Unit 2: Adapt URDF for Gazebo Simulator

URDF files are very useful, as you have seen, for having a virtual representation of the different links and joints.  
But you are not simulating its weight, its inertia, what sensors it has, how it collides with other objects, the friction with the floor, or how the servo position control will react to the robot. These are just some of the topics that you are going to learn about in this unit.

0. What you need to appear in Gazebo

To make a model appear in Gazebo you need to add the following:

* Inertias: The inertias and mass of the model is important if you want it to have physics applied
* Gazebo Physical Properties: Here you specify the frictions and material properties like colour or softness.
* Collisions: Without collisions the Robot models would just go through the floor.
* Visual Properties: We add colour to the visual, which only will affect the URDF in RVIZ, the colour in the simulation is defined in the **Gazebo Physical Properties**.
* Gazebo Sensors: Here is when you add cameras, motor controllers and all the robots systems.

1. Mira\_Simple Gazebo Spawn

We will Spawn the most simple urdf we have created, and we will go from there:



roscd my\_mira\_description

touch urdf/mira\_simple\_collisions\_inertias.urdf

We have to add the **collisions**, **innertias** and **gazebo physical properties** to make it appear in the simulated world.

1.1 Innertias

Here is an example of how the inertias would be added to the base\_link:

In [ ]:



**<!--** **\*** **\*** **\*** Link Definitions **\*** **\*** **\*** **-->**

**<**link name**=**"base\_link"**>**

**<**inertial**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**mass value**=**"0.18" **/>**

**<**inertia ixx**=**"0.0002835" ixy**=**"0.0" ixz**=**"0.0" iyy**=**"0.0002835" iyz**=**"0.0" izz**=**"0.000324"**/>**

**</**inertial**>**

**<**visual**>**

**<**origin rpy**=**"0 0 0" xyz**=**"0 0 0"**/>**

**<**geometry**>**

**<**cylinder radius**=**"0.06" length**=**"0.09"**/>**

**</**geometry**>**

**</**visual**>**

**</**link**>**

Here are some elements to comment on:

In [ ]:



**<**inertial**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**mass value**=**"0.18" **/>**

**<**inertia ixx**=**"0.0002835" ixy**=**"0.0" ixz**=**"0.0" iyy**=**"0.0002835" iyz**=**"0.0" izz**=**"0.000324"**/>**

**</**inertial**>**

The Inertial tag has:

* **origin**: As you already know, this will move the inertia element based on the axis frame of the base\_link. For example, it's used for lowering the center of mass of the link element, if needed.
* **mass**: It's very important that the weight of this link element in Kg is coherent.  
  The reason is that this will condition the dynamics and the power of the actuators used afterwards to move the joints. But because it is not always possible to get the weight of each part of a robot, it at least has to be proportional to the total weight, which is much easier to get.
* **inertia**: As it states, this is the inertia matrix of the link. It will condition how the link behaves when spinning and moving around.  
  There could be a whole course on inertia calculations. But for you, as the user, you will always assume that the material the link is made of is homogeneous, and the matrix is diagonal. This means that you will only have values in the ixx, iyy, and izz. They will always be calculated based on primary geometric shapes.  
  This link has the basic inertia moments : https://en.wikipedia.org/wiki/List\_of\_moments\_of\_inertia

Although this way of calculating the Inertial tag data might seem poor, bear in mind that by approximating inertias to basic shapes, the results are quite close to reality. And if you really need a perfect simulation, you will have to dedicate time to obtaining the exact data from the real model.

Here you have a simple python script created to calculate the three basic inertia moments based on the mass and different variables.

In [ ]:



roscd my\_mira\_description

mkdir scripts

touch scripts**/**inertia\_calculator.py

chmod **+**x scripts**/**inertia\_calculator.py

inertia\_calculator.py

In [ ]:



*#!/usr/bin/env python*

**import** math

​

​

**class** InertialCalculator(object):

​

**def** \_\_init\_\_(self):

print "InertialCalculator Initialised..."

​

**def** start\_ask\_loop(self):

​

selection **=** "START"

​

**while** selection **!=** "Q":

print "#############################"

print "Select Geometry to Calculate:"

print "[1]Box width(w)\*depth(d)\*height(h)"

print "[2]Sphere radius(r)"

print "[3]Cylinder radius(r)\*height(h)"

print "[Q]END program"

selection **=** raw\_input(">>")

self.select\_action(selection)

​

print "InertialCaluclator Quit...Thank you"

​

**def** select\_action(self, selection):

**if** selection **==** "1":

mass **=** float(raw\_input("mass>>"))

width **=** float(raw\_input("width>>"))

depth **=** float(raw\_input("depth>>"))

height **=** float(raw\_input("height>>"))

self.calculate\_box\_inertia(m**=**mass, w**=**width, d**=**depth, h**=**height)

**elif** selection **==** "2":

mass **=** float(raw\_input("mass>>"))

radius **=** float(raw\_input("radius>>"))

self.calculate\_sphere\_inertia(m**=**mass, r**=**radius)

**elif** selection **==** "3":

mass **=** float(raw\_input("mass>>"))

radius **=** float(raw\_input("radius>>"))

height **=** float(raw\_input("height>>"))

self.calculate\_cylinder\_inertia(m**=**mass, r**=**radius, h**=**height)

**elif** selection **==** "Q":

print "Selected Quit"

**else**:

print "Usage: Select one of the give options"

​

​

**def** calculate\_box\_inertia(self, m, w, d, h):

Iw **=** (m**/**12.0)**\***(pow(d,2)**+**pow(h,2))

Id **=** (m **/** 12.0) **\*** (pow(w, 2) **+** pow(h, 2))

Ih **=** (m **/** 12.0) **\*** (pow(w, 2) **+** pow(d, 2))

print "BOX w\*d\*h, Iw = "**+**str(Iw)**+**",Id = "**+**str(Id)**+**",Ih = "**+**str(Ih)

​

**def** calculate\_sphere\_inertia(self, m, r):

I **=** (2**\***m**\***pow(r,2))**/**5.0

print "SPHERE Ix,y,z = "**+**str(I)

​

**def** calculate\_cylinder\_inertia(self, m, r, h):

Ix **=** (m**/**12.0)**\***(3**\***pow(r,2)**+**pow(h,2))

Iy **=** Ix

Iz **=** (m**\***pow(r,2))**/**2.0

print "Cylinder Ix,y = "**+**str(Ix)**+**",Iz = "**+**str(Iz)

​

**if** \_\_name\_\_ **==** "\_\_main\_\_":

inertial\_object **=** InertialCalculator()

inertial\_object.start\_ask\_loop()

END inertia\_calculator.py

This will speed up the process of obtaining the inertias of all the links you work on in this course.

To calculate the innertias of the **base\_link** cylinder execute the **inertial\_calculator.py** input the cylinders values and mass:

In [ ]:



rosrun my\_mira\_description inertia\_calculator.py

1.2 Gazebo Physical properties

Here you have an example of how to add physical properties to the base\_link in Gazebo:

Add this snippet inside the **mira\_simple\_collisions\_inertias.urdf**

In [ ]:



**<**gazebo reference**=**"base\_link"**>**

**<**kp**>**100000.0**</**kp**>**

**<**kd**>**100000.0**</**kd**>**

**<**mu1**>**10.0**</**mu1**>**

**<**mu2**>**10.0**</**mu2**>**

**<**material**>**Gazebo**/**Grey**</**material**>**

**</**gazebo**>**

The "gazebo" tag that refers to the link that is setting the physical properties for.

Here the main physical properties of the link are defined:

* mu1: The static friction coefficient. In simple terms, it would be how much friction there is until the object starts moving.
* mu2: The dynamic friction coefficient. In simple terms, it would be how much friction there is when the object is moving.

Here we have the same issue. These values should be calculated through friction tests with elements with the same mass as the links you are setting these values to. You should also bear in mind the materials they are made of, and so on. But in reality, it's setting them with the values that makes the robot behave correctly, not necessarily the real ones.

* kp: This coefficient sets the static contact stiffness. This determines if the link material is closer to marble (very rigid, bigger values) or more like rubber (soft material, lower values).
* kd: This coefficient sets the dynamic contact stiffness. This determines if the link material is closer to marble (very rigid, bigger values) or more like rubber (soft material, lower values). It's essentially how much it deforms over a long period of time, exerting its pressure.

And then we have the material, that is not more than the colour of the geometric visual shape. If its a dae, then it serves no purpose because the colours are enmbeded in the 3Dfile.

* material: Gazebo/Grey, in this case.

In Mira Robot's case, this is not as relevant as with Gurdy Robot, where you will work a bit longer on where locomotion depends on friction with the floor.

