MXET 300
Mechatronics I

Final Project Report
SCUTTLE Robot Obstacle & Stop Sign Detection

Authors:
Mark Perez
Ricardo Rodriguez
Jacob Reyes

Due Date:
05/09/2025

Objective & Introduction:

The objective of this project is to develop an autonomous vehicle that follows a designated path, avoids obstacles, and stops at stop signs. The vehicle is programmed to follow a green road using computer vision and to detect any obstacles in its path. If the vehicle detects an object, it will stop. If it detects a stop sign, it will stop for five seconds before proceeding. We will be implementing all our knowledge from past labs and lectures to develop this project. The main sensor we are using for obstacle detection is the SICK TiM651 LIDAR sensor which features a 270° field of view. We will be using a small Logitech HD webcam in conjunction with computer vision on the Raspberry Pi to locate and follow a path.

We will be using and modifying our past programs that interface with the LIDAR sensor and color tracking capabilities of computer vision. We had to modify the LIDAR sensor's field of view to focus on a certain angle, as well as change between two set HSV values in the computer vision program.

Procedures:

In terms of the procedures for this project, the totality can be broken down into two main subsections, which include the hardware of the SCUTTLE robot and the software created for its functionality to detect obstacles and the stop sign using the LIDAR sensor and webcam respectively. In the sections found below, you will find an overview of the procedures in which the hardware setup was completed and tested before implementing our program to ensure all components were functioning correctly.

SCUTTLE Hardware:

Over the course of the semester, we implemented various hardware components within our SCUTTLE robot, which included the Raspberry Pi module, the battery pack, motors and H-bridges, the encoders, the Light Detection and Ranging (LIDAR) sensor used for obstacle detection, and the webcam used for the color vision tracking.

In order to mount the Raspberry Pi module, battery pack, and H-Bridge to the chassis of the SCUTTLE robot, a DIN railing was ordered and shortened to fit the chassis of the SCUTTLE robot, in which we then attached each of the part to the DIN rail via the clip-on fasteners. In order to power all the components on our SCUTTLE robot, power was routed from the battery pack to the Raspberry Pi module and H-Bridge, in which approximately 12 V is required to allow all components to function accordingly. To connect the motors and encoders to the Raspberry Pi module, we used the QWIIC cables included in our package to connect from the Raspberry Pi to an adapter that would allow us to connect to our encoder pins found at each motor. After completing these steps, the SCUTTLE robot is fully functional on its own, and in Figure 1, you will find an image showing the components attached to the chassis as well as the connections between the hardware components and the Raspberry Pi module. We also implemented the LIDAR sensor, in which the power to the LIDAR sensor was routed from the H-Bridge to the sensor, and the USB connection for data collection was connected to the USB

port on the Raspberry Pi. Lastly, we also implemented color tracking through the use of a webcam, which was also connected to the USB port on the Raspberry Pi.

