Documents

OpenSim Developer's Guide

OpenSim Release 2.2

November, 2010

OpenSim Developer's Guide

Authors

Scott Delp

Matt DeMers

Ayman Habib

Samuel Hamner

Chand John

Joy Ku

Peter Loan

Jack Middleton

Jeff Reinbolt

Ajay Seth

Michael Sherman

Acknowledgments

Trademarks and Copyright and Permission Notice

Brief Description

OpenSim enables users to create computer models of the musculoskeletal system and create dynamic simulations of movement. Individuals who read this guide will learn advanced programming features of OpenSim through illustrations and exercises. Completing the exercises will enable you to create simulations, analyze simulations, and add new functionality to the software.

Audience

This guide is recommended for those who desire in-depth knowledge of musculoskeletal modeling and simulation with OpenSim and wish to use the Application Programming Interface (API) to create or contribute novel models or algorithms not currently supported by the OpenSim application. Programming experience in C++ will help you complete the exercises.

Table of Contents

1 Getting Started

1.1 Guide Overview

This guide introduces the Application Programming Interface (API) for OpenSim, a freely available software package for musculoskeletal modeling and dynamic simulation of movement. For more information on OpenSim, visit the OpenSim project site at http://simtk.org/home/opensim. The project site provides a forum for users to ask questions and share expertise, as well as many other resources.

This guide summarizes the tools and capabilities of OpenSim available to the C++ programmer. It includes five chapters:

- Chapter 1 describes the prerequisites to build and run the examples in the chapters to follow
- Chapter 2 illustrates how to write a main program that performs a dynamic simulation
- Chapter 3 demonstrates how to write a plug-in to analyze a simulation
- Chapter 4 describes how to add functionality to OpenSim
- Chapter 5 provides some background on SimTK

1.2 Prerequisites for Programming OpenSim and Running the Example Programs

To run the examples provided in this guide, you will need:

- A computer running Windows PC or Mac OSX with Windows BootCamp or VMWare
- Microsoft's Visual Studio (version 2005 or 2008) or Visual C++ 2008 Express Edition (http://www.microsoft.com/express/vc/Default.aspx)
- CMake 2.6 or later (http://www.cmake.org/cmake/resources/software.html)

We also recommend the following tools:

- An XML Editor, for example,
  - notepad++ (http://notepad-plus.sourceforge.net/uk/site.htm)
  - XMLMarker (http://symbolclick.com/download.htm)
- Dependency Walker (http://www.dependencywalker.com/): a third-party, free tool for checking the interdependency between modules (dll, sys, exe, etc.) on Windows. It is useful for troubleshooting installation issues, particularly those related to having multiple versions of OpenSim, SimTK, and/or Visual Studio on the same machine.

1.3   Installing OpenSim

1. To install OpenSim with the OpenSim API, download the self-extracting executable from the download page of OpenSim (go to http://simtk.org/home/opensim and click on "Downloads"). The API is installed by default along with the OpenSim application.

   Due to incompatibility between various versions of Microsoft Visual Studio, you need to download/install the distribution of OpenSim that is consistent with your development environment: either Visual Studio 2005, 2008, or 2008 Express (2008 Express is free and recommended).
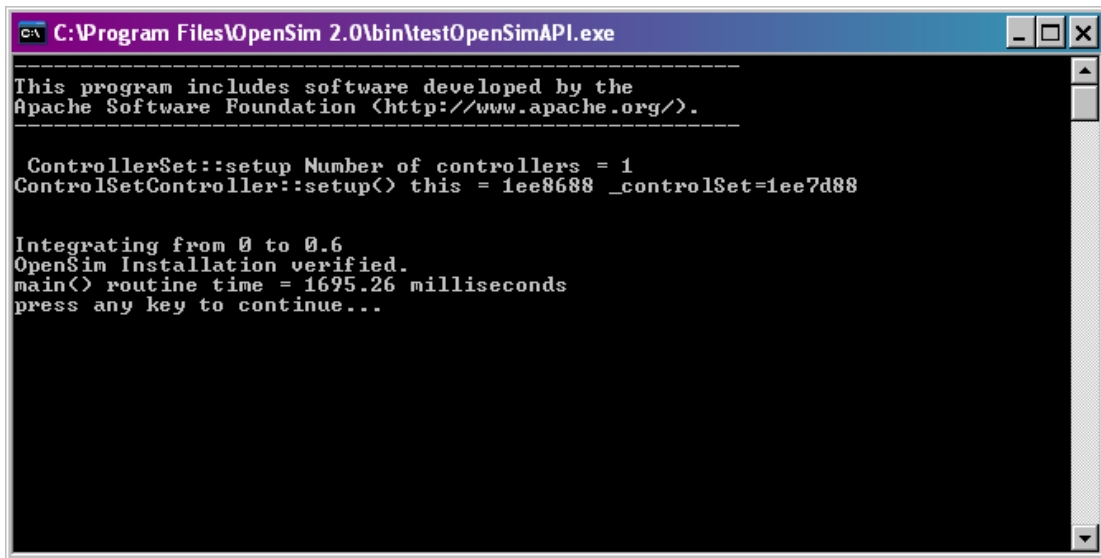
2. Run the executable, following the on-line instructions.
   To be able to run the main programs from the command line (outside Visual Studio), you need to add the OpenSim libraries to your PATH. This can be done during installation by selecting the radio button as illustrated below.
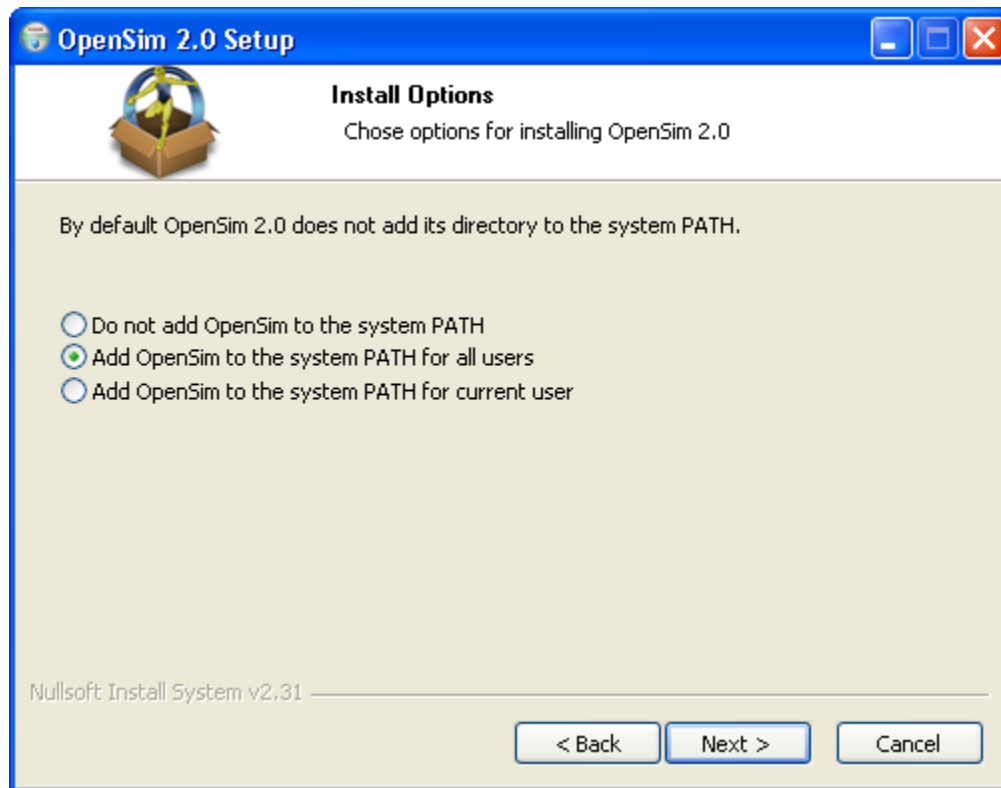
Warning: Earlier installations of OpenSim will continue to be accessible but only through the GUI, which sets its own environment (PATH) variable.

3. Test your installation. Go to the *\bin* directory for the OpenSim installation (if you installed in the default location, the full directory is *C:\ OpenSim 2.x\bin*). Copy the file *testOpenSimAPI.exe* to the *C:\OpenSim 2.x\sdk\APIExamples\ExampleMain* and then double-click on it to run the test. If the OpenSim libraries were added to the PATH while installing then you can just browse for the folder above then type *testOpenSimAPI.exe* at the command prompt. If everything was installed correctly, a window should pop up with message like that shown below:

1.4   Obtaining the Example Programs

The examples come with the OpenSim distribution and are located in the *sdk\APIExamples* directory for the OpenSim installation (if you installed in the default location, the full directory is *C:\OpenSim 2.x\sdk\APIExamples*).

2   Performing a Simulation

2.1   An Example main Program

In this chapter, we will write a main program to perform a forward dynamic simulation using the OpenSim API. We will build it up in pieces, starting from the simplest possible OpenSim model, a single block experiencing the force of gravity. In the end, we will have an OpenSim model with two muscles performing a tug-of-war on the block, with the muscles and ground reactions counteracting the gravitational force. The resulting source code and associated files for this example come with the OpenSim distribution under the directory:

```
C:\OpenSim2.x\sdk\APIExamples\ExampleMain
```

Performing a forward dynamic simulation in OpenSim involves a series of steps. Each of these steps usually requires only a few lines of code. The following sections explain the steps by gradually developing a complete program, stopping at points where the partial program can be compiled, run, and its results visualized.

For your convenience, we have provided the source code as a series of exercise-labeled development snapshots, gradually leading up to the complete program which is called TugOfWar_Complete.cpp. After most steps, we will be using the OpenSim GUI to visualize the model and the motions that result from running the simulations. See the OpenSim User's Guide for details on using the OpenSim GUI. Each of the steps below generates output files with the same names.

2.2   Create an OpenSim Model

To perform a simulation, we first create an OpenSim model and set its name in our main program.

```cpp
#include <OpenSim/OpenSim.h>
using namespace OpenSim;

int main()
{
    try {
        // Create an OpenSim model and set its name
        Model osimModel;
        osimModel.setName("tugOfWar");
    }
    catch (OpenSim::Exception ex) {
        std::cout << ex.getMessage() << std::endl;
        return 1;
    }

    std::cout << "OpenSim example completed successfully.\n";
    return 0;
}
```

Exercise 1: This version of the example is available as *TugOfWar1_CreateModel.cpp*. While the main program above compiles and runs, the OpenSim model it creates is "empty" and no information is saved to a file. Note that you only need to include the header file <OpenSim/OpenSim.h> at the top of the file. Also, note the line:

```cpp
using namespace OpenSim;
```

This line is required to avoid having to prefix every symbol with OpenSim:: since all OpenSim classes live in the namespace OpenSim. Another namespace that will appear later in this guide is SimTK which is utilized by OpenSim for many fundamental classes. Please see Chapter 5 for details.

You should now compile and run the program:

1. Run CMake to generate a Visual Studio solution file. If you installed OpenSim in a location different from the default then you need to change the value of the CMake variable OPENSIM_INSTALL_DIR to point to the actual installation directory. This can be done either in the CMake interface (after you "Configure" and before you "Generate") or alternatively it could be done directly in the CMakeLists.txt file.

2. Launch Visual Studio and load in the solution file that CMake just created.

3. Within Visual Studio, set the Configuration to "RelWithDebInfo" (that is "release with debug information" – unfortunately it won't work in Debug).

4. Compile and run the program. If it is working, it should output "`OpenSim example completed successfully.`" and do nothing else. If it doesn't work, be sure to resolve the problem before attempting to move any further through the exercise.

2.3 Get the Model's Ground Body

A new OpenSim model comes with a ground body. This ground body, however, has no geometry attached to it. After we have created an OpenSim model, we get a reference to the model's ground body. We can then add display geometry to it so we can visualize it in the OpenSim GUI:

```cpp
// Get a reference to the model's ground body
OpenSim::Body& ground = osimModel.getGroundBody();

// Add display geometry to the ground to visualize in
   the GUI
ground.addDisplayGeometry("ground.vtp");
ground.addDisplayGeometry("anchor1.vtp");
ground.addDisplayGeometry("anchor2.vtp");
```

OpenSim allows for files of type *.vtp*, *.stl* and *.obj* as display geometry. At this point we still haven't saved any information to a file, so the model cannot be opened or visualized within the OpenSim GUI. We'll fix that next.

2.4 Save the Model to a File

After we have created a ground body and added its display geometry, we save the model to a file with the ".osim" extension in order to visualize the model we have created.

```cpp
// Save the model to a file
osimModel.print("tugOfWar_model.osim");
```

Exercise 2: After we compile and run the TugOfWar2_AddGround.cpp main program, we can open the model file *tugOfWar_model.osim* in the OpenSim GUI and visualize the ground body (*highlighted with green for this guide*).
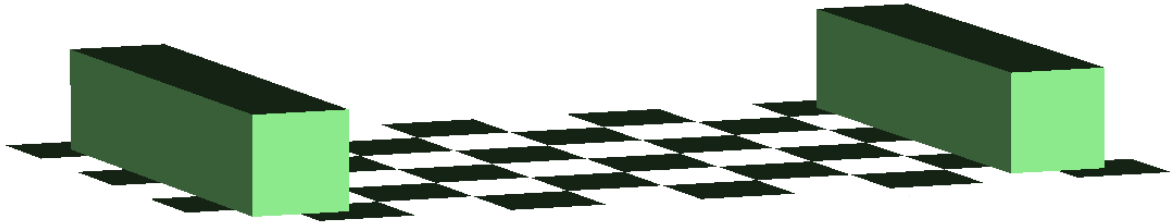
Figure 2.1: Model with only visible ground geometry.

Except for the colors, you should see an image in the GUI like the one above.

2.5   Create a New Block Body

To add an additional body to the OpenSim model, we create a new block body with inertial properties and add display geometry to it.

```cpp
using namespace SimTK;
⋮
            // Specify properties of a 20 kg, 0.1 m^3 block body
            double blockMass = 20.0, blockSideLength = 0.1;
            Vec3 blockMassCenter(0);
            Inertia blockInertia =
                blockMass*Inertia::brick(blockSideLength,
                blockSideLength, blockSideLength);

            // Create a new block body with specified properties
            OpenSim::Body *block = new OpenSim::Body("block",
                blockMass, blockMassCenter, blockInertia);

            // Add display geometry to the block to visualize in the
                GUI
            block->addDisplayGeometry("block.vtp");
```

The classes Vec3 and Inertia live in the namespace SimTK and are explained in Chapter 5. You can write them as SimTK::Vec3 and SimTK::Inertia or include a "using namespace" statement as we did above.

Also, note that the units for mass and length are kilograms and meters, respectively. OpenSim uses the SI convention (length in meters; mass in kilograms; time in seconds; forces in Newtons; and moments/torques are in Newton-meters). Angles can be in degrees or radians; internally, OpenSim uses radians.

Programming Note: *OpenSim model objects in this example are allocated on the heap using "new". Whenever they are added to the model, the model takes ownership of these objects, you shouldn't call "delete" on these objects otherwise the model will be left holding to stale pointers, these objects will be destructed by the model destructor.*

At this point, the block body is not connected to the OpenSim model and cannot be used or visualized in the GUI. To achieve that, the block body has to be connected to the ground body (or any other body already in the model) with a joint. We'll do that next.

2.6   Create a Joint between the Ground and the Block

Before we add the block body to the OpenSim model, we create a new free joint (i.e., 6 degrees-of-freedom) between the block and ground.

```cpp
// Create a new free joint with 6 degrees-of-freedom
   (coordinates) between the block and ground bodies
Vec3 locationInParent(0, blockSideLength/2, 0),
   orientationInParent(0), locationInBody(0),
   orientationInBody(0);
FreeJoint *blockToGround = new
   FreeJoint("blockToGround", ground, locationInParent,
   orientationInParent, *block, locationInBody,
   orientationInBody);

// Get a reference to the coordinate set (6 degrees-of-
   freedom) between the block and ground bodies
CoordinateSet& jointCoordinateSet =
               blockToGround->getCoordinateSet();

// Set the angle and position ranges for the coordinate
   set (SimTK:: prefix not actually needed here)
double angleRange[2] = {-SimTK::Pi/2, SimTK::Pi/2};
double positionRange[2] = {-1, 1};
jointCoordinateSet[0].setRange(angleRange);
jointCoordinateSet[1].setRange(angleRange);
jointCoordinateSet[2].setRange(angleRange);
jointCoordinateSet[3].setRange(positionRange);
jointCoordinateSet[4].setRange(positionRange);
jointCoordinateSet[5].setRange(positionRange);
```

At this point, the block body and corresponding free joint are ready to be added to the OpenSim model.

Although we defined a FreeJoint in this example, different kinds of joints are available, with corresponding constructors:

- WeldJoint

- PinJoint

- SliderJoint

- BallJoint

- EllipsoidJoint

- CustomJoint: This is a more general joint that enables joint motion about/along six spatial axes to be specified as user-supplied functions of joint coordinates.

- Joint: an abstract definition that is sub-classed to create new types of joints (e.g., EllipsoidJoint)

2.7   Add the Block Body to the Model

To finish this step, we simply add the block body to the OpenSim model.

```
// Add the block body to the model
osimModel.addBody(block);
```

Exercise 3: After we compile and run the current main program, we can open the model in the OpenSim GUI (same file name *tugOfWar_model.osim* as in the earlier step) and visualize the ground body (*highlighted with green for this guide*) and the block body (*highlighted with blue for this guide*). You'll also be able to open the "coordinate viewer" within the GUI and interactively change the coordinates. For the FreeJoint, the built-in names of the coordinates are "X-rotation", "Y-rotation", "Z-rotation" followed by the three translations "X-translation", "Y-translation", and "Z-translation". These names, however, can be changed by calling the coordinate's `setName()` method directly.
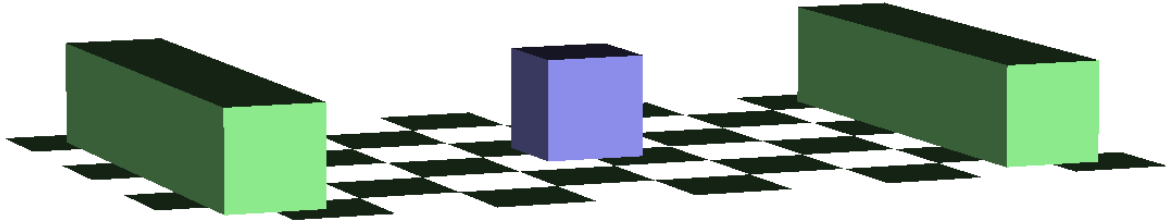
Figure 2.2: Model of a moving free block between two fixed anchors.

Except for the colors, the model in the GUI should look like the image above. Be sure to go to the "coordinates" pane, move the sliders corresponding to the six coordinates, and note the effect that has on the block's position and orientation.

