Robot Lab Report

Hazem Ibrahim (hi387) & Mounir Elgharabawy (mke255)

Abstract:

This final project required the creation of a control system to allow for a robot, which comes in the form of a car, to complete a number of different tasks. The tasks required the robot to firstly, travel from an arbitrary point to another. Secondly, the robot must follow another robot without crashing into the leader. Finally, the robot must follow a predetermined trajectory consisting of an array of x and y tuples.

In order to track the location of the robot, reacTIVision was used. ReacTIVision is a software which tracks fiducial markers and provides a number of different properties about the fiducial marker to the user, including x and y position, angle, and id of the fiducial. To utilize this software, a fiducial marker was placed on top of the robot, which in turn allowed for a constant feedback loop between the robot's movement and the camera.

Using the python library Pytuio, the information relayed from reacTIVision was converted to python objects to allow for easy manipulation within the framework built.

The code for this project and videos of the robot in motion can be found at the following github repository:

https://github.com/hazemibrahim97/Embedded_Systems_Final_Project

Setup:

The project utilized a number of different components including :

1. 51duino Robot/Car

    - The main component of the project, which houses a number of different components, including a camera, a microcontroller, as well as a WIFI module.
2. Logitech Camera
    - A simple camera which was used to capture a live stream of the area where the robots are moving.
3. ReacTIVision software
    - Software to detect and relay information about fiducial markers in the camera's field of view [1].
4. Pytuio library
    - A python library which is able to receive and parse data using the TUIO protocol, which is specially designed to transmit the state of objects [2].
5. Python Network Sockets

The fourth fiducial id shown below was used to interact with reacTIVision.

Figure 1 : Fiducial Marker Used

The system works by transmitting commands from a controller PC to the robot through a Wi-Fi network created by the robot's Wi-Fi module. The commands the robot can receive are pre-set and can be found in the command datasheet [3]. The PC identifies and locates the robot and the target by using the reacTiVision software. The information extracted by the video stream is then sent to our python script through the TUIO protocol [4]. The script makes all the necessary calculations based on the robot's location, orientation and the target's location and sends the robot its next command using python network sockets [5].

Implementation (Procedure):

Before the tasks were implemented, a `Vehicle` class was created. This class contains all of the commands that can be sent to the robot, and acts as a wrapper for the socket communication. This class also contains the logic that allows the robot to rotate and make turns at different speeds. Additionally, it tracks the current action the robot is undertaking through the usage of the `status` attribute.

Basic Commands:

The basic commands implemented in the `Vehicle` class are shown below. These commands are almost never used in main.py, as they are very low-level commands. The commands include:

1. Stop
2. Forward/Backward

3. Rotate Left/Right
4. Set Left/Right wheel speed
5. Connect

The robot command methods (1-4) handle sending basic commands to the robot. The hexadecimal values of the command taken from the datasheet were stored in constants as bytearrays (variables written in capitals). It is worth noting that the commands SETLEFT and SETRIGHT were kept in their hexadecimal string format rather than a bytearray, to easily modify the value of the speed before converting and sending. The methods also include status checking for illegal status transitions, as well as setting the status if the command is sent, to signify that the robot is now performing a different action. Generally, illegal status transitions happen if the direction of the wheels needs to be reversed (ex. Forward -> Backward) as these transitions can cause damage to the wheel motors [command datasheet]. Additionally, if the robot is already performing the required action, the command is not sent as that would put unnecessary load on the Wi-Fi module.

