ENGR–UH 3530: Embedded Systems

Prof. Anthony Tzes

Final Project

Siba Siddique (ss8315)
Yasmin Farhan (yf740)

Date of submission: May 2, 2018

Introduction

For this project, we used a DS Robot which comprises of a 51duino Mainboard, four DC motors, a Wi-fi module for communication, and an on-board camera.
We started with implementing control over the robot's functions using the TCP/IP protocol and keyboard shortcuts by connecting to Wi-Fi access point created by the Wi-Fi module. This was used to implement a communication 'language' such that robot moves according to specified user input.

Fiducial markers were a main part of the project, and is used in the implementation of the control logic where the program detects the fiducial markers on the individual robots then execute the relevant commands. reacTIVision was identified as a suitable computer vision framework for detecting and identifying fiducial markers, used to identify the robot and the specified start/end locations.

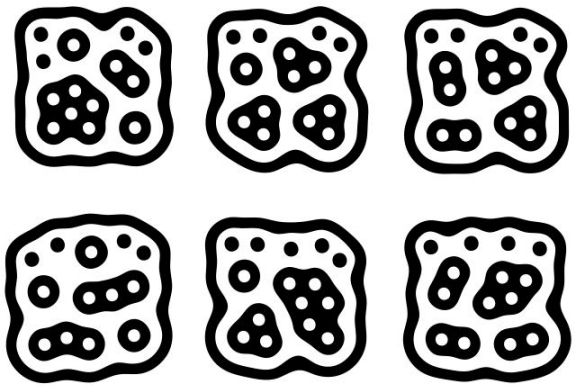

Figure 1. Reactivision Fiducial Markers

The frame rate of reacTIVision is fixed at 30 frames/sec (fps), which is a typical value perceptible to the human eye. The reacTIVision software includes orientation and x-,y-coordinates information, and uses the TUIO protocol, which is used by the reacTIVision software to identify the robot and the specified end location.

The TUIO protocol is encoded using the Open Sound Control (OSC) format, and requires the OSC implementation to transmit on the channel. Two main class of messages are defined by the TUIO protocol: ALIVE messages and SET messages. As can be understood from the naming, the ALIVE messages return a list of unique Session IDs of objects which are 'alive' or active, while SET messages include information on the orientation and x- and y-coordinates (position). Each bundle of SET messages includes an ALIVE message containing the session IDs of all currently active TUIO components, and each bundle starts with a SOURCE message to identify the TUIO source. Furthermore, each packet is marked with a frame sequence ID message contains data acquired at a given time and is useful in error-checking [4].

Our tasks for this project were divided into four parts: firstly, we ensured that we were able to manually control the robot; then, there was the go-to-goal task where the robot must go from any initial pose to any target pose. The third task is to have the robot follow another one, i.e. a leader-follower model, and optimize the experimental procedure by either minimizing the distance or the time required for the robot to reach point B from A. In the final Trajectory tracking task, the robot must follow a pre-specified trajectory within an optimum amount of time.

To implement our application, we used the TCP/IP protocol in Python to transmit the TUIO data. A TCP socket is a form of Inter Process Communication (IPC), and is commonly used in cross-platform communication. In order to use socket programming for communication, which is essentially a way of connecting two nodes on a network to common node, the client socket and server socket needs to be defined [3].

Procedure and Results

To get started with the project, the following setup shown in Figure 1 was used for the project environment. The TCP/IP protocol was used in communicating with the robot, and reacTIVision on the computer was used to identify the fiducial markers both on the robot and around it.

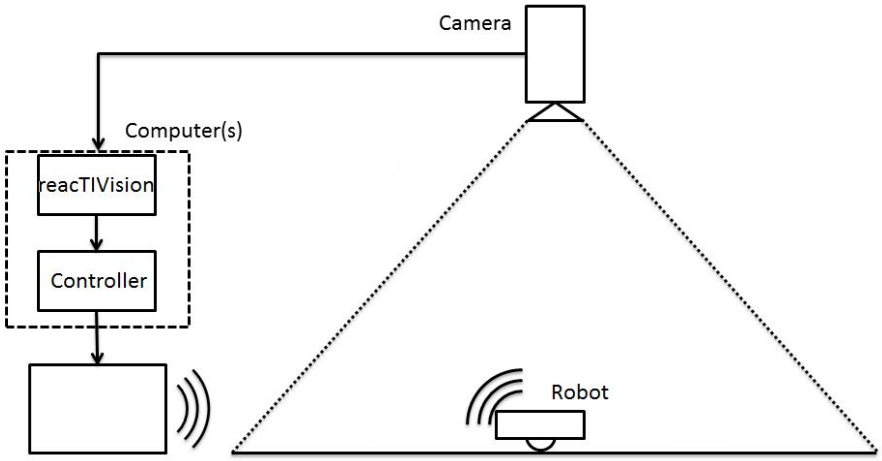

Figure 2. Robot and camera setup [2]

In addition to being controlled in open loop with user input, the robot should be capable of performing three tasks, as follows:

1. Go-to-goal task
2. Leader-follower task

3. Trajectory tracking task, where the robot has to follow the trajectory and keep a log of its position while minimizing the error and time it takes for the robot to complete the task.

Firstly, the pyTUIO python library was installed which implements the TUIO protocol. On command line, after installing pytuio into a directory on your Python import path, we can start with the subtasks.

