



Advanced Programming with SimTK

Release 1.5

August 8, 2008

Website: SimTK.org/home/dissemination

Copyright and Permission Notice

Portions copyright (c) 2008 Stanford University and Peter Eastman.

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 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
1.1	Extending SimTK.....	1
1.2	Realization Revisited.....	2
1.3	The First Four Stages	3
2	A CUSTOM SUBSYSTEM	5
2.1	A First Subsystem.....	5
2.2	A ForceSubsystem	7
2.3	Adding Parameters to the Subsystem	9
2.4	Creating a State Variable.....	12
2.5	Other Subsystem Features	15
3	OTHER EXTENSIONS	18
3.1	A Custom Force	18
3.2	A Custom Constraint.....	20
3.3	Simple Constraints	24
3.4	A Custom MobilizedBody (The Easy Case).....	27
3.5	A Custom MobilizedBody (The Hard Case).....	29

1 Introduction

1.1 Extending SimTK

Before reading this, you should already have read the SimTK Tutorial. In that tutorial, you learned how to build SimTK Systems out of Subsystems, MobilizedBodies, Constraints, and other objects. You can go a very long way doing nothing but that—building up Systems out of prewritten parts—but you may reach a point where it is not enough. Perhaps you need a Constraint that does not correspond to any of the standard constraint types. Or maybe you need to model an object that cannot be represented with the standard MobilizedBody classes. It's not enough to just use the classes provided by the toolkit; you need to write new ones. That is what this document will teach you how to do.

Let's begin by reviewing some things you learned in the tutorial. A System is made up of Subsystems. Each Subsystem can do any of the following:

1. Define state variables, which can be categorized into generalized coordinates (**q**), generalized speeds (**u**), auxiliary variables (**z**), and discrete variables (**d**).
2. Calculate information to be stored in the realization cache of a State object.
3. Calculate the time derivatives of continuous state variables.
4. Define constraint equations.
5. Define event trigger functions and event handlers.

You will learn how to write new Subsystems that may do any or all of these. This is the most general way that you can extend SimTK Systems with custom code.

It often isn't the most convenient way, though. If all you want is to define one new constraint type, you shouldn't need to write an entire Subsystem. SimTK provides simpler mechanisms for extending a System in common ways. In fact, you have already seen one of them: rather than writing a new Subsystem to define an event handler, you simply write an EventHandler or EventReporter object, then add it to the default Subsystem. You also can

write custom subclasses of Force, Constraint, and MobilizedBody. We will see examples of all of these.

1.2 Realization Revisited

As you learned in the SimTK Tutorial, a State object holds two types of information: *state variables* and *cached results*. Cached results are stored in the *realization cache*, which is divided up into *stages*. Before you can access information in the cache, you must first make sure the State has been realized to the appropriate stage. For example, it must be at Position stage or later to access Cartesian coordinates, and at Velocity stage or later to access Cartesian velocities.

Stages can also be thought about in another way. Every state variable is associated with a particular cache stage:

Variable	Stage
t	Time
q	Position
u	Velocity
z	Dynamics
d	any

(A discrete state variable may be associated with any stage except Empty or Topology. When a Subsystem defines a discrete variable, it specifies what stage to associate it with.)

When a State is being realized to a particular stage, the values calculated and stored in the cache may only depend on state variables for *that stage* or *earlier stages*. They may not depend in any way on state variables associated with later stages.

Why is this? Because whenever a state variable is modified, the cache is automatically reverted back to the stage immediately *before* the stage associated with that variable. If you modify a generalized coordinate, the state is reverted back to Time stage. If you modify a

generalized speed, the state is reverted back to Position stage. This ensures that any information in the cache which might depend on that variable is discarded.

Suppose that a Subsystem failed to obey this rule. Suppose that, while realizing a State to Position stage, it made use of the generalized speeds. Then, at some later point, the speeds were modified. Those cached values would no longer be consistent with the state variables. But because the Position stage cache entries would not be discarded, they would still be present in the cache and accessible to anyone who looked for them.

When you were simply using classes written by other people, you didn't have to worry about this. You simply trusted that information in the cache would always be correct. But now that you are preparing to write extensions to SimTK, you need to be aware of it. You need to know what promises are made by the System, and you have a responsibility to make sure your code does not break them.

1.3 The First Four Stages

There are a total of ten stages. The last six were discussed in the SimTK tutorial, but very little mention was made of the first four stages: Empty, Topology, Model, and Instance. When running a simulation, there is usually not much reason to think about these stages, since all information you are likely to want is associated with one of the later stages. But when you are writing extensions to SimTK, these stages are very important. They are where the State gets constructed and configured to hold data. Let's examine each one of them.

Empty: This is the stage a newly constructed State object is in before it has been realized. It contains no information at all, and is not specific to any particular System.

Topology: When a State gets realized to Topology stage, it is configured to become a State for a particular System. In practice, this usually means allocating space in the cache for whatever data the System needs to store, and creating Model stage state variables. The Topology stage is unique in that no state variables may correspond to it. There is no such thing as a "Topology stage state variable". What happens when you realize a State to

Topology stage can only depend on intrinsic properties of the System (“topological properties”), not on the value of any state variable.

Model: When a State is realized to Model stage, its complete set of state variables and cache entries becomes fixed. This means that the set of state variables may depend on the value of a Model stage state variable. For example, Simbody allows rotations to be modeled with either quaternions or Euler angles. You select which representation to use by calling `setUseEulerAngles()` on the `SimbodyMatterSubsystem`. Your choice is stored in a Model stage discrete state variable. If you select Euler angles, three generalized coordinates will be created for each rotation. If you select quaternions, there will be four.

This has an important consequence: Model stage variables may not be changed in the middle of a simulation. If you did, States created before the change might contain different state variables than ones created after, and they could not be compared to each other. Model stage variables are used to configure a System *before* beginning a simulation, not once the simulation has started.

Instance: At Instance stage, the set of constraint functions and event trigger functions becomes fixed. This means that the set of constraints and event handlers may potentially change during a simulation. For example, Simbody uses an Instance stage variable to record which Constraints are disabled. Instance stage is also useful for precalculating values that will only change during discrete events, not during continuous integration intervals.

