Imperial College London

Imperial College London

Department of Electrical and Electronic Engineering

Mars Rover Project

Members:	*Subsystem*	*CID:*
Tan Chern Heng	*Control*	*01566453*
Foo Xin Yue	*Command*	*01782083*
Lim Tian Yi	*Vision*	*01564851*
Pek Zhen Wen Edwyn	*Drive*	*01327203*
Tharmarajan Praveen	*Energy*	*01517322*
Barua Nitu	*Integration*	*01735532*

ELEC50003/8 — Engineering Design Project 2

Supervisors — Mr Adam Bouchaala and Mrs Esther Perea

Page Count — 40

June 15, 2021

Contents

1. Introduction

This aim of this project was to design and build an autonomous rover capable of identifying the presence of objects in its immediate surroundings and building a map of the local area on a web server. The rover should be able to approach objects of interest while avoiding obstacles along the way. Modern rovers comprise sophisticated technology designed to operate in unfamiliar environments. This interdisciplinary project took inspiration from the approaches taken in developing a real-world Mars Rover (National Aeronautics and Space Administration, U. S. A., 2021) to understand the challenges real rovers face on extra-terrestrial worlds.

1.1 Required Features of Rover

1.1.1 Functional Requirements

Required Capabilities	Functional Requirements
Autonomous rover system – used in a remote location without direct supervision.	Control needs a built-in program to run autonomously using information from Drive, Energy and Vision. The rover should be able to map its surroundings autonomously.
Able to accurately measure and move the rover to a desired destination despite external disturbances	Drive to implement closed-loop position control and send measurement to other subsystems for accurate data processing.
Able to detect and avoid obstacles in its working area.	Vision to be able to detect and distinguish between the defined arena elements.
Able to build a map of its local working area (including obstacles) to an offsite data store.	Command to process data points of rover's surroundings and translate it to a grid map display on web browser
Able to act autonomously when battery is low to conserve power and preserve battery lifespan.	Control to monitor battery status from Energy and shut down systems when battery drops below critical level.

Table 1: Functional requirements

1.1.2 Non-Functional Requirements

Required Capabilities	Non-Functional Requirements
With the Sun as the only power source, the unreliability of the Sun rays dictate that the rover should be power efficient to prolong its active lifespan.	All subsystems should make decisions that increase power efficiency.
The rover is not maintained and traverses a hostile and volatile environment, so it should be able to handle unexpected situations.	Defensive programming and error handling to be employed in all subsystems

Table 2: Non-functional requirements

1.2 Introduction and Relationship Between Subsystems

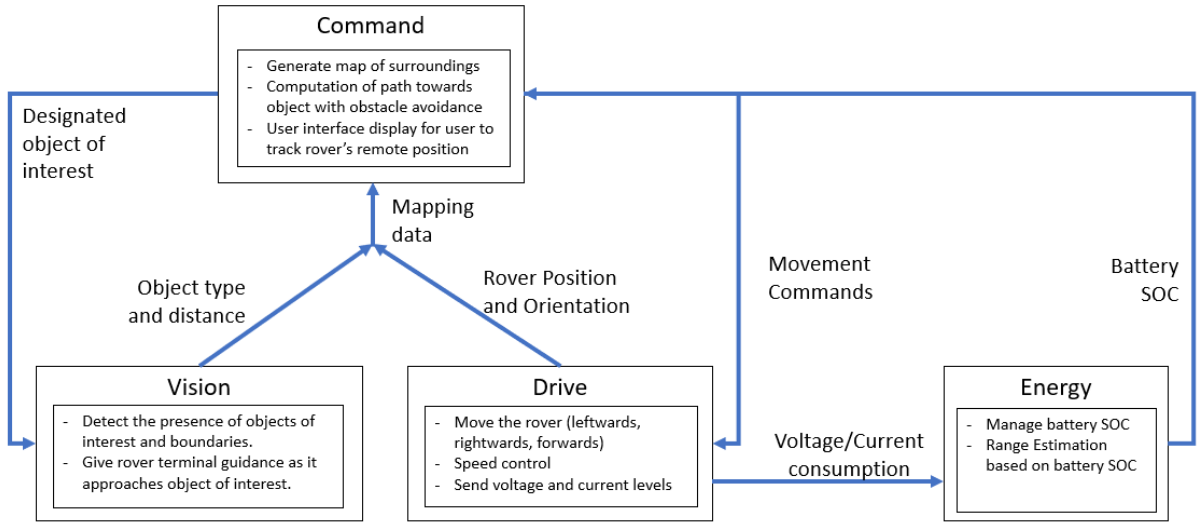


Figure 1: Data dependencies between subsystems

Figure 1 above illustrates the relationship between the different subsystems, and maps what data each subsystem needs and where it comes from. Control is not represented in the figure because it is an intermediary between subsystems, hence all subsystems inherently depend on Control.

1.3 Structural Diagram

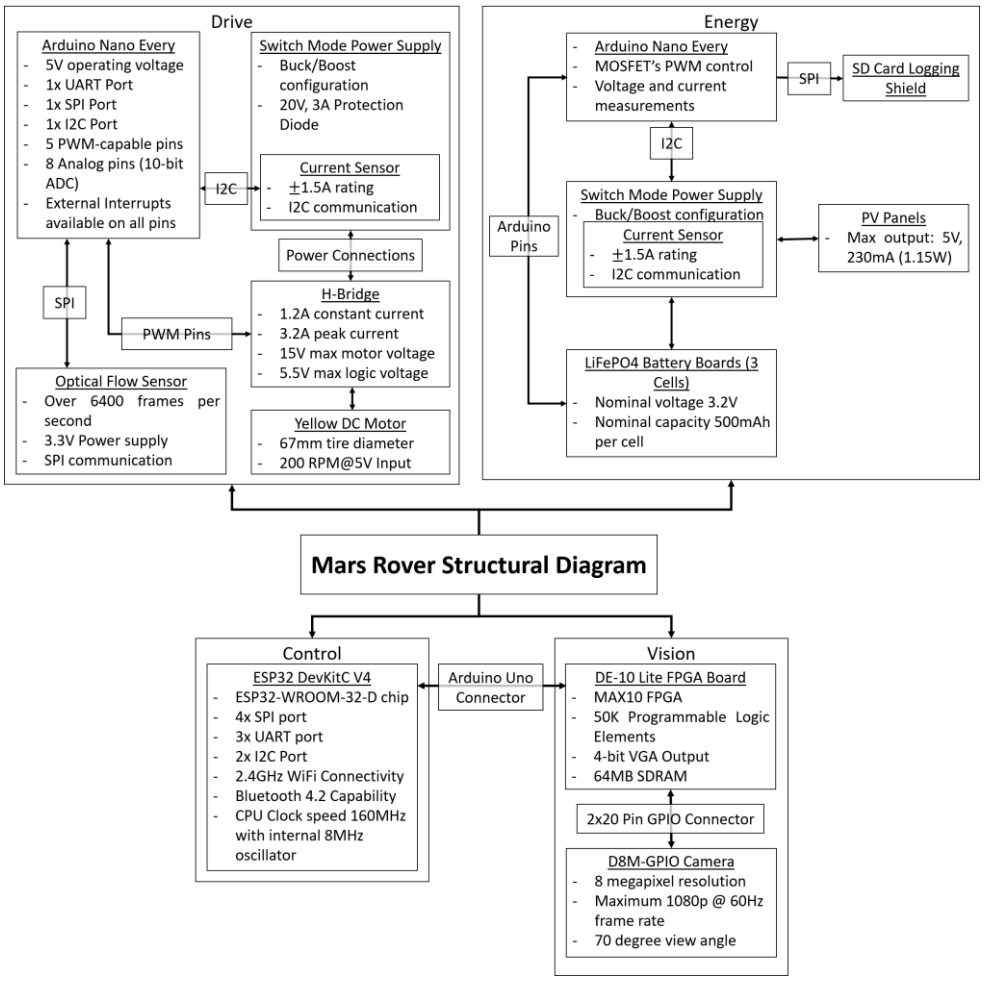


Figure 2: Structural Diagram

2. Design, Implementation and Evaluation of Subsystems

2.1 Control Subsystem

The main task of the Control subsystem was to communicate with other subsystems. Hence, the main program flow and logic of the autonomy of the rover was built into Control. Thus, Control took up the secondary role of maintaining and updating the state of the system and managing the timing of communication with other subsystems.

