Simple way to build a robot and control it in Python with CoppeliaSim ZeroMQ Remote API

BnZr

ENSTA Bretagne

December 2023

Basic Start

- Foreword
- Installation
- Remote API
- First test program
- Creating the arena
- Creating the robot
- Adding sensors
- Testing the robot

Foreword

To make a dynamic model of a mobile robot operating in a dynamic environment with only skills in Python, we will use a robot simulator

- Many simulators are specialized in robotics arms simulation, they are not the best choice
- Gazebo (https://gazebosim.org/home) is the reference, it's open source, but the learning step is a bit harsh for beginners. Although Gazebo can be used in a stand alone way, the key benefit to use it is it's strong link with ROS2 robot operating system.
- CoppeliaSim (https://www.coppeliarobotics.com/), formerly known as V-Rep, is quite simple to use and offers a free license for educational purpose.
- Webots (https://cyberbotics.com/) is also simple to use and is open-source since December 2018

Foreword - Choice

Here we will use CoppeliaSim as we practice it in teaching since around 10 years and we have experienced that it's quite easy for students to dynamically model their first robots.

Webots could have been a good choice too, but, 10 years ago it was not open-source.

We will use a remote API (Application Programming Interface) to avoid coding inside the robot simulator.
https://www.coppeliarobotics.com/helpFiles/en/remoteApiOverview.htm

We may have used C++ or Java, but we will use Python to keep the programming simple.

Installation

Installation is quite simple as we will use the latest version of CoppeliaSim (now 4.6.0)

Depending of your operating system (Linux, MacOS or Windows), you can choose the right version on the download page : (https://www.coppeliarobotics.com/downloads

If, for some reason, you need a former version, you can get it here : https://www.coppeliarobotics.com/previousVersions

Remote API

- We will call API functions to exactly reproduce what we do when interactively creating a robot with the graphical interface.
- This approach looks a bit more complex at a first view, but it turns out that it is far more easy to modify the robot's design afterward. It also allows to create scenes with many elements placed using mathematical functions (instead of manually !)
- The former remote API, called "legacy", was used before version 4.4. It requires to add some code inside the simulator to define a communication link between the simulator and our external Python program. Only a subset of the API functions were available remotely.
  https://www.coppeliarobotics.com/helpFiles/en/legacyRemoteApiOverview.htm
- Here we will use the ZeroMQ remote API, more simple to use and giving access to all the API functions.
  https://www.coppeliarobotics.com/helpFiles/en/zmqRemoteApiOverview.htm

First Test Program

First we start the execution of CoppeliaSim

Then we create a file (test_zeromq_remote_api.py for example) with the following Python code :

```
from coppeliasim_zmqremoteapi_client import RemoteAPIClient

client = RemoteAPIClient()
sim = client.require('sim')
sim.setStepping(True)

sim.startSimulation()
while (t := sim.getSimulationTime()) < 10:
    print(f'Simulation time: {t:.2f} [s]')
    sim.step()
sim.stopSimulation()
```

Open a terminal and type (here we are on Linux, command line may be different on Windows) :

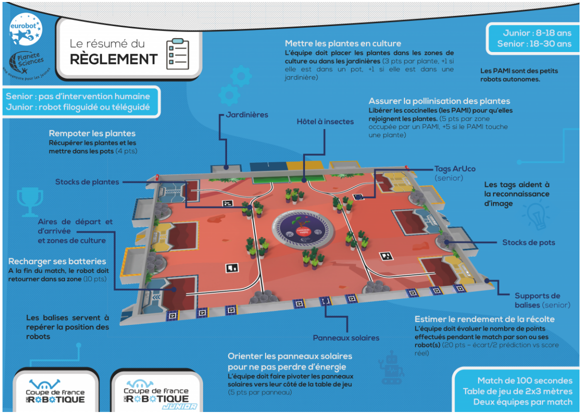
```
python3 test_zeromq_remote_api.py
```

The simuator time should be displayed on the terminal for 10 seconds.

