A Very Very Brief Intro to Mujoco

Professor: Oliver Kroemer
TA: Steven Brills

About the Simulator:

Mujoco uses a generalized co-ordinate representation under the hood to maintain all the state information of the model at any given instant of time. What this means is that the model state is represented in the "joint" configuration at any given time and Cartesian states are derived from these generalized coordinates. As a result, many typical operations required for robotics can be implemented very efficiently.

The World & Robot Model (Internally referenced as mjModel):

Your typical Mujoco program will contain two data structures that you will initialize at the start. The **`mjModel`** and the **`mjData`**. The **`mjModel`** reads in the model description from an MJCF scene description language which is an xml file format. If you have worked with URDFs in the past, the MJCF has similar underlying ideas.

In the course homework assignments, we will be using MJCF descriptions for the FrankArm that are a reproduction of the ones found in **[1]**. The ones for this assignment are distinct from the one found in the source in that the actuators in the model are torque controlled actuators at the lowest level instead of position controlled which will allow you to implement gravity compensation, impedance, force and other controllers over it.

The names of the MJCF files that are available are listed below:

1. **panda_nohand_torque.xml** : An empty world with a single Franka arm without an end-effector
2. **panda_nohand_torque_fixed_board.xml** : A world with a single Franka arm and a whiteboard
3. **panda_nohand_torque_sine_board.xml** : A world with a single Franka arm and an actuated whiteboard that moves based on a sine wave control input against the end-effector.

The first step in setting up the simulation would be to load up the model file as shown in the snippet below, snippet 1 and snippet 2:

```
# Set the XML filepath
xml_filepath = "../franka_emika_panda/panda_nohand_torque.xml"
```

Figure 1: XML Filepath to the MJFC Model Description

```
# Load the xml file here
model = mj.MjModel.from_xml_path(xml_filepath)
```

Figure 2: Model Object Instantiation

This imports all model data into the simulation environment. MJCF files can include descriptions of light sources, camera viewports in the world and many other objects that can influence simulation, environmental interaction and rendering. More details on the MJCF XML elements and their properties can be found at **[2]**. Take a look at a simple XML example from the python tutorial notebook **[3]**. Once you are done with that try to go through the Franka MJCF description. If the first bit that lies between the "<default>" tags confuse you, just remember that everything defined between the default tag are exactly that, default values so that you don't repeat yourself when defining the actual world. Each set of defaults are mapped to a user-defined string name which you can attach parts of your model to so that it copies default settings from there. For example you can define a joint default if 5 out 7 joints have the same joint limits, and name "joint_default". Every time you define a joint, just derive it from the defaults class and if you wish to override the joint limits, explicitly over write it when you are defining the 6th and 7th instance of a joint.

The World & Robot State (Internally referenced as mjData):

Next up is the **`mjData`** data structure. This data structure stores all the important states of the elements in the model in generalized coordinates. Now, given that Mujoco maintains state in generalized coordinates, the Cartesian poses of objects in the world are never updated unless explicitly asked to do so by calling **`mj_kinematics`** which takes the current state in generalized coordinates and computes all the Cartesian poses for the objects.

```
data = mj.MjData(model)
```

Figure 3: Instantiating Data Object

When running the homework files, we simply load the model and instantiate the mjData object after which we open the interactive simulator. Once we call the interactive simulator, under the hood it calls **`mj_forward`** which in turn calls **`mj_kinematics`** along with the forward simulation physics update step. So you will not see an explicit call to `mj_kinematics` in the code since we are offloading that task to the interactive simulator.

The mjData object stores some of the following important state and input data:

1. Generalized coordinate states for all joints
2. Cartesian position and orientation for all the bodies, sites and joints in the model
3. Control inputs for all actuators

These can be accessed as shown in the snippets below:

```
print("Joint errors: ", (desired_joint_positions-data.qpos[:7]))
```

Figure 4: Accessing Generalized Co-ordinates for Joint Positions

`data.qpos` in snippet 4 stores all the joint states in the model world. Remember that this includes not just the robot joints but other joints in the world such as the double slider joint that a hockey puck might have on a 2D plane floor. You may only want to access the robot's joint values at any given time.

```
# Instantite a handle to the desired body on the robot
body = data.body("hand")
```

Figure 5: Instantiating a Handle to the Desired Body in the Model

In snippet 5, `data.body("hand")` instantiates a handle to the body named "hand" from the MJCF file. Once you have this handle, you can access attributes such as `body.xpos` and `body.xquat` which give you the Cartesian position and orientation of the body.

```
data.ctrl[:7] = data.qfrc_bias[:7]
```

Figure 6: Accessing Control Inputs and the Coriolis and Gravity Terms in Data Object

Finally, in snippet 6 `data.ctrl` gives you access to all the controlled actuators defined as part of the model. There may be a multitude of actuators in the world and slicing will allow you to address those specific actuators that you intend to control. `data.qfrc_bias` above stores all the computed Coriolis and gravity force in generalized coordinates, aka, as joint torques for revolute joints on your robot. Gravity compensation is simply feeding this back into the joint controller as seen in the snippet above.

