Lab report for CISC 3060 Lab Assignment #4

Fall 2022

Jan C. Bierowiec

1.0 Objective

The objective of lab assignment #4, is to explore having a robot carry out a pre-planned path. Taking a step back and looking at the objective from afar seems straightforward. The user inputs points for the robot to travel on the path and the robot goes to each point in the path. There are a few issues that need to be addressed however. One of the issues is that a preplanned path is dense with points with very little distance between them. This includes a path that can be a straight line for the robot to travel, or for a more extreme case, when a robot follows a path with sharp turns. The locomotion mechanism of the robot, i.e. the wheels and motors may swivel, causing the robot to stray off track causing it to not follow the path exactly.

2.0 Robot Following a Preplanned Path

2.1 Step 1

For Step 1 of the lab, five files were downloaded from Blackboard. Three of them were pp.py, astar.py, and pp_encmap.py. The first two, pp_encmap.py and astar.py are path-planning files. These path-planning files use an occupancy-map based tree search algorithm discussed in class. The astar.py file differs however from the bfs.py file gone over in class in that the astar.py file uses a more efficient tree search algorithm called A*. The file pp_encmap.py is similar to the maps1.py file downloaded in class to test the path planner. This file however automatically loads the enclosed_ocmap.txt file into a numpy array gMap and it includes a function planPath(start, goal) which facilitates to call the oath planner on an (x, y) start and end couple. planPath displays the path overlayed on the occupancy grid and returns a list of paths. The pp.py file includes two parts. The main program within the file calls the path planner for a pre-specified start and end position. In the file this was set to (2, 0.01) and (2, -4) respectively. Once it starts to move towards those points the follower node will issue points on the oath one by one within the movegoal topic. The MoveIt.py node was also downloaded from Blackboard as well in order for the MoveIt node to work in the program.

These files were downloaded into the same directory, entitled CISC3060, and then three linux shells were opened. One shell was opened to open the TurtleBot Gazebo. Another shell

was then opened and the MoveIt.py node was started. In the third shell, pp.py was opened. Once the files were opened the planned path was shown. It was similar, if not exact to the one provided in the lab instructors. Figure I shows the two plots. These maps have been dilated to allow for the consideration of a non-point robot. The blue central obstacle is the dilated center of the wall. The red path is the planned robot path. Once the planned path window was shut down, pp.py starts to issue the (x, y) goals to the MoveIt node through the movegoal topic. Figure II shows snapshots of the robot moving along the path from start to finish. (In chronological order from a to d) The pp.py file also records every position that the robot has occupied (the gTrackX and gTrackY are global lists that update in the positionCallback function of pp.py). Once the path finishes, pp.py displays a matplotlib window with a scatter graph of these locations, as shown in Figure III. At the top, the disconnected part of the path in Figure III results from the robot moving from (0,0) to the start location (2,0.1) as the first step in pp.py. Once this step 1 was completed, with everything working and the results matching those provided in the lab instructions, step 2 was started.

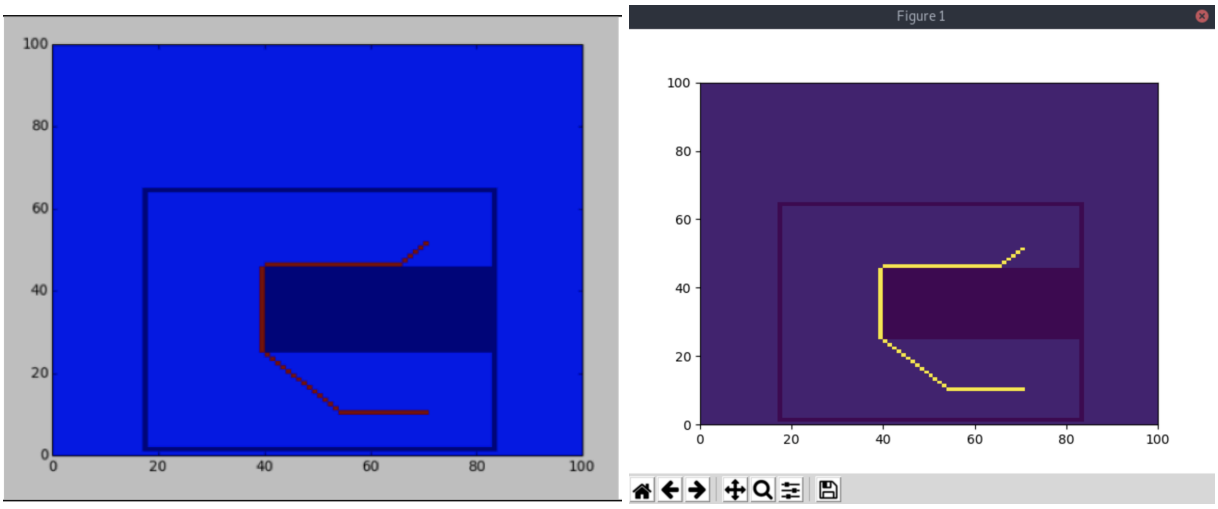


Figure I: The plot provided in the lab instructions is present on the left, and the plot generated following the steps is on the right. The graphs are identical regarding the path of the robot, and not including the color or other additional properties of the graph.

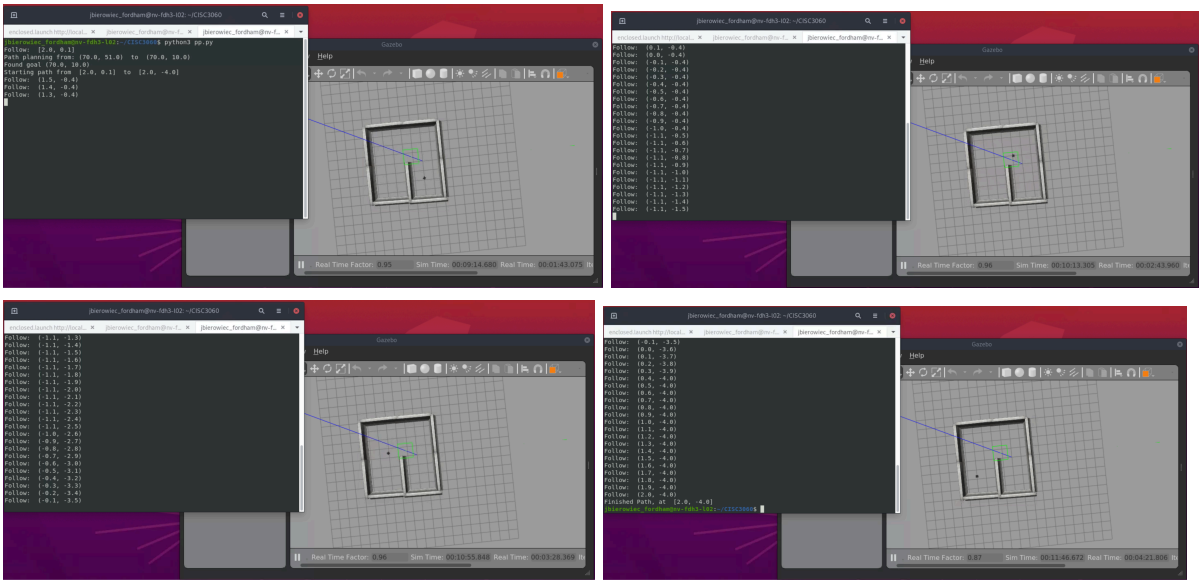


Figure II: Here are screenshots of the robot traveling from the starting point to the goal point, and the coordinates generated, where the robot was found at each point traveling its pre-planned path.

