Author: Mohammad Kassas
Date: 2/1/2019

Overview:

We are designing a harness that can load and unload the body to study balance control. The harness will be attached to a balance robot that will simulate the standing balance control of humans. Because our experiments are on the order of hours, we require these harnesses to both safely and comfortably load and unload the body for that period of time.

Fig1a&b: unloading and loading harness with the stiffness labels. Refer to "our model is parametrized by" to see what k1, k2, and ks represent.

We hypothesize that a harness that evenly distributes loads between the attachment points to the body will be the most comfortable. To study this, we need to first optimize a harness design that achieves even load distributions over a range of body configurations of interest.

To model load distributions, we used an OpenSim musculoskeletal model of the full body with a simulated harness. We chose OpenSim because it is open source and easily extensible to additional scenarios outside the current scope of the study. Briefly, we used a full body OpenSim musculoskeletal model (Gait10dof18musc.osim) and attached additional rigid body and spring elements to represent the harness, with attachment points at the hips, thighs, and top of the shoulders. We additionally removed all the muscles and locked the joints to create a single rigid body model. We can unlock these joints (e.g. hip flexion) to increase the complexity of the model for future investigations.

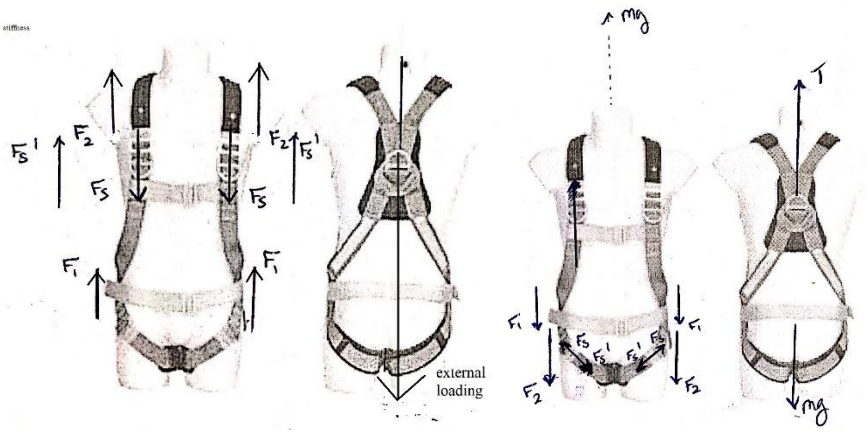
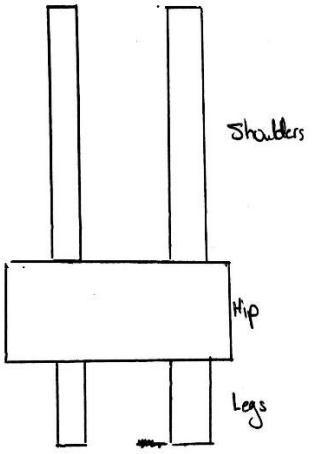


Fig2&3

We are modeling two scenarios: loading (left) and unloading (right). Loading involves the shoulders and hip. Unloading involves the legs and hip. To model additional harness designs with other attachment points, users can add additional rigid body segments and/or spring elements attached to the body to model additional harness sections and body tissue interactions respectively.

The reason the reaction forces of Fs are included is because we section the harness through that legs and the shoulder straps, getting two systems. F2 and Fs are one system; F1 and Fs' are another. F1 represents the hip-body force. F2 is the shoulder-body force for the left image, while being the leg-body force for the right image. Fs is the strap force. (refer to fig2&3)



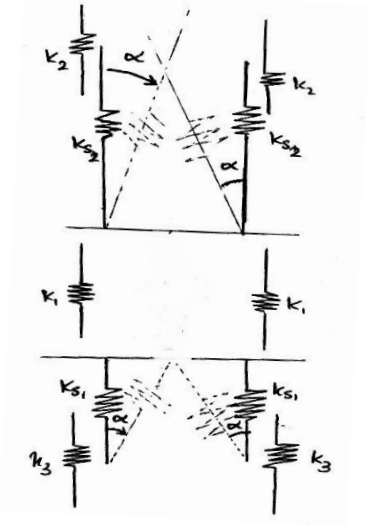


Fig4 *Fig5*

Our model is parameterized by:

1) Total body mass: m in mg (fig2&3)
2) Soft tissue material properties (Young's modulus or stiffness) at the attachment points: k1 and k2 in fig1a&b
3) Harness material properties (Young's modulus or stiffness) connecting harness sections (e.g. hip to shoulder or hip to legs): ks in fig1a&b.
4) Angle of shoulder-hip or leg-hip straps relative to the vertical.

