**Python script to control robot using waypoints**

**Objective:**

Objective is to control the robot in the virtual machine (gazebo environment) using the python script on the workstation (ubuntu machine) using ROS communication.

There are two ways to control the robot, one from the same virtual machine and the other from the workstation PC.

1. **Control the robot in same virtual machine:**

1. In the virtual machine open the gazebo empty launch file from the desktop.

2. Open the terminal and run

| **cd ~/catkin\_ws/src** |
| --- |

3. Create a package using

| **catkin\_create\_pkg\_prakash\_scripts std\_msgs rospy** |
| --- |

4. Do **cd ~/catkin\_ws** and type **catkin\_make** to build the package

5. Before opening the prakash\_scripts folder source the ros setup using **source /opt/ros/melodic/setup.bash** and type source catkin\_ws/devel/setup.bash

6. Change the directory using

| **roscd prakash\_scripts** |
| --- |

7**.** In the folder place the below zip file and run

| **python model\_predictive\_trajectory\_generator.py** |
| --- |

Use the link below to download the folder.

<https://drive.google.com/drive/folders/1wIYWa4u_cvffzo6nCpwJkALDajFpqA4I?usp=sharing>

8. The code will first generate the trajectory using MPTG program and call the navigator model to move the robot in the gazebo environment using ROS commands

1. **Control the robot through Workstation PC:**

1. This can be achieved by creating the connection between the workstation and the virtual machine using the virtual machine IP address.

2. On the workstation enable zeroconf using the below link

| sudo apt install libnss-mdns avahi-daemon avahi-utils |
| --- |

1. In the workstation terminal window set the environment variables.ROS assumes that the computer it is set up on is the robot. But we are running on the workstation, not on the robot. To tell ROS how to communicate with the robot, type

| export ROS\_MASTER\_URI=[http://](http://ip) IP address:11311 >> ~/.bashrc export ROS\_HOSTNAME=$(hostname).local >> ~/.bashrc |
| --- |

In the IP address place, replace it with your virtual machine’s IP address.r

1. In the virtual machine open the gazebo empty launch file from the desktop.
2. Now download the below file and place it in the desired location catkin\_ws/src/

<https://drive.google.com/drive/folders/1wIYWa4u_cvffzo6nCpwJkALDajFpqA4I?usp=sharing>

1. Change the directory to the above location and launch the python script using

| python model\_predictive\_trajectory\_generator.py |
| --- |

1. The code will first generate the trajectory using the MPTG program and call the navigator model to move the robot in the gazebo environment using ROS commands.

**Reference link:**

<https://learn.ubiquityrobotics.com/workstation_setup>

**3. Place to change the waypoints points**

1. The tar list variable will be above the def main() function in model\_predictive\_trajectory\_generator.py. (MPTG) The variable can be edited to different waypoints, for now the program uses the below values as waypoints to create a trajectory between them.

| tar=[[0.0,0.0,0.0],[3.0,1.0,0.0],[5.0,2.0,45.0]] |
| --- |

4. **Functions to generate trajectory**

This is the main function which calls the test\_optimize\_trajectory and gets the trajectory between the given 3 points and finally, calls the waypoint\_nav model and makes the robot move in the gazebo environment.

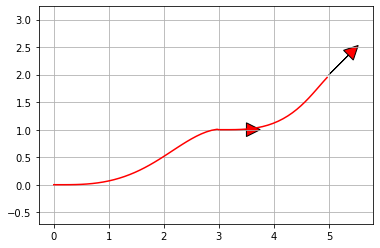
| def main():   temp1=time.time()  x,y= test\_optimize\_trajectory(tar)  l=len(x)  WAYPOINTS=[]  for i in range(0,l):  if(i>0):  WAYPOINTS.append([x[i],y[i]])  waypoint\_nav.move\_function(WAYPOINTS) |
| --- |

**The test\_optimize trajectory** consists of a motion model function to generate the trajectory and a cost calculator to find the cost of the generated trajectory path, and a learning parameter selector function to update the distance vector.

