Moco: Predict a Squat-to-stand

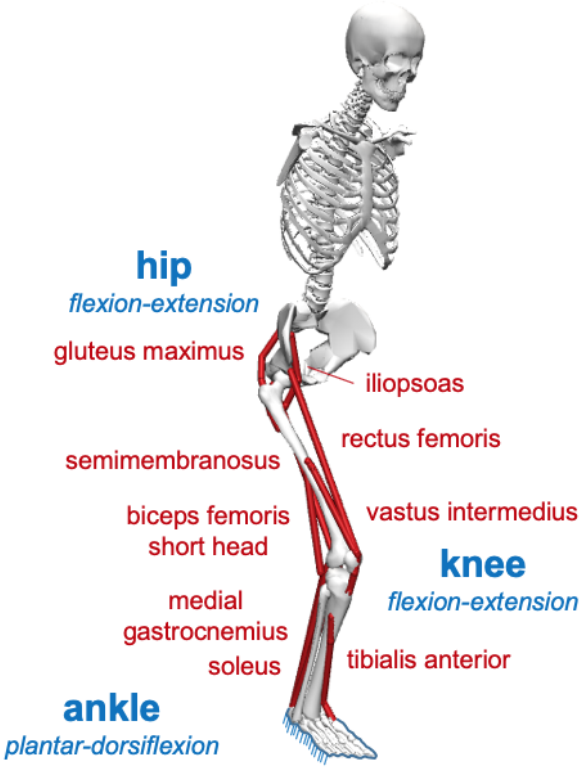
OpenSim Moco is a software toolkit built on top of OpenSim for solving musculoskeletal optimal control problems. Moco can solve "inverse" problems (given experimental motion data, estimate quantities that were not measured during an experiment) and predictive problems (predicting a walking motion, without experimental motion data), as well as problems in between. Moco solves these problems using direct collocation, which is a generic method for solving optimal control problems. Learn more from Moco's website and bioRxiv preprint.

This example will show you how to use Moco to predict a squat-to-stand motion in Matlab. More specifically, the goals of this lab are as follows:

1. Predict a new motion.
2. Track motion data using a torque-driven model.
3. Estimate muscle activity from motion data.
4. Estimate the effect of a passive assistive device on a motion and muscle activity.

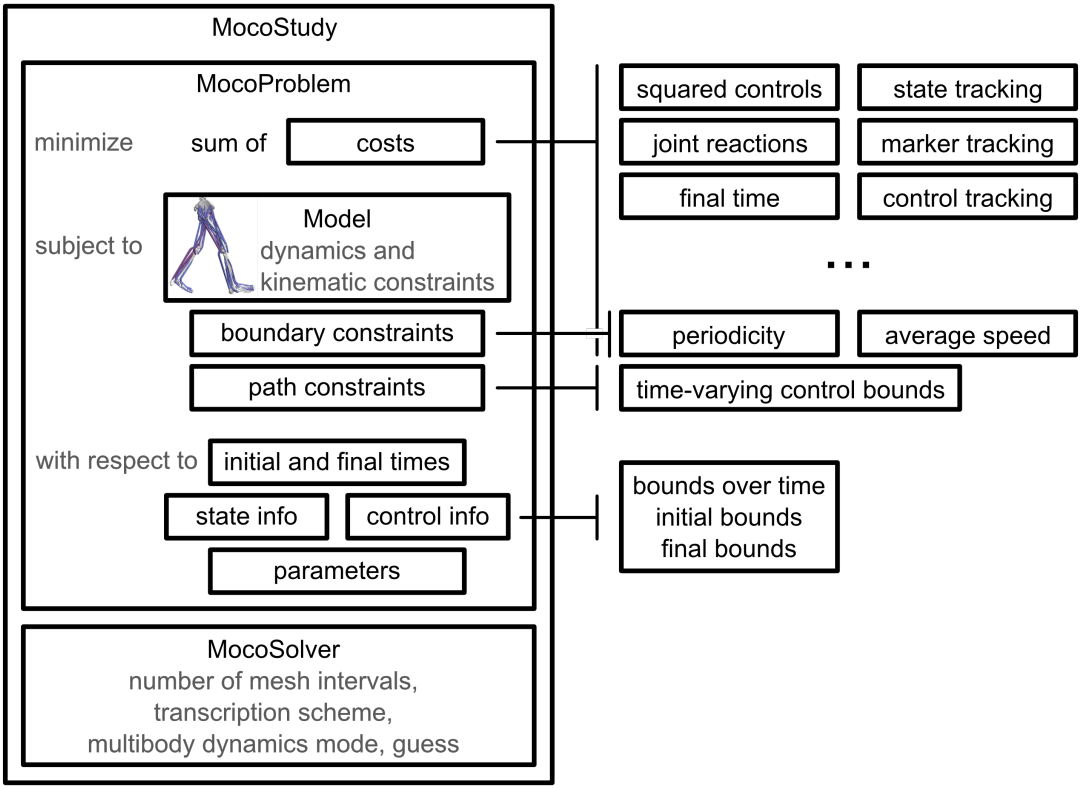
Here is a video of the motion you will predict:

