Assignment 2
Publishers and Subscribers

EECS 4421, Intro. to Robotics

Course Instructor: Michael Jenkin
Teaching Assistant: Robert Codd-Downey

Report Contributors

Ramavath, Sai Varun	213506936, varun333@my.yorku.ca
Patel, Henilkumar	215245707, henil@my.yorku.ca
Rego, Jared	214843874, jaredr14@my.yorku.ca

4th Year
Electrical Engineering
and Computer Science

Contents

List of Figures

Listings

1 Question 1

1.1 Add noise to odometry data

The odometry data published by Gazebo is perfect data that drives the block robot exactly where the user wants it to go without the need for error correction methods. The problem with this is that in the real world, robot sensors are prone to noise and sending noisy signals to the computer on board the rover that can have significant detrimental effects to the location of the robot. This section will cover how to add noise to the perfect odometry data using Python scripts that would read in /odom messages, extract the distance travelled in each of the x, y and theta channels, add noise to each of those channels and publish the message to a new topic /noisy_odom.

The first step in this process is to read in /odom data. A subscriber must be created for this that listens to the /odom topic published by Gazebo in an __init__ function. The publisher and any variables required to store variables in the noise adding functions are also initialized here.

```python
        self.pub.publish(msg)

    def __init__(self):
        self.sub  = rospy.Subscriber("odom", Odometry, self.odom_callback) # Set the subscriber.
        self.pub  = rospy.Publisher('noisy_odom', Odometry, queue_size=10) # Set the publisher.
        self.msg  = Odometry() # Instantiate variable that stores the Odometry data from /odom.
        self.last_odom = None # Odometry variable.
        self.pose = [0.0, 0.0, 0.0] # x, y and rotational pose array.
        self.std = 0.10 # Standard deviation of noise per meter.
        self.rate = rospy.Rate(0.9) # /odom is published to at rate of 20 Hz.
```

Listing 1: __init__ function that initializes the subscriber, publisher and any variables that are needed to store data globally.

The subscriber initiation has a callback feature that runs everytime /odom receives a publication from Gazebo, which can be used to extract the current odometry information by assigning the message to the self.msg variable.

```python
    def odom_callback(self, msg):
        self.msg = msg
```

Now, the self.msg contains the perfect odometry data that we want and can be referenced anywhere in the rest of the class to process the information; for example, in a function that produces the required zero mean Gaussian noise.

```python
    # This section is inspired by code from Xuan Sang's article on adding noise to odom data at: https://blog.lxsang.me/post/id/16
    def compute_noise(self, msg):
        # Determine the perfect quaternion data from /odom.
        quat = [msg.pose.pose.orientation.x,
                msg.pose.pose.orientation.y,
                msg.pose.pose.orientation.z,
                msg.pose.pose.orientation.w]
        (r, p, thetac) = tf.transformations.euler_from_quaternion(quat)
```

Listing 3: Processing the odometry data assigned to self.msg in the callback function.

The problem statement requires the added noise to be a function of the distance travelled by the robot since the previous /odom message, which can be computed by first storing the perfect /odom data in a variable called self.last_odom and thereon using its value as a reference to the previous position of the robot in the simulated world. Also, the first corrupted message is simply the current perfect /odom message which can be obtained by checking if this is the first message received.

```
        if (self.last_odom == None):
            self.last_odom = msg
            self.pose[0] = msg.pose.pose.position.x # Current x position.
            self.pose[1] = msg.pose.pose.position.y # Current y position.
            self.pose[2] = thetac # Current deviation angle.
            self.pub.publish(self.last_odom)
```

Listing 4: Store perfect /odom data for distance calculations.