2 A Custom Subsystem

2.1 A First Subsystem

In this chapter, we will write a custom Subsystem. We will build it up in pieces, starting from the simplest possible Subsystem: one that does nothing at all.

```
#include "SimTKsimbody.h"
#include "SimTKcommon/internal/SubsystemGuts.h"

using namespace SimTK;

class ExampleSubsystemImpl : public Subsystem::Guts {
public:
    Subsystem::Guts* cloneImpl() const {
        return new ExampleSubsystemImpl();
    }
};

class ExampleSubsystem : public Subsystem {
public:
    ExampleSubsystem(MultibodySystem& system) {
        adoptSubsystemGuts(new ExampleSubsystemImpl());
        system.adoptSubsystem(*this);
    }
};
```

To understand this code, you first need to know that a Subsystem is actually defined by two different classes. Subsystem defines the “public interface” to it—those properties that most people access most of the time. There also are many properties related to the implementation, which most users of the Subsystem do not care about most of the time. To keep the interface clean, these properties are split off into a separate class called Subsystem::Guts. An object of this type is created automatically for each Subsystem and can be accessed by calling getSubsystemGuts() on it.

To define a new type of Subsystem, you must create a subclass of each of these classes. The Subsystem subclass defines the public API, while the Subsystem::Guts subclass defines the implementation.

Now let's look at the example above. We define a class called ExampleSubsystemImpl that will provide our implementation. Since our Subsystem doesn't currently do anything, there isn't much to implement. The only method it is required to implement is cloneImpl(). There are many others which it *can* implement, and we will see some of those later. But all the others are only required if you need to provide certain features in your Subsystem.

ExampleSubsystem is equally simple. It defines only a constructor:

```
ExampleSubsystem(MultibodySystem& system) {
    adoptSubsystemGuts(new ExampleSubsystemImpl());
    system.adoptSubsystem(*this);
}
```

The Subsystem we are planning to create will work only with MultibodySystems, so we require one as a constructor argument. The constructor creates an ExampleSubsystemImpl, registers it by calling adoptSubsystemGuts(), and adds itself to the System.

You can now create an ExampleSubsystem and add it to a System exactly as you would any other Subsystem:

```
MultibodySystem system;
SimbodyMatterSubsystem matter(system);
ExampleSubsystem example(system);
```

Notice that the ExampleSubsystem is created as just a local variable. It will disappear as soon as that variable goes out of scope. The ExampleSubsystemImpl is the true persistent object. For this reason, you should never add fields to a Subsystem subclass or try to store information in it. Instead, store all information in the Subsystem::Guts subclass. A Subsystem object is merely a glorified pointer to it.

One other point is worth mentioning before we go on. Notice that we had to include an extra header file:

```
#include "SimTKcommon/internal/SubsystemGuts.h"
```

Why was this necessary? In the past, it has always been enough to just include SimTKsimbody.h, which includes all the header files you need. Usually, that is true. It includes all the headers that most people need most of the time. But it does not include headers that are needed only when writing extensions to SimTK. Those must be included separately.

2.2 A ForceSubsystem

Now let's make our Subsystem actually do something. It will cause all the MobilizedBodies in the System to repel each other with a force proportional to $1/r^2$:

```
#include "SimTKsimbody.h"
#include "simbody/internal/ForceSubsystemGuts.h"

using namespace SimTK;

class ExampleSubsystemImpl : public ForceSubsystem::Guts {
public:
    ExampleSubsystemImpl() : ForceSubsystem::Guts("Example", "1.0") {
    }
    Subsystem::Guts* cloneImpl() const {
        return new ExampleSubsystemImpl();
    }
    int realizeSubsystemDynamicsImpl(const State& state) const {
        const MultibodySystem& system =
            static_cast<const MultibodySystem&>(getSystem());
        const SimbodyMatterSubsystem& matter = system.getMatterSubsystem();
        Vector_<SpatialVec>& forces = system.updRigidBodyForces(state,
            Stage::Dynamics);
        for (MobilizedBodyIndex i(0); i < matter.getNBodies(); i++) {
            const MobilizedBody& body1 = matter.getMobilizedBody(i);
            for (MobilizedBodyIndex j(0); j < i; j++) {
                const MobilizedBody& body2 = matter.getMobilizedBody(j);
                Vec3 r = body1.getBodyOriginLocation(state)-
                    body2.getBodyOriginLocation(state);
                Real distance = r.norm();
                Vec3 force = r/(distance*distance*distance);
                forces[i][1] += force;
                forces[j][1] -= force;
            }
        }
    }
}
```

```

        return 0;
    }
    Real calcPotentialEnergy(const State& state) const {
        const MultibodySystem& system = static_cast<const
            MultibodySystem&>(getSystem());
        const SimbodyMatterSubsystem& matter = system.getMatterSubsystem();
        double energy = 0.0;
        for (MobilizedBodyIndex i(0); i < matter.getNBodies(); i++) {
            const MobilizedBody& body1 = matter.getMobilizedBody(i);
            for (MobilizedBodyIndex j(0); j < i; j++) {
                const MobilizedBody& body2 = matter.getMobilizedBody(j);
                Vec3 r = body1.getBodyOriginLocation(state)-
                    body2.getBodyOriginLocation(state);
                energy -= 1.0/r.norm();
            }
        }
        return energy;
    }
};

class ExampleSubsystem : public ForceSubsystem {
public:
    ExampleSubsystem(MultibodySystem& system) {
        adoptSubsystemGuts(new ExampleSubsystemImpl());
        system.addForceSubsystem(*this);
    }
};

```

The first thing to notice is that we are using different parent classes: `ForceSubsystem` and `ForceSubsystem::Guts`. `ForceSubsystem` is a subclass of `Subsystem` defined by `Simbody`, which you should use for any `Subsystem` that applies forces to bodies. This ensures that `Subsystems` will be realized in the proper order, and also defines a `calcPotentialEnergy()` method in which you can calculate your force's contribution to the potential energy of the system. Also notice that we add the `Subsystem` to the `System` by calling `addForceSubsystem()`, instead of `adoptSubsystem()` like we did in the previous example.

We have added a method to `ExampleSubsystemImpl`, `realizeSubsystemDynamicsImpl()`. This will be called each time a `State` is being realized to `Dynamics` stage. There are similar methods for all the other stages. Each one has a default implementation that does nothing, so you only have to implement the ones you need.

Very little of the code in this method should look unfamiliar. We begin by calling `getSystem()` to get a reference to the `System` this `Subsystem` is part of:

```
const MultibodySystem& system =
    static_cast<const MultibodySystem&>(getSystem());
```

We look up its `SimbodyMatterSubsystem`, then call `updRigidBodyForces()` to get a writeable reference to the vector of spatial forces acting on the bodies:

```
Vector_<SpatialVec>& forces = system.updRigidBodyForces(state, Stage::Dynamics);
```

Now we have a pair of nested loops over each pair of bodies. We calculate the displacement between them, use that to calculate a force, and add it to the appropriate entries in the vector.

