IoT Lab 1

Dixon Liang

September 15, 2021

Abstract

This report is for Lab 1 of the course Internet of Things (CS-437) at the University of Illinois Urbana-Champaign as part of the Masters in Computer Science program. The lab is broken down into two parts. A video of compiling each part of the lab can be found here: https://youtu.be/t4X64sTeJ_0. The source code can be found here: https://github.com/dixonliang/IOTSDCLAB1

1 Part 1

1.1 Design Considerations

The car was assembled using the generic SunFounder instructions. Figure 1 shows the car from a bird's eye view.The main design consideration which was different was the addition lsof the Pi Camera. In order to fit the camera connection, I created a small slit in the plastic casing to mount the camera in the front as shown in Figure 2.

1.2 Topology Report

Figure 3 shows the topology report for the car. This was done using Fritzing. I used a blank 4WD Hat part in the report with the pin level details being approximated on location on the hat. Another note of detail is that the 2-ch Photo Interrupter and 3-ch Gray Channel Sensor parts were not available, so I used two parts that looked similar and are as labeled. The 3-ch Gray Channel Sensor should also have 5 pin connections, instead of the 4 pin connections as indicated by the diagram.

1.3 Basic Driving

The first step of configuring the car to drive was to create code for basic navigation through an obstacle. A snippet of the code can be found below. The code follows a basic if, then, else logic for each scan of in front of the car. A sleep counter of 0.3 seconds is used after each move to give the car time to complete the move appropriately. From trial and error, I found this to be an appropriate amount of time.

```
if tmp != [2,2,2,2]: # if object is detected 20cm ...
    fc.stop() # stop
     fc.time.sleep(0.3)
    fc.backward(speed) # move backwards
    fc.time.sleep(0.3)
    fc.turn_right(turn) # turn to the right
    fc.time.sleep(0.3)
else:
    fc.forward(speed) # if not, move forward
    fc.time.sleep(0.5)
```

A full demo video of the basic driving and navigation of the car can be found at this link: https://youtu.be/D3rjOI1J1YU

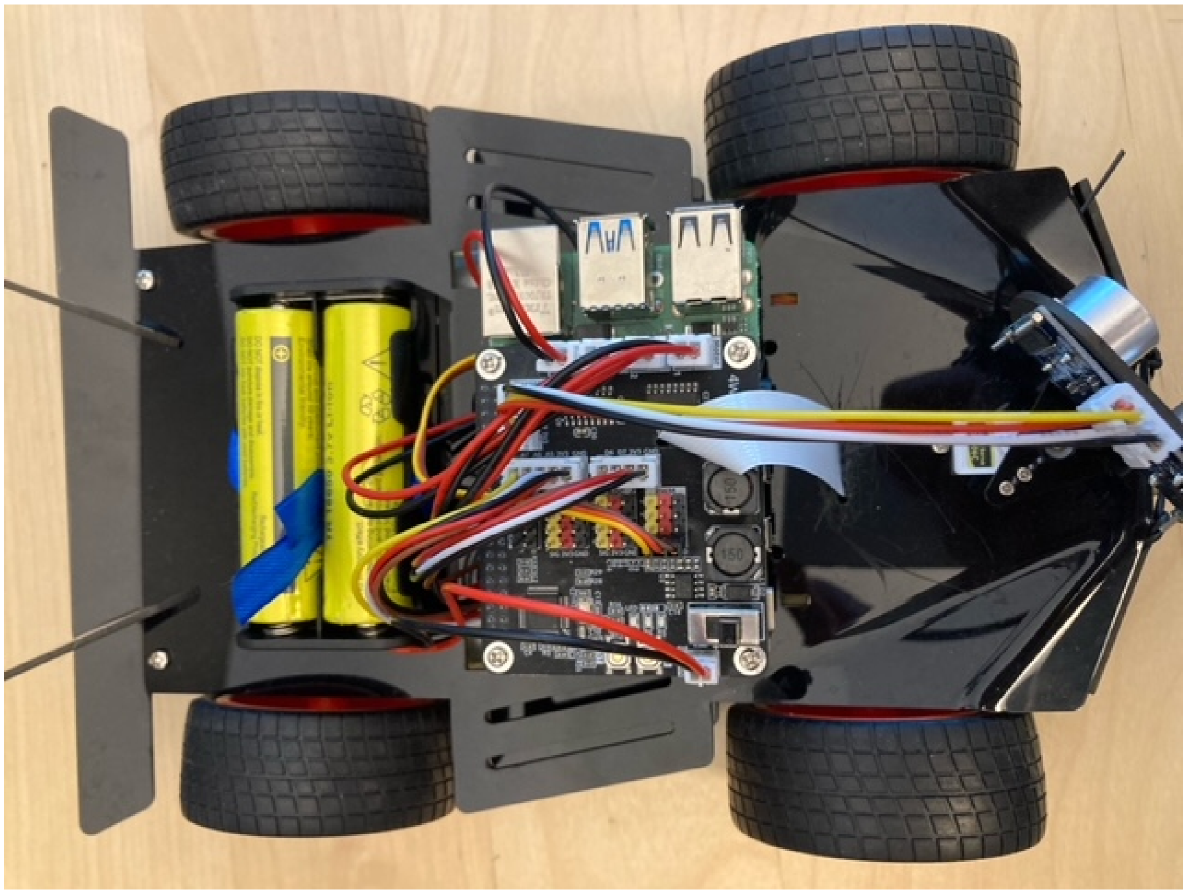


Figure 1: Assembled Car.

