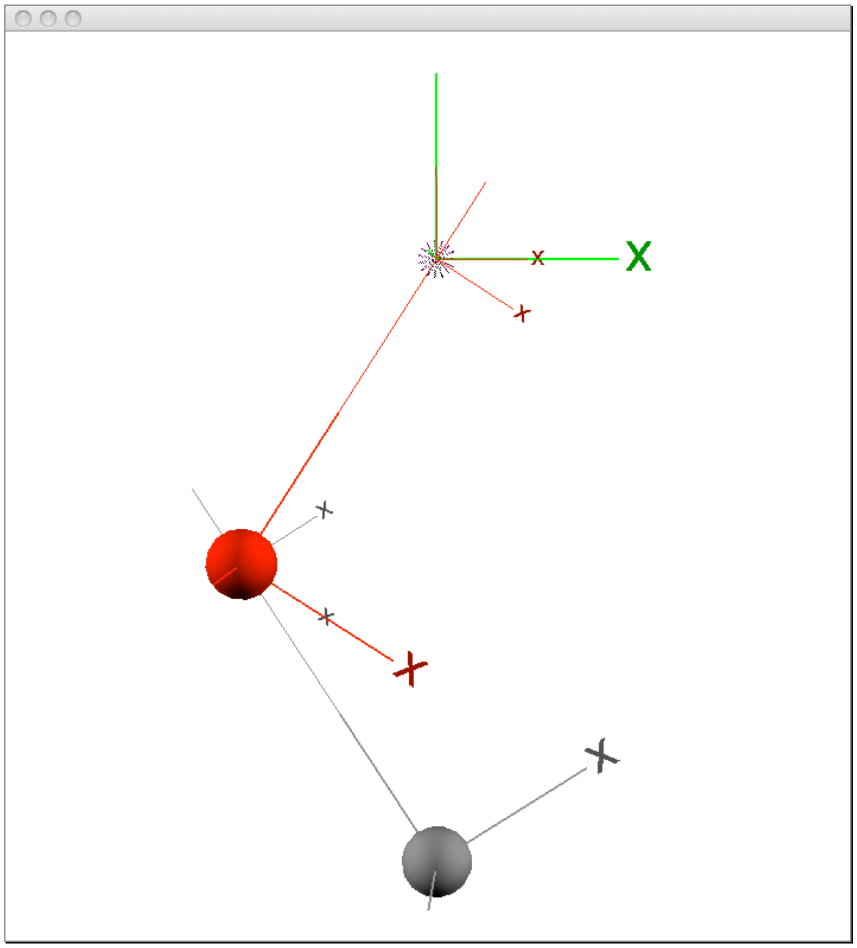
3.1 A First Example

It's now time to look at our 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 "SimTKsimbody.h"
#include "SimTKsimbody_aux.h"

using namespace SimTK;

int main() {

    // 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)));
    system.updDefaultSubsystem().addEventReporter(
            new VTKEventReporter(system, 0.01));

    // Initialize the system and state.

    system.realizeTopology();
    State state = system.getDefaultState();
    pendulum2.setOneU(state, 0, 5.0);

    // Simulate it.

    VerletIntegrator integ(system);
    TimeStepper ts(system, integ);
    ts.initialize(state);
    ts.stepTo(50.0);
}
```

Before you can compile and run this program, you need to run the SimTK Core installer. It will create a folder with "include" and "lib" subfolders. Make sure the "include" folder is part of your compiler's include path, and the "lib" folder is available to the program at runtime. 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 a couple of include statements:

```
#include "SimTKsimbody.h"
#include "SimTKsimbody_aux.h"
```

I mentioned SimTKsimbody.h earlier, and said it was the only file you needed to include to access all the SimTK classes. That was almost true, but not quite. If you want to use VTK for visualizing your system, you also need to include SimTKsimbody_aux.h. If your program does not display a graphical user interface, or does not use the SimTK user interface classes, you can omit including it.

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

```
using namespace SimTK;
```

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·m^2 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)));
```

MobilizedBody::Pin defines a pin joint. It 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 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.

