Common Scripting Commands

Scripting environments like Matlab, Python and the OpenSim GUI shell allow users to interact with the classes of the OpenSim API (see Introduction to the OpenSim API). There are many example scripts that are located in the OpenSim scripting folder, available with the distribution to help you get started. This page is resources of some of the common calls that you will make in the scripting environment of your choice.

Beginner Scripting Resources

- Run the example scripts found in the distribution.
- As you run the example scripts, find the related Classes and Methods in Doxygen
- The structure for classes can often be found in OpenSim's files (.xml or .osim). For example, Setup_Scale.xml shows the hierarchy of the ScaleTool.
- Use the OpenSim Search bar to search for a Class or Method in the documentation, on the forum or on the Doxygen pages.
- If you are having trouble with a Class or Method, ask a question on the OpenSim forum.

Inline Method and Type getting

These are inline commands that help you find the methods available for a class

Command	Platform	Description
methodsview('Model') or methodsview (osimModel)	GUI, Matlab	Examine the methods available for a class (e.g. Model) or for an existing object that you've created (e.g. osimModel)
type(aObject)	GUI, Python	Prints the full qualified type of aObject
Up/Down Arrows at the command prompt	GUI, Interactive Python	Recalls past commands entered at the prompt
Tab completion	Matlab, Interactive Python	Define an OpenSim model or object then use tab completion in Matlab to see the available methods
dir(aObject)	Python	Returns a list of methods available on aObject

Packages and Libraries

Packages and libraries are collections of classes and methods (see Introduction to the OpenSim API for background) that have a well-defined interface and can be imported into your programming environment to utilize. These can be packages to browse files, read and write files, do mathematics operations, and run simulations. The commands below are common packages or libraries you will find useful.

Package	Platform	Description
java.swing	GUI	Swing is the generic GUI kit for Java, allows for I/O and creating windows if needed. Examples of usage can be found here.
java.lang	GUI	Lang is the Java language core libraries e.g String, Math, etc. To use particular libraries (for example Math), do "from java. lang import Math"
java.io	GUI	IO is the package in Java that's used to perform input/output operations including file reading/writing.
org. opensim. modeling	GUI and Matlab	Modeling classes from the OpenSim API. This package is automatically imported in the GUI Shell. In Matlab, excute command import org.opensim.modeling.* to avoid having to type org.opensim.modeling.'Class'

org.opensim.utils	GUI	Provides some convenient file browsing options

To import packages in the GUI, type, for example:

```
>>> import java.io as io                              # Import Read/Write package
>>> trcFolder = io.File(trcDataFolder)        # Use package to read TRC file
```

Creating New Objects

Creation of objects is performed by the constructor method of the Class. We can create an object and populate its properties, or pass the class a correctly formatted xml file.
Note: 'path' indicates your system path to the file (i.e., C:/Users/<username>/Documents/OpenSim/4.0/Models/...).

Note: Delete "modeling." from code for Matlab.

Action	Class Information	Default construction (Scripting shell)	XML construction (Scripting shell)
Marker set object	OpenSim::MarkerSet	markerSet = modeling.MarkerSet()	markerSet = modeling.MarkerSet([*path* 'markerSet.xml'])
Marker data object	OpenSim::MarkerData	markerData = modeling.MarkerData()	markerData = modeling.MarkerData([*path* 'walkingData.trc'])
Scaling tool object	OpenSim::ScaleTool	scaleTool = modeling.ScaleTool()	scaleTool = modeling.ScaleTool([*path* 'scale_setup.xml'])
Inverse Dynamics analysis object	OpenSIm::InverseDynamicsTool	idTool = modeling.InverseDynamicsTool()	idTool = modeling.InverseDynamicsTool([*path* 'ID_setup.xml'])
Control Set	OpenSim:ControlSet	controlSet = modeling.ControlSet()	controlSet = modeling.ControlSet([*path* 'controls.mot'])

