REPORT - LAB 5

Name: Puya Fard (pfard@uci.edu)
Course: EECS221-B
Subject: Motion Planning Using ROS (Part 3)

Instructor notes:

Objective

This lab provides an introduction to Gazebo capabilities for Robot Simulation, a tool made for teaching Navigation and Manipulation, making it suitable for robotic applications. This lab provides a set of examples/exercises for motion planning, focused in the topics of path planning and low-level control.

Experiment

Problem : Full Path Planning Stack

We will first set up VM provided to us by lab manual and run it using VMware

- first need to download UTM, choose the package corresponding to your operating system. Download the LAB3 MAC M1 file and open it with UTM to import the VM.

Then we will run our **gazebo bot command** and start the robot by clicking the play button.

>>ros2 launch turtlebot4_ignition_bringup turtlebot4_ignition.launch.py model:=lite

Then once ran, we will do the following steps:

1. Open a terminal "Terminal 1" and run the node called **Motion_Planner**

>>ros2 run TurtleBot Motion_Planner

2. Open another terminal "Terminal 2" and run a node called **rrt_node**

>> ros2 run TurtleBot rrt_node

3. Open another terminal "Terminal 3" and run a node called **PID_Controller**

>>ros2 run TurtleBot PID_Controller

Now we have successfully run all nodes required in order to run our full path planning stack. Our program will generate a **plot** of our trajectory, **the gazebo won't be moving yet.** Once we **close** the plot by clicking the **x button**, the gazebo will **start moving** through the trajectory drawn by our rrt_node.

We can further analyze the flowchart of our system on the **fig. 1** below:

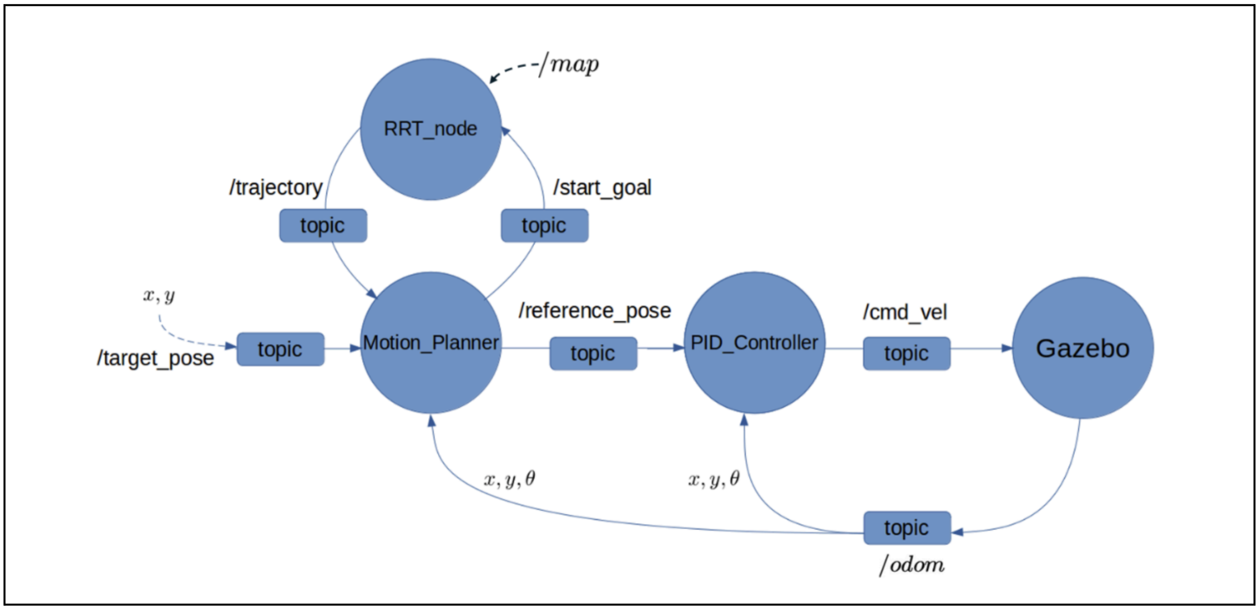

Fig 1: System flowchart

Approach Breakdown:

1. In this system, our code heavily relies on our **Motion_Planner** node. Motion_Planner will ask users to input **(x,y)** coordinates and it will send them to our RRT_node via **/start_goal** topic.

