**Real-time Image Processing for Autonomous Guidance of Maze Following Robot**

**Introduction**

**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.[2] Moreover, autonomous robots have also shown their usage in cleaning as autonomous vacuums or vacuum cleaners. [1] 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.

**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 decisions 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). [4] 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.

**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 feather/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.

**Theory:**

**Hardware:**

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). [8] 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]

**Processor:** This is the basic part of Raspberry Pi which is responsible for all the computations, decision making, data processing, and fetching the results. BCM2837B0 in this Raspberry Pi is a fast and power-efficient quad-core processor running with a clock speed of GHz.

**GPU:** It is a hardware component that is used to perform operations on images and videos that’s why it is called a Graphics Processing Unit. GPU with OpenGL (Open Graphics Library) version 2.0 and hardware-accelerated OpenVG API which is a 2-D Vector Graphics Application Programming Interface is used in this Raspberry Pi 3 B+.

**Storage:** Raspberry Pi 3 B+ offers a storage capacity of 1GB and RAM used in it is LPDDR2 SDRAM which is the low power double data rate synchronous dynamic random access memory designed to work on low power without risking performance.

**Wireless LAN & Bluetooth 4.2:** Raspberry Pi 3 B+ comes with built-in Bluetooth and Wi-Fi features, allowing its users to set up Wi-Fi wirelessly. The on-board connectivity options of Raspberry Pi 3 B+ provide with successful set up of wireless networks in terms of both Wi-Fi and hotspot without using any kind of USB dongles.

**HDMI & USB Ports:** Raspberry Pi 3 B+ consists of 4 USB ports so there would be no need to connect extra ports using the USB hub. Standard HDMI (Type A) connector is used in Raspberry Pi 3 B+ model. The standard (Type A Male to Type A Male) cable helps Ethernet, 3D, and 4k videos.

**Power Input:** Raspberry Pi 3 B+ requires a power supply of 5V/2.5A for the Pi to work and a minimum of 2A would be sufficient to maintain its stability. [11] &[12]

**40-pin GPIO (General Purpose Input Output):**

To interface the robot with the Raspberry Pi so that robot can be operated as per the coding done on the Raspberry Pi, it is necessary to connect the 40-pin GPIO bus cable. It is called GPIO (General Purpose Input Output) because these pins provide standard interfacing by connecting the Raspberry Pi to external input/output devices. The GPIO has power pins such as 5V pins, 3V pins, and 0V (ground pins) and plenty of input/output pins. The processor of the Raspberry Pi works on 3V and if a sensor that runs on 3.3V is used then this power pin becomes vital for it. However, the Raspberry Pi board needs 5V to get powered with also the 5V power pin can be used as a supply for the motors that need 5 volts to run. Then there are TX and RX are the transmission and receiving pins that are used for the communication. The number of input/output multifunctional pins used with I2C (Inter-Integrated Circuit) and SPI (Serial Peripheral Interface) communication protocols.

**Operating System:**

Raspberry Pi 3 B+ contains an operating system specially made for it and the Linux it has is Debian version 11 and the version of kernel it supports is 5.15. The operating system in this Raspberry Pi is Raspberry Pi OS lite and it doesn’t have GUI (Graphical User Interface) in lite version.

**SSH:**

1. **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.

**Software:**

1. **Anaconda:**

Anaconda is a 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 Python distributor. It is a free software that comes with Python interpreter, various Python packages, and editors. After installing Anaconda, we gain access to environments called as 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 programme 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:

1. **Jupyter Notebook:** The Jupyter notebook is an interactive open-source environment that allows users to write 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, 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. **Spyder:** Spyder, Scientific Python Development Environment, is an IDE that allows a programmer to interact with code very effiently as it offers advance editing, introspection features and vibrant debugging and testing of code.

* **Advance editing** helps in following the correct syntax and indentation in the code by heighlighting 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 bug in it

Furthermore, Spyder also has a window to display the varibales that are initiazed in the code this is why it gets easier to debug and Programmer is not required to go through a hassle of printing every variable to debug it. Lastly, Spyder is extremely customizeable this would help programmer select settings that are easy on the eyes as they spend number of hours infront of the screen. [25] & [26]

1. **Python:**

We constantly invent things for our own ease and the trend of adapting things that are easier to grasp is no secret. Similarly, the need of implementing a language that a programmer can easily understand and code results in the implementation of programming language with name Python. The project for developing Python started in December 1989 and the first working version of python was released in Feb, 1991. Today, Python is one of the most popular programming language. The rapid increase in the popularity of python is due to its readability, simple syntax and great learning curve for junior coders. Moreover, python is an open source programming language under OSI-approved license this makes python developer to easily access the source code and amend it according their need which results in pre-developed libraries by many developers which can be easily used by programmers to help them in completing the project. There are multiple tasks that 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