Background



This example uses a 3-degree-of-freedom planar model, with a weld constraint between the foot and the ground. For Parts 1 and 2, we use a torque-driven version of this model. In Parts 4 and 5, we use a muscle-driven version. The muscle-driven model contains 9 muscles.

When using Moco, we begin with a **MocoStudy**, which contains a **MocoProblem** and a **MocoSolver.** A **MocoProblem** contains a description of the optimal control problem (e.g., costs, Model, constraints), while a **MocoSolver** converts the problem into a nonlinear program via direct collocation.



Prepare your environment

Moco is currently a separate software package from OpenSim, but includes OpenSim internally.

1. Download Moco from its website.
2. Unzip Moco anywhere; for example, C:\Users\<username>\opensim-moco. We will refer to this directory as **<opensim-moco>**.
    a. On macOS 10.15 Catalina or later, open a Terminal and run the following command (otherwise, macOS will not permit you to run the Moco code):

    ```
    xattr -r -d com.apple.quarantine <opensim-moco>
    ```

    b. On Windows, update your **PATH** environment variable to include **<opensim-moco>\bin**.
        i. Open the Start Menu, search for "environment", select **Edit the system environment variables**, click **Environment Variables...**.
        ii. Under System variables, select **PATH**, click **Edit...**, and add **<opensim-moco\bin** to the beginning (you may need to append a semicolon to the path).
        iii. Remove any entries for previous OpenSim installations (or, make such paths invalid by appending nonsense text to them, such as "TODO").
3. Start Matlab (as Administrator on Windows) and run **configureMoco.m** in **<opensim-moco>/Resources/Code/Matlab**.
4. Restart Matlab and change your current directory to **<opensim-moco>/Resources/Code/Matlab/exampleSquatToStand**.
5. Open the file **exampleSquatToStand.m**. In this example, you will fill in the blanks in this file.

Part 1: Torque-driven Predictive Problem

Create a new **MocoStudy**. Fill in Part 1a in **exampleSquatToStand.m**.

Part 1a

```
study = MocoStudy();
```

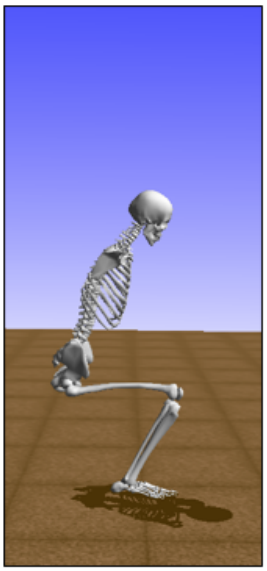
Obtain the **MocoProblem** from within the **MocoStudy** and set the OpenSim **Model** to use (**torqueDrivenModel** is defined at the top of **exampleSquatToStand.m**).

