Design and Specifications of Autonomous Robot

Harsha Cuttari, Emanuelle Crespi, Nikhil Badami (Team 1)

Contents

1 Overview of Task

Our main goal for this project was to design and implement an autonomous robot capable of communicating with other autonomous robots to move in formations and avoid obstacles. The overall task was separated into different challenges: obstacle avoidance, line formation, rank formation and class challenge. For purposes of this paper, we will consider only the first challenge, line formation, as this was the furthest accomplishment by our class.

Obstacle avoidance focuses on motor control of the individual robot and its response to the environment. For this task, a robot is expected to move forward and avoid any obstacles present in front of it. We were given Ping sensors for this task to detect the distances of the obstacles. If the robot comes across a corner, it would be expected to navigate out of the trap.

For line formation, our autonomous robot was expected to move in line with other robots, a leader-follower configuration, one behind another. Each robot would have to maintain a fixed spacing. The line of robots would then have to traverse a simple region without obstacles and then advance to a complex region with obstacles. If the robot leaves the line, it must then rejoin the line as soon as possible.

After successfully completing the line formation task of following each other, the next step would be to switch positions of the leaders. For example, a random robot in the line is designated as the leader and moves to the front of the line, assuming the duties of the leader.

2 Requirements Analysis

For the main objective of line formation, we broke the project down into smaller subsections. After mastering each of the subtasks, we would then be able to create a fully functioning autonomous robot capable of completing the assigned challenge. Specifically, we broke the objective into 9 different subtasks:

- Basic Motor Control
- Obstacle Avoidance with Ping Sensors
- Avoid Trapping Behavior
- Tennis Ball Tracking
- Arduino 101/ Kangaroo Communication
- Communication via Server/Clients
- Robot Identification

After completing each subtask successfully, we would then combine them with other subtasks to complete a larger task. While some of the subtasks were successful, we pivoted from the original direction in the later subtasks due to limitations presented by the equipment.

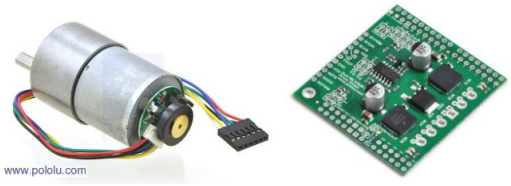
The robot must first move forward, stop and make directional changes. It then would detect obstacles via Ping sensors and determine maneuvers accordingly - including if it is trapped. Next, the robot should identify tennis ball and determine the location respective of the screen. It will communicate this information to the Arduino 101 where it can then maneuver towards the ball and maintain distance. For communication between the multiple robots, a server must be set up where the robots would act as clients to relay information to the server. The server would identify and sort the robot's location based on

the data. After these subtasks are completed in succession, we can proceed with the line formation objective.

3 Hardware Component and Use

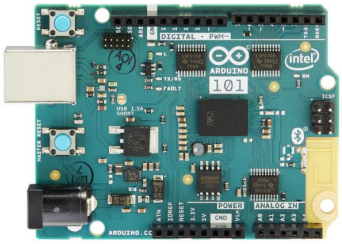
The following section will list all the hardware used in the project with their main functions in the robot.

3.1 Motors and Motor Control



We used two bidirectional brushed DC motors to power the wheels for our robot. The motors were connected to a dual motor driver which then connected to the Arduino 101. The motor driver allowed us to operate at five volts and accept the Pulse Width Modulation (PWM) from the Arduino 101. The motor drivers were set to slightly different speeds to account for the different specifications of both the motors and the wheels in order to maintain a straight line while moving forward.

3.2 Arduino 101



We used the the Arduino 101 Board as the low level controller for the robot. The board managed maneuver control and the robot's response to the low level (Ping) sensors for obstacle avoidance. The board also communicates with the Kangaroo via OpenCV. The board provided output the five volts to our Ping sensors as well. The Arduino 101 was powered by the Kangaroo.