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.

Now let's look at the second and fourth arguments, which are both Transform objects. These are the *inboard* (toward the root of the tree) and *outboard* (away from the root of the tree) transforms. A pin joint allows the body to rotate around a central point. The inboard transform defines the location of that central point relative to the parent body. In our case, we specify a vector of length 0, so each body simply rotates around its parent. The outboard transform defines the location of the body relative to the central point. We specify a vector of length 1 pointing in the y direction, so when both joints are in their neutral positions (that is, when **q**=[0, 0]), the pendulum hangs straight down.

Our System is almost complete at this point, but there's one more line:

```
system.updDefaultSubsystem().addEventReporter(new VTKEventReporter(system, 0.01));
```

As you can see, we are adding an event reporter to the System. 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. VTKEventReporter uses VTK 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 fast your computer is, one second of simulation time may take more or less than one second to calculate and display.)

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

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

This function 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 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.setOneU(state, 0, 5.0);
```

We now have a System to simulate, and an initial State to begin the simulation from. It's time to do some simulating!

```
VerletIntegrator 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 VerletIntegrator, which uses the velocity Verlet algorithm.

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 SimTK 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.

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

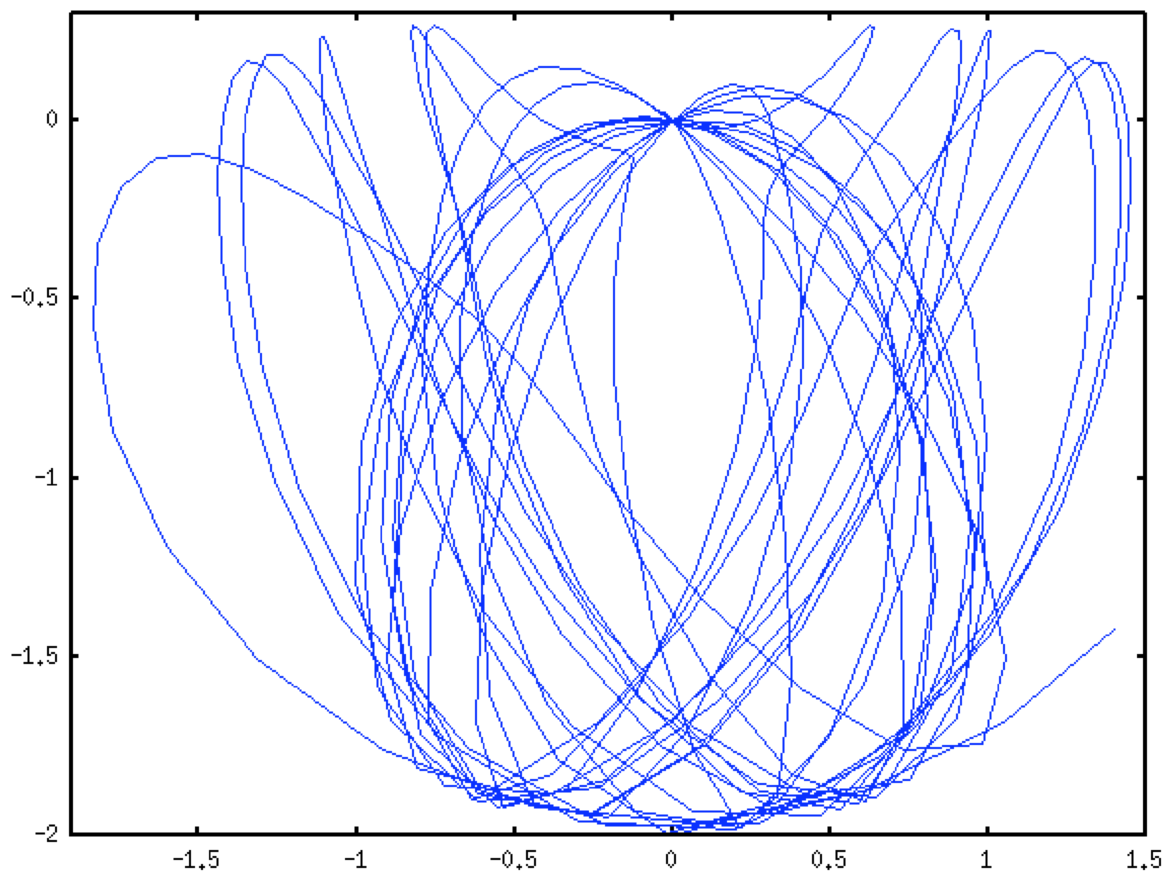
We can add this event reporter to our System with the following line:

```
system.updDefaultSubsystem().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& body,
    Real reportInterval) : PeriodicEventReporter(reportInterval),
    system(system), body(body) {
}
```