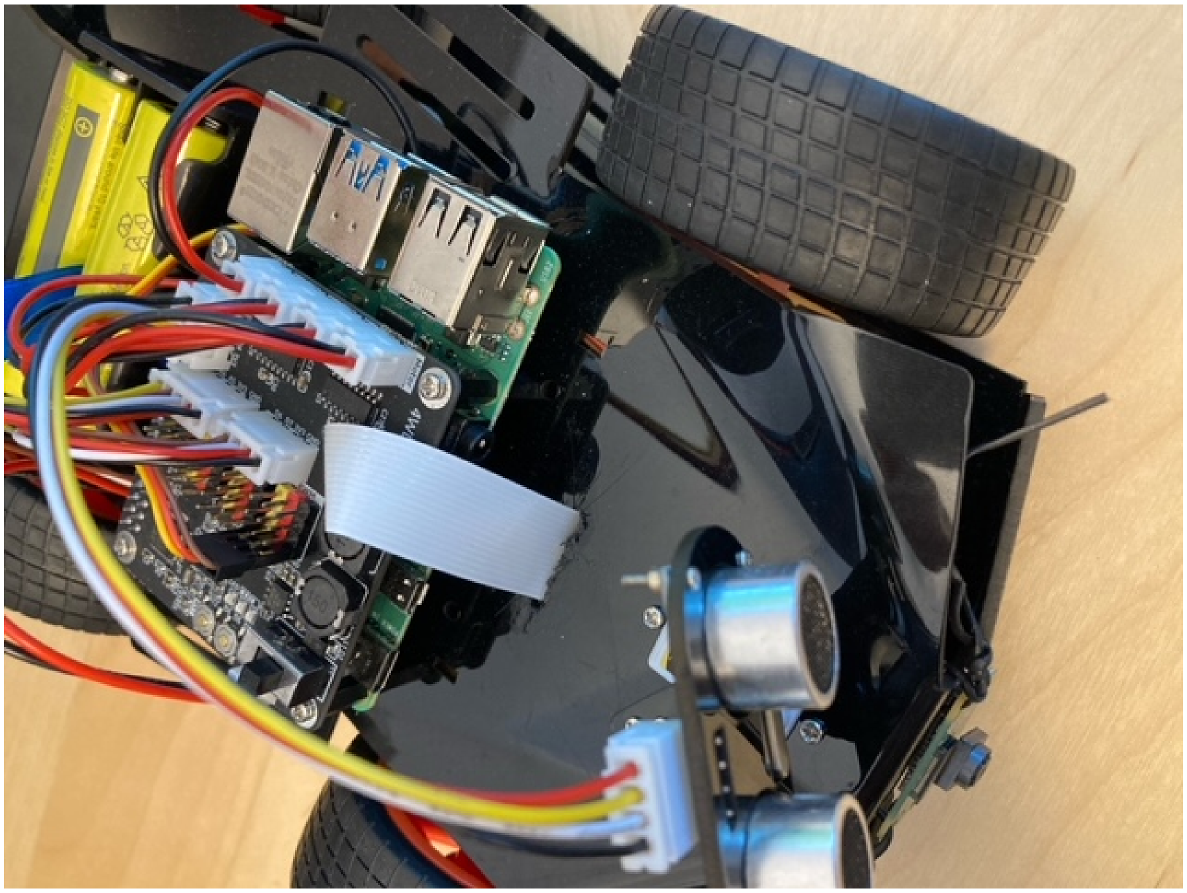


Figure 2: Pi Camera mounted on the front of the car using slit.

