School of Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Sharan Govinden Umavassee

2^{nd} May 2023

Low-cost vision system for autonomous outdoor robots

Project Supervisor: *Doctor Klaus-Peter Zauner*
Second Examiner: *Professor Christopher Freeman*

A project report submitted for the award of
MEng Computer Science with Industrial Studies

Abstract

The advancement of computer vision and machine learning has driven the development of robust navigation systems for autonomous agents. Fiducial markers, such as ArUco markers, are artificial landmarks that can be easily detected and tracked by computer vision algorithms, providing reliable reference points for navigation in complex environments. In this report, we present the development of a low-cost computer vision-based navigation system, focusing on accurately detecting and recognising ArUco markers arranged in a gate-like configuration to guide autonomous agents. We leverage the OpenCV library to detect ArUco markers.

Background research on fiducial markers, marker detection and computer vision-based navigation is provided, followed by a detailed methodology and implementation of the system's development. We then evaluate the system's performance through a series of experiments, testing its robustness under various conditions such as distance, angles, motion blur and lighting variations. The results demonstrate the system's high detection rate and robustness under most conditions, highlighting its potential for real-world applications. Challenges and limitations are identified, and future work is suggested to further enhance the system's capabilities. This dissertation aims to contribute to the growing body of research in computer vision-based navigation and foster new avenues for development in this rapidly evolving field.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

```
I have acknowledged all sources, and identified any content
taken from elsewhere.
```

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

```
I have used the open-source python libraries OpenCV and numpy
in the code development to implement specific functionalities
Images helping the explanation of the methodology for
fiducial marker systems were reproduced from public sources.
This was appropriately referenced and explained throughout
the report.
```

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

```
I did all the work myself, or with my allocated group, and
have not helped anyone else.
```

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

```
The material in the report is genuine, and I have included
all my data/code/designs.
```

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

```
I have not submitted any part of this work for another
assessment.
```

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

```
My work did not involve human participants, their cells or
data, or animals.
```

Acknowledgements

I would like to thank my project supervisor, Doctor Klaus-Peter Zauner, who supported and guided me throughout the project. Weekly meetings were very helpful to gain direct insights in his expertise on the subject of electronics.

I would also like to thank my second examiner, Professor Christopher Freeman, who provided me with valuable insights about related work during our meeting.

Contents

List of Figures vii

List of Tables viii

List of Listings viii

1 Introduction 1

2 Background 4
2.1 Fiducial Markers 4
2.2 ArUco Markers 6
2.3 Marker Detection 7
2.4 Raspberry Pi Zero Set-up 10
2.5 OpenCV and Raspberry Pi OS 12
2.6 Computer Vision-based Navigation 13

3 Design and Methodology 14

4 Implementation 17
4.1 Development Implementation 18
4.2 Migration to Raspberry Pi Zero 22
4.2.1 Hardware Architecture 22
4.2.2 Software Architecture 24

5 Testing and Evaluation 26
5.1 Marker and Gate Detection 26
5.1.1 Varying Distances 28
5.1.2 Varying Horizontal Angles 29
5.1.3 Varying Vertical Angles 31
5.2 Character Output Range 32
5.3 Motion Blur 34
5.4 Lighting Variations 35

6 Project Management **36**

7 Conclusion **38**

References **40**

A Project Brief **45**

B Code for Development Implementation **47**

C Code for Final Implementation on Raspberry Pi **54**

List of Figures

1.1 Path course 3

2.1 Types of Fiducial Markers 5
2.2 Steps required to detect ArUco markers 7
2.3 Image process for automatic marker detection 9
2.4 Raspberry Pi Zero 10
2.5 DFRobot 11
2.6 Zero4U USB hub 11
2.7 PuTTY SSH and Serial Connection 13

3.1 Selected ArUco tags 15
3.2 Formation of Gates 15
3.3 Marker Card Design 16

4.1 Drawing ArUco Marker on the frame 19
4.2 ArUco Marker Size Calculation 19
4.3 Gate Target Point 20
4.4 Gate Detected ASCII Character Display 21
4.5 Gate Not Detected 21
4.6 Raspberry Pi Zero Set-up 22
4.7 Software Flowchart 24
4.8 Target ASCII Character Output Example 25

5.1 Pin Board Set up 26
5.2 Detected Markers and Gates on Pin Board 27
5.3 Horizontal angle testing 30
5.4 Vertical angle testing 31
5.5 Diagonal angle testing 31
5.6 Character Output Test 32
5.7 Adjusted Scale 33

6.1 Gantt Charts 37

List of Tables

2.1 Advantages and disadvantages of fiducial marker packages . . 6

5.1 Marker detection test at varying distances 29
5.2 Marker detection test at varying angles 30
5.3 Motion Blur Test 34
5.4 Lighting Variation Test 35

Listings

2.1 Predifined ArUco dictionary with 50 markers with bit size 4X4 8
2.2 Custom Dictionary Generation 8
2.3 ArUco Marker Detection in image frame 8
4.1 Gate Formation Algorithm 20

Chapter 1

Introduction

The field of computer vision has experienced tremendous growth over the past few decades, driven by advancements in machine learning, digital image processing, and camera technology [1][2]. One of the key applications of computer vision is in navigation systems, where the ability to detect and recognise objects in real-time is critical for guiding autonomous vehicles, robots, or even humans in complex environments [3][4].

Fiducial markers, often utilized in augmented reality applications, are artificial landmarks that can be easily detected and tracked by computer vision algorithms [5]. These markers provide robust and reliable reference points, encoded in a planar pattern enabling reliable detection in images, accurate pose estimation and facilitating navigation in both indoor and outdoor environments [6]. ArUco markers, a specific type of fiducial markers, are square-shaped patterns with a unique identification code that allows for rapid detection and decoding [7].

In this project, we present the development and implementation of an ArUco gate detection system for computer vision-based navigation. The primary objective of this project is to create a system capable of accurately detecting and recognising ArUco markers arranged in a gate-like configuration, as shown in Figure 1.1, estimating their relative position and alignment, and providing navigation information to guide autonomous robots through these gates. This project has the potential to impact various fields, including robotics, unmanned aerial vehicles (UAVs), and outdoor and indoor navigation systems [8].

To achieve this objective, we leverage the power of the OpenCV library, a widely-used open-source computer vision library that provides a rich set

of tools and algorithms for image processing and computer vision tasks [9]. OpenCV includes a dedicated module for ArUco marker detection and decoding, which forms the basis of our ArUco marker computer vision-based navigation system [10].

In the following sections, we will provide a detailed background research on fiducial markers, ArUco markers, marker detection, Raspberry Pi, OpenCV and computer vision-based navigation. We will then present our methodology, outlining the algorithms and techniques employed in the development of the system. Finally, we will discuss the implementation, results, limitations, and future work to further enhance the system's capabilities and applications.

By the end of this dissertation, we aim to provide a comprehensive understanding of fiducial marker systems and ArUco marker detection systems and their potential use in computer vision-based navigation, opening new avenues for research and development in this rapidly evolving field.

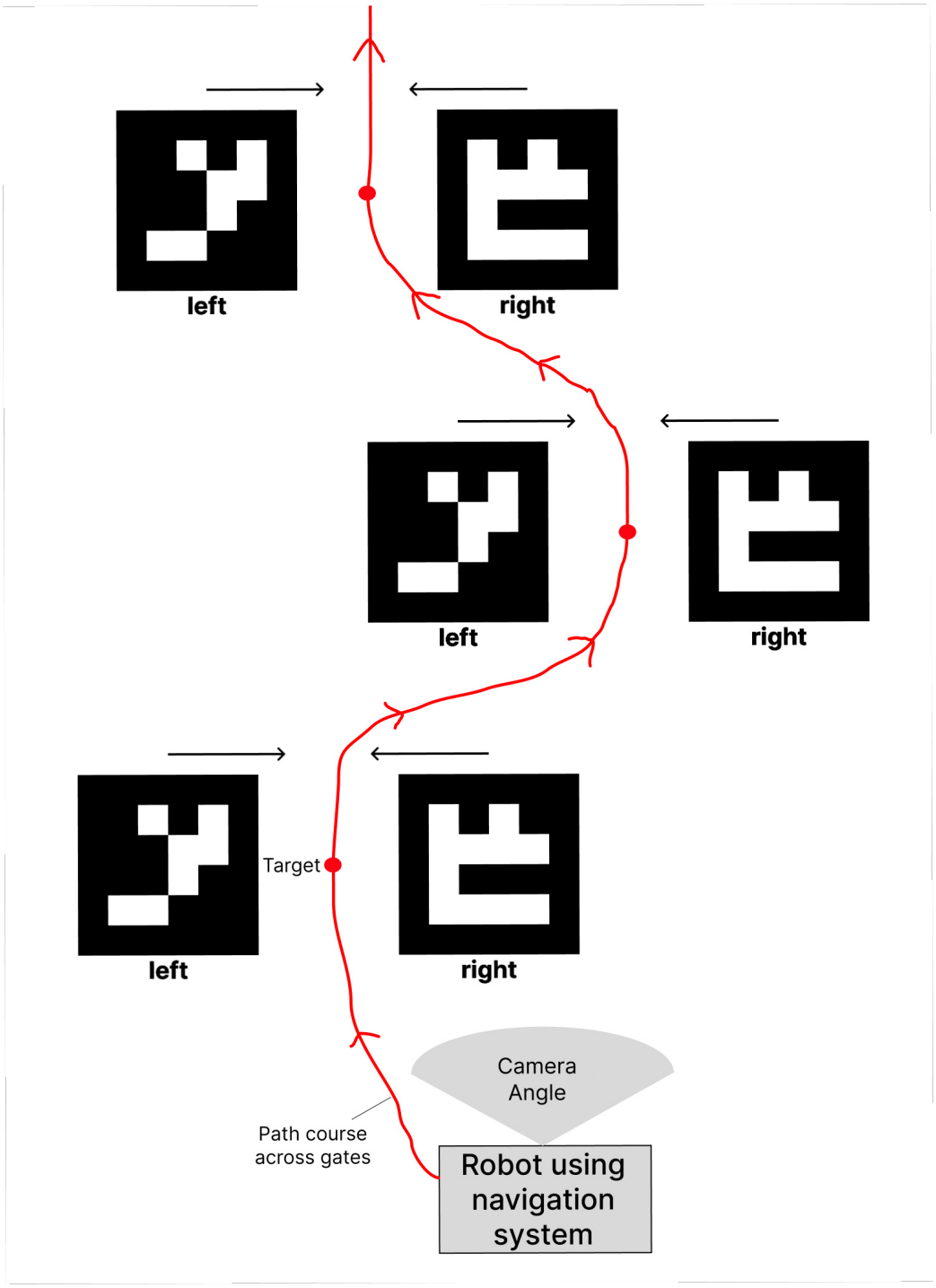


Figure 1.1: Overview of the path course and gate-like configuration of ArUco markers for a robot using the navigation system

Chapter 2

Background

2.1 Fiducial Markers

Fiducial markers refer to artificial landmarks added to a scene to facilitate locating point correspondences between images, or between images and a known model [11]. They are 2D binary black and white patterns that computer vision algorithms can easily detect.

They are useful when object recognition or pose determination is needed with high reliability and when natural features are insufficient in quantity and uniqueness. Fiducial markers also offer a highly distinguishable pattern and have an encoding that acts as a fail-safe against misdetections. They provide a robust and reliable means of determining the camera pose in relation to the marker, enabling accurate position and orientation estimation in real-time. Their applications include augmented reality (AR), input devices for human-computer interaction (HCI) and robot navigation [11]. They are attractive because of their high reliability and small size.

Various types of fiducial markers exist, including square markers, circular markers, and natural feature markers. Among these, square markers have been widely used due to their simple geometric structure and ease of detection. The most commonly used fiducial markers are ARTag [12], AprilTag [13], ArUco [6] and STag [14], as shown in Figure 2.1 from **a** to **d**.

Most square shaped marker frameworks are based on the ARToolKit framework [15], initially created for AR video conferences [5]. ARtag is based on ARToolKit but uses digital coding theory to create the marker's internal pattern [16].

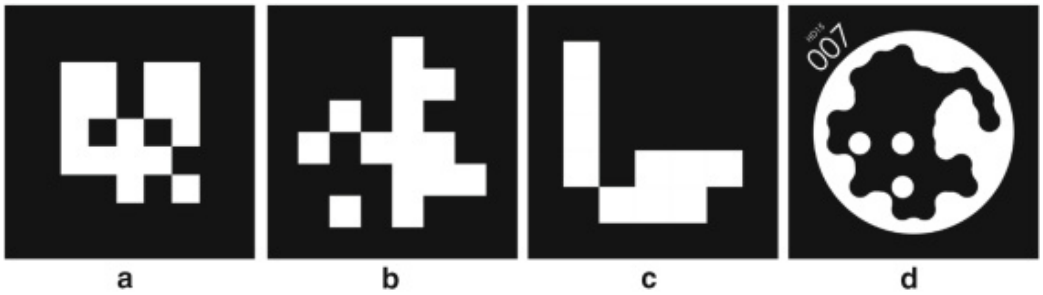


Figure 2.1: Types of Fiducial Markers: **a** ARTag, **b** AprilTag, **c** ArUco , **d** STag. Reproduced from [16]

AprilTag and ArUco are both instances of marker systems that build on the ARTag framework. They each offer several improvements on the framework. AprilTag introduced a graph-based image segmentation algorithm that analyses gradient patterns on images and a new programming system to take care of issues stemming from the 2D barcode system. Because they deal with the issues of incompatibility with rotation and false positives in outdoor applications, AprilTags provide high robustness against occlusion and warping and a decreased quantity of misdetections [16].