Part 1b

```
problem = study.updProblem();
problem.setModel(torqueDrivenModel);
```

Set bounds on the variables (time, generalized coordinates, generalized speeds, and controls) in the optimal control problem, including bounds on the initial and final generalized coordinates and speeds. Use the values shown in the following image:

$t_i = 0$

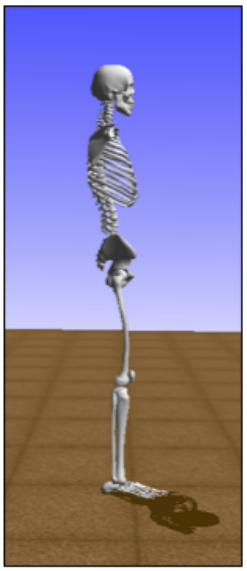


$q_i^{\text{hip}} = -2$

$q_i^{\text{knee}} = -2$

$q_i^{\text{ankle}} = -0.5$

$t_f = 1$



$q_f^{\text{hip}} = 0$

$q_f^{\text{knee}} = 0$

$q_f^{\text{ankle}} = 0$

Generalized coordinates

$q^{\text{hip}} \in [-2, 0.5]$

$q^{\text{knee}} \in [-2, 0]$

$q^{\text{ankle}} \in [-0.5, 0.7]$

Generalized speeds

$u \in [-50, 50]$

$u_i = 0$

$u_f = 0$

Here is a description of the functions on **MocoProblem** for setting bounds:

Setting bounds on variables

```
% problem.setTimeBounds(initial_bounds, final_bounds)
% problem.setStateInfo(path, trajectory_bounds, inital_bounds, final_bounds)
%
% All *_bounds arguments can be set to a range, [lower upper], or to a
% single value (equal lower and upper bounds). Empty brackets, [], indicate
% using default bounds (if they exist). You may set multiple state infos at
% once using setStateInfoPattern():
%
% problem.setStateInfoPattern(pattern, trajectory_bounds, inital_bounds, ...
%       final_bounds)
%
% This function supports regular expressions in the 'pattern' argument;
% use '.*' to match any substring of the state/control path
% For example, the following will set all coordinate value state infos:
%
% problem.setStateInfoPattern('/path/to/states/.*/value', ...)
```

Part 1c

```
% Time bounds
problem.setTimeBounds(0, 1);

% Position bounds: the model should start in a squat and finish
% standing up.
problem.setStateInfo('/jointset/hip_r/hip_flexion_r/value', ...
    [-2, 0.5], -2, 0);
problem.setStateInfo('/jointset/knee_r/knee_angle_r/value', ...
    [-2, 0], -2, 0);
problem.setStateInfo('/jointset/ankle_r/ankle_angle_r/value', ...
    [-0.5, 0.7], -0.5, 0);

% Velocity bounds: all model coordinates should start and end at rest.
problem.setStateInfoPattern('/jointset/.*/speed', [], 0, 0);
```

Specify the goals to minimize; in this case, we minimize the sum of squared controls.

To learn more about **MocoControlGoal**, view the Doxygen documentation for this class on Moco's website.

Part 1d

```
problem.addGoal(MocoControlGoal('myeffort'));
```

Initialize and obtain the **MocoSolver** from the **MocoStudy**. Set the number of mesh intervals (this is related to the number of time points over which the problem is discretized), and the numerical tolerance on the objective function and constraint functions.

Part 1e

```
solver = study.initCasADiSolver();
solver.set_num_mesh_intervals(25);
solver.set_optim_convergence_tolerance(1e-4);
solver.set_optim_constraint_tolerance(1e-4);
```

If using Moco version 0.4.0 or earlier, replace **moco.initCasADiSolver()** with **study.initCasADiSolver()**, as shown above.