Viewer:

The viewer is called by passing the model and data to the viewer.launch method in Mujoco. Underneath the hood, Mujoco calls it's simulate.cc program which runs the physics engine and integrates the states over time. The viewer opens an interactive window where you can monitor sensor outputs, visualize the world, frames, joints, objects, inertias and many other model features as you would in any other simulator as well as interact with the robot directly by applying forces and torques using your mouse. Figure 8 demonstrates this interaction.

```
# Launch the simulate viewer
viewer.launch(model, data)
```

Figure 7: Calling the Viewer Launch Method for the Interactive Viewer

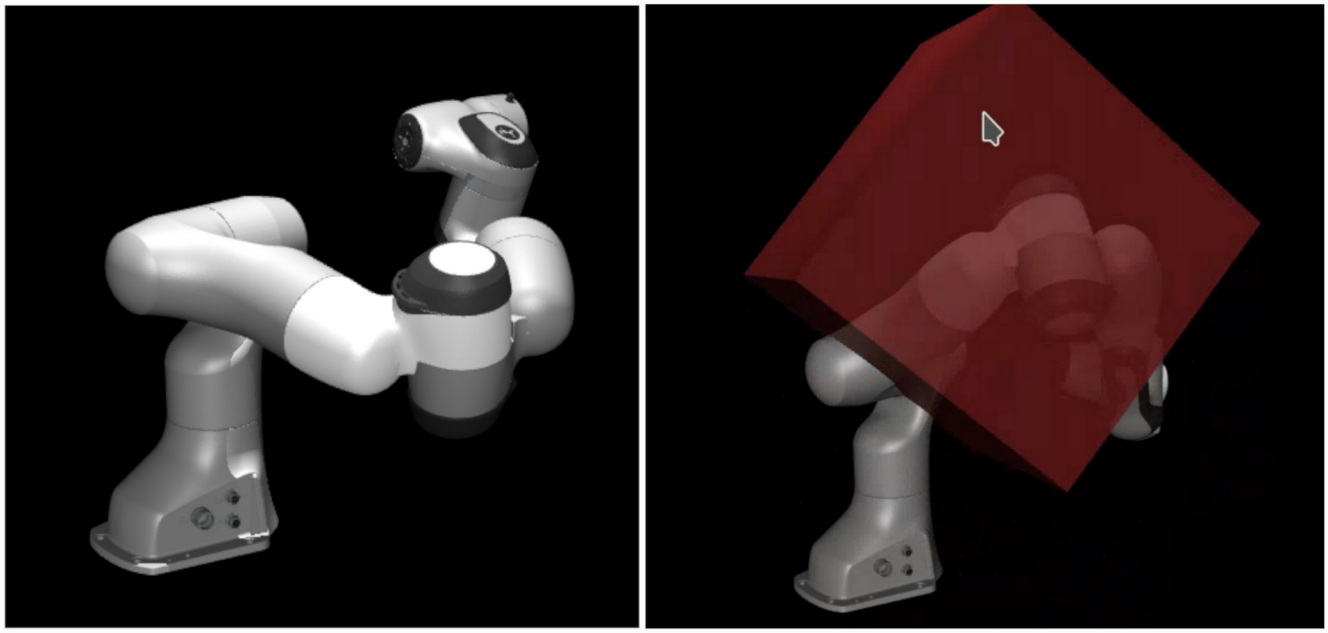

Figure 8: (Left) Selecting the 4th Link on the Robot from the Base, (Right) Applying a Force on the Link

Custom Callbacks:

Mujoco gives you the option to define custom callbacks to generate your own derived data. You may want this to output your own processed data, say for a custom sensor. When set to a user-defined callback, the simulator will call on your defined function to fill the data that you want to provide to it.

We will be making use of this feature while implementing the controllers for the homework. Before we instantiate the viewer above, we'll tell Mujoco that we want it to use our own controllers by writing the following code seen in snippet 8:

```
mj.set_mjcb_control(position_control)
```

Figure 9: Setting Custom Callbacks for Control

In fact, for any callback `mjcb_foo` that is defined in the C API for Mujoco, you can set your own callback in python by calling `mujoco.set_mjcb_foo(<your callable>)` . In snippet 9, we have a callable function named "position_control". Note that any callable function must be defined with the model and data parameters as these are passed to it during simulation.

Possibly Useful Sections from the Documentation

The API reference section of the Mujoco documentation does a good job of highlighting all the available functions, data structures and methods that are exposed to the programmer. Some of the useful sections of the API reference are linked below. The main page is at **[4]**. The following links are subsections.

1. **`mjData`**: https://mujoco.readthedocs.io/en/stable/APIreference.html#mjdata
2. **`mjModel`**: https://mujoco.readthedocs.io/en/stable/APIreference.html#mjmodel
3. **mj_jac and related**: https://mujoco.readthedocs.io/en/stable/APIreference.html#mj-jac (read mj_jac and associated jacobian shortcuts)
4. **Math functions**: https://mujoco.readthedocs.io/en/stable/APIreference.html#standard-math
5. **Vector math**: https://mujoco.readthedocs.io/en/stable/APIreference.html#vector-math
6. **Quaternion math**: https://mujoco.readthedocs.io/en/stable/APIreference.html#quaternions

Credits

The MJCF model was adapted from the one made by Kevin Zakka and Saran Tunyasuvunakool **[1]**.

References

[1] https://github.com/deepmind/mujoco_menagerie/tree/main/franka_emika_panda
[2] https://mujoco.readthedocs.io/en/stable/XMLreference.html#xml-reference
[3] https://colab.research.google.com/github/deepmind/mujoco/blob/main/python/tutorial.ipynb
[4] https://mujoco.readthedocs.io/en/stable/APIreference.html