ArUco markers, similar to their "sibling frameworks", offer the possibility of using predefined libraries. However, their major feature is that they allow users to create configurable libraries. This allows users to generate custom libraries based on their specific needs and only comprising markers with maximum Hamming distance [16]—the distance between two binary vectors of equal length, that is, the number of positions at which the corresponding bits in the patterns are different [17]. ArUco's capacity to create custom libraries of desired length removes the need to include all possible markers in standard libraries, hence offering a low-computational cost.

A number of studies [18][19] have compared the different fiducial marker systems available. In Michail Kalaitzakis et al.'s paper [16], the authors experimented on the following monochromatic and square shaped frameworks: ARTag, AprilTag, ArUco and STag [14]. They tested performance metrics such as varying marker distance, varying marker orientation, motion blur and computational cost. Table 2.1 displays a summary of the results obtained from the experiments carried out outlining the advantages and disadvantages of each marker system.

Table 2.1: Main advantages and disadvantages of the four evaluated packages in Michail Kalaitzakis et al.'s paper [16]

Marker	Advantages	Disadvantages
ARTag	- Lowest computational cost	- Low detection rate for single markers - Extreme outliers & high standard deviation in marker bundles
AprilTag	- Great orientation results - Great detection rate - Good position results	- Most computationally expensive - Most sensitive to motion blur - Worst results in the non-planar set-up
ArUco	- Good position results - Great detection rate - Low computational cost for single markers - Good orientation results	- Sensitive to smaller marker sizes - Sensitive to larger distances - Computational cost scales with multiple markers
STag	- Great position results and detection rate - Good orientation results	- Sensitive to smaller marker sizes and larger distances

2.2 ArUco Markers

ArUco markers have gained popularity in computer vision applications due to their simplicity, robustness, ease of detection and rapid decoding. These black and white square binary patterns provide a balance between efficiency and accuracy, allowing for rapid and reliable marker identification and tracking. ArUco markers are invariant to scale, rotation, and illumination changes, which makes them suitable for real-time applications like augmented reality, robotics, and navigation [6]. They can be easily generated, and their size and pattern can be customized to suit specific requirements. Additionally, ArUco markers support error detection and correction, further enhancing their reliability in challenging conditions in outdoors environment such as lighting variations and camera motion blur.

ArUco markers have been used in a wide range of applications, such as robot localisation and tracking [6]. In robotics, ArUco markers have been employed to localise robots within their environment, facilitating navigation tasks and improving overall system performance [20]. In unmanned aerial vehicles (UAVs), ArUco markers have been utilized for precision landing and way-point navigation [21]. The use of ArUco markers in augmented reality applications enables the overlay of virtual objects onto real-world scenes, enhancing user experience and interaction [5][22].

2.3 Marker Detection

This section explains how to detect the markers in an image using OpenCV [9]. We aim at extracting the binary code from the detected square markers.

OpenCV is a widely-used open-source computer vision library that provides a comprehensive set of tools and algorithms for image processing and computer vision tasks [23][9]. Example applications: to detect faces or identify objects in an image and classify human actions in videos.

OpenCV includes a dedicated module for ArUco marker detection [24] and decoding [25][26], making it an ideal choice for this project. The ArUco detection module in OpenCV employs the fast corner detection algorithm and adaptive thresholding for robust marker detection [7].

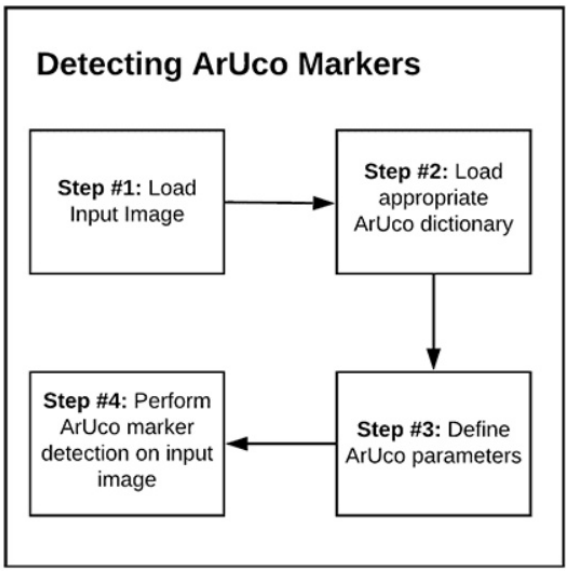


Figure 2.2: Flowchart of steps required to detect ArUco markers with OpenCV.
Reproduced from [27]

Marker dictionaries, or libraries, broadly refer to a set of a specific type of markers. We use them to generate and detect ArUco markers. There are 21 different ArUco dictionaries built into the OpenCV ArUco dedicated module. They are categorised based on number of bits and number of markers in the dictionary, ranging from 50 to 1000. The NxN value is the 2D bit size of the ArUco marker. The integer M following the grid size specifies the total number of unique ArUco IDs that can be generated with the dictionary.

ArUco markers can be generated through code or generator tools such as the ArUco Marker Generator project by Oleg Kalachev obtained through their open-source github project [28].

```
cv2.aruco.Dictionary_get(cv2.aruco.DICT_4X4_50)
```

Listing 2.1: Predifined ArUco dictionary with 50 markers with bit size 4X4

As mentioned in section 2.1, the ArUco marker system also offers the possibility to create configurable libraries to adhere to the user's specific needs. The method cv2.aruco.Dictionary_create(X, Y) takes two parameters X and Y and creates a customised dictionary composed of X markers of YxY bits. In cases where we require less than 50—the minimum number of markers available in predefined dictionaries—or a specific number of unique markers, the ArUco marker system greatly helps as it heavily reduces the computational cost [16].

Markers from the customised dictionary have maximum Hamming distance between them. The Hamming distance between two equal-length strings of symbols is defined as the minimum number of errors or substitutions that can transform one string into the other [29]. In simpler terms, it is the difference in bits between two bit sequences. This ensures that the markers will be distinguishable from one another considering variables in the environment or algorithm that might impact the detection of markers.

Detection process of ArUco markers for custom dictionaries for four ArUco markers with bit size four x four

```
arucoDictionary = cv2.aruco.Dictionary_create(4,4)
arucoParameters = cv2.aruco.DetectorParameters_create()
```

Listing 2.2: Custom Dictionary Generation

```
(corners, ids, rejected) = cv2.aruco.detectMarkers(
frame,arucoDictionary, parameters=arucoParameters)
```

Listing 2.3: ArUco Marker Detection in image frame

After performing the ArUco marker detection algorithm, we obtain three key information that helps us identify the marker:
corners: a list containing the (x,y)-coordinates of the detected ArUco markers' corners
ids: a list of IDs of the detected markers
rejected: a list of potential marker candidates that were found but rejected due to the inner code of the marker being unable to be parsed

The steps involved in the detection process [6] are:
Image Segmentation: extracts the most important contour using a local adaptive thresholding approach
Contour extraction and filtering: obtains the polygons in the marker image
Marker Code extraction: extracts the internal code from the inner regions of the extracted contours
Marker identification and error correction: determines which marker candidates obtained belongs to the dictionary

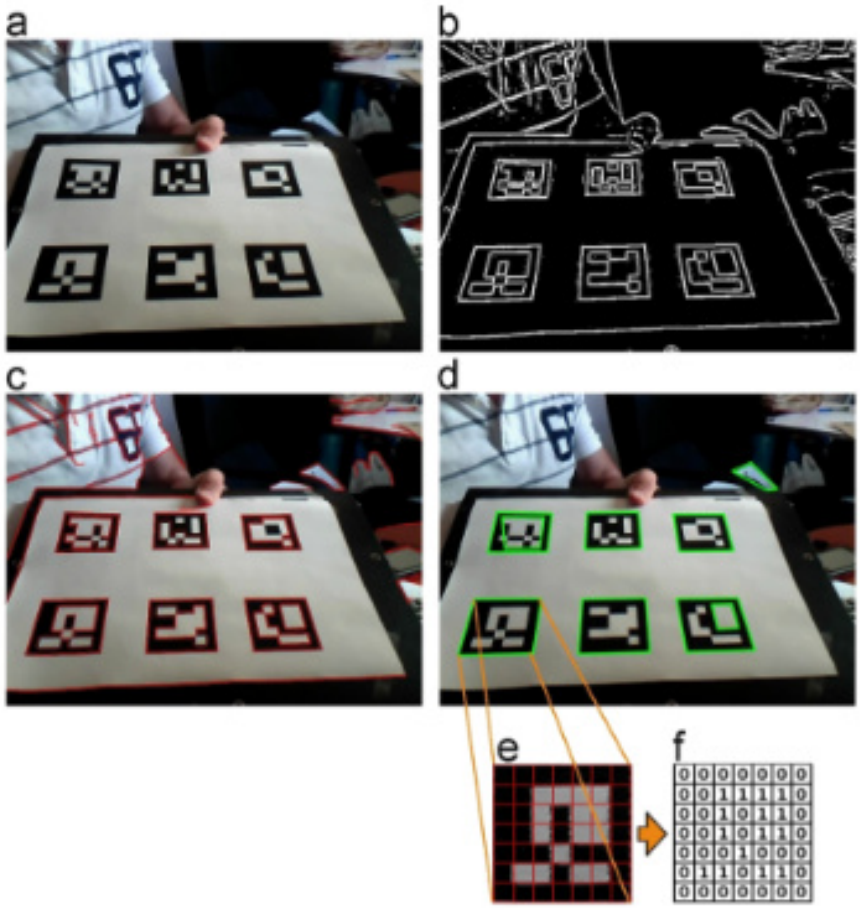


Figure 2.3: Image process for automatic marker detection. (a) Original image. (b) Result of applying local thresholding. (c) Contour detection. (d) Polygonal approximation and removal of irrelevant contours. (e) Perspective transformation. (f) Bit assignment for each cell. Reproduced from [6]

2.4 Raspberry Pi Zero Set-up

This section provides an overview of the hardware components used in the project. We use a Raspberry Pi (RasPi), specifically a Raspberry Pi Zero (Pi Zero) [30] model—a low-cost single-board small computer. The Pi Zero is listed at £5 [31][32] and has a 1 GHz single-core Central Processing Unit (CPU) and 512 MB of Random Access Memory (RAM). It has an unpopulated 40-pin General-Purpose Input/Output (GPIO) connector—with the same pinout as other Raspberry Pi models A+/B+/2B, a micro-SD card slot for storage, mini-HDMI (High Definition Multimedia Interface) socket for 1080p60 video output and micro-USB (Universal Serial Bus) sockets for data and power [32]. Figure 2.4 shows a picture of a Pi Zero.

Figure 2.4: A Raspberry Pi Zero. Picture Credits to The Pi Hut [31].

Alongside the Pi Zero, we have a DFRobot IO (Input/Output) Expansion HAT (Hardware Attached on Top) 40 pin header [33] that provides a series of additional ports to the Raspberry Pi. Figure 2.5 shows the set up of ports and pins on the IO Expansion HAT. The most important port to us is the Universal Asynchronous Receiver/Transmitter (UART) port that enables a serial connection with the Pi Zero.

To connect to the Pi Zero through UART serial connection, we use a USB to serial cable. Our choice for a serial cable is the USB to Hi-Speed UART Serial Adapter Cable with Embedded Electronics—part number: C232HD-DDHSP-0—from FTDI Chip [34]. Section 2.5 provides more details about the software process involved in connecting to the Raspberry Pi.

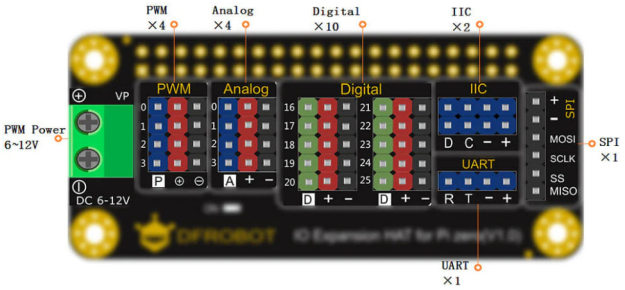


Figure 2.5: DFRobot IO Expansion HAT displaying the set up and purpose of the pin set. The board classifies the I/O ports in Digital, Analog, I2C, PWM, UART and SPI ports. Picture Credits to The Pi Hut [33].

The next piece of hardware used is a Zero4U USB hub by UUGear [35] with 4 USB ports mounted to the Pi Zero back-to-back. Figure 2.6 shows a picture of a Zero4U. Its 4 USB ports can transfer data in USB 2.0 high-speed. Because the Pi Zero individually only has a micro-USB port, the Zero4U aids us to connect USB peripherals, such as a keyboard and a USB camera, to the RasPi, acting as a USB On-The-Go (USB OTG).

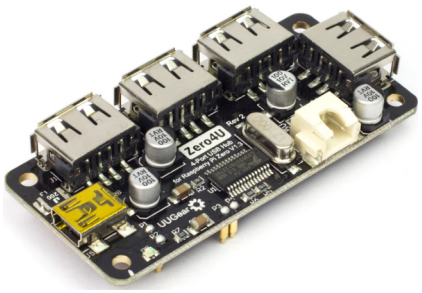


Figure 2.6: A Zero4U USB hub. Picture Credits to Pimoroni.

2.5 OpenCV and Raspberry Pi OS

This section describes the software setup in the project. It provides information about the installation of the OpenCV package and the Raspberry Pi Operating System. It also shows the process of using the software PuTTy to connect to our Raspberry Pi using serial connection and SSH.