The connect command connects the PC's network socket to the robot's Wi-Fi module on port 2001.

```
def connect(self, ipAddr, port):
    try:
        self.sock.connect((ipAddr, port))
        self.connected = True
        self.stop()
    except:
        self.connected = False

def stop(self):
    if self.status == 0: return
    print "Stop"
    self.sock.send(STOP)
```

```
            self.status = 0

      def forward(self):
            if self.status == 3 or self.status == 5 or self.status == 6:
return
            print "curving"
            self.sock.send(FORWARD)

      def backward(self):
            if self.status == 4: return
            self.sock.send(BACKWARD)
            self.status = 4

      def rotate_left(self):
            if self.status == 2: return
            print "rotating left"
            self.sock.send(ROTL)
            self.status = 2

      def rotate_right(self):
            if self.status == 1: return
            print "rotating right"
            self.sock.send(ROTR)
            self.status = 1

      def set_left_speed(self, leftspeed):
            if self.status != 0 and self.status != 3 and self.status != 5 and
self.status != 6: return
            commandleft = SETLEFT.replace("x", hex(leftspeed).replace("0x",
"").zfill(2))
            self.sock.send(bytearray.fromhex(commandleft))

      def set_right_speed(self, rightspeed):
            if self.status != 0 and self.status != 3 and self.status != 5 and
self.status != 5: return
            rightspeed = hex(int(rightspeed))
            rightspeed = rightspeed.replace("0x","")
            rightspeed = rightspeed.zfill(2)
            commandright = SETRIGHT.replace("x", rightspeed)
            self.sock.send(bytearray.fromhex(commandright))
```
Code Snippet 1: Robot basic commands

Composite Commands:

It makes sense to group some of these basic commands together into a function to perform a more complex action. These composite commands include:

1. Go_forward
2. Curve_left/right

The go_forward method commands the robot to move forward with a certain speed specified in the argument passed to the function. The curve left/right methods command the robot to move forward with a difference in speed applied to each of the wheels. The difference is used as a percentage of the forward speed. This allows the robot to make turns instead of just rotating in place. The forward speed and the difference are passed in as arguments into the method. These methods are almost not used at all in main.py.

```
def curve_right(self, forwardSpeed, difference):
      mySpeedRight = int(forwardSpeed - difference*forwardSpeed/100)
      mySpeedLeft = int(forwardSpeed + difference*forwardSpeed/100)
      print "curve right"
      self.set_right_speed(mySpeedRight)
      self.set_left_speed(mySpeedLeft)
      self.forward()
      self.status = 5

def curve_left(self, forwardSpeed,difference):

      mySpeedRight = int(forwardSpeed + difference*forwardSpeed/100)
      mySpeedLeft = int(forwardSpeed - difference*forwardSpeed/100)
      print "curve left"
      self.set_right_speed(mySpeedRight)
      self.set_left_speed(mySpeedLeft)
      self.forward()
      self.status = 6

def go_forward(self, speed):
      print "moving forward"
      self.set_left_speed(speed)
      self.set_right_speed(speed)
      self.sock.send(FORWARD)
      self.status = 3
```

Code Snippet 2: Composite commands

High-level Methods:

The Vehicle’s high-level methods are the ones primarily utilized in main.py. These methods are:

1. Curve_to
2. Rotate_to

Curve_to and Rotate_to take the same set of arguments: The current orientation, the required orientation, the current distance from the target, and the acceptable error for the angle. Curve_to takes an additional argument, the forward speed, which sets the average forward speed of the motors.

```
def rotate_to(self, cur_angle, req_angle, dist, error):
      angle = req_angle - cur_angle

      anglediff = abs(angle)
      if anglediff > 180:
            anglediff = 360 - anglediff

      if anglediff < error: return False
      # we are definitely going to turn
      if self.status != 1 and self.status != 2:
            self.stop()

      direction = 1 # CW
      if angle > 180 or (angle < 0 and angle > -180):
            direction = 0

      rotation_speed = int(anglediff / (180 * 4) * 100)
      # setting the speeds is an action in itself
      self.set_left_speed(rotation_speed)
      self.set_right_speed(rotation_speed)
      if direction == 0:
            if self.status == 1: self.stop()
            self.rotate_left()
      else:
            if self.status == 2: self.stop()
            self.rotate_right()

      return True
```

```
    def curve_to(self, cur_angle, req_angle, error, dist, fwd):
        angle = req_angle - cur_angle

        anglediff = abs(angle)
        if anglediff > 180:
            anglediff = 360 - anglediff

        if anglediff < error: return False
        if anglediff < 10:
            self.go_forward(fwd)
            return True
        # we are definitely going to turn

        if self.status != 5 and self.status != 6:
            self.stop()

        direction = 1 # CW
        if angle > 180 or (angle < 0 and angle > -180):
            direction = 0

        rotation_speed = int( anglediff/50 *100)
        # setting the speeds is an action in itself

        if direction == 1:
            self.curve_left(fwd, rotation_speed)
        else:
            self.curve_right(fwd, rotation_speed)

        return True
```
Code Snippet 3: Curve_to and rotate_to functions

