Mobile Robotics Lab Report

Contents

Lab 1: Introduction to ROS

Task 1-2

To create the hello world package, access to the ROS commands was firstly obtained by running `echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc` followed by `source ~/.bashrc`, then the package was created by running `catkin_create_pkg hello_world rospy` in the `~/catkin_ws/src` directory. After this, boilerplate code for the `hello_world` package was created in the `~/catkin_ws/src/hello_world` directory, which has the following directory structure:

```
omar@omar-VirtualBox:~/catkin_ws/src/hello_world$ tree
.
├── CMakeLists.txt
├── launch
│   └── node_launcher.launch
├── package.xml
└── src
```

The program, which prints “HELLO WORLD - ROS”, was implemented within the `~/catkin_ws/src/hello_world/src/printer.py` file, which was first created and opened in Visual Studio Code by running `code printer.py` in the `~/catkin_ws/src/hello_world/src` directory. The code shown below was written to the file:

```
#! /usr/bin/python
import rospy
rospy.init_node('printer_node')
print '\n\n HELLO WORLD - ROS\n\n'
```

Next, the hello_world program was compiled by first giving the `printer.py` file execution permission by running `chmod +x printer.py`, and then running `catkin_make` in the `~/catkin_ws` directory. Before running `catkin_make`, `~/catkin_ws/devel/setup.bash` was added to `~/.bashrc` so ROS can find the packages’ files on any terminal window, by running `echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc` followed by `source ~/.bashrc`.

Then, to run the `hello_world` program, `roscore` was ran in the `~/catkin_ws` directory to start the ROS core service, and another separate terminal window was opened (while the ROS core service was being ran on the other terminal window) to trigger the running of the `hello_world` package by running `rosrun hello_world printer.py` in the `~/catkin_ws` directory. Since `~/catkin_ws/devel/setup.bash` was previously added

to `~/.bashrc`, this terminal window was able to find the files and run the package successfully. After running the program, “HELLO WORLD - ROS” appeared in the terminal, as expected.

An alternate way of running the program was also explored using `roslaunch`. Firstly, the `~/catkin_ws/src/hello_world/launch` directory was created, and the `node_launcher.launch` file was created and opened in Visual Studio Code within, by running `cd ~/catkin_ws/src/hello_world/launch` followed by `code node_launcher.launch`. The code shown below was written to the file:

```
<launch>
     <node pkg="hello_world" type="printer.py" name="printer_node"
output="screen">
     </node>
</launch>
```

and then the `hello_world` package was run by running `roslaunch hello_world node_launcher.launch` (also in the new terminal window, with `roscore` running on the other). As expected, “HELLO WORLD - ROS” appeared in the terminal.

Task 1-3

To launch and run both of the publisher and subscriber nodes simultaneously, they were both added to a file named `pubsub_launcher.launch` in `~/catkin_ws/src/hello_world/launch`. Firstly, `code pubsub_launcher.launch` was run in the `~/catkin_ws/src/hello_world/launch` directory to create and open the file in Visual Studio Code, and the code shown below was written to the file:

```
<launch>
   <node pkg="hello_world" type="publisher.py" name="publisher_node"
output="screen"></node>
   <node pkg="hello_world" type="subscriber.py" name="subscriber_node"
output="screen"></node>
</launch>
```

Then, the `hello_world` package was re-compiled by running `catkin_make` in the `~/catkin_ws` directory, and the publisher and subscriber nodes of the `hello_world` package were run by running `roslaunch hello_world pubsub_launcher.launch`. Below shows the terminal output after running the publisher and subscriber nodes of the `hello_world` package simultaneously using `roslaunch`:

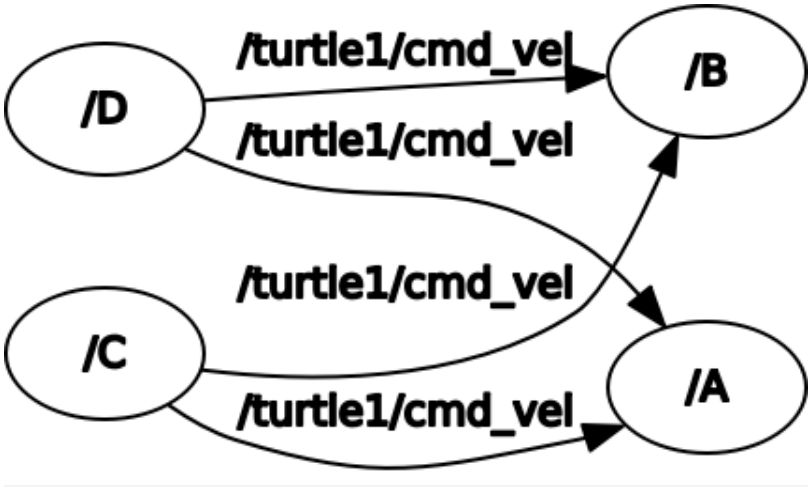
```
omar@omar-VirtualBox:~/catkin_ws$ roslaunch hello_world pubsub_launcher.launch
... logging to /home/omar/.ros/log/184df5ca-85de-11ec-a2eb-080027185f53/roslaunch-omar-VirtualBox-2623.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://omar-VirtualBox:46203/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES
  /
    publisher_node (hello_world/publisher.py)
    subscriber_node (hello_world/subscriber.py)

ROS_MASTER_URI=http://localhost:11311

process[publisher_node-1]: started with pid [2640]
process[subscriber_node-2]: started with pid [2641]
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
HELLO WORLD - ROS
^C[subscriber_node-2] killing on exit
[publisher_node-1] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

Note: CTRL+C was pressed after a short amount of time to stop the program, which otherwise would have printed “HELLO WORLD - ROS” to the terminal indefinitely, and `roscore` was running in a separate terminal window.

Lab 2: Simulation with ROS

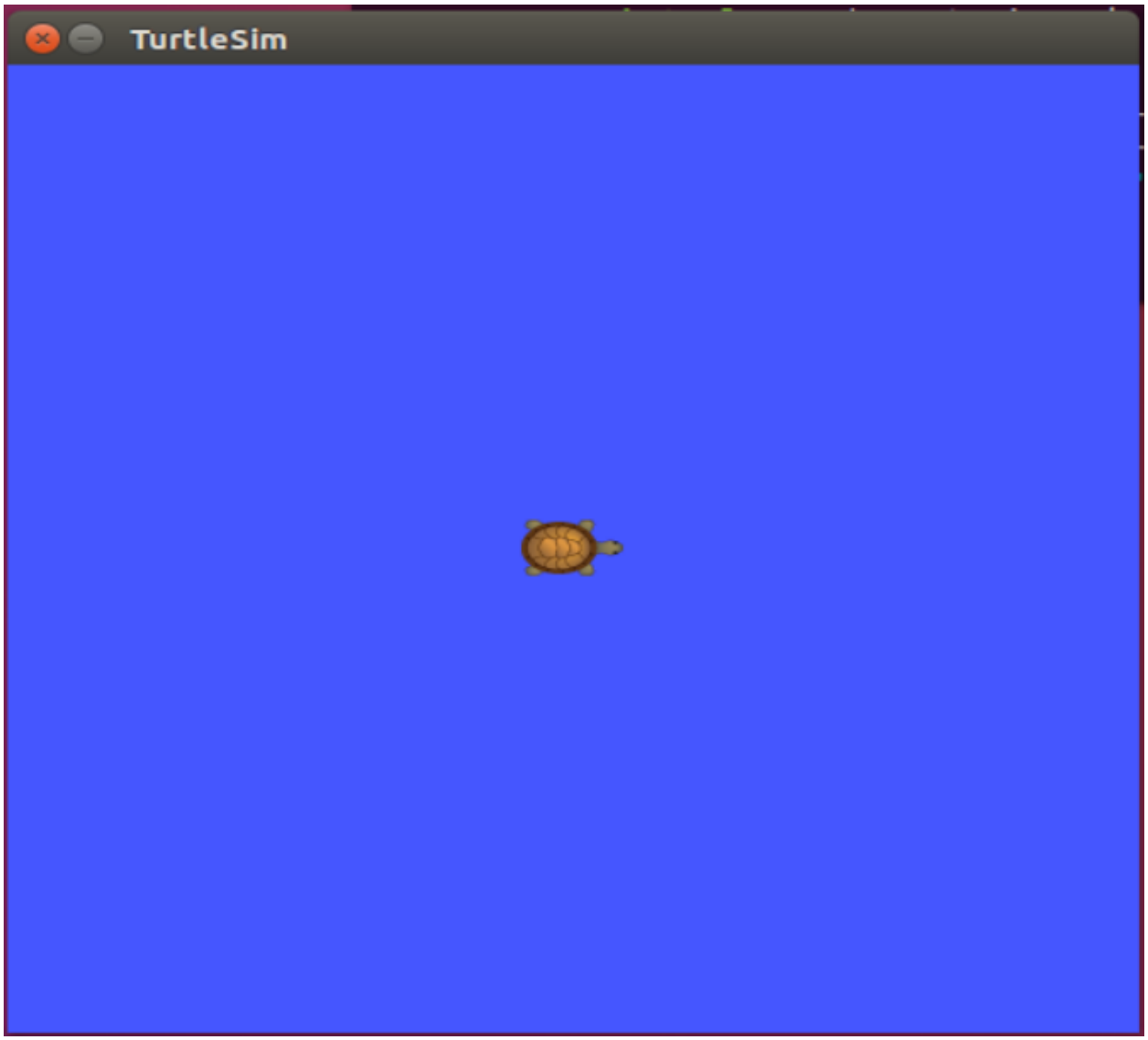
Task 2-1

Four nodes were created: two `turtlesim_node` nodes named A and B respectively, and two `turtle_teleop_key` nodes named C and D respectively. Below shows a visualisation of the four nodes, by running `rosrun rqt_graph rqt_graph`:

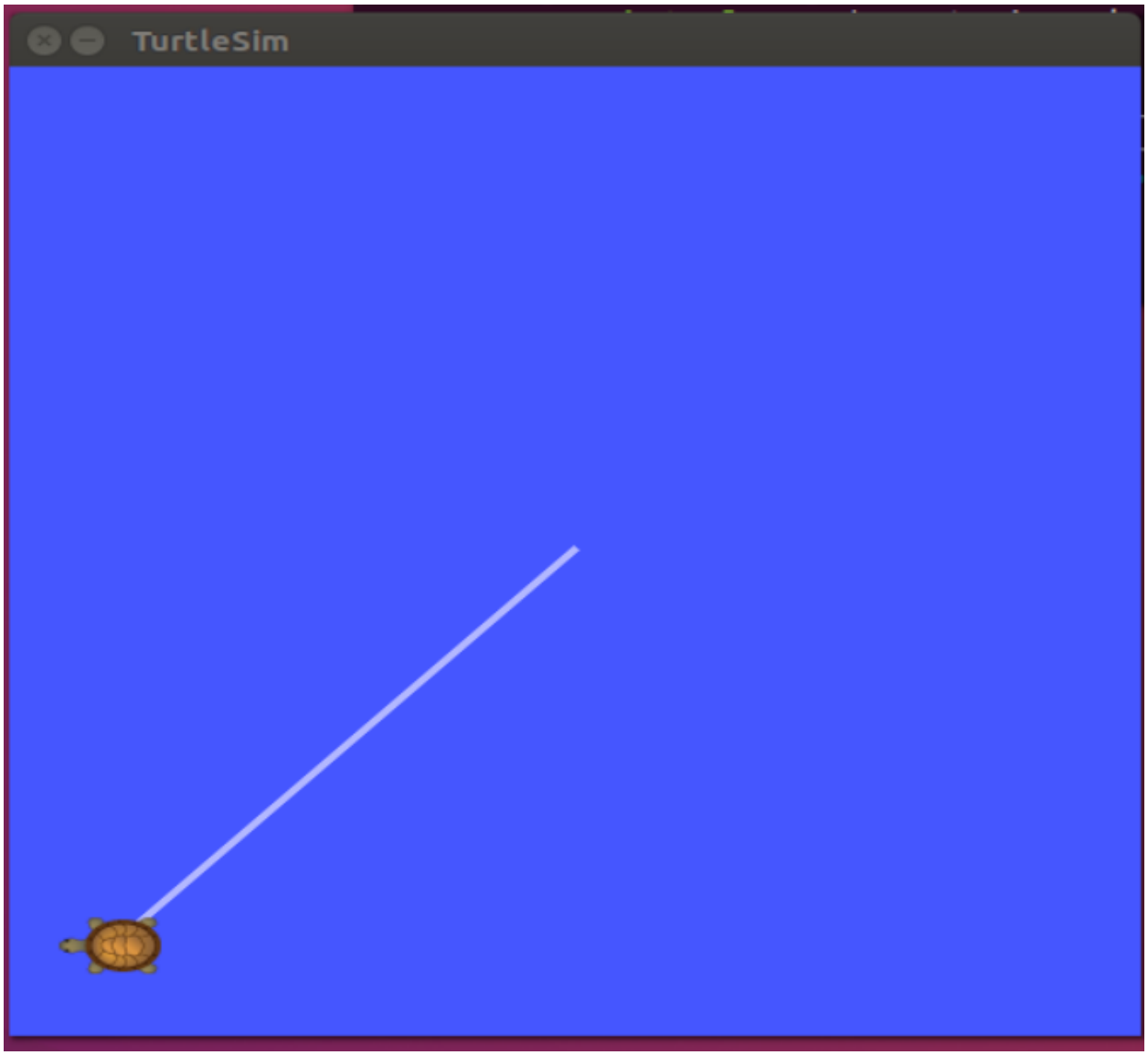


Task 2-2

A TurtleSim node was initialised by running `rosrun turtlesim turtlesim_node`. The turtle's initial position is shown below:



Then, in another terminal window, the turtle was teleported to the absolute position `(x=1, y=1)`, with a rotation `theta=pi` radians, by running `rosservice call /turtle1/teleport_absolute 1.0 1.0 3.1415`. The turtle's position after running the command is shown below:

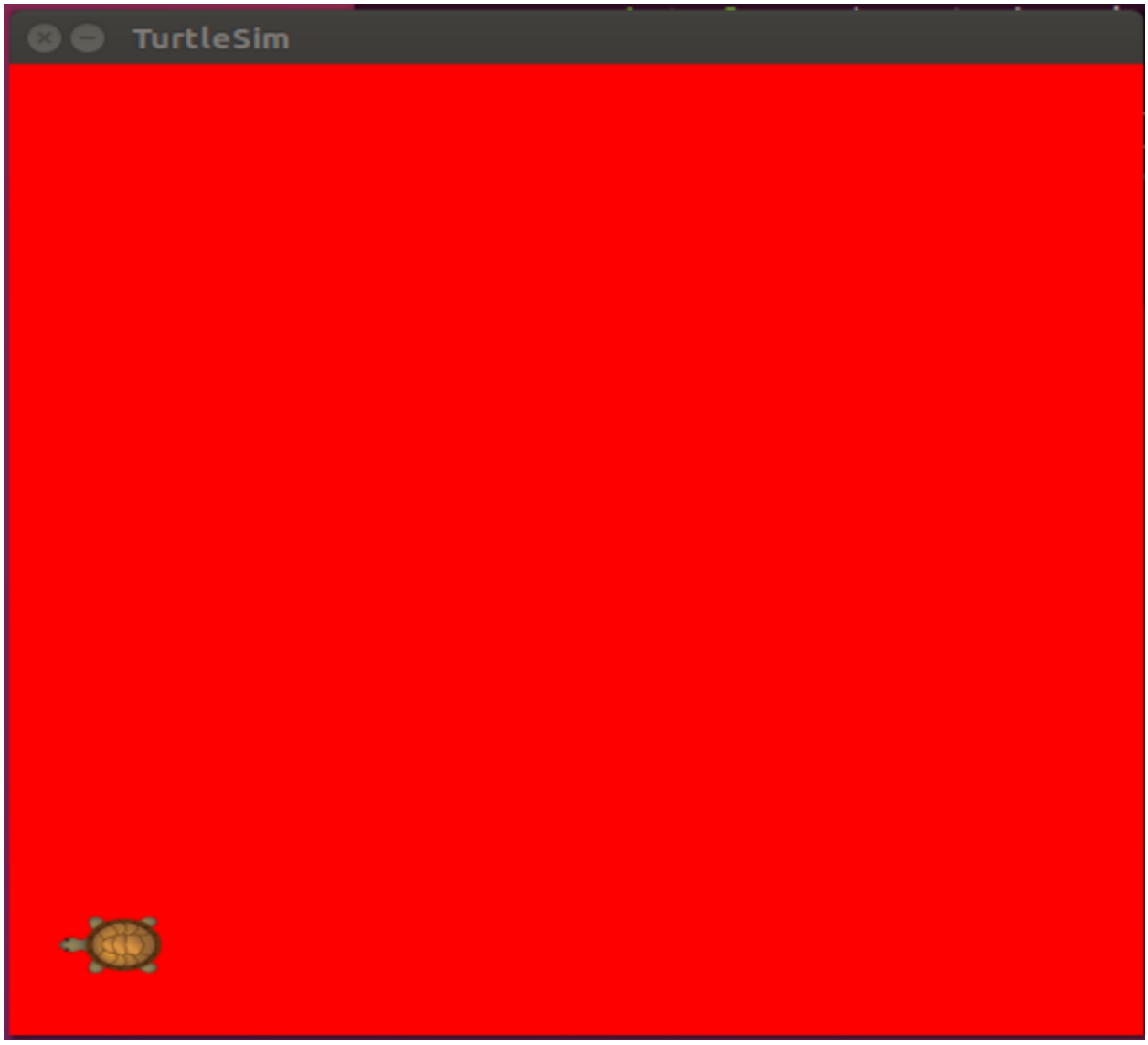


Then, the background colour was set to red by first changing background's ROS Parameters to an RGB value of `(255,0,0)` by running the following commands:

```
rosparam set background_r 255
rosparam set background_g 0
rosparam set background_b 0
```

and then calling the `clear` service to make the changes happen within TurtleSim by running `rosservice call /clear`.

Below shows the TurtleSim window after updating the background colour to red:



Task 2-3

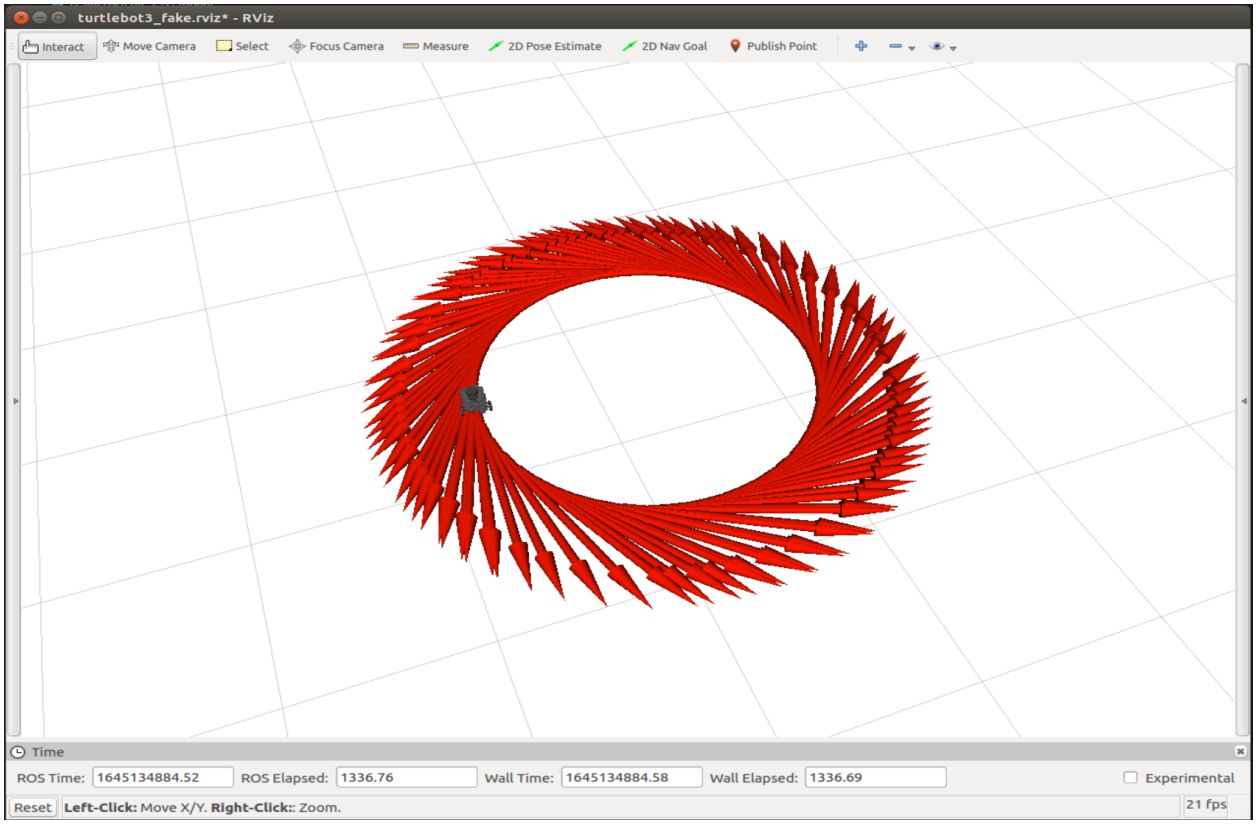
To make the Turtlebot drive in a circle, firstly a ROS Package named `turtlebot_move` was created, and a file named `trajectory.py` was created within the package under the directory `~/catkin_ws/src/turtlebot_move/src`. Within that file, the following code was entered:

```
#! /usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Vector3

def mover():
   rospy.init_node('vel_publisher')
```

```
    rospy.loginfo("Press Ctrl + C to terminate")
    pub = rospy.Publisher('cmd_vel', Twist, queue_size=10)
    move = Twist()