3.3 ELP 170 degree Fisheye Wide Angle Camera

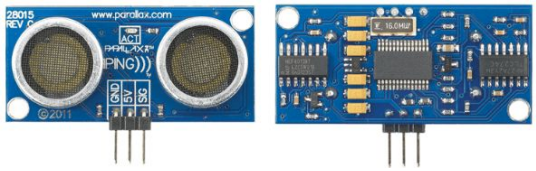
The 170 degree wide angle camera was used to mainly perform image processing and determine the distance and position of the tennis ball required for following the robots. The camera was connected to the Kangaroo where it relayed all the image data. The camera was able to detect colors and use auto brightness settings.

3.4 Kangaroo

The Kangaroo Mobile Desktop runs Windows 10, though other operating systems can probably be installed. It can be programmed using a high level language, such as Python which was used in this project. We ran OpenCV for image processing. It has built in BLE and WiFi, both useful for our project. The Kangaroo has 2 USB ports, 1 HDMI port, and an SD card reader for I/O.

We first used the Kangaroo to read in the data from the camera and send the signals required for motor control on the Arduino 101. Later in the class, we used Kangaroo to also communicate to a central server to receive additional commands to relay to the Arduino 101.

3.5 Ping Sensors



We used ultrasound sensors to measure distance over short ranges. Each of the autonomous robots were equipped with 2 Ping sensors for obstacle avoidance. The sensors have a limited distance and were somewhat noisy.

4 Software Design, Components, and Implementation

4.1 Basic Motor Control

The motor control has been implemented in the source code for the Arduino 101. Each wheel has it's own attributes pins/values for defining movement and direction. These include, default speed, motor pin, and a-b pin values. We defined values on pins *aX* and *bX* for each respective *motorX* (X = 1 or 2). At the head of the source code, we will define these values as constants for the following code.

```
const static int MAX_SPEED = 55;
const static int MAX_DIST = 9;
const static int SPEED_UP = 1;
```

The setup() assigns these pins as INPUT types on the hardware. The threshold values approximate the pulse-width-modulation (pwm) on the motor controller to signal which wheel we would like to drive at some *default* speed (*mX_default* has been tested to work for our circuit but could vary in others).

```
const static int MAX_SPEED = 55;
const static int MAX_DIST = 9;
const static int SPEED_UP = 1;

//left
const static int motor1 = 6;
const static int b1_dig = 13;
const static int a1_dig = 7;
const static int m1_default = 40;
static int m1_speed = m1_default;
static int dir1 = 1;

//right
const static int motor2 = 5;
const static int b2_dig = 11;
const static int a2_dig = 8;
const static int m2_default = 50;
static int m2_speed = m2_default;
static int dir2 = 1;
```

```
void setup() {
  //wheel 1 setup
  pinMode(a1_dig,INPUT);
  pinMode(b1_dig,INPUT);
  pinMode(motor1, INPUT_PULLUP);

  //wheel 2 setup
  pinMode(a2_dig, INPUT);
  pinMode(b2_dig, INPUT);
  pinMode(motor2, INPUT_PULLUP);

  Serial.begin(115200);

  //Serial.begin(9600);
}
```

NOTE: *Serial.begin(115200) sets up a hardware link to an available COM port on the hardware. It is useful as a debugging mechanism in this phase of the motor control.*

We use the function call "drive(dir,...)" in the loop() to specify the wheel, direction, and speed we would like to assign on that object. This is the basic setup before adding further logic to the code for turning and escaping procedures. [It is always assumed dir1 = dir2 = 1]

```
//before driving the motor want to check that speed isn't over the max
if( m1_speed == MAX_SPEED ){ m1_speed = m1_default; }
if( m2_speed == MAX_SPEED ){ m2_speed = m2_default; }

//motion on wheel1 and wheel2
drive( dir1, motor1, m1_speed, a1_dig, b1_dig );
drive( dir2, motor2, m2_speed, a2_dig, b2_dig );
```