There is a very similar method called `calcPotentialEnergy()`. This method is defined by `ForceSubsystem::Guts`. When someone calls `calcPotentialEnergy()` on a `MultibodySystem`, it loops over each of its `ForceSubsystems`, calls `calcPotentialEnergy()` on each one, and returns the sum of their potential energies.

2.3 Adding Parameters to the Subsystem

The force we implemented in the last section resembles Coulomb repulsion: lots of charged bodies all repelling each other. Wouldn't it be useful if you could change the strength of the force (to represent changing the charge on each body)? Of course, you could just edit the source code and recompile, but what if you want to run lots of simulations, each with a different strength for the force? It would be much easier if you could set the strength programmatically.

Here is a version that allows that:

```
class ExampleSubsystemImpl : public ForceSubsystem::Guts {
public:
    ExampleSubsystemImpl() : ForceSubsystem::Guts("Example", "1.0"),
        defaultForceStrength(1.0) {}
    Subsystem::Guts* cloneImpl() const {
        return new ExampleSubsystemImpl();
    }
};
```

```

}
int realizeSubsystemDynamicsImpl(const State& state) const {
    const MultibodySystem& system =
        static_cast<const MultibodySystem&>(getSystem());
    const SimbodyMatterSubsystem& matter = system.getMatterSubsystem();
    Vector_<SpatialVec>& forces = system.updRigidBodyForces(state,
        Stage::Dynamics);
    for (MobilizedBodyIndex i(0); i < matter.getNBodies(); i++) {
        const MobilizedBody& body1 = matter.getMobilizedBody(i);
        for (MobilizedBodyIndex j(0); j < i; j++) {
            const MobilizedBody& body2 = matter.getMobilizedBody(j);
            Vec3 r = body1.getBodyOriginLocation(state)-
                body2.getBodyOriginLocation(state);
            Real distance = r.norm();
            Vec3 force = defaultForceStrength*r/(distance*distance*distance);
            forces[i][1] += force;
            forces[j][1] -= force;
        }
    }
    return 0;
}

Real calcPotentialEnergy(const State& state) const {
    const MultibodySystem& system =
        static_cast<const MultibodySystem&>(getSystem());
    const SimbodyMatterSubsystem& matter = system.getMatterSubsystem();
    double energy = 0.0;
    for (MobilizedBodyIndex i(0); i < matter.getNBodies(); i++) {
        const MobilizedBody& body1 = matter.getMobilizedBody(i);
        for (MobilizedBodyIndex j(0); j < i; j++) {
            const MobilizedBody& body2 = matter.getMobilizedBody(j);
            Vec3 r = body1.getBodyOriginLocation(state)-
                body2.getBodyOriginLocation(state);
            energy -= defaultForceStrength/r.norm();
        }
    }
    return energy;
}

void setDefaultForceStrength(Real strength) {
    defaultForceStrength = strength;
    invalidateSubsystemTopologyCache();
}

Real getDefaultForceStrength() const {
    return defaultForceStrength;
}

private:
    Real defaultForceStrength;
};

class ExampleSubsystem : public ForceSubsystem {
public:
    ExampleSubsystem(MultibodySystem& system) {

```



```

        adoptSubsystemGuts(new ExampleSubsystemImpl());
        system.addForceSubsystem(*this);
    }
    void setDefaultForceStrength(Real strength) {
        updImpl().setDefaultForceStrength(strength);
    }
    Real getDefaultForceStrength() const {
        return getImpl().getDefaultForceStrength();
    }
private:
    ExampleSubsystemImpl& updImpl() {
        return static_cast<ExampleSubsystemImpl&>(updRep());
    }
    const ExampleSubsystemImpl& getImpl() const {
        return static_cast<const ExampleSubsystemImpl&>(getRep());
    }
};

```

First look at `ExampleSubsystemImpl`. We have added a field called `defaultForceStrength`. Why “default”? Because in the next section we will introduce a state variable for storing the force strength, and the value stored in the `Subsystem` will simply be the default value for newly created `States`. We have added a pair of accessor methods for getting and setting it, and modified the force and energy calculation to use it.

There is one line that needs explanation. In `setDefaultForceStrength()`, we call `invalidateSubsystemTopologyCache()`. The default force strength is stored in the `Subsystem`. That means it is a topological property. Whenever you modify a topological property, it is very important that you call this method. It marks that topology has changed, and all existing `States` need to be realized from `Topology` stage onward. If you try to use an old `State` object without doing that, you will get an error.

You might wonder why this is necessary. If a topological change actually affects the data stored in a `State` (such as the set of cache entries), obviously old `States` will no longer be valid, but why does it matter for a simple change to the force constant?

The answer is that it is *especially* important for changes like this, because that is the only way to catch a variety of errors. A topological change should only ever be made before the start of a simulation, not in the middle. Otherwise, a saved `State` from earlier in the simulation (created based on an old value of the force constant) could easily get passed to a

routine that would try to analyze it based on the new value. This is a very insidious sort of bug, because there is no way to detect it by looking at the State object itself. Calling `invalidateSubsystemTopologyCache()` ensures that all such errors will be caught.

If this seems restrictive, don't worry. There's an easy solution, which we'll see in the next section: if you want to be able to change something in the middle of a simulation, make it a state variable instead of a topological property.

The changes to `ExampleSubsystem` are very simple. We added two accessor methods, which just invoke the corresponding methods of `ExampleSubsystemImpl`. For convenience, note that we created two methods for looking up the `ExampleSubsystemImpl`:

```
ExampleSubsystemImpl& updImpl() {
    return static_cast<ExampleSubsystemImpl&>(updRep());
}
const ExampleSubsystemImpl& getImpl() const {
    return static_cast<const ExampleSubsystemImpl&>(getRep());
}
```

Since we will be accessing it many times, this saves us from having to write a `static_cast` every time.