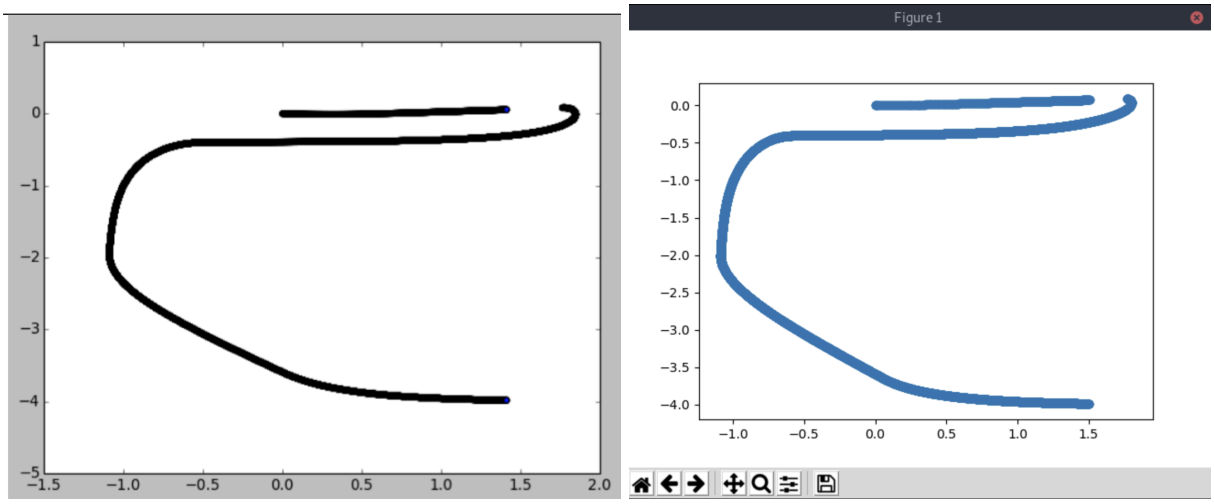


Figure Iii: The graph provided in the lab instructions is present on the left, and the graph generated following the steps is on the right. The graphs are identical regarding the path of the robot, and not including the color or other additional properties of the graph.

2.2 Step 2

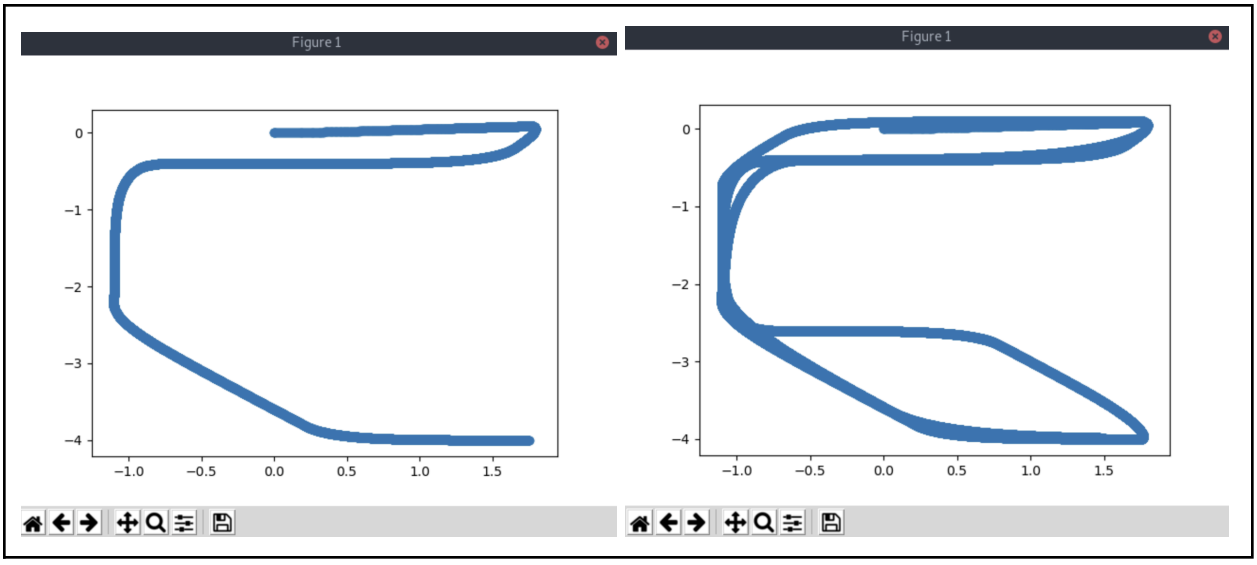
The follower node sequences the (x, y) locations to the *movegoal* topic by checking to see when the robot has approached the previous goal. This point in the code is shown below in Figure IV. The variable “pursuit” present in the figure above, is what is responsible for controlling how close the robot has to be to each point, before the next point is issued. This variable also controls a lot of the features of the path that is generated in Figure III including how closely it follows the planned path Figure I. For this part of the lab, “pursuit” was modified for the values to be in the range of {0.25, 0.3, 0.5, 1.0, 2.0}. This was done by changing the code within the pp.py file. The changes can be seen in the main section of the program shown in Figure V below. The default value for pursuit which was set to 0.5 in the follower node was uncommented to not interfere with the pursuit array values in the main function. Figure VI shows the generated plots of the results for each value in the pursuit range.

```
78              msg.x,msg.y = goalx,goaly
79              pub.publish(msg)
80              while np.hypot(gLoc[0]-goalx,gLoc[1]-goaly)>pursuit:
81                  rate.sleep()
```