NOTE: *There is some logic above the calls to drive(...) to make sure we do not exceed a threshold on the wheels speeds. This applies to cases (turns, etc…) where m1_speed or m2_speed have to increase.*

```
/* Accepts appropriate pins for motor a, and b to
 * power the motor at pwm speed mSpeed.
 * dir = 1 will indicate forward motion
 * dir = -1 will indicate backward motion
 */
void drive(int dir, int motor, int mSpeed, int a, int b) {
  analogWrite(motor,mSpeed);

  switch( dir ){
    case 1:
      digitalWrite(a,HIGH);
      digitalWrite(b,LOW);
      break;

    case -1:
      digitalWrite(a,LOW);
      digitalWrite(b,HIGH);
      break;

    default:
      digitalWrite(a,HIGH);
      digitalWrite(b,LOW);
      Serial.print("Default in drive(...)???");
      break;
  }

}
```

4.2 Obstacle Avoidance with Ping Sensors

The obstacle avoidance is dependent on the source code that reads from the ping sensors. Again at the head of the source code we need to assign a pin for the ping sensors in our circuit.
We also decided to assign boolean flags to indicate an object within the threshold distance of

```
//flags
static boolean leftObs = 0;
static boolean rightObs = 0;

//will toggle betwwen 2 and 4
static int pingPin = 2;
```

We proceed with the following logic on variable *pingpPin* to make sure one ping sensor is used every other loop() cycle to measure distances.

```
//process pingPin 2/pingPin 4
pinMode(pingPin, OUTPUT);
digitalWrite(pingPin, LOW);
delayMicroseconds(2);
digitalWrite(pingPin, HIGH);
delayMicroseconds(5);
digitalWrite(pingPin, LOW);
pinMode(pingPin, INPUT);
duration = pulseIn(pingPin, HIGH);

inches = microsecondsToInches(duration);
cm = microsecondsToCentimeters(duration);

if( pingPin == 2 ){
  right = inches;
}else if( pingPin == 4 ){
  left = inches;
}else{
  Serial.print("??????????");
}

//setting flag to signal left/right turn
leftObs = (left <= MAX_DIST);
rightObs = (right <= MAX_DIST);

// toggles between pin2 and pin4 for the ping
pingPin ^= 6;
```

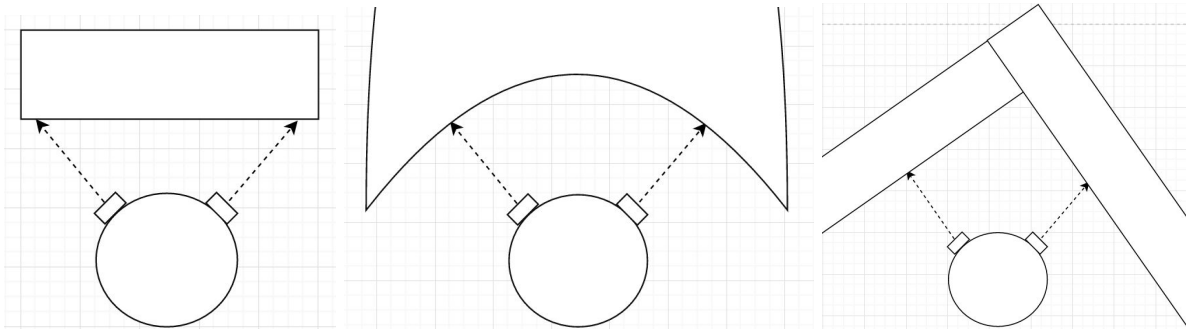
NOTE: *Before the next cycle, we have pingPin ^= 6 to toggle between values 2 and 4.*

