SimTK Documents

An Introduction to OpenSim

November 16, 2010

SIMPAR 2010 | Darmstadt, Germany

OpenSim Workshop Agenda

Tuesday, November 16, 2010

8:30am – 8:45am	Welcome and goals of workshop – *Sam Hamner & Massimo Sartori*
8:45am – 9:00am	Introduction to the GUI – *Sam Hamner*
9:00am – 10:00am	Guided GUI example and exploration on your own – *Sam & You*
10:00am – 10:30am	Break
10:30am – 10:45am	Introduction to the API – *Massimo Sartori*
10:45am – 11:55am	Guided API example and exploration on your own – *Massimo & You*
11:55am – 12:00pm	Closing Remarks – *Sam & Massimo*

Trademarks and Copyright and Permission Notice

Acknowledgments

OpenSim was developed as a part of SimTK and funded by the Simbios National Center for Biomedical Computing through the National Institutes of Health and the NIH Roadmap for Medical Research, Grant U54 GM072970. Information on the National Centers can be found at http://nihroadmap.nih.gov/bioinformatics.

Table of Contents

1 Introduction to the GUI

1.1 What is the OpenSim GUI?

OpenSim is a freely available software package that enables you to build, exchange, and analyze computer models of the musculoskeletal system and dynamic simulations of movement. OpenSim version 1.0 was introduced at the American Society of Biomechanics Conference in 2007. Since then, many people have begun to use the software in a wide variety of applications, including biomechanics research, medical device design, orthopedics and rehabilitation science, neuroscience research, ergonomic analysis and design, sports science, computer animation, robotics research, and biology and engineering education.

The software provides a platform on which the biomechanics community can build a library of simulations that can be exchanged, tested, analyzed, and improved through multi-institutional collaboration. The underlying software is written in C++ and the graphical user interface (GUI) is written in Java. OpenSim plug-in technology will make it possible to develop customized controllers, analyses, contact models, and muscle models among other things. These plug-ins can be shared without the need to alter or compile source code. You can analyze existing models and simulations and develop new models and simulations and visualize them within the GUI.

OpenSim is built on top of SimTK, an open-source simulation toolkit developed to create mathematical models of biological dynamics. SimTK is being developed by Simbios, an NIH National Center for Biomedical Computation based at Stanford University. Open-source, third-party tools are used for some basic functionality, including the Xerces Parser from the Apache Foundation for reading and writing XML files (xml.apache.org/xerces-c) and the Visualization Toolkit (VTK) from Kitware for visualization (www.vtk.org). Use of plug-in technology allows low-level computational components such as integrators and optimizers to be updated as appropriate without extensive restructuring.

1.2 Capabilities

OpenSim includes a wide variety of features to support modeling, simulation and analysis of musculoskeletal models. You can find out about these features by completing the tutorials and browsing the user guide and this handout. Some of the most useful features are:

- Scaling a Musculoskeletal Model
- Performing Inverse Kinematics Analyses
- Performing Inverse Dynamics Analyses
- Performing Static Optimization Analyses
- Generating Forward Dynamics Simulations
- Analyzing Dynamic Simulations
- Plotting Results
- Creating Snapshots and Making Animations

1.3 Model and Simulation Repository

You can create your own models of musculoskeletal structures and dynamic simulations of movement in OpenSim, as well as take advantage of computer models and dynamic simulations that other users have developed and shared. For example, you can use existing computer models of the human lower limb, upper limb, cervical spine, and whole body which have already been developed and posted at https://simtk.org/home/nmblmodels. You can also use dynamic simulations of walking and other activities that have been developed, tested and posted on Simtk.org. We encourage you to share your models and simulations with the research community by setting up a project on SimTK.org.

1.4 Additional Resources and Help

You can learn more at the OpenSim project site at http://simtk.org/home/opensim. The project site provides a forum for users to ask questions and share expertise. You can also get additional information in the following article: Delp, S.L., Anderson, F.C., Arnold, A. S., Loan, P., Habib, A., John, C., Guendelman, E.G., Thelen, D.G., OpenSim: Open-source software to create and analyze dynamic simulations of movement. *IEEE Transactions on Biomedical Engineering*, vol. 54, no. 11, pp. 1940-1950, 2007.

The following chapters elaborate on some of the tools in both the GUI and API (Application Programming Interface) that will be used in the workshop.

2 Inverse Kinematics

2.1 The inverse Kinematics Problem:

The Inverse Kinematics Tool solves a problem faced by practically any user who has a (OpenSim) model and is trying to find the best set of joint angles that the model can assume that fits recorded experimental data. The experimental data is usually given in the form of marker trajectories, reported as a sequence of frames.

2.2 How It Works

The Inverse Kinematics Tool goes through each time step (frame) of motion and computes generalized coordinate values which position the model in a pose that "best matches" experimental marker and coordinate values for that time step. Mathematically, the "best match" is expressed as a weighted least squares problem, whose solution aims to minimize both marker and coordinate errors (if a guess is provided).

Figure 2-1: Inverse Kinematics Tool Overview. Experimental markers are matched by model markers throughout the motion by varying the generalized coordinates (e.g., joint angles) through time.

2.2.1 Marker Errors

A *marker error* is the distance between an experimental marker and the corresponding marker on the model (Figure 2-1) when the model is positioned using the generalized coordinates computed by the Inverse Kinematics solver. Each marker has a weight associated with it, specifying how strongly that marker's error term should be minimized.

2.2.2 Coordinate Errors

A *coordinate error* is the difference between an "experimental coordinate value" and the generalized coordinate value computed by the Inverse Kinematics Tool. Experimental coordinate values can be joint angles obtained directly from a motion capture system (i.e., built-in mocap inverse kinematics capabilities), or may be computed from experimental data by various specialized algorithms (e.g., defining anatomical coordinate frames and using them to specify joint frames that, in turn, describe joint angles) or by other measurement techniques that involve other measurement devices (e.g., a goniometer). A fixed desired value for a coordinate can also be a specified constant (e.g., if we know that a specific joint angle should stay at 0°). The inclusion of experimental coordinate values is optional; the Inverse Kinematics Tool can solve for the generalized coordinates using marker matching alone.

2.2.3 Weighted Least Squares Equation

The weighted least squares problem solved by the Inverse Kinematics Tool is

$$\min_{\mathbf{q}} \left[\sum_{i \in \text{markers}} w_i \left\| \mathbf{x}_i^{\text{exp}} - \mathbf{x}_i(\mathbf{q}) \right\|^2 + \sum_{j \in \text{unprescribed coords}} \omega_j \left(q_j^{\text{exp}} - q_j \right)^2 \right]$$

$$q_j = q_j^{\text{exp}} \text{ for all prescribed coordinates } j$$

where $\mathbf{q}$ is the vector of generalized coordinates being solved for, $\mathbf{x}_i^{\text{exp}}$ is the experimental position of marker i, $\mathbf{x}_i(\mathbf{q})$ is the position of the corresponding marker on the model (which depends on the coordinate values), and q_j^{exp} is the experimental value for coordinate j.

2.3 Inputs

Three files are required as input by the Inverse Kinematics Tool:

arm26_elbow_flex.trc: Experimental marker trajectories for a motion trial.

arm26_InverseKinematics_Tasks.xml: Contains the inverse kinematics tasks (i.e., a specification of which markers should be matched up during the inverse kinematics solution) and their relative weightings. Matching is based on names.

arm26.osim: The current model loaded in OpenSim

2.4 Outputs

The Inverse Kinematics Tool generates a single file:

arm26_InverseKinematics.mot: Motion file containing the time histories of generalized coordinates that describe the movement of the model.

2.5 Inverse Kinematics Tool

To launch the Inverse Kinematics Tool, select **Inverse Kinematics...** from the **Tools** menu. The **Inverse Kinematics Tool** dialog (Figure 2-2) like all other OpenSim tools, operates on the *Current Model* open and selected in OpenSim (e.g., *arm26*). Inverse kinematics requires that a marker set is associated with the model and the number of markers is reported (e.g., *3 markers*). The **IK Trial** section specifies the experimental marker data that the Inverse Kinematics Tool will match with the current model. A *Trial name* can be associated with the trial to uniquely identify the resultant motion. The *Marker data for trial* field must contain the path to the marker data (in *.trc* format) and OpenSim will report the information it recognizes from the file such as the number of markers, the number of frames and sampling frequency as well as the start and end times of the data set in the **Marker Data** pane. Any subset of the time range can be specified for performing inverse kinematics in the *Time range* field, but by default the complete time range is specified. If the *Coordinate data for trial* flag is checked, then the Inverse Kinematics Tool will require coordinate values specified in a motion (.mot) file to be loaded.