Figure IV: This is where the follower node sequences the (x, y) locations to the movegoal topic which checks to see if the robot approached the previous goal.

```
#------------------------------MAIN  program---------------------
if __name__ == '__main__':
    try:
        # pursuit values in an array
        pursuit = {0.25, 0.30, 0.50, 1.0, 2.0}
        # for loop is created to take in starting values and goal values passing in
        # each number in the pursuit array
        for p in pursuit:
            start=[2.0, 0.1]
            goal=[2.0, -4.0]
            # follow node now takes in two parameters as opposed to one
            # one for the sart value and the other for the pursuit values
            follow_node([start], p)
            path = pp_encmap.planPath(start, goal)
            print("Starting path from ",start," to ",goal)
            # follow_node called again to generate coordinates of robot reaching
            # final point of pre-planned path
            follow_node(path, p)
            print("Finished Path, at ",goal)
            # plot genrated of robot path
            plt.scatter(gTrackX,gTrackY)
            plt.show()
            plt.clf()
            # takes the robot back to the starting point
            pathResart = pp_encmap.planPath(goal, start)
            print("Starting path from ",start," to ",goal)
            follow_node(pathback, p)
            # output message saying robot went back to starting point
            print("Robot returned back to ", start)
```

Figure V: Pursuit array code created within the main program taking in the parameters of the starting point and the pursuit values and generating a graph when the robot travels from the start to the pre-planned point.



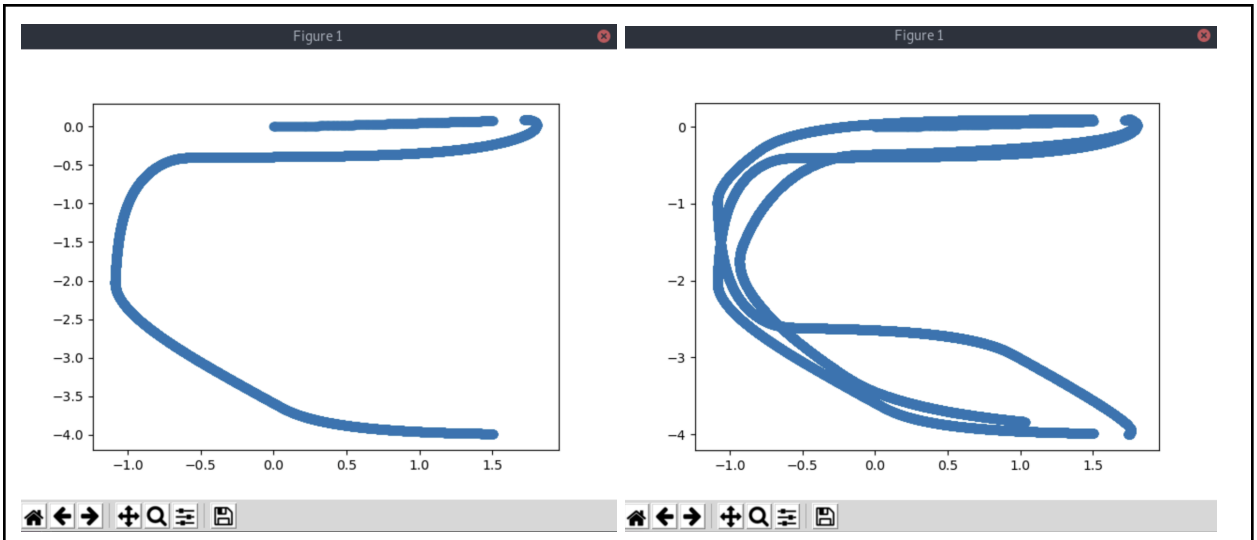


Figure VI: 0.25 pursuit value graph is on the top left corner, 0.3 pursuit value graph is on the top right corner, 0.5 pursuit value graph is on the bottom left corner, 1.0 pursuit value graph is on the bottom right corner. A 2.0 pursuit value graph was not generated.

Through placing the pursuit values in an array and outputting the graphs after each iteration, some pros and cons can be observed of the motion of the robot. One pro of these values was that as the robot increased its pursuit value it traveled faster to the pre-planned path, so in regards to speed it was more efficient. However since the pre-planned path is dense with points with very little distance between them the robot might have missed its proper path generating a graph with a lot of uncertain points with path noise (noise being imperfections in the path). This is a major con in regards to the behavior of the robot. Another pro observed is that although the graphs may have differed with path noise and disconnections in the graph, the graph shape remained relatively the same. This is good because when analyzing these graphs it can be deduced that the algorithm works, and what remains is to find out what the most optimal pursuit value would be for the robot to travel both quickly and reach the pre-planned path point as efficiently and as accurately as possible.

This can be a reason as to why the graphs look the way they do in FigureVI. When the robot traveled at a pursuit value of 0.25, the graph was smooth and there was no disconnection between the robot moving from (0, 0) to the start location of (2, 0.1). Similarly when the robot traveled at a pursuit value of 0.3, there was more noise in the path graph, however the graph was still uniform without any disconnection from (0, 0) to the start location (2, 0.1). It is when the robot reaches pursuit values of 0.5 and higher that there was more noise, along with disconnections from (0, 0) and (2, 0.1). A 0.5 pursuit value generated the same graph as was when the code ran originally with 0.5 being set as the default value in step 1, and although there was not a lot of path noise, there was a disconnection. At a pursuit value of 1.0, the path

noise was significantly larger than that of 0.3, and there was also a disconnection present. Lastly, at a pursuit value of 2.0, no graph was generated as there was a constant bump message generated in the terminal as the robot would hit the center wall. This means that at a pursuit value that was too high the robot would not be able to reach its pre-planned path. This reasoning therefore supports the idea that the higher the pursuit value is, the quicker the robot traveled disconnecting from (0, 0) and the start location, as well as creating path noise within the path, not generating a smooth graph, and reaching a point where no graph was generated at all.

2.3 Step 3

For step 3 of the lab, the follower node was modified in a way that before a new point was issued, the node checks to see if subsequent points are in a line and skips ahead to the last point in the line. This change can be seen in Figure VII, which provides the changed code for the follower node. The first change was done in the main loop which checks for straight lines. This was done creating a while loop looking for the variable current which was by default set to 0 before the while loop. If conditions are met, regarding the current being greater than or equal to 0, then the goalx and goaly check for straight lines in the path. A print statement is then generated outputting to the user where the current is, as well as the path current. Once the while loop goes through the nextCurrentGoal function is called which takes the index of the last point on the current and the path. In this main loop, this then returns the index of the next point to be sent.

```
115    pursuit = 0.5 # how far 'in front' to set the moveit goal
116
117    gLogging = True
118
119    current=0
120
121    while current >= 0:
122        goalx,goaly=path[current][0],path[current][1]
123        print("The current is ", current," and the path current is ",path[current])
124        if np.hypot(gLoc[0]-goalx,gLoc[1]-goaly)>=pursuit:
125            print("Follow: ",goalx,goaly)
126            msg.x,msg.y = goalx,goaly
127            pub.publish(msg)
128            while np.hypot(gLoc[0]-goalx,gLoc[1]-goaly)>pursuit:
129                rate.sleep()
130        current = nexLinearGoal(current,path)
131    gLogging=False
132    return
```

Figure VII: Here is an image of the main loop that was changed in the follower node. The changes include a variable current which looks at whether a current point in a line is straight, and once the while loop makes one iteration the nextLinearGoal function is called.