In the fig4, we used blocks to reperesent the different parts of the harness. Fig5 shows the spring model of the harness. The springs that seem to float are the ones that take into account the forces between harness and body tissue.

For our simulations, we fixed

1. Total body mass (73.062 kg)
2. Soft tissue material properties
    a. hip-body strap stiffness k1: 1000 N/m
    b. shoulder-body strap stiffness k2 (fig1b): 3000 N/m
    c. leg-body strap stiffness k2 (fig1a): 3000 N/m

Required software:

1) OpenSim 3.3 https://simtk.org/frs/index.php?group_id=91
2) Python 3.6 (we recommend downloading the Anaconda package: https://www.anaconda.com/distribution/)

Downloading OpenSim:

1) setup a simTk account
2) https://simtk.org/frs/index.php?group_id=91
3) go to previous releases and select 3.3
4) download the 64-bit version

Ensure that the Python environment and IDE are 64-bit to match the OpenSim 64-bit version

Linking libraries:

follow the instructions:
https://simtk-confluence.stanford.edu/display/OpenSim/Scripting+in+Python
To test if the linking of libraries worked, in your command window type "import opensim as osim"
If there are no errors, great; it works!
In the case where you get an import error, follow the troubleshooting instructions found in the above link
If an import error occurs, that means that you either have an inconsistent bit version of either program or you need to "tell" your Python Environment where to locate the OpenSim program on your computer.

Repository Folder Layout:

1. hangingHarnessModel: 2 unloading models with optimizer and assistive functions
    a. hangingHarnessModel: simplest unloading model → **unloadingOptimizer**
    b. hangingHarnessModel_limitedHip: has a "CoordinateLimitForce" that helps account for the discomfort from the hip harness pushing onto the rib cage → **unloading_angle_optimizer_limitedHip**
2. loadingHarnessModel: 2 loading model with optimizer and assistive functions:
    a. loadingHarnessModel: simplest loading model → **loadingOptimizer**
    b. loadingHarnessModel_limitedHip: has a "CoordinateLimitForce" that helps account for the discomfort from the hip harness pushing onto the iliac crest → **loadingOptimizer_limHip**
3. friction: contains 4 loading and unloading models that allow optimization for the padding material of the harness by adding friction forces in the model.
    a. hangingHarnessModel_fric: same as 1.a with an additional frictional force that helps estimate the ideal padding material to use in the hip harness. → **fric_optimizer**
    b. hangingHarnessModel_fric_lim: same as 1.b with an additional frictional force that helps estimate the ideal padding material to use in the hip harness. → **fric_lim_opt & loading_fric_lim_angle_opt**
    c. loadingHarnessModel_fric: same as 2.a with an additional frictional force that helps estimate the ideal padding material to use in the hip harness. → **fric_optimizer**

d. loadingHarnessModel_fric_lim: same as 2.b with an additional frictional force that helps estimate the ideal padding material to use in the hip harness. → **fric_lim_opt & loading_fric_lim_angle_opt**

4. all_opt: contains modularized code and optimizers that change the pelvis tilt angle
    a. hangingHarnessModel_horiz_lim_hip: same as 1.b in addition to horizontal forces that model the new weight distribution due to pelvis tilting. → **optimizer_pelvis_tilt_horiz**
    b. hangingHarnessModel_lim_hip: same as 1.b → **optimizer_pelvis_tilt & optimizer_pelvis_tilt_angle**
    c. loadingHarnessModel_horiz_lim_hip: same as 2.b in addition to horizontal forces that model the new weight distribution due to pelvis tilting. → **optimizer_pelvis_tilt_horiz**
    d. loadingHarnessModel_lim_hip: same as 2.b → **optimizer_pelvis_tilt & optimizer_pelvis_tilt_angle**

    optimizers is a general optimizer that can be used with any of the models within the same directory.

