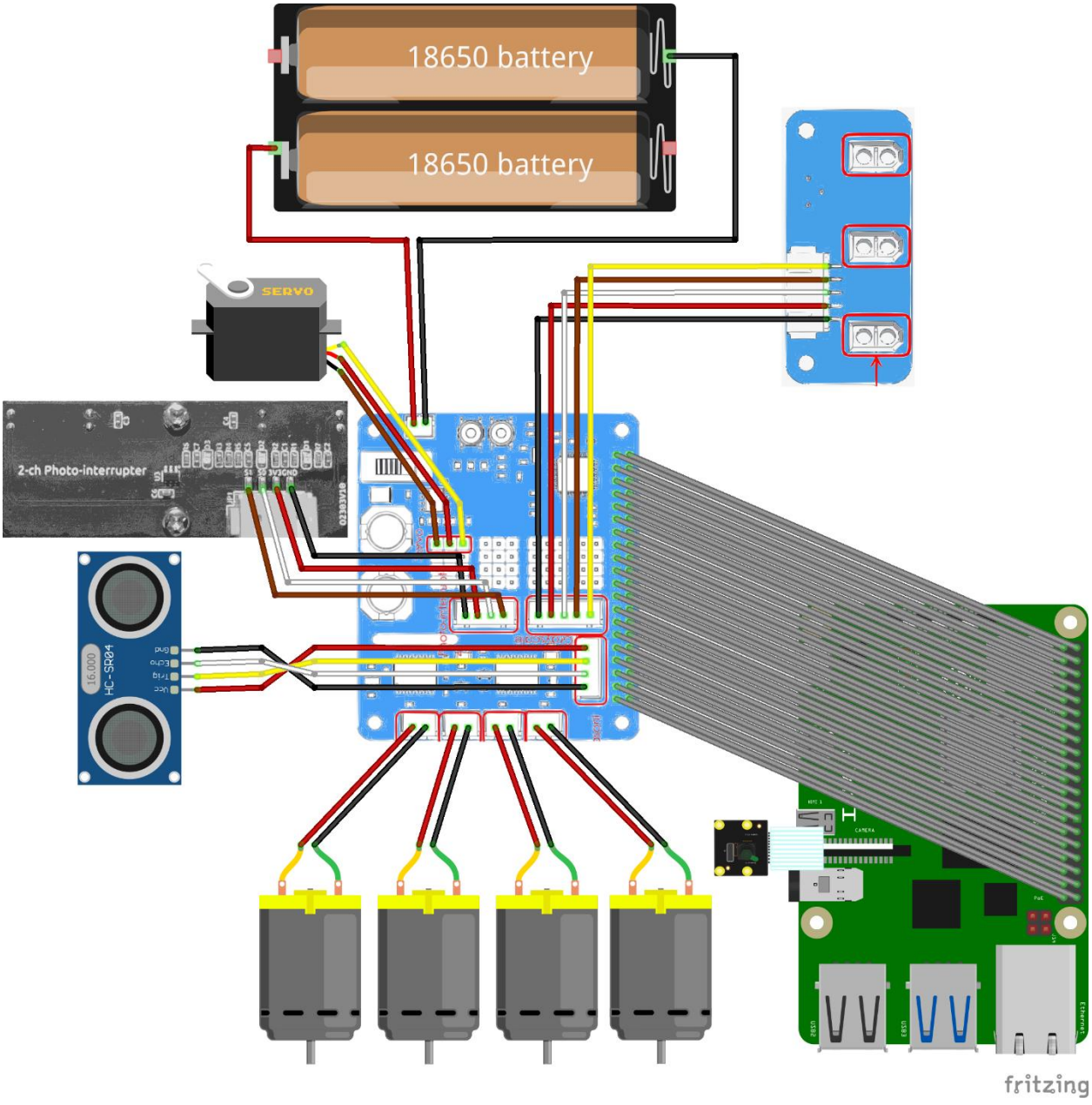
Project Overview

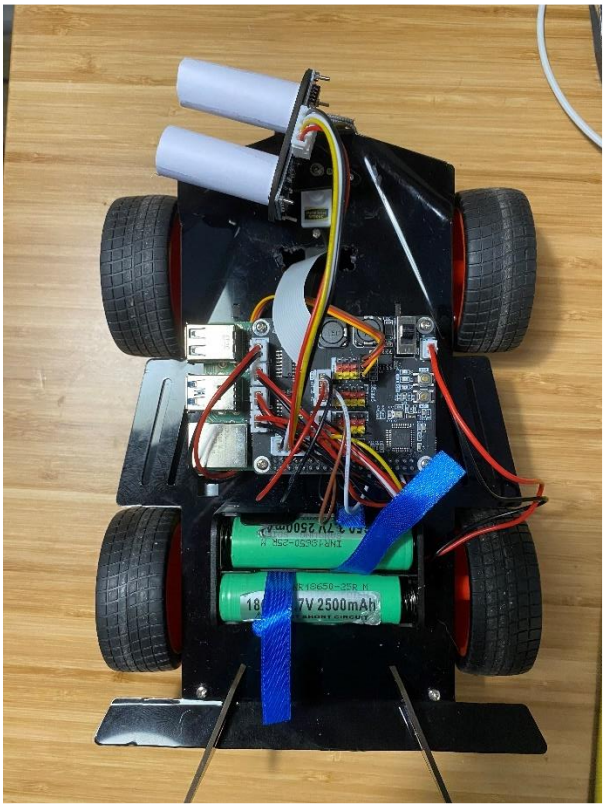
Greg Harrison

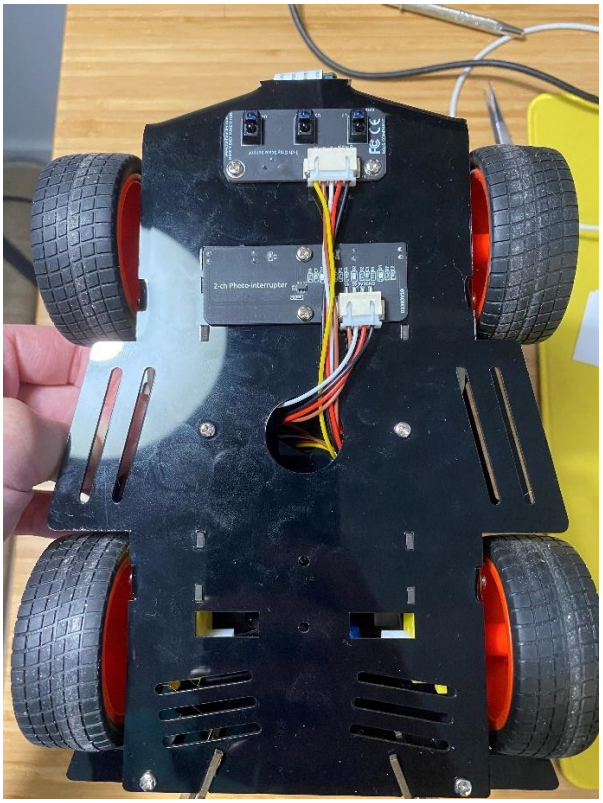
Assembly:

The car is wired per the diagram below.



The following photos show how this was assembled onto the car.





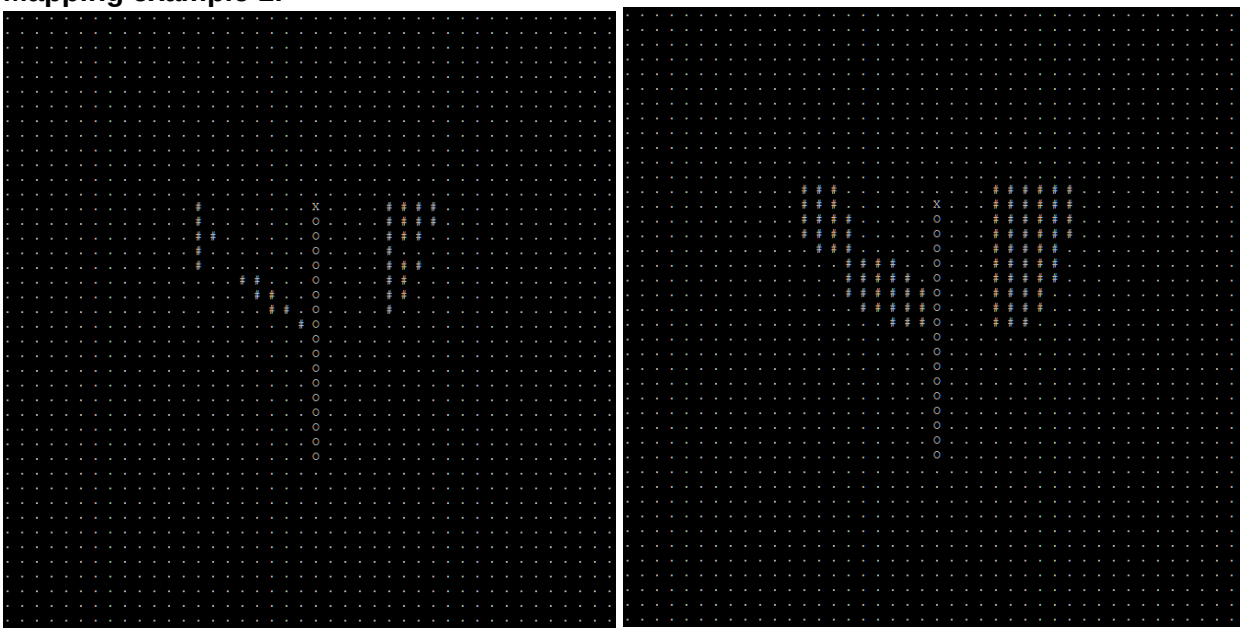
Design Considerations – Mapping:

The ultrasonic sensor was used to make a map of the car's surroundings. This was done by taking the distance measurement from the sensor and the angle of the sensor to find a coordinate on a grid, where each cell of the grid represents a 5-cm x 5-cm square area. I found that the ultrasonic sensor was unreliable at measuring objects at distances beyond about 50-cm, therefore for every scan, the sensor would only look for obstacles that were within a 50-cm radius of the sensor. To speed up the scanning process, the ultrasonic sensor would take a measurement every 10-degrees. If two subsequent measurements returned a distance less than 50-cm, then it was assumed that those measurements were taken on the same obstacle and obstacle markers would be placed along a line between those two points.

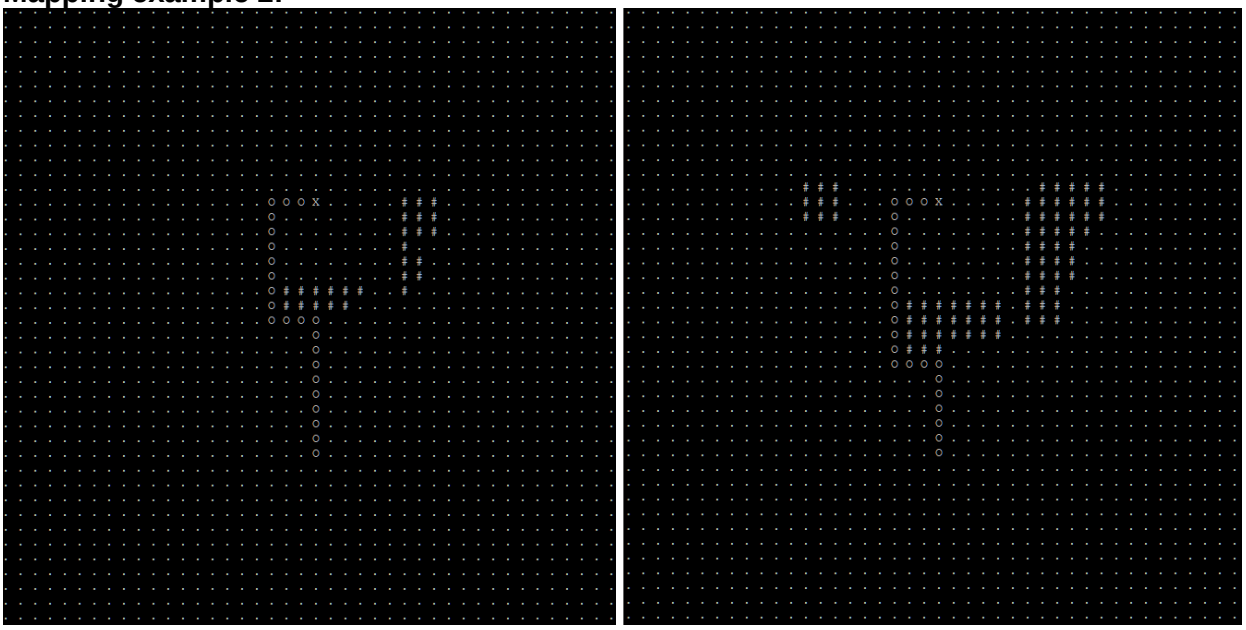
The A* algorithm was used to create a path around obstacles. The implementation used was from https://www.redblobgames.com/pathfinding/grids/graphs.html. I found that this tended to create paths around obstacles that hugged the obstacle. Since the ultrasonic sensor is mounted in middle of the car, this meant that the algorithm would track a path that would cause the edges of the car to bump into the obstacle. To prevent this from happening, a 5-cm boundary was created around every obstacle. This meant that the path from the A* algorithm would hug this obstacle boundary instead of the physical obstacle. This boundary was created on all obstacles except for ones that were measured within a 10-cm radius of the car. This was done to prevent issues that would arise if the car took a measurement while it was within or very close to the 5-cm obstacle boundary.

The images below show examples of the mapping. Obstacles are represented as "#", The location that the ultrasonic sensor took measurements from is represented by "X", and the path from the A* algorithm is represented by "O". Example 1 shows an unobstructed path from start to destination. Example 2 shows an obstructed path and demonstrates how the A* algorithm creates a path around obstacles. In both examples, the image on the left shows the scan without the 5-cm boundary, and the image on the right shows the map after a 5-cm boundary is created around each obstacle.

