8/19/2022

Real-Time Image Processing for Autonomous Guidance of Maze Following Robot

EE990 Individual MSc Final Report

Aqsa Khan

202175534

Dr. Graeme M West

MSc in Wind Energy Systems

The Department of Electronic and Electrical Engineering

The University of Strathclyde

Declaration of Authorship

I, **Aqsa Khan**, hereby declare that this work has not been submitted for any other degree/course at this University or any other institution and that, except where reference is made to the work of other authors, the material presented is original and entirely the result of my own work at the University of Strathclyde under the supervision of **Dr. Graeme M West.**

Signed: 

Date: **06.08.2022**

Abstract

The project includes image processing to guide the robot independently inside the maze. The project's working principle comprises three parts. i) Digital image processing ii) Wall/feature detection algorithm iii) Logics for the movements of the robot. Python is used for the coding in the project. A maze is created so that the images can be taken from the Raspberry Pi camera mounted on the robot. Image processing steps include image rotation, image conversion to grayscale, image blurring, canny edge detection, and contouring. The wall detection algorithm is designed based on the wall of the maze created, with the help of OpenCV functions. Once the robot takes a picture with a front view and the canny edge detection operator is performed, the algorithm starts working depending on the number of edges the image has. The wall detection algorithm works on the canny image and cropping the image to only show the edges of the front wall. It draws the colored contours on the edges of the wall making it more obvious for the robot. The logic for movements combines forward, left, and right movements. The robot decides to move either left or right based on the corner edge of the maze in the image taken by it. The code is the integration of image processing, wall detection, and logic for movements. The robot takes the images, keeps moving forward, decides on where to take the turn, and detects the wall ahead of it using the algorithm.

Table of Contents

List of Figures

Chapter 1

1 Introduction

1.1 Project Outline

The title of the project signifies that we aim to design an autonomous robot that is nothing but an intelligent machine that knows how to perform specific tasks without having direct control by humans in real-time. Autonomous robots offer quite significant applications and their implementations in the fields like locomotion (flying, crawling, and wheeled robots), navigation, mapping, orientation & object detection [1]. Moreover, autonomous robots have also shown their usage in cleaning as autonomous vacuums or vacuum cleaners [2]. One of the major applications of autonomous robots can be seen as autonomous vehicles like self-driving cars whose ultimate goal is to cultivate a technology that could enhance road efficiency and lessen traffic accidents [3].

In this project, the autonomous robot is being guided to move inside the maze by detecting walls that come in front of it and then making either a right or left turn accordingly. First, the robot takes the images of the maze in real-time from the camera mounted on it. There will be multiple images of the maze taken by the robot's camera. For this purpose, Raspberry Pi is to be connected to the motor using the bus cable. By using python code, we can enable the robot's camera and take pictures of the maze as per the requirements. The pictures of the maze taken by the robot would be generic digital images similar to the images taken by a normal camera. Then each image is gone through image processing which is a method of performing any particular operation on an image so that we could get an enhanced yet desirable image as per our requirements by extracting fruitful information from it. However, image processing is a tool that helps us fetch the information or data from the image that could help our robot to perform its operations autonomously. Therefore, the main objective of the robot is to follow the algorithm that is written for it to detect walls. The Wall detection algorithm combines multiple steps in which edge detection is one of the vital parts of it. Using image processing, the image of the maze is taken and converted into a grayscale image which can later be used for edge detection.

1.2 Objectives

The main objectives that we aim to achieve in this project include image processing in real-time so that the robot could behave like an intelligent machine without straightforward human control and make the decision to travel in the maze by avoiding any obstacle in the form of walls. The first and foremost aim to achieve in the project is to capture representative images of the maze for the robot. This is an initial step as for the robot to start following the maze, it must have the images stored in it to make the decision accordingly. The objective here is to first take the images from the robot's camera and then do image processing with it. Once we are successful in taking the representative images from the robot's camera it will be easier to implement an image processing tool in it. Ideally, the robot is supposed to take real-time images of the maze from its camera, and then image processing is done for every image taken by it. The main advantage of real-time image processing helps the robot navigate through the maze without getting into the phase of preprocessing. The task of preprocessing would have involved taking the images of the maze beforehand and then with image processing, the path would be formed for the robot to follow. However, in real-time image processing, the robot takes the images of the maze in real-time, does image processing, and follow the maze. The other main goal to achieve in this project would be an algorithm for the robot for wall detection. The whole concept of image processing in the project is linked with writing the wall detection algorithm. Previously, maze-solving robots have been designed like line following robots or robots that use maze-solving algorithms. In line following robots, a path/line is already defined in the maze for the robot, and sensors are mounted on it and with the help of the controller, it solves the maze. On the other hand, one of the most popular maze-solving algorithms is used called the wall-follower algorithm in which a robot focuses on either the right wall (Right-Hand Rule) or the left wall (Left-Hand Rule). Wall-follower algorithm also requires sensors in the structure of the robot to detect the wall by receiving the information from the external environment [4]. However, in this project, we focus on writing an algorithm for the robot to detect features (walls) without the help of any controller or sensor. Our goal is to design a robot that takes the images and by using Raspberry Pi, real-time image processing is done and the robot would detect the wall ahead of it by the algorithm written for it rather than solving the whole maze. The algorithm includes an edge detection mechanism for the images of the maze, by doing so, this edge detection tool enhances the edges of the maze for the robot to recognize the wall ahead of it. Writing a wall detection algorithm could be a daunting task as it defines the

precision of the robot to take left, right, forward and backward movements. Once an algorithm is successfully written for the robot to detect walls, it is required to define the logic for the robot's movement. These logics are defined in the way to guide the robot to make a turn by first evaluating a wall and since it would be capturing the images in real-time as it moves inside the maze, it would take turns accordingly. Therefore, by taking real-time images, image processing, and following the wall detection algorithm, the robot would be able to follow the maze. The last objective would be the implementation of this entire coding on the robot.

1.3 Motivation / Background

As previously mentioned, there have been numerous autonomous maze-solving / maze-following robots that gave the motivation for this project. Maze-solving robot, also known as a micro-mouse is based on the function to find the route in the maze itself as its coding is done in a way that allows it to interpret the path on its own without taking any help or assistance. This autonomous maze-solving robot is already serving mankind in the applications like pipe inspection, bomb disposal, material handling, carrying goods, and warehouse management. This kind of maze-solving robot is quite unlike the line-following robot in which the robot has to follow a predefined path. It works on the logic done in Arduino code and ultrasonic sensors are used to help navigate the robot in the maze [5]. Another maze solver robot has been implemented using Artificial intelligence with Arduino which operates on two basic principles i.e. the robot is responsible to locate its way out of the maze no matter where it is put inside the maze and as soon as a robot finds its path, it ought to enhance the solution by finding the shortest path from start to end. This maze-solving robot is designed to first find the walls of the maze and then follow them to move inside it by implementing either left or right hand on the wall algorithm at Arduino code with the help of multiple sensors. First, the algorithm allows the robot to follow the wall by the left/right-hand rule to complete the route in the maze, and next time by using artificial intelligence, the robot would be able to take the shortest path [6]. Then there comes an autonomous robot that uses image processing and a pathfinding algorithm to find the best route for maze solving. In this method, a top view image of the maze is taken, and then preprocessing is done to find the path from start to end point. The image processing in this method involves preprocessing of the image which gives a virtual line (well-defined path in the image) for the robot to follow. After that, an algorithm is designed for the robot to accomplish the aim of traversing [7].

So far, we have seen autonomous robots that have been built to solve the maze using sensors, maze solving algorithms, Arduino, or preprocessing of the image. One of the many reasons that became the motivation of this project was not just aiming to solve the maze using any pre-defined or virtual path, but it's rather focused on feature/wall detection without the help of sensors. Furthermore, in this project, Raspberry Pi is being used for real-time image processing. The purpose of real-time image processing is quite different than preprocessing in which a robot is already defined with a path to escape through a maze. In this project, we are motivated to write a wall detection algorithm with the help of image processing. Not only is image processing responsible for providing real-time images of the robot's camera but it is also involved in writing the logic for the robot to detect walls in front of it and move in a maze without hitting them or any edges.

Chapter 2

2 Theory of Practical Work

2.1 Tools

2.1.1 Raspberry-Pi 3 Model B+

Raspberry Pi is nothing but a small-sized computer having a single board that runs Linux on it. It is quite a cheap and affordable device to help programmers operate electronic devices, physical computing, and the Internet of Things (IoT). Raspberry Pi gets easily plugged into a desktop monitor or a normal TV and can be operated using a computer's keyboard and mouse. Having the ability to interact with the outside world, Raspberry Pi is capable of doing everything one would expect from a normal computer [8]. It allows its users to do programming in Python to control a robot, helps students to run their hardware projects, is used as a gaming server, helps take pictures, makes videos, etc. [9]. In this project, Raspberry Pi 3 Model B+ has been used which is a 3rd generation single-board computer with the following specifications:

- 1.4GHz 64-bit quad-core processor
- Wireless LAN
- Bluetooth 4.2
- 40-pin GPIO (General Purpose Input Output)
- Camera Serial Interface (CSI), Camera port for connecting a Raspberry Pi camera
- Display Serial Interface (DSI), Display port for connecting Raspberry Pi display screen
- HDMI port
- 4 USB 2.0 ports
- 5V / 2.5A DC power input [10]

2.1.2 Camera

To take the digital images of the maze, a Raspberry Pi camera is used which has a Raspberry Pi operating system, a derivative of Debian. The Raspberry camera board allows direct connection to

the Raspberry Pi. It comes with a 15cm flat strip ribbon cable with a 15-pin MIPI CSI connector that goes into the CSI port on the Raspberry Pi, specifically made for interfacing with the cameras. The camera module has 5MP (Million Pixels) and state image resolution is 2592 x 1944 and gives an HD (High Definition) image i.e. 1080p.

2.1.3 Anaconda

Anaconda is software available to allow the programmers to work in different environments where it is easier to write codes in Python that's why it is commonly known as a Python distributor. It is free software that comes with a Python interpreter, various Python packages, and editors. After installing Anaconda, we gain access to environments called Integrated Development Environments (IDEs) which are capable of giving comfort in creating codes for different projects. The useful functions offered by IDEs, such as the ability to develop, edit, and debug codes, view and examine data, store variables, and present results, are always in demand. Before Anaconda, CPython served as the Python distribution tool for programmers. However, Anaconda is a relatively straightforward Graphical User Interface (GUI) that offers useful, well-sustained libraries and IDEs. Important libraries like Scipy, Numpy, and Matplotlib are all present in Anaconda, along with the open-source python program called Biopython. With a single installation, Anaconda eliminates the need for several Python installations with various libraries, functions, and IDEs. For coding in Python, Anaconda offers a variety of useful IDEs that automatically come with the installation of Anaconda. The commonly used IDEs available with Anaconda are as follows.