Creating the Arena

We will use the example of the French Robotic Cup 2024. All the elements of the scene can be found here :

https://www.coupederobotique.fr/edition-2024/le-concours/reglement-2024


eurobot
Planète Sciences
une aventure pour les jeunes
Le résumé du
RÈGLEMENT
Jardinières
Hôtel à insectes
Stocks de plantes
Tags ArUco (senior)
Aires de départ et d'arrivée et zones de culture
Stocks de pots
Supports de balises (senior)
Panneaux solaires
Coupe de France de ROBOTIQUE
Coupe de France de ROBOTIQUE JUNIOR

Senior : pas d'intervention humaine
Junior : robot filoguidé ou téléguidé

Junior : 8-18 ans
Senior : 18-30 ans

Mettre les plantes en culture
L'équipe doit placer les plantes dans les zones de culture ou dans les jardinières (3 pts par plante, +1 si elle est dans un pot, +1 si elle est dans une jardinière)

Les PAMI sont des petits robots autonomes.

Assurer la pollinisation des plantes
Libérer les coccinelles (les PAMI) pour qu'elles rejoignent les plantes. (5 pts par zone occupée par un PAMI, +5 si le PAMI touche une plante)

Rempoter les plantes
Récupérer les plantes et les mettre dans les pots (4 pts)

Les tags aident à la reconnaissance d'image

Recharger ses batteries
A la fin du match, le robot doit retourner dans sa zone (10 pts)

Les balises servent à repérer la position des robots

Estimer le rendement de la récolte
L'équipe doit évaluer le nombre de points effectués pendant le match par son ou ses robot(s) (20 pts – écart/2 prédiction vs score réel)

Orienter les panneaux solaires pour ne pas perdre d'énergie
L'équipe doit faire pivoter les panneaux solaires vers leur côté de la table de jeu (5 pts par panneau)

Match de 100 secondes
Table de jeu de 2x3 mètres
Deux équipes par match

Creating this arena and a two wheels robot will show the key elements of dynamic modeling with CoppeliaSim ZeroMQ remote API.

Creating the Arena - Convert STEP file

We will describe how to create the table, the principle will be the same for the others elements of the arena.

The CAD elements are downloaded from :
https://www.coupederobotique.fr/wp-content/uploads/3D_2024_FINAL_V1.zip After decompressing the zip archive, the table is in :
3D_FINALE_V1/step/table_2024.stp

The STEP format (.stp) cannot be imported directly in CoppeliaSim, so we convert it in STL (.stl) using the Python module **cadquery** :
https://cadquery.readthedocs.io/en/latest/index.html

If not installed, cadquery can be installed with pip :

```
python3 -m pip install cadquery
```

The conversion is simply achieved with this Python script executed in 3D_FINALE_V1/step folder :

```
import cadquery as cq
cad_data=cq.importers.importStep("table_2024.stp")
cq.exporters.export(cad_data,"table_2024.stl")
```

Creating the Arena - Connect

- To connect our Python script to CoppeliaSim, we take the first lines of the test program we used before to test the ZeroMQ remote API.
- As we do not want to control the simulation steps from the Python program, we put **setStepping** to False.
- We create a file called "create_arena.py" and put this lines in it :

```
from coppeliasim_zmqremoteapi_client import RemoteAPIClient

client = RemoteAPIClient()
sim = client.require('sim')
sim.setStepping(True)
```

After that we will able to use all the API functions listed here by families :
https://www.coppeliarobotics.com/helpFiles/en/apiFunctions.htm

Creating the Arena - Table

- To create the table in CoppeliaSim, we just need to create a shape using the STL file **"table_2024.stl"** we just converted from STEP. For this, we use **sim.importShape()** API function.
- The function return an **handle** which is an integer value that can be used to change the properties of the table.
- For example, to give a name to this shape, we can use its handle (**table_handle**) in **sim.importShape()** function.
- The following Python code is added in **"create_arena.py"**:

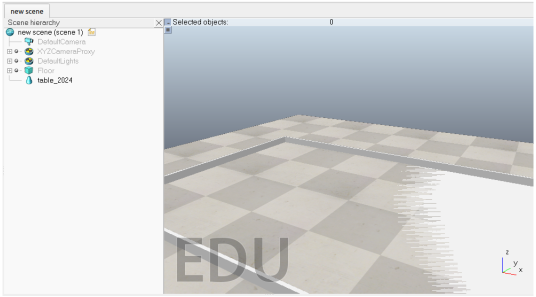
```
# change dir_path_stl to your own path if needed
dir_path_stl = os.path.join(os.getcwd(),"3D_FINALE_V1/stl/")
file_format = 0 # auto detect format
file_name = os.path.join(dir_path_stl,'table_2024.stl')
options = 16 # tries to preserve textures
identical_vertice_tolerance = 0.0 # default
scaling_factor = 0.001 # mm to m
table_handle = sim.importShape(file_format, file_name, options,
                               identical_vertice_tolerance,
                               scaling_factor)
sim.setObjectAlias(table_handle, "table 2024")
```