Mapping example 1:



Mapping example 2:

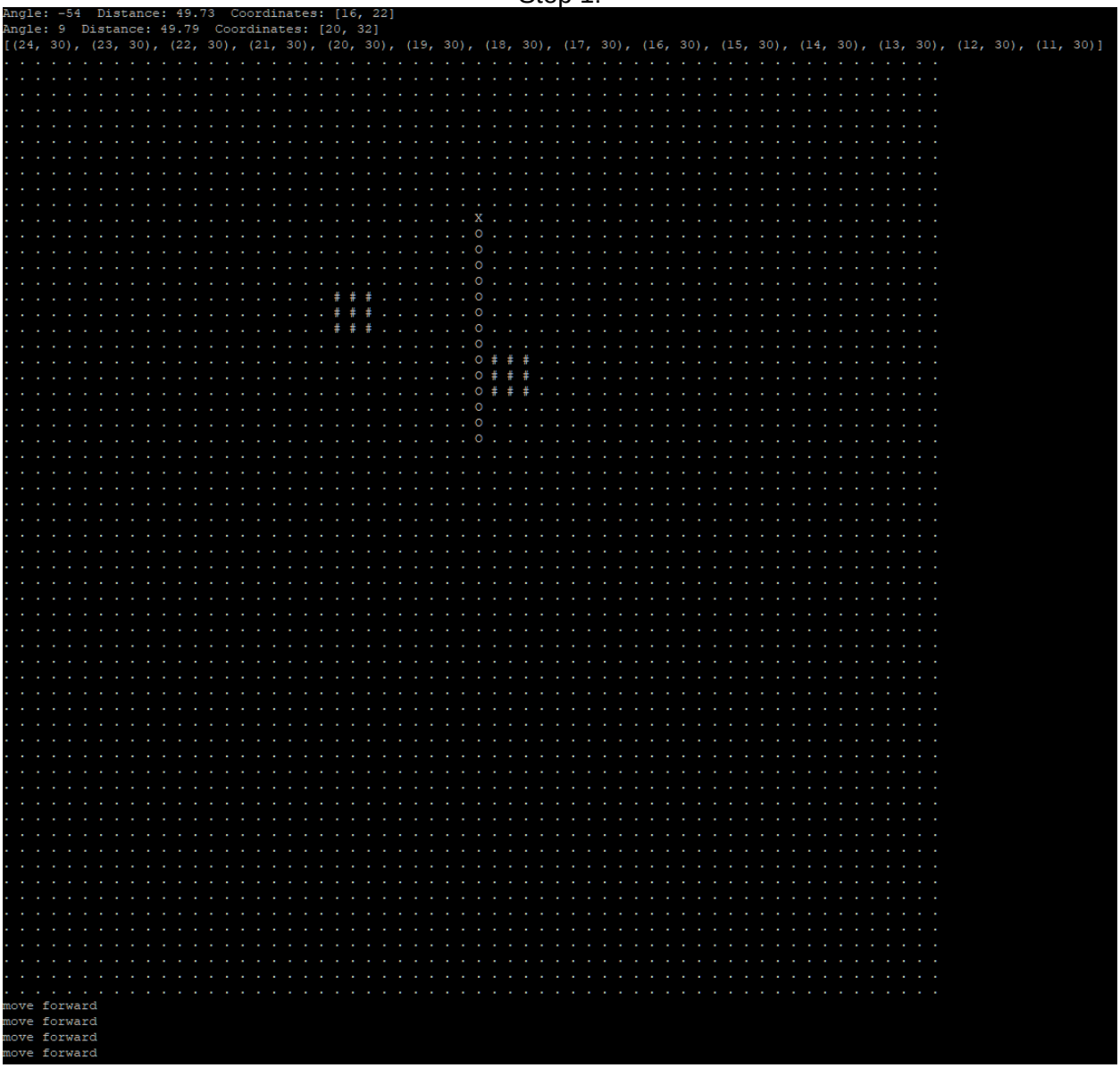


You will notice that the boundaries are generally square. This is because a box was created around each obstacle to create the boundary. I experimented with using a circle around each obstacle, but this made the paths more stairstep-like when navigating around obstacles. Due to some of the underlying hardware limitations, for example, the car not being able to consistently make perfect 90-degree turns, this stairstep motion tended to make the car move off course. Making the boundaries square, reduced the number of stairstep-like paths and allowed the car to more reliable follow the path to its destination.