The required distance can be calculated using simple geometrical relations, since the x and y components are related by the rotational pose of the robot at any given point in time. The one consideration that needs to be made in this section is that the current angular pose of the robot is a culmination of the robot's previous angular pose and the angle of the headed direction w.r.t the origin. Here dist_rot1 is the rotational distance from the previous robot pose to the current robot pose, and dist_rot2 is the total angular deviation from the origin to the current robot's pose.[1]

```
        else:
            # Compute the distance travelled since previous reading.
            dist_x = msg.pose.pose.position.x - self.last_odom.pose.pose.position.x
            dist_y = msg.pose.pose.position.y - self.last_odom.pose.pose.position.y
            dist_hyp = math.sqrt(dist_x ** 2 + dist_y ** 2)

            # Extract previous quaternion data; this is perfect /odom data used to
    calculate angular deviation since previous reading.
            quat = [self.last_odom.pose.pose.orientation.x,
                    self.last_odom.pose.pose.orientation.y,
                    self.last_odom.pose.pose.orientation.z,
                    self.last_odom.pose.pose.orientation.w]
            (r, p, thetap) = tf.transformations.euler_from_quaternion(quat)

            # Compute the angular deviation since previous reading using x and y
    distances.
            dist_rot1 = math.atan2(dist_y, dist_x) - thetap
            dist_rot2 = thetac - thetap - dist_rot1
```

Listing 5: Compute the distance travelled in x, y and theta.

[1] *Adding noise to odometry data published by the Gazebo simulator*, Xuan Sang, https://blog.lxsang.me/post/id/16

The next step in the process, then, is to add the zero mean Gaussian noise to the distances computed above which can be done simply by sampling a zero mean, (distance * std) standard deviation function; the std here is set to 10 different values to gauge the impact of noise on the robot's path. The self.pose list stores the corrupted data, and it is required that subsequent corrupted data is a summation of the previous corrupted data and the current distance travelled.

```
            # Add noise to the distances computed above.
            dist_hyp  += np.random.normal(loc=0,
                                          scale=math.fabs(dist_hyp * self.std))
            dist_rot1 += np.random.normal(loc=0,
                                          scale=math.fabs(dist_rot1 * self.std))
            dist_rot2 += np.random.normal(loc=0,
                                          scale=math.fabs(dist_rot2 * self.std))

            # Compute the individual channels from absolute data above.
            self.pose[0] += dist_hyp * cos(thetap + dist_rot1)
            self.pose[1] += dist_hyp * sin(thetap + dist_rot1)
            self.pose[2] += dist_rot1 + dist_rot2
```

Listing 6: Add noise to the computed distances

Finally, the corrupted data must be published to the /noisy_odom topic and this has been done by overwriting the msg packet that was received earlier from /odom, since its value is not required anymore.

```
            # Store current perfect /odom data for distance calculations in next
    iteration.
            self.last_odom = msg

            # Overwrite the original /odom message for publishing.
            msg.pose.pose.position.x = self.pose[0]
            msg.pose.pose.position.y = self.pose[1]

            eqt = tf.transformations.quaternion_from_euler(0, 0, self.pose[2])
            msg.pose.pose.orientation.x = eqt[0]
            msg.pose.pose.orientation.y = eqt[1]
            msg.pose.pose.orientation.z = eqt[2]
            msg.pose.pose.orientation.w = eqt[3]

        self.pub.publish(msg)
```

Listing 7: Publish the corrupted Odometry data to /noisy_odom.

The functions are executed using a __main__ function that runs continuously while the node is running. Note how the rate function is set to sleep for a specified amount of time before running the compute_noise function; this is to ensure that the first message has been properly received and that the default values of the variables are not being set as the first /noisy_odom message. The code in this section has been inspired by course code from the drive_to_goal.py file provided in the chapter 2 cpmr zip.

```
if __name__ == '__main__':
    # Initialize the noisy_odom node.
    rospy.init_node('noisy_odom_node', anonymous=False)
    # Initialize an instance of the NoisyOdometry() class.
```

```python
    odom = NoisyOdometry()
    # Run the compute_noise function while the node is active.
    try:
        # Wait a bit to get the first /odom message.
        odom.rate.sleep()
        while not rospy.is_shutdown():
            # Compute noise and publish to /noisy_odom.
            odom.compute_noise(odom.msg)
            odom.rate.sleep()
    except rospy.ROSInterruptException:
        pass

    # Unregister /noisy_odom.
    rospy.loginfo(f"Unregistering from /odom (Clean shutdown)")
    odom.sub.unregister()
```

Listing 8: Main function for /noisy_odom node.

1.2 Plot different noise levels

This section will cover two script files created for the purposes of examining the effect of noise levels on the robot's path: drive_squares.py and and plot_noise.py.