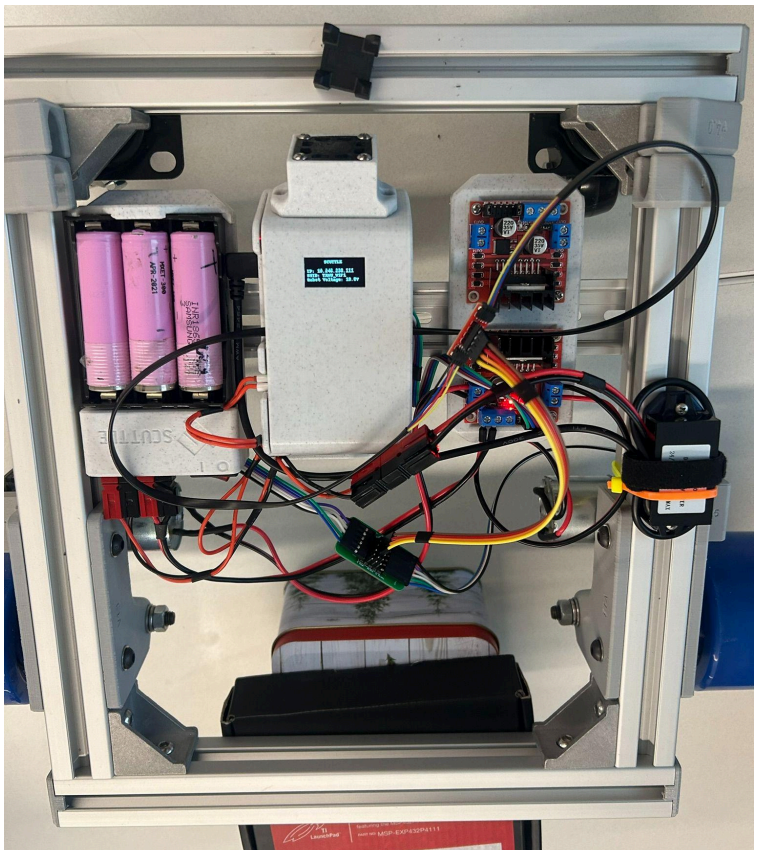

Figure 1: Hardware Components & Connections for SCUTTLE Robot

For the implementation of both the LIDAR sensor and webcam, we needed both components to be placed at the centerpoint of the front of the SCUTTLE robot, and in order to do so, we 3D printed three six inch aluminum framing rails using ABS plastic at half density to ensure it does not add any unnecessary weight to the robot. Using the printed framing rails, we were able to create a bridge over the LIDAR sensor allowing us to mount the webcam at the centerpoint of the robot while not obstructing the LIDAR sensor's viewpoints. Below in Figure 2, you will find an image depicting a front view of the SCUTTLE robot showing the mounted LIDAR and webcam used for the implementation of our final project.

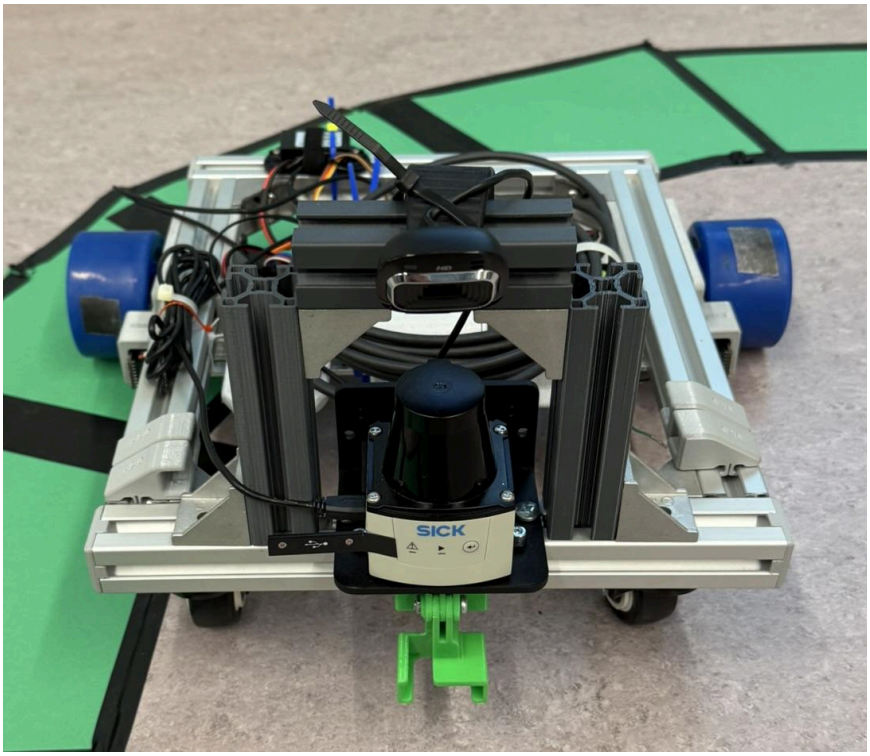

Figure 2: LIDAR & webcam Mounting for Front Facing Centerpoint Position

SCUTTLE Software Programming:

The software used to develop this scuttle robot mainly consist of the two topics learned in Lab 6 and Lab 8, the "*Lidar and Obstacle Detection*" & "*Computer Vision*" respectively. The first edit we created for our SCUTTLE robot was to edit the actual results for the "lidar.py" file in such a way that the SCUTTLE robot would only see between -80 to 80 degrees due to the mounting constructed for the camera. This would allow the scuttle robot to stop if it detects an individual in the designated range.

```
def minimized_range():
    lidarData = polarScan(54)
    valid_dis =[]
    valid_ang = []
    x = 0
    for i in range(54):
        element = lidarData[i][1]
        if ((element > -80) and (element < 80)):
            valid_dis.append(lidarData[i][0])
            valid_ang.append(lidarData[i][1])

    my = [[] for _ in range(len(valid_dis))]
    combined = np.stack((valid_dis, valid_ang), axis=1)
    return(combined)
```

Figure 3: LIDAR .py Function created to minimize range.

As we can see in *Figure 3,* we initialize the full array of all the 270 degree lidar detection into an empty array called, “lidarData”. Then we will implement a food loop that will look for every simple data point and will only accept (append) data points that fall between the range described in the code of -80 and 80 degrees. However, we will need to turn this rudimentary array into an array that “*Vector.py* ” will accept so that the vector program will only get the nearest value from that specific array.