For further details, this Gazebo page gives you all the options for the Gazebo tag. Go to the section on **"Elements For Links"**: <http://gazebosim.org/tutorials/?tut=ros_urdf>

1.3 Collisions

One of the most crucial elements in being able to simulate a robot is how it interacts with the world around it.  
At the moment, your URDF Mira Robot would be a Ghost in a simulation. There is a visual representation, but it can't interact with anything. It will go through the floor and the objects.  
So, the first thing you have to do is add collisions to your URDF model. Here you have an example for the base\_link:

In [ ]:



**<**link name**=**"base\_link"**>**

**<**inertial**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**mass value**=**"0.18" **/>**

**<**inertia ixx**=**"0.0002835" ixy**=**"0.0" ixz**=**"0.0" iyy**=**"0.0002835" iyz**=**"0.0" izz**=**"0.000324"**/>**

**</**inertial**>**

**<**collision**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**geometry**>**

**<**cylinder radius**=**"0.06" length**=**"0.09"**/>**

**</**geometry**>**

**</**collision**>**

**<**visual**>**

**<**origin rpy**=**"0.0 0 0" xyz**=**"0 0 0"**/>**

**<**geometry**>**

**<**cylinder radius**=**"0.06" length**=**"0.09"**/>**

**</**geometry**>**

**</**visual**>**

**</**link**>**

As you can see, the only difference is that there is a new tag called **collision** that specifies the collision geometry. This is the shape used for calculating the physical contacts.  
As you might have guessed, you can also use 3D meshes here, like this:

In [ ]:



**<**link name**=**"base\_link"**>**

**<**inertial**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**mass value**=**"0.18" **/>**

**<**inertia ixx**=**"0.0002835" ixy**=**"0.0" ixz**=**"0.0" iyy**=**"0.0002835" iyz**=**"0.0" izz**=**"0.000324"**/>**

**</**inertial**>**

​

**<**collision**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**geometry**>**

**<**mesh filename**=**"package://my\_mira\_description/models/mira/meshes/mira\_body\_v3.dae"**/>**

**</**geometry**>**

**</**collision**>**

​

**<**visual**>**

**<**origin rpy**=**"0.0 0 0" xyz**=**"0 0 0"**/>**

**<**geometry**>**

**<**mesh filename**=**"package://my\_mira\_description/models/mira/meshes/mira\_body\_v3.dae"**/>**

**</**geometry**>**

**</**visual**>**

**</**link**>**

This is not advised, as the physics calculations are more intensive as the mesh gets more complex.  
That's why the collisions are normally basic geometric shapes, while the visuals are meshes.  
Another alternative if the geometry of the contact is crucial, is to use a lower poly version of the virtual mesh. That way, the shape is maintained, but less calculation power is needed.

In [ ]:



**<**link name**=**"base\_link"**>**

**<**inertial**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**mass value**=**"0.18" **/>**

**<**inertia ixx**=**"0.0002835" ixy**=**"0.0" ixz**=**"0.0" iyy**=**"0.0002835" iyz**=**"0.0" izz**=**"0.000324"**/>**

**</**inertial**>**

**<**collision**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**geometry**>**

**<**mesh filename**=**"package://mira\_description/models/mira/meshes/mira\_body\_v3\_lowpolygons.dae"**/>**

**</**geometry**>**

**</**collision**>**

**<**visual**>**

**<**origin rpy**=**"0.0 0 0" xyz**=**"0 0 0"**/>**

**<**geometry**>**

**<**mesh filename**=**"package://mira\_description/models/mira/meshes/mira\_body\_v3.dae"**/>**

**</**geometry**>**

**</**visual**>**

**</**link**>**

1.4 Visual Properties

This tag only is usefull for the RVIZ visualization which wont correspond with the gazebo Material given in the physical properties.

In [ ]:



**<**material name**=**"grey"**>**

**<**color rgba**=**"0.75 0.75 0.75 1"**/>**

**</**material**>**

​

**<**link name**=**"base\_link"**>**

**<**inertial**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**mass value**=**"0.18" **/>**

**<**inertia ixx**=**"0.0002835" ixy**=**"0.0" ixz**=**"0.0" iyy**=**"0.0002835" iyz**=**"0.0" izz**=**"0.000324"**/>**

**</**inertial**>**

**<**collision**>**

**<**origin xyz**=**"0 0 0" rpy**=**"0 0 0"**/>**

**<**geometry**>**

**<**cylinder radius**=**"0.06" length**=**"0.09"**/>**

**</**geometry**>**

**</**collision**>**

**<**visual**>**

**<**origin rpy**=**"0.0 0 0" xyz**=**"0 0 0"**/>**

**<**geometry**>**

**<**cylinder radius**=**"0.06" length**=**"0.09"**/>**

**</**geometry**>**

**<**material name**=**"grey"**/>**

**</**visual**>**

**</**link**>**

This is defined before as you can see. We define all the colours we might need in the visuals tags:

In [ ]:



**<**material name**=**"grey"**>**

**<**color rgba**=**"0.75 0.75 0.75 1"**/>**

**</**material**>**

​

...

​

**<**material name**=**"grey"**/>**

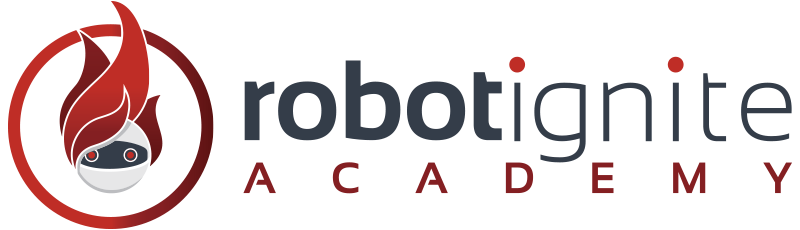
Exercise U2-0

* Add the necessary elements for the **roll\_M1\_link**
* Note that the roll\_M1\_link doesnt have to collide with anything because its an internal elemnt, so sollisions \*\*wont be needed.
* Note also that we **need** it to have inertias for the physiscs calculations, but it has to be negligible, therefore use a mas of **0.001** Kg.

END Exercise U2-0

Solution Exercise U2.0

Please try to do it by yourself, unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.



Follow this link to open the solutions notebook for Adapt URDF for Gazebo Simulator: [solutions\_Course\_urdfROS\_Unit\_2](https://i-0500135e5b3ce7d04.robotigniteacademy.com/d7e6f789-1484-4bd3-aac3-7d6fe70eb589/jupyter/notebooks/course_urdf_ROS/extra_files/solutions_Course_urdfROS_Unit_2.ipynb)

1.4 Spawn into Simulation

Now its time to see your creation inside the **simulated world!**.

Now, how do you spawn a URDF file-defined robot into the simulated world?  
Through these two launch files:

This first one's name is **spawn\_urdf.launch**:



roscd my\_mira\_description

touch launch/spawn\_urdf.launch

In [ ]:



**<?**xml version**=**"1.0" encoding**=**"UTF-8"**?>**

​

**<**launch**>**

​

**<**arg name**=**"x" default**=**"0.0" **/>**

**<**arg name**=**"y" default**=**"0.0" **/>**

**<**arg name**=**"z" default**=**"0.0" **/>**

​

**<**arg name**=**"urdf\_robot\_file" default**=**"" **/>**

**<**arg name**=**"robot\_name" default**=**"" **/>**

​

**<!--** This Version was created due to some errors seen **in** the V1 that crashed GAzebo **or** went too slow **in** spawn **-->**

**<!--** Load the URDF into the ROS Parameter Server **-->**

**<**param name**=**"robot\_description" command**=**"cat $(arg urdf\_robot\_file)" **/>**

​

**<!--** Run a python script to the send a service call to gazebo\_ros to spawn a URDF robot **-->**

**<**node name**=**"urdf\_spawner" pkg**=**"gazebo\_ros" type**=**"spawn\_model" respawn**=**"false" output**=**"screen"

args**=**"-urdf -x $(arg x) -y $(arg y) -z $(arg z) -model $(arg robot\_name) -param robot\_description"**/>**

**</**launch**>**

This first launch spawns the given URDF file into the given point in space, if a gazebo simulation is running.  
You can call this first launch through a second launch, passing the necessary arguments to it:

This second one's name is **spawn\_mira.launch**:



roscd my\_mira\_description

touch launch/spawn\_mira\_simple\_collisions\_inertias.launch

In [ ]:



**<?**xml version**=**"1.0" encoding**=**"UTF-8"**?>**

**<**launch**>**

**<**include file**=**"$(find my\_mira\_description)/launch/spawn\_urdf.launch"**>**

**<**arg name**=**"x" value**=**"0.0" **/>**

**<**arg name**=**"y" value**=**"0.0" **/>**

**<**arg name**=**"z" value**=**"0.2" **/>**

**<**arg name**=**"urdf\_robot\_file" value**=**"$(find my\_mira\_description)/urdf/mira\_simple\_collisions\_inertias.urdf" **/>**

**<**arg name**=**"robot\_name" value**=**"mira" **/>**

**</**include**>**

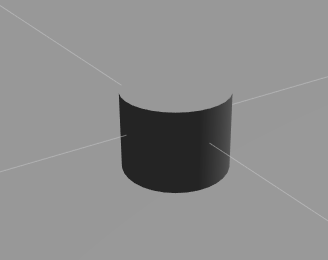
**</**launch**>**

And now execute the launch and you should get something like this:

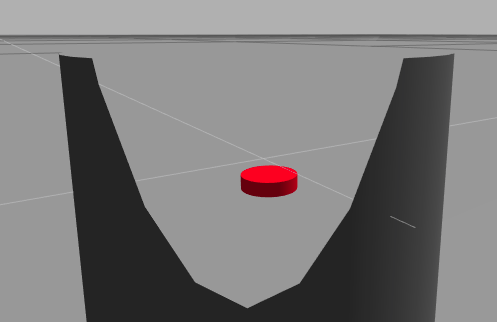
In [ ]:



roslaunch my\_mira\_description spawn\_mira\_simple\_collisions\_inertias.launch



You can see the **Red** **roll\_M1\_link** if you get closer enough to go through the baselink.



1.5 Delete a robot model from Gazebo:

Of course we want to do more tests, and spawn the complete **mira robot**. So How do I delete the previous one I spawned, because it failed or because I don't want it?

* First : Check you have the model in the world. For that execute the following command:

|  |
| --- |
| Execute in WebShell #1 |

In [ ]:



rosservice call **/**gazebo**/**get\_world\_properties "{}"

In [ ]:



user urdf $ rosservice call **/**gazebo**/**get\_world\_properties "{}"

sim\_time: 17173.946

model\_names: ['ground\_plane', 'mira']

rendering\_enabled: **True**

success: **True**

status\_message: GetWorldProperties: got properties

* Remove it from the world: In case you have the model inside, get the name given in gazebo, which is the one that appears in **model\_names**, in this case **mira**. We remove it because there can't be tow robot models with the same name in Gazebo.

|  |
| --- |
| Execute in WebShell #1 |

In [ ]:



rosservice call **/**gazebo**/**delete\_model "model\_name: 'mira'"

Or you can make your life easier by creating a client that removes the model:



roscd my\_mira\_description

touch scripts/delete\_mira.py

chmod +x scripts/delete\_mira.py

In [ ]:



*#! /usr/bin/env python*

​

**import** rospy

**from** gazebo\_msgs.srv **import** DeleteModel, DeleteModelRequest *# Import the service message used by the service /gazebo/delete\_model*

**import** sys

​

rospy.init\_node('remove\_model\_service\_client') *# Initialise a ROS node with the name service\_client*

print "Waiting for Service /gazebo/delete\_model"

rospy.wait\_for\_service('/gazebo/delete\_model') *# Wait for the service client /gazebo/delete\_model to be running*

delete\_model\_service **=** rospy.ServiceProxy('/gazebo/delete\_model', DeleteModel) *# Create the connection to the service*

kk **=** DeleteModelRequest() *# Create an object of type DeleteModelRequest*

kk.model\_name **=** "mira" *# Fill the variable model\_name of this object with the desired value*

print "Deleting model ="**+**str(kk)

status\_message **=** delete\_model\_service(kk) *# Send through the connection the name of the object to be deleted by the service*

print status\_message

You can execute it like so:

|  |
| --- |
| Execute in WebShell #1 |

In [ ]:



rosrun my\_mira\_description delete\_mira.py

It's possible that **none of these systems will work** because there was an error in the spawn or something similar. If this happens, you can remove the model just by **right-clicking on it and selecting Delete**.

But if you can't */gazebo/delete\_model*, the gazebo has probably died. In that case, you should reboot your gazebo, or in RobotIgniteAcademy, just return to Unit 0 and come back to this unit again. That will restart everything.

Exercise U2-1

* Create a new launch called **spawn\_mira\_geometric\_collisions\_inertias.launch** that spawns the entire **mira\_geometric\_collisions\_inertias.urdf** URDF file.
* For that, you will have add all of the **collisions,innertias, visual properties and gazebo properties** to the links that need to have them.
* **Collisions**: roll\_M1\_link, pitch\_M2\_link, yaw\_M3\_link, camera\_link, and right\_eye\_link/left\_eye\_link **don't** need the **collisions**.  
  Why?  
  The eyes and camera don't because they are too small or don't protrude to have any significant effect on collisions. As for the roll, pitch, and yaw, they don't have to have collisions because they are virtual links; they don't exist in reality and it would just cause problems that the real robot doesn't have.
* **Innertias**:Add all of the inertial tags for all the links. Use the python program to calculate the inertias of each link.All of the inertia calculations will be based on the geometry of the collision geometries.

Here you have the weight of all the links:

* base\_link: mass value="0.18"
* roll\_M1\_link: value="0.001"
* pitch\_M2\_link: value="0.001"
* yaw\_M3\_link: value="0.001"
* head\_link: mass value="0.02"
* left\_eye\_link: mass value="0.001"
* right\_eye\_link: mass value="0.001"
* camera\_link: mass value="0.001"

The links with mass set to **0.001** are just symbolic mass that has no real effect on the dynamics of the robot, but allow it to be represented in Gazebo.

* Gazebo Physical properties:

In [ ]:



**<**gazebo reference**=**"LINK\_NAME"**>**

**<**kp**>**100000.0**</**kp**>**

**<**kd**>**100000.0**</**kd**>**

**<**mu1**>**10.0**</**mu1**>**

**<**mu2**>**10.0**</**mu2**>**

**<**material**>**Gazebo**/**COLOUR**</**material**>**

**</**gazebo**>**

Choose the colours you like and the other properties set them the same as the base\_link to avoid any problems. Actually the only link that touches anything is the base link, the other don't touch the ground or interact in anyway with anything.

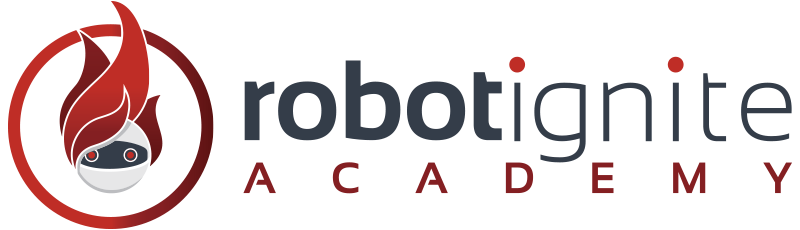
Here you have a list of all the **Gazebo Materials** available: <https://bitbucket.org/osrf/gazebo/src/default/media/materials/scripts/gazebo.material>

* Visual Properties: Choose the colours that will apear in RVIZ. I recommend choosoing the same as the materials defined in the **Gazbeo Physical properties material**.

Finally, spawn your URDF and see what happens.

END Exercise U2-1

Please try to do it by yourself, unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.



Follow this link to open the solutions notebook for Adapt URDF for Gazebo Simulator: [solutions\_Course\_urdfROS\_Unit\_2](https://i-0500135e5b3ce7d04.robotigniteacademy.com/d7e6f789-1484-4bd3-aac3-7d6fe70eb589/jupyter/notebooks/course_urdf_ROS/extra_files/solutions_Course_urdfROS_Unit_2.ipynb)

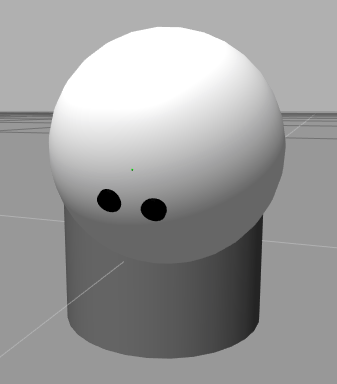
When executing the following command you should get something like this in Gazebo Simulator window:

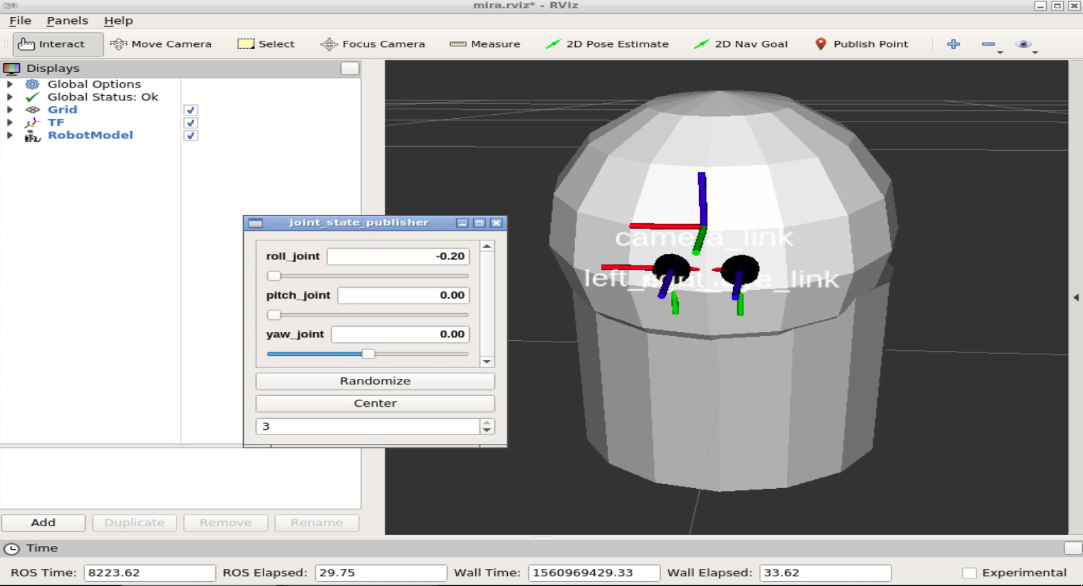
|  |
| --- |
| Execute in WebShell #1 |

In [ ]:



roslaunch my\_mira\_description spawn\_mira\_geometric\_collisions\_inertias.launch





Check also the URDF visualization in RVIZ

|  |
| --- |
| Execute in WebShell #1 |



roslaunch my\_mira\_description urdf\_visualize.launch model:='$(find my\_mira\_description)/urdf/mira\_geometric\_collisions\_inertias.urdf'

Note that even if you move the joints sliders , the simulation doesnt care. This sis because we havent added the controllers yet. Ths URDF is juts a representation of the in RVIZ of the file, nothing to do with the simulation state.

2. Add 3D meshes:

Now to add the meshes instead of the **geometric shapes** you will have to edit only the **visual** tags. The collisions will be used the geometric shapes. This is a common practice because it reduces significantly the physics calculations vs using meshes.

**THIS is VERY IMPORTANT**:

* Gazebo needs the 3D models to be inside a certain folder to find them. Depends on the system, but RobotIgniteAcademy, the package with the meshes has to be inside:



/usr/share/gazebo/models/

So you have to always clean and copy your files inside that folder if the meshes have been changed:

|  |
| --- |
| Execute in WebShell #1 |



sudo rm -rf /usr/share/gazebo/models/my\_mira\_description

sudo cp -r /home/user/catkin\_ws/src/my\_mira\_description /usr/share/gazebo/models/

**ALWAYS WHEN YOU SIGN IN AGAIN IN ROBOT IGNITE ACADEMY, you will have to copy this folder AGAIN**. (this is a limitation of our learning platform, not necessary on any other ROS system)

Exercise U2-2

Create a new spawn launch file and a urdf file called:

* mira\_no\_controllers.urdf
* spawn\_mira\_no\_controllers.launch

Add the meshes for:



Base\_link: <mesh filename="package://my\_mira\_description/models/mira/meshes/mira\_body\_v3.dae"/>

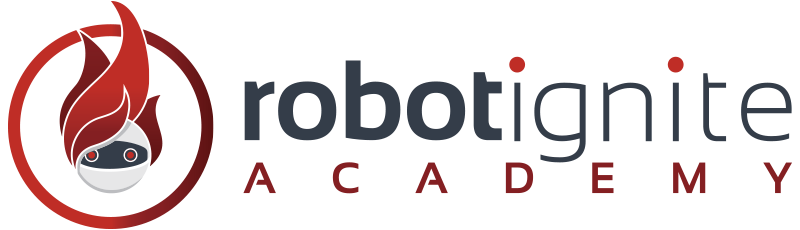
Head\_link: <mesh filename="package://my\_mira\_description/models/mira/meshes/mira\_head\_v5.dae"/>

Eyes: <mesh filename="package://my\_mira\_description/models/mira/meshes/mira\_eye\_v4.dae"/>

END Exercise U2-2

Solution Exercise U2.2

Please try to do it by yourself, unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.



Follow this link to open the solutions notebook for Adapt URDF for Gazebo Simulator: [solutions\_Course\_urdfROS\_Unit\_2](https://i-0500135e5b3ce7d04.robotigniteacademy.com/d7e6f789-1484-4bd3-aac3-7d6fe70eb589/jupyter/notebooks/course_urdf_ROS/extra_files/solutions_Course_urdfROS_Unit_2.ipynb)

Remember to delete the previous **mira** you spawned:



rosrun my\_mira\_description delete\_mira.py

or



rosservice call /gazebo/delete\_model "model\_name: 'mira'"

Or if everything fails Change to Unit0 and back again.

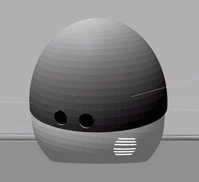
When executing the following command you should get something like this in Gazebo Simulator window:

In [ ]:



roslaunch my\_mira\_description spawn\_mira\_no\_controllers.launch

When spawning, you should get a perfectly rendered Mira Robot, though it will be limp. Because we haven't set up any friction in the joints, it will continue to move with the residual spawn movement forever.



Exercise U2-2-EXTRA

Add your own Custom mesh for the head. We provide you here with one, but you can upload any mesh you want through the IDE.

* Here we provide you with two files:

Cromulon (Rick & Morty) by unbrandedlx is licensed under the Creative Commons - Attribution license. Extracted from here: <https://www.thingiverse.com/thing:1587203>

This was extracted from this link: <https://www.thingiverse.com/thing:2640506> Created by: Low poly skull by Agesianax is licensed under the Creative Commons - Attribution license. It was scaled and coloured for your convenience.

[cromulon.dae](https://i-0500135e5b3ce7d04.robotigniteacademy.com/d7e6f789-1484-4bd3-aac3-7d6fe70eb589/jupyter/notebooks/course_urdf_ROS/extra_files/cromulon.dae)

[skull\_v2.dae](https://i-0500135e5b3ce7d04.robotigniteacademy.com/d7e6f789-1484-4bd3-aac3-7d6fe70eb589/jupyter/notebooks/course_urdf_ROS/extra_files/skull_v2.dae)

* To upload just **righ click on the meshes folder** inside the **my\_mira\_description** and select **upload**.
* To avoid strange things, download the original head mesh and compare its size with the one you want to replace with. You can use blender for that.
* **Highly recommend** to use **Low Poly Meshes**. It will make loading much faster and smooth.

Remember that to update the meshes insid ethe system you have to execute the following command:

|  |
| --- |
| Execute in WebShell #1 |



rm -rf /usr/share/gazebo/models/my\_mira\_description

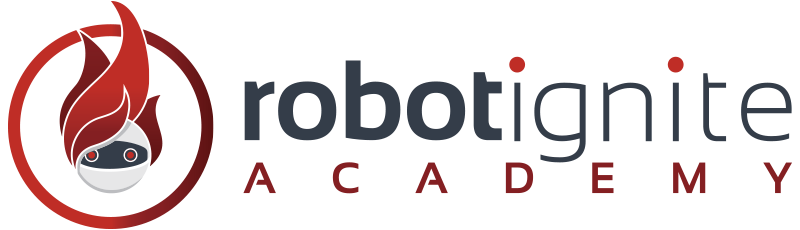
cp -r /home/user/catkin\_ws/src/my\_mira\_description /usr/share/gazebo/models/

**ALWAYS WHEN YOU SIGN IN AGAIN IN RIOBOTIGNTE ACADEMY, you will have to copy this folder AGAIN**.

END Exercise U2-2-EXTRA

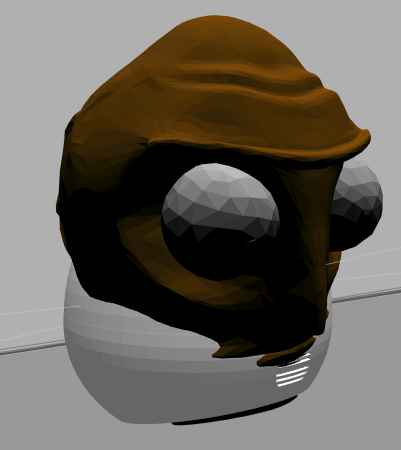
Solution Exercise U2.2

Please try to do it by yourself, unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.