Creating the Arena - Table

We first make sure that CoppeliaSim has been launched and we execute the Python script :

```
python3 create_arena.py
```

The table should appear in the 3D window of CoppeliaSim :



Creating the Arena - Table

The default floor creates confusing view in the previous picture. As it will be useless, we can remove it.

- In the scene tree, in the upper left part of the previous picture, we expand the "Floor" object to find that it uses a "box" shape.
- First we need to find the handles of "Floor" and "box" with **sim.getObject()**. The name of the object must take into account the path in the scene tree. "Floor" is a the top of the tree, its name is "/Floor", "box" is under "Floor", its name is "/Floor/box".
- Then we can call **sim.removeObjects()** function with an array containing all the handles of the object we want to remove
- The following Python code is added in **"create_arena.py"**:

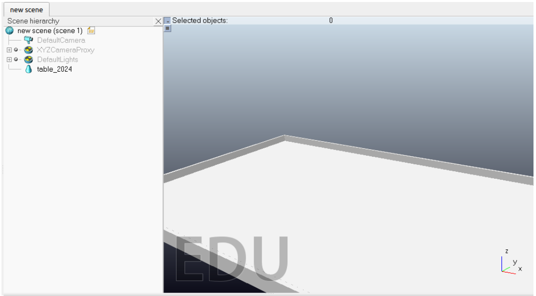
```
# remove default floor
try:
    floor_box_handle = sim.getObject("/Floor/box")
    floor_handle = sim.getObject("/Floor")
    sim.removeObjects([floor_box_handle,floor_handle])
except:
    print ("Default Floor already removed ...")
```

Creating the Arena - Table

Before executing **"create_arena.py"** Python script, we need to remove the "/table 2024" object. We can remove it using **sim.removeObjects()** or simply by using the graphical user interface. Another way is to simply close the scene in CoppeliaSim "File" menu.
Note that if we do not delete the existing "/table 2024" object, the execution of **"create_arena.py"** will create another "/table 2024" object.

```
python3 create_arena.py
```

The default floor should have disappeared in the 3D window of CoppeliaSim :



Creating the Arena - Table

We will now add the texture of the floor of the arena. We can get it from:
https://www.coupederobotique.fr/wp-content/uploads/vinyle_2024_FINAL_V1.zip
After decompressing, we get the file:
"vinyle_FINALE_V1/vinyle_table_2024_FINAL_V1.svg".

The texture is created with **sim.createTexture()** function and Python code looks like this :

```
dir_path_textures = os.path.join(os.getcwd(),"vinyle_FINALE_V1")
file_name = os.path.join(dir_path_textures,
                         'vinyle_table_2024_FINAL_V1.svg')
options = 0
plane_sizes = [3.0,2.0]
scaling_UV = [3.0,3.0]
xy_g = [0,0,math.radians(180)] # or [0,0,0]
fixed_resolution = 0
resolution = None
tapis_handle, id, res = sim.createTexture(file_name, options,
                            plane_sizes, scaling_UV, xy_g,
                            fixed_resolution, resolution)
sim.setObjectAlias(tapis_handle, "tapis")
```

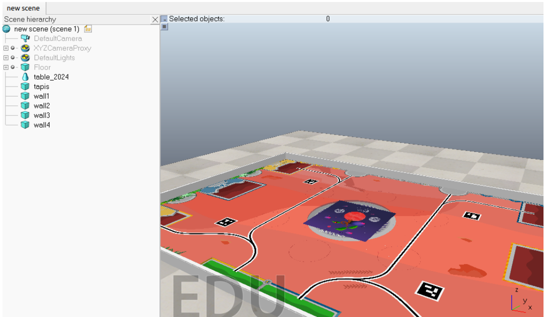
Creating the Arena - Table

To see the texture we move the texture plane up 0.5 mm in the z direction using **sim.setObjectPosition()**.

```
sim.setObjectPosition(tapis_handle, [0,0,0.0005],
                      sim.handle_world)
```