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. By default it corresponds to double precision, but you can configure it to be something else instead.

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 = body.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.

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

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

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

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 SimTK that events should not occur on rising sign transitions (when the trigger function increases from negative to positive). We only care about falling sign transitions.

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

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

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. A good rule to follow is, “When in doubt, call realize().”

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.

3.4 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, const Vector& yWeights, const
Vector& ooConstraintTols, Stage& lowestModified, bool& shouldTerminate) const {
        state.updU() *= 0.2;
        lowestModified = Stage::Velocity;
    }
};
```

We add it to the System with the following line:

```
system.updDefaultSubsystem().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, const Vector& yWeights, const
Vector& ooConstraintTols, Stage& lowestModified, bool& shouldTerminate) const {
```

That's a *lot* 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 three arguments related to the accuracy requirements for the event handler. Remember, 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.

The simplest way in which the user can specify these requirements is with a single number for the overall accuracy of the simulation. That is the "accuracy" argument. 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.

The System may also provide weights for the individual state variables and constraint functions. These are the next two arguments. You should multiply the errors by the corresponding weights when comparing them to the requested accuracy.

Finally, there are two arguments which are used for output: lowestModified and shouldTerminate. If you modify the State in any way, you should set lowestModified to the appropriate value to tell SimTK what you have modified. In some cases, this can allow the Integrator to avoid work if it knows that certain aspects of the State have not been modified.

The actual implementation of this method is quite simple:

```
state.updU() *= 0.2;
lowestModified = Stage::Velocity;
```

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 of "upd".

Finally, since we have modified **u**, we set lowestModified to Velocity. This lets the Integrator know that we have modified velocities but not positions.

3.5 Constraints

Working in internal coordinates greatly reduces the number of constraints needed for most systems, but it often cannot eliminate them altogether. Fortunately, SimTK makes it very easy to add constraints to your System.

Each constraint is represented by an object of the Constraint class. Subclasses of Constraint represent different kinds of constraints. SimTK 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 which 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 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.

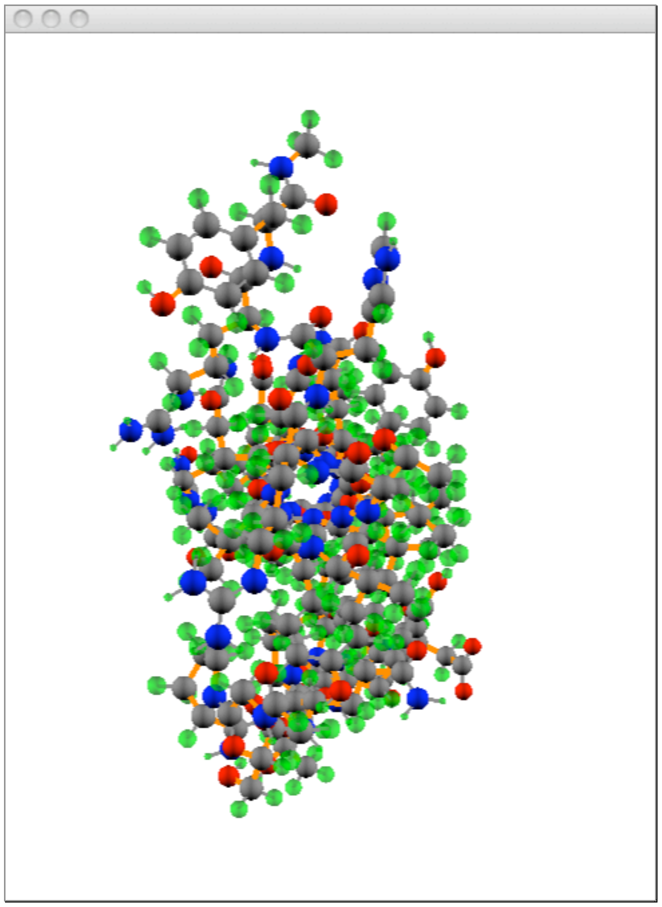
4 Example: A Protein Simulation

4.1 Creating a Protein

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