2.8   Define Gravity

In order for the block to actually fall during the simulation, we define the acceleration of gravity to pull the block towards the ground. The actual direction of the vector is arbitrary; however, OpenSim uses the convention that gravity is in the negative Y-direction in the models included with the OpenSim distribution.

```
// Define the acceleration of gravity
osimModel.setGravity(Vec3(0,-9.80665,0));
```

2.9   Initialize the OpenSim Model System and Get the State

An OpenSim model is backed by a SimTK::System (see Chapter 5), which actually performs the computations. As such, the model itself is a stateless object with the state being stored externally in an instance of SimTK::State. To begin simulating the block falling, we initialize the SimTK::System associated with the OpenSim model and create an instance of the system state. **After the call to initSystem(), no changes should be made to the structure of the model.** For example, adding forces or constraints would require the re-creation of the system and a fresh call to initSystem() since these objects may have a state of their own that needs to be incorporated into the system's state.

```
// Initialize the system
SimTK::State& si = osimModel.initSystem();
```

2.10   Define Initial Position and Velocity States of the Block

Next, we define the initial position and velocity of the block. For the free joint, the position coordinates and their velocities are ordered 0 (x-rotation), 1 (y-rotation), 2 (z-rotation), 3 (x-translation), 4 (y-translation), and 5 (z-translation).

```
// Define non-zero (defaults are 0) states for the free joint
CoordinateSet& modelCoordinateSet =
   osimModel.updCoordinateSet();
// set x-translation value
modelCoordinateSet[3].setValue(si, blockSideLength);
// set x-speed value
modelCoordinateSet[3].setSpeedValue(si, 0.1);
// set y-translation value
modelCoordinateSet[4].setValue(si, blockSideLength/2+0.01);
```

2.11   Create the Integrator and Manager for the Simulation

We create the integrator and manager for the simulation in order to perform the numerical integration of the system equations of motion during the forward dynamics simulation. An OpenSim Manager object collects together all the resources need to perform a simulation, including the Model, the numerical methods to be employed, the current State, storage for the trajectory, and runtime options for controlling the simulation.

```
// Create the integrator and manager for the simulation.
SimTK::RungeKuttaMersonIntegrator
   integrator(osimModel.getMultibodySystem());
integrator.setAccuracy(1.0e-4);
Manager manager(osimModel, integrator);
```

2.12   Integrate the System Equations of Motion

We integrate the system equations of motion from the initial time to the final time of the simulation. Depending on your computer speed, this numerical integration could take from a few to several seconds.

```
// Define the initial and final simulation times
double initialTime = 0.0;
double finalTime = 4.0;

// Integrate from initial time to final time
```

```
manager.setInitialTime(initialTime);
manager.setFinalTime(finalTime);
std::cout<<"\n\nIntegrating from "<<initialTime<<" to "
    <<finalTime<<std::endl;
manager.integrate(si);
```

2.13   Save the Simulation Results

After we have performed the integration for the forward dynamics simulation, we save the resulting motion in order to visualize the simulation we have created. Note that OpenSim uses radians internally but degrees are required in a .mot file, so we have to convert to degrees before writing out the .mot file for visualization.

```
// Save the simulation results
Storage statesDegrees(manager.getStateStorage());
statesDegrees.print("tugOfWar_states.sto");
osimModel.updSimbodyEngine().convertRadiansToDegrees(statesDeg
    rees);
statesDegrees.setWriteSIMMHeader(true);
statesDegrees.print("tugOfWar_states_degrees.mot");
```

Exercise 4: After we compile and run the current main program, we can load the model and the motion in the OpenSim GUI and visualize the simulation. (To load the motion, go to File → Load Motion. Select the motion file *tugOfWar_states_degrees.mot* that we just wrote out.)
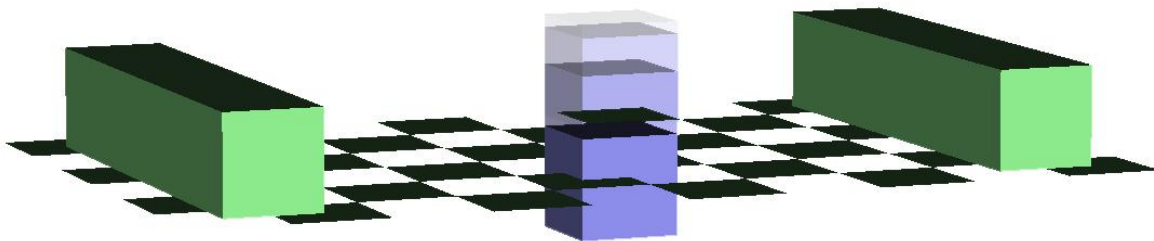
Figure 2.3: Block is falling in the presence of gravity.

Except for the colors, you should see the above in the GUI. Using the motion slider and video controls, visualize the motion. You should see the block falling under gravity.

2.14   Add Two Opposing Muscles

To prevent the block from falling through the ground, we create two opposing muscles between the ground and block. Note that this must be done before the call to `initSystem`, or else the muscles are not included in the simulation.

```cpp
// Create two new muscles with the these parameters
double maxIsometricForce = 1000.0, optimalFiberLength = 0.1,
    tendonSlackLength = 0.2, pennationAngle = 0.0,
    activation = 0.0001, deactivation = 1.0;
// Create new muscle 1 using the Shutte 1993 muscle model
// Note: activation/deactivation parameters are set differently
    between the models.
Thelen2003Muscle *muscle1 = new
    Thelen2003Muscle("muscle1", maxIsometricForce,
        optimalFiberLength, tendonSlackLength, pennationAngle);
muscle1->setActivationTimeConstant(activation);
muscle1->setDeactivationTimeConstant(deactivation);
// Create new muscle 2 using the Thelen 2003 muscle model
Thelen2003Muscle *muscle2 = new
    Thelen2003Muscle("muscle2", maxIsometricForce,
        optimalFiberLength, tendonSlackLength, pennationAngle);
muscle2->setActivationTimeConstant(activation);
muscle2->setDeactivationTimeConstant(deactivation);

// Specify the paths for the two muscles
// Path for muscle 1
muscle1->addNewPathPoint("muscle1-point1", ground, Vec3(0.0,0.05,-
    0.35));
muscle1->addNewPathPoint("muscle1-point2", *block, Vec3(0.0,0.0,-
    0.05));
// Path for muscle 2
muscle2->addNewPathPoint("muscle2-point1", ground,
    Vec3(0.0,0.05,0.35));
muscle2->addNewPathPoint("muscle2-point2", *block,
    Vec3(0.0,0.0,0.05));

// Add the two muscles (as forces) to the model
osimModel.addForce(muscle1);
osimModel.addForce(muscle2);
```

2.15   Prescribe Muscle Controls from Functions

We define the control values for each muscle as a linear function of time defined by the slope of the line and its intercept (value when time=0). We define two linear function one for each muscle in the tug-of-war.

```cpp
// Create a prescribed controller that simply applies controls as
    function of time
PrescribedController *muscleController = new PrescribedController();
```

```cpp
        muscleController->setActuators(osimModel.updActuators());
// Define linear functions for the control values for the two muscles
    Array<double> slopeAndIntercept1(0.0, 2);  // array of 2 doubles
    Array<double> slopeAndIntercept2(0.0, 2);
// muscle1 control has slope of -1 starting 1 at t = 0
    slopeAndIntercept1[0] = -1.0/(finalTime-initialTime);
    slopeAndIntercept1[1] = 1.0;
// muscle2 control has slope of 1 starting 0.05 at t = 0
    slopeAndIntercept2[0] = 1.0/(finalTime-initialTime);
    slopeAndIntercept2[1] = 0.05;

// Set the indiviudal muscle control functions for the prescribed
    muscle controller
    muscleController->prescribeControlForActuator("muscle1", new
        LinearFunction(slopeAndIntercept1));
    muscleController->prescribeControlForActuator("muscle2", new
        LinearFunction(slopeAndIntercept2));
```

2.16   Define the Initial Activation and Fiber Length States

In addition, we define the initial activation and fiber length of each muscle. Once these parameters are set, we initialize the states for each muscle.

```cpp
        // Define the initial states for the two muscles
        // Initial activation correspond to control at time=0
        muscle1->setDefaultActivation(slopeAndIntercept1[1]);
        muscle2->setDefaultActivation(slopeAndIntercept2[1]);
        // Fiber length
        muscle2->setDefaultFiberLength(0.1);
        muscle1->setDefaultFiberLength(0.1);
        // Compute initial conditions for muscles
        osimModel.computeEquilibriumForAuxiliaryStates(si);
```

Exercise 5: After we compile and run the current main program, we can load the motion in the OpenSim GUI (same file name *tugOfWar_states_degrees.mot* as before) and visualize the simulation.
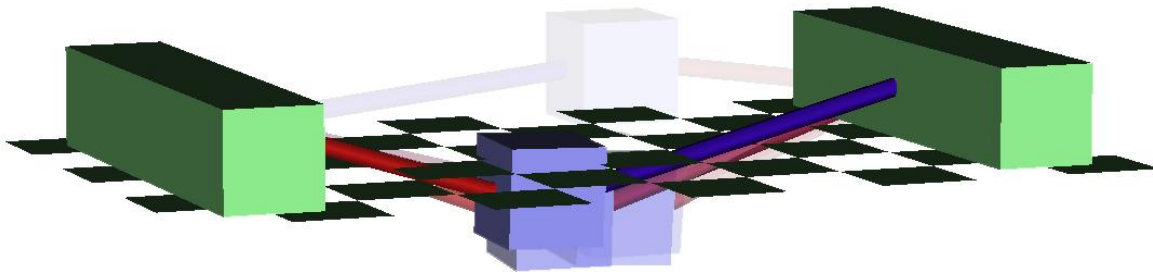
Figure 2.4: Simulation of a falling block suspended by muscles

Except for the colors, you should see something like the above in the GUI. Using the motion slider and video controls, visualize the motion. You should see the block falling under gravity but then restrained by the muscles. Then, you should see the block respond to the controls.

2.17   Add Contact Geometry and Elastic Foundation Force

As you have seen, *display* geometry does not cause contact forces. To prevent the block from penetrating the floor, we create some *contact* geometry and an elastic foundation force between the floor and a cube.

```cpp
// Create new contact geometry for the floor and a cube
// Create new floor contact halfspace
ContactHalfSpace *floor = new ContactHalfSpace(SimTK::Vec3(0),
    SimTK::Vec3(0, 0, -0.5*SimTK::Pi), ground);
floor->setName("floor");
// Create new cube contact mesh
OpenSim::ContactMesh *cube = new
    OpenSim::ContactMesh("blockRemesh192.obj", SimTK::Vec3(0),
    SimTK::Vec3(0), *block);
cube->setName("cube");

// Add contact geometry to the model
osimModel.addContactGeometry(floor);
osimModel.addContactGeometry(cube);

// Create a new elastic foundation force between the floor and
    cube.
OpenSim::ElasticFoundationForce *contactForce = new
    OpenSim::ElasticFoundationForce();
OpenSim::ElasticFoundationForce::ContactParameters
    contactParams;
contactParams.updGeometry().append("cube");
contactParams.updGeometry().append("floor");
contactParams.setStiffness(1.0e8);
contactParams.setDissipation(0.01);
contactParams.setDynamicFriction(0.25);
contactForce->updContactParametersSet().
    append(contactParams);
contactForce->setName("contactForce");

// Add the new elastic foundation force to the model
osimModel.addForce(contactForce);
```

Exercise 6: The complete program up to this point can be found in the file *TugOfWar6_AddContact.cpp*. You can use CMake to generate a new solution file with this as the TARGET, or manually replace the previous source file with this one, from within Visual Studio. Make sure the file *blockRemesh192.obj* is in your working directory.

After we compile and run the current main program, we can load the motion in the OpenSim GUI (same file name *tugOfWar_states_degrees.mot* as before) and visualize the simulation.
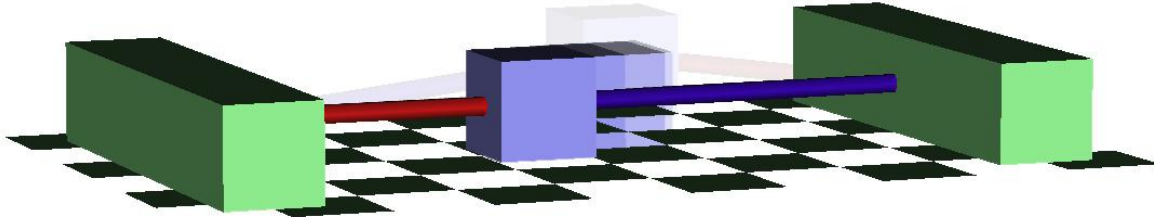
Figure 2.5: Muscle actuated block gliding on a contact surface

Except for the colors, you should see something like the above in the GUI. Using the motion slider and video controls, visualize the motion. You should see that the block no longer falls through the floor but settles there and then responds to the controls.

2.18   Add a Prescribed Force

To push the block during the tug-of-war, we create a prescribed force to apply to the block. The prescribed force is applied in the x-direction in the block body's frame. The point of application varies linearly from (0, 0, 0) to (0.1, 0, 0) during the simulation.

```
// Specify properties of a force function to be applied to the
   block
double time[2] = {0, finalTime}; // time nodes for linear
   function
double fXofT[2] = {0, -blockMass*9.80665*3.0}; // force values
   at t1 and t2
double pXofT[2] = {0, 0.1}; // point in x values at t1 and t2

// Create a new linear functions for the force and point
   components
PiecewiseLinearFunction *forceX = new
   PiecewiseLinearFunction(2, time, fXofT);
PiecewiseLinearFunction *pointX = new
   PiecewiseLinearFunction(2, time, pXofT);

// Create a new prescribed force applied to the block
PrescribedForce *prescribedForce = new PrescribedForce(block);
prescribedForce->setName("prescribedForce");

// Set the force and point functions for the new prescribed
   force
```

```
prescribedForce->setForceFunctions(forceX, new Constant(0.0),
    new Constant(0.0));
prescribedForce->setPointFunctions(pointX, new Constant(0.0),
    new Constant(0.0));

// Add the new prescribed force to the model
osimModel.addForce(prescribedForce);
```

Exercise 7: After we compile and run the current main program, we can load the motion in the OpenSim GUI (same file name *tugOfWar_states_degrees.mot* as before) and visualize the simulation.
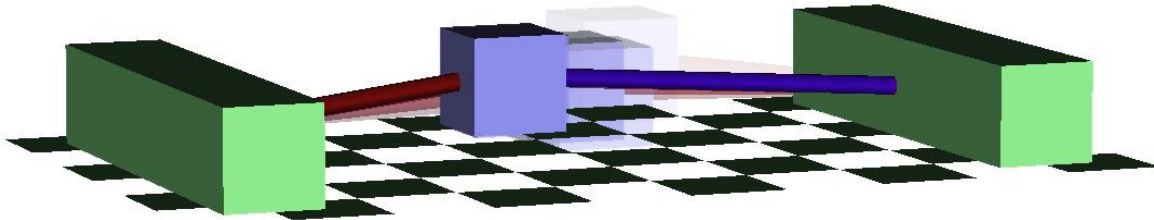
Figure 2.6: Muscle-actuated block with additional perpendicular prescribed force

Except for the colors, you should see something like the above in the GUI. Using the motion slider and video controls, visualize the motion. You should see that the block now responds to the prescribed force as well as the muscle controls.

2.19   Adding a Built-in Analysis

Generally, we would like to report various quantities while running a simulation. In this example, we'd like to report the forces that were applied to the model while running the forward simulation, so that we can troubleshoot the simulation and validate it. To get this effect, we will add in one of the built-in analyses that come with OpenSim. The specific Analysis subclass we will use in this case is ForceReporter. Attaching this analysis to the simulation will cause the values of the forces applied to the model to be reported in a storage file at the end of the simulation.

OpenSim provides a set of Analysis subclasses for convenience, in particular:

- Kinematics

- PointKinematics

- Actuation

- ForceReporter

- InverseDynamics

- StaticOptimization

The last two analyses are more advanced and are used by the corresponding tools in the GUI. To create the analysis for this step requires adding the following lines of code before we integrate the model forward:

```
ForceReporter* reporter = new ForceReporter(&osimModel);
osimModel.addAnalysis(reporter);
```

After the integration is done, we add the line:

```
reporter->getForceStorage().print("tugOfWar_forces.mot");
```

This will create a file with columns corresponding to the forces in the muscles and the applied prescribed forces.

2.20   Add a Constraint

In this section, our goal is to create a constraint such that the motion of the block is along a specified line. The line we specify will be represented by the vector **(1, 0, -1)**, which will constrain the motion of the block on a 45° angle between the two anchor points.

```
// Specify properties of a point on a line constraint to limit
   the block's motion
Vec3 lineDirection(1,0,-1);
Vec3 pointOnLine(1,0,-1);
Vec3 pointOnFollowerBody(0,-0.05,0);

// Create a new point on a line constraint
PointOnLineConstraint *lineConstraint = new
   PointOnLineConstraint(ground, lineDirection, pointOnLine,
   *block, pointOnFollowerBody);

// Add the new point on a line constraint to the model
osimModel.addConstraint(lineConstraint);
```

Exercise 8: This is the final step of this example. The complete program can be found in the file *TugOfWar_Complete.cpp*. You can use CMake to generate a new solution file with this as the TARGET, or manually replace the previous source file with this one, from within Visual Studio.
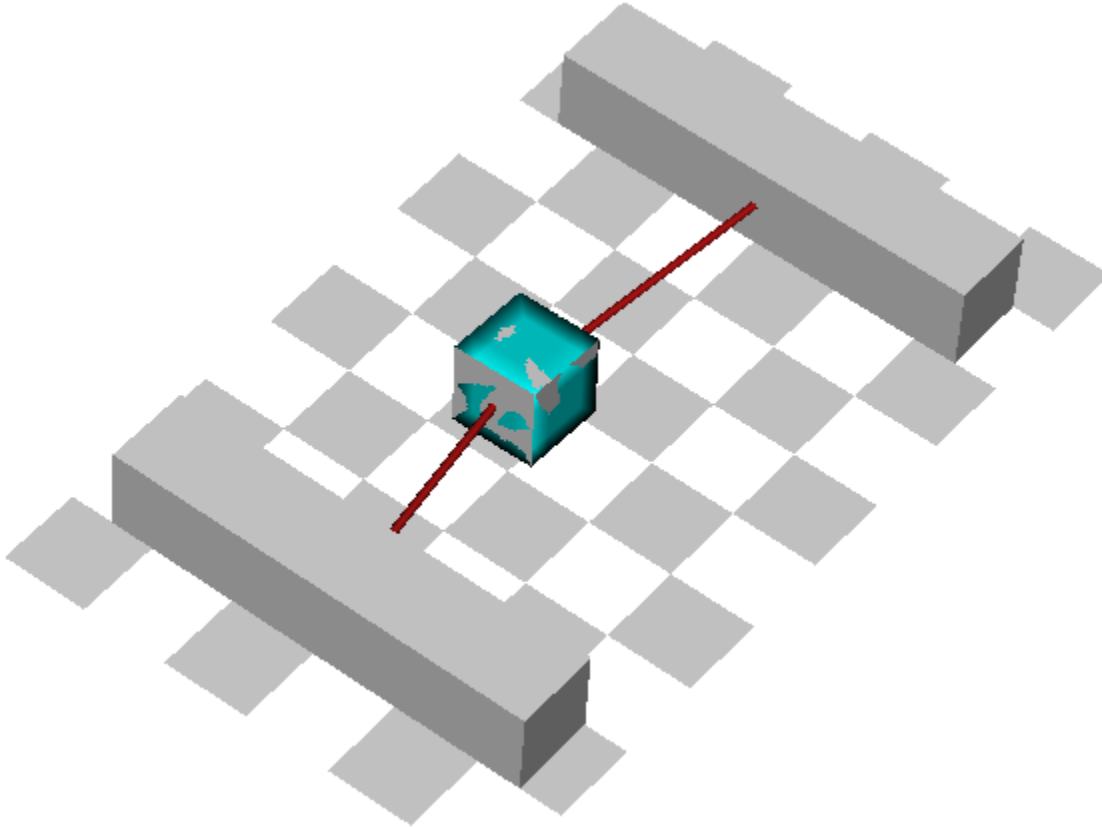
Figure 2.7: Final simulation.

