Custom Muscle Model Part One

The steps covered in part one are:

The header file (FatigableMuscle.h)

We start defining our new muscle model by creating a header file. The model extends the Millard2012EquilibriumMuscle, and will include fatigue effects that are loosely based on 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).

The model defines active and fatigued motor unit states which are the percentages of motor units in either pool. The motor units not in the active or fatigued pools are considered "recovered" and are able to become active. The actual activation used to compute the contraction dynamics is thus a active motor unit pool percentage times the desired activation resulting from the neural excitation of the muscle. Refer to the Doxygen documentation for a complete description of the *Muscle* interface and details of the*Millard2012EquilibriumMuscle* model.

We will call our model FatigableMuscle. At the top of *Fatigable Muscle.h*, we include the header file for the base class and derive our new class.

```
#include <OpenSim/Actuators/Millard2012EquillibriumMuscle.h>
namespace OpenSim {

class FatigableMuscle : public Millard2012EquilibriumMuscle {
OpenSim_DECLARE_CONCRETE_OBJECT(FatigableMuscle,
                                Millard2012EquilibriumMuscle);
```

Defining properties

FatigableMuscle will have ~~all of~~ the properties of the Millard2012EquilibriumMuscle model, plus ~~two~~ four additional ones to define the rates at which active muscle fibers fatigue and the rate at which fatigued fibers recover and defaults for the active and fatigued motor unit states.

```
//=============================================================================
// PROPERTIES
//=============================================================================
    /** @name Property declarations
    These are the serializable properties associated with this class. **/
    /**@{**/
    OpenSim_DECLARE_PROPERTY(fatigue_factor, double,
        "percentage of active motor units that fatigue in unit time");
    OpenSim_DECLARE_PROPERTY(recovery_factor, double,
        "percentage of fatigued motor units that recover in unit time");
       OpenSim_DECLARE_PROPERTY(default_active_motor_units, double,
        "default state value for the fraction of motor units that are active");
    OpenSim_DECLARE_PROPERTY(default_fatigued_motor_units, double,
        "default state value for the fraction of motor units that are fatigued");
    /**@}**/
```

The fatigue and recovery factors (new properties) define the behavior of the FatigableMuscle and we therefore define a constructor to initialize these properties upon creation.NOTE that the constructors also construct the properties and initialize the methods and this is typically done by calling a private constructProperties() methods.

```
//-----------------------------------------------------------------------
// CONSTRUCTION
//-----------------------------------------------------------------------
FatigableMuscle();
FatigableMuscle(const std::string &name, double maxIsometricForce,
                        double optimalFiberLength, double tendonSlackLength,
                        double pennationAngle, double fatigueFactor,
                        double recoveryFactor);
```

Provide accessor methods to get and set the new properties. Note, specified properties automatically provide methods get/set_<property_name>() but we include these methods for completeness.

```
      //-----------------------------------------------------------------------
      // GET & SET Properties
      //-----------------------------------------------------------------------
      /** Fatigue and recovery factors are properties of this muscle */
      double getFatigueFactor() const { return get_fatigue_factor(); }
      void setFatigueFactor(double aFatigueFactor);
      double getRecoveryFactor() const { return get_recovery_factor(); }
      void setRecoveryFactor(double aRecoveryFactor);
      /** default values for states */
      double getDefaultActiveMotorUnits() const;
      void setDefaultActiveMotorUnits(double activeMotorUnits);
      double getDefaultFatiguedMotorUnits() const;
      void setDefaultFatiguedMotorUnits(double fatiguedMotorUnits);
```

Defining states

Millard2012EquilibriumMuscle has two states: activation and fiber length. We will add ~~two~~ three more states: one for the target activation, the fraction of active motor units (range 0.0 to 1.0), and one for the fraction of fatigued motor units (range 0.0 to 1.0). These states will be added to the system of equations representing the dynamics of the muscle in *FatigableMuscle.cpp*, but in the header we specify the accessor methods:

```
      //-----------------------------------------------------------------------
      // GET & SET States and their derivatives
      //-----------------------------------------------------------------------
      /** Fatigued activation state and time derivative accessors */
      double getTargetActivation(const SimTK::State& s) const;
      void setTargetActivation(SimTK::State& s, double fatiguedAct) const;
      double getTargetActivationDeriv(const SimTK::State& s) const;
      void setTargetActivationDeriv(const SimTK::State& s,
                                    double fatiguedActDeriv) const;
      /** Active motor units state and time derivative accessors */
      double getActiveMotorUnits(const SimTK::State& s) const;
      void setActiveMotorUnits(SimTK::State& s, double activeMotorUnits) const;
      double getActiveMotorUnitsDeriv(const SimTK::State& s) const;
      void setActiveMotorUnitsDeriv(const SimTK::State& s,
                                  double activeMotorUnitsDeriv) const;
      /** Fatigued motor units state accessors */
      double getFatiguedMotorUnits(const SimTK::State& s) const;
      void setFatiguedMotorUnits(SimTK::State& s,
                           double fatiguedMotorUnits) const;
      double getFatiguedMotorUnitsDeriv(const SimTK::State& s) const;
      void setFatiguedMotorUnitsDeriv(const SimTK::State& s,
                                    double fatiguedMotorUnitsDeriv) const;
```