2. **RRT_node** will generate the path starting from coordinates (0,0) to goal (x,y) set by the user on an empty 1**00x100 np.matrix map with step size 5 and resolution 1.** It will publish the coordinates to **/trajectory** topic and will also output the path generated on a table for debugging purposes.

3. Motion_Planner will take the coordinates generated by our RRT_node and publish them on topic **/reference_pose** for it to be used in our PID_Controller node.

4. **PID_Controller** will adjust the **velocity** and **angle** of our gazebo bot accordingly in order to follow the trajectory drawn by the rrt_node via **/cmd_vel topic.**

Explaining Code (Motion_Planner)

```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from std_msgs.msg import Float64MultiArray
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Point
from math import sqrt, atan2

class MotionPlanner(Node):

def __init__(self):
super().__init__('motion_planner')

self.current_position = None
self.target_position = None
self.trajectory = []
self.trajectory_index = 0
self.reached_goal = False

self.start_goal_pub = self.create_publisher(Float64MultiArray, '/start_goal', 10)
self.reference_pose_pub = self.create_publisher(Float64MultiArray, '/reference_pose',
10)
self.target_pose_pub = self.create_publisher(Float64MultiArray, '/target_pose', 10)
self.subscription_odom = self.create_subscription(Odometry, '/odom',
self.odom_callback, 10)
self.subscription_trajectory = self.create_subscription(Float64MultiArray,
'/trajectory', self.trajectory_callback, 10)

self.timer = self.create_timer(0.1, self.timer_callback)
```

In the code given above, we will start by an initialization process that our node will subscribe to and publish to the required topics such as /start_goal, /reference_pose, /target_pose, /odom and /trajectory.

```
def get_user_coordinates(self):
while rclpy.ok():
try:
goal_x = float(input("Enter goal x coordinate (meters): "))
goal_y = float(input("Enter goal y coordinate (meters): "))
```

```
self.target_position = (goal_x, goal_y)

if self.current_position:
self.publish_start_goal(self.current_position[0], self.current_position[1], goal_x,
goal_y)
self.publish_target_pose(goal_x, goal_y)

except ValueError:
self.get_logger().error("Invalid input. Please enter numeric values for coordinates.")
except rclpy.exceptions.ROSInterruptException:
break
```

In the code given above, motion planner will ask user to input x and y coordinates then publish them in topic /start_goal and /target_pose.

```
def odom_callback(self, msg):
self.current_position = (msg.pose.pose.position.x, msg.pose.pose.position.y,
self.quaternion_to_euler(msg.pose.pose.orientation))
self.get_logger().info(f"Updated current position: {self.current_position}")
```

In the code given above, motion planner will get gazebo bots current position as starting x and y coordinates.

```
if not self.target_position:
self.get_user_coordinates()

def trajectory_callback(self, msg):
self.trajectory = []
data = msg.data
for i in range(0, len(data), 2):
self.trajectory.append((data[i], data[i + 1]))
self.trajectory_index = 0
self.reached_goal = False
self.get_logger().info("Received trajectory:")
for point in self.trajectory:
self.get_logger().info(str(point))
```

In the code given above, motion planner will log the data collected by trajectory received.

```
def timer_callback(self):
if self.trajectory and not self.reached_goal:
```

```
if self.is_close_to_goal(self.trajectory[self.trajectory_index]):
self.trajectory_index += 1
if self.trajectory_index < len(self.trajectory):
self.publish_reference_pose(self.trajectory[self.trajectory_index])
else:
self.get_logger().info("Reached the final goal.")
self.reached_goal = True
else:
self.publish_reference_pose(self.trajectory[self.trajectory_index])

def publish_start_goal(self, start_x, start_y, goal_x, goal_y):
start_goal_msg = Float64MultiArray()
start_goal_msg.data = [start_x, start_y, goal_x, goal_y]
self.start_goal_pub.publish(start_goal_msg)
self.get_logger().info(f"Published start ({start_x}, {start_y}) and goal ({goal_x},
{goal_y}) coordinates.")

def publish_reference_pose(self, pose):
reference_pose_msg = Float64MultiArray()
reference_pose_msg.data = [pose[0], pose[1]]
self.reference_pose_pub.publish(reference_pose_msg)
self.get_logger().info(f"Published reference pose ({pose[0]}, {pose[1]}).")
```

