Enabling Autonomous Robotic Vehicle Navigation on Mars: An External Camera Aerial View Approach through Colour Detection Algorithm

Conducted by: Yann Raphael Janssen, Rafid Hossain, Leyan Ouyang, Adrian Murselaj, and Antoine Valadon.
Written by: Yann Raphael Janssen, Rafid Hossain, Leyan Ouyang, Adrian Murselaj, and Antoine Valadon.

Table of Contents

Word Count: 4997

Abstract:

This project aimed to develop a navigation system for a Mars rover to autonomously navigate from Point A to Point B using colour detection technology. An external camera captured aerial images of the rover and a target object, both identified by a distinct green colour, simulating the green lab notebooks. These images underwent a colour detection algorithm to determine the coordinates of the rover and target object in the camera's plane. The algorithm, implemented on a Raspberry Pi, converted pixel distances into real-world measurements, enabling precise navigation. Results showed a linear relationship between distances in the camera's plane and their real-world counterparts, determined through linear interpolation. The navigation system algorithm was run multiple times for three arbitrary path routes, with the rover averaging 16.9% off from the centre point of the target object's coordinates. Despite this deviation, the rover successfully reached the vicinity of the target object in all tests. This outcome demonstrates the system's effectiveness in reaching intended destinations within an acceptable margin. This project showcases the feasibility of autonomous robotic vehicle navigation using colour detection technology and offers valuable insights for subsequent research endeavours to build upon and progress further.

I. Introduction

I.1 - Objective

The primary objective of the project was to construct a navigation system for the robot to be able to travel to a target location. This was done through the use of an external camera with an aerial view, which would take pictures and send it to the laptop to undergo a colour detection algorithm. For the sake of the experiment, the rover and target object, both have a distinct colour. Bearing in mind this colour detection is applicable to any colour, and in this case, the green lab notebook's HSV was known, and used to represent the rover and target object’s initial position. The colour detection algorithm being run on the images captured by the external camera can detect the coordinates of the rover, and target object in the camera’s plane, due to the range of the colour being detected, matching the colour of the rover, and the target object. The coordinates in pixel distance analysed by the algorithm are then sent to the Mars rover, where it converts the coordinates into a distance metric. An input time value can be retrieved by linear interpolation to the equivalent distance value discovered, directing the rover to travel for that amount of time. The algorithm for the navigation system was developed in the Raspberry Pi, by one of the group members. Moreover, the aim is to demonstrate successful autonomous robotic vehicle navigation, through a colour detection approach, implemented through an external camera with an aerial view.

I.2 - Background Information

Various navigation systems enable the rover's movement and positioning. Odometry involves computing the rover's positional changes relative to its starting point over time (GM0.org, n.d.).

Triangulation utilises three external reference points forming a triangle to determine the object's position based on the distances between these points and the object (Advnture, 2022). In-built cameras on the rover allow it to spin continuously until it detects an object in the centre of its view, upon which it stops and advances towards the recognised object (Medium, 2019). Line tracking relies on the rover's capability to perceive light levels beneath it, enabling it to track and follow a black line (Raspberry Pi Foundation, n.d.). External cameras, mounted on the ceiling, capture images of the area, aiding in object detection and determining the rover's location (Instructables, n.d.).

Why were the external camera chosen as a navigation technique:

In a lab, the structure of the environment may change. The ability to reposition an external camera would be beneficial to maintain a clear view of the surroundings showcasing flexibility.

An External camera can provide high resolution images compared to an inbuilt PiCamera on the rover. Higher resolution can improve accuracy of tasks such as object detection.

External cameras can provide real time feedback by capturing images at specific time intervals, enabling the rover to adapt to changes in environment such as moving objects or obstacles.

External camera eliminates the need of additional hardware required by other navigation systems such as triangulation, or line tracking, simplifying the set up.

External cameras are less susceptible to lighting, meaning they are not affected by the surface textures, ensuring consistent and reliable imaging. In contrast to other navigation systems such as line tracking, and ultrasound mapping.

How Does Colour Detection Work?

The most commonly used colour model is composed of red, green, and blue (RGB). Different intensities of each colour can be combined to make colours across the spectrum. Computer screens are made up of pixels. In each pixel, there is a combination of specific intensities of the three colours (RGB), since a computer screen has thousands of pixels, a coloured image can be created.

Hue, Saturation, and Value (HSV) is another colour model used to represent colours. Hue is an angle around a colour wheel, saturation is a percentage of the maximum intensity, and value is a percentage of the maximum brightness. HSV models are used more than RGB in computer vision applications where an image needs to be analysed since this model is more intuitive for humans in the way that we perceive light.

Colour Detection is an essential component of the navigation system algorithm, where the rover successfully travels to the target object. Both, the rover, and the target object, will have a distinct colour, in comparison to the plane seen from the camera at an aerial view.

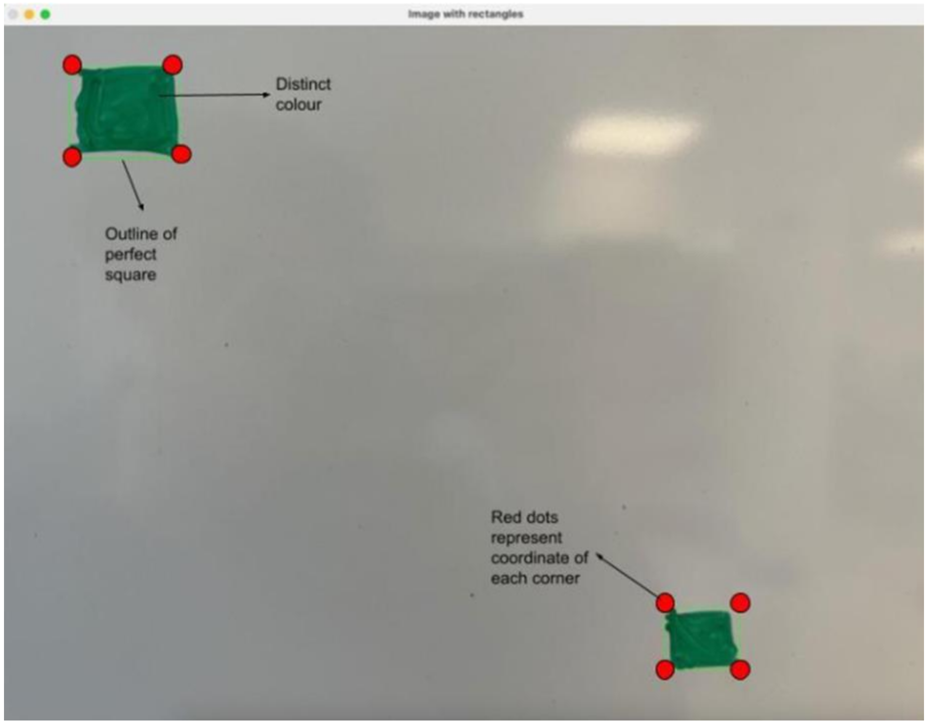


Image with rectangles
Distinct colour
Outline of perfect square
Red dots represent coordinate of each corner

Figure 1: abstract diagram showing colour detection of green shade.

After being given a coloured image of the area by the external camera, the colour detection algorithm would outline a rectangle around the distinct contours of colours (Fig.1). Assuming that the target object and rover from an aerial viewpoint are a quadrilateral, a box can outline the colours, and the coordinates in each corner are identified (Fig.1). Further, the midpoint of the quadrilaterals is calculated.

Colour detection can be effective as a navigation system technique, it can easily identify stationary or moving obstacles in the rover's path. This is only true if there is a constant feed of live updates from the external camera. The broad aerial view that the external camera provides and the efficacy of identifying the coordinates of these objects, through colour detection, demonstrates their positive association as a navigation methodology.

Colour detection will be applied to the navigation algorithm and will be written using OpenCV python (ProjectPro.io, 2023). This is further explained in detail in the methodology section, as well as the logistics of the experimental procedure including the setup of the rover's hardware, and the external camera.

II. Methodology:

II.1 - Hardware Components of the Rover

The rover itself consists of different sets of hardware components. These components are classified under four specific categories according to their different functions: power supply, Raspberry Pi board, multiple sensors, and motors. Figure 2 presents a diagram of the rover with its hardware components, and the functionalities of these components are explained below.

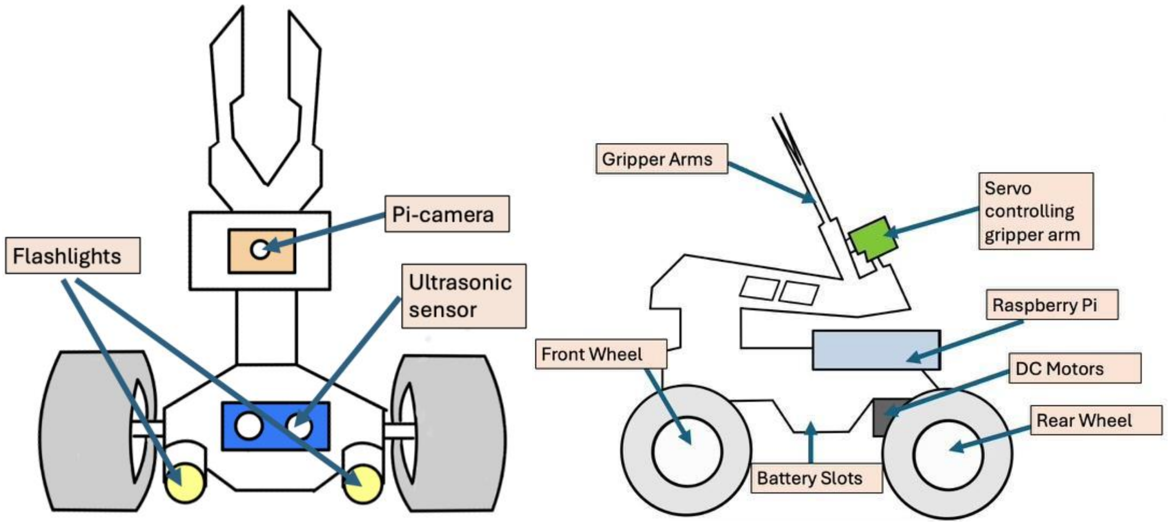


Figure 2: Diagram of the different Hardware Components of Rover.

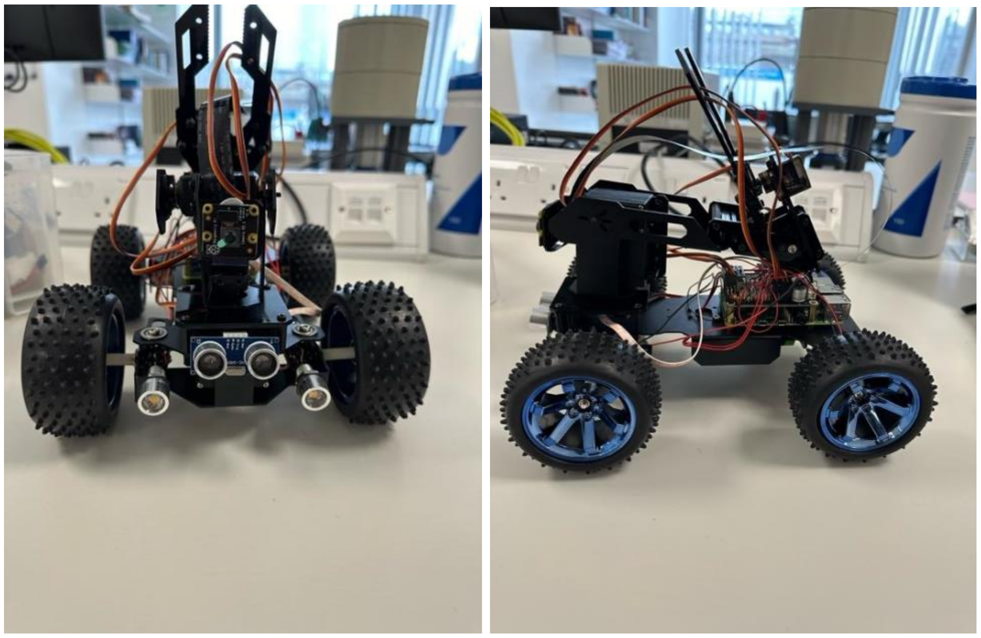


Figure 3: Overall view of the rover.