Follow this link to open the solutions notebook for Adapt URDF for Gazebo Simulator: [solutions\_Course\_urdfROS\_Unit\_2](https://i-0500135e5b3ce7d04.robotigniteacademy.com/d7e6f789-1484-4bd3-aac3-7d6fe70eb589/jupyter/notebooks/course_urdf_ROS/extra_files/solutions_Course_urdfROS_Unit_2.ipynb)

If you used the provided **cromulon.dae** you should see something like this:



If you used the provided **skull\_v2.dae** you should see something like this:



3. Add controllers:

As you have experienced in the previous exercises, Mira's joints don't have any effort applied and, therefore, they fall due to their weight. And we can't move them through the joints publisher gui, as you should know, because that's only a topic that indicates the sensor readings of the encoders inside the actuators (virtual or physical).  
So, you have to add real actuators to move Mira's joints. These are the steps for a single joint, in this case the "roll\_joint":

Create the URDF and the launch file accordingly:

|  |
| --- |
| Execute in WebShell #1 |



roscd my\_mira\_description

touch urdf/mira\_with\_onecontroller.urdf



roscd my\_mira\_description

touch launch/spawn\_mira\_with\_onecontroller.launch

spawn\_mira\_with\_onecontroller.launch



<?xml version="1.0" encoding="UTF-8"?>

<launch>

  <include file="$(find my\_mira\_description)/launch/spawn\_urdf.launch">

      <arg name="x" value="0.0" />

      <arg name="y" value="0.0" />

      <arg name="z" value="0.2" />

      <arg

      name="urdf\_robot\_file"

      value="$(find my\_mira\_description)/urdf/mira\_with\_onecontroller.urdf"

      />

      <arg name="robot\_name" value="mira" />

  </include>

</launch>

END spawn\_mira\_with\_onecontroller.launch

Step 1: Add a transmission tag linked to the joint you want to move:

In [ ]:



**<**gazebo**>**

**<**plugin name**=**"gazebo\_ros\_control" filename**=**"libgazebo\_ros\_control.so"**>**

**<**robotNamespace**>/**mira**</**robotNamespace**>**

**</**plugin**>**

**</**gazebo**>**

**<**joint name**=**"roll\_joint" type**=**"revolute"**>**

**<**parent link**=**"base\_link"**/>**

**<**child link**=**"roll\_M1\_link"**/>**

**<**origin xyz**=**"0.0023 0 -0.0005" rpy**=**"0 0 0"**/>**

**<**limit lower**=**"-0.2" upper**=**"0.2" effort**=**"0.1" velocity**=**"0.005"**/>**

**<**axis xyz**=**"1 0 0"**/>**

**</**joint**>**

​

**<**transmission name**=**"tran1"**>**

**<**type**>**transmission\_interface**/**SimpleTransmission**</**type**>**

**<**joint name**=**"roll\_joint"**>**

**<**hardwareInterface**>**EffortJointInterface**</**hardwareInterface**>**

**</**joint**>**

**<**actuator name**=**"motor1"**>**

**<**hardwareInterface**>**EffortJointInterface**</**hardwareInterface**>**

**<**mechanicalReduction**>**1**</**mechanicalReduction**>**

**</**actuator**>**

**</**transmission**>**

Tags explained:

First, let's have a look at the joint limits:

In [ ]:



**<**limit lower**=**"-0.2" upper**=**"0.2" effort**=**"0.1" velocity**=**"0.005"**/>**

As you can see, there is a maximum effort of 0.1, and a maximum velocity of 0.005. This is how fast the joint will be able to go and how much effort they will allow the actuator to apply. This is like selecting the kind of servo that you want. Select a servo that can move the weight of that link and the ones connected to it down the tree, but don't select a huge servo because it will just break your robot, or simply make it fly due to the inertia forces. Bear this in mind because it's very easy to get carried away in virual design, but then the robot behaves in the most bizarre way.

In [ ]:



**<**gazebo**>**

**<**plugin name**=**"gazebo\_ros\_control" filename**=**"libgazebo\_ros\_control.so"**>**

**<**robotNamespace**>/**mira**</**robotNamespace**>**

**</**plugin**>**

**</**gazebo**>**

This activates the gazebo control plugin for the Mira namespace. This namespace will be set in the main spawn launch file.

In [ ]:



**<**transmission name**=**"tran1"**>**

Here you define the name of the transmission, which will have to be unique through the URDF file.

In [ ]:



**<**type**>**transmission\_interface**/**SimpleTransmission**</**type**>**

The type of transmission that, for the moment, is the only one implemented: "transmission\_interface/SimpleTransmission"

In [ ]:



**<**joint name**=**"roll\_joint"**>**

**<**hardwareInterface**>**EffortJointInterface**</**hardwareInterface**>**

**</**joint**>**

Here you link the joint to the transmission and select the hardware interface, which also only has an effort interface working.

In [ ]:



**<**actuator name**=**"motor1"**>**

**<**hardwareInterface**>**EffortJointInterface**</**hardwareInterface**>**

**<**mechanicalReduction**>**1**</**mechanicalReduction**>**

**</**actuator**>**

Here you set a unique name for the actuator and again select the hardware interface. As for the reduction, it's self-explanatory.

All these values are almost always the same, so don't worry too much because, in most cases, putting these values is more than enough.

Step 2: Create a Control config yaml to set the PID for each transmission/motor

Create a file called **mira\_control.yaml** inside the **config** folder of your package, **my\_mira\_description**.

|  |
| --- |
| Execute in WebShell #1 |



roscd my\_mira\_description

touch config/mira\_onecontroller.yaml

In [ ]:



mira:

*# Publish all joint states -----------------------------------*

joint\_state\_controller:

type: joint\_state\_controller**/**JointStateController

publish\_rate: 50

​

*# Position Controllers ---------------------------------------*

roll\_joint\_position\_controller:

type: effort\_controllers**/**JointPositionController

joint: roll\_joint

pid: {p: 1.0, i: 1.0, d: 0.0}

*# To add more just add them here as the first one*

Here you will set the publish rate of the joints position, that is now controlled by the joint\_state\_controller.  
And you set the joint controllers:

* The name of the controller is recommended to have the exact same joint name, proceeded by "\_position\_controller"
* Then, you select the type, which we recomment as the JointPositionController because we want to control it through the position of the joints.
* The JointName: This is linking the joint to the controller.
* pid: Sets the PID values. This is absolutely vital to making the actuators reach the desired position without oscillations or going too slow.

If you are uncertain of how PID work, although it's a deep subject, this tutorial is magnificent for giving a practical idea of how to diagnose PID settings and how to use them: <https://www.flitetest.com/articles/p-i-and-sometimes-d-gains-in-a-nutshell>

Also, you will set a sensible PID at the start, so you can then calibrate it during the execution of movements through the DynamicReconfigure <http://wiki.ros.org/dynamic_reconfigure> and its GUI, rqt\_reconfigure <http://wiki.ros.org/rqt_reconfigure>.  
Through rqt\_reconfigure, you can change any parameter that is made accesible at runtime. This way, you just have to move the robot and change the values until you see that it behaves in the correct way.

To use it, you will just have to execute the following command when you spawn Mira with its controllers:

|  |
| --- |
| Execute in WebShell #1 |

In [ ]:



rosrun rqt\_reconfigure rqt\_reconfigure

Step 3: Create the control launch file that gets everything running

Now, it's time to turn on the switch for the controllers and the actuators. For this, you have to create the following launch file, called **mira\_control.launch**:

|  |
| --- |
| Execute in WebShell #1 |



roscd my\_mira\_description

touch launch/mira\_onejoint\_control.launch

mira\_onejoint\_control.launch

In [ ]:



**<**launch**>**

**<!--** Load joint controller configurations **from** YAML file to parameter server **-->**

**<**rosparam file**=**"$(find my\_mira\_description)/config/mira\_onecontroller.yaml" command**=**"load"**/>**

​

**<!--** load the controllers **-->**

​

**<**node name**=**"controller\_spawner" pkg**=**"controller\_manager" type**=**"spawner" respawn**=**"false"

output**=**"screen" ns**=**"/mira" args**=**"roll\_joint\_position\_controller joint\_state\_controller --shutdown-timeout 3"**/>**

​

**<!--** convert joint states to TF transforms **for** rviz, etc **-->**

**<**node name**=**"robot\_state\_publisher" pkg**=**"robot\_state\_publisher" type**=**"robot\_state\_publisher"

respawn**=**"false" output**=**"screen"**>**

**<**remap **from=**"/joint\_states" to**=**"/mira/joint\_states" **/>**

**</**node**>**

​

**</**launch**>**

END mira\_onejoint\_control.launch

**Be careful with name spaces; they're a source of errors.**

If you don't understand what's going on in this launch, please refer to our course on **TF\_ROS** for a more detailed explanation.  
Esentially here we are loading the **yaml file** and launching the controllers with the robot state publisher for the TF publication.