To have dynamic objects not falling through the floor, we need the floor to respond to the forces exerted by the objects. We dot it by making the floor "respondable" with **sim.setObjectInt32Param()** function.

```
sim.setObjectInt32Param(tapis_handle,
                        sim.shapeintparam_respondable,1)
```



Creating the Arena - Table

The table object can be made dynamic and visible to the sensors. However, to reduce computation time, it is better to add 4 walls. The example is given for one wall :

- length : 2.00 m, width : 2 cm and height 7.5 cm
- position (of the center) : x = -1.51 m, y = 0 m and z = 3.75 cm
- orientation : rotation of 90 ^{o} around z
- allow collision with other objects : respondable
- object is static , if we want the wall to be pushable or destroyable by a robot, static must be set to 0
- object is used only for computation, we make it invisible
- usable by all sensors : collidable, measurable and detectable

Creating the Arena - Table

The Python code to create the wall and set all the properties we listed is ad follows :

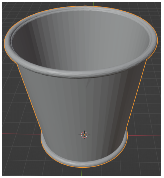
```
wall4_handle = sim.createPrimitiveShape(sim.primitiveshape_cuboid,
                                        [2.00, 0.02, 0.15], options)
sim.setObjectAlias(wall4_handle, "wall4")
sim.setObjectPosition(wall4_handle, [-1.51,0,0], sim.handle_world)
sim.setObjectOrientation(wall4_handle, [0,0,math.pi/2],wall4_handle)
prop = sim.objectspecialproperty_collidable
prop += sim.objectspecialproperty_measurable
prop += sim.objectspecialproperty_detectable
sim.setObjectInt32Param(wall4_handle,sim.shapeintparam_respondable,1)
sim.setObjectInt32Param(wall4_handle,sim.shapeintparam_static,1)
sim.setObjectInt32Param(wall4_handle,sim.objintparam_visibility_layer,0)
sim.setObjectSpecialProperty(wall4_handle, prop)
```

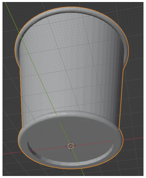
All other elements of the arena can be added is the same way. An object is defined by 2 or 3 components :

- a visual component that can be complex
- a dynamic component of very simple (primitive) shape (sphere, cuboid, cylinder, ...) to speed up the computation of the dynamics
- a collision component of simple shape. Often the dynamic and the collision components can be the same
- if the shape is complex, it can be approximated by a group of primitive shapes

Creating a Complex Shape

We can take as example the pot shape. The shape of the pot, retrieved for the STL CAD file looks like this :





We have to add 36 pots (6 batches of 6 pots), if we make this shape "dynamic" and "respondable", the simulation becomes extremely slow

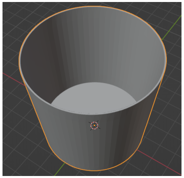
$\rightarrow$ this shape is far too complex to be used for dynamic and collisions computations.

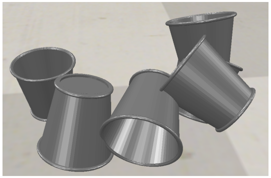
Creating a Complex Shape

An idea could be to simplify the mesh by, at least, 2 methods:

1. decimating the shape (reducing the number of polygons)
2. creating a simpler shape using an outer truncated cone extruded by an inner truncated cone

The left figure shows the simplified mesh using the second method.





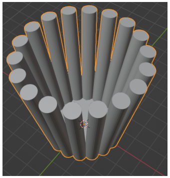
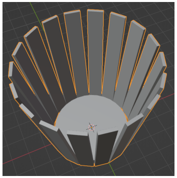
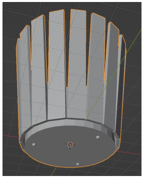
The center image show a batch of 6 pots before starting the simulation After a few seconds of simulation, some pots becomes unstable and fall (right image). The initial falling of the center back upper pot in the lower pot is not well simulated. Simulation is very slow (less the 20% of real-time).

$\rightarrow$ this shape is still too complex to be used for dynamics and collisions.

Creating a Complex Shape

The shape is made by grouping simple primitive shapes.

- The basis is a cylinder
- The side can be a combination of cylinders (left figure) or of cuboids (center figure)
- The interface with the floor is replace by 3 hemispherical contacts (right figure)

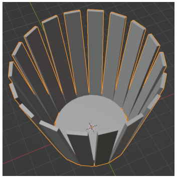
→ this shape will be the parent of the visible shape, so that it looks like the visible shape obeys to dynamics and collides with other objects in the scene.