2.1.1 Overall Program Flow

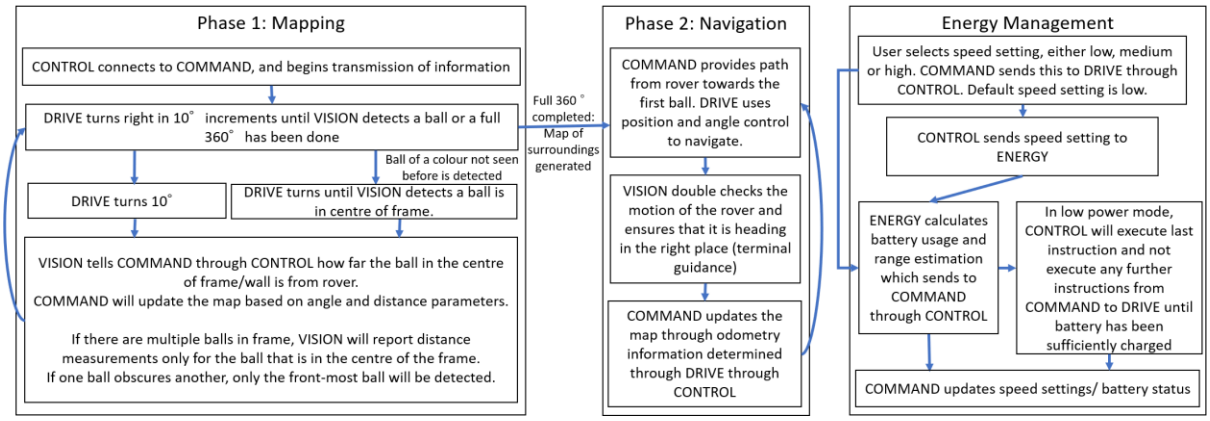


Figure 3: Program Flow of Control

To achieve the functional requirements of the rover, the main program was split into 3 parts.

Phase 1 gathered information about the rover's immediate surroundings and sent it to Command to build a map of the local area. This was done via a full 360° rotation in 10° increments. 10° was chosen as it provided sufficient precision in identifying objects and boundaries around a rover, while taking a reasonable number of rotations. Each time the rover stopped after a 10° rotation, Control polled Drive for its actual angle of turn and Vision for the objects detected with their corresponding estimated distances. This information was then sent to Command to build the map.

Phase 2 allowed the rover to navigate its immediate vicinity using the map, and approach objects of interest while avoiding obstacles. Command used the A* path-finding algorithm to calculate the shortest path that avoided any previously seen obstacles and sent instructions to the rover to follow said path. After every instruction, Vision sent objects that it saw to Command via Control to confirm that the motion was correct. This prevented the rover from colliding with previously unseen objects, as Command would update the map with the new information. This would repeat until the rover reached the object of interest. Vision would then adjust the rotation so that the object was in the centre of the frame.

Low-power Mode was also built into the program, with Control monitoring battery health from Energy. If battery dropped below 15%, Control would finish executing any instruction it was on and update Command. Then, Control would stop executing further instructions until the battery returned to 80%. This autonomy on Control would prevent over-discharging of the battery and removed the need for a user to manually shut down the rover.

2.1.2 Timing Diagram of Communication

To start the program, Command sends a {“start”: 1} message in JSON format to Control, which causes the program to enter Phase 1.

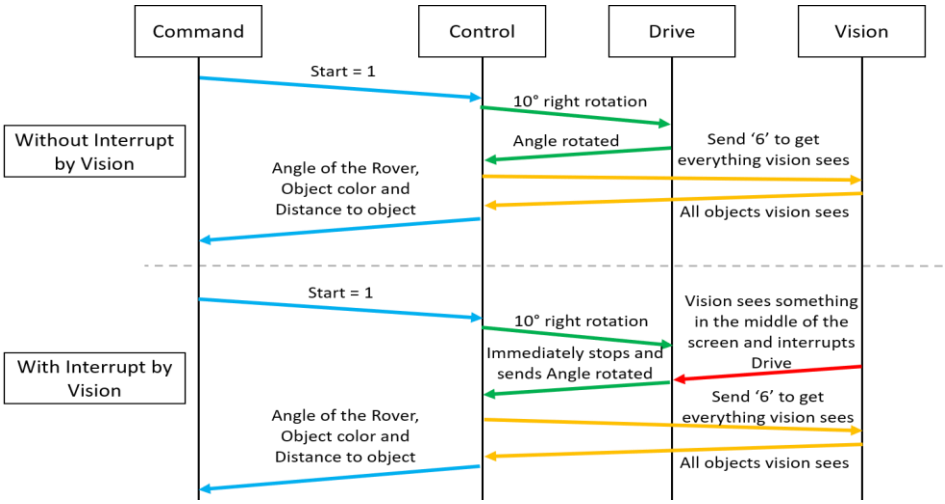


Figure 4: Phase 1 Timing Diagram

Phase 1 begins with a hard coded 10° rotation sent to Drive. Control does nothing until the rover stops. Rotation can stop in two ways: finishing the 10° rotation or by a hardware interrupt from Vision when it detects a coloured ball in the centre of the frame. Since Vision can only report an accurate distance when the object is in the centre of the frame, having a hardware interrupt ensures Drive stops the rover immediately, allowing Control to read accurate distance data from Vision. This information of angle, object and distance would be then sent to Command to build the map.

After Control records a full revolution in Phase 1, Control updates Command that it is proceeding with Phase 2 via a {“done”: 1} message in JSON format. Command then begins to build up the map with the information it received.

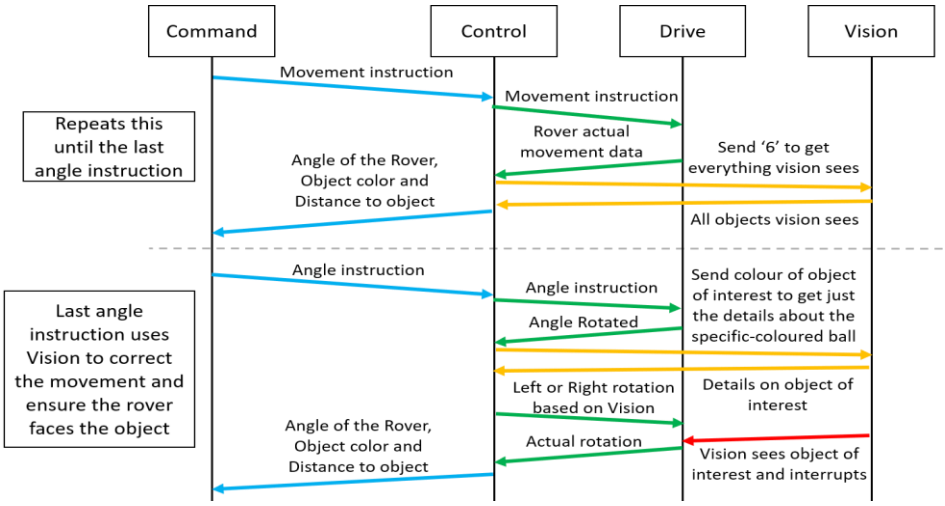


Figure 5: Phase 2 Timing Diagram

Command first sends instructions to the rover based on the shortest path. After Drive completes its movement, Control retrieves data of any objects from Vision before sending it to Command. This repeats until the last rotation since the rover should be facing the selected object. If so, Control reads from Vision after executing the last rotation to ensure that the ball is in the centre of the frame. If not, Control sends a rotation instruction to Drive in the direction indicated by Vision, with Vision interrupting Drive once the object is in the centre of the frame.

