

Embedded Systems

Final Project

Professor Antonis Tzes

Hamdan Alhosani - Christos bin Zoukos

Abstract

A control system is a system of devices that allows you to manage, command and control the behavior of several devices or systems. It usually shows the relationship between the input and output of the system. The main two types of control systems are; open loop control system and closed loop system. In an open loop system the process output has no impact on the desired process response, meanwhile in a closed loop control system, the output of the system affects the input of the process. This is shown in the figures below.

In this project, an Arduino robot is placed in a two-dimensional map and tracked by using reacTIVision; a software that tracks fiducial markers and provide the x position and y position and angle of the robot. Using pytuio, the output of reacTIVision was converted to a python object. The movement of the robot is controlled by applying a first order control algorithm. The four tasks of this project were to move the robot from one arbitrary point to another arbitrary point, creating an open loop controller of the robot, allowing the robot to follow another leader robot's motion and follow a trajectory and keep a log of its path.

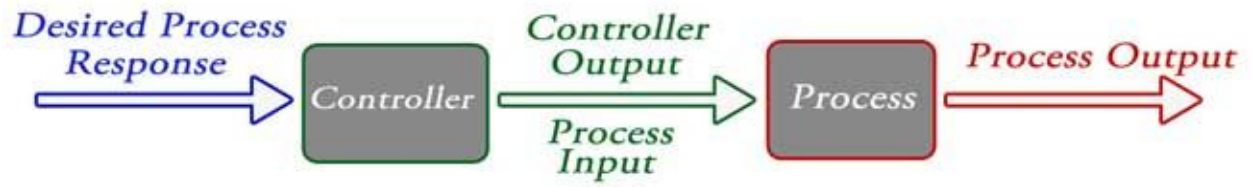


Image 1: Example of a open loop control system

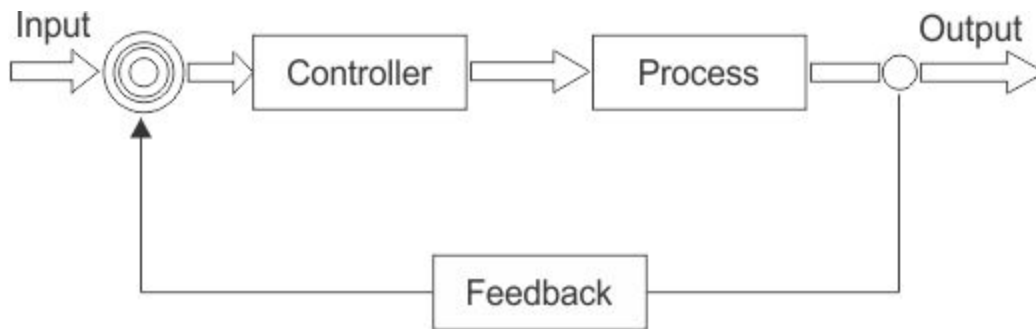


Image 2: Example of a open loop control system

Setup

The materials and software used in this project is as follows:

1. Logitech Camera
2. reacTIVision: is an open source computer vision framework used for fast and robust of fiducial markers attached on an object.
3. Pytuio: a python library that transmits the received data as OSC messages via an UDP socket to a client software.
4. Python Modules:
 - a. Socket: Low-level networking interface used to ssh to the robot.
 - b. Keyboard: a library that allows to simulate key presses and Take full control of your keyboard with this small Python library.



Image 3: A reacTIVision fiducial marker

Procedure

After installing reactTIVision, pytuio and the required python modules, connect the camera to the laptop to ensure everything is installed correctly. The camera must detect a fiducial marker. It is also important to make sure to calibrate the camera according to the camera's field of view in order to create a 2-D coordinate system. Place the robot in a point that you would like to consider as your zero point (0,0), in addition assign the angle of the robot to know its orientation. Figure 1 shows our coordinate map system with the robot at the point (0,0). Please note that because of the camera's coordination, our point (0,0) is set differently than the markers on the floor. For that reason, our point(0,0) is at the (1,1) marker while our (x,y) point is at the (0,0) marker on the floor.