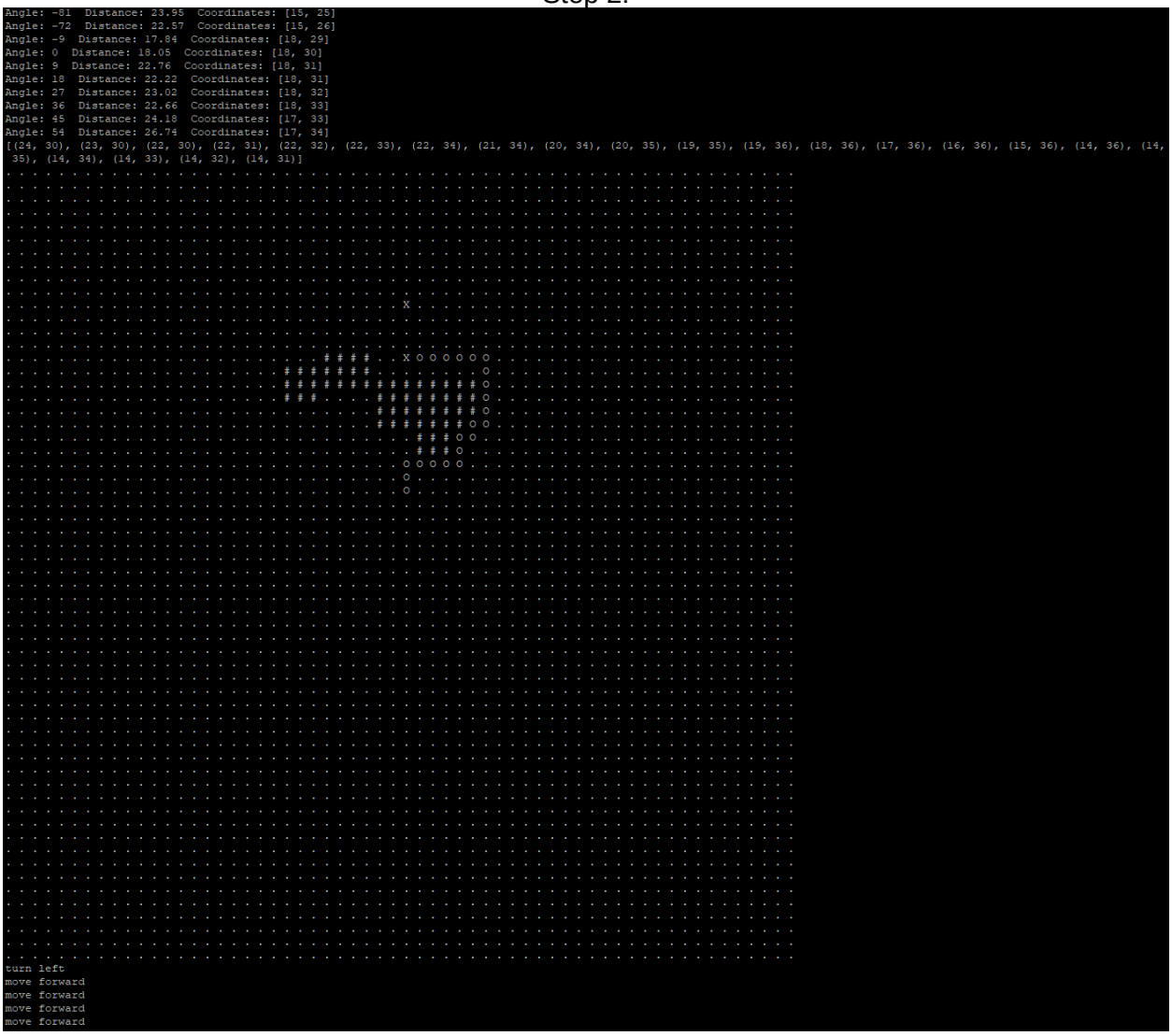
To get the car to its destination, functions were made to move the car forward, turn right 90-degrees, and turn left 90-degrees. The car would follow the path from the starting location, to a predetermined destination, taking scans every three steps. A step was defined as either a left 90-degree turn and a 5-cm movement forward, a right 90-degree turn and a 5-cm movement forward, two 90-degree turns (if the car needs to turn around) and a 5-cm movement forward, or a single 5-cm movement forward if the path was straight. After every scan, the map would update with any obstacle detected and a new path would be created to get the car from the current location to the destination.

The following series of images show what is printed to the console as the car moves to its destination. When looking at the map, notice that the car is moving from the top to the bottom. Each “X” is a location that of the car when it makes a scan. The first thing that is printed is the angle that the ultrasonic sensor is taking a measurement from, the distance in centimeters to an obstacle returned from the ultrasonic sensor, and the location in map coordinates where that obstacle is located. This is printed for every obstacle detected. The next thing that is printed is the path from the car’s destination to the car’s current position; this is the location of the “O”s on the map. The map of the environment is then printed, followed by the instructions to the car. The instructions for step 1 are move forward 4 times, and instructions for step 2 are turn left and move forward 4 times. After each set of instructions is carried out by the car, the environment is scanned again, and new instructions are given.