2.1.3.1 Jupyter Notebook

The Jupyter notebook is an interactive open-source environment that allows users to write the number of codes in the notebook and run individual blocks of code. To examine the outcomes for each individual block, they can use this functionality to execute their code in chunks. The web-based environment enables programmers to efficiently employ a range of texts, writing code and execution of results in one place. Moreover, it includes visualizations, simulations, figures, equations, and plots all in a single notebook. Because of its web-based feature, it becomes easier to share the Jupyter notebooks with others in an organized way. Jupyter notebooks may be edited in the browser and are pretty similar to Word documents. Although they have the ".ipynb" file extension, notebooks can also be downloaded in other formats, such as PDF and HTML.

2.1.3.2 Spyder

Spyder, Scientific Python Development Environment, is an IDE that allows a programmer to interact with code very efficiently as it offers advanced editing, introspection features, and vibrant debugging and testing of code.

- **Advance editing** helps in following the correct syntax and indentation in the code by highlighting and predicting keywords
- The **debugging** using this IDE gets easier as it offers full control after each line so that a programmer can easily know the line that has a bug in it

Furthermore, Spyder also has a window to display the variables that are initialized in the code this is why it gets easier to debug and the Programmer is not required to go through the hassle of printing every variable to debug it. Lastly, Spyder is extremely customizable this would help programmers select settings that are easy on the eyes as they spend several hours in front of the screen. [11] & [12]

2.1.4 Python

Python is an open source programming language under an OSI-approved license this makes python developers to easily access the source code and amend it according to their needs which results in pre-developed libraries by many developers which can be easily used by programmers to help them in completing the project [13]. Multiple tasks can be done using Python language some of them are listed below:

- Creating a machine learning model
- Image processing
- NLP
- Creating a dashboard
- Making scripts
- Server-side programming

Image processing was one of the vital steps that had to be done for the project, and there were two options available: MATLAB and Python. Since, image processing helps the programmers do a lot of tasks like image manipulation by cropping, flipping the images, image classification, etc. In the project, image processing was meant to use for rotating the images taken off the robot's camera

and later for image segmentation, feature extraction, image recognition, and restoration. Therefore, Python was a better choice to perform these tasks as it comes with a variety of image processing tools with their free availability. Python appears to be a simpler language for the user to translate robot-related ideas into code. Since image processing is a subset of machine learning, python is the most widely used, developed, and well-supported programming language in machine learning. Python provided us with numerous pros like the comfort of coding. Python's feature would make it easier to concentrate on designing rather than coding in even complex settings. In addition to being simple to code, it is also free, unlike MATLAB. Because it is open source, programmers can distribute their work for free on many websites. Python distinguishes out from the competition for programmers working on image processing because the majority of image processing could be handled by the python libraries [14].

2.1.4.1 Libraries

When it comes to programming using Python, there are quite useful libraries available to perform which are specifically designed to perform certain tasks. Similarly, numerous libraries come in handy for image processing as well which could make a programmer's life easier. The Python libraries that are used in the project for image manipulation are as follows:

2.1.4.1.1 OpenCV

Open CV (Open source Computer Vision) is the library that is widely used on various platforms to process, enhance or reconstruct an image as well as for different computer vision applications. The project for the Open CV library was first initiated in early 1999 by Intel researchers for the advancement of CPU-intensive applications. The library was originally designed to contain functions that programmers could use to help them with real-time computer vision. Open CV comes with a feature of GPU acceleration for real-time operations that was initially started in 2011 whose purpose is to enhance the speed of the processing of an image, bluffing it as real-time processing. Many useful implementations can be achieved using the Open CV library, some of them are listed below:

- Detection of a face
- Detection of edges
- Adding filters
- Removing watermark from an image

- Converting images to greyscale or binary
- Detection of a certain color in an image
- Removing background noise in an image
- Converting a regular image into a cartoon

In this project, the Open CV library is used to carry out the processing of the captured images of the maze using a digital Raspberry Pi camera. This processing mainly includes finding the edges of a wall which would help later to guide the robot autonomously to avoid collision and do the movements accordingly.

2.1.4.1.2 NumPy (Numeric Python)

NumPy is one of Python's most significant libraries which is mostly used in programming to process arrays. The primary components of the NumPy library are multidimensional array objects and a collection of functions for handling arrays. Although lists in Python can already be used as arrays their processing speed is relatively poor. However, NumPy comes with an array object called 'ndarray' (N-dimensional array) which is over 50 times quicker than Python's default lists and fills this gap. A processed image is essentially a NumPy array with pixels and data points. The pixel values of an image can be changed by using fundamental operations of NumPy such as indexing, slicing, and masking. A Python package called Skimage loads the image, which Matplotlib may then display [15] & [16]. NumPy is utilized in the project to manipulate numerous arrays and various matrices of numerical data using mathematical and logical operations on arrays. NumPy is used in conjunction with other Python libraries like SciPy (Scientific Python) and Matplotlib, a plotting library. MATLAB could have been considered for image processing but this combination is frequently employed in place of MATLAB. Moreover, another benefit of NumPy is that it is open source and so publicly available.

2.1.4.1.3 Matplotlib

The Python module Matplotlib is used to plot graphs, making it a tool for data visualization. Like Numpy, Matplotlib is a free and open-source program. Matplotlib is being utilized in the project for static visualization however, it is a complete library for both static and interactive visualizations. Python-based Matplotlib can be used to generate 2D graphs from data contained in arrays. It can be used in conjunction with NumPy, a Python extension for numerical mathematics [17].

2.2 Techniques

2.2.1 Image Processing

As previously discussed, image processing is a technique in which an image is taken from a digital camera or through scanning and then can be reconstructed, manipulated, and enhanced into a desirable form. In this project, its main task is to compress or analyze the taken image and collect the information from it so that it can be later used for writing an algorithm for wall detection. Image rotation comes first in the image processing phase, then the image is converted to grayscale, blurred to minimize noise, and finally edge detection, which is the cornerstone of information extraction from the images. Later, contours are found and then drawn in the image.

2.2.1.1 Image Rotation

One of the image processing procedures is called "image rotation" which rotates an image by a predetermined amount of degrees around its center. A geometric change known as image rotation can be carried out using either a forward transformation/mapping or an inverse transformation/mapping. In the project, the images that were taken by the robot's camera were upside down and therefore, the first step in image processing was to rotate the taken images. There are several functions available to rotate the images such as PillowIt is a Python image processing package that rotates the images counterclockwise by utilizing inverse transformation with the number of degrees as a parameter. Utilizing the image processing library "imutils" with OpenCV is a different approach that also rotates the image by a defined angle [18]. However, the images taken by the robot's camera were rotated using the OpenCV function. There are two functions in OpenCV used for image rotation, one is cv2.rotate( ) and the other is cv2.getRotationMatrix2D( ). Cv2.rotate is a function that accepts two parameters, the image that is to be rotated and the kind of rotation to be applied. The image can only be rotated with this function in multiples of 90 degrees. For example, it rotates the image in a clockwise direction by 90 degrees, counter-clockwise by 90 degrees, and in a clockwise direction by 180 degrees. In contrast, the function cv2.getRotationMatrix2D( ) rotates the image by angles other than 90 or multiples of 90 degrees [19]. Therefore, we used cv2.rotate( ) as we only needed to rotate the images by 180 degrees.

2.2.1.2 Image into Grayscale

The conversion of colorful or RGB images into grayscale in image processing is advantageous because it makes it simpler for programmers to work with the grayscale image and requires less

computing power. Because colorful photos contain extraneous information that increases the quantity of data needed to attain high performance. Therefore, conversion to grayscale is initiated by simplifying the algorithm and speeding up the processing time. The project has implemented the conversion of the BGR(Blue, Green, and Red) image to grayscale. In contrast to grayscale, which only permits a single channel of processing, BGR images were relatively challenging to handle. Grayscale conversion had to be done before extracting edges since edge detection had to be done afterward for the wall detection algorithm, and the BGR picture information wouldn't let the edge detection identify all the essential edges in the image. Writing an algorithm is already a daunting task to do, and edge detection on colored images would have required extensive work. Our focus was to minimize the complexity of the code, debugging, and extra support [20] & [21]. The OpenCV function cvtColor( ) is used to convert an image from one color space to another. This function contains two input arguments, the first input argument receives the rotated image of the maze. However, the code for color space conversion is sent as the second input argument. We used COLOR_BGR2GRAY since we needed to convert the BGR color space to grayscale in our situation [22].

2.2.1.3 Image blurring (Noise Reduction)

One of the important preprocessing procedures in image processing is blurring the images since the captured images may have noises that need to be removed through smoothing and blurring. Blurring is required to lessen the high-frequency noise and edges before applying edge detection. The pixels that give an image its brightness and color fluctuation are the sources of noise; blurring removes those pixels, leaving them hazy and fuzzy. Many blurring operations, including basic blurring, weighted gaussian blurring, median blurring, and bilateral blurring, are supported by OpenCV. However, to reduce noise, the image soothing method that we employed is Gaussian blurring and the blurring function it uses is GaussianBlur( ). Sharp edges in the image are smoothed while excessive blurring is reduced using this technique for image soothing. A low-pass filter is used for the Gaussian blurring to take out the high-frequency components. This function is used to remove noise that roughly follows Gaussian distribution, which causes the final image to become less and less artificially blurred. Although, Gaussian blur is slightly slower than the other blurring methods of OpenCV but it seemed to be the better option as we're working on natural images [23] & [24] & [25] & [26].

This function's syntax consists of various arguments, where the first argument 'src' describes the source or input image, and the second argument describes the size of the kernel. The size is indicated as [height width] and both height and width may have distinct values but should be odd. The third argument 'sigmaX' describes the gaussian kernel standard deviation along the horizontal direction / X-axis. The fourth argument describes the destination 'dst' or output image, and 'SigmaY' is the fifth argument kernel standard deviation along the vertical axis / Y-axis. The last argument 'borderType' signifies the boundaries of the image with possible values such as borderType = cv.BORDER_REPLICATE / cv.BORDER_CONSTANT / cv_REFLECT, cv.WRAP / cv.TRANSPARENT / cv.DEFAULT and cv.ISOLATED [26] & [27].