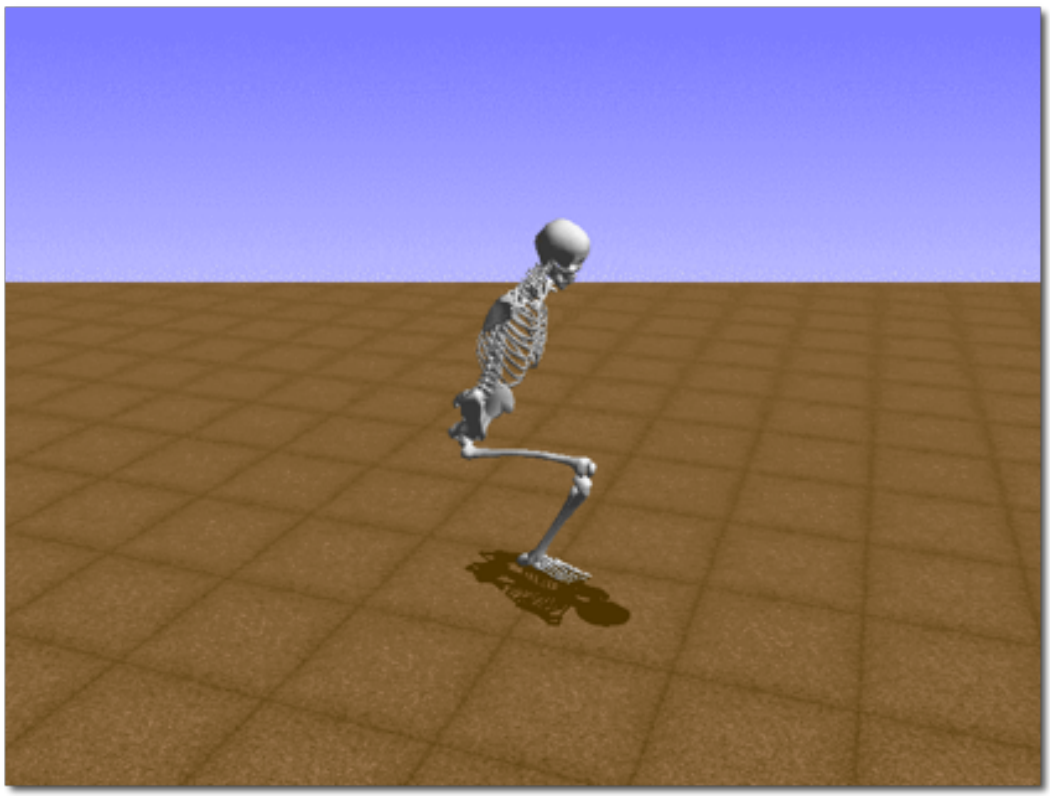
Moco provides two solvers: **MocoTropterSolver** and **MocoCasADiSolver**; here, we use **MocoCasADiSolver**, which supports parallelization and leverages the CasADi library. Internally, solvers employ the Ipopt nonlinear solver.

The problem is now fully configured. Solve the problem, save the solution to a Storage (tab-delimited) file, and visualize the solution trajectory using the Simbody Visualizer.

Part 1f

```
predictSolution = study.solve();
predictSolution.write('predictSolution.sto');
study.visualize(predictSolution);
```

Run your code for Part 1. Here is a snapshot of the visualization that appears:



Moco predicted this motion without tracking any motion data!

Part 2: Torque-driven Tracking Problem

Next, we will show how to solve a motion tracking problem with Moco. Specifically, we will track the motion we just predicted and ensure that the resulting joint moments match our original (predicted) joint moments. This process gives us confidence that Moco can produce the correct forces in when tracking experimental data.

Load and filter the predicted motion. The **TableProcessor** accepts a base table and allows appending operations to modify the table. Note that the operations are not actually performed until **tableProcessor.process()** is invoked (Moco will invoke this function internally).

Part 2a

```
tableProcessor = TableProcessor('predictSolution.sto');
tableProcessor.append(TabOpLowPassFilter(6));
```

Add a goal to track the predicted motion, using **MocoStateTrackingGoal**. Enable the **setAllowUnusedReferences()** setting to ignore the controls in the predictive solution.