```
#include "SimTKmolmodel.h"
#include "SimTKsimbody_aux.h"

using namespace SimTK;

int main() {

    // Load the PDB file and construct the system.

    CompoundSystem system;
    SimbodyMatterSubsystem matter(system);
    DecorationSubsystem decoration(system);
    TinkerDuMMForceFieldSubsystem forces(system);
    forces.loadAmber99Parameters();
    PDBReader pdb("1PPT.pdb");
    pdb.createCompounds(system);
    system.modelCompounds();
    system.updDefaultSubsystem().addEventHandler(new VelocityRescalingThermostat(
        system, SimTK_BOLTZMANN_CONSTANT_MD, 293.15, 0.1));
    system.updDefaultSubsystem().addEventReporter(new VTKEventReporter(system,
        0.025));
    system.realizeTopology();

    // Create an initial state for the simulation.

    State& state = system.updDefaultState();
    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);
}
```

When you run this program, a window should appear (after some delay for building the model and doing energy minimization) displaying 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 bonds 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.

SimTK 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 SimTKmolmodel.h instead of SimTKsimbody.h. This gets us the molecular modeling classes along with the rest of SimTK. We begin by creating our CompoundSystem and Subsystems:

```
CompoundSystem system;
SimbodyMatterSubsystem matter(system);
DecorationSubsystem decoration(system);
TinkerDuMMForceFieldSubsystem 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.

TinkerDuMMForceFieldSubsystem 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 solvation model to represent the effect of water.

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.updDefaultSubsystem().addEventHandler(new VelocityRescalingThermostat(
        system, SimTK_BOLTZMANN_CONSTANT_MD, 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 Sytem. We tell it to maintain a temperature of 293.15 K (20°C), and to rescale the velocities every 0.1 ps.

Notice that one of the arguments is the value of Boltzmann's constant. This may seem a little odd. Doesn't it know the value of Boltzmann's constant already? The answer is no, because it doesn't know what system of units you are using. In the previous chapter we used SI units. In this chapter we use a different set of units known as MD units. It measures length in nm, mass in Daltons, time in ps, and energy in kJ/mol. SimTK provides predefined constants that give the values of various physical constants in both systems of units.

We also add a VTKEventReporter to show a movie of our simulation, and ask it to show a frame every 0.025 ps:

```
system.updDefaultSubsystem().addEventReporter(new VTKEventReporter(system,
        0.025));
```

After creating the System, we need to create an initial State. Once again we turn to the PDBReader:

```
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 configures 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. If we started the simulation from this State, the protein would experience very large forces that might 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: 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.

4.2 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 slightly 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, $\mathbf{r}_\mathrm{i}$ is the location of the i’th atom, and $\mathbf{r}_\mathrm{avg}$ 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. 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 vector or matrix itself. For example, Vector_<Vec3> is a 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). 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(atoms[i], state);
        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.updDefaultSubsystem().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(atoms[i], state);
```