2.2.1.4 Edge detection

Edge detection is one of the primary tools of image processing in which the amount of data or pixels is reduced to process and conserves the structure of the image. In edge detection, the regions in the image where there are differences in the intensities of the pixel are identified as edges by using the matrix method. It is also the initial step in writing the algorithm for wall detection. Edge detection itself is an algorithm containing the following steps:

a) **Filtration:** Digital images taken by the camera need to be filtered out as they might contain various oddities such as noise that might reduce the performance of an edge detector. However, filtering needs to be done in such a way that an image doesn't lose the strength of the edges. In the images of the maze, there had been quite a few shadows of the light which were filtered out while doing edge detection.

b) **Enhancement:** Before getting into the final step of edge detection, it is important to recognize the change in intensities of the pixel in the taken image. Therefore, the enhancement mode focuses on indicating how brighter or lighter the points are than the other in an image. Since the blocks of the maze were white-colored, there had been a change in intensity between the front block and the blocks on the sides of it. Therefore, enhancement was needed to emphasize the pixels in the image with a major change in local intensity.

c) **Detection:** To detect the edges, points with strong edges are to be considered. It may be the case that an image has a few points that are not qualified as edges for the desired

application [28]. It does a thresholding operation by extracting the edge pixels, qualified to be reserved and the rest should be rejected as noise.

There are multiple functions available to carry out image processing for edge detection, each with its pros and cons. The three widely used edge detection functions in Open CV are as follows:

- Sobel
- Canny
- Laplacian

2.2.1.4.1 Sobel Edge Detector

The Sobel edge detector is a gradient-based method to detect edges using first-order derivatives. It measures the gradient of an image which is a directional change in the image intensity to highlight the areas of maximum rate of change of image intensity values in the spatial domain that relates to edges. It determines the first derivative of the image in 2D by taking the derivative of both X and Y axes. Sobel edge detector feature uses a pair of 3x3 convolution kernels for the derivatives, one for Y (vertical) axis and the other for X (horizontal) axis. However, convolution is a method of multiplying two arrays with different sizes but the same dimensions to produce a resultant array having a similar dimension to the two arrays. Whereas, kernel is a matrix used in image convolution, usually a small matrix with different sizes and configuration of numbers [29] & [30]. In gradient-based edge detection methods, when there is a peak or steep, it is declared as an edge pixel but the drawback of such techniques is that it can be difficult to determine what is a peak and what isn't that's why it's hard to discriminate between a peak (edge) and noise [31].

2.2.1.4.2 Laplacian Edge Detector

Laplacian is a second derivative operator that is used to find edges in the image. Laplacian calculates second-order derivatives, which distinguishes it from other edge detector operators, and unlike Sobel, it uses a single kernel for the derivative. The other different feature of laplacian that sets it apart from the other edge detection operators is that the edges it highlights are random, either inward or outward with no particular direction. Laplacian comes with the idea of zero-crossing which represents a point of the edge location where a picture's intensity fluctuates quickly. However, the one disadvantage of using laplacian is that since it takes a second-order derivative, it becomes extremely sensitive to noise [31] & [32].

After measuring the pros and cons of all three canny was decided as the best algorithm for carrying out edge detection in this project.

2.2.1.4.3 Comparison of Edge detection methods

As discussed earlier, it is evident that edges have a higher pixel intensity than the other points in its surrounding. Generally, edge detection techniques are classified into two categories:

a) Gradient-based edge detectors
b) Laplacian edge detectors

Gradient-based operators take the first derivative of the image with respect to time 't', and the first derivative is at a maximum at the center of the edge. Then the threshold value is set to compare with the gradient value, if the gradient value exceeds the threshold value, an edge is detected. However, if the first derivative is at a maximum then the second derivative would be zero. Therefore, the laplacian operator locates the edges by finding the zero crossings in the second derivative [33]. Examples of gradient-based edge detectors are Sobel, Roberts, Prewitt, and Canny, whereas Marr-Hildreth / Laplacian of Gaussian (LoG) and Zero Crossing (ZC) are laplacian-based operators. All three renowned edge detectors were examined prior to choosing canny for the project's edge detection, and it was determined that canny produced the best results despite time-consuming and complex computations. Given that the Sobel and laplacian are both quite sensitive to noise, the canny operator detects weak edges even in noise conditions, better in calculating error rate and improving signal to noise ratio. Canny is also able to detect continuous, thin edges as well as edges in the black areas that the other edge detection operators were unable to pick up on [34] & [35] & [36]. Normally the gradient-based operators tend to detect thick edges but due to non-maximal suppression, canny is capable to detect thin and smooth edges in the image. Another big difference between commonly used Sobel and canny operators is that Sobel's final output is a blurred image whereas, canny gives the clarity of the image by producing an output with sharp edges [37].

2.2.1.4.4 Canny Edge Detector

For edge detection, the canny operator was chosen as it is the most commonly used and seemed to be the most effective method for detecting walls of the maze among all available edge detection operators. Canny edge detector maximizes the probability of detecting the actual edges and minimizes the chances of highlighting the non-edgy points in the image thus reducing the error

rate. Canny edge detection is a technique used to filter the edges in the image. It also ensures localization by detecting the edge closer to the actual edge. Lastly, it is responsible to generate a single edge detection response and avoid creating multiple responses to a single edge point. Moreover, it is an edge detection method consisting of multiple algorithms as follows:

a) **Smoothing & Noise Removal:** The initial stage in edge detection is to eliminate undesired noise from the original image; as a result, blurring is used to filter out the noise and provide a smooth output. Canny edge detector accomplishes this using a Gaussian filter.

b) **Finding Gradients of the Image**: The next step is to find the gradients of the image which is done by taking the first derivative with the help of the Sobel kernel. This phase involves taking the first derivative for computing vertical and horizontal components as well as the magnitude and direction of the gradient. The direction of the gradient is perpendicular to the edges and it is rounded to a single one of the four angles demonstrating directions like horizontal, vertical, and two diagonals.

c) **Non-Maximal Suppression:** Once getting the magnitude and direction of the gradient, the next step is to get rid of the thick edges in the source image, and this process of thinning the edges can be done by non-maximal suppression. Moreover, the local maxima are located in the direction of the gradient and designated as edges, while the others are suppressed, lowering the number of false edges [37] & [38].

d) **Double Thresholding:** This step is called double thresholding as two threshold values are taken in it to decide what actual edges are and what needs to be discarded. The two threshold values are called T1 = Max. Value and T2 = Min. value respectively. The pixel intensity values of the grayscale are then compared with the threshold values. If the values of pixels are greater than T1 then the pixels are declared to be the strong/sure edges and if the values of gray scale are less than T2 then pixels are weak / non-edges. If the values of the pixels lie between T1 & T2 then the outcome depends on the neighboring pixels.

e) **Hysteresis Edge Tracking:** Once it is decided what strong and weak edges are, strong/sure edges are considered in the final output image, and weak/uncertain edges that are not linked with the strong edges are rejected. However, weak edges that are connected to the strong edges are included in the final output image. Hence, it is very important to choose threshold values to get accurate results [37] & [39].

2.2.1.5 Finding Contours

Finding the contours is the next approach used in the image processing stage after the application of the canny edge detector. The points and the boundaries of an object are joined to create the curves or contours. To find contours, an OpenCV function called 'findContours( )' is used that searches for contours of the corners. It is advised to utilize this function in binary images that's why it becomes easier to locate contours after the application of canny edge detection on the images. This function's target contours should be white, while the background should be black [40] & [41].

2.2.1.6 Drawing Contours

After extracting the contours by using findContours( ) function, the next OpenCV function 'drawContours( )' is used to draw the contours. With the given boundary points, it is possible to draw the contours of any shape or color. In the project, the main goal was to draw the contours across the boundary points of the front wall.

2.2.2 Methods for Wall Detection

Following edge detection, the algorithm would then investigate several methods that could be used on the image to assist the robot in detecting walls. The image after edge detection would only highlight the edges of the maze, now wall discrimination is needed so that robot doesn't collide with every wall that comes ahead of it and takes a turn every time it detects the wall. The following techniques have been considered to apply one by one for wall detection. One method could be coloring the wall that comes in front of the robot with a different color. The robot can then more easily recognize the wall based on the difference in coloration between the front and side walls.

Another approach would be to draw colored contours, which would simply paint the boundaries of the wall and highlight them so that the robot can only see the colored edges of the wall. Contours are simply the outlines drawn along the boundary of the edges in the image with some intensity or color. The colored boundaries help in object recognition and in this scenario, the colored contours make wall identification for the robot easier. Contours tend to be an important tool and it gives accurate results when the image is binary that's why it is necessary to use contours after edge detection. Since the canny operator is already applied to the image, the contour function is expected

to generate precise output. Using the function "cv.drawContours," contours can be used to draw any shape once the boundary points are known [40].

Numerous feature detection techniques are available in OpenCV for examining significant characteristics in the image. A feature in the image can be anything like edges, corners, spots, points, etc. The black and white version of the image should be used for the feature detection method because it makes the features easier to see for the algorithms. Haris corner detection is one of the feature detection algorithms used to detect the corners of the image. It locates the corners of the image using a slider box, and after applying a threshold, the corners are indicated. Another feature detection algorithm is FAST (Features from Accelerated Segment Test) algorithm which is fast enough to use with real-time devices like mobile phones and cameras because it has a very fast computing time. It can detect corners as well as blobs that's why also considered a blob detector [42]. FAST efficiently detects features more than other feature detection algorithms like Haris corner, SIFT, and SURF and is thus suitable for resource-intensive applications like real-time image processing in the project.

Chapter 3

3 Experimental & Modelling Procedure

3.1 Prerequisite steps of working

Testing the functionality of the Raspberry Pi that was already installed on the robot was the first step in starting the project's operation. The Raspberry Pi-3 model B+ has been utilized in the project to carry out all necessary robot-related tasks, including taking the images of the maze and controlling the movement of the robot. Therefore, the whole setup of Raspberry Pi was done by integrating it with the local PC. Putty software was installed which is a software application that allows us to access the Raspberry Pi command line interface by running it from our laptop or desktop. By installing Putty, a connection has been built between the windows operating system of the laptop and Raspberry pi which is a remote device. After the quick installation of Putty on the laptop, a tested file was transferred from the laptop to Raspberry Pi by sending commands to it [43]. After the Raspberry Pi and laptop were successfully integrated, a Python setup was completed because Python was chosen for image processing in the project. To build Python code for image processing and a wall recognition technique, we downloaded Anaconda and a Jupyter notebook. Since completing the entire maze is not the project's major goal, feature detection, a wall of the maze, is. Therefore, a maze was designed so that we could take pictures of the first wall from the entrance of the maze. The following step was to capture the appropriate images after establishing a desirable maze. Real-time picture processing is required, as the project's title suggests. Before that, though, a set of representative images had to be taken to serve as the basis for image processing coding in Python that would later enable the robot to take and interpret pictures in real-time. The robot already had a Raspberry Pi camera placed on it so that it could capture images of the maze's front wall. To capture the photos from the Raspberry Pi camera, Python code was created. For image processing and writing wall identification algorithm, many images were obtained. Since the whole python code was to be implemented on the robot, it was important to test the movements of the robot. Therefore, the only hardware part of the project is the robot’s movement inside the maze with the help of real-time image processing and feature detection. Before starting the software part of the project, the robot’s movement with the help of

Raspberry Pi was tested by Python code. The Left and right movements of the robot were effectively achieved by the coding.