After defining direction in every cycle, we applied further logic to before calling drive(...) so we can have the wheels dodge any obstacles within the range of *MAX_DIST*. This involves re-defining the attributes of the wheels so we can add turning behavior to the robot in particular situations after reading distance from the pings (*leftObs == true or rightObs == true*).

```
//for the pings
//assume forward motion
dir1 = 1;
dir2 = 1;

/*************************************
 * Routines to process for the the wheels
 * speeds & direction deepending on pings.
 *
 * ***********************************
 * //FLAGS/FILTERS Initializations
 * static boolean leftObs = 0;
 * static boolean rightObs = 0;
 *
 * static boolean escape = false;
 * static int eFilter = 0;
 * static boolean backup = false;
 *************************************/
if( leftObs && !rightObs ){
    //turning right
    Serial.print("Dodge right");
    m1_speed += SPEED_UP;
    m2_speed = 0;
}else if( rightObs && !leftObs ){
    //turning left
    Serial.print("Dodge left");
    m1_speed = 0;
    m2_speed += SPEED_UP;
}else if( leftObs && rightObs && eFilter < 40 ){
    if( data != 48 ) eFilter++;          //process this request for 40 cycles
    Serial.print("TRAPPED"); //debug on SERIAL
    m1_speed = 0;
    m2_speed = 0;
}else{
    m1_speed = 0;//m1_default;
    m2_speed = 0;//m2_default;
}
```

4.3 Avoid Trapping Behavior

A special case to consider is one where both *leftObs == true and rightObs == true.* This is an event to consider where the robot has been trapped. This only happens when the robot has somehow positioned itself in front of a wall, curved obstacle or a narrow corner. A response has been programmed to enter a routine where the robot will “escape” by first backing up and turning around.



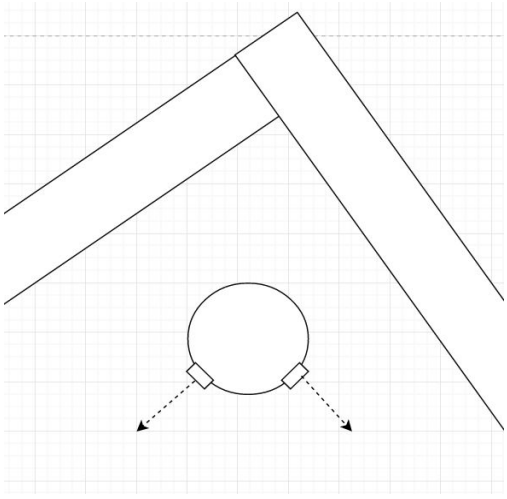
Case 1 Case2 Case3

The arduino code uses a similar approach as the obstacle avoidance and keeps track of 2 boolean flags (‘backup’ and ‘escape’) so the wheels can react appropriately. It is always assumed backup = escape = false until the robot has processed this event in a variable, *eFilter*, for 40 cycles.

```
}else if( leftObs && rightObs && eFilter < 40 ){
    if( data != 48 ) eFilter++;          //process this request for 40 cycles
    Serial.print("TRAPPED"); //debug on SERIAL
    m1_speed = 0;
    m2_speed = 0;
}else{
    m1_speed = 0;//m1_default;
    m2_speed = 0;//m2_default;
}

//only gets here if it was trapped for 40 cycles (both pings read 9)
if ( eFilter >= 40 ){
    //process this state in the filter
    eFilter++;
    m1_speed = m1_default;
    m2_speed = m2_default;

    //keep track of state flags
    if( eFilter > 40 && eFilter <= 80 ){
        //need to backup because I am trapped
        Serial.print("backup"); //debug on SERIAL
        backup = true;
        escape = false;
    }else if( eFilter > 80 && eFilter != 100 ){
        //will escape after handling backup routine
        Serial.print("Escape"); //debug on SERIAL
        backup = false;
        escape = true;
    }else{
        eFilter = 0;
        backup = false;
        escape = false;
    }

    //routine for backing up
    if( backup == true ){
        //reverse both wheels
        dir1 = -1;
        dir2 = -1;
    }

    //routine for escaping
    if( escape == true ){
        //reverse only one wheel
        dir1 = -1;
    }
```