Part 2b

```
tracking = MocoStateTrackingGoal();
tracking.setName('mytracking');
tracking.setReference(tableProcessor);
tracking.setAllowUnusedReferences(true);
problem.addGoal(tracking);
```

Reduce the control goal weight so the goal acts as only a regularization term (e.g., a term that doesn't significantly affect the final objective value but aids convergence).

Part 2c

```
problem.updGoal('myeffort').setWeight(0.001);
```

Set the initial guess to be the predictive solution from Part 1. Tighten the convergence tolerance to ensure the resulting controls are smooth.

Part 2d

```
solver.setGuessFile('predictSolution.sto');
solver.set_optim_convergence_tolerance(1e-6);
```

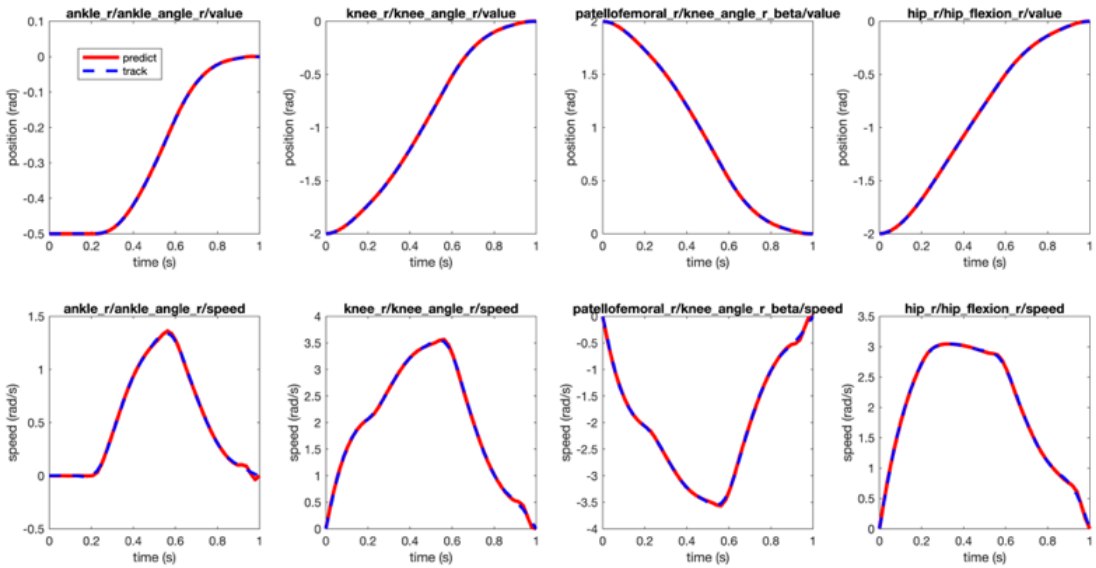
Solve the tracking problem, write the solution to a file, and visualize the solution.

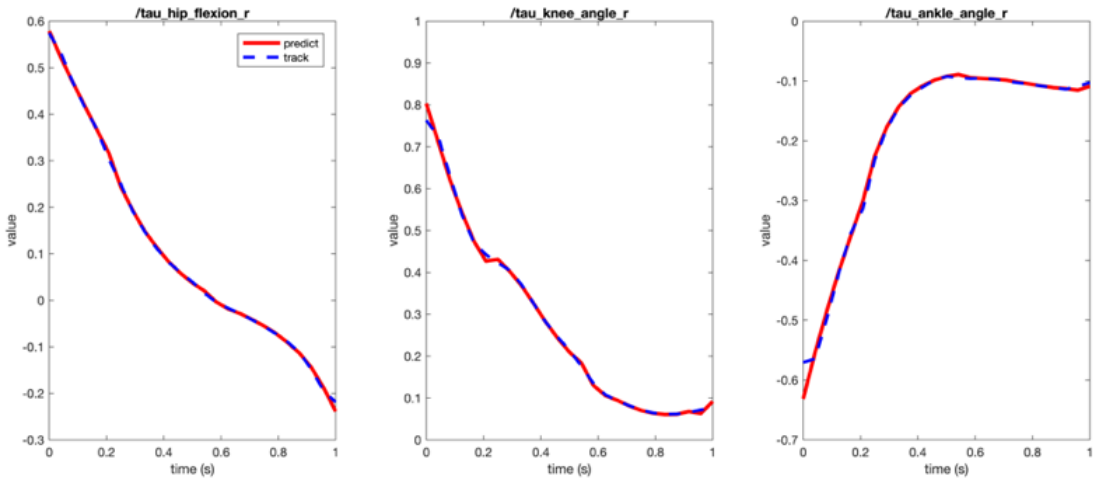
Part 2e

```
trackingSolution = study.solve();
trackingSolution.write('trackingSolution.sto');
study.visualize(trackingSolution);
```