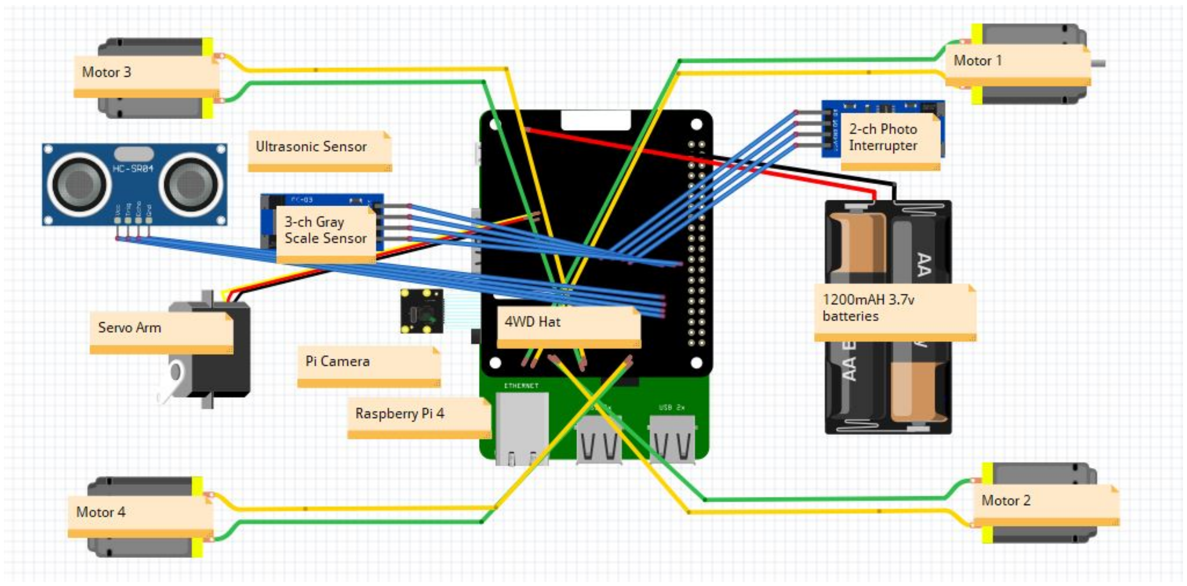


Figure 3: Topology Report.

2 Part 2

2.1 Mapping

The first step of Part 2 of the lab required was to create a mapping of the area in front of the car. The basic logic of performing this to create a numpy arrays representing a map of the car's surroundings. 0 will represent nothing in the grid area, while a 1 will represent something in the grid area.

For the most accurate reading, I decided to create a run with taking a distance reading every degree using "fc.get-distance-at()". This function returns the reading from the ultrasonic sensor of the estimated distance of the first object based on the time for the emitted ultrasonic sound wave to return. Below is a snippet of the code for the conversion into coordinates. The video demo also demonstrates the car taking an initial scan and reading.

To begin, I initialized a 100cm*100cm grid which represents the area in front of the car. The original of the car is initialized in the middle of the horizontal axis (left/right movement), and at the bottom of the vertical axis (forward/backward movement). A for loop generates the readings for each degree over 120 degrees in front of the car. The coordinates are then calculated using trigonometry from the feedback distance of the ultrasonic sensor. Following one scan, the array is updated based on each reading in the for loop.

```
angle_step = 1 # angle step
grid = np.zeros((100,100)) # initialize numpy array (origin is 50,0)

for i in range (-60,61): #calculate all points of objects
 tmp = fc.get_distance_at(i*angle_step)
 x = 49 + np.int(tmp * np.sin(np.radians(i*angle_step)))
 y = np.int(tmp * np.cos(np.radians(i*angle_step)))
 if x > 99: # if outside of the range, just change to the max in the array
  x = 99
 if y > 99:
  y = 99
 grid[x,y] = 1 # if object detected then change to 1

plt.imshow(grid, origin='lower')
plt.show()
```

In Figures 4-6, maps visualizing the numpy array are generated using pyplot. Each yellow dot represents a "1" in the grid or the instance of an obstacle for that 1cm*1cm square. Note that the horizontal movement of the car is represented by the y-axis, and vertical movement is represented by the x-axis. The left of the car is higher on the plot, and the right of the car is towards the bottom of the plot. The car is located at (0,50) if looking at the visual plots.

In Figure 4, the sensor gives readings for when there is nothing in the immediate vicinity of the car. We can see one or two sporadic readings 50cm in front of the car, but this can be accounted for some margin of error. We see most the readings are on the far edges of the map, which indicates that there are no obstacles.

In Figure 5, the sensor gives readings for when there is an immediate object (kettle bell) in front of the car. This can be seen in the map with the line of yellow dots in the 20-30cm area of the horizontal access and 40-70cm in the vertical access.

In Figure 6, the sensor gives readings for when there are two objects in front of the vehicle. One object (dog) is to the left of the car and the other object (kettle bell) is to the right of the car. The map shows both of these objects as lines as in the previous figure, and a gap between the two can also be seen. Figure 7 shows what the actual arrangement looks like.

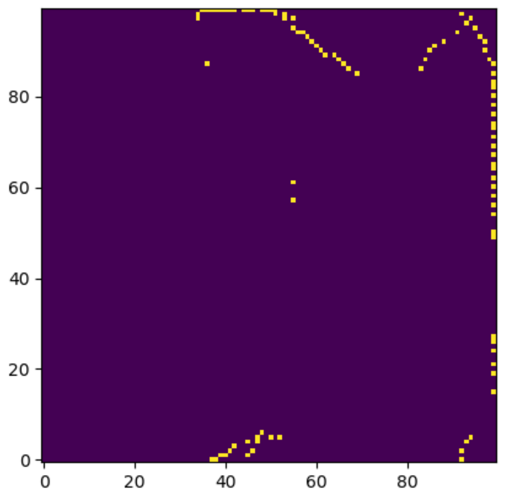


Figure 4: Using 1 reading per degree with no obstacles.

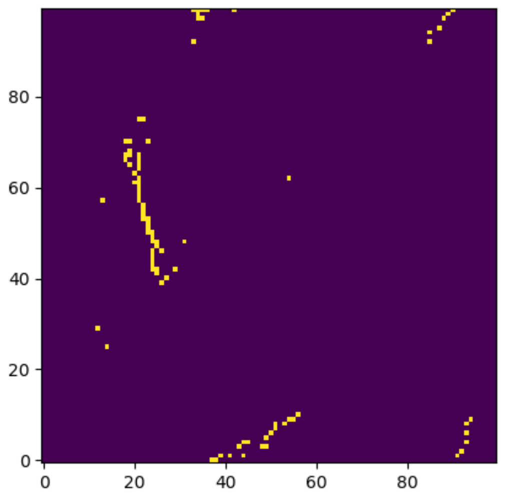


Figure 5: Using 1 reading per degree with 1 obstacle.