Creating a Complex Shape

we will used for dynamics and collisions the shape with side made of cuboids (left figure)

The batch of 6 pots (center image) is now well handled by the simulation (right image).

In particular, the initial falling of the center back upper pot in the lower pot is now well simulated.



→ we keep this shape to represent the pot for dynamics and collision.

Creating a Complex Shape

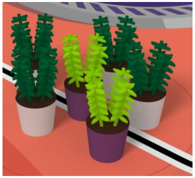
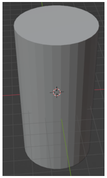
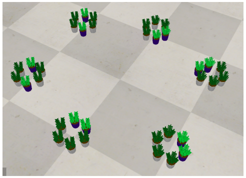
The API functions used for creating the dynamics and collision shape are :

- **sim.createPrimitiveShape()** to create the simple shapes
- **sim.setObjectOrientation()** to change the orientation of a shape
- **sim.setObjectPosition()** to translate the shape at the right place
- **sim.setObjectInt32Param()** with **sim.shapeintparam_respondable** parameter to allow the object to collide with others collidable objects
- **sim.setObjectInt32Param()** with **sim.shapeintparam_static** parameter to make the object dynamic
- **sim.setObjectInt32Param()** with **sim.objintparam_visibility_layer** parameter to make the object non visible
- **sim.setObjectSpecialProperty()** to make it usable with sensors
- **sim.groupShapes()** to group the shapes in a single shape entity
- **sim.setObjectParent()** to make the dynamic shape the parent of the visible shape, so that the visible shape follows the motion of the dynamic shape

Creating a Complex Shape

The plants (left image) can be represented for dynamics and collisions with a simple vertical cylinder (center picture).

The simulation of the 36 plants (6 batches of 6) is fast and stable (right image).

→ Note that if the cylinder is not realistic enough to simulate the plants, we can use the same kind of assembly of simple shapes we used before for the pots.

Creating the robot - Body & Wheels

As for the pots and the plants, the robot will require a simple shape for dynamics and collisions and a more complex shape for its visualization

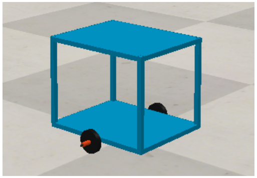
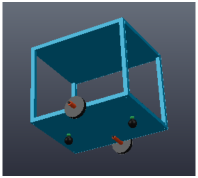
The visualization shape is generally imported from the CAD design of the robot. A visualization shape is simply attached as a child to its corresponding dynamic shape, so they it can follow its motion.

For the dynamics and collision shapes, the main difference with pots and plants is that the robot needs moving parts to, for example, get into motion or grab some objects.

Creating the robot - Body & Wheels

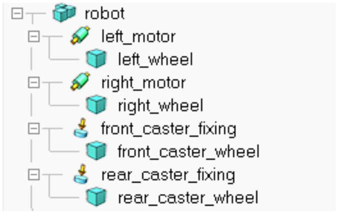
Here we define a simple differential drive robot (figures below) with :

- a body made of a top and a bottom rectangular plates and 4 vertical square section poles
- two main wheels
- two motors , one per main wheel, to link the wheels to the body
- two spherical caster wheels for stability, one front and one rear
- two force sensors, to attach the caster wheels to the body

Creating the robot - Robot Tree

The shapes of the elements constituting the robot are arranged in a tree :


robot
left_motor
left_wheel
right_motor
right_wheel
front_caster_fixing
front_caster_wheel
rear_caster_fixing
rear_caster_wheel

- The robot's body is the root of the tree
- The motors are attached to the body
- The left and right wheels are attached to the motors
- The caster wheels are just spherical contacts to stabilize the robot in pitch.
- As they are not rolling, joints are not required and they are attached to the body with a force sensor
- The force sensor is not used to measure the force, it is used as a fix attachment that can brake if the force applied on the fixation is above a predefined threshold

Creating the robot - Motors

The dynamics and collision shapes are created using primitives shapes :

- 6 cuboids (top & bottom plates, 4 corner poles) grouped together to make the body. *(Note : the 6 shapes are grouped and not merged for faster simulation)*
- Wheels are simulated with cylinders
- Caster wheels are simulated with sphere