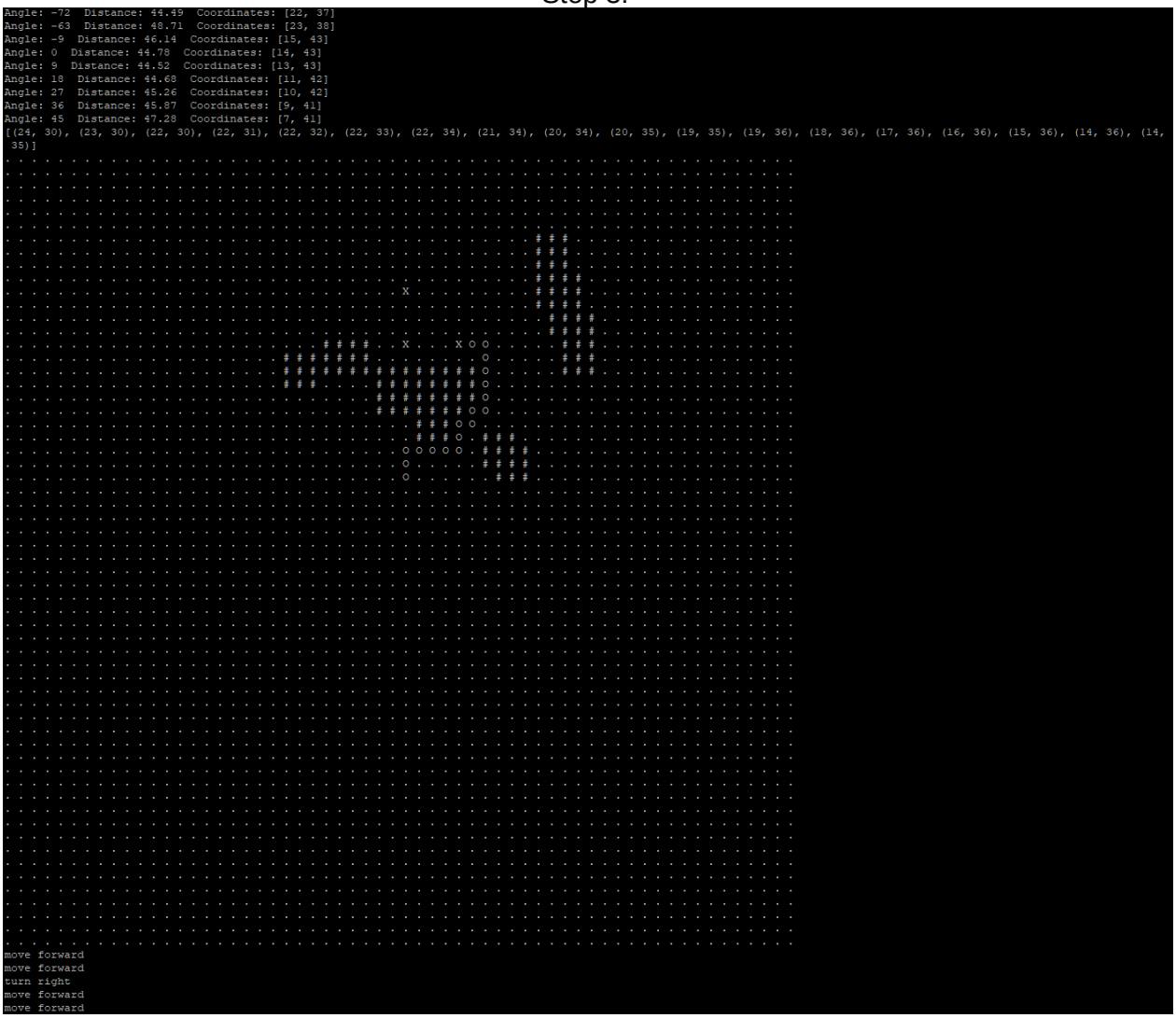
Step 1:


Angle: -54 Distance: 49.73 Coordinates: [16, 22]
Angle: 9 Distance: 49.79 Coordinates: [20, 32]
[(24, 30), (23, 30), (22, 30), (21, 30), (20, 30), (19, 30), (18, 30), (17, 30), (16, 30), (15, 30), (14, 30), (13, 30), (12, 30), (11, 30)]
move forward
move forward
move forward
move forward