After we re-compile and run the current main program, we can load the new motion file in the OpenSim GUI (same file name *tugOfWar_states_degrees.mot* as before) and visualize the simulation. If you look at the animation from a top view as above, you should see that the motion of the block is now restricted to traveling along a diagonal line.

3   Creating Your Own Analysis

3.1   Setup

In the previous chapter, we created a main program that built a new OpenSim model and performed a forward simulation on it. Another way to utilize the OpenSim API is to use it to create new kinds of objects that are not available in OpenSim.

One particularly common example of such objects is an Analysis. In OpenSim, an Analysis object defines a computation that gets performed repeatedly either during integration of the equations of motion in a forward simulation or during analysis of a trajectory. While OpenSim comes with a set of Analysis objects, users often wish to perform new kinds of computations to support their work/research needs. Instead of trying to anticipate all possible uses, OpenSim allows users to write their own Analysis and attach it to a simulation or to other tools in OpenSim that process trajectories.

When creating an Analysis, you'll have to decide which of two different ways to implement it:

1. ***Dynamic Library Option (Option A):*** In this scenario, your class implementing the OpenSim::Analysis interface will be built into a separate dynamic link library (.dll on Windows). Then, you will be able to load this dynamic library into the GUI or into your main program or have it be used by different tools. This approach allows you to share or distribute your Analysis more easily and to reuse it in your work without recompiling it. The disadvantage is having to define the class that includes properties that permit serialization to/from XML files.

2. ***Main Program Option (Option B)***: In this scenario, you compile your Analysis along with your main program for immediate use. The advantage of this is that you can, optionally, do away with implementing the parts of the Analysis API specific to serialization, so it is faster to write and test. The disadvantage is that you cannot use it in the GUI or embed it in XML files, which are the setup files for most of the

OpenSim tools. Also, you will have to manually point your build system to the Analysis code you are using, rather than link to or load a library that you build once.

Your use case will dictate which approach to take for the long term. In some cases, you may start out using option B, do some testing, and then migrate to option A for the long term.

The OpenSim distribution includes a *templates* directory (if you installed to the default location, the full path is *C:\Program Files\OpenSim 2.0\sdk\templates*) showing the use of option A, as it is the more general use case and is more involved, as well. The distribution also contains the same code built as a plug-in (by default, this is available in *C:\Program Files\OpenSim 2.0\sdk\examples\plugin*). In the example below, we will use the plug-in to build our Analysis.

3.2   Build a Body Position Analysis from the Template

In this example, you will learn how to a build and use an Analysis. The Analysis itself is a simple one that outputs the position of the center of mass of each body in the model.

1. **Prepare a working directory:** Copy the folder containing the Analysis template from the installation folder (by default, this is *C:\Program Files\OpenSim 2.0\sdk\examples\plugin*) to a working folder. In this example, we will assume the working folder is *C:\OpenSimPlugin\plugin*, but the name and path are arbitrary.

2. **Rename   Template.** In   your   working   plug-in   directory   (e.g., *C:\OpenSimPlugin\plugin*), rename the *AnalysisPlugin_Template.h* and *AnalysisPlugin_Template.cpp* files to *MyAnalysis.h* and *MyAnalysis.cpp*, respectively. (Any other name that is unique from the built-in analyses will also be acceptable.) The template analysis simply reports the center-of-mass position of selected bodies.

3. **Run CMake.** Launch CMake. In the dialog box that appears:
    a. For the field "Where is the source code," select the working plug-in directory you just created (e.g., *C:\OpenSimPlugin\plugin*).
    b. Select a directory for "Where to build the binaries." For this example, we will use the folder *C:\OpenSimPlugin\pluginBuild*.

    c. **Populate the "Cache Values" as shown below**. Make sure that the correct installation directory is specified for OPENSIM_INSTALL_DIR, and update it if needed. If you use any directory other than the default for installation, you will also need to update CMAKE_INSTALL_PREFIX to have the same value as OPENSIM_INSTALL_DIR. The PLUGIN_NAME is the name of the dll that will be created by the solution file (in this case myAnalysisPlugin).

    d. Click **Configure** and then **OK**.
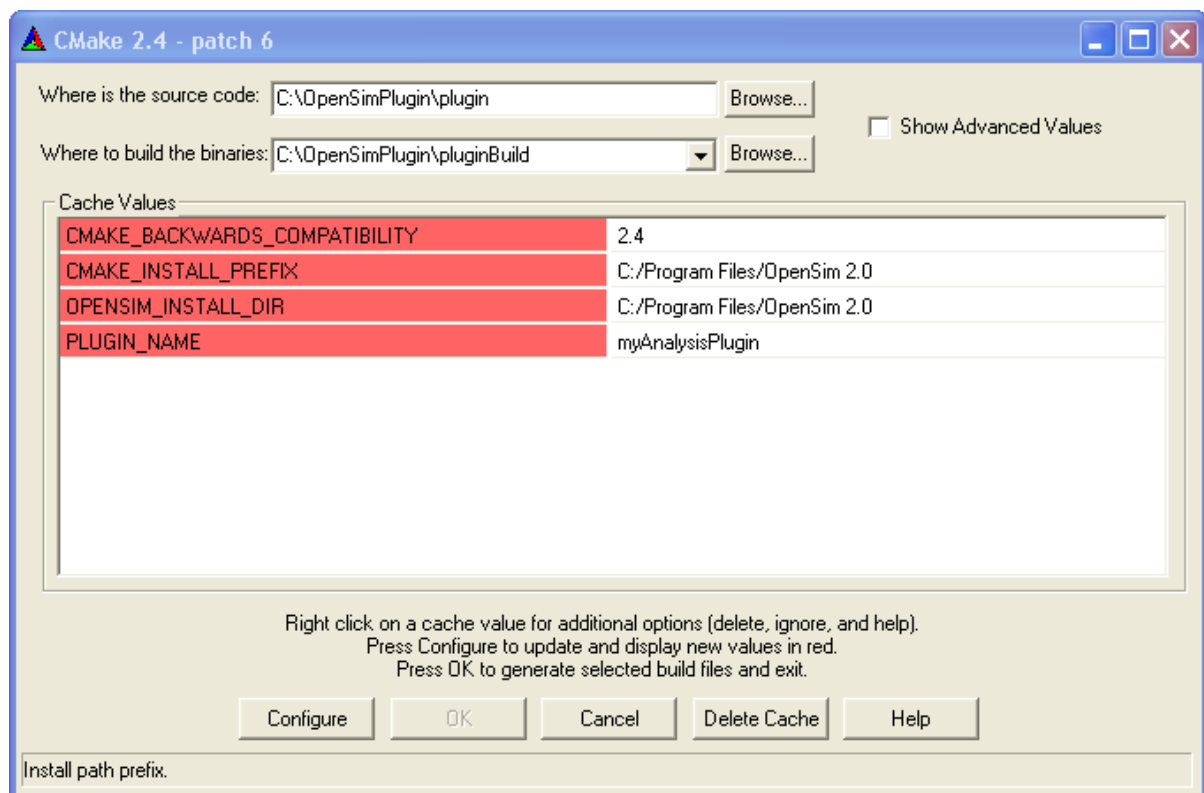
4. **Open the solution file OsimPlugin.sln** (located in whatever directory you instructed CMake to build the binaries). This will **launch Visual Studio**. Do a search and replace (on the entire solution) to replace *AnalysisPlugin_Template* with *MyAnalysis* or whatever name you gave your analysis in Step 2.

5. **Build solution.** Use Visual Studio's "Build" menu to compile your analysis into a dll (plugin). You need to switch from "Debug" to either "Release" or "RelWithDebInfo" if you do not have debuggable OpenSim libraries against which to link.

After building the dll, build the Install project within Visual Studio. It should install the dll in *<OpenSimInstallDir>\plugins*.

3.2.1 Using your Analysis

Option A: The section above demonstrated Option A, creating an analysis that can be built into a dynamic link library. In this case, the dynamic library *myAnalysisPlugin.dll* was created. Below, we examine three of the ways this library can be used:

- **With the OpenSim GUI:** To use your analysis within the OpenSim GUI, place the dynamic library *myAnalysisPlugin.dll* in the *plugins* folder under the OpenSim installation directory. This enables the GUI to load the Analysis, making it accessible to models and available through the GUI menu option (Tools->User Plugins). You can also accomplish this by executing the Install project within Visual Studio.
- **With other OpenSim tools:** To use your Analysis with other OpenSim tools, you need to:
    1. Change the setup file for the tool to include your Analysis in the set of Analyses to be executed. You would use the following XML tags for your new analysis, replacing MyAnalysis with whatever name you actually selected for your class:

```
<MyAnalysis name="">
      <start_time>      0.0 </start_time>
      <end_time>        1.0 </end_time>
      <in_degrees> true </in_degrees>
      <!--Names of the bodies on which to perform the
      analysis.The key word 'All' indicates that the
      analysis should be performed for all bodies.-->
      <body_names> all </body_names>
</MyAnalysis>
```

    2. Run the OpenSim tool from the command line, including the option

```
-L myAnalysisPlugin.dll
```

replacing the name *myAnalysisPlugin.dll* with the name you gave to the dynamic library.

For example, to test your Analysis during an InverseDynamics run of the arm26 model you would do the following:

1. Create a new empty test directory (e.g, *OpenSimPlugin\test*).

2. Copy the files *arm26.osim*, *arm26_InverseKinematics.mot*, and *arm26_Setup_InverseDynamics.xml* from the installation folder (under *examples\Arm26\InverseDynamics*) to the test folder. (Note: if you use a model from an older version of OpenSim, you may see a lot of "illegal tag" messages. You can clean up the model by reading it into the OpenSim GUI and then saving it.)

3. Run InverseDynamics using the command:

```
analyze -S arm26_Setup_InverseDynamics.xml
```

4. Modify the setup file *arm26_Setup_InverseDynamics.xml* by adding the tags for the analysis you created directly below the `</InverseDynamics>` tag.

5. Save the new setup file as *arm26_myAnalysis_ID.xml*

6. Make sure your dll is either in the PATH or that you have copied it to the test directory. Then, run the tool again from the command line:

```
analyze -S arm26_myAnalysis_ID.xml -L myAnalysisPlugin
```

You should see the line "Loaded library myAnalysisPlugin" printed on stdout. You should also see a new file *arm26__pos.sto* in the *Results* directory that

contains your analysis results (the *.sto* file consists of 6 columns per body, with the columns containing position and orientation relative to ground).

7. Now load the model *arm26.osim* that you used above in the GUI. Then, load the motion *arm26_InverseKinematics.mot*. Plot the quantity r_ulna_radius_hand_Y (from the file *arm26__pos.sto*) and the location of the marker r_radius_styloid_ty from the motion. The results should look like that shown below:
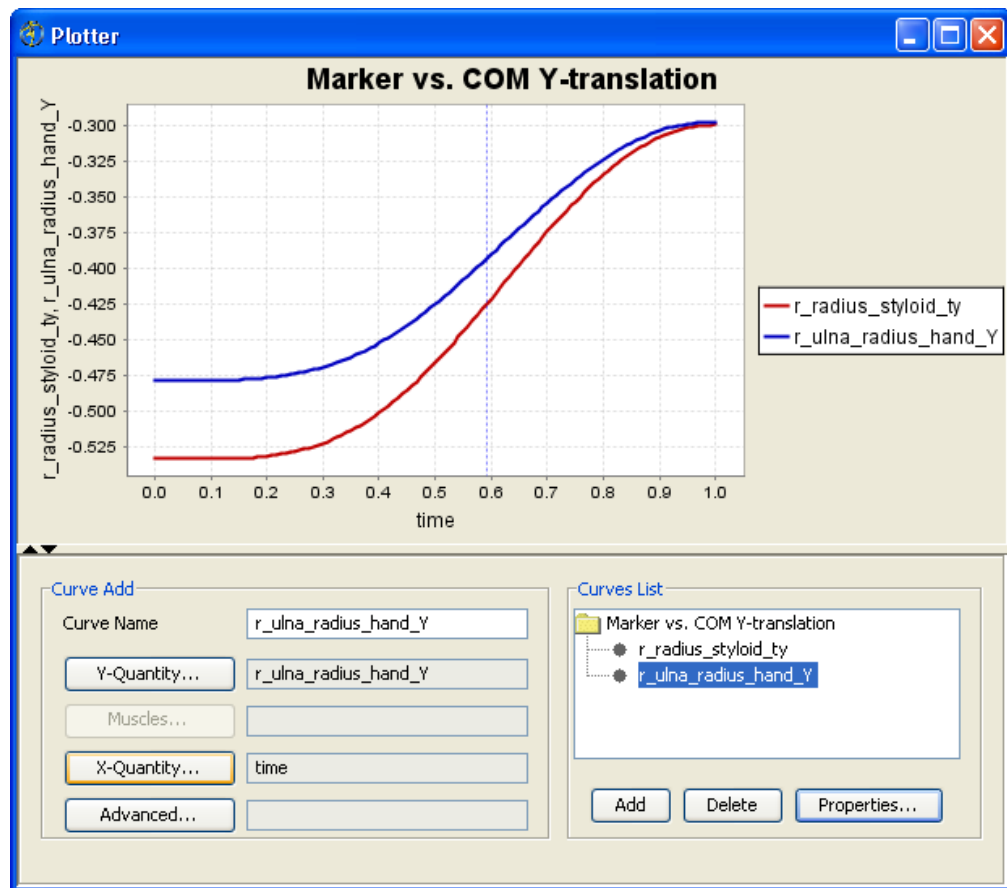
• **With a main program:** If you are writing a main program, you can use your analysis by including the header files and linking to the export library (.lib on Windows).

Option B: For Option B, where you wrote your Analysis as part of your main program, there is no plug-in to build separately. You just compile and link the code for your Analysis along with the rest of the code of your main program. The analysis can be added to the model using the calls:

```
MyAnalysis* comReporter = new MyAnalysis (&osimModel);
osimModel.addAnalysis(comReporter);
```

3.3   Build a Body Position, Velocity, and Acceleration Analysis

The Analysis from the previous section outputs a body's position. We will now extend it to also output the body's velocity and accelerations. Below, we will show you snippets of the existing code that outputs positions. You should search for that code snippet and use it as a model to add the code necessary to output velocities and acceleration. Note that if you want to use the same set of column labels as are defined here for positions, you need to output the same number of columns as that for positions (6 per body).

The following steps outline the procedure:

1. Declare additional output storage and internal working arrays. The number of outputs has changed. Before we had one storage file with position data:

   ```
   /** Storage for recording body positions. */
   Storage _storePos;
   ```

   and one internal working array:

   ```
   /** Internal work array to hold the computed positions. */
   Array<double> _bodypos;
   ```

   Add additional storage for velocities and accelerations and provide the needed working arrays for velocities and accelerations in the .h file.

2. Update the description of the Analysis in `constructDescription()` in the .cpp file.

3. Setup the storage for the velocity and acceleration results, following the example for positions:

```
setupStorage()
{
        // Positions
        _storePos.reset(0);
        _storePos.setName("Positions");
        _storePos.setDescription(getDescription());
        _storePos.setColumnLabels(getColumnLabels());
        …
}
```

4. Correctly size the working arrays :

```
setModel(Model& aModel)
{       …
        int numBodies = aModel.getNumBodies();
        _kin.setSize(6*numBodies);
        …
}
```

5. An Analysis' `record()` method is the heart of the analysis. It collects and, if necessary, computes the data to output the results of an analysis. In this case, the analysis requires adding a calculation (call to the SimbodyEngine) to get the model accelerations.

```
/*
 * Compute and record the results.
 *
 * This method, for the purpose of example, records the position and
 * orientation of each body in the model. You can customize it
 * to perform your analysis.
 *
 * @param aState Current state of the system.
 */


record(const SimTK::State& aState)
{
        …

        // After setting the state of the model and applying forces
        // Compute the derivative of the multibody system (speeds and
        // accelerations)


        // POSITION
        const BodySet& bodySet = _model->getBodySet();
        int numBodies = bodySet.getSize();
        for(int i=0;i<numBodies;i++) {
                const Body& body = bodySet.get(i);
```

```cpp
            SimTK::Vec3 com;
            body.getMassCenter(com);

            // GET POSITIONS AND EULER ANGLES
            _model->getSimbodyEngine ().getPosition(body,com,vec);
            _model->getSimbodyEngine ()
                    .getDirectionCosines(body,dirCos);
            _model->getSimbodyEngine ()
                    .convertDirectionCosinesToAngles(dirCos,
                    &angVec[0],&angVec[1],&angVec[2]);

            // CONVERT TO DEGREES?
            if(getInDegrees()) {
                    angVec *= SimTK_RADIAN_TO_DEGREE;
            }

            // FILL KINEMATICS ARRAY
            int I=6*i;
            memcpy(&_bodypos[I],&vec[0],3*sizeof(double));
            memcpy(&_bodypos[I+3],&angVec[0],3*sizeof(double));
        }
    _storePos.append(aT,_bodypos.getSize(),&_bodypos[0]);

    // VELOCITY
    …
```

Repeat this process for velocities and accelerations. Check the Doxygen documents for calls to the SimbodyEngine to get velocities and accelerations.

6. In `begin()` reset the storage objects at the specified time.

```cpp
// RESET STORAGE
_storePos.reset(aT);
```

7. An analysis is finalized by printing the results out to file:

```cpp
/*
 * Print results.
 *
 * The file names are constructed as
 * aDir + "/" + aBaseName + "_" + ComponentName + aExtension
 *
 * @param aDir Directory in which the results reside.
 * @param aBaseName Base file name.
 * @param aDT Desired time interval between adjacent storage vectors.
 *      Linear interpolation is used to print the data out at the
 *      desired interval.
 * @param aExtension File extension.
 *
 * @return 0 on success, -1 on error.
 */
```

```
printResults(const string &aBaseName,const string &aDir,double aDT,
const string &aExtension)
{
    // POSITIONS
_storePos.scaleTime(_model->getTimeNormConstant());
Storage::printResult(&_storePos,aBaseName+"_"+getName()+"_pos",aDir,
    aDT,aExtension);
// VELOCITIES
…
```

8. Compile and debug in Visual Studio.

9. Build install when satisfied (this will overwrite the previous *osimplugin.dll* if you do not change the name).

10. Repeat the process from the previous section to run and test.

Try plotting both the position and velocity of the COM y coordinate r_ulna_radius_hand_Y. You should see a plot like the one below:
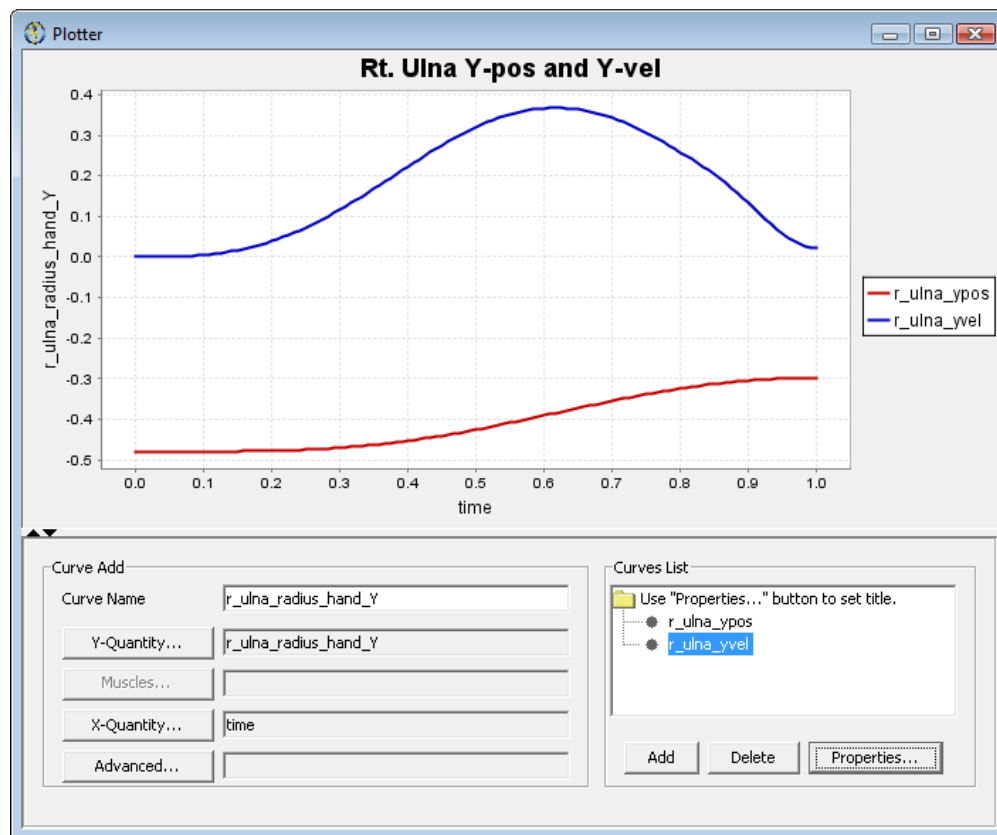
4   Adding New Functionality

4.1   Creating a Controller

In this section, we will add to the tug-of-war example from Chapter 2 by creating a controller that will calculate excitations for the two muscles in the model. The controller we will build computes excitations that naively try to track a desired trajectory of the block in the tug-of-war model. Through this example, you will see how to implement a proportional-derivative (PD) controller and control a model using the OpenSim API.