The next step for our software would be to implement the lidar in conjunction with the “*Computer Vision*”. Computer Vision’s program “*color tracking*” will work as our main base of code where the only difference between past iterations would be the HSV values for green (*our track*) and the width of the pixel required for it to be considered aligned. However, to implement the lidar sensor, we made the main program code run only when the lidar sensor did not detect an obstacle within .35 meters as shown in *Figure 4.*

```
var1 = vec.getNearest()
try:
    while (red_detected == 0):
        while not((var1[0] > 0.04) and (var1[0] < 0.35)):
            sleep(.05)

            ret, image = camera.read()  # Get image from camera

            # Make sure image was grabbed
            if not ret:
                print("Failed to retrieve image!")
                break
```

Figure 4: Color Tracking.py Function created to operate only when an object is not detected.

Meanwhile, if the lidar sensor is between these two ranges of 4 to 35 cm then we will make the robot stop in its track by providing a 0% duty cycle as well as changing the HSV value from green to red to detect if the object in the Lidar range is a stop or not. Once we are able to distinguish if the object is a stop, we will sleep the system with a five second delay, while setting our “*red_detected*” variable to a positive value of 1.

```
while (var1[0] > 0.04) and (var1[0] < 0.35):
    sc.driveOpenLoop(np.array([0.,0.]))        # stop if no targets detected
    ret, image = camera.read()  # Get image from camera

    # Make sure image was grabbed
    if not ret:
        print("Failed to retrieve image!")
        break

    image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)             # Convert image to HSV

    height, width, channels = image.shape                      # Get shape of image

    thresh = cv2.inRange(image, (red[0], red[1], red[2]), (red[3], red[4], red[5]))   # Find all pixels in color range

    kernel = np.ones((5,5),np.uint8)                           # Set kernel size
    mask = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel)    # Open morph: removes noise w/ erode followed by dilate
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)     # Close morph: fills openings w/ dilate followed by erode
    cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[-2]                    # Find closed shapes in image
```

Figure 5: When Object is detected. Part 1.

```
if len(cnts) > 0:
    c = max(cnts, key=cv2.contourArea)                    # return the largest target area
    if np.any(c > 100):
        print("Red Detected")
        red_detected = 1
        print("First:",red_detected)
        sleep(5)
        break
    else:
        print("Not Enough Red!")
        var1 = vec.getNearest()
else:
    print("Object Detected!")
var1 = vec.getNearest()
print(var1)
```

Figure 6: When Object is detected. Part 2.

The way we were able to make sure the SCUTTLE robot would continue after only a five second delay after detecting a stop sign would be by implementing a function as the main function of our color tracking program. However, the only difference would be that we will not be utilizing the lidar sensor for a hundred iterations. This effectively caused us to ignore lidar readings of the next approximate object for roughly three to four seconds. This can be seen in *Figure 7,* down below.

```
def no_lidar(time):
    # Try opening camera with default method
    try:
        camera = cv2.VideoCapture(0)
    except: pass

    # Try opening camera stream if default method failed
    if not camera.isOpened():
        camera = cv2.VideoCapture(camera_input)

    camera.set(3, size_w)                       # Set width of images that will be retrived from camera
    camera.set(4, size_h)                       # Set height of images that will be retrived from camera
    initialized_time = 0
    try:
        while (initialized_time <= time):
            sleep(.05)

            ret, image = camera.read()  # Get image from camera

            # Make sure image was grabbed
            if not ret:
                print("Failed to retrieve image!")
                break

            image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)              # Convert image to HSV

            height, width, channels = image.shape                       # Get shape of image
```

Figure 7: Function that ignores lidar racing with input iterations. Part 2

Testing & Results:

In order to ensure that our SCUTTLE robot was functional based on its assigned tasks to follow a color coordinated track, avoid obstacles, and detect and stop at a stop sign, we broke our testing phase of the project into four separate phases, which included testing the color tracking for the track via the webcam, obstacle detection via the LIDAR, combination of LIDAR and color tracking, and stop sign interruption using the webcam.

Color Tracking of Vehicle Path Testing:

For our program, we first implemented the color tracking using the webcam and green construction paper, which we used to construct the track that the SCUTTLE robot will follow. First, we used the color tracking NodeRED file to tune the hue, saturation, and value (HSV) minimum and maximum values that would allow our program to only detect and follow the particular shade of green used for our track. Once we had the HSV minimum and maximum values, we then created the track to test if the robot were to follow the track based on the HSV values, which is where we encountered another region of testing. Within our program, we were able to adjust the target pixel width of the tracked object, which we were able to use trial and error to adjust this value to fit the average width of each paper on our track in pixels. Once we configured these two variables within our program, we tested the SCUTTLE on the track and it was able to follow the track.

Another area of testing in regards to the color tracking and path following was based on the angle at which the camera was facing the track. In the primary tests of the SCUTTLE, we noticed that if the camera was angle too high, the camera would view further ahead of where the SCUTTLE was currently at thus making the robot view curves and turn too early.