Power Supply:
The entire rover was powered by two 5V batteries. The batteries were directly connected to the Raspberry Pi board, and all other relevant sensors and motors.

Raspberry Pi:
The Raspberry Pi board served as the centre controlling system of the rover. The codes were compiled and uploaded onto the Raspberry Pi board to control the motors and the sensors. The board also enabled transmission of information between the laptop and the rover (Upton et al, 2012). If any errors were encountered, the motherboard could communicate with the laptop in an attempt to fix any problems. Please refer to Appendix A, where the software section is discussed for specific details on the configuration of the Raspberry Pi.

Sensors:
There were sensors on the rover which were primarily responsible for data gathering from the surrounding environment. There was a front mounted ultrasonic sensor capable of detecting distances between the rover and obstacles. A pi camera was attached to the rover which could ideally detect objects encountered in front of it.

Motors:
The motors were directly powered by the power supply which facilitated the rover to propose motions. There were two DC motors which powered the rear wheel. The gripper arm was controlled by another servo motor which enabled it to pick up objects.

The four components on the rover complement each other and achieve their collective functionality. The Raspberry Pi board serves as the central processing unit, providing computational power and interfacing abilities. Paired with the motors which enable physical movements and sensors which provide crucial feedback from the surrounded environment to help propel decision making. The batteries ensure that the whole setup is powered and functioning properly.

In order to achieve navigation for the rover, the implementation of an external camera is crucial, to observe from an aerial view, and to calculate its path route towards the target object. In the next section, the procedure and set up of implementing an external camera to provide an aerial view of the plane is described.

II.2 – Implementing an External Camera (Aerial View)

The setup of the external camera system involves an “Arduino nano 33 ble” board and an OV7670 camera module. The Arduino board is connected with the camera module according to the circuit diagram in Figure 4 (Github, 2021), so that the board can trigger the camera to capture the image and send it to the board. These data bytes are stored inside the Arduino board, and through a USB cable, which connects the Arduino board to the laptop, allowing the experimenters to obtain the data for further image processing. This image processing will be discussed in the next section, The Colour Detection Algorithm.

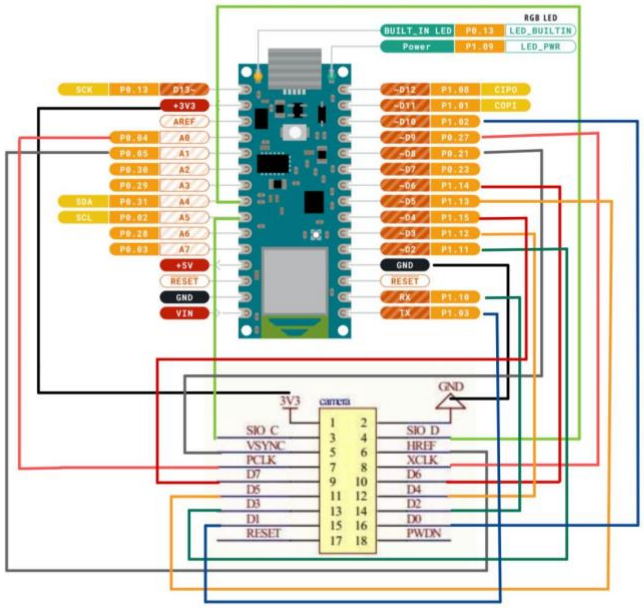
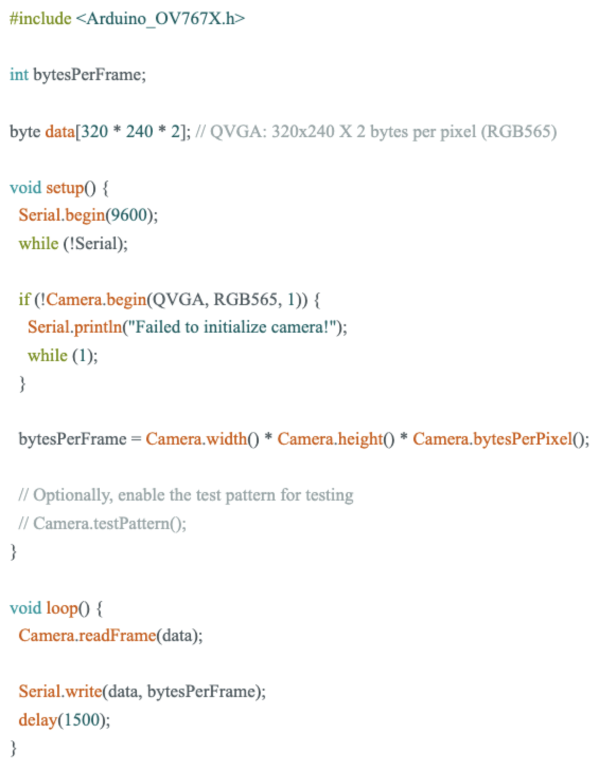


Figure 4: Circuit Diagram of How Camera is Connected.

Figure 5 below, shows the code for the Arduino board. The key purpose of the code is to use the camera module to capture image data in raw bytes which instead of showing the image in a direct picture on a screen, is shown in a series of digital symbols and numbers. After that the data is sent to the laptop through the Arduino board.



```
#include <Arduino_OV767X.h>

int bytesPerFrame;

byte data[320 * 240 * 2]; // QVGA: 320x240 X 2 bytes per pixel (RGB565)

void setup() {
 Serial.begin(9600);
 while (!Serial);

 if (!Camera.begin(QVGA, RGB565, 1)) {
  Serial.println("Failed to initialize camera!");
  while (1);
 }

 bytesPerFrame = Camera.width() * Camera.height() * Camera.bytesPerPixel();

 // Optionally, enable the test pattern for testing
 // Camera.testPattern();
}

void loop() {
 Camera.readFrame(data);

 Serial.write(data, bytesPerFrame);
 delay(1500);
}
```

Figure 5: Arduino Code for External Camera Aerial Viewpoint

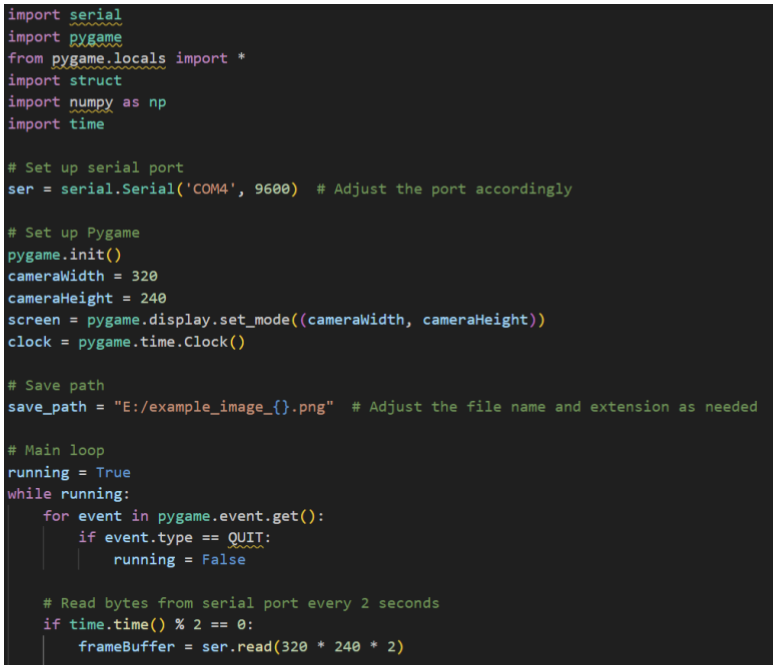
Initially, a library called “Arduino-OV767X” was imported to allow simpler coding functionalities with the camera module (OV7670). Then, a global variable 'BytesPerFrame' that defines the size of recorded data, was created. The camera used for navigation of the rover from an aerial view captures images with a length of 320 pixels, width of 240 pixels and records the data with two bytes per pixel (Mehendale et al 2022).

Afterwards, the monitor serial connection is opened within a frequency of 9600 Hz. This proposes data communication between the Arduino board and the laptop. In addition, the camera module is initialised, and if it fails to initialise, an error message will be sent to the computer through the serial connection.

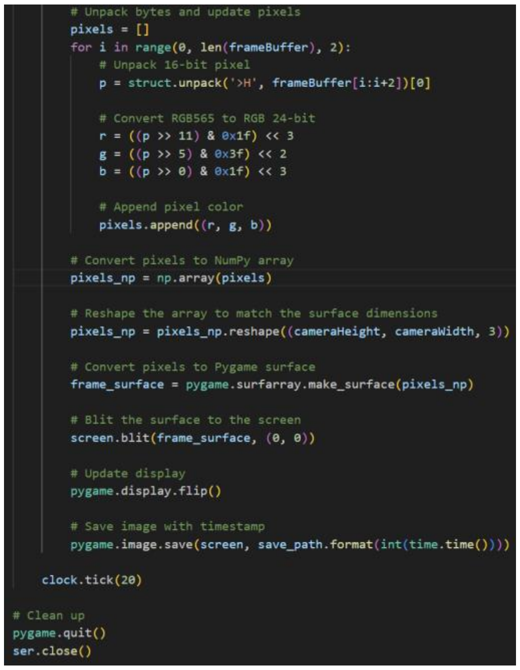
Finally, the Arduino board will now execute and loop through the code until there is a disconnection from the power source, or if told to break, a command disrupts the code. The raw image data per frame is read from the camera module and sent back to the laptop through serial connection with the board. For each captured frame, the board needs to send the data with the correct size specified by the 'BytesPerFrame' variable, previously described, which would be computational taxing for the Arduino board to process and transmit, due to its significantly large size. As a result, a delay of 2.2s is placed before the camera reads another image frame. After the codes is uploaded to the Arduino board, it is time to figure out how to read the data from the Arduino board in the laptop.

Figure 6: Captured data from the Arduino shown on the laptop serial communicator.