| def optimize\_trajectory(target, k0, p, val):  flag=0  for i in range(max\_iter):  # motion model consist of vehicle dynamics model and generates the #trajectory based on it.  xc, yc, yawc = motion\_model.generate\_trajectory(p[0], p[1], p[2], k0,val)  dc = np.array(calc\_diff(target, xc, yc, yawc)).reshape(3, 1)  print("dc",dc)  # Using the vector above difference vetor the cost is calculated using euclidean distance formula  cost = np.linalg.norm(dc)  print(cost) |
| --- |
| # The learning parameters are updated using the below function  alpha = selection\_learning\_param(dp, p, k0, target,val)  print("alpha",alpha)  p += alpha \* np.array(dp) |

Once the cost becomes less than the threshold value, the algorithm stops generating trajectory and gets out of the loop and starts to plot the final output and passess the trajectory points to the navigation model to control the robot.

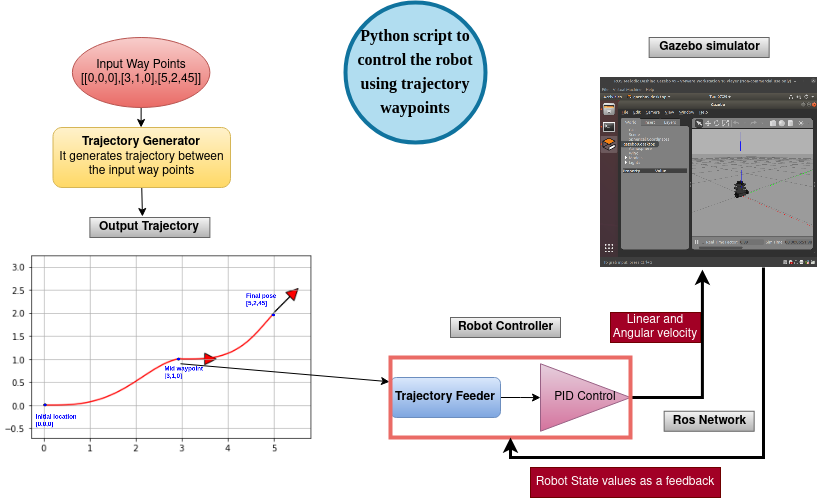
The figure-1 shows how the path has generated the waypoints [0,0,0],[3,1,0] , and [5,2,45].



*Figure-1 Trajectory generated by the model predictive trajectory generator (MPTG)*

**5. Architecture Diagram**

1. Trajectory has been generated from the model predictive trajectory generator script,
2. The generated trajectory is send as a waypoints to the robot controller code
3. The robot controller python script uses a PID control algorithm to generate the linear and angular velocity based on the reference trajectory and robot odometry.
4. The generated velocities are transferred to virtual machine running gazebo simulator using ROS network
5. The robot's movements are visualized in the gazebo simulator.

****

**6. Move\_function in waypoint navigator**

The move function is called from the main function at model\_predictive\_trajectory\_generator.py. with the input arguments as generated trajectory points. The move function resides in the waypoint\_nav.py model and that calls the turtlebot\_move function to move the robot in the gazebo simulator.

| def move\_function(waypoints):  try:  turtlebot\_move(waypoints)  except rospy.ROSInterruptException:  rospy.loginfo("Action terminated.") |
| --- |

The move function calls move\_to\_point function by sending each trajectory waypoints as an argument. It helps in computing theta and direction vector, then, the linear and angular velocities are calculated from it.

| def move\_to\_point(self, x, y):  # Compute orientation for angular velocity and direction #vector for linear velocity  diff\_x = x - self.x  diff\_y = y - self.y  direction\_vector = np.array([diff\_x, diff\_y])  direction\_vector = direction\_vector/sqrt(diff\_x\*diff\_x + diff\_y\*diff\_y) # normalization  theta = atan2(diff\_y, diff\_x) |
| --- |

Move\_to\_point calls the update function to calculate the velocity values using the theta and direction vector. **The P\_value, I\_value, and D\_value** are added together and they are used as control signals to control the robot.