Following this change in code, a function was created for nextLinearGoal. This function consisted of the slope formula y = mx + b, or in the case of this lab, y = mx + c, with c being the offset of the line. m remains to be the slope of the line in the equation, and y and x are coordinate points on the line. The equation for slope was also included within the function, that being y2 - y1 / x2 - x1. With the robot knowing the slope and offset, then whenever it is given a point (x1,y1), then the robot can determine whether it lies on the line or not through evaluating the equation: r = y1 - m * x1 - c. This (y=mx+c) formula will be zero for a point on the line, and non-zero otherwise. Testing the function, 0.1 was used in the function as the default threshold value. Once the formulas were written, taking in the current and path parameters, the calculations were made, and the output of those calculations were generated to the screen. This was done to keep track of the number of lines the program thinks were produced in the path of the robot. Figure VIII shows an image of the nextLinearGoalFunction.

```
def nextLinearGoal(current,path):
    if len(path) <= 1:
        return -1

    # prints the length of the path
    print(len(path))

    # default values for the sume of r, the index, and the threshold
    threshold = 0.1
    rsum = 0
    goalIndex = current + 1

    print("Index =",goalIndex)

    # x1 and y1 initial points
    x1 = path[current][0]
    y1 = path[current][1]
    print(x1, y1)

    # th eindex is incremented by if the current path equals the index path
    while path[current] == path[goalIndex]:
        goalIndex+=1
    print("Index point is found at ",path[goalIndex])

    # x2 and y2 are the end points
    x2 = path[goalIndex][0]
    y2 = path[goalIndex][1]
    print(x2, y2)

    # if statement to check if x2 and x1 are the same
    if x2-x1 == 0:
        while path[current][0] == path[goalIndex][0]:
            goalIndex+=1
        current = goalIndex
        return current

    # slope formula
    m = (y2-y1)/(x2-x1)
    print("The slope of the line is (",y2, " - ",y1,")/(", x2 , " - ", x1,") = ",m)

    # adjustment
    c = y1-(m*x1)
    print(c,"=",y1," - ",m," * ",x1)
```

```
    # x2 and y2 are the end points
    x2 = path[goalIndex][0]
    y2 = path[goalIndex][1]
    print(x2, y2)

    # if statement to check if x2 and x1 are the same
    if x2-x1 == 0:
        while path[current][0] == path[goalIndex][0]:
            goalIndex+=1
        current = goalIndex
        return current

    # slope formula
    m = (y2-y1)/(x2-x1)
    print("The slope of the line is (",y2, " - ",y1,")/(", x2 , " - ", x1,") = ",m)

    # adjustment
    c = y1-(m*x1)
    print(c,"=",y1," - ",m," * ",x1)

    # index is incremented
    goalIndex+=1

    # if the rsum is less than the threshold, then the following loop is executed
    while rsum < threshold:
        if goalIndex >= len(path):
            return -1
        while path[current] == path[goalIndex]:
            if goalIndex >= len(path) - 1:
                return -1
            goalIndex+=1

        # path index is printed and the initial x1 and y1 are
        # set to be equal to the index of the path
        print(path[goalIndex])
        x1 = path[goalIndex][0]
        y1 = path[goalIndex][1]

        # formula to determine if a point is on the line or not
        r = y1-(m*x1)-c
        print("r = ", r, " = ",y1," - ",m," * ",x1," - ",c)

        rsum += abs(r)
        print("The sum of r is ",rsum)
        print(goalIndex)
        current = goalIndex
    return current
```

Figure VIII: Images of the nextLinearGoal function with the included equations to calculate the slopes of the line as the robot moves from point to point and printing the messages out to the screen.

When this program was written and compiled, the turtlebot simulation was brought to the screen, MoveIt.py was run, and pp.py was run with this new code. For the first run of the program the threshold value was by default set to be 0.1. When running the program, this produced a tracker graph, which can be seen below in Figure IX. This graph is similar to step 1 and step 2 of the graph in that this tracker graph resembles the graph of when the pursuit value

in the follower node function was set by default equal to 0.5. Of course there is a disconnection towards the top of the graph that results from the robot moving from (0,0) to the start location (2,0.1) as the first step in pp.py. This has a pro and a con. A pro in that the algorithm has not changed but remained consistent in sensing the path of the robot, but a con in regards to the robot tracker graph not being able to output a complete graph, but a disconnected graph from the robots starting location to the ending location.