    # Publish at the rate of 1Hz
    rate = rospy.Rate(1)
    while not rospy.is_shutdown():
        move.linear.x = 1
        move.angular.z = 1
        pub.publish(move)
        rate.sleep()

if __name__ == '__main__':
    try:
        mover()
    except rospy.ROSInternalException:
        rospy.loginfo("Action terminated.")
```

This code publishes a ROS message of type `Twist` to the `cmd_vel` topic at a rate of 1Hz, where the `Twist` message has a linear velocity on the x-axis of 1, and an angular velocity about the z-axis of 1. This particular combination of linear and angular velocity was found to move the Turtlebot in a circle, as seen in RViz by firstly running `roslaunch turtlebot3_fake turtlebot3_fake.launch` to start and open the RViz window with the Turtlebot inside, and then running the `trajectory.py` script within the newly created `turtlebot_move` package by running `rosrun turtlebot_move trajectory.py` (after firstly giving execution permission to the `trajectory.py` file and compiling the `turtlebot_move` package).

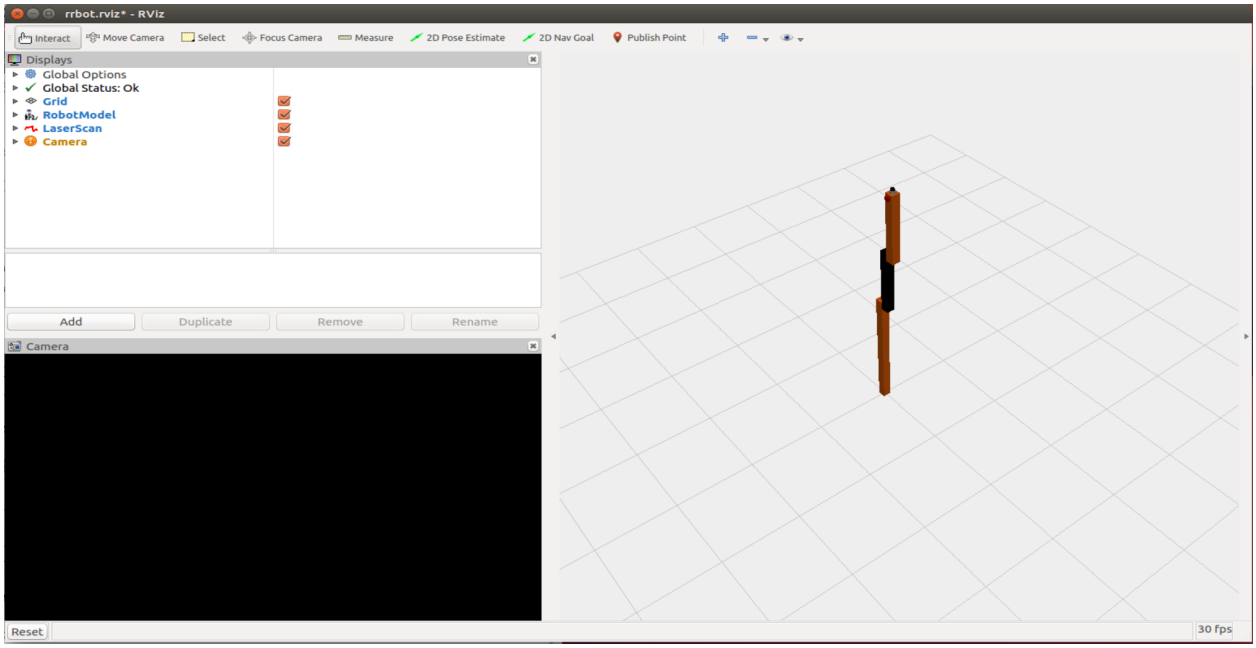
Below shows a screenshot from the RViz window while both the virtual Turtlebot in RViz and the `trajectory.py` script were running:



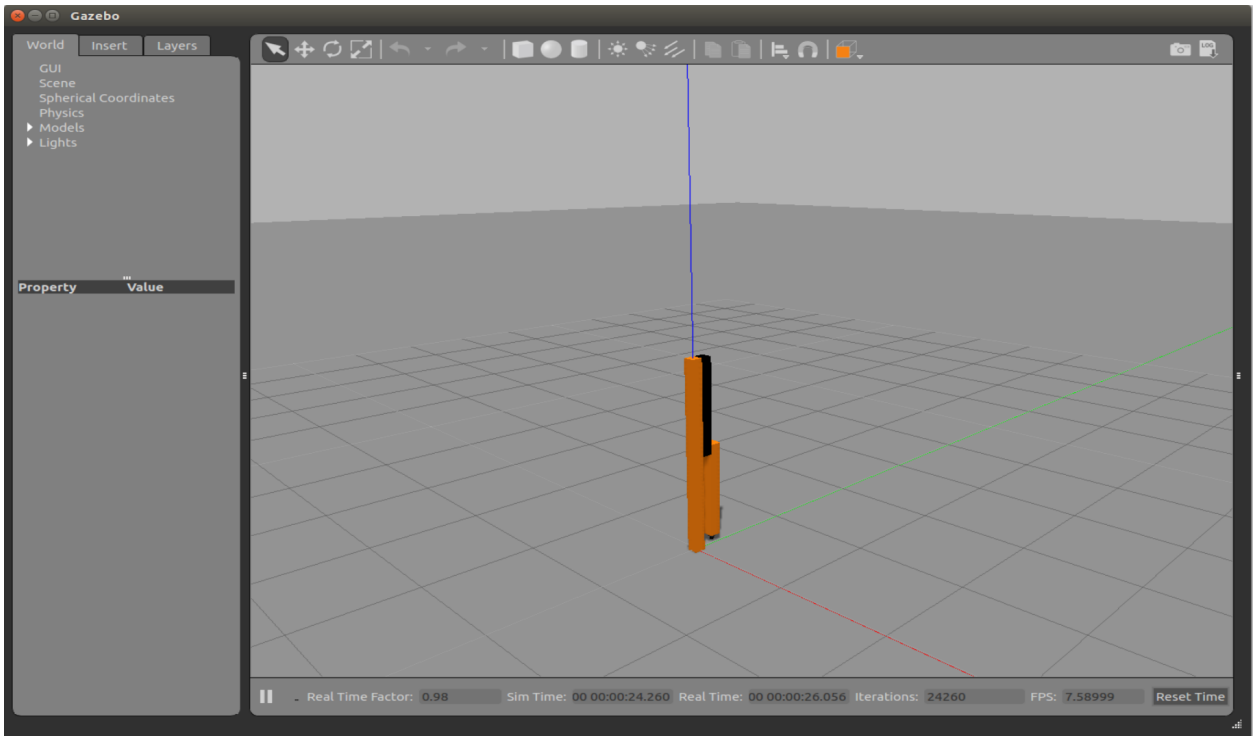
From the above screenshot, the Turtlebot can be seen to be moving in a circle when being controlled by the `trajectory.py` script.

Task 2-4

Firstly, `rrbot` was visualised in RViz by running `roslaunch rrbot_description rrbot_rviz.launch`, shown in the screenshot below:

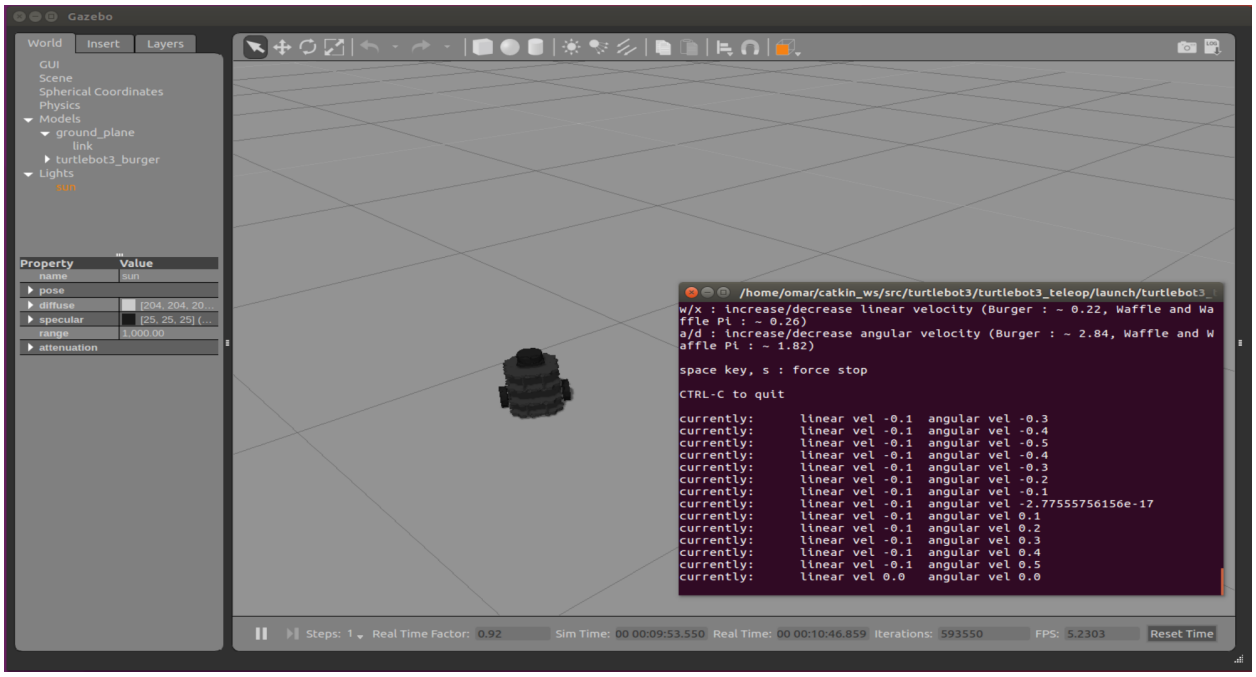


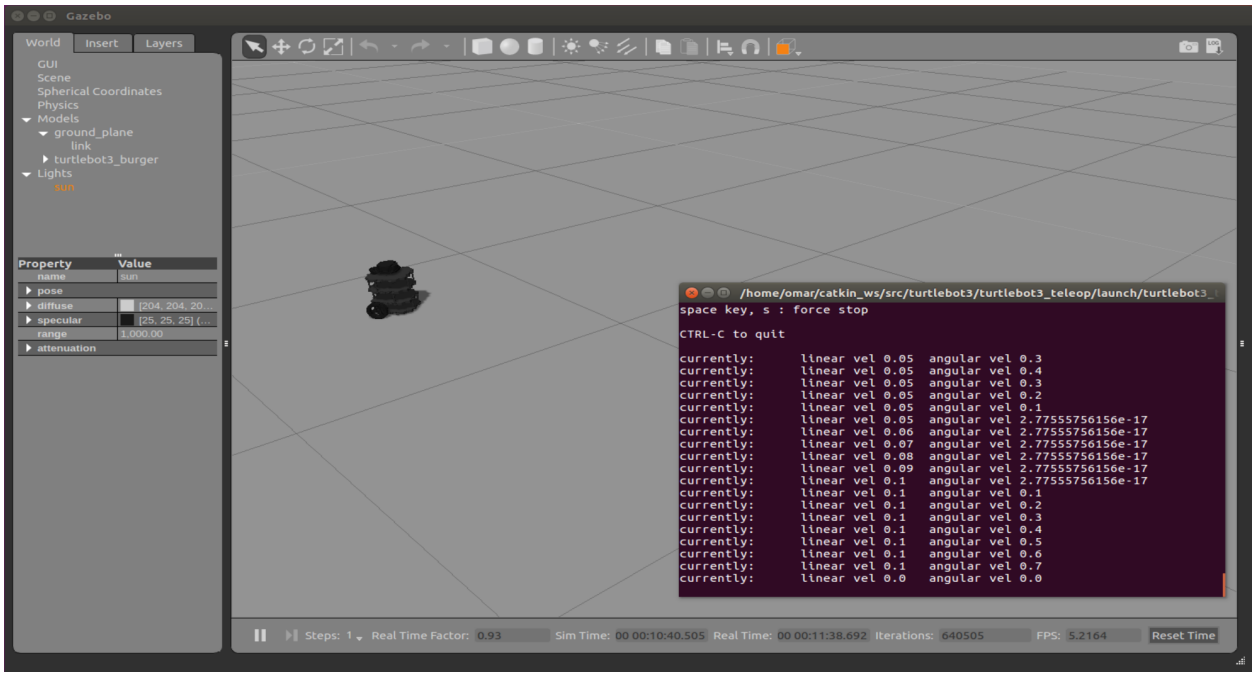
Then, `rrbot` was visualised in Gazebo by running `roslaunch rrbot_gazebo rrbot_world.launch`, shown in the screenshot below:



Next, Turtlebot was run in Gazebo within the world `empty_world` by running `roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch`. In a separate terminal window, Turtlebot3 was driven within the Gazebo window by running `roslaunch`

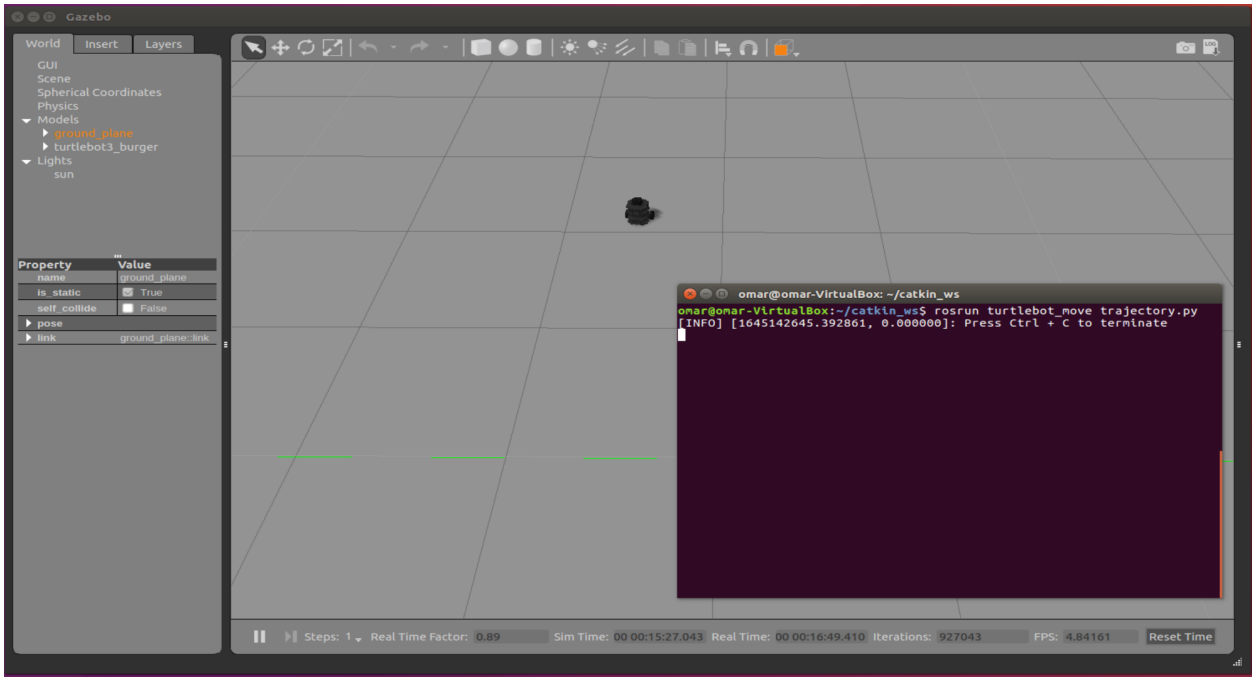
`turtlebot3_teleop turtlebot3_teleop_key.launch`. The below two screenshots show the Gazebo window and the terminal window running teleop_key simultaneously, with the Turtlebot in different locations, moved by pressing the w, a, s, d, x and spacebar keys within the `teleop_key` prompt on the terminal window.





Finally, Turtlebot was driven in a circle by the previously created `trajectory.py` script within the `turtlebot_move` ROS Package, and visualised using Gazebo within the world `empty_world`. First, the Gazebo window was opened with the Turtlebot in the `empty_world`

world by running `roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch`. Then, the `trajectory.py` script was run on a separate terminal window by running `rosrun turtlebot_move trajectory.py`. Below shows a screenshot of both the Gazebo window and the terminal window running the `trajectory.py` script simultaneously, with the Turtlebot traversing the circle.



Lab 3: Motion

Task 3-1

Firstly, a ROS Package named `lab3t3-1` was created, and a Python script named `move_square.py` with execution permission was created in the `~/catkin_ws/src/lab3t3-1/src` directory.

Then, the following code was written in the `move_square.py` file:

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from math import pi

class Turtlebot3():
    def __init__(self):
        rospy.init_node("turtlebot3_move_square")
        rospy.loginfo("Press Ctrl + C to terminate")
        self.vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=10)
        self.f_pub = 60 # Publishing frequency.
        self.spa = 5 # Seconds per action.
        self.rate = rospy.Rate(self.f_pub)
        self.run()


    def run(self):
        sides_completed = 0 # Number of sides of the square traversed.
        side_meter_count = 0 # Current number of meters traversed on the
current side of the square.
        side_num_meter = 4 # Number of meters on each side of the square.
        move = Twist()
        # Move the robot while it has not completed all of the sides of the
square.
        while not rospy.is_shutdown() and sides_completed < 4:
            if side_meter_count == side_num_meter: # At a corner.
                # Rotate by 90 degrees.
                move.linear.x = 0 # No forwards movement while rotating at
a corner.
                move.angular.z = (pi / 2) / self.spa # Rotate by pi/2
radians = 90 degrees over the duration of an action.
```

```
                side_meter_count = 0 # Reset meter count for next side.
                sides_completed += 1
            else: # Still traversing a side of a square.
                move.linear.x = 1.0 / self.spa # Move forward by 1 meter
over the duration of an action.
                move.angular.z = 0 # No rotation while traversing a side of
a square.
                side_meter_count += 1

            # Publish commands corresponding to one action at the
publishing frequency.
            # f_pub publishes takes 1 second, thus f_pub * spa publishes
takes spa seconds: the number of seconds per action.
            for _ in range(0, self.f_pub * self.spa):
                self.vel_pub.publish(move)
                self.rate.sleep()

        # Completed all of the sides of the square, thus stop rotating the
robot (cleaning up)
        move.angular.z = 0
        self.vel_pub.publish(move)

if __name__ == '__main__':
    try:
        robot = Turtlebot3()
    except rospy.ROSInterruptException:
        rospy.loginfo("Action terminated.")
```

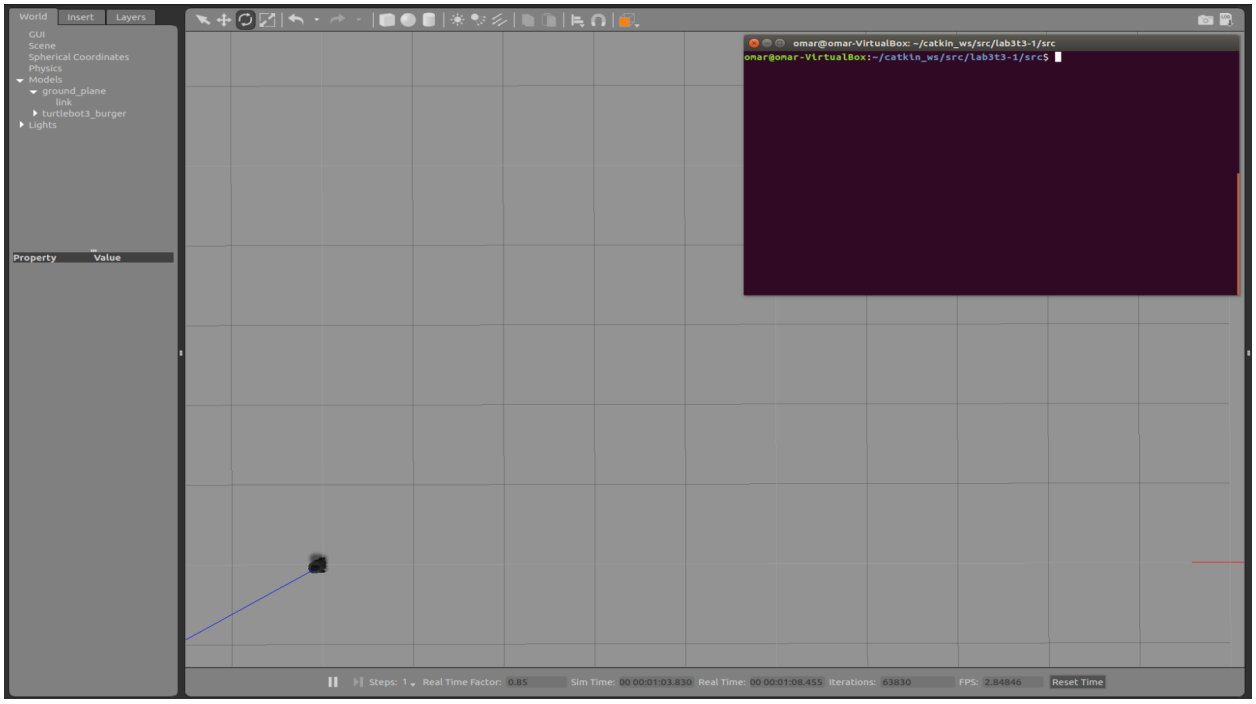
This code moves the robot in a square. The suggested organisation of the code into a class was used. ROS commands are published at a frequency of `f_pub` (which is set to 60Hz), and we divide the movements of the robot into “actions” which each take spa seconds to complete. There are two actions: move forward by 1 metre, and rotate by 90 degrees (`pi/2` radians). For each action, we divide it over each second by dividing its components `(x,y,z)` by `spa`, and then send `f_pub * spa` ROS commands for the divided action to execute it. This makes sense because the publishing rate is set to `f_pub`, `f_pub` publishes takes 1 second, thus `f_pub * spa` takes `spa` seconds: the number of seconds to complete the action. This approach was taken because ROS may not be able to handle too high of a publishing rate, but we may need to divide the actions over more intervals for higher accuracy, which can be achieved by additionally spreading the action over seconds. Thus, `spa` controls how long each action takes, and `f_pub` controls the rate of publishing, where higher values of both result in better accuracy (but at the trade-off of performance and time, respectively). It was found,

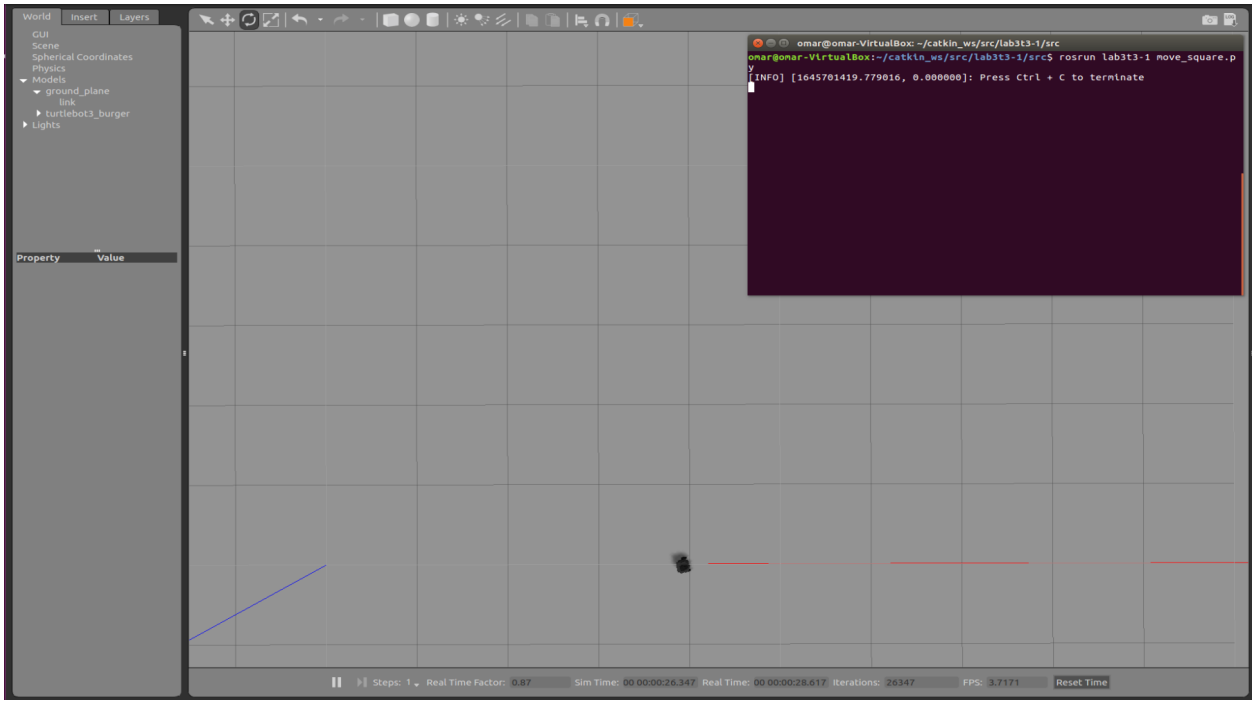
empirically, that an `f_pub` of 20Hz and a `spa` of 5 seconds were high enough to accurately execute the actions with an error less than 1 metre on each corner of the square. Higher values result in higher accuracy but at the expense of performance and time, respectively.