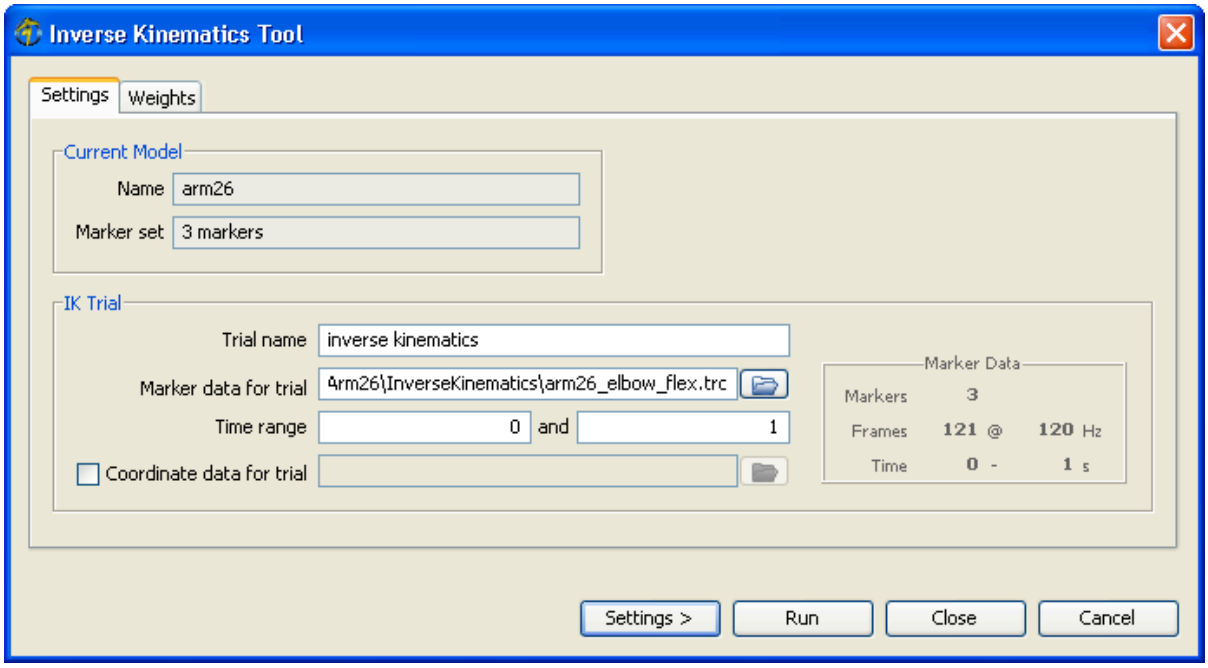


Figure 2-2: Inverse Kinematics Tool Dialog

Once a marker file, and possibly a coordinate file, are specified, the behavior of the Inverse Kinematics Tool can be modified under the ***Weights*** tab (Figure 2-3). Each entry in the table represents a weight in the least-squares equation for either a marker (top table) or a coordinate (lower table). By selecting a row (or multiple rows), the entry fields below the panes become editable allowing the marker(s) or coordinate(s) to be enabled and allowing the user to specify a weight. The weight value will affect to what degree a match should be satisfied with larger weights penalizing errors for that marker or coordinate more heavily and thus attempting to match the experimental value more closely. For coordinates, the coordinate value to be matched can come from a specified motion file or set to its default or a user-specified (*manual*) constant value.

When running the Inverse Kinematics Tool from the GUI, the results from inverse kinematics are not automatically saved to file but are associated with the model under the **Motions** category in the model Navigator. One can view multiple Inverse Kinematics results before saving to file. To save a motion, right click on the motion in the Navigator and select "Save as."

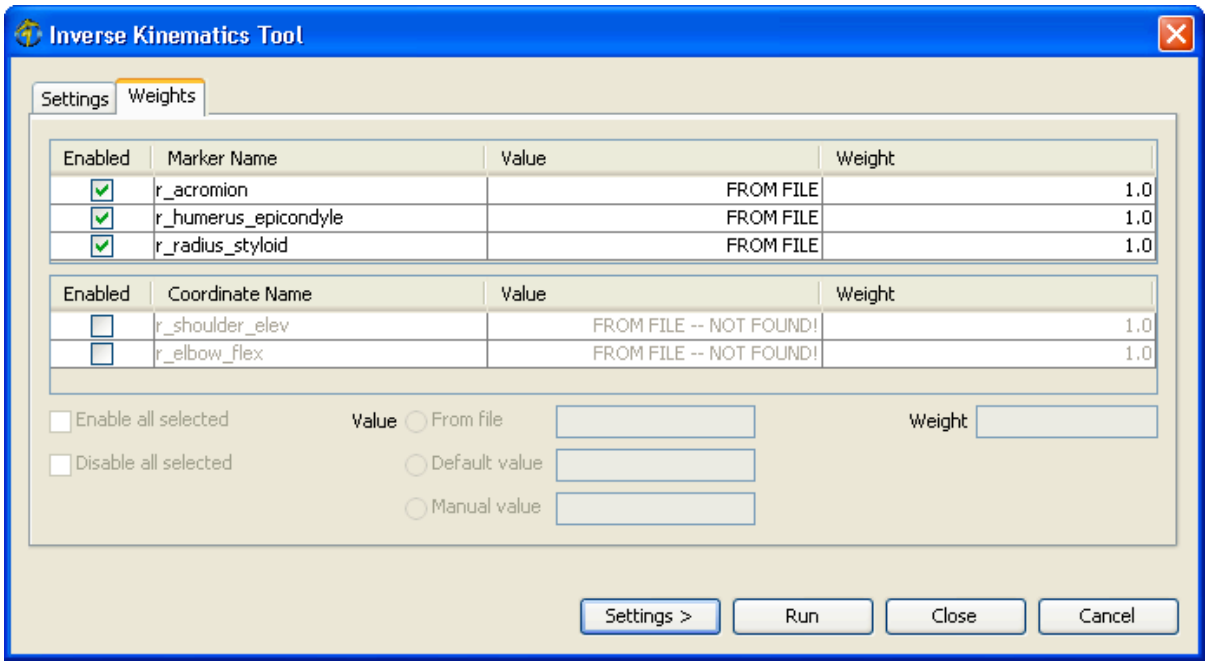


Figure 2-3: Specifying Inverse Kinematics Tool Weights

3 Inverse Dynamics

3.1 How it Works

The equations of motion for a multibody system may be written in the following form:

$$\underbrace{\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q},\dot{\boldsymbol{q}}) + \boldsymbol{G}(\boldsymbol{q})}_{\text{knowns}} = \underbrace{\boldsymbol{\tau}}_{\text{unknowns}}$$

where N is the number of degrees of freedom; $\boldsymbol{q},\dot{\boldsymbol{q}},\ddot{\boldsymbol{q}} \in \boldsymbol{R}^N$ are the vectors of generalized positions, velocities, and accelerations, respectively; $\boldsymbol{M}(\boldsymbol{q}) \in \boldsymbol{R}^{N\times N}$ is the system mass matrix; $\boldsymbol{C}(\boldsymbol{q},\dot{\boldsymbol{q}}) \in \boldsymbol{R}^N$ is the vector of Coriolis and centrifugal forces; $\boldsymbol{G}(\boldsymbol{q}) \in \boldsymbol{R}^N$ is the vector of gravitational forces; and $\boldsymbol{\tau} \in \boldsymbol{R}^N$ is the vector of generalized forces.

The motion of the model is completely defined by the generalized positions, velocities, and accelerations. Consequently, all of the terms on the left-hand side of the equations of motion are known. The remaining term on the right-hand side of the equations of motion is unknown. The Inverse Dynamics Tool uses the known motion of the model to solve the equations of motion for the unknown generalized forces (e.g., joint torques).

3.2 Inputs

Two files are required as input by the Inverse Dynamics Tool:

arm26_InverseKinematics.mot: Motion file containing the time histories of generalized coordinates that describe the movement of the model. This file may be generated by the Inverse Kinematics Tool.

arm26.osim: The current model loaded in OpenSim.

***Note**: When analyzing movements with external forces (e.g., ground reaction forces during walking), this data can be specified on the External Forces tab.*

3.3 Outputs

The Inverse Dynamics Tool generates a single file in a specified folder:

arm26_InverseDynamics_force.sto: Storage file containing the time histories of the net forces and torques at each joint.

3.4 Inverse Dynamics Tool

To launch the Inverse Dynamics Tool select, **Inverse Dynamics...** from the **Tools** menu. The **Inverse Dynamics Tool** dialog (Figure 3-1) like all other OpenSim tools operates on the *Current Model* open and selected in OpenSim (e.g., *arm26*). The Inverse Dynamics Tool is controlled by a dialog with two tabbed panes. The **Main Settings** pane specifies parameters relating to the input kinematics of the current model, the time range for the analysis, and the output of the results. The **External Loads** pane specifies parameters relating to the external loads applied to the model during the analysis.

The Main Settings pane (Figure 3-1) is organized into four main sections entitled **Current Model**, **Input**, **Time**, and **Output**. The **Current Model** section displays an uneditable name for the current model being used for the inverse dynamics analysis. The **Input** section displays editable information specifying the kinematics (e.g., states or motion) describing the movement of a model. The **Time** section displays editable information specifying the start and end time for the inverse dynamics analysis. The **Output** section displays editable information specifying the prefix appended to the resulting output file, the directory to which the file is saved, and the precision (number of decimal places) used when writing results. You may use the  button to browse for a directory to save the output files, and the  button to explore to the specified directory.