Building a MarkerSet Object from file and attaching it to a Model

```
>>> myModel = modeling.Model("gait2354.osim")              # Load a Model from file
>>> markerSetFile = "gait2354_MarkerSet.xml"               # Define the full path to the MarkerSet file
>>> newMarkers = modeling.MarkerSet(myModel, markerSetFile) # Construct a MarkerSet Object
>>> myModel.updateMarkerSet(newMarkers)                    # Append newMarkers to the Model's MarkerSet,
replacing markers with the same name
```

Building a MarkerSet Object Programmatically and attaching it to a Model

```
>>> myModel = modeling.Model("gait2354.osim")                                  # Create a Model Object
>>> pelvisbody = myModel.getBodySet().get("pelvis")                            # Get a handle to the pelvis body
>>> newMarker = modeling.Marker("LASI", pelvisbody, modeling.Vec3(1,1,1))   # Create a Marker called 'LASI'
that is attached to the pelvis at (1,1,1)
>>> myModel.addMarker(newMarker)                                               # Add newMarker to the Model
```

Building a Body Programmatically in Matlab

```
>> newBody  = Body();                                        % Creates a body
>> massCenter = Vec3(-0.0707,0.0,0.0);                       % Create a Vec3 Object with mass center information
>> inertia = Inertia(0.128, 0.0871, 0.0579, 0, 0, 0);        % Create an Inertia object with only principal axes
>> newBody.setName('pelvis');                                % Set the name of the body
>> newBody.setMass(11);                                      % Set the mass of the body
>> newBody.setMassCenter(massCenter);                        % Set the mass center
>> newBody.setInertia(inertia);                              % Set the body's inertia

%% Alternatively, a body can be created by sending all of these properties directly to the constructor
>> massCenter = Vec3(-0.0707,0.0,0.0);                       % Create a Vec3 Object with Mass center information
>> inertia = Inertia(0.128, 0.0871, 0.0579, 0, 0, 0);        % Create an Inertia object with only principal axes
>> bodyName = 'pelvis';                                      % Set the name of the body
>> bodyMass = 11;                                            % Set the mass of the body
>> newBody = Body(bodyName, bodyMass, massCenter, inertia);  % Creates a body from properties sent to
constructor
```

Create a Joint programmatically in Matlab

```
>> pBody = Body('pelvis', bodyMass, massCenter, inertia);       % Create the parent body
>> cBody = Body('femur_r', bodyMass, massCenter, inertia);      % Create the child body
>> name = 'hip_r';                                              % Joint name
>> locInParent = Vec3(-0.0707,-0.0661,0.0835);                  % Location of the joint origin expressed in the
parent frame
>> oriInParent = Vec3(0,0,0);                                   % Orientation of the joint frame in the parent
frame
>> locInChild = Vec3(0,0,0);                                    % Location of the joint origin expressed in the
child frame
>> oriInChild = Vec3(0,0,0);                                    % Orientation of the joint frame in the child
frame
>> rHip = BallJoint(name, pBody, locInParent, oriInParent, cBody, locInChild, oriInChild); % Construct the hip
joint
```

Create a Torque Actuator for the Knee Joint (flex/ext) programmatically in Matlab

```
>> myModel = Model('gait2354.osim');               % Create a Model Object from file
>> femur_r = myModel.getBodySet().get('femur_r');  % Get a handle to the femur
>> tibia_r  = myModel.getBodySet().get('tibia_r'); % Get a handle to the tibia
>> zAxis = Vec3(0,0,1);                            % A Vec3 of the z-axis
>> torqueActuator = TorqueActuator();              % Create a TorqueActuator Object
>> torqueActuator.setBodyA(femur_r);               % Set BodyA
>> torqueActuator.setBodyB(tibia_r);               % Set BodyB
>> torqueActuator.setAxis(zAxis);                  % Set the axis about which the torque will act
>> torqueActuator.setOptimalForce(10);             % Set the optimal force (gain) of the actuator
```