Stage 1: Manual Control of Robot

The first stage of the project involved setting the connection by initialising the robot as a socket and sending hexadecimal commands over Wi-Fi which is used to control the robot. The socket library is first imported before initialising a socket, as shown in the following code:-

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

The first parameter refers to the address family ipv4, and the SOCK_STREAM means connection oriented TCP protocol. As the robot will be connected to a server on the local network, the following line is used to connect to the server:

```
s.connect((tcp_ip, port))
```

To understand how the robot responds to the hex commands, the code was structured as a user-input program, as shown in Figure 2 below. This code was implemented in the leader robot for the leader-follower task.

```
#imported modules
import socket
import time

TCP_IP = '192.168.1.1'
TCP_PORT = 2001
BUFFER_SIZE = 1024
forward = b'\xff\0\x01\0\xff'
bckwd = b'\xff\0\x02\0\xff'
stop = b'\xff\0\x00\0\xff'
rot_r = b'\xFF\x00\x04\x00\xFF'
rot_l = b'\xFF\x00\x03\x00\xFF'
save_cam_angle = b'\xFF\x32\x00\x00\xFF'
reset_cam_angle = b'\xFF\x33\x00\x00\xFF'

#functions defining robot commands
def move_fwd(skt):
  skt.send(forward)
```

```
def move_bckwd(skt):
  skt.send(bckwd)

def move_stp(skt):
  skt.send(stop)

def rotate_r(skt):
  skt.send(rot_r)

def rotate_l(skt):
  skt.send(rot_l)

def set_right_speed(skt):
  speed_no = int(input("Enter speed from 0-100:"))
  speed_no_hex = str(hex(speed_no))
  if(speed_no < 16): #ensuring correct string length for single digit hexadecimal values
    speed_msg_str = 'FF02010' + speed_no_hex[2:len(speed_no_hex)] + 'FF'
  else:
    speed_msg_str = 'FF0201' + speed_no_hex[2:len(speed_no_hex)] + 'FF'
  speed_msg_bytes = bytes.fromhex(speed_msg_str)
  skt.send(speed_msg_bytes)

def set_left_speed(skt):
  speed_no = int(input("Enter speed from 0-100:"))
  speed_no_hex = str(hex(speed_no))
  if(speed_no < 16):
    speed_msg_str = 'FF02020' + speed_no_hex[2:len(speed_no_hex)] + 'FF'
  else:
    speed_msg_str = 'FF0202' + speed_no_hex[2:len(speed_no_hex)] + 'FF'
  speed_msg_bytes = bytes.fromhex(speed_msg_str)
  skt.send(speed_msg_bytes)

#these functions allow for the attached camera to be rotated, but were not ultimately used given the availability of the overhead camera to track the fiducial markers instead
def rot_cam_LR(skt):
    angle_no = int(input("Enter angle from 0-180:"))
    angle_no_hex = str(hex(angle_no))
    angle_msg_str = 'FF0107' + angle_no_hex[2:len(angle_no_hex)] + 'FF'
    angle_msg_bytes = bytes.fromhex(angle_msg_str)
    skt.send(angle_msg_bytes)

def rot_cam_UD(skt):
    angle_no = int(input("Enter angle from 0-180:"))
    angle_no_hex = str(hex(angle_no))
    angle_msg_str = 'FF0108' + angle_no_hex[2:len(angle_no_hex)] + 'FF'
    angle_msg_bytes = bytes.fromhex(angle_msg_str)
    skt.send(angle_msg_bytes)

def request_cmnd(socket):
  end = False
```

```
#a while loop to  ensure that commands controlling the robot can be continuously received
  while(end == False):
    cmnd = int(input("Enter command:\nForward: 1\nBackward: 2\nStop: 3\nRotate right:
4\nRotate left: 5\nChange right motor speed: 6\nChange left motor speed: 7\nClose
connection: 0\n"))
    if(cmnd == 1):
      move_fwd(socket)
    elif(cmnd == 2):
      move_bckwd(socket)
    elif(cmnd == 3):
      move_stp(socket)
    elif(cmnd == 4):
      rotate_r(socket)
    elif(cmnd == 5):
      rotate_l(socket)
    elif(cmnd == 6):
      set_right_speed(socket)
    elif(cmnd == 7):
      set_left_speed(socket)
    elif(cmnd == 0):
        end = True


def main():

  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #creating the socket object
  s.connect((TCP_IP, TCP_PORT)) #establishing the connection
  request_cmnd(s) #calling the function to enable user to command the robot
  s.close() #closing the connection

main()
```

Figure 3: Initial working code for Wi-Fi robot - Manual control by user

Stage 2: Moving Robot From Point A to B

The next stage of the project called for the robot to move from a point A, defined by the starting position of the robot, to point B. It was determined that we would be using the overhead camera set-up outside Lab A5-014, and therefore, in combination with the reacTIVision module, we were able to integrate the information gathered from the camera with the commands sent over TCP to the robot, thus substituting user input with the information received by the camera to guide the robot's behavior.

This information pertains specifically to the previously mentioned fiducial markers that reacTIVision is capable of recognizing. The following information for each fiducial marker, or TUIO object, as dictated by the objects.py file of the pyTUIO module, is available to us. In our

implementation, we made use of the object's id, x and y coordinates, as well as angle, all highlighted below. Therefore, given this available information, a fiducial marker (fiducial marker with ID no. 0) was associated with point A, i.e., the robot's position, and another fiducial marker (ID no. 3) with point B, the final robot destination.

```
class Tuio2DObject(TuioObject):
    def __init__(self, objectid, sessionid):
        super(Tuio2DObject, self).__init__(objectid, sessionid)
        self.id = objectid
        Self.sessionid = sessionid
        self.xpos = self.ypos = self.angle = 0.0
        self.xmot = self.ymot = self.mot_accel = self.mot_speed = 0.0
        self.rot_speed = self.rot_accel = 0.0
```

Figure 4. Tuio2DObject class in objects.py file of pyTUIO module

The first steps towards our implementation was assuming 4 possible scenarios, guided by the assumption that the destination fiducial marker could be in only 1 of 4 possible positions relative to the starting position of the robot (northeast, northwest, southeast, southwest), with the robot serving as a reference to the origin in this theoretical coordinate plane. In each of these 4 scenarios, the plane is divided into five sections, namely the four general quadrants, with the quadrant containing the vector pointing from the robot to the end location further split to account for the different code logic to be implemented for this special fifth case.