The left and right motors are simulated with "revolute" joints, that are controlled in velocity. *(Note : joints can also be controlled in position or force, they can also left free)*. The API functions used for creating the joints are :

- **sim.createJoint()** to create the joints
- **sim.setObjectOrientation()** to define the orientation of a joint
- **sim.setObjectPosition()** to translate the joint at the right place
- **sim.setObjectAlias()** to give a name to the joint (e.g. "left_motor")
- **sim.setObjectParent()** to make the joint the parent of the wheel attached to it
- **setObjectInt32Param()** with **sim.jointintparam_dynctrlmode** parameter to set the control mode of the joint (here "velocity")
- **sim.setJointTargetVelocity()** to set the velocity set point of the joint

Creating the robot - Caster Wheels

Caster wheels are spheres attached to the body with a force sensor.

The API functions used for creating the force sensors are :

- **sim.createForceSensor()** to create the force sensors
- **sim.setObjectOrientation()** to define its orientation
- **sim.setObjectPosition()** to define its position
- **sim.setObjectAlias()** to give a name to the force sensor

Creating the robot - Adding Sensors

We would like to add a LIDAR (2D) and a video camera to our robot.

CoppelaSim offers predefined sensors families : proximity, vision and force.

The video camera is simulated as a vision sensor.

The LIDAR may have been simulated with a rotating proximity (laser) sensor, but this turn out to be slow. The 360 degrees LIDAR is made with a combination of 3 vision sensors with 120 degrees FOV (Field Of View). The depth information in the vision sensor is used to model the LIDAR range.

Adding sensors - Video Camera

To be explained

Adding sensors - 2D LIDAR

To be exlained

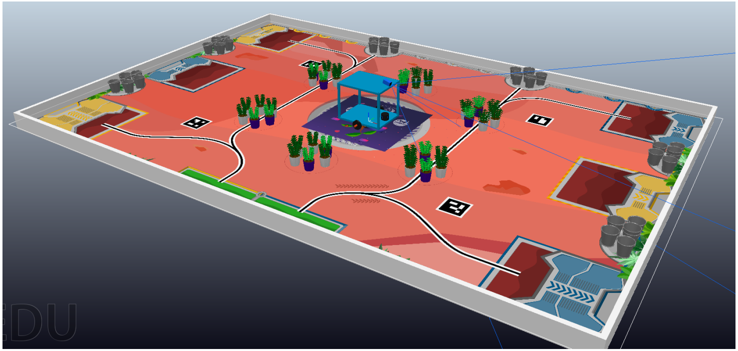
Creating the robot - Test It!

We can now build and test our simulation :

```
# open an empty CoppeliaSim scene
# create the arena, the robot, the plants and the pots
python create_arena.py
python create_robot.py
python add_plants.py
python add_pots.py
# click on the start simulation button on Coppelia Sim
# then test the robot motion with :
python test_robot_motion.py
# and test the sensors with :
python test_robot_sensors.py
```

Creating the robot - Test It!

The "Coupe de France 2024" arena looks like this :

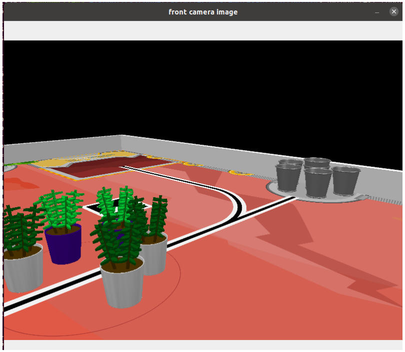


→ Simulation speed is around 60 % of real-time on a Rizen 5 with Nvidia GTX 1050.

→ Making some plants and pots static will improve simulation speed.

Creating the robot - Test It!

The video camera (left) and the 2D LIDAR (right) looks like this :


front camera image

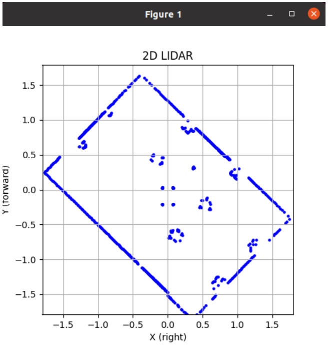


Figure 1
2D LIDAR
Y (forward)
X (right)