NOTE: *eFilter keeps track of cycles*

After 40 cycles have passed, a new event is triggered, setting the *backup* = true and *escape* = false. This sets dir1 = dir2 = -1 so that the drive(dir,....) function runs the wheels 1 and 2 in reverse for 40 more cycles. At this point, eFilter = 80, and a final event, ‘backup = false, *escape = true*’, is triggered. The direction dir1 = -1 and this is processed for the last 20 cycles. By this time, a total of 100 cycles have been processed *eFilter* = 100, the bot has backed up and finished a 180 degree turn. (*eFilter* is set back to 0, *backup* = *escape* = false)



4.4 Tennis Ball Tracking

This section covers how we programmed our Kangaroo to use the camera to detect a green tennis ball, draw a circle and track the movements of the ball.

For this section, it is important to install the required programs: Anaconda, Python 3.5, OpenCV 2.4. To install OpenCV, we typed the following code in IPython:

!conda install -c https://conda.binstar.org/menpo opencv3

Import cv2 should work right after and we can make use of OpenCV. After installation of OpenCV, we need to use certain necessary packages. To do this, we typed the following into the command prompt:

OpenCV Track Object Movement — Shell

```
$ pip install imutils
```

Next, we made use of Spyder 3 and imported the necessary packages:

OpenCV Track Object Movement — Python

```
# import the necessary packages
from collections import deque
import numpy as np
import argparse
import imutils
import cv2

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video",
    help="path to the (optional) video file")
ap.add_argument("-b", "--buffer", type=int, default=32,
    help="max buffer size")
args = vars(ap.parse_args())
```

We can now proceed to track the ball:

OpenCV Track Object Movement — Python

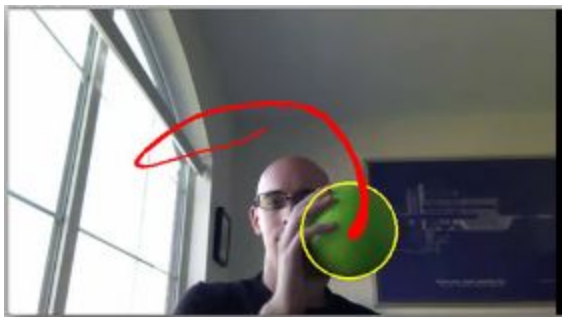
```
# define the lower and upper boundaries of the "green"
# ball in the HSV color space
greenLower = (29, 86, 6)
greenUpper = (64, 255, 255)

# initialize the list of tracked points, the frame counter,
# and the coordinate deltas
pts = deque(maxlen=args["buffer"])
counter = 0
(dX, dY) = (0, 0)
direction = ""

# if a video path was not supplied, grab the reference
# to the webcam
if not args.get("video", False):
    camera = cv2.VideoCapture(0)

# otherwise, grab a reference to the video file
else:
    camera = cv2.VideoCapture(args["video"])
```

As you can see, we first set the upper and lower bounds of the green in the tennis ball. This would account for the hue, saturation and value. We then initialize our variables and deque our buffer.