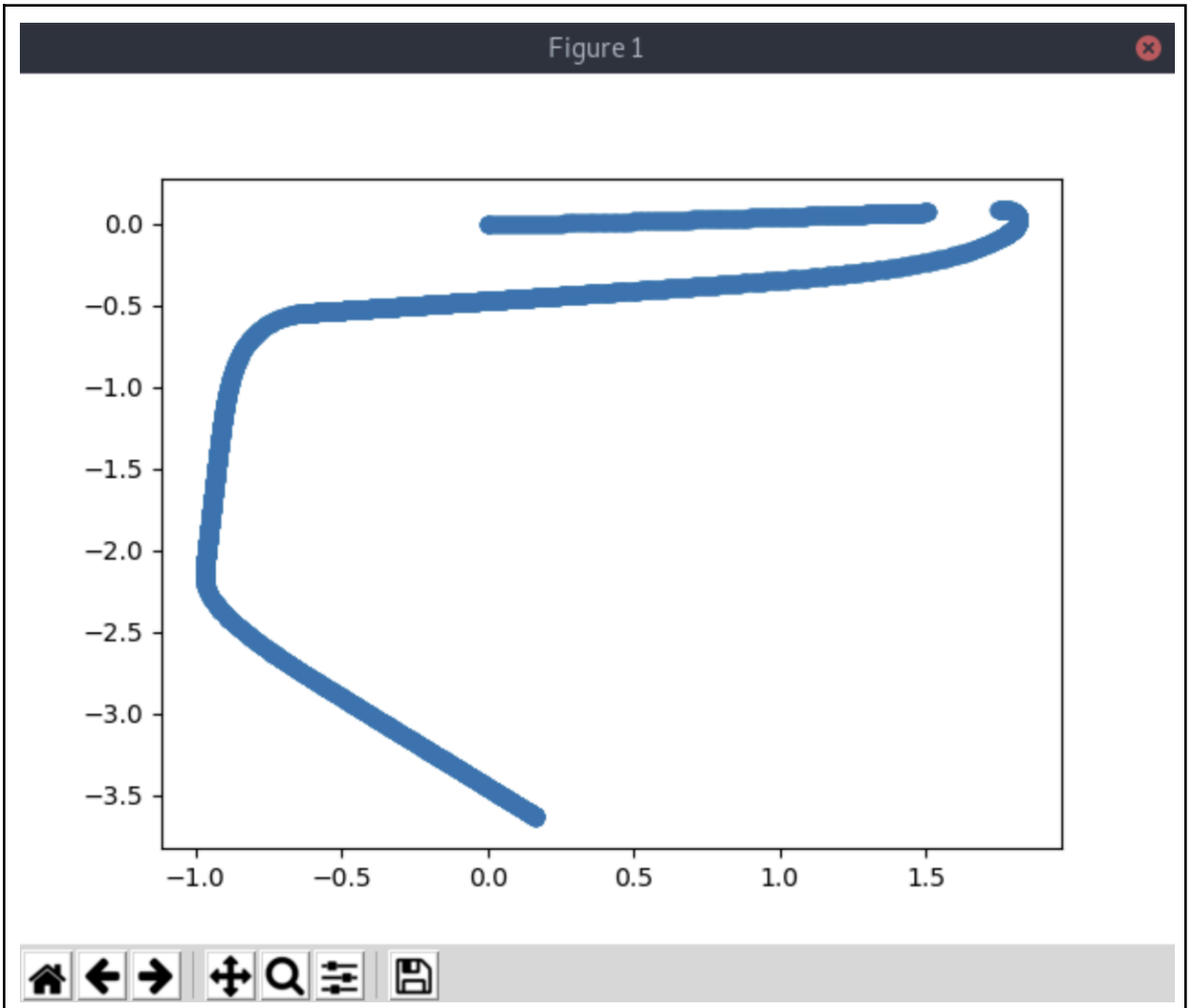


Figure IX: This is an image of the graph generated using the default threshold value 0.1. This graph is very similar to the step 1 and 2 tracker graphs where the pursuit value was set by default equal to 0.5.

This experiment was rerun with different values for the threshold. The values used were 0.25, 0.3 and 0.5 from step 2, the graphs in Figure X show the graphs of the different thresholds in an order from least to greatest. What can be noted is that the graphs do not change, despite

changing the threshold number. The graphs all follow the same path and all of them have a disconnection when the robot travels from the starting point to the ending point. This can be a pro and a con. A con because the graphs still output a disconnection, meaning that the robot at some point as it travels linearly from point to point, there is still some sort of disconnection between points. A pro however can also be observed, because although the graphs are disconnected, they are the same. This is a good observation, because this can help a robot travel a pre-planned path accurately from point to point and at a reasonable speed.

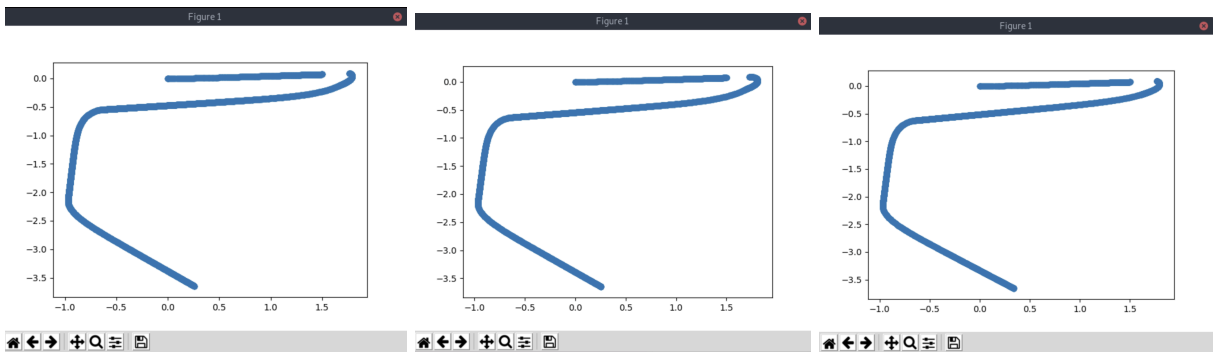


Figure X: Graphs generated of threshold points being set to default in the following order: 0.25, 0.3, and 0.5.

3.0 References

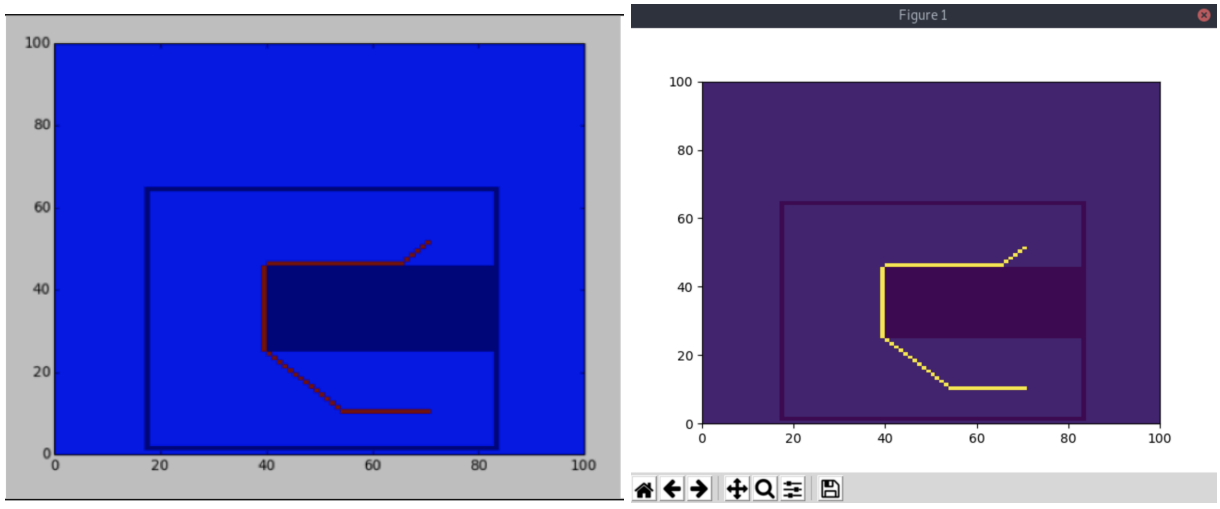


Figure I: Plot generated for step 1 of the lab.