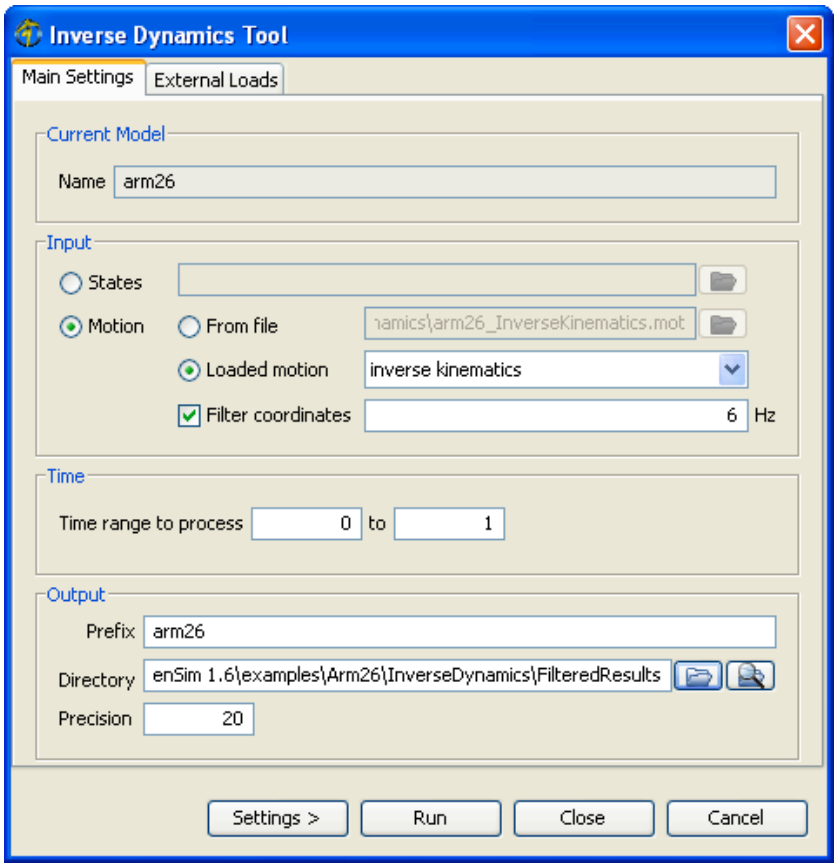


Figure 3-1: Dialog for the Inverse Dynamics Tool. The main settings pane.

4 Example: Inverse Kinematics and Inverse Dynamics

4.1 Performing Inverse Kinematics

1. Open the Model. To open the model, select **Open Model…** from the **File** menu, then navigate the directory with the Arm26 example (e.g., *C:\OpenSim 2.2\examples\Arm26\arm26.osim*).
2. **Open Inverse Kinematics Tool**. To open the tool (Figure 2-2), select **Inverse Kinematics…** from the **Tools** menu.
3. **Specify Trial Name**. Begin in the **Settings** pane and enter a name (e.g., *inverse kinematics*) for the trial.
4. **Specify Marker Data for Trial**. Browse to the inverse kinematics directory (e.g., *..\InverseKinematics*) and select the marker file (e.g., *arm26_elbow_flex.trc*).
5. **Specify Time Range**. Enter a range from 0 to 1 seconds corresponding to the interval in the marker file.
6. **Confirm Weights**. Move to the **Weights** pane and set all marker weights to 1.
7. **Save Settings**. Use the **Settings >** button to save your settings to a setup file (e.g., *arm26_Setup_InverseKinematics.xml*).
8. **Run and Close Inverse Kinematics Tool**. The model's elbow will begin to flex as the optimization to best match marker trajectories. When the analysis has completed by reaching the end of the specified time range, the **Motions** branch under the model in the **Navigator** will be populated by the inverse kinematics motion.

4.2 Viewing Inverse Kinematics Results

1. **View Motion**. Use the motion viewer to play back the inverse kinematics motion.
2. **Save Results from Inverse Kinematics**. Right click the new motion in under the model in the **Navigator** and select **Save As** to save the file (e.g., *arm26_InverseKinematics.mot*).
3. **Plot Joint Angles from Inverse Kinematics**. From the resulting motion (e.g., *inverse kinematics*), plot *r_shoulder_elev* and *r_elbow_flex* versus *time*. (see Chapter 13 of the OpenSim User's Guide for complete details on using the plotter).

4.3 Performing Inverse Dynamics

1. **Open Inverse Dynamics Tool**. To open the tool (Figure 3-1), select **Inverse Dynamics…** from the **Tools** menu.
2. **Specify Unfiltered Input Motion**. Use the radio buttons to select the **Loaded motion** (e.g., *inverse kinematics*) and uncheck the **Filter coordinates** option.
3. **Specify Time Range**. Enter a range from 0 to 1 seconds corresponding to the interval in the motion.
4. **Specify Output Directory**. Set the output **Directory** (e.g., *..\InverseDynamics\UnfilteredResults*), so that you are able to compare the results of inverse dynamics analyses using unfiltered and filtered input motions.
5. **Save Settings**. Use the **Settings >** button to save your settings to a setup file (e.g., *arm26_Setup_InverseDynamics.xml*).
6. **Run Inverse Dynamics Tool**. You will see the model begin to move as the analysis flexes the elbow while computing joint torques. When the analysis has completed by reaching the end of the specified time range, the specified output directory will be populated by a storage file (e.g., *arm26_InverseDynamics_force.sto*).
7. **Specify New Filtered Input Motion**. Check the **Filter coordinates** option and enter a cutoff frequency of 6 Hz.
8. **Specify New Output Directory**. Rename the output **Directory** (e.g., *..\InverseDynamics\FilteredResults*).
9. **Run and Close Inverse Dynamics Tool**.

4.4 Comparing Inverse Dynamics Results

1. **Plot Noisy Joint Torques from Inverse Dynamics**. From the resulting file (e.g., *..\UnfilteredResults\arm26_InverseDynamics_force.sto*), plot *r_shoulder_elev* and *r_elbow_flex* versus *time*. Leave the **Plotter** dialog open to compare subsequent joint torques.
2. **Plot Smooth Joint Torques from Inverse Dynamics**. From the resulting file (e.g., *..\FilteredResults\arm26_InverseDynamics_force.sto*), plot *r_shoulder_elev* and *r_elbow_flex* versus *time*. Compare with noisy joint torques.

5 Computed Muscle Control

5.1 Why is Computed Muscle Control Necessary

With musculoskeletal models, we are also typically interested in estimating muscle forces and controls. Traditional approaches to solve for muscle controls computed, like static optimization (see Chapter 17 of the OpenSim User's Guide), often fail to reproduce the observed motion (the inputs to inverse dynamics and static optimization) when applied in a forward dynamics simulation. There are three principle causes for this discrepancy: 1) forward and inverse musculoskeletal models do not share identical dynamics, 2) experimental noise and sampling results in dynamically inconsistent kinematics and 3) musculoskeletal models are nonlinear dynamical systems and inherently chaotic. Cause 3) is often overlooked but it is important to realize that even if identical models where used in an inverse and then forward analysis with noiseless and error-free kinematics (i.e. synthetic data) a forward simulation will fail to reproduce the initial performance if the initial states of the simulation are not identical, since even the smallest of differences (to machine precision) can lead to diverging solutions. Cause 2) stems from the reality that data acquired (from a subject) does not match what could be generated by the model (satisfying modeled dynamics) and the estimates of joint kinematics (from IK) does not take into the continuity of system dynamics from one instant to the next given discrete samples of position data. The largest source of discrepancies is the fact that different models are used to perform inverse dynamics and static optimization versus that of a forward simulation. Even when static optimization includes force-length and force-velocity relationships, the estimate of muscle length and velocity are determined by the length of the whole muscle-tendon unit (inelastic tendon) and activations do not satisfy excitation-to-activation dynamics present in forward.

5.2 How it Works

Computed muscle control (CMC) attempts to bridge the gap between forward and inverse methods by combining: PD feedback control to track experimental kinematic, static optimization to estimate the feed forward controls (muscle excitations) in order to generate desired accelerations at a small time (T) in the future, and then forward integration to generate new states and step forward in time.

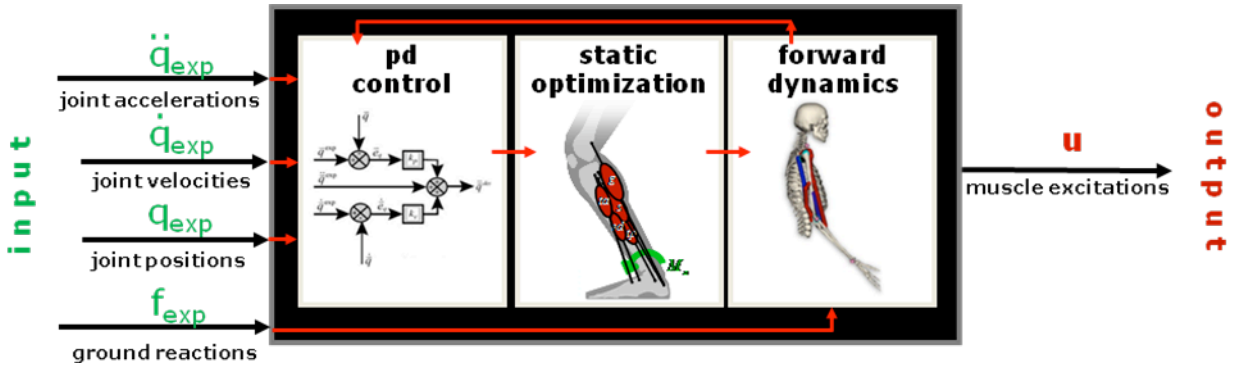


Figure 5-1: Overview of Computed Muscle Control

PD: $$\ddot{\bar{q}}^*(t+T) = \ddot{\bar{q}}_{exp}(t+T) + \bar{k}_v \cdot [\dot{\bar{q}}_{exp}(t) - \dot{\bar{q}}(t)] + \bar{k}_p \cdot [\bar{q}_{exp}(t) - \bar{q}(t)]$$

Two formulations of the static optimization problem are currently available in CMC. The first formulation, called the slow target, consists of a performance criterion () that is a weighted sum of squared actuator controls plus the sum of desired acceleration errors:

Slow Target: $$J = \sum_{i=1}^{nx} x_i^2 + \sum_{j=1}^{nq} w_j (\ddot{q}_j^* - \ddot{q}_j)^2$$

The second formulation, called the fast target, is the sum of squared controls augmented by a set of equality constraints (C_j=0) that requires the desired accelerations to be achieved within the tolerance set for the optimizer:

Fast Target: $$J = \sum_{i=1}^{nx} x_i^2 \; ; \qquad C_j = \ddot{q}_j^* - \ddot{q}_j, \;\; for \;\; all \;\; j$$

5.3 Inputs

The primary inputs to CMC consist of:

arm26_InverseKinematics.mot: Desired kinematics [.mot or .sto file] to be tracked

arm26_ComputedMuscleControl_Tasks.xml : Tracking tasks [.xml file] specifying which coordinates are to be tracked

(**arm26_CMC_Control_Constraints.xml**): Optional control constraints [.xml file] used to limit the allowed values of the actuator controls.

arm26.osim: The current model loaded in OpenSim.