OpenCV Track Object Movement — Python

```
37 # keep looping
38 while True:
39     # grab the current frame
40     (grabbed, frame) = camera.read()
41
42     # if we are viewing a video and we did not grab a frame,
43     # then we have reached the end of the video
44     if args.get("video") and not grabbed:
45         break
46
47     # resize the frame, blur it, and convert it to the HSV
48     # color space
49     frame = imutils.resize(frame, width=600)
50     blurred = cv2.GaussianBlur(frame, (11, 11), 0)
51     hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
52
53     # construct a mask for the color "green", then perform
54     # a series of dilations and erosions to remove any small
55     # blobs left in the mask
56     mask = cv2.inRange(hsv, greenLower, greenUpper)
57     mask = cv2.erode(mask, None, iterations=2)
58     mask = cv2.dilate(mask, None, iterations=2)
59
60     # find contours in the mask and initialize the current
61     # (x, y) center of the ball
62     cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
63         cv2.CHAIN_APPROX_SIMPLE)[-2]
64     center = None
```

We then loop the capturing of the camera frames and process them one at a time to detect the green values and use blur function remove small noise. After creating a circle shape around the object, we can then calculate two important values: radius of the ball and the x-axis location of the ball relative to the screen. The radius of the ball is calculated by the changing size of the drawn circular shape on the green object detected by the camera. The x-axis location is the center point of the drawn circular shape relative to the screen. In our case, the screen frame width is 600. This means, the ball would be in center at around 300. The picture below shows the circle drawn on the ball.



The rest of the code is available in the attached files with elaborate comments. The purpose here is to guide a beginner in the first steps of installing and running OpenCV as it was difficult to find the proper documentation from the source. Also note that when starting the code, it is important to have the

ball in view of the camera lens or there might be occurrences of deque index errors. However, this is fixed to a point in the attached version of the code.

4.5 Arduino 101/ Kangaroo Communication

We now have to send the radius and x-axis values detected to the arduino. For the communication to take place, we must first install the following in the command line: pip install pyserial

Next, we setup the Arduino and Python with the right configurations. The following shows the configurations respectively:

Arduino:

```
Serial.begin(115200);
```

Python:

```
#Choose COM port and baud rate accordingly
arduino = serial.Serial('COM5', 115200, timeout=.1)
time.sleep(1) #give the connection a second to settle
```

We then send out the information to the Arduino with the following:

```
    if rad >= 45:
        movement = 0

    if xAxis >= 250 and xAxis <= 350 and radiusInRange:
        movement = 1

    if xAxis <= 250 and radiusInRange:
        movement = 2

    if xAxis >= 350 and radiusInRange:
        movement = 3

    if rad < 10:
        movement = 4

    if turn == True:
        movement = 5
        i = 0
        print("Sending 80 times bro....")
        while i < 80:
            i = i + 1
            arduino.write("%d".encode('ascii') % movement)
        turn = False
        continue

    movement = 5
    #arduino will process this state accordingly
    arduino.write("%d".encode('ascii') % movement)
```

Arduino receives the information and will have to send the information back to the Kangaroo. This is because the COM port can only be used for one instance. In our case, it would be better to debug with Python, so we will read everything the Arduino outputs on the iPython console with the following code:

Python:

```
####### DEBUG ON SERIAL ########
# data = arduino.readline()
# if data:
#     print(data)
################################
```

Arduino:

```
void loop() {
      static long duration, inches, cm;
      static long left;
      static long right;

        //Read values from Python app
        if(Serial.available() > 0) {
              int data = Serial.read();
              char str[2];
              str[0] = data;
              str[1] = '\0';
              //Serial.print("Arduino101: ");
              Serial.print(str);
```

We completed all the subtasks and combined them to complete the first objective. We were able to detect the green tennis ball via the camera and analyze the information on the Kangaroo. We used the radius and x-position of the ball to send movement commands to the Arduino to make the turn accordingly. For example, the robot would go straight if the radius was below 45 and x-axis values detected would be between 250 and 350 (to allow for camera disturbances). The priority of the robot was to first avoid obstacles and then follow the tennis ball. This is done with nested if statements in the motor code. If the ball is out of range, the robot would then spin around to find the ball and resume to follow it.

Some of the problems we faced were the lighting issues presented by the fluorescent bulbs. The tennis balls were of similar brightness and caused the camera to make multiple mistakes and caused the robot to confuse similar colors.