In [ ]:



**<**node name**=**"controller\_spawner" pkg**=**"controller\_manager" type**=**"spawner" respawn**=**"false"

output**=**"screen" ns**=**"/mira" args**=**"roll\_joint\_position\_controller joint\_state\_controller"**/>**

This part is the only one that is really relevant. Here you state that all the controllers that you defined in the yaml are active. In this case, you are only turning on the "roll\_joint\_position\_controller" and the "joint\_state\_controller." When adding more controllers, just add them here.

Step 4: Spawn and launch the controllers in their own name space:

The last step is creating a launch that spawns the Mira Robot in its own name space and starts the controllers.  
Spawning in its own name space allows it to be spawned alongside more robots using the urdf\_spawner and other systems.

You have to define the **"spawn\_with\_controllers\_mira.launch"** that will do the real work.

|  |
| --- |
| Execute in WebShell #1 |



roscd my\_mira\_description

touch launch/start\_mira\_withonecontroller.launch

start\_mira\_withonecontroller.launch

In [ ]:



**<?**xml version**=**"1.0" encoding**=**"UTF-8"**?>**

**<**launch**>**

**<**include file**=**"$(find my\_mira\_description)/launch/spawn\_mira\_with\_onecontroller.launch"**/>**

**<**include file**=**"$(find my\_mira\_description)/launch/mira\_onejoint\_control.launch"**/>**

**</**launch**>**

END start\_mira\_withonecontroller.launch

These two files **(spawn\_mira\_with\_onecontroller.launch and mira\_onejoint\_control.launch)** you already have, so now you just have to launch your **"start\_mira\_withonecontroller."** It should spawn your Mira URDF model, with the roll joint as fully controllable.

Remember to delete the previous **mira** you spawned:



rosrun my\_mira\_description delete\_mira.py

or



rosservice call /gazebo/delete\_model "model\_name: 'mira'"

Or if everything fails Change to Unit0 and back again.

Remember that to update the meshes insid ethe system you have to execute the following command:

|  |
| --- |
| Execute in WebShell #1 |



rm -rf /usr/share/gazebo/models/my\_mira\_description

cp -r /home/user/catkin\_ws/src/my\_mira\_description /usr/share/gazebo/models/

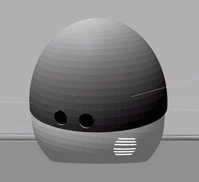
**ALWAYS WHEN YOU SIGN IN AGAIN IN RIOBOTIGNTE ACADEMY, you will have to copy this folder AGAIN**.

|  |
| --- |
| Execute in WebShell #1 |



roslaunch my\_mira\_description start\_mira\_withonecontroller.launch

You should get apparently the exact same robot as before:



But now you have the power of moving the **roll\_joint**, so lets have a look.

To test that the controllers are working, you can:

* Publish directly in the controllers topics:
* Publish through the rqt\_gui Topic Publishing.

To publish directly, pick a controller command topic; in this case, there must be only one; and publish a position inside the limits of that joint (in this case lower="-0.2" upper="0.2"):

|  |
| --- |
| Execute in WebShell #2 |

In [ ]:



rostopic pub **/**mira**/**roll\_joint\_position\_controller**/**command std\_msgs**/**Float64 "data: 0.2"

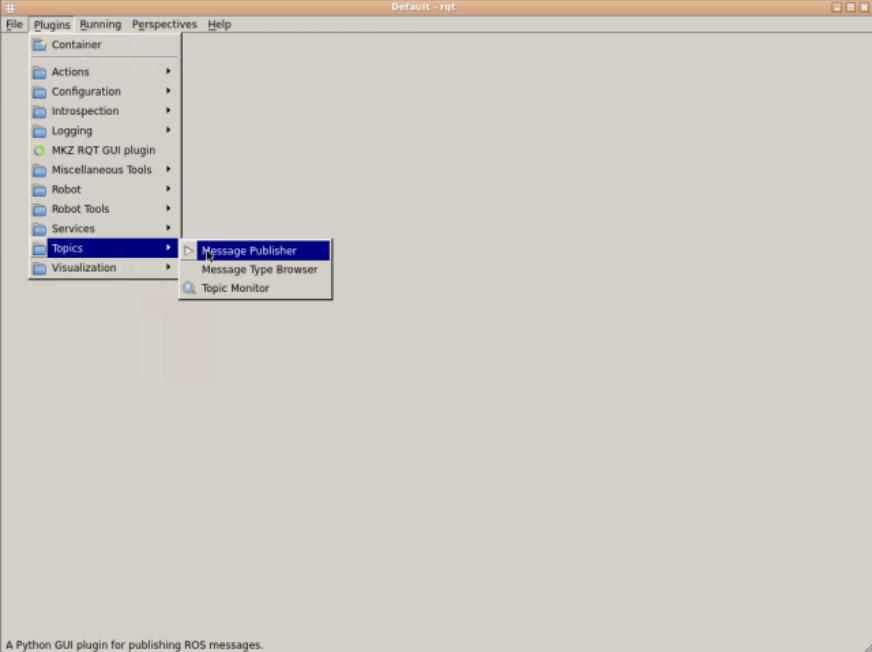
Mira Robot should move its head in the roll direction.

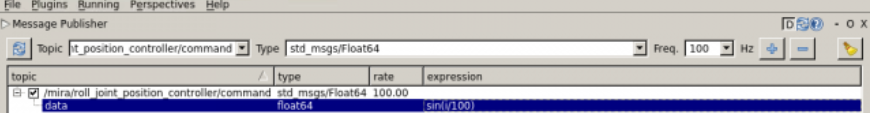
To publish through the gui, just execute:

|  |
| --- |
| Execute in WebShell #2 |

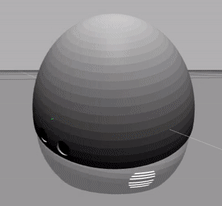


rosrun rqt\_gui rqt\_gui





Introducing a sinus based on the time (i) and divided by the frequency, Mira should move its head side to side. So, because the yaw is not yet controlled, it continues to move freely.



Exercise U2-3

Add two more transmissions; one for the pitch\_joint and the other for the yaw\_joint.  
When you have it working by commands, create a python script that moves Mira.  
Try to make a simple set of movements and try to mimic how it moves in reality.  
Remember that if you find that the movements are too fast or oscillate too much, you can always use the rqt\_reconfigure to tune the PID values.

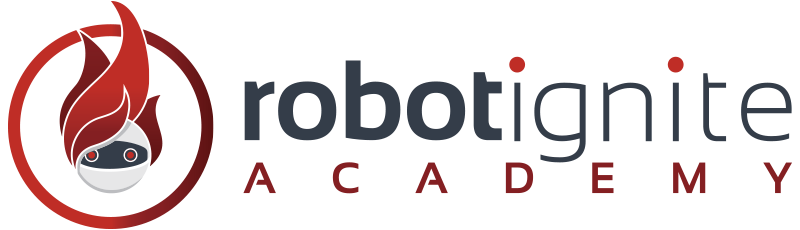
END Exercise U2-3

Warning

If you have to remove the robot to restart again, due to a gazebo issue, you have to delete the model by RIGHT-clicking and hitting Delete. Don't forget to also close the control launch.

Solution Exercise U2.3

Please try to do it by yourself, unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.



Follow this link to open the solutions notebook for Adapt URDF for Gazebo Simulator: [solutions\_Course\_urdfROS\_Unit\_2](https://i-0500135e5b3ce7d04.robotigniteacademy.com/d7e6f789-1484-4bd3-aac3-7d6fe70eb589/jupyter/notebooks/course_urdf_ROS/extra_files/solutions_Course_urdfROS_Unit_2.ipynb)

Remember to delete the previous **mira** you spawned:



rosrun my\_mira\_description delete\_mira.py

or



rosservice call /gazebo/delete\_model "model\_name: 'mira'"

Or if everything fails Change to Unit0 and back again.