In the code given above, node will publish the reference pose x and y to /reference_pose topic.

```
def publish_target_pose(self, goal_x, goal_y):
target_pose_msg = Float64MultiArray()
target_pose_msg.data = [goal_x, goal_y]
self.target_pose_pub.publish(target_pose_msg)
self.get_logger().info(f"Published target pose ({goal_x}, {goal_y}).")
```

In the code given above, node will publish the target pose our goal coordinates.

```
def is_close_to_goal(self, goal, threshold=0.1):
if not self.current_position:
return False
distance = sqrt((self.current_position[0] - goal[0]) ** 2 + (self.current_position[1]
- goal[1]) ** 2)
return distance < threshold

def quaternion_to_euler(self, orientation):
"""
Convert quaternion to euler angles.
```

```
"""
x, y, z, w = orientation.x, orientation.y, orientation.z, orientation.w
t0 = +2.0 * (w * x + y * z)
t1 = +1.0 - 2.0 * (x * x + y * y)
roll_x = atan2(t0, t1)

t2 = +2.0 * (w * y - z * x)
t2 = +1.0 if t2 > +1.0 else t2
t2 = -1.0 if t2 < -1.0 else t2
pitch_y = sqrt(1 - t2 * t2) # asin(t2)

t3 = +2.0 * (w * z + x * y)
t4 = +1.0 - 2.0 * (y * y + z * z)
yaw_z = atan2(t3, t4)
```

In the code given above, node converts a quaternion orientation to Euler angles (yaw in this case).

```
return yaw_z # in radians

def main(args=None):
rclpy.init(args=args)
motion_planner = MotionPlanner()

rclpy.spin(motion_planner)

motion_planner.destroy_node()
rclpy.shutdown()

if __name__ == '__main__':
main()
```

Explaining Code (rrt_node)

```
import rclpy
from rclpy.node import Node
from nav_msgs.msg import OccupancyGrid
from std_msgs.msg import Float64MultiArray
import numpy as np
import random
import math
import matplotlib.pyplot as plt

class RRTNode(Node):
def __init__(self):
super().__init__('rrt_node')
# Subscription to start goal topic
self.subscription_start_goal = self.create_subscription(
Float64MultiArray, '/start_goal', self.start_goal_callback, 10)
# Publisher to trajectory topic once you connect start and goal points
self.trajector_publisher = self.create_publisher(
Float64MultiArray, '/trajectory', 50)
# Global variables
self.resolution = 1 # Assuming each cell represents 1 unit of space
self.origin = [0, 0] # Assuming the map origin is at (0, 0)
self.map_width = 8 # 8x8 map
self.map_height = 8
self.map_data = np.zeros((self.map_height, self.map_width), dtype=int) # 8x8 empty map
self.map_img = None
print(self.map_data)
```

In the code given above, we will initialize our node via its constructor, and then we will make sure that we generate an empty map to be used for the trajectory. The empty map will contain np.zeros matrix size 8x8 with resolution 1.