OpenCV, being the heart of the code development, is an open-source computer vision library that provides tools and algorithms for image processing and computer vision tasks. It is installed using pip [36], a package installer for Python. OpenCV has two installable packages: *opencv-python* that offers the main modules and basic functionalities or *opencv-contrib-python* that provides the full package of OpenCV. *opencv-contrib-python* includes additional modules [37], containing the ArUco dedicated module which is a crucial part of the code development. We use the command *pip install opencv-contrib-python* to install the OpenCV package with the ArUco module. Installing OpenCV on the RasPi requires additional system packages such as libjasper-dev and cmake. A tutorial [27][38] explaining how to install OpenCV on a Raspberry Pi was used that includes a full list of site packages and details every step required to successfully install the *opencv-contrib-python* package.

The Raspberry Pi Operating System, an operating system based on Debian, can be installed on a micro-SD card using the software Raspberry Pi Imager [39]. The operating system has two installable versions. The Full Raspberry Pi OS provides a desktop environment with the full operating system capacity and recommended software for convenience. The Raspberry Pi OS Lite or Headless Raspberry Pi OS only includes a shell terminal to input commands manually using Linux. It is similar to the Command Prompt in Windows OS that allows for manual command-line inputs.

Once the OS is installed and configured, there are several ways to connect to our Raspberry Pi Zero: directly using a display, micro-HDMI to display cable and peripherals, or remotely over a Secure Shell (SSH) connection—a network communication protocol that allows two computers to communicate—or over serial connection.

The SSH connection can be made using the Raspberry Pi's Internet Protocol (IP) address or configured hostname and the command *ssh username@<The Pi's IP Address>* on the second computer. If the computer used does not have SSH, we can use the PuTTY application to help connect to the RasPi. To use SSH on PuTTY, select "SSH" under "Connection Type", and type the Pi's IP address under "Host Name (or IP address)".

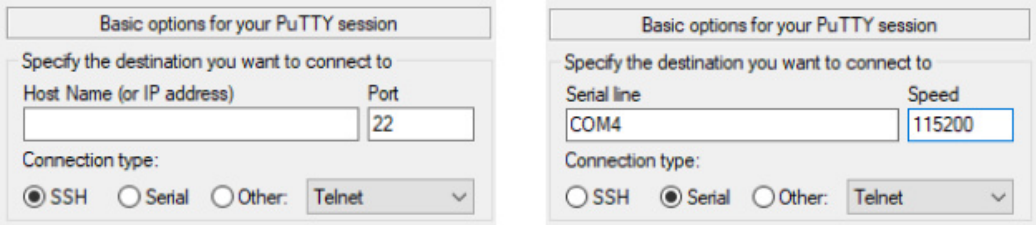


Figure 2.7: Example of PuTTY app to create SSH and Serial connection

The serial connection can be made using the software PUTTy and a serial cable connected to a USB port on the computer. To use the serial line on PuTTy, select "Serial" under "Connection Type", and type the port number used to connect the serial cable—which can be found using Windows Device Manager—in "Serial line" and the desired baud rate in "Speed". The baud rate is the rate at which information is transferred in the communication channel. It indicates the maximum number of bits that can be transferred per second. Figure 2.7 displays the section of the PuTTY software used to input these information for both the SSH and serial connection. Once connected and logged in using our configured username and password, we then use Linux-written commands to operate the RasPi.

2.6 Computer Vision-based Navigation

Computer vision-based navigation leverages the power of image processing and computer vision algorithms to guide autonomous agents through complex environments [40]. The use of fiducial markers has been shown to improve navigation performance by providing accurate reference points [41].

Despite the advantages of computer vision-based navigation, several challenges remain to be addressed. These challenges include occlusion, lighting variations, and camera motion blur, which can affect the detection and decoding of fiducial markers [42][43]. To overcome these challenges, various approaches have been proposed, such as adaptive thresholding, multi-scale detection, and robust estimation techniques [7].

The research literature demonstrates the potential of ArUco markers in facilitating computer vision-based navigation tasks. By understanding the existing research and techniques, we can develop and implement an effective ArUco gate detection system as part of this project. The following sections of this report will detail the methodology, system design, implementation, and evaluation of the proposed project.

Chapter 3

Design and Methodology

In this chapter, we propose and describe our methodology and concepts to understand for the success of this project. We aim to create a low cost computer vision-based navigation system for outdoor autonomous robots. We set fiducial markers, acting as gates, on a path and the vision system will provide the robot with navigational/directional information.

The ArUco marker system is selected for our fiducial markers because of its low computational cost, great detection rate in real-time applications and configurable dictionary functionality and reliability in challenging outdoor conditions as mentioned in section 2.2. The black and white pattern, enabling rapid detection and decoding, allows for a greater chance for real-time detection on a small low-cost computer.

A customised dictionary, made for the project's specific purpose, of 4 ArUco markers with 4x4 bit size is generated, as shown in section 2.3. Using the four-by-four bit size, the smallest marker bit size available for ArUco, offers a bigger and more distinguishable pattern. The four markers required are the two directional markers: left and right markers, and the start and stop markers. Figure 3.1 shows the four ArUco markers generated by our custom dictionary. Each ArUco ID is appointed to a specific marker for uniqueness. The configurable dictionary and small number of tags provide markers with the maximum Hamming distance, helping to make them distinguishable from one another, which reduces chances of misdetection and false positives [16].

OpenCV allows us to use computer vision algorithms to load and process the image frames and, with its ArUco detection module, enables us to generate the custom dictionary and detect the markers as explained in section 2.3.

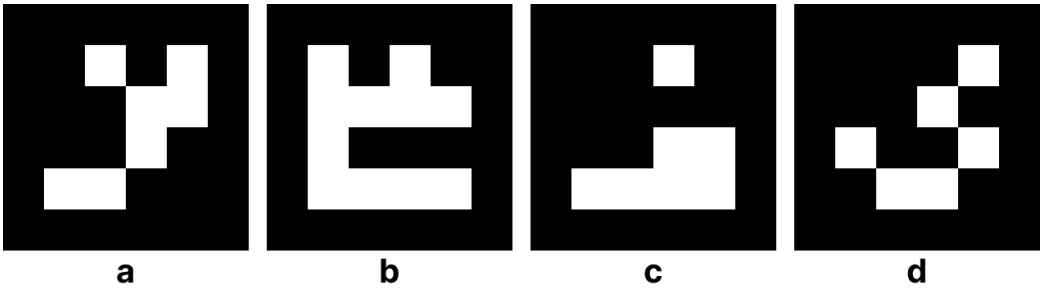


Figure 3.1: Selected ArUco tags generated by custom dictionary: **a** left, **b** right, **c** start, **d** stop markers.

The process is described as follows:
Once the start tag is detected, the vision system will start looking for the two directional markers. A gate is formed if two detected markers are left and right markers, in parallel, have approximately the same edge size and are in their correct respective positions—left marker on the left and right marker on the right, as shown in Figure 3.2. Figure 3.3 displays card designs for each ArUco tag, made from previous work [44].

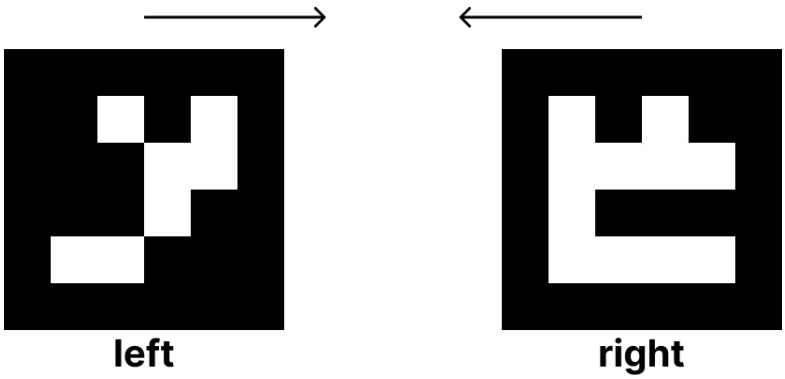


Figure 3.2: Formation of Gates: using the Left and Right ArUco markers

A list of detected markers is created and sorted by size of marker to obtain the closest tags relative to the camera. The closest markers are checked against the gate requirements. The system then computes the target point by calculating the midpoint of the line segment across the diagonally opposite corners of the two markers. This target point is then used as reference with the camera centre point to calculate the direction to guide the robot towards.

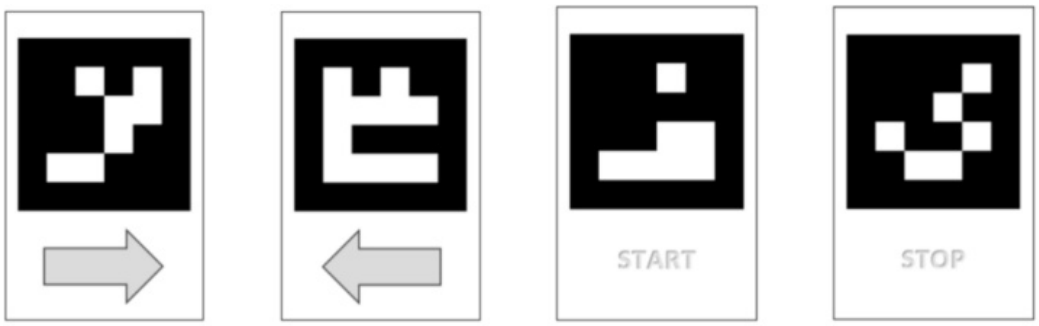


Figure 3.3: Marker Design: The cards include the markers from Figure 3.2 and a visual aid that helps setting up the path correctly.

A Raspberry Pi can be used to run the navigation system which will provide the robot with directional instructions. This can be done in two ways:

1. through its GPIO pins—one pin to go straight, one pin to go left and one pin to go right.
2. over UART serial connection—encoding the target point's x-coordinate as an ASCII character by applying a scaling factor of 25 divided by the width of the frame to the coordinate and converting it to an uppercase character.

The methodology demonstrates the concepts that would be used in implementing the computer vision-based navigation system. By understanding the marker design and setup techniques, we can effectively carry out each step in our implementation phase in the following section.

Chapter 4

Implementation

In this chapter, we detail the implementation of the computer vision-based navigation system, which was carried out in two stages:

1. First Implementation Stage: Development on a computer with an 8-core CPU, 16 GB of RAM and using the Windows 10 operating system.
2. Second Implementation Stage: Migration to a Raspberry Pi Zero with a 1 GHz single-core CPU and 512 MB of RAM.

The following sections describe the software design for both implementation stages. The software for this project was first developed on a more computationally powerful machine to facilitate code development. The desired outcome of the project is to create a low computational cost software that can be run on a low-processing computer. Implementing the program on a more resourceful machine first allows for more capacity in developing, editing and optimising the software as required. It gives more resource for the program to do tasks such as image processing and adaptive thresholding—done by the ArUco module.

The second implementation stage involves adapting and optimising the program to be transported onto the Raspberry Pi Zero and designing the physical layout for the system. We describe the process to set up and connect to the Pi Zero, download the required packages and run the code on the system with examples of results produced.

4.1 Development Implementation

To develop the software, we use Python version 3.11 [45][46] as our programming language with the following packages/libraries:

- *OpenCV*: A popular library for computer vision tasks, used for marker detection and image processing [23][26].
- *NumPy*: A library for numerical computing in Python, used for mathematical calculations [47].
- *sys*: A standard Python library used for system-specific operations, such as command-line arguments [48].

The ArUco dictionary is initialised using the OpenCV ArUco module. As described in section 2.3, we use the cv2.aruco.Dictionary_create() method requiring two parameters—for the number of markers to include and the number of bits or pixels in a marker's edge—to provide four markers of size four x four bits. We set up the camera using the OpenCV cv2.VideoCapture() method to allow for a real-time video stream and read the image frame inside of an infinite loop that runs until program exit.

After reading the frame and tweaking its dimensions, the marker detection process is carried out by the method cv2.aruco.detectMarkers() as explained in section 2.3. The detectMarkers() method gives us information about all corners and ID of each detected marker. This information will represent each marker detected as a form of corners and ids. To facilitate the display of markers on the image frame shown in real-time using the cv2.imshow() method, we draw a bounding box for detected ArUco marker using the cv2.aruco.drawDetectedMarkers() method and display their respective ID, as shown Figure 4.1.

Since we require specific processes involving the detected markers, a Class gate is used to represent each marker. It takes the marker's corners and ID as parameters and initialises the size of the marker in pixels by calculating the length of the diagonal across the top-left and bottom-right corners of the marker, as displayed in Figure 4.2.

An object from Class gate is instantiated and appended to a list of markers that will represent all markers detected. The list of markers is then sorted by size in descending order, which updates for each frame loop. As a bigger marker in the image frame indicates that it is closer relative to the camera, this ensures having the closest marker always at the beginning of the list.

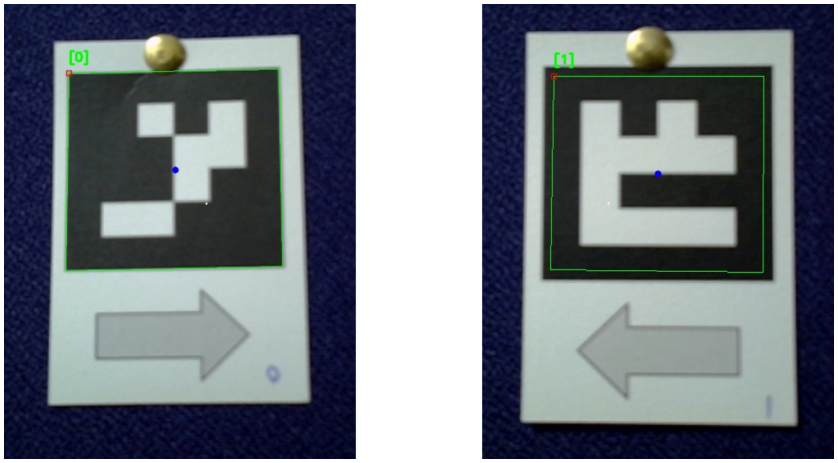


Figure 4.1: A bounding box, a blue circle at the centre of the detected markers and their respective IDs are drawn on the frame

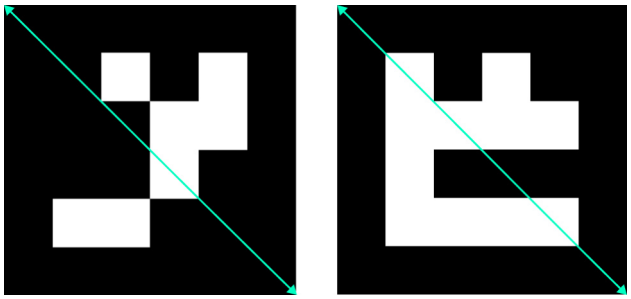


Figure 4.2: The diagonal line, drawn from the top-left to the bottom-right corners of the marker, used to calculate the size of the marker.