In order, the driving function is quite straightforward to implement as it only requires publishing to the /cmd_vel topic in sequences of driving forward and rotating which involves writing a Twist command appropriately. The publisher setup is similar to the previous part, except it is specified as /cmd_vel in this case. The code in this section is inspired by Michael Ferguson's code on the ROS wiki's mini_max tutorials.[2]

```python
    def drive_squares(self, cmd_vel):
        # Start driving squares here; using drive_to_goal.py as inspiration.
        twist = Twist()
        twist.linear.x = 0.2
        rospy.loginfo(f"Starting to publish on {cmd_vel}...")
        for d in range(100): # Send forward command for 10 Hz * 10 seconds = 100 iterations
            self.pub.publish(twist)
            self.rate.sleep()

        twist = Twist()
        twist.angular.z = -0.5236 # deg2rad(30) clockwise; requires 3 seconds to turn 90 degrees.
        rospy.loginfo(f"Starting to turn at rate of {twist.angular.z} rad/s...")
        for a in range(34):
            self.pub.publish(twist)
            self.rate.sleep()
```

Listing 9: Function to drive the robot in squares.

Note how the amount of time required to turn the robot 90 degrees is just north of the required 3 seconds here; this is because the robot was not able to complete a 90 degree rotation in 3 seconds. A better approach would be to

[2]http://docs.ros.org/en/groovy/api/mini_max_tutorials/html/square_8py_source.html

check the amount of rotation and stop rotating when the robot had reached the target, but this is a very close approximation.

The function showcased above can be called in a __main__ function to perform the required steps to drive a square, since the drive_squares function only does this one time.

```python
if __name__ == '__main__':
    # Instantiate the drive_squares node and create Class object.
    rospy.init_node("drive_squares")
    driver = DriveSquares()

    # Grab the current command velocity topic and pass that to the drive_squares function.
    cmd_vel = rospy.get_param("~cmd_vel", "cmd_vel")
    for i in range(4):
        driver.drive_squares(cmd_vel)

    # Stop the robot after completing a full square.
    twist = Twist()
    driver.pub.publish(twist)
```

Listing 10: Driving the robot in squares.

The robot is now capable of driving in squares and we can move on to plotting the robot's path to examine the impact of the added noise.

The plot_noise.py script handles plotting the noisy path and the clean path of the robot by subscribing to both the perfect /odom and corrupted /noisy_odom topics.

```python
    def __init__(self):
        self.sub_noisy = rospy.Subscriber("noisy_odom", Odometry, self.odom_callback_noisy) # Set the subscriber.
        self.sub_perfect = rospy.Subscriber("odom", Odometry, self.odom_callback_perfect) # Set the subscriber.
        self.msg_noisy = Odometry() # Instantiate variable that stores the Odometry data from /noisy_odom.
        self.msg_perfect = Odometry() # Instantiate variable that stores the Odometry data from /odom.
        self.noisy_data = [] # Store the /noisy_odom data while driving.
        self.perfect_data = [] # Store the /noisy_odom data while driving.
        self.rate = rospy.Rate(20) # Odometry data is published at rate of 20 Hz.
```

Listing 11: Subscribing to the perfect and noisy odometry topics.

The noisy and perfect data are both stored in lists that can be appended to with odometry data from the robot's entire path as it traverses a square (self.noisy_data, self.perfect_data). Both subscribers have callback functions that follow similar functionality as the previous part and assign the incoming odometry messages to the two variables created to store them (self.msg_noisy, self.msg_perfect).

```python
    def odom_callback_noisy(self, msg):
        self.msg_noisy = msg
    def odom_callback_perfect(self, msg):
        self.msg_perfect = msg
```

Listing 12: Callback functions to store odom messages.