Furthermore, our node will subscribe to topic /start_goal and publish to /trajectory to be used by our motion planner node.

```
def start_goal_callback(self, msg):
print("Got points to calculate")
# Get start and goal coordinates from /start_goal topic subscription
x_start_real, y_start_real, x_goal_real, y_goal_real = msg.data
x_start_index, y_start_index = self.get_index_from_coordinates(x_start_real,
y_start_real)
x_goal_index, y_goal_index = self.get_index_from_coordinates(x_goal_real, y_goal_real)
start, goal = ([x_start_index, y_start_index], [x_goal_index, y_goal_index])

traj_msg = Float64MultiArray()
print("About to find path")
path = self.rrt(start, goal)
print("Came Back from path\n")
print(path)

if path is not None:
flattened_path = [coord for point in path for coord in point]
print(f"Flattened path {flattened_path}")
traj_msg.data = [float(value) for value in flattened_path]
print(f"traj_msg.data {traj_msg.data}")
self.plot_path_map(path)
else:
traj_msg.data = []
print(f"publishing {traj_msg}\n")
self.trajector_publisher.publish(traj_msg)
print(f"Successfully published {traj_msg}\n")

def get_index_from_coordinates(self, x_real, y_real):
x_index = int(round(x_real / self.resolution))
y_index = int(round(y_real / self.resolution))
return x_index, y_index

def rrt(self, start, goal, max_iter=10000, step_size=10):
nodes = [start]
parents = {tuple(start): None}

for i in range(max_iter):
rand_point = (random.randint(0, self.map_data.shape[1] - 1),
random.randint(0, self.map_data.shape[0] - 1))

nearest_node = self.find_nearest_node(nodes, rand_point)
new_node = self.extend_towards(nearest_node, rand_point, step_size)
```

```
if new_node is not None:
nodes.append(new_node)
parents[tuple(new_node)] = nearest_node

if self.distance(new_node, goal) < step_size:
parents[tuple(goal)] = new_node
path = self.reconstruct_path(parents, start, goal)
print("Path found:")
print(path)
print("Map after path discovery:")
print(self.map_data)
return path

return None

def find_nearest_node(self, nodes, point):
distances = [(self.distance(node, point), node) for node in nodes]
return min(distances, key=lambda x: x[0])[1]

def extend_towards(self, node, target, step_size):
direction = (target[0] - node[0], target[1] - node[1])
distance = self.distance(node, target)

if distance < step_size:
return target

unit_vector = (direction[0] / distance, direction[1] / distance)
new_node = (int(node[0] + unit_vector[0] * step_size),
int(node[1] + unit_vector[1] * step_size))

if self.is_free(new_node):
return new_node
else:
return None

def is_free(self, point):
x, y = point
if 0 <= x < self.map_data.shape[1] and 0 <= y < self.map_data.shape[0]:
return True # All points are free in the empty world
return False
```

```
def distance(self, point1, point2):
return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)

def reconstruct_path(self, parents, start, goal):
path = [goal]
current_node = goal

while current_node != start:
current_node = parents[tuple(current_node)]
path.append(current_node)
print(f"Visiting node: {current_node}")

path.reverse()
return path

def plot_path_map(self, path):
plt.imshow(self.map_data, cmap='binary', origin='lower')
path = np.array(path)
plt.plot(path[:, 0], path[:, 1], 'r', linewidth=2)
plt.colorbar()
plt.title('Occupancy Grid with Path')
plt.xlabel('X (cells)')
plt.ylabel('Y (cells)')
plt.show()
```

This code given above will make sure to plot the trajectory drawn by our rrt_node.

```
def main(args=None):
rclpy.init(args=args)
rrt_node = RRTNode()
rclpy.spin(rrt_node)
rrt_node.destroy_node()
rclpy.shutdown()

if __name__ == '__main__':
main()
```

Finally, this node is a template used from our previous lab4, which will draw out a trajectory path for the given coordinates and will be plotted for the user to debug and analyze the drawn out path.