Energy is always in constant communication with Control. Control will send the speed of the rover to Energy every second to calculate battery life and range estimation data, and Energy will send this data back to Control. The battery life and range estimation data will be sent to Command at every instance that Control sends data to Command.

2.1.3 Inter-Module Communication

2.1.3.1 Communication Protocol

2.1.3.1.1 Energy and Drive

The choice of communication protocol with Energy and Drive was limited due to hardware. The PCB (Printed Circuit Board) that Energy and Drive used only exposes the Receiver and Transmitter pins of the Arduino Nano Every, which meant that only communication via UART (Universal Asynchronous Receiver/Transmitter) was possible.

2.1.3.1.2 Vision

There were 3 options to choose from, UART, SPI (Serial Peripheral Interface) and I2C (Inter-Integrated Circuit). However, the UART IP was found to take up too much on-chip resources. Hence, a much lighter-weight communication protocol was required, either SPI or I2C. After reviewing the requirements with the pros and cons of each protocol, SPI was chosen as the communication protocol with Vision.

A common criticism of SPI is that it requires 4 wires instead of 2 wires required by I2C, and I2C can have more slaves in communication (On Amlendra, 2018). However, the rover only required 1 master and 1 slave and there were sufficient GPIO pins on the ESP32. Hence, this was no issue. SPI could transmit data faster than I2C with less power drawn, reducing latency of communication, and increasing power efficiency, which was important for the rover's implementation (On Amlendra, 2018). SPI could support full duplex as well, and this could bring greater utility for implementation than I2C.

2.1.3.1.3 Command

MQTT (Message Queuing Telemetry Transport) network protocol was touted to be a superior protocol for Internet-of-Things application. MQTT provides services that are useful when delivering small amounts of data and is especially useful with many clients and sensors (MQTT, 2021). However, the initial aims of the rover were to not only send small packets of data describing the state of the rover system, but also to send pictures or even stream a video of what Vision sees. MQTT is not suited to do video streaming (IDA HÜBSCHMANN, 2021), but other application protocols like WebSocket and HTTP could fulfil that functionality (Santos Rui & Santos Sara, 2019) (Ted Young, 2019). However, the video streaming functionality could not be developed in time for the deployment of the rover, but WebSocket was still chosen as the communication protocol for its potential to further develop this functionality in the future. WebSocket was more suitable compared to HTTP due to its higher speed as bi-directional data can be passed continuously over a single open connection, unlike HTTP. (Arpit Asati, 2019) Tests conducted between WebSocket and HTTP confirmed that WebSocket was superior due to its speed and low latency.

2.1.3.2 Data Field Design

2.1.3.2.1 Energy and Drive

Communication with the Arduinos was via UART, and the data was transmitted over `Serial`. The data was read as a string and the string was split into the individual data using the newline character ('\n').

The sequence of data sent and received was fixed, so each byte sent corresponded to a particular data field. There were no wasted bytes as all fields were required for each transmission and provided useful information. Table 3 shows the byte fields to and from each component in the order which was implemented in the programme as well as its corresponding data type.

Control to Drive	Drive to Control	Control to Energy	Energy to Control
Direction (int)	Angle/Distance (int)	Speed (int)	Battery Life (int)
Distance/Angle (int)			Range estimation (int)
Speed (int)			

Table 3: Data transmitted and the order in which it was transmitted from top to bottom

2.1.3.2.2 Vision

The data field design with Vision aimed to send the most information in the fewest bytes, saving memory and increasing power efficiency. The information describing each object was sent using 2 bytes. The coloured balls were given numerical ID's from 1-5. Walls were designated as ID 6, and any other values were considered "invalid". This meant that 3 bits were required for object identification. In addition, the Correction Factor indicated if the object was to the right, left or centre as values 1, 2, or 0, respectively. This was used in Phase 2 to correct the angle of rotation for final angle instruction.

When there was no data, the registers would be set to 0xFF, or -1 in two's complement. Error handling was put in place to ensure that the information received was not corrupted, in which case it would be 0x00, triggering another read of Vision's data. Table 4 shows the format of the bytes sent by Vision, and more details on the r and θ values can be found in Section 2.2.2.2.5.

	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
Byte 1 (Ball)	Distance measurement							
Byte 2 (Ball)	Correction Factor		**X**	**X**	**X**	Ball Colour ID (1-5)		
Byte 1 (Wall)	**X**	r value						
Byte 2 (Wall)	θ value					1	1	0

Table 4: Format of the 2 bytes sent by Vision for each object

Conversely, instructions to Vision were in the form of a command enumeration.

Number	Meaning
0	Idle. Clear all state flags.
1	Seek out Red Ball (colour 1)
2	Seek out Green Ball (colour 2)
3	Seek out Blue Ball (colour 3)
4	Seek out Yellow Ball (colour 4)
5	Seek out Purple Ball (colour 5)
6	Seek out all colours (Phase 1 Exploration)
7	Low-power mode

Table 5: Values to Vision

When seeking out a specified colour, Vision only reports the presence of that colour to Control. When seeking out all colours, Vision reports the presence of all colours in the frame to Control.

2.1.3.2.3 Command

Bi-directional communication with Command was done via 1 URL. The data was sent in JSON file format, which is a lightweight data format and is easy to implement. 4 different JSON formats were used, 2 sent by Command and 2 sent by Control, as shown in the following Tables 6-8:

Sent by **Command** to start the program		Sent by **Control** to indicate end of Phase 1	
Start	'1' to start	Done	'1' to indicate end of Phase 1

Table 6: Data sent to start the program and to start Phase 2

Data from **Control** to **Command** to update the state of Rover	
Object Colour (int)	'0' represents data referring to rover's movement, 1-5 represents data about a ball colour and '6' indicates data about a wall
Angle (int)	Angle of rotation done by the rover / Angle of an object with respect to 0°
Distance (int)	Distance moved by the rover / Distance of an object with respect to rover
Battery (int)	Battery health
Range Estimation (int)	Estimation of range of rover

Table 7: Data sent to Command during Phase 1 and 2

Data from **Command** to **Control** as instructions to move the rover	
Instruction tag (int)	This is used as the database primary key
Angle (int)	Instruction for angle of rotation to be done by the rover
Distance (int)	Instruction for distance to be moved by the rover
Object Colour (int)	This gave the object of interest for Vision to focus on
Speed (int)	Speed of the rover
Last path (int)	This indicated the last 2 instructions so Vision could look for the object of interest. As the instructions alternated between moving forwards or rotating, the last rotation instruction would be in 1 of the last 2 instructions. '1' is the 2nd last instruction, while '2' is the last instruction. '0' corresponded to all other instructions.

Table 8: Data received from Command during Phase 2

2.1.4 Implementation

2.1.4.1 Hardware

There was a problem during the implementation of SPI communication with the FPGA. The autonomy of the rover meant that FPGA had to be the Master to initiate the communication. For example, in Phase 1, when Vision detected an object of interest in the middle of its view, Vision needed to send information of this to stop the rover's rotation. However, the SPI library in the Arduino IDE does not support ESP32 as slave, hence an alternative method was developed, which used the FPGA to perform a wired hardware interrupt to initiate a communication with the ESP32. While the interrupt was initially designed to be wired to the ESP32, having the wired interrupt directly to Drive could bypass the ESP32 and reduce latency which was crucial in Phase 1. Control's program was then adapted to only retrieve data from Vision once Drive communicated that the rover had stopped, so the data read from Vision was static and accurate. This provided the FPGA a Master-like behaviour in communicating with the ESP32.