The following angle was common amongst all final rotational angle, or bearing calculations:

$$\theta = arctan(\Delta y/\Delta x)$$

where Ay is the projected length of the vector along the y-axis, and Ax is the projected length of the vector along the x-axis. In the diagrams below, the red line represents the shortest path drawn between the robot and final destination, the R the position of the Robot, and the X its destination. The sections are defined from 1 to 5 starting from quadrant I above the positive x-axis. Angle *a* defines the starting angle of the robot, i.e. the angle information received by the camera about TUIO object with ID 0. Also, we note the follower/pursuer has twice the speed of the evader, and the trajectory is not known in advance. The derivation of the individual sections and the corresponding direction and angles of rotation are illustrated in Fig. 2 and has been compiled in Table 1 below.

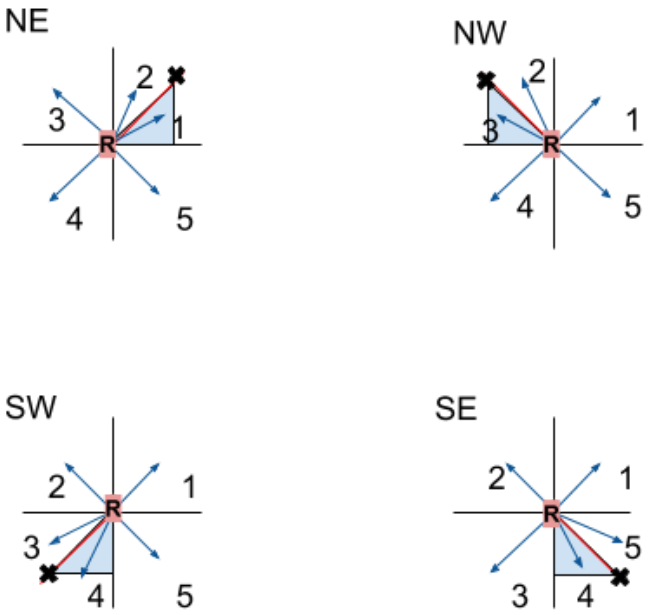

Figure 5. Diagrams depicting the sections for each possible positional scenario

Table 1. Table with different sections and corresponding rotation angles

		Section range	Rotation direction + bearing
NE	**1.**	$0 < a < \theta$	L: $\theta - a$
	2.	$\theta < a < 90$	R: $a - \theta$
	3.	$90 < a < 180$	R: $a - \theta$
	4.	$180 < a < 270$	R: $a - \theta$
	5.	$270 < a < 360$	L: $360 - a - \theta$
NW	**1.**	$0 < a < 90$	L: $180 - a - \theta$
	2.	$90 < a < (180 - \theta)$	L: $180 - a - \theta$
	3.	$(180 - \theta) < a < 180$	R: $a - 180 + \theta$
	4.	$180 < a < 270$	R: $a - 180 + \theta$

	5.	$270 < a < 360$	R: $a - 180 + \theta$
SW	**1.**	$0 < a < 90$	L: $180 - a + \theta$
	2.	$90 < a < 180$	L: $180 - a + \theta$
	3.	$180 < a < (180 + \theta)$	L: $180 - a + \theta$
	4.	$(180 + \theta) < a < 270$	R: $a - 180 - \theta$
	5.	$270 < a < 360$	R: $a - 180 - \theta$
SE	**1.**	$0 < a < 90$	R: $a + \theta$
	2.	$90 < a < 180$	R: $a + \theta$
	3.	$180 < a < 270$	L: $360 - a - \theta$
	4.	$270 < a < (360 - \theta)$	L: $360 - a - \theta$
	5.	$(360 - \theta) < a < 360$	R: $a - 360 + \theta$