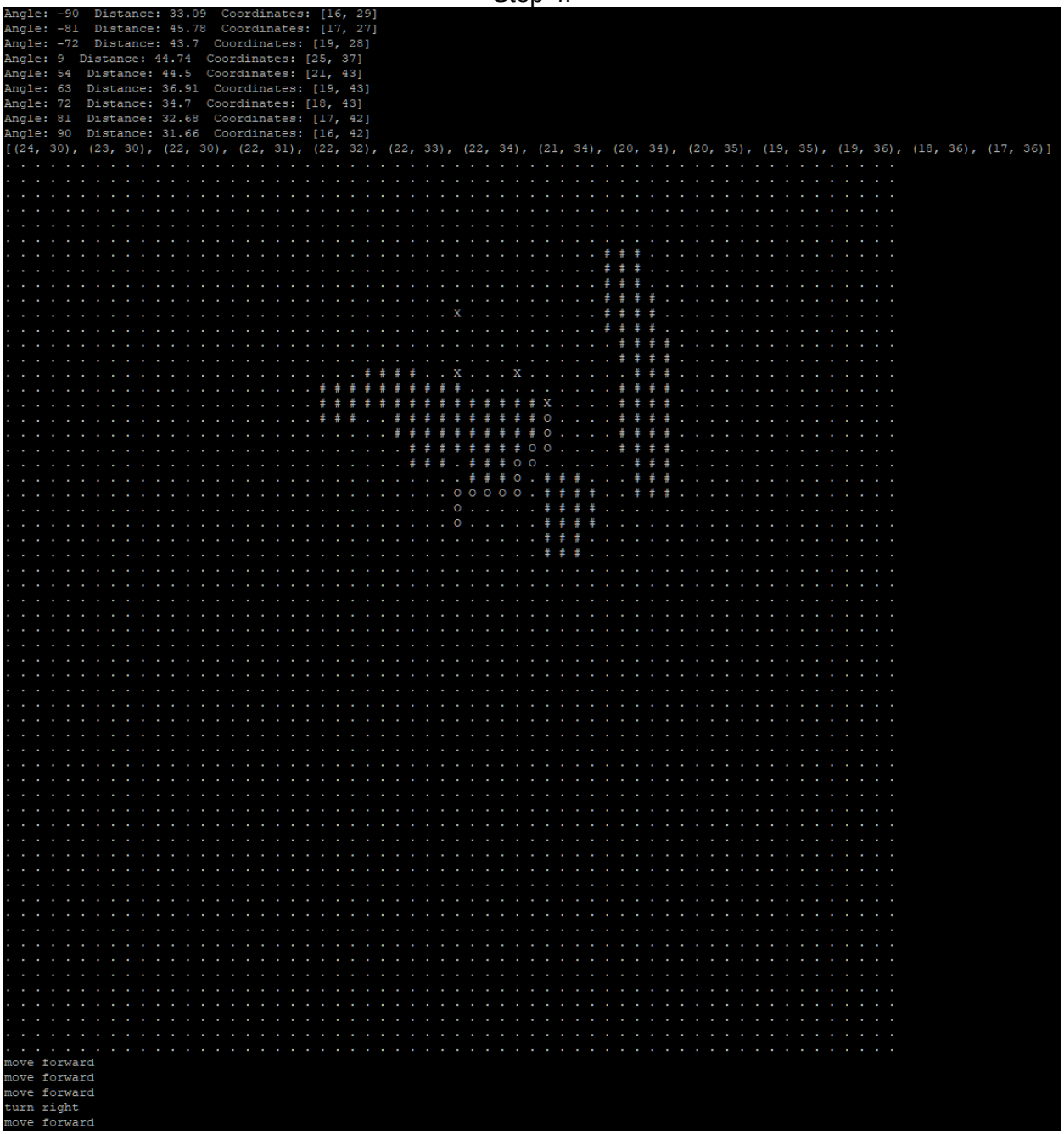
Step 2:


Angle: -81 Distance: 23.95 Coordinates: [15, 25]
Angle: -72 Distance: 22.57 Coordinates: [15, 26]
Angle: -9 Distance: 17.84 Coordinates: [18, 29]
Angle: 0 Distance: 18.05 Coordinates: [18, 30]
Angle: 9 Distance: 22.76 Coordinates: [18, 31]
Angle: 18 Distance: 22.22 Coordinates: [18, 31]
Angle: 27 Distance: 23.02 Coordinates: [18, 32]
Angle: 36 Distance: 22.66 Coordinates: [18, 33]
Angle: 45 Distance: 24.18 Coordinates: [17, 33]
Angle: 54 Distance: 26.74 Coordinates: [17, 34]
[(24, 30), (23, 30), (22, 30), (22, 31), (22, 32), (22, 33), (22, 34), (21, 34), (20, 34), (20, 35), (19, 35), (19, 36), (18, 36), (17, 36), (16, 36), (15, 36), (14, 36), (14, 35), (14, 34), (14, 33), (14, 32), (14, 31)]
turn left
move forward
move forward
move forward
move forward

Step 3:


Angle: -72 Distance: 44.49 Coordinates: [22, 37]
Angle: -63 Distance: 48.71 Coordinates: [23, 38]
Angle: -9 Distance: 46.14 Coordinates: [15, 43]
Angle: 0 Distance: 44.78 Coordinates: [14, 43]
Angle: 9 Distance: 44.52 Coordinates: [13, 43]
Angle: 18 Distance: 44.68 Coordinates: [11, 42]
Angle: 27 Distance: 45.26 Coordinates: [10, 42]
Angle: 36 Distance: 45.87 Coordinates: [9, 41]
Angle: 45 Distance: 47.28 Coordinates: [7, 41]
[(24, 30), (23, 30), (22, 30), (22, 31), (22, 32), (22, 33), (22, 34), (21, 34), (20, 34), (20, 35), (19, 35), (19, 36), (18, 36), (17, 36), (16, 36), (15, 36), (14, 36), (14, 35)]
move forward
move forward
turn right
move forward
move forward