Image 4: Custom Coordinate System

In order to configure the coordinate system for the robot, we had to convert the x,y values of reactIVision to ensure they are (0,0) at point (0,1) and (1,0) at point (1,1). This was done with the code snippet below (Image 4), which shows how the values were converted to the correct coordinate system. Also, please note that the below values of 0.06 and 0.9 were based on practical measurements of reactIVision's measurements and were manually changed to adjust the coordinate system. In case a different camera is used or the camera is placed in another position, the coordinate system might have to be readjusted.

```
57     # Identifying x,y position of the target
58     x_base = round(base.xpos,5) - 0.06
59     y_base = -1*(round(base.ypos,5) - 0.9)
```

Image 5: Configuring the coordinate system

Similarly, the angles of the robot had to be readjusted so the angle is 0 when the robot points towards point (x,0), 90 when it points towards (0,y), 180 when it points towards (-x,0) and 270 when it points towards (0,-y) respectively. The code snippet below (Image 5) shows the above-mentioned conversion to the appropriate coordinate system for the angles:

```

42     # Converting robot's angle to correct axis system
43     angle_robot = -1*(math.floor(robot.angle) - 270)
44     if angle_robot >= 360:
45         angle_robot -= 360
46     elif angle_robot < 0:
47         angle_robot = 360 - (-angle_robot)
48     if angle_robot == -0:
49         angle_robot = 0

```

Image 6: Configuring the angle of the robot according to the new coordinate system

The next step, was to calculate the angle of vector between the robot and the marker. This way, we are able to command the robot which side to turn in order to point towards the target. This was done by measuring the x,y values of the robot and the target (via reacTIVision) and then using arctan to calculate the angle of the vector. The formula used to calculate the vector between the robot and the target is as follows:

$$Vector = \tan^{-1} \frac{y_{target} - y_{robot}}{x_{target} - x_{robot}}$$

This procedure can be seen on the code snippet below (Image 6), while Image 7 shows a diagram of the logic implemented:

```
69     # Converting target's angle to correct axis system
70     ang_vect = math.floor(math.atan2(x_vect,y_vect)*180/math.pi)
71     ang_vect = -1*(ang_vect+270)
72
73
74     # Ensuring angle vector takes values from 0 to 360 degrees
75     if ang_vect < 0 and abs(ang_vect) > 360:
76         ang_vect = 720 - abs(ang_vect)
77     elif ang_vect < 0:
78         ang_vect = 360-abs(ang_vect)
79     elif ang_vect >= 360:
80         ang_vect -= 360
81     elif ang_vect < 0 and abs(ang_vect) > 360:
82         ang_vect = 720 - abs(ang_vect)
83     if ang_vect == -0:
84         ang_vect = 0
--
```

