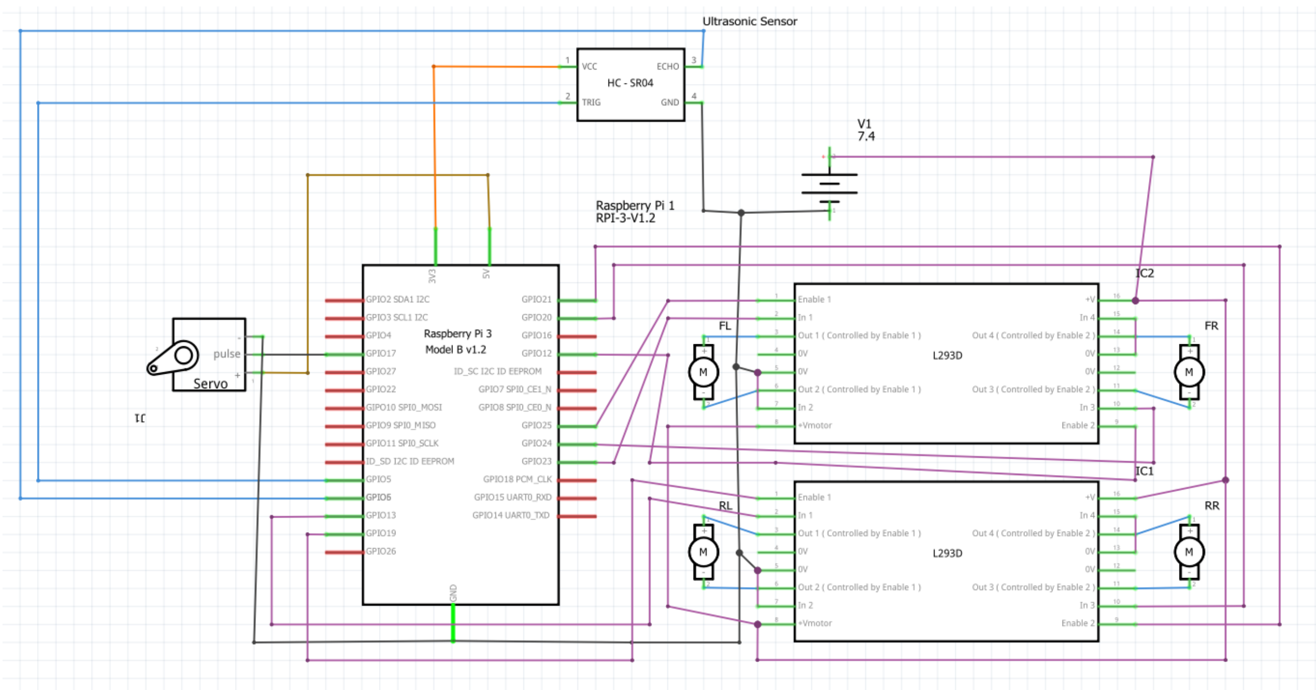
Self-Driving PiCar

Joshua Montoya

For this project, a toy car was programmed to self-drive a route while avoiding obstacles and obeying some simple traffic laws. The SunFounder PiCar hardware was purchased for the body of the car, as well as a Pi camera fitted to the front. SunFounder provides some sample code on their website, and this was used for basic I/O setup with pinouts. All other code was built on top of this foundation.

Topology for the device wiring can be seen below:



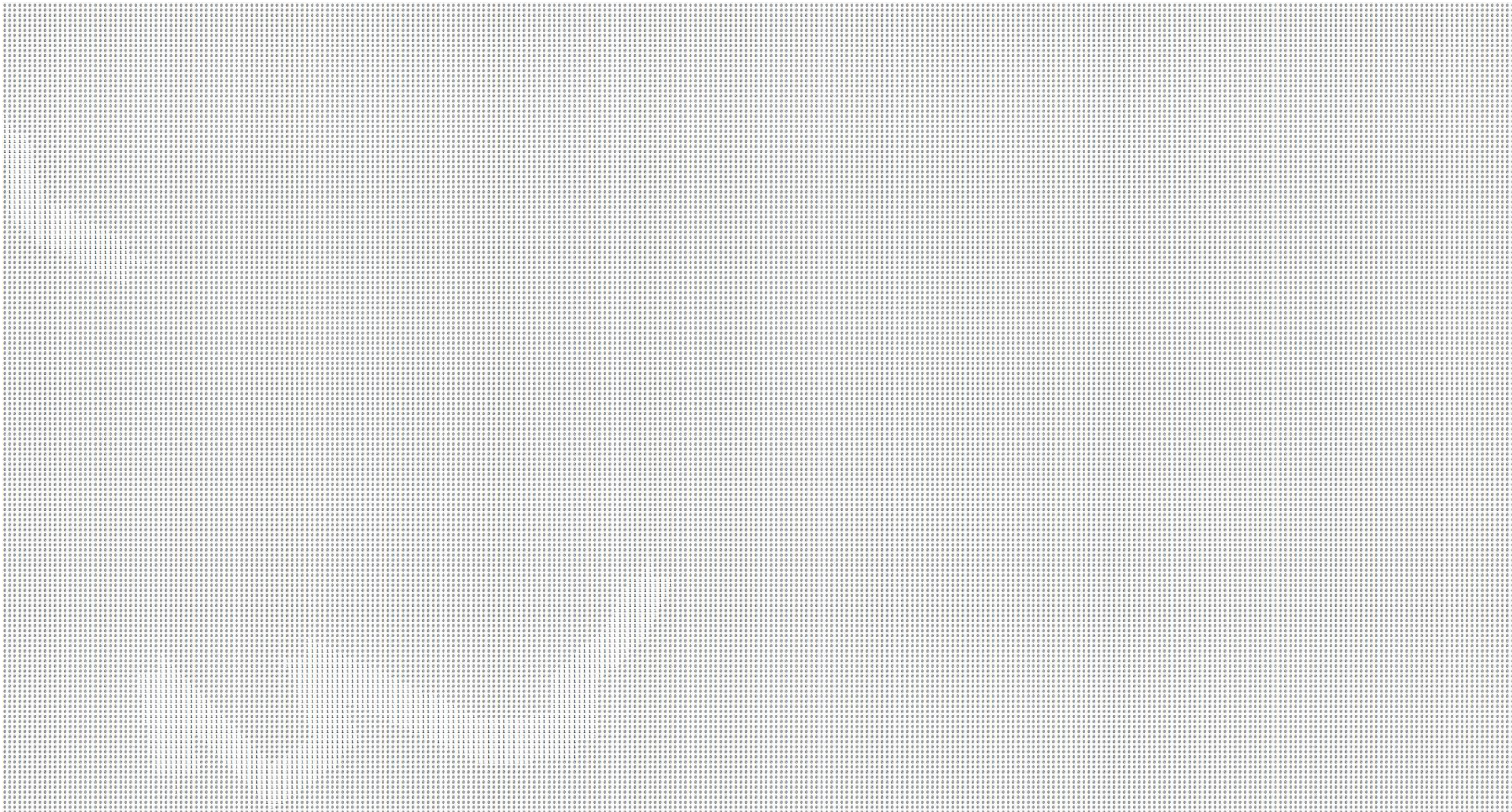
Moving the Car and Detecting Objects

To start, I set up the servo motor controlling the position of the ultrasonic sensor to take readings at regular intervals over a 180-degree range. I found a 0.01 second pause at each position was enough to collect accurate object detection information.

Following this, I wrote the code for moving the car forward and backward, turning it by a given degree, and rerouting it in the event of an obstacle. The results were somewhat inconsistent, limited by the hardware, so precision was not good. This could be improved by attaching a compass to the Pi.