Figure 6 above shows the captured data bytes from Arduino. The characters are written in ASCII code where each of the characters tells a colour component of a frame pixel for the program to understand (Waters et al, 2021). After the data is captured and sent to the laptop, the data bytes are converted into coloured images, by the image processing algorithm. This algorithm reads the data bytes in RGB565 colour format, which is a language that specifies each byte's weight in red, green, and blue colour scales (Xijun et al 2015).



```
import serial
import pygame
from pygame.locals import *
import struct
import numpy as np
import time

# Set up serial port
ser = serial.Serial('COM4', 9600)  # Adjust the port accordingly

# Set up Pygame
pygame.init()
cameraWidth = 320
cameraHeight = 240
screen = pygame.display.set_mode((cameraWidth, cameraHeight))
clock = pygame.time.Clock()

# Save path
save_path = "E:/example_image_{}.png"  # Adjust the file name and extension as needed

# Main loop
running = True
while running:
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

    # Read bytes from serial port every 2 seconds
    if time.time() % 2 == 0:
        frameBuffer = ser.read(320 * 240 * 2)
```



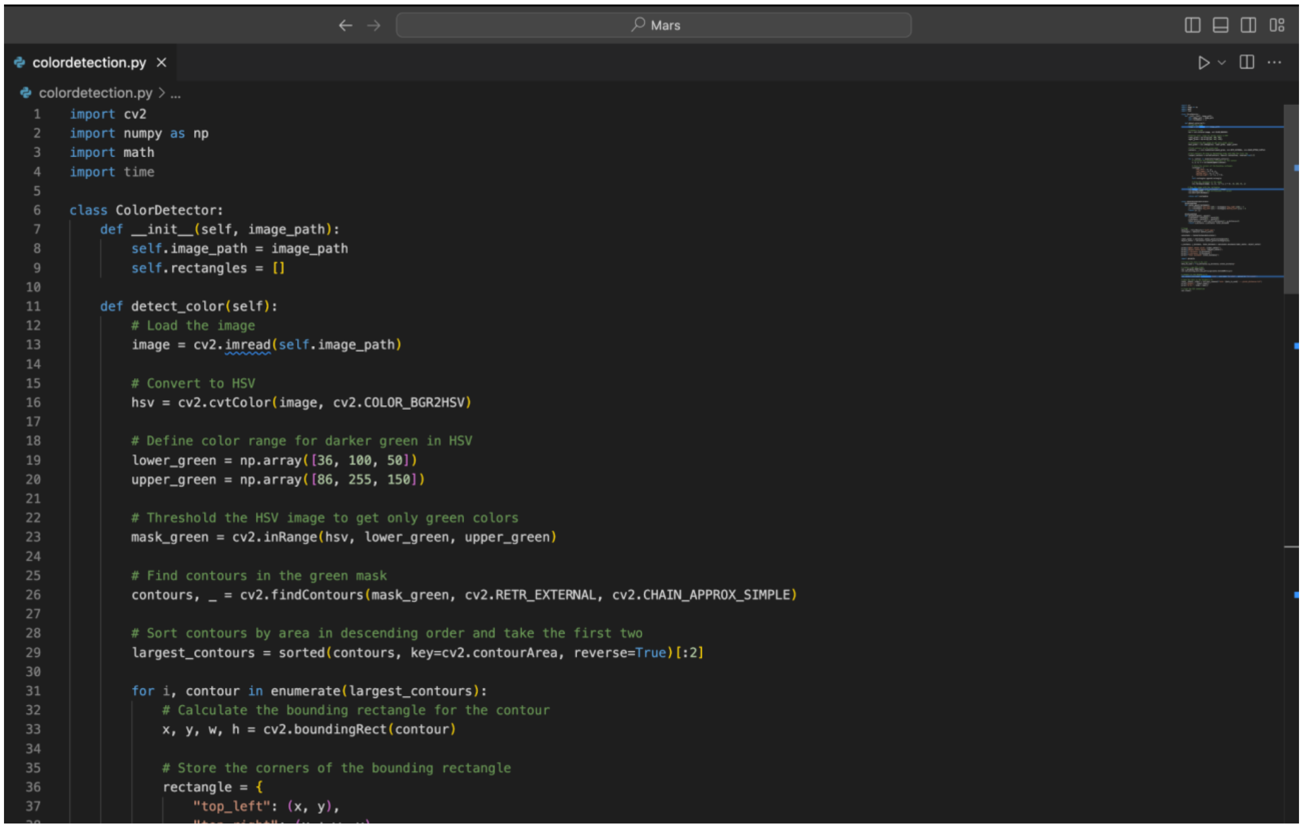
```
        # Unpack bytes and update pixels
        pixels = []
        for i in range(0, len(frameBuffer), 2):
            # Unpack 16-bit pixel
            p = struct.unpack('>H', frameBuffer[i:i+2])[0]

            # Convert RGB565 to RGB 24-bit
            r = ((p >> 11) & 0x1f) << 3
            g = ((p >> 5) & 0x3f) << 2
            b = ((p >> 0) & 0x1f) << 3

            # Append pixel color
            pixels.append((r, g, b))

        # Convert pixels to NumPy array
        pixels_np = np.array(pixels)

        # Reshape the array to match the surface dimensions
        pixels_np = pixels_np.reshape((cameraHeight, cameraWidth, 3))

        # Convert pixels to Pygame surface
        frame_surface = pygame.surfarray.make_surface(pixels_np)

        # Blit the surface to the screen
        screen.blit(frame_surface, (0, 0))

        # Update display
        pygame.display.flip()

        # Save image with timestamp
        pygame.image.save(screen, save_path.format(int(time.time())))

    clock.tick(20)

# Clean up
pygame.quit()
ser.close()
```

Figure 7: Code used to transfer the data bytes from the board to the laptop as images visible to the human eye.

Four libraries are imported here which are serial, pygame, struct and time. Where the serial library helped to set up the serial port connections between the laptop and the board. The struct library was used to process binary data which helped unpack and process the data packets from the Arduino board. Then the unpacked data is translated by the image processing algorithm mentioned above and the original data is rearranged into RGB indexed arrays. Before the full coloured image was generated, the pygame library was applied to build a new frame window where the RGB numbers are shown as a visible coloured image to the human eye. At last, the time library helped to control the time float, to take the delay into account, so that a frame is read from the port every 2.2s. The generated images were saved in the predetermined route inside the laptop for the colour detection algorithm, which is further explained in detail in section II.3.

II.3 - Colour Detection Algorithm

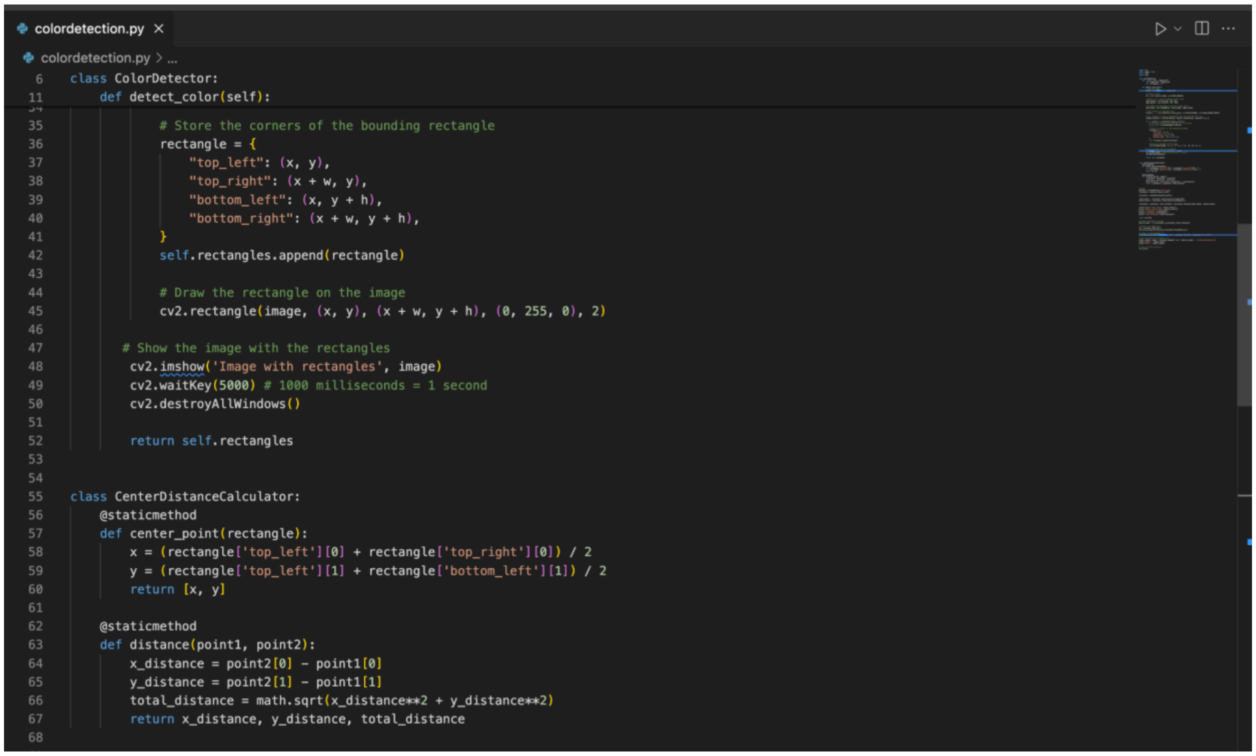
There will be screenshots attached below, showing the code written by one of the members of the group for the colour detection algorithm along with some explanation of it. OpenCV is a computer vision library widely used for image processing tasks, and it is a new library to the coder, so there was the assistance of AI in the use of the cv2 library code (Github, 2023). Additionally, paramiko library enables in python SSH commands and the transfers of files, and there was also the assistance of AI for this (Github, 2023).



```python
import cv2
import numpy as np
import math
import time

class ColorDetector:
    def __init__(self, image_path):
        self.image_path = image_path
        self.rectangles = []

    def detect_color(self):
        # Load the image
        image = cv2.imread(self.image_path)

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

        # Define color range for darker green in HSV
        lower_green = np.array([36, 100, 50])
        upper_green = np.array([86, 255, 150])

        # Threshold the HSV image to get only green colors
        mask_green = cv2.inRange(hsv, lower_green, upper_green)

        # Find contours in the green mask
        contours, _ = cv2.findContours(mask_green, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

        # Sort contours by area in descending order and take the first two
        largest_contours = sorted(contours, key=cv2.contourArea, reverse=True)[:2]

        for i, contour in enumerate(largest_contours):
            # Calculate the bounding rectangle for the contour
            x, y, w, h = cv2.boundingRect(contour)

            # Store the corners of the bounding rectangle
            rectangle = {
                "top_left": (x, y),
```