We will build the example up in pieces. First, we will write a function that returns the desired position of the model in the z-direction (the direction of the line connecting the origins of both muscles in the ground) at a given time. Then, we will implement a PD controller that extends OpenSim's base `Controller` class to calculate excitations for the muscles based on the desired position of the model and the model's current state. Finally, we will connect the controller to the model to run a forward dynamics simulation that attempts to make the model follow the desired trajectory by controlling the model's muscle excitations. We can observe the final trajectory of the model to assess how well this controller is able to reproduce the model's desired motion. The resulting source code and associated files for this example come with the OpenSim 2.0 distribution under the directory:

```
C:\Program Files\OpenSim 2.0\sdk\APIExamples\ControllerExample
```

The following sections will describe the blocks of code needed to implement this example.

4.1.1   Defining the desired trajectory of the model

The desired trajectory for the model is a sinusoid that starts out exactly halfway in-between the left and right walls (at the origin $z = 0$), moves toward the right wall (to $z = +0.15$ m) then toward the left wall (to $z = -0.15$ m), and then moves back to the starting position. The whole motion will last from $t = 0$ seconds to $t = 2$ seconds. The equation for the z-coordinate of the model to follow this motion is $z(t) = 0.15 \sin(2\pi f t)$, where the frequency is $f = \frac{1}{2}$ Hz,

which simplifies to $z(t) = 0.15 \sin(\pi t)$. We will keep the desired values for all other coordinates of the model at zero.

To implement this desired trajectory, we will write a function in our *ControllerExample.cpp* file that calculates the value of $z(t)$ given a value of t, according to the equation above.

```
double desiredModelZPosition( double t ) {
    // z(t) = 0.15 sin( pi * t )
    return 0.15 * sin( Pi * t );
}
```

The velocity of the model's *z*-coordinate is just the time-derivative of its position: $z'(t) = 0.15\pi \cos(\pi t)$. We will implement this as a function as well.

```
double desiredModelZVelocity( double t ) {
    // z'(t) = (0.15*pi) cos( pi * t )
    return 0.15 * Pi * cos( Pi * t );
}
```

The velocities of all of the other coordinates in the model shall be set to zero (the derivative of their position values, which are also zero). A function implementing the desired acceleration is written similarly (see the *ControllerExample.cpp* file).

4.1.2 Designing a controller to track a desired trajectory

We will design a controller that computes control values (excitations) for the model's two muscles in an effort to make the model follow the desired trajectory we implemented above. The controller will be a *proportional-derivative (PD) controller*: we will compute excitations based on deviations of the model's current position from its desired position, as well as on deviations of the model's current velocity from its desired velocity.

We will pretend that each of the model's two muscles is an *idealized linear actuator* that instantaneously applies forces with magnitude $F = xF_{opt}$ at both ends of the actuator (directed from each end to the middle of the actuator), given an input excitation value $0 \leq x \leq 1$. F_{opt} may be a different number for each actuator and is a constant indicating the maximum force an actuator can produce when given a control value $x = 1$. We set F_{opt} for

each actuator equal to the maximum isometric force (specified in the model's *.osim* file) for the corresponding muscle. Unlike idealized actuators, muscles have activation and contraction dynamics that transform an input control (excitation) value into a muscle force, and this force production is not an instantaneous process. A model containing idealized actuators instead of muscles that is controlled by a PD controller can instantaneously produce the necessary forces needed to make the model follow a desired trajectory. However, since our model consists of muscle actuators (which cannot instantaneously produce a desired force from a given excitation value) instead of idealized actuators, we expect that the controller we implement will not track the desired trajectory perfectly. But, we are curious to see just how close we can get with a simple controller!

Continuing to pretend that our model contains idealized actuators instead of muscles, we will now implement a PD controller that computes control values that would cause the actuators to produce the forces that would make the model follow the desired motion. At time t, we know the current position $z(t)$ and velocity $z'(t)$ of the model, as well as the desired position $z_{des}(t)$, velocity $z_{des}'(t)$, and acceleration $z_{des}''(t)$ of the model. First, we compute the total desired acceleration:

$$a_{des}(t) = [z_{des}''(t) + k_v[z_{des}'(t) - z'(t)] + k_p[z_{des}(t) - z(t)]],$$

where k_p and k_v are constants called the position and velocity gains, respectively. In dynamics, k_p and k_v represent the "stiffness" (force response due to position change) and "damping" (force response due to velocity change) properties of a system. If k_v is too high, the system will be *overdamped*, so the system will lose energy too quickly and settle too quickly to some equilibrium state. If k_v is too low, the controller will be *underdamped* and the system will oscillate about an equilibrium state without settling to it quickly. It is common practice to choose k_v so that the system is *critically damped*, i.e., the system settles quickly to a state but without oscillating about the equilibrium state. Similarly, a PD controller is considered critically damped if $k_v = 2*\text{sqrt}(k_p)$ for a second-order linear system. Thus, we choose $k_p = 1600$ and $k_v = 80$ in our implementation of this PD controller.

Next, we compute the net desired force on the block in the model:

$$F_{des}(t) = m\, a_{des}(t),$$

where m is the mass of the block. Since muscles only pull and do not push, we will only excite (i.e., send a non-zero control value to) one muscle at a time (we will set the control value of the other muscle to zero). If $F_{des}(t) < 0$, then we want to pull the block to the left, so we will excite the left muscle at time t. If $F_{des}(t) > 0$, then we want to pull the block to the right, so we will excite the right muscle at time t. In any case, the non-zero control value $x(t)$ that we send to a muscle at any time t will be:

$$x = |F_{des}(t)| \,/\, F_{opt},$$

where F_{opt} is the maximum isometric force of the muscle being excited at time t. This equation is the *control law* we will implement for our PD controller below.

4.1.3 Implementing the controller

In this example, we will write a class called `TugOfWarPDController` that implements the PD controller we designed above. To implement our controller with the desired control law, we derive our controller from `CustomController`:

```
class TugOfWarPDController : public CustomController {
public:
    TugOfWarPDController( Model& aModel, double aKp, double aKv ) :
        CustomController( aModel ), kp( aKp ), kv( aKv ) {

        // Read the mass of the block.
        blockMass = aModel.getBodySet().get( "block" ).getMass();
        std::cout << std::endl << "blockMass = " << blockMass
            << std::endl;
    }
```

The constructor above says that when the controller is created, it should have all the properties of its parent `CustomController` (i.e., it knows what model it will be controlling) and set its member variables `kp,` `kv,` and `blockMass` equal to the input values `aKp,` `aKv,` and `aMass,` respectively.

The behavior of the controller is determined by its `computeControl` function, which implements the intended control law. Two arguments are passed into this function: the current state, `s`, of the system and a number, `index`, that refers to the current actuator whose control is being computed. In our model, index 0 refers to the left muscle and index 1 refers to the right muscle. The `computeControl` function computes and returns a control value for the current actuator based on the current state and desired position, velocity, and acceleration:

```cpp
virtual double computeControl( const SimTK::State& s, int index )
const {

    // Get the current time in the simulation.
    double t = s.getTime();

    // Get a pointer to the current muscle whose control is being
    // calculated.
    Muscle* act = dynamic_cast<Muscle*>
        ( &_actuatorSet.get( index ) );

    // Compute the desired position of the block in the tug-of-war
    // model.
    double zdes  = desiredModelZPosition(t);

    // Compute the desired velocity of the block in the tug-of-war
    // model.
    double zdesv = desiredModelZVelocity(t);

    // Compute the desired acceleration of the block in the tug-
    // of-war model.
    double zdesa = desiredModelZAcceleration(t);

    // Get the z translation coordinate in the model.
    const Coordinate& zCoord = _model->getCoordinateSet().
        get( "blockToGround_zTranslation" );

    // Get the current position of the block in the tug-of-war
    // model.
    double z  = zCoord.getValue(s);

    // Get the current velocity of the block in the tug-of-war
    // model.
    double zv = zCoord.getSpeedValue(s);

    // Compute the correction to the desired acceleration arising
    // from the deviation of the block's current position from its
    // desired position (this deviation is the "position error").
    double pErrTerm = kp * ( zdes  - z  );

    // Compute the correction to the desired acceleration arising
    // from the deviation of the block's current velocity from its
```

```cpp
            // desired velocity (this deviation is the "velocity error").
            double vErrTerm = kv * ( zdesv - zv );

            // Compute the total desired acceleration based on the initial
            // desired acceleration plus the correction terms we computed
            // above: the position error term and the velocity error term.
            double desAcc = zdesa + vErrTerm + pErrTerm;

            // Compute the desired force on the block as the mass of the
            // block times the total desired acceleration of the block.
            double desFrc = desAcc * blockMass;

            // Get the maximum isometric force of the current muscle.
            double Fopt = act->getMaxIsometricForce();

            // Now, compute the control value for the current muscle.
            double newControl;

            // If desired force is in direction of current muscle's pull
            // direction, then set the muscle's control based on desired
            // force.  Otherwise, set the current muscle's control to
            // zero.
            if( desFrc < 0 && index == 0 || // index 0: left muscle
                    desFrc > 0 && index == 1 )  // index 1: right muscle
                    newControl = abs( desFrc ) / Fopt;
            else
                    newControl = 0.0;

            // Don't allow the control value to be less than zero.
            if( newControl < 0.0 ) newControl = 0.0;

            // Don't allow the control value to be greater than one.
            if( newControl > 1.0 ) newControl = 1.0;

            // Return the final computed control value for the current
            // muscle.
            return newControl;
    }
```

This function returns a control value based on deviation of the current state (position and velocity of the block) of the system from the desired state (position and velocity of the block). This is an implementation of the control law we described earlier.

Finishing off the definition of the `TugOfWarPDController` class is the declaration of the member variables, `kp,` `kv,` and `blockMass`:

```cpp
private:

    /** Position gain for this controller */
    double kp;
```

```
    /** Velocity gain for this controller */
    double kv;

    /**
     * Mass of the block in the tug-of-war model, used to compute the
     * desired force on the block at each time step in a simulation
     */
    double blockMass;
};
```

4.1.4 Writing the main()

To run a forward dynamics simulation using our controller, we can write a main program (as in Chapter 2). Our main program below initializes the model, attaches a controller to the model, and runs a forward dynamics simulation.

```
int main()
{
    try {

        // Need to load this DLL so muscle types are recognized.
        LoadOpenSimLibrary( "osimActuators" );

        // Create an OpenSim model from the model file provided.
        Model osimModel( "tugOfWar_model_ThelenOnly.osim" );
```

4.1.5 Creating the controller and attaching it to the model

Next in our main program, we create an instance of the TugOfWarPDController, passing it the model we read in as well as values we choose for the position and velocity gains:

```
        // Create the controller.
        TugOfWarPDController *pdController = new
            TugOfWarPDController( osimModel, kp, kv );

        // Add the controller to the Model.
        osimModel.addController( pdController );
```

4.1.6 Initializing the system

Next, we set the initial states of the system, which include the position and speed values for all the coordinates in the model, as well as the muscle activations and fiber lengths:

```cpp
// Initialize the system and get the state representing the
// system.
SimTK::State& si = osimModel.initSystem();

// Define non-zero (defaults are 0) states for the free joint.
CoordinateSet& modelCoordinateSet =
    osimModel.updCoordinateSet();
// Get the z translation coordinate.
Coordinate& zCoord = modelCoordinateSet.
    get( "blockToGround_zTranslation" );
// Set z translation speed value.
zCoord.setSpeedValue( si, 0.15 * Pi );

// Define the initial muscle states.
const Set<Actuator>& actuatorSet = osimModel.getActuators();
Muscle* muscle1 =
    dynamic_cast<Muscle*>( &actuatorSet.get(0) );
Muscle* muscle2 =
    dynamic_cast<Muscle*>( &actuatorSet.get(1) );
muscle1->setDefaultActivation( 0.01 ); // muscle1 activation
muscle1->setDefaultFiberLength( 0.2 ); // muscle1 fiber length
muscle2->setDefaultActivation( 0.01 ); // muscle2 activation
muscle2->setDefaultFiberLength( 0.2 ); // muscle2 fiber length

// Compute initial conditions for muscles.
osimModel.equilibrateMuscles( si );
```

4.1.7 Creating the integrator

We create the integrator for the simulation in order to perform the numerical integration of the system equations of motion during the forward dynamics simulation.

```cpp
// Create the integrator and manager for the simulation.
SimTK::RungeKuttaMersonIntegrator
    integrator( osimModel.getMultibodySystem() );
integrator.setAccuracy( 1.0e-4 );
integrator.setAbsoluteTolerance( 1.0e-4 );
Manager manager( osimModel, osimModel.getMultibodySystem(),
    integrator );
```

4.1.8 Running the simulation

We run the simulation by setting the initial and final times for the integrator and then instructing the manager to integrate starting from the initial state of the system.

```cpp
// Integrate from initial time to final time.
manager.setInitialTime( initialTime );
```

```
manager.setFinalTime( finalTime );
std::cout << "\n\nIntegrating from " << initialTime
        << " to " << finalTime << std::endl;
manager.integrate( si );
```

At this point, you can run the main program. We can then visualize the result of the forward dynamics simulation to see how well the controller was able to make the model follow the desired trajectory. The figure below shows the desired trajectory (magenta) and the actual trajectory (dark blue) taken by the model during the simulation. The vertical axis is the position of the block (*z*-coordinate) and the horizontal axis is time (in seconds):
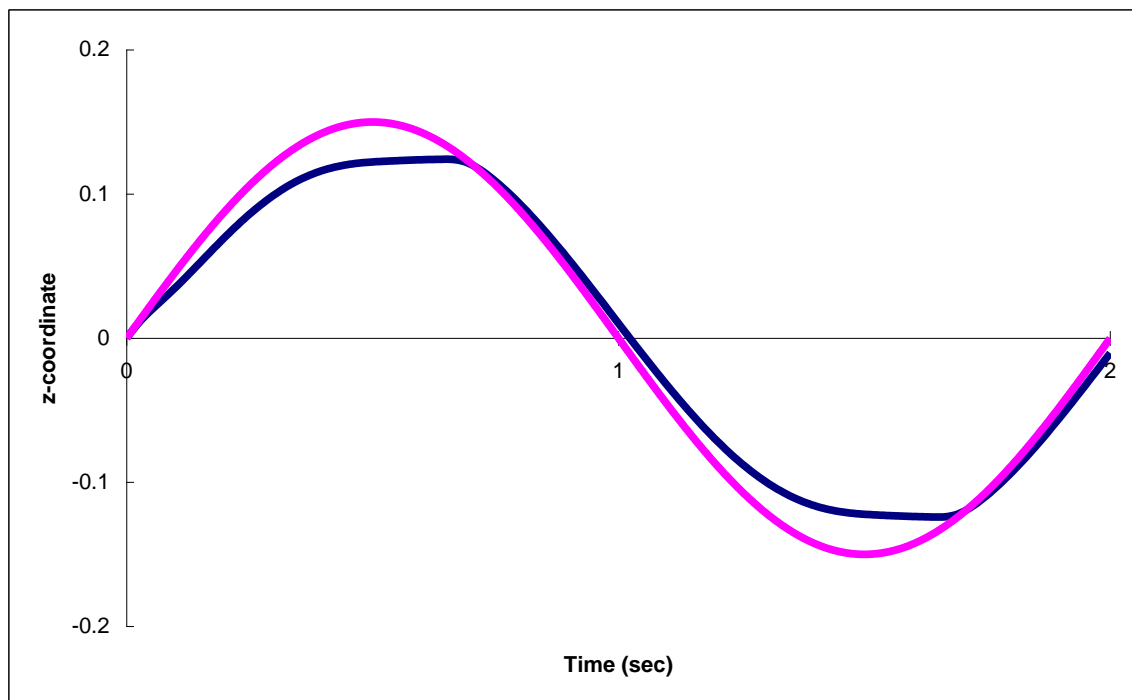
4.2 Creating an Optimization
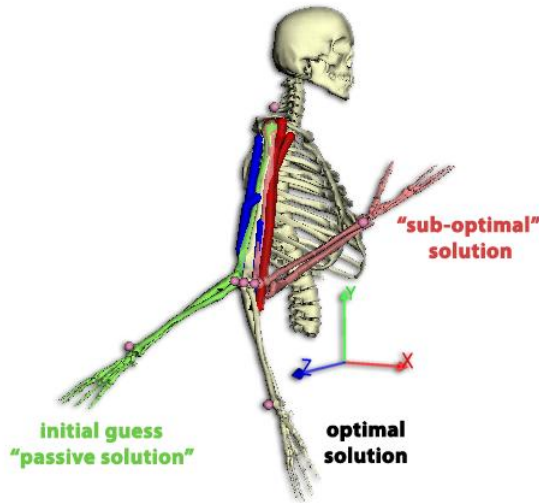
In this section, we will write a main program to perform an optimization study using OpenSim. We will build it up in pieces, starting by programmatically loading an existing OpenSim model. The model will be a simple arm model named *Arm26.osim*, consisting of 2 degrees of freedom and 6 muscles. We will then define an optimization problem that finds a set of muscle controls to **maximize the forward velocity** of the forearm/hand segment mass center. The resulting source code and associated files for this example come with the OpenSim 2.0 distribution under the directory:

```
C:\Program Files\OpenSim 2.0\sdk\APIExamples\OptimizationExample_Arm26
```

As in Chapter 2, the following sections explain the steps to create your own main program. Additionally, we will be extending existing optimizer classes in the OpenSim API. For more information on the OptimizerSystem class, see the SimTKmath User's Guide available on the SimTK project site (https://simtk.org/home/simtkcore under the "Documents" tab).