In our proposed methodology in chapter 3, we state the requirement for the start and stop markers in the ArUco dictionary to control our gate formation algorithm. The program checks for the start marker having ID 2 in the list of markers. It then moves on to checking the conditions that satisfy the formation of a marker gate.

It should be noted that the start and stop markers do not affect the functionality of the algorithms. The need for start and stop markers can be bypassed by structuring the algorithms to wait for at least two directional markers to be detected that would satisfy the gate requirement. They are used for control and convenience but are not crucial for the success of the project. Hence, this functionality was not implemented in the code development.

The next stage is to check the conditions to satisfy the formation of a gate. As

explained in chapter 3, a gate is formed if the two closest markers detected relative to the camera are the left and right markers with ID 0 and 1 respectively and aligned appropriately. The gate alignment is then checked by comparing their x-coordinate in the image frame and finding their locations respective to each other. A code snippet of algorithm is listed below.

```
"""Compare the IDs of markers in sorted list"""
if (tags[0].ID[0] == 0 and tags[1].ID[0] == 1):
    """Compare the positions of the markers"""
    if (marker0Center[0] < marker1Center[0]):
        # Gate detected
        # Target Point Calculated
        # Scale and convert the target point
        # x-coordinate to uppercase character
        asc = chr(int((target[0] * scale_factor) + 64))

elif (tags[0].ID[0] == 1 and tags[1].ID[0] == 0):
    if (marker0Center[0] > marker1Center[0]):
        # Gate detected
        # Target Point Calculated
        # Scale and convert the target point
        # x-coordinate to uppercase character
        asc = chr(int((target[0] * scale_factor) + 64))
```

Listing 4.1: Gate Formation Algorithm

This process checks the ID of the markers detected. If the first marker or the closest marker relative to the camera has ID 0 and the second closest marker has ID 1, or vice-versa, it compares their x-coordinates to gain their position respective to each other and assigns them as a gate if the left marker and right marker are positioned and aligned appropriately, as shown in Figure 3.2. A target point is computed calculating the midpoint of the line segment across the diagonally opposite corners of the two markers, as shown in Figure 4.3.

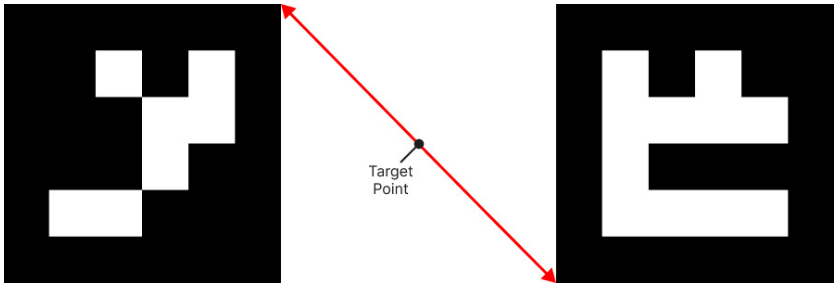


Figure 4.3: The target point is the midpoint of the line segment between diagonally opposite corners of the two markers.

To compute the directional output given to the robot, we use the target point's x-coordinate. If a gate is detected, we use an ASCII character (A to Z) to represent the navigational information dictating the position of the target in the image frame. ASCII (American Standard Code for Information Interchange) is an 8-bit character code where each individual bit represents a unique character: punctuations, alphabets, numbers, and other special characters [49]. We allocate 25 sections (A (65) to Z (90)) across the frame x-axis to dictate the position of the target. A scale factor of 25 divided by the width in pixels of the frame is applied to the target's x-coordinate. The target coordinate is scaled and converted to an ASCII uppercase character using the formula: $asc = chr(int((target[0] * scale\_factor) + 64))$.
A red circle is drawn on the frame at the target point's coordinates alongside its corresponding character, as shown in Figure 4.4.

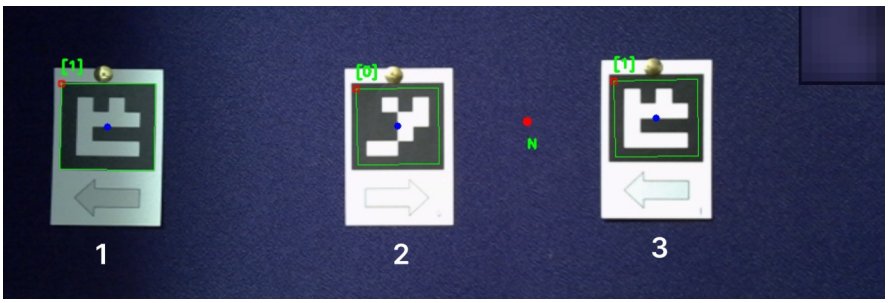


Figure 4.4: A gate is detected between marker 2 and 3 and the ASCII character of the scaled target coordinate, N, is displayed on the frame

If no gate is detected, a hyphen ("-") will be given as navigational output, as shown in Figure 4.5. This enables the user to control and personalise their solution in treating the hyphen ("-") case to move their robot.

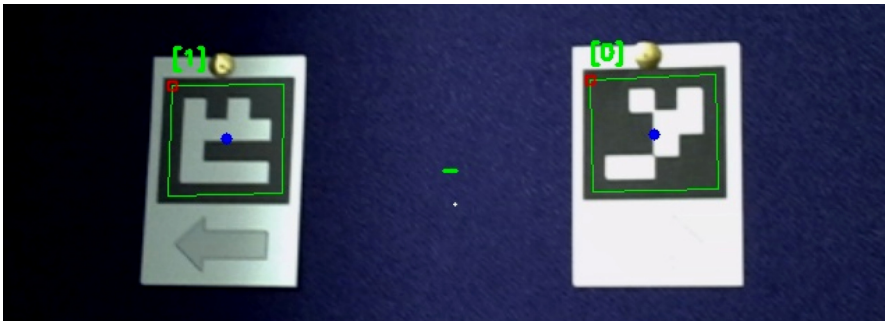


Figure 4.5: A gate is not detected between the markers in image. The ASCII character output and displayed is "-".

Upon testing the system, it was noted that the gate formation algorithm had issues for cases of more than two markers in the frame. This was caused by the algorithm comparing only the two closest markers in the image. Hence, the process was adapted to reflect this scenario of three or more detected markers to account for a decision check made to compare three markers with each other to find a valid gate.

The program then outputs a summary of the information providing the coordinates of detected markers and the target point of the gate in the image and its corresponding character. It then terminates after the clean up process for the OpenCV video capture frame.

4.2 Migration to Raspberry Pi Zero

The second stage of development is to migrate our program to a Raspberry Pi Zero. This constitutes stripping away computation-costly parts of the program to allow for an efficient detection process. The aim of this stage is to allow the output of navigational information using the serial line. As stated in chapter 3, the directions can be given to a connected robot using either the GPIO pins or the UART serial connection. The serial connection proved to be the most efficient method as it is easy to implement and handle on the robot's side.

4.2.1 Hardware Architecture

Section 2.4 provided an overview of the hardware components used in the project. This section now describes the physical layout of the system designed, as shown in Figure 4.6.

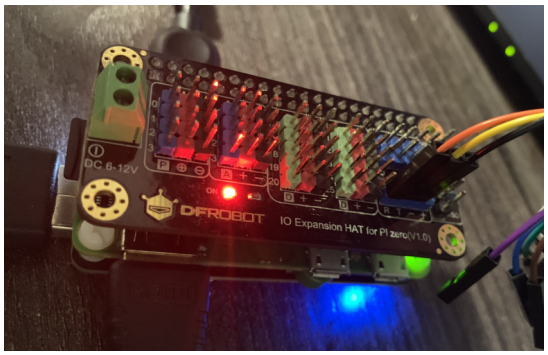


Figure 4.6: Raspberry Pi Zero Set-up showing connection to a display monitor using a mini-HDMI to HDMI cable and serial connection for output

The set-up consists of a Raspberry Pi Zero, a DFRobot IO Expansion HAT, a Zero4U USB hub, a 16 GB micro-SD card, USB to Serial line cable, a display monitor, a mini-HDMI to HDMI cable, a keyboard, a power cable and a USB camera. We use a mini-HDMI to HDMI cable to connect our Pi Zero to a display monitor. The Zero4U USB hub allows us to connect a keyboard to operate the Pi Zero headless terminal.

It was noted that the Zero4U is supposed to take power from the Pi Zero and work in 'self-power-mode' [50], however, this did not provide enough power to the USB hub to recognise the peripherals. Hence, a cable was made to provide enough power through a power plug to the Zero4U to connect to the peripherals. This enables us to connect the USB camera to detect ArUco markers, which removes the need to specifically use a Pi Camera to use the video feed.

To connect to our Pi Zero over serial connection, the USB to serial cable connects to the UART pins on the IO Expansion HAT. The black, yellow and orange cables of the serial cable, representing ground (GND), reception (RX) and transmission (TX), connect to the UART pins labelled "-", "T" and "R" respectively on the IO Expansion HAT. The black ground cable is connected as reference for the signal voltages on RX and TX.

The following parameters are needed to connect to the Raspberry Pi console over UART serial connection using PuTTy:
Port: COM4
Speed (baud rate): 115200
Bits: 8
Parity: None
Stop Bits: 1
Flow Control: None

We reserve the serial communication channel for the output of navigational information. The serial line can be used to log into the Pi Zero and connect to the shell terminal. Hence, we use the command *sudo raspi-config* to access the configuration tool and disable the boot messages for the serial interface.

The Raspberry Pi Zero can also be connected using SSH connection. We use the command *ssh username@<Pi's IP Address>* on the second computer. We can also use the PuTTY application to connect to the RasPi, as mentioned in section 2.4. This allows a shell terminal to be used directly on the connecting computer similar to a virtual machine (VM).

4.2.2 Software Architecture

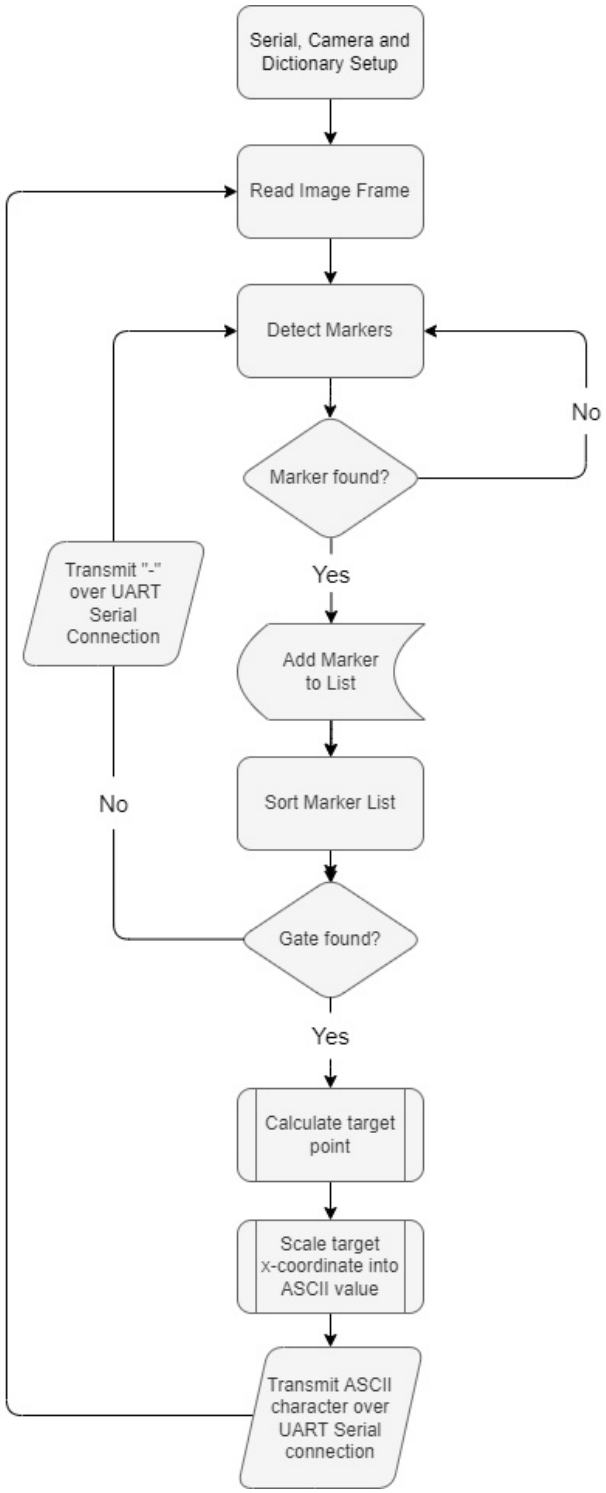


Figure 4.7: The flowchart shows the rough steps that the system goes through. It involves one main loop that loops over the image frames to read the video stream and two conditional statements that checks the detection the markers and the formation of a valid gate.