2.4 Creating a State Variable

We really want to be able to change the force strength at any time, not just before starting a simulation. To do that, it needs to be stored in a state variable. The following example shows how to do it. (Some methods are omitted to avoid repeating code you have already seen.)

```
class ExampleSubsystemImpl : public ForceSubsystem::Guts {
public:
    ...

    int realizeSubsystemTopologyImpl(State& state) const {
        forceStrengthIndex = state.allocateDiscreteVariable(getMySubsystemIndex(),
            Stage::Dynamics, new Value<Real>(defaultForceStrength));
        return 0;
    }
}
```

```

int realizeSubsystemDynamicsImpl(const State& state) const {
    const MultibodySystem& system =
        static_cast<const MultibodySystem&>(getSystem());
    const SimbodyMatterSubsystem& matter = system.getMatterSubsystem();
    Vector_<SpatialVec>& forces = system.updRigidBodyForces(state,
        Stage::Dynamics);
    Real forceStrength = getForceStrength(state);
    for (MobilizedBodyIndex i(0); i < matter.getNBodies(); i++) {
        const MobilizedBody& body1 = matter.getMobilizedBody(i);
        for (MobilizedBodyIndex j(0); j < i; j++) {
            const MobilizedBody& body2 = matter.getMobilizedBody(j);
            Vec3 r = body1.getBodyOriginLocation(state)-
                body2.getBodyOriginLocation(state);
            Real distance = r.norm();
            Vec3 force = forceStrength*r/(distance*distance*distance);
            forces[i][1] += force;
            forces[j][1] -= force;
        }
    }
    return 0;
}

void setForceStrength(State& state, Real strength) const {
    static_cast<Value<Real>&>(updDiscreteVariable(state,
        forceStrengthIndex)).upd() = strength;
}

Real getForceStrength(const State& state) const {
    return static_cast<const Value<Real>&>(getDiscreteVariable(state,
        forceStrengthIndex)).get();
}

private:
    Real defaultForceStrength;
    mutable int forceStrengthIndex;
};

```

Let's start by looking at realizeSubsystemTopologyImpl(). The first thing to notice is that, unlike realizeSubsystemDynamicsImpl(), it receives a non-const reference to the State. Topology and Model are the only stages where the State can be modified during realization (such as by defining new state variables).

That may seem strange. Isn't the whole point of realization to modify the State and store new information in it? The answer is that all cache entries are mutable. They are exactly what the name suggests: a cache. The state variables are the true information. The cache entries can always be regenerated based on them. So a const State reference still allows you to modify the cache, but not the state variables.

Now look at what happens when we realize Topology stage:

```
forceStrengthIndex = state.allocateDiscreteVariable(getMySubsystemIndex(),  
    Stage::Dynamics, new Value<Real>(defaultForceStrength));
```

We call `allocateDiscreteVariable()` on the State to create a new discrete variable. There are three arguments: the index of the Subsystem it belongs to, the stage it should be associated with, and a Value object to hold the actual value. We specify Dynamics stage, since that is the earliest stage that uses it. Notice that Value is a template. A discrete state variable can hold any type of value, even a large data structure. It is not restricted to just real values the way continuous state variables are.

The return value from `allocateDiscreteVariable()` is an index, which we will use whenever we want to access the value. We store that index in a mutable field of `ExampleSubsystemImpl`.

I hope you just cried out in horror at that last sentence? If not, please reread it and think carefully about why it's such a shocking statement! A System (including all its Subsystems) is supposed to be immutable during a simulation. All mutable information is supposed to go into the State. And here we are, realizing a State object... and we *modified a field of the Subsystem*? Didn't we just violate that design principle? And what will happen when we realize a different State, and write a new value to the field?

In any other situation, you would be completely correct. Topology stage is the *only* time when we are allowed to modify the System during realization. And not just any change; only properties that are calculated as a result of realizing the State. Indices are calculated for the state variables and cache entries as they are allocated, and we need to store them for future reference.

But what happens when we realize another State? Won't the indices get overwritten with new values? That's where Topology stage is special. Remember, there are no Topology stage state variables. What happens in `realizeTopology()` can depend *only* on topological properties of the System, which means you will get exactly the same indices every time it is called (unless you modify a topological property, in which case your previously created States will become invalid anyway.)

But wait a minute—what about Model stage? You can also allocate state variables then. Does that mean it's also alright to store indices in the System while realizing Model stage?

The answer is a very emphatic *NO!* What happens during Model stage can depend on state variables, so you might get different indices for different States. Instead, create a data structure to hold any indices that will be calculated at Model stage. Then at *Topology* stage, allocate a discrete variable to hold an instance of that structure. That way, all the indices calculated at Model stage will be stored in the State, not the System.

Now look at how we access the state variable:

```
void setForceStrength(State& state, Real strength) const {
    static_cast<Value<Real>&>(updDiscreteVariable(state,
        forceStrengthIndex)).upd() = strength;
}
Real getForceStrength(const State& state) const {
    return static_cast<const Value<Real>&>(getDiscreteVariable(state,
        forceStrengthIndex)).get();
}
```

We invoke `updDiscreteVariable()` to get a writable reference to the Value object, or `getDiscreteVariable()` to get a const reference. We cast the result to the appropriate type (`Value<Real>&`), then call `get()` or `upd()` to get a reference to the actual value.

Finally, `realizeSubsystemDynamicsImpl()` and `calcPotentialEnergy()` need to be modified to use the value stored in the state variable. This is trivial: we just call `getForceStrength()`, then use that value instead of the default value.

2.5 Other Subsystem Features

There are many other things a Subsystem can do. Some are trivial to implement, while others are rarely used, so rather than discuss extended examples of each one, we will simply go through them quickly in this section.

If a Subsystem needs to create a cache entry, that works almost exactly like creating a discrete state variable. You call `allocateCacheEntry()` to create it, and `getCacheEntry()` or `updCacheEntry()` to access a value. Like state variables, each cache entry is associated with a particular stage, but the meaning is different. For a cache entry, the stage is used to determine when the value should be invalidated. If you define a cache entry to be associated with Position stage, that means it should be invalidated (and hence made inaccessible) if any Position stage state variable is modified.

Allocating continuous state variables is also very simple. You invoke `allocateQ()`, `allocateU()`, or `allocateZ()` on the State to allocate a contiguous block of state variables. Each of these methods takes a Vector containing the initial values of the state variables to allocate. The length of the Vector determines how many variables will be allocated. The return value is the index of the start of the block within the Vector of state variables *for that Subsystem*. For example, if you allocate a block of **q**'s by writing

```
qindex = state.allocateQ(getMySubsystemIndex(), initialValues);
```

then you would look up the value of the first one by calling

```
value = state.getQ(getMySubsystemIndex())[qindex];
```

If you allocate continuous state variables, you will usually also want to implement `realizeSubsystemAccelerationImpl()` to calculate their derivatives. You set them by calling `updQDot()`, `updQDotDot()`, `updUDot()`, and `updZDot()` on the State.

Another feature of Subsystems is the ability to define event handlers. This is not a widely used feature, because it is usually easier to use the `EventHandler` and `EventReporter` classes. But sometimes you might prefer to have a Subsystem do its own event handling, especially if the events are closely related to other functions of the Subsystem.