(**arm26_Reserve_Actuators.xml**): Optional set of actuators (reserve are ideal torques) to append or replace the model's current set of actuators. Falls under the "Actuators and External Loads" tab of the **CMCTool**. In this case, ideal torques supplement muscles if muscles are unable to generate the required net joint moments.

5.4 Outputs

The following primary CMC outputs are placed in the specified output directory:

arm26_controls.xml: Actuator control [.xml file] (e.g., muscle excitations) computed by CMC that will drive a forward dynamic simulation.

arm26_controls.sto: Actuator controls [.sto file] computed by CMC in a format suitable for plotting.

arm26_states.sto: Model states file [.sto file] containing the time histories of all model states that occurred during the CMC simulation.

arm26_Kinematics_q.mot: Motion file [.mot file] containing the time histories of the generalized coordinates resulting from CMC (Joint angles are expressed in degrees rather than radians).

5.5 Computed Muscle Control (CMC) Tool

To launch the Computed Muscle Control Tool, select **Computed Muscle Control...** from the **Tools** menu. The Computed Muscle Control Tool is controlled by a dialog with three tabbed panes (Fig. 8-2). The **Main Settings** pane specifies parameters relating to the controls and states that will be input into the model, the time range for the simulation, and the output of the results. The **Actuators and External Loads** pane specifies the actuator set and the external loads applied to the model during the simulation. The **Integrator Settings** pane specifies integrator step sizes and tolerances used to solve the simulation. Limits on the range of controls can be defined by selecting the option **Actuator constraints** check box.

The Main Settings pane (Fig. 8-2) is organized into five main sections entitled **Current Model**, **Input**, **Reduce Residuals**, **Time**, and **Output**. The **Current Model** section displays uneditable information about the current model being used for analysis by the Computed Muscle Control Tool. The **Input** section displays editable information specifying the desired kinematics to be tracked by the CMC Tool. You may use the  button to browse

for the desired kinematics as either a storage (.sto) or motion (.mot) file. Filtering options are the next set of inputs, followed by the Tasks (.xml) file that specifies the kinematics to be tracked, their relative weightings and PD controller gains. The **Time** section displays editable information specifying the start and end time for the forward simulation during CMC as well at the look-ahead time window CMC uses to estimate accelerations in the future from current controls. The **Output** section displays editable information specifying the prefix appended to all of the resulting output files, the directory to which the files are saved, and the precision (number of decimal places) used when writing results. You may use the button to browse for a directory to save the output files, and the button to open an explorer to the specified directory.

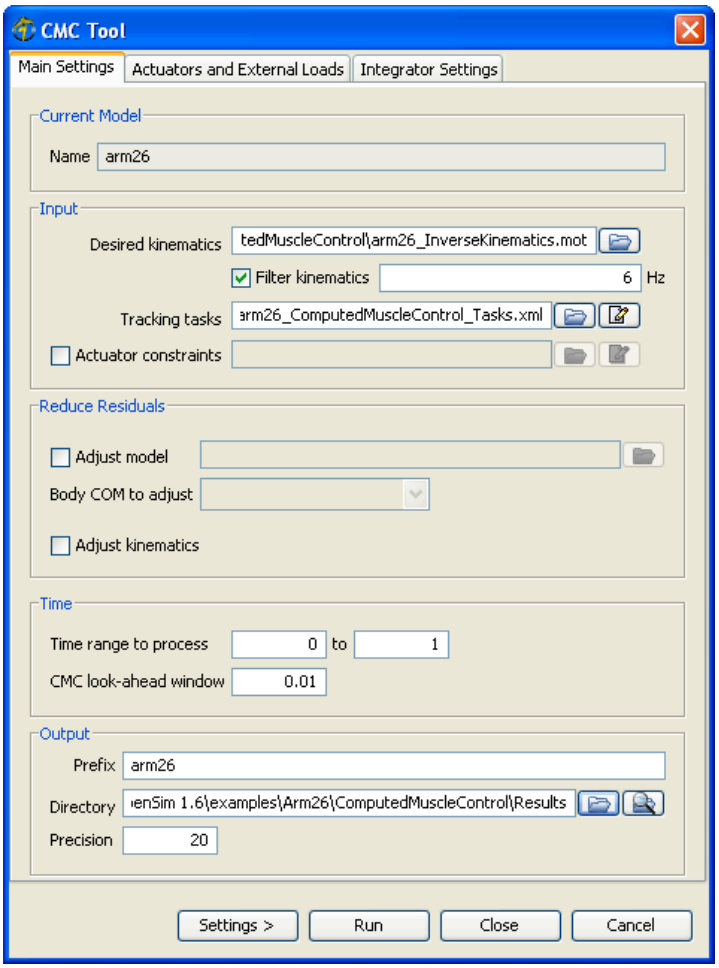


Figure 5-2: Dialog for the Compute Muscle Control Tool. The main settings pane.

6 Example: Computed Muscle Control

6.1 Using Computed Muscle Control

1. **Open Computed Muscle Control Tool**. To open the tool (**Error! Reference source not found.**), select **Compute Muscle Control…** from the **Tools** menu.
2. **Specify Filtered Input Motion**. Browse to the inverse kinematics directory and select the **Desired kinematics** file (e.g., *..\InverseKinematics\arm26_InverseKinematics.mot*), check the **Filter kinematics** option, and enter a cutoff frequency of 6 Hz.
3. **Specify Tracking Tasks**. Browse to select the tasks file (e.g., *arm26_ComputedMuscleControl_Tasks*.xml) specifying the joint coordinates for CMC to track, their relative weightings, and the Kp and Kv gains on the errors.
4. **Uncheck Adjust Model**. This option is used to specify an adjusted model when performing residual reduction to obtain more dynamically consistent simulations. For our example, residual reduction is not necessary.
5. **Specify Time Range**. The **time range** for the forward simulation is specified and these should be set from 0 to 1 sec to correspond to the interval upon which the controls from static optimization were computed.
6. **Set CMC look-ahead window.** A time window of 0.01 is generally sufficient for muscle activations to change enough to produce the desired accelerations.
7. **Specify Output Directory**. Set the output **Directory** (e.g., *..\ComputedMuscleControl\Results*), so that you are able to compare the results of a CMC simulation with Forward Dynamics simulations using Nonphysiological and Physiological controls generated earlier by Static Optimization.
8. **Specify Additional Model Actuators**. Set the actuator settings to **Append** (rather than replace) edit the **Additional actuator set files** field by adding an actuator set file (e.g., *arm26_Reserve_Actuators.xml*) containing reserve torques.
9. **Save Settings**. Use the **Settings >** button to save your settings to a setup file (e.g., *arm26_Setup_ComputedMuscleControl.xml*).

10. **Run CMC Tool.** You will see the model begin to move as muscles contract and accelerate the model. When the simulation has completed by reaching the end of the specified time range, the specified output directory will be populated by states files (e.g., *arm26_states_degrees.mot*) and the corresponding motion file (e.g., *arm26_states*) will be associated with the model in the GUI.

6.2 Modifying the Muscle Controls

1. **Open Excitation Editor** by selecting **Excitation…** from the **Edit** menu.
2. **Load Controls**. In the Excitation Editor dialog, load the controls file (e.g., *arm26_controls.xml*) from the static optimization results that use the muscle force-length-velocity relation. These results are located in your specified output directory (e.g., *..\ComputedMuscleControl\Results*). Select muscle excitations to edit (e.g., *BIClong*).
3. **Modify Controls**. The controls appear as individual graphs with moveable control nodes, which enable you to reshape the controls as desired. To select an individual control node, hold the **Ctrl** key and click the control node. To select multiple control nodes, hold the **Ctrl** key and drag a box (from top left to bottom right) around control nodes of interest. Use the left mouse button to drag selected control node(s). Increase the excitation to Biceps Long (*BIClong*) by 25% or so.
   Note: To zoom in, click and drag from the top left to bottom right. To zoom out, click and drag from bottom right to top left.
4. **Save Modified Controls**. Use the **Save As** button to save the modified controls to a new file (e.g., *\ComputedMuscleControl\Results\arm26_Modified_controls.xml*).
5. **Open Forward Dynamics Tool**. To open the tool, select **Forward Dynamics…** from the **Tools** menu.
6. **Specify the Controls**. In the Forward Dynamics Tool dialog, select the **Controls** file (e.g., *arm26_Modified_controls.xml*) from the static optimization results that do not use the muscle force-length-velocity relation.
7. **Specify Initial States**. An initial **States** file was calculated in the CMC results (e.g.,*\ComputedMuscleControl\Results\arm26_states.sto*) and the Forward Dynamics Tool should use this file.
8. **Solve for Equilibrium**. This option makes sure that the initial actuator states (muscle activation, fiber length) are in static equilibrium, that is the rate of contraction is zero. This is a useful option for setting initial states when one does not

have reliable estimates and the model is starting from rest. It utilizes the input state to determine the position of the model and initial activation of the muscles from which the initial muscle fiber lengths are computed.