Remember that to update the meshes insid ethe system you have to execute the following command:

|  |
| --- |
| Execute in WebShell #1 |



rm -rf /usr/share/gazebo/models/my\_mira\_description

cp -r /home/user/catkin\_ws/src/my\_mira\_description /usr/share/gazebo/models/

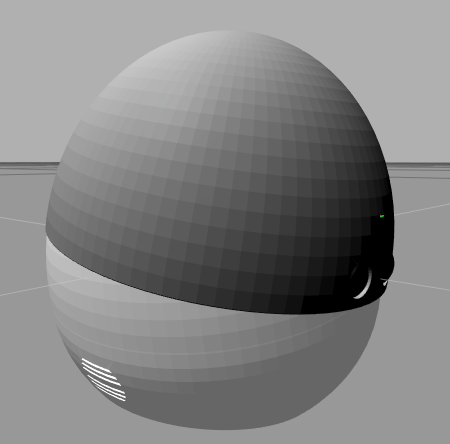
**ALWAYS WHEN YOU SIGN IN AGAIN IN RIOBOTIGNTE ACADEMY, you will have to copy this folder AGAIN**.

|  |
| --- |
| Execute in WebShell #1 |



roslaunch my\_mira\_description start\_mira\_with\_controllers.launch

You should get a static Mira like this:



3.1: Creating a python class that moves Mira

Here you have a tiny example of a class created to move Mira thorugh the **commands** topics.

|  |
| --- |
| Execute in WebShell #1 |



roscd my\_mira\_description

touch scripts/my\_mira\_demo.py

chmod +x scripts/my\_mira\_demo.py

my\_mira\_demo.py

In [ ]:



*#!/usr/bin/env python*

**import** time

**import** rospy

**from** math **import** pi, sin, cos, acos

**import** random

**from** std\_msgs.msg **import** Float64

**from** sensor\_msgs.msg **import** JointState

"""

Topics To Write on:

type: std\_msgs/Float64

/mira/pitch\_joint\_position\_controller/command

/mira/roll\_joint\_position\_controller/command

/mira/yaw\_joint\_position\_controller/command

"""

​

**class** MiraJointMover(object):

​

**def** \_\_init\_\_(self):

rospy.init\_node('jointmover\_demo', anonymous**=True**)

rospy.loginfo("Mira JointMover Initialising...")

self.pub\_mira\_roll\_joint\_position **=** rospy.Publisher('/mira/roll\_joint\_position\_controller/command',

Float64,

queue\_size**=**1)

self.pub\_mira\_pitch\_joint\_position **=** rospy.Publisher('/mira/pitch\_joint\_position\_controller/command',

Float64,

queue\_size**=**1)

self.pub\_mira\_yaw\_joint\_position **=** rospy.Publisher('/mira/yaw\_joint\_position\_controller/command',

Float64,

queue\_size**=**1)

joint\_states\_topic\_name **=** "/mira/joint\_states"

rospy.Subscriber(joint\_states\_topic\_name, JointState, self.mira\_joints\_callback)

mira\_joints\_data **=** **None**

**while** mira\_joints\_data **is** **None**:

**try**:

mira\_joints\_data **=** rospy.wait\_for\_message(joint\_states\_topic\_name, JointState, timeout**=**5)

**except**:

rospy.logwarn("Time out " **+** str(joint\_states\_topic\_name))

**pass**

​

self.mira\_joint\_dictionary **=** dict(zip(mira\_joints\_data.name, mira\_joints\_data.position))

​

**def** move\_mira\_all\_joints(self, roll, pitch, yaw):

angle\_roll **=** Float64()

angle\_roll.data **=** roll

angle\_pitch **=** Float64()

angle\_pitch.data **=** pitch

angle\_yaw **=** Float64()

angle\_yaw.data **=** yaw

self.pub\_mira\_roll\_joint\_position.publish(angle\_roll)

self.pub\_mira\_pitch\_joint\_position.publish(angle\_pitch)

self.pub\_mira\_yaw\_joint\_position.publish(angle\_yaw)

​

**def** move\_mira\_roll\_joint(self, position):

"""

limits radians : lower="-0.2" upper="0.2"

:param position:

:return:

"""

angle **=** Float64()

angle.data **=** position

self.pub\_mira\_roll\_joint\_position.publish(angle)

​

**def** move\_mira\_pitch\_joint(self, position):

"""

limits radians : lower="0" upper="0.44"

:param position:

:return:

"""

angle **=** Float64()

angle.data **=** position

self.pub\_mira\_pitch\_joint\_position.publish(angle)

​

**def** move\_mira\_yaw\_joint(self, position):

"""

Limits : continuous, no limits

:param position:

:return:

"""

angle **=** Float64()

angle.data **=** position

self.pub\_mira\_yaw\_joint\_position.publish(angle)

​

**def** mira\_joints\_callback(self, msg):

"""

sensor\_msgs/JointState

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

string[] name

float64[] position

float64[] velocity

float64[] effort

​

:param msg:

:return:

"""

self.mira\_joint\_dictionary **=** dict(zip(msg.name, msg.position))

​

**def** mira\_check\_joint\_value(self, joint\_name, value, error**=**0.1):

"""

Check the joint by name 'pitch\_joint', 'roll\_joint', 'yaw\_joint' is near the value given

:param value:

:return:

"""

similar **=** self.mira\_joint\_dictionary.get(joint\_name) **>=** (value **-** error ) **and** self.mira\_joint\_dictionary.get(joint\_name) **<=** (value **+** error )

​

**return** similar

​

**def** convert\_angle\_to\_unitary(self, angle):

"""

Removes complete revolutions from angle and converts to positive equivalent

if the angle is negative

:param angle: Has to be in radians

:return:

"""

*# Convert to angle between [0,360)*

complete\_rev **=** 2 **\*** pi

mod\_angle **=** int(angle **/** complete\_rev)

clean\_angle **=** angle **-** mod\_angle **\*** complete\_rev

*# Convert Negative angles to their corresponding positive values*

**if** clean\_angle **<** 0:

clean\_angle **+=** 2 **\*** pi

​

**return** clean\_angle

​

**def** assertAlmostEqualAngles(self, x, y,):

c2 **=** (sin(x) **-** sin(y)) **\*\*** 2 **+** (cos(x) **-** cos(y)) **\*\*** 2

angle\_diff **=** acos((2.0 **-** c2) **/** 2.0)

**return** angle\_diff

​

**def** mira\_check\_continuous\_joint\_value(self, joint\_name, value, error**=**0.1):

"""

Check the joint by name 'pitch\_joint', 'roll\_joint', 'yaw\_joint' is near the value given

We have to convert the joint values removing whole revolutions and converting negative versions

of the same angle

:param value:

:return:

"""

joint\_reading **=** self.mira\_joint\_dictionary.get(joint\_name)

clean\_joint\_reading **=** self.convert\_angle\_to\_unitary(angle**=**joint\_reading)

clean\_value **=** self.convert\_angle\_to\_unitary(angle**=**value)

​

dif\_angles **=** self.assertAlmostEqualAngles(clean\_joint\_reading, clean\_value)

similar **=** dif\_angles **<=** error

​

**return** similar

​

**def** mira\_movement\_sayno(self):

"""

Make Mira say no with the head

:return:

"""

check\_rate **=** 5.0

position **=** 0.7

​

​

rate **=** rospy.Rate(check\_rate)

**for** repetition **in** range(2):

similar **=** **False**

**while** **not** similar:

self.move\_mira\_yaw\_joint(position**=**position)

*# WARNING: THE COMMAND HAS TO BE PUBLISHED UNTIL THE GOAL IS REACHED*

*# This is because, when publishing a topic, the first time doesn't always work.*

similar **=** self.mira\_check\_continuous\_joint\_value(joint\_name**=**"yaw\_joint", value**=**position)

​

rate.sleep()

position **\*=** **-**1

​

​

​

**def** mira\_movement\_look(self, roll, pitch, yaw):

"""

Make Mira look down

:return:

"""

check\_rate **=** 5.0

position\_roll **=** roll

position\_pitch **=** pitch

position\_yaw **=** yaw

​

similar\_roll **=** **False**

similar\_pitch **=** **False**

similar\_yaw **=** **False**

rate **=** rospy.Rate(check\_rate)

**while** **not** (similar\_roll **and** similar\_pitch **and** similar\_yaw):

self.move\_mira\_all\_joints(position\_roll, position\_pitch, position\_yaw)

similar\_roll **=** self.mira\_check\_continuous\_joint\_value(joint\_name**=**"roll\_joint", value**=**position\_roll)

similar\_pitch **=** self.mira\_check\_continuous\_joint\_value(joint\_name**=**"pitch\_joint", value**=**position\_pitch)

similar\_yaw **=** self.mira\_check\_continuous\_joint\_value(joint\_name**=**"yaw\_joint", value**=**position\_yaw)

rate.sleep()

​

​

**def** mira\_lookup(self):

​

self.mira\_movement\_look(roll**=**0.0,pitch**=**0.3,yaw**=**1.57)

​

**def** mira\_lookdown(self):

​

self.mira\_movement\_look(roll**=**0.0,pitch**=**0.0,yaw**=**1.57)

​

**def** mira\_movement\_laugh(self, set\_rpy**=False**, roll**=**0.05, pitch**=**0.0, yaw**=**1.57, n\_giggle**=**15):

"""

Giggle in a given pitch yaw configuration

:return:

"""

position\_roll **=** roll

position\_pitch **=** pitch

position\_yaw **=** yaw

**for** repetitions **in** range(n\_giggle):

**if** set\_rpy:

self.move\_mira\_all\_joints(position\_roll, position\_pitch, position\_yaw)

**else**:

self.move\_mira\_roll\_joint(position\_roll)

time.sleep(0.1)

position\_roll **\*=** **-**1

​

**def** mira\_moverandomly(self):

roll **=** random.uniform(**-**0.15, 0.15)

pitch **=** random.uniform(0.0, 0.3)

yaw **=** random.uniform(0.0, 2**\***pi)

self.mira\_movement\_look(roll, pitch, yaw)

​

**def** movement\_random\_loop(self):

"""

Executed movements in a random way

:return:

"""

rospy.loginfo("Start Moving Mira...")