Figure 4.7 shows a flowchart of the general steps in our vision system for the Raspberry Pi Zero. The code development is described in section 4.1. To adapt our program to our Pi Zero system, we add the *pyserial* Python module to encapsulate the access for the serial port through Python properties [51][52]. We set up the UART serial connection using the pyserial module with the required parameters.

```
ser = serial.Serial(
    port='/dev/serial0',
    baudrate=115200,
    bytesize=serial.EIGHTBITS,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    timeout=1)
```

After setting up the serial connection, we read the image frames in a loop, detect the markers and run the gate formation algorithm as detailed in section 4.1. However, upon migrating our code to the Pi Zero, because the Pi Zero has lower computational power, we remove processes unnecessary to our final serial output—we do not draw the frame and marker bounding boxes and do not output the summary of information to the console. We use the *ser.write(asc.encode())* line to encode the *asc* ASCII character representing the target coordinate to the serial line.

After connecting to our Raspberry Pi Zero over the serial connection using PuTTy, the output of the target will appear as in Figure 4.8. This output will then be transmitted over the serial line to an autonomous agent using a serial cable. The robot would then move accordingly per the character received from our navigation system by implementing an algorithm that control its motor functions using those characters.

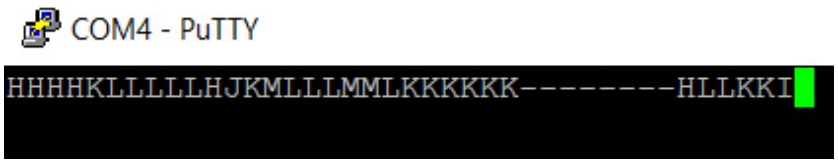


Figure 4.8: Example of the output display on PuTTy serial connection for the target ASCII Character as gate are being detected. When a gate is detected, an uppercase character ("A"-"Z") is encoded on the serial line. When no gate is detected, a hyphen ("-") is encoded.

Chapter 5

Testing and Evaluation

This chapter details a series of tests that were carried out throughout the project to prove the robustness of the system developed.

5.1 Marker and Gate Detection

Experiments were conducted to test the detection of ArUco markers and the formation of gates, using a set-up of markers placed on a pin board, as displayed in Figure 5.1. The detection test was run on the computer used for the development implementation in section 4.1 using code in Appendix B.

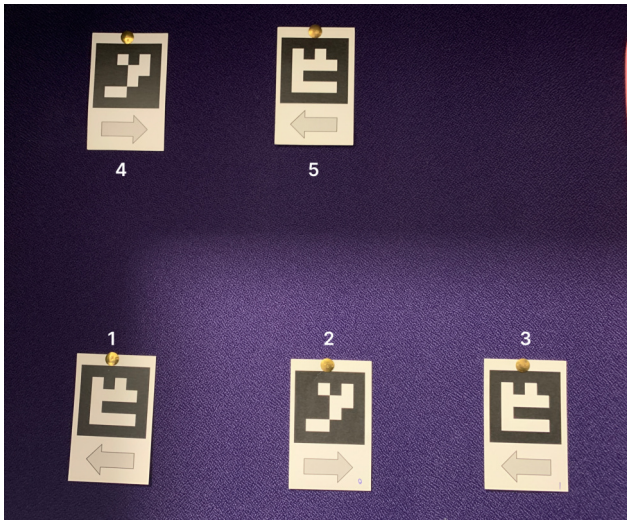


Figure 5.1: ArUco markers set-up on pin board to test the detection algorithm

The desired outcome of the marker and gate detection test was successfully achieved with a bounding box drawn around all ArUco markers indicating they were detected and a target point with its corresponding ASCII character for a valid gate between markers 2 and 3, as shown in Figure 5.2.

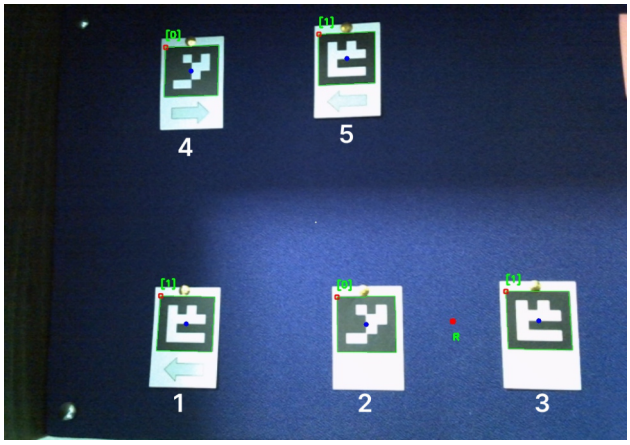


Figure 5.2: Marker and Gate Detection Test to detect ArUco markers and check for a valid gate formed on pin board set-up

A series of tests were carried out to check for successful detection of markers at varying distances and angles, as described below. Here's a step-by-step explanation of how to calculate the detection rate:

- Conduct multiple trials for detecting markers or gates under the given conditions—different distances and angles.
- Count the number of successful detections in each trial. A successful detection means that the system correctly identified the marker or gate.
- Calculate the detection rate by dividing the total number of successful detections by the total number of trials, then multiply by 100 to obtain a percentage.

The detection rate serves as a performance metric for the detection system, indicating its effectiveness in identifying markers or gates under various conditions. A high detection rate implies that the system is robust and reliable, while a lower detection rate indicates that the system may struggle to detect markers or gates under specific conditions or configurations. We allow a total of 25 attempts to detect the markers and counting the number of successful detections.

5.1.1 Varying Distances

In this experiment, we aimed to evaluate the performance of the ArUco gate detection system concerning the distance between the camera and the markers or gates. The distance at which markers and gates can be accurately detected and decoded is a critical factor in determining the effectiveness of the system in real-world applications.

We conducted the experiment by positioning the camera at a range of distances from the gate, varying from 10 cm to 200 cm, in increments of 10 cm. For each distance, we recorded the detection rate of the markers and the gate. The camera's angle of view was kept constant, facing the gate perpendicularly.

The results of the experiment demonstrated that the system could effectively detect and decode markers and gates at distances up to 200 cm. As expected, the detection rate and decoding accuracy decreased as the distance increased, but the system maintained an acceptable performance throughout the tested range. The following data summarises the results obtained from this experiment:

The results indicate that the detection system can effectively detect and identify markers and gates at a wide range of distances, with a steeper decrease from 150 cm. This versatility enables the system to provide reliable navigation information to autonomous agents operating in diverse environments and at various distances from the gates.

Table 5.1: Marker detection test results at varying distances

Distance (in cm)	Total number of trials	Successful detections	Detection Rate (%)
10 cm	25	25	100 %
20 cm	25	25	100 %
30 cm	25	25	100 %
40 cm	25	24	96 %
50 cm	25	25	100 %
60 cm	25	24	96 %
70 cm	25	23	92 %
80 cm	25	23	92 %
90 cm	25	22	88 %
100 cm	25	21	84 %
110 cm	25	21	84 %
120 cm	25	21	84 %
130 cm	25	20	80 %
140 cm	25	19	76 %
150 cm	25	17	68 %
160 cm	25	16	64 %
170 cm	25	16	64 %
180 cm	25	15	60 %
190 cm	25	13	52 %
200 cm	25	13	52 %

5.1.2 Varying Horizontal Angles

One of the critical factors that affect the performance of an ArUco gate detection system is the angle at which the markers are detected. To evaluate the system's capability to accurately detect and identify markers and gates at different angles, we conducted an experiment where the camera's angle of view was systematically varied relative to the markers. The objective of this experiment was to analyse the detection performance of the system when dealing with markers oriented at non-perpendicular angles to the camera.

The camera was positioned at a fixed distance of 50 cm from the gate and markers 2 and 3 from Figure 5.1, and the angle between the camera's optical axis and the plane of the gate was altered in increments of 15 degrees, ranging from 0 degrees (directly facing the gate) to 75 degrees (oblique angle). For each angle, we recorded the immediate detection rate of the markers.

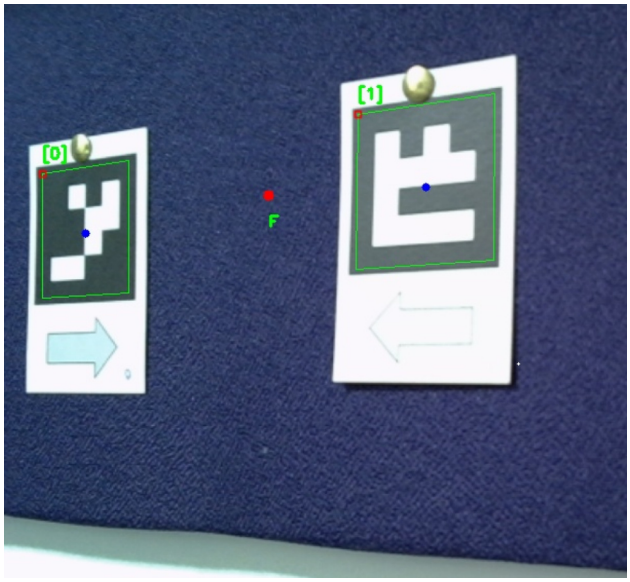


Figure 5.3: Detecting ArUco markers and gate at a 60 ° horizontal angle.

The experiment results showed that the detection system performed well at immediately detecting markers and gates, even at relatively large angles. The detection rate decreased gradually as the angle increased, but the system maintained a satisfactory performance up to 75 degrees. The following data summarizes the results obtained from this experiment:

Table 5.2: Marker detection test results at varying horizontal angles

Angle (in degrees °)	Total detection trials	Successful detections	Detection Rate (%)
0 °	25	25	100 %
15 °	25	25	100 %
30 °	25	24	96 %
45 °	25	24	96 %
60 °	25	22	88 %
75 °	25	21	84 %

These results indicate that the detection system is capable of effectively detecting and identifying markers and gates at various horizontal angles. The robust performance of the system under different angles ensures that it can provide reliable navigation information to robots, even when they approach the gates from diverse directions.

5.1.3 Varying Vertical Angles

Our system was also tested with varied vertical and diagonal angles with similar results to those described above. To evaluate the system's capability to accurately detect and identify markers and gates at different vertical and diagonal angles, we conducted an experiment where the camera's angle of view was systematically varied relative to the markers.

The camera was positioned in a similar set up as in the above experiment, at a fixed distance of 50 cm from the gate and markers 2 and 3 from Figure 5.1 and the angle between the optical axis and the gate was incremented by 15°, ranging from 0° to 75°. Figures 5.4 and 5.5 showcase a series of screenshots taken during the testing experiment for vertical and diagonal angles. The diagonal angles were tested by vertically tilting then horizontally translating the position of the camera to detect the markers, to simulate irregular terrain traversed by the autonomous agent.

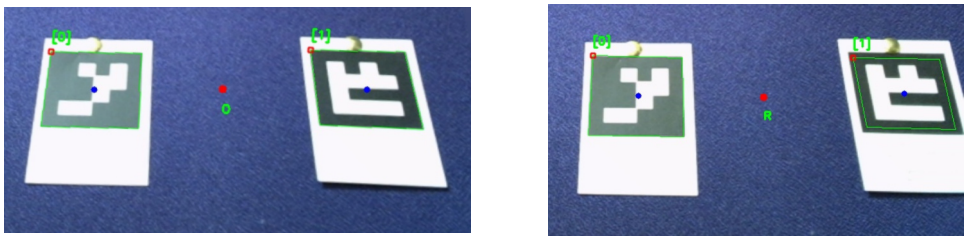


Figure 5.4: Detecting ArUco gate at 30 ° and 15 ° vertical angles.

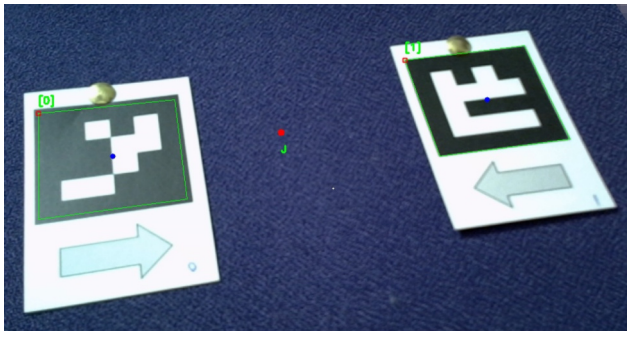


Figure 5.5: Detecting ArUco markers and gate at a 60 ° vertical angle and 20 cm left-horizontal translation.

The results from these experiments demonstrate that the detection system is capable of effectively detecting and identifying markers and gates at various vertical angles. The system has shown to be robust and reliable, even when the robots would approach from irregular angles.

5.2 Character Output Range

This test was conducted to test the character output as navigation information to an autonomous agent. We analyse and explain the reasoning behind the range of obtainable outputs for given conditions. This test was run using the code developed in Appendix B as it allows for the display of the ASCII character on the frame at the target coordinates—providing a convenient testing environment.

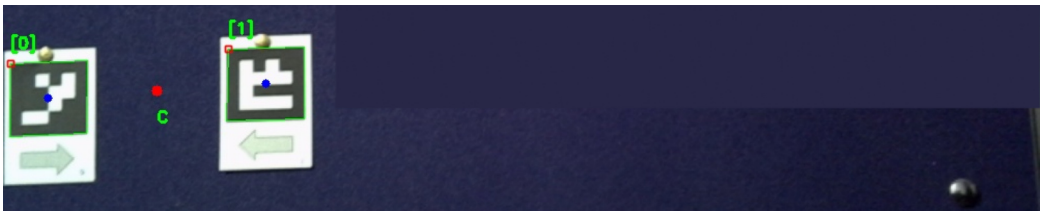


Figure 5.6: Character C output as gate target point at a range of 80 cm and a perpendicular angle to the centre of the pin-board set-up

The experiment was carried out at a distance of 80 cm from the plane and at a perpendicular angle to the pin-board set-up of markers. The camera was positioned at the centre of the board viewing the front of the markers tested at an slight-left angle, positioning them to the left of the camera optical axis. The camera was adjusted to have the marker with ID 0 at the extremity of the frame.