9. **Specify Time Range**. The **time range** for the forward simulation is specified and these should be set from 0 to 1 sec to correspond to the interval upon which the controls from static optimization were computed.
10. **Specify Output Directory**. Set the output **Directory** (e.g., *..\ForwardDynamics_ModifiedCMC_Results*), to save the results of the simulation.
11. **Specify Model's Actuators**. Set the actuator settings to **Append** (rather than replace).
12. **Save Settings**. Use the **Settings >** button to save your settings to a setup file (e.g., *arm26_Setup_ForwardDynamics.xml*).
13. **Run Forward Dynamics Tool.** You will see the model begin to move as muscles contract and accelerate the model. When the simulation has completed by reaching the end of the specified time range, the specified output directory will be populated by states files (e.g., *arm26_states_degrees.mot*).
14. **View New Motion**. Use the motion viewer controls to play back the forward dynamics motion. Was performance improved?

6.3 Experiment On Your Own

1. **Experiment with Different Controls**. Repeat section 6.2 with other muscle controls.
2. **View Results Simultaneously**. Multiple models can be open at once to visualize simulation results simultaneously. Open another model (e.g., *arm26.osim*) by selecting **Open model...** from the **File** menu. The new model will appear offset from the original model. You can associate other motions (e.g., *arm26_states_degrees.mot*) to this model by selecting **Load motion...** from the **File** menu.
3. **Make a Movie**. Use the camera tool to take snapshots or use the movie-camera to generate animations. The camera dolly allows the view point of the movie-camera to change during when the animation is being captured, by interpolating between user defined views.

7 Introduction to the API

7.1 API Overview

This chapter introduces the Application Programming Interface (API) for OpenSim, a freely available software package for musculoskeletal modeling and dynamic simulation of movement. For more information on OpenSim, visit the OpenSim project site at http://simtk.org/home/opensim. The project site provides a forum for users to ask questions and share expertise, as well as many other resources. The API is intended for users who want to utilize OpenSim functionality inside their own framework and whose needs are not satisfied by the GUI.

7.2 Prerequisites for Programming with OpenSim

To run the examples provided in this guide, you will need:

- A computer running Windows XP, Vista or Windows7 (or Mac OSX with Windows, BootCamp, or VMWare)
- Microsoft's Visual Studio (version 2005 or 2008) or Visual C++ 2008 Express Edition (http://www.microsoft.com/express/vc/Default.aspx)
- CMake 2.6.0 or later (http://www.cmake.org/cmake/resources/software.html)

We also recommend the following tools:

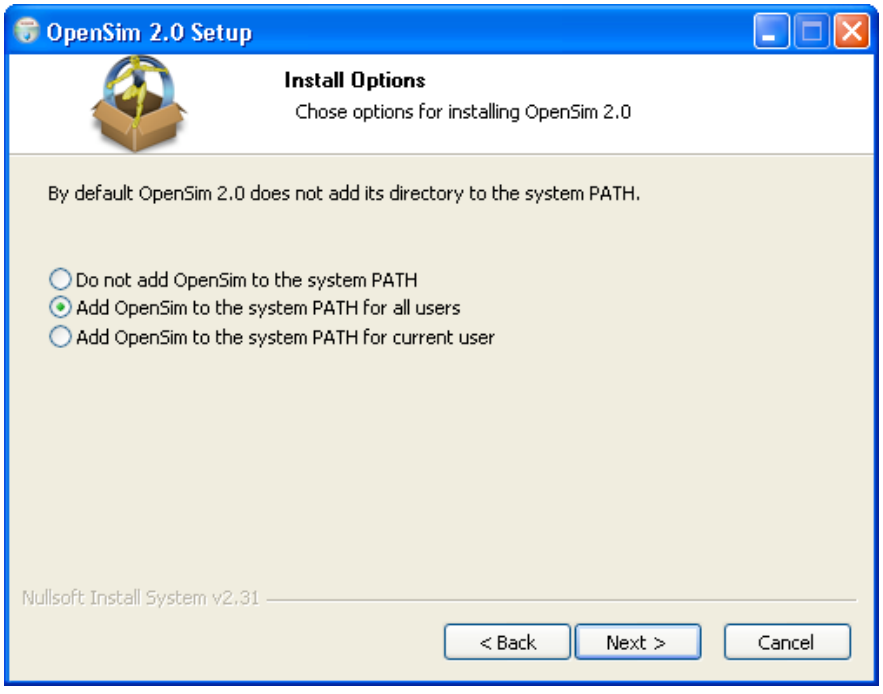
- An XML Editor, for example,
  - notepad++ (http://notepad-plus.sourceforge.net/uk/site.htm)
  - XMLMarker (http://symbolclick.com/download.htm)
- Dependency Walker (http://www.dependencywalker.com/): a third-party, free tool for checking the interdependency between modules (dll, sys, exe, etc.) on Windows. It is useful for troubleshooting installation issues.

7.3 Installing OpenSim

1. To install OpenSim API, download the self-extracting executable from the download page of OpenSim (go to http://simtk.org/home/opensim and click on "Downloads". The API is installed by default along with the OpenSim application).

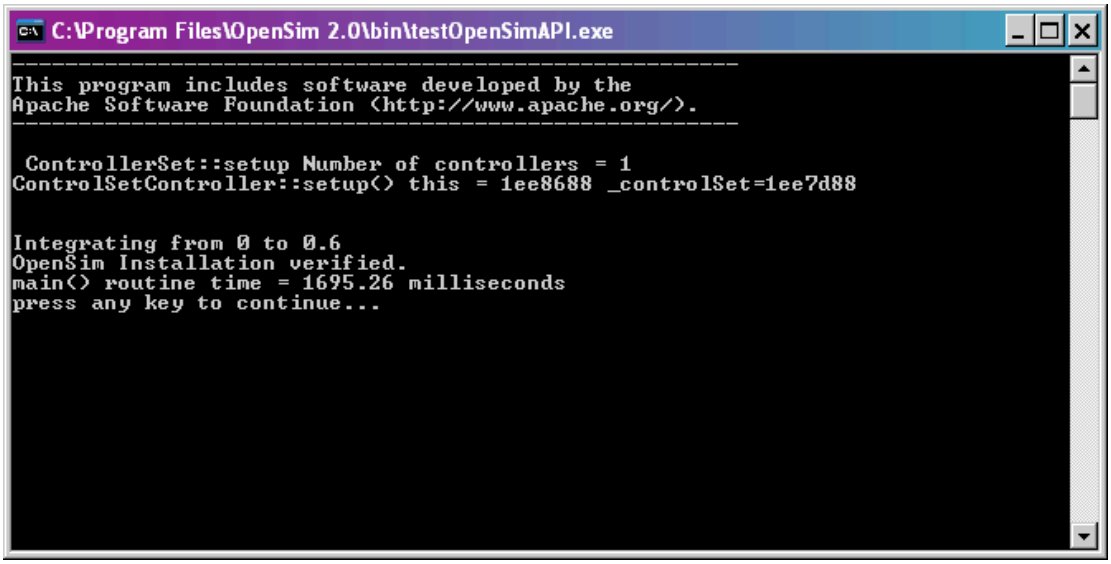
   Due to incompatibility between various versions of Microsoft Visual Studio, you need to download/install the distribution of OpenSim that is consistent with your development environment: either Visual Studio 2005, 2008, or 2008 Express
   Visual Studio 2008 Express is recommended, as it is free and tested!

2. Run the executable, following the on-line instructions.
   To be able to run the main programs from the command line (outside Visual Studio), you need to add the OpenSim libraries to your PATH. This can be done during installation by selecting the radio button as illustrated below.



Warning: *Earlier installations of OpenSim will continue to be accessible but only through the GUI, which sets its own environment (PATH) variable.*

3. Test your installation. Go to the *\bin* directory for the OpenSim installation (if you installed in the default location, the full directory is *C:\OpenSim2.2\bin*). Copy the file *testOpenSimAPI.exe* to the directory *C:\OpenSim 2.2\sdk\APIExamples\ExampleMain* and then double-click on it to run the test. If everything was installed correctly, a window should pop up with message like that shown below:



7.4 Obtaining the Example Programs

The examples come with the OpenSim distribution and are located in the *sdk\APIExamples* directory for the OpenSim installation (if you installed in the default location, the full directory is *C:\OpenSim 2.2\sdk\APIExamples*).

8 Using the OpenSim API

8.1 An Example main Program ("Tug of War")

In this chapter, we will write a main program to perform a forward dynamic simulation using the OpenSim API. We will build it up in pieces, starting from the simplest possible OpenSim model, a single block experiencing the force of gravity. In the end, we will have an OpenSim model with two muscles performing a tug-of-war on the block, with the muscles and ground reactions counteracting the gravitational force. The resulting source code and associated files for this example come with the OpenSim 2.2 distribution under the directory:

```
C:\OpenSim 2.2\sdk\APIExamples\ExampleMain
```

Performing a forward dynamic simulation in OpenSim involves a series of steps. Each of these steps usually requires only a few lines of code. The following sections explain the steps by gradually developing a complete program, stopping at points where the partial program can be compiled, run, and its results visualized.

For your convenience, we have provided the source code as a series of exercise-labeled development snapshots, gradually leading up to the complete program, which is called *TugOfWar_Complete.cpp*. After most steps, we will be using the OpenSim GUI to visualize the model and the motions that result from running the simulations. Each of the steps below generates output files with the same names.