In our initial implementation of this task, the process of moving the robot to point B was split into 2 parts, the first being the robot rotating from its starting angle to that which would orient it in the destination's direction, and the second having it move forward until the correct x and y positions were reached within a specified range of error. The bulk of this initial code is shown below. In this initial version, no error correction along the path of movement was executed.

```
def new_cnct(): #function allowing for new socket connection to be established
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  s.connect((TCP_IP, TCP_PORT))
  return s

def check_case(sx, sy, ex, ey): #function to determine position of end fm relative to robot and calculate
theta, we pass in start and end coordinates

  theta = math.degrees(math.atan(abs(ey-sy)/abs(ex-sx)))

  if(sx>ex): #determining which of the four scenarios is applicable
    if(sy<ey):
      case = 'nw'
    else:
      case = 'sw'
```

```
  else:
    if(sy<ey):
      case = 'ne'
    else:
      case = 'se'

  return theta, case

def check_angle(): #returning robot's object angle

  tracking = tuio.Tracking()

  end = False
  while end == False:
    tracking.update()
    for obj in tracking.objects():
      if(obj.id == 0):
        print(obj.angle)
        tracking.stop()
        return obj.angle

def inc_rot_r(agl): #rotating robot to the right incrementally depending on final angle position
  end = False
  while end == False:
    rotate_r_3()
    # move_stp_2()
    time.sleep(0.02) #we found incorporating even a small delay was necessary to be able to register angle information within good time
    a = check_angle() #store current value of robot's angle
    print(agl)
    if((a < (agl+3)) and (a > (agl-3))): #if within desired range, end rotation
      end = True

def inc_rot_l(agl): #rotating robot to the left incrementally depending on final angle position
  end = False
  while end == False:
    rotate_l_3()
    time.sleep(0.02)
    a = check_angle()
    if((a < (agl+3)) and (a > (agl-3))):
      end = True

def check_overflow(agl_r, agl_s, is_right): #ensuring angle overflows are dealt with appropriately by wrapping around to correct final rotational position
  if(is_right):
    dif = agl_s - agl_r #rotating right results in a decrease in angle value
    if(dif < 0):
      agl_s += 360
  else:
    sum_agl = agl_s + agl_r #rotating left results in an increase in angle value
```

```
    if(sum_agl > 360):
      agl_s -= 360
  return agl_s #the adjusted final angle is returned

def det_rot_angle(agl_s, theta, case):
 #agl_s corresponds to starting angle of the robot
 #rot_angle is the angle the robot must rotate by to reach correct axis

  if(case == "nw"):
    if((agl_s > (180-theta)) and (agl_s < 360)):
      rot_angle = agl_s - 180 + theta
      is_right = 1
    else:
      rot_angle = 180 - agl_s - theta
      is_right = 0

  elif(case == "se"):
    if(agl_s > 0) and (agl_s <= 180):
      rot_angle = agl_s + theta
      # rot_angle = -theta
      is_right = 1
    elif(agl_s > 180) and (agl_s <= (360 - theta)):
      rot_angle = 360 - agl_s - theta
      is_right = 0
    else:
      rot_angle = agl_s - 360 + theta
      is_right = 1

  elif(case == "ne"):
    if(agl_s < theta):
      rot_angle = theta - agl_s
      is_right = 0

    elif((agl_s > theta) and (agl_s < 270)):
      rot_angle = agl_s - theta
      is_right = 1

    elif(agl_s > 270) and (agl_s < 360):
      rot_angle = 360 - agl_s + theta
      is_right = 0

  elif(case == "sw"):
    if(agl_s < (180 + theta)):
      rot_angle = 180 - agl_s + theta
      is_right = 0
    elif(agl_s >= (180 + theta)):
      rot_angle = agl_s - 180 - theta
      is_right = 1

  agl_s = check_overflow(rot_angle, agl_s, is_right) #accounting for overflow of >360 or <0,
```

```
change starting angle value if necessary

  return agl_s, rot_angle, is_right

def det_fm_info_end(): #function to retrieve information of goal/end point
  tracking = tuio.Tracking()
  while 1:
      tracking.update()
      for obj in tracking.objects():
          if(obj.id == 3):
            print obj
            tracking.stop()
            return obj.angle, obj.xpos, obj.ypos

def det_fm_info_start(): #function to retrieve initial information of robot's starting point (fiducial marker 0 information)
  tracking = tuio.Tracking()
  while 1:
      tracking.update()
      for obj in tracking.objects():
          if(obj.id == 0):
            print obj
            tracking.stop()
            return obj.angle, obj.xpos, obj.ypos

def move_dest(goal_x, goal_y): #function to move robot to correct x and y position
  end = False
  tracking = tuio.Tracking()
  while (end == False):
    tracking.update()
    for obj in tracking.objects():
        if(obj.id == 0):
          if(obj.xpos <= (goal_x + 0.01)) and (obj.xpos >= (goal_x - 0.01)):
            if(obj.ypos <= (goal_y + 0.01)) and (obj.ypos >= (goal_y - 0.01)):
              move_stp_2()
              end = True
def main():
  agl_e, xe, ye = det_fm_info_end() #getting info of goal fm
  agl_s, xs, ys = det_fm_info_start() #getting info of robot's starting position
  theta, case = check_case(xs, ys, xe, ye) #getting case and theta calculated
  agl_s, agl_r, ir = det_rot_angle(agl_s, theta, case) #retrieving the angle robot needs to turn from start position to align on axis of end fm

  #part 1: orienting robot to correct angle position
  if(ir):
    inc_rot_r(agl_s - agl_r)
    print(agl_s - agl_r)
  else:
    inc_rot_l(agl_s + agl_r)
    print(agl_s + agl_r)
```

```
    #part 2: moving forward to reach destination
     move_dest(xe, ye)

main()
```

Figure 6. Initial code for moving robot to a final point B

The `check_case` function is used to check which of the four possible positions the destination fiducial marker is placed in relative to the robot. The function `det_rot_angle` determines how much to rotate depending on the position and orientation of the robot, i.e. which of the 5 sections its starting angle falls in, as depicted in Figure 4. The boolean variable `ir` is used to store the initial direction of rotation required of the robot when orienting itself towards its destination. The methods `check_overflow` is used to account for the fact that the angle range is from [0,360), so

Stage 3. Implementation of leader-follower mechanism + optimizing stage 2

In this stage, we implemented the leader-follower model of the Wi-Fi robot, as well as made adjustments to the previous stage of moving the robot to a fixed point to include along the path error correction. The key to the switch in logic was creating an open loop, so that for as long as the angle was incorrect and the x and y coordinates of the goal, be it the fixed point or moving leader robot, were not in range, the follower robot would rotate to the appropriate orientation. If the angle of the follower robot is within range, it moves forward, but because the loop allows us to continuously check for these conditions, it will move forward only as long as the relevant one is satisfied. Otherwise it will adjust itself to the new conditions it has received in this feedback loop. Below is the bulk of the code for the implementation. Functions reused from the previous stage are not shown.

```
#a new socket was opened and closed every time a command was sent to avoid addressing clashes with the tuio tracking function
def move_fwd_2():
        skt = new_cnct()
        skt.send(b'\xFF\x02\x01\x18\xFF')
        skt.send(b'\xFF\x02\x02\x18\xFF')
        skt.send(forward)
        skt.close()
def move_stp_2():
        skt = new_cnct()
        skt.send(stop)
        skt.close()
```

```
#adjusted incrementing rotation function with included delay to allow for relatively high resolution information to be registered in real time
def rotate_r_3(delay):
	skt = new_cnct()
	skt.send(rot_r)
	# time.sleep(0.07)
	time.sleep(delay)
	skt.send(stop)
	skt.close()

def rotate_l_3(delay):
	skt = new_cnct()
	skt.send(rot_l)
	time.sleep(delay)
	# time.sleep(0.07)
	skt.send(stop)
	skt.close()

#function to return final destination angle according to bearing, start angle, and direction of rotation
def check_direction(agl_r, agl_s, is_right):
	if(is_right):
		final_angle = agl_s - agl_r
	else:
		final_angle = agl_s + agl_r
	return final_angle

#while this function originally turned the bearing/angle to rotate, this version returns the final desired angle position instead
def det_rot_angle(agl_s, theta, case):
	….#bulk of function remained the same as in previous version and has been removed for redundancy - changes made to end of function are shown below
	rot_angle = check_direction(rot_angle, agl_s, is_right)
	return rot_angle, is_right

def check_xy(xr, yr, goal_x, goal_y, xy_range): #checking to see if follower robot is within specified xy range
	if(xr <= (goal_x + xy_range)) and (xr >= (goal_x - xy_range)):
		if(yr <= (goal_y + xy_range)) and (yr >= (goal_y - xy_range)):
			return True
	return False

def main():
	#choosing which mode to operate the robot in, either moving to a fixed point or following a moving robot
	mode = int(input("Choose your mode:\n1: Move to point B\n2: Follow another robot\n"))
```