The markers and gate were correctly detected and identified and the target point's ASCII character was provided. At a range of 80 cm, the most obtainable character was C. If the camera angle would move any more to the right, the left marker would be occluded, thus the system would not detect both markers to form a valid gate. From this, we can assume that the right side of the frame would provide similar observations.

The camera's distance from the plane of markers was then increased to check for the maximum range of characters. It was observed that the detection rate of markers decreased as the camera was moved further from the plane of markers. The character "B" and "Y" could be obtained depending on the stability of the system and optimal environmental conditions. "A" could not be obtained as the left-most marker would need to take less than 40 pixels in the image, which would drastically decreased the likelihood of detection for the marker. Hence, characters "A" and "Z" would be impossible to obtained using a scale that covers the whole frame. We can derive that the effective character range—for a scale covering the complete frame width and a distance

of 80 cm—is from "C" to "X".

The camera distance from the plane is a critical factor dictating the obtainable character range. The camera distance proved to be directly proportional to the character range.

Camera Distance α Character Range

As the camera distance increases, the character range would also increase. This also means that the obtainable character range at a short distance is short and closer to the centre alphabet character "M"—representing the centre of the image. Markers appear bigger in the image as the camera gets closer to the plane, causing the target's ASCII character to be closer to "M". This can be demonstrated in Figure 4.8, where the ASCII characters are relatively close to character "M".

The cutoff of obtainable characters was due to marker occlusion when markers were viewed at extremities of the frame of the camera. It was observed that the left-most or right-most markers would not be detected if part of the markers were outside of the frame. To account for this, a possible solution would be to reduce the length of the scale used to convert our target coordinate into the ASCII character. The adjusted scale, as displayed in Figure 5.7 would:
start at x-coordinate= $0 + (framewidth/25)$ and
end at x-coordinate= $framewidth - (framewidth/25)$

Figure 5.7: The adjusted scale to cater for the first position being undetectable.

This adjusted scale functionality, however, was not implemented and will be included in future works for the project.

5.3 Motion Blur

The objective of this experiment is to assess the ability of the system to detect and recognise ArUco markers and gates under different levels of motion blur. Motion blur is a common challenge in computer vision real-time applications, as it can lead to a loss of detail in the captured frames, making it difficult for the detection algorithm to identify and decode the markers accurately.

To simulate motion blur in a controlled manner, I held the camera and moved my arm at different speeds while capturing images of ArUco markers at a fixed distance. An altered version of the program was created to detect markers and gates in images, rather than real-time video capture. This setup allowed us to create three distinct levels of motion blur in the captured images: low, medium and high.

For each level of motion blur, we analyse a set of images containing ArUco markers and formed gates. We then calculated the detection rate for 10 trials and the gate detection rate, which is the percentage of correctly identified gates compared to the total number of gates present in the image of 10.

Table 5.3: Marker detection test results at three levels of motion blur

Motion Blur Level	Detection Rate (%)	Gate Detection Rate (%)
Low	70 %	80 %
Medium	50 %	60 %
High	40 %	30 %

By comparing the detection rate and gate detection rate for each level of motion blur, we can understand how the performance of the detection system is affected by motion blur. This provided insights into the system's fragility under different levels of motion blur and helped identify areas for improvement to be included in future works to enhance the performance of the navigation system in real-world scenarios.

5.4 Lighting Variations

The objective of this experiment is to evaluate the ability of the system to detect and recognise ArUco markers and gates under varying lighting conditions. Lighting variations are a common challenge in computer vision outdoor applications, as they can affect the appearance of the captured frames, making it difficult for the detection algorithm to identify and decode the markers accurately in real-time.

To assess the system's robustness to lighting variations, we conducted the experiment indoors in a dark and controlled environment and used a flashlight with 3 distinct levels of luminosity: low—dark or dim-light room, medium—well lit room, high—outdoor sunlight. The light was then directed on the marker to add glare and increase the brightness in the frames. A camera was then used to detect and identify the markers in real-time. We then calculated the detection rate for the number of successful trials in detecting markers for the varying levels of luminosity.

Table 5.4: Marker detection test results at three levels of luminosity

Luminosity Level	Detection Rate (%)
Low	100 %
Medium	80 %
High	70 %

This experiment aims to simulate the varying levels of luminosity encountered in real-world applications. As the autonomous agent using the navigation system would be an outdoor robot, the system's good performance in high luminosity, representing sunlight, is crucial. As demonstrated in Table 5.4, the system performed at satisfactory level in indoors and outdoors. The system would need to be tested further in outdoors application using luminosity sensors in future work to understand more details about how it performs in direct sunlight.

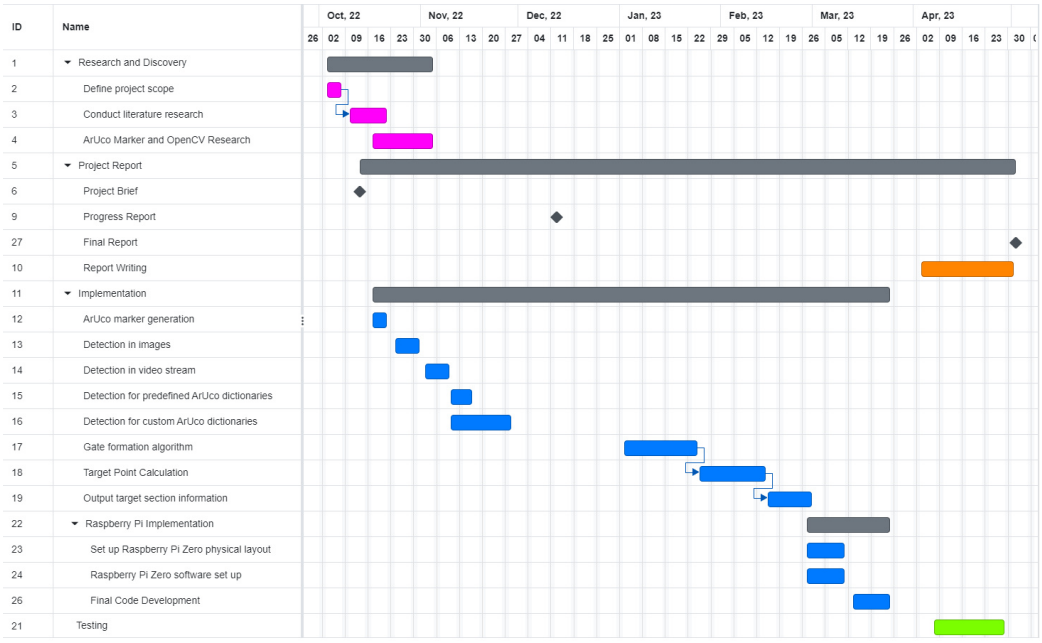
Chapter 6

Project Management

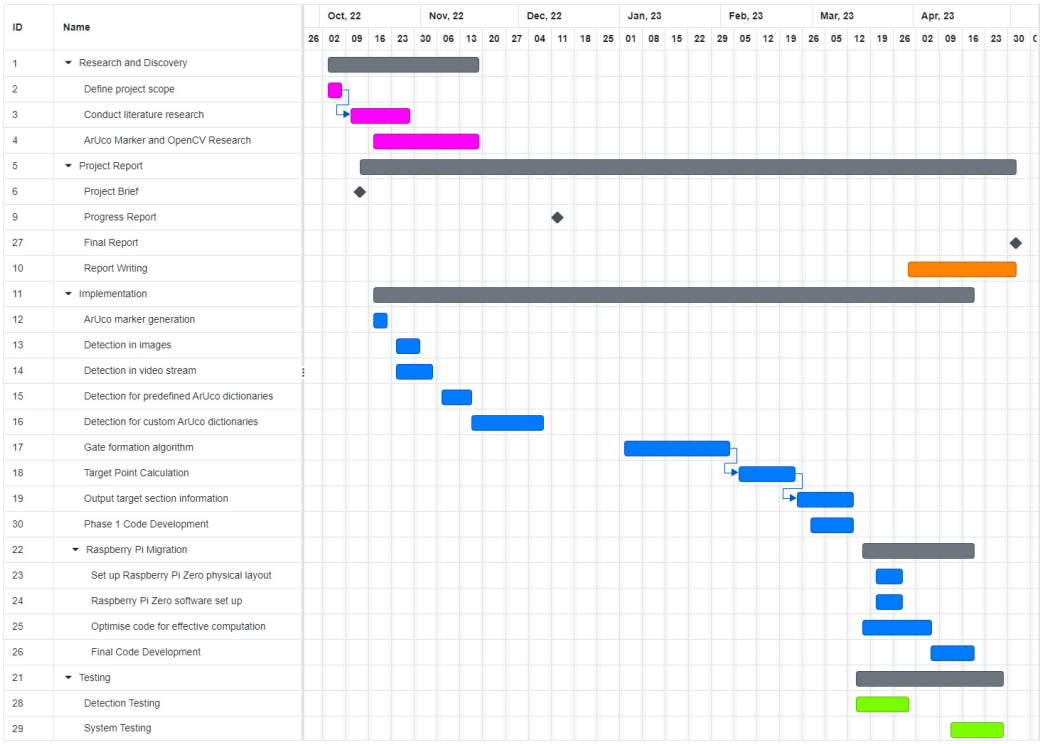
This chapter explains the project plan over the course of the year. The Gantt charts A and B in Figure 6.1 display the expected plan for the project and the actual progress made at the end of the project respectively. They both list a series of task and the amount of weeks taken for the completion of each task, represented by the rectangles. The initial stage of project was focused on literature research about fiducial markers and tutorials to learn about OpenCV and ArUco markers. After defining the scope, the implementation was split in three: detecting markers, creating the gate formation algorithm and implementation on Raspberry Pi Zero.

The first implementation stage started at the end of October after submitting the Project Brief. Up to the milestone of submitting a progress report in mid-December, effort was concentrated on understanding how to use the OpenCV to detect the markers. The second implementation stage focused on creating our gate formation algorithm to detect valid marker gates and calculate their target points. The final implementation stage constituted optimising and transporting our developed code onto a Raspberry Pi Zero as discussed above. Testing was carried out in two stages: one for the marker detection that was done after the second implementation stage and one for the system characterisation to test out our final developed system.

Time was allocate accordingly over the process of the project. It was noted that the first and second implementation stages took longer than expected as it was the core part of the project—understanding new concepts and developing the main program. Project development was finished in early-to-mid-April, allowing time for alterations, testing and report writing.



A



B

Figure 6.1: Gantt Charts for expected (A) and actual (B) project plan

Chapter 7

Conclusion

In this project report, we presented the development and implementation of a low-cost computer vision-based navigation system. The primary objective of this project was to create a system capable of accurately detecting and recognising ArUco markers arranged in a gate-like configuration, calculating their target points, and providing navigation information to guide autonomous agents through these gates. The system leverages the power of the OpenCV library, which includes a dedicated module for ArUco marker detection and decoding, to calculate and output navigation information to the autonomous agent over a serial connection in the form of ASCII characters, dictating the position of a gate detected in the frame in real-time.

A comprehensive background research was provided, discussing fiducial markers, ArUco markers, detection methods, the Raspberry Pi Zero, OpenCV, and computer vision-based navigation. Our methodology was presented, outlining the algorithms and techniques employed in the development of the system for the two implementation stages on a high-computation-capable machine and a Raspberry Pi Zero. Subsequently, we conducted a series of experiments to evaluate the system's performance and robustness under varying conditions, such as motion blur, lighting variations, distance, and angles.

Our experiments demonstrated that the system exhibits a high detection rate under most conditions. However, as expected, the performance decreases under extreme occlusion or severe motion blur. The system also proved to be robust under different lighting conditions, showcasing its potential for real-world applications.

Despite its success, several challenges and limitations were identified, such as marker occlusion and motion blur. Future work could focus on improving

the system's performance under these challenging conditions and further optimisation of the algorithm could be explored to improve the overall system's usability and efficiency.

This project showcases the potential for low-cost computer vision-based navigation systems in real-world applications, such as robotics and UAVs. Our work contributes to and opens new paths for further development in this rapidly-growing field.

References

[1] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer Science & Business Media, 2010.

[2] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[3] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.

[4] Francisco Bonin-Font, Alberto Ortiz, and Gabriel Oliver. "Visual navigation for mobile robots: A survey". In: *Journal of Intelligent and Robotic Systems* 53.3 (2008), pp. 263–296.

[5] Hirokazu Kato and Mark Billinghurst. "Marker tracking and HMD calibration for a video-based augmented reality conferencing system". In: *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality*. 1999. DOI: 10.1109/IWAR.1999.803809.

[6] Sergio Garrido-Jurado et al. "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6 (2014), pp. 2280–2292.

[7] Sergio Garrido-Jurado et al. "Generation of fiducial marker dictionaries using mixed integer linear programming". In: *Pattern Recognition* 51 (2016), pp. 481–491.

[8] Federico Boniardi et al. "Autonomous indoor robot navigation using a sketch interface for drawing maps and routes". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 2896–2901. DOI: 10.1109/ICRA.2016.7487453.

[9] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, Inc., 2008.

[10] Francisco J Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. "Speeded up detection of squared fiducial markers". In: *Image and Vision Computing* 76 (2018), pp. 38–47.

[11] Mark Fiala. "Designing Highly Reliable Fiducial Markers". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.7 (2010), pp. 1317–1324. DOI: 10.1109/TPAMI.2009.146.