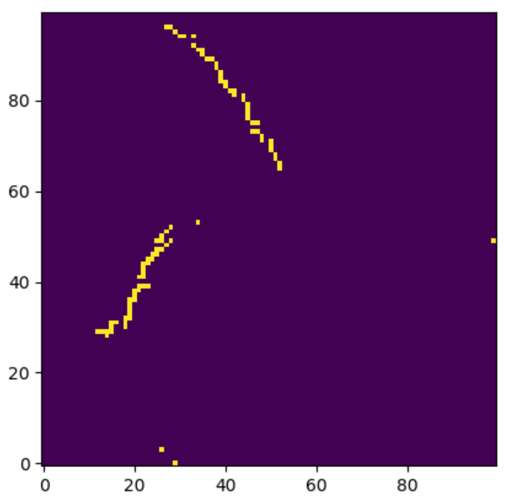


Figure 6: Using 1 reading per degree with 2 obstacles.

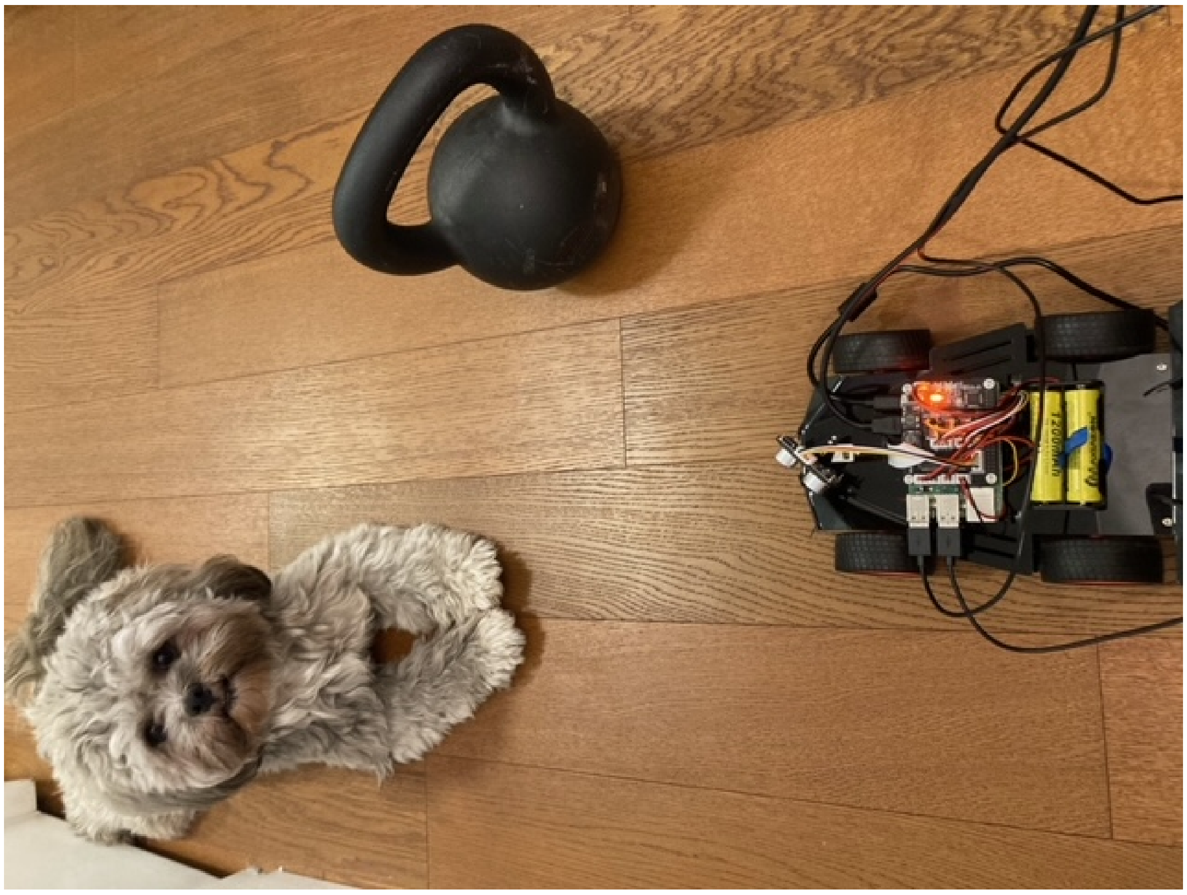


Figure 7: 2 obstacles in front of car.