Mapping a Route with Obstacle Avoidance

The data collected from the ultrasonic sensor can be represented as follows:



A line of ones was drawn between two adjacent points where an object was detected by the ultrasonic sensor. Then, a 1 was placed in the map at any position within 4 cells of these lines. You can see these sections at the bottom left and top left of the image. These are areas to be avoided. This map was condensed to allow for ease of implementation. The condensed map was created by breaking the large map into 10x10 sections and placing a 1 if any ones exist within that section, and a 0 if all cells in that section are 0. Code for the A-star algorithm was implemented to draw a route on this condensed map. This condensed map is shown below:

```
0001111001100000000000000000000
0011111111110000000000000000000
0011111110011000000000000000000
0000001000011000000000000000000
0000000000001100000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
vvvvvvvvvvvvvvvvvvvvvvvvv000000
A================<<<<<<<<<00000
^<<<<<<<<<<<<<<<==<<<<<<<<<0000
^11^<<<<<<<<<<<<<==<<<<<<<<<000
111^^<<<<<<<<<<<<<==<<<<<<<<<v0
10>^^^<<<<<<<^<<<<<===========Z
100^^^^^^^^^^^^^^^^^^^^^^^^^^^0
0000000000000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
```

This map is flipped vertically from the large map. “A” represents the start position of the car. “Z” represents the desired destination. “1” represents an object. “0” is empty space. Arrows represent iterations of the A-star algorithm, and “=” represents the current calculated route. The approximate real-world area of the map is 10 feet wide and 20 feet long.

Obeying Traffic Laws

Images were periodically collected from the Pi camera to view the area in front of the car. TensorFlow libraries were used to detect objects. This was done using objects that have been pretrained and are available with the TensorFlow libraries, such as stop signs, traffic lights, traffic cones, and people. Code was written to lay out a procedure when these objects are encountered. Processing each image was the most challenging part of the project. Because of the limited processing power of the Raspberry Pi, running object detection was extremely slow. This resulted in a scenario where the frame rate from the camera needed to be lowered substantially. Occasionally, this would cause the processed frames to miss important events, as the car moved too quickly to capture important objects while they were in the field of view. Increasing the frame rate to compensate overloaded the processor and caused unpredictable behavior. Consequently, some traffic laws were violated, with no clear solution with existing hardware.

One possible option to solve this problem would be hardware acceleration. The TensorFlow documentation mentions that a Coral Edge TPU processor is supported. The processor is optimized for machine learning algorithms and could be plugged into a USB port on the Pi, which would increase the processing speed significantly over the built-in CPU. If needed, this hardware could be purchased for 60 dollars.

Combining Obstacle Avoidance and Route Navigation with Object Detection

Multithreaded code was created to simultaneously run route navigation and obstacle avoidance in one thread (as both of these involve the ultrasonic sensor), and object detection in another thread.

Code Discussion

Main function with threads, for reference:

```python
def main():

    parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument(
        '--model',
        help='Path of the object detection model.',
        required=False,
        default='efficientdet_lite0.tflite')
    parser.add_argument(
        '--cameraId', help='Id of camera.', required=False, type=int, default=0)
    parser.add_argument(
        '--frameWidth',
        help='Width of frame to capture from camera.',
        required=False,
        type=int,
        default=640)
    parser.add_argument(
        '--frameHeight',
        help='Height of frame to capture from camera.',
        required=False,
        type=int,
        default=480)
    parser.add_argument(
        '--numThreads',
        help='Number of CPU threads to run the model.',
        required=False,
        type=int,
        default=4)
    parser.add_argument(
        '--enableEdgeTPU',
        help='Whether to run the model on EdgeTPU.',
        action='store_true',
        required=False,
        default=False)
    args = parser.parse_args()

    thread1 = threading.Thread(target=run_nav)#, args=(False,))
    thread2 = threading.Thread(target=run_detect, args=(args.model, int(args.cameraId), args.frameWidth,
                                args.frameHeight,int(args.numThreads), bool(args.enableEdgeTPU),))
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    stop()
    servo.set_angle(0)

if __name__ == "__main__":
    try:
        main()
    finally:
        servo.set_angle(0)
        stop()
```