For mode 1, where the robot is moving to a fixed point, the delay, angle range, and xy_range are set to be lower than in mode 2, in the leader-follower mode, as the latter was deemed to benefit

less from being perfectly accurate and faster response time was preferred.

```
#different settings for angle range, delay, and xy_range are applied depending on mode
                if(mode == 1):
                        agl_range = 6
                        delay = 0.07
                        xy_range = 0.07
                else:
                        agl_range = 10
                        delay = 0.15
                        xy_range = 0.1
        while 1: #while loop continuously checks for current start and end conditions so that adjustments to follower can be made if necessary
                ae, xe, ye = det_fm_info_end()
                ar, xr, yr = det_fm_info_start()
                theta, case = check_case(xr, yr, xe, ye)
                agl, is_right = det_rot_angle(ar, theta, case)

                if(((ar < (agl+agl_range)) and (ar > (agl-agl_range))) and not(check_xy(xr, yr, xe, ye, xy_range))):
                        move_fwd_2() #moves forward if within angle but not xy_range

                elif(not((ar < (agl+agl_range)) and (ar > (agl-agl_range)))): #if not within angle range rotate to correct orientation
                        if(is_right):
                                rotate_r_3(delay)
                        else:
                                rotate_l_3(delay)
                else: #if within both angle and xy_range, will stop
                        move_stp_2()

                        if(mode == 1): #if we are moving to a fixed point, the while loop is exited, otherwise, the follower robot will wait for further movement from leader
                                break

main()
```

Figure 7. Code for stages 2 and 3 - Moving robot to point B, and following another robot

As explained in the comments of the code, the while loop implemented allowed for continuous checking of the conditions of both the start fiducial marker 0, that on the follower robot, as well as that of the end fiducial marker 3, designated to either the fixed final destination, in mode 1, which corresponds to stage 2 of the project, or to the leader robot, in mode 2. A larger angle and xy_range were set in mode 2, as accuracy was deemed to be less important than response time to the changes in the leader robot's movement. For example, a larger angle range allowed for the follower robot to move more quickly towards the leader, as opposed to a smaller one which would cause for more time spent re-adjusting the angle.

Stage 3: Trajectory tracking task

We are given the following trajectory in Figure x and the robot is required to follow the trajectory while keeping a log of its path. The objective is to minimise the time and the error in the implementation. In the first part is a path version of the implementation, which was demonstrated, and the second is a trajectory version which prioritizes timed transitions above all else.

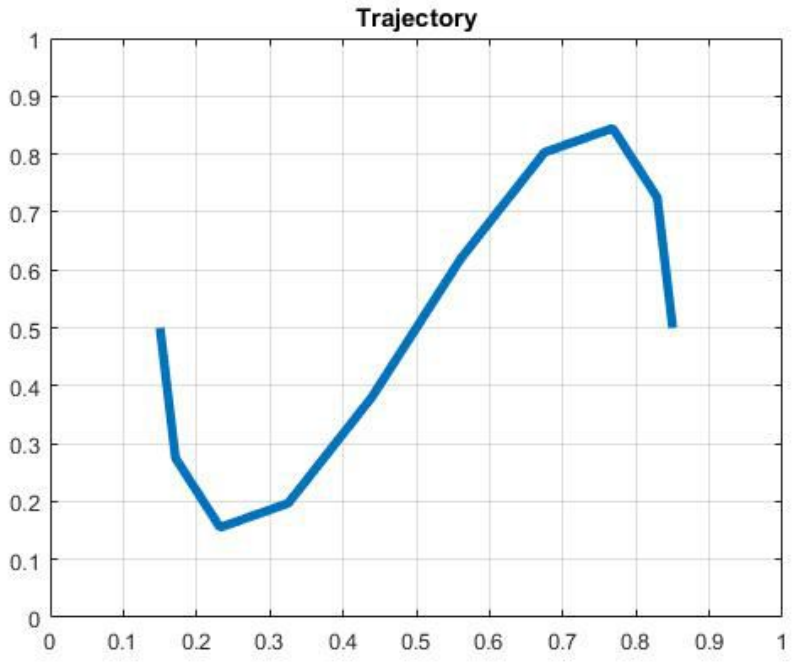

Figure 8. Assigned Trajectory

Table 2: Trajectory coordinates for tracking task

Time	**x**	**y**
0	0.85	0.50
1 ΔT	0.83	0.72
2 ΔT	0.77	0.84
3 ΔT	0.68	0.80
4 ΔT	0.56	0.62
5 ΔT	0.44	0.38
6 ΔT	0.32	0.20
7 ΔT	0.23	0.16
8 ΔT	0.17	0.28
9 ΔT	0.15	0.50

