

I. Objective

Methods for estimating muscle forces from experimental data are an essential part of the biomechanist's simulation toolkit. Static optimization is a widely used tool for estimating muscle activity due to its speed and ease-of-use, and many OpenSim users rely on the [Static Optimization Tool](#) as a key part of their simulation research pipelines. Despite its popularity, OpenSim's implementation of this method only permits one type of cost function and a fixed set of constraints, and therefore is only applicable to research questions that can accept these limitations. In this lab, you will write your own code to better understand the elements required to solve the static optimization problem, draw comparisons to OpenSim's implementation, and learn how extend the basic tool to fit new research applications.

By working through this lab, you will:

- Learn the basics of the static optimization problem, including OpenSim's implementation and alternative implementations.
- Become familiar with using the OpenSim API through MATLAB.
- Learn how to write your own static optimization code.

II. Background

Problem Definition

The goal of static optimization is to solve for muscle activations that produce the dynamics of an observed motion. Since there are more muscles than degrees-of-freedom in the human body, this problem is "non-unique" (i.e., many possible solutions exist), hence the need for optimization. We often describe this as the "muscle redundancy problem". We use the term "static" since no dynamics appear in the optimization problem: activation dynamics are ignored, tendons are assumed rigid, and multibody dynamics are prescribed via motion data. Most static optimization problems take the following form:

$$\begin{aligned} &\text{minimize} \quad \sum_{i=1}^{N_m} (a_i)^p \\ &\text{subject to} \quad M_j = \sum_{i=1}^{N_m} r_{i,j} * f_i^m && \text{for } j = 1 \dots N_q \\ & \quad f_i^m = F_i^{\max} (a_i f_l(\tilde{l}_i^m) f_v(\tilde{v}_i^m) + f_p(\tilde{l}_i^m)) && \text{for } i = 1 \dots N_m \\ & \quad 0 \leq a_i \leq 1 && \text{for } i = 1 \dots N_m \end{aligned}$$

where

a_i – activation [Pages](#) / ... / [Advanced Examples](#)

p – positive integer

M_j – net joint moment about j^{th} joint

$r_{i,j}$ – moment arm for i^{th} muscle about j^{th} joint

f_i^m – force in i^{th} muscle

F_i^{max} – max isometric force in i^{th} muscle

\tilde{l}_i^m – normalized fiber length in i^{th} muscle

\tilde{v}_i^m – normalized fiber velocity in i^{th} muscle

f_l – active fiber force-length curve

f_v – active fiber force-velocity curve

f_p – passive fiber force-length curve

N_m – number of muscles

N_q – number of coordinates

The labels "minimize" and "subject to" denote the problem cost function and constraints, respectively. The muscle activations are the only problem "design variables", or the values that the optimizer can change to minimize the cost and satisfy the constraints. Net joint moments are typically computed from inverse dynamics, and inverse kinematics is used to compute all muscle kinematic states.


OpenSim Implementation

OpenSim's Static Optimization Tool is similar to the problem definition above (see [How Static Optimization Works](#)). OpenSim uses a similar cost function where activation raised to a user-defined power is minimized. Typically, an activations-squared cost function ($p = 2$) is used for most research applications. However, OpenSim's implementation does not include passive muscle contributions when computing total muscle force, f^m . Lastly, rather than matching muscle-generated moments to net joint moments from inverse dynamics, OpenSim's StaticOptimizationTool requires that joint accelerations generated from muscle forces match joint accelerations computed from motion data. In this lab, we will use the moment-matching approach, but think about how you would solve the acceleration-matching problem as you work on your code.

(Optional) Optimization Theory

One important property of static optimization is that the problem is *linear* in its design variables. While the characteristic curves that define muscle forces are non-linear, muscle activations are the only design variables in the problem and appear linearly in the muscle-generated joint moment equations. If possible, it is beneficial to write optimization problems with linear constraints only, as these problems can usually be solved more efficiently than problems with non-linear constraints. Linear constraints are so advantageous that entire subfields in optimization theory are dedicated to solving problems with this form (see [Linear Programming](#) and [Quadratic Programming](#) to learn more).

III. Materials

We've provided materials to help you get started writing your own optimization code. You may click on the links provided here, or click the menu icon, , in the top right of the page and "Attachments" to download all files at once. Place all the materials in a common working directory, preferably something not write-protected (e.g., C:\Users\<account_name>\CustomStaticOptimization).

- Files from the [Rajagopal model distribution](#):
 - [subject_walk_adjusted.osim](#) – RRA-adjusted Rajagopal2016 model.
 - [grf_walk.mot](#) – Ground reaction force and torque data.
 - [grf_walk.xml](#) – ExternalLoads to be added to the model to apply ground reaction loads.
 - [coordinates.mot](#) – Coordinate values from an inverse kinematics solution.
- [loadFilterCropArray.m](#) – Utility to load STO files into MATLAB array, filter, and crop to a specified time range. This function relies on the osimTableToStruct MATLAB utility that comes with OpenSim.

i If you don't have MATLAB scripting with OpenSim set up already, refer to [Scripting with Matlab](#).

IV. Coding Static Optimization

In this section, you will code your own version of the static optimization problem described in the Background section. Write your problem in CustomStaticOptimization.m included in the materials. To guide you, we've structured CustomStaticOptimization.m to be completed in individual parts, which are labeled via comments in the starter code (e.g., Part 1). The sections you need to fill in will be denoted with commented brackets like the following:

```
% TODO {  
  
% }
```

Before you start coding, first look through the whole file and the parts you need to fill in to get a high-level view of the assignment. You may then continue reading through this section for more guidance on the code sections you need to complete.