[12] Mark Fiala. "ARTag, a fiducial marker system using digital techniques". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. 2005, 590–596 vol. 2. DOI: 10.1109/CVPR.2005.74.

[13] Edwin Olson. "AprilTag: A robust and flexible visual fiducial system". In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3400–3407. DOI: 10.1109/ICRA.2011.5979561.

[14] Burak Benligiray, Cihan Topal, and Cuneyt Akinlar. "STag: A stable fiducial marker system". In: *Image and Vision Computing* 89 (2019), pp. 158–169. ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2019.06.007.

[15] Washington Publications. *ARToolKit*. URL: http://www.hitl.washington.edu/artoolkit.html. (accessed on 8 December 2022).

[16] Michail Kalaitzakis et al. "Fiducial Markers for Pose Estimation: Overview, Applications and Experimental Comparison of the ARTag, AprilTag, ArUco and STag Markers". In: *Journal of Intelligent & Robotic Systems* 101 (Apr. 2021). DOI: 10.1007/s10846-020-01307-9.

[17] Najib Altawell. "Introduction to Machine Olfaction Devices". In: Academic Press, 2022, pp. 47–62. ISBN: 978-0-12-822420-5. DOI: https://doi.org/10.1016/B978-0-12-822420-5.00009-X.

[18] Ksenia Shabalina et al. "ARTag, AprilTag and CALTag Fiducial Systems Comparison in a Presence of Partial Rotation: Manual and Automated Approaches". In: *Informatics in Control, Automation and Robotics*. Ed. by Oleg Gusikhin and Kurosh Madani. Cham: Springer International Publishing, 2020, pp. 536–558. ISBN: 978-3-030-11292-9.

[19] Mark Fiala. "Comparing ARTag and ARToolkit Plus fiducial marker systems". In: *IEEE International Workshop on Haptic Audio Visual Environments and their Applications*. 2005, 6 pp. DOI: 10.1109/HAVE.2005.1545669.

[20] Arturo Gil et al. "A comparative evaluation of interest point detectors and local descriptors for visual SLAM". In: *Machine Vision and Applications* 21 (Oct. 2010), pp. 905–920. DOI: 10.1007/s00138-009-0195-x.

[21] Adam Marut, Konrad Wojtowicz, and Krzysztof Falkowski. "ArUco markers pose estimation in UAV landing aid system". In: *2019 IEEE 5th International Workshop on Metrology for AeroSpace (MetroAeroSpace)*. 2019, pp. 261–266. DOI: 10.1109/MetroAeroSpace.2019.8869572.

[22] Daniel Wagner et al. "Pose Tracking from Natural Features on Mobile Phones". In: *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*. ISMAR '08. USA: IEEE Computer Society, 2008, pp. 125–134. ISBN: 9781424428403. DOI: 10.1109/ISMAR.2008.4637338.

[23] OpenCV. *About*. 2020. URL: https://opencv.org/about/. (accessed on 8 December 2022).

[24] OpenCV. *Detection of ArUco Markers*. URL: https://docs.opencv.org/4.1.0/d5/dae/tutorial_aruco_detection.html. (accessed on 23 April 2023).

[25] OpenCV. *aruco.hpp File Reference*. URL: https://docs.opencv.org/4.1.0/d9/d53/aruco_8hpp.html. (accessed on 23 April 2023).

[26] OpenCV. *ArUco marker detection*. 2020. URL: https://docs.opencv.org/4.x/d9/d6d/tutorial_table_of_content_aruco.html. (accessed on 8 December 2022).

[27] Adrian Rosebrock. *ArUco Markers—PyImageSearch*. 2021. URL: https://pyimagesearch.com/category/aruco-markers/. (accessed on 8 December 2022).

[28] Oleg Kalachev. *ArUco Marker Generator*. URL: https://chev.me/arucogen/. (accessed on 23 April 2023).

[29] Bill Waggener. *Pulse code modulation techniques with applications in communications and data recording*. New York: Van Nostrand Reinhold, 1995.

[30] Raspberry Pi. *Raspberry Pi Zero*. URL: https://www.raspberrypi.com/products/raspberry-pi-zero/. (accessed on 25 April 2023).

[31] The Pi Hut. *Raspberry Pi Zero v1.3*. URL: https://thepihut.com/products/raspberry-pi-zero?src=raspberrypi. (accessed on 25 April 2023).

[32] Raspberry Pi. *Raspberry Pi Zero: the $5 computer*. 2021. URL: https://www.raspberrypi.com/news/raspberry-pi-zero/. (accessed on 8 December 2022).

[33] The Pi Hut. *IO Expansion HAT for Raspberry Pi*. URL: https://thepihut.com/products/io-expansion-hat-for-raspberry-pi. (accessed on 25 April 2023).

[34] FTDI Chip. *USB to Hi-Speed UART Serial Adapter Cable w/Embedded Electronics, LEDs, 3.3V*. URL: https://ftdichip.com/products/c232hd-ddhsp-0/. (accessed on 25 April 2023).

[35] UUGear. *ZERO4U: 4-PORT USB HUB FOR RASPBERRY PI ZERO (V1.3 AND W)*. URL: https://www.uugear.com/product/zero4u/. (accessed on 25 April 2023).

[36] PyPi. *pip*. URL: https://pypi.org/project/pip/. (accessed on 25 April 2023).

[37] OpenCV. *OpenCV modules*. URL: https://docs.opencv.org/4.x/. (accessed on 25 April 2023).

[38] Tony G. *How to install OpenCV on Raspberry Pi 4*. 2021. URL: https://singleboardbytes.com/647/install-opencv-raspberry-pi-4.htm. (accessed on 25 April 2023).

[39] Raspberry Pi. *Raspberry Pi OS*. URL: https://www.raspberrypi.com/software/. (accessed on 26 April 2023).

[40] Andrew J. Davison et al. "MonoSLAM: Real-Time Single Camera SLAM". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29.6 (2007), pp. 1052–1067. DOI: 10.1109/TPAMI.2007.1049.

[41] Piotr Gaska et al. "Simple Optical Coordinate Measuring System, Based on Fiducial Markers Detection, and its Accuracy Assessment". In: *Advances in Science and Technology Research Journal* 14 (Dec. 2020), pp. 213–219. DOI: 10.12913/22998624/127082.

[42] Torsten Sattler, Bastian Leibe, and Leif Kobbelt. "Fast image-based localization using direct 2D-to-3D matching". In: Nov. 2011, pp. 667–674. DOI: 10.1109/ICCV.2011.6126302.

[43] Emilio Maggio and Andrea Cavallaro. *Video tracking: Theory and practice*. Wiley, 2011.

[44] Elias Lageder. "A Computer Vision System using Fiducial Markers". In: (May 2021). URL: https://secure.ecs.soton.ac.uk/notes/comp3200/e_archive/COMP3200/2021/el6g18/pdfs/Report.pdf. (accessed on 8 December 2022).

[45] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[46] Python Documentation. *Python 3.11*. URL: https://docs.python.org/3/whatsnew/3.11.html. (accessed on 23 April 2023).

[47] Numpy. *Numpy Python Library*. URL: https://numpy.org/. (accessed on 23 April 2023).

[48] Python Documentations. *sys—System-specific parameters and functions*. URL: https://docs.python.org/3/library/sys.html. (accessed on 23 April 2023).

[49] Injosoft Co. *ASCII table according to Windows-1252*. URL: https://www.ascii-code.com/. (accessed on 26 April 2023).

[50] UUGear. *Zero4U 4-Port USB Hub for Raspberry Pi Zero User Manual (revision 1.20)*. URL: https://www.uugear.com/doc/Zero4U_UserManual.pdf. (accessed on 25 April 2023).

[51] PyPi. *pyserial*. URL: https://pypi.org/project/pyserial/. (accessed on 27 April 2023).

[52] Chris Liechti. *pySerial Documentations*. URL: https://pythonhosted.org/pyserial/. (accessed on 27 April 2023).

Appendix A

Project Brief

Low-cost vision system for autonomous outdoor robots

Student: Sharan Govinden Umavassee

Project Supervisor: Klaus-Peter Zauner

Computer vision (CV) is a field of artificial intelligence (AI) that enables computers to derive meaningful information from digital images, videos and other visual inputs – and take actions or make recommendations based on that information (IBM, n.d.). It remains one of the great challenges in AI to achieve computer vision's fullest potential. With the advancement of technology – more powerful hardware such as graphics card and processors and use of techniques such as deep convolutional neural networks – computer vision solutions are constantly setting the bar higher. Competitions, commonly known as 'hackathons', that compare different solutions using the same data set is one of the driving forces behind progress in computer vision. However, they tend to focus on accuracy and ignore efficiency on hardware with limited resources (Lu, 2022).

This project is inspired by the gates used in competitions of remote-controlled (RC) rock crawlers. With the use of 'ArUco' markers – synthetic square markers composed by a wide black border and an inner binary matrix determining their identifier (OpenCV, n.d., p. ArUco markers) – acting as gates and a microprocessor, we aim to produce a computer vision module that will be connected to and allow rovers to autonomously follow a path laid out by the ArUco marker gates. It will test the efficiency of computer vision technologies on low resource hardware.

The scope of the project is:

1. Literature review on computer vision, embedded systems and ArUco markers.
2. Build an electrical system using a raspberry pi zero microprocessor and a camera to obtain feed of images/frames about the field of view.
3. Compute ArUco markers data from OpenCV libraries to obtain coordinates for navigations and calculation.
4. Use an error correction algorithm on coordinates data from ArUco markers to correctly assess position of gates in images/frames.
5. Compute and output navigational information to robot through the microprocessor and signal indicating directions to follow.
6. Create user interface using web server to use, calibrate and control the system.

References

IBM, n.d. What is computer vision?. [Online]
Available at: `https://www.ibm.com/uk-en/topics/computer-vision`
Lu, G. T. a. Y., 2022. Efficient Computer Vision for Embedded Systems. no. 04 ed. s.l.:Computer, vol.55.
OpenCV, n.d. Detection of ArUco Markers. [Online]
Available at: `https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html`

Appendix B

Code for Development Implementation

```
# import the necessary packages
import sys
import numpy as np
import cv2

# Create custom dictionary of aruco markers
arucoDict = cv2.aruco.Dictionary_create(4,4)
arucoParams = cv2.aruco.DetectorParameters_create()

if len(sys.argv) > 1:
    camera_index = int(sys.argv[1])
else:
    camera_index = 0  # Default camera index

# Camera Setup
print("[INFO] Starting video stream...")
cam = cv2.VideoCapture(camera_index)
if not cam.isOpened():
    print("Cannot open camera")
    sys.exit()

# Class to represent the aruco markers as form of gates
class gate:
    def __init__(self, corners, ID):
        self.corners = corners
        self.ID = ID
        self.size = cv2.norm(corners[0] - corners[2]) #
   diagonal length of the marker

```

```
# Function to calculate the midpoint between two markers
def calculate_midpoint(marker1, marker2):
    x1, y1 = marker1[1] # top-right corner of marker 1
    x2, y2 = marker2[3] # bottom-left corner of marker 2

    # calculate midpoint
    x_mid = (x1 + x2) / 2
    y_mid = (y1 + y2) / 2

    return (x_mid, y_mid)

# Function to draw a circle at the midpoint between two markers
def drawtarget(image, target, asc):
    x_mid, y_mid = target

    # draw a circle at the midpoint when gate is detected
    if not asc == "-":
        cv2.circle(image, (int(x_mid), int(y_mid)), 5, (0, 0, 255), -1)

    # draw the ASCII character on the frame
    cv2.putText(frame, asc,
        (int(x_mid), int(y_mid + 30)),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.5, (0, 255, 0), 2)

# loop over the frames from the video stream
while True:
    # Capture frame-by-frame video stream
    _, frame = cam.read()

    # grab the frame from the threaded video stream and resize it
    # to have a maximum width of 1000 pixels
    # and maximum height of 700 pixels
    frame = cv2.resize(frame, (1000, 700))

    # Get the height and width of the image
    height, width, _ = frame.shape

    # Draw circle at centre of the frame
    centre = (width // 2, height // 2)
    cv2.circle(frame, centre, 1, (255, 255, 255), -1)

    # List of tags in view
    tags  = []
    tagID = []