There are many advantages of Python language some of them are listed below:

* Python is an interpreter language it runs each line in sequence and does not need compilation this make it easier to get to an error
* It is easier to read, write and understand
* It is free and has Open source licensee making it easier to change
* There are almost all the functions required for making an efficient program

Pros always comes with cons and some of the features in which Python lacks are:

* Python is time taking due to its nature of being an interpreter language
* Since, in python’s syntax is somewhat similar to English language it takes up more space so it is not memory efficient
* The processing of using python for connecting to database is fairly slower
* Python cannot be used for coding at lower level. [27]

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 help the programmers doing 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 clearly a better choice to perform these tasks as it comes with a variety of image processing tools with their free availability [30]. 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 the numerous pros like 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 are able to 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. [30]

**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:

1. **OpenCV**

Open CV (Open source Computer Vision) is the library which 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.

1. **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 but their processing speed is relatively poor. However, NumPy comes with an array object called ‘ndarray’ (N-dimentional array) which is over 50 times quicker than Python's default lists, 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. [31] & [32]. NumPy is utilised 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 processingbut this combination is frequently employed in place of MATLAB. Moreover, another benefit of NumPy is that it is open source and so publicly available.

1. **Matplotlib**

The Python module Matplotlib is used to plot graphs, making it a tool for data visualisation. Like Numpy, Matplotlib is a free and open-source programme. Matplotlib is being utilised in the project for static visualisation however, it is a complete library for both static and interactive visualisations. 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. [33]

**Techniques:**

**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 minimise 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.

1. **Image Rotation**

One of the image processing procedures called "image rotation" which rotates an image by a predetermined amount of degrees around its centre. A geometric change known as image rotation can be carried out using either a forward transformation / mapping or an inverse transformation / mapping [35]. 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. [35] There are several functions available to rotate the images such as PillowIt is a Python image processing package that rotates the images counterclockwise by utilising inverse transformation with the amount 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 [35]. However, the images taken by the robot’s camera were rotated using OpenCv function. [34] 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 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 [34]. Therefore, we used cv2.rotate() as we only needed to rotate the images by 180 degree.

1. **Image into Grayscale**

The conversion of colourful 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 colourful 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 speed 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 afterwards for the wall detection algorithm, and the BGR picture information wouldn't let the edge detection identify all the essential edges in the image. Writng an algorthim 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. [36] & [37] The OpenCV function cvtColor() is used to convert an image from one colour space to another. This function contains two input arguments, the first input argument receives the rotated image of the maze. However, the code for colour space conversion is sent as the second input argument. We used COLOR\_BGR2GRAY since we needed to convert the BGR colour space to grayscale in our situation. [38]

1. **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 in order to lessen the high frequency noise and edges prior to applying edge detection. The pixels that give an image its brightness and colour fluctuation are actually the sources of noise; blurring actually removes those pixels, leaving them hazy and fuzzy. Many blurring operations, including basic blurring, weighted guassian blurring, median blurring, and bilateral blurring, are supported by OpenCV. However to reduce noise, the image soothing method that we employed is Guassian 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 Guassian blurring to take out the high-frequency components. This function is used to remove noise that roughly follows Guassian distribution, which causes the final image to become less and less artificially blurred. Although, Guassian blur is slightly slower than the other blurring methods of OpenCV but it seemed to be the better option as were working on the natural images. [39],[40],[41],[42]

This function's syntax consists of various arguments, where the first argument ‘src’ describes the source or input image, 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 guassian kernel standard deviation along horizintol direction / X-axis. The fourth argument describes the destination ‘dst’ or output image, and ‘SigmaY’ is the fifth arguement kernel standard deviation along 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. [43] & [44]

**iv)** **Edge detection**

[13] 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:

1. **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.
2. **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.
3. **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 [13]. 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

**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. [14] & [15]. 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[16].

**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. [16] & [17]

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

**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:

1. Gradient-based edge detectors
2. 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 [18]. 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. [19] & [20] & [21]. 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. [22] (Radhika chandwakar)

**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:

1. **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.
2. **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. (meaning)??
3. **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. [22] & [23]
4. **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.
5. **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. [22] & [24]
6. **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 utilise 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. [45] & [46]

1. **Drawing Contours:**

After extracting the contours by using findcontour() 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 colour. In the project, the main goal was to draw the contours across the boundary points of the front wall.

**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 the every wall comes ahead of it and take a turn eveytime 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 infront of the robot with a different color. The robot can then more easily recognise the wall based on the difference in coloration between the front and side walls.

Another approach would be to draw coloured 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 coloured 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, canny operator is already applied to the image, 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. [28]

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 mobiles phones and cameras because it has a very fast computing time. It can detect corners as well as blobs that’s why also considered as a blob detector. [29]. FAST detects features in an efficient way more than other feature detection algorithms like Haris corner, SIFT and SURF and thus suitable for resource-intesive application like real-time image processing in the project.