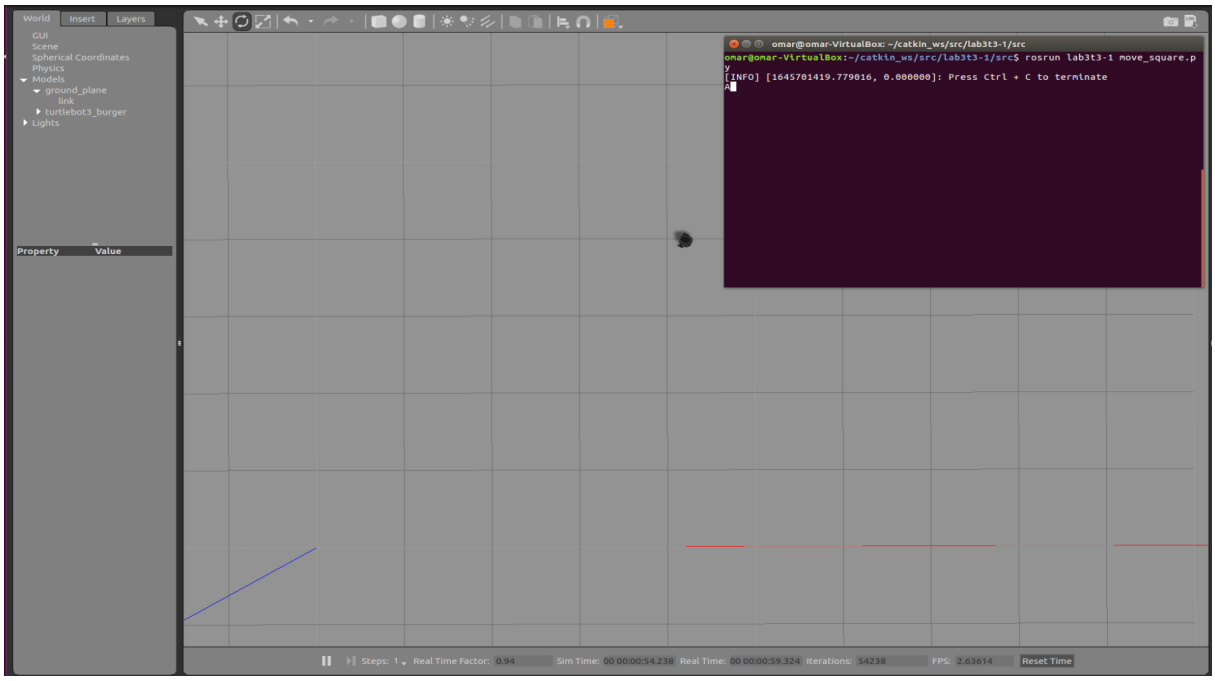
We maintain two state variables to determine the robot's progress traversing the square: one tracking how far it is along a side named `side_meter_count`, and another named `sides_completed` tracking how many sides it has traversed. When `side_meter_count` is less than the number of metres per side, the robot is still traversing a side, thus we execute the action to move forwards with a velocity of 1, and increment it. When `side_meter_count` is equal to the number of metres per side, we are at a corner, thus we execute the action to rotate the robot by 90 degrees (an angular velocity on the z axis of `pi/2` radians), increment the number of sides traversed, and reset `side_meter_count` for the next side. When `sides_completed` reaches 4, we have traversed all of the sides of the square, thus we set its rotation to 0 to halt the robot, then the script finishes.

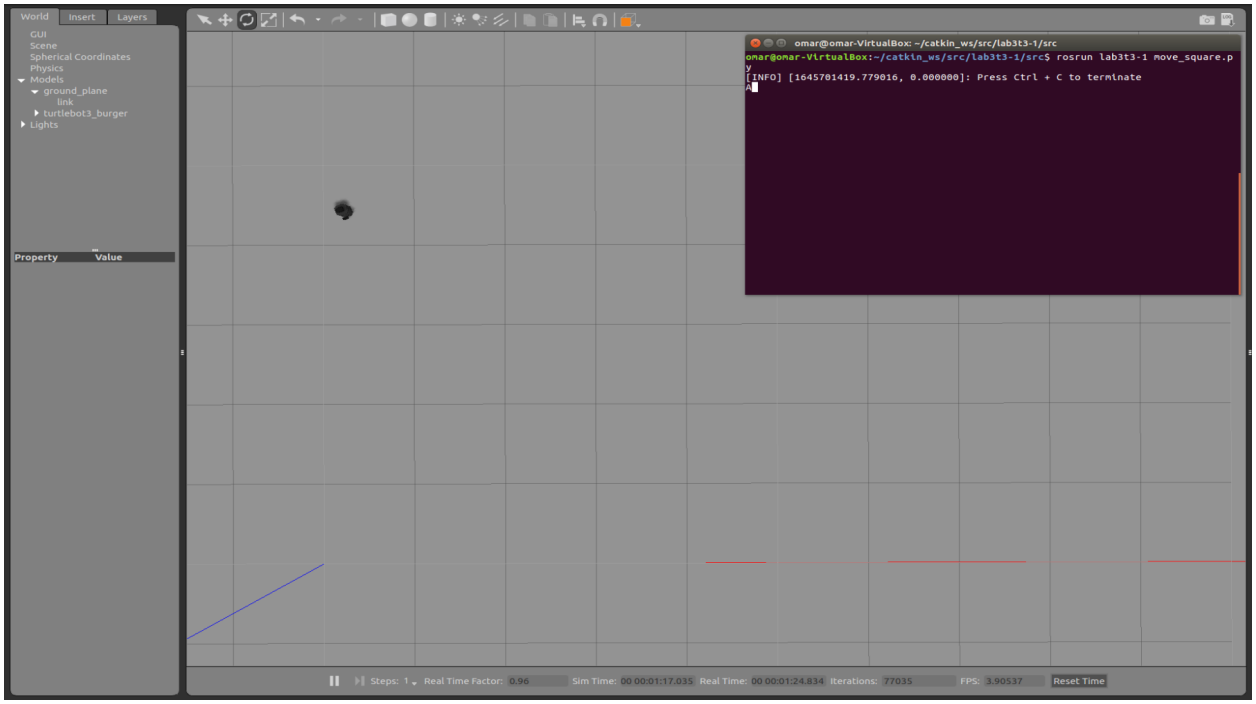
The below screenshots show the robot in the following respective positions:

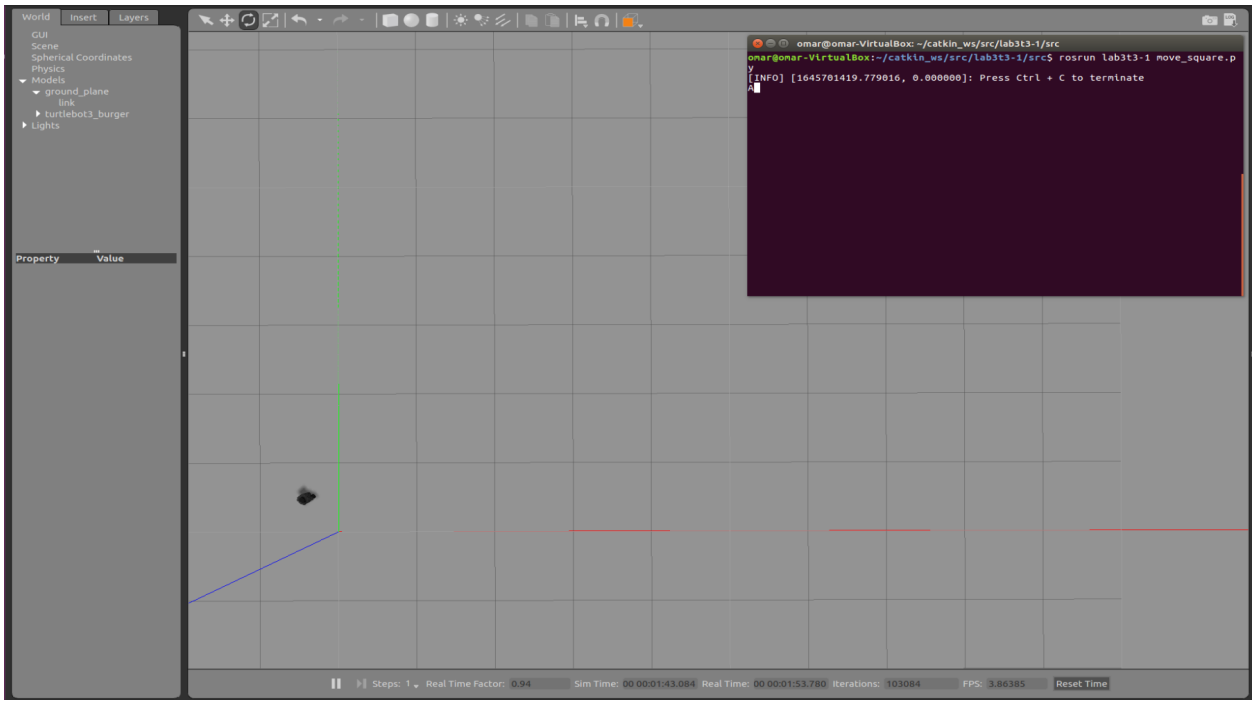
- The initial position, before executing the script.
- The first corner, after executing the script.
- The second, corner after executing the script.
- The third, corner after executing the script.
- The fourth and final corner after executing the script.




World
Insert
Layers
GUI
Scene
Spherical Coordinates
Physics
Models
ground_plane
link
turtlebot3_burger
Lights
Property
Value
omar@omar-VirtualBox: ~/catkin_ws/src/lab3t3-1/src
omar@omar-VirtualBox:~/catkin_ws/src/lab3t3-1/src$ rosrun lab3t3-1 move_square.p
y
[INFO] [1645701419.779016, 0.000000]: Press Ctrl + C to terminate
Steps: 1
Real Time Factor: 0.87
Sim Time: 00 00:00:26.347
Real Time: 00 00:00:28.617
Iterations: 26347
FPS: 3.7171
Reset Time


World
Insert
Layers
GUI
Scene
Spherical Coordinates
Physics
Models
ground_plane
link
turtlebot3_burger
Lights
Property
Value
omar@omar-VirtualBox: ~/catkin_ws/src/lab3t3-1/src
omar@omar-VirtualBox:~/catkin_ws/src/lab3t3-1/src$ rosrun lab3t3-1 move_square.p
y
[INFO] [1645701419.779016, 0.000000]: Press Ctrl + C to terminate
Steps: 1
Real Time Factor: 0.94
Sim Time: 00 00:00:54.238
Real Time: 00 00:00:59.324
Iterations: 54238
FPS: 2.63614
Reset Time



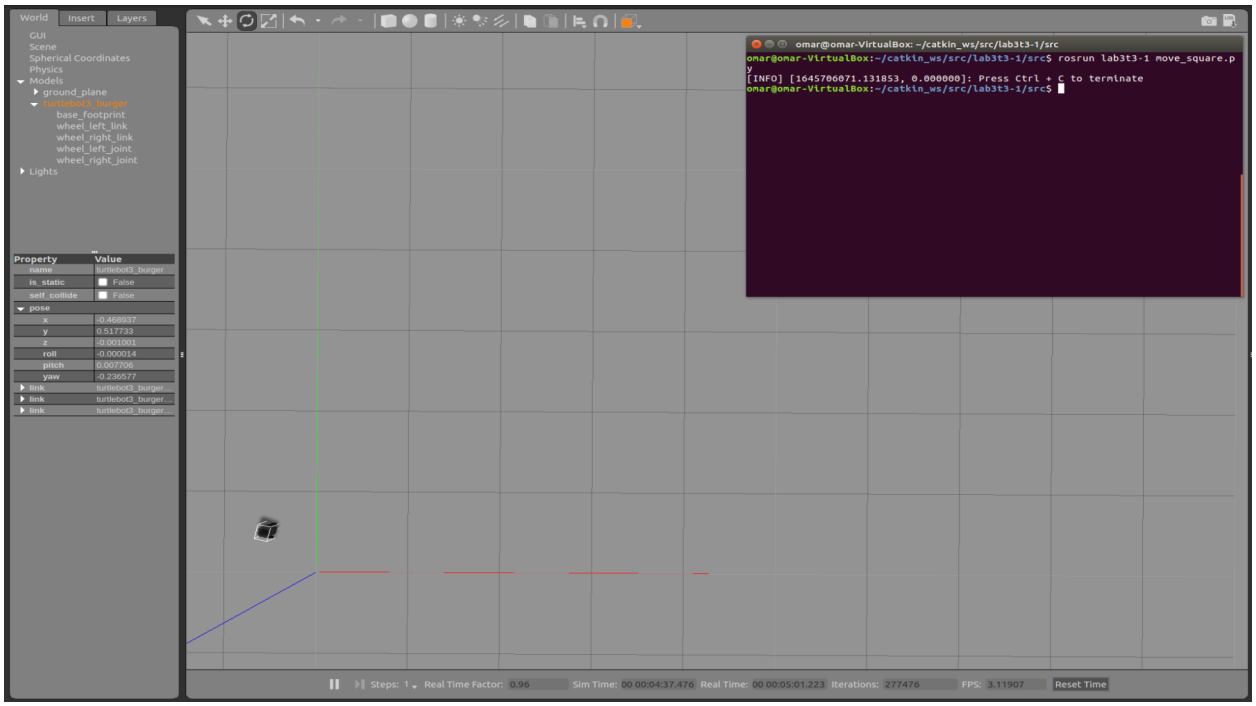


As can be seen from the screenshots, the robot reaches each checkpoint with an error less than one metre.

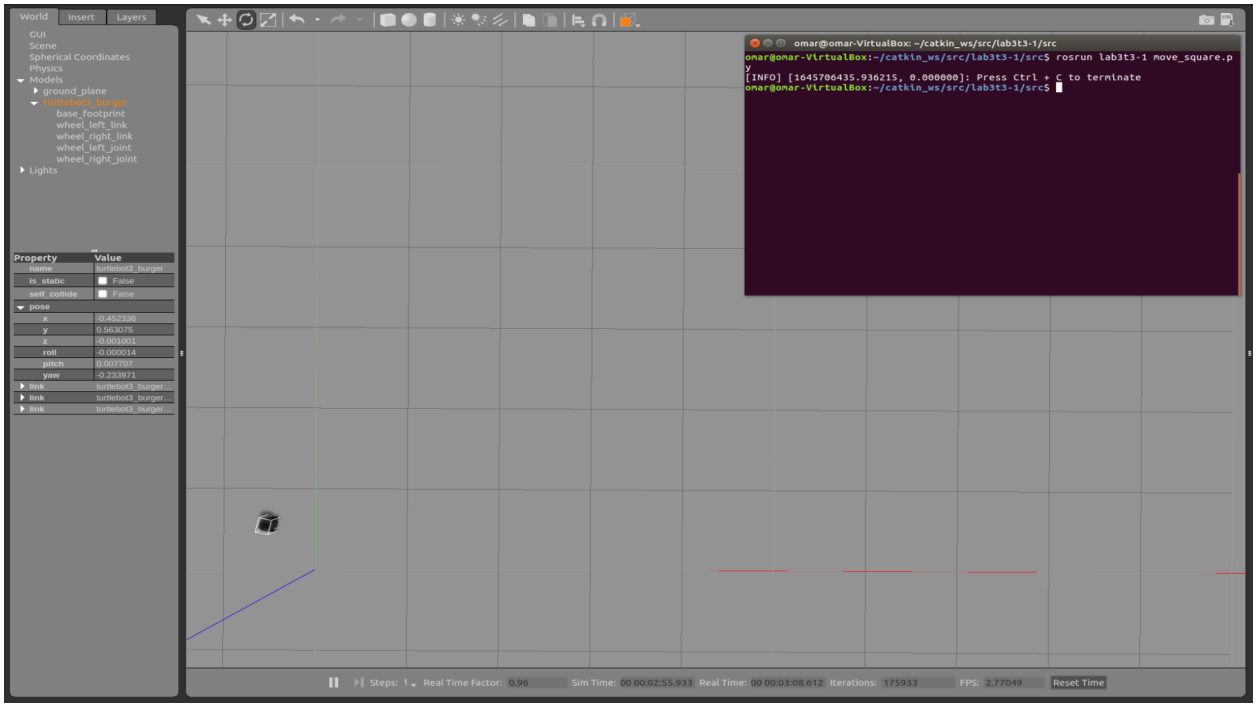
Task 3-2

After running move_square.py 10 times, the robot ended at different positions with different covariance matrices. Below we show a screenshot of each final position, and the (x,y) position of the robot after each respective run.