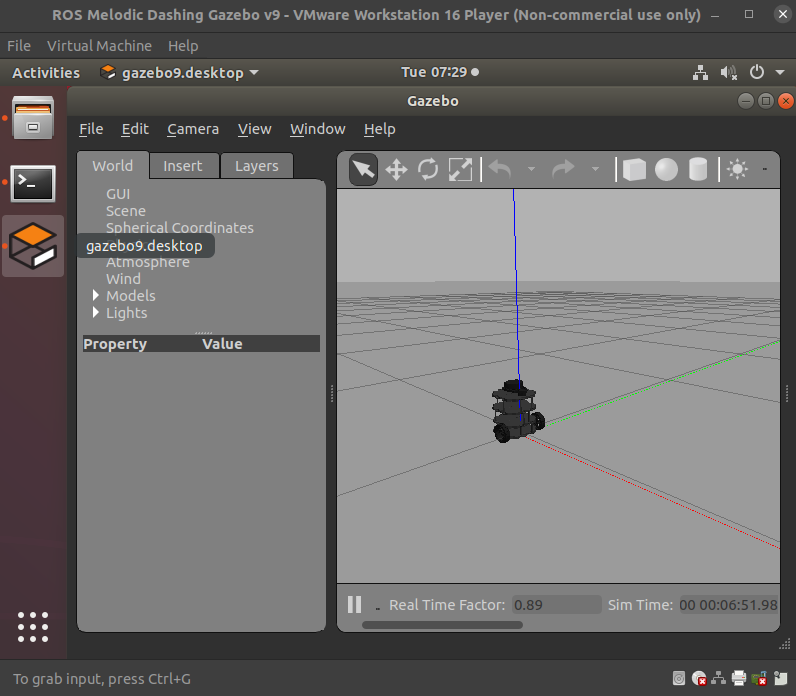
| def update(self, current\_value):  self.error = self.set\_point - current\_value  if self.error > pi: self.error = self.error - 2\*pi  elif self.error < -pi:  self.error = self.error + 2\*pi  self.P\_value = self.Kp \* self.error  self.D\_value = self.Kd \* ( self.error - self.Derivator)  self.I\_value = self.Integrator \* self.Ki  PID = self.P\_value + self.I\_value + self.D\_value  return PID |
| --- |

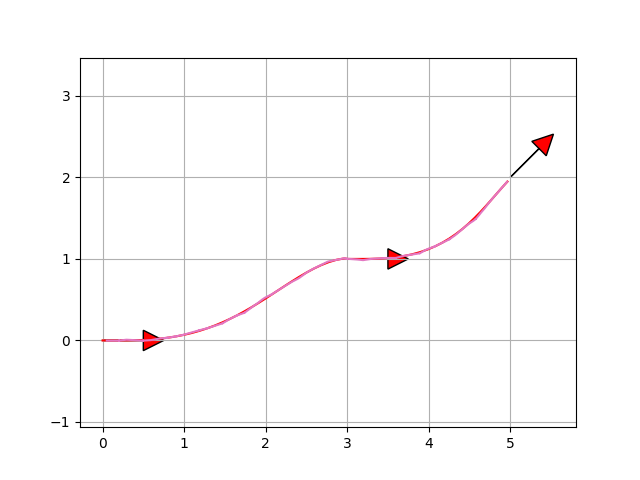
Finally the linear and angular velocity values are published through the ROS via cmd/vel topic. These signals are received by the turtlebot running on the gazebo simulator and they move in the world environment by following the commands.

| self.vel.linear.x = linear self.vel.angular.z = angular self.vel\_pub.publish(self.vel) self.rate.sleep() |
| --- |