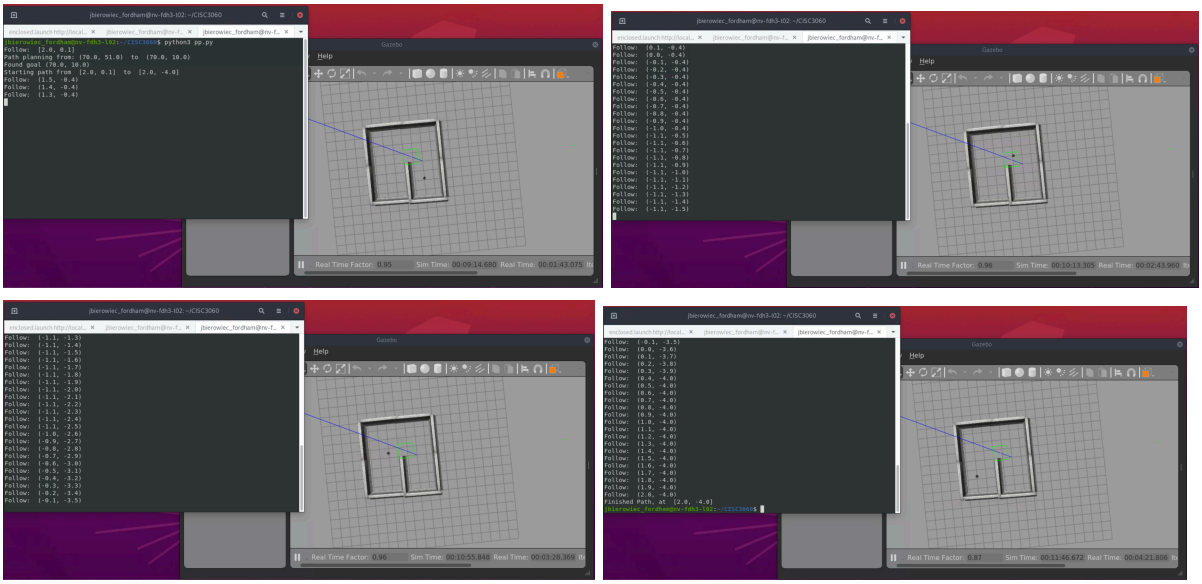


Figure II: Screenshots of the robot moving from its starting to ending point.

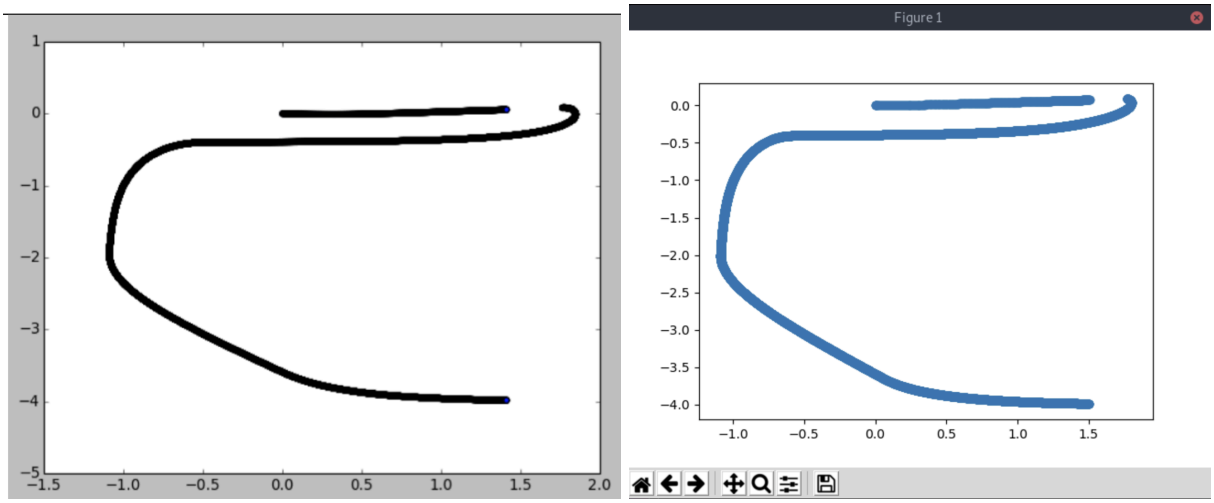


Figure Iii: Tracker graph generated in step 1.

```
78              msg.x,msg.y = goalx,goaly
79              pub.publish(msg)
80              while np.hypot(gLoc[0]-goalx,gLoc[1]-goaly)>pursuit:
81                  rate.sleep()
```

Figure IV: Code where follower node sequences locations to the movegoal topic.

```
85 #------------------------------MAIN  program---------------------
86 if __name__ == '__main__':
87     try:
88         # pursuit values in an array
89         pursuit = {0.25, 0.30, 0.50, 1.0, 2.0}
90         # for loop is created to take in starting values and goal values passing in
91         # each number in the pursuit array
92         for p in pursuit:
93             start=[2.0, 0.1]
94             goal=[2.0, -4.0]
95             # follow node now takes in two parameters as opposed to one
96             # one for the sart value and the other for the pursuit values
97             follow_node([start], p)
98             path = pp_encmap.planPath(start, goal)
99             print("Starting path from ",start," to ",goal)
100            # follow_node called again to generate coordinates of robot reaching
101            # final point of pre-planned path
102            follow_node(path, p)
103            print("Finished Path, at ",goal)
104            # plot genrated of robot path
105            plt.scatter(gTrackX,gTrackY)
106            plt.show()
107            plt.clf()
108            # takes the robot back to the starting point
109            pathResart = pp_encmap.planPath(goal, start)
110            print("Starting path from ",start," to ",goal)
111            follow_node(pathback, p)
112            # output message saying robot went back to starting point
113            print("Robot returned back to ", start)
```