Part 1. Path

A similar logic to part 2 of the project was utilized for the tracking task. The main change implemented was the introduction of a for loop, which would loop for as many points in the path there were to traverse. The adjustments to the code, namely in the main function, are shown below. Different speeds were set at different points on the path in an attempt to speed up the total time where possible, as demonstrated by the called functions, set_speed_low(), set_speed_med(), and set_speed_high(). This variation in speed was eliminated in the trajectory portion of the task.

```
def det_next_pos(idx):
	x_points = [0.85, 0.83, 0.77, 0.68, 0.56, 0.44, 0.32, 0.23, 0.17, 0.15]
	y_points = [0.5, 0.72, 0.84, 0.8, 0.62, 0.38, 0.2, 0.16, 0.28, 0.5]

	return x_points[idx], y_points[idx]

def main():

	move_stp_2()
	set_speed_med() #set at 0x0f
	agl_range = 6
	xy_range = 0.03

	for i in range(1, 10):
		if(i==9):
			set_speed_low() #set at 0x0a
			xy_range = 0.06
		elif(i==3) or (i==4) or (i==5) or (i==6):
			set_speed_high() #set at 0x12

		ar, xr, yr = det_fm_info_start()
		xn, yn = det_next_pos(i)
		theta, case = check_case(xr, yr, xn, yn)
		agl, is_right = det_rot_angle(ar, theta, case)
		end = False

		while end == False: #while loop set up within each position traversal
			ar, xr, yr = det_fm_info_start()
			theta, case = check_case(xr, yr, xn, yn)
			agl, is_right = det_rot_angle(ar, theta, case)

			if(((ar < (agl+agl_range)) and (ar > (agl-agl_range))) and
not(check_xy(xr, yr, xn, yn, xy_range))):
				move_fwd_2()
			elif(not((ar < (agl+agl_range)) and (ar > (agl-agl_range)))):

				if(is_right):
					rotate_r_3()
				else:
					rotate_l_3()
```

```
                        else:
                                ts = time.time()
                                st =
datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S')

                                print(xr, yr, xn, yn, st)
                                move_stp_2()

                                end = True #the while loop is broken once conditions pertaining to x,y
coordinates and orientation are satisfied
main()
```

Figure 9. Code for Path implementation

The total time for this implementation which utilizes continual error checking is shown below in Tables 3-4. While we did increase the speeds in an attempt to minimize the time taken to traverse the path, we found that higher speeds were less reliable, and that while they did occasionally result in a much quickly total path time, often they would take much longer as a result of moving too quickly to register the correct angle position, and would continue rotating in place until the condition was satisfied. This dramatic increase in time can be shown in the following tables for different speeds.

In order to calculate the relative error of distance, the equations for relative percentage in x and in y were used.

Table 3a. Path-following implementation data (`med speed=0x0f, high speed=0x12, error = ±0.03`)

	Actual x	Actual y	Goal x	Goal y	Timestamp	x %error	y %error
0	0.85	0.5	0.85	0.5	18:07:40	0	0
1	0.849901676	0.697465003	0.83	0.72	18:07:42	2.4	3.13
2	0.798361003	0.824388146	0.77	0.84	18:07:45	3.68	1.86
3	0.708096623	0.810482204	0.68	0.8	18:07:48	4.13	1.31
4	0.576442301	0.630725563	0.56	0.62	18:07:51	2.94	1.73
5	0.456993014	0.408231348	0.44	0.38	18:07:54	3.86	7.43
6	0.338199317	0.213068187	0.32	0.2	18:07:56	5.69	6.53
7	0.256282777	0.167671919	0.23	0.16	18:07:57	11.43	4.79
8	0.195563823	0.25657779	0.17	0.28	18:08:00	15.04	8.37
9	0.178458259	0.444828093	0.15	0.5	18:08:04	18.97	11.03

Table 3b. Path-following implementation data (`med speed=0x18, high speed=0x18, error = ± 0.03, agl_range = 8`)

	Actual x	Actual y	Goal x	Goal y	Timestamp	x %error	y %error
0	0.85	0.5	0.85	0.5	19:20:40	0	0
1	0.840461075	0.690464735	0.83	0.72	19:20:42	1.26	4.1
2	0.797574282	0.815464735	0.77	0.84	19:20:44	3.58	2.92
3	0.707604885	0.814321101	0.68	0.8	19:20:57	4.06	1.79
4	0.589647055	0.631548703	0.56	0.62	19:21:00	5.29	1.86
5	0.430610806	0.370134026	0.44	0.38	19:21:07	2.13	2.6
6	0.346612751	0.228715032	0.32	0.2	19:21:13	8.32	14.36
7	0.255900353	0.166928917	0.23	0.16	19:21:14	11.26	4.33
8	0.194793493	0.252935588	0.17	0.28	19:21:17	14.58	9.67
9	0.172765523	0.443276495	0.15	0.5	19:21:19	15.18	11.34

Table 3c. Path-following implementation data (`med speed=0x24, high speed=0x24, error = ± 0.03, agl_range = 8`)

	Actual x	Actual y	Goal x	Goal y	Timestamp	x %error	y %error
0	0.85	0.5	0.85	0.5	19:23:45	0	0
1	0.8286713362	0.6978437901	0.83	0.72	19:23:49	0.16	3.08
2	0.7504261732	0.8541229963	0.77	0.84	19:23:54	2.54	1.68
3	0.6669689417	0.8049460649	0.68	0.8	19:24:08	1.92	0.62
4	0.5676354766	0.6488344669	0.56	0.62	19:24:22	1.36	4.65
5	0.4473667145	0.3771780431	0.44	0.38	19:24:26	1.67	0.74
6	0.3332878053	0.1821168363	0.32	0.2	19:24:30	4.15	8.94
7	0.223093316	0.1644886285	0.23	0.16	19:24:36	3	2.81
8	0.1828944385	0.2839088142	0.17	0.28	19:24:41	7.58	1.4
9	0.1546547115	0.5510271192	0.15	0.5	19:24:46	3.1	10.21