Image 7: Calculating the angle of the vector from the robot towards the target

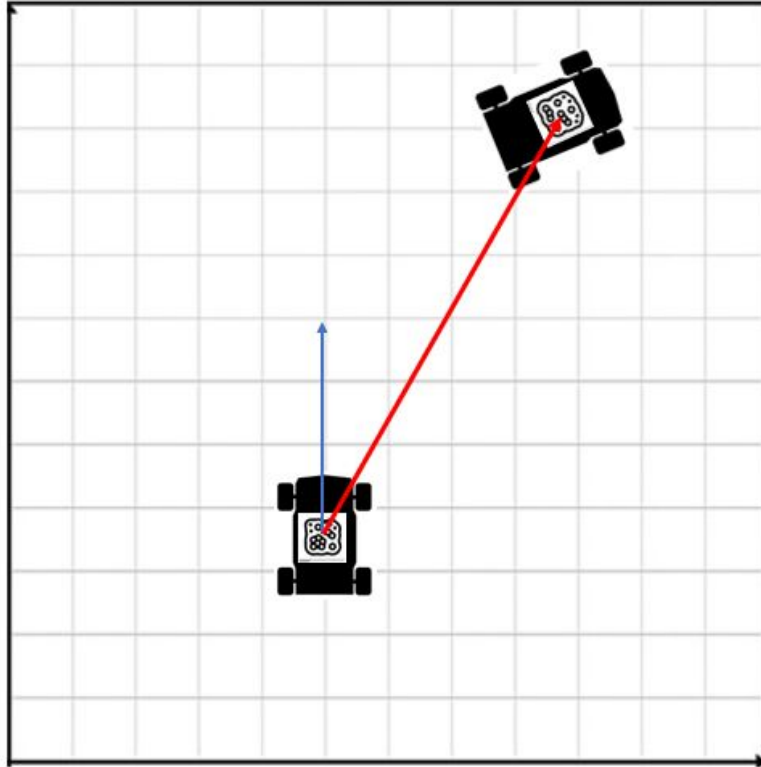


Image 8: Vector between the robot and the target

(The blue line shows the robot's orientation, while the red line the vector that points from the robot towards its target)

After calculating the angle of the vector, we then had to build logic that will allow the robot to turn left or right depending on the target. This was done by calculating the two arcs from the vector of the robot towards the vector that points from the robot to the target. Then depending on which arc is smaller the robot would turn left or right. The above-mentioned logic can be seen on Image 8 below, while Image 9 shows the code used for this implementation.

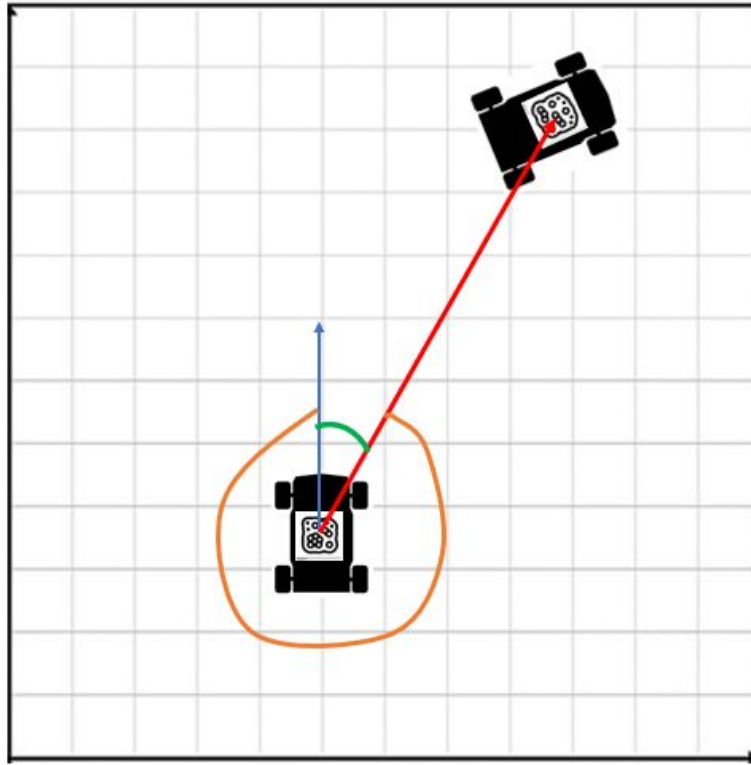


Image 9: Calculation of the two arcs

```

86     # Calculate angles
87     right_angle = ang_vect - angle_robot
88     if right_angle < 0:
89         right_angle += 360
90     left_angle = angle_robot - ang_vect
91     if left_angle < 0:
92         left_angle += 360
--

```

Image 10: Logic for calculating the two arcs

After deciding which arc is smaller, we then had to make the robot turn accordingly.

First, if the smaller arc is more than 80 degrees or the robot is close to the target then the robot will rotate towards the target instead of turning. Otherwise, the robot will turn by having all motors rotating but at different speeds. Image 10 below shows the code used for this logic:

```

118 rotate = False
119 # Rotate instead of turning in case robot too close to the target
120 if dist > 0.22:
121     if left_angle <= right_angle:
122         if left_angle >= 80:
123             rotate = True
124         else:
125             # Turn left
126             s.send('FF020105FF'.decode('hex')) # Left motor speed
127             s.send('FF020230FF'.decode('hex')) # Right motor speed
128             sleep(0.01)
129             s.send(forward)
130             sleep(0.05)
131             tracking.update()
132             s.send(stop)
133
134     else:
135         # If angle between target and robot too high, rotate instead of turn
136         if right_angle >= 80:
137             rotate=True
138         else:
139             # Turn Right
140             s.send('FF020130FF'.decode('hex')) # Left motor speed
141             s.send('FF020205FF'.decode('hex')) # Right motor speed
142             sleep(0.01)
143             s.send(forward)
144             sleep(0.05)
145             tracking.update()
146             s.send(stop)
147
148     else:
149         rotate = True
150 # Rotate Left
151 if rotate == True:
152     if left_angle <= right_angle:
153         s.send(r_left)
154     else:
155         # Rotate Right
156         s.send(r_right)
157         sleep(0.01)
158         tracking.update()
159         s.send(stop)