Obstacle Detection Tracking:

After ensuring our SCUTTLE could follow the track, we then implemented obstacle detection using the LIDAR sensor. Based on the alterations made to the robot in order to attach the webcam to the centerpoint of the robot, we noticed that the LIDAR sensor was detecting the side framing rails used to fix the camera to the SCUTTLE as the nearest object. In order to have the program disregard the framing rails as objects, we tested to see at which angle we could detect objects without the SCUTTLE detecting the framing rails, which after trial and error was from a range of -80° to 80°. After implementing conditional statements to limit the angular range for valid detections from -80° to 80°, we tested the LIDAR program to ensure that the sensor was detecting objects and displaying the object with the closest proximity to the SCUTTLE while ignoring the readings from the framing rails. After this proved to be successful, our next steps was to integrate the two portions of programming in order to combine path following and obstacle detection.

Combination of Path Tracking & Obstacle Detection Testing:

Once the individual codes for path tracking and obstacle detection were combined, our primary goal was to test to ensure that the detection of an obstacle halted the vehicle until the obstacle was no longer being detected by the LIDAR sensor. To do this, we tested variations of our program until we reached a procedure that would allow for this to happen, which entailed a while statement that essentially stated that if the lidar did not detect an obstacle within a certain distance from the robot, the SCUTTLE would continue on the tracked path until an obstacle was detected, and at that point the SCUTTLE would halt all motion until the obstacle was removed from the path of the SCUTTLE. Once we tested this version of the program, we noticed that the SCUTTLE was able to successfully halt its motion when an obstacle was detected a certain range from the LIDAR and continued motion once the object was removed from the SCUTTLE's viewpoint.

Stop Sign Interruption Testing:

Our last area of testing included using the previously combined codes of path tracking and obstacle detection to include using the webcam to identify a stop sign and stop the motion of the SCUTTLE robot. Our first area of testing included performing the same procedures of obtaining the HSV minimum and maximum values for the webcam detection of the red hue of the stop sign. After attaining these values, we next tested various versions of programming to determine a way to have the identification of the stop sign act as an interruption to cease all other functions for five seconds mimicking the average stop time of a vehicle at a stop sign and resuming normal function after the stop. One of the biggest issues that we encountered that required extensive testing was the positioning of the webcam in order to allow the SCUTTLE to view both the defined path to follow as well as determine if there is a stop sign detected to interrupt all other functions and stop the SCUTTLE for our predetermined time of five seconds. After testing various camera positions using the NodeRED flow from the Computer Vision laboratory assignment, we were able to determine the most accurate camera position to have the SCUTTLE view both the path as well as detect any stop signs.

Results:

After the testing phase, we noticed that the overall function of the SCUTTLE robot was satisfactory although there are points that could be improved upon. One main region of improval would be the use of a particular webcam that would allow for a wider range of view since this caused some of our issues with our project. Since the view of our webcam is very narrow, the webcam had to be positioned to view both the path it was following as well as the stop sign, which proved to be very difficult. If we were to use a webcam that would allow for either a wider and larger view or if the webcam itself can be moved by a controller, it would allow for better view of the track directly in front of the SCUTTLE as well as the stop sign positioned along the track.

Another area of improvement we noticed could be implemented with more time is a Proportional-Integral-Derivative (PID) controller that would allow for smoother motion of the

vehicle. Based on the video found below showing the testing phases of our SCUTTLE project, you can see that the SCUTTLE moves very jittery and does not smoothly drive across our predefined path. Through previous experimentations, we saw how the implementation of various combinations of PID controllers can cause the movements of the robot to transform from jittery to smooth. If we were allotted a greater amount of time to complete this project, we could have had the time to implement a controller of this type, which could have possibly allowed our SCUTTLE to have smoother transitions on the track especially when following curves in the track.

Conclusion:

In conclusion, we were able to successfully replicate the objectives designed for this SCUTTLE robot. However, we could still improve upon this project by getting a better camera that could allow us to get a better view of both the track and the view in the near vicinity. We could also improve upon the code, in such a way that we would be able to minimize the amount of time it takes for one iteration for our robot to move.

All in all, this project allowed us to grow in our understanding of computer vision, as well as our understanding on how the lidar sensor worked. This project idea was hard to implement due to several constraints such as time, availability, and inexperience with camera implementation.

Video Demonstration of SCUTTLE Vehicle:

In the video linked below, you will find a demonstration of the SCUTTLE robotic vehicle as it maneuvers the track created while detecting obstacles as well as stop signs that have been placed around the track.

Demo of SCUTTLE Robot

In the link below, you will find a full folder filled with all the python programs we have edited or constructed for the SCUTTLE robotic vehicle. All files must be in the same directory for the “Final_Project.py” file to work as intended.

Final Project MXET 300 GitHub Repository