Because taking a reading every degree was not fast enough, I decided to use a different approach to speed up the mapping process by taking less readings, but using a slope interpretation to fill in the gaps. I eventually decided on a reading every 6 degrees, resulting in 20 total readings. This is compared to 120 different readings from the previous 1 degree iteration, which theoretically should cut down the scanning down to 1/6 of the previous time. Below is a snippet of the code I used to interpolate between the points from each angle reading.

```
if (prev_x and prev_y) and (y - prev_y) != 0: # check if the slope is not 0
      diff = abs(y - prev_y)
      #print(diff)
      slope = (x-prev_x) / (y-prev_y) # calculate the slope
      if slope < 0.5: # check if slope meets the threshold
          if (y > prev_y):
               for j in range (0,diff): # for each point between the two angle readings
                   new_y = y+j
                   new_x = np.int(x+slope*j)
                   if new_x > 99:
                       new_x = 99
                   if new_y > 99:
                       new_y = 99
                   grid[new_x,new_y] = 1
          else:
               for j in range (0,diff): # if horizontal coordinate less than previous ho
                   new_y = prev_y+j
                   new_x = np.int(x+slope*j)
                   if new_x > 99:
                       new_x = 99
                   if new_y > 99:
                       new_y = 99
                   grid[new_x,new_y] = 1
  prev_x = x
  prev_y = y
```

A basic slope finding formula is used simply by dividing the differences of the vertical change by the horizontal change of two points. As discussed earlier, x is actually the vertical change and y is the horizontal change.

Given this rudimentary approach, there is an increase in the potential for error. In order to mitigate some of this error, I've set a threshold for the slope for areas of the grid to be interpolated. The logic behind this is that if the slope is more than a certain amount, it's more likely that the next point feeding is not part of the same object detected but either another object or empty space. Thus, only points that are within a certain slope threshold will be filled in as part of a smoothing process.

Figures 8-10 show the mapping when this new approach is taken. From a first impression comparing with the 1 degree readings, it can be seen that there is a noticeable amount of error. However, for the most part the readings are usable and is able to detect either no obstacles or obstacles within a close vicinity of the car. Noise starts becoming more apparent past the 50cm vertical mark.

2.2 Object Detection

Following basic mapping, the next step was to configure object detection using the Pi Camera. For this task, I used TensorFlow Lite. In Figure 11, a snapshot of the detection algorithm in the car in process can be seen through a computer monitor. The code implementation of the object detection and inference can be seen in section 2.4, when combined with routing for a full test. The only difference between the two coding processes was that the image preview and annotation boxes were removed. I decided to use COCO labels for the object annotation and inference ability as I wanted primarily to detect stop signs which the label set provided. To classify an image, the inference function provides an output and a score. For an object to be annotated, the object has to meet a certain threshold for

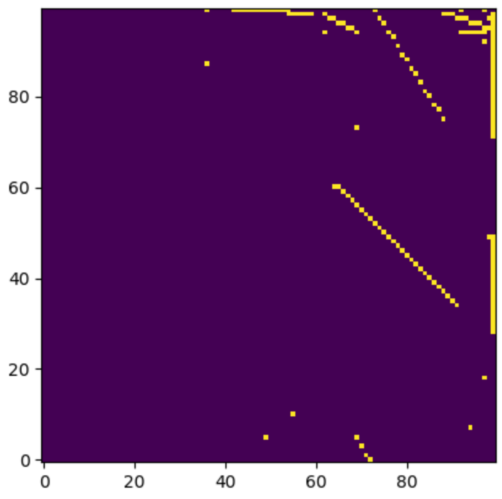


Figure 8: Using 1 reading per 6 degrees with no obstacles.

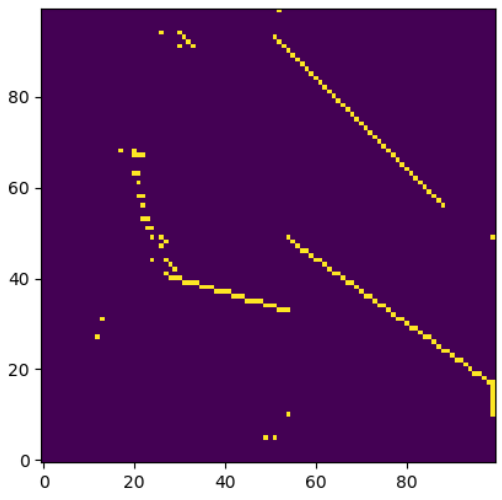


Figure 9: Using 1 reading per 6 degrees with 1 obstacle.

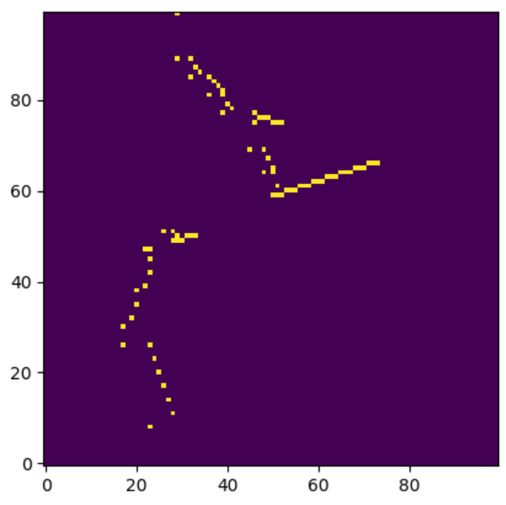


Figure 10: Using 1 reading per 6 degrees with 2 obstacles.

score - the default setting is 0.5.