Rotate_to decides whether the car needs to rotate or not, based on the absolute difference of the angles. If the angle difference is lower than the acceptable error, the vehicle does not rotate, and the method returns False, signalling that the rotate command was not transmitted. If the vehicle is to rotate, the rotation speed is calculated based on the angle difference using the formula: $rotationSpeed = \frac{difference}{180} \times 25$. The formula is used to map the difference in angles to a value between 0-25. 25 was chosen as the maximum value for rotation speed to avoid the effects of motion blur, or turning too fast for the camera's refresh rate, resulting in a cycle of overshooting and corrections. The relative angle difference is also used to determine the direction of rotation needed. The method used is in code snippet 3.

As for the curve_to function, a similar algorithm is used to determine the direction the robot needs to go in, as well as the acceptable orientation. Additionally, if the angle difference is less than 10, the robot just moves forward with no curving. Otherwise, the difference parameter for the curve function is calculated as:

$$diff = \frac{angleDifference \times 50}{100}.$$

Main Script:

The main script receives the tracking information from reacTiVision using the TUIO protocol. It then passes it into the methods in the Vehicle class. All decision-making is left to the vehicle, and the main function only calls the rotate_to, curve_to, connect and stop methods. The main function also handles calculating the distance and the required orientation using the following functions:

```
def get_required_orientation(srcx, srcy, dstx, dsty):
      if(dstx < srcx):
            return math.degrees(math.atan((dsty - srcy) / (dstx - srcx))) +
180
      elif (dstx > srcx):
            if(dsty < srcy):
                  return math.degrees(math.atan((dsty - srcy) / (dstx -
srcx))) + 360
            elif (dsty > srcy):
                  return math.degrees(math.atan((dsty - srcy) / (dstx -
srcx)))

def get_distance(srcx, srcy, dstx, dsty):
      return math.sqrt((dstx - srcx)**2 + (dsty - srcy)**2)
```

Code Snippet 4: calculation helper functions

In addition to calling the Vehicle class, the main function also keeps updating the TUIO tracker for any new information. To avoid receiving the same data multiple times due to the camera's relatively slow refresh rate, an if statement makes sure that the values were updated before going into the main body of the code (snippet 4).

```
if prev_x == robot.xpos and prev_y == robot.ypos and myVehicle.status != 0:
      continue
elif cur_angle == prev_angle and myVehicle.status != 0 and myVehicle.status !=
3 and myVehicle.status != 4 and prev_x == robot.xpos and prev_y == robot.ypos:
      continue
else:
      prev_angle = cur_angle
      prev_x = robot.xpos
      prev_y = robot.ypos
```

Code Snippet 5: making sure the values were updated

Finally, the main script handles the event when the robot reaches its target. If the distance of the robot from the target is less than the desired distance, the robot stops. On the other hand, in the trajectory mode, the trajectoryindex is updated so that the target of the robot is the next point in the path. When the robot arrives at a point in trajectory mode, the robot's coordinates and time since start are written into a csv file, to track its error and speed.

Results:

Task 1 (Point A to Point B):

In this task, the robot was required to move from a point A to an arbitrary point B, which is marked by a fiducial marker to be placed on a robot or on the ground. If the robot was within a ± 50 degree angle from the target point, the robot would begin curving towards the point using the process previously described. Otherwise, the robot would rotate either right or left, depending on which side is nearer to the target, and then begin curving towards the target.