Part 2. Trajectory

The next step was correctly implementing the time dimension into the code. While in the first part, the while loop was broken only when the position was reached within the specified range of error, in this implementation, ΔT determines when the robot will begin its attempt to move to the next specified position. It is for this reason that while in the path implementation, all

of the actual x and y values fall within the range of error specified in the code, in this trajectory version the condition of having the x and y values fall within the specified range of error is not always satisfied, as shown below in Tables 4(a-c) , but the time between transitions is always 4s. The changes made to the code are shown below in Figure . The speed was made constant at 0x12, the angle range at 8, and the xy_range at 0.01, corresponding roughly to a goal of 2% error. ΔT was set to be 4 seconds.

```
def main():
        #applying desired conditions
        move_stp_2()
        set_speed_med()
        agl_range = 8
        xy_range = 0.01

        for i in range(1, 10):
                ts = time.time()
                cur_ts = ts

                ar, xr, yr = det_fm_info_start()
                xn, yn = det_next_pos(i)
                theta, case = check_case(xr, yr, xn, yn)
                agl, is_right = det_rot_angle(ar, theta, case)

                while cur_ts < ts + 4: #while loop is broken once current position path surpasses 4s,
cur_time is compared with time the robot began at for this position
                        cur_ts = time.time() #current time is tracked
                        ar, xr, yr = det_fm_info_start()
                        theta, case = check_case(xr, yr, xn, yn)
                        agl, is_right = det_rot_angle(ar, theta, case)

                        if(((ar < (agl+agl_range)) and (ar > (agl-agl_range))) and
not(check_xy(xr, yr, xn, yn, xy_range))):
                                move_fwd_2()
                        elif(not((ar < (agl+agl_range)) and (ar > (agl-agl_range)))):

                                if(is_right):
                                        rotate_r_3()
                                else:
                                        rotate_l_3()
                        else:
                                move_stp_2()
                print(xr, yr, xn, yn, cur_ts)
main()
```

Figure 10. Code for Trajectory implementation

The trajectory implementation data for different time step increments, x-y range, and speeds are shown in Tables 4(a-c) below.

Table 4a. Trajectory implementation data (`ΔT = 4, xy_range = 0.03, speed = 0x12`)

	Actual x	Actual y	Goal x	Goal y	Timestamp	x %error	y %error
0	0.85	0.5	0.85	0.5	1525332100	0	0
1	0.8245902061	0.7138840556	0.83	0.72	1525332104	0.65	0.85
2	0.7754206657	0.8311188817	0.77	0.84	1525332108	0.7	1.06
3	0.6889204383	0.7982444763	0.68	0.8	1525332112	1.31	0.22
4	0.5725470185	0.6194565892	0.56	0.62	1525332117	2.24	0.09
5	0.4962631166	0.4268939197	0.44	0.38	1525332121	12.79	12.34
6	0.3334735632	0.201930359	0.32	0.2	1525332125	4.21	0.97
7	0.22823973	0.1684586257	0.23	0.16	1525332129	0.77	5.29
8	0.1741695702	0.2691287994	0.17	0.28	1525332133	2.45	3.88
9	0.1479895115	0.494828105	0.15	0.5	1525332137	1.34	1.03

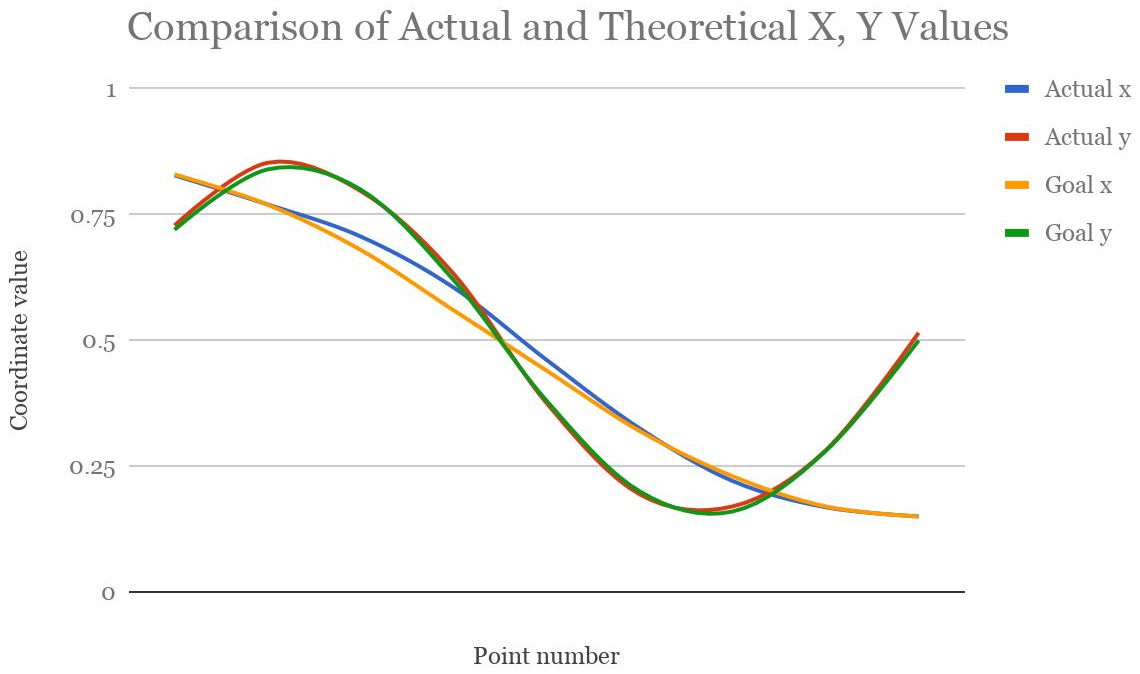

Figure 11. Plot of actual and theoretical x, y values for Table 4a

Table 4b. Trajectory implementation data (ΔT = 4, xy_range = 0.01, speed = 0x12)