run_detect, or the code for object detection:

```
def run_detect(model: str, camera_id: int, width: int, height: int, num_threads: int, enable_edgetpu: bool) -> None:

    global stop_sign
    global red_light
    global person

    # Variables to calculate FPS
    counter, fps = 0, 0
    start_time = time.time()

    # Start capturing video input from the camera
    cap = cv2.VideoCapture(camera_id)
    cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)

    # Visualization parameters
    row_size = 20  # pixels
    left_margin = 24  # pixels
    text_color = (0, 0, 255)  # red
    font_size = 1
    font_thickness = 1
    fps_avg_frame_count = 10

    # These assist in detecting the color of traffic light
    traffic_light_column = 0 # pixel column value of center of traffic light
    red_light_row = 0 # pixel row value of center of red light
    yellow_light_row = 0 # pixel row value of center of yellow light
    green_light_row = 0 # pixel row value of center of green light
    red_rgb = [0,0,0] # RGB value at center of red light
    yellow_rgb = [0,0,0] # RGB value at center of yellow light
    green_rgb = [0,0,0] # RGB value at center of green light
    red_intensity = 0 # light intensity of red light
    yellow_intensity = 0 # light intensity of yellow light
    green_intensity = 0 # light intensity of green light

    detected_object_size = 0 # used to estimate how close object is to car
    stop_counter = 0 # used to allow car to pass stop sign before beginning to look for one again

    # Initialize the object detection model
    options = ObjectDetectorOptions(
        num_threads=num_threads,
        score_threshold=0.3,
        max_results=3,
        enable_edgetpu=enable_edgetpu)
    detector = ObjectDetector(model_path=model, options=options)

    # Continuously capture images from the camera and run inference
    while cap.isOpened():
        success, image = cap.read()
        if not success:
            sys.exit('ERROR: Unable to read from webcam. Please verify your webcam settings.')

        counter += 1
        image = cv2.flip(image, 1)

        # Run object detection estimation using the model.
        detections = detector.detect(image)

        # Draw keypoints and edges on input image
        image = visualize(image, detections)
        if stop_counter > 0: #stop_counter give time to get past stop sign
            stop_counter -= 1
            print("Stop counter:")
            print(stop_counter)

```

```
        for detection in detections:
            for category in detection.categories:
                detected_object_size = detection.bounding_box.bottom - detection.bounding_box.top # Uses bounding box to detemine object size
                if category.label == "person" and detected_object_size > 30:
                    person = True
                else:
                    person = False
                if category.label == "stop sign" and detected_object_size > 10:
                    if stop_counter == 0:
                        stop_counter = 25 # begin stop_counter to allow car to pass stop sign before looking for a stop sign again
                        stop_sign = True
                if category.label == "traffic light":

                    #use bounding box to estimate individual light locations
                    traffic_light_column = round((detection.bounding_box.left + detection.bounding_box.right)/2)
                    red_light_row = round((detection.bounding_box.bottom - detection.bounding_box.top)/5 + detection.bounding_box.top)
                    yellow_light_row = round((detection.bounding_box.bottom - detection.bounding_box.top)/2 + detection.bounding_box.top)
                    green_light_row = round((detection.bounding_box.bottom - detection.bounding_box.top)*4/5 + detection.bounding_box.top)

                    #use locations to determine rbg at those points in image
                    red_rgb = image[red_light_row, traffic_light_column]
                    yellow_rgb = image[yellow_light_row, traffic_light_column]
                    green_rgb = image[green_light_row, traffic_light_column]

                    #use rgb to calculate overall intensity of pixel
                    red_intensity = (red_rgb[0]+red_rgb[1]+red_rgb[2])/3
                    yellow_intensity = (yellow_rgb[0]+yellow_rgb[1]+yellow_rgb[2])/3
                    green_intensity = (green_rgb[0]+green_rgb[1]+green_rgb[2])/3

                    if red_intensity > yellow_intensity and red_intensity > green_intensity and detected_object_size > 10:
                        red_light = True
                    elif yellow_intensity > red_intensity and yellow_intensity > green_intensity and detected_object_size > 10:
                        red_light = True
                    else:
                        red_light = False

        # Calculate the FPS
        if counter % fps_avg_frame_count == 0:
            end_time = time.time()
            fps = fps_avg_frame_count / (end_time - start_time)
            start_time = time.time()

        # Show the FPS
        fps_text = 'FPS = {:.1f}'.format(fps)
        text_location = (left_margin, row_size)
        cv2.putText(image, fps_text, text_location, cv2.FONT_HERSHEY_PLAIN,
                font_size, text_color, font_thickness)

        # Stop the program if the ESC key is pressed.
        if cv2.waitKey(1) == 27:
            break
        cv2.imshow('object_detector', image)

    cap.release()
    cv2.destroyAllWindows()
```