Figure 8: Part 1 of the colour detection code

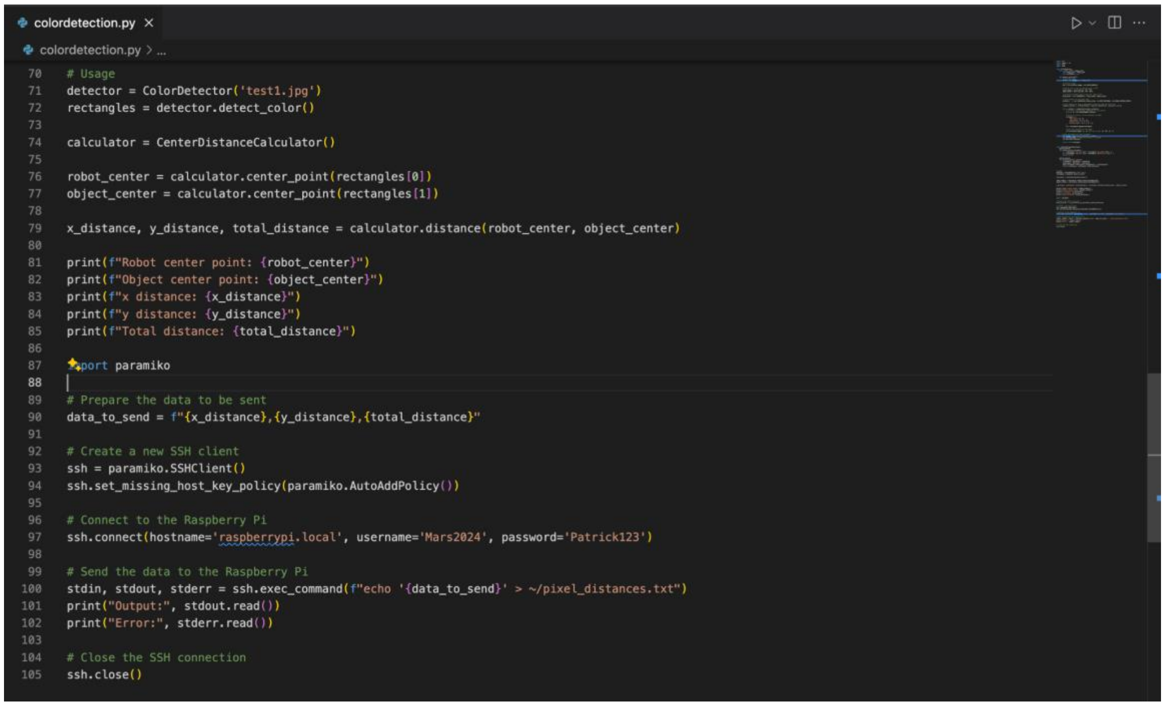
For starters the necessary libraries are imported in the python file. A class “ColorDetector” is defined, with an initializer that takes an image path as an input and initialises an empty list of ‘rectangles’. The detector_color function reads the image for image processing tasks (height, width, channels, etc). Then the cvtColor function in the variable ‘hsv’ converts the image from a RGB colour space to a HSV colour space. The lower and upper boundaries for the colour green are defined with their corresponding HSV values for the colour green (of the lab notebook). Then the mask_green line creates a binary mask where pixels within the green colour range are set to 255 (highest value upper range), and everything else is set to 0. The find contours function is used to find a curve joining all continuous points of the same colour or intensity. Then these contours are sorted out by their area in descending order and take the two largest ones. A loop is iterating through the two largest contours calculating the bounding rectangles, and the function returns the x and y coordinate of the top left corner, as well as the width and height of the rectangle.



```python
class ColorDetector:
    def detect_color(self):
                # Store the corners of the bounding rectangle
                rectangle = {
                    "top_left": (x, y),
                    "top_right": (x + w, y),
                    "bottom_left": (x, y + h),
                    "bottom_right": (x + w, y + h),
                }
                self.rectangles.append(rectangle)

                # Draw the rectangle on the image
                cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)

           # Show the image with the rectangles
            cv2.imshow('Image with rectangles', image)
            cv2.waitKey(5000) # 1000 milliseconds = 1 second
            cv2.destroyAllWindows()

            return self.rectangles


class CenterDistanceCalculator:
    @staticmethod
    def center_point(rectangle):
        x = (rectangle['top_left'][0] + rectangle['top_right'][0]) / 2
        y = (rectangle['top_left'][1] + rectangle['bottom_left'][1]) / 2
        return [x, y]

    @staticmethod
    def distance(point1, point2):
        x_distance = point2[0] - point1[0]
        y_distance = point2[1] - point1[1]
        total_distance = math.sqrt(x_distance**2 + y_distance**2)
        return x_distance, y_distance, total_distance
```

Figure 8: Part 2 of the colour detection code

A dictionary is created to represent the bounding rectangle with the corresponding coordinates of all the corners of that rectangle, and this rectangle is then added to the rectangle list initialised as empty earlier. The image is displayed with rectangles outlining the colour green in the image for 5 seconds. The class CentreDistanceCalculator is defined, where the centre point of these two rectangles is calculated by finding the middle point in the width and length of the rectangle (x and y coordinates). And the distance is calculated through Pythagoras theorem.



```python
# Usage
detector = ColorDetector('test1.jpg')
rectangles = detector.detect_color()

calculator = CenterDistanceCalculator()

robot_center = calculator.center_point(rectangles[0])
object_center = calculator.center_point(rectangles[1])

x_distance, y_distance, total_distance = calculator.distance(robot_center, object_center)

print(f"Robot center point: {robot_center}")
print(f"Object center point: {object_center}")
print(f"x distance: {x_distance}")
print(f"y distance: {y_distance}")
print(f"Total distance: {total_distance}")

port paramiko

# Prepare the data to be sent
data_to_send = f"{x_distance},{y_distance},{total_distance}"

# Create a new SSH client
ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

# Connect to the Raspberry Pi
ssh.connect(hostname='raspberrypi.local', username='Mars2024', password='Patrick123')

# Send the data to the Raspberry Pi
stdin, stdout, stderr = ssh.exec_command(f"echo '{data_to_send}' > ~/pixel_distances.txt")
print("Output:", stdout.read())
print("Error:", stderr.read())

# Close the SSH connection
ssh.close()
```

Figure 9: Part 3 of the colour detection code

Now, the ColorDetector class is used to detect green coloured objects in an image, because the parameters chosen to define the colour green (HSV values) and presents them as rectangles. It uses the CentreDistanceCalculator class to calculate the robot's (starting position) and object's (final position) centre point, and prints out the x, y and total distance in terms of pixels between the two rectangles.

After importing the paramiko library, there is an SSH connection being established between the laptop and the Raspberry Pi through the hostname, username and password. Then it prepares to send the x, y, and total pixel distance through a txt file "pixel_distance.txt" in the home directory of the Pi. It will print out any errors (if there are any when executing this command), and finally closes the SSH connection.

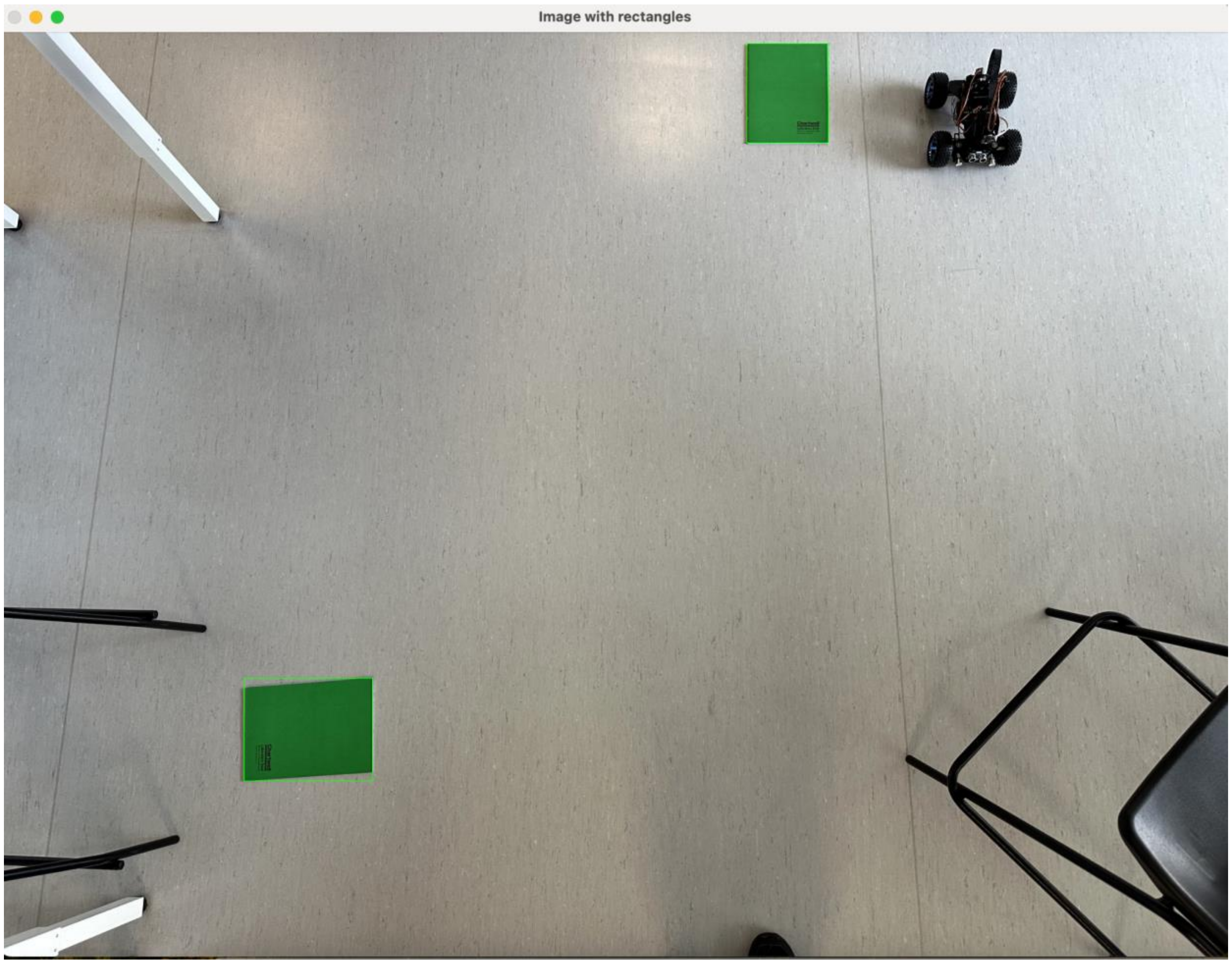
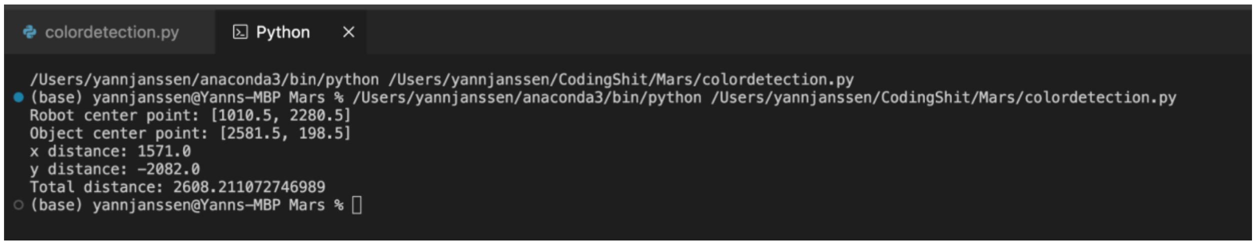


Figure 10: Test 1 of Colour Detection Algorithm

Above, Figure 10, was the first attempt of using two green lab notebooks, one was the initial position, and the other notebook was the end position. The OpenCV library in python defines a left-handed coordinate system where origin is defined in the top left corner of the image, and going down in the y-direction is positive and to the right in the x-direction is also positive. As long as the orientation of picture is taken with the robot facing down, the algorithm is able to detect as the notebook with the smallest y distance as the starting position (the notebook at the top of the image), and if they were both to be on the same y-axis, then the one with the smallest x distance is also considered into account. The code can be modified to interpret a green rectangle and a red rectangle, to distinguish the starting and end position with more ease.



```
/Users/yannjanssen/anaconda3/bin/python /Users/yannjanssen/CodingShit/Mars/colordetection.py
(base) yannjanssen@Yanns-MBP Mars % /Users/yannjanssen/anaconda3/bin/python /Users/yannjanssen/CodingShit/Mars/colordetection.py
Robot center point: [1010.5, 2280.5]
Object center point: [2581.5, 198.5]
x distance: 1571.0
y distance: -2082.0
Total distance: 2608.211072746989
(base) yannjanssen@Yanns-MBP Mars %
```

Figure 11: Printed out results robot centre point, object centre point, x,y coordinates and total distance (pixel units)

Figure 11 shows the coordinates of where the robot and target object are in the plane of the external camera's aerial perspective, and the distance values between the two objects. It should be noted that these distance values were measured in the units of pixels, as the images were captured from the camera and measured by the computer, not by a human being with a ruler. The distance travelled by the robot is measured in centimetres, so it is evident that there is the need for a conversion rate between the units of pixels to a distance metric in the real world (centimetres). To be able to have this transformation from the camera's plane to the robot's plane, an experiment was conducted to find what values of distance in centimetres correspond to the pixels caught on the camera. So, for every pixel needed to travel a certain distance in the computer's plane, there will be a corresponding distance, in centimetres, for the rover to travel in the real-world plane. The experimental set up and procedure for this conversion rate is explained below in section II.4.