Functions:

1. Optimizer_callBack: runs a simulation and returns the result of the cost function
2. External_tension: call back function to set the stiffness of the external cable carrying the model during unloading.
3. Shoulder_hip_right: call back function to set the stiffness of the right shoulder-hip strap
4. Shoulder_left_right: call back function to set the stiffness of the left shoulder-hip strap
5. Hip_body_right: call back function to set the stiffness of the right-side strap representing the reaction of hip and tissue
6. Hip_body_left: call back function to set the stiffness of the left-side strap representing the reaction of hip and tissue.
7. Shoulder_body_right: call back function to set the stiffness of the right-side strap representing the reaction of shoulder and tissue.
8. Shoulder_body_left: call back function to set the stiffness of the left-side strap representing the reaction of shoulder and tissue.
9. Angle_shifter: sets the angle of the straps
10. Length_calc: calculates the length of the straps
11. Normalize: returns an array that represents a Euclidian vector with respect to the new reference frame
12. Refactor: returns an array that represents a Euclidian vector with respect to the original reference frame

*note: functions 2 through 8 are dependent on the index of the forces in the force set of the model. The indices are hard-coded into these functions.

In “horiz optimizer”, there are four functions:

1. coordinates: respsonsible for housing, normalizing, and refactoring all the coordinates of the bodies and forces

2. geometry_func: contains sub-programs that change and retrieve the geometry of the straps and springs (angle, length, and GeometryPath)
3. model_returner: returns an OpenSim model, a copy of that model, a file path, the indices of the forces, the forceSet, and whether it is loading or unloading. Moreover, this function performs all the file I/O to create a copy of the model, so that any alterations would not affect the original model. If you are changing parameters, ensure the paths are consistent and the indices returned align properly to the column indices of the force reporter file (f1Index and f2Index). If you are adding additional complexity to the model or changing the entire model, make sure to have the correct forceSet indices be returned (frcSet). The frcSet returns the indices of force 1 and force 2 (hip-strap and shoulder-strap reaction resp.)
4. vector_operations: normalizes and refactors the coordinates of the springs to any specified coordinate.

Scipy minimize & optimizer_callBack():

The function takes an array of parameters as in input. Values of the array are varied and simulated until the scipy minimize functions finds the ideal solution for the provided cost function that is found at the end of the call back function as the return statement. For example, if the elements of the parameter array are the strap stiffness (ks) and strap angle (alpha), the callback sets the angles and stiffnesses of the OpenSim model to the current parameter set. It then runs a forward dynamics tool with a force analysis for 3 seconds, which we found was sufficient to reach steady-state. From the force analysis, we extract the forces on the body segments (shoulder/legs and the torso), in the straps, and on the external loading cable. In the simplest case the cost function is $(F1 - F2)^2$, so we simply use the body segment forces to compute and return the cost.

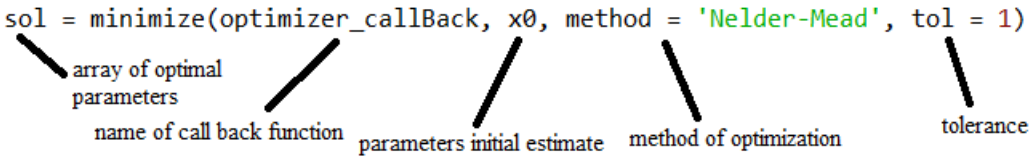


Fig6: shows the scipy minimize function; how it is called and what parameters to include. This function returns an array of values that represent the ideal parameters for the cost function. The size of x0 must match the number of parameters being varied. We found the Nelder-Mead optimization method to be the fastest (partly because it does not optimize based on gradient). Tolerance allows the optimizer to find a solution within a range of ±tol.

<u>Examples of cost functions and x0's:</u>

1. cf: $(F1 - F2)^2 + x[1]^2$ // x0: [stiffness value, strap angle value]

   this minimizes the difference between F1 and F2 (making them as equal as possible) and minimizes the strap angle at the same time. The optimizer will find the best pair of parameters to minimize both terms of the cost function.

2. cf: $(F1 - F2)^2 + (100 * x[1]^2)$ // x0: [stiffness value, strap angle value]

   this will do the same as (1); however, it will favor the parameter variation towards minimizing the value of the angle. Due to the weight (coefficient in front of the term) of $x[1]^2$, having a minimum strap angle is more important than the closeness of F1 and F2's values.

3. cf: $x[0]^2 - x[1]^2$ // x0: [friction value, strap angle value]

   this will minimize the vale of friction and minimize that of the strap angle. Since both weights are equal, both have equal importance to the optimizer.