run_nav, or the code for object detection and route navigation:

```python
def run_nav():
    angle = 90 #angle at which we will start
    servo.set_angle(angle)
    opp = MAPPING_ARRAY_WIDTH + 100 #opposite edge calculated from distance (hypotenuse) and stored
    adj = MAPPING_ARRAY_LENGTH + 100 #adjacent edge calculated from distance (hypotenuse) and stored
    time.sleep(1)
    step = 5 #step between angles from which an ultrasonic distance reading will be taken
    direction = -1 #determines direction of servo motor motion
    object_map = np.zeros((MAPPING_ARRAY_WIDTH + 1,MAPPING_ARRAY_LENGTH + 1))
    line = False #Helps determine if we need to draw a line between two points
    reached_end = False #Will be true when end of path is reached
    make_map = False #Returns true when scan is
    start, goal = (0, 8), (30, 12)
    while reached_end == False:
        # "draw" map at each angle
        current = [start[1]*10,start[0]*10] # converts new starting coordinates from compressed array to full array
        distance = get_distance_at(angle) #returns object distance at current angle
        if distance < 0:
            opp = MAPPING_ARRAY_WIDTH + 100
            adj = MAPPING_ARRAY_LENGTH + 100
            line = False
        else:
            prev_opp = opp
            prev_adj = adj
            angle_radians = math.radians(angle)
            opp = math.sin(angle_radians)*distance #from sine = opposite/hypotenuse
            adj = math.cos(angle_radians)*distance #from cosine = adjacent/hypotenuse
            opp = round(opp + current[0])
            adj = round(adj + current[1])
        if opp <= MAPPING_ARRAY_WIDTH and adj <= MAPPING_ARRAY_LENGTH and opp >= 0 and adj >= 0:
            object_map = write_ones(object_map, opp, adj, False) #put ones at point and all points within a given distance
            if line == True:
                object_map = draw_line(object_map, prev_opp, prev_adj, opp, adj)
            line = True #if object detected for next angle then we will know to draw a line
        else:
            line = False

        if angle < -90: #if max angle is reached change servo direction
            direction = 1
            g = GridWithAdjustedWeights(MAPPING_ARRAY_LENGTH/10 + 1, MAPPING_ARRAY_WIDTH/10 + 1)
            g.walls = make_diagram_walls(object_map, start, goal)
            came_from, cost_so_far = a_star_search(g, start, goal)
            path = reconstruct_path(came_from, start, goal)
            reached_end, start = build_route(path, start)
            object_map = np.zeros((MAPPING_ARRAY_WIDTH + 1,MAPPING_ARRAY_LENGTH + 1))
        if angle > 90:
            direction = -1
            g = GridWithAdjustedWeights(MAPPING_ARRAY_LENGTH/10 + 1, MAPPING_ARRAY_WIDTH/10 + 1)
            g.walls = make_diagram_walls(object_map, start, goal)
            came_from, cost_so_far = a_star_search(g, start, goal)
            path = reconstruct_path(came_from, start, goal)
            reached_end, start = build_route(path, start)
            object_map = np.zeros((MAPPING_ARRAY_WIDTH + 1,MAPPING_ARRAY_LENGTH + 1))
        if direction > 0:
            angle = angle + step #steps servo in positive direction
        if direction < 0:
            angle = angle - step #steps servo in negative direction
```

write_ones:

```
def write_ones(mat, x, y, transpose):
    mat = mat.copy()
    width = MAPPING_ARRAY_WIDTH
    length = MAPPING_ARRAY_LENGTH
    if transpose:
        width = MAPPING_ARRAY_LENGTH
        length = MAPPING_ARRAY_WIDTH

    for a in range(5):
        for b in range(5)
            if x + a < width and y + b < length:
                mat[x + a, y + b] = 1
            if x - a > 0 and y - b > 0
                mat[x - a, y - b] = 1
            if x - a > 0 and y + b < length:
                mat[x - a, y + b] = 1
            if x + a < width and y - b > 0:
                mat[x + a, y - b] = 1
    return mat
```