To define a scheduled event or scheduled report, the Subsystem should call `createScheduledEvent()` when realizing Instance stage. This allocates a globally unique event ID that is thereafter used to refer to that event. Next, you need to implement one or both of the two methods `calcTimeOfNextScheduledEvent()` and

`calcTimeOfNextScheduledReport()`. These methods return the next time at which any events or reports will occur, and the IDs of all events/reports that will occur at that time. Finally, you need to implement one or both of the methods `handleEvents()` and `reportEvents()`. These are called when an event/report occurs, and are given the IDs of all events/reports that occurred.

To define a triggered event or triggered report, call `createTriggeredEvent()` when realizing Instance stage. This returns both an event ID and an index into the Vector of trigger function values. When realizing the appropriate stage, you should calculate the value of the trigger function, then store it in the State by calling

```
state.updEventsByStage(getMySubsystemIndex(), stage)[eventIndex] = value;
```

You then implement `handleEvents()` to handle the event when it occurs.

Another function Subsystems can perform is to define constraints. Once again, doing this directly from a Subsystem is rarely the easiest way. In the next chapter, we will see other ways to define constraints.

A constraint equation may be applied to coordinates, velocities, or accelerations. When realizing Instance stage, call `allocateQErr()` to create a constraint on coordinates, `allocateUErr()` to create a constraint on velocities, and `allocateUDotErr()` to create a constraint on accelerations. The actual quantities you will calculate and store in the State are the “constraint errors”. Every constraint is defined by an equation of the form $c(\mathbf{y}, \mathbf{d}, t) = 0$. You calculate the value of the function $c(\mathbf{y}, \mathbf{d}, t)$, and SimTK then tries to keep it equal to 0. You set the constraint errors when realizing the appropriate stages by calling `updQErr()`, `updUErr()`, and `updUDotErr()`.

3 Other Extensions

3.1 A Custom Force

Writing a new Subsystem is the most general way to add custom features to a SimTK System, but it is not always the easiest way. Simbody provides special classes for writing common types of extensions: custom Forces, custom Constraints, and custom MobilizedBodies. We will look at these classes in this chapter and see examples of how to use them.

In the last chapter, we wrote a custom Subsystem that caused all the bodies in a MultibodySystem to repel each other. In this section, we will implement the same force in a simpler way. You will remember from the tutorial that GeneralForceSubsystem provides a general mechanism for adding arbitrary Force objects to a System. Simbody provides a number of Force subclasses that implement common sorts of forces: Force::UniformGravity, Force::TwoPointLinearSpring, etc. It also provides a class called Force::Custom that can be used to define completely new forces.

To use it, you must write a subclass of Force::Custom::Implementation. Here is one that implements our global repulsion force:

```
class ExampleForce : public Force::Custom::Implementation {
public:
    ExampleForce(SimbodyMatterSubsystem& matter) : matter(matter) {
    }
    void calcForce(const State& state, Vector_<SpatialVec>& bodyForces,
                  Vector_<Vec3>& particleForces, Vector& mobilityForces) const {
        for (MobilizedBodyIndex i(0); i < matter.getNBodies(); i++) {
            const MobilizedBody& body1 = matter.getMobilizedBody(i);
            for (MobilizedBodyIndex j(0); j < i; j++) {
                const MobilizedBody& body2 = matter.getMobilizedBody(j);
                Vec3 r = body1.getBodyOriginLocation(state)-
                    body2.getBodyOriginLocation(state);
                Real distance = r.norm();
                Vec3 force = r/(distance*distance*distance);
                bodyForces[i][1] += force;
                bodyForces[j][1] -= force;
            }
        }
    }
}
```



```

}
Real calcPotentialEnergy(const State& state) const {
    double energy = 0.0;
    for (MobilizedBodyIndex i(0); i < matter.getNBodies(); i++) {
        const MobilizedBody& body1 = matter.getMobilizedBody(i);
        for (MobilizedBodyIndex j(0); j < i; j++) {
            const MobilizedBody& body2 = matter.getMobilizedBody(j);
            Vec3 r = body1.getBodyOriginLocation(state)-
                body2.getBodyOriginLocation(state);
            energy -= 1.0/r.norm();
        }
    }
    return energy;
}
bool dependsOnlyOnPositions() const {
    return true;
}
private:
    SimbodyMatterSubsystem& matter;
};

```

The `calcForce()` method is called to calculate the force. Notice that it has three different arguments for storing forces into: `bodyForces`, `particleForces`, and `mobilityForces`. Use `bodyForces` to apply Cartesian forces and torques to bodies. That is what we are doing in this example. You also can use `mobilityForces` to apply forces directly to individual degrees of freedom. That is, there is only element corresponding to each generalized speed. A Force object may apply either or both types of force.

(Currently, `particleForces` is ignored. That is because `Simbody` does not yet support particles. It is expected that they will be added in a future version, so the interface includes them for forward compatibility.)

Similarly, `calcPotentialEnergy()` is called to calculate the potential energy due to the force. Both it and `calcForce()` are nearly identical to the corresponding methods we wrote in the last chapter.

Finally, there is an optional method called `dependsOnlyOnPosition()`. The default implementation returns false. If you override it to return true, that enables an optimization to avoid recalculating the force and energy when a generalized speed or auxiliary state variable is modified. Since our force depends only on \mathbf{q} , not on \mathbf{u} or \mathbf{z} , we return true. This will potentially make our simulations run faster.

Here is how to add the custom force to a MultibodySystem:

```
MultibodySystem system;
SimbodyMatterSubsystem matter(system);
GeneralForceSubsystem forces(system);
Force::Custom(forces, new ExampleForce(matter));
```

As you see, it works just like any other Force object. We simply create a Force::Custom, passing an instance of our Implementation class as an argument.

3.2 A Custom Constraint

Writing a custom Constraint is very similar to writing a custom Force: you create a class that extends Constraint::Custom::Implementation, then pass an instance to the constructor of a Constraint::Custom. But before showing an example, I need to give you warning: writing Constraints is significantly more difficult than writing Forces. It isn't that the programming interface is hard. There are several methods to implement, but that's not a big deal. The problem is that most constraints just involve a lot of math, and if your Constraint fails to work correctly, it can be difficult to figure out exactly where you made the mistake.

In principle, constraints are simple. A constraint is just an equation of the form $c(\mathbf{y}, \mathbf{d}, t) = 0$. How hard can that be to implement? Actually, there are some constraints that really are as simple as that, and Simbody offers a special mechanism that lets you implement them in a truly easy way. We will discuss it in the next section. The problem is that, in many cases, the constraint function depends in some enormously complex way on a very large set of state variables.

