Creating a Controller Part One

The steps covered in part one are:

- Defining the desired trajectory of the model
- Designing a controller to track a desired trajectory
- Implementing the controller

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 $z(t) = 0.15 \sin(2ft)$, where the frequency is $f = ½$ Hz, which simplifies to $z(t) = 0.15 \sin(t)$. 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: $z'(t) = 0.15 \cos(t)$. 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).

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 $0 > x > 1$. 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*, and the system damping will dominate the response slowing down the time to reach the equilibrium state. If k_v is too low, the controller will be *underdamped* and the system will overshoot and oscillate about an equilibrium state which also slows the settling time. 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.

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 Controller:

```
class TugOfWarPDController : public Controller {
OpenSim_DECLARE_CONCRETE_OBJECT(TugOfWarController, Controller);

public:
        TugOfWarController(double aKp, double aKv) : Controller(), kp( aKp ), kv( aKv )
        {
        }
...
```

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

The behavior of the controller is determined by its computeControls function, which implements the intended control law. Two arguments are passed into this function: the current state, s, of the system and a vector of controls, which are the model controls to be computed. In our model, index 0 refers to the left muscle and index 1 refers to the right muscle. The computeControls function computes and adds in the values for controls for each of its actuators based on the current state and desired position, velocity, and acceleration. You will need to fill out the computation of velocity and acceleration:

```
void computeControls( const SimTK::State& s, SimTK::Vector controls ) const
{
        // Get the current time in the simulation.
        double t = s.getTime();
        // Read the mass of the block.
        double blockMass = getModel().getBodySet().get( "block" ).getMass();
        // Get pointers to each of the muscles in the model.
        Muscle* leftMuscle = dynamic_cast<Muscle*>        ( &getActuatorSet().get(0) );
        Muscle* rightMuscle = dynamic_cast<Muscle*> ( &getActuatorSet().get(1) );
        // Compute the desired position of the block in the tug-of-war
        // model.
        double zdes  = desiredModelZPosition(t);

        //////////////////////////////////////////////////////////////
        // 3) Compute the desired velocity of the block in the tug- //
        //    of-war model.  Create a new variable zdesv to hold    //
        //    this value.                                           //
        //////////////////////////////////////////////////////////////

        // 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);

        //////////////////////////////////////////////////////////////
        // 4) Get the current velocity of the block in the tug-of-  //
        //    war model.  Create a new variable zv to hold this     //
        //    value.                                                //
        //////////////////////////////////////////////////////////////

        // 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  );

        //////////////////////////////////////////////////////////////
        // 5) Compute the correction to the desired acceleration    //
        //     arising from the deviation of the block's current    //
        //     velocity from its desired velocity (this deviation   //
        //     is the "velocity error").  Create a new variable     //
        //     vErrTerm to hold this value.                         //
        //////////////////////////////////////////////////////////////

        //////////////////////////////////////////////////////////////
        // 6) In the computation of desAcc below, add the velocity  //
        //    error term you created in item #5 above.  Please      //
        //    update the comment for desAcc below to reflect this   //
        //    change.                                               //
        //////////////////////////////////////////////////////////////

        // Compute the total desired acceleration based on the initial
        // desired acceleration plus the position error term we
        // computed above.
        double desAcc = zdesa + 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 for the left muscle.
        double FoptL = leftMuscle->getMaxIsometricForce();
        // Get the maximum isometric force for the right muscle.
        double FoptR = rightMuscle->getMaxIsometricForce();
        // If desired force is in direction of one muscle's pull
        // direction, then set that muscle's control based on desired
        // force.  Otherwise, set the muscle's control to zero.
        double leftControl = 0.0, rightControl = 0.0;
        if( desFrc < 0 ) {
                leftControl = abs( desFrc ) / FoptL;
                rightControl = 0.0;
        }
        else if( desFrc > 0 ) {
                leftControl = 0.0;
                rightControl = abs( desFrc ) / FoptR;
        }
        // Don't allow any control value to be greater than one.
        if( leftControl > 1.0 ) leftControl = 1.0;
        if( rightControl > 1.0 ) rightControl = 1.0;
        // Thelen muscle has only one control
        Vector muscleControl(1, leftControl);
        // Add in the controls computed for this muscle to the set of all model controls
        leftMuscle->addInControls(muscleControl, controls);
        // Specify control for other actuator (muscle) controlled by this controller
        muscleControl[0] = rightControl;
        rightMuscle->addInControls(muscleControl, controls);
}
```

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:

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

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

};
```

Next: Creating a Controller Part Two

Previous: Creating a Controller

Home: Scripting and Development | Developer's Guide | Adding New Functionality