   Weight of a cost is directly proportional to its importance to the optimizer.

   The closer the initial guess, x0, is to the actual answer; the better and faster is the optimizer

Interchanging models with optimizers:

Every optimizer should use the models within the same directory.

There are two cases of interchanging:

1. angle being optimized:
   use the optimizer and model corresponding to the same configuration (loading model and a loading optimizer). The names of the files should make it clear which optimizer corresponds to which model
2. angle not being optimized:
   simply change the path of one model to another model within the same directory that has the mirrored features of the model being switched out (e.x. friction force and limited hip).
   Two models are considered replaceable with each other if:
   1. They have the same number of forces in force set
   2. Forces (hip-body, leg/shoulder-body, hip limit, etc…) are in the same order in the force sets of each model.
   3. All the bodies are the same
   4. Force reporters have the same column order similar to (2)
   5. Only difference is leg harness forces vs. shoulder harness forces

Refer to *optimizers.py*, *optimizers_pelvis_tilt.py*, and optimizer*_pelvis_tilt_horiz.py* under *all_opt* for clearer separation between loading and unloading models

Using the OpenSim GUI:

1. Load the OpenSim model (.osim file extension) from the file tab
2. Only the pelvis_ty, hip_ty, legR_ty, and legL_ty coordinates should be unlocked.
   a. In the case of loading, only the pelvis_ty, hip_ty, shR_ty, and shL_ty coordinates should be unlocked.
3. In the Tool tab, select Forward Dynamics.
4. In the pop window, check “solve for equilibrium states”
5. Go to analyses tab, and select add “Force analysis”
6. Press “run”
7. After running the forward dynamics simulation, the force reporter file will be produced in the same directory as your OpenSim model. The file name starts with the name of your “modelName_ Force_Reporter_forces.sto”
   a. Other outputs:
      i. “modelName_controls.sto” will be empty in our case
      ii. “modelName_states.sto” will contain the initial states of the model

iii. “model_Name_states_degrees.mot” is a motion file that has the angles of certain bodies of the model.

If the model being loaded contains actuators (models with friction forces), a controls file must be loaded; otherwise, the friction forces will be zero by default. Load the controls file between steps 4 and 5. You can also adjust the activation values by pressing on the edit (pencil) icon next to the controls file path.

*note: the “modelName” prefix is only added to the name of the file if the simulation is run manually through OpenSim. In the case of running the simulation through Python, it will be “_ForceReporter_forces.sto”.

OpenSim components/methods:
For the adept user, included are some notes on the most relevant OpenSim methods and classes. For more information on interfacing with OpenSim, visit
https://simtk.org/api_docs/opensim/api_docs20b/main.html

a) import opensim as osim: imports all the necessary OpenSim methods
b) osim.Model("..."): loads in the OpenSim model that will be simulated
c) model.getForceSet(): gets all the added external forces that are acting on model
d) model.getForceSet().get(0): gets the base of the first force from the forceSet()
e) PathSpring is a type of spring in OpenSim that only produces force during tension (similar to a string)
f) osim.PathSpring.safeDownCast(model.getForceSet().get(0)): this down casts the force to a PathSpring --> having access to all the PathSpring functions and attributes.
g) <PathSpring Object>.setStiffness(double): sets the stiffness of the spring
h) model.initSystem(): sets up the initial states of the model before simulation
i) osim.ForceReporter(<Model Object>): tells the forward tool to produce a force file after running
j) model.addAnalysis(reporter): adds the force reporter to the list of analyses performed during the simulation
k) osim.ForwardTool(): specifies the type of simulation to be a forward dynamics simulation
l) setModel(<Model Object>): sets the model which will be simulated
m) setFinalTime(double): specifies the end time of the simulation
n) osim.Vec3(double, double, double): produces a 3-D vector
o) <PathSpring Object>.setRestingLength(double): sets the resting length of the spring
p) upd_GeometryPath(): returns a GeometryPath Object
q) <GeometryPath Object>.updPathPointSet(): returns the set of pathpoints that define the geometry of the spring
r) <PathPoint Object>.setLocation(SimTK init State, Vec3): changes the coordinates of the PathPoint

important information:
PathSprings only produce tension forces (no compression)
PathPoint coordinates are specified in reference to parent bodies. In order to find lengths or manipulate different springs and points, one must normalize everything to a single origin (parent body)