II.4 – Distance Conversion Rate

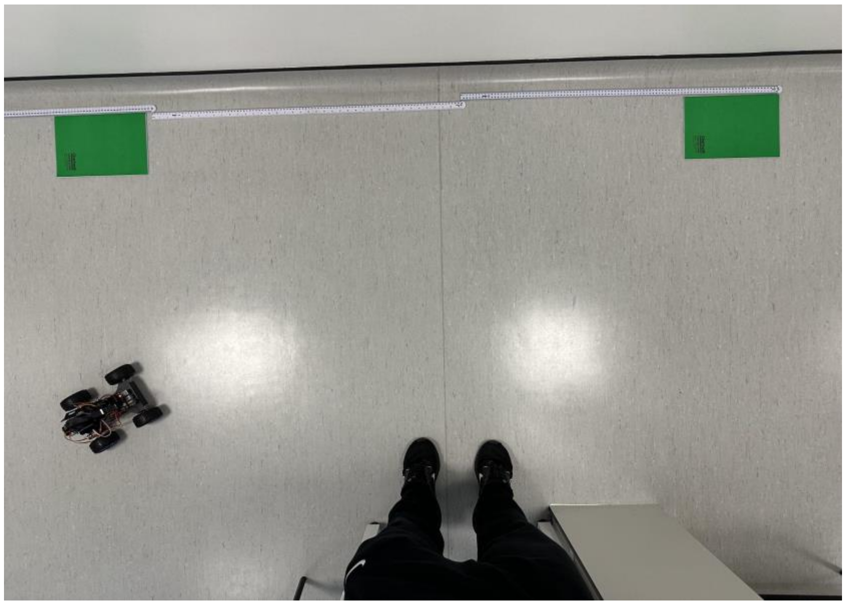


Figure 12: Experimental Setup for The Conversion Rate

The experimental setup depicted in Figure 12 aims to determine the conversion rate between pixel distances in the camera's plane and real-world distances in centimetres. There were three ruler metre sticks placed down next to one another, one green lab notebook placed on the right side (starting position), and the other green lab notebook placed at the end position of the rover. Once given the command for the rover to move forward in a straight line it requires two inputs, the speed, and the time duration for how long the robot is set to travel. The speed input is a percentage value of the max speed, so the input could be 75, representing 75% of the max speed. After a few trials with different speeds the movement of the rover tends to deviate the most with speeds higher than 60%, so for simplicity and effectiveness it was decided to have a constant speed set at 50% (controlled variable) for the forward and backward movement of the rover. The distances will be measured for a time input (independent variable) ranging from 0 to 4.5 seconds, in increments of 0.5 seconds. The procedure on how to attain these measurements goes as the following.

Firstly, the rover needs to be placed parallel to the green notebook in the starting position, and after commanding it to execute a specific time input at speed 50%, it will travel a certain distance. Then that distance is measured through the use of the metre sticks attaining a result in centimetres in the real-world plane. Afterwards, the second green notebook will be placed parallel to where the rover ended, a picture will be taken from an aerial view, and processed through the colour detection algorithm. The algorithm is being used as a tool to measure the distance in pixels between the two notebooks. This was repeated five times per time trial to ensure precision. Now, for that time input, there is a corresponding distance value in centimetres (real-world plane) and in pixels (camera's plane). It should be noted that all measurements are taken from the centre point of the green notebook, and the centre point of the rover. This data was processed and is analysed in section III.

III. Results & Discussion:

To explore the relationship between the distance travelled by the rover, and time, the velocity was calculated in the real-world plane (centimetres/sec), and in the camera's plane (pixels/sec), and they are displayed in the graphs below.

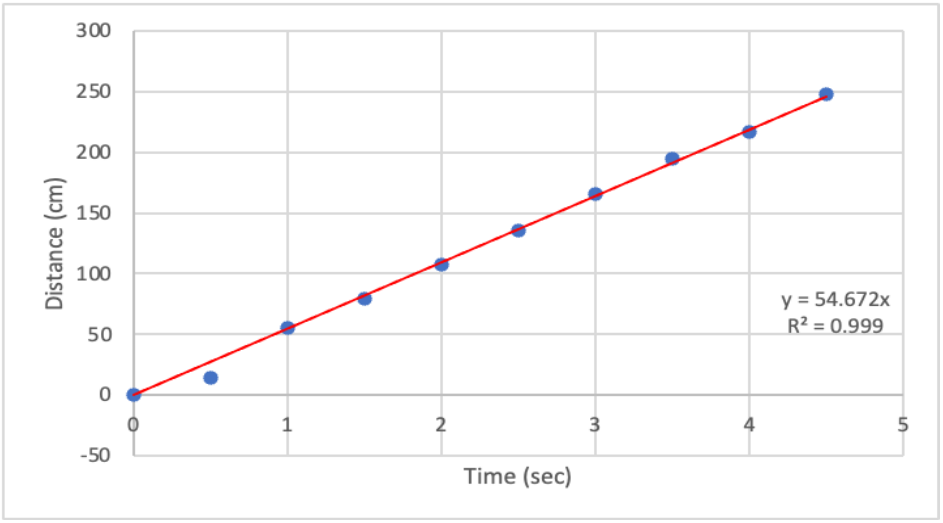


Figure 13: Distance (cm) plotted against Time (sec)

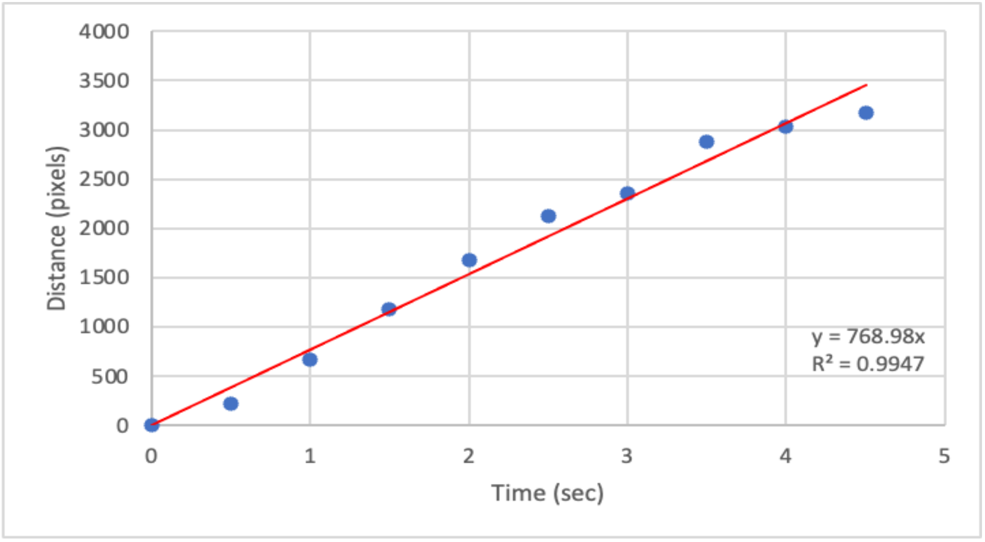


Figure 14: Distance (pixels) plotted against Time (sec)

It is evident that in the graph above that velocity is constant when the rover is travelling in a straight line, for both, camera's plane, and the real-world plane. The initial velocity at a time interval 0 to 0.5 seconds, does not follow the same linear relationship of the rover's velocity throughout the rest of its route. This is due to the initial acceleration to get the rover to move initially from rest to constant speed, requiring the initial velocity to not be constant, which can be seen from the first two data points in both graphs. This may lead to some inaccuracies when commanding the robot to travel a distance corresponding to time 0 to 0.5 seconds.

There was an observation made about the braking distance, when the rover was instructed to stop, the distance travelled was minuscule in comparison to the distances measured throughout the experiment, and for that reason it will be neglected.

As for the uncertainty measurements on the distance travelled by the robot in the real-world plane, it was calculated that if a measurement were to lie in between 0 and 100cm, the corresponding uncertainty would be 0.05cm + 0.05cm = 0.1cm. For 100cm to 200cm, it would be 0.2cm, and for 200cm to 300cm would be 0.3cm, since each metre stick has an uncertainty value of 0.1cm coming from the use of both ends of the ruler. This calculation was included in the error bars for the graphs above.

The values for the error bars are between 0.1cm and 0.3cm, which are merely visible to the human eye in comparison to the data measured.

In the camera's plane, the colour detection algorithm was used as a tool to measure the corresponding distance travelled by the rover in terms of pixels. The computer measuring the distance has utmost accuracy, and the uncertainty values are minuscule in comparison to the measurements done by the human eye. To ensure precision each time interval has been repeated five times, and the processed results are shown in Figure 13 and 14...

The greatest source of error comes from the deviation of the rover which will be discussed in the evaluation and improvements section. The relationship between the distance in the two planes is established in Figure 15, below.

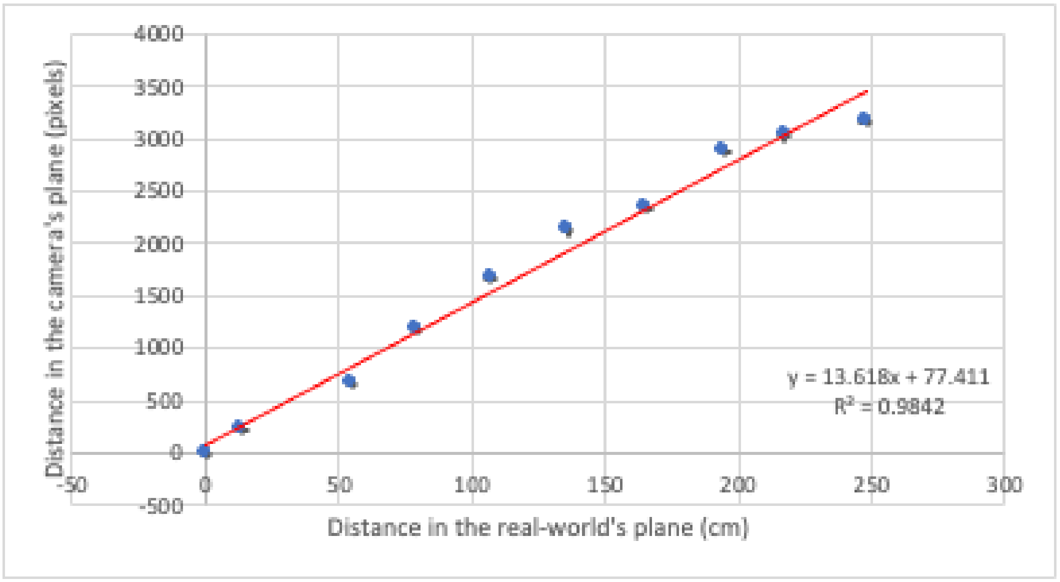


Figure 15: The distance in camera's plane and real-world's plane

It is evident that there exists a linear relationship between the data points representing distances in the camera's plane and their corresponding real-world distances. To accurately determine a distance value lying between two given data points, linear interpolation was employed.