```

```
75      # detect ArUco markers in the input frame
76      (corners, ids, rejected) = cv2.aruco.detectMarkers(frame,
        arucoDict, parameters=arucoParams)
77
78      # verify *at least* one ArUco marker was detected
79      if len(corners) > 0:
80          # Zip corners and ids inside tuple
81          markers = zip(corners, ids)
82
83          for x in range(len(ids)):
84              # Draw bounding box for detected aruco markers
85              cv2.aruco.drawDetectedMarkers(frame, corners)
86
87              # add detected ArUco markers to tags list
88              g = gate(corners[x][0], ids[x])
89              tags.append(g)
90
91          # Sort tags list by size of ArUco markers in
    ascending order
92          tags.sort(key = lambda gate: gate.size, reverse =
    True)
93
94          # loop over the detected ArUCo corners
95          for (markerCorner, markerID) in markers:
96              # extract the marker corners (which are always
    returned in
97              # top-left, top-right, bottom-right, and bottom-
    left order)
98              # corners[0][0], corners[0][1], corners[0][2],
    corners[0][3]
99              corners = markerCorner.reshape((4, 2))
100             (topLeft, topRight, bottomRight, bottomLeft) =
    corners
101
102             # convert each of the (x, y)-coordinate pairs to
    integers
103             topRight = (int(topRight[0]), int(topRight[1]))
104             bottomRight = (int(bottomRight[0]), int(
    bottomRight[1]))
105             bottomLeft = (int(bottomLeft[0]), int(bottomLeft
    [1]))
106             topLeft = (int(topLeft[0]), int(topLeft[1]))
107
108             # compute and draw the center (x, y)-coordinates
    of the ArUco marker
109             cX = int((topLeft[0] + bottomRight[0]) / 2.0)
110             cY = int((topLeft[1] + bottomRight[1]) / 2.0)
111             cv2.circle(frame, (cX, cY), 4, (255, 0, 0), -1)
112
```

```
113             # draw the ArUco marker ID on the frame
114             cv2.putText(frame, str(markerID),
115         (topLeft[0], topLeft[1] - 15),
116         cv2.FONT_HERSHEY_SIMPLEX,
117         0.5, (0, 255, 0), 2)
118 
119         gatedetect = False
120 
121         # Scale factor to find convert coordinates to an
    ascii value
122         # 25 sections (26 characters, 0-25).
123         scale_factor = 25 / width # 0.025
124 
125         # Check if the gates are aligned correctly
126         if (len(tags) == 2):
127             # Extracting the corners of the markers
128             marker0Corners = tags[0].corners
129             marker1Corners = tags[1].corners
130 
131             # Calculate the centre of the markers
132             marker0Center = np.mean(marker0Corners, axis=0).
    astype(int)
133             marker1Center = np.mean(marker1Corners, axis=0).
    astype(int)
134 
135             # Calculate the midpoint between the two markers
    to act as target
136             target = calculate_midpoint(marker0Corners,
    marker1Corners)
137 
138             # Default value for ASCII character and if no
    gate is detected
139             asc = "-"
140 
141             # Check if the markers are aligned correctly
142             if (tags[0].ID[0] == 0 and tags[1].ID[0] == 1):
143                 # Compare the positions of the markers
144                 if (marker0Center[0] < marker1Center[0]):
145                     # Gate detected
146                     # Scale and convert the target point
147                     # x-coordinate to uppercase character
148                     asc = chr(int((target[0] * scale_factor)
    + 64))
149                     gatedetect = True
150 
151             elif (tags[0].ID[0] == 1 and tags[1].ID[0] == 0):
152                 # Compare the positions of the markers
153                 if (marker0Center[0] > marker1Center[0]):
154                     # Gate detected
```

```
155                     # Scale and convert the target point
156                     # x-coordinate to uppercase character
157                     asc = chr(int((target[0] * scale_factor)
    + 64))
158                     gatedetect = True
159 
160             # Print the ASCII character to the console
161             print(asc)
162 
163             # Draw a target point between the two markers
164             drawtarget(frame, target, asc)
165 
166         if len(tags) >= 3:
167             # Extracting the corners of the markers
168             marker0Corners = tags[0].corners
169             marker1Corners = tags[1].corners
170             marker2Corners = tags[2].corners
171 
172             # Calculate the centre of the markers
173             marker0Center = np.mean(marker0Corners, axis=0).
    astype(int)
174             marker1Center = np.mean(marker1Corners, axis=0).
    astype(int)
175             marker2Center = np.mean(marker2Corners, axis=0).
    astype(int)
176 
177             # Default value for ASCII character
178             asc = "-"
179 
180             # Compare the IDs of the markers
181             if (tags[0].ID[0] == 0 and tags[1].ID[0] == 1):
182                 # Calculate the midpoint between the two
    markers to act as target
183                 target = calculate_midpoint(marker0Corners,
    marker1Corners)
184 
185                 # Compare the positions of the markers
186                 if (marker0Center[0] < marker1Center[0]):
187                     # Gate detected
188                     # Scale and convert the target point
189                     # x-coordinate to uppercase character
190                     asc = chr(int((target[0] * scale_factor)
    + 64))
191 
192                     # Draw a target point between the two
    markers
193                     drawtarget(frame, target, asc)
194 
195                     gatedetect = True
```

```
196
197             elif (tags[0].ID[0] == 1 and tags[1].ID[0] == 0):
198                 # Calculate the midpoint between the two
    markers to act as target
199                 target = calculate_midpoint(marker0Corners,
    marker1Corners)
200
201                 # Compare the positions of the markers
202                 if (marker0Center[0] > marker1Center[0]):
203                     # Gate detected
204                     # Scale and convert the target point
205                     # x-coordinate to uppercase character
206                     asc = chr(int((target[0] * scale_factor)
    + 64))
207
208                     # Draw a target point between the two
    markers
209                     drawtarget(frame, target, asc)
210
211                     gatedetect = True
212
213             elif (tags[1].ID[0] == 1 and tags[2].ID[0] == 0):
214                 # Calculate target point
215                 target = calculate_midpoint(marker1Corners,
    marker2Corners)
216
217                 # Compare the positions of the markers
218                 if (marker1Center[0] > marker2Center[0]):
219                     # Gate detected
220                     # Scale and convert the target point
221                     # x-coordinate to uppercase character
222                     asc = chr(int((target[0] * scale_factor)
    + 64))
223
224                     # Draw a target point between the two
    markers
225                     drawtarget(frame, target, asc)
226
227                     gatedetect = True
228
229             elif (tags[1].ID[0] == 0 and tags[2].ID[0] == 1):
230                 # Calculate target point
231                 target = calculate_midpoint(marker1Corners,
    marker2Corners)
232
233                 # Compare the positions of the markers
234                 if (marker1Center[0] < marker2Center[0]):
235                     # Gate detected
236                     # Scale and convert the target point
```

```
                    # x-coordinate to uppercase character
                    asc = chr(int((target[0] * scale_factor)
    + 64))

                    # Draw a target point between the two
    markers
                    drawtarget(frame, target, asc)

                    gatedetect = True

            # Print the ASCII character to the console
            print(asc)

            # Draw a target point between the two markers
            drawtarget(frame, target, asc)

        tagID = [tag.ID[0] for tag in tags]
        tagsize = [tag.size for tag in tags]

    # show the output frame
    cv2.imshow("Fiducial Marker Computer Vision-based
    Navigation System", frame)
    key = cv2.waitKey(1) & 0xFF

    # if the `s` key was pressed, take a screenshot
    if key == ord("s"):
        cv2.imwrite("screenshot.jpg", frame)
        print("[INFO] Screenshot saved.")

    # if the `q` key was pressed, break from the loop
    if key == ord("q"):
        print("[INFO] stopping video stream...")
        break

if not(tagID == []):
    print("[INFO] Detected Tags ID: " + str(tagID))
    print("[INFO] Tag Sizes (pixels): " + str(tagsize))
    if gatedetect:
        print("[INFO] Gate detected.")
        print("[OUTPUT] Target Gate Coordinates: " + str(
    target))
    else:
        print("[INFO] No Gate detected.")
    print("[OUTPUT] ASCII value of target:", asc)

# cleanup
cam.release()
cv2.destroyAllWindows()
```

Appendix C

Code for Final Implementation on Raspberry Pi

```
# import the necessary packages
import sys
import numpy as np
import cv2
import serial

# Initialize UART serial communication
ser = serial.Serial(
    port='/dev/serial0',
    baudrate=115200,
    bytesize=serial.EIGHTBITS,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    timeout=1)ser = serial.Serial('/dev/serial0', 115200)

# Class to represent the aruco markers as form of gates
class gate:
    def __init__(self, corners, ID):
        self.corners = corners
        self.ID = ID
        # diagonal length of the marker
        self.size = cv2.norm(corners[0] - corners[2])

# Function to calculate the midpoint between two markers
def calculate_midpoint(marker1, marker2):
    x1, y1 = marker1[1] # top-right corner of marker 1
    x2, y2 = marker2[3] # bottom-left corner of marker 2

    # calculate midpoint
```

```
30      x_mid = (x1 + x2) / 2
31      y_mid = (y1 + y2) / 2
32
33      return (x_mid, y_mid)
34
35  # create custom dictionary of aruco markers
36  arucoDict = cv2.aruco.Dictionary_create(4,4)
37  arucoParams = cv2.aruco.DetectorParameters_create()
38
39  if len(sys.argv) > 1:
40      camera_index = int(sys.argv[1])
41  else:
42      camera_index = 0  # Default camera index
43
44  # Camera Setup
45  print("[INFO] Starting video stream...")
46  cam = cv2.VideoCapture(camera_index)
47  if not cam.isOpened():
48      print("Cannot open camera")
49      sys.exit()
50
51  # loop over the frames from the video stream
52  while True:
53      # Capture frame-by-frame video stream
54      _, frame = cam.read()
55
56      # grab the frame from the threaded video stream and
        resize it
57      # to have a maximum width of 1000 pixels
58      # and maximum height of 750 pixels
59      frame = cv2.resize(frame, (1000, 700))
60
61      # Get the height and width of the image
62      _, width,_ = frame.shape
63
64      # List of tags in view
65      tags = []
66
67      # detect ArUco markers in the input frame
68      (corners, ids, rejected) = cv2.aruco.detectMarkers(frame,
        arucoDict, parameters=arucoParams)
69
70      # verify *at least* one ArUco marker was detected
71      if len(corners) > 0:
72          # Add detected ArUco markers to tags list
73          for x in range(len(ids)):
74              g = gate(corners[x][0], ids[x])
75              tags.append(g)
76
```

```
77          # Sort tags list by size of ArUco markers in
    ascending order
78          tags.sort(key = lambda gate: gate.size, reverse =
    True)
79
80          # Scale factor to convert coordinates
81          # to an ascii value
82          # 25 sections (26 characters, 0-25).
83          scale_factor = 25 / width # 25 / 1000 = 0.025
84
85          # Gate formation algorithm for 2 markers detected
86          if (len(tags) == 2):
87              # Extracting the corners of the markers
88              marker0Corners = tags[0].corners
89              marker1Corners = tags[1].corners
90
91              # Calculate the centre of the markers
92              marker0Center = np.mean(marker0Corners, axis=0).
    astype(int)
93              marker1Center = np.mean(marker1Corners, axis=0).
    astype(int)
94
95              # Calculate the midpoint between the two markers
    to act as target
96              target = calculate_midpoint(marker0Corners,
    marker1Corners)
97
98              # Default value for ASCII character and if no
    gate is detected
99              asc = "-"
100
101             # Compare the IDs of the markers to verify gate
    alignment
102             if (tags[0].ID[0] == 0 and tags[1].ID[0] == 1):
103                 # Compare the positions of the markers
104                 if (marker0Center[0] < marker1Center[0]):
105                     # Gate detected
106                     # Scale and convert the target point
107                     # x-coordinate to uppercase character
108                     asc = chr(int((target[0] * scale_factor)
    + 64))
109
110             elif (tags[0].ID[0] == 1 and tags[1].ID[0] == 0):
111                 # Compare the positions of the markers
112                 if (marker0Center[0] > marker1Center[0]):
113                     # Gate detected
114                     # Scale and convert the target point
115                     # x-coordinate to uppercase character
116                     asc = chr(int((target[0] * scale_factor)
```

```
    + 64))
117
118             # Send the ASCII character over UART
119             ser.write(asc.encode())
120
121         # Gate formation algorithm for at least 3 markers
    detected
122         if len(tags) >= 3:
123             # Extracting the corners of the markers
124             marker0Corners = tags[0].corners
125             marker1Corners = tags[1].corners
126             marker2Corners = tags[2].corners
127
128             # Calculate the centre of the markers
129             marker0Center = np.mean(marker0Corners, axis=0).
    astype(int)
130             marker1Center = np.mean(marker1Corners, axis=0).
    astype(int)
131             marker2Center = np.mean(marker2Corners, axis=0).
    astype(int)
132
133             # Default value for ASCII character and if no
    gate is detected
134             asc = "-"
135
136             # Compare the IDs of the markers to verify gate
    alignment
137             if (tags[0].ID[0] == 0 and tags[1].ID[0] == 1):
138                 # Calculate the midpoint between the two
    markers to act as target
139                 target = calculate_midpoint(marker0Corners,
    marker1Corners)
140
141                 # Compare the positions of the markers
142                 if (marker0Center[0] < marker1Center[0]):
143                     # Gate detected
144                     # Scale and convert the target point
145                     # x-coordinate to uppercase character
146                     asc = chr(int((target[0] * scale_factor)
    + 64))
147
148             elif (tags[0].ID[0] == 1 and tags[1].ID[0] == 0):
149                 # Calculate the midpoint between the two
    markers to act as target
150                 target = calculate_midpoint(marker0Corners,
    marker1Corners)
151
152                 # Compare the positions of the markers
153                 if (marker0Center[0] > marker1Center[0]):
```

```
                    # Gate detected
                    # Scale and convert the target point
                    # x-coordinate to uppercase character
                    asc = chr(int((target[0] * scale_factor)
    + 64))

            elif (tags[1].ID[0] == 1 and tags[2].ID[0] == 0):
                # Calculate target point
                target = calculate_midpoint(marker1Corners,
    marker2Corners)

                # Compare the positions of the markers
                if (marker1Center[0] > marker2Center[0]):
                    # Gate detected
                    # Scale and convert the target point
                    # x-coordinate to uppercase character
                    asc = chr(int((target[0] * scale_factor)
    + 64))

            elif (tags[1].ID[0] == 0 and tags[2].ID[0] == 1):
                # Calculate target point
                target = calculate_midpoint(marker1Corners,
    marker2Corners)

                # Compare the positions of the markers
                if (marker1Center[0] < marker2Center[0]):
                    # Gate detected
                    # Scale and convert the target point
                    # x-coordinate to uppercase character
                    asc = chr(int((target[0] * scale_factor)
    + 64))

            # Send the ASCII character over UART
            ser.write(asc.encode())

    key = cv2.waitKey(1) & 0xFF

    # if the 'q' key was pressed, break from the loop
    if key == ord("q"):
        print("[INFO] stopping video stream...")
        break

# close the serial port
ser.close()

# cleanup
cam.release()
cv2.destroyAllWindows()
```