4.2.1 Extending the *OptimizerSystem* class

Before we get into extending the class, we need to include the proper header files and define a few global variables.

```cpp
#include <OpenSim/OpenSim.h>
using namespace OpenSim;
using namespace SimTK;
int stepCount = 0;

// Global variables to define integration time window, optimizer step
// count, the best solution.
double initialTime = 0.0;
double finalTime = 0.25;
double bestSoFar = Infinity;
```

In OpenSim, optimization problems are set up within an OptimizerSystem, which uses the SimTK-level algorithms to determine a solution. To set up our optimization problem, we need to create our own OptimizerSystem, called `ExampleOptimizationSystem`, by extending the existing base `OptimizerSystem` class.

```
class ExampleOptimizationSystem : public OptimizerSystem {
   public:

   /* Constructor class. Parameters accessed in objectiveFunc() class */
   ExampleOptimizationSystem(int numParameters, State& s, Model& aModel):
      numControls(numParameters), OptimizerSystem(numParameters),
      si(s), osimModel(aModel){}

   /* The objectiveFunc() class will go here. */

   private:
      int numControls;
      State& si;
      Model& osimModel;

};
```

4.2.2 Writing the main()

We can perform an optimization by creating our own main program that will invoke our OptimizerSystem.

```
int main()
{
      try {
            // Create a new OpenSim model
            LoadOpenSimLibrary("osimActuators");
            Model osimModel("Arm26_Optimize.osim");

            /* The guts of your main() will go here */
      }
      catch (std::exception ex)
      {
            std::cout << ex.what() << std::endl;
            return 1;
      }

      // End of main() routine.
      return 0;
}
```

4.2.3 Defining controls and initializing muscle states

As in Chapter 2, we need to define the controls (i.e., muscle excitations), initial activation, and fiber length of each muscle. Moreover, we initialize the states for each muscle after setting the states.

```cpp
// Define the initial and final controls
ControlLinear *control_TRIlong = new ControlLinear();
ControlLinear *control_TRIlat = new ControlLinear();
ControlLinear *control_TRImed = new ControlLinear();
ControlLinear *control_BIClong = new ControlLinear();
ControlLinear *control_BICshort = new ControlLinear();
ControlLinear *control_BRA = new ControlLinear();

/* NOTE: Each contoller must be set to the corresponding
 *       muscle name in the model file. */
control_TRIlong->setName("TRIlong"); control_TRIlat->setName("TRIlat");
control_TRImed->setName("TRImed"); control_BIClong->setName("BIClong");
control_BICshort->setName("BICshort"); control_BRA->setName("BRA");

ControlSet *muscleControls = new ControlSet();
muscleControls->append(control_TRIlong);
muscleControls->append(control_TRIlat);
muscleControls->append(control_TRImed);
muscleControls->append(control_BIClong);
muscleControls->append(control_BICshort);
muscleControls->append(control_BRA);

ControlSetController *muscleController = new ControlSetController();
muscleController->setControlSet(muscleControls);
muscleController->setName("MuscleController");

// add the controller to the model
osimModel.addController(muscleController);

// Initialize the system and get the state
State& si = osimModel.initSystem();

// Define the initial muscle states
const OpenSim::Set<OpenSim::Actuator> &muscleSet =
        osimModel.getActuators();
for(int i=0; i< muscleSet.getSize(); i++ ){
   Actuator* act = &muscleSet.get(i);
   OpenSim::Muscle* mus = dynamic_cast<Muscle*>(act);
   mus->setDefaultActivation(0.5);
   mus->setDefaultFiberLength(0.1);
}

// Make sure the muscles states are in equilibrium
osimModel.equilibrateMuscles(si);
```

4.2.4 Define the optimizer

In SimTK and OpenSim, an Optimizer operates on an OptimizationSystem, which we will initialize as an `ExampleOptimizerSystem`. We then define the bounds for the parameters of the problem, the optimizer tolerance, and the numerical gradient flag before finally invoking the optimizer.

```cpp
// Number of controls will equal the number of muscles in the model
int numControls = 6;

// Initialize the optimizer system we've defined.
ExampleOptimizationSystem sys(numControls, si, osimModel);
Real f = NaN;

/* Define and set bounds for the parameter we will optimize */
Vector lower_bounds(numControls);
Vector upper_bounds(numControls);

for(int i=0;i<numControls;i++) {
   lower_bounds[i] = 0.01;
   upper_bounds[i] = 1.0;
}
sys.setParameterLimits( lower_bounds, upper_bounds );

// set the initial values (guesses) for the controls
Vector controls(numControls);
controls[0]  = 0.01; controls[1]  = 0.01;
controls[2]  = 0.01; controls[3]  = 0.01;
controls[4]  = 0.01; controls[5]  = 0.01;


try {
   // Create an optimizer. Pass in our OptimizerSystem
   // and the name of the optimization algorithm.
   Optimizer opt(sys, SimTK::LBFGSB);

   // Specify settings for the optimizer
   opt.setConvergenceTolerance(0.05);
   opt.useNumericalGradient(true);
   opt.useNumericalJacobian(true);

   // Optimize it!
   f = opt.optimize(controls);
}
catch(const std::exception& e) {
   std::cout << "Caught exception :"  << std::endl;
   std::cout << e.what() << std::endl;
}
```

4.2.5 Writing the objective function

Within `ExampleOptimizationSystem`, we need to define our objective function as a public member of the class. This member function will take the parameters of the muscle controls that we want to vary and will return a real number about the performance we want to optimize (i.e., forward velocity of the hand) , which will then be minimized (*Note: To maximize a value, just multiply it by -1*). In this case, the parameters we want to vary are the muscle control values, and we will return a real number determined by our objective function (i.e., the forearm/hand mass center velocity). Additionally, if we had an analytical gradient or Jacobian function for our system, they could also be defined as member functions of the `ExampleOptimizationSystem`.

```
int objectiveFunc(const Vector &newControls, const bool new_coefficients,
   Real& f ) const {

   // make a copy of out initial states
   :State s = si;

   // Access the controller set of the model and update the control values
   ((ControlSetController *)(&osimModel.updControllerSet()[0]))
      ->updControlSet()->setControlValues(initialTime, &newControls[0]);
   ((ControlSetController *)(&osimModel.updControllerSet()[0]))
      ->updControlSet()->setControlValues(finalTime, &newControls[0]);

   // Create the integrator for the simulation.
   RungeKuttaMersonIntegrator integrator(osimModel.getMultibodySystem());
   integrator.setAccuracy(1.0e-4);

   // Create a manager to run the simulation
   Manager manager(osimModel, osimModel.getMultibodySystem(), integrator);

   // Integrate from initial time to final time
   manager.setInitialTime(initialTime);
   manager.setFinalTime(finalTime);
   osimModel.getMultibodySystem().realize(s, Stage::Acceleration);
   manager.integrate(s);

   /* Calculate the scalar quantity  for the optimizer to minimize
   *  In this case, we're maximizing forward velocity of the
   *  forearm/hand mass center so compute the velocity and
   *  just multiply it by -1.*/
   Vec3 massCenter;
   Vec3 velocity;
   osimModel.getBodySet().get("r_ulna_radius_hand")
      .getMassCenter(massCenter);
   osimModel.getMultibodySystem().realize(s, Stage::Velocity);
   osimModel.getSimbodyEngine().getVelocity(s,
      osimModel.getBodySet().get("r_ulna_radius_hand"),
      massCenter, velocity);
```
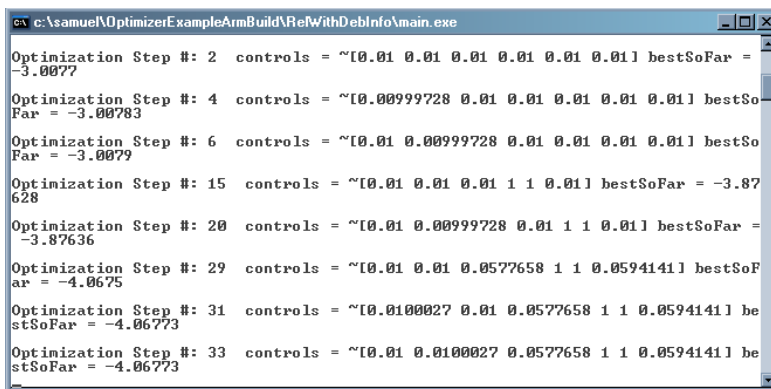
```
f = -velocity[0];
stepCount++;

/* Use an if statement to only store and print the results of an
 *  optimization step if it is better than a previous result.
 */
if( f < bestSoFar){
    Storage statesDegrees(manager.getStateStorage());
    osimModel.updSimbodyEngine().convertRadiansToDegrees(statesDegrees);
    statesDegrees.print("bestSoFar_states_degrees.sto");
    bestSoFar = f;
    std::cout << "\nOptimization Step #: " << stepCount <<
        " controls = " << newControls <<  " bestSoFar = " << f <<
        std::endl;
}

return(0);

}
```


Now you can build and run your main program, and then load the model and results into OpenSim to visualize the optimized control pattern and resulting kinematics.
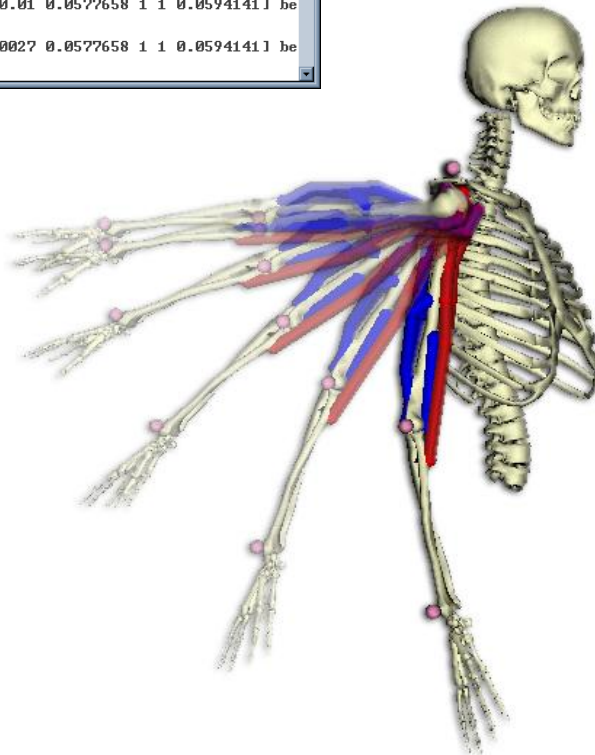
4.3   Creating a Customized Actuator

In this exercise, we will create a specific type of actuator that implements a spring with controllable stiffness. The source code and associated files for this example come with the OpenSim 2.0 distribution under the directory:

```
C:\Program Files\OpenSim 2.0\sdk\APIExamples\CustomActuatorExample
```

When defining a new actuator, you can either start from scratch by deriving from the base class, `CustomActuator`, or if your actuator builds on an existing class, you can derive from that class. In this example we will implement a controllable stiffness spring by deriving from the `PistonActuator` class.

4.3.1 Actuator Overview

We define an actuator as something that produces controllable loads between two bodies. These could be torques applied between two bodies along a common axis, forces applied between two points defined on two different bodies, or some combination of loads applied according to some geometry and state parameters. The key function of any actuator class is to calculate and apply loads to its associated bodies based on its control value and the state variables at any time step.

4.3.1.1   The PistonActuator class

In this exercise we wish to create a spring with controllable stiffness that acts between two points located on different bodies. Instead of building this actuator from the generic, pure virtual class, `CustomActuator`, we will instead derive our new class from the pre-existing `PistonActuator` class. This class is a copy of the `LineActuator` class defined within OpenSim. However, to serve as an example of how we design our actuator classes we have implemented and included the renamed version, `PistonActuator`, within the source material of this example. The figure below illustrates the `PistonActuator` class. This actuator applies a force between two points fixed on two bodies. These bodies do not need to be consecutive bodies in a kinematic chain. This class calculates the magnitude of its force

as the product (optimalForce x control value) and uses the convention that a positive force magnitude acts to increase the distance between points P_A and P_B.
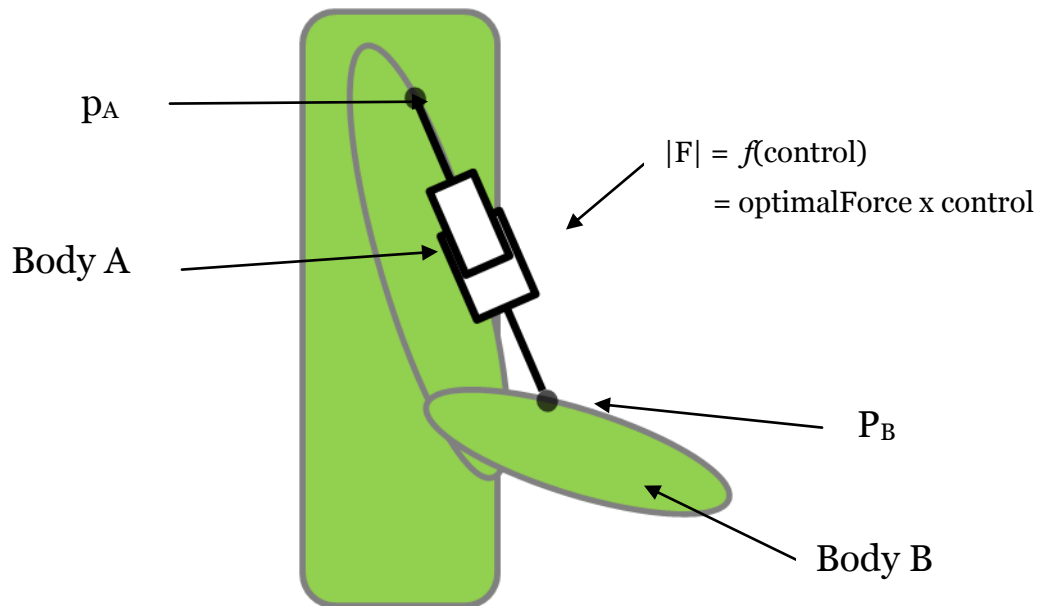
Illustration of the PistonActuator class

4.3.1.2   *The ControllableSpring class*

The figure below illustrates the `ControllableSpring` class that we will define. Just like `PistonActuator`, `ControllableSpring` will act between two points fixed on two different bodies. However, the force magnitude will not simply be calculated as the product of the optimal force and the control value. Instead, this product will represent the spring stiffness: **k = (optimalForce x control value)**. We will also have to define a rest length at which the spring produces no force. The force magnitude will then be calculated as **F = k*(restLength – currentLength).**

K = *f*(control)

= optimalForce x control
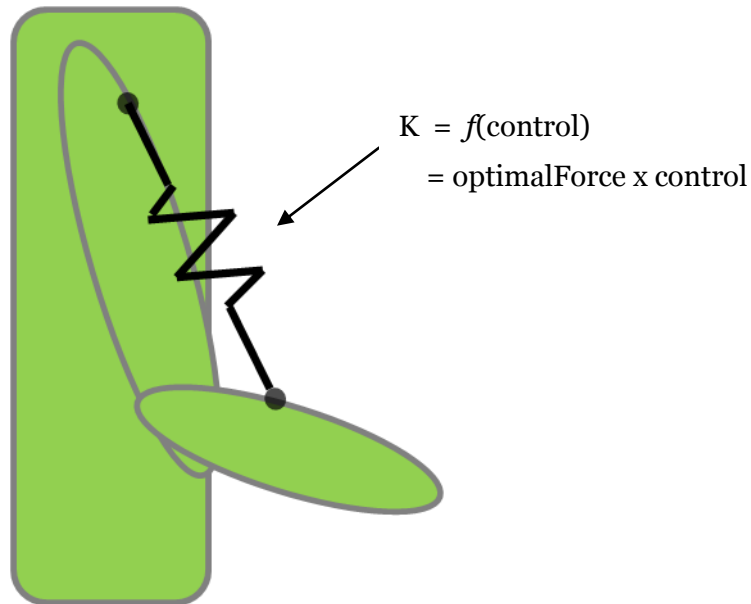
Illustration of the ControllableSpring class to be implemented

4.3.2 Creating Your New Class

4.3.2.1   Setting up a working directory

Before we examine the code, you will need to set up a working directory. This process is very similar to that described in Section 3.2.

1. Launch CMake. Set the */CustomActuatorExample* directory as the source code location, and create any directory you wish for the build location.

2. Click Configure. Be sure to point the OpenSim installation property to the correct location of your OpenSim 2.0 installation folder. By default, the CMAKE_INSTALL_PREFIX (this flag shows up if you set CMake to show "Advanced View") is set to the same directory as your source code. This will ensure that you will not have to move any associated files to visualize your results in the GUI later on.

3. Click Configure again and then click Generate. Then close CMake.

4.3.2.2   Defining the ControllableSpring class (ControllableSpring.h)

The following instructions will outline ALL the steps for defining the `ControllableSpring` class. The `ControllableSpring` class is defined by the file *ControllableSpring.h*. It only contains a partial definition of the class, though. You will need to fill in a few key lines that have been omitted.