Now that we are receiving the odometry messages, we need to extract the x, y and theta channels (much like the previous part) and append them to the lists created above to store this data; this is done in an extract_data function.

```python
    def extract_data(self, msg, msg2):
        # Extract data from the /noisy_odom Odometry messages.
        x = msg.pose.pose.position.x
        y = msg.pose.pose.position.y
        quat = [msg.pose.pose.orientation.x,
                msg.pose.pose.orientation.y,
                msg.pose.pose.orientation.z,
                msg.pose.pose.orientation.w]
        (r, p, theta) = tf.transformations.euler_from_quaternion(quat)

        # Store the data in list for plotting.
        self.noisy_data.append([x, y, theta])

        xp = msg2.pose.pose.position.x
        yp = msg2.pose.pose.position.y
        quat = [msg2.pose.pose.orientation.x,
                msg2.pose.pose.orientation.y,
                msg2.pose.pose.orientation.z,
                msg2.pose.pose.orientation.w]
        (r, p, thetap) = tf.transformations.euler_from_quaternion(quat)

        # Store the data in list for plotting.
        self.perfect_data.append([xp, yp, thetap])
```

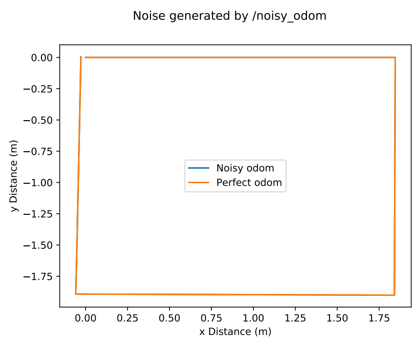
Listing 13: Extracting odometry data and appending to the created lists.

The nuance in this script file is that the data extraction occurs while the robot is driving, and stops when rospy.is_shutdown() is true (user inputs Ctrl-C). This allows us to first obtain all the data in real time and then process it afterwards to ensure that no packets are missed during collection.

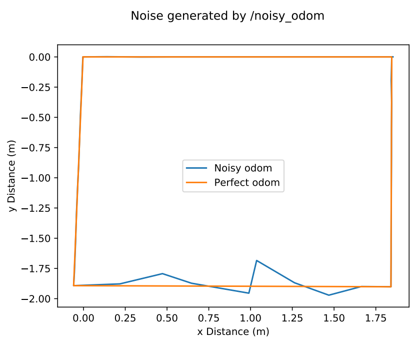
```python
    def plot_noise(self, noisy_data, perfect_data):
        # Convert self.noisy_data to a 2D array containing /noisy_odom data.
        data = np.reshape(np.array(noisy_data), (-1, 3))
        datap = np.reshape(np.array(perfect_data), (-1, 3))

        # Create line to plot theta value of odometry data.
        # Plot robot path.
        fig, ax = plt.subplots()
        ax.plot(data[:, 0], data[:, 1], datap[:, 0], datap[:, 1])
        fig.suptitle('Noise generated by /noisy_odom')
        ax.legend(['Noisy odom', 'Perfect odom'])
        plt.xlabel('x Distance (m)')
        plt.ylabel('y Distance (m)')
        plt.show()
```

Listing 14: Function that plots the data published to /odom and /noisy_odom.

The plot function first converts the list into an array of arrays (2D array) which is then used with the matplotlib Python library for an intuitive approach at plotting in Python. The array's first column contains all the x values, second column all the y values and last column all the theta values. Both the perfect and noisy data are plotted on the same axis to see exactly how much the noise affects the path compared to the perfect odometry data. As expected, the plot



(a) std = 0.00



(b) std = 0.10

Figure 1: The plot for std = 0 shows how there is no noise in the signal whatsoever, as the noisy blue line is not visible.

for std = 0 pertaining to zero added noise (Fig. 1a) shows no deviation in the robot's path and as explained earlier, the slight increment in time required to turn 90 degrees is a very close approximation since the path shows a near perfect square. The noise becomes an issue with a standard deviation of std = 0.10 (Fig. 1b), where at a certain point the robot's noisy path is quite far off from the perfect path.

The plot does not appear to show the proper result, however, since the expectation is to see the path deviate significantly even before reaching the first turning point; it is unclear what caused this error in the plotting but many debugging attempts were fruitless. Further work is being done on this to figure out a solution to the problem.