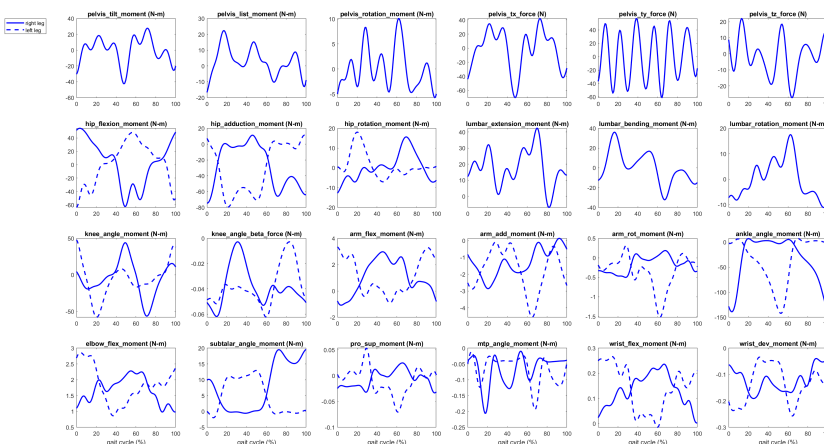
Part 1: Fill in the missing InverseDynamicsTool commands.

To get started, you will need to fill in the missing OpenSim API commands to compute the net joint moments. Refer to the Doxygen page for the [InverseDynamicsTool](#) to find the missing commands you'll need.

i If you have never used Doxygen or the API, refer to [Guide to Using Doxygen](#) and [Introduction to the OpenSim API](#). You can also type 'methodsvIEW InverseDynamicsTool' into the MATLAB command window to get a list of available commands.

Part 2: Plot and inspect the generalized forces from inverse dynamics.

Before you continue on to the next steps, compare the generalized forces (i.e., the net joint moments, M_j) from inverse dynamics to the plot below to make sure you have computed them correctly. Note that these results are specific to the model and data included with this example.



Click on this plot to enlarge and download.

Part 3: Remove any unneeded generalized force data columns.

Your static optimization solution should provide a reasonable estimate of muscle activity for walking. Therefore, we should only solve the problem across degrees-of-freedom with reliable experimental measurements that also

contribute to the muscle act
interested in lower extremity [Pages / ... / Advanced Examples](#)

coordinates from the optimi
freedom that may have poor inverse kinematics solutions. Fill in the 'forcesToRemove' array as instructed in the CustomStaticOptimization.m file to remove unnecessary forces.

Part 4: Store max isometric force values and disable muscle dynamics

The model provided for you includes 80 lower-extremity [Millard2012EquilibriumMuscles](#). We need to store an array of max isometric force values so we can compute total muscle force in the optimization. We will also disable activation and tendon compliance dynamics in each muscle, since static optimization does not include muscle dynamics. For a list of the available API commands for the muscles to complete these tasks, see the [Millard2012EquilibriumMuscle API docs](#) [🔗](#).

i Since OpenSim returns muscles retrieved from the model ForceSet as the base type Muscle, we need to "downcast" each muscle object to the Millard2012EquilibriumMuscle type. To visualize downcasting, open the inheritance diagram at the top of the Millard2012EquilibriumMuscle API page and you'll see the Millard muscle below the base Muscle class. This shows that Millard2012EquilibriumMuscle "inherits" from Muscle, meaning that Millard2012EquilibriumMuscle can use all the commands and properties defined in Muscle (*mostly, this is a simplification, see [Inheritance](#) [🔗](#) and [Inheritance in C++](#) [🔗](#)). You need to downcast to the Millard2012EquilibriumMuscle type if you want to use the commands specific to that type in other places in your code.

Part 5: Perform static optimization.

Use FMINCON, MATLAB's constrained optimization solver, to solve the static optimization problem with an activations-squared cost function (i.e., $\sum_{i=1}^{N_M} (a_i)^2$). This is the main part of the coding assignment and will likely take the most time to complete. There is some starter code to select which data points to solve the problem on and to construct the FMINCON options structure; the rest is up to you!

Some things to consider:

- Try outlining your problem in pseudocode before writing any actual MATLAB code.
- Which elements of the problem are the same across all time steps and which elements need to be computed at each time step?
- Similarly, which computations need to happen "inside" the optimization, and which can be pre-computed?
- How will you implement the problem cost function? The first argument to FMINCON accepts a [MATLAB function handle](#) [🔗](#), which is a MATLAB variable type that represents a function. Function handles can be the typical named functions or [anonymous functions](#) [🔗](#). Note that the function handle passed to FMINCON can only be a function of the problem design variables.
- How will you implement the problem constraints? Consider the discussion from the Background section.

Part 6: Plot results.

Plot your muscle activation solution and any other relevant outputs.

V. Follow-up Questions

1. Describe the muscle activation solution you plotted in Part 6. Does your solution match muscle activity levels for normal walking? Compare your solution to electromyography (EMG) data from the literature. [What are potential explanations for any discrepancies?](#)
2. Provide 3 research questions that you could use your new static optimization code to answer. For each question, briefly describe any additions or changes to the data, model or code necessary.
3. Reflection questions.
 - a. Approximately how long did it take you to complete this assignment?
 - b. Was the assignment too hard/too easy?
 - c. Do you think the skills learned were mostly worth the time invested in this assignment?
 - d. If you could make one improvement to this assignment for future classes, what would it be?