This function changes, in the object_map, from 0 to 1, the point we take in and all points within 4 spaces of it (for car clearance reasons).

draw_line:

```
def draw_line(mat, x0, y0, x1, y1):
    mat = mat.copy()
    if (x0, y0) == (x1, y1):
        return mat
    mat = write_ones(mat, x1, y1, False)
    # Swap axes if Y slope is smaller than X slope
    transpose = abs(x1 - x0) < abs(y1 - y0)
    if transpose:
        mat = mat.T
        x0, y0, x1, y1 = y0, x0, y1, x1
    # Swap line direction to go left-to-right if necessary
    if x0 > x1:
        x0, y0, x1, y1 = x1, y1, x0, y0
    # Compute intermediate coordinates using line equation
    x = np.arange(x0 + 1, x1)
    y = np.round(((y1 - y0) / (x1 - x0)) * (x - x0) + y0).astype(x.dtype)
    # Write intermediate coordinates
    for a in range(len(x)):
        if transpose:
            mat = write_ones(mat, x[a], y[a], True)
        else:
            mat = write_ones(mat, x[a], y[a], False)
    return mat if not transpose else mat.T
```

This function constructs an array of points between two object points and passes these points into the write_ones function to construct a wall of ones.

make_diagram_walls:

```
def make_diagram_walls (object_map, start, goal):
    walls_array = []
    for x in range(int(MAPPING_ARRAY_WIDTH/10)):
        for y in range(int(MAPPING_ARRAY_LENGTH/10)):
            array_slice = object_map[x*10:x*10+9,y*10:y*10+9]
            exists = 1 in array_slice
            if exists == True and (y,MAPPING_ARRAY_WIDTH/10-x-1) != start and (y,MAPPING_ARRAY_WIDTH/10-x-1) != goal:
                walls_array.append((y,MAPPING_ARRAY_WIDTH/10-x-1))
    return walls_array
```

This function is responsible for creating the array of points that will fill in the “walls” of the graph on which a_star_search is performed. The graph is smaller than the object_map by a factor of 10. The walls array is made by looking at a 10x10 section of the map, and if there are any ones that appear in that section, there will be a one in the point of the graph corresponding to that section. This smaller graph makes for smoother navigation between the start point and the goal and provides plenty of clearance for the car. The only drawback is if there are a lot of objects detected, especially if they are close to the start or end points, a_star_search may not be able to find a path. For this application, that wasn’t a problem.

build_route:

```
def build_route(path, start): # Builds an array of directions, then calls go_route function to move that route
    END_LOOP = 15
    current = start
    step_count = 0
    moves = []
    while step_count < END_LOOP:
        if len(path) <= step_count:
            go_route(moves)
            return True, start
        if path[step_count][0] > current[0]: # check if next leg is forward
            moves.append("forward")
        elif path[step_count][1] > current[1]: # check if next leg is right
            if step_count + 1 >= len(path) or step_count + 1 >= END_LOOP: # check if it's the last leg of the path
                moves.append("right")
            elif path[step_count+1][0] > path[step_count][0]: # check if leg after next is forward, if so we have a diagonal
                moves.append("diagonal_right")
                step_count += 1
            else:
                moves.append("right")
        elif path[step_count][1] < current[1]: # check if next leg is left
            if step_count + 1 >= len(path) or step_count + 1 >= END_LOOP: # check if it's the last leg of the path
                moves.append("left")
            elif path[step_count+1][0] > path[step_count][0]: # check if leg after next is forward, if so we have a diagonal
                moves.append("diagonal_left")
                step_count += 1
            else:
                moves.append("left")
        current = path[step_count]
        step_count += 1
    go_route(moves)
    if len(path) <= END_LOOP: #
        return True, start
    else:
        return False, path[END_LOOP-1]
```