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:

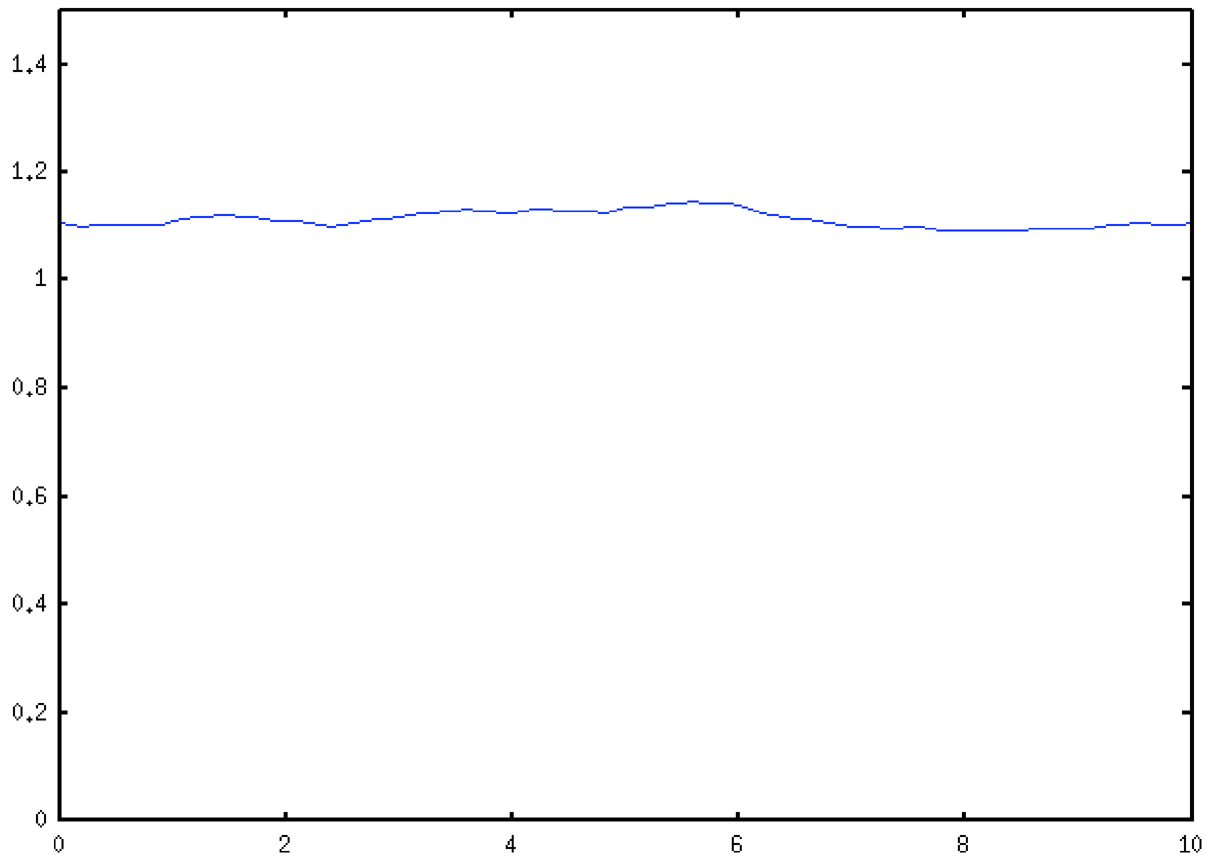
```
pos -= mean(pos);
```

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. 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.