Image Process Considerations

There were some additional considerations in the implementation of image processing. Because of the limitations in the Raspberry Pi computational power, video processing capabilities are limited. Upon starting the application, it's quickly evident that inference time begins to slow to prevent overheating. Below are three questions that I considered in approaching this issue.

1. Would hardware acceleration help in image processing? Have the packages mentioned above leveraged it? If not, how could you properly leverage hardware acceleration?

The first consideration is additional hardware which the TensorFlow Lite package actually does mention. Specifically, the package mentions the Coral USB Accelerator which adds the Edge TPU ML accelerator to the Raspberry Pi. I did not choose to go this route, as I did not want to add more hardware to my car.

2. Would multi-threading help increase (or hurt) the performance of your program?

Multi-threading may help the performance in that the inferences from Tensorflow Lite could be delayed briefly against the live video feed which would give more processing power to the inference speed (although delayed from real-time). Technically the performance of both processes would be improved, but the usefulness of this improvement would be questionable. I don't think this would make sense in my case given the necessity of speed of detection for a moving car.

3. How would you choose the trade-off between frame rate and detection accuracy?

I think for the purpose of this project, slowing the frame rate makes sense given that the speed of the car is relatively low. Ultimately, I think there would be enough time before approaching an object for the car the react. In my code, I used a continuous feed for each scan with the ultrasonic sensor (which can be seen in part 2.3/2.4). I set the frame rate to the default 30 FPS. However, if the car were to approach speeds that of a real car, then there may be some issues in this approach. In the case of faster car, I think frame rate would become more important. A solution would be to simplify the object detection so that the car is biased towards stopped when any type of object is detected, it

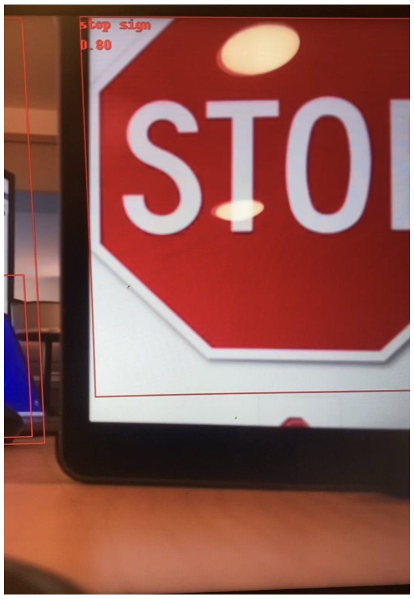


Figure 11: Detection of a Stop Sign in interface.

makes the necessary adjustments so that it can better classify what it is.

2.3 Routing

Using the map generated earlier, the car can now be programmed for routing. To begin, the routing code starts with the generation of the obstacle map. Then, I set a distance I want the car to travel. This process required a bit of calibration through trial and error, but my distance ended up being closer to being measured in inches. The primary logic behind the routing process of my car is for the car to stop maneuvering once it has reached a certain forward distance through the use of a while loop. Sections of the routing code can be seen below.

```
def test():
 total_forward = 0
 distance = 20
 while total_forward < distance:
        ...
        ...
        if np.all(grid[40:60,0:10]==0):
        # detect if something is in front of the car, if not then move forward
         #print(grid[40:60,0:10])
         fc.forward(100)
         x = 0
         fc.time.sleep(0.25)
         speed = speed4()
         x += speed * 0.1
         print(x)
         speed4.deinit()
         total_forward = x + total_forward # add to the total distance
         fc.stop()
        else:
         fc.backward(50)
         fc.time.sleep(0.25)
         fc.stop()
        if np.all(grid[55:80,0:10]==0): # detect if something to the left for the car
            print('left')
            fc.turn_left(100) # turn to the left
            fc.time.sleep(3)
            fc.stop()
        else:
            print('right')
            fc.turn_right(100)
            fc.time.sleep(1)
            fc.stop()
```

In the default code, I have set the distance to "20". This can be changed to whatever the user wants. While the total distance traveled by the car is less than the distance set, the car will iterate through the scanning process. The scanning process is the same used as per in the mapping process. Once the scanning process is complete, if the car does not detect anything directly in front of it, then it will move forward. Otherwise, if the car does not detect anything to the left of it, it will turn to the left, otherwise it will turn to the right. Figure 12 shows what this looks like printed out through each iteration through an obstacle course. As can be seen, the car records the forward distances it moves, and then adds to the total forward distance working its way to the distance that is set for routing. Also printed out is whenever the car turns left or to the right. Figure 13 shows what this obstacle course looked like.