1.3 Create beacons

In order to generate the structure of the beacons, a gazebo description of their shape is required. This was achieved in a slightly different manner than used in the previous assignment. Specifically, the cylindrical mesh of the beacons was modeled in the open-source blender animation program and exported in the COLLADA file format (.dae) which transcribes the graphical model from blender to an xml description compatible with ROS. The generated 'beacon.dae' file provides a description of the geometry of the beacons. The rest of the xml description of the beacons follows the same general structure as the a URDF description and includes definitions for physical properties such as the coefficients of friction as well as inertia and mass. The main difference was the description of the geometry, which was simplified greatly. Specifically, in order to define the geometry for the beacon, all that was required was to reference the 'beacon.dae' file in the xml description of the beacon as per Fig. 2. This uses the mesh described by the 'beacon.dae' file as the shape for the beacon.

```
<geometry>
   <mesh>
   <uri>model://beacon/meshes/beacon.dae</uri>
   </mesh>
</geometry>
```

Figure 1: Using a .dae file generated from blender simplifies definition of the geometry of the shape, using this method, complex shapes can be modeled visually on blender and imported into a gazebo compatible xml description

The resulting beacon shape is shown in figure 2.

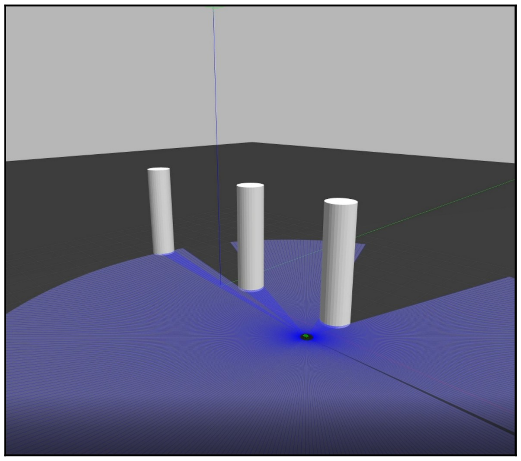


Figure 2: Overall beacon structure. The beacons are have a radius of 0.25m and a height of 2m. These numbers can be adjusted by modifying the factors in the beacon.dae file as shown in figure 3

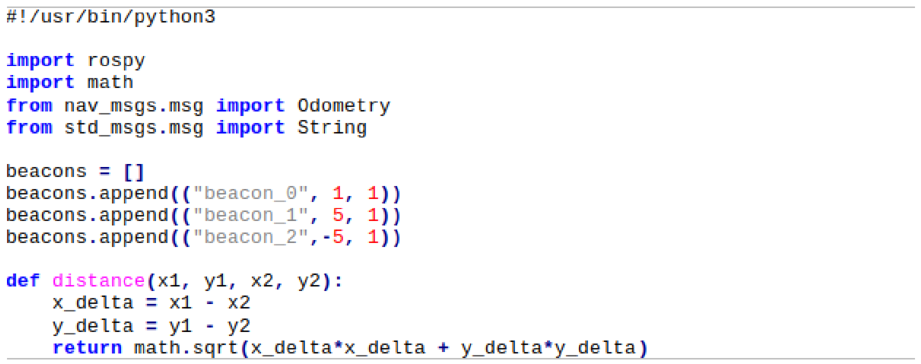
```
<library_visual_scenes>
   <visual_scene id="Scene" name="Scene">
     <node id="Cylinder" name="Cylinder" type="NODE">
       <matrix sid="transform">0.5 0 0 0 0 0.5 0 0 0 0 2 0 0 0 0 1</matrix>
       <instance_geometry url="#Cylinder-mesh" name="Cylinder"/>
     </node>
   </visual_scene>
 </library_visual_scenes>
```

Figure 3: By modifying the 'transform' parameters in the highlighted line of the beacon.dae mesh, the size of the beacon can very easily be altered.

Figure 2: Relevant details for question 1.3

1.4 Measuring distance from beacons

To measure the distance from each beacon to the position of the robot, the two required parameters are the location of each beacon, as well as the location of the robot. The robot's location is derived from the /odom topic, which provides the current position of the robot based on the wheel odometer, while the location of the beacons is provided and fixed in the world. As such, the distance can be calculated by using the formula for a distance between 2 points. The python code in the figure defines a function that calculates the distance between 2 points, as well as defines the locations of the N=3.



```python
#!/usr/bin/python3

import rospy
import math
from nav_msgs.msg import Odometry
from std_msgs.msg import String

beacons = []
beacons.append(("beacon_0", 1, 1))
beacons.append(("beacon_1", 5, 1))
beacons.append(("beacon_2",-5, 1))

def distance(x1, y1, x2, y2):
    x_delta = x1 - x2
    y_delta = y1 - y2
    return math.sqrt(x_delta*x_delta + y_delta*y_delta)
```

Figure 3: Python code implementing calculation of distance between two points.