Hence, an extra wire was needed to connect Drive and Vision through the ESP32. Table 9 shows the wired connections between ESP32 and other subsystems:

Energy Arduino to Control ESP32	**Drive Arduino to Control ESP32**
GPIO 2 of ESP32 to TX pin on Arduino	GPIO 16 of ESP32 to TX pin on Arduino
GPIO 4 of ESP32 to RX pin on Arduino	GPIO 17 of ESP32 to RX pin on Arduino
GND of ESP32 to GND on Arduino	GND of ESP32 to GND on Arduino
	GPIO 15 of ESP32 to Digital Pin 4 on Arduino for interrupt

Table 9: Physical wires connecting different components

2.1.4.2 Platform for Programming

To program the ESP32, the Arduino IDE was used. There were other options considered, namely MicroPython and ESP32-IDF. However, to use MicroPython, the chip must run a Python interpreter on top of the actual code, which increases power consumption and reduces performance (Damien George & Daniel Campora, 2014). To increase power efficiency, C++ was chosen over MicroPython. Furthermore, after assessing the project requirements, the libraries available in the Arduino IDE were sufficient in achieving the requirements, hence using ESP32-IDF was not necessary.

2.1.5 Evaluation of Subsystem

2.1.5.1 Evaluation of testing results

Evaluation of Control was done together with other subsystems explained in Section 3.3.

2.1.5.2 Future Extensions

More processing could be done on the ESP32. On Mars, communication with the rover from Earth takes 20 minutes (National Aeronautics and Space Administration, U. S. A., 2021), hence building up a simple map could be done locally on the ESP32 without relying on commands from the web server. Video streaming could be implemented with WebSocket as originally planned since video footage is important when exploring a new planet.

2.2 Vision Subsystem

2.2.1 Design considerations

Apart from the Optical Flow sensor, the rover only senses its environment through Vision. Therefore, it must deliver accurate and prompt data to Control for the rover to attain the goals in Section 1.1.

As the system is FPGA-based, there was a lot of freedom in design, with constraints being imposed mostly by other subsystems. However, the available resources on the provided MAX-10 FPGA provided an upper bound on the complexity of the algorithms implemented.

2.2.2 Implementation of Subsystem

There were two challenges to be solved by Vision: Detecting objects of interest and avoiding obstacles.

The area for the rover to “explore” was defined using an arena with walls made of black cardboard, and a lightly coloured floor. This created a line between the floor and the walls, which should be detected as an obstacle by the rover. In addition, the rover would also need to detect and distinguish between differently coloured balls. More information about the arena is in Section 3.1.

During Phase 1, all coloured balls would be considered as objects of interest, as Command had not yet determined the colour of ball to be investigated. Subsequently, after a colour has been specified in Phase 2, all other colours would be considered as obstacles.

The coloured balls and uncoloured lines represent different challenges in object detection, requiring different methods of detection. A block diagram of the Vision subsystem is below.

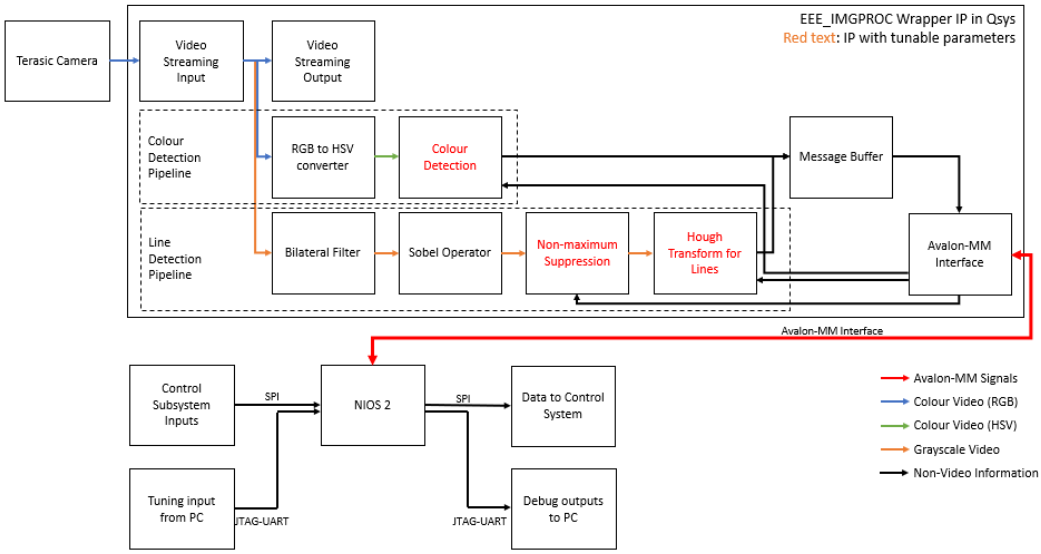


Figure 6: Block Diagram of Vision Subsystem

There are two detection pipelines that process image data in parallel – the colour detection pipeline for detecting balls and the line detection pipeline to detect walls. These pipelines output data to both a Message Buffer to transfer their output to the Nios II processor, and also to the Video Streaming output (not shown in the diagram) for visual debugging.

A summary of the resources used by the design can be found in Appendix 1.

2.2.2.1 Colour Detection Pipeline

RGB data is not robust to external lighting changes. Given that the objects of interest are spheres, their apparent brightness varies across the object, with the areas on top (directly exposed to light) appearing lighter with higher individual RGB values, and areas below (in shadow) appearing darker, with lower individual RGB values. This makes it difficult to set exact thresholds for detection.

In contrast, using the HSV (Hue, Saturation and Value) colour space separates the "colour" component of information into the Hue component, while giving information about the Saturation and brightness (Value) of the colour as well (H. D. Cheng et al., 2017).

This is useful for detecting colours, as the colour of the pixel seen can be determined by setting thresholds directly on its Hue value. The Saturation and Value parameters can also be tuned based on the specific colour of interest. After thresholds are appropriately tuned for each colour, bounding boxes can then be drawn.

2.2.2.1.1 RGB to HSV

A formula for RGB to HSV conversion was used (Rapid Tables, 2021). However, it was modified to meet the limitations of the FPGA platform. Details on how this was implemented are in Appendix 2.

2.2.2.1.2 Colour Detection Module

The HSV values are fed into a Colour Detection Module. One module is instantiated per colour of interest.

Thresholding and comparison are performed for each colour, with the varied parameters being `hue_h`, `hue_l`, `satThresh`, and `valThresh`. An upper and lower bound are required for the Hue value as different colours are represented as different ranges from 0-255, while only a lower bound is required for Saturation and Value as they represent intensity of colour and brightness, respectively.

The colour Red is special, as it wraps around the colour wheel, with Hue values ranging from 300° to 30°. This requires special treatment of Hue thresholding. The formulae to determine whether a given colour is detected are given below.

```
detected = hue_detect && sat_detect && val_detect
sat_detect = Sat >= satThresh
val_detect = Val >= valThresh
hue_detect = colour_red ? (Hue>=hue_h||Hue<=hue_l):(Hue<=hue_h&&Hue>=hue_l)
```

To avoid the effects of coloured background noise, for instance speckles of colour in the background, the module only generates a bounding box (BB) for the largest contiguous volume for each colour in the image. This is attained with two accumulators.

Contiguous volumes are defined by their individual bounding boxes (top, bottom, left and right coordinates) and their sizes.

Pixels are read from the top left of frame to the bottom right. Thus, in the x-direction, a pixel is adjacent to an existing volume if its current x-coordinate is within the proximity threshold of the right edge of the volume's bounding box.