8.2 Setting up Visual Studio with CMake

To get started we will use CMake to generate a Visual Studio project:

1. Open CMake to generate a Visual Studio solution file. In textbox labeled ***Where is the source code:***, specify the example directory, e.g. *C:\OpenSim 2.2\sdk\APIExamples\ExampleMain*. In the textbox labeled ***Where to build the binaries***, specify a new build directory, such as *C:\OpenSim 2.2\sdk\APIExamples\ExampleMain_BUILD*.
2. Press Configure. CMake will ask you to create the directory (say "Yes") and will ask you to specify your compiler (e.g., *Visual Studio 9 2008*).
3. If you installed OpenSim in a location other than the default then you need to change the value of the CMake variable OPENSIM_INSTALL_DIR to point to the actual installation directory. This can be done either in the CMake interface (after you "Configure" and before you "Generate") or alternatively it could be done directly in the CMakeLists.txt file.
4. Press **Configure** again. Then press **Generate**.
5. If you're using Visual Studio, navigate to the new build directory and find the file named *OpenSimTugOfWar.sln*. Double click the file to launch Visual Studio and load in the solution file that CMake just created.
6. Within Visual Studio, set the Configuration to "RelWithDebInfo" (that is "release with debug information" – unfortunately it won't work in Debug).
7. Compile and run the program. If it is working, it should output "`OpenSim example completed successfully.`" and do nothing else. If it doesn't work, be sure to resolve the problem before attempting to move any further through the exercise.

8.3 Create an OpenSim Model

To perform a simulation, we first create an OpenSim model and set its name in our main program.

```
#include <OpenSim/OpenSim.h>
using namespace OpenSim;

int main()
{
   try {
      // Create an OpenSim model and set its name
      Model osimModel;
      osimModel.setName("tugOfWar");
   }

   catch (OpenSim::Exception ex) {
      std::cout << ex.getMessage() << std::endl;
      return 1;
   }

   std::cout << "OpenSim example completed successfully.\n";
   return 0;
}
```

Exercise 1: This version of the example is available as *TugOfWar1_CreateModel.cpp*. While the main program above compiles and runs, the OpenSim model it creates is "empty" and no information is saved to a file. Note that you only need to include the header file <OpenSim/OpenSim.h> at the top of the file. Also, note the line:

```
using namespace OpenSim;
```

This line is required to avoid having to prefix every symbol with OpenSim:: since all OpenSim classes live in the namespace OpenSim. Another namespace that will appear later in this guide is SimTK, which is utilized by OpenSim for many fundamental classes.

8.4 Get the Model's Ground Body

A new OpenSim model comes with a ground body. This ground body, however, has no geometry attached to it. After we have created an OpenSim model, we get a reference to the model's ground body. We can then add display geometry to it so we can visualize it in the OpenSim GUI:

```
// Get a reference to the model's ground body
OpenSim::Body& ground = osimModel.getGroundBody();

// Add display geometry to the ground to visualize in the GUI
ground.addDisplayGeometry("ground.vtp");
ground.addDisplayGeometry("anchor1.vtp");
ground.addDisplayGeometry("anchor2.vtp");
```

OpenSim allows for files of type *.vtp*, *.stl* and *.obj* as display geometry. At this point we still haven't saved any information to a file, so the model cannot be opened or visualized within the OpenSim GUI. We'll do that next.

8.5 Save the Model to a File

After we have created a ground body and added its display geometry, we save the model to a file with the ".osim" extension in order to visualize the model we have created.

```
// Save the model to a file
osimModel.print("tugOfWar_model.osim");
```

Exercise 2: After we compile and run the main program, we can open the model file *tugOfWar_model.osim* in the OpenSim GUI and visualize the ground body (*highlighted with green for this guide*).

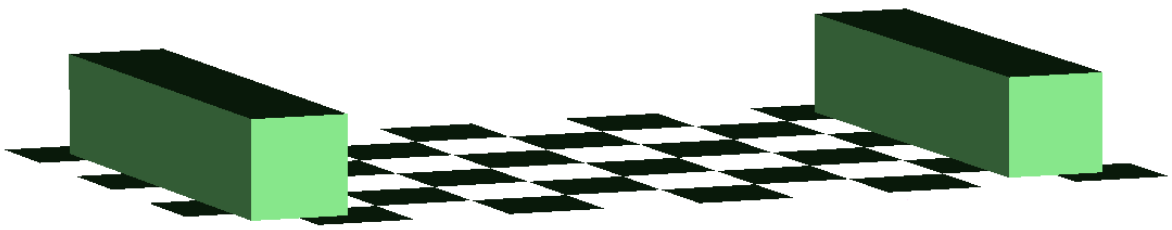


Figure 8.1: Model with only visible ground geometry.

Except for the colors, you should see an image in the GUI like the one above.

8.6 Create a New Block Body

To add an additional body to the OpenSim model, we create a new block body with inertial properties and add display geometry to it.

```
using namespace SimTK;

// Specify properties of a 20 kg, 0.1 m^3 block body
double blockMass = 20.0, blockSideLength = 0.1;
Vec3 blockMassCenter(0);
Inertia blockInertia = blockMass*Inertia::brick(blockSideLength,
   blockSideLength, blockSideLength);

// Create a new block body with specified properties
OpenSim::Body *block = new OpenSim::Body("block", blockMass,
   blockMassCenter, blockInertia);

// Add display geometry to the block to visualize in the GUI
block->addDisplayGeometry("block.vtp");
```

The classes Vec3 and Inertia live in the namespace SimTK. You can write them as SimTK::Vec3 and SimTK::Inertia or include a "using namespace" statement as we did above.

Also, note that the units for mass and length are kilograms and meters, respectively. OpenSim uses the SI convention (length in meters; mass in kilograms; time in seconds; forces in Newtons; and moments/torques are in Newton-meters). Angles can be in degrees or radians; internally, OpenSim uses radians.

Programming Note: *OpenSim model objects in this example are allocated on the heap using "new". Whenever they are added to the model, the model takes ownership of these objects, you shouldn't call "delete" on these objects otherwise the model will be left holding to stale pointers, these objects will be destructed by the model destructor.*

At this point, the block body is not connected to the OpenSim model and cannot be used or visualized in the GUI. To achieve that, the block body has to be connected to the ground body (or any other body already in the model) with a joint. We'll do that next.

8.7 Create a Joint

Before we add the block body to the OpenSim model, we create a new free joint (i.e., 6 degrees-of-freedom) between the block and ground.

```
// Create a new free joint with 6 degrees-of-freedom (coordinates)
   between the block and ground bodies
Vec3 locationInParent(0, blockSideLength/2, 0),
   orientationInParent(0), locationInBody(0), orientationInBody(0);
FreeJoint *blockToGround = new FreeJoint("blockToGround", ground,
   locationInParent, orientationInParent, *block, locationInBody,
   orientationInBody);

// Get a reference to the coordinate set (6 degrees-of-freedom)
   between the block and ground bodies
CoordinateSet& jointCoordinateSet =
                 blockToGround->getCoordinateSet();

// Set the angle and position ranges for the coordinate set (SimTK::
   prefix not actually needed here)
double angleRange[2] = {-SimTK::Pi/2, SimTK::Pi/2};
double positionRange[2] = {-1, 1};
jointCoordinateSet[0].setRange(angleRange);
jointCoordinateSet[1].setRange(angleRange);
jointCoordinateSet[2].setRange(angleRange);
jointCoordinateSet[3].setRange(positionRange);
jointCoordinateSet[4].setRange(positionRange);
jointCoordinateSet[5].setRange(positionRange);
```

At this point, the block body and corresponding free joint are ready to be added to the OpenSim model. Although we defined a FreeJoint in this example, different kinds of joints are available, with corresponding constructors:

- WeldJoint
- PinJoint
- SliderJoint
- BallJoint
- EllipsoidJoint
- CustomJoint
- Joint (abstract class)

8.8 Add the Block Body to the Model

To finish this step, we simply add the block body to the OpenSim model.

```
// Add the block body to the model
osimModel.addBody(block);
```

Exercise 3: After we compile and run the current main program, we can open the model in the OpenSim GUI (the model will be overwritten with the same file name *tugOfWar_model.osim* as in the earlier step) and visualize the ground body (*highlighted with green for this guide*) and the block body (*highlighted with blue for this guide*). You'll also be able to open the "coordinate viewer" within the GUI and interactively change the coordinates. For the FreeJoint, the built-in names of the coordinates are "X-rotation", "Y-rotation", "Z-rotation" followed by the three translations "X-translation", "Y-translation", and "Z-translation". These names, however, can be changed by calling the coordinate's `setName()` method directly.

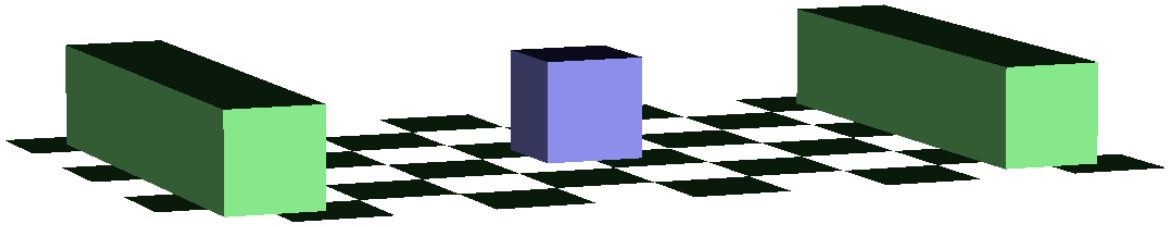


Figure 8.2: Model of a moving free block between two fixed anchors.