At the top of the header file, we include the header for the base class, call the OpenSim namespace, and begin defining the class as a derived class of `PistonActuator`.

```
#include "PistonActuator.h"
namespace OpenSim {

class ControllableSpring : public PistonActuator
{
```

4.3.2.2.1 Defining properties

Our new actuator will have all of the properties of the `PistonActuator` class, plus one more for defining the rest length of the spring.

```
protected:

    /** rest length of the spring */
    PropertyDbl _propRestLength;

    // REFERENCES
    /** rest length */
    double &_restLength;
```

4.3.2.2.2 The constructors

Next we define the constructors. The constructors take the same form as the `PistonActuator` constructors for consistency. Both the constructor and copy constructor call the `setNull` method (to be defined later) which initializes some of the basic elements of the class. The copy constructor also copies the rest length from the existing `ControllableSpring`. The default destructor is used.

```
/* _restLength reference must be initialized in the initialization list */
ControllableSpring( std::string aBodyNameA="", std::string aBodyNameB="")
:
    PistonActuator(aBodyNameA, aBodyNameB),
    _restLength(_propRestLength.getValueDbl())
```

```
{
      setNull();
}
/* The copy constructor must also copy the _restLength since the base
class
** version doesn't know about it. */
ControllableSpring(const ControllableSpring &aControllableSpring) :
      PistonActuator(aControllableSpring),
      _restLength(_propRestLength.getValueDbl())
{
      setNull();
      _restLength = aControllableSpring.getRestLength();
}
/* use the default destructor */
virtual ~ControllableSpring() {};
```

4.3.2.2.3 Setup methods

We will define two private member methods that are used during construction to initialize the ControllableSpring instance. First, setupProperties() is used to set up the properties of the ControllableSpring from values read in from an XML file. The only property added in this class is the rest length.

```
/* define private utilities to be used by the constructors. */
private:
void setupProperties()
{
      _propRestLength.setName("rest_length");
      _propRestLength.setValue(1.0);
      _propRestLength.setComment("The equilibrium length of the spring.");
      _propertySet.append( &_propRestLength);
}
```

Next we define setNull(), which is called when a ControllableSpring object is constructed. It calls setupProperties() and sets some other basic elements of the actuator class, such as its type ("ControllableSpring") and its number of states.

```
void setNull()
{
      setType("ControllableSpring");
      setupProperties();
      setNumStateVariables(0);

}
```

4.3.2.2.4 Get and Set methods

Since the rest length was defined as a private member variable, we must define some public methods to get and set its value.

```
public:
// REST LENGTH
void setRestLength(double aLength) { _restLength = aLength; };
double getRestLength() const { return _restLength; };
```

4.3.2.2.5 computeForce()

The `computeForce()` method is the heart of any actuator class. It is called by OpenSim to calculate and apply any loads associated with the actuator. The `computeForce()` method is defined to be purely virtual in the `CustomActuator` base class, so any derived classes must define its behavior. `PistonActuator` has already defined its own implementation of `computeForce()`, but we will redefine it here so that the `ControllableSpring` actuator behaves like a spring instead of like an ideal actuator. This method begins by checking that the model and bodies are defined.

```
void computeForce(const SimTK::State& s) const
{
      // make sure the model and bodies are instantiated
      if (_model==NULL) return;
      const SimbodyEngine& engine = getModel().getSimbodyEngine();

      if(_bodyA ==NULL || _bodyB ==NULL)
            return;
```

Next, it determines the locations of the application points in both the body and ground frames by doing some transformations. _pointA and _pointB, as well as the bool _pointsAreGlobal, are defined in the `PistonActuator` base class.

```
      /* store _pointA and _pointB positions in the global frame.  If not
      ** alread  in  the  body  frame,  transform _pointA and _pointB into
their
      ** respective body frames. */

      SimTK::Vec3 pointA_inGround, pointB_inGround;

      if (_pointsAreGlobal)
      {
```

```
        pointA_inGround = _pointA;
        pointB_inGround = _pointB;
        engine.transformPosition(s, engine.getGroundBody(), _pointA,
*_bodyA, _pointA);
        engine.transformPosition(s,  engine.getGroundBody(),  _pointB,
*_bodyB, _pointB);
    }
    else
    {
        engine.transformPosition(s,          *_bodyA,          _pointA,
engine.getGroundBody(), pointA_inGround);
        engine.transformPosition(s,          *_bodyB,          _pointB,
engine.getGroundBody(), pointB_inGround);
    }
```

Now we find the vector pointing from point B to point A expressed in the ground frame and then decompose it into its magnitude and direction.

```
    // find the dirrection along which the actuator applies its force
    SimTK::Vec3 r = pointA_inGround - pointB_inGround;
    SimTK::UnitVec3 direction(r);
    double length = sqrt(~r*r);
```

To compute the magnitude of the force, we first must know the spring stiffness. Since we want stiffness to be the product of optimalForce and the control value, we simply use the `computeActuation()` method from the base class, which outputs exactly this calculation.

```
    double stiffness = computeActuation(s);
```

Now we find the magnitude of the force from the stiffness and the deflection of the spring. We then form the force vector.

```
    // find the force magnitude and set it. then form the force vector
    double forceMagnitude = (_restLength - length)*stiffness;
    setForce(s,  forceMagnitude );
    SimTK::Vec3 force = forceMagnitude*direction;
```

The last operation `computeForce()` performs is to apply the equal and opposite point forces to the two bodies.

```
    // appy equal and opposite forces to the bodies
    applyForceToPoint(*_bodyA, _pointA, force);
```

```
        applyForceToPoint(*_bodyB, _pointB, -force);
}
```

4.3.2.2.6 Finish the class definition and close the namespace

```
//=============================================================
};      // END of class ControllableSpring

} //Namespace
//=============================================================
//=============================================================
```

4.3.3 Using the ControllableSpring (toyLeg_example.cpp)

We can now use the `ControllableSpring` class in an example to see its effects. The *toyLeg_example.cpp* file we have provided implements a toy leg model that is driven by a `PistonActuator` (see toyLeg figure below). The model is built up in the sequence ground->linkage1->linkage2->block with pin joints between all the segments. The block is constrained to move only in the vertical direction. A `PistonActuator` called "piston" acts between the distal end of linkage1 and the center of the block. We will modify the `main()` routine to replace the piston actuator with a variable stiffness spring.
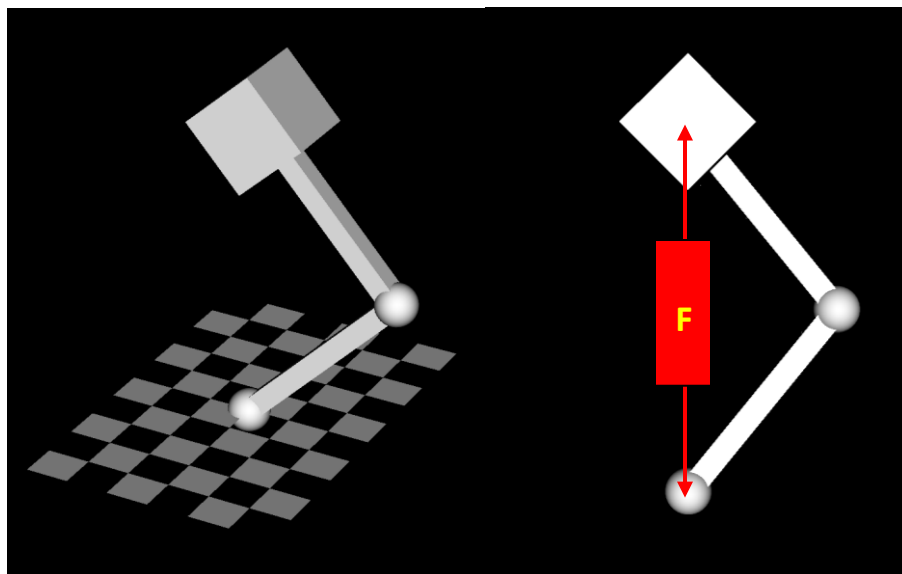
toyLeg example

4.3.3.1   Ready toyLeg_example.cpp to use the ControllableSpring

Open the *toyLeg_example.cpp* file, if you have not already done so. Add the ControllableSpring class to the included files as shown below.

```
#include "PistonActuator.h"
#include "ControllableSpring.h"
#include <OpenSim/OpenSim.h>

using namespace OpenSim;
using namespace SimTK;
```

Within Visual Studios, locate the Actuators_examples project. Right click it and select "Build" in order to rebuild *toyLeg_example.cpp* and force the first build of *ControllableSpring.h*. You will need to switch from "Debug" to either "Release" or "RelWithDebInfo" if you do not have debuggable OpenSim libraries with which to link.

4.3.3.2   Add a ControllableSpring to the model

Find the line after the piston is added to the model. At this location, create a ControllableSpring. Set it up to have the identical geometry as the piston, and add it to the model.

```
osimModel.addForce(piston);
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Add ControllableSpring between the first linkage and the second
block
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ControllableSpring *spring = new ControllableSpring;
spring->setName("spring");
spring->setBodyA(block);
spring->setBodyB(&ground);
spring->setPointA(pointOnBodies);
spring->setPointB(pointOnBodies);
spring->setOptimalForce(2000.0);
spring->setPointsAreGlobal(false);
spring->setRestLength(0.8);

osimModel.addForce(spring);
```

4.3.3.3 Modify the control values given to the actuator

Comment out the line defining the control values for the piston. Below it, add a series of control values that will be applied to the spring.

```
// defing the control values for the piston
//double controlT0[1] = {0.982}, controlTf[1] = {0.978};

// define the control values for the spring
double controlT0[1] = {1.0}, controlT1[1] = {1.0},
          controlT2[1] = {0.25}, controlT3[1] = {.25},
          controlT4[1] = {5};
```

4.3.3.4 Point the controls to the spring

After the definition of control1, modify the `setName` call to apply control1 to the spring instead of the actuator.

```
ControlLinear *control1 = new ControlLinear();
control1->setName("spring");//change this from 'piston' to 'spring'
```

4.3.3.5 Point the control set to the new control values

Comment out the section that sets the controlSet values to the piston controls and then point controlSet to the spring controls you just defined.

```
// set control values for the piston
/*controlSet->setControlValues(t0, controlT0);
controlSet->setControlValues(tf, controlTf);*/

// set control values for the spring
controlSet->setControlValues(t0, controlT0);
controlSet->setControlValues(4.0, controlT1);
controlSet->setControlValues(7.0, controlT2);
controlSet->setControlValues(10.0, controlT3);
controlSet->setControlValues(tf, controlT4);
```

4.3.3.6 Save the resulting motion as a different file

Change the Save Results section in order to print the resulting toyLeg kinematics under a new file name.

```
// Save results

Storage statesDegrees(manager.getStateStorage());
osimModel.updSimbodyEngine().convertRadiansToDegrees(statesDegrees);
//statesDegrees.print("PistonActuatedLeg_states_degrees.mot");
statesDegrees.print("SpringActuatedLeg_states_degrees.mot");
```

4.3.3.7 Build and run the example

1. Build the **Actuator_examples** project again (see Section 4.3.3.1 above).
2. Then build the **INSTALL** project.
3. Make sure that the *<OpenSim2.0_intall_dir>/bin* directory appears at the front of your PATH. To check and/or set your PATH, go to Start -> System Properties (or System). Click on the Advanced tab and then select the Environment Variables button.
4. Navigate to the install directory and run the executable file, *toyLeg_example*. After running the executable, use the GUI to open the model *toyLeg.osim* and load the new motion file (*SpringActuatedLeg_states_degrees.mot*). Upon visualizing the motion, you should see the block oscillate at different magnitudes and frequencies as the spring stiffness is varied over time.

4.4 Creating a Customized Muscle Model

In this section, we will create a muscle model that characterizes fatigue. We will then adapt the example from Chapter 2 to use this new type of muscle model. The resulting source code and associated files for this example come with the OpenSim 2.0 distribution under the directory:

```
C:\Program Files\OpenSim 2.0\sdk\APIExamples\MuscleExample
```

When creating a new muscle model, you can start from scratch by deriving from the base class, Muscle, or you can derive alter an existing muscle model. In this example, we will add the effects of muscle fiber fatigue to Thelen2003Muscle, but we could just as easily do this to Schutte1993Muscle or Delp1990Muscle.

4.4.1 Muscle modeling overview

A muscle is defined by a path and a set of force-generating parameters. The path of a muscle is stored in a GeometryPath object owned by the base class, Muscle. The force-generating parameters are usually different for each type of muscle, so they are stored in the derived muscle classes. A muscle also typically has one or more states (though it can have zero) whose differential equations describe the force, length, and activation behavior of the muscle. Thelen2003Muscle has 2 states.

4.4.2 The header file (LiuThelen2003Muscle.h)

We start defining our new muscle model by creating a header file. The model will be based on Thelen2003Muscle, and will include fatigue effects as defined in the following paper:

> Liu, Jing Z., Brown, Robert, Yue, Guang H., "A Dynamical Model of Muscle Activation, Fatigue, and Recovery," *Biophysical Journal*, Vol. 82, Issue 5, pp. 2344-2359 (2002).

So we will call our model LiuThelen2003Muscle. At the top of *LiuThelen2003Muscle.h*, we include the header file for the base class and derive our new class.

```
#include <OpenSim/Actuators/Thelen2003Muscle.h>

namespace OpenSim {

class LiuThelen2003Muscle : public Thelen2003Muscle
{
```

4.4.2.1   Defining properties

Our new muscle model will have all of the properties of the Thelen2003Muscle model, plus two additional ones to define the rates at which active muscle fibers fatigue and the rate at which fatigued fibers recover.

```cpp
class LiuThelen2003Muscle : public Thelen2003Muscle
{
    protected:
        // the rate at which active muscle fibers become fatigued
        PropertyDbl _fatigueFactorProp;
        double &_fatigueFactor;

        // the rate at which fatigued fibers recover (become active)
        PropertyDbl _recoveryFactorProp;
        double &_recoveryFactor;
```

4.4.2.2   Defining states

Thelen2003Muscle has two states: activation and fiber length. We will add two more states: one for the number of active motor units (range 0.0 to 1.0), and one for the number of fatigued motor units (range 0.0 to 1.0). In *LiuThelen2003Muscle.cpp*, we will set the indices for the new states. In the header file we just define the names.

```cpp
    protected:
            static const int STATE_ACTIVE_MOTOR_UNITS;
            static const int STATE_FATIGUED_MOTOR_UNITS;
```

4.4.2.3   Required functions

The muscle base class (`Muscle`) has a number of pure virtual functions. These functions must be defined in all classes that derive from `Muscle`. To see the complete set of these functions, look for function declarations in *Muscle.h* that end in " = 0;". Thelen2003Muscle defines all of these functions, such as `getFiberLength()` and `computeIsometricForce()`. Because our new class derives from Thelen2003Muscle and not directly from Muscle, we only have to define the base class functions whose behavior we want to change. We will also be adding additional functions to compute the new states for fatigue. The base class functions that we will focus on in this example are:

```cpp
    public:
        virtual void computeEquilibrium(SimTK::State& s ) const;
        virtual double computeActuation(const SimTK::State& s) const;
```

```cpp
    virtual double computeIsometricForce(SimTK::State& s, double act)
const;
    virtual void equilibrate(SimTK::State& state) const;
private:
    void setNull();
    void setupProperties();
```

4.4.3 The source file (LiuThelen2003Muscle.cpp)

We will put the function definitions for our class into *LiuThelen2003Muscle.cpp*. This source file will include all of the required functions that we need to override in Thelen2003Muscle, as well as new functions that describe the behavior of the fatigue states.

4.4.3.1   *Enumerating the states*

At the top of the source file, we define the indices of the muscle states that we are adding to the base class. Thelen2003Muscle defines 0 as the activation state and 1 as the fiber length state. It is important to use 2 and 3 for the states we are adding because **the indices for all of the muscle states must start at 0 and be contiguous.** Once we define the integer indices `STATE_ACTIVE_MOTOR_UNITS` and `STATE_FATIGUED_MOTOR_UNITS`, those names will be used throughout the code to access the state variables, rather than the numbers 2 and 3.

```cpp
// States 0 and 1 are defined in the base class, Thelen2003Muscle.
const int LiuThelen2003Muscle::STATE_ACTIVE_MOTOR_UNITS = 2;
const int LiuThelen2003Muscle::STATE_FATIGUED_MOTOR_UNITS = 3;
```

4.4.3.2   *setNull()*

The function `setNull()` is used to set some of the basic elements of the muscle class, such as its type (name) and its number of states. It is called by OpenSim when an object of this class is constructed.

```cpp
void LiuThelen2003Muscle::setNull()
{
  setType("LiuThelen2003Muscle");

  setNumStateVariables(4);

  _stateVariableSuffixes[STATE_ACTIVE_MOTOR_UNITS]="active_motor_uni
ts";
```

```
    _stateVariableSuffixes[STATE_FATIGUED_MOTOR_UNITS]="fatigued_motor
_units";
}
```

4.4.3.3   setupProperties()

The function `setupProperties()` is used to define the properties of the muscle class and add them to the set of all OpenSim properties. This enables you to define them in an OpenSim model file. This function is called by OpenSim when an object of this class is constructed. As we did in the header file, we need to define a property for the rate at which active muscle fibers fatigue (which we will call *fatigue_factor*) and one for the rate at which fatigued fibers recover (*recovery_factor*). These factors have a default value of 0.0, are assumed to be in the range 0.0 to 1.0, and are normalized (so they are usually the same for all muscles).

```
void LiuThelen2003Muscle::setupProperties()
{
    _fatigueFactorProp.setName("fatigue_factor");
    _fatigueFactorProp.setValue(0.0);
    _fatigueFactorProp.setComment("percentage of active motor units
that
        fatigue in unit time");
    _propertySet.append(&_fatigueFactorProp, "Parameters");

    _recoveryFactorProp.setName("recovery_factor");
    _recoveryFactorProp.setValue(0.0);
    _recoveryFactorProp.setComment("percentage of fatigued motor
units that
        recover in unit time");
    _propertySet.append(&_recoveryFactorProp, "Parameters");
}
```

4.4.3.4   equilibrate() and computeEquilibrium()

The function `equilibrate()` computes values of the muscle states assuming the muscle is in an equilibrium state. It should be called for each muscle before you begin a dynamic simulation. In our muscle class, this function initializes the states to reasonable values, realizes the Simbody model to the velocity stage, and then calls `computeEquilibrium()`. `computeEquilibrium()` gets the current activation of the muscle and calls `computeIsometricForce()`, which is described in Section 4.4.3.6.

```
void LiuThelen2003Muscle::equilibrate(SimTK::State& state) const
```

```
{
    // Reasonable initial activation value
    setActivation(state, 0.01);
    setFiberLength(state, getOptimalFiberLength());
    setActiveMotorUnits(state, 0.0);
    setFatiguedMotorUnits(state, 0.0);
    _model->getMultibodySystem().realize(state,
SimTK::Stage::Velocity);

    // Compute isometric force to get starting value of _fiberLength.
    computeEquilibrium(state);
}

void LiuThelen2003Muscle::computeEquilibrium(SimTK::State& s) const
{
    double force = computeIsometricForce(s, getActivation(s));
}
```

4.4.3.5   *computeActuation()*

computeActuation() computes the values of the muscle states and their derivatives. It is called by the integrator or other code whenever the musculoskeletal model's state has changed and the muscle must be updated accordingly. The code for this function in the LiuThelen2003Muscle class is very similar to the code in Thelen2003Muscle::computeActuation(), but includes calculations of the fatigue states. This function performs four basic steps:

1. Calculate normalized values of the muscle states from the State object passed in. The State object contains the current values and derivatives of the model's coordinates, muscle states, and any other states in the musculoskeletal model. During a forward dynamics simulation, the integrator operates on the state; it uses the current state, their derivatives, and the equations of motion to determine the state at the next time step. So that the same differential equations can be used for every muscle of the same type, the equations are normalized to certain muscle parameters. For example, tendon length is usually normalized by dividing it by the resting length of the tendon for that muscle, thus producing tendon strain. Tendon force is usually normalized by dividing it by the maximum isometric force of the muscle fibers. Once this is done, the same force vs. tendon strain equation can be used for all muscles.

2. Compute normalized derivatives of the muscle states, using the current values of the muscle states and the muscle state equations. For the fatigue and recovery states in our LiuThelen2003Muscle, the equations look like this:

```
normStateDeriv[STATE_ACTIVE_MOTOR_UNITS] =
     normStateDeriv[STATE_ACTIVATION] -
     getFatigueFactor() * getActiveMotorUnits(s) +
     getRecoveryFactor() * getFatiguedMotorUnits(s);
normStateDeriv[STATE_FATIGUED_MOTOR_UNITS] =
     getFatigueFactor() * getActiveMotorUnits(s) -
     getRecoveryFactor() * getFatiguedMotorUnits(s);
```

The first equation means that the rate of change in the percentage of active motor units is equal to the rate of change in the activation level minus the fatigue factor times the number of active motor units, plus the recovery factor times the number of fatigued motor units. The second equation means that the rate of change in the percentage of fatigued motor units is equal to the fatigue factor times the number of active motor units minus the recovery factor times the number of fatigued motor units.

3. Un-normalize the state derivatives and store them in the State object, so they can be accessed by the integrator.

4. Store the muscle force and other state-dependent variables so they can be accessed by the integrator, muscle analysis, or other components of the program.