**Experimental, Modelling or Analysis Procedure:**

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 utilised in the project to carry out all necessary robot-related tasks, including taking the images of the maze and control 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 basically a software application that allows you to to access the Raspberry Pi command line interface by running it from your own laptop or desktop. By installing Putty, a connection has been built between 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 being transferred from laptop to Raspberry Pi by sending commands to it. [47] After the Raspberry Pi and laptop were successfully integrated, a Python setup was completed because Python was chosen for image processing in the project. In order 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. Prior to that, though, a set of representative images had to be taken in order 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. In order 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 in 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 the Python code. Left and right movements of the robot were effectively achieved by the coding.

The software part of the project comprises of 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 off by first importing the libraries. Four libraries that were imported are as follows:

* 1. Matplotlib.pylab
  2. Numpy
  3. CV2
  4. Math

Among above four libraries, CV2 is actually an OpenCV library which 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 comprises of multiple steps in the 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,coloumn). The image that we used has 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. After the image was uploaded, it had to be rotated because the images taken from the camera were all 180 degree reversed. To rotate the image, another OpenCV function ‘cv2.rotate()’ was used which contains two input arguemnts, 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, its always necessary to change the colored image to grayscale. Unlike colored image, the grayscale image has pixels that carries single color channel which is gray. Again, an OpenCV function ‘cv2.cvtColor()’ was used to convert the BGR image into 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 blurring function of OpenCV 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 of the image to get blurred. The last argument which is sigmax is set to be 0 representing the kernel standard deviation along x-axis. In the last step of image processing, a matplotlib function has been used ‘imshow’ which was actually 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 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.

**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.

- 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 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 minimum and maximum respectively. The minimum threshold value is taken as 40 where as 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 blurred (grayscale) image are greater 140, they are considered as strong edges and the pixels gradients less than 40 are rejected by declared as weak edges.

- The final canny image which is stored in a variable ‘canny’ is actually a binary image (black & white) having only two pixel values i.e. 0 represents black and 255 represents white. Since these pixel values representing 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 being saved in variables ‘y\_axis’ & ‘x\_axis’.

- Since in our image of the maze, three edges were detected which were in vertical direction therefore, it is needed to find out the maximum and minimum values of the vertical edges that are in 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.

- Once we are able to calculate the maximum and minimum values of y-axis, the next step is to set up a condition which will be calculating the number of edges for the corresponding axes. Therefore, ‘if’ condition is applied in which first we are taking the 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 the half of the maximum value of y-axis, numpy function ‘np.where’ finds out where the middle value of y-axis lies in ‘y-axis’ variable. When the required y-axis coordinate value is achieved, the corresponding coordinate values against the x-axis is fetched from ‘x-axis’ variable. As per the image that is taken, the maximum number of edges were three, therefore, 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 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 are set to be ‘3’ and ‘2’.

- A ‘for’ loop is implemented in the next step of 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 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, ‘for’ loop continues to run till the total number of edges with respect to y-axis and its corresponding x-axis values, reaches the num\_edge value. When ‘if’condition gets satisfied, it stores the single y coordinate value of the required edges in a variable ‘y’. Since there will always be number of edges greater than 1 and we will be getting more than 1 x-axis coordinate values, another numpy function ‘np.sort()’ is being used. It actually sorts the x coordinate values against the y-axis by arranging them in an ascending order and save this array in a variable ‘x’. Lastly, after getting the required results we don’t want ‘for’ loop to continue to run therefore, if the condition gets failed, the loop will be stopped to run using break command so that there will be no overwrite of ‘x’ and ‘y’ values.

- 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. In order to separate the front wall from the rest of the image, it is being cropped by adjusting both 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 wont be including that third element of the array. For our canny image, we subtracted 20 from the y-coordinate value, stored in variable ‘y’ and for 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 which we named ‘canny’. Lastly, the cropped image would be displayed using ‘imshow’ function and since the final cropped image was in binary, fixcolor was used to change the color from binary to RGB.

-The final step in 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 OpenCV function ‘cv2.findContours’ to find the contours in the cropped image ‘canny’. Cv2.findContours having three input arguements 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 the 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 end points 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 drawing function ‘cv2.drawContours’ having 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 contour index which is a parameter of drawing a contour which 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 color scheme (255,0,0). The final argument represents thickness of the lines of the drawn contours which we selected as ‘2’.

-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.

**Logics for movement:**

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 logics 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 travelling forward and make a turn to the right or left.

**-Forward Movement:**

The forward logic that has been designed for the robot mainly depends on detecting the wall that come 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 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.

**Forward Logic:**

**-**Logic starts with ‘while’ loop followed by 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, converting to grayscale and applying blurring to it in order to reduce noise. Moving on, canny edge detection function is being performed followed by finding the edges with respect to x and y axes, taking out the maximum and minimum values of y coordinates of the edges.