In this code, the import definitions import the description of the data formats for /odom topic as well as the std_msg format to be published. The rospy and math libraries provide functions to interface with ros and basic math calculations. The beacons array holds the position and name of each beacon (from 0 to 2 for a total of 3 beacons).

In the code highlighted in Fig. 4, the current position of the robot is defined by addressing by accessing the position tag of the /odom message and stored in the corresponding x or y variables. In order to publish to 3 topics, 3 publishers are defined. Publish_b0 represents beacon_0 and prints to a topic labelled beacon_0 with a string data type. Publish_b1 and publish_b2 represent beacon_1 and beacon_2 and print to topics with labelled beacon_1 and beacon_2 respectively. The for loop iterates over the list of beacons and locations and calculates the distance between all three beacons and the robot at all positions that the robot traverses. The if statements select the correct topic to publish to based on the current beacon to robot distance being calculated in the for loop. Finally, the main function initializes the rosnode and subscribes to the /odom topic to receive the odometer messages from which the distance is calculated, while the final rospy.spin() line runs the node until the code crashes or is interrupted through the terminal. The operation of the node is shown in the video at the following link: https://youtu.be/aZc4JV4-S2M

```
def callback(msg):
    x = msg.pose.pose.position.x
    y = msg.pose.pose.position.y

    publish_b0 = rospy.Publisher('beacon_0', String, queue_size=10)
    publish_b1 = rospy.Publisher('beacon_1', String, queue_size=10)
    publish_b2 = rospy.Publisher('beacon_2', String, queue_size=10)

    for beacon_name, beacon_x, beacon_y in beacons:
        dist = distance(x, y, beacon_x, beacon_y)

        #rospy.loginfo('current robot position x:{}, y:{}'.format(x, y))
        #rospy.loginfo('distance from {}, d = {}'.format(beacon_name, dist))

        data_str = 'source: {}, position: x:{} y:{}, distance:
{}'.format(beacon_name, beacon_x, beacon_y, dist)

        if beacon_name == "beacon_0":
            publish_b0.publish(data_str)
        if beacon_name == "beacon_1":
            publish_b1.publish(data_str)
        else:
            publish_b2.publish(data_str)

def main():

    rospy.init_node('track_robot')
    rospy.Subscriber("/odom", Odometry, callback)
    rospy.spin()

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

Figure 4: Callback function to handle /odom data and create publishers

2 Question 2

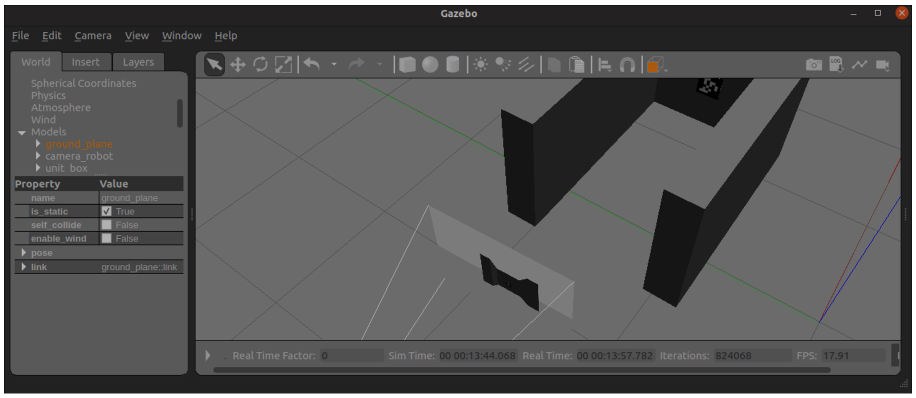
2.1 Create a docking station

In gazebo the U-shaped docking station was created with 3 cube instances and the ARUCO tag was placed in the middle of the back of the U. The cube objects is_static value were set to true. In Fig. 5 the robot's camera simulation can be seen having the whole docking station in view. Unfortunately, when trying to save the gazebo world as a file, gazebo would freeze and become unresponsive.

2.2 Use OpenCV target detection

Lighting was added to the gazebo environment to illuminate the docking station, you can see a different between the shades of colour between Fig. 5 and Fig. 7. The code provided inside the cpmr_ch5 scripts folder aruco.py, canny_edges.py, good_features.py, harris_corners.py and view_camera.py were run with the camera robot and the following windows were presented.

The closest the camera robot got to the wall was as close as the ARUCO target was still fully visible in the camera view. Some failures were that the larger radiuses for example 10xradius, the camera was unable to locate the



Property	Value
name	ground_plane
is_static	✓ True
self_collide	False
enable_wind	False
pose	
link	ground_plane::link