Figure V: Changed pursuit array in the main function of pp.py

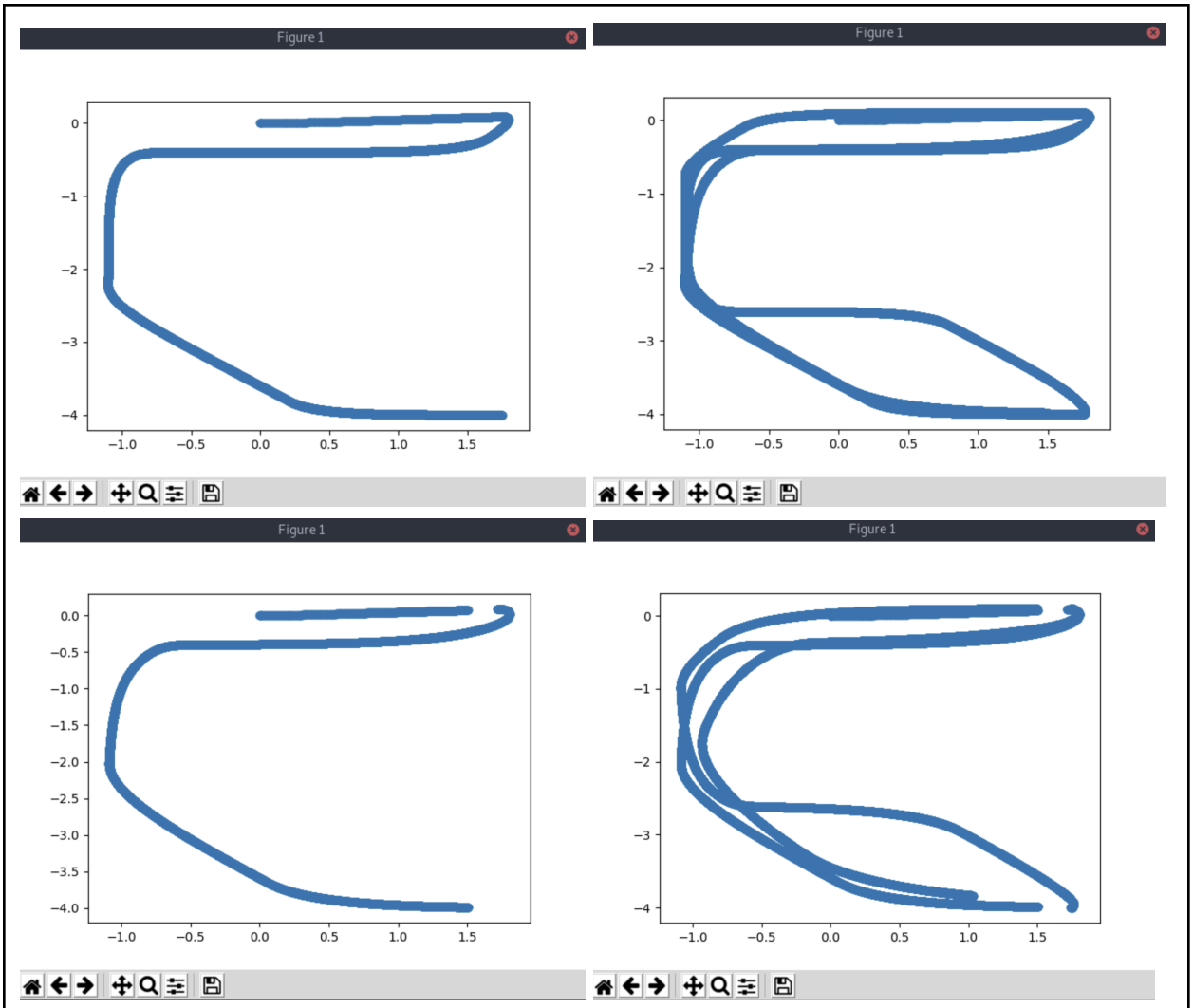


Figure VI: Four graphs generated for pursuits of 0.25, 0.3, 0.5, and 1.0. 2.0 was not generated.

```
    pursuit = 0.5 # how far 'in front' to set the moveit goal

    gLogging = True

    current=0

    while current >= 0:
        goalx,goaly=path[current][0],path[current][1]
        print("The current is ", current," and the path current is ",path[current])
        if np.hypot(gLoc[0]-goalx,gLoc[1]-goaly)>=pursuit:
            print("Follow: ",goalx,goaly)
            msg.x,msg.y = goalx,goaly
            pub.publish(msg)
            while np.hypot(gLoc[0]-goalx,gLoc[1]-goaly)>pursuit:
                rate.sleep()
        current = nexLinearGoal(current,path)
    gLogging=False
    return
```

Figure VII: Changing the main while loop in the follower node in step 3.

```
def nextLinearGoal(current,path):
    if len(path) <= 1:
        return -1

    # prints the length of the path
    print(len(path))

    # default values for the sume of r, the index, and the threshold
    threshold = 0.1
    rsum = 0
    goalIndex = current + 1

    print("Index =",goalIndex)

    # x1 and y1 initial points
    x1 = path[current][0]
    y1 = path[current][1]
    print(x1, y1)

    # th eindex is incremented by if the current path equals the index path
    while path[current] == path[goalIndex]:
        goalIndex+=1
    print("Index point is found at ",path[goalIndex])

    # x2 and y2 are the end points
    x2 = path[goalIndex][0]
    y2 = path[goalIndex][1]
    print(x2, y2)

    # if statement to check if x2 and x1 are the same
    if x2-x1 == 0:
        while path[current][0] == path[goalIndex][0]:
            goalIndex+=1
        current = goalIndex
        return current

    # slope formula
    m = (y2-y1)/(x2-x1)
    print("The slope of the line is (",y2, " - ",y1,")/(", x2 , " - ", x1,") = ",m)

    # adjustment
    c = y1-(m*x1)
    print(c,"=",y1," - ",m," * ",x1)

    # x2 and y2 are the end points
    x2 = path[goalIndex][0]
    y2 = path[goalIndex][1]
    print(x2, y2)

    # if statement to check if x2 and x1 are the same
    if x2-x1 == 0:
        while path[current][0] == path[goalIndex][0]:
            goalIndex+=1
        current = goalIndex
        return current

    # slope formula
    m = (y2-y1)/(x2-x1)
    print("The slope of the line is (",y2, " - ",y1,")/(", x2 , " - ", x1,") = ",m)

    # adjustment
    c = y1-(m*x1)
    print(c,"=",y1," - ",m," * ",x1)

    # index is incremented
    goalIndex+=1

    # if the rsum is less than the threshold, then the following loop is executed
    while rsum < threshold:
        if goalIndex >= len(path):
            return -1
        while path[current] == path[goalIndex]:
            if goalIndex >= len(path) - 1:
                return -1
            goalIndex+=1

        # path index is printed and the initial x1 and y1 are
        # set to be equal to the index of the path
        print(path[goalIndex])
        x1 = path[goalIndex][0]
        y1 = path[goalIndex][1]

        # formula to determine if a point is on the line or not
        r = y1-(m*x1)-c
        print("r = ", r, " = ",y1," - ",m," * ",x1," - ",c)

        rsum += abs(r)
        print("The sum of r is ",rsum)
        print(goalIndex)
        current = goalIndex
    return current
```

Figure VIII: nextLinearGoal function to generate lines between each point as the robot traveled.