Step 4:


Angle: -90 Distance: 33.09 Coordinates: [16, 29]
Angle: -81 Distance: 45.78 Coordinates: [17, 27]
Angle: -72 Distance: 43.7 Coordinates: [19, 28]
Angle: 9 Distance: 44.74 Coordinates: [25, 37]
Angle: 54 Distance: 44.5 Coordinates: [21, 43]
Angle: 63 Distance: 36.91 Coordinates: [19, 43]
Angle: 72 Distance: 34.7 Coordinates: [18, 43]
Angle: 81 Distance: 32.68 Coordinates: [17, 42]
Angle: 90 Distance: 31.66 Coordinates: [16, 42]
[(24, 30), (23, 30), (22, 30), (22, 31), (22, 32), (22, 33), (22, 34), (21, 34), (20, 34), (20, 35), (19, 35), (19, 36), (18, 36), (17, 36)]
move forward
move forward
move forward
turn right
move forward

Design Considerations – Object Detection:

For this project, I focused on having the car recognize a stop sign and stop when seeing it. To do this, I repurposed the code found here: https://github.com/tensorflow/examples/blob/master/lite/examples/object_detection/raspberry_pi/README.md, which takes frames from the Raspberry Pi camera mounted on the front of the car, and uses TensorFlow Lite to process them with the Coco image recognition model. This code will annotate each frame with the objects that it recognizes and outputs it to a display. I removed all the annotation and display output functionality which increased processing speed by about 3%. I created a queue and had the function push a 1 to the queue when a stop sign was detected in a frame.

To do this while the car was mapping the environment and moving to its destination, the object detection tasks were run on a separate thread, so that while the car was moving it was processing frames and looking for stop signs. At the end of each movement step, the queue was checked. If the queue was not empty, it means that the car saw a stop sign in that movement step. If the car saw a stop sign, it would pause for 5-seconds before rescanning the environment and continuing moving to its destination.

A 4-in x 4-in stop sign was used for this project. The distance in which the object recognition model could reliably recognize the small stop sign was about 40-cm, which is smaller than a movement step; therefore, the car would stop its movement step and pause before passing the stop sign. The only odd behavior that I noticed from implementing the stop sign behavior in this way is that if the movement step in which a stop sign was detected had a turn, the car would turn before stopping. This behavior can be seen in one of the videos associated with this report.

Object Detection Question Responses:

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

Hardware acceleration would help in image processing. The TensorFlow Lite object detection code that was repurposed for this project (https://github.com/tensorflow/examples/blob/master/lite/examples/object_detection/raspberry_pi/README.md) recommends using the Coral USB Accelerator (https://coral.ai/products/accelerator), which is Google's Edge TPU coprocessor. It is an ASIC that can be used to run the image recognition model. The TensorFlow Lite API is compatible with the Coral USB Accelerator so you can easily delegate model execution to it and speed up machine learning inference time.

Question 2: Would multithreading help increase (or hurt) the performance of your program?

Multithreading will improve the short-term performance of the program. For instance, frame rate of image processing could be improved by preprocessing frames in one thread and then performing object detection on those frames in another thread. I suspect that the performance over an extended period will be diminished without adequate CPU cooling, since multithreading will raise temperatures of the Raspberry Pi quicker and cause performance throttling.

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

To assess this, I would have to consider what the design goals are. If the goal is to detect relatively large objects or objects that are moving at the same relative velocity as the vehicle, traffic signs or other cars moving the same direction on the roadway for example, frame rate is not as important, since these objects are more likely to be in the frame for a relatively long period of time. It is however very important that these objects are accurately predicted. It is easy to imagine the negative outcomes of mistaking a stop sign or traffic light for a speed limit sign.

Frame rate becomes exceedingly important for detecting small fast objects that move into the path of the vehicle, like a person or an animal running across the road. The importance of accuracy in these situations is an ethical question. For instance, do we need to be able to know that the object in front of the car is an animal vs. a soccer ball for instance? If so, is it worth reducing the frame rate and slowing the vehicle reaction time, or potentially missing the object entirely, to obtain that accuracy?

I think it is important to answer these questions in the context of the limitations of the specific vehicle performing the object detection. If the breaking distance of the car at 30-mph is 45-ft, and an object appears in front of the car 50-ft away. To not hit the object, the vehicle needs to detect the object and begin breaking in about $1/10^{th}$ of a second.