Linear interpolation relies on a linear relationship between the consecutive data points to estimate intermediate distances between known data points. By using the linear interpolation formula (Wikipedia, 2019),

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$$

Where x and y represent the distance in the camera's plane and the real-world plane respectively, and x0, x1, y0 and y1 represent the distance values of adjacent data points. Inputting a pixel distance from the camera plane, will provide a distance in the real-world, and linear interpolation is then repeated, if the data point lies in between two other measurements, for that distance value to find its corresponding time value.

This interpolation allows for the calculation of real-world distance values that correspond to distances in the camera's plane. These calculated conversion values, from the distances in the camera's plane, are crucial for navigation purposes, as they provide the rover with precise time input values that result in the distance measurements calculated to travel in the real-world plane. Furthermore, each interpolated distance value is accompanied by a corresponding time value, ensuring accurate navigation to the desired distance within the specified time frame.

Moreover, implementing this distance conversion factor between the real world and pixel coordinate plane, in the Raspberry Pi, for the rover's navigation system is crucial. The technicalities of the code for the algorithm are explained in detail in the appendix B section.

After running the navigation system algorithm ten times for the 'test 1' path shown in Figure 10, the rover's end position was measured in comparison to its intended destination (midpoint of the notebook). After measuring the distance of the rover's end point and the target object's position, from the rover's starting point, the percentage difference was calculated, to demonstrate its accuracy. The results were found to be, on average, 14.6% off from the centre point of the green lab notebook (target object's coordinates). This was repeated for two different arbitrary paths, where the rover was found to be 18.9% and 17.3% respectively off the target object's coordinates. It's worth mentioning that despite the navigation system's limitations, the rover successfully reached the vicinity of the green notebook. **This outcome showcases successful navigation, as being only, on average, 16.9% off the centre point of the notebook, which is within an acceptable margin for the rover to activate its grippers and pick up the object at its intended destination.** Moreover, the evaluation and improvements section seek to critically assess several aspects of the experiments that may affect the accuracy and precision of the results found.

IV. Evaluation and Improvements:

Throughout this project, a thorough analysis was conducted of various aspects that may be affecting the data reliability and precision. Identifying solutions to these issues and recognising strengths and weaknesses were essential steps in enhancing the efficiency of the navigation system.

Manoeuvring the rover to turn posed significant challenges. When attempting to turn the rover, the motor attached to the front wheels would initiate the rotation, causing the front wheels to pivot. However, this approach resulted in a considerable turning radius when run, often displaying inconsistency in both size and position. This issue stemmed from the rover's relatively large wheelbase (distance between the front and rear wheels), and due to the limited angle, the front wheels could pivot. Unfortunately, due to the fixed chassis of the rover, adjustment to the wheel positions would not be a viable option to pursue. Inspired by the way tanks are able to turn with such a tight turning radius without being able to pivot their tracks relative to their hull, differential steering was implemented.

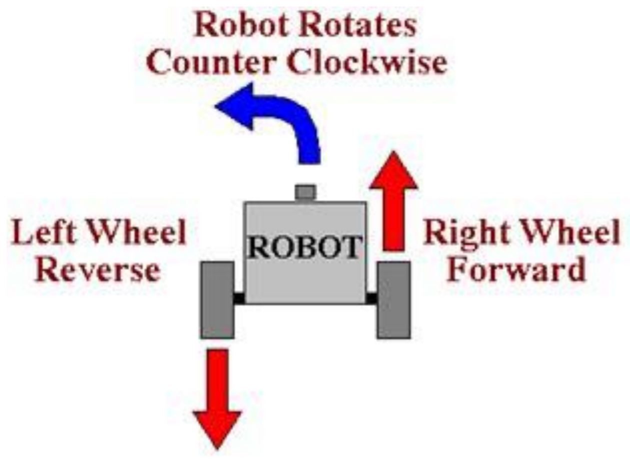


Figure 16: Turning as a result of differential steering (Differential steering, n.d.)

Differential steering works by spinning the wheels on one side of the vehicle forwards and the wheels on the other side of the vehicle backward (Fig.16). This method of steering proved to be extremely successful as contrary to conventional steering, not only was the turning radius of the rover reduced by a significant amount, but the rover was able to rotate within a fixed position. Moreover, another key strength of differential steering is the consistency in the angle the rover turns with multiple trials, improving the accuracy and precision of the rover's turning.

Additionally, another issue that was often observed was the rover deviating away from a straight-line path and following a curved path when in forward motion. After further investigation, it became apparent that it was due to the rover's front wheels being unstable and pivoting slightly. The solution for this issue was implemented by placing blu-tack on the gear system which could lock the front wheels from pivoting, thus providing axial stability. This proved to be very effective as the blu-tack successfully managed to stabilise the front wheels, and also prevented them from pivoting. In practice, this solution resulted in the rover following a straight path and only deviating in a small number of trials. However, this solution did have some shortcomings, at high speeds (60%+), the rover began to deviate, due to the blu-tack not being strong enough to damp the vibrations that cause the pivoting of the wheels. To further improve on this issue, it may be possible to use a stronger adhesive substance that is capable of entirely locking the front wheels from pivoting or implementing two DC motors to control the front two wheels to ensure stability, instead of having an axle.

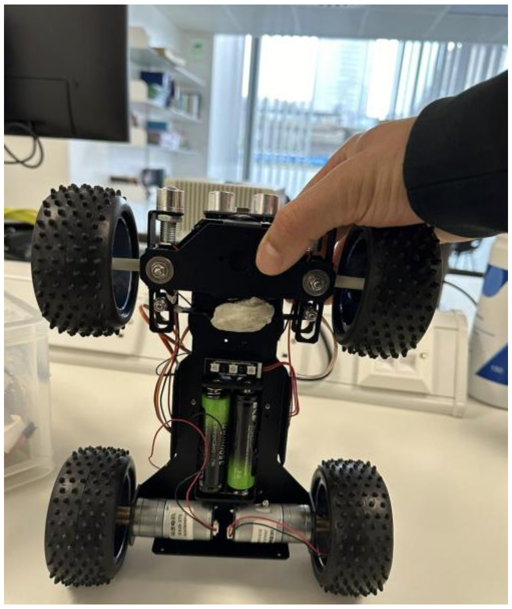


Figure 17: Blu-tack providing axial stability to front wheels.

Moreover, a significant strength of this report lies in the rover's navigation system's ability to navigate any arbitrary path within the external camera's view. This capability owes much to the accurate conversion rate between the camera's pixel plane and the physical plane in centimetres. Additionally, the colour detection algorithm plays a pivotal role in identifying objects in the external camera's image in pixel units. Furthermore, the conversion rate experiment is essential for enabling the rover to traverse any distance to reach the target object. The linear interpolation of the data further facilitated the extraction of a time input value necessary to command the rover to operate long enough to cover the required distance. Overall, the methodologies employed in establishing an effective navigation system are crucial strengths of this report.

V. Conclusion

The seamless integration of object detection technology into the autonomous rover represents a significant milestone in the project. By successfully employing a sophisticated colour recognition algorithm, which mapped the rover's location and identified a specific target area. This enabled precise calculation of the pixel-based distance between the rover and the target, facilitating the determination of a path for navigation.

After numerous measurements and trials, a reliable conversion rate was established for translating pixel distance into physical distance, as well as establishing the rover's speed to be linear. Leveraging these metrics, the rover efficiently followed the calculated route with 16.9% error (on average) in reaching the target location. Despite encountering various challenges, accurate navigation was achieved, fulfilling the primary objective of this project.

Further improvements for the next cohort would be to enhance the rover's capabilities further by integrating additional features such as ultrasonic sensors and the Pi camera, which the group had explored but not implemented. These enhancements would enable the rover to have increased accuracy in its navigation, paving the way for even greater success in future endeavours. Nevertheless, the success of this project showcases the effectiveness of colour detection from an external camera, with an aerial viewpoint, being used for navigation.

VI. Bibliography:

Adeept (2023). *adeept/adeept_picarpro*. [online] GitHub. Available at: https://github.com/adeept/adeept_picarpro [Accessed 3 Apr. 2024].

GM0.org. (n.d.). Odometry. Retrieved from https://gm0.org/en/latest/docs/software/concepts/odometry.html.

Advnture. (2022). How to Triangulate. Retrieved from https://www.advnture.com/how-to/triangulate.

Medium. (2019). Autonomous Mars Rover using Raspberry Pi, Arduino, and Pi Camera. Retrieved from https://medium.com/@rishavrajendra/autonomous-mars-rover-using-raspberry-pi-arduino-and-pi-camera-5b285be452c1.

Raspberry Pi Foundation. (n.d.). Line Following with Raspberry Pi – 4. Retrieved from https://projects.raspberrypi.org/en/projects/rpi-python-line-following/4.

Instructables. (n.d.). OV7670 Camera and Image Sensor With Nano 33 BLE. Retrieved from https://www.instructables.com/OV7670-Camera-and-Image-Sensor-With-Nano-33-BLE/.

Workwithcolor.com. (n.d.). Green Color Hue Range. Retrieved from http://www.workwithcolor.com/green-color-hue-range-01.htm.

ProjectPro.io. (2023). Detect Specific Colors from Image – OpenCV. Retrieved from https://www.projectpro.io/recipes/detect-specific-colors-from-image-opencv.

TechTarget. (n.d.). RGB (red, green and blue). Retrieved from https://www.techtarget.com/whatis/definition/RGB-red-green-and-blue#:~:text=RGB%20(red%2C%20green%20and%20blue)%20refers%20to%20a%20system,red%2C%20green%20and%20blue%20colors.

Differential steering. (n.d.). https://mae.ufl.edu/designlab/Class%20Projects/Background%20Information/Steering.htm

GitHub. “GitHub Copilot · Your AI Pair Programmer.” *GitHub*, 2023, github.com/features/copilot.

Wikipedia Contributors. “Linear Interpolation.” *Wikipedia*, Wikimedia Foundation, 27 Apr. 2019, en.wikipedia.org/wiki/Linear_interpolation.

Upton, Eben and Gareth Halfacree. “Raspberry Pi User Guide.” (2012).

Mehendale, Ninad Dileep. “Interfacing Camera Module OV7670 with Arduino.” *SSRN Electronic Journal* (2022): n. pag.

Waters, Ian. “ASCII Table.” *PowerShell for Beginners* (2021): n. pag.

Xijun, Yan and Jiang Lei. “A Method of Lossy Compression for RGB565 Format True Color Image.” *International Journal of Signal Processing, Image Processing and Pattern Recognition* 8 (2015): 279-288.