The software part of the project comprises two major steps. The first one is image processing and the second one is wall detection algorithm. The coding for image processing in Python started by first importing the libraries. Four libraries that were imported are as follows:

a) Matplotlib.pylab
b) Numpy
c) CV2
d) Math

Among the above four libraries, CV2 is an OpenCV library that is used particularly in the project for image processing. However, OpenCV is a library that comes with ties in C++, C, Java, and Python but here we have used it for Python only. The Python libraries used in the coding include matplotlib.pylab, numpy, and math, and they are supporting here the OpenCV library to carry out image processing and wall detection algorithms. Moreover, both matplotlib.pylab and math are given short names such as 'plt' and 'np' respectively. Image processing was performed comprised of multiple steps in Python coding. It starts with the image loading. We first uploaded one of the captured images from the robot's camera and for that 'cv2.imread( )' function has been used which is an OpenCV function as indicated by CV2. The image that was taken by the Raspberry Pi camera was a colored image comprising of pixels(intensity) and each pixel has the dimension represented as (row, column). The image that we used has a total number of 1024 rows and 1280 columns of pixels. Usually, in a colored image, a pixel carries three color channels either R, G, B (Red, Green, Blue) or B, G, R (Blue, Green, Red) but the image that we uploaded was B, G, R therefore among 1024x1280 numbers of pixels, each pixel had B, G, R color channel.

3.2 Wall detection

After the image was uploaded, it had to be rotated because the images taken from the camera were all 180 degrees reversed. To rotate the image, another OpenCV function 'cv2.rotate( )' was used which contains two input arguments, one is the reversed image and the other is the rotating function signifying the degree of rotation as 'cv2.Rotate_180'. Since the captured had noise and unwanted pixels in it, therefore, blurring was needed. However, before blurring the image for noise reduction,

it's always necessary to change the colored image to grayscale. Unlike the colored image, the grayscale image has pixels that carry a single gray color channel. Again, an OpenCV function 'cv2.cvtColor( )' was used to convert the BGR image into a gray image having two input arguments. The first argument represents the image that is needed to be converted and the second argument carries the function mentioning the required conversion as 'cv2.cvtColor_BGR2GRAY'. Now the blurring function of OpenCV is applied as 'cv2.GuassianBlur' that carries three input arguments in the coding. The first argument represents the source image which is the gray image, the second argument contains the size of the kernel. We set the kernel size as [5,5] with respect to our requirement for the image to get blurred. The last argument which is sigmaX is set to be 0 representing the kernel standard deviation along the x-axis. In the last step of image processing, a matplotlib function has been used 'imshow' which was used to plot the blurred image. The matplotlib function 'imshow' is well known for displaying images or drawing figures by using 2-dimensional numpy arrays. Since the blurred image was in grayscale, therefore, 'fixcolor' command has been used to change the color of the blurred image from grayscale into an RGB color channel. Moreover, 'imread' is a function that reads the image in BGR whereas 'imshow' is the function that prints the image in RGB, as a result of that, the final image that we got was with the color sequence of RGB.

3.2.1 Steps of Wall Detection Algorithm

As the project's name implies, the robot uses autonomous guidance to move through the maze, which is based on real-time images captured by the robot's camera as well as a wall detector. Since there have been many different maze-solving algorithms for the robot, as was previously said, we created one that focuses on teaching the robot how to find the walls in the maze.

a. Edge detection, which is used to highlight only the edges in the image, is the first stage in the wall detection technique. Canny edge detection is the operator that is used here which is found to be the better choice among all the other edge detection operators. Therefore, an OpenCV function for canny edge detection 'cv2.Canny( )' is being used that takes up three input arguments. The first input argument takes the blurred image that is in grayscale whereas, the second and third arguments are setting the threshold levels as a minimum and maximum respectively. The minimum threshold value is taken as 40 whereas the maximum threshold value is taken as 140 and it compares the pixel intensity values with the threshold values.

Moreover, following the hysteresis step of canny edge detection, if the pixel gradients of the blurred (grayscale) image are greater than 140, they are considered strong edges, and the pixel gradients less than 40 are rejected by declared as weak edges.

b. The final canny image which is stored in a variable 'canny' is a binary image (black & white) having only two-pixel values i.e. 0 represents black and 255 represents white. Since these pixel values represent the intensity of each pixel, therefore, the 0 value shows the black background whereas, 255 shows the white edges. Once the edges are detected in the image, we have to signify all the x-axis and y-axis coordinates of the edges in the canny image. For this, we have used a numpy function 'np.where' and this function returns all the x-axis and y-axis coordinates of canny image where we have edges. After finding the x-axis and y-axis values of the edges, these values are saved in variables 'y_axis' & 'x_axis'.

c. Since in our image of the maze, three edges were detected which were in the vertical direction, therefore, it is needed to find out the maximum and minimum values of the vertical edges that are on y-axis. Again a numpy functions 'np.min' and 'np.max' have been used that aim to take out the minimum and maximum values of the y-axis.

d. Once we can calculate the maximum and minimum values of the y-axis, the next step is to set up a condition which will be calculating the number of edges for the corresponding axes. Therefore, the 'if' condition is applied in which first we are taking half of the maximum value of the y-axis which we calculated in a previous line of command, then we are rounding this number off using a math library function 'math.ceil( )'. Math.ceil is a function that is used to round off the number making it a whole number. As we don't have any decimal value in our 'y-axis' variable, we couldn't apply a condition with a decimal value of 'max/2'. After rounding off half of the maximum value of the y-axis, numpy function 'np.where' finds out where the middle value of the y-axis lies in the 'y-axis' variable. When the required y-axis coordinate value is achieved, the corresponding coordinate values against the x-axis are fetched from the 'x-axis' variable. As per the image that is taken, the maximum number of edges was three, therefore, a maximum number of values along the x-axis would also be three. Hence 'len' representing the length of edges, would calculate the number of edges with respect to the x-axis. By equating the whole condition to '3', the overall 'if' condition goes like if the length of the edges is equal to '3' then it's a true condition and it would populate three edges in a variable 'num_edge' and if this condition is false it populates two edges in the same variable.

This condition implies that the number of edges in our designed wall detection algorithm is set to be '3' and '2'.

e. A 'for' loop is implemented in the next step of the wall detection algorithm for finding out the coordinates of y and x axes from starting to ending edges. 'For' loop runs by taking the range of y-axis coordinate values. The range has been selected between the minimum and maximum coordinate values of the y-axis that have been calculated before, with a step size of 10 for the estimation purpose. These y-axis values ranging between minimum and maximum having 10 step size will be stored in a variable 'i' followed by a condition. The continuation of this 'for' loop depends upon the 'if' condition which is based on first finding y-axis coordinates values from the range 'i' and then calculating the corresponding x-axis values against it. The 'if' condition compares the length of x-axis values with the num_edge value and therefore, the 'for' loop continues to run till the total number of edges with respect to the y-axis and its corresponding x-axis values, reaches the num_edge value. When the 'if 'condition gets satisfied, it stores the single y coordinate value of the required edges in a variable 'y'. Since there will always be the number of edges greater than 1 and we will be getting more than 1 x-axis coordinate value, another numpy function 'np.sort ( )' is being used. It sorts the x coordinate values against the y-axis by arranging them in ascending order and saving this array in a variable 'x'. Lastly, after getting the required results we don't want the 'for' loop to continue to run therefore if the condition gets failed, the loop will be stopped to run using the break command so that there will be no overwrite of 'x' and 'y' values.

f. After achieving the required edges and their coordinate values, we need to crop the canny image in such a way that the robot would only see the front wall. To separate the front wall from the rest of the image, it is being cropped by adjusting both the x and y axes. It is necessary to verify the image before making any adjustments, and then we must select how accurate we want our image to be with respect to the wall detection. Checking the requirements of our canny image, we set both the coordinates by adjusting them to get the desirable cropped image. Since we don't want to include the third edge as it is not the part of the front wall, we won't be including that third element of the array. For our canny image, we subtracted 20 from the y-coordinate value, stored in the variable 'y', and for the x-axis, subtracted 10 from the first element of the array stored in variable 'x', then added 10 to the second element of the array. However, it is noted that cropping at this stage of the algorithm varies with the size of an image,

in our image we did cropping because we wanted to keep the upper, left, and right sides of the edges for further processing. The concluding cropped image will be saved in a variable that we named ‘canny’. Lastly, the cropped image would be displayed using the ‘imshow’ function and since the final cropped image was in binary, fixcolor was used to change the color from binary to RGB.

g. The final step in the wall detection algorithm is to find the contours along the boundary of the wall ahead of the robot and then we will be aiming to draw contours across it. We will be using the OpenCV function ‘cv2.findContours’ to find the contours in the cropped image ‘canny’. Cv2.findContours has three input arguments here, in the first argument we will be generating a copy image of ‘canny’ as canny.copy( ). We copied the cropped canny image because we intend to find the contours in this very image. The second input argument that is used is one of the contour retrieval/extraction modes ‘RETR_EXTERNAL’ and by using this flag we will be able to extract only the outer contours. Since, we aim to find only the extreme outer contours, leaving behind all the child contours. The third and final argument is ‘cv2.CHAIN_APPROX_SIMPLE’ which is a contour approximation method. As we know, contours are the limits of a shape with (x,y) coordinates for each boundary. We can either store all of the boundary points along the line or simply its ends can be stored. Because of this, "cv2.CHAIN APPROX SIMPLE" tends to simply store the endpoints by deleting all the other points, which conserves memory. The coordinates after being found are stored in a variable ‘cnts’. To finally draw the contours, we wish to do that in the original image that we took from the robot’s camera. Therefore before drawing contours, we will copy the rotated image as ‘img.copy’ and save the copied image in a variable ‘coins’. Now using the drawing function ‘cv2.drawContours( )’ has five input arguments. The first argument would be the rotated copied image (coins) and we first cropped it using the same adjustment of the coordinates as we did with the canny image. However, we are doing the cropping of the image only in the argument to draw contours on the edges of the wall. Then the second argument would be the input contours that were found and stored in cnts, third argument would represent the contour index which is a parameter of drawing a contour that we assigned ‘-1’. The negative sign of the contour index parameter signifies that all the contours are to be drawn. The fourth argument indicates the desired color of the contours which we selected as blue by specifying the color

scheme (255,0,0). The final argument represents the thickness of the lines of the drawn contours which we selected as ‘2’.

h. To display the output matplotlib’s function ‘imshow’ is used and since the image was in BGR, we used fixColor to change the color channel of the contour image from BGR to RGB.