Analysis

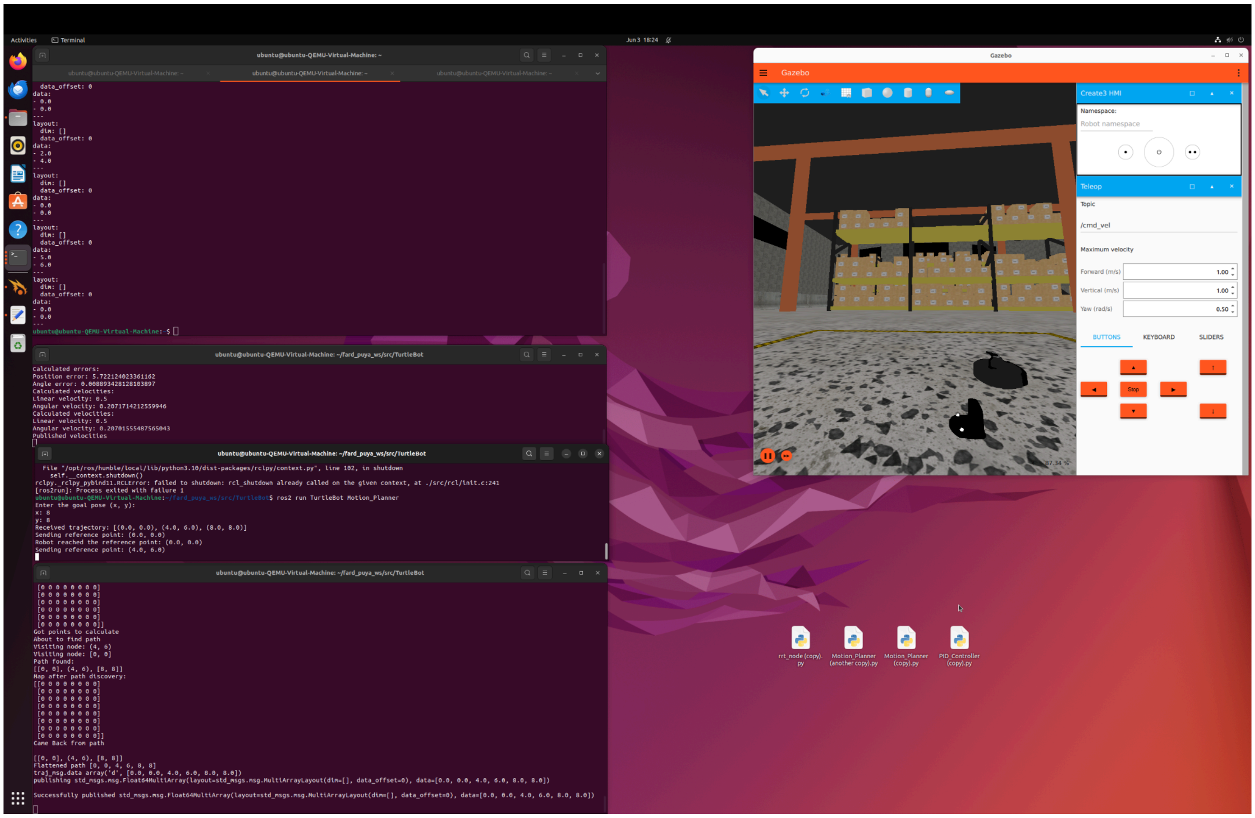

Fig 2: Full autonomous system 1

On the code above, we can analyze the entire system running simultaneously. We have our nodes **Motion_Planner, rrt_node, and PID_Controller running along with our Gazebo turtlebot4.**

We can further analyze the plot of the trajectory drawn out by our rrt_node on the screen for further analysis and debugging of our path generated.

We can finally observe the gazebo bot moving towards the path as generated by our **Motion_Planner.**