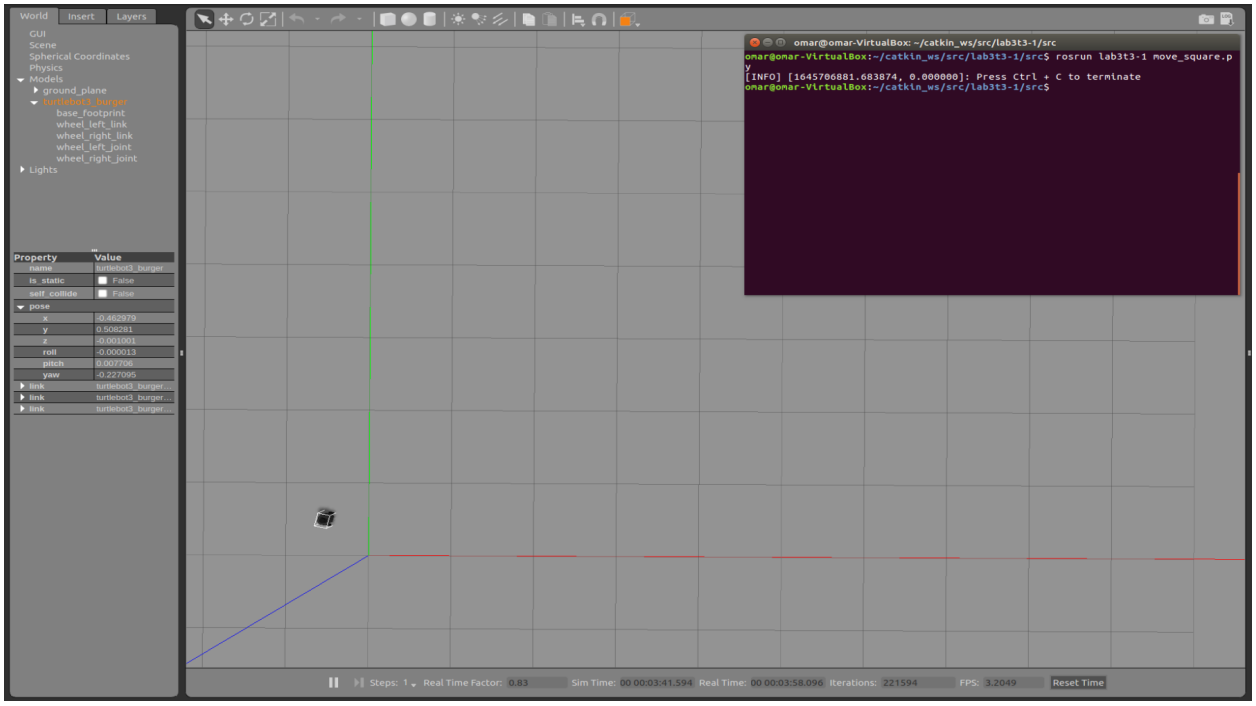
`x=-0.468937, y=0.517733`



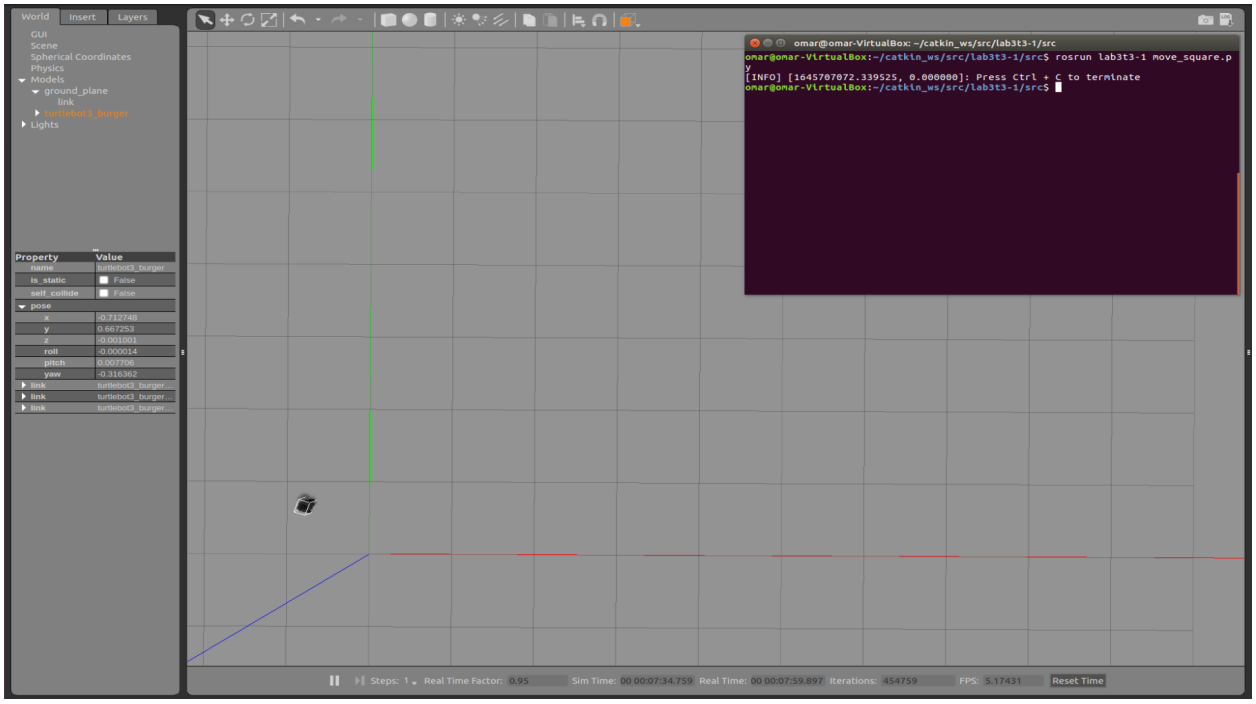
x=-0.452336, y=0.563075



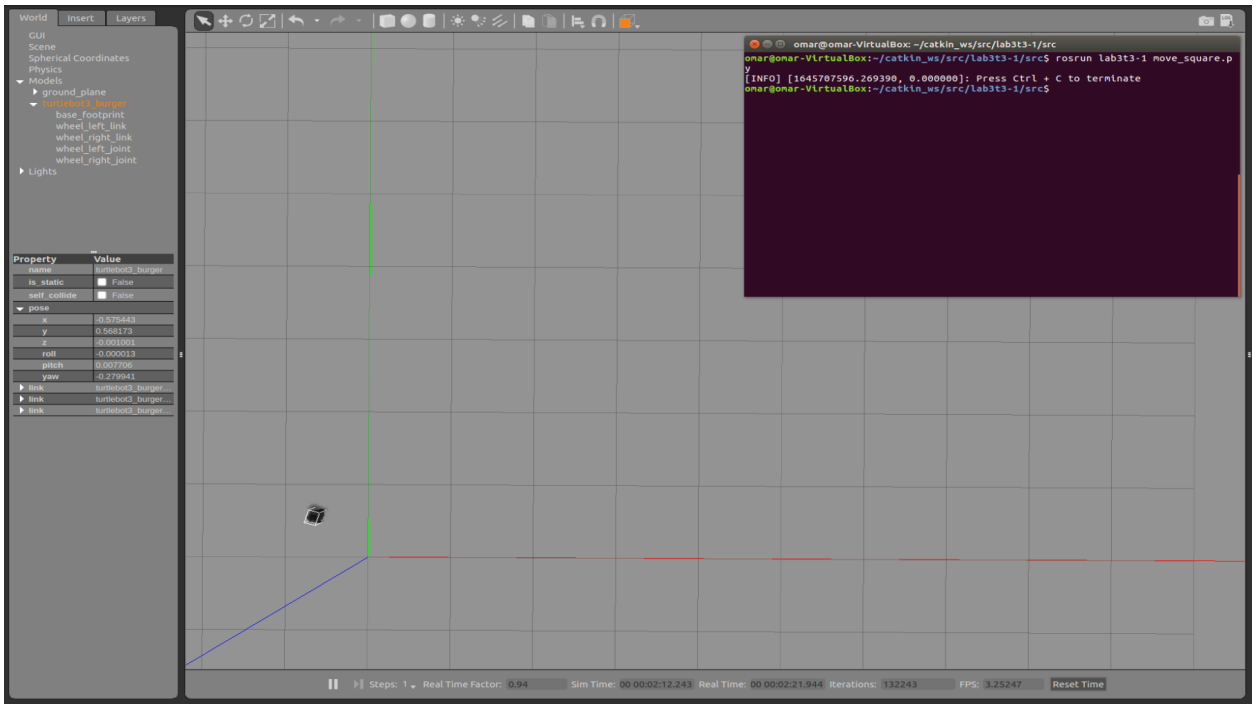
x=-0.462979, y=0.508281



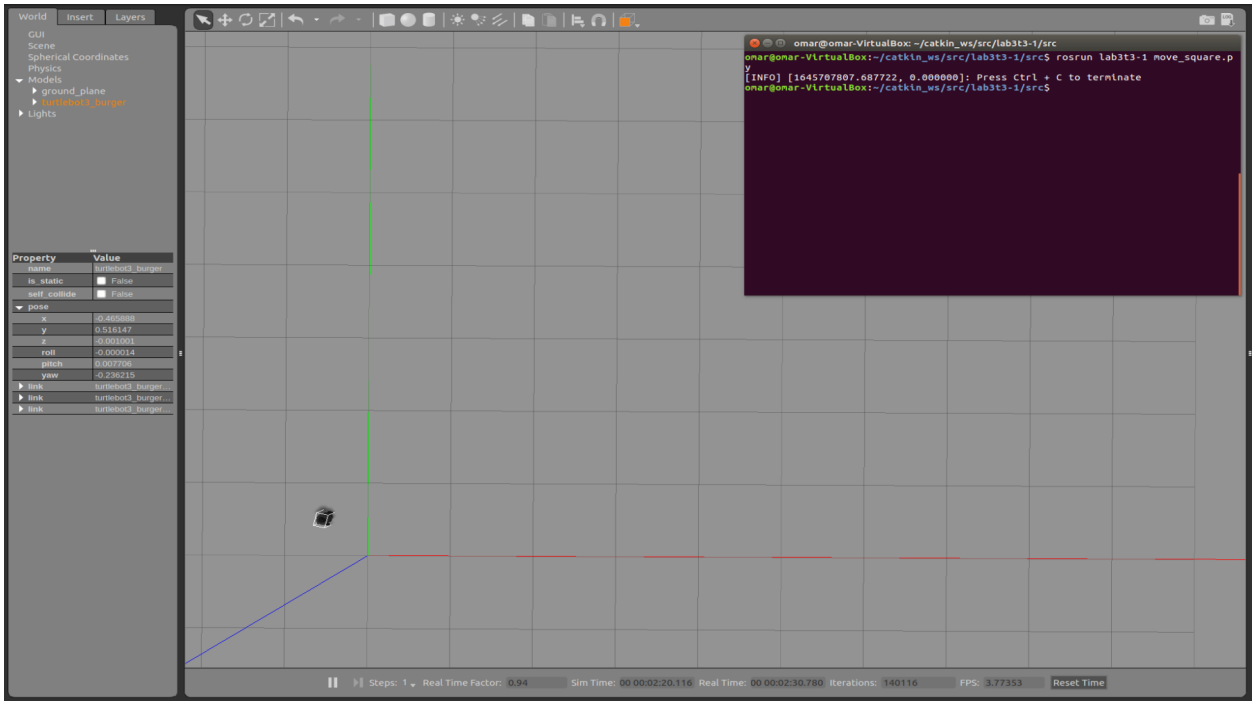
x=-0.712748, y=0.667253



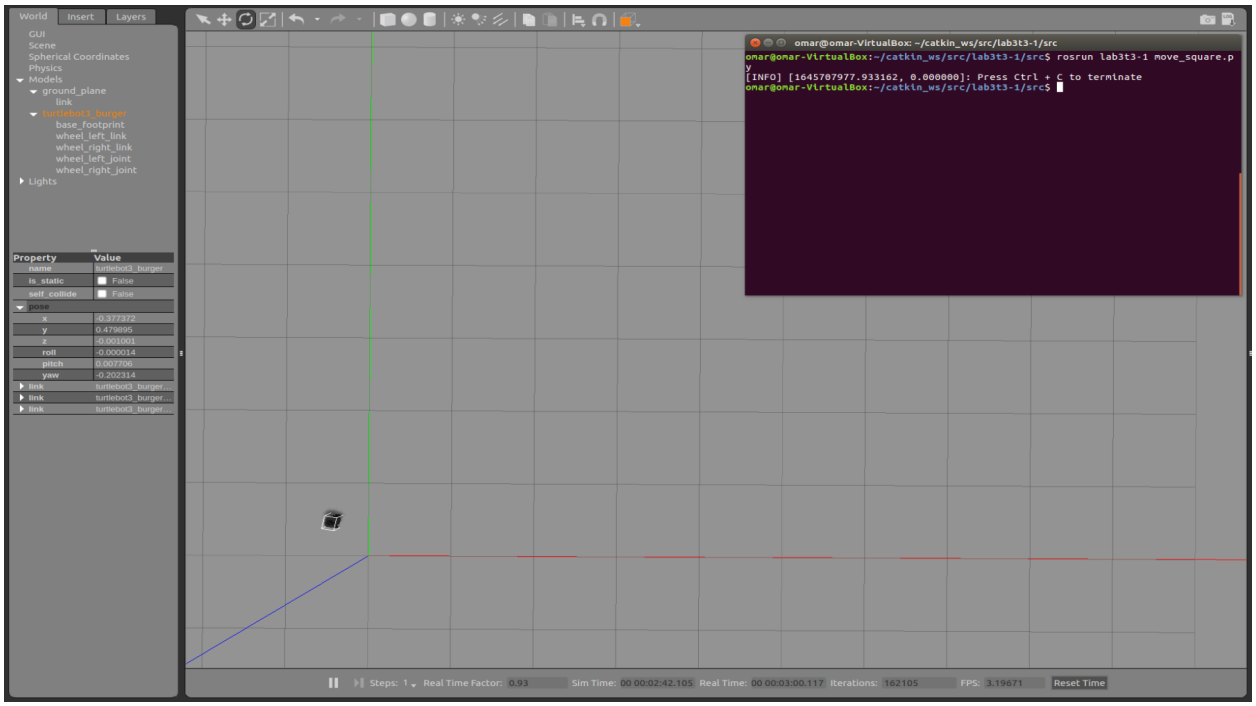
x=-0.575443, y=0.568173



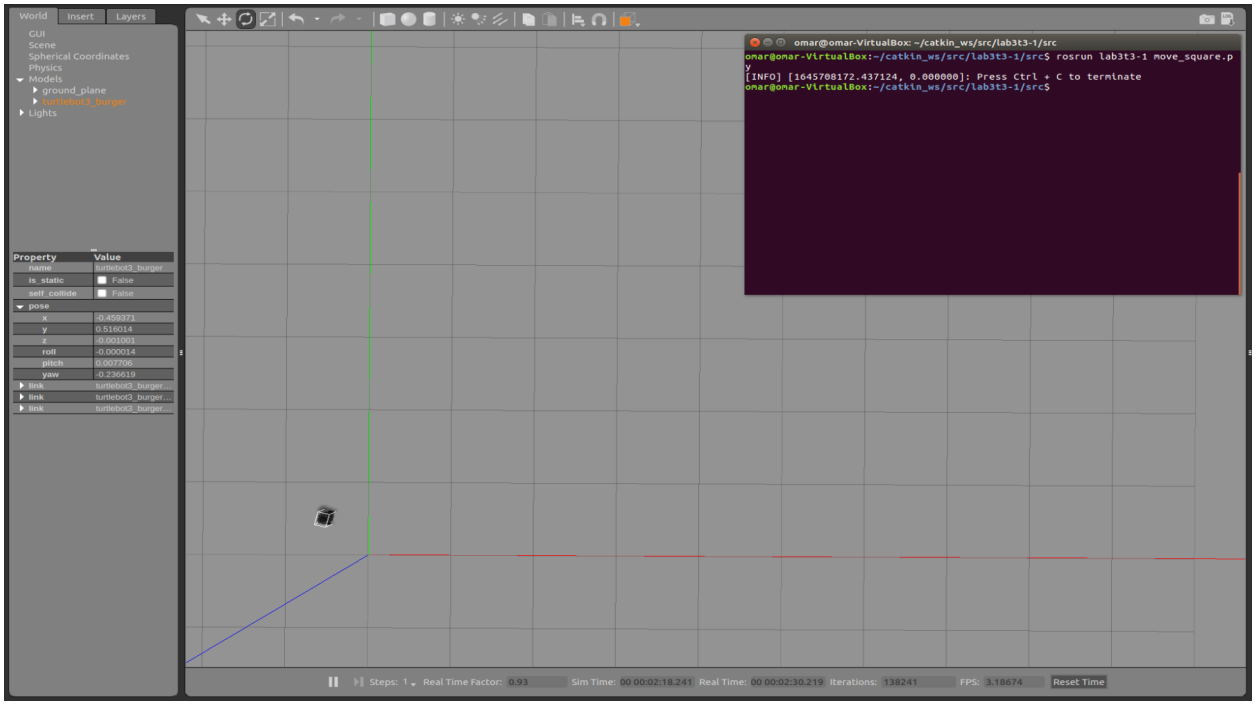
x=-0.465888 y=0.516147



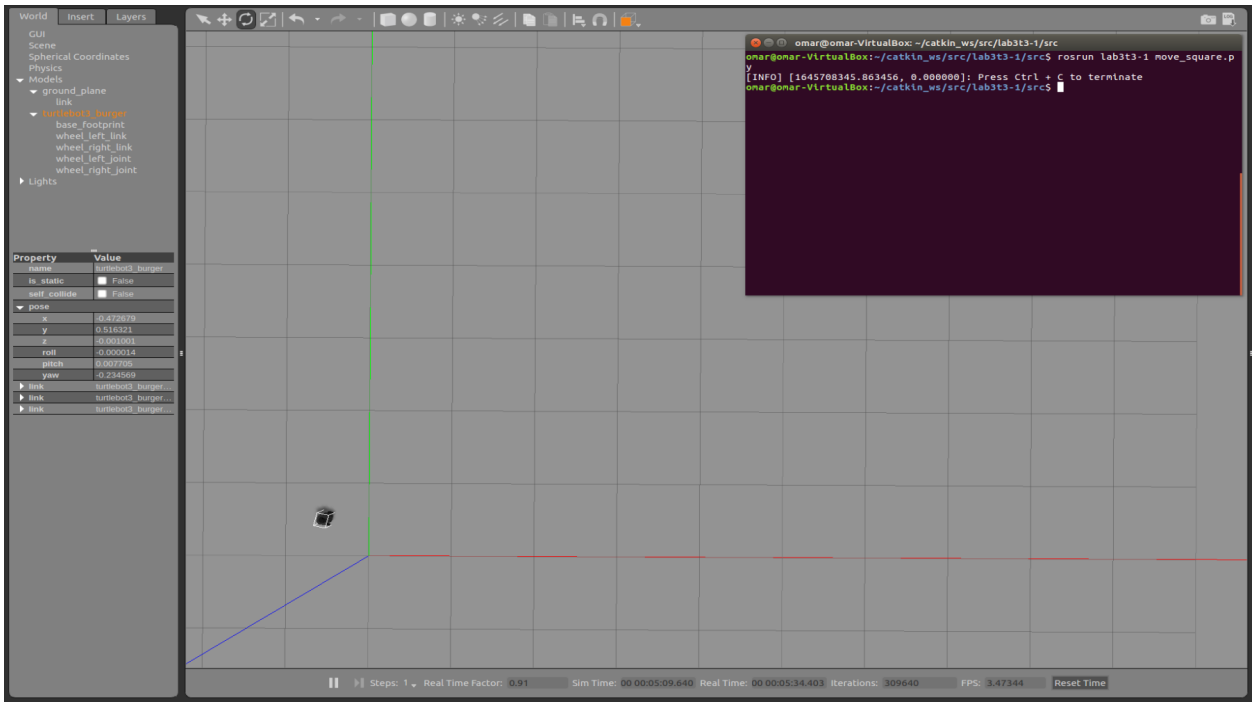
x=-0.377372 y=0.479895



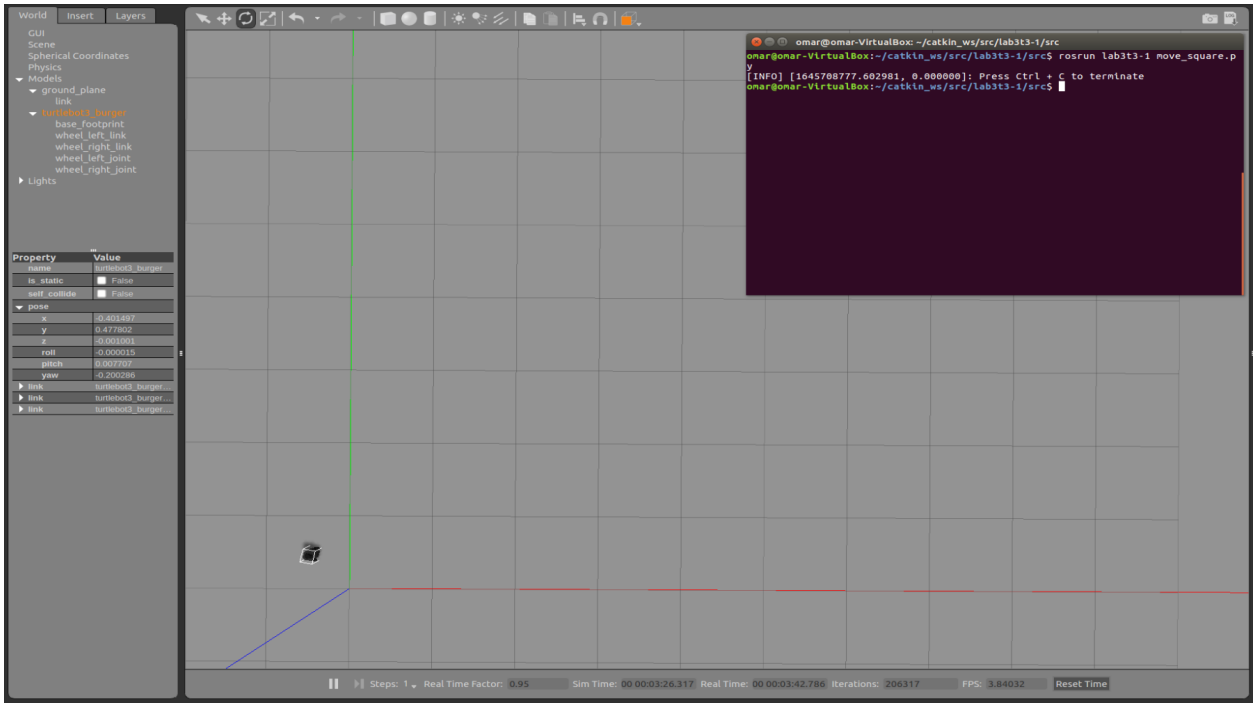
x=-0.459371 y=0.516014



x=-0.472679 y=0.516321



`x=-0.401497 y=0.477802`



A Python script was created to compute the covariance matrix, with the code as below:

```
xys = [
    [-0.468937,0.517733],
    [-0.452336,0.563075],
    [-0.462979,0.508281],
    [-0.712748,0.667253],
    [-0.575443,0.568173],
    [-0.465888,0.516147],
    [-0.377372,0.479895],
    [-0.459371,0.516014],
    [-0.472679,0.516321],
    [-0.401497,0.477802]
]

tl = 0
tr = 0
bl = 0
br = 0

x_bar = 0
y_bar = 0
```

```
for xy in xys:
   x = xy[0]
   y = xy[1]
   x_bar += x
   y_bar += y

n = len(xys)

x_bar /= n
y_bar /= n

for xy in xys:
   x = xy[0]
   y = xy[1]

   dx = x - x_bar
   dy = y - y_bar

   tl += dx ** 2
   br += dy ** 2

   tr += dx * dy
   bl += dy * dx

tl /= n
tr /= n
bl /= n
br /= n

print("tl", tl, "tr", tr, "bl", bl, "br", br)
```

In the script, `tl`, `tr`, `bl` and `br` stand for the “top left”, “top right”, “bottom left” and “bottom right” entries of the covariance matrix, respectively.

After running the script, the following was output to the terminal:
`('tl', 0.008158937142800003, 'tr', -0.004485197127600001, 'bl', -0.004485197127600001, 'br', 0.0027727272724399994).`

This output corresponds to the following covariance matrix:

```
[ 0.008158937142800003, -0.004485197127600001  ]
[-0.004485197127600001,  0.0027727272724399994 ]
```

As can be seen, the obtained covariance matrix is symmetric. The values on the diagonal show that there is some deviation/scatter in the values of the final x and y positions of the robot after traversing its square. This scatter can be reduced by reducing the motion uncertainty of the robot. As explained in Task 3-1, this can be done by increasing the publishing frequency for the velocity command and the number of seconds per action, by increasing `f_pub` and `spa` respectively in the `move_square.py` script. Such increases will respectively make the simulation more computationally expensive and longer to run, but yield higher accuracy as the commands are smaller in magnitude and spread over a longer duration, thus being less prone to physical errors, and more measurements can be made, increasing our confidence in the true location of the robot.