GitHub. “Arduino_OV767X Library for Arduino” *GitHub*, 2021, https://github.com/arduino-libraries/Arduino_OV767X/tree/master

Appendix A: Procedure for SSH Connection - Software Configuration

In order to control the movement of the robot and its hardware components, there must be a connection made between a device to code on, and the raspberry pi motherboard. The connection was established through WIFI and Bluetooth via the SSH connection, which stands for Secure Shell Network. The Raspberry Pi motherboard has its own operating system called Raspbian, the Raspberry Pi imager was downloaded on one of the, and then configuration took place. For the configuration to take place, an SD card is necessary to download the Raspbian onto, and then to input on to the motherboard for connection. When launching the app, a window pops up, like the one in Figure 18.

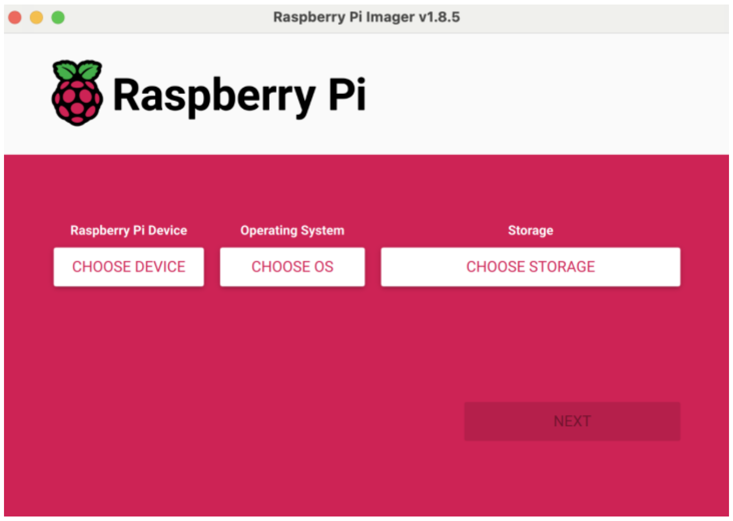


Figure 18: Raspberry Pi Imager - Configure SD Card

The device chosen was the Raspberry Pi 4, and the operating system chosen was the Raspberry Pi OS (32 bit), and the storage would be the micro-SD card used (which is connected to the laptop while downloading and configuring the Raspbian). Afterwards, Raspbian allows you to modify the username: "Mars2024" and the password: "Patrick123". Raspbian also allow wireless LAN configuration, the first attempt was to connect to Eduarom, the university WIFI, however due to various authentication issues, it would only be possible using Mobile Data, where the username was the name of the iPhone's data used, and the corresponding password of the Personal Hotspot ("Patrick123") which were also registered into the Raspberry Pi when configuring the wireless LAN.

After configuring the SD card, it is plugged into the motherboard of the robot vehicle. To establish a connection with the robot and a device, it is done through SSH, the IDLE (integrated development and learning environment) chosen to do all of the coding is Visual Studio Code

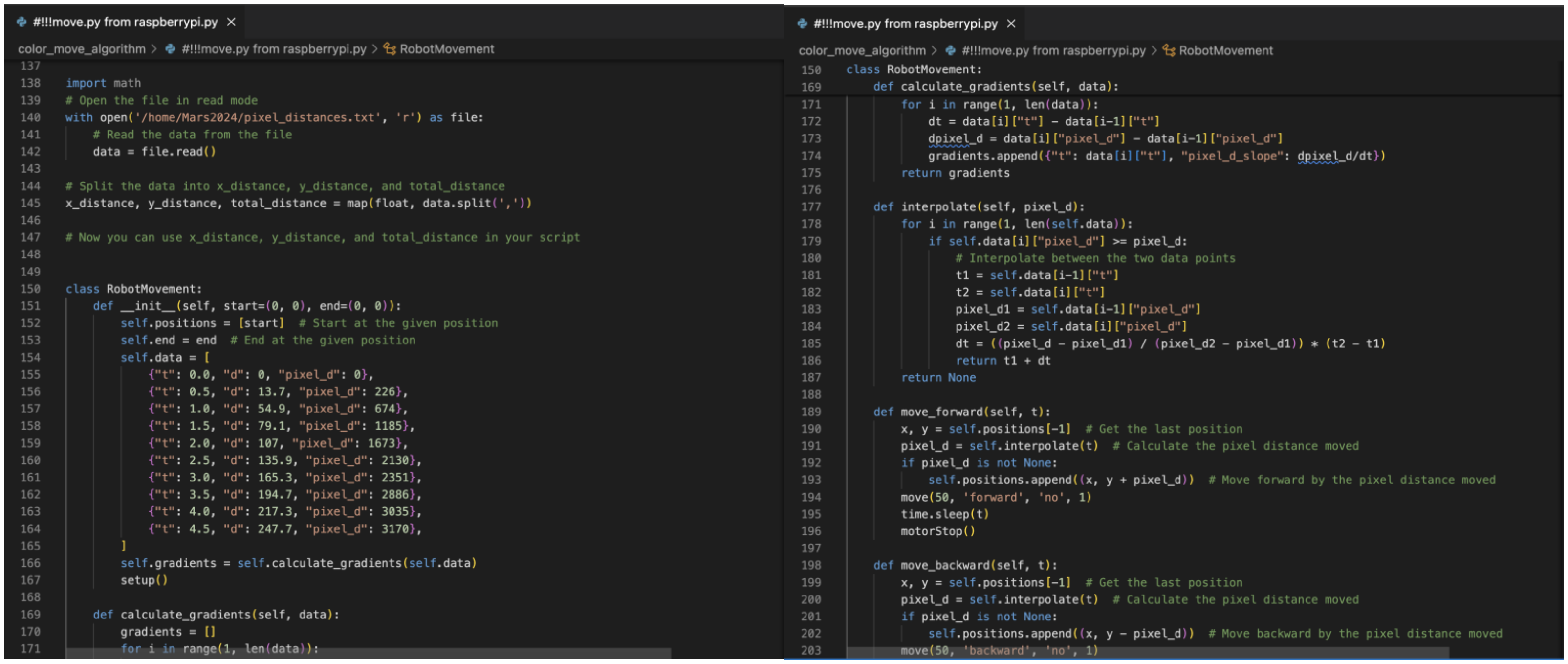
(VScode). In the terminal section of this program, the command “ping raspberrypi.local” is used to check if the Pi board is emitting a WIFI connection. Afterwards, to establish an SSH connection between the Pi and the device (laptop), this command line is used, “ssh username@raspberrypi.local” where the username would be Mars2024 in this case. The computer will request the password for the Pi (“Patrick123”), after inputting that, you will have access to the raspberry pi's directory.

The code for the car’s movement can be found in the “move.py” file, which comes from the manufacturer’s code, found on GitHub [adeept_picarpro](Adeept, 2023), for the functionality of the robotic vehicle. The command line to make a copy of this code onto the Pi was “git clone {link to the GitHub repository}”.

Once all these steps are completed, further development on the code for the external camera navigation method can take place. To take a picture from a bird eye view, and for this picture to be transmitted from the Arduino to the main device (laptop). Afterwards, the laptops run the colour detection algorithm constructed by one of the group’s members and send these coordinates to the raspberry pi.

Appendix B: Aerial View Navigation System Algorithm based off the Colour Detection Procedure

The adeept_picarpro GitHub repository (Adeept, 2023) move.py file, has the commands to allow the robot to move forward, backwards, turn left and right. There are other files containing other functionalities of the robot. The objective is to have the rover travel from any arbitrary point A to point B in a plane. Rewriting the move.py file, to implement this conversion rate was done. In the colour detection algorithm, the robot's, and object's (at the end position) centre point coordinate, as well as the x, y and total distance between them was sent to the Pi from the laptop through SSH. The screenshots below show the algorithm written by one of the group members, along with some explanation of the code.



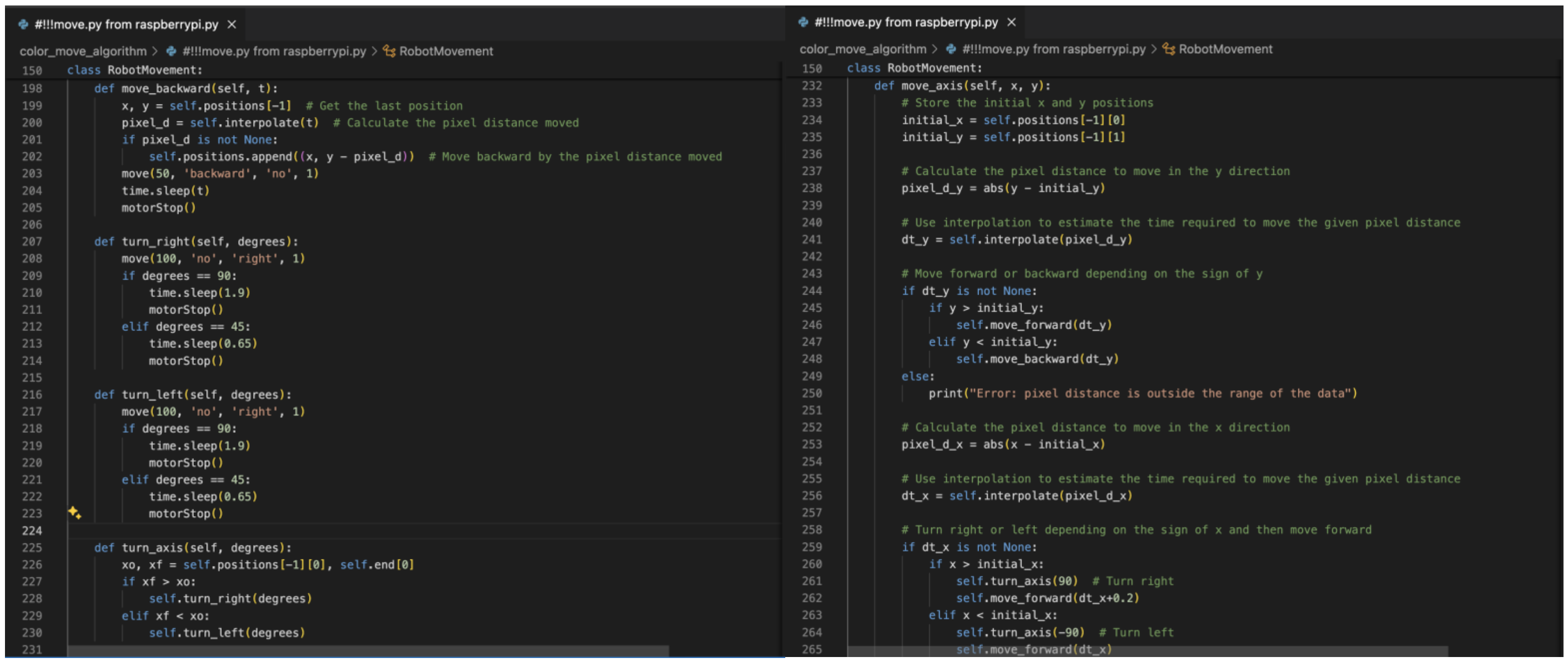
#!!!move.py from raspberrypi.py
color_move_algorithm > #!!!move.py from raspberrypi.py > RobotMovement

```python

import math
# Open the file in read mode
with open('/home/Mars2024/pixel_distances.txt', 'r') as file:
    # Read the data from the file
    data = file.read()

# Split the data into x_distance, y_distance, and total_distance
x_distance, y_distance, total_distance = map(float, data.split(','))

# Now you can use x_distance, y_distance, and total_distance in your script


class RobotMovement:
    def __init__(self, start=(0, 0), end=(0, 0)):
        self.positions = [start]  # Start at the given position
        self.end = end  # End at the given position
        self.data = [
            {"t": 0.0, "d": 0, "pixel_d": 0},
            {"t": 0.5, "d": 13.7, "pixel_d": 226},
            {"t": 1.0, "d": 54.9, "pixel_d": 674},
            {"t": 1.5, "d": 79.1, "pixel_d": 1185},
            {"t": 2.0, "d": 107, "pixel_d": 1673},
            {"t": 2.5, "d": 135.9, "pixel_d": 2130},
            {"t": 3.0, "d": 165.3, "pixel_d": 2351},
            {"t": 3.5, "d": 194.7, "pixel_d": 2886},
            {"t": 4.0, "d": 217.3, "pixel_d": 3035},
            {"t": 4.5, "d": 247.7, "pixel_d": 3170},
        ]
        self.gradients = self.calculate_gradients(self.data)
        setup()

    def calculate_gradients(self, data):
        gradients = []
        for i in range(1, len(data)):
```

```python
class RobotMovement:
    def calculate_gradients(self, data):
        for i in range(1, len(data)):
            dt = data[i]["t"] - data[i-1]["t"]
            dpixel_d = data[i]["pixel_d"] - data[i-1]["pixel_d"]
            gradients.append({"t": data[i]["t"], "pixel_d_slope": dpixel_d/dt})
        return gradients

    def interpolate(self, pixel_d):
        for i in range(1, len(self.data)):
            if self.data[i]["pixel_d"] >= pixel_d:
                # Interpolate between the two data points
                t1 = self.data[i-1]["t"]
                t2 = self.data[i]["t"]
                pixel_d1 = self.data[i-1]["pixel_d"]
                pixel_d2 = self.data[i]["pixel_d"]
                dt = ((pixel_d - pixel_d1) / (pixel_d2 - pixel_d1)) * (t2 - t1)
                return t1 + dt
        return None

    def move_forward(self, t):
        x, y = self.positions[-1]  # Get the last position
        pixel_d = self.interpolate(t)  # Calculate the pixel distance moved
        if pixel_d is not None:
            self.positions.append((x, y + pixel_d))  # Move forward by the pixel distance moved
        move(50, 'forward', 'no', 1)
        time.sleep(t)
        motorStop()

    def move_backward(self, t):
        x, y = self.positions[-1]  # Get the last position
        pixel_d = self.interpolate(t)  # Calculate the pixel distance moved
        if pixel_d is not None:
            self.positions.append((x, y - pixel_d))  # Move backward by the pixel distance moved
        move(50, 'backward', 'no', 1)
```