**while** **not** rospy.is\_shutdown():

self.mira\_moverandomly()

self.mira\_movement\_laugh()

​

​

**if** \_\_name\_\_ **==** "\_\_main\_\_":

mira\_jointmover\_object **=** MiraJointMover()

mira\_jointmover\_object.movement\_random\_loop()

​

​

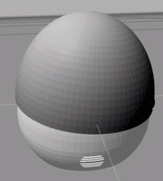
END my\_mira\_demo.py

|  |
| --- |
| Execute in WebShell #2 |



rosrun my\_mira\_description my\_mira\_demo.py

You should get movements like these:



Basically, this class publishes in the controller topics **( /mira/roll\_joint\_position\_controller/command, /mira/pitch\_joint\_position\_controller/command /mira/yaw\_joint\_position\_controller/command )** and checks the **/mira/joint\_states** topic to make sure that the robot has arrived at the given position.  
It also deals with the fact that joint angles sometimes have negative or complete turn versions of the same angle, and you have to make a system similar to the one in the function "assertAlmostEqualAngles" to be sure that the angles are the same, even though they might be different versions of the same angle.

For more details:  
<http://gazebosim.org/tutorials/?tut=ros_control>  
<http://wiki.ros.org/urdf/XML/Transmission>

4. Adding Sensors

The last step is to add sensors to the robot. In this case, you have to add a camera where the "camera\_link" is.  
But thanks to the gazebo plugin system, this is a plug and play operation. You just have to add the following tag to the end of the URDF file:

In [ ]:



**<!--** camera **-->**

**<**gazebo reference**=**"camera\_link"**>**

**<**sensor type**=**"camera" name**=**"camera1"**>**

**<**update\_rate**>**15.0**</**update\_rate**>**

**<**camera name**=**"head"**>**

​

**<**pose**>**0 0 0 0 0 1.57**</**pose**>**

​

**<**horizontal\_fov**>**1.3962634**</**horizontal\_fov**>**

**<**image**>**

**<**width**>**400**</**width**>**

**<**height**>**400**</**height**>**

**<**format**>**R8G8B8**</**format**>**

**</**image**>**

**<**clip**>**

**<**near**>**0.01**</**near**>**

**<**far**>**100**</**far**>**

**</**clip**>**

**<**noise**>**

**<**type**>**gaussian**</**type**>**

**<**stddev**>**0.007**</**stddev**>**

**</**noise**>**

**</**camera**>**

**<**plugin name**=**"camera\_controller" filename**=**"libgazebo\_ros\_camera.so"**>**

**<**alwaysOn**>**true**</**alwaysOn**>**

**<**updateRate**>**0.0**</**updateRate**>**

**<**cameraName**>**mira**/**camera1**</**cameraName**>**

**<**imageTopicName**>**image\_raw**</**imageTopicName**>**

**<**cameraInfoTopicName**>**camera\_info**</**cameraInfoTopicName**>**

**<**frameName**>**camera\_link**</**frameName**>**

**<**hackBaseline**>**0.07**</**hackBaseline**>**

**<**distortionK1**>**0.0**</**distortionK1**>**

**<**distortionK2**>**0.0**</**distortionK2**>**

**<**distortionK3**>**0.0**</**distortionK3**>**

**<**distortionT1**>**0.0**</**distortionT1**>**

**<**distortionT2**>**0.0**</**distortionT2**>**

**</**plugin**>**

**</**sensor**>**

**</**gazebo**>**

As you can see, there are many variables that can be changed to adapt the camera to the real specifications. But the most important variable to set is the **reference="camera\_link,"** which will link the camera to the camera\_link position. The **pose** tag is also important. This allows it to point the camera where it should go and not based on the camera\_link axis. In this case, you can see that it's turned 90º to point in the correct direction.

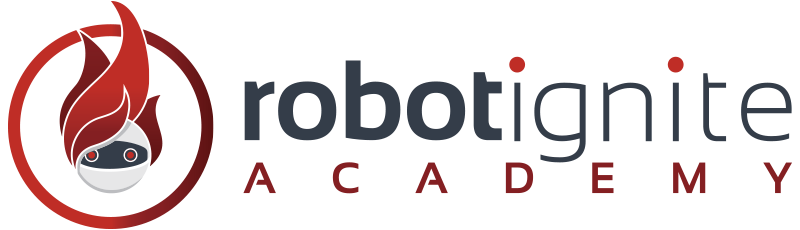
Exercise U2-4

* Add the camera sensor into a final version of URDF called **mira.urdf**.
* Also create a **spawn\_mira.launch** and **start\_mira.launch** to spawn this mira.urdf and start the **mira\_control.launch**.

END Exercise U2-4

Solution Exercise U2.4

Please try to do it by yourself, unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.



Follow this link to open the solutions notebook for Adapt URDF for Gazebo Simulator: [solutions\_Course\_urdfROS\_Unit\_2](https://i-0500135e5b3ce7d04.robotigniteacademy.com/d7e6f789-1484-4bd3-aac3-7d6fe70eb589/jupyter/notebooks/course_urdf_ROS/extra_files/solutions_Course_urdfROS_Unit_2.ipynb)

Spawn an object to make the camera see something more than an empty world. Lets spawn a **banana** and a **tennisball**:

|  |
| --- |
| Execute in WebShell #2 |

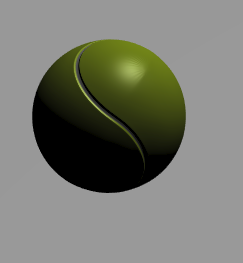
In [ ]:



roslaunch models\_spawn\_library\_pkg spawn\_banana.launch

roslaunch models\_spawn\_library\_pkg spawn\_tennisball.launch





And make it move the head to see the image feed change:

|  |
| --- |
| Execute in WebShell #2 |

In [ ]:



rosrun my\_mira\_description my\_mira\_demo.py

To view what the camera sees, just execute rqt\_image\_view and select the image topic:

|  |
| --- |
| Execute in WebShell #3 |

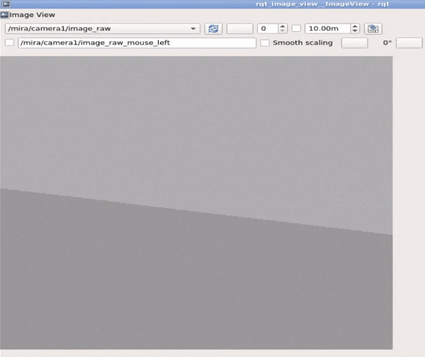
In [ ]:



rosrun rqt\_image\_view rqt\_image\_view

Then, open GraphicalTools:





Warning

In this case, removing the robot **with sensors** will crash Gazebo in any system. This is a known issue and they are working on solving it. For you as user, when you delete a robot model with sensors, just restart your Gazebo or, in RobotIgniteCase, just go to Unit 0 and come back again, restarting all of the systems.

Another solution is executing the command "rosservice call /gazebo/reset\_simulation TAB-TAB." This will kill the Gazebo process that might be a zombie. Once Gazebo is killed, RobotIgniteSystem will detect it and relaunch a fresh, new Gazebo for you.

Don't forget that you can remove any object of the scene like this:

In [ ]:



rosservice call **/**gazebo**/**delete\_model "model\_name: 'banana1'"

rosservice call **/**gazebo**/**delete\_model "model\_name: 'tennis\_ball1'"

rosservice call **/**gazebo**/**delete\_model "model\_name: 'mira'"

Congratulations! You can now create any robot from scratch and simulate it in Gazebo

In the next unit, you will learn how to create the Gurdy Robot and how to convert URDF files into XACRO format files for cleaner robot model definitions.