4.6 Communication via Server/Clients

In order to implement internet connectivity between our robot and our laptops, we decided to set up a UDP connection. This is required for the robots to “communicate” with each other. UDP does not implement clients and servers in the same way a TCP/IP connection might. There is no handshake or setup routine to establish a connection between machines. Rather, the concept of which machine is the server and which is the client is largely dependent on what each machine is doing. In our implementation, we decided to have our laptop act as the “host,” the machine that would send out commands, and our robot act as the “client,” the machine that would receive commands. Our server/client code is displayed below, with the server code appearing first:

```python
#Server
import socket

def Main():
    #The host variable stores the hosts computer IP address
    #and the serverport defines the port clients should try
    #and connect to. The IP address can be found using your
    #computers terminal and the port number is arbitrary, though
    #it is advised to use a large number like 5000 since lower
    #number ports might be reserved for specific tasks
    host = '8.8.8.8'
    serverport = 5003

    #This creates the socket object. AF_INET means the socket
    #is a streaming socket and SOCK_DGRAM defines it as a UDP
    #socket. "s.bind()" is required so that the computer knows
    #which IP address and port number the socket is associated with.
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.bind((host, serverport))

    while True:
        #Receive data and client IP/port
        data, addr = s.recvfrom(1048576)
        if not data:
            break
        print("from connected user: " + str(data))
        data = str(data).upper()
        print("sending: " + str(data))
        #send data back to client
        s.sendto(str.encode(data), (str(addr[0]), addr[1]))
    s.close()