Figure 19: Part 1 Navigation System Code

For starters the Raspberry Pi reads the pixel coordinates as x, y, and total distance, and converts them to floating point numbers. Then it defines a "RobotMovement" class which represents the movement of the robot from start to end position. The class contains the data found from the experiment for each time interval's distance in centimetres there is its corresponding distance in pixels. The class then calculates the rate over changes of the pixel distance over time. The interpolate function is a method in the class used to estimate the time at which the robot reaches a given pixel distance, based on the data points and gradients. The move forward and move backward functions control the robot's movement based on a given time. They calculate the pixel distance the robot should move in that time, update the robot's position accordingly, and initiate its physical movement. After the robot moves the specified time, the methods stop the robot's motors.


#!!!move.py from raspberrypi.py
color_move_algorithm > #!!!move.py from raspberrypi.py > RobotMovement
#!!!move.py from raspberrypi.py
color_move_algorithm > #!!!move.py from raspberrypi.py > RobotMovement

```python
class RobotMovement:
    def move_backward(self, t):
        x, y = self.positions[-1]  # Get the last position
        pixel_d = self.interpolate(t)  # Calculate the pixel distance moved
        if pixel_d is not None:
            self.positions.append((x, y - pixel_d))  # Move backward by the pixel distance moved
        move(50, 'backward', 'no', 1)
        time.sleep(t)
        motorStop()

    def turn_right(self, degrees):
        move(100, 'no', 'right', 1)
        if degrees == 90:
            time.sleep(1.9)
            motorStop()
        elif degrees == 45:
            time.sleep(0.65)
            motorStop()

    def turn_left(self, degrees):
        move(100, 'no', 'right', 1)
        if degrees == 90:
            time.sleep(1.9)
            motorStop()
        elif degrees == 45:
            time.sleep(0.65)
            motorStop()

    def turn_axis(self, degrees):
        xo, xf = self.positions[-1][0], self.end[0]
        if xf > xo:
            self.turn_right(degrees)
        elif xf < xo:
            self.turn_left(degrees)
```

```python
class RobotMovement:
    def move_axis(self, x, y):
        # Store the initial x and y positions
        initial_x = self.positions[-1][0]
        initial_y = self.positions[-1][1]

        # Calculate the pixel distance to move in the y direction
        pixel_d_y = abs(y - initial_y)

        # Use interpolation to estimate the time required to move the given pixel distance
        dt_y = self.interpolate(pixel_d_y)

        # Move forward or backward depending on the sign of y
        if dt_y is not None:
            if y > initial_y:
                self.move_forward(dt_y)
            elif y < initial_y:
                self.move_backward(dt_y)
        else:
            print("Error: pixel distance is outside the range of the data")

        # Calculate the pixel distance to move in the x direction
        pixel_d_x = abs(x - initial_x)

        # Use interpolation to estimate the time required to move the given pixel distance
        dt_x = self.interpolate(pixel_d_x)

        # Turn right or left depending on the sign of x and then move forward
        if dt_x is not None:
            if x > initial_x:
                self.turn_axis(90)  # Turn right
                self.move_forward(dt_x+0.2)
            elif x < initial_x:
                self.turn_axis(-90)  # Turn left
                self.move_forward(dt_x)
```

Figure 20: Part 2 Navigation System Code

The turn right and turn left functions in the RobotMovement class, initiates a turn for the robot, and pauses for a specific duration based on the desired rotation angle before stopping the motors. It was measured that for a 90-degree turn, the time taken was 1.9 seconds, it is necessary to have both motors at 100% speed going in opposite directions. The turn axis method decides the direction of the turn (right or left) based on the comparison between the robot's current and end x-coordinates.

The move axis function in the RobotMovement class moves the robot to a specified x, y position. It calculates the pixel distances and corresponding times to move in the y and x directions using interpolation. Depending on whether y is positive or negative, it moves the robot forward or backward. Similarly, whether x is positive or negative it turns the robot right or left accordingly, and then moves forward again. If the calculated pixel distance is outside the range of the data, it will print an error message (only if distance to travel in a straight-line is bigger than 2.5m).

Further improvements would be to allow travel larger than 2.5m by repeating the navigation system's moving forward method the number of times necessary to meet that distance. Moreover, constantly live updating pictures from an aerial view of the system, would allow further accuracy in travelling in the x and y coordinates. Additionally, the use of two distinct colours on the robot would allow the orientation of the robot to always be known, so if there were any deviations caused when travelling, the angle to turn the robot back to its normal can be calculated, and the rotation can be executed.

Appendix C: Further Improvement - Orientation of the Rover whilst Differential Steering

One of the issues identified in the path finding algorithm was the lack of orientation detection. The algorithm would only detect the position of the square, and the robot faces would be difficult to determine, with solely the single shape. The solution is to use a two-shape system.

It was considered that if two dots, with distinct colours, were defined on the robot, that relative axis could be created to find the angle, and the direction in which it faces. This way the same colour detection algorithm explained could identify the distinguishable points on the robot, for example front red and back blue. To determine an angle in the xy axes, where the blue dot (the rear of the vehicle) is on the origin, a simple trigonometric calculation can be done to find the angle. The front dot (front of the vehicle) being the furthest away from the origin is important as it dictates which direction the rover is facing.

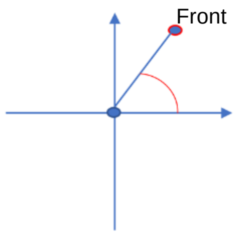


Figure 21: Exemplar Situation of the Rover's Markers

Next, the direction should be addressed, as if the rover were commanded to turn 90°, it may deviate by some error margin. For drastic purposes, let's say it were to be undertaken by 30° or so. The angle required to turn the rover back to its original position facing the y axis, would be restricted to 0° < x < 90°. However, the issue is to identify which direction the rover needs to turn, in respect to what quadrant(Fig.22). An algorithm was considered where the axes would be dissected into four unique quadrants which could be used to identify the direction of the angle.

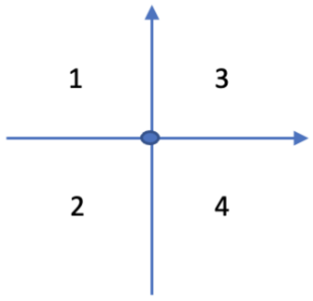
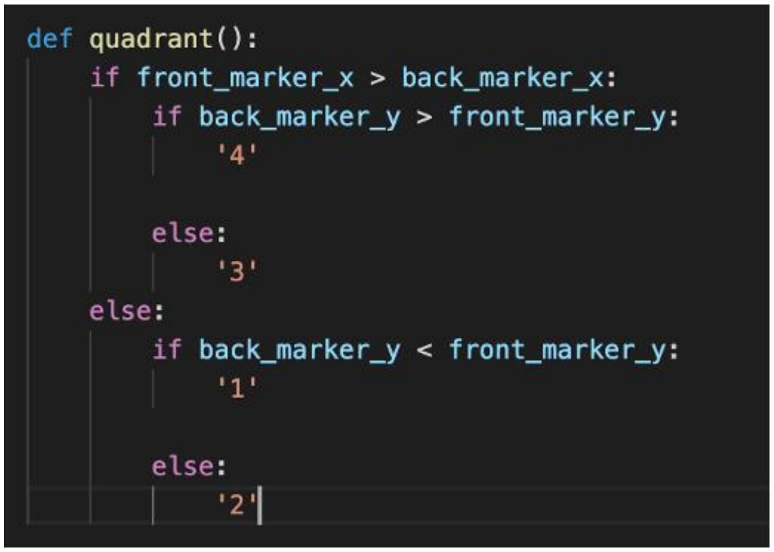


Figure 22: Quadrant Designation

The quadrant could then show which way the robot faces. The direction could then break into two pieces of information, angle and quadrant. With these pieces of information, you can determine a full 360° range of direction for the robot.



```
def quadrant():
    if front_marker_x > back_marker_x:
        if back_marker_y > front_marker_y:
            '4'

        else:
            '3'
    else:
        if back_marker_y < front_marker_y:
            '1'

        else:
            '2'
```

Figure 23: Code to Identify Rover’s Orientation

Implementing this code into the colour detection algorithm, the location of the dots can be identified, further explaining the process to obtain which quadrant the robot is facing, its orientation. For example, if the front marker’s x-direction is larger than the back marker’s x-direction, and the back marker’s y-direction is larger than the front marker’s y-direction, it becomes apparent that the rover orientation is facing the 4th quadrant in Figure 22. This logic is similarly implemented for the different quadrants, and with this information, and the destination quadrant, the angle needed to turn can be determined. After, the algorithm can calculate the time it takes to turn that specific angle into the right quadrant.

The trigonometric formula to calculate the angle is shown below,

$$\tan^{-1}\left(\frac{\|front_y - back_y\|}{\|front_x - back_x\|}\right) \tag{1}$$

This further improvement can be implemented onto the ongoing existing navigation system algorithm to drastically reduce any errors. It should be noted that the external camera can give constant live updates to identify the front and back dots of the rover, or at least before and after turning the robot.