4.4.3.6   *computeIsometricForce()*

To compute the isometric force in the muscle, we assume that the muscle has reached an equilibrium position in which the fiber velocity is zero, and the fatigue and recovery rates are constant. According to the Liu fatigue model, in this equilibrium position the number of active motor units is independent of the activation level (as long as the activation is greater than some threshold that depends on the fatigue and recovery rates). So we can use the fatigue and recovery rates to calculate the steady-state activation level, and then pass that activation to the `computeIsometricForce` function in the base class to compute the force in the muscle.

```cpp
double LiuThelen2003Muscle::
computeIsometricForce(SimTK::State& s, double aActivation) const
{
    if (_optimalFiberLength < ROUNDOFF_ERROR) {
        return 0.0;
    }

    // This muscle model includes two fatigue states, so this
function
    // assumes that t=infinity in order to compute the [steady-state]
    // isometric force. When t=infinity, the number of active motor
    // units is independent of the activation level (as long as
activation
    // > _recoveryFactor / (_fatigueFactor + _recoveryFactor)). So
the
    // passed-in activation is not used in this function (unless the
fatigue
    // and recovery factors are both zero which means there is no
fatigue).
    if ((_fatigueFactor + _recoveryFactor > 0.0) && (aActivation >=
            _recoveryFactor / (_fatigueFactor + _recoveryFactor))) {
        setActiveMotorUnits(s, _recoveryFactor / (_fatigueFactor +
            _recoveryFactor));
        setFatiguedMotorUnits(s, _fatigueFactor / (_fatigueFactor +
            _recoveryFactor));
    } else {
        setActiveMotorUnits(s, aActivation);
        setFatiguedMotorUnits(s, 0.0);
    }

    aActivation = getActiveMotorUnits(s);

    // Now you can call the base class's function with the steady-
state
    // activation.
    return Thelen2003Muscle::computeIsometricForce(s, aActivation);
}
```

4.4.4 Example program

To illustrate the new muscle model, we will modify the tug-of-war example described in Chapter 2 so that a LiuThelen2003Muscle pulls against a Thelen2003Muscle. We will run two five-second forward dynamics simulations, one with the fatigue and recovery parameters set to zero, and one with them set to abnormally high values, to exaggerate the fatigue effect. We will quantify the fatigue effect by plotting the lengths of the two muscles during the simulations.

4.4.4.1   Modified tug-of-war example

The source file *mainFatigue.cpp* contains the `main()` function from the tug-of-war simulation, with a few modifications so we can use it with our new muscle model. The source file in the *MuscleExample* folder already contains all of the necessary modifications, but here is a description of how it was changed. First, we want to lock the rotational degrees of freedom in the ground->block joint so that the block does not twist as the muscles pull on it (twisting interferes with the straight back-and-forth length changes).

```
// Free joint states
CoordinateSet &coordinates = osimModel.updCoordinateSet();
coordinates[0].setValue(si, 0, true);
coordinates[1].setValue(si, 0, true);
coordinates[2].setValue(si, 0, true);
coordinates[3].setValue(si, 0, true);
coordinates[4].setValue(si, 0, true);
coordinates[5].setValue(si, 0, true);
coordinates[0].setLocked(si, true);
coordinates[1].setLocked(si, true);
coordinates[2].setLocked(si, true);
```

Next, we want to increase the simulation time from 4.0 seconds to 5.0 to better see the fatigue effect.

```
// Define the initial and final simulation times
double initialTime = 0.0;
double finalTime = 5.0;
```

Lastly, we want to change the names of the output files.

```
// Save the simulation results
Storage statesDegrees(manager.getStateStorage());
osimModel.updSimbodyEngine().convertRadiansToDegrees(statesDegrees);
statesDegrees.print("tugOfWar_fatigue_states_degrees.mot");

// Save the OpenSim model to a file
osimModel.print("tugOfWar_fatigue_model.osim");
```

4.4.4.2   Using LiuThelen2003Muscle

The final change we need to make to the original example is to use a LiuThelen2003Muscle instead of a Schutte1993Muscle. We are also going to set the maximum isometric force of the Liu muscle to a value twice that of the Thelen muscle, so that it initially pulls the block

towards its side before fatiguing and letting the Thelen muscle pull it back. After including the header file, *LiuThelen2003Muscle.h*, at the top of *mainFatigue.cpp*, we can modify the code that creates the muscles to look as shown below. Note that the fatigue and recovery factors are both 0.0. We will change them to non-zero values before running the second simulation.

```cpp
// Create two muscles
double maxIsometricForce1 = 4000.0, maxIsometricForce2 = 2000.0,
    optimalFiberLength = 0.2, tendonSlackLength = 0.2;
double pennationAngle = 0.0, activation = 0.0001,
    deactivation = 1.0, fatigueFactor = 0.0, recoveryFactor = 0.0;

// muscle 1 (model with fatigue)
LiuThelen2003Muscle muscle1("Liu", maxIsometricForce1,
    optimalFiberLength, tendonSlackLength, pennationAngle,
    fatigueFactor, recoveryFactor);
muscle1.setActivationTimeConstant(activation);
muscle1.setDeactivationTimeConstant(deactivation);
// muscle 2 (model without fatigue)
Thelen2003Muscle muscle2("Thelen", maxIsometricForce2,
    optimalFiberLength, tendonSlackLength, pennationAngle);
muscle2.setActivationTimeConstant(activation);
muscle2.setDeactivationTimeConstant(deactivation);

// Define the path of the muscles
muscle1.addNewPathPoint("Liu-point1", ground,
    SimTK::Vec3(0.0,0.05,-0.35));
muscle1.addNewPathPoint("Liu-point2", block,
    SimTK::Vec3(0.0,0.0,-0.05));

muscle2.addNewPathPoint("Thelen-point1", ground,
    SimTK::Vec3(0.0,0.05,0.35));
muscle2.addNewPathPoint("Thelen-point2", block,
    SimTK::Vec3(0.0,0.0,0.05));
```

We also want to change the activation controls for the muscles so that they both start inactivated and ramp up to full activation at the same rate.

```cpp
// Define the initial and final controls
double initialControl[2] = {0.0, 0.0};
double finalControl[2] = {1.0, 1.0};
```

4.4.4.3   *Analyzing the fatigue effect*

After compiling and running the simulation, we can load the model into the OpenSim GUI and plot muscle lengths to analyze the fatigue effect. The model that is used by the example program is saved to the file *tugOfWar_fatigue_model.osim*. This model contains a

Thelen2003 muscle and a *LiuThelen2003* muscle. However, because *LiuThelen2003* is a new muscle model that OpenSim does not know about, we cannot load that model into OpenSim without first creating a plug-in containing the *LiuThelen2003Muscle* class. Rather than create a plug-in for this example, we will instead make a small modification to the model file so that it can be loaded into OpenSim. All we need to do is change the XML tag for the muscle named "Liu" from *LiuThelen2003Muscle* to *Thelen2003Muscle*. This changes its type to a Thelen2003 muscle, meaning that it will not include any fatigue effects. But this is OK for our example. The example program still uses the correct muscle types, but the OpenSim GUI will load the modified model. The muscles in this modified model have the same paths as in the original model, so we can still plot muscle-tendon lengths to analyze the fatigue effects. This modified model is provided for you, and is in the file *tugOfWar_fatigue_model_GUI.osim*. So we now want to load the model *tugOfWar_fatigue_model_GUI.osim* into OpenSim along with the motion file with the simulation results, *tugOfWar_fatigue_states_degrees.mot*. Plotting muscle-tendon length for both muscles as a function of the motion should generate a plot like the one below.
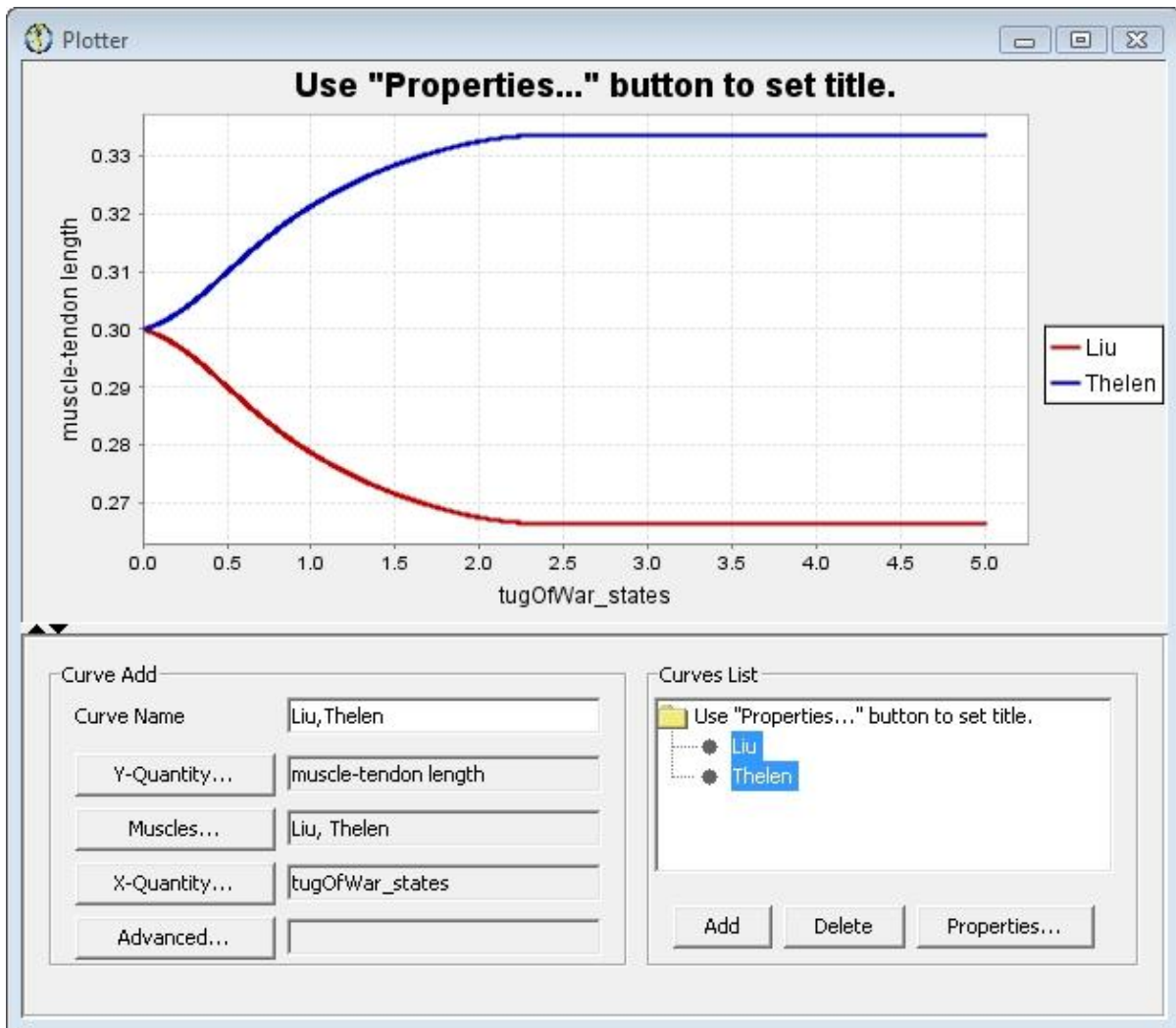
Both muscles start with zero activation at a length of 0.3 meters. Because the Liu muscle is twice as strong as the Thelen muscle, as they both develop force the Liu muscle "wins" the tug-of-war and stretches the Thelen muscle.

We can now change the fatigue and recovery parameters and see what difference this makes in the tug-of-war. Normal values for the parameters are fatigue = 0.02 and recovery = 0.008. But we will exaggerate the effect for this example and use 0.1 and 0.04, respectively.

```
// Create two muscles
double maxIsometricForce1 = 4000.0, maxIsometricForce2 = 2000.0,
    optimalFiberLength = 0.2, tendonSlackLength = 0.2;
double pennationAngle = 0.0, activation = 0.0001,
    deactivation = 1.0, fatigueFactor = 0.1, recoveryFactor = 0.04;
```

After compiling and running a second simulation, we can load the same (overwritten) motion file into the OpenSim GUI and plot the muscle lengths as a function of the new motion, adding the curves to the first plot. The result looks like this:
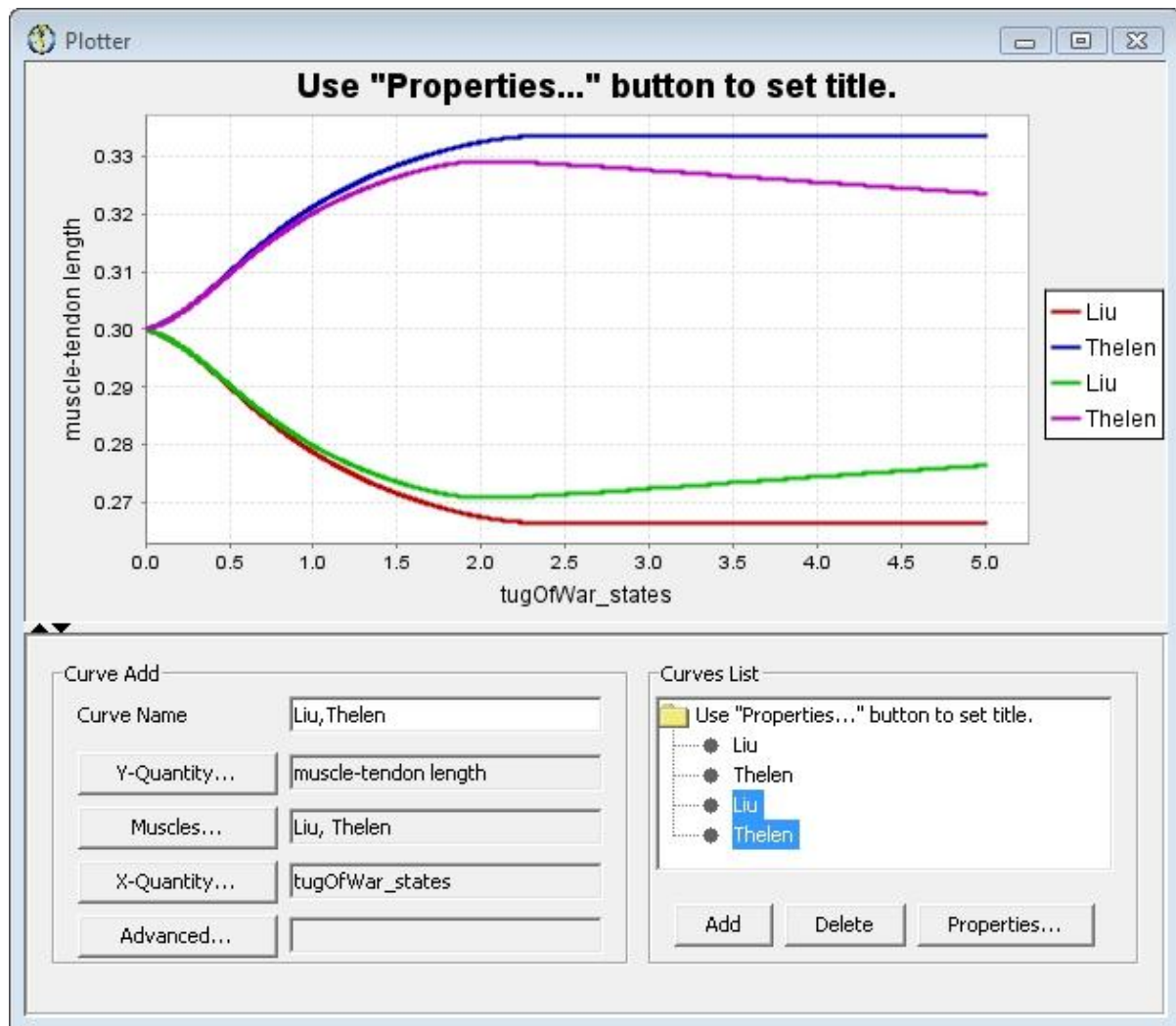
The green and magenta curves show the Liu and Thelen muscle lengths, respectively, during the second simulation. In this case, the Liu muscle is still stronger initially, and stretches the Thelen muscle. But it quickly starts to fatigue, and at about one second, has weakened enough so that the Thelen muscle begins to lengthen it.

5   SimTK Basics

The OpenSim API uses the Simbios "simulation toolkit" SimTK as its low-level, domain-independent computational layer. Some familiarity with SimTK is required to use the OpenSim API because some of the SimTK objects are visible there. This chapter presents a brief introduction to the parts of SimTK that are most commonly used with the OpenSim API; you can find more documentation for SimTK elsewhere (see https://simtk.org/home/simtkcore, Documents tab). The intent, however, is that there is enough material here so that you can use the OpenSim API for many purposes without having to look further.

SimTK provides a broad base of functionality. We will discuss the subset needed for the OpenSim API in four groups:

- Numerical objects (numbers, constants, vectors, matrices)
- Numerical methods (linear algebra, optimization, integration)
- Multibody dynamics (Simbody)
- Simulation (System and State)

Together, the numerical objects and numerical methods provide Matlab-like functionality accessible from C++. We will cover the low-level numerical objects in detail, provide an introduction to the available numerical methods, and introduce basic concepts for Simbody and the SimTK simulation architecture as employed by OpenSim.

5.1   Naming Conventions

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

```
using namespace SimTK;
```

Alternatively, you can introduce symbols selectively with statements like

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

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

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

5.2   Numbers and Constants in SimTK

SimTK supports both float (single) and double precision, and is compiled with one of those as its default, which is then referred to in SimTK as type `Real`. `SimTK::Real` is simply a `typedef` to either the built-in `float` or `double` type. OpenSim *always* uses double precision, so for our purposes the `SimTK::Real` type is always defined

```
typedef double Real;
```

Thus `SimTK::Real` and `double` are interchangeable in the OpenSim API. We will use `double` everywhere because it is more conventional and requires fewer characters. However, if you look elsewhere at SimTK documentation, you will see type `Real` used instead; just interpret that as `double` when you see it. Similarly, SimTK has a type `SimTK::Complex` which is interchangeable here with the C++ built-in type `std::complex<double>`. You probably will not need to use complex numbers with OpenSim, but they are available if you need them.

SimTK pre-defines a number of constants to machine precision; we recommend you use those rather than defining your own. The most useful ones are: `Pi`, `E` (*e*), `Infinity`, `NaN`

(not-a-number), and `I` ($i = \sqrt{-1}$). Note that the first letter of constants are capitalized and in the `SimTK` namespace.

5.3 Vectors and Matrices

SimTK has two different sets of classes for vector and matrix objects. You have seen types `Vec3` (a 3-vector) and `Vector` (a variable-length vector) in OpenSim examples. Here we will discuss those in more detail. If you want to know more about the design goals and implementation of these classes, see the SimTK Simmatrix document here: https://simtk.org/home/simtkcore, Documents tab.

First, there are classes to represent small, fixed size vectors and matrices with zero runtime overhead: `Vec` for column vectors, and `Mat` for matrices[*]. These classes are templatized based on size and element type. Synonyms (`typedef`s) are defined for common combinations; for example, `Vec3` is a synonym for `Vec<3,double>`, while `Mat22` is a synonym for `Mat<2,2,double>`. You can also create other combinations, such as `Mat<2,10,double>` or `Vec<4,std::complex<double>>`. However, the size must always be determinable at compile time. The in-memory representation of these small objects is minimal: only the data elements are stored.

Second, there are classes to represent large vectors and matrices whose sizes are determined at runtime: `Vector_` for column vectors and `Matrix_` for matrices[†]. These classes are templatized based on element type. Types `Vector` and `Matrix` are synonyms for `Vector_<double>` and `Matrix_<double>`. Again, you can use other element types. In fact, the element type can even be one of the fixed-size vector or matrix objects. For example, `Vector_<Vec3>` is a vector, where each element is itself a three-component vector. The type `Vec<2,Vec3>`, called a *spatial vector*, is useful for combining rotational and translational quantities into a single object representing a spatial velocity or spatial force, for example. However, it is not permissible to use the variable-size `Vector_` or `Matrix_` objects as element types. The in-memory representation of these objects includes,

---

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

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

in addition to the data, an opaque descriptor containing the length and information on how the data is laid out; the declared objects actually consist only of a pointer (essentially a void*) to the descriptors. This has many advantages for implementation and binary compatibility, but makes it difficult to look through these objects in a debugger as you can with the small `Vec` and `Mat` classes.