**-**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 ‘elseif’ condition.

- Now ‘elseif’ condition is applied for forward movement which first finds the half of the maximum value of y coordinate of the vertical edges and checks 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 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 its 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 are more than three. When this ‘elseif’ condition fails it goes to the else body.

-‘Elseif’ condition gets failed when the number of edges are not greater than 3 which means it becomes equal to or less than 3. This is when the logic enters into ‘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.

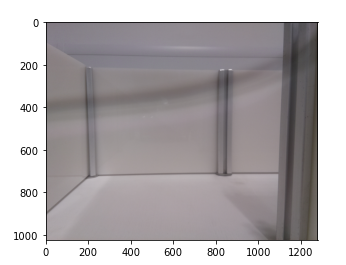
**Results:**

-The first major task that has been done in the working of the project was taking pictures from 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 a number of photographs from the camera and obtained the results shown below.

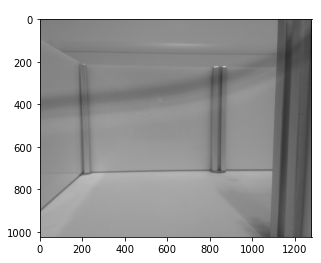


The above image was taken at the entrance of the maze and it is clearly shown in the image that the picture that was captured by the robot’s camera was upside down.

-Using OpenCV function, the image was rotated as follows.

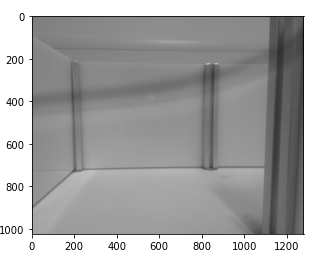


-The above rotated image was in BGR and in order to do edge detection on the image, we needed to convert it from BGR to grayscale. Therefore by using OpenCV function, following result was generated in grayscale.



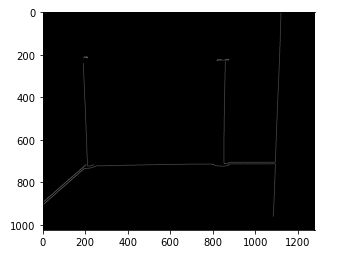
Since the original image was white or offwhite in colour, the colour change in grayscale is a little less pronounced, but it is still evident that the image has distinct grey shades in grayscale. This is due to the fact that grayscale only has one colour channel with values that range from 0 (black) to 255 (white).

-The result for a blurred image is given below.



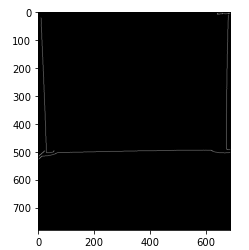
The original images always contain noise in them, thus blurring which is nothing more than a noise reduction is a smart strategy and needs to be done for improving edge detection outcomes.

- 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.

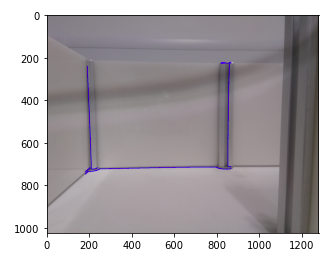


Because of the threshold settings that we chose for canny edge detection, the result above shows the edges pretty precisely. We first varied both the upper and lower limits of the threshold until we obtained the desired outcome mentioned above. The programmer's preference is all that determines the threshold 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 the 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. Howerver, the third vertical image wouldn’t be needed as it wasn’t the 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.



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



The above result shows the original image with the blue contours drawn on the edges of the front wall. Everytime the robot takes an image and gets the number of edges either 3 or 2, wall detection algorithm implements by finding the number of edges and checking them with the set value, then finding the required coordintes, crop them and draw 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 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 drawn on the edges.

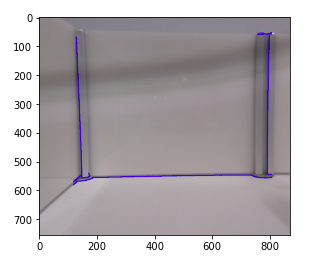
**Results of forward movement:**

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 robot’s camera to create the algorithm and develop the logics 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 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 while loop. Now since count is a variable having 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’. Keeping in mind that the variable count is only used here for the offline mode images. In the next command the count will be reduced by a value 1 but since ‘image 3’ is already stored in ‘img\_name’ this counter wont affect the first loop. As the first ‘if’ condition gets true only when the image with 0 edges are found therefore, the logic checks ‘elseif’ condition which also gets failed as it works for the number of edges greater than 3. Now the ‘img\_name’ gets verified by next condition which works when the number of edges reach 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 the ‘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 it still verified else condition and performed wall detection algorithm to the end. To check the forward logic we printed the following output.



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



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