Create a Coordinate Actuator programmatically in Python

```
>>> myModel = opensim.Model("arm26.osim")
>>> coordActuator = opensim.CoordinateActuator()
>>> coordActuator.setName('r_elbow_actuator')
>>> coordActuator.setCoordinate(myModel.getComponent('r_shoulder/r_shoulder_elev'))
```

Loading, Creating, and Initializing Models

Loading models and dealing with model states is very common. Below are the methods for loading, copying and initializing a model

Action	GUI Command	Matlab/Python Command
Loads the specified model in the GUI	loadModel(modelFileName)	

Creates a handle to the current model in the GUI	myModel = getCurrentModel()	
Load a model from file (Create Model Object)	myModel = modeling.Model(modelFileName)	myModel = Model('gait2354.osim')
Creates a copy of myModel.	myModelCopy = modeling.Model(myModel)	myModelCopy = Model(myModel)
Initialize the model and get the default state	myState = myModel.initSystem()	myState = myModel.initSystem()

Loading a Model and States in the GUI Scripting Window

```
>>> modelFileName = modelFolder+"/gait2354_simbody.osim" # Define the full path to the model file
>>> loadModel(modelFileName)                             # Load the model into the GUI
>>> myModel = getCurrentModel()                          # Create a handle to the current model
>>> myState = myModel.initSystem()                       # Initialize the model and obtain the default state
```

Exploring and Editing Models and Model Components

You can use the functionality of the OpenSim API to access the properties of a model and change their values. For examples, refer to the distributed scripts muscleScaler and alterTendonSlackLength. The OpenSim API Doxygen lists all of the available functions. For example, executing the following commands in the GUI scripting shell, sets and gets the name of the current model.

```
>>> myModel = getCurrentModel()
>>> myModel.setName("My Model")
>>> mymodel.getName()
>>> "my Model"
```

You should generally avoid adding and removing objects from a model that is "live" in the OpenSim GUI. Instead you should make a copy of the model, make additions and deletions, then reload in the GUI.

The API also allows you to access and edit the components of an OpenSim model, like its bodies, muscles, and joints. Some properties have custom "get" and "set" functions - see the respective classes for details.

Action	Class Information	Getting a Handle to a Set	Reference an Object by index	Get object from a name	Alternative 'long path'
Body Set	OpenSim::BodySet	bodySet= myModel.getBodySet()	rightFemur = bodySet().get(1)	rightFemur = bodySet().get("femur_r")	rightFemur =myModel.getBodySet().get("femur_r")
Joint Set	OpenSim::JointSet	jointSet = myModel.getJointSet()	rightHip = jointSet ().get(7)	rightHip = jointSet().get("hip_r")	rightHip = myModel.getJointSet().get("hip_r")
Coordinate Set	OpenSim::CoordinateSet	cordSet= myModel.getCoordinateSet()	hip_coord = cordSet().get(4)	hip_coord = cordSet().get("hip_flexion_r")	hip_coord = cordSet().get("hip_flexion_r")
Muscle Set	Muscles	muscleSet= myModel.getMuscles()	recFemR = muscleSet().get(3)	rec_fem_r = recFemRt().get("'recFem_r")	rec_fem_r= myModel.getMuscles().get("'recFem_r")
Path Point of a Muscle	OpenSim; PathPoint	pathPoints = myModel.getMuscles().get("'recFem_r").getGeometryPath().getPathPointSet()	recFemPathPoint1 = pathPoints.get(0)	recFemPathPoint1 = pathPoints.get("rect_fem_r-P1")	recFemPathPoint1 = myModel.getMuscles().get("'recFem_r").getGeometryPath().getPathPointSet()

Once a Handle to the component has been created you can edit its properties and methods.