```

```python
#Client
import socket

def Main():
    host = '8.8.8.8'
    client = '7.7.7.7'
    #This variable is needed because the client
    #will be sending and receiving data from the host
    clientport = 5007
    serverport = 5002

    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    print ('Created socket')
    s.bind((client, clientport))

    message = input("-> ")
    while message != 'q':
        s.sendto(str.encode(message), (host, serverport))
        data = s.recvfrom(1048576)
        print("Received from server: " + str(data))
        message = input("-> ")
    s.close()

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

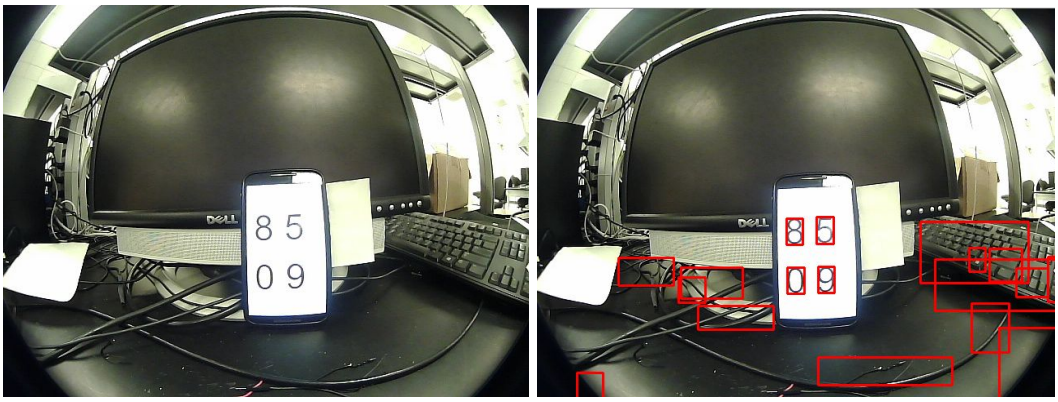
The code displayed above is the minimum amount of code needed to set up a UDP connection in Python. Any additional commands or information that the user desires to send can be added on top of this framework. It should be noted, however, that since UDP does not establish a constant connection between computers, it is possible that the data the host sends may not be received by the client. If data delivery is a priority for the user, it is recommended that they use a TCP/IP connection instead.

4.7 Robot Identification

One of the more challenging aspects of this project is to use the given hardware to detect positions of other robots without the aid of GPS. While there might be many solutions, we decided to go with the following approach:

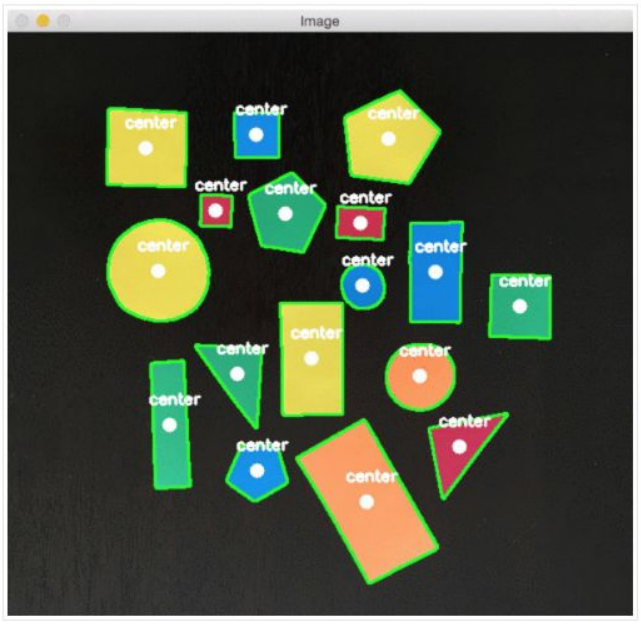
- Use a unique identification on each robot to be read by nearby robots
- Relay the read identification as well as the communicating robot's own identification to the server
- Server analyzes data from all the robots and sorts the information into an array where we can now know relative positions of each robot with respect to each other.

To accomplish these tasks, we had to choose between coloring the tennis balls a different primary and secondary colors (red, blue, yellow, etc) or another approach. The color method, we discovered with various camera tests on printed color circles, was not a viable solution. Due to the variety of colors in the ever changing lab environment, the robots would confuse their target and would relay wrong information to the server at various times. Our next solution was to use learning in robotics to detect numbers using Optical Character Recognition (OCR). This method had acceptable results but required pattern detection.



We can see in the above image that 8, 5, 0, and 9 were detected successfully, however due to large amounts of noise (mainly detecting 1 and 7 from the edges), we have to include a square number pattern and crop the image to a certain location for optimal results. Due to time constraints, we abandoned the approach as noise reduction required extensive research on unfamiliar topics.

Similar to using OCR for identification, we tried the shape detection approach with extremely optimal results.



The above image represents some of the theoretical results but real world results were just as good. We used red as the color for all objects due to unique identification properties of the color and detected a variety of shapes. Python works best when detecting multiple sided shapes and was accurate majority of the time. However, we were not able to implement this portion of the project in the final due to time constraints.

Note: It is important to realize the video from the camera is distorted due the wide angle lens used. This causes the straight lines towards the edges to become distorted and thus confuse the shapes. Some anti distortion measures in the code will help solve the problem.

5 Demonstration Results

Our final results were demonstrated on 7th of December 2016. Our section consisted of five teams. We were able to successfully form a line by following a tennis ball placed behind each of the robots. Our robot was also able to avoid obstacles and rejoin the line when the formation broke apart.

Due to lighting situations and volatility of other robots among other variables, our robot did not have a chance to perform exactly as anticipated but did manage to accomplish the given tasks. Other teams also managed to follow through on the course and followed our robot which led the line at certain points.

While communication was not completed in time, we do appreciate the hardwork and efforts of all the teams who combined efforts to help our team to achieve the number detection and shape detection code. Other subtasks were also delegated to and completed by teams 2, 3, 4 and 5 but did not get a chance to be used.

All the code written for this project (including tutorials, examples, tests, etc.) are on a shared Google Drive with the following link:
https://drive.google.com/drive/folders/0B_DjFbXv25Z3bWVyWFk1VklnRUU?usp=sharing