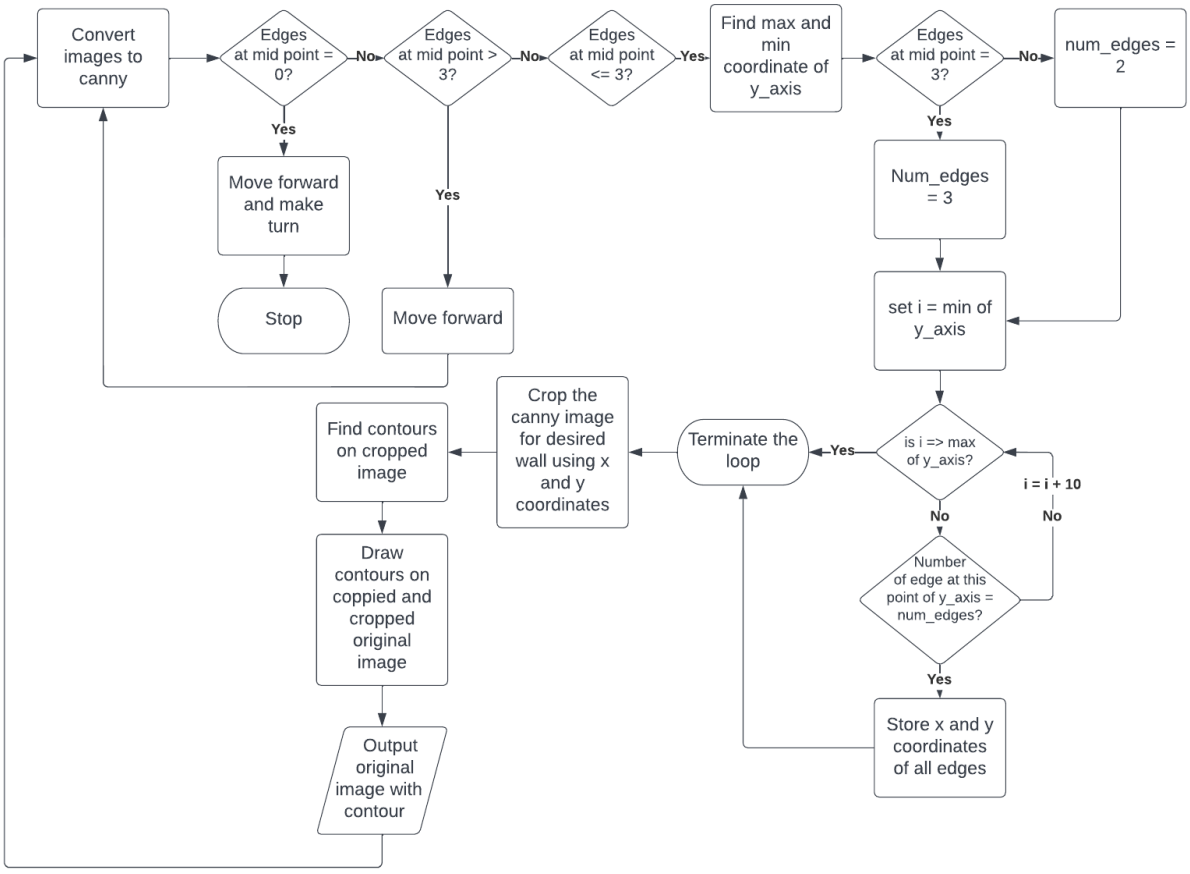

Figure 1 Wall Detection Algorithm

3.3 Logic for movements

One of the important aspects of the project is the robot's movements within the maze, such as forward, left, and right movements. The major goal of the project is to create logic for these movements in such a way that the robot can determine when to begin moving forward. Additionally, how long it must continue on the same path, as well as when it must stop traveling forward and make a turn to the right or left.

3.3.1 Forward Movement

The forward logic that has been designed for the robot mainly depends on detecting the wall that comes ahead of it. Once the wall detection algorithm is designed, the forward movement logic needs to be integrated with the wall detection. It is necessary because when the robot enters the maze and starts taking the images, it has to perform two tasks simultaneously: one is moving forth and the other is detecting the wall. It should be noted, though, that we created forward logic, which causes the robot to always start moving forward if it gets images with more than three edges. On the other hand, it was previously stated that our wall detecting algorithm was established and that it begins to function once it receives three edges.

3.3.1.1 Steps of Forward Logic

a) Logic starts with a 'while' loop followed by a robot taking the pictures in real-time and the same process will be performed which has been designed as the initial steps of image processing and noise reduction. Starting from loading the image and the same will be rotated, converted to grayscale, and applying blurring to it to reduce noise. Moving on, the canny edge detection function is being performed followed by finding the edges with respect to the x and y axes, taking out the maximum and minimum values of y coordinates of the edges.

b) Now 'if' condition is applied by equating either variable 'y-axis' or x-axis' with null array which is true when the robot gets the image with 0 edges. In that case, the robot will move forward for 5 seconds (timer could be set), break the loop, and will be ready to take either right or left turn. If the condition is false it will go to the next 'else if' condition.

c) Now 'else if' condition is applied for forwarding movement which first finds the half of the maximum value of y coordinate of the vertical edges and checks the length of corresponding x coordinate values i.e. number of edges. If the length is greater than 3, the robot continues to move forward for 10 seconds which signifies that the wall is far from the robot. Here '3' is the threshold level being set for the robot which means among the images taken by the robot, if it's getting greater than 3 edges, it will be moving towards the forward path. The while loop continues to run as long as the length of the edges is more than three. When this 'else if' condition fails it goes to the else body.

d) 'Elseif' condition gets failed when the number of edges is not greater than 3 which means it becomes equal to or less than 3. This is when the logic enters into the 'else' body which states that when the image got three edges or less, the rest of the wall detection algorithm starts

implementing including setting the number of edges, finding the coordinates of the starting and ending edges of the wall, cropping the image, finding and drawing contours. Along with following the algorithm, the robot keeps moving forward but for 5 seconds since the wall is getting near to it.

3.3.2 Left/Right Movement

3.3.2.1 Steps of Left/Right Logic

a) This logic for left and right starts within the 'else' body of forwarding movement i.e the turn will be decided once a picture with less than equal to 3 appears for the very first time in the code. The first condition we have to check 'if' condition, the turn is equal to 0 then this condition gets true and our code for calculating the turn will run otherwise the code will continue to perform wall detection. Outside the while loop, we define the same variable 'turn' and populate it with 0. This variable will later change to either left or right at the end of our direction decisioning logic hence the value of 'turn' will get changed and the condition which says that if 'turn' is equal to zero will get false and the module for deciding the direction would not run.

b) Once it is checked out that the turn is not decided yet then we have to run the direction decisioning logic. For doing this, we must look into the 'y_axis' and 'x_axis' variables that were created earlier and take out the starting coordinates of the lengthiest edge. Since y_axis and x_axis contain the coordinates of edges only and are stored in ascending order of y_axis, it can be concluded that at position y_axis[0] and x_axis[0] we will get the coordinate of the first edge and it would always be the starting point of the lengthiest edge.

c) Since the starting point of the line is known, a variable 'final_x' is initiated which will be later used for populating the x coordinate of the end of the lengthiest line. Currently, it is set to zero which means that end coordinates are not found yet. If they are found and final_x is populated with some numbers greater than 0, the loops will break.

d) After obtaining all the helping variables and starting coordinate of the lengthiest line, a loop will be started from maximum to minimum i.e the loop will run in descending order of y_axis with a step size of -10. The main purpose of this loop is to find out the ending coordinate of the lengthiest line. First, we find out the number of edges and the values of x_axis elements corresponding to each y_axis variable decrementing in the loop. Again the number of edges and the values in the x_axis elements corresponding to the midpoint of the y_axis are

calculated. Now validate if it already has the ending point of the lengthiest line. If not the condition gets true and the loop breaks otherwise, it moves forward to finding the end of this line. For doing this, another loop is started from 0 to the number of elements of the x_axis at the midpoint of the y_axis. Within a loop, we compare each value of the x_axis at the midpoint with the starting x_axis coordinate of the lengthiest line. If any of the values in the x_axis of the midpoint lies within the +-100 range of the starting point of the x_axis, it can be concluded that the middle point of the line is found and the condition gets true. If this condition gets failed, another element in the array of the midpoint is checked against the starting x_axis coordinate of the lengthiest edge after running the 'else' body.

e) Once it is known that the line exists at the midpoint, another loop is started from 0 to the size of the x_axis coordinate at the point where the max to min loop is running. The same 'if' condition is followed that checks if the element in the array lies in a range of the starting x_axis coordinate of the lengthiest line with the margin of +100 and -100 then it can be concluded that the coordinates of the end of the line are found. The y coordinate is taken out from looping variable 'i' and placed in 'final_y' variable while the x coordinate is taken out from one of the elements of 'lowx_value' variable and placed in 'final_x' variable. The range of +100 and -100 is given because the line is not straight and we will not have a constant x_axis coordinate throughout the line.

f) Finally, the ending coordinates of the lengthiest line are found, and the loop breaks and recheck if the final_x value is now populated. If it is populated then the max to min loop will break as well. After getting out of the loop, 'final_x' value is checked if it lies at the right side of the midpoint of the x_axis. If it does, store 'Right' in the variable 'turn' and calculate the length of the lengthiest line by passing the starting and ending coordinates to the already defined function 'calculateDistance'. However, if this condition gets failed we store 'left' in the 'turn' variable and calculate the distance similarly. Now we have a value in the 'turn' variable so the direction decisioning code will not run again.