Except for the colors, the model in the GUI should look like the image above. Be sure to go to the "coordinates" pane, move the sliders corresponding to the six coordinates, and note the effect that has on the block's position and orientation.

8.9 Define Gravity

In order for the block to actually fall during the simulation, we define the acceleration of gravity to pull the block towards the ground. The actual direction of the vector is arbitrary; however, OpenSim uses the convention that gravity is in the negative Y-direction in the models included with the OpenSim distribution.

```
// Define the acceleration of gravity
osimModel.setGravity(Vec3(0,-9.80665,0));
```

8.10 Initialize the OpenSim Model System

An OpenSim model is backed by a SimTK::System, which actually performs the computations. As such, the model itself is a stateless object with the state being stored

externally in an instance of SimTK::State. To begin simulating the block falling, we initialize the SimTK::System associated with the OpenSim model and create an instance of the system state. **After the call to initSystem(), no changes should be made to the structure of the model.** For example, adding forces or constraints would require the re-creation of the system and a fresh call to initSystem() since these objects may have a state of their own that needs to be incorporated into the system's state.

```
// Initialize the system
SimTK::State& si = osimModel.initSystem();
```

8.11 Define Initial Position and Velocity States of the Block

Next, we define the initial position and velocity of the block. For the free joint, the position coordinates and their velocities are ordered 0 (x-rotation), 1 (y-rotation), 2 (z-rotation), 3 (x-translation), 4 (y-translation), and 5 (z-translation).

```
// Define non-zero (defaults are 0) states for the free joint
CoordinateSet& modelCoordinateSet = osimModel.updCoordinateSet();
// set x-translation value
modelCoordinateSet[3].setValue(si, blockSideLength);
// set x-speed value
modelCoordinateSet[3].setSpeedValue(si, 0.1);
// set y-translation value
modelCoordinateSet[4].setValue(si, blockSideLength/2+0.01);
```

8.12 Create the Integrator and Manager for the Simulation

We create the integrator and manager for the simulation in order to perform the numerical integration of the system equations of motion during the forward dynamics simulation. An OpenSim Manager object collects together all the resources need to perform a simulation, including the Model, the numerical methods to be employed, the current State, storage for the trajectory, and runtime options for controlling the simulation.

```
// Create the integrator and manager for the simulation.
SimTK::RungeKuttaMersonIntegrator
   integrator(osimModel.getMultibodySystem());
integrator.setAccuracy(1.0e-4);
Manager manager(osimModel, integrator);
```

8.13 Integrate the System Equations of Motion

We integrate the system equations of motion from the initial time to the final time of the simulation. Depending on your computer speed, this numerical integration could take from a few to several seconds.

```
// Define the initial and final simulation times
double initialTime = 0.0;
double finalTime = 4.0;


// Integrate from initial time to final time
manager.setInitialTime(initialTime);
manager.setFinalTime(finalTime);
std::cout<<"\n\nIntegrating from "<<initialTime<<" to "
   <<finalTime<<std::endl;
manager.integrate(si);
```

8.14 Save the Simulation Results

After we have performed the integration for the forward dynamics simulation, we save the resulting motion in order to visualize the simulation we have created. Note that OpenSim uses radians internally but degrees are required in a .mot file, so we have to convert to degrees before writing out the .mot file for visualization.

```
// Save the simulation results
Storage statesDegrees(manager.getStateStorage());
statesDegrees.print("tugOfWar_states.sto");
osimModel.updSimbodyEngine().convertRadiansToDegrees(statesDegrees);
statesDegrees.setWriteSIMMHeader(true);
statesDegrees.print("tugOfWar_states_degrees.mot");
```

Exercise 4: After we compile and run the current main program, we can load the model and the motion in the OpenSim GUI and visualize the simulation. (To load the motion, go to File → Load Motion. Select the motion file *tugOfWar_states_degrees.mot* that we just wrote out.)

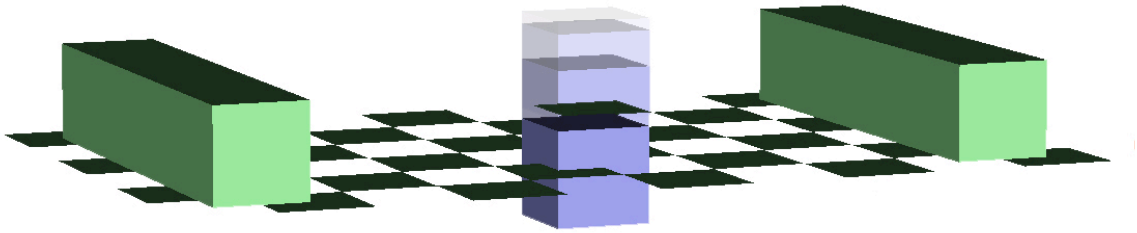


Figure 8.3: Block is falling in the presence of gravity.

Except for the colors, you should see the above in the GUI. Using the motion slider and video controls, visualize the motion. You should see the block falling under gravity.

8.15 Add Muscles

To prevent the block from falling through the ground, we create two opposing muscles between the ground and block. ***Note: this code must be added before the call to `initSystem`, or else the muscles are not included in the simulation.***

```
// Create two new muscles using the Thelen 2003 muscle model
double maxIsometricForce = 1000.0, optimalFiberLength = 0.1,
   tendonSlackLength = 0.2, pennationAngle = 0.0,
   activation = 0.0001, deactivation = 1.0;

Thelen2003Muscle *muscle1 = new
   Thelen2003Muscle("muscle1", maxIsometricForce,
      optimalFiberLength, tendonSlackLength, pennationAngle);
muscle1->setActivationTimeConstant(activation);
muscle1->setDeactivationTimeConstant(deactivation);

Thelen2003Muscle *muscle2 = new
   Thelen2003Muscle("muscle2", maxIsometricForce,
      optimalFiberLength, tendonSlackLength, pennationAngle);
muscle2->setActivationTimeConstant(activation);
muscle2->setDeactivationTimeConstant(deactivation);

// Specify the paths for the two muscles
muscle1->addNewPathPoint("muscle1-point1", ground, Vec3(0.0,0.05,-
   0.35));
muscle1->addNewPathPoint("muscle1-point2", *block, Vec3(0.0,0.0,-
   0.05));
muscle2->addNewPathPoint("muscle2-point1", ground,
   Vec3(0.0,0.05,0.35));
muscle2->addNewPathPoint("muscle2-point2", *block,
   Vec3(0.0,0.0,0.05));

// Add the two muscles (as forces) to the model
osimModel.addForce(muscle1);
osimModel.addForce(muscle2);
```

8.16 Prescribe Muscle Controls from Functions

We define the control values for each muscle as a linear function of time defined by the slope of the line and its intercept (value when time=0). We define two linear function one for each muscle in the tug-of-war.

```
// Create a prescribed controller that simply applies controls as
   function of time
   PrescribedController *muscleController = new
   PrescribedController();
      muscleController->setActuators(osimModel.updActuators());
// Define linear functions for the control values for the two
   muscles
   Array<double> slopeAndIntercept1(0.0, 2);  // array of 2 doubles
   Array<double> slopeAndIntercept2(0.0, 2);
// muscle1 control has slope of -1 starting 1 at t = 0
   slopeAndIntercept1[0] = -1.0/(finalTime-initialTime);
   slopeAndIntercept1[1] = 1.0;
// muscle2 control has slope of 1 starting 0.05 at t = 0
   slopeAndIntercept2[0] = 1.0/(finalTime-initialTime);
   slopeAndIntercept2[1] = 0.05;

// Set the indiviudal muscle control functions for the prescribed
   muscle controller
   muscleController->prescribeControlForActuator("muscle1", new
      LinearFunction(slopeAndIntercept1));
   muscleController->prescribeControlForActuator("muscle2", new
      LinearFunction(slopeAndIntercept2));
```

8.17 Define the Initial Activation and Fiber Length States

In addition, we define the initial activation and fiber length of each muscle. Once these parameters are set, we initialize the states for each muscle.

```
// Define the initial states for the two muscles
// Initial activation correspond to control at time=0
muscle1->setDefaultActivation(slopeAndIntercept1[1]);
muscle2->setDefaultActivation(slopeAndIntercept2[1]);
// Fiber length
muscle2->setDefaultFiberLength(0.1);
muscle1->setDefaultFiberLength(0.1);
// Compute initial conditions for muscles
osimModel.computeEquilibriumForAuxiliaryStates(si);
```

Exercise 5: After we compile and run the current main program, we can load the motion in the OpenSim GUI (same file name *tugOfWar_states_degrees.mot* as before) and visualize the simulation.

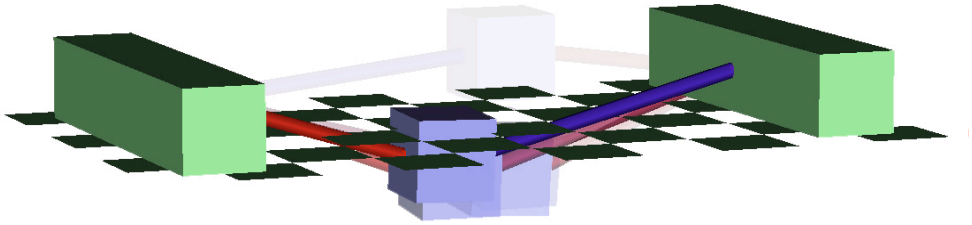


Figure 8.4: Simulation of a falling block suspended by muscles

Except for the colors, you should see something like the above in the GUI. You should see the block falling under gravity but then restrained by the muscles.