A pixel is adjacent to an existing volume in the y-direction if it is within the proximity threshold of the bottom edge of the volume's bounding box, and its x-coordinates are between the right and left edges of its bounding box. Adjacent pixels will be considered for addition to contiguous volumes.

To illustrate, orange pixels in the diagram below are already in a contiguous volume, while green pixels are potential pixels to be added to the existing volume. The yellow lines represent the current bounding boxes of the volumes. Assume that the proximity threshold given is 2 pixels.

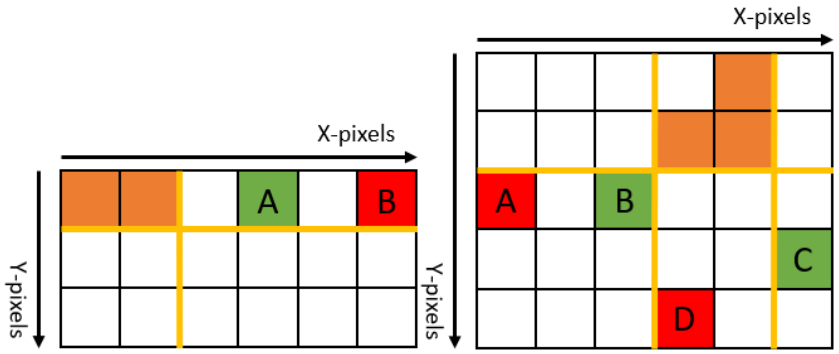


Figure 7: Illustrating adding pixels in the (a) x-direction (left) or (b) y-direction (right)

In Figure 7 (a), pixel A will be added to the existing volume, as it is within two pixels of the right edge of the bounding box. In contrast, pixel B would not, as it is further than two pixels away.

In Figure 7 (b), pixel A will not be added to the volume, as while it is one pixel below the bottom edge of the bounding box, it is more than two pixels from the left edge. Similarly, pixel D will not be added, as it is more than two pixels from the bottom edge. Pixels B and C will be added, due to their proximity to the edges.

Two contiguous volumes are tracked with separate accumulators to obtain one with the largest size.

When a new pixel is detected, there are three possibilities:

1. The pixel is to be added to both volumes.
2. The pixel is to be added to only one volume.
3. The pixels is not to be added to either volume.

In the first case, the two volumes need to be merged. The value of the resulting bounding box is taken as the union of both volumes' bounding boxes, while the size of the new volume is equal to the sum of both volumes' sizes plus one. The value of this resultant bounding box is written to the larger of the two original volumes, while the smaller of the two is cleared.

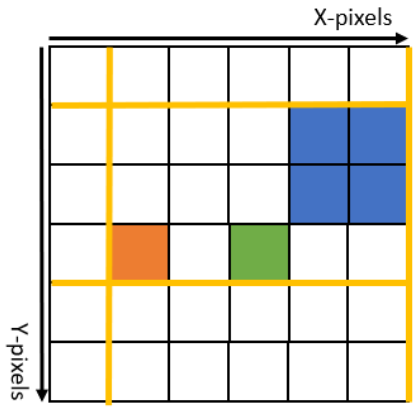


Figure 8: Logic for merging volumes

If the orange and blue regions have already been designated as volumes, and the green pixel is detected next, the resulting bounding box is taken as the union of the two volumes, shown in yellow.

The size of the resultant volume is 4+1+1, the sum of the size of the constituent volumes plus one.

After this, the updated bounding box values and size will be assigned to the blue volume, while the values for the orange volume will be cleared.

In the second case, the pixel is added to the related volume, and its bounding box and size are updated.

In the third case, the smaller of the two volumes by size is overwritten with the coordinates of the new pixel.

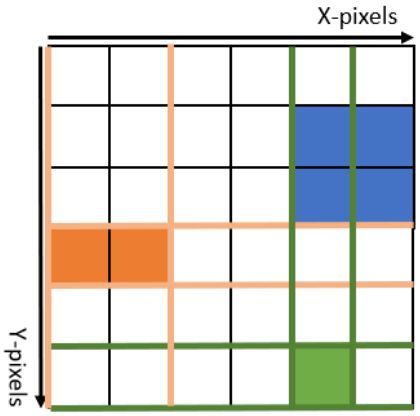


Figure 9: Logic for overwriting volumes

If the green pixel is detected after the orange and blue volumes have previously been detected, the orange volume will be overwritten by the green volume.

The size of the new volume is then 1 instead of 2, and its bounding box values will also correspond to the green ones instead of the orange ones.

Finally, at the end of the frame, the size the volumes are then put through a size threshold, to prevent bounding boxes to be drawn when only coloured speckles are detected with no ball in the image frame. The bounding box values of the larger of the two volumes (by size) are then output as the bounding boxes for that specific colour.

If there is no valid volume detected in the entire frame, then the bounding box values are set to -1 for error handing. An example of the output of the Colour detection is in Appendix 5.

2.2.2.2 Line Detection Pipeline

The distance to walls can be detected using the lines at the transition between floor and wall.

The walls were made of black cardboard to make it easier to distinguish between the dark-coloured wall and light-coloured floor. To detect lines, the image is converted from colour to grayscale. Following this, Canny Edge Detection, followed by the Hough Transform for lines, was used.

There are many ways (Wilhelm Burger & Mark J. Burge, 2010) to convert RGB to grayscale, such as colourimetric conversion, but the sample code was unchanged: `grey = red/4 + green/2 + blue/4`. It is simple and yields satisfactory results. The green component is only divided by 2 because

Bayer sensors, like in the camera provided, have two green sensor elements for each red and blue sensor to mimic the human eye.

Canny Edge Detection is a multi-stage algorithm to extract edges from a given grayscale image. After image smoothing to remove noise, the intensity gradients are found, corresponding to the vertical, horizontal, and diagonal edges in the image (John Canny, 2017).

Subsequently, gradient magnitude thresholding is applied, finding the locations with the sharpest change of intensity value, thinning edges to be one pixel wide.

Lastly, a threshold is applied to filter out false negatives. Hysteresis thresholding can be used for a more accurate threshold.

After edge detection is applied, the Hough Transform is then used to find orientation of lines in the sample space. This can then be used to calculate the distance of the wall.

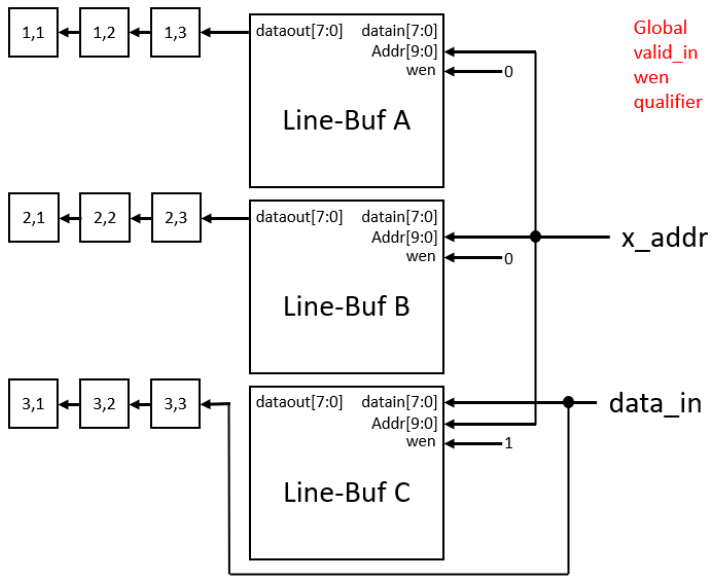
2.2.2.2.1 Convolution Implementation

The first three components of the Canny Edge Detection Algorithm are convolution operations, which involve replacing the original value of pixels with the weighted sum of the pixels in their vicinity, based on the given convolution kernel (Jamie Ludwig, 2007).