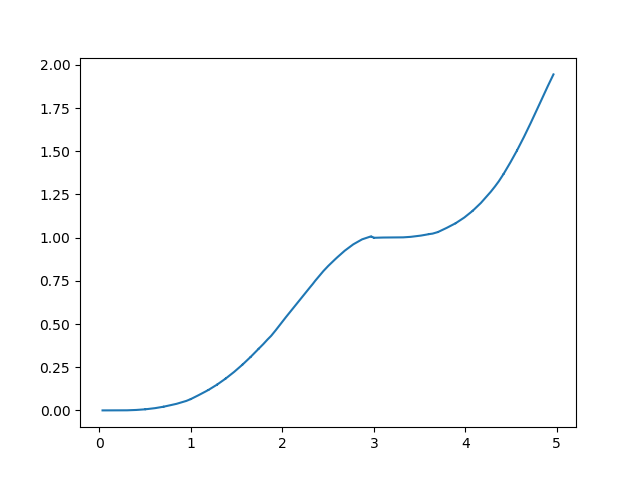
**7. Results and Discussions**

The figure-2 is the screenshot of turtlebot running in the gazebo environment by receiving commands from a python script running on a workstation PC.



*Figure-2 Turtlebot running in Gazebo environment*

*Figure-3, output trajectory of robot is infused over the trajectory generated by MPTG model*



*Figure-4 output trajectory of the turtlebot*

Figure 3 shows both the trajectory generated by the MPTG model and the trajectory of the turtlebot plotted by receiving the odometry values through a callback function from the turtlebot. Figure-5 shows the output of the odometry values . Based on the output, it is evident that the robot has followed the same trajectory which is generated by the MPTG model and it is stopped once it reaches the endpoint.

Figure 4 shows only the output trajectory of the robot by receiving the odometer values from the robot.

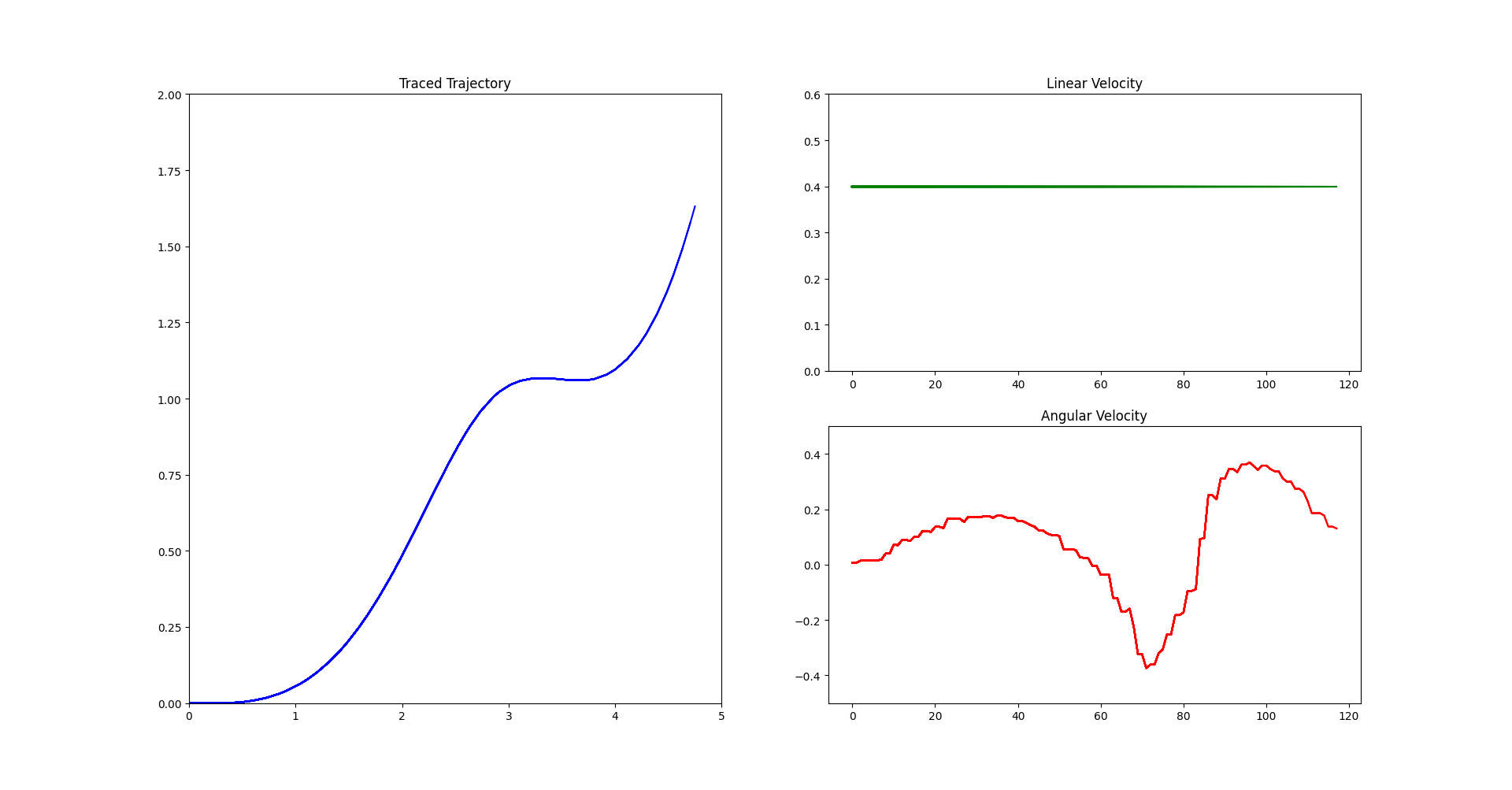
| [INFO] [1623955671.260464, 369.576000]: odom: x=4.9654277802164; y=1.9495111855553189; theta=0.7459953092290167 [INFO] [1623955671.981655, 370.236000]: odom: x=4.965446385872352; y=1.9495285080642446; theta=0.7456385564531847 [INFO] [1623955672.720293, 370.935000]: odom: x=4.965464889291778; y=1.9495457217077772; theta=0.7452836633274034 [INFO] [1623955673.428274, 371.579000]: odom: x=4.965483263345221; y=1.9495628015072373; theta=0.7449311519464924 [INFO] [1623955674.180554, 372.259000]: odom: x=4.965501563775552; y=1.9495797995068904; theta=0.7445799543889209 [INFO] [1623955674.928456, 372.939000]: odom: x=4.965519763475242; y=1.949596690709086; theta=0.7442305921496647 |
| --- |