Part 3: Compare Predictive and Tracking Solutions

You have now generated a predictive solution, **'predictSolution.sto'**, and a tracking solution, **'trackingSolution.sto'**. Use mocoPlotTrajectory() to compare the two solutions; they should match closely:





This result gives us confidence that tracking a motion can recover the original actuator forces that generated the tracked motion.

Part 4: Muscle-driven Inverse Problem

This section shows how to use Moco to solve a problem in which motion data is prescribed exactly—the model cannot deviate from the provided motion. This type of problem, which we solve with the **MocoInverse** tool, excludes multibody dynamics and is therefore more computationally efficient that solving a problem that includes multibody dynamics. In this case, we will use **MocoInverse** with a muscle-driven model to estimate muscle forces that could have generated the motion we predicted in Part 1.

Create a **MocoInverse** tool:

Part 4

```
inverse = MocoInverse();
```

Provide the MocoInverse tool with a muscle-driven model. The **ModelProcessor** can add operations to modify a base model, such as adding reserve actuators (CoordinateActuators) to the model.

Part 4a

```
modelProcessor = ModelProcessor(muscleDrivenModel);
modelProcessor.append(ModOpAddReserves(2));
inverse.setModel(modelProcessor);
```

Set the reference kinematics using the same **TableProcessor** we used in the tracking problem.

Part 4b

```
inverse.setKinematics(tableProcessor);
```

Set the time range, mesh interval, and convergence and constraint tolerances.

Part 4c

```
inverse.set_initial_time(0);
inverse.set_final_time(1);
inverse.set_mesh_interval(0.05);
inverse.set_convergence_tolerance(1e-4);
inverse.set_constraint_tolerance(1e-4);
```

If using Moco version 0.4.0 or earlier, replace **inverse.set_tolerance()** with the two tolerance lines shown above.

The example code already contains additional lines to further configure the **MocoInverse** tool:

```
% Allow extra (unused) columns in the kinematics and minimize activations.
inverse.set_kinematics_allow_extra_columns(true);
inverse.set_minimize_sum_squared_activations(true);

% Append additional outputs path for quantities that are calculated
% post-hoc using the inverse problem solution.
inverse.append_output_paths('.*normalized_fiber_length');
inverse.append_output_paths('.*passive_force_multiplier');
```

If using Moco version 0.4.0 or earlier, replace **inverse.set_minimize_sum_squared_states(true)** with **inverse.set_minimize_sum_squared_activations(true)** as shown above.

Solve the **MocoInverse** problem and write the resulting **MocoSolution** to a file.

Part 4d

```
inverseSolution = inverse.solve();
inverseSolution.getMocoSolution().write('inverseSolution.sto');
```

You can get the outputs that **MocoInverse** calculated from the inverse solution (we won't use this; this is just for reference).

Part 4e

```
inverseOutputs = inverseSolution.getOutputs();
STOFileAdapter.write(inverseOutputs, 'muscleOutputs.sto');
```

Part 5: Muscle-driven Inverse Problem with Passive Assistance

In the final part of this example, we modify the previous **MocoInverse** tool to reduce the effort to perform the squat-to-stand via a passive device—a torsional spring at the knee.

Create a spring to assist the knee joint, set the stiffness of the spring to 50 N-m/rad, and add the spring to the model.