8.18 Add Contact

As you have seen, *display* geometry does not cause contact forces. To prevent the block from penetrating the floor, we create some *contact* geometry and an elastic foundation force between the floor and a cube.

```
// Create new contact geometry for the floor and a cube
// Create new floor contact halfspace
ContactHalfSpace *floor = new ContactHalfSpace(SimTK::Vec3(0),
   SimTK::Vec3(0, 0, -0.5*SimTK::Pi), ground);
floor->setName("floor");
// Create new cube contact mesh
OpenSim::ContactMesh *cube = new
   OpenSim::ContactMesh("blockRemesh192.obj", SimTK::Vec3(0),
   SimTK::Vec3(0), *block);
cube->setName("cube");

// Add contact geometry to the model
osimModel.addContactGeometry(floor);
osimModel.addContactGeometry(cube);

// Create a new elastic foundation force between the floor and cube.
OpenSim::ElasticFoundationForce *contactForce = new
   OpenSim::ElasticFoundationForce();
OpenSim::ElasticFoundationForce::ContactParameters contactParams;
contactParams.updGeometry().append("cube");
contactParams.updGeometry().append("floor");
contactParams.setStiffness(1.0e8);
contactParams.setDissipation(0.01);
contactParams.setDynamicFriction(0.25);
contactForce->updContactParametersSet(). append(contactParams);
contactForce->setName("contactForce");

// Add the new elastic foundation force to the model
osimModel.addForce(contactForce);
```

Exercise 6: After we compile and run the current main program, we can load the motion in the OpenSim GUI (same file name *tugOfWar_states_degrees.mot* as before) and visualize the simulation. ***Note: Make sure the file* blockRemesh192.obj *is in your working directory.***

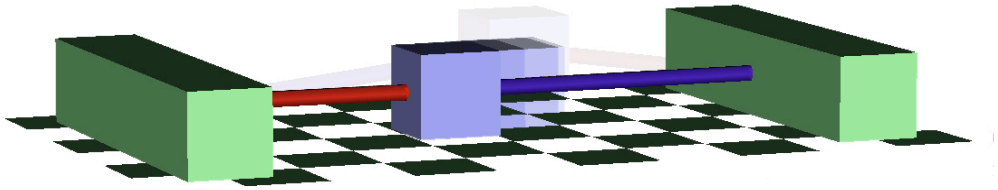


Figure 8.5: Muscle actuated block gliding on a contact surface

Except for the colors, you should see something like the above in the GUI. You should see that the block no longer falls through the floor.

8.19 Add a Prescribed Force

To push the block during the tug-of-war, we create a prescribed force to apply to the block. The prescribed force is applied in the x-direction in the block body's frame. The point of application varies linearly from (0, 0, 0) to (0.1, 0, 0) during the simulation.

```
// Specify properties of a force function to be applied to the block
double time[2] = {0, finalTime}; // time nodes for linear function
double fXofT[2] = {0, -blockMass*9.80665*3.0}; // force values at t1
   and t2
double pXofT[2] = {0, 0.1}; // point in x values at t1 and t2

// Create a new linear functions for the force and point components
PiecewiseLinearFunction *forceX = new PiecewiseLinearFunction(2,
   time, fXofT);
PiecewiseLinearFunction *pointX = new PiecewiseLinearFunction(2,
   time, pXofT);

// Create a new prescribed force applied to the block
PrescribedForce *prescribedForce = new PrescribedForce(block);
prescribedForce->setName("prescribedForce");

// Set the force and point functions for the new prescribed force
prescribedForce->setForceFunctions(forceX, new Constant(0.0), new
   Constant(0.0));
prescribedForce->setPointFunctions(pointX, new Constant(0.0), new
   Constant(0.0));

// Add the new prescribed force to the model
osimModel.addForce(prescribedForce);
```

Exercise 7: After we compile and run the current main program, we can load the motion in the OpenSim GUI (same file name *tugOfWar_states_degrees.mot* as before) and visualize the simulation.

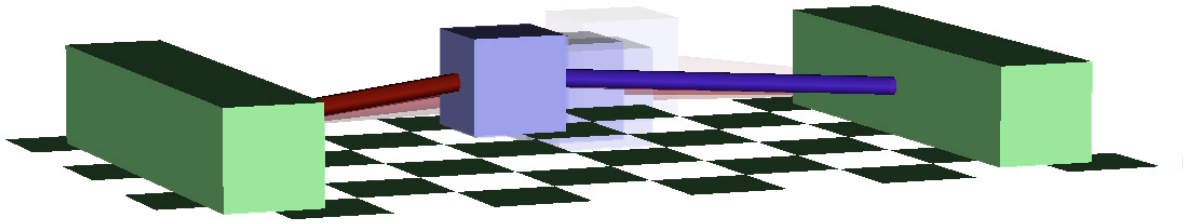


Figure 8.6: Muscle-actuated block with additional perpendicular prescribed force

Except for the colors, you should see something like the above in the GUI. Using the motion slider and video controls, visualize the motion. You should see that the block now responds to the prescribed force as well as the muscle controls.

8.20 Adding a Built-in Analysis

Generally, we would like to report various quantities while running a simulation. In this example, we'd like to report the forces that were applied to the model while running the forward simulation, so that we can troubleshoot the simulation and validate it. To get this effect, we will add in one of the built-in analyses that come with OpenSim. The specific Analysis subclass we will use in this case is ForceReporter. Attaching this analysis to the simulation will cause the values of the forces applied to the model to be reported in a storage file at the end of the simulation.

OpenSim provides a set of Analysis subclasses for convenience, in particular:

- Kinematics
- PointKinematics
- Actuation
- ForceReporter

To create the analysis for this step requires adding the following lines of code before we integrate the model forward:

```
ForceReporter* reporter = new ForceReporter(&osimModel);
osimModel.addAnalysis(reporter);
```

After the integration is done, we add the line:

```
reporter->getForceStorage().print("tugOfWar_forces.sto");
```

This will create a file with columns corresponding to the forces in the muscles and the applied prescribed forces.

8.21 Add a Constraint

In this section, our goal is to create a constraint such that the motion of the block is along a specified line. The line we specify will be represented by the vector **(1, 0, -1)**, which will constrain the motion of the block on a 45° angle between the two anchor points.

```
// Specify properties of a point on a line constraint to limit the
   block's motion
Vec3 lineDirection(1,0,-1);
Vec3 pointOnLine(1,0,-1);
Vec3 pointOnFollowerBody(0,-0.05,0);

// Create a new point on a line constraint
PointOnLineConstraint *lineConstraint = new
   PointOnLineConstraint(ground, lineDirection, pointOnLine, *block,
   pointOnFollowerBody);

// Add the new point on a line constraint to the model
osimModel.addConstraint(lineConstraint);
```

Exercise 8: This is the final step of this example. The complete program can also be found in the file *TugOfWar_Complete.cpp*. You can use CMake to generate a new solution file with this as the TARGET, or manually replace the previous source file with this one, from within Visual Studio.

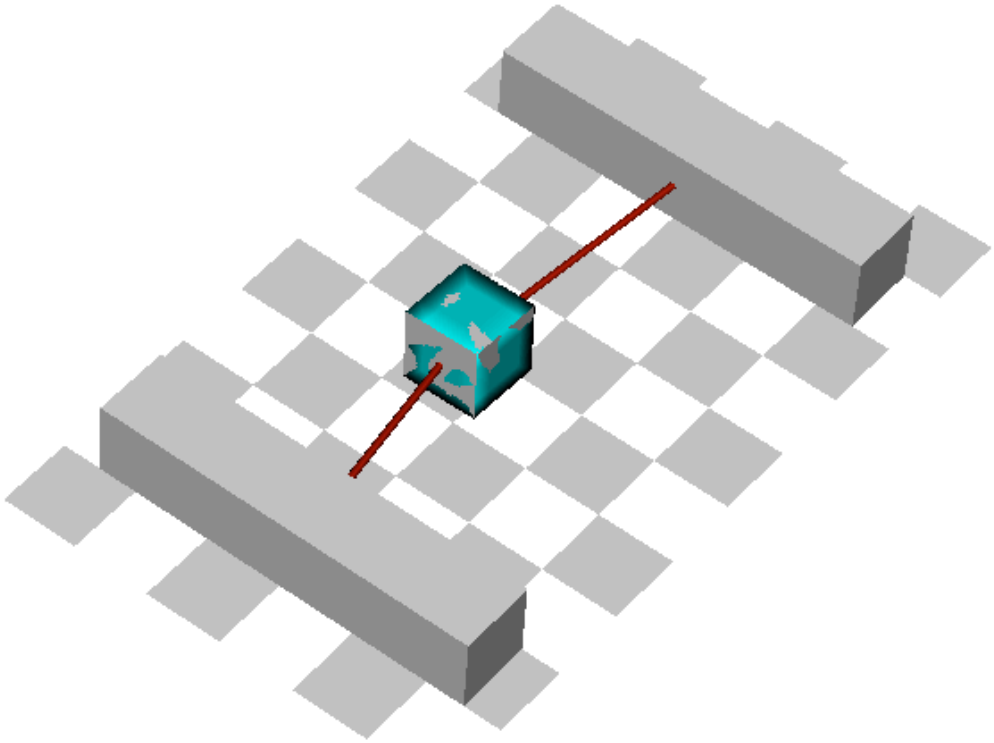


Figure 8.7: Final simulation.

After we re-compile and run the current main program, we can load the new motion file in the OpenSim GUI (same file name *tugOfWar_states_degrees.mot* as before) and visualize the simulation. If you look at the animation from a top view as above, you should see that the motion of the block is now restricted to traveling along a diagonal line.