Consider, for example, a simple Constraint::Rod. This specifies that the distance between points on two different bodies must remain fixed. The positions of those points depend on the positions and orientations of the two bodies. And those, in turn, depend on the generalized coordinates for the two bodies and for *every one of their parent bodies back to ground*. Trying to actually write the constraint equation explicitly would be completely impractical.

On its own, that isn't usually a problem. After all, SimTK will calculate the locations of the points for you, and it's easy enough to then calculate the distance between them. But there is a second issue that complicates matters. Most "constraints" actually involve multiple constraint equations. The Rod constraint, for example, is actually three constraints: a position-level constraint requiring the distance between two points to be fixed; a velocity-level constraint requiring their relative velocity to be zero; and an acceleration-level constraint requiring their relative acceleration to be zero. You must implement all of these and make sure they are all consistent with each other.

Constraints can be divided into three categories: holonomic, non-holonomic, and acceleration-only. A *holonomic* constraint is one defined at the position level (such as a Rod constraint). A holonomic constraint implies three constraint equations, as discussed above. A *non-holonomic* constraint is defined at the velocity level. An example is `Constraint::ConstantSpeed`: it sets no restriction on the allowed values of coordinates, only on how those coordinates change with time. A non-holonomic constraint implies two constraint equations (one at the velocity level and one at the acceleration level). An acceleration-only constraint is defined at the acceleration level, and implies only a single constraint equation.

Alright, let's take a look at an example. Here is a custom constraint that requires the distance between two bodies' origins to remain fixed. This is, of course, just a special case of a Rod constraint. So this isn't a very useful class, but it is a fairly easy one to understand, so it makes a good example.

```
class ExampleConstraint : public Constraint::Custom::Implementation {
public:
    ExampleConstraint(MobilizedBody& b1, MobilizedBody& b2, Real distance) :
        Implementation(b1.updMatterSubsystem(), 1, 0, 0), distance(distance) {
        body1 = addConstrainedBody(b1);
        body2 = addConstrainedBody(b2);
    }
    Implementation* clone () const {
        return new ExampleConstraint(*this);
    }
    void realizePositionErrors(const State& state, int mp,
        Real* perr) const {
        Vec3 r1 = getBodyOriginLocation(state, body1, true);
        Vec3 r2 = getBodyOriginLocation(state, body2, true);
        perr[0] = (r1-r2).norm()-distance;
    }
};
```

```

}

void realizePositionDotErrors(const State& state, int mp,
    Real* pverr) const {
    Vec3 r1 = getBodyOriginLocation(state, body1, true);
    Vec3 r2 = getBodyOriginLocation(state, body2, true);
    Vec3 r = r2-r1;
    Vec3 v1 = getBodyVelocity(state, body1, true)[1];
    Vec3 v2 = getBodyVelocity(state, body2, true)[1];
    Vec3 v = v2-v1;
    pverr[0] = dot(v, r);
}

void realizePositionDotDotErrors(const State& state,
    int mp, Real* paerr) const {
    Vec3 r1 = getBodyOriginLocation(state, body1, true);
    Vec3 r2 = getBodyOriginLocation(state, body2, true);
    Vec3 r = r2-r1;
    Vec3 v1 = getBodyVelocity(state, body1, true)[1];
    Vec3 v2 = getBodyVelocity(state, body2, true)[1];
    Vec3 v = v2-v1;
    Vec3 a1 = getBodyAcceleration(state, body1, true)[1];
    Vec3 a2 = getBodyAcceleration(state, body2, true)[1];
    Vec3 a = a2-a1;
    paerr[0] = dot(a, r) + dot(v, v);
}

void applyPositionConstraintForces(const State& state, int mp,
    const Real* multipliers, Vector_<SpatialVec>& bodyForcesInA,
    Vector& mobilityForces) const {
    Vec3 r1 = getBodyOriginLocation(state, body1, true);
    Vec3 r2 = getBodyOriginLocation(state, body2, true);
    Vec3 r = r2-r1;
    Vec3 force = multipliers[0]*r;
    addInStationForce(state, body2, Vec3(0), force, bodyForcesInA);
    addInStationForce(state, body1, Vec3(0), -force, bodyForcesInA);
}

private:
    ConstrainedBodyIndex body1, body2;
    Real distance;
};

```

There's a lot to discuss here, so let's begin at the beginning. The constructor takes two bodies to constrain and the required distance between them. Notice the three integers that get passed to the superclass constructor. Those are the numbers of holonomic, non-holonomic, and acceleration-only constraints defined by this class. We are creating a single holonomic constraint, so we pass 1, 0, 0. (Don't confuse that with the number of constraint equations. A single holonomic constraint involves three constraint equations.)

The constructor calls `addConstrainedBody()` to register the `MobilizedBodies` the constraint acts on. This is an important optimization, since the cost of enforcing a constraint depends on the number of bodies involved (directly or indirectly). By telling `Simbody` what bodies are constrained, you allow it to avoid calculations for degrees of freedom that have no effect on the constraint.

This optimization has a very profound impact on how you write constraints. If you look at the example above, you will notice that none of the calculation routines ever reference a `MobilizedBody` object, a `MultibodySystem`, or a `SimbodyMatterSubsystem`. Instead, `Constraint::Custom::Implementation` defines its own methods that you use instead, like `getBodyOriginLocation()` and `getBodyVelocity()`. These methods refer to bodies with a `ConstrainedBodyIndex`, not a `MobilizedBodyIndex`.

After you call `addConstrainedBody()` to register all of the constrained bodies, `Simbody` identifies an “ancestor body”, which is the nearest common ancestor shared by all the constrained bodies. This allows it to define a “constrained system”, consisting of the constrained bodies and all of their parents going back to the ancestor body. Often this will only be a small fraction of the bodies in the full `System`, but all the other bodies are guaranteed to have no impact on whether the constraint is satisfied. This can save a huge amount of computation. When you call `getBodyOriginLocation()`, it actually returns the location in the ancestor body’s reference frame, *not* in the ground frame.

Now look at the methods that implement the constraint. This is a holonomic constraint, so it involves three constraint equations. There is a virtual method corresponding to each one: `realizePositionErrors()`, `realizePositionDotErrors()`, and `realizePositionDotDotErrors()`. Each one calculates the error in the appropriate constraint equation. In this case, the position level equation is $|\mathbf{r}| - d = 0$, the velocity level equation is $\mathbf{v} \cdot \mathbf{r} = 0$, and the acceleration level equation is $\mathbf{a} \cdot \mathbf{r} + \mathbf{v} \cdot \mathbf{v} = 0$. (The acceleration equation is obtained by simply taking the time derivative of the velocity equation.)