Lab 4: Control

Task 4-1

The PD controller was implemented with the following code:

```
class Controller:
    def __init__(self, P=0.0, D=0.0, set_point=0):
        self.Kp = P
        self.Kd = D
        self.set_point = set_point # reference (desired value)
        self.previous_error = 0

    def update(self, current_value):
        # calculate P_term and D_term
        error = self.set_point - current_value
        P_term = self.Kp * error
        D_term = self.Kd * (error - self.previous_error)
        self.previous_error = error
        return P_term + D_term

    def setPoint(self, set_point):
        self.set_point = set_point
        self.previous_error = 0

    def setPD(self, P=0.0, D=0.0):
        self.Kp = P
        self.Kd = D
```

We first compute the error `e` at the current time `t` as the reference (`r` in the figure, equal to `self.set_point` in the code) minus the control output (`y` in the figure, equal to `current_value` in the code), since in the diagram it shows that `e = r - y`.

Next, we compute k_p `* e(t)` and store it in the variable `P_term` by multiplying `self.Kp` by the previously computed error at time t (stored in the variable `error`), and we compute k_d `* (e(t) - e(t - Δt))` and store it in the variable `D_term` by subtracting the previously computed error at time `t` (stored in the variable `error`) by the error at time `t - Δt` (stored in the variable `self.previous_error`).

Finally, the computed error, `P_term` and `D_term` variables are combined appropriately (given by the code skeleton) to store the error at time `t` for the next call to update (where it will be

equal to the error at time `t - Δt`), and return the control input `u` by summing `P_term` and `D_term`.

Task 4-2

Discussion

PD Controllers for both the forward velocity (`x` component of the pose) and the rotation (`theta` component of the pose) were used to make the robot accurately drive in a square. Empirically it was found that k_p=`0.9` and k_d=`0.3` made the PD Controllers sufficiently accurate.

To overcome angle comparison issues with the angles within the `[pi,-pi]` radians range, we use `atan2(sin(x-y), cos(x-y))` to subtract angles `x` and `y`. Since `x-y` may be outside the `[pi,-pi]` radians range, we use `x-y` to define a point on the unit circle, which is at the coordinates `(cos(x-y), sin(x-y))`. Then, we obtain the signed angle to that point, by applying `atan2`. This technique was found here: https://stackoverflow.com/a/2007279. The PD Controllers accept a comparator for comparing the current state with the reference, for which `atan2(sin(x-y), cos(x-y))` is used for angles.

When the robot travels forwards along an edge of the square, a PD Controller is used to control the forward velocity of the robot, with either the x or y coordinate of the goal point used as the reference (which-ever is different from the current coordinate). We scale the linear velocity output by the PD Controller so it's not too high to cause too much uncertainty. In addition, a PD Controller is used to control the rotation of the robot, keeping it equal to the rotation of the side of the square, correcting the perturbations in rotation resulting from the forward linear velocity.

When the robot is rotating at a corner of the square, a PD Controller is used to control the angular velocity of the robot, with the `theta` component of the pose used as the reference. We scale the angular velocity output by the PD Controller so it's not too high to cause too much uncertainty. No PD Controller for the forward velocity is used here, because the forward velocity is close to zero and the angular velocity doesn't cause significant perturbations in the forward linear velocity.

Since the control input output by the controller is proportional to the error, it was found that when the robot approaches the reference, the error becomes small and thus the robot takes a long time to progress further to the reference. An improvement to my implementation to alleviate this issue could be to use `min(u,m)` as the control input output by the controller, for some capped minimum value `m > 0` (for positive control inputs `u`).

In response to the questions:

- "Which causes a larger effect in your robot, uncertainty in drive distance or rotation angle?"

    - In general, the rotation angle seems to cause a larger effect in my robot. At high linear velocities and leaving the angular velocity untouched and originally equal to 0, the rotation angle changes significantly. However, when at high angular velocities and leaving the linear velocity untouched and originally equal to 0, the linear velocity does not change much. Thus, the linear velocity appears to affect the angular velocity significantly, however the angular velocity does not appear to affect the linear velocity much.
- "Can you think of any robot designs which would be able to move more precisely?"
    - One way would be to move the robot with linear and angular velocities of smaller magnitudes to reduce the uncertainty in the robot's pose, thus enabling more precise movement.
      Another way would be to use additional PD controllers to control the robot's linear velocity, to reduce the uncertainty in the robot's linear velocity.
- "How should we go about equipping a robot to recover from the motion drift?"
    - The robot could have its linear velocity capped such that it does not exceed some threshold which causes significant motion drift.
      However, if motion drift does occur, in order to fix it during motion (and thus reduce its extent), PD controllers can be used to control the robot's linear velocity in addition to its angular velocity, correcting its pose from the motion drift. Control inputs can be obtained from the PD controllers while they are outside of some threshold acceptable error, correcting the robot such that its pose stays within the threshold error. For example, if the robot is travelling forwards and it faces some motion drift which changes the robot's rotation angle, the PD controller controlling the rotation angle can correct the change in rotation angle as it occurs, ensuring it stays within some threshold near 0.
      Since the previous approach only reduces the extent of the motion drift, in order to recover from any motion drift that has happened (even after attempting to reduce it), the robot could be separately directed to travel to its original destination from the position it arrived at after the motion drift. If the goal position of the robot is `(x*,y*)` and the robot ends up at the position `(x',y')` after its first attempt to travel to `(x*,y*)` (and `(x', y') != (x*,y*)` due to motion drift), then afterwards the robot can be separately directed to travel from `(x',y')` to `(x*,y*)`.

Code

Below shows the code used to drive the robot in a more accurate square. For modularity, the PD Controller with a comparator was implemented in a file named `pdcontroller.py`, and the ROS Controller was implemented in a file named `move_square_pd.py`.

`pdcontroller.py` code:

```
class Controller:
   def __init__(self, P=0.0, D=0.0, set_point=0, sub=(lambda a, b : a -
b)):
```

```
        self.Kp = P
        self.Kd = D
        self.set_point = set_point # reference (desired value)
        self.previous_error = 0
        self.sub = sub

    def update(self, current_value):
        # calculate P_term and D_term
        error = self.sub(self.set_point, current_value)
        P_term = self.Kp * error
        D_term = self.Kd * (error - self.previous_error)
        self.previous_error = error
        return P_term + D_term

    def setPoint(self, set_point):
        self.set_point = set_point
        self.previous_error = 0

    def setPD(self, P=0.0, D=0.0):
        self.Kp = P
        self.Kd = D
```

`move_square_pd.py` code:

```
#!/usr/bin/env python
from math import pi, sqrt, atan2, cos, sin
from xml.dom.expatbuilder import theDOMImplementation
import numpy as np

import rospy
import tf
from std_msgs.msg import Empty
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose2D

import pdcontroller

# Subtracts two angles in the range [+pi,-pi] radians
# https://stackoverflow.com/a/2007279
def sub_angles(theta1, theta2):
```

```
        delta = theta1 - theta2
        return atan2(sin(delta), cos(delta))

class Turtlebot3():
    def __init__(self):
        rospy.init_node("turtlebot3_move_square")
        rospy.loginfo("Press Ctrl + C to terminate")
        self.vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=10)
        self.rate = rospy.Rate(10)

        # subscribe to odometry
        self.pose = Pose2D()
        self.logging_counter = 0
        self.trajectory = list()
        self.odom_sub = rospy.Subscriber("odom", Odometry,
self.odom_callback)

        try:
            self.run()
        except rospy.ROSInterruptException:
            rospy.loginfo("Action terminated.")
        finally:
            # save trajectory into csv file
            np.savetxt('./trajectory.csv', np.array(self.trajectory),
fmt='%f', delimiter=',')

    def run(self):
        # add your code here to adjust your movement based on 2D pose
feedback
        angles = [pi/2,pi,-pi/2,0]
        goal_positions = [4,4,0,0]
        goal_positions_x = [True, False, True, False]

        angle = 0.0

        for i in range(4):
            goal_pos = goal_positions[i] # Goal position to travel to (x or
y coordinate, depending on `goal_pos_x`)
            goal_pos_x = goal_positions_x[i] # Whether `goal_pos` is an x
or y coordinate
```

```
            new_angle = angles[i] # Angle to rotate to after `goal_pos`

            # PD Controller controlling the forward velocity of the robot
            # Goal position is `goal_pos`
            pd_vx = pdcontroller.Controller(0.9, 0.3, goal_pos)

            # PD Controller controlling the rotation of the robot while it
travels forwards towards its goal
            # Keeps the robot's angle equal to `angle`, which is the
current angle of the robot (it should not change while travelling forward)
            # Use the `sub_angles` comparator for comparing angles
            pd_theta0 = pdcontroller.Controller(0.9, 0.3, angle,
sub_angles)

            # Difference in position of the robot to the goal, based upon
whether the goal is an x or y coordinate
            def delta_goal():
                pos = self.pose.x if goal_pos_x else self.pose.y
                return goal_pos - pos

            # Move the robot forwards while it's not within 0.05 meters of
its goal
            while abs(delta_goal()) > 0.05:
                # Get velocity from the controller, giving the current
position as the x or y component of the pose depending on `goal_pos_x`
                u_vx = pd_vx.update(self.pose.x if goal_pos_x else
self.pose.y)
                # Get the rotation from the controller, giving the theta
component of the pose
                u_theta = pd_theta0.update(self.pose.theta)
                # Goal positions are to the left and below the robot for
the last two sides, which cause the controller to output a negative
velocity.
                # Fix the velocity.
                if i >= 2: u_vx *= -1
                # Scale the obtained forwards and angular velocity from the
PD Controllers, and execute the movement command
                move = Twist()
                move.linear.x = u_vx / 10
                move.angular.z = u_theta
```

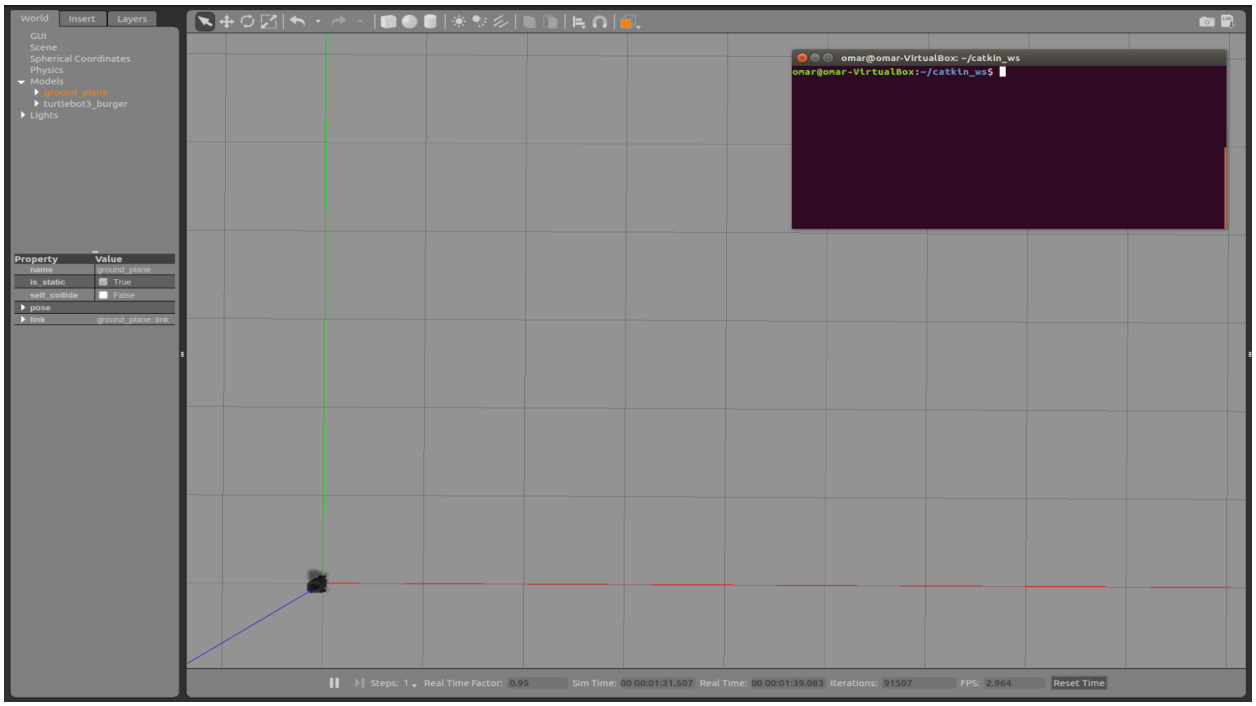
```
                self.vel_pub.publish(move)
                self.rate.sleep()

            # PD Controller controlling the rotation of the robot around
the corner
            # Goal rotation is `new_angle`
            # Use the `sub_angles` comparator for comparing angles
            pd_theta = pdcontroller.Controller(0.9,0.3, new_angle,
sub_angles)

            # Rotate the robot to face `new_angle` (while its rotation is
not within 0.05 radians of `new_angle`)
            while abs(sub_angles(self.pose.theta, new_angle)) > 0.05:
                # Get the rotation from the controller
                u_theta = pd_theta.update(self.pose.theta)
                # Scale the obtained angular velocity from the PD
Controller and execute the movement command
                move = Twist()
                move.angular.z = u_theta / 5
                self.vel_pub.publish(move)
                self.rate.sleep()

            # The robot is now facing `new_angle` (with some uncertainty)
            angle = new_angle

        # Stop robot from moving after traversing the square
        move = Twist()
        move.linear.x = 0
        move.angular.z = 0
        self.vel_pub.publish(move)
        self.rate.sleep()

    def odom_callback(self, msg):
        # get pose = (x, y, theta) from odometry topic
        quaternion =
[msg.pose.pose.orientation.x,msg.pose.pose.orientation.y,\
                    msg.pose.pose.orientation.z,
msg.pose.pose.orientation.w]
        (roll, pitch, yaw) =
tf.transformations.euler_from_quaternion(quaternion)
```

```
        self.pose.theta = yaw
        self.pose.x = msg.pose.pose.position.x
        self.pose.y = msg.pose.pose.position.y

        # logging once every 100 times (Gazebo runs at 1000Hz; we save it
at 10Hz)
        self.logging_counter += 1
        if self.logging_counter == 100:
            self.logging_counter = 0
            self.trajectory.append([self.pose.x, self.pose.y])  # save
trajectory
            rospy.loginfo("odom: x=" + str(self.pose.x) +\
                ";  y=" + str(self.pose.y) + ";  theta=" + str(yaw))

if __name__ == '__main__':
    robot = Turtlebot3()
```

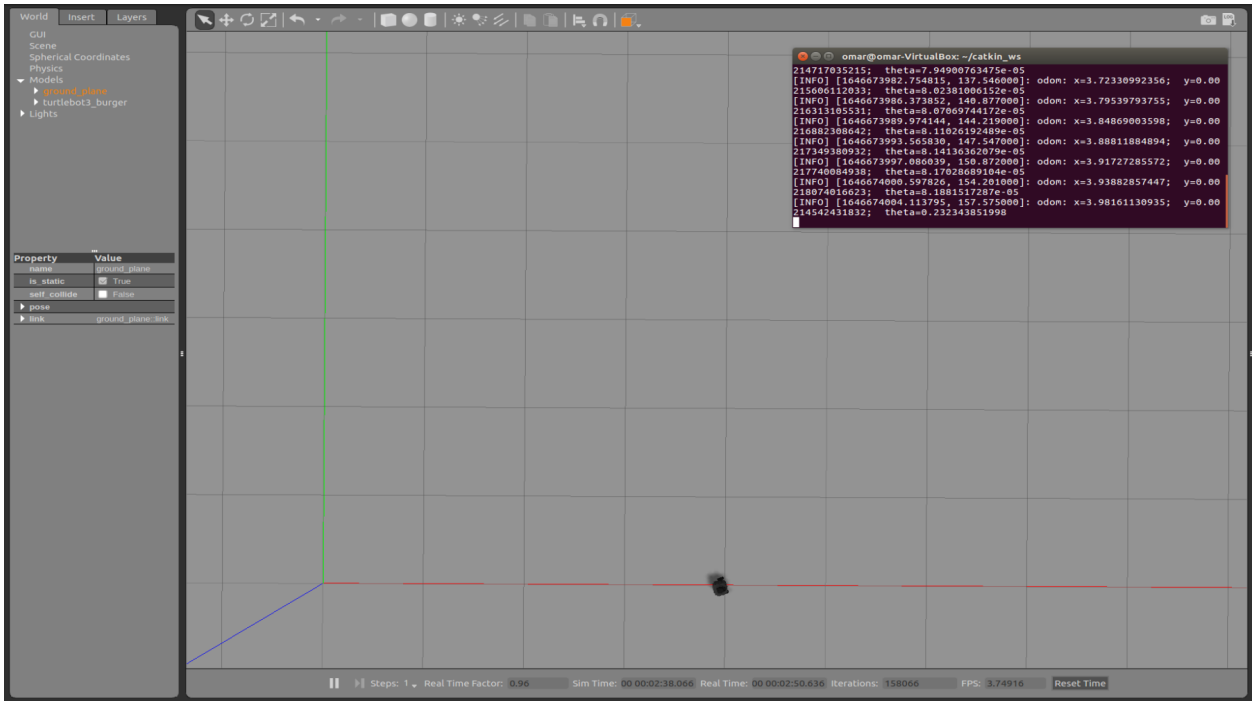
Screenshots

Below are screenshots of the robot's position at each of the four corners of the square and the plotted trajectory when being controlled using the ROS Controller with PD Control in `move_square_pd.py`:

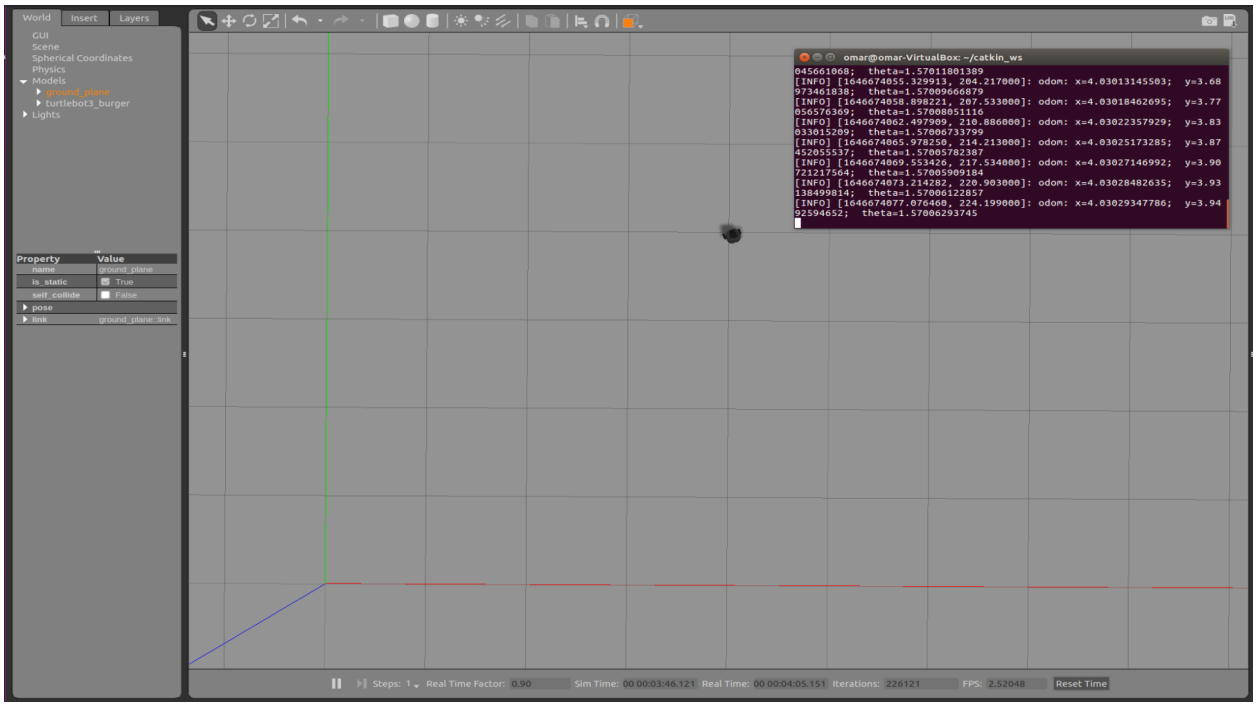
Waypoint 0 / Initial position (bottom-left)



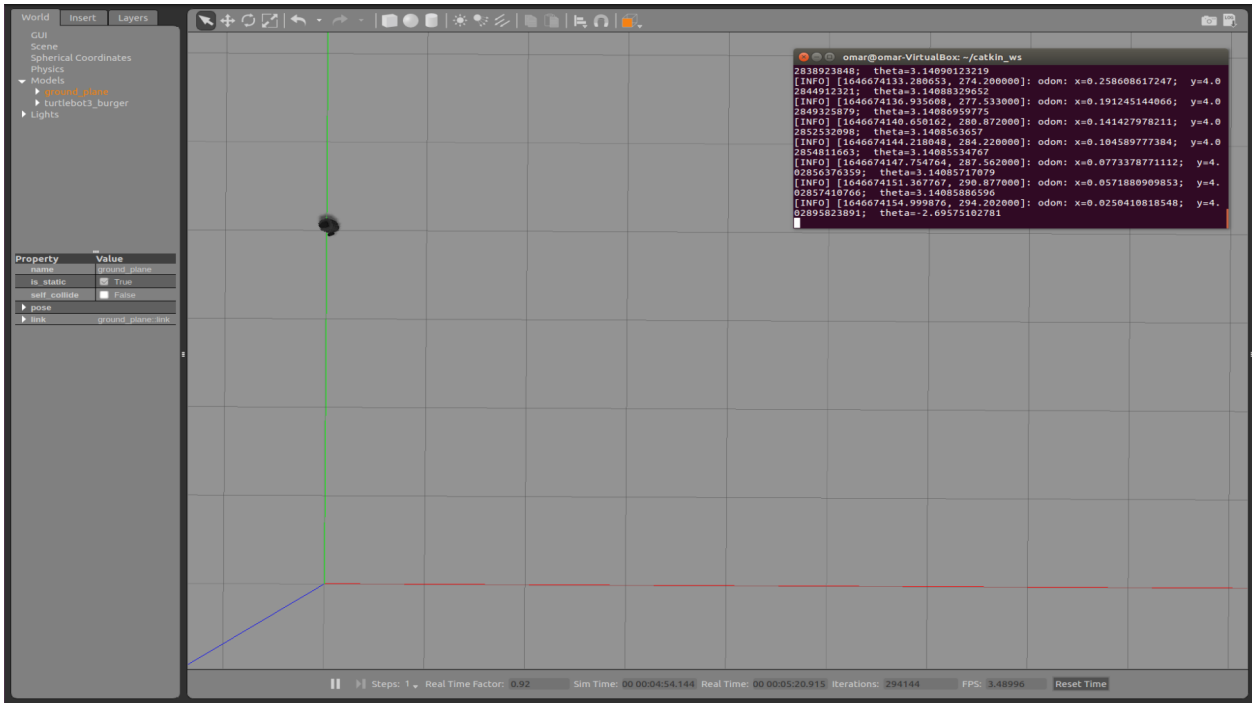
Waypoint 1 (bottom-right)