```

Image 11: Code used for the robots turning and rotating functions

Next, since locating the target and turning the robot towards the target is accomplished the robot will now have to move forward in case it points at the target and stop if it is close enough to the target. Also note that the speed of the follower robot is twice the speed of the leader robot. The code used for this implementation can be seen on Image 11 below:

```

101     # Stop if close to target
102     if dist <= 0.2:
103         tracking.update()
104         s.send(stop)
105         sleep(0.1)
106
107     # Move straight if pointing at target
108     elif abs(ang_vect - angle_robot) <= 10:
109         s.send(stop)
110         s.send('FF020125FF'.decode('hex')) # Left motor speed
111         s.send('FF020225FF'.decode('hex')) # Right motor speed
112         sleep(0.01)
113         s.send(forward)
114         sleep(0.05)
115         tracking.update()
116         s.send(stop)

```

Image 12: Code for moving the robot forward or stopping in case it is close to its target

As far as the trajectory part is concerned the logic was similar, with only a small difference, which is the coordinates of the target and timestamp of each point. In order to ensure that the robot will go to all the points of the trajectory, two arrays were made with the x and y values of the trajectory. At each iteration of the array the x,y values were the target for the robot. Then, the main open loop was modified and would iterate on the arrays everytime the robot went close to its target. This way, the robot would move towards the entire trajectory and would even move to the starting point no matter at which initial point it is located at. Also, a timer would go off as soon as the robot starts moving and the total time is printed every time the robot reaches its destination in order to show the results of the speed. Also, for each point, the x,y positions as well as the timestamps for each point were recorded and saved in a .csv file. The code used for this part can be seen on Image 12, 13, 14 and 15 below. Also, please note that the coordinates of the robot while it moves in the trajectory are saved into a csv

file in order to compare results and the accuracy of the path moved compared to the trajectory.

```
20 x_arr = [0.85, 0.83, 0.77, 0.68, 0.56, 0.44, 0.32, 0.23, 0.17, 0.15]
21 y_arr = [0.5, 0.72, 0.84, 0.8, 0.62, 0.38, 0.2, 0.16, 0.28, 0.5]
```

Image 12: Arrays of the x,y values

```
106         if dist <= 0.12:
107             tracking.update()
108             s.send(stop)
109             sleep(0.1)
110             counter += 1
```

Image 13: Iterating the array when the robot reaches its target until it goes over the entire array

```
39     while counter < 10:
```

Image 14: Ending the main loop when the robot traverses through all the points of the trajectory

```

34 # Open and write in new file
35 output = open("trajectory.csv", "w")
36 output.write("x_val, y_val, time" + "\n")
37
38 try:
39     start_t = time.time()
40     while counter < 10:
41         tracking.update()
42         tmp = 0
43         for obj in tracking.objects():
44             robot = obj
45             x_robot = round(robot.xpos,3) - 0.06
46             y_robot = -1*(round(robot.ypos,3) - 0.9)
47             end_t = time.time()
48             dt = (end_t - start_t)
49             output.write(str(x_robot) + ", " + str(y_robot) + ", " + str(dt) + "\n")