Required Interfaces

As a Muscle and a ModelComponent, the FatigableMuscle must satisfy the required interfaces put in place by the Muscle and ModelComponent classes from which it derives. Because our new class derives from a concrete (meaning it implements all the methods to satisfy the interfaces) Muscle class, the Millard2012EquilibriumMuscle, and not directly from Muscle, we only have to specify (override) the methods for which we want to change the behavior. We are adding new states to model fatigue and this is done by overriding the ModelComponent method addToSystem(). Methods to initialize the new states from properties and to store state values in properties are also part of the ModelComponent interface.

```
protected:
       // Model Component Interface
       /** add new dynamical states to the multibody system corresponding
           to this muscle */
       void addToSystem(SimTK::MultibodySystem& system) const OVERRIDE_11;
       /** initialize muscle state variables from properties. For example, any
           properties that contain default state values */
       void initStateFromProperties(SimTK::State& s) const OVERRIDE_11;
       /** use the current values in the state to update any properties such as
           default values for state variables */
       void setPropertiesFromState(const SimTK::State& s) OVERRIDE_11;
```

Finally we want to change how the Millard2012EquilibriumMuscle computes the activation (and thus force) of the muscle based on available active motor units. In this case we need to: 1) specify how the new states of the model are initialized and 2) specify the dynamics (derivatives) of all the muscle states by overriding the methods that perform these computations.

```
       //--------------------------------------------------------------------------
       // COMPUTATIONS
       //--------------------------------------------------------------------------
       /** Determine the initial state values based on initial fiber equilibrium.
             For the FatigableMuscle, the active and fatigued motor units
             states are set to 1.0 and 0.0 respectively, which translates to
             no fatigue as the initial state. */
       void computeInitialFiberEquilibrium(SimTK::State& s) const OVERRIDE_11;

       /** Compute the derivatives for state variables added by this muscle */
       SimTK::Vector computeStateVariableDerivatives(const SimTK::State& s)
             const OVERRIDE_11;
```

The source file (FatigableMuscle.cpp)

We implement the methods that we defined for our class in FatigableMuscle.cpp. This source file will include the methods that we need to override in Maillard2012EquilibriumMuscle that implement the fatigue behavior of the muscle.

constructProperties()

The function constructProperties() is used to create the properties of the muscle class, specify their default values, and add them to the set of all OpenSim properties. This enables you to define them in an OpenSim model file.-FatigableMuscle constructors **must** call this method. As we did in the header file, we need to construct 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). Two additional properties define the default values for the state variables that describe the fraction of motor units that are active and fatigued.

```
/*
 * Construct and initialize properties.
 * All properties are added to the property set. Once added, they can be
 * read in and written to files.
 */
void FatigableMuscle::constructProperties()
{
    constructProperty_fatigue_factor(0.0);
    constructProperty_recovery_factor(0.0);
       constructProperty_default_active_motor_units(0.0);
    constructProperty_default_fatigued_motor_units(0.0);
}
```

Adding state variables

Adding new variables to the underlying computational system of equations, whether they be continuous or discrete state variables, cache variables, or modeling options (flags) must be specified in the addToSystem method implemented by all ModelComponents. In the FatigableMuscle example, we will have the base, Millard2012EquilibriumMuscle, add its variables and then add the state variables necessary to model fatigue. Specifically, we add a targetActivation state, which represents the activation state if fatigue was not present and is equivalent to the activation state of the base class. The remaining two states are the fraction of active and fatigued motor units of the muscle.

```
// Define new states and their derivatives in the underlying system
void FatigableMuscle::addToSystem(SimTK::MultibodySystem& system) const
{
       // Allow Millard2012EquilibriumMuscle to add its states, cache, etc.
       // to the system
       Super::addToSystem(system);

       // Now add the states necessary to implement the fatigable behavior
       addStateVariable("target_activation");
       addStateVariable("active_motor_units");
       addStateVariable("fatigued_motor_units");
       // and their corresponding dervivatives
       addCacheVariable("target_activation_deriv", 0.0, SimTK::Stage::Dynamics);
       addCacheVariable("active_motor_units_deriv", 0.0, SimTK::Stage::Dynamics);
       addCacheVariable("fatigued_motor_units_deriv", 0.0, SimTK::Stage::Dynamics);
}
```

computeInitialFiberEquillibrium()