In addition to calculating the constraint errors, we also need to calculate the constraint forces that should be applied at each time step to maintain the constraint. `Simbody`

automatically calculates the Lagrange multipliers corresponding to each constraint and passes them to `applyPositionConstraintForces()`. We use them to calculate the force to apply to each body. (If you aren't familiar with Lagrange multipliers, they are a little beyond the scope of this document, but you can easily find descriptions of them online.) We call `addInStationForce()`, which is a convenience method to apply a linear force at a specific point on a specific body. It works out the correct force and torque to apply, and adds them to the Vector of SpatialVecs.

Creating a non-holonomic or acceleration-only constraint is very similar. For a non-holonomic constraint, there are three virtual methods to implement: `realizeVelocityErrors()`, `realizeVelocityDotErrors()`, and `applyVelocityConstraintForces()`. For an acceleration-only constraint, there are two methods: `realizeAccelerationErrors()` and `applyAccelerationConstraintForces()`. `Constraint::Custom::Implementation` also has methods corresponding to each of the standard stages (`realizeTopology()`, `realizeModel()`, etc.), which can be overridden exactly as in a `Subsystem`. This allows you to define custom state variables and cache entries, which is useful if you want your constraint to depend on adjustable parameters.

3.3 Simple Constraints

Some constraints really are simple. Sometimes it is easy to write a function of the state variables that you want to constrain. In these cases, `Simbody` provides special classes that let you implement new constraints with very little work.

There are three special cases for which `Simbody` offers simple constraint classes. The first is a holonomic constraint that can be written as a simple function of the generalized coordinates: $c(\mathbf{q}) = 0$. This class is called `Constraint::CoordinateCoupler`, since it defines a coupling between some set of coordinates.

The second is a non-holonomic constraint that can be written as a simple function of the generalized coordinates and generalized speeds: $c(\mathbf{q}, \mathbf{u}) = 0$. This class is called `Constraint::SpeedCoupler`. Although the constraint equation may involve coordinates, it is

strictly a constraint on the speeds. It considers \mathbf{q} to be fixed, and manipulates \mathbf{u} to satisfy the equation given the current value of \mathbf{q} .

The third case is a holonomic constraint that explicitly specifies the behavior of one generalized coordinate as a function of time: $q_i = f(t)$. This class is called `Constraint::PrescribedMotion`, since the motion of one coordinate is explicitly prescribed in advance.

Each of these classes requires you to provide a function of some set of state variables. This is done with the `Function` class. A `Function` object defines an n -valued function of m arguments. That is, it provides a method with the following signature:

```
Vec<N> calcValue(const Vector& x) const
```

`Function` is a templated class, with the number of output values as a template parameter. All of the `Constraint` classes require a `Function<1>`, so `calcValue()` will return a `Vec1`.

Suppose we want a constraint that requires two generalized coordinates to always be equal to each other. This is done with a `Constraint::CoordinateCoupler` that enforces $c(\mathbf{q}) = q_1 - q_2 = 0$. Here is a `Function` class that implements $c(\mathbf{q})$:

```
class ConstraintFunction : public Function<1> {
    Vec1 calcValue(const Vector& x) const {
        return Vec1(x[0]-x[1]);
    }
    Vec1 calcDerivative(const std::vector<int>& derivComponents,
        const Vector& x) const {
        if (derivComponents.size() == 1)
            return Vec1(derivComponents[0] == 0 ? 1 : -1);
        return Vec1(0);
    }
    int getArgumentSize() const {
        return 2;
    }
    int getMaxDerivativeOrder() const {
        return std::numeric_limits<int>::max();
    }
};
```

The implementation of `calcValue()` is simple: it just returns the difference between its two arguments. Since `Function` is templated only on the number of output values, not the number of input arguments, we also must implement `getArgumentSize()` to return the expected number of input arguments (2 in this case).

A `Function` class also must implement `calcDerivative()` to calculate the partial derivatives of the function. This takes a `std::vector<int>`, which lists all arguments with respect to which to take the derivative. If the vector is of length 1 (that is, a first derivative), we return either 1 or -1, depending on whether a derivative with respect to the first or second argument is requested. If the vector length is greater than 1 (a second derivative or higher), we return 0.

A `Function` need not calculate all possible derivatives, since usually only the first few orders are required. It just needs to implement `getMaxDerivativeOrder()` to report the highest order derivative it can calculate. In this example there is no limit to which ones we can calculate (all derivatives higher than first order are 0), so we return the maximum possible integer value. The `Functions` used for constraints must support derivatives up to second order.

Here is how we add the `Constraint` to a `System`:

```
std::vector<MobilizedBodyIndex> coordBody(2);
std::vector<MobilizerQIndex> coordIndex(2);
coordBody[0] = body1.getMobilizedBodyIndex();
coordBody[1] = body2.getMobilizedBodyIndex();
coordIndex[0] = MobilizerQIndex(0);
coordIndex[1] = MobilizerQIndex(0);
Constraint::CoordinateCoupler constraint(matter, new ConstraintFunction(),
    coordBody, coordIndex);
```

In addition to telling the `CoordinateCoupler` what function to use, we also must tell it which coordinates to pass as arguments. For each coordinate, we specify the `MobilizedBody` it belongs to and the index of that coordinate for the `MobilizedBody`. In this example, we constrain the first coordinate of `body1` to always equal the first coordinate of `body2`.

This example could actually be made even simpler. `SimTK` provides `Function` subclasses for common function types, such as linear functions, polynomials, and splines.

Function::Linear represents a linear function of its arguments. For two arguments, for example, the function is $f(x, y) = Ax + By + C$. You provide the coefficients. We want $(A, B, C) = (1, -1, 0)$, so we create the constraint as follows:

```
Vector_<Vec1> coefficients(3);
coefficients[0] = Vec1(1);
coefficients[1] = Vec1(-1);
coefficients[2] = Vec1(0);
Constraint::CoordinateCoupler constraint(matter,
    new Function<1>::Linear(coefficients), coordBody, coordIndex);
```

Now we don't even need to write our own Function subclass! This is a Constraint that truly is easy to implement.

3.4 A Custom MobilizedBody (The Easy Case)

Simbody also allows you to write new types of MobilizedBodies. There is some good news and some bad news about this. First the bad news: in the general case, writing a custom MobilizedBody is even more difficult than writing a custom Constraint. Now the good news: as with Constraints, Simbody provides a class that lets you implement some, but not all, MobilizedBody types in a fairly easy way.