```

Image 15: Starting the timer and writing to the .csv file

Results

Task 1 : Controller

In the first task of the project, we had to create a remote controller that can control the robot's movement based on an input. The robot accepts 5-byte commands through plain TCP on port 2001. The robot has the commands already coded on its arduino, which were given to us in the beginning of the project. The robot is controlled using a keyboard, in which whenever a key is pressed, it will send a hexadecimal code to the robot via the TCP/IP connection. The code snippet below shows the different keys used to control the robot. The four keys used are w, a, s and d which correspond to forward, left, reverse and right respectively. This is considered as an open loop control system, since the output of the system (robot's motion) does not affect the input of the system.

```

import keyboard

from time import sleep

# All commands

forward = bytes.fromhex('FF000100FF')
backward = bytes.fromhex('FF000200FF')
stop = bytes.fromhex('FF000000FF')
r_left = bytes.fromhex('FF000300FF')
r_right = bytes.fromhex('FF000400FF')
#####
##### CHANGE TO 1 TO ENABLE SSH #####
#####
enable_socket = 1

if enable_socket:
    # Connect to robot
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("192.168.1.1", 2001))

while True:
    if keyboard.is_pressed('w'): # Move Straight
        s.send(bytes.fromhex('FF020130FF')) # Left motor speed
        s.send(bytes.fromhex('FF020230FF')) # Right motor speed
        s.send(forward)
        sleep(0.05)
        s.send(stop)
    if keyboard.is_pressed('a'): # Turn Left
        #s.send(bytes.fromhex('FF020130FF')) # Left motor speed
        #s.send(bytes.fromhex('FF020210FF')) # Right motor speed
        s.send(r_right)
        sleep(0.05)
        s.send(stop)
    if keyboard.is_pressed('d'): # Turn Right
        #s.send(bytes.fromhex('FF020110FF')) # Left motor speed
        #s.send(bytes.fromhex('FF020230FF')) # Right motor speed
        s.send(r_left)
        sleep(0.05)
        s.send(stop)
    if keyboard.is_pressed('s'): # Reverse
        s.send(bytes.fromhex('FF020130FF')) # Left motor speed
        s.send(bytes.fromhex('FF020230FF')) # Right motor speed
        s.send(backward)
        sleep(0.05)
        s.send(stop)

```

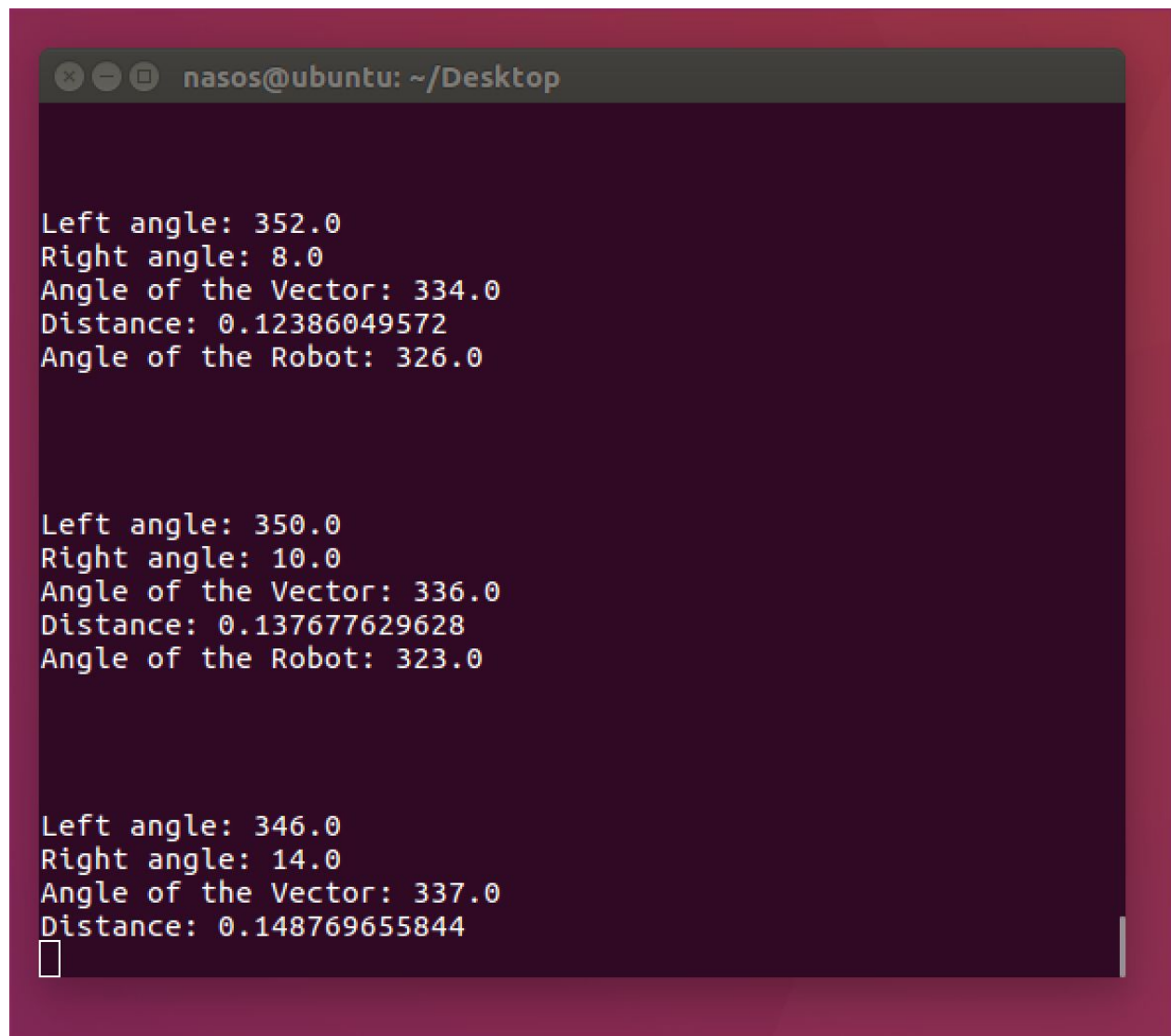
Image 16: Code snippet of the robot controller