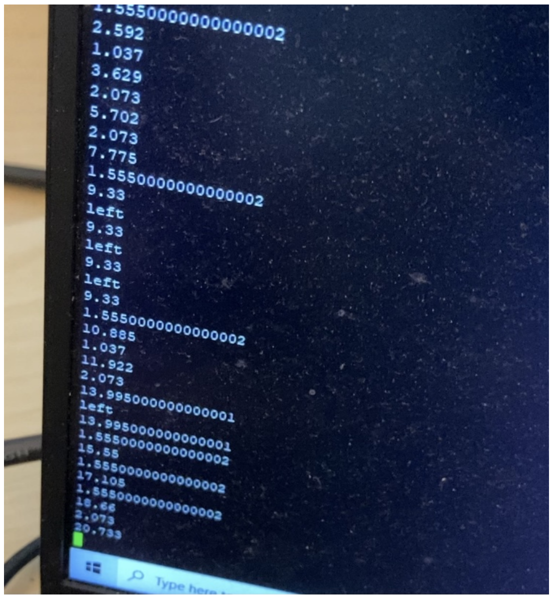
2.4 Testing

Finally, the last objective of the lab is to combine the above into one test - specifically, the routing which includes object detection and navigation with the image detection. To create the full test, the routing code and object detection code were merged together. Below I have included the snippets new parts of the code added to the routing code.

```
labels = load_labels('/home/pi/picar-4wd/examples/coco/coco_labels.txt')
 interpreter = Interpreter('/home/pi/picar-4wd/examples/coco/detect.tflite')
 interpreter.allocate_tensors()
 _, input_height, input_width, _ = interpreter.get_input_details()[0]['shape']

 with picamera.PiCamera(
      resolution=(CAMERA_WIDTH, CAMERA_HEIGHT), framerate=30) as camera:
   #camera.start_preview()
   try:
     stream = io.BytesIO()
     distance = 25
     total_forward = 0
     while total_forward < distance:
        for _ in camera.capture_continuous(stream, format='jpeg', use_video_port=True):
         stream.seek(0)
         image = Image.open(stream).convert('RGB').resize(
           (input_width, input_height), Image.ANTIALIAS)
         results = detect_objects(interpreter, image, threshold)

         ...
         ...
        for obj in results:
           if (labels[obj['class_id']] == 'stop_sign'):
            print(labels[obj['class_id']])
            fc.stop()
            fc.time.sleep(4)
         stream.seek(0)
         stream.truncate()
```



```
1.5550000000000002
2.592
1.037
3.629
2.073
5.702
2.073
7.775
1.5550000000000002
9.33
left
9.33
left
9.33
left
9.33
1.5550000000000002
10.885
1.037
11.922
2.073
13.995000000000001
left
13.995000000000001
1.5550000000000002
15.55
1.5550000000000002
17.105
1.5550000000000002
18.66
2.073
20.733
```

Figure 12: Step by Step and Distance Reading for Simple Course.

The code begins with loading the labels that will be used, which as discussed earlier, I used the COCO set. The camera is loaded with the default parameters, and the routing logic for distance traveled is the same as in the previous examples. Before the mapping is done, the camera captures a continuous stream at 30 FPS detecting objects. Following an iteration of mapping, the code revisits the array formed by the interpreter and checks if a 'stop sign' is one of them. If it meets the score threshold set by the user, then it will be pushed to the list of objects detected. The score threshold is set within the code which is not shown in the above snippet. If there is a stop sign detected, then the vehicle will stop for 4 seconds. The stream from the camera resets and the process starts over. A full walk through of this process can be seen in the video demo too.

Figure 14 shows the displayed readings from the full obstacle course of the car. The total distance set was longer than the only routing example from earlier. As can be seen, the car recognizes the stop sign and prints it while navigating as done in routing with left and right turns in addition to the forward distance tracking also printing. Figure 15 shows the snapshot of the car navigating.

2.5 Improvements

Through the implementation process, I came across several challenges, some of which still have not been properly dealt with.

Calibration of Distance

The first challenge I encountered was trying to calibrate the distance parameters which is a variable based on the motor power and the floor the car is running on. The default setting seemed to measure in cm, but I ended up tweaking and finding that the measurement of "100" on the forward motor relative to "25" on speed ended up each unit being measured in inches. The floor I did all of my testing was a type of hardwood. It is important to note that this will likely have to be tweaked if any other type of floor is used; the unit can also be tweaked to resemble cm as well.

Measuring Lateral Distances

The next step in improving routing capabilities is to add the ability to measure lateral distance. Currently with my implementation it is not possible to know just how much the car moves to the left or the right after turning. A potential approach would be to measure the angle of turn given how long the car