Part 5a

```
device = SpringGeneralizedForce('knee_angle_r');
device.setStiffness(50);
device.setRestLength(0);
device.setViscosity(0);
muscleDrivenModel.addForce(device);
```

Create a new **ModelProcessor** using the assisted model; tell **MocoInverse** to use this new **ModelProcessor**.

Part 5b

```
modelProcessor = ModelProcessor(muscleDrivenModel);
modelProcessor.append(ModOpAddReserves(2));
inverse.setModel(modelProcessor);
```

Solve the MocoInverse problem again, reusing the settings we applied previously. Write the solution to a file.

Part 5c

```
inverseDeviceSolution = inverse.solve();
inverseDeviceSolution.getMocoSolution().write('inverseDeviceSolution.sto');
```

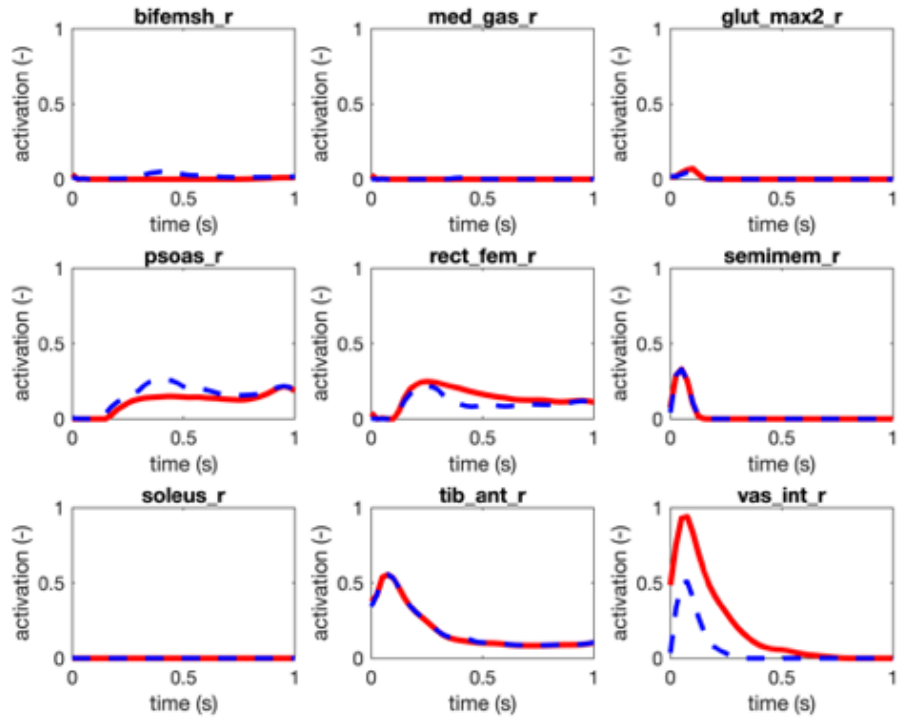
Part 6: Compare unassisted and assisted Inverse Problems

We have now solved for muscle activity without and with an assistive device. Does the assistive device reduce the muscle effort required to achieve the motion we originally predicted? The example provides code to print the objective function value without and with the assistive device, and to plot the unassisted and assisted solutions.

Part 6

```
fprintf('Cost without device: %f\n', ...
        inverseSolution.getMocoSolution().getObjective());
fprintf('Cost with device: %f\n', ...
        inverseDeviceSolution.getMocoSolution().getObjective());
% This is a convenience function provided for you. See below for the
% implementation.
compareInverseSolutions(inverseSolution, inverseDeviceSolution);
```

This code generates the following result:



As expected, the spring at the knee reduces the activity of vastus intermedius muscle.

Bonus

Explore how changes to the model or the MocoProblem affect your results. Here are some ideas:

1. Perform Parts 1 and 2 using the muscle-driven model instead of the torque-driven model. Does the optimization take longer to solve? Does the model require more time to reach its standing position?
2. Add additional goals to the cost.
3. Change the mass properties of the model.