Task 2 : From point A to B

The second task of the project was make the robot move from arbitrary point in to another arbitrary point. The robot had to find the target fiducial marker and move towards it. This is a closed loop control system, since the output of the system, which is the (x,y) coordinates of the target marker affect the values of the vector between the robot and target. Hence, the commands sent to the robot will change according to the vector's angle and distance (whether to move forward, rotate or turn). In the figures below, the feedback of the system is displayed.



Image 17: Calculation of the coordinates and the angles

A terminal window with a dark purple background and a grey title bar. The title bar contains the text 'nasos@ubuntu: ~/Desktop'. The terminal displays three sets of numerical data, each set consisting of five lines. The first set shows values for Left angle, Right angle, Angle of the Vector, Distance, and Angle of the Robot. The second set shows similar values. The third set shows similar values. A cursor is visible at the end of the third set of data.

```
nasos@ubuntu: ~/Desktop

Left angle: 352.0
Right angle: 8.0
Angle of the Vector: 334.0
Distance: 0.12386049572
Angle of the Robot: 326.0

Left angle: 350.0
Right angle: 10.0
Angle of the Vector: 336.0
Distance: 0.137677629628
Angle of the Robot: 323.0

Left angle: 346.0
Right angle: 14.0
Angle of the Vector: 337.0
Distance: 0.148769655844
█
```

Image 18: Feedback of the system

Task 3: Leader-slave

For this task, our robot was able to successfully chase the leader robot and stop when it came too close, while maintaining the orientation of the leader. Our robot also moved at a high speed, which allowed it to catch up with the leader quite fast. A demonstration of this can be seen on the video that is uploaded along with the report. Image 19 below shows a screenshot of our robot chasing the leader:

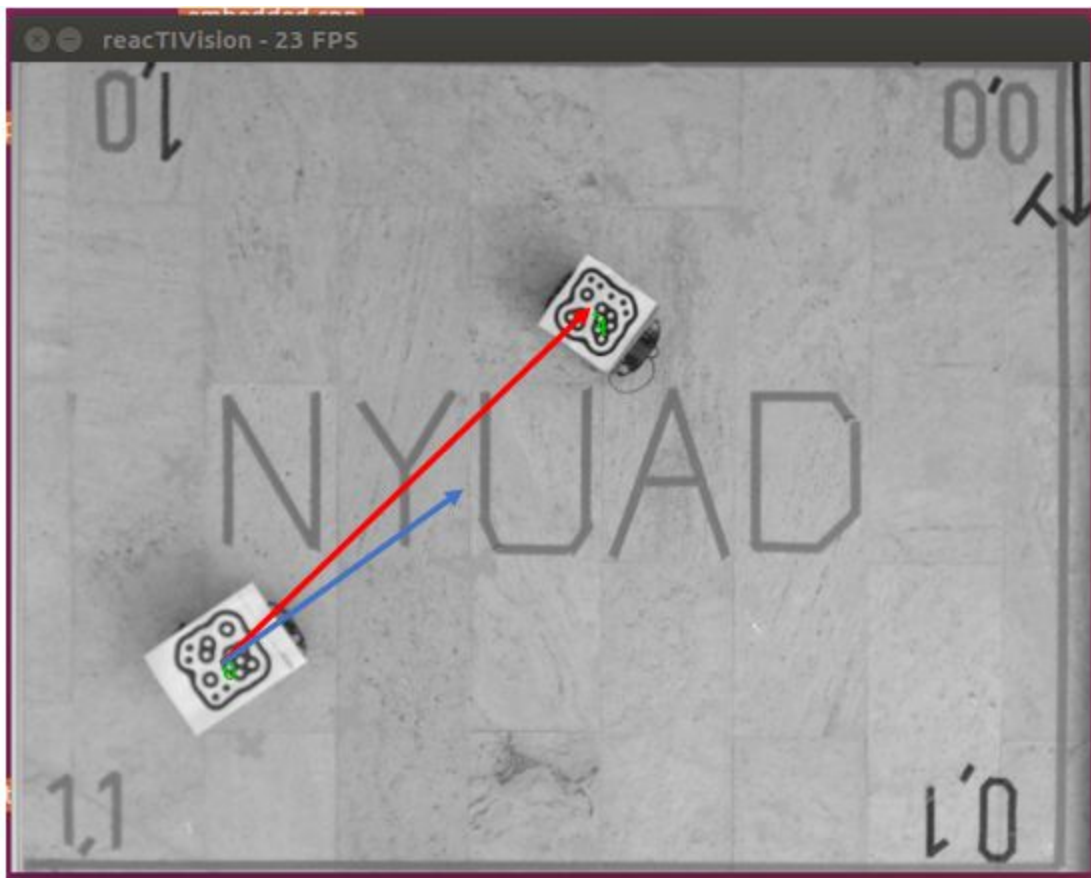


Image 19: Follower robot (left) chasing the leader robot (right)

Task 4: Trajectory

In this part our robot was successful in moving toward the target trajectory with a small deviation for performance purposes. We assumed that the time in which the robot should complete the trajectory is 10 seconds and the time to go from one point to another point is one second. Our robot was able to complete the trajectory in around 10.8 seconds and in the table below the timestamp when it reach the point. In addition, we calculated the percentage absolute error between the given points and the robot's trajectory. Image 21 below shows the outputs of the trajectory code, while Image 20 shows a graph that compares the robot's trajectory versus the targeted trajectory:



Image 20: The trajectory of the robot and the (x,y) values of the target and the robot.