```
Real Time Factor: 0   Sim Time: 00 00:13:44.068   Real Time: 00 00:13:57.782   Iterations: 824068   FPS: 17.91
```

Figure 5: U-shaped docking station created with 3 cube objects and ARUCO tag positioned in the middle of the furthest most cube.

ARUCO target even if it was directly in front of the camera.

2.3 Create a distractor object

The texture in 8 would have been used as the Gazebo distractor object on the back wall of the docking station. The ARUCO tags are very robust and this would not have done much to the camera robots' ability to trace the ARUCO target since the harris_corners.py script would a lot of the dots as corners or white squares. This would still allow the robot to preform as it usually does. But like before if the robot were further away it would become increasingly difficult for the ARUCO tags to be located. The worst possible texture would be a texture that makes the ARUCO target into a part of another ARUCO target which the robot will only be able to recognize the larger more pronounce target instead of the smaller one we would want it to be tracking. So, the ARUCO tag that we want it to track would be seen as a white square in the larger ARUCO tag. This would also lead to the robot thinking it is closer to the docking station then it really is since the ARUCO targets are supposed to be a fixed size, leading to improper docking of the camera robot.

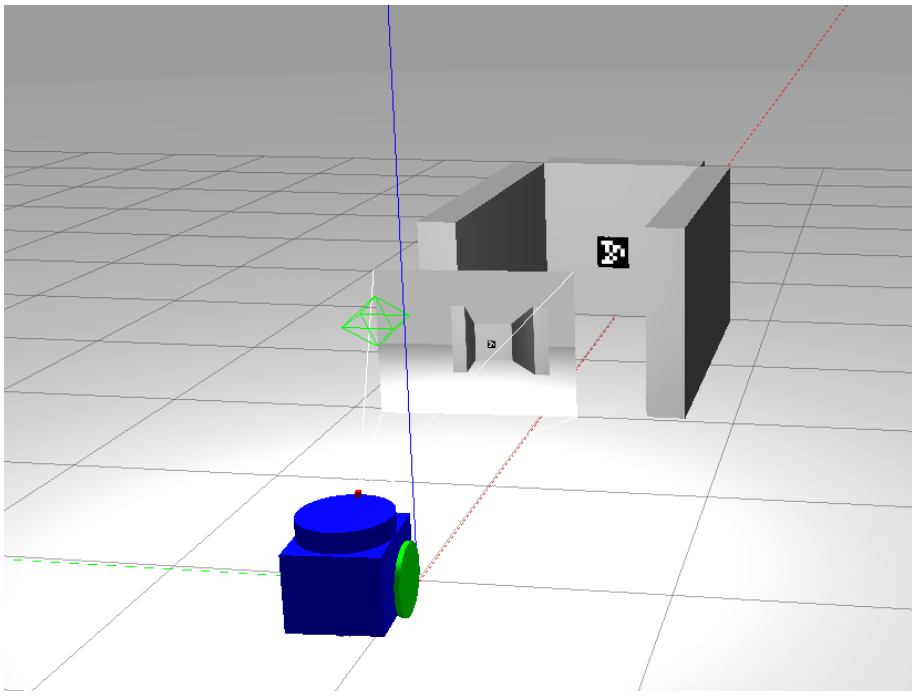
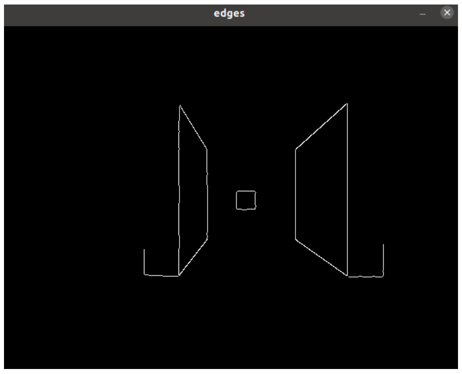
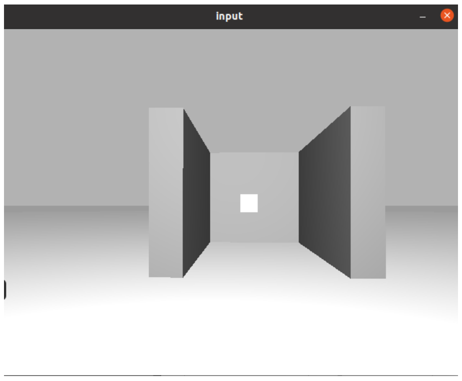


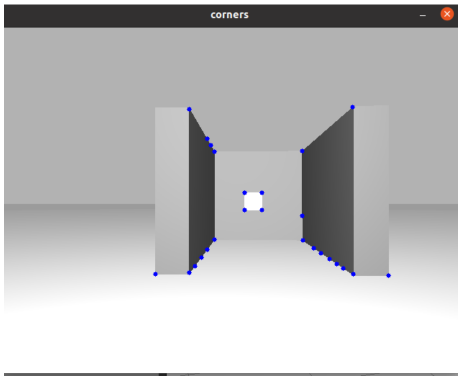
Figure 6: Estimated position farthest from docking station to still be successful.



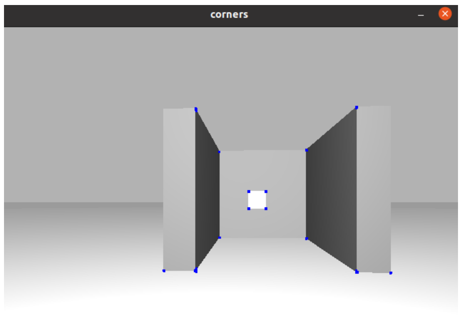
(a) Canny edges 1



(b) Canny edges 2



(c) Good features



(d) Harris corners

Figure 7: U-shaped docking station created with 3 cube objects and ARUCO tag positioned in the middle of the furthest most cube.

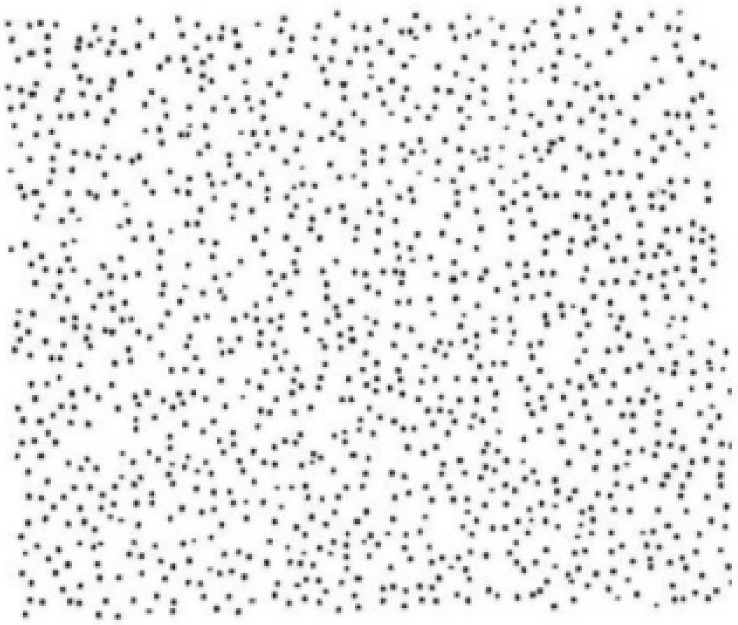


Figure 8

A Appendix

The code for this report can be found at this Github repository:
https://github.com/svarunr/IntroRobotics_York.git