	Actual x	Actual y	Goal x	Goal y	Timestamp	x %error	y %error
1	0.8281140327	0.7284964919	0.83	0.72	1525332413	0.23	1.18
2	0.7704327106	0.8531468511	0.77	0.84	1525332417	0.06	1.57
3	0.7066761255	0.797151804	0.68	0.8	1525332421	3.92	0.36
4	0.6051409841	0.6324737668	0.56	0.62	1525332425	8.06	2.01
5	0.4613090158	0.3754953444	0.44	0.38	1525332429	4.84	1.19
6	0.3251966834	0.1941943467	0.32	0.2	1525332433	1.62	2.9
7	0.2205747366	0.1700684726	0.23	0.16	1525332438	4.1	6.29
8	0.1684331298	0.2815195024	0.17	0.28	1525332442	0.92	0.54
9	0.1502130628	0.5153263211	0.15	0.5	1525332446	0.14	3.07

Table 4c. Trajectory implementation data (ΔT = 3.5, xy_range = 0.005, speed = 0x12)

	Actual x	Actual y	Goal x	Goal y	Timestamp	x %error	y %error
0	0.85	0.5	0.85	0.5	1525332870	0	0
1	0.8292176127	0.7291375399	0.83	0.72	1525332874	0.09	1.27
2	0.768635273	0.8439612985	0.77	0.84	1525332878	0.18	0.47
3	0.7467220426	0.8175408244	0.68	0.8	1525332881	9.81	2.19
4	0.5771634579	0.602163434	0.56	0.62	1525332885	3.06	2.88
5	0.5278190374	0.4496066272	0.44	0.38	1525332889	19.96	18.32
6	0.3465198874	0.1893648058	0.32	0.2	1525332892	8.29	5.32
7	0.230124563	0.1604385227	0.23	0.16	1525332896	0.05	0.27
8	0.168908447	0.2777971923	0.17	0.28	1525332899	0.64	0.79
9	0.1423131526	0.4938374162	0.15	0.5	1525332903	5.12	1.23

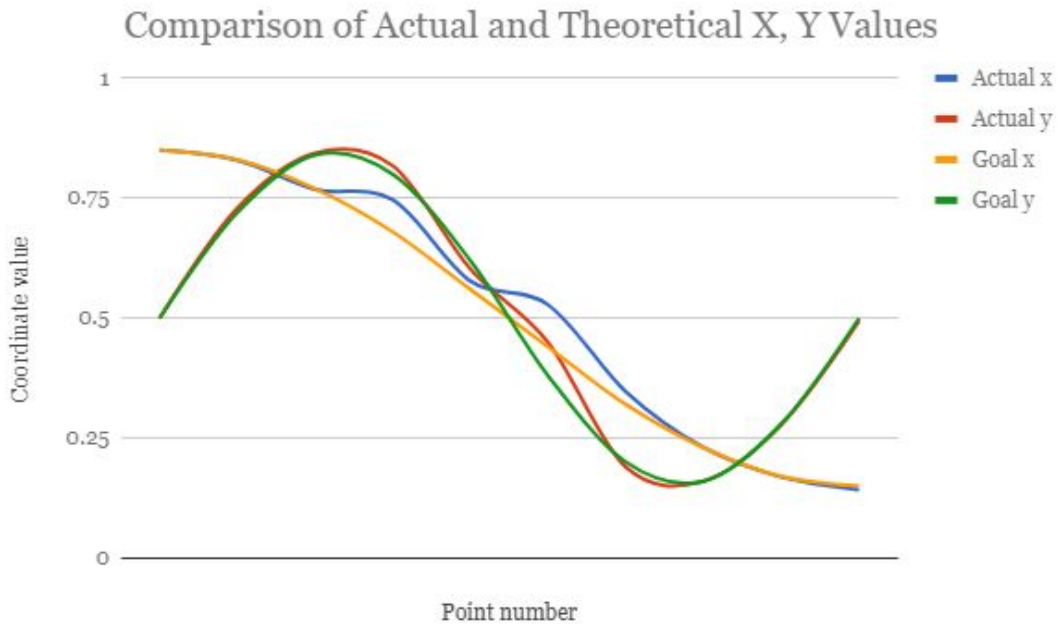

Figure 12. Plot of actual and theoretical x, y values for Table 4c

Conclusion

The results from the path following and the trajectory tracking tasks indicate that there is an inverse relation between the speed and time spent at each node. For a smaller angle range, there is more adjusting required along the path and this may lead to a greater (or lower) accuracy in the x- and y-coordinates depending on the speed of the robot. Another factor we had to consider is the motion blur, which imposed a limit on the speed of the robot due to the blurring of the input video feed taken by the overhead camera for processing on reacTIVision.

References

[1] R. Becina, M. Kaltenbrunner and S. Jordá, "Improved Topological Fiducial Tracking in the reacTIVision System.," Barcelona, 2005.
[2] Ramos , João Luiz. “Visual Control of Mobile Robots.” Portfolio Home Page, MIT, web.mit.edu/jlramos/www/visual_control.html.
[3] “TcpCommunication” , Python Wiki, wiki.python.org/moin/TcpCommunication.
[4] “TUIO 1.1 Protocol Specification.” TUIO Protocol Specification 1.1, TUIO.org, www.tuio.org/?specification.