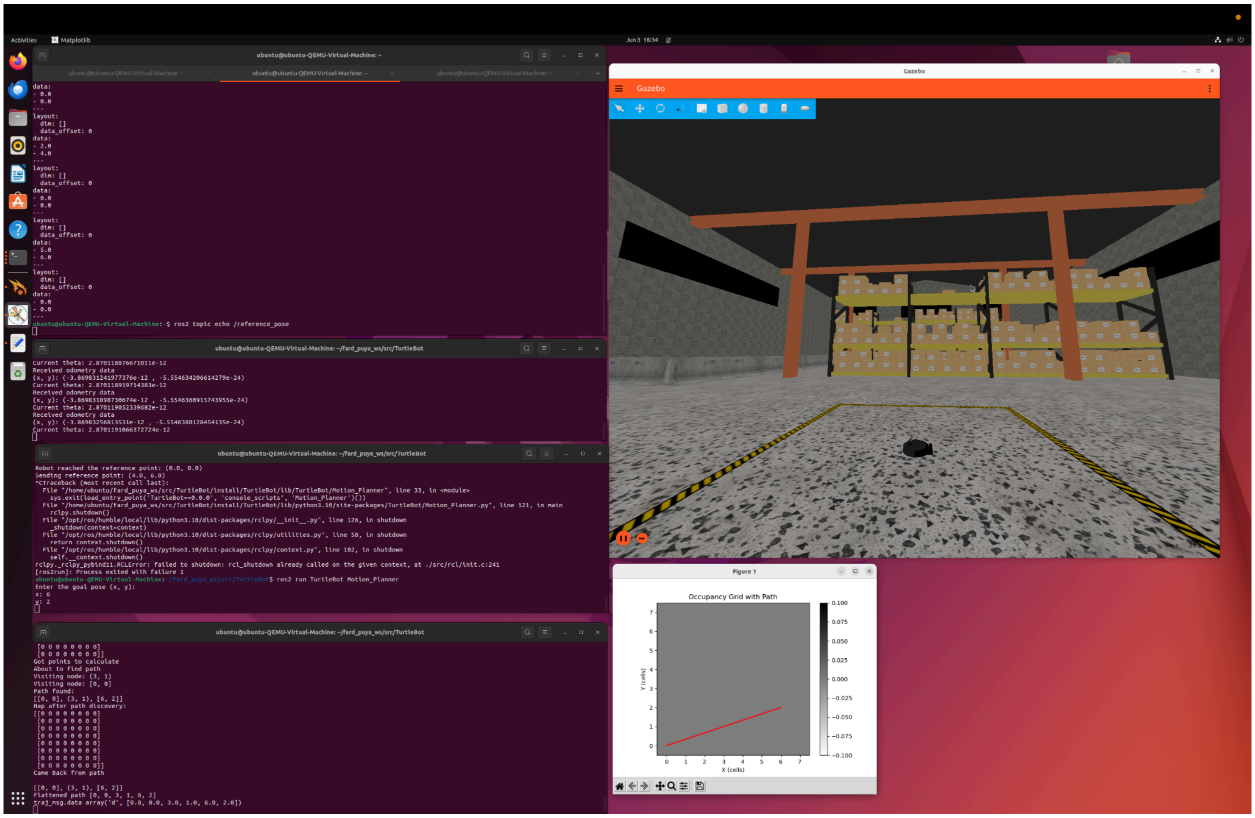

Fig 3: Full autonomous system 2

On the figure above, we can analyze the second trial of another coordinate inputted by the user. We can observe the path being generated and so is the gazebo bot taking its way to reach the goal via PID_Controller. Below, we can see that the path generated by our rrt_node contains different coordinate points until it reaches the goal destination.