4.3 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 **x** is a vector of parameters, and f(**x**) is a real valued function of those parameters. Optimizer searches for a set of values for the parameters **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, const bool new_params,
            Real& f) const {
        Rotation r;
        r.setRotationToBodyFixedXYZ(Vec3(params[0], params[1], params[2]));
        Transform t(r, Vec3(params[3], params[4], params[5]));
        Vector_<Vec3> diff = t*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 applied to pos1. We therefore use them to create a Transform object, which combines a rotation (represented by a Rotation object) and a translation (represented by a Vec3):

```
Rotation r;
r.setRotationToBodyFixedXYZ(Vec3(params[0], params[1], params[2]));
Transform t(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 = t*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(atoms[i], state);
    }
    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(atoms[i], state);
        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);
```

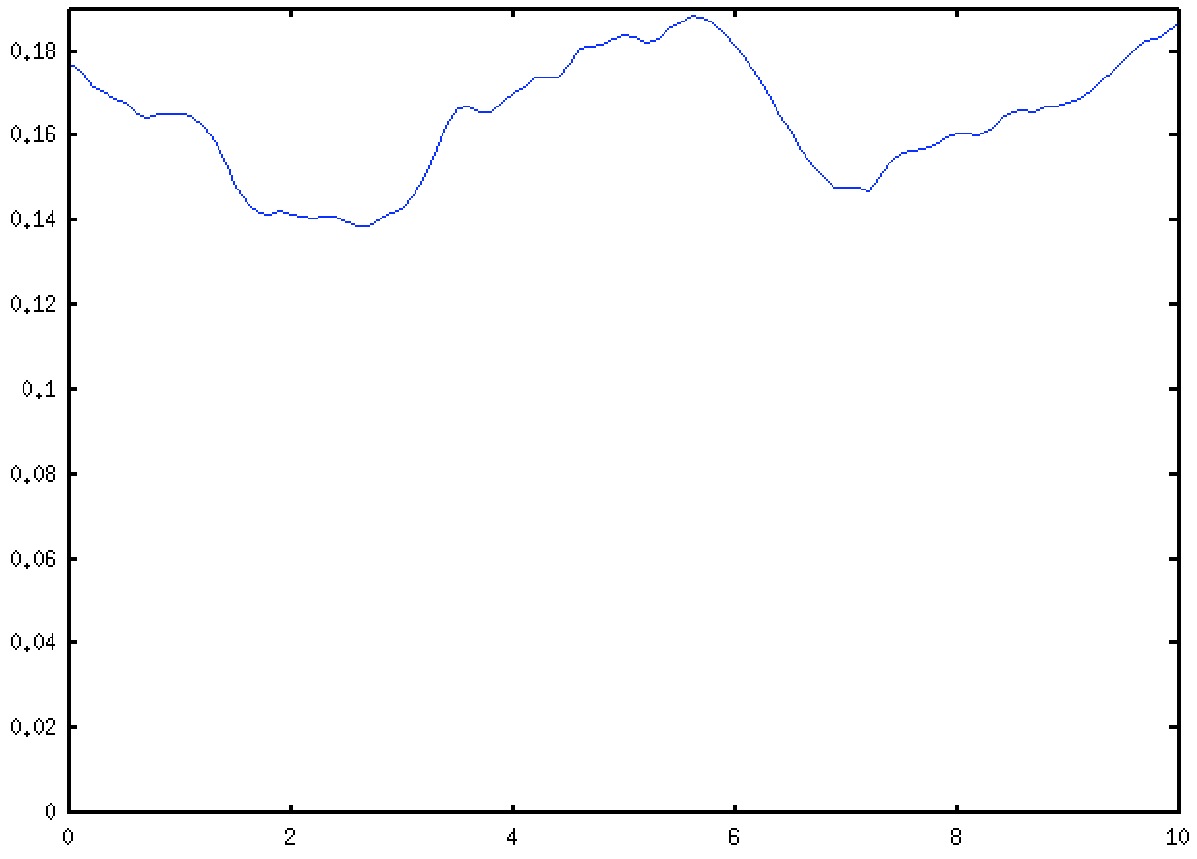
At the time we add the RMSDReporter to our System, we don't yet have a State representing the native structure, so we need to keep a reference to it:

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