Figure 2 : Task 1

Task 2 (Follow another robot):

In this task, the robot is required to follow another robot without crashing into it. This is done using the same procedure as the previous task, where if the angle between the robot's current orientation is ± 50 degrees, the robot would rotate until is is within ± 50 degrees from its required orientation, and then proceed to curve to the target. The only difference between this task and the previous task, is that the target's position is constantly changing, however, the code behind both of these tasks is the same. The minimum distance between the robot and the target was set to be 0.2. The robot was successfully able to follow the evading robot without crashing into it.

Figure 3 - Task 2

Trajectory Task:

In this task, the robot was required to follow a specific trajectory marked by 10 points. The required trajectory can be seen in the figure below.

Figure 4 - Ideal Trajectory of the Robot

A graph comparing the robot’s path and the ideal path can be seen below.

Figure 6 - Robot Path vs Ideal Path in Trajectory Task

The total time for the robot to move from the first trajectory point and the final trajectory point was found to be approximately 7.712 seconds. Below, a table indicating a number of values regarding the path taken by the robot can be seen.

Table 1 : Values regarding robot path in trajectory task

Discussion:

Task 1 and 2:

In the task, the robot would rotate if the angle between the robots current orientation and the target was greater than 50 degrees. If the minimum required rotation angle was set lower than 50, the robot would be spending more time rotating rather than moving towards the target in a curve, whereas if the value was set much greater than 50, the robot will travel at a large curve resulting in a larger overall travel time. Therefore, setting the angle at 50 is a good compromise between time spent rotating towards the correct orientation, and time spent travelling at a large curve to arrive at the target. In the second task (following another robot), the minimum distance was set to 0.2 as to not crash into the other robot. This value was set after many trials with various values. Although the minimum distance can be set to a smaller value, depending on how the fiducial marker is placed on the evading robot, the following robot may crash. This is because the camera sends the location of the fiducial rather than the robot, therefore a value of 0.2 was set as a precaution to not crash into the evading robot.

Trajectory Task:

As can be seen in Figure 6, the robot takes a similar curve when compared to the ideal path. In the implementation of the trajectory task, the distance required between the robot and the trajectory point before moving on to the next point was set to 0.1. It should be noted that in this setup, 0.1 does not hold any units, but it is relative to the field of view of the camera, where maximum distance along both the x and y axes were set to 1. If this required distance was set to a value lower than 0.1, the robot would sacrifice time for error, as decreasing the distance would require a greater amount of rotating for angle corrections, especially at small distances. Therefore, this setup was determined to be a good trade-off between time and error.

General Discussion:

The project could be improved by a number of different factors. Firstly, the control of the robot in both task 2 and the trajectory task can be greatly improved by operating the robot using assembly functions or register manipulation rather than high level functions. This would speed up the process of controlling the motors and allow for a quicker response time by the robot. In addition, the information sent by reacTIVision, particularly during instances where the robot is moving at a high speed, can be unreliable due to motion blur. To fix this, certain settings on reacTIVision, including adjusting exposure and manually adjusting focus can theoretically eliminate these issues, however these modifications were not tested.

Moreover, using a PID controller could have also improved our rotation script, by accurately arriving at the correct orientation instead of overshooting the target [5]. Bounds checking could have also been implemented to stop the robot from hitting the boundaries of the arena set up.

References:

[1] - “ReacTIVision 1.5.1.” ReacTIVision, reactivision.sourceforge.net/.

[2] - “Pytuio.” PyPI, pypi.org/project/pytuio/.

[3] - “TUIO.org.” TUIO, www.tuio.org/.

[4] - NYUClasses. “DSRobot Quickstart Guide.” DSRobot Documentation.

[5] - “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.

[6] - “PID Controller.” Wikipedia, Wikimedia Foundation, 24 Apr. 2018, en.wikipedia.org/wiki/PID_controller.