This time we'll start with the easy case. MobilizedBody::FunctionBased allows you to create new MobilizedBodies that have the following properties:

1. There is a fixed number of generalized coordinates.
2. There is a one-to-one correspondence between generalized coordinates and generalized speeds. That is, $d\mathbf{q}/dt = \mathbf{u}$.
3. The motion of the body can be described with six functions of the generalized coordinates, where three of them return translations along fixed axes, and the other three return rotation angles around fixed axes.

As an example, let's create a MobilizedBody with two generalized coordinates: a translation along X and a rotation around Y. I'm not sure exactly what a joint like that would be useful for, but I'll leave that up to you to decide! Let's see how to implement it.

First, we need six Functions. Two of them will be linear functions (for the X translation and Y rotation), and the other four will be constant functions that always return 0.

```
Vector_<Vec1> coefficients(2);
coefficients[0] = Vec1(1);
coefficients[1] = Vec1(0);
std::vector<const Function<1*>> functions(6);
functions[0] = new Function<1>::Constant(Vec1(0), 0);
functions[1] = new Function<1>::Linear(coefficients);
functions[2] = new Function<1>::Constant(Vec1(0), 0);
functions[3] = new Function<1>::Linear(coefficients);
functions[4] = new Function<1>::Constant(Vec1(0), 0);
functions[5] = new Function<1>::Constant(Vec1(0), 0);
```

Notice that the linear functions are the second and fourth entries in the array. The first three functions return the rotation angles and the last three return the translations. By default, the axes are X, Y, and Z respectively, but you can modify them to have translations along arbitrary directions and rotations around arbitrary axes.

Next we need to tell Simbody what coordinates to pass to each function. We want to pass the first generalize coordinate to function 1, the second one to function 3, and no coordinates at all to the other functions. We create a `std::vector<int>` for each function listing the coordinates to pass to it:

```
std::vector<std::vector<int>> coordIndices(6);
coordIndices[1].push_back(0);
coordIndices[3].push_back(1);
```

Now we can go ahead and create our custom MobilizedBody:

```
MobilizedBody::FunctionBased customBody(parent, body, 2, functions, coordIndices);
```

There are several other constructors that let you specify other options, such as the axes to use and the inboard and outboard transforms.

A common use of `MobilizedBody::FunctionBased` is to create joints based on experimental data. Consider a human knee, for example. It has only one degree of freedom, but it rotates

and translates in a complex way as a function of that degree of freedom. You can simply take experimental data describing the motion of a real knee, fit a set of splines to that data, and then use those splines to define a custom MobilizedBody.

3.5 A Custom MobilizedBody (The Hard Case)

Many MobilizedBodies can be implemented with MobilizedBody::FunctionBased, but not all. For example, if you want to represent a rotation with a quaternion, it will not work. Instead, you need to use MobilizedBody::Custom.

Here is a simple example of a custom MobilizedBody. Its behavior is identical to MobilizedBody::Translation: three generalized coordinates which are interpreted as displacements along the X, Y, and Z axes respectively. So it is not actually a useful class, but it makes a good example, since it is one of the simplest of all MobilizedBodies to implement.

```
class CustomTranslation : public MobilizedBody::Custom::Implementation {
public:
    CustomTranslation(SimbodyMatterSubsystem& matter) :
        Implementation(matter, 3, 3, 0) {
    }
    Implementation* clone() const {
        return new CustomTranslation(*this);
    }
    Transform calcMobilizerTransformFromQ(const State& s, int nq,
        const Real* q) const {
        return Transform(Vec3(q[0], q[1], q[2]));
    }
    SpatialVec multiplyByHMatrix(const State& s, int nu,
        const Real* u) const {
        return SpatialVec(Vec3(0), Vec3(u[0], u[1], u[2]));
    }
    void multiplyByHTranspose(const State& s, const SpatialVec& F, int nu,
        Real* f) const {
        Vec3::updAs(f) = F[1];
    }
    SpatialVec multiplyByHDotMatrix(const State& s, int nu,
        const Real* u) const {
        return SpatialVec(Vec3(0), Vec3(0));
    }
    void multiplyByHDotTranspose(const State& s, const SpatialVec& F, int nu,
        Real* f) const {
        Vec3::updAs(f) = Vec3(0);
    }
}
```

```

void setQToFitTransform(const State&, const Transform& X_FM, int nq,
    Real* q) const {
    Vec3::updAs(q) = X_FM.T();
}
void setUToFitVelocity(const State&, const SpatialVec& V_FM, int nu,
    Real* u) const {
    Vec3::updAs(u) = V_FM[1];
}
};

```

We pass three integers to the superclass constructor: the number of generalized speeds (3), the number of generalized coordinates (3), and the number of those coordinates that correspond to rotation angles (0).

The first method we need to implement is `calcMobilizerTransformFromQ()`. This is a straightforward method: given the list of generalized coordinates for this `MobilizedBody`, it calculates the mobilizer transform. In this example we simply take the three coordinates as the X, Y, and Z components of a translation:

```
return Transform(Vec3(q[0], q[1], q[2]));
```

Next comes a group of four related methods: `multiplyByHMatrix()`, `multiplyByHTranspose()`, `multiplyByHDotMatrix()`, and `multiplyByHDotTranspose()`. The H matrix transforms the generalized speeds \mathbf{u} into the spatial velocity introduced by the joint: $\mathbf{v} = \mathbf{H} \cdot \mathbf{u}$. We need to provide methods to multiply by H (to convert \mathbf{u} to \mathbf{v}) and \mathbf{H}^T (to convert \mathbf{v} to \mathbf{u}). We also need to provide methods that multiply by the time derivatives of H and \mathbf{H}^T . In this example H does not change with time, so the latter two routines just return 0 for all components.

Finally, there are two methods that set \mathbf{q} and \mathbf{u} based on a Transform or spatial velocity. These implement the standard `MobilizedBody` methods of the same names.

This `MobilizedBody` has a simple one-to-one relationship between generalized coordinates and generalized speeds, such that $d\mathbf{q}/dt = \mathbf{u}$. If this were not true, we would have to implement three additional methods: `multiplyByQMatrix()`, `multiplyByQTranspose()`, and `multiplyByQDotMatrix()`. The Q matrix transforms the generalized speeds \mathbf{u} into the derivatives of the generalized coordinates: $d\mathbf{q}/dt = \mathbf{Q} \cdot \mathbf{u}$. The default implementations of

these methods assume Q is an identity matrix, so you only need to implement them when that is not true.

Like `Constraints`, `MobilizedBody::Custom::Implementation` has virtual methods you can implement to define new state variables, customize the appearance of the body, and various other things. See the reference documentation for details.