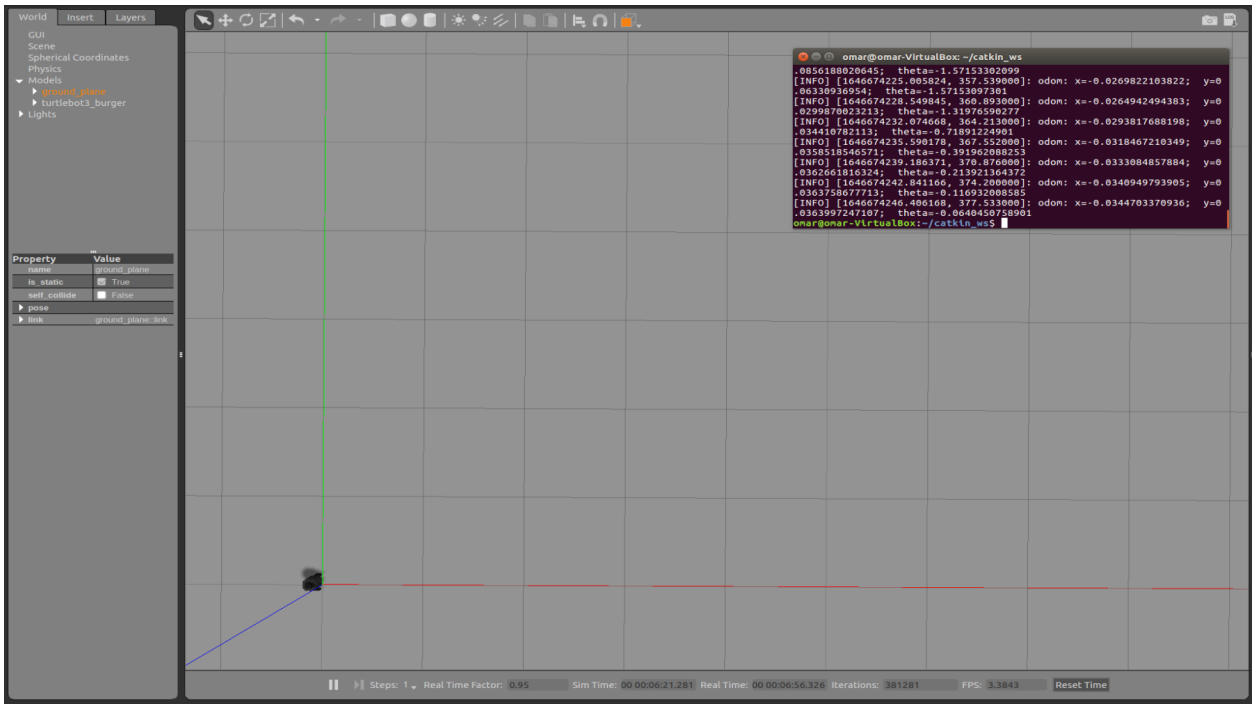
Waypoint 2 (top-right)



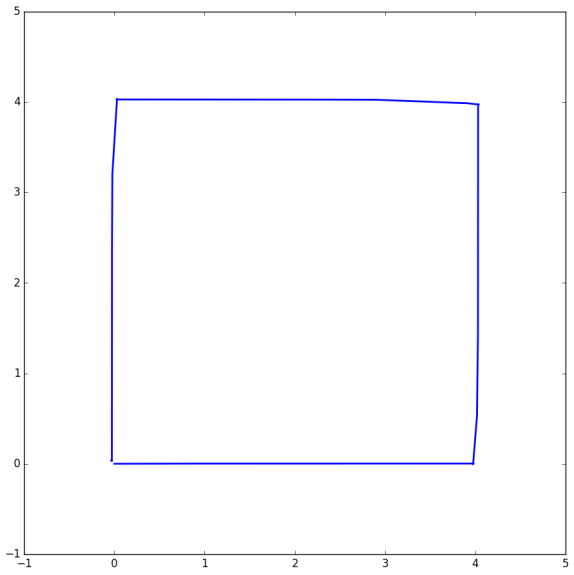
Waypoint 3 (top-left)



Waypoint 4 / Final position (bottom-left)



Plotted trajectory (with `0.0,0.0` added to the first line of `trajectory.csv`, to show the initial position):



Lab 5: Sensing

Task 5-1

A Python script named `laser_data.py` was created within a newly created ROS Package named `lab5`, which extracts the laser data and prints the distance measured in the four directions received by the `scan` topic. The code for the `laser_data.py` script is below:

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan

class Turtlebot3():
    def __init__(self):
        rospy.init_node("turtlebot3_laser_data_subscriber")
        rospy.loginfo("Press Ctrl + C to terminate")
        self.rate = rospy.Rate(10)

        self.scan_sub = rospy.Subscriber("scan", LaserScan,
self.scan_callback)

        try:
            self.run()
        except rospy.ROSInterruptException:
            rospy.loginfo("Action terminated.")
        finally:
            pass

    def run(self):
        while True:
            self.rate.sleep()

    def scan_callback(self, msg):
        d_front = msg.ranges[0]
        d_left = msg.ranges[89]
        d_back = msg.ranges[179]
        d_right = msg.ranges[269]
        print("Front: " + str(d_front))
        print("Left: " + str(d_left))
        print("Back: " + str(d_back))
        print("Right: " + str(d_right) + "\n")
```

```
if __name__ == '__main__':
   robot = Turtlebot3()
```

Task 5-2

Closed loop obstacle avoidance

Closed loop obstacle avoidance was implemented by subscribing to the `scan` topic to receive laser data as in Task 5-1, and publishing to the `cmd_vel` topic to send movement commands. In our Turtlebot3 class, we have a `near_obstacle` member variable which is `True` if near an obstacle, and `False` otherwise.
In the scan callback, we check if the distance to an obstacle in-front of the robot is less than 0.4m (predefined threshold) and if so we set `near_obstacle` to `True`. Otherwise, it's set to `False`.
At every step of the rate, we publish a movement command with a linear x velocity of 0 if `near_obstacle` is `True`, otherwise 0.1. Thus, when the robot is near an obstacle, its velocity is set to 0 and it halts, otherwise it drives forwards by our predefined velocity.
Closed loop obstacle avoidance was implemented in a script named `obstacle_avoidance_closed_loop.py` in our `lab5` ROS Package, and the code is below:

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

class Turtlebot3():
   def __init__(self):
       rospy.init_node("turtlebot3_laser_data_subscriber")
       rospy.loginfo("Press Ctrl + C to terminate")
       self.rate = rospy.Rate(10)

       self.scan_sub = rospy.Subscriber("scan", LaserScan,
self.scan_callback)
       self.vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=10)

       self.near_obstacle = False

       try:
           self.run()
```

```
        except rospy.ROSInterruptException:
            rospy.loginfo("Action terminated.")
        finally:
            pass

    def run(self):
        while True:
            move = Twist()
            move.linear.x = 0 if self.near_obstacle else 0.1
            self.vel_pub.publish(move)
            self.rate.sleep()

    def scan_callback(self, msg):
        d_front = msg.ranges[0]
        self.near_obstacle = d_front < 0.4

if __name__ == '__main__':
    robot = Turtlebot3()
```

Open loop obstacle avoidance

Closed loop obstacle avoidance was implemented by first creating a queue `d_fronts` to store the distance measurements to the obstacle in-front of the robot, with size `num_d`. This is to overcome occasionally highly incorrect measurements. In the scan callback, we add `d_front` to the head of the queue and pop the tail if the queue is full. We set the queue size to 5, so we take the mean of the 5 previous distance measurements.

On each step of the rate, we use a P Controller with a goal distance to the obstacle in-front of the robot as 0.3m (with the same implementation of the PD Controller as in Lab 4). We compute the mean of all of the distance measurements to the obstacle in-front of the robot in the `d_fronts` queue. If the queue wasn't full, we have not made enough measurements yet, so we set the velocity of the robot to 0. However, if it was full then we compute the distance to the obstacle in-front of the robot `d_front` by taking the mean of all the values in the queue.

Then, we update our P Controller with `d_front` to obtain the difference in distance to the obstacle in-front of the robot to the goal distance (0.3m). We then multiply that value by -1 since the difference will be negative when the robot is further away, and scale it by dividing by 5 so the velocity isn't too large (determined experimentally), and set the velocity of the robot to the scaled value from the P Controller.
In terms of the gain of our P Controller, we began with a PD Controller with $k_p=0.9$ and $k_d=0.3$ since those values worked well in Lab 4. However, it was found that k_d could be 0 (likely due to

our use of the queue for the distance measurements and averaging over them to reduce the error), and for slightly higher velocity k_p was increased to 0.95. $k_p=0.95$, $k_d=0$ and our scaling of the P Controller's outputs yields the expected behaviour of the robot: beginning fast and smoothly decreasing its velocity to stop near the wall when its distance reaches 0.3m.

Open loop obstacle avoidance was implemented in a script named `obstacle_avoidance_open_loop.py` in our `lab5` ROS Package, and the code is below (we copied the implementation of the PD Controller from Lab 4 into our package and imported it from the `pdcontroller.py` file):

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from pdcontroller import Controller

class Turtlebot3():
    def __init__(self):
        rospy.init_node("turtlebot3_laser_data_subscriber")
        rospy.loginfo("Press Ctrl + C to terminate")
        self.rate = rospy.Rate(10)

        self.scan_sub = rospy.Subscriber("scan", LaserScan,
self.scan_callback)
        self.vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=10)

        self.d_fronts = [] # Last `num_d` distances to the obstacle
in-front of the robot
        self.num_d = 5 # How many distances to the obstacle in-front of the
robot to average over

        try:
            self.run()
        except rospy.ROSInterruptException:
            rospy.loginfo("Action terminated.")
        finally:
            pass

    def run(self):
        # P Controller with goal distance to obstacle in-front as 0.3m.
        # Use P=0.95 and D=0 from testing.
        pd_d_front = Controller(0.95, 0, 0.3)
```

```
        while True:
            # Get mean distance to obstacle in-front of the robot
            d_front = self.mean_d_front()
            # Set linear x velocity of robot to 0 if `d_front` is None (in
which case we haven't made enough measurements)
            # Otherwise compute the difference in distance to the nearby
obstacle to the goal distance from the P Controller,
            # and set the robot's linear x velocity proportional to that
change in distance (determined experimentally).
            v = 0 if d_front is None else -pd_d_front.update(d_front) / 5
            move = Twist()
            move.linear.x = v
            self.vel_pub.publish(move)
            self.rate.sleep()

    def scan_callback(self, msg):
        d_front = msg.ranges[0]
        # Add `d_front` to the `d_fronts` queue
        if len(self.d_fronts) == self.num_d:
            self.d_fronts.pop(0)
        self.d_fronts.append(d_front)

    # Compute the mean of the distances in-front of the robot (or None if
there aren't enough)
    def mean_d_front(self):
        if len(self.d_fronts) < self.num_d:
            # Not enough distances yet
            return None
        # Compute mean of all the distances in `d_fronts`
        return sum(self.d_fronts) / float(self.num_d)

if __name__ == '__main__':
    robot = Turtlebot3()
```

Non-zero velocity of the obstacles

In order to get the robot car to match the non-zero velocity of the car-front rather than stop at the required distance, we could still use a PD Controller for the distance to the car in-front of the robot (with goal equal to the required distance to be behind the car in-front of the robot), but when setting the velocity of the robot from its output, we should set the velocity to $v_c + ku$

where v_c is the velocity of the car-front, u is the output from the PD Controller and k is some scaling constant for the output from the PD Controller (it should be negative since being further away from the car causes it to be negative but we want an increase in velocity in that case, and higher absolute value results in faster velocity of the car, thus faster to reach the required distance and catch up to the car in-front).
This approach assumes we know v_c (the velocity of the car in-front). If we don't know this, it could be estimated by taking two measurements of the distance to the car in-front d_1, d_2, then the estimated velocity of the car in-front is $(d_2 - d_1 + vt) / t$ where v = velocity of the robot during the measurements and t = time between the two measurements. This is because $velocity = distance / time$, and the other car travelled $d_2 - d_1 + vt$ distance in time t (vt comes from the fact that the robot will have moved towards the car by vt during the measurements, making d_2 smaller than it should be, specifically by vt).

Task 5-3

For the task of wall following, we use laser measurements up to 45 degrees to the right of the robot, rather than 90 degrees as suggested. We divide the 45 degrees into 15 degree intervals and take distance measurements at each, taking the mean over the last 5 measurements for each angle with the queue approach as in the previous implementation for open loop obstacle avoidance.

At each step, we compute the minimum measured mean distance across all of our measured angles to get the distance to the closest obstacle within 45 degrees to the right of the robot. If that minimum distance is less than 0.3, then the robot is too close and facing a wall, thus we rotate the robot left by applying to it an angular velocity of 0.2. The robot rotates until the minimum measured distance becomes greater than or equal to 0.3, immediately after which it is parallel to the wall thus we apply a linear velocity of 0.1 to make it follow the wall. While applying this velocity, the robot may have slightly overturned left when it was facing the wall making the robot travel slightly non-parallel, thus we also employ a PD maintaining a goal minimum measured distance of 0.3 to the wall. When the minimum measured distance to the wall becomes greater than 0.3, the robot has an angular velocity applied to it proportional to the difference in its current distance and 0.3 via the PD, rotating the robot right while it is travelling forward to slightly correct its rotation and remain parallel to the wall. The multiplier on the output of this PD corresponds to the "stickiness" of the robot - if we have a high multiplier on the output then the correcting angular velocity would be larger, making the robot correct itself more quickly. At the start of the episode the robot is not close to a wall so the minimum measured distance to a nearby obstacle is larger than 0.3, causing the robot to move forward with a linear velocity. However, the PD correcting the robot's rotation to be parallel to a wall only applies when it has first reached a wall (which is determined by a boolean flag).

This approach has been tested and works on both the `turtlebot3_stage_1` and `turtlebot3_world` worlds. It's important that we measure angles 0 and 45 degrees to maintain the threshold 0.3 distance from the front and right walls respectively, and that no

angles beyond 45 degrees are measured, otherwise the robot will over-turn from the wall when getting close and facing it. Using more angles within the 0 to 45 degree interval improves the motion of the robot.

The wall following behaviour was implemented in a script named `wall_following.py` in our `lab5` ROS Package, and the code is below:

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from pdcontroller import Controller

class Turtlebot3():
   def __init__(self):
       rospy.init_node("turtlebot3_laser_data_subscriber")
       rospy.loginfo("Press Ctrl + C to terminate")
       self.rate = rospy.Rate(10)

       self.scan_sub = rospy.Subscriber("scan", LaserScan,
self.scan_callback)
       self.vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=10)

       self.d_angles = [0, 344, 329, 314] # Angles to measure distances
from the laser sensors
       self.distances = [[],[], [], []] # Queues of the measured distances
from the laser sensors at each angle
       self.num_d = 5 # How many distances to the obstacle in-front of the
robot to average over

       try:
           self.run()
       except rospy.ROSInterruptException:
           rospy.loginfo("Action terminated.")
       finally:
           pass

   def run(self):
       # P Controller to maintain a 0.3 measured distance to a nearby
obstacle
       pd = Controller(0.95,0,0.3)
```

```
        on_wall = False # Whether the robot has reached a wall yet or not
        while True:
            mean_ds = self.mean_ds() # Compute the mean measured distances
to obstacles for each angle
            move = Twist()
            if mean_ds is not None: # May be None if haven't measured
enough distances yet
                min_d = min(mean_ds) # Compute the minimum measured
distance to an obstacle
                if min_d < 0.3:
                    # Facing towards the wall and too close, rotate left to
avoid it
                    on_wall = True
                    move.angular.z = 0.2
                else:
                    # Either far away from a wall or parallel to it, move
forwards
                    move.linear.x = 0.1
                    if on_wall:
                        # If the robot is further than 0.3 from the wall
then rotate it right to remain parallel.
                        # This fixes the overshoot that may occur from the
left rotation when close to the wall.
                        # Use a PD with a desired distance of 0.3 for the
rotation so the rotation is proportional to the difference
                        move.angular.z = pd.update(min_d)

            self.vel_pub.publish(move)
            self.rate.sleep()

    def scan_callback(self, msg):
        # Extract the laser distance measurements for each angle we are
measuring
        # and add the measurements to their respective queues
        for i, angle in enumerate(self.d_angles):
            d = msg.ranges[angle]
            dsi = self.distances[i]
            if len(dsi) == self.num_d:
                dsi.pop(0)
            dsi.append(d)
```

```
    def mean_ds(self):
        # Compute the mean measured distance for each angle we are
measuring
        # Mean of each queue
        mean_ds = []
        for ds in self.distances:
            if len(ds) < self.num_d:
                # Not enough distances yet
                return None
            mean_ds.append(sum(ds) / float(self.num_d))
        return mean_ds


if __name__ == '__main__':
    robot = Turtlebot3()
```

Lab 6: Localisation

Task 6-1

For the task of estimating the robot poses using the particle set, the code to drive the robot in a square from Task 4-2 was extended.

Particle Class

Firstly, a `Particle` class was implemented which stores the parameters of the particles (`x`, `y`, `theta` and the weight `w`) and updates them based upon the motion model specified in the brief. Specifically, the functions `Particle:update_straight_line_motion` and `Particle::update_rotation_motion` apply the motion equations for a movement of a distance `d` or a rotation of an angle `alpha` respectively. In both, we sample the random Gaussian errors via the `Particle.gen_noise` method, which uses numpy to sample `n` random points from a Gaussian with 0 mean and standard deviation given by the `noise_gaussian_deviation` global variable. We found that a value of 0.001 for the deviation of the Gaussian errors caused noticeable but not too much error in the particle's positions after the robot traverses the square – the value is so small because the robot travels very slowly around the square to achieve the accuracy it does in Task 4-2, thus the effect of the errors are amplified.

Particle Set

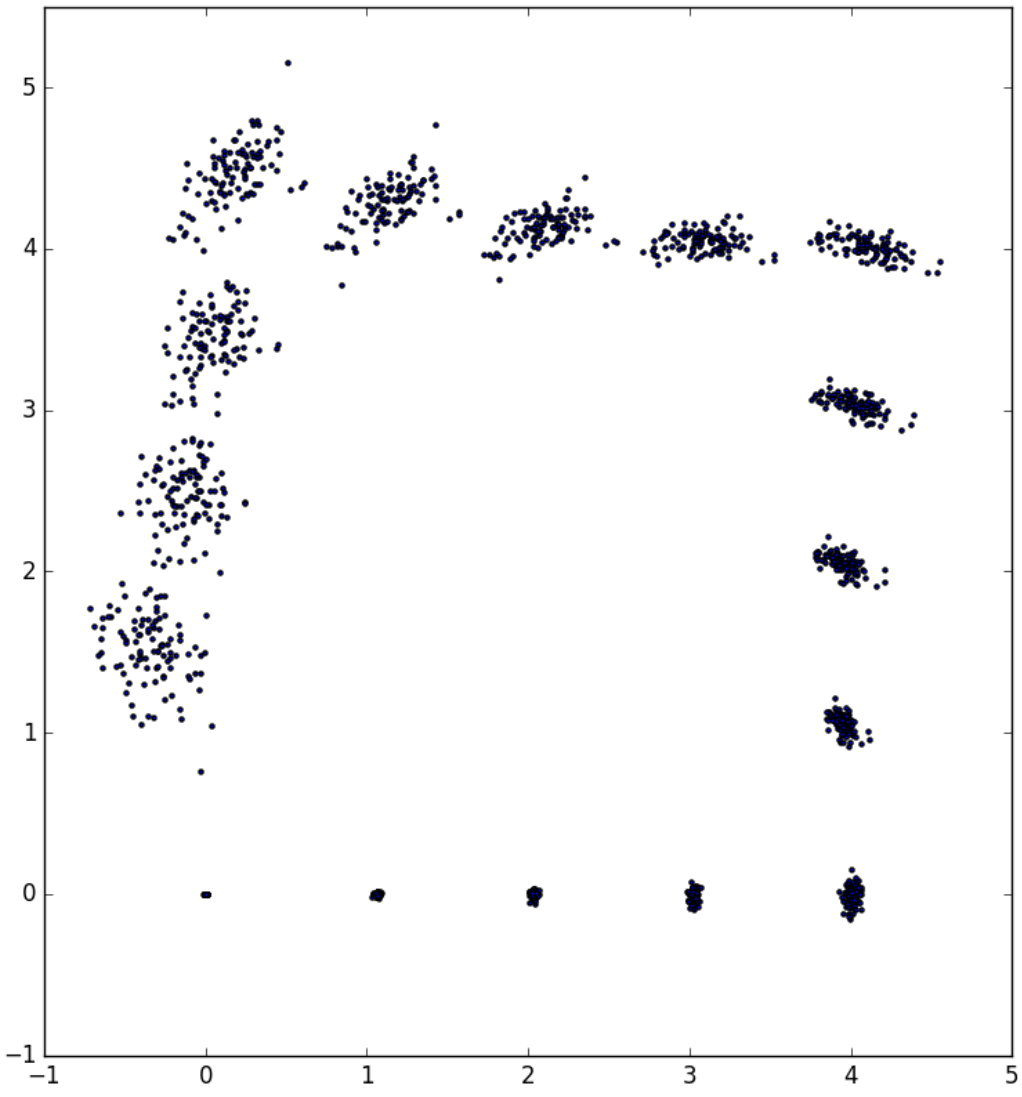
With the particle motion implemented, the particle set was implemented by maintaining an array of 100 particles (specified by the `num_particles` global variable), and calling the particle motion update functions after publishing each velocity command. At each timestep we publish a velocity command of type `Twist`, which may have non-zero linear x velocity and angular z velocity. We convert the linear x velocity and angular z velocity to a distance moved and angle rotated during the time-step by applying the equations for velocity and angular velocity respectively. Specifically, in a time-step, the robot's current velocity is applied over an interval of `t=1/hz` seconds, where `hz` is the rate, i.e. the number of steps per second. Since `velocity = distance travelled / time`, and `angular velocity = change in angle / time`, we have that `distance travelled = velocity * time`, and `change in angle = angular velocity * time`. Thus, we obtain the distance moved in the timestep `d` and the change in angle `alpha` as `d = (linear x velocity) * t` and `alpha = (angular z velocity) * t`. With `d` and `alpha`, we apply `Particle:update_straight_line_motion` and `Particle::update_rotation_motion` with `d` and `alpha` to each particle respectively, to update their positions based upon the robot's movement commands.