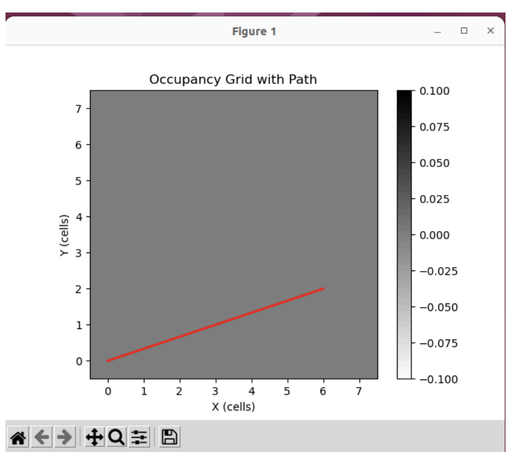

Fig 4: Path generated by rrt_node

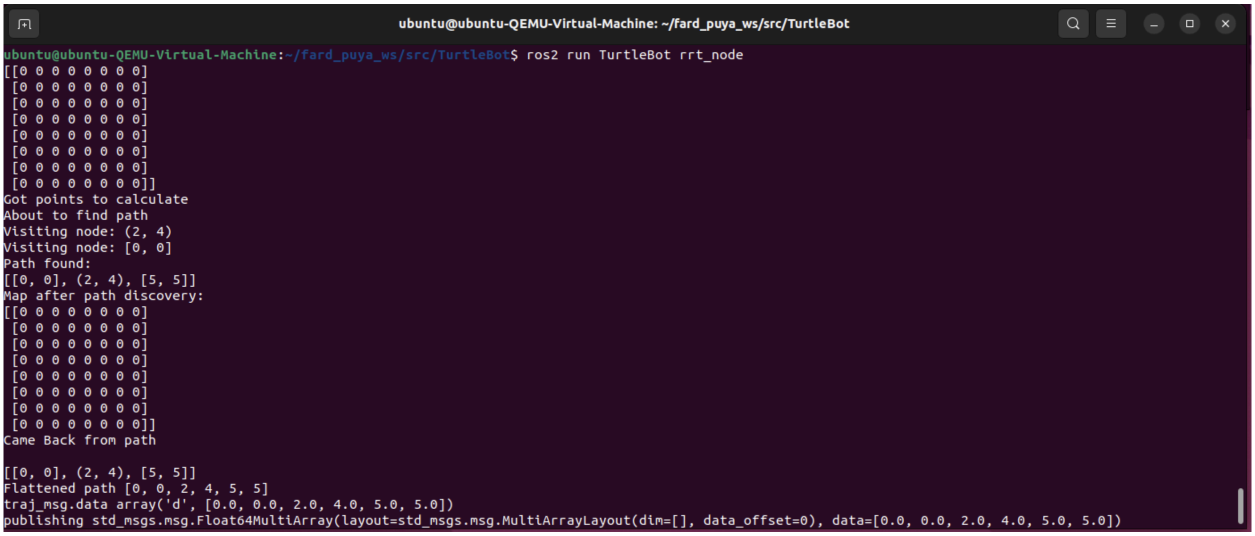

Fig 5: rrt_node

As we can see on the figure above, our rrt_node takes the input from user coordinates and it publishes the **trajectory** and **plot** to visualize.

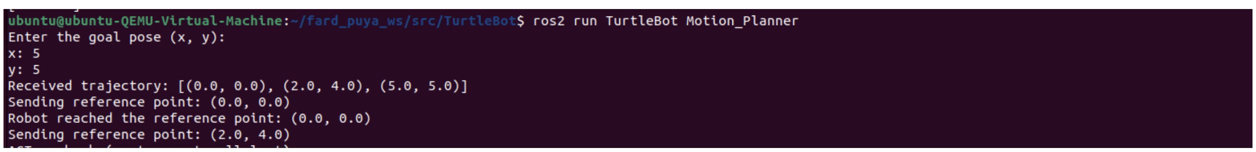

Fig 6: Motion_Planner

As we can see on the figure above, our motion_planner takes the input from user coordinates and it publishes the **start_goal** topic to be used by our rrt_node, then further publishes the trajectory coordinates to **reference_pose** topic to be used by our PID_Controller.

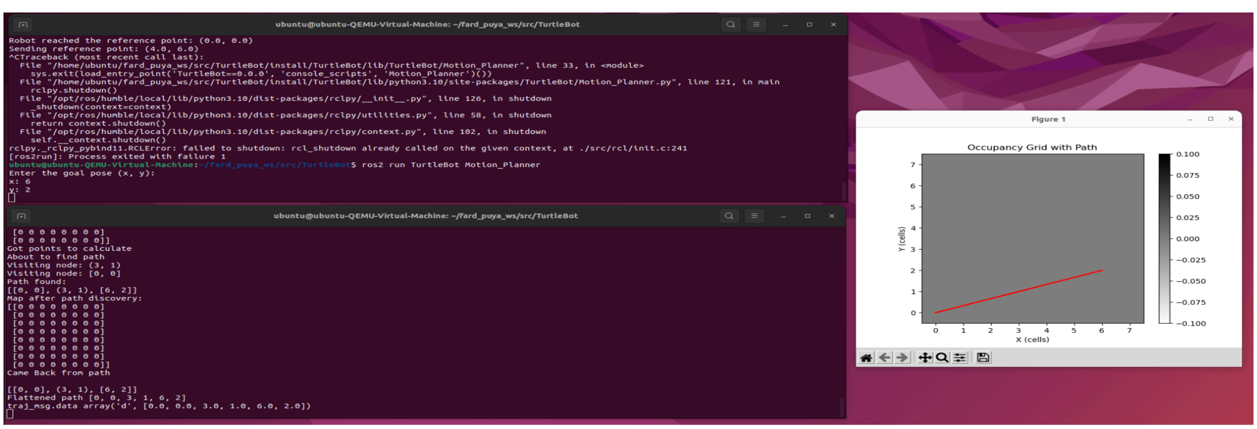

Fig 7: PID_Controller and rrt_node combined

As we can see on the figure above, our **rrt_node and PID_Controller** are shown running along with the path generated by our **trajectory** plot.

Conclusion

Successfully completed the project by following steps given in the lab manual. Learned how to create a full stack autonomous system that generates a trajectory via rrt_node and then publishes it to PID_Controller to move gazebo via reference coordinates generated by the path given. This lab was challenging yet another successful work completed.