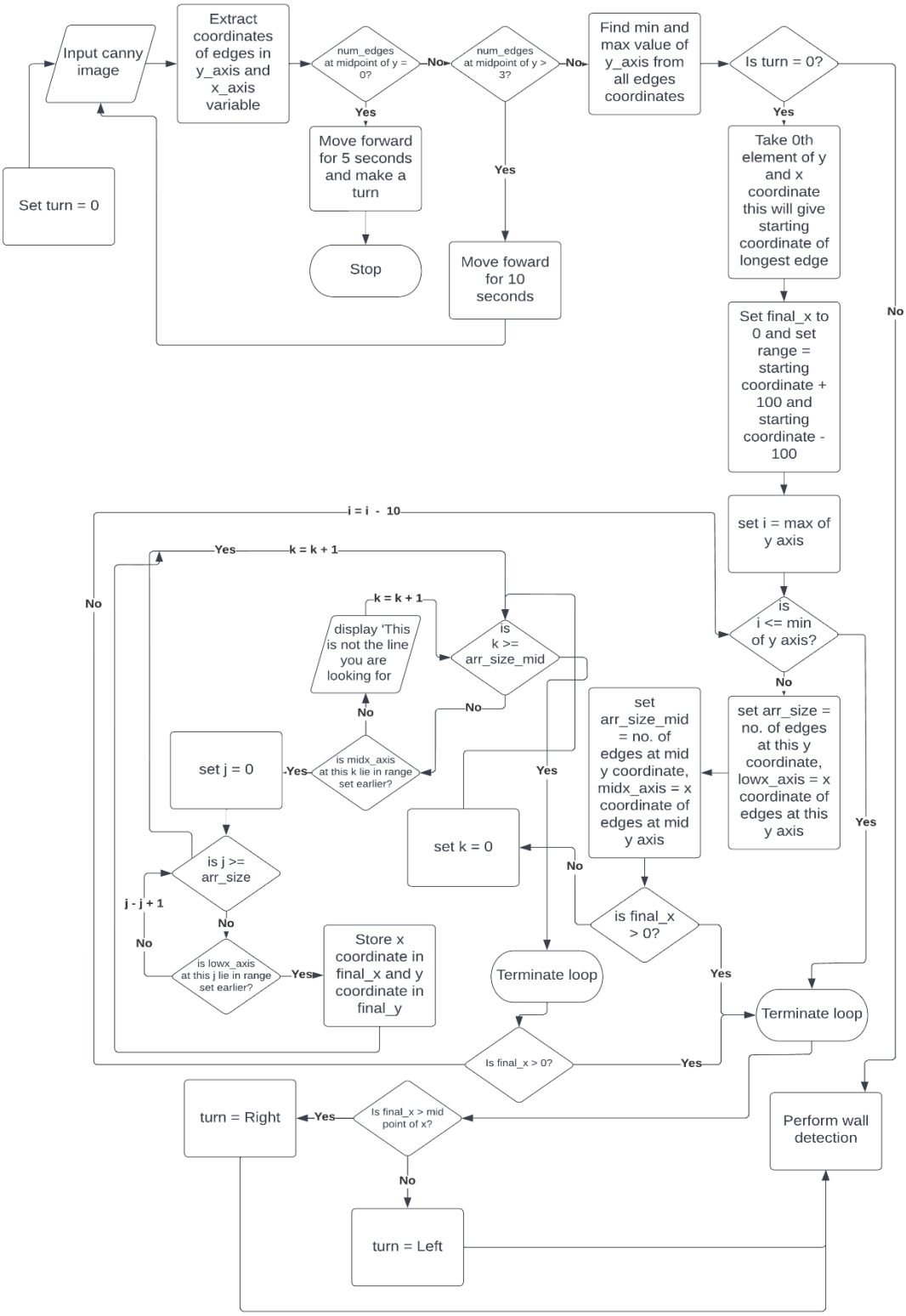

Figure 2 Logic for movements

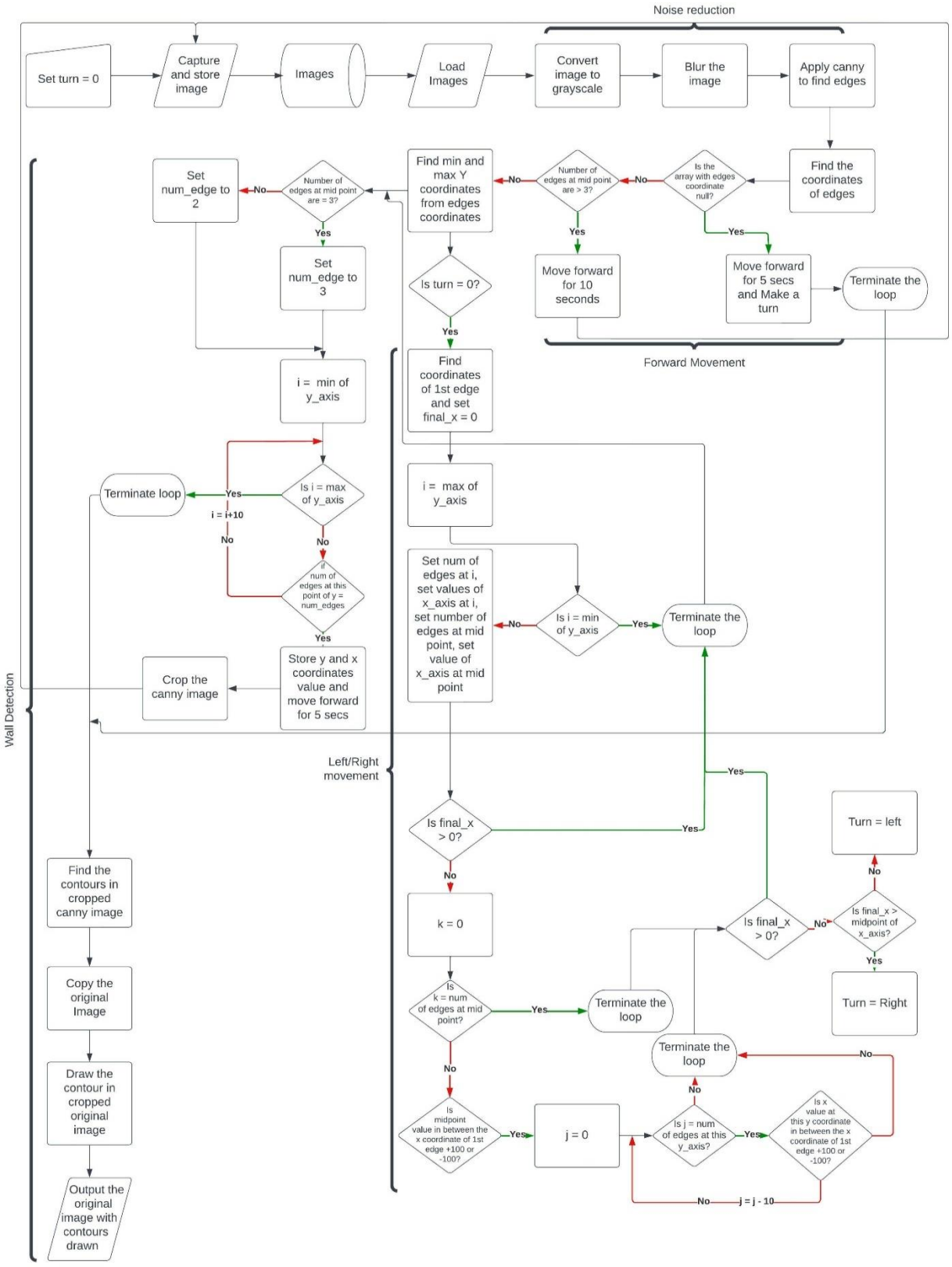

Figure 3 Methodology

Chapter 4

4 Results

4.1 Noise Reduction

The first major task that has been done during the working of the project was taking pictures from a Raspberry Pi camera. To write the wall detection algorithm and perform image processing, we must have the actual images of the maze created by us. Using Python code, we captured several photographs from the camera and obtained the results shown below.

Figure 4 Original Image from Robot's camera

The above image was taken at the entrance of the maze and it is clearly shown in the image that the picture that was captured by the robot's camera was upside down. Using the OpenCV function, the image was rotated as follows.

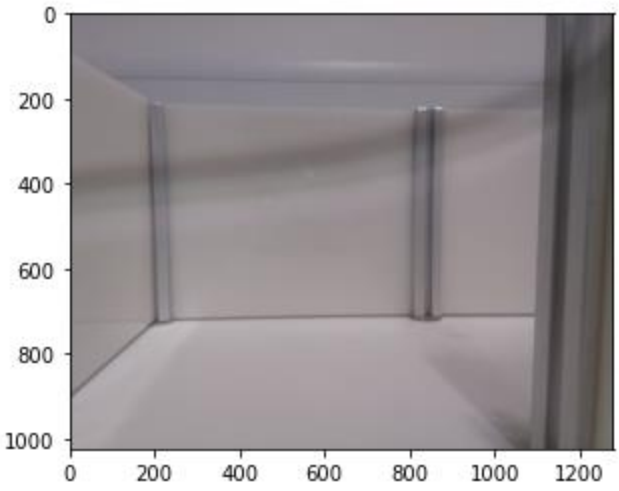

Figure 5 Rotated Image

The above-rotated image was in BGR and to do edge detection on the image, we needed to convert it from BGR to grayscale. Therefore by using the OpenCV function, the following result was generated in grayscale.

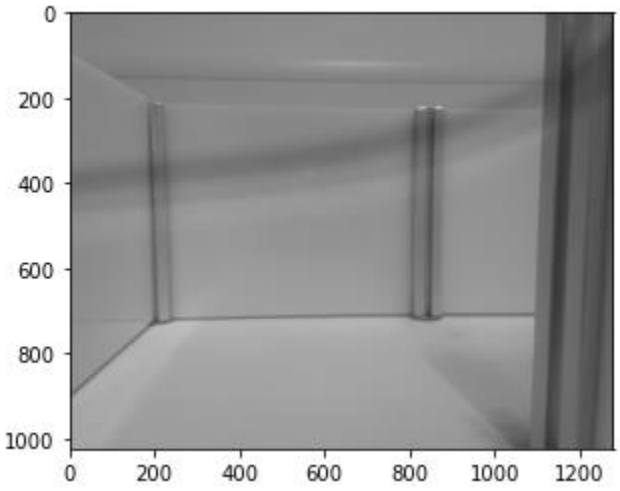

Figure 6 Grayscale Image

Since the original image was white or off-white, the color change in grayscale is a little less pronounced, but it is still evident that the image has distinct grey shades in grayscale. This is because grayscale only has one color channel with values that range from 0 (black) to 255 (white). The result for a blurred image is given below.

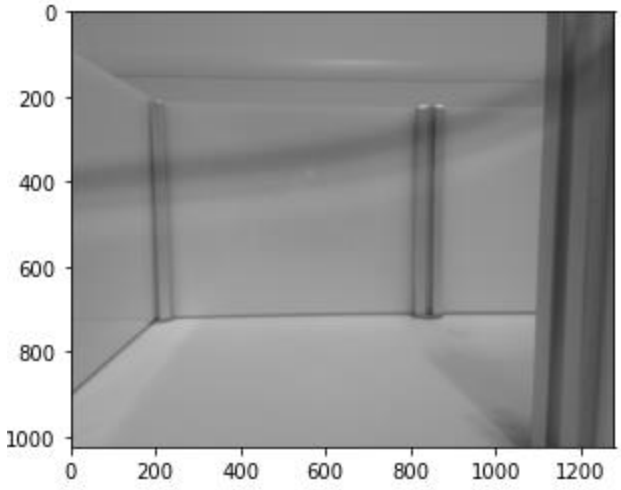

Figure 7 Blurred Image

The original images always contain noise in them, thus blurring which is nothing more than noise reduction is a smart strategy and needs to be done for improving edge detection outcomes.