Robot Motion

Next, the robot was made to stop at 1 metre intervals (0.5 metres resulted in too dense of a graph of the particles), by checking if the robot's position was near 1 metre intervals on each side of the square (which we call "checkpoints"). When the robot comes within 0.05 metres of a checkpoint, we publish movement commands with 0 linear and angular velocity for 2 seconds (specified by the stop_seconds global variable), after which we report the robot's position and covariance matrix from its particles. We obtain the position estimate from the particles by computing the mean (x,y) position across all of the particles in the particle set, and we compute the covariance matrix by applying the formula given in Part 2 of Lab 3 with the (x,y) positions of the particles in the particle set. We add the (x,y) positions of all of the particles to the `particles_trajectory` array after each stop of the robot at the checkpoints in order to visualise the positions of the particles at each checkpoint.

Results

After the robot has traversed its square, we write the `particles_trajectory` array to a CSV file as in Part 2 of Lab 4, and visualise it with a scatter plot using `matplotlib` in a similar way. Below shows the plotted positions of the particles at each checkpoint after the robot has traversed the square.


5
4
3
2
1
0
−1
−1
0
1
2
3
4
5

Output

Below we show the output of `particle_square.py`, which contains the position estimates and covariance matrices from the particles at each checkpoint:

```
Position estimate: (0.000451589339915,1.11188704068e-06)
Covariance matrix:
[[  1.97713520e-05  -1.35576824e-09]
 [ -1.35576824e-09   5.40301415e-10]]

Position estimate: (1.06419967209,-0.000272056806021)
Covariance matrix:
[[  8.18420426e-05   4.46298477e-06]
 [  4.46298477e-06   5.56647564e-05]]

Position estimate: (2.03477031093,-0.00116118526624)
Covariance matrix:
[[  1.66322978e-04   8.35980834e-06]
 [  8.35980834e-06   2.91757284e-04]]

Position estimate: (3.02524732538,-0.00311859579183)
Covariance matrix:
[[  2.51377563e-04   1.78462497e-05]
 [  1.78462497e-05   1.05899766e-03]]

Position estimate: (4.00804415463,-0.00436333354913)
Covariance matrix:
[[ 0.00073627  0.00017724]
 [ 0.00017724  0.0030172 ]]

Position estimate: (3.9486731381,1.06016741644)
Covariance matrix:
[[ 0.00240186 -0.00134118]
 [-0.00134118  0.00300677]]

Position estimate: (3.94588310177,2.05280909394)
Covariance matrix:
[[ 0.00741999 -0.00279493]
 [-0.00279493  0.00317854]]

Position estimate: (4.01950390418,3.03803830485)
Covariance matrix:
[[ 0.01569489 -0.00441549]
 [-0.00441549  0.00342812]]
```

```
Position estimate: (4.10777622429,4.01135335609)
Covariance matrix:
[[ 0.02758719 -0.00659373]
 [-0.00659373  0.00427219]]

Position estimate: (3.08603470834,4.04830803817)
Covariance matrix:
[[ 0.02797104  0.00017163]
 [ 0.00017163  0.00404322]]

Position estimate: (2.10116069272,4.13264401809)
Covariance matrix:
[[ 0.0289989   0.006909  ]
 [ 0.006909    0.00967598]]

Position estimate: (1.13927491409,4.28674153873)
Covariance matrix:
[[ 0.03041471  0.01447398]
 [ 0.01447398  0.02121394]]

Position estimate: (0.170702725163,4.45767652579)
Covariance matrix:
[[ 0.03229184  0.02234847]
 [ 0.02234847  0.04098147]]

Position estimate: (0.0496100741827,3.44153912938)
Covariance matrix:
[[ 0.02194958  0.01103637]
 [ 0.01103637  0.04307987]]

Position estimate: (-0.116331368529,2.4689867744)
Covariance matrix:
[[ 0.0207843  -0.00060375]
 [-0.00060375  0.04661979]]

Position estimate: (-0.349596841291,1.52251399111)
Covariance matrix:
[[ 0.0284222  -0.01365724]
 [-0.01365724  0.0527707 ]]
```

Code

The particle set and moving the robot in a square was implemented in a script named `particle_square.py` within a ROS Package named `lab6`, and its code is shown below. The visualisation of the particle positions at each checkpoint within the CSV outputted by

`particle_square.py` was implemented in a script named `vis_particles.py` in the `lab6` ROS Package, and is also shown below.

`particle_square.py` code:

```
#!/usr/bin/env python
from math import pi, sqrt, atan2, cos, sin
import numpy as np

import rospy
import tf
from std_msgs.msg import Empty
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose2D

import pdcontroller

num_particles = 100
noise_gaussian_deviation = 0.001 # Standard deviation for the randomness
of the particle's motion
stop_seconds = 2 # Number of seconds to stop at each checkpoint
stop_points = [0,1,2,3,4] # Distances along a side of a square to stop at
(we stop at 1m intervals)


# Subtracts two angles in the range [+pi,-pi] radians
# https://stackoverflow.com/a/2007279
def sub_angles(theta1, theta2):
       delta = theta1 - theta2
       return atan2(sin(delta), cos(delta))

# Class for a particle and its motion in the world
class Particle:
   def __init__(self):
       self.x = 0.0
       self.y = 0.0
       self.theta = 0.0
       self.weight = 1.0 / num_particles

   # Generate random errors for the robot's motion
   @staticmethod
```

```
    def gen_noise(n):
        # Return n samples from a gaussian with deviation
`noise_gaussian_deviation`
        return np.random.normal(0, noise_gaussian_deviation, n)

    # Apply straight-line motion of distance d formula to update
(x,y,theta) based upon d and Gaussian sampled errors
    def update_straight_line_motion(self, d):
        noise = Particle.gen_noise(2)
        e = noise[0]
        f = noise[1]
        self.x = self.x + (d + e) * cos(self.theta)
        self.y = self.y + (d + e) * sin(self.theta)
        self.theta = self.theta + f

    # Apply pure rotation of angle alpha formula to update theta based upon
alpha and Gaussian sampled error
    def update_rotation_motion(self, alpha):
        noise = Particle.gen_noise(1)
        g = noise[0]
        self.theta = self.theta + alpha + g


class Turtlebot3():
    def __init__(self):
        rospy.init_node("turtlebot3_move_square")
        rospy.loginfo("Press Ctrl + C to terminate")
        self.vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=10)
        self.hz = 10
        self.rate = rospy.Rate(self.hz)


        # Initialise particles
        self.particles = list()
        for _ in range(num_particles):
            self.particles.append(Particle())
        # Initialise particles trajectory list, which stores the positions
of all the particles at each 1m checkpoint
        self.particles_trajectory = list()
```

```
        # subscribe to odometry
        self.pose = Pose2D()
        self.logging_counter = 0
        self.trajectory = list()
        self.odom_sub = rospy.Subscriber("odom", Odometry,
self.odom_callback)

        try:
            self.run()
        except rospy.ROSInterruptException:
            rospy.loginfo("Action terminated.")
        finally:
            # save particles trajectory into csv file
            np.savetxt('./particle_trajectory.csv',
np.array(self.particles_trajectory), fmt='%f', delimiter=',')
            # save trajectory into csv file
            np.savetxt('./trajectory.csv', np.array(self.trajectory),
fmt='%f', delimiter=',')

    # Apply the particles' dynamics to all of the particles based upon a
movement command `move` to the robot (of type Twist)
    def update_particles(self, move):
        # Formulas to obtain the distance moved and angle rotated:
        #   d = vt
        #   delta_theta = angular_velocity * delta_t
        t = 1.0 / self.hz # Delta time across one timestep
        d = move.linear.x * t # Distance moved computed via d = vt
        alpha = move.angular.z * t # Angle rotates obtained by delta_theta
= angular_velocity * delta_t
        # Apply the dynamics to each particle, for the computed `alpha` and
`d`
        for particle in self.particles:
            particle.update_rotation_motion(alpha)
            particle.update_straight_line_motion(d)

    # Compute the position estimate of the robot from the positions of the
particles
    def particles_position_estimate(self):
        mean_pos = [0.0,0.0]
```

```
        # Compute mean (x,y) position of particles weighted by their
weights
        for particle in self.particles:
            mean_pos[0] += particle.weight * particle.x
            mean_pos[1] += particle.weight * particle.y
        return mean_pos

    # Compute the covariance matrix of the positions of the particles
    def particles_covar_matrix(self):
        sum_tl, sum_tr, sum_bl, sum_br = 0.0, 0.0, 0.0, 0.0 # Sum top-left,
sum top-right, sum bottom-left, sum bottom-right in the formula
respectively
        mean_pos = self.particles_position_estimate() # Compute [x bar, y
bar]
        # Update the sums for each element of the covar matrix for each
particle based upon the formula of the covar matrix
        for particle in self.particles:
            dx = particle.x - mean_pos[0]
            dy = particle.y - mean_pos[1]
            sum_tl += dx ** 2
            sum_tr += dx * dy
            sum_bl += dy * dx
            sum_br += dy ** 2
        # Normalise and return the matrix
        n = len(self.particles)
        return [[sum_tl / n, sum_tr / n],[sum_bl / n, sum_br / n]]


    def run(self):
        # add your code here to adjust your movement based on 2D pose
feedback
        angles = [pi/2,pi,-pi/2,0]
        goal_positions = [4,4,0,0]
        goal_positions_x = [True, False, True, False]

        angle = 0.0

        for i in range(4):
            goal_pos = goal_positions[i] # Goal position to travel to (x or
y coordinate, depending on `goal_pos_x`)
```

```
            goal_pos_x = goal_positions_x[i] # Whether `goal_pos` is an x
or y coordinate
            new_angle = angles[i] # Angle to rotate to after `goal_pos`

            # PD Controller controlling the forward velocity of the robot
            # Goal position is `goal_pos`
            pd_vx = pdcontroller.Controller(0.9, 0.3, goal_pos)

            # PD Controller controlling the rotation of the robot while it
travels forwards towards its goal
            # Keeps the robot's angle equal to `angle`, which is the
current angle of the robot (it should not change while travelling forward)
            # Use the `sub_angles` comparator for comparing angles
            pd_theta0 = pdcontroller.Controller(0.9, 0.3, angle,
sub_angles)

            # Difference in position of the robot to the goal, based upon
whether the goal is an x or y coordinate
            def delta_goal():
                pos = self.pose.x if goal_pos_x else self.pose.y
                return goal_pos - pos

            # Closest absolute distance to a stopping checkpoint
            def delta_stop_point():
                pos = self.pose.x if goal_pos_x else self.pose.y
                min_d = 99999999
                for stop_point in stop_points:
                    min_d = min(min_d, abs(pos - stop_point))
                return min_d

            # Stop the robot at a checkpoint for `stop_seconds` seconds.
            # Compute the position estimate and covariance matrix for the
particles, and output it.
            # Save the positions of the particles in the particles
trajectory.
            def stop():
                # Stop the robot for `stop_seconds` seconds.
                for _ in range(stop_seconds * self.hz):
                    move = Twist()
                    self.vel_pub.publish(move)
```

```
                    self.update_particles(move)
                    self.rate.sleep()

                # Compute position estimate
                mean_pos = self.particles_position_estimate()
                print("Position estimate: (" + str(mean_pos[0]) + "," +
str(mean_pos[1]) + ")")

                # Compute covariance matrix
                particles_covar = self.particles_covar_matrix()
                print("Covariance matrix:")
                print(np.matrix(particles_covar))
                print() # Padding

                # Save particle positions in particle positions trajectory
                for particle in self.particles:
                    self.particles_trajectory.append([particle.x,
particle.y])

            stopped = False # Whether the robot had stopped at the previous
timestep or not

            # Move the robot forwards while it's not within 0.05 meters of
its goal
            while abs(delta_goal()) > 0.05:
                if delta_stop_point() < 0.05 and not stopped:
                    # Within a stopping checkpoint and we're in a new
checkpoint since we didn't previously stop, so stop the robot
                    stopped = True
                    stop()
                if delta_stop_point() > 0.05 and stopped:
                    # Outside of a checkpoint and we previously stopped, so
set stopped to false so we stop at the next checkpoint
                    stopped = False

                # Get velocity from the controller, giving the current
position as the x or y component of the pose depending on `goal_pos_x`
                u_vx = pd_vx.update(self.pose.x if goal_pos_x else
self.pose.y)
```

```
                # Get the rotation from the controller, giving the theta
component of the pose
                u_theta = pd_theta0.update(self.pose.theta)
                # Goal positions are to the left and below the robot for
the last two sides, which cause the controller to output a negative
velocity.
                # Fix the velocity.
                if i >= 2: u_vx *= -1
                # Scale the obtained forwards and angular velocity from the
PD Controllers, and execute the movement command
                move = Twist()
                move.linear.x = u_vx / 10
                move.angular.z = u_theta
                self.vel_pub.publish(move)
                self.update_particles(move)
                self.rate.sleep()

            # PD Controller controlling the rotation of the robot around
the corner
            # Goal rotation is `new_angle`
            # Use the `sub_angles` comparator for comparing angles
            pd_theta = pdcontroller.Controller(0.9,0.3, new_angle,
sub_angles)

            # Rotate the robot to face `new_angle` (while its rotation is
not within 0.05 radians of `new_angle`)
            while abs(sub_angles(self.pose.theta, new_angle)) > 0.05:
                # Get the rotation from the controller
                u_theta = pd_theta.update(self.pose.theta)
                # Scale the obtained angular velocity from the PD
Controller and execute the movement command
                move = Twist()
                move.angular.z = u_theta / 5
                self.vel_pub.publish(move)
                self.update_particles(move)
                self.rate.sleep()

            # The robot is now facing `new_angle` (with some uncertainty)
            angle = new_angle
```

```
        # Stop robot from moving after traversing the square
        move = Twist()
        move.linear.x = 0
        move.angular.z = 0
        self.vel_pub.publish(move)
        self.update_particles(move)
        self.rate.sleep()

    def odom_callback(self, msg):
        # get pose = (x, y, theta) from odometry topic
        quaternion =
[msg.pose.pose.orientation.x,msg.pose.pose.orientation.y,\
                    msg.pose.pose.orientation.z,
msg.pose.pose.orientation.w]
        (roll, pitch, yaw) =
tf.transformations.euler_from_quaternion(quaternion)
        self.pose.theta = yaw
        self.pose.x = msg.pose.pose.position.x
        self.pose.y = msg.pose.pose.position.y

        # logging once every 100 times (Gazebo runs at 1000Hz; we save it
at 10Hz)
        self.logging_counter += 1
        if self.logging_counter == 100:
            self.logging_counter = 0
            self.trajectory.append([self.pose.x, self.pose.y])  # save
trajectory
            # rospy.loginfo("odom: x=" + str(self.pose.x) +\
            #     ";  y=" + str(self.pose.y) + ";  theta=" + str(yaw))

if __name__ == '__main__':
    robot = Turtlebot3()
```

`vis_particles.py` code:

```
#!/usr/bin/env python

import numpy as np
import matplotlib.pyplot as plt
```

```
def visualization():
    # load csv file and plot trajectory
    _, ax = plt.subplots(1)
    ax.set_aspect('equal')

    trajectory = np.loadtxt("./particle_trajectory.csv", delimiter=',')
    plt.scatter(trajectory[:, 0], trajectory[:, 1], s=5)

    plt.xlim(-1, 5)
    plt.ylim(-1, 5.5)
    plt.show()

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

Task 6-2

For Task 6-2, we extended the code for the particle set in Task 6-1, and the code for the laser measurements in Task 5-3. We implemented the code for Task 6-2 in a script named `part_2.py` in the `lab6` ROS Package.

Preliminaries

Firstly, the code to move the robot to each point of the square was generalised for any arbitrary sequence of points, given by the global `goal_positions` array, containing each point of the robot's path in the maze. In addition, we generalise the stopping of the robot to positions that are a multiple of 2, so the robot stops at 0.5m intervals.

In order to obtain the map data to compute the expected measurements, we harnessed the `map_server` ROS Package. After downloading the PGM and YAML files to a directory, we ran `rosrun map_server map_server map.yaml` to publish the parsed map data to the map topic. After publishing the map data, it was read from the map topic via a subscriber on the map topic with the `nav_msgs.msg.OccupancyGrid` type, with a callback function `map_callback` which extracts and saves the map data from the data field of the `OccupancyGrid`.

With the map data available in the map array, we thus implemented the `index_map` utility function, which returns the value in the map array corresponding to an (x,y) coordinate in the world space. We firstly account for the difference in centres in the map and the world by adding 3.75 to the x and y coordinate (since the map image is 150x150 pixels and the centre is roughly (0,0) at `0.05 * (150/2) = 3.75` on the x and y axis), then account for the difference in

resolution between the two by dividing the offset x and y coordinates by 0.05. Now, we have the coordinate in the map space, and we obtain the index in the map data array by converting it to row-major order. With the index in the map, we return the value in the map at the index.

Measurement Update Step

With the preliminaries implemented, we thus implement the measurement update step, which occurs every time the robot stops at 0.5m intervals. We do not perform the measurement update at every time-step because it takes too long for the simulation to keep up.

First, in order to compute the expected measurement for a particle, we first cast 36 “rays” at 10 degree increments around it, seeking the first point in the map that is occupied for each ray and recording the distance to such points. To perform each raycast, we implemented a function `raycast` which projects a ray from the world position (x,y) at a given angle, which tests points at 0.005 metre increments from the position in the direction of the ray, returning the first position which is occupied. These distances from the raycasts form the expected measurement.

Then, to obtain the weight of a particle in the measurement update, we compare the expected measurement distances with the actual measurements from the laser data. We sum the absolute differences between the distances at each angle, and normalise the value by the sum across all particles. Then, we compute 1 minus the value to obtain the weight of the particle, and then re-normalise by dividing by the number of particles minus 1, such that the weight is largest when the absolute difference between the expected and actual measurements is the smallest. Indeed, we could have used a 0 mean Gaussian here instead, however for simplicity we don’t, since it would have required tuning of its variance (which may have been time-consuming).

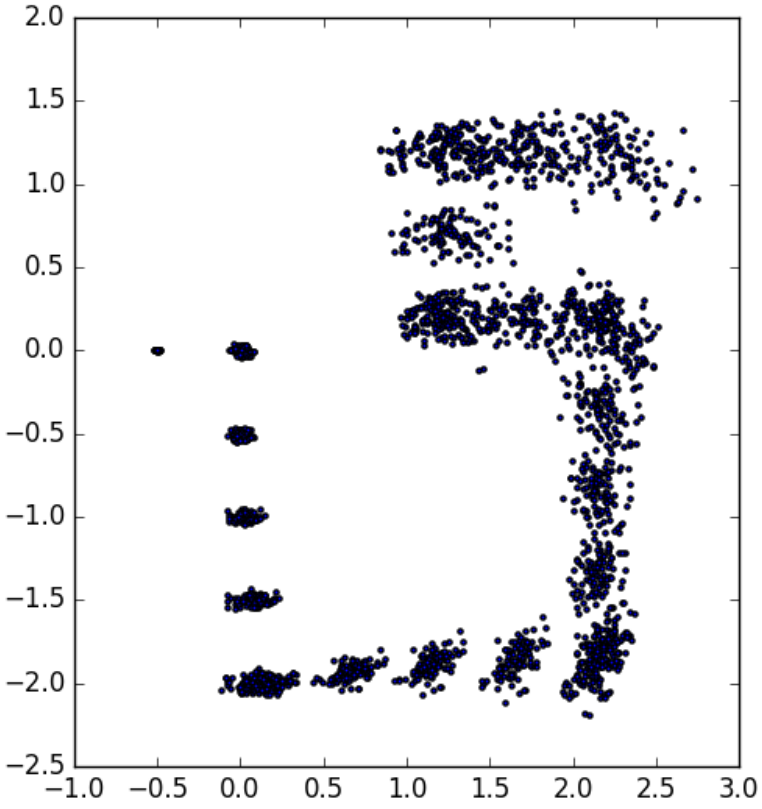
With the new weights of the particles, we thus re-sample them by drawing a random number between 0 and 1 for each particle, and pick the particle corresponding to this random number (with larger weight particles being more likely), using the Stochastic Universal Sampling technique as explained in Lecture 1 of Week 6, which completes the measurement update step.

With the particle set and measurement update implemented, we conclude the implementation of Task 6-2.

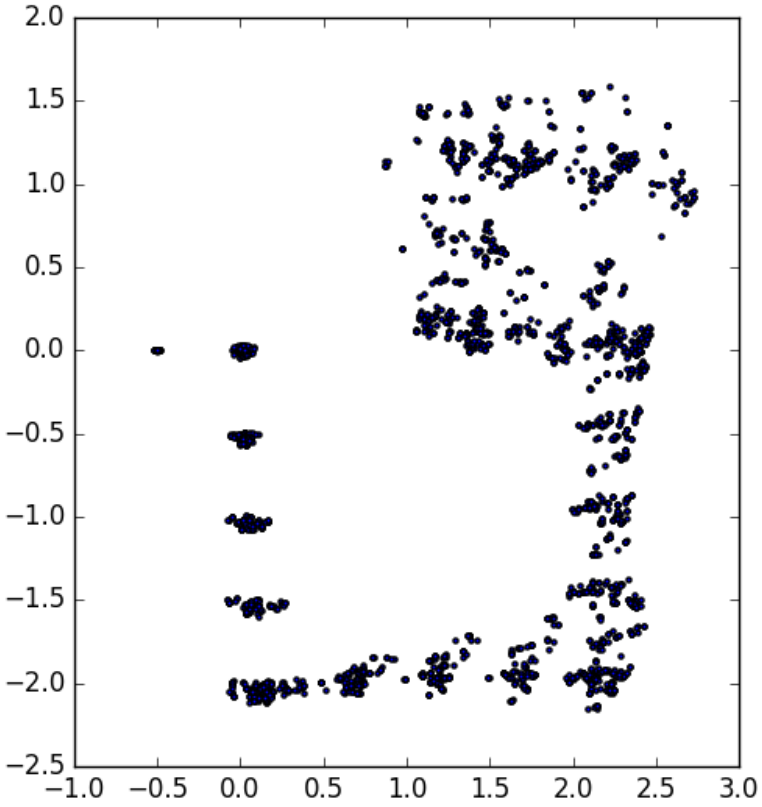
Results

We ran the robot in the maze with and without the measurement update, and plotted the particle set in both cases using the vis_particles.py script from Task 6-1.

Without the measurement update step, the particle trajectory looked like:



However, with the measurement update step, the particle trajectory looked like:



Indeed, we can see the particles inhabiting less positions, verifying the effectiveness of the measurement update step, however the step hasn't seemed to reduce their spread by much.

The measurement update step wasn't as effective as we expected, and there are many potential reasons:

- We do not perform the measurement update frequently enough,
- We do not sample the weights of the particles from a 0-mean Gaussian,
- We do not use enough angles in our expected and actual measurements.

Further work may help in improving the effectiveness of the measurement update step, however for our purposes it worked well enough in this lab.

Below, we list the output of the `part_2.py` script (the position estimate and covariance matrix from the particles after performing the measurement update at each stop position), and the code implementing the `part_2.py` script.

Output

Output position estimates and covariance matrices of the particles after performing the measurement update at each 0.5m checkpoint:

```
Position estimate: (-0.50518857573,-0.000588924286596)
Covariance matrix:
[[  1.58339502e-05   4.75215926e-07]
 [  4.75215926e-07   1.42624028e-08]]

Position estimate: (-0.00143467184755,-0.00946169052623)
Covariance matrix:
[[  3.80860170e-36   1.35416949e-35]
 [  1.35416949e-35   4.81482486e-35]]

Position estimate: (0.0167738844077,0.0123797816023)
Covariance matrix:
[[  3.89035396e-06  -8.75927046e-06]
 [ -8.75927046e-06   1.97218093e-05]]

Position estimate: (-0.0115877955905,-0.502787512359)
Covariance matrix:
[[  3.64121130e-34  -2.11852294e-32]
 [ -2.11852294e-32   1.23259516e-30]]

Position estimate: (-0.0131414062365,-1.01687210092)
Covariance matrix:
[[  5.06746009e-09  -1.08507729e-07]
 [ -1.08507729e-07   2.32343759e-06]]

Position estimate: (0.00420617456733,-1.52987767156)
Covariance matrix:
[[  3.14859411e-08  -7.81844427e-07]
 [ -7.81844427e-07   1.94144016e-05]]

Position estimate: (0.0509323334597,-2.04607530132)
Covariance matrix:
[[  3.90000814e-33  -1.10933565e-31]
 [ -1.10933565e-31   3.15544362e-30]]

Position estimate: (0.0466151969768,-2.06216140321)
Covariance matrix:
[[  8.13705402e-33   2.40356057e-31]
 [  2.40356057e-31   7.09974815e-30]]

Position estimate: (0.610817475707,-1.98147975031)
Covariance matrix:
[[  1.24352885e-07   1.22444338e-07]
 [  1.22444338e-07   1.20565082e-07]]
```

```
Position estimate: (1.10547889192,-1.94722470284)
Covariance matrix:
[[  1.97215226e-31  -7.88860905e-31]
 [ -7.88860905e-31   3.15544362e-30]]

Position estimate: (1.60771347053,-1.93150897278)
Covariance matrix:
[[  1.77493704e-30   6.80392531e-30]
 [  6.80392531e-30   2.60817137e-29]]

Position estimate: (2.09909142602,-1.92260742597)
Covariance matrix:
[[  1.13904691e-05  -4.67600176e-06]
 [ -4.67600176e-06   1.91958665e-06]]

Position estimate: (2.10833959914,-1.89881476425)
Covariance matrix:
[[  5.97027453e-07  -4.10815581e-06]
 [ -4.10815581e-06   2.82682883e-05]]

Position estimate: (2.09095329752,-1.36767934807)
Covariance matrix:
[[  3.38275649e-07   1.49415715e-06]
 [  1.49415715e-06   6.59966388e-06]]

Position estimate: (2.10876962055,-0.851641120032)
Covariance matrix:
[[  1.62247687e-08  -1.09710953e-07]
 [ -1.09710953e-07   7.41859151e-07]]

Position estimate: (2.14676633887,-0.365230127536)
Covariance matrix:
[[  7.65347952e-08  -7.03158575e-08]
 [ -7.03158575e-08   6.46022479e-08]]

Position estimate: (2.18986501382,0.135727538456)
Covariance matrix:
[[  7.88860905e-31   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00]]

Position estimate: (2.20225145846,0.153872135814)
Covariance matrix:
[[  8.61132685e-06  -8.57774081e-06]
 [ -8.57774081e-06   8.54428576e-06]]
```

```
Position estimate: (1.73085470049,0.206531271352)
Covariance matrix:
[[  1.00667472e-06  -2.74732806e-07]
 [ -2.74732806e-07   7.49776594e-08]]

Position estimate: (1.24347471851,0.268424497003)
Covariance matrix:
[[  4.93038066e-32  -7.39557099e-32]
 [ -7.39557099e-32   1.10933565e-31]]

Position estimate: (1.22142126942,0.297083961721)
Covariance matrix:
[[  6.11979311e-07  -2.85329926e-06]
 [ -2.85329926e-06   1.33032547e-05]]

Position estimate: (1.2578256239,0.810935123103)
Covariance matrix:
[[  8.12194334e-08  -2.50611388e-07]
 [ -2.50611388e-07   7.73288674e-07]]

Position estimate: (1.28126657581,1.31827333087)
Covariance matrix:
[[  1.10933565e-29   4.43734259e-30]
 [  4.43734259e-30   1.77493704e-30]]

Position estimate: (1.27829455383,1.36567796592)
Covariance matrix:
[[  1.33662150e-08  -9.09579581e-07]
 [ -9.09579581e-07   6.18974792e-05]]

Position estimate: (1.76896371441,1.32763789474)
Covariance matrix:
[[  9.65930629e-08   4.00428877e-08]
 [  4.00428877e-08   1.65998759e-08]]

Position estimate: (2.26302763003,1.30165234876)
Covariance matrix:
[[  7.30974816e-06  -4.85796223e-07]
 [ -4.85796223e-07   3.22853764e-08]]
```

Code

`part_2.py` code:

```
#!/usr/bin/env python
from math import pi, atan2, cos, sin
import numpy as np

import rospy
import tf
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose2D

from nav_msgs.msg import OccupancyGrid
from sensor_msgs.msg import LaserScan

import pdcontroller

num_particles = 100 # Number of particles
noise_gaussian_deviation = 0.001 # Standard deviation for the randomness
of the particle's motion
stop_seconds = 2 # Number of seconds to stop at each checkpoint
d_angles = [10 * k for k in range(36)] # Angles to measure distances at
(increments of 10 between 0 and 360 degrees)
do_measurement_update = True # Whether to do the measurement update or not
num_d_avg = 5 # How many distances to the obstacle in-front of the robot
to average over

# Positions to drive to, in the form (x,y,theta) where theta is the angle
to turn to upon arriving
goal_positions = [
    (0.0,0.0,-pi/2),
    (0.0,-2.0,0.0),
    (2.0,-2.0,pi/2),
    (2.0,0.0,-pi),
    (1.0,0.0,pi/2),
    (1.0,1.0,0.0),
    (2.0,1.0,0.0)
]
```

```
# Subtracts two angles in the range [+pi,-pi] radians
# https://stackoverflow.com/a/2007279
def sub_angles(theta1, theta2):
       delta = theta1 - theta2
       return atan2(sin(delta), cos(delta))

# Class for a particle and its motion in the world
class Particle:
   def __init__(self):
       self.x = 0.0
       self.y = 0.0
       self.theta = 0.0
       self.weight = 1.0 / num_particles

   # Generate random errors for the robot's motion
   @staticmethod
   def gen_noise(n):
       # Return n samples from a gaussian with deviation
`noise_gaussian_deviation`
       return np.random.normal(0, noise_gaussian_deviation, n)

   # Apply straight-line motion of distance d formula to update
(x,y,theta) based upon d and Gaussian sampled errors
   def update_straight_line_motion(self, d):
       noise = Particle.gen_noise(2)
       e = noise[0]
       f = noise[1]
       self.x = self.x + (d + e) * cos(self.theta)
       self.y = self.y + (d + e) * sin(self.theta)
       self.theta = self.theta + f

   # Apply pure rotation of angle alpha formula to update theta based upon
alpha and Gaussian sampled error
   def update_rotation_motion(self, alpha):
       noise = Particle.gen_noise(1)
       g = noise[0]
       self.theta = self.theta + alpha + g

       grid = OccupancyGrid()
```

```
        grid.data


class Turtlebot3():
    def __init__(self):
        rospy.init_node("turtlebot3_move_square")
        rospy.loginfo("Press Ctrl + C to terminate")
        self.vel_pub = rospy.Publisher("cmd_vel", Twist, queue_size=10)
        self.hz = 10
        self.rate = rospy.Rate(self.hz)
        self.scan_sub = rospy.Subscriber("scan", LaserScan,
self.scan_callback)

        # Subscriber for the map
        self.map_sub = rospy.Subscriber("map", OccupancyGrid,
self.map_callback)

        self.distances = [[] for _ in d_angles] # Queues of the measured
distances from the laser sensors at each angle

        # Initialise particles
        self.particles = list()
        for _ in range(num_particles):
            p = Particle()
            p.x = -0.7
            self.particles.append(p)
        # Initialise particles trajectory list, which stores the positions
of all the particles at each 1m checkpoint
        self.particles_trajectory = list()


        # subscribe to odometry
        self.pose = Pose2D(x=-0.7)
        self.odom_sub = rospy.Subscriber("odom", Odometry,
self.odom_callback)

        try:
            self.run()
        except rospy.ROSInterruptException:
            rospy.loginfo("Action terminated.")
```

```
        finally:
            # Save particles trajectory into csv file
            np.savetxt('./particle_trajectory_maze.csv',
np.array(self.particles_trajectory), fmt='%f', delimiter=',')

    def map_callback(self, map):
        self.map = map
        print("got map")

    def expected_measurement(self, pos):
        # Increments of 10 degrees from 0
        angle_distances = []
        for angle in [k * 10 * pi / 180 for k in range(36)]:
            nearest_obstacle = self.raycast(pos, angle)
            if nearest_obstacle is None: angle_distances.append(None)
            else:
                d = abs(pos[0] - nearest_obstacle[0]) + abs(pos[1] -
nearest_obstacle[1])
                angle_distances.append(d)
        return angle_distances


    def raycast(self, pos, angle):
        d = 0.0
        def get_pos():
            return (pos[0] + d * sin(angle), pos[1] + d * cos(angle))

        while (d < 8.0 and self.index_map(*get_pos()) != 100):
            d += 0.005

        if self.index_map(*get_pos()) == 100:
            return get_pos()
        return None

    def index_map(self, x, y):
        x += 3.75
        y += 3.75
        i = int(round(x / 0.05) + round(y / 0.05) * self.map.info.width)
        if (i >= len(self.map.data)): return -1
        return self.map.data[i]
```

```
    def scan_callback(self, msg):
        # Extract the laser distance measurements for each angle we are
measuring
        # and add the measurements to their respective queues
        for i, angle in enumerate(d_angles):
            d = msg.ranges[angle]
            dsi = self.distances[i]
            if len(dsi) == num_d_avg:
                dsi.pop(0)
            dsi.append(d)


    def mean_ds(self):
        # Compute the mean measured distance for each angle we are
measuring
        # Mean of each queue
        mean_ds = []
        for ds in self.distances:
            if len(ds) < num_d_avg:
                # Not enough distances yet
                return None
            non_inf_ds = []
            for d in ds:
                if d != float("inf"):
                    non_inf_ds.append(d)
            if len(non_inf_ds) == 0:
                mean_ds.append(None)
            else:
                mean_ds.append(sum(non_inf_ds) / float(len(non_inf_ds)))
        return mean_ds

    # Do measurement update for each particle
    def measurement_update(self):
        m = self.mean_ds() # Compute actual measurement
        # Don't do measurement update if disabled or missing data
        if not do_measurement_update or m is None or self.map is None:
return
        sum_weight = 0.0 # Sum of the particle weights
        # Compute the expected measurement for each particle
```

```
        # and set its weight to the difference between the
        # actual measurement and its expected measurement
        for particle in self.particles:
            exp_m = self.expected_measurement((particle.x, particle.y))
            delta = 0.0 # Difference between expected and actual
measurement
            # Sum the distances between each distance at each angle
            for i in range(len(exp_m)):
                if exp_m[i] is None or m[i] is None:
                    continue
                delta += abs(exp_m[i] - m[i])
            particle.weight = delta
            sum_weight += particle.weight
        # Normalise particle weights
        for particle in self.particles:
            if sum_weight == 0: particle.weight = 1.0 / num_particles
            else: particle.weight = particle.weight / sum_weight
        # Invert particle weights
        if sum_weight > 0:
            for particle in self.particles:
                particle.weight = (1 - particle.weight) / (num_particles -
1)
        # Resample particles
        new_particles = []
        for i in range(num_particles):
            rand = np.random.uniform(0, 1)
            # find particle with weight in this region
            sum_weight = 0.0
            p = self.particles[num_particles - 1]
            for particle in self.particles:
                sum_weight += particle.weight
                if sum_weight >= rand:
                    p = particle
                    break
            new_particle = Particle()
            new_particle.x = p.x
            new_particle.y = p.y
            new_particle.theta = p.theta
            new_particles.append(new_particle)
        # Replace particles with the resampled ones
```

```
        self.particles = new_particles

    # Apply the particles' dynamics to all of the particles based upon a
movement command `move` to the robot (of type Twist)
    def update_particles(self, move):
        # Formulas to obtain the distance moved and angle rotated:
        #   d = vt
        #   delta_theta = angular_velocity * delta_t
        t = 1.0 / self.hz # Delta time across one timestep
        d = move.linear.x * t # Distance moved computed via d = vt
        alpha = move.angular.z * t # Angle rotates obtained by delta_theta
= angular_velocity * delta_t
        # Apply the dynamics to each particle, for the computed `alpha` and
`d`
        for particle in self.particles:
            particle.update_rotation_motion(alpha)
            particle.update_straight_line_motion(d)

    # Compute the position estimate of the robot from the positions of the
particles
    def particles_position_estimate(self):
        mean_pos = [0.0,0.0]
        # Compute mean (x,y) position of particles weighted by their
weights
        for particle in self.particles:
            mean_pos[0] += particle.weight * particle.x
            mean_pos[1] += particle.weight * particle.y
        return mean_pos

    # Compute the covariance matrix of the positions of the particles
    def particles_covar_matrix(self):
        sum_tl, sum_tr, sum_bl, sum_br = 0.0, 0.0, 0.0, 0.0 # Sum top-left,
sum top-right, sum bottom-left, sum bottom-right in the formula
respectively
        mean_pos = self.particles_position_estimate() # Compute [x bar, y
bar]
        # Update the sums for each element of the covar matrix for each
particle based upon the formula of the covar matrix
        for particle in self.particles:
            dx = particle.x - mean_pos[0]
```

```
            dy = particle.y - mean_pos[1]
            sum_tl += dx ** 2
            sum_tr += dx * dy
            sum_bl += dy * dx
            sum_br += dy ** 2
        # Normalise and return the matrix
        n = len(self.particles)
        return [[sum_tl / n, sum_tr / n],[sum_bl / n, sum_br / n]]


    def run(self):
        cur_pos = (-0.7,0.0,0.0) # Current position of the robot
        # Visit each goal position
        for i in range(len(goal_positions)):
            goal_pos = goal_positions[i] # Goal position to travel to
            new_angle = goal_pos[2] # Angle to rotate to after `goal_pos`
            angle = cur_pos[2] # Current angle
            goal_x = cur_pos[0] != goal_pos[0] # Whether the goal is along
an x trajectory or not

            # PD Controller controlling the forward velocity of the robot
            # Goal position is `goal_pos`
            pd_vx = pdcontroller.Controller(0.95, 0.3, goal_pos[0] if
goal_x else goal_pos[1])

            # PD Controller controlling the rotation of the robot while it
travels forwards towards its goal
            # Keeps the robot's angle equal to `angle`, which is the
current angle of the robot (it should not change while travelling forward)
            # Use the `sub_angles` comparator for comparing angles
            pd_theta0 = pdcontroller.Controller(0.95, 0.3, angle,
sub_angles)

            # Difference in position of the robot to the goal, based upon
whether the goal is an x or y coordinate
            def delta_goal():
                if goal_x:
                    return goal_pos[0] - self.pose.x
                else:
                    return goal_pos[1] - self.pose.y
```

```
            # Closest absolute distance to a stopping checkpoint
            def delta_stop_point():
                pos = self.pose.x if goal_x else self.pose.y
                # Stopping checkpoints are multiples of a half
                # Thus each stopping point is an integer when multipied by
2
                return abs(round(2 * pos) - 2 * pos)

            # Stop the robot at a checkpoint for `stop_seconds` seconds;
            # Perform the measurement update;
            # Compute the position estimate and covariance matrix for the
particles, and output it;
            # Save the positions of the particles in the particles
trajectory.
            def stop():
                # Stop the robot for `stop_seconds` seconds.
                for _ in range(stop_seconds * self.hz):
                    move = Twist()
                    self.vel_pub.publish(move)
                    self.update_particles(move)
                    self.rate.sleep()

                # Measurement update
                self.measurement_update()

                # Compute position estimate
                mean_pos = self.particles_position_estimate()
                print("Position estimate: (" + str(mean_pos[0]) + "," +
str(mean_pos[1]) + ")")

                # Compute covariance matrix
                particles_covar = self.particles_covar_matrix()
                print("Covariance matrix:")
                print(np.matrix(particles_covar))
                print() # Padding

                # Save particle positions in particle positions trajectory
                for particle in self.particles:
```

```
                    self.particles_trajectory.append([particle.x,
particle.y])

            stopped = False # Whether the robot had stopped at the previous
timestep or not

            # Move the robot forwards while it's not within 0.05 meters of
its goal
            while abs(delta_goal()) > 0.005:
                if delta_stop_point() < 0.05 and not stopped:
                    # Within a stopping checkpoint and we're in a new
checkpoint since we didn't previously stop, so stop the robot
                    stopped = True
                    stop()
                if delta_stop_point() > 0.05 and stopped:
                    # Outside of a checkpoint and we previously stopped, so
set stopped to false so we stop at the next checkpoint
                    stopped = False

                # Get velocity from the controller, giving the current
position as the x or y component of the pose depending on `goal_pos_x`
                u_vx = pd_vx.update(self.pose.x if goal_x else self.pose.y)
                # Get the rotation from the controller, giving the theta
component of the pose
                u_theta = pd_theta0.update(self.pose.theta)
                # Goal positions are to the left and below the robot for
the last two sides, which cause the controller to output a negative
velocity.
                # Fix the velocity.
                if (goal_x and goal_pos[0] < cur_pos[0]) or (not goal_x and
goal_pos[1] < cur_pos[1]):
                    u_vx *= -1
                # Scale the obtained forwards and angular velocity from the
PD Controllers, and execute the movement command
                move = Twist()
                move.linear.x = u_vx / 10
                move.angular.z = u_theta
                self.vel_pub.publish(move)
                self.update_particles(move)
                self.rate.sleep()
```

```
            # PD Controller controlling the rotation of the robot to face
`new_angle`
            # Use the `sub_angles` comparator for comparing angles
            pd_theta = pdcontroller.Controller(0.9,0.3, new_angle,
sub_angles)

            # Rotate the robot to face `new_angle` (while its rotation is
not within 0.05 radians of `new_angle`)
            while abs(sub_angles(self.pose.theta, new_angle)) > 0.005:
                # Get the rotation from the controller
                u_theta = pd_theta.update(self.pose.theta)
                # Scale the obtained angular velocity from the PD
Controller and execute the movement command
                move = Twist()
                move.angular.z = u_theta / 5
                self.vel_pub.publish(move)
                self.update_particles(move)
                self.rate.sleep()

            # The robot is now at goal_pos
            cur_pos = goal_pos

        # Stop robot from moving after traversing the points
        move = Twist()
        move.linear.x = 0
        move.angular.z = 0
        self.vel_pub.publish(move)
        self.update_particles(move)
        self.rate.sleep()

    def odom_callback(self, msg):
        # get pose = (x, y, theta) from odometry topic
        quaternion =
[msg.pose.pose.orientation.x,msg.pose.pose.orientation.y,\
                    msg.pose.pose.orientation.z,
msg.pose.pose.orientation.w]
        (roll, pitch, yaw) =
tf.transformations.euler_from_quaternion(quaternion)
        self.pose.theta = yaw
```

```
        self.pose.x = msg.pose.pose.position.x
        self.pose.y = msg.pose.pose.position.y

if __name__ == '__main__':
    robot = Turtlebot3()
```