Action	Class Information	Example
Get the optimal fiber length	OpenSim::Muscle::getOptimalFiberLength ()	recFemFiberLength = RectusFemoris.getOptimalFiberLength()
Set the optimal fiber length	OpenSim::Muscle::setOptimalFiberLength()	RectusFemoris.setOptimalFiberLength(0.23)
Set the tendon slack length	OpenSim::Muscle::setTendonSlackLength()	RectusFemoris.setTendonSlackLength(0.2105)
Get the muscle maximum isometric force	OpenSim::Muscle::getMaxIsometricForce()	recFemMaxForce = RectusFemoris.getMaxIsometricForce()

Change the ECRB muscle properties in the GUI

```
>>> ECRB = myModel.getMuscles().get("ECRB")              # Get a handle to the ECRB
>>> backupTendonSlackLength = ECRB.getTendonSlackLength() # Back up the original tendon slack length (just in
case)
>>> ECRB.setTendonSlackLength(0.2105)                    # Prescribe a new Tendon slack length
>>> myModel.initSystem()                                 # Re-initialize the model
```

You may need to downcast an object from an abstract class (e.g., Muscle) to a derived class (e.g., Thelen2003Muscle) in order to gain access to its properties and methods. Here is an example:

```
>> import org.opensim.modeling.*
>> myModel = Model('arm26.osim');
>> mcl_TRIlong = Thelen2003Muscle.safeDownCast( myModel.getMuscles().get('TRIlong') );
>> mcl_TRIlong.setFmaxTendonStrain( 0.5*mcl_TRIlong.getFmaxTendonStrain() );
```

Parent and Concrete Classes

If you are calling a method or function (e.g., getting or setting properties) that you are pretty sure should work, but you are getting an error that the method doesn't exist, this may mean that you need to downcast. In the C++ programming language, programmers use a concept called "inheritance" to build up complexity without re-writing the same code multiple times. For example, in OpenSim Thelen2003Muscle and MillardMuscle both rely on code in the common parent Muscle class that they "inherit" from.

If you have a handle to a base class object (e.g. Muscle) you may need to downcast the object to one of its derived (or concrete) classes, like the Thelen2003Muscle, in order to gain access to properties and methods specific to the concrete class. In the below example, we get a reference to a muscle in the model and return the class name and concrete class name;

```
>>> model = Model(path2model)
>>> muscle = model.getMuscles().get(0);
>>> muscle.getClassName()
muscle
>>> muscle.getConcreteClassName()
Thelen2003Muscle
```

Then to get a reference to the concrete class you use the safeDownCast() method.

```
>>> muscle = model.getMuscles().get(0);    # The object you get here is of base class Muscle
>>> thelenMuscle = Thelen2003Muscle.safeDownCast(muscle)  # To use a method specific to Thelen2003Muscle you
need to safeDownCast
>>> timeConstant = thelenMuscle.getActivationTimeConstant() # Get property that is present is specific to
Thelen2003Muscle
```

Outputs

OpenSim 4.0 uses component outputs and reporters to collect variables of interest and print them to file. To display the output names for a component, use the method getOutputNames();

```
>>> muscle.getOutputNames();
```

Reading Data from TRC Files into TimeSeriesTable

To read marker trajectories from a trc file into a TimeSeriesTableVec3, construct a table by passing the filename into the TimeSeriesTableVec3 constructor. Afterwards, you can query the TimeSeriesTable for the data as shown in the code snippet below.

```
>>> markerData  = TimeSeriesTableVec3('markerdata.trc');
>>> numDataRows = markerData.getNumRows();
>>> numMarkersInTable = markerData.getNumColumns();
```

Using the Tools

Tools contain a number of grouped Methods that allow you to run a study. For example, to scale a Model to match experimental data the ScaleTool groups GenericModelMaker(), ModelScaler() and MarkerPlacer() together. The AnalyzeTool() can group StaticOptimization() and MuscleAnalysis() together to output muscle states of a Static Optimization. Tools can be initialized from a setup file (.xml) that has stored settings. They also have methods that methods can be called to change the input models, data files, and some settings. Use the API documentation or methodsview() in Matlab (described above) to explore the methods that are available for the Tool you are using.