While arbitrary access to pixels is possible when images are stored in memory, the FPGA views images as a stream of pixels, with no ability to access elements in the past or future. Hence, additional memory is required to implement the convolution operation, to "save" past values.

The line-buffers are implemented as 1-port RAM, holding the grayscale value of one pixel. They have an element for each pixel in the x-direction. For timing reasons, their outputs are not registered.

As the Sobel and Non-Maximum Suppression kernels are both 3x3 kernels, it was decided to use three line-buffers. They take turns to be written to, depending on the current line of the convolution. The `wen` signals of the line buffers are grouped in a one-hot bus that is shifted every time a line comes to an end, as per the code snippet below.



The buffer written to is rotated every line, with the previous two lines existing in the past two buffers.

The graphic illustrates this. Registers are used to keep track of the three lines of interest, with values being shifted down from right to left as new data enters.

For any given line read, the input data is fed into the bottom-most line-buffer, with the new data also passing directly into the shift registers on the bottom row.

Figure 10: Illustration of Line-buffer read/write logic

Line buffers A, B, and C are rotated every new line. {A,B,C} rotates from {1,2,3} to {2,3,1} to {3,1,2} as lines enter the buffers. Hence, the data from pixels in lines above the current pixel at the same x-coordinate are fed into the registers above for calculation.

This register allocation makes it easy to apply a convolution kernel, as the convolution sum is now easily computable by multiplying the nine individual values in registers (1,1) to (3,3) with their corresponding coefficients according to the convolution kernel and then summing the result.

Exceptions occur at the edges of a frame, where data might be invalid or missing. However, to simplify computation, these pixels are ignored in further stages of calculation.

2.2.2.2.2 Bilateral Filter

Before edge detection, an image is typically filtered to remove noise or grain, for instance as a result of high camera gain (ISO) settings.

The Bilateral Filter (Haarith Devarajan & Harold Nyikal, 2008) was selected as an imaging pre-processing filter. While a Gaussian filter (Shapiro & Stockman, 2011) is commonly used for such purposes, it blurs both edges and imaging noise. In addition, while the Gaussian filter is a linear filter, which means that it can be computed more efficiently conventionally, this does not apply to the streaming application in this case.

In contrast, while the Bilateral filter requires more computation, it preserves edges in the input image while still smoothing imaging noise by using weights based on both proximity and pixel value, leading to greater filter effectiveness. In addition, it can be performed with no time penalty compared to the Gaussian filter in a streaming application. Thus, it was chosen over the Gaussian filter.

Implementation details of the Bilateral filter are in Appendix 3.

Following filtering, a brightness threshold was applied across the entire image. As the walls are black while the floor is lightly coloured, to remove the impact of unwanted lines in the image, pixels with a value smaller than a threshold were set to 0 (black) while pixels with a value greater than that threshold were set to 255 (white). Results before and after the filtering are in Appendix 5.

2.2.2.2.3 Edge Detection with Sobel Operator

The Sobel operator (Irwin Sobel, 2014)is a discrete differentiation operator which gives the gradient of the intensity of an image at each point, in the x or y directions. Two kernels are used to compute the gradient of the image in the x and y directions. The resultant gradients are called G_x and G_y. The magnitude of the gradient is then calculated as $G = \sqrt{(G_x^2 + G_y^2)}$. The values are then calculated using the convolution machinery described above. Detailed implementation is in Appendix 4, with results in Appendix 5.

2.2.2.2.4 Non-Maximum Suppression

Non-maximum Suppression looks at the magnitude and direction of the gradients produced by the Sobel operator. It then picks out lines with the maximum intensity along the direction of the gradient.

Typically, the direction of the gradient is found as $\arctan\left(\frac{G_y}{G_x}\right)$. However, the inverse tangent function is difficult to compute in hardware. In addition, only four directions need to be considered: horizontal, vertical, and the two diagonal orientations. Hence, an approximation is done instead.

To afford more resolution, the value of G_y is first left-shifted by 3 bits, so the quotient of $\frac{G_y \ll 3}{G_x}$ is computed.

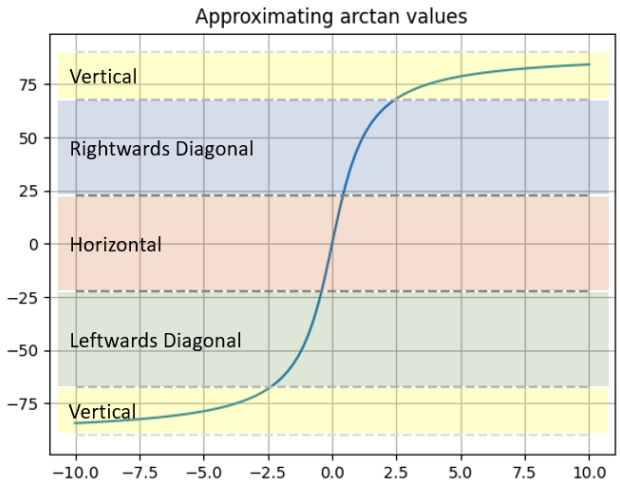


Figure 11: Quantization of Arctan function

From this graph of the arctan function, it is possible to evenly divide the graph into regions centred around 0° (horizontal), ±45° (rightwards and leftwards diagonal, respectively) and ±90° (vertical).

The lines dividing the four regions are set at the midpoint between their y-values, at ±22.5° (x=±0.414) and ±67.5° (x=±2.414).

Therefore, taking the right shift into account, the threshold values for the quotient also need to be multiplied by 8, to be ±3.3 and ±22.5, respectively.

After orientation detection, the centre pixel is compared with its relevant neighbours. The four possibilities are shown below, with the values to be compared against in orange, and the centre pixel in green.

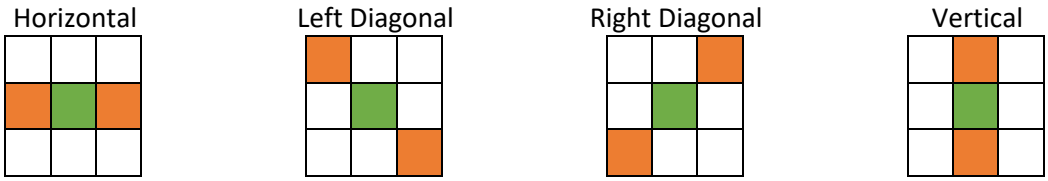


Figure 12: Pixels to be compared for different orientations

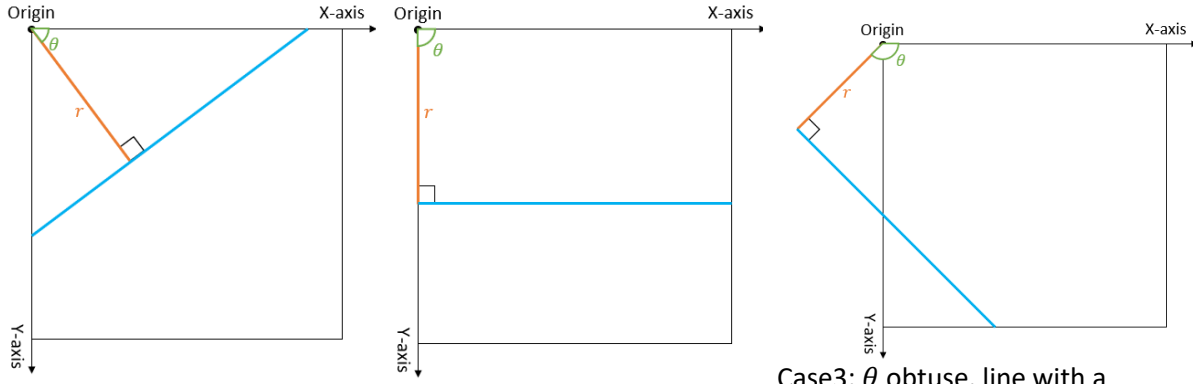
If the centre pixel is greater than its neighbours, then it is a maximum value. Else, it is set to 0.