The function computeInitialFiberEquilibrium() computes values of the muscle states assuming the muscle is in an equilibrium state. It is typically called for each muscle before beginning a dynamic simulation. In our muscle class, this function initializes the new states to reasonable values prior to simulation.

```
/* Determine the initial state values based on initial fiber equlibrium. */
void FatigableMuscle::computeInitialFiberEquilibrium(SimTK::State& s) const
{
       // initialize th target activation to be the actual.
       setTargetActivation(s, getActivation(s));
       // assume that all motor units can be activated initially and there is
       // no appreciable fatigue
       setActiveMotorUnits(s, 1.0);
       setFatiguedMotorUnits(s, 0.0);

       // Compute the fiber & tendon lengths according to the parent Muscle
       Super::computeInitialFiberEquilibrium(s);
}
```

Specify the dynamics of the FatigableMuscle

With the states defined and initialized appropriately, all that remains is to specify how the states should evolve in time. The base class (Millard2012EquilibriumMuscle) already defines *activation* and *fiber_length* state variables and we added *target_activation*, *active_motor_units* and *fatigued_motor_units*, for a total of five muscle states and their derivatives must be provided by computeStateVariableDerivatives() in the same order. The dynamics for the target_activation state are already defined by the activation dynamics model employed by the Millard2012EquilibriumMuscle and we can use it to compute the targetActivationRate. Given that the actual activation the muscle experiences is the product of the fraction of active motor units (Ma) and the target activation (a) we can compute the activation rate of the muscle knowing the time derivative of the two quantities such that activationRate = dMa/dt*a + Ma*da/dt. The rate of conversion to/from active and fatigued motor units is based on Liu et al. 2008.

```
/**
 * Compute the derivatives of the muscle states.
 *
 * @param s  system state
 */
SimTK::Vector FatigableMuscle::
computeStateVariableDerivatives(const SimTK::State& s) const
{
       // vector of the derivatives to be returned
       SimTK::Vector derivs(getNumStateVariables(), SimTK::NaN);
       int nd = derivs.size();

       SimTK_ASSERT1(nd == 5, "FatigableMuscle: Expected 5 state variables"
        " but encountered  %f.", nd);

       // compute the rates at which motor units are converted to/from active
    // and fatigued states based on Liu et al. 2008
       double activeMotorUnitsDeriv = - getFatigueFactor()*getActiveMotorUnits(s)
                  + getRecoveryFactor() * getFatiguedMotorUnits(s);

       double fatigueMotorUnitsDeriv = getFatigueFactor()* getActiveMotorUnits(s)
                  - getRecoveryFactor() * getFatiguedMotorUnits(s);

       //Compute the target activation rate based on the given activation model
       const MuscleFirstOrderActivationDynamicModel& actMdl
            = get_MuscleFirstOrderActivationDynamicModel();

       double excitation = getExcitation(s);
       // use the activation dynamics model to calculate the target activation
    double targetActivation = actMdl.clampActivation(getTargetActivation(s));
       double targetActivationRate = actMdl.calcDerivative(targetActivation, excitation);

       // specify the activation derivative based on the amount of active motor
       // units and the rate at which motor units are becoming active.
       // we assume that the actual activation = Ma*a       then,
       // activationRate = dMa/dt*a + Ma*da/dt  where a is the target activation
       double activationRate = activeMotorUnitsDeriv*targetActivation +
                           getActiveMotorUnits(s)*targetActivationRate;

       // set the actual activation rate of the muscle to the fatigued one
       derivs[0] = activationRate;
       // fiber length derivative
       derivs[1] = getFiberVelocity(s);
       // the target activation derivative
       derivs[2] = targetActivationRate;
       derivs[3] = activeMotorUnitsDeriv;
       derivs[4] = fatigueMotorUnitsDeriv;

       // cache the results for fast access by reporting, etc...
       setTargetActivationDeriv(s, targetActivationRate);
       setActiveMotorUnitsDeriv(s, activeMotorUnitsDeriv);
       setFatiguedMotorUnitsDeriv(s, fatigueMotorUnitsDeriv);

       return derivs;
}
```

Next: Custom Muscle Model Part Two

Previous: Creating a Customized Muscle Model

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