Here are some sample declarations:

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

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

typedef Vec<2,Vec3> SpatialVec;
```

5.3.1 Operators

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

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

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

There also are versions of many standard math functions that operate on vectors and matrices: `sin()`, `exp()`, `sqrt()`, etc. and additional functions `abs()`, `min()`, `max()`,

`sort()`, `mean()`, `median()`. These allow many calculations to be written in a very concise way.

5.3.2 Construction and assignment

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

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

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

```
Vec3   v;
Mat22  m;

Vec3 v(0);   // v=(0,0,0)
v=0;         //      "
v=3;         // v=(3,3,3)
Mat33 m(0); // m=( 0,0 ) zero
m=0          //    ( 0,0 )
Mat33 m(1); // m=( 1,0 ) identity
m=1;         //    ( 0,1 )

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

This convention is especially apt for matrices, because the matrix resulting from such a scalar assignment "acts like" that scalar. That is, if you multiply by this matrix the result is identical to a scalar multiply by the original scalar. Two important special cases enabled by

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

5.3.3 Indexing

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

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

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

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

5.3.4 Output

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

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

5.4   Basic Geometry and Mechanics

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

5.4.1 Stations (points)

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

5.4.2 Directions (unit vectors)

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

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

5.4.3 Rotations

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

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

$$^{G}R^{B} \equiv \begin{bmatrix} \mathbf{x}^{B} \end{bmatrix}_{G} \quad \begin{bmatrix} \mathbf{y}^{B} \end{bmatrix}_{G} \quad \begin{bmatrix} \mathbf{z}^{B} \end{bmatrix}_{G}$$

(The notation $\mathbf{v}^{B}$ indicates a (column) vector quantity $\mathbf{v}$ fixed to reference frame B, with measure numbers expressed in B's frame, while the operator $[\,]_{F}$ indicates that the measure numbers of some physical quantity are re-expressed in coordinate frame F.) So the symbol $^{G}R^{B}$ should be read "the axes of frame B expressed in frame G," or "the orientation of frame B in G," or just "B in G." We never use "R" alone for a rotation matrix and neither should you; that is a recipe for certain disaster. Instead, always provide the two frames. Using this notation, you can simply match up superscripts to rotate vectors or compose rotations. When under tight typographical restrictions, as in source code, we write $^{G}R^{B}$ in "monogram" notation as `R_GB`. Also, since these are orthogonal, the inverse of a rotation matrix is just its transpose, which serves simply to swap the superscripts. Use the SimTK "~"

operator to indicate rotation inversion: $\sim^{G}R^{B} = {}^{B}R^{G}$. As an example, if you have a rotation ${}^{G}R^{B}$ and a vector $[\mathbf{v}]_{B}$ expressed in B, you can re-express that same vector in G like this: $[\mathbf{v}]_{G} = {}^{G}R^{B}[\mathbf{v}]_{B}$. To go the other direction, we can write $[\mathbf{v}]_{B} = {}^{B}R^{G}[\mathbf{v}]_{G} = \sim^{G}R^{B}[\mathbf{v}]_{G}$. As a C++ code fragment, this can be written:

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

…

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

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

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

5.4.4 Transforms

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

$$
{}^{G}X^{B} \approx \left(\begin{bmatrix} \mathbf{x}^{B} \\ 0 \end{bmatrix}_{G} \quad \begin{bmatrix} \mathbf{y}^{B} \\ 0 \end{bmatrix}_{G} \quad \begin{bmatrix} \mathbf{z}^{B} \\ 0 \end{bmatrix}_{G} \quad \begin{bmatrix} {}^{G}p^{B} \\ 1 \end{bmatrix}_{G} \right)
$$

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

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

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

$$
{}^{G}X^{B} \equiv \left(\begin{pmatrix} & & \\ & {}^{G}R^{B} & \\ & & \end{pmatrix} \begin{pmatrix} {}^{G}p^{B} \\ \\ \end{pmatrix} \\ \begin{matrix} 0 & 0 & 0 & 1 \end{matrix} \right)
$$

In our implementation, the physical layout of a Simbody `Transform` is just the three columns of the rotation matrix followed immediately in memory by the translation vector, that is, ${}^{G}X^{B} = \begin{bmatrix} {}^{G}R^{B} & | & {}^{G}p^{B} \end{bmatrix}_{3\times4}$. There is no need for the fourth row to be stored in memory since it is always the same.

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

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

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

5.4.5 Inertia

The SimTK `Inertia` class is a 3x3 symmetric matrix. The class provides some convenient constructors and methods for shifting to and from a body's center of mass. The OpenSim API uses this class for specifying body inertia properties. If you want to see what else you can do with this class, look it up in the Doxygen documents available at https://simtk.org/home/simtkcore, Documents tab.

5.5   Available SimTk Numerical Methods

Most of the SimTK numerical methods you will need are wrapped by the OpenSim API, so you will not need to access them directly through SimTK. However, many such numerical methods are available if you need them. Some of the most commonly used are:

- Linear algebra (various object-oriented factorization and eigenvalue classes, as well as direct access to Lapack and Blas if needed)
- Optimization (constrained and unconstrained)
- Numerical integration
- Numerical differentiation
- Random number generation
- Polynomial root finding

For more information, see https://simtk.org/home/simtkcore, Documents tab. SimTKlapack and Simmath documents are available from there, as well as the detailed Doxygen documents that describe individual classes and methods.

5.6   Multibody Dynamics Concepts (Simbody)

OpenSim's dynamics capability is based on the open-source Simbody package that is part of SimTK. Simbody is a full-featured, high-performance multibody dynamics toolset using internal coordinates (that is, generalized coordinates relating one body to the next, rather than Cartesian coordinates) and capable of modeling open- and closed-topology systems. Computation is performed using a recursive O(n) method so that performance scales linearly with problem size.

Simbody provides some basic components (objects) that OpenSim uses to construct the multibody system that underlies an OpenSim dynamic model. In most cases you will interact with these objects through the OpenSim API; however, you can get direct access to them for more advanced functionality. The basic Simbody concepts are:

- Body (mass properties and geometry)
- Mobilizer (internal coordinate joint)
- Constraint
- Force

Forces are important, but they are typically domain-specific (like muscles) rather than being part of the multibody system itself so are discussed elsewhere.

Simbody provides a wide set of built-in mobilizer (joint) and constraint types and allows arbitrary new ones to be constructed; OpenSim makes extensive use of that capability. For a comprehensive discussion of custom mobilizers, see the preprint of the paper

> Ajay Seth, Michael Sherman, Peter Eastman, and Scott Delp, Minimal formulation of joint motion for biomechanics. *Nonlinear Dynamics, Submitted* (2009).

The most important Simbody base classes are:

- `MobilizedBody` (combines a body and its inboard mobilizer)
- `Constraint`

You can obtain direct access to the concrete objects derived from these types from the OpenSim API, and then make use of the Simbody API to interact with them.

For more information, see https://simtk.org/home/simtkcore, Documents tab. The SimTK Tutorial, Simbody Theory Manual, and detailed Doxygen documents available there describe the individual classes and methods.

5.7   SimTK Simulation Concepts

The figure below shows the primary objects involved in computational simulation of a physical system in SimTK: *System*, *State*, and *Study*. OpenSim creates and manages specific objects of these types that are suitable for the domain of neuromuscular biomechanics. In particular, the OpenSim `Model` class implements a SimTK System, and the OpenSim `Manager` represents a Study. OpenSim uses a `SimTK::State` object directly to represent the state of an OpenSim `Model`.
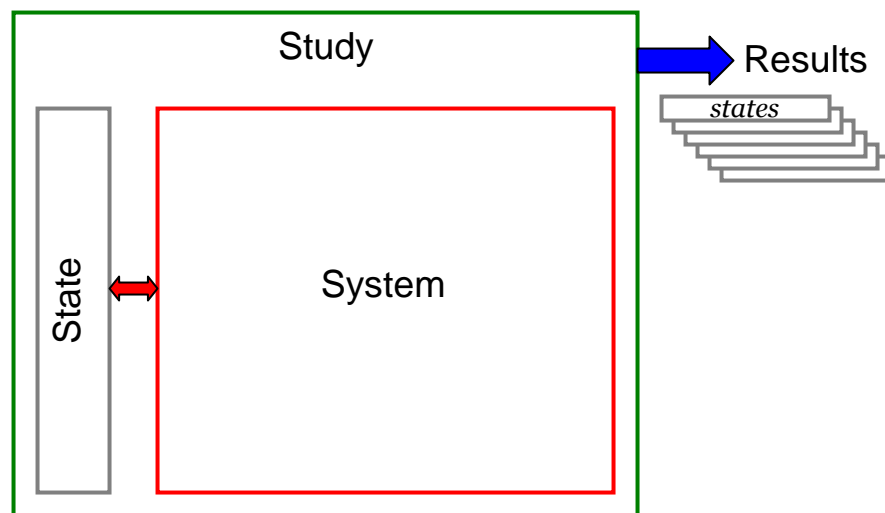
A *System* is the computational embodiment of a mathematical model of the physical world. A System typically comprises several interacting, separately meaningful subsystems. A System contains models for physical objects and the forces that act on them and specifies a set of variables whose values can affect the System's behavior. However, the System itself is an unchangeable, state-free ("const") object. Instead, the values of its variables are stored in a separate object, called a *State* (more details below). Finally, a *Study* couples a System and one or more States, and represents a computational experiment intended to reveal something about the System. By design, the results of *any* Study can be expressed as a State value or set of State values which satisfies some pre-specified criteria, along with results which the System can calculate directly from those State values. Such a set of State values is often called a *trajectory*.

It is important to note that our notion of "state" is somewhat more general than the common use of the term. By state, we mean *everything* variable about a System. That includes not only the traditional continuous time, position and velocity variables, but also discrete variables, memory of past events, modeling choices, and a wide variety of parameters that we call *instance variables*. The System's State has entries for the values of all of these variables.

In an internal coordinate representation, our position coordinates are generalized coordinates q; our velocity coordinates are generalized speeds u. There are also auxiliary continuous state variables we denote z; these are used as state variables for force models, controllers, etc.

5.7.1 Structure of a System

A System is composed of a set of interlocking pieces, which we call *subsystems*.
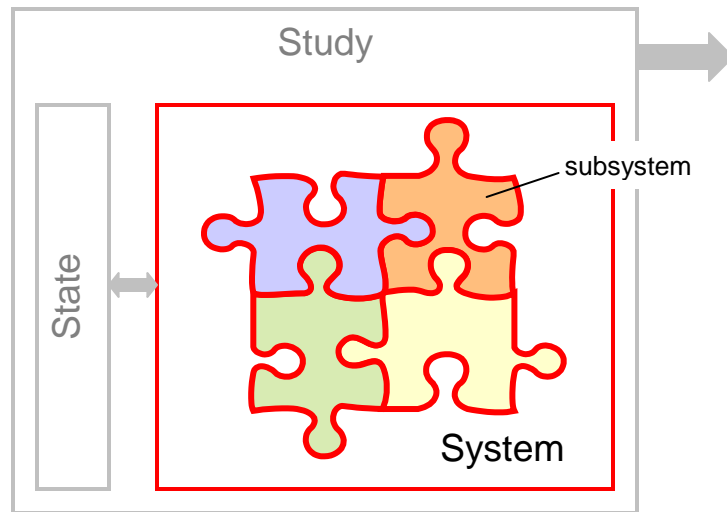
In this jigsaw puzzle analogy, you can think of the System as providing the "edge pieces" which frame the subsystems into a complete whole.

In general, any subsystem of a System may have its own state variables, as can the System itself. The System ensures that its subsystems' state needs are provided for within the overall System's State. The calculations performed by subsystems are interdependent in the sense of having interlocking computational dependencies.

Note that by design this is *not* a hierarchical structure. It is a flat partitioning of a System into a small number of Subsystems. In a higher-level modeling layer like OpenSim, you will find hierarchical models, which are a powerful way to represent the physical world. However, computational resources are flat, not hierarchical, and the SimTK System/Subsystem scheme is a computational device, not a modeling system. The intent is that a modeling layer (or user program) assembles a System from a small library of Subsystems just at the point when it is ready to perform resource-intense computations.

5.7.2 Structure of a multibody system

Let's look at Simbody in this context. Simbody primarily provides *one* computational subsystem (one puzzle piece) of a complete multibody mechanics System. This piece, called the `SimbodyMatterSubsystem`, manages the representation of interconnected massive objects (that is, bodies interconnected by joints). Simbody can use this representation to

perform computations which permit a wide variety of useful studies to be performed. For example, given a set of applied forces, Simbody can very efficiently solve a generalized form of Newton's 2nd law F=ma. On the other hand, Simbody is agnostic about the forces F, which come from domain-specific models. That is, Simbody fully understands the concept of *forces*, and knows exactly what to do with them, but hasn't any idea from where they might have come. OpenSim provides the remaining pieces, such as muscle force subsystems.

A complete System thus consists of both the matter subsystem implemented by Simbody, and user-written or OpenSim-provided force subsystems. So for a multibody system, the general SimTK System described above is specialized to look something like this:
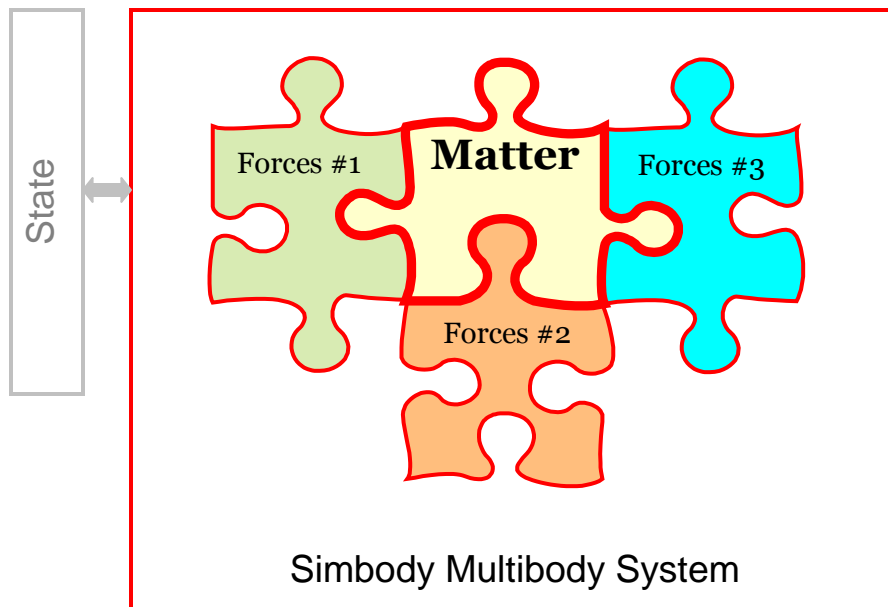
Although both the `SimbodyMatterSubsystem` and the forces from subsystems require state variables, as discussed above, any SimTK System (including an OpenSim `Model`) is a stateless object once constructed. Its subsystems collectively define the System's parameterization, but the parameter values themselves are stored externally in a separate `SimTK::State` object.

For more information, see https://simtk.org/home/simtkcore, Documents tab. The SimTK Advanced Programming Guide, Simbody Theory Manual, and detailed Doxygen documents available there describe the individual classes and methods.
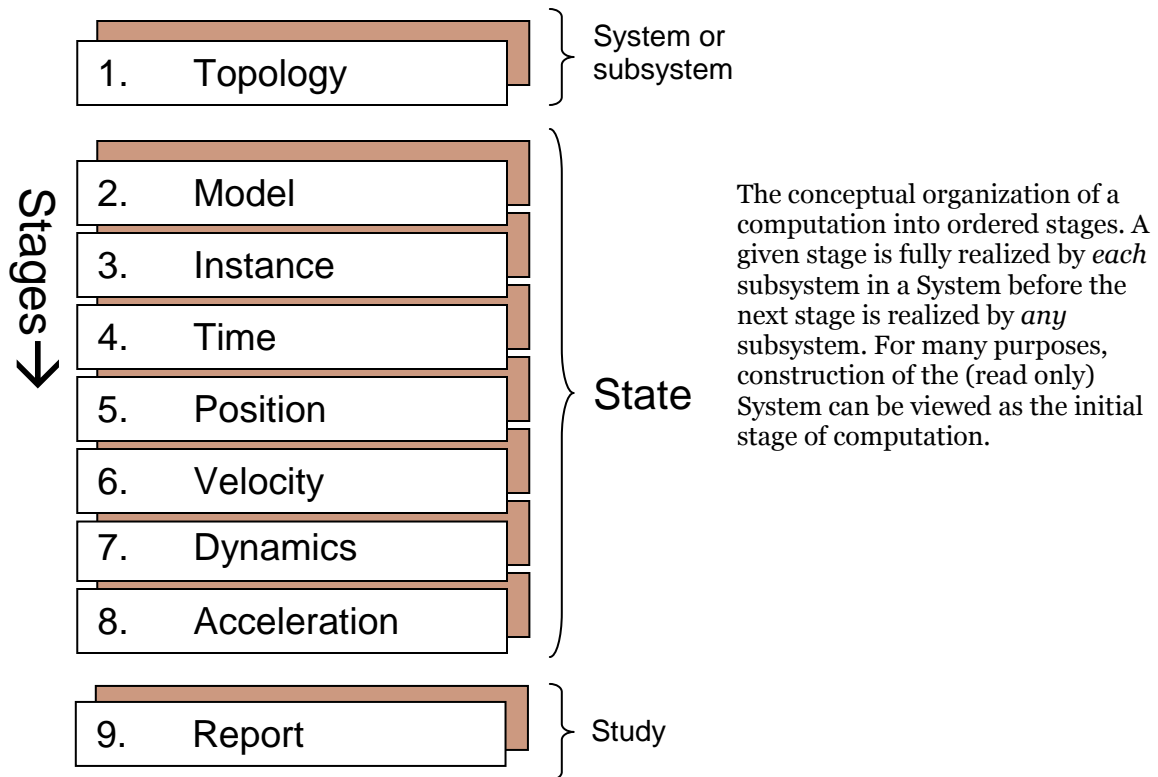
5.7.3 State realization

The state variables collectively represent a complete description of the state of a 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 generalized acceleration of each internal coordinate

These are not independent pieces of 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 to which that information belongs. The figure below shows all the stages.

The conceptual organization of a computation into ordered stages. A given stage is fully realized by *each* subsystem in a System before the next stage is realized by *any* subsystem. For many purposes, construction of the (read only) System can be viewed as the initial stage of computation.

The "Topology" stage is not part of the State; it represents the fixed contents of the System. "Model" stage is used for setting modeling choices, such as whether to use quaternions or Euler angles for joint orientation. "Instance" stage sets instance variables, such as masses, spring constants, attachment points, etc. Those stages are fixed during a simulation. The remaining stages change dynamically:

Time: At this stage, time has advanced and state variables have their new values, but no derived information has yet been calculated. You can query the State for time and any of the state variables, but nothing else.

Position: At this stage, the spatial positions of all bodies are known, along with related quantities such as separation distances.

Velocity: At this stage, the spatial velocities of all bodies are known, along with related quantities.

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.

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 for data output. 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:

- Changing an instance variable, such as a mass or spring constant, brings the State back to Model stage.
- Modifying time t will bring the State back to Instance stage.
- Modifying a generalized coordinate q will bring the State back to Time stage.
- Modifying a generalized speed u will bring the State back to Position stage.
- Modifying an auxiliary variable 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.

5.8 For More Information about SimTK

The above provides a quick look at SimTK, but there is much more to learn if you are interested. A good place to start is https://simtk.org/home/simtkcore, Documents tab, where you will find several tutorials and theory manuals. More information and source code for the projects making up SimTK can be found on Simtk.org under projects: simbody, simmath, cpodes, simtkcommon, and lapack.