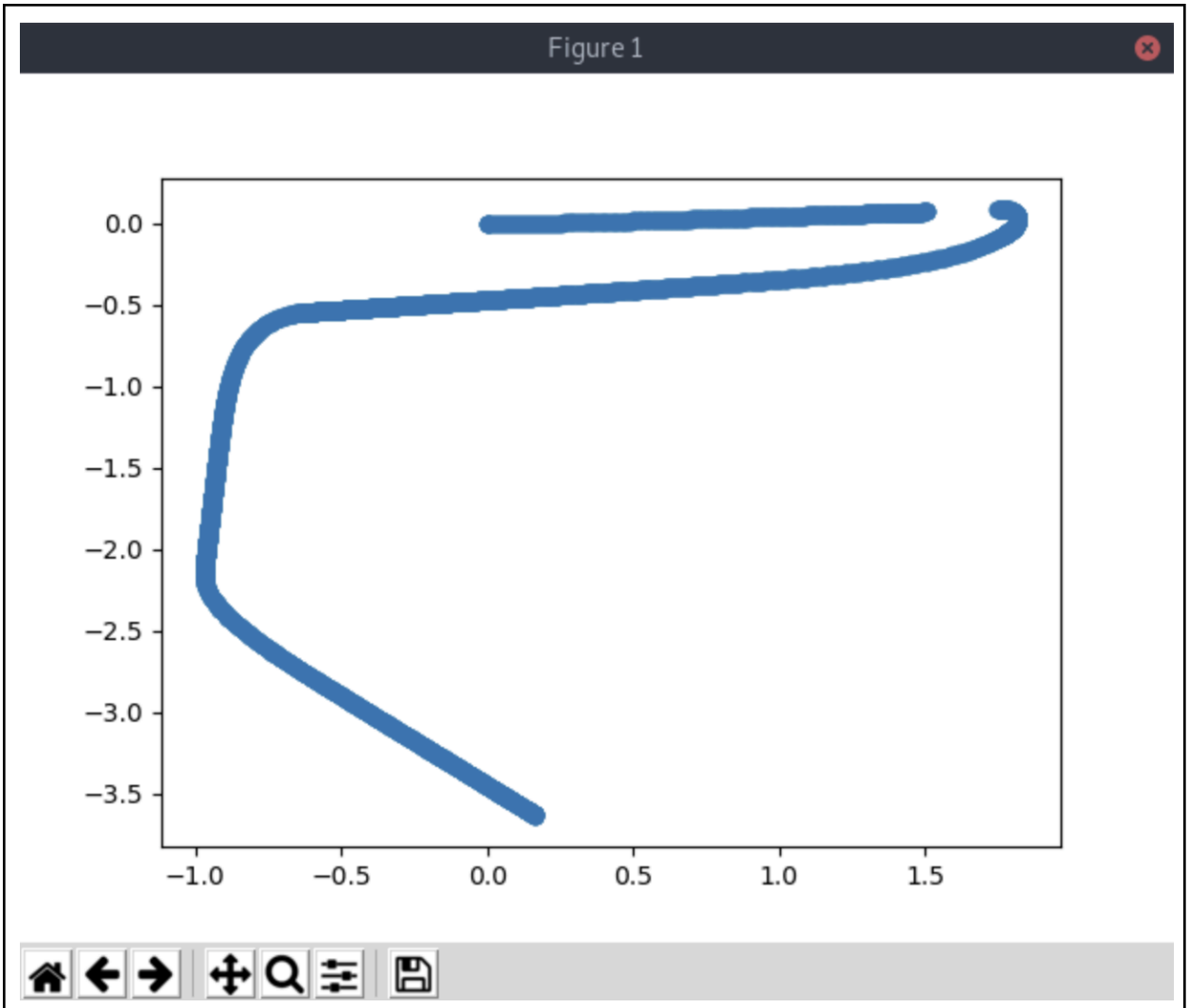


Figure IX: Generated graph when the threshold in nextLinearGoal was set to 0.1.

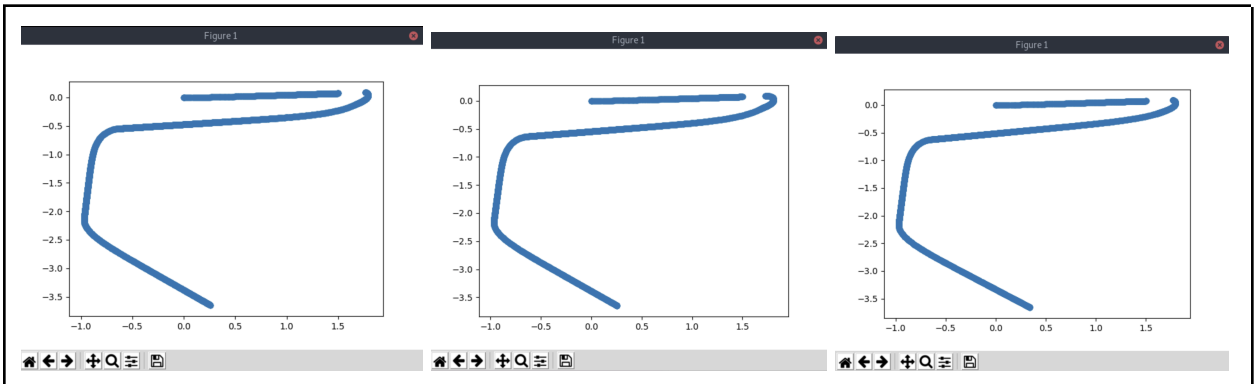


Figure X: Graphs generated of threshold points being set to default in the following order:

0.25, 0.3, and 0.5.

4.0 Summary

Based on the experiment done on the robot, it can be observed that when a robot follows a pre-planned path, the pursuit value should be considered when a robot reaches a given point. If the pursuit value is too high, this will generate path noise, and the robot might not reach the point it intended to reach accurately, if however the pursuit value is too small, the robot can take a lot of time to reach its pre-planned point. It is therefore important to find a pursuit value in which the robot can travel to its pre-planned path at an efficiently quick speed, and with accuracy reaching the set points. Through creating a nextLinearGoal function however, this allows the robot to more accurately travel from point to point linearly, which increases the accuracy of a robot traveling from its initial starting position to its final destination, and increasing the robot's speed going from start to end.