Creating a Customized Actuator

In this exercise, we will create a specific type of linear actuator that implements a spring with controllable stiffness. The source code and associated files for this exercise are located in the OpenSim source code repository:

```
https://simtk.org/svn/opensim/Branches/OpenSim_BuiltOn_SimTK_1_1/OpenSim/Ex
amples/CustomActuatorExample
```

In Windows Explorer, create a new directory, e.g., C:\Projects\CustomActuatorExample. Navigate into that directory, right-click, and choose SVN Checkout. For URL of Repository, enter the above URL. Once you have obtained the example files, launch CMake. Point to the /CustomActuatorExample directory as the source code location, and create any directory you wish for the build location. 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. Click Configure again and then click Generate. Then close CMake.

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 theLinearActuator class.

1. Actuator Overview

We define an actuator as something that produces 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 the state variables at any time step.

2. The LinearActuator 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 LinearActuator class. This class will eventually be incorporated into the OpenSim 2.0 distribution, but to serve as an example of how we design our actuator classes we have implemented and included it within the source material of this example. Figure 1 illustrates the LinearActuator 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 A and B.

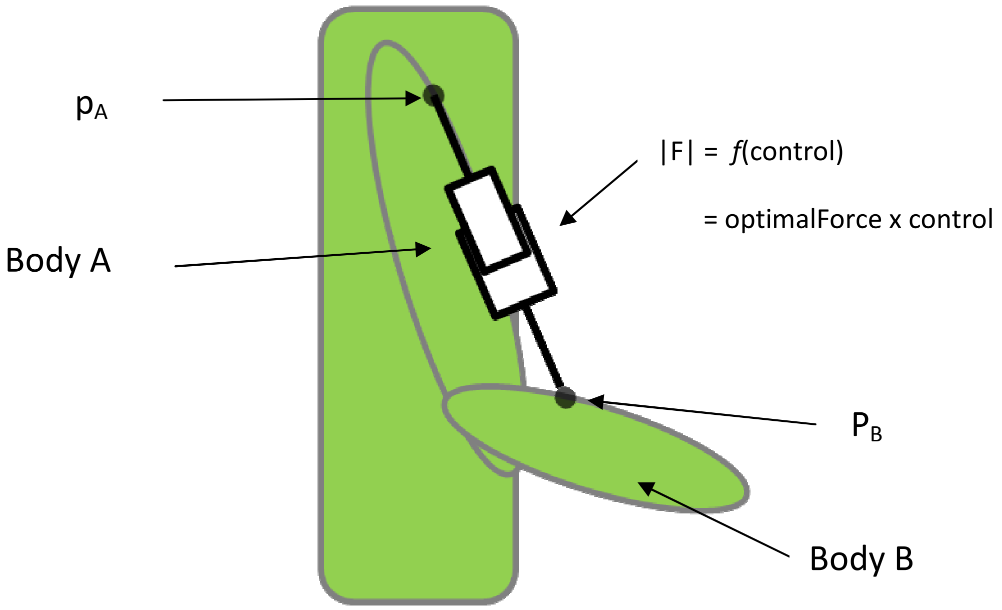
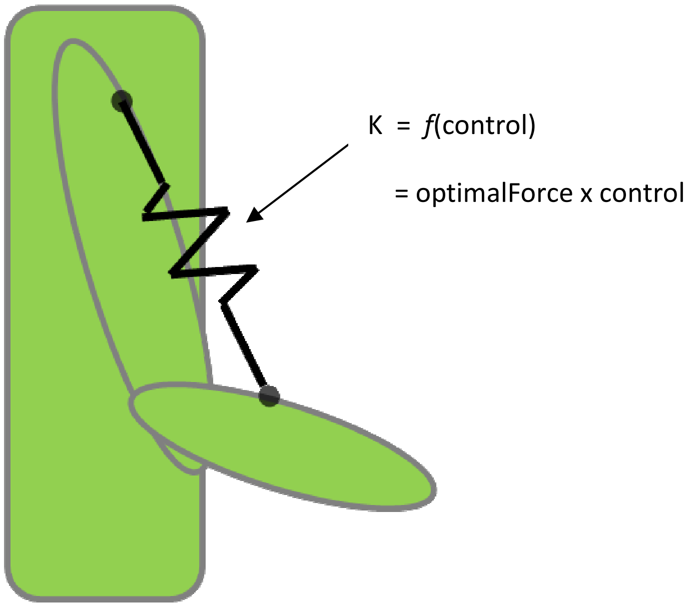


Figure 1. Illustration of the LinearActuator

3. The ControllableSpring class

Figure 2 illustrates the ControllableSpring class that we will define. Just like LinearActuator, ControllableSpring will act between two points fixed on two different bodies. However, the force magnitude will not simply be calculated as the product of optimal force and control value. Instead, the spring stiffness will be calculated by **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).**