4.2 Wall Detection Algorithm

- The initial step in developing a wall detection algorithm is edge detection, and for this project, canny edge detection was proven to be the most effective edge detection operator. After edge detection, it outputs the image as seen below.

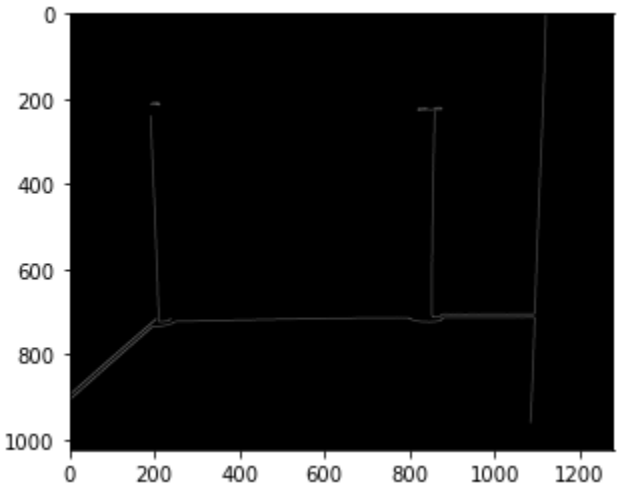

Figure 8 Canny Image

Because of the threshold settings that we chose for canny edge detection, the result above shows the edges pretty precisely. We first varied both the upper and lower limits of the threshold until we

obtained the desired outcome mentioned above. The programmer's preference is all that determines the threshold limit for edge detection. As the front wall of the maze was our primary target for detection, it is clear that our main goal was to obtain correct edges, particularly on the front wall.

After getting the required edges, we needed to crop the image by making adjustments in the x and y coordinate values. The main objective was to crop the canny image in such a way that the top portion of the two vertical edges of the front wall would remain in the final image. However, the third vertical image wouldn't be needed as it wasn't part of the wall. Likewise, the left and right side of the vertical edges of the front wall was required to be snipped. Lastly, the bottom side of the canny image remained the same as it depends on the distance taken by the robot as it moves forward towards the wall. The final cropped image is shown below.

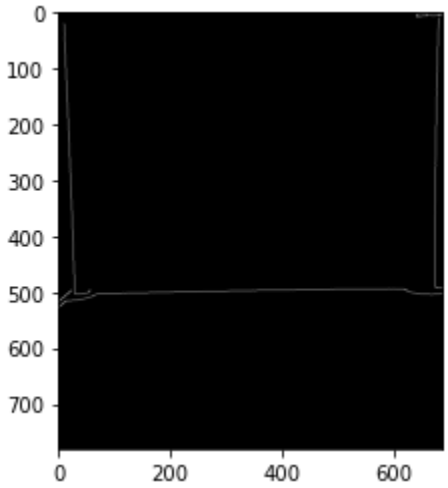

Figure 9 Cropped Canny Image

The final result of the wall detection algorithm would be as follows.

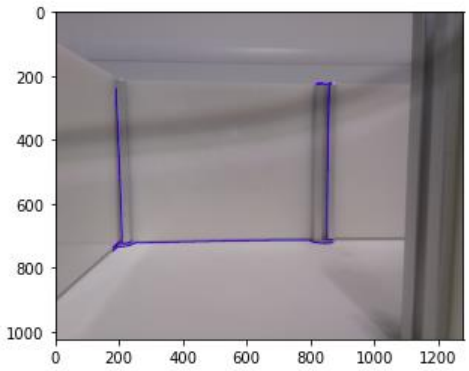

Figure 10 Wall detection with Colored Contours

The above result shows the original image with the blue contours drawn on the edges of the front wall. Every time the robot takes an image and gets the number of edges either 3 or 2, the wall detection algorithm is implemented by finding the number of edges and checking them with the set value, then finding the required coordinates, cropping them, and drawing contours on them. The entire process would enable the robot to obtain the image seen above, which would direct him in determining whether walls are present as he approaches them. Thus, feature detection and real-time image processing enable the robot to operate independently. It can also be observed that the final image is not as cropped as the canny image because we didn't store the cropped image in any variable but we did the cropping as the argument of drawContours function. Therefore, the final result of the image was stored after the contours had been drawn on the edges.

4.3 Forward movement

Originally, the robot would have taken real-time photographs inside the maze, but because that wasn't possible at this point, we used sample images taken by the robot's camera to create the algorithm and develop the logic for the movements. It began by taking pictures at the maze's entrance and continued until the front wall's two borders were all that remained visible to the camera. As a consequence, we obtained a total of three photographs, which we refer to as "3," "2," and "1". Given the actual distance between the entrance of the maze and the front wall, the first two images '3' and '2' featured three edges, but obviously at different distances, whilst the final image '1' only had two edges of the front wall.

We stored '3' in a variable 'count' and starts the logic with a while loop. Now since the count is a variable having a numeric value i.e. 3, we first need to convert it into string type and then after adding '.jpg' to it, the image will be saved in a variable 'img_name'. Keep in mind that the variable count is only used here to validate te code in offline mode. In the next command, the count will be reduced by a value of 1 but since 'image 3' is already stored in 'img_name' this counter won't affect the first loop. As the first 'if' condition gets true only when the image with 0 edges is found, therefore, the logic checks the' else if' condition which also gets failed as it works for the number of edges greater than 3. Now the 'img_name' gets verified by the next condition which works when the number of edges reaches 3 or less than 3 and the wall detection algorithm starts from here. After applying wall detection on 'image 3', the while loop repeats itself with 'image 2' which also had three edges, else condition gets verified and the same process takes place. The last image,

‘1’ although had two edges but still verified else condition and performed a wall detection algorithm to the end. To check the forward logic we printed the following output.

```
Now performing wall detection on image  3.jpg because wall is near now and moving forward for 5 secs
Now performing wall detection on image  2.jpg because wall is near now and moving forward for 5 secs
Now performing wall detection on image  1.jpg because wall is near now and moving forward for 5 secs
```

Figure 11 Output of Forward logic

After ‘image 1’ if we use an image with 0 edges, the ‘if’ condition gets true and it will break the while loop after moving forward for 5 seconds. Following is the output of ‘image 1’ using the designed forward logic.

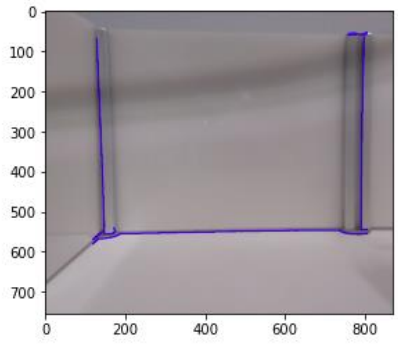

Figure 12 Wall Detection with Forward Logic

The above result demonstrates how the robot will move forward and recognize the wall while taking the images inside the maze.

4.4 Left / Right Movement

As mentioned earlier, the captured images can only be used for the simulation of results at this point. Since the images stored were for the right turning in the maze, for the sake of verification, a horizontally flipped image is being used to check if the code can produce the output and detect the difference between the correct turn. Following is the comparison between both the images and their outputs.

Figure 14 Flipped Imaged

Figure 13 Original image

The image on the left is flipped and has a left side turn while the image on the right has a right side turn. Now let's see what the outputs are.

```
Now performing wall detection on image  6.jpg because wall is near now and moving forward for 5 secs
Looking for turn....
195 This is the value
We have found a break in a line or this is not the line you are looking for searching again...
We have found a break in a line or this is not the line you are looking for searching again...
your turning is on the left since the lenght of edge on left side is:  951.6438409405065
Now performing wall detection on image  5.jpg because wall is near now and moving forward for 5 secs
Now performing wall detection on image  4.jpg because wall is near now and moving forward for 5 secs
Now performing wall detection on image  3.jpg because wall is near now and moving forward for 5 secs
Now performing wall detection on image  2.jpg because wall is near now and moving forward for 5 secs
Moving forward for 5 secs then making Left turn
```

Figure 15 Output with Left Turn

```
Now performing wall detection on image  5.jpg because wall is near now and moving forward for 5 secs
Looking for turn....
We have found a break in a line or this is not the line you are looking for searching again...
We have found a break in a line or this is not the line you are looking for searching again...
1083 This is the value
Your Turning is on the right since the lenght of edge on right side is:  955.6782931509955
Now performing wall detection on image  4.jpg because wall is near now and moving forward for 5 secs
Now performing wall detection on image  3.jpg because wall is near now and moving forward for 5 secs
Now performing wall detection on image  2.jpg because wall is near now and moving forward for 5 secs
Moving forward for 5 secs then making Right turn
```

Figure 16 Output with Right Turn

The output for the left turn is on the left-hand side while the output for the right turn is on the right hand side. The code for determining the direction is written in else block of forward movement which only runs if edges are less than equal to 3. However, the turn is only decided once and the decision gets stored in a variable 'turn'. The 'if' condition within the else body of the forward movement first checks whether the variable 'turn' is 0 or not if it is not 0 then the logic for determining the 'turn' would not run else it would. Moreover, the number in the output after the line 'Your turning is on the right since the length of the edge on right side is: xxxx' indicates the length of this edge this is printed for validating if the longest edge is being used for determining the turning of the robot later.

Chapter 5

5 Conclusion

This project focuses on image processing in such a way that enables the robot to navigate the maze on its own. A maze was created for the robot to take the representative images from the Raspberry Pi camera. Python code was written for taking the pictures from Raspberry Pi. Later, Raspberry Pi was integrated with the local desktop and tried taking images from the camera with the help of the code. After capturing front images of the maze with the camera mounted on the robot, it would be able to identify the wall in front of it.

A wall detection algorithm was written for the robot. Before writing the algorithm, image processing was performed on the images taken by the robot earlier. We couldn't proceed to the algorithm before processing the images using various OpenCV functions. Since the images taken from the Raspberry Pi camera on the robot were upside down, the image was rotated. Then to suppress the noise, blurring needed to be done, and before that image was converted from BGR to grayscale.