Command or Class	Platform	Action
scale = ScaleTool('Scale_Setup_file.xml')	Matlab	Returns a Scaletool object with properties defined in the Scale_Setup_File.xml
ik = InverseKinematicsTool() ik.run()	Matlab	Returns an InverseKinematicsTool object with default properties. You must set the properties of the tool. Begin the Inverse Kinematics simulation by calling the run() method.
rra = RRATool'RRA_Setup.xml') rra.setAdjustCOMToReduceResiduals(1) rra.run()	Matlab	Returns an RRATool object with properties defined in the RRA. Allow RRA to alter the trunk COM by using the setAdjustCOMToReduceResiduals() method. Begin the RRA simulation by calling the run() method.
so = AnalyzeTool('SO_Setup.xml')	Matlab	Returns an AnalzeTool that has a Static Optimization analysis included.

Working with Vectors, Matrices, and Other SimTK Classes

we've exposed the most commonly used SimTK classes. In particular:

Command	Platform	Description
modeling.Vec3() modeling.Vec3(double e) modeling.Vec3(double e0, double e1, double e2)	All	Creates a Simtk Vec3 object (a vector of length 3). If passed only one argument (double e) all elements will be set to e.
modeling.Vector(int length, double e)	All	Creates a Simtk Vector with specified length. All elements are set to e.
modeling.Mat33(double e) modeling.Mat33(double e0, ... , double e8)	All	Creates a Simtk Mat33 object (a 3x3 matrix). If passed only one argument (double e), the diagonal elements will be set to e and other elements set to zero.
modeling.Inertia() *See doxygen link at right for additional constructors*	All	Creates a Simtk Inertia object. All constructors are available except symmetric matrix constructors.
modeling.State()	All	Creates a Simtk State object. See the doxygen link for more information.
modeling.Stage() modeling.Stage(int level)	All	Creates a Simtk Stage object, optionally realized to level l. See the doxygen link for more information.

Helpful tips:

- For Simbody doxygen links above Vec3P corresponds to a modeling.Vec3 object and RealP corresponds to a double value.
- To see the available methods for these objects, use methodsview() or tab completion (Matlab only).

- You can find more information in the SimTK Basics section of the User's Guide. Note that only the SimTK classes listed above are available through scripting.
- You can also find more information in the section below on Handling C++ Templates.

ArrayDouble

In many cases the function you're trying to call takes an argument type different from the object you have already. This is true even moving data between Matlab and Java objects, and between objects in the SimTK namespace and those in the OpenSim namespace. In particular, many OpenSim methods return an ArrayDouble (Array<double) and you would prefer to convert the array to a different type. The following set of convenience methods are "Adaptors" intended to help you pass data around between OpenSim objects and low level SimTK objects.

For Matlab, leave off "modeling". For Python, change "modeling" to "opensim".

Command	Platform	Description
modeling.ArrayDouble.createVec3([0.0,0.05,0.35])	All	Creates a SimTK::Vec3
modeling.ArrayDouble.getAsVec3()	All	returns SimTK::Vec3 populated from ArrayDouble of size 3.
modeling.ArrayDouble.getAsVector()	All	return SimTK::Vector populated from ArrayDouble
modeling.ArrayDouble.populateFromVector(SimTK::Vector aVector)	All	populate an ArrayDouble from the passed in SimTK Vector
modeling.ArrayDouble.getValuesFromVec3(SimTK::Vec3 vec3)	All	returns an ArrayDouble populated from the passed in SimTK Vec3

When referring to indexed elements remember that Matlab begins indexing at 1 while OpenSim data structures begin at 0.

Advanced Multibody Calculations with Simbody

In 4.0, you can perform advanced multibody calculations in scripting via the SimbodyMatterSubsystem inside the OpenSim Model (model. getMatterSubsystem()). The SimbodyMatterSubsystem class allows you to compute the mass matrix, Jacobians, inverse dynamics moments, etc.