Typically, hysteresis thresholding is performed on the output of non-maximum suppression, with an upper and lower confidence bound implemented. Pixels with value greater than the upper bound are designated “strong” edges while those with values below the lower bound are discarded. The values in between the two bounds are designated as “weak” edges. The picture is iterated through, and any “weak” edges connected to “strong” edges are preserved, while “weak” edges without any such connections are discarded.

However, this method requires the entire frame to be within memory and is difficult to do with the given streaming implementation. Therefore, a single threshold is used to determine whether a line is an edge or not. Testing revealed that this approach yielded satisfactory results, shown in Appendix 5.

2.2.2.2.5 Hough Transform for Lines

The Hough Transform (Wilhelm Burger & Mark J. Burge, 2010) detects features with specified parameters using a voting procedure. In this case, lines with parameters r and θ are detected, where r is the distance from the origin to the closest point on the line, and θ is the angle between the x-axis and the line connecting the origin with the closest point.



Case 1: θ acute, line with a negative gradient.

Case 2: $\theta = 90$, horizontal line.

Case3: θ obtuse, line with a positive gradient.

Figure 13: Illustrations of possible values of r, θ

The Hough transform is able to determine the most likely r, θ parameters for the given image. With these parameters, the most prominent line in the image can be inferred.

The Hough transform algorithm iterates through each point where an edge is present, and also through a range of angles. It calculates a corresponding radius with the formula $r = x * \cos(\theta) + y * \sin(\theta)$, and increments the value at an accumulator matrix with coordinate r, θ. After iterating through all valid values of x, y, and θ, the coordinates of the element with the highest value in the accumulator matrix are chosen as the output values r, θ.

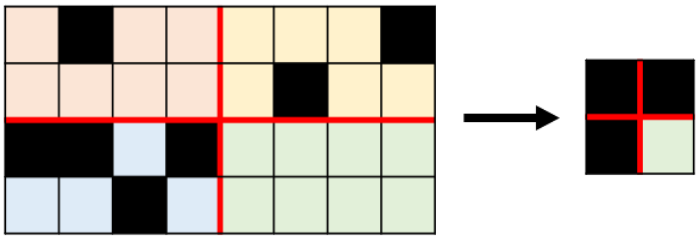
In pseudo-code, the algorithm is as follows:

```
for x in X_pixels:
  for y in Y_pixels:
for theta in Theta_range:
  if img[x][y] is edge:
    r = x*cos(theta)+y*sin(theta)
    acc[r][theta] += 1  # voting procedure to find r,theta
max = 0
r_out = 0
theta_out = 0
for theta in Theta_range:
  for r in r_range:
if acc[r][theta] > max:
  max = acc[r][theta]
      r_out = r
  theta_out = theta
return r_out, theta_out
```

However, it is not feasible to scan the entire image at the full resolution of 640x480, as the FPGA is unable to store the entire frame in memory. In addition, it is also not possible to iterate through the full range of 0-180° for values of θ. Hence, only a sub-section of the frame is sampled, with values being compressed in resolution to save memory.

Lines appear on the ground, which will not cross into the top half of the frame. For accurate distance measurements, only lines near the frame's centre are considered. Hence, only the area in the middle of the frame with resolution 80x240 is relevant. In addition, the image is compressed by 4x in the x-direction and by 2x in the y-direction. The resolution of the image buffer is thus 20x120.

Through empirical testing, the range of values for θ is unlikely to exceed 50-130°. A resolution of 2° was deemed acceptable, so θ had 41 possible values. By taking the Pythagorean distance of the image buffer, r has a maximum distance of 121. The accumulator thus has resolution 41x121.



4x2 pixel groups are compressed using an OR operation. If any of the pixels in the grid are considered edges (in black), then the resultant pixel in the compressed image buffer is considered an edge as well.

Figure 14: Illustration of OR compression operation

Only processing part of the frame also saves memory, since only one image buffer and accumulator are required, as they can be written to, read from, and cleared while different sections of the frame are read. As there is time to individually clear each memory address in the image buffer and the accumulator each frame, an On-Chip RAM implementation can be used, instead of a register-based application with a global clear.

2D arrays, as used in the pseudocode, are implemented by multiplying the second axis coordinate by the size of the first axis. For instance, to access `img_buf[x][y]`, address `x+y*20` is used, as the x-coordinate.

The accumulator addresses are obtained using a look-up table to approximate the sine and cosine functions at possible values of θ. To reduce quantization error, the r-axis was deemed the first axis. As it is not multiplied, quantization error has a smaller effect on indexing of the accumulator.

The lookup tables are implemented in FPGA ROM and contain an integer representing a numerator value, with a right shift by a constant value representing a denominator value. They are indexed by the value of θ. To further reduce quantization error and ensure that the correct accumulator bin is selected, instead of simply shifting right, rounding off is also performed, as shown below.

```
input [13:0] in;
output [7:0] out; // shift right by 6 bits
out = in[5] ? in[13:6]+8'd1 : in[13:6];
// look at most-significant truncated bit to determine round up or down.
```

The Hough transform output is then passed on to the Nios II for further processing. Output images are shown in Appendix 5.

2.2.2.3 Processing of Raw Data

The data collected by the two detection pipelines is written into the message buffer every frame. The Colour Detection fields are repeated for each of the five colours given.

Data Fields	Field 1	Field 2	Field 3	Field 4
Colour Detection	Left BB value	Right BB value	Top BB value	Bottom BB value
Line Detection	r	θ	-	-

Table 10: Data fields read by Nios II from hardware registers

The Nios II processor periodically reads the message buffer to further process the data.

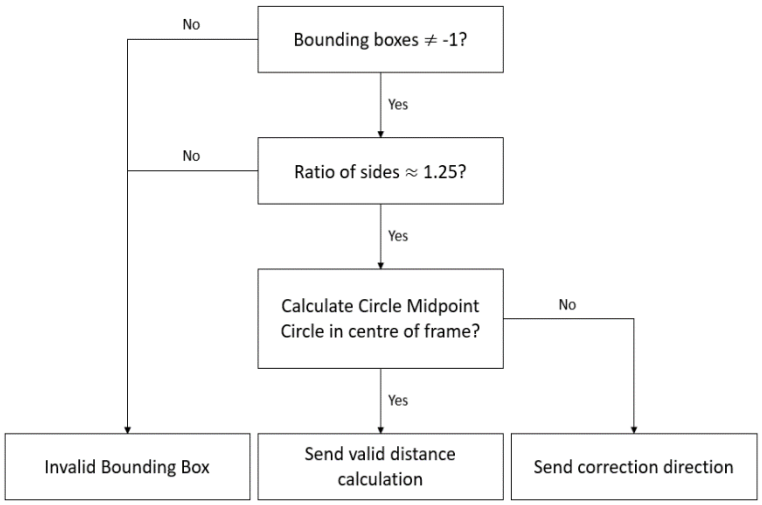


Figure 15: Nios II Post-processing data flow

Processing for the Colour Detection pipeline checks input from hardware registers.

The length and height of the bounding box are calculated as the difference between the left and right, and top and bottom, BB values, respectively.

Their ratios are checked in software, as the data rate is relatively low, and for ease of debugging.

If a bounding box is valid, the ball's (x,y) coordinates can then be calculated as the average of its left and right, and top and bottom, coordinates respectively. If a ball is near the centre of a frame, its distance can be calculated from its y-coordinate. If not, the processor flags it as being on the left or right of the frame, for further action by Control. Details and derivation of the calculations are in Appendix 6.