*Figure-5 Odometer values received from the robot*

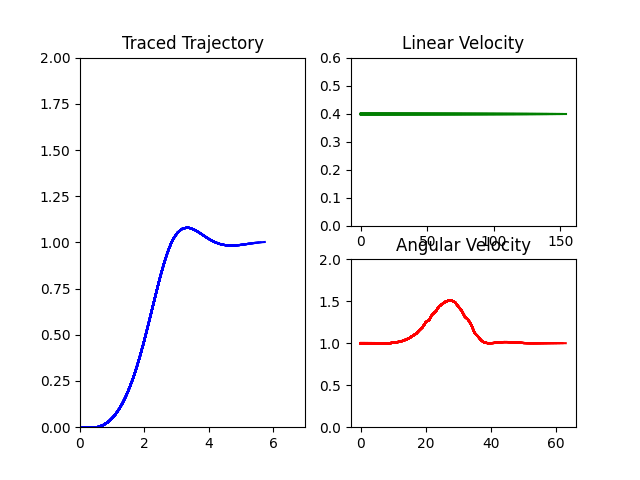
The state feedback from the turtlebot is extracted through a callback function and the output x,y state values are plotted.

In the respective subplots, the linear and angular velocity values are also plotted.

The figure 6 shows the traced trajectory, linear, and angular velocity plots, plotted in real time while the robot is moving in a gazebo environment.



*Figure-6 traced trajectory, linear, and angular velocity plots*

Figure-7 shows the output of angular velocity after differentiation of yaw value with respect to theta. By comparing it with figure-6, the angular velocity curve is smooth

*Figure -7 smoothed angular velocity curve by differentiating the yaw value*