The build_route function builds an array of instructions based on the path generated from A*. The array contains a maximum of 15 instructions before calling the go_route function, then returning for another sweep with the ultrasonic sensor from the new location. The code also returns True when the goal is reached. This code is meant to detect diagonal moves. I noticed that A* generated lots of sequences of left, forward, left, forward…, and right, forward, right, forward…. If the code sees a sequence of left and forward, that would be a diagonal left move. The effect of this is instead of moving the car left 10 cm and forward 10 cm, we move the car 45 degrees to the left and move 14.14 cm in that direction. It allowed for much smoother navigation of the car.

go_route:

```python
stop_sign = False
red_light = False
person = False

def go_route(moves):

    global stop_sign
    global red_light
    global person

    prev = "forward"

    for a in range(len(moves)):
        while person == True:
            stop()
        while red_light == True:
            stop()
        if stop_sign == True:
            stop()
            time.sleep(5)
            stop_sign = False
        if moves[a] == "left":
            if prev == "diagonal_left":
                left_45()
            elif prev == "forward":
                left_90()
            elif prev == "diagonal_right":
                left_135()
            elif prev == "right":
                turn_180()
            forward(10)
            time.sleep(0.35)
        elif moves[a] == "diagonal_left":
            if prev == "left":
                right_45()
            elif prev == "forward":
                left_45()
            elif prev == "diagonal_right":
                left_90()
            elif prev == "right":
                left_135()
            forward(10)
            time.sleep(0.5)
        elif moves[a] == "forward":
            if prev == "left":
                right_90()
            elif prev == "diagonal_left":
                right_45()
            elif prev == "diagonal_right":
                left_45()
            elif prev == "right":
                left_90()
            forward(10)
            time.sleep(0.35)
        elif moves[a] == "diagonal_right":
            if prev == "left":
                right_135()
            elif prev == "diagonal_left":
                right_90()
            elif prev == "forward":
                right_45()
            elif prev == "right":
                left_45()
            forward(10)
            time.sleep(0.5)
        elif moves[a] == "right":
            if prev == "left":
                turn_180()
            elif prev == "diagonal_left":
                right_135()
            elif prev == "forward":
                right_90()
            elif prev == "diagonal_right":
                right_45()
            forward(10)
            time.sleep(0.35)
        prev = moves[a]
    if prev == "left":
        right_90()
    elif prev == "diagonal_left":
        right_45()
    elif prev == "diagonal_right":
        left_45()
    elif prev == "right":
        left_90()
    stop()
```

This is where our car is directed to physically move. This is also where the global Boolean variables are checked. If the other thread sets them to True, this function follows the appropriate procedure. Our car turns and moves based on the current position of the car (determined by the previous instruction in the moves array) and the current instruction in the moves array. After all the instructions in the moves array are carried out and the for loop is exited, the car is directed forward again for the new scan.

The turn functions are the final functions from the code and are as follows:

```python
def turn_180():
    stop()
    time.sleep(0.1)
    turn_left(70)
    time.sleep(1.9)
    stop()
    time.sleep(0.1)

def left_135():
    stop()
    time.sleep(0.1)
    turn_left(70)
    time.sleep(1.4)
    stop()
    time.sleep(0.1)

def left_90():
    stop()
    time.sleep(0.1)
    turn_left(70)
    time.sleep(0.65)
    stop()
    time.sleep(0.1)

def left_45():
    stop()
    time.sleep(0.1)
    turn_left(70)
    time.sleep(0.32)
    stop()
    time.sleep(0.1)
```

```python
def right_45():
    stop()
    time.sleep(0.1)
    turn_right(70)
    time.sleep(0.35)
    stop()
    time.sleep(0.1)

def right_90():
    stop()
    time.sleep(0.1)
    turn_right(70)
    time.sleep(0.7)
    stop()
    time.sleep(0.1)

def right_135():
    stop()
    time.sleep(0.1)
    turn_right(70)
    time.sleep(1.4)
    stop()
    time.sleep(0.1)
```

I found I got more consistent results with the turns when stopping for a short moment before and after making them.