Use the Simbody Inverse Dynamics Operator in Matlab

```
model = Model('arm26.osim');
s = model.initSystem();
% For the given inputs, we will use the inverse dynamics operator
% (calcResidualForce()) to compute the first column
% of the mass matrix. We accomplish this by setting all inputs to 0
% except for the acceleration of the first coordinate.
%   f_residual = M udot + f_inertial + f_applied
%              = M ~[1, 0, ...] + 0 + 0
model.realizeVelocity(s);
appliedMobilityForces = Vector();
appliedBodyForces = VectorOfSpatialVec();
knownUdot = Vector(s.getNU(), 0.0);
knownUdot.set(0, 1.0);
knownLambda = Vector();
residualMobilityForces = Vector();
smss = model.getMatterSubsystem();
smss.calcResidualForce(s, appliedMobilityForces, appliedBodyForces, ...
                       knownUdot, knownLambda, residualMobilityForces);
```

Class Templates (Vec3(), Array<double>, Vector())

A summary of Templatized Class use in scripting can be found on the Scripting Versions of OpenSim C++ API Calls page.

Templates are advanced C++ constructs that are used extensively throughout the OpenSim API and Simbody API. If you see notation like Array<double> in the doxygen or C++ code that you are trying to replicate, this means you're working with a templatized class and, to use that class in scripting, will need to find its appropriate mapping in the scripting environment. For more information, and a easy to use ummary of the C++ to scripting mappings, see the Scripting Versions of OpenSim C++ API Calls page.

Obtaining Position and Velocity Information

For more information regarding multibody system states, refer to the SimTK Simulation Concepts documentation in the Developer's Guide.

In order to obtain simulation position or velocity state information you must have a State object in hand.

Using State Objects

```
>>> state = myModel.initSystem();
>>> myModel.equilibrateMuscles(state);
```

Now you can use methods on Body objects (actually, from the Frame class) to obtain the location or velocity of a point in the ground frame.

```
>>> body = myModel.getBodySet().get('r_ulna_radius_hand')
>>> massCenter = body.getMassCenter(massCenter)
>>> body.getStationVelocityInGround(state, massCenter)
>>> dir(body) # List other methods available on Body.
```

Using the Simbody Visualizer

You can use the visualizer from Simbody in Matlab and Python. To do so call the "setUseVisualizer" method and pass in the parameter "true", and when you run the simulation the Simbody Visualizer GUI will pop up. The example of usage is described in "TugOfWar_CompleteRunVisualizer.m"

```
osimModel = Model('tug_of_war_muscles_controller.osim');
osimModel.setUseVisualizer(true);
```

Set verbosity or logging level

Applies to OpenSim 4.2 (unreleased) and above.

You can control how much detail is printed to the console using the following:

```
Logger.setLevel("Off");
Logger.setLevel("Critical");
Logger.setLevel("Error");
Logger.setLevel("Warn");
Logger.setLevel("Info"); % default
Logger.setLevel("Debug");
Logger.setLevel("Trace");
```

When performing batch processing or optimization you can turn off logging completely by calling Logger.setLevel("Off").

You can always add a new file during runtime to start logging into (in addition to whatever the current logging behavior is). This can be done by calling

```
Logger.addFileSink('logs_folder/full_path.log');
```

Make sure you have permission to write to the logs_folder.

Batch Processing

There are several examples in the Matlab scripts and GUI scripts that show how to perform batch processing by calling the OpenSim API (e.g. Analyze, IK). We encourage you to use this approach rather than using Matlab's xml parsing tools. To read more about why this is the case, please see the scripting FAQ:

Frequently Asked Questions

References

- Java File IO: http://docs.oracle.com/javase/1.4.2/docs/api/java/io/File.html
- Using Java in Jython: http://www.jython.org/jythonbook/en/1.0/JythonAndJavaIntegration.html
- Jython Swing examples: http://wiki.python.org/jython/SwingExamples

Next: Scripting in the GUI

Previous: Scripting

Home: Scripting and Development