The time elapsed after the robot followed trajectory

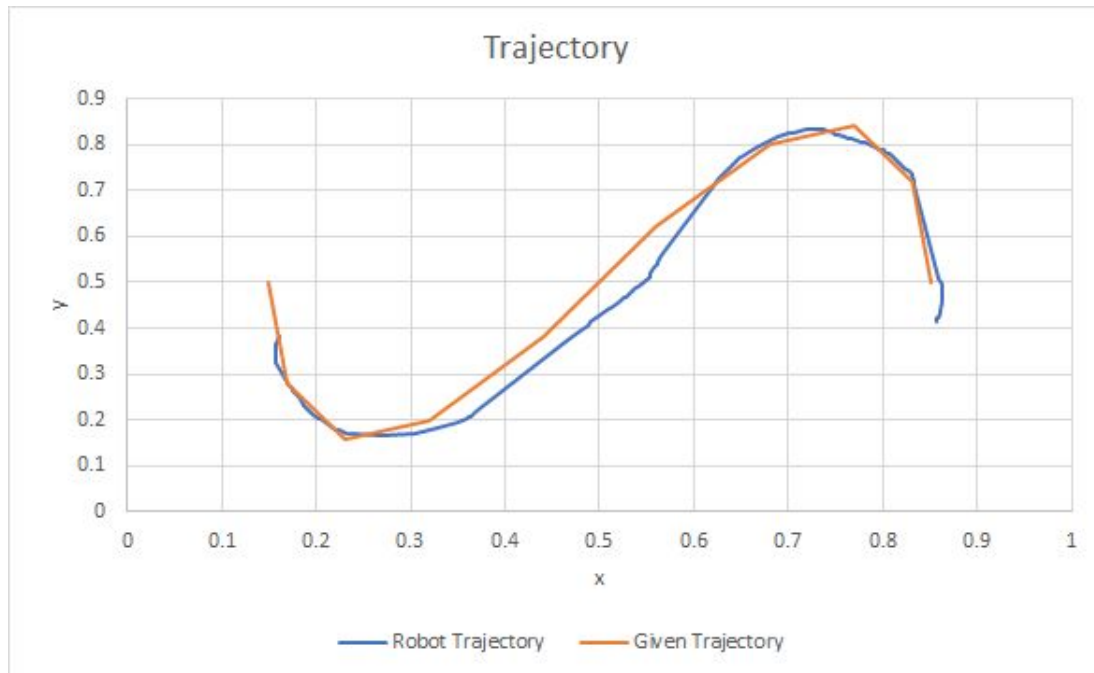


Image 21: The robot's trajectory versus the target trajectory

Table 1: Values of the trajectory part and the percentage error

Given X	Given Y	Robot X	Robot Y	Timestamp (s)	ABS X Error (%)	ABS Y Error (%)
0.85	0.5	0.859	0.508	1.07	1.058824	1.6
0.83	0.72	0.833	0.72	1.2	0.361446	0
0.77	0.84	0.778	0.806	2.68	1.038961	4.047619
0.68	0.8	0.68	0.809	5.015	0	1.125
0.56	0.62	0.563	0.554	5.80	0.535714	10.64516
0.44	0.38	0.468	0.376	7.184	6.363636	1.052632
0.32	0.2	0.327	0.181	7.89	2.1875	9.5
0.23	0.16	0.233	0.17	10.08	1.304348	6.25
0.17	0.28	0.176	0.269	10.6	3.529412	3.928571
0.15	0.5	0.158	0.48	10.8	5.333333	4

Discussion

Last but not least, our robot was able to complete all of our tasks with high accuracy and speed, as shown in both the results and the video. However, our results could be optimized via techniques of machine learning and control theory where the robot would adjust the speeds of its motors automatically for maximum efficiency. On the contrary our robot was still able to deliver accurate results with a low percentage of error (3.2%) on the trajectory part and was able to follow the leader robot with no major lag in speed.

Furthermore, during the demo the robot seemed to make some quick stops while moving forward. That is because of two reasons; First, the camera does not always detect the fiducial on the robot when moving too fast and second the camera seems to occasionally autofocuses depending on environmental movement or at random times, which makes the robot stop as it does not take any data from reacTIVision. In conclusion, in order to improve the robot's motion and the time of the trajectory a second order control algorithm needs to be applied, which will ensure smooth curves with no delay but such algorithms is beyond our knowledge.

References

- “ReacTIVision 1.5.1.” *ReacTIVision*, reactivision.sourceforge.net/.
- *Control System | Closed Loop Open Loop Control System*,
www.electrical4u.com/control-system-closed-loop-open-loop-control-system/.
- “Pytuio.” *PyPI*, pypi.org/project/pytuio/.
- “Keyboard.” *PyPI*, pypi.org/project/keyboard/.
- “17.2. Socket - Low-Level Networking Interface¶.” *17.2. Socket - Low-Level Networking Interface - Python 2.7.15 Documentation*,
docs.python.org/2/library/socket.html.
- “16.3. Time - Time Access and Conversions¶.” *16.3. Time - Time Access and Conversions - Python 3.6.5 Documentation*,
docs.python.org/3/library/time.html.