Firgure 2. Illustration of the ControllableSpring

3.1. Defining the ControllableSpring class (ControllableSpring.h)

Open ControllableSpring.h, which contains a partial definition of the ControllableSpring class. The following instructions will outline ALL the steps for defining the ControllableSpring class. However, since the file is mostly completed, you will only 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 derived class of LinearActuator.

```
#include "LinearActuator.h"

namespace OpenSim {

class ControllableSpring : public LinearActuator
{
```

3.1.1. Defining Properties

Our new actuator will have all of the properties of the LinearActuator 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;
```

3.1.2. The Constructors

Next we define the constructors. The constructors take the same form as the LinearActuator 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="") :
      LinearActuator(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) :
      LinearActuator(aControllableSpring),
      _restLength(_propRestLength.getValueDbl())
```

```
{
	setNull();
	_restLength = aControllableSpring.getRestLength();
}
/* use the default destructor */
virtual ~ControllableSpring() {};
```

3.1.3. Setup Methods

We will define two private member methods that are used during construction to initialize the ControllableSpring instance. First, setupProperties() is used to setup the properties of the ControllableSpring from values read in from a 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);

}
```

3.1.4. 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; };
```

3.1.5. 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 pure virtual in the CustomActuator base class, so any derived classes must define its behavior. LinearActuator has already defined its own implementation of computeForce(), but we will redefine it here so that ControllableSpring 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 LinearActuators 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 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);
}
```

3.1.6. Finish the class definition and close the namespace

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

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

3.2. Using the ControllableSpring (toyLeg_example.cpp)

Open the toyLeg_example.cpp file. This file implements a main() program that builds a toy leg model that is driven by a LinearActuator (Figure 3). 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 LinearActuator called "piston" acts between the distal end of linkage1 and the center of the block. We will modify the main routing to replace the piston actuator with a variable stiffness spring.

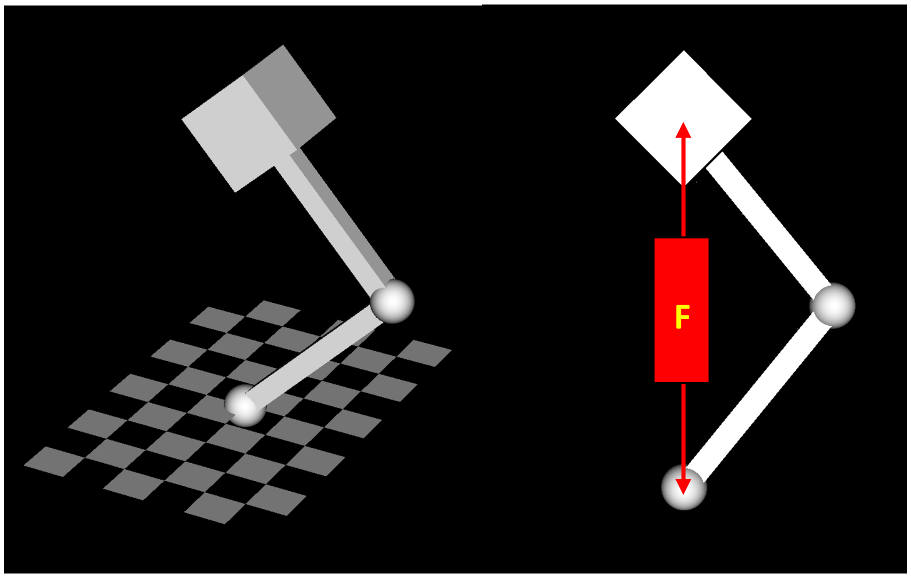


Figure 3. toyLeg example

3.2.1. Ready toyLeg.cpp to be able to use the ControllableSpring

Add the ControllableSpring class to the included files as shown below. Look in the Visual C++ Solution Explorer to find the Actuators_examples project. Right click is and select build in order to rebuild toyLeg_example.cpp and to force the first build of ControllableSpring.h.

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

using namespace OpenSim;
using namespace SimTK;
```

3.2.2. 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 identical geometry to 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);
```

3.2.3. 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};
```

3.2.4. 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'
```

3.2.5. Point the control set to the new control values

Comment 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);
```

3.2.6. 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.

3.2.7. Build and Run

Build the **Actuator_examples** project again, then build the **INSTALL** project. Navigate to the install directory and find the executable file, toyLeg_example.. Before running this executable, you must set you must ensure that the <OpenSim2.0_intall_dir>/bin directory appears at the front of your PATH. After running the excutable, you can now use the GUI to open the model toyLeg.osim and load the motionfile you renamed. Upon visualizing the motion you should see the block oscillate at different magnitudes and frequencies as the spring stiffness is varied over time.