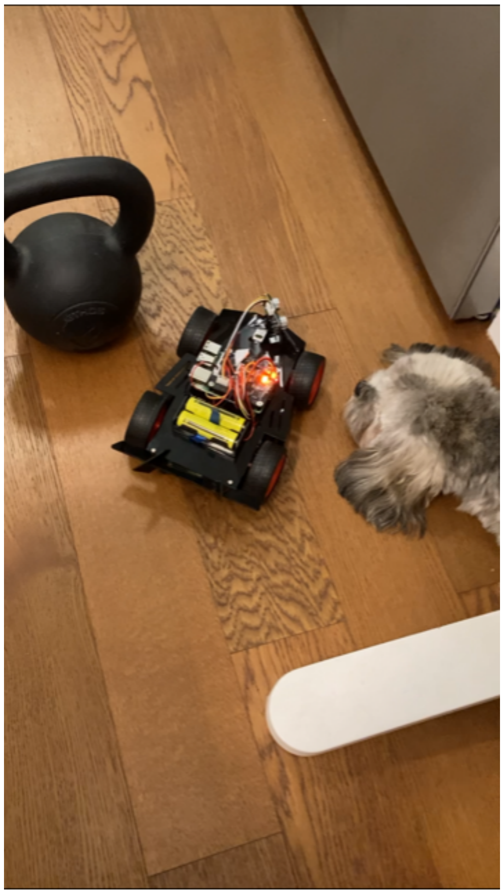
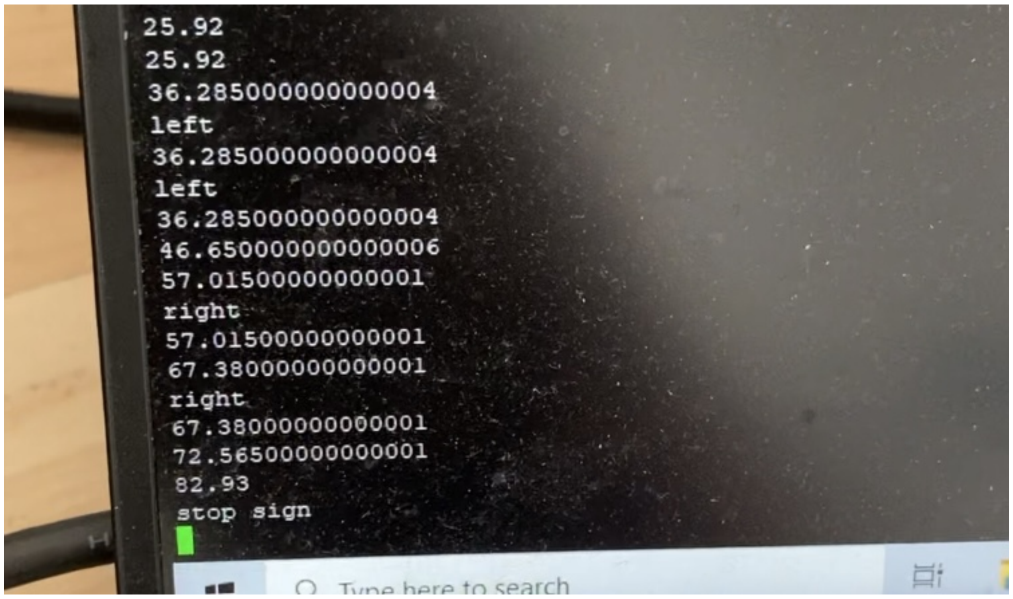


Figure 13: Routing the Simple Obstacle Course.



```
25.92
25.92
36.285000000000004
left
36.285000000000004
left
36.285000000000004
46.650000000000006
57.01500000000001
right
57.01500000000001
67.38000000000001
right
67.38000000000001
72.56500000000001
82.93
stop sign
```

Figure 14: Step by Step and Distance Reading for Full Course.

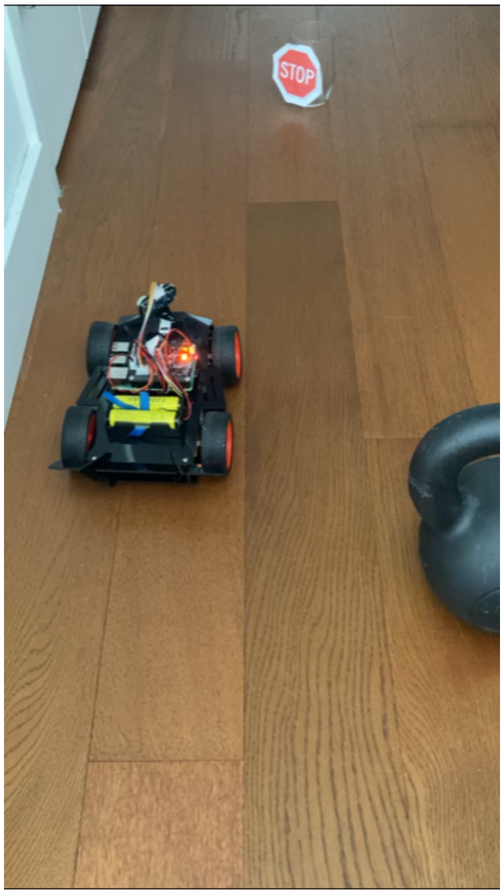


Figure 15: Routing the Full Obstacle Course.

is given to turn, similar to the trial and error process of distance tracking. Once the car turns a certain amount, a tracker can be used for that direction. The orientation of the car will also have to be tracked too. However, like the forward tracking, this would require tweaking any time the car changes surfaces.

Object Detection

With regards to image classification, an issue I realized came up often was that sometimes the camera would recognize an object from a further distance than I would have liked or would recognize an object twice as it moves even closer. In the case of the stop sign, the car ends up stopping twice when this happens which is not the intention. To prevent this from happening too often, I ended up setting the threshold for detection higher. A further improvement would be to set a distance parameter for when the vehicle detects the stop sign and to only stop when that distance parameter is met. This would require further tweaking with the mapping process.

2.6 Conclusion

Through the process of the lab, I was able to gain a basic understanding of the IoT devices and autonomous driving processes while implementing these myself in the car. At this point in time, the car I have created has basic object avoidance, mapping/navigation, and image recognition capabilities. There is the potential for further improvements to be made.