The wall detection algorithm that has been built for the project is distinct from the algorithms that were previously used. To navigate the maze, the project doesn't make use of any sensors or other turning algorithms. One of the fundamental phases in creating the wall detection algorithm is canny edge detection. Among various edge detection operators, canny edge detection seemed to be a better option for the project. The blurred image after noise removal was used for edge detection. Threshold levels were to be set as per our requirement for edge detection. Then the coordinate values of all the canny edges were found with respect to the x and y axes. After setting the number of edges as '3' or '2', the coordinates of the x and y axes for the required edges were found. The image was then cropped as per our requirements i.e the upper, left and right sides of the edges were cropped. Lastly, contours were found and then drawn using OpenCV functions. Blue-colored contours were neatly drawn on the edges of the front wall. The robot can recognize the front wall as soon as it enters the maze with the help of a wall detection algorithm. By cropping the image and drawing colorful contours on the edges of the wall, the robot avoids itself to get hit by the wall.

The forward logic works even if the robot is outside the maze, it would still be moving forward with more than three edges seen by it. By getting more than three edges, the robot would be moving forward for 10 seconds, and this would be the case when the robot is outside the maze. As soon as it gets three or two edges, it gets the logic to move forward but for 5 seconds as it would be approaching the wall ahead. At the same point, the wall detection algorithm starts working as it begins from the edges equal to 3. There would be a point when the image gets 0 edges and that would be the point when the robot stops moving forward after 5 seconds of moving.

When it has three edges—two of the front wall's edges and one on the corner—it records the decision to turn using left or right logic and begins working on the wall detection algorithm while continuing to move forward. The robot's decision to turn right or left is primarily based on the corner side of the edge thanks to the left/right logic. As per forward logic, when the robot gets 0 edges, it would stop moving ahead after a delay and turn accordingly using previously stored left/right decision .

The results are generated for each phase of the project i.e. wall detection algorithm, forward, left and right movements showing the successful implementation of the working and coding in Python. At the start of the project, the robot’s left, right, forward and backward movements was successfully tested using a simple Python code. However, final working of the entire coding is not integrated with the robot itself. Therefore for future work, it is recommended that the algorithm with the movement logics can be collaborated with the robot to validate the entire software work that has been done in the project. The hardware implementation could make it clear how the robot could move inside the maze, detecting walls and performing turns in accordance with the designed logics and code.

Bibliography

[1] G. A. Bekey, "Autonomous Robots," in *Biological Inspiration to Implementation and Control*, 2017.

[2] H. H. Prayash, M. R. Shaharear, M. F. Islam, S. Islam and N. Hossain, "Designing and Optimization of An Autonomous Vacuum Floor Cleaning Robot," in *IEEE International Conference on Robotics* , Dhaka, 2021.

[3] IEEE, "Google Self-Driving Car," Robots.IEEE, [Online]. Available: https://robots.ieee.org/robots/googlecar/. [Accessed July 2022].

[4] A. Y. Z. Dao, "Maze Escape with Wall-Following Algorithm," Andrew Yong Zheng Dao, 2021.

[5] D. S.Suryanarayana, "AUTONOMOUS MAZE SOLVING ROBOT USING ARDUINO," *International Journal of Advanced Research in Engineering and Technology,* vol. 12, no. 3, 02 March 2021.

[6] Mjrovai, "Maze Solver Robot, Using Artificial Intelligence With Arduino," Instruction Tables Circuits, [Online]. Available: https://www.instructables.com/Maze-Solver-Robot-Using-Artificial-Intelligence-Wi/. [Accessed July 2022].

[7] O. Kathe, V. Turkar, A. Jagtap and G. Gidaye, "Maze solving robot using image processing," 2015.

[8] H. Ghael, "A Review Paper on Raspberry Pi and its Applications," *International Journal of Advances in Engineering and Management,* vol. 2, no. 12, pp. 225-227, 2020.

[9] C. Cawley, "26 Awesome Uses for a Raspberry Pi," Make use of, 10 Dec 2019. [Online]. Available: https://www.makeuseof.com/tag/different-uses-raspberry-pi/. [Accessed July 2022].

[10] Deepak.S.Kumbhar, H. Chaudhari, S. S.Bhatambrekar and S. M.Taur, "IoT Based Home Security System Using Raspberry Pi-3," *International Journal of Research and Analytical Reviews (IJRAR),* vol. 6, no. 1, 2019.

[11] D. Rolon-Merette, M. Ross, T. Rolon-Merette and K. Church, "Introduction to Anaconda and Python: Installation and setup," University of Ottawa, Ottwa.

[12] A. SebastiánYagüe, "Python for scientists," [Online].

[13] K. R. Srinath, "Python – The Fastest Growing Programming Language," *International Research Journal of Engineering and Technology (IRJET),* vol. 4, no. 12, 20 July 2017.

[14] D. C. S. Yadav, "Basic Fundamental of Python Programming Language and The Bright Future," *Peer Reviewed Refereed and UGC Listed Journal,* vol. 8, no. 2, 02 05 2019.

[15] S. v. d. Walt, S. C. Colbert and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *IEEE Xplore.*

[16] W3 Schools, "NumPy Introduction," W3 Schools, [Online]. Available: https://www.w3schools.com/python/numpy/numpy_intro.asp. [Accessed July 2022].

[17] Tutorials Point, "Matplotlib," 2016. [Online]. Available: https://www.tutorialspoint.com/matplotlib/matplotlib_tutorial.pdf. [Accessed July 2022].

[18] GeekforGeeks, "How to rotate an image using Python?," GeekforGeeks, 3 June 2022. [Online]. Available: https://www.geeksforgeeks.org/how-to-rotate-an-image-using-python/. [Accessed July 2022].

[19] Project Pro, "How to rotate an image using OpenCV," Project Pro, 20 June 2022. [Online]. Available: https://www.projectpro.io/recipes/rotate-image-opencv. [Accessed July 2022].

[20] G. W. Cottrell and C. Kanan, "Color-to-Grayscale: Does the Method Matter in Image Recognition?," Plos One, 10 January 2012. [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0029740#:~:text=The%20main%20reason%20why%20grayscale,algorithm%20and%20reduces%20computational%20requirements. [Accessed July 2022].

[21] M. K. Khobragade, "A Comparative study of Converting Coloured Image to Gray-scale Image using Different and Technologies," Pune, 2012.

[22] techtutorialsx , "Python OpenCV: Converting an image to gray scale," techtutorialsx , 2019. [Online]. Available: https://techtutorialsx.com/2018/06/02/python-opencv-converting-an-image-to-gray-scale/. [Accessed July 2022].

[23] A. Rosebrock, "OpenCV Smoothing and Blurring," py image search, 28 April 2021. [Online]. Available: https://pyimagesearch.com/2021/04/28/opencv-smoothing-and-blurring/. [Accessed July 2022].

[24] S. Maharaj, "Advanced OpenCV: Blurring An Image using the Renowned OpenCV Library," Analyticsvidhya, 25 August 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/08/advanced-opencv-blurring-an-image-using-the-renowned-opencv-library/. [Accessed July 2022].

[25] Tutorials Point, "OpenCV - Gaussian Blur," Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/opencv/opencv_gaussian_blur.htm. [Accessed July 2022].

[26] Tutorials Kart, "OpenCV Python Image Smoothing – Gaussian Blur," Tutorials kart, [Online]. Available: https://www.tutorialkart.com/opencv/python/opencv-python-gaussian-image-smoothing/. [Accessed July 2022].

[27] Tutorial Point, "OpenCV - Gaussian Blur," Tutorial Point, [Online]. Available: https://www.tutorialspoint.com/opencv/opencv_gaussian_blur.htm. [Accessed July 2022].

[28] R. Jain, R. Kasturi and B. G. Schunck, "Edge Detection," in *Machine Vision*, McGraw-Hill, 1995.

[29] R. Fisher, S. Perkins, A. Walker and E. Wolfart, "Sobel Edge Detector," 2003. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm.

[30] L. Han, Y. Tian and Q. Qi, "Research on edge detection algorithm based on improved sobel operator," in *Institute of Information Engineering, Beijing Institute of Graphic Communication*, Beijing, 2020.

[31] U. Sinha, "The Sobel and Laplacian Edge Detectors," 2010. [Online]. Available: https://aishack.in/tutorials/sobel-laplacian-edge-detectors/.

[32] Tutorials Point, "Laplacian Operator," Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/dip/laplacian_operator.htm. [Accessed July 2022].

[33] R. Maini and D. H. Aggarwal, "Study and Comparison of Various Image Edge Detection Techniques," *International Journal of Image Processing,* vol. 3, no. 1.

[34] R. Maini and D. H. Aggarwal, "Study and Comparison of Various Image Edge Detection Techniques," *International Journal of Image Processing (IJIP),* vol. 3, no. 1.

[35] N. A. Channar, "A Comparative Study of Edge Detection Techniques in Digital Images," Grin, 2016. [Online]. Available: https://www.grin.com/document/512938.

[36] S. Singh and R. Singh, "Comparison of Various Edge Detection Techniques," 2015.

[37] R. Chandwadkar, S. Dhole, V. Gadewar, D. Raut and P. S. A. Tiwaskar, "Comparison of Edge Detection Techniques," in *Proceedings of Sixth IRAJ International Conference*, Pune, 2013.

[38] J. A. M. Saif, M. H. Hammad and I. A. A. Alqubati, "Gradient Based Image Edge Detection," *IACSIT International Journal of Engineering and Technology,* vol. 8, no. 3, 2016.

[39] S. Sahir, "Canny Edge Detection Step by Step in Python — Computer Vision," Towards Data Science, 25 January 2015. [Online]. Available: https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123. [Accessed July 2022].

[40] OpenCV, "Contours : Getting Started," OpenCV, [Online]. Available: https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html. [Accessed Jjuly 2022].

[41] DefltStack, "OpenCV Find Contours," DefltStack, 23 January 2022. [Online]. Available: https://www.delftstack.com/howto/python/opencv-find-contours/. [Accessed July 2022].

[42] geeksforgeeks, "Feature detection and matching with OpenCV-Python," geeksforgeeks, 28 Nov 2021. [Online]. Available: https://www.geeksforgeeks.org/feature-detection-and-matching-with-opencv-python/. [Accessed July 2022].

[43] S. Campbell, "Use PuTTY to Access the Raspberry Pi Terminal from a Computer," Circuit Basics, [Online]. Available: https://www.circuitbasics.com/use-putty-to-access-the-raspberry-pi-terminal-from-a-computer/#:~:text=PuTTY%20is%20a%20software%20application,and%20receive%20data%20fro. [Accessed July 2022].

Appendix A

Reporsitory Structure

Code can be clone from the link: https://github.com/aqsakhan2022/Msc-Project.git4

The folders in the git are as follows:

- Images
  - Contains the images that are used in validation of the code
- Docs
  - Contains the final report of the project
  - Contains the poster presentation
- Code
  - Contains the final notebook of the code in Python