Similarly, the y-coordinate of a wall can be calculated from its r, θ values. However, as this calculation requires use of trigonometric functions, it is unable to fit in the Nios II's program memory. Therefore, this calculation is performed on Command.

2.2.2.4 External Communication

2.2.2.4.1 Communication with Control

The Nios II communicates with Control as an SPI slave. In software, this was managed by creating a SPI data buffer, which the Nios II then populates with the relevant information. The SPI core generates an interrupt when a read transaction occurs. During this interrupt, the next item to be read is transferred from the SPI data buffer into the Transmit buffer of the SPI core, which allows Control to read the desired values.

To ensure that buffer content was not read whilst being modified, interrupts are disabled when the SPI buffer is being written to. This leads Control to read zeroes from Vision, triggering a re-read of the data. While two or more data buffers could have been utilized to ensure a set of data is always "clean" for reading, such an implementation would involve more logic. In addition, the time required to update the data buffer is short, meaning that the probability of a read error is low. The more basic implementation was thus used.

In all modes except Low-power mode, status LEDs are shown indicating the current command from Control. In Low-power mode, the LEDs are turned off to save power.

2.2.2.4.2 Communication with Drive

For accurate distance measurements, Vision uses a PIO connected to GPIO to trigger an external interrupt whenever a ball of interest is in the middle of the frame. This was to ensure reliable stopping, even when the current state of the Drive microcontroller is unknown, allowing for accurate measurement of ball distance.

To ensure that the same ball does not trigger the same interrupt twice, and to prevent other coloured objects from interfering with Phase 1, state flags are used to ensure that an interrupt is not generated more than once. These state flags are cleared whenever Control sends the "0" command to Vision.

2.2.3 Testing of Subsystem

2.2.3.1 Development Flow

The subsystem was developed with best practices learned from previous digital design modules. Verilog hardware was first developed and simulated using Icarus Verilog with testbenches to ascertain their correctness before deployment onto actual hardware.

Algorithms, such as the Hough Transform, were validated using Python and OpenCV to determine their effectiveness. This made it easy to determine design choices, for instance the level of quantization required for the Sine and Cosine lookup tables.

The display mode was multiplexed between different sources for a visual representation of the output of each IP core. The video output was also compared with "model" images from OpenCV.

Lastly, the SignalTap Logic Analyzer was heavily used to identify further issues with hardware flashed onto the FPGA. The TimeQuest Timing Analyzer was used to trim down the design's critical path.

2.2.3.2 Tuning of Values

Due to the long compile time of Quartus, run-time configurable thresholds were a key feature of the detection IPs. The Memory-mapped interface of the EEE_IMGPROC IP provided was expanded with more write-only addresses which corresponded to thresholds in the detection IPs. This considerably streamlined the tuning process, especially when adapting to areas with differing lighting conditions.

2.2.3.3 Testing Methodology

For independent testing of Vision, a distance calibration rig was made out of cardboard. The FPGA was affixed to the cardboard at the same height it would be on the rover.

Balls were put at various distances and positions along the rig to confirm the accuracy of the distance measurements, and to confirm that each individual colour could be detected and distinguished from the rest. The same material for the walls was also placed at various angles and distances on the rig, testing the accuracy of the wall detection.

An image of the distance calibration rig is in Appendix 7.

2.2.4 Evaluation of Subsystem

2.2.4.1 Evaluation of testing results

Testing using the distance rig showed that distance detection of balls was highly accurate, with an error of ±2cm, given that the bounding box drawn around the ball encompassed it entirely. However, this was difficult to ensure. Accounting for the dark underside of balls sometimes meant that parts of the background or floor would also be detected, giving inaccurate bounding boxes.

In addition, choice of floor material was important. The floor could not be too reflective, as it might lead to false positives from the reflections from the ball. In addition, it could not be coloured too similarly to any of the ball colours.

Wall detection worked, but with a larger ±5cm error margin. This was due in part to the lack of resolution available to the algorithms used. The distances were typically over-estimates, which was undesirable as the rover might hit a wall that was unexpectedly closer.

In conclusion, while Vision could attain its stated goals, it was unable to be robust to a variety of external circumstances. Further error handling would be needed to improve its reliability.

2.2.4.2 Future Extensions

Extensions to Vision would mainly be in their ability to improve the robustness of detection despite variable external circumstances. Autofocus (Xin Xu et al., 2011), and Auto-Exposure capabilities (Jarosław Bernacki, 2020) could be added to ensure broadly similar detection conditions despite lighting conditions.

Auto White Balance (Becki Robins, 2019) could be used to eliminate the effects of different shades of lighting in the environment. This would allow for the hue thresholds for each colour to remain constant, adding another layer of robustness. Alternatively, the rover could carry its own lighting sources with known colour temperature, which would also allow it to operate at night.

The Hough Transform used for lines could also be used for circles, allowing a secondary detection mechanism for the coloured balls. Accurate detection of circles would allow for more consistent distance measurements of balls, less tied to the accuracy of the colour detection mechanism.

Lastly, to reduce power consumption in Low-power mode, Vision could be configured to process data at a slower rate or deactivate itself completely.

2.3 Drive Subsystem

2.3.1 Design Considerations

2.3.1.1 Hardware Design

At the hardware level, the Arduino Nano Every is a microcontroller which controls three main modules in Drive, seen in Table 11. (See Appendix 8 for provided power circuit board connections and hardware design.)

<table>
<tr><th>Main Modules</th><th>Function</th></tr>
<tr><td>Buck closed loop mode</td><td>

Regulates the voltage output, <code>Vm</code>, at nearly fixed value
<code>Vm</code> can be increased or decreased to adjust rover speed
Alternatively, changing the pulse-width modulation (PWM) wave values sent to pins 5 (<code>pwmr</code>) and 9 (<code>pwml</code>) changes the rover speed. (Refer to Section 2.3.2.2)
To stop the rover, a PWM value of 0 is sent to pin 5 and 9.

</td></tr>
<tr><td>H-bridges</td><td>

Arduino controls signals sent to the H-bridges (pins 20, 21)
Sending <code>DigitalWrite</code> to pins 20 (<code>DIRL</code>) and 21 (<code>DIRR</code>) sets the HIGH/LOW state of the motors which determines the direction of movement.

<table>
<tr><td>DIRR</td><td>DIRL</td><td>Rover movement</td></tr>
<tr><td>HIGH</td><td>LOW</td><td>Forwards</td></tr>
<tr><td>LOW</td><td>HIGH</td><td>Backwards</td></tr>
<tr><td>HIGH</td><td>HIGH</td><td>Leftwards</td></tr>
<tr><td>LOW</td><td>LOW</td><td>Rightwards</td></tr>
</table>
</td></tr>
<tr><td>Optical flow sensor (via SPI port)</td><td>

Measures the distance moved by the rover, returning 2 values, namely <code>total_x</code> and <code>total_y</code>.
<code>total_y</code> records the actual distance moved in the vertical direction. It increases or decreases accordingly as rover moves forward or backward
<code>total_x</code> records the circumference that the sensor has swept when the rover is turning. It increases or decreases accordingly as the rover turns left or right
Used for closed loop position control (Refer to Section 2.3.2.4)

</td></tr>
</table>

Table 11: Main modules controlled by Arduino

2.3.1.2 Software Design

The software aspect of the subsystem was broken down into 4 main segments, consisting of data format, speed control, turning method, closed-loop position control and 2 testing portions, namely PID tuning and accurate distance and angle measurement. The flowchart below shows how these different parts are related.

A key feature includes the ability to reset the Arduino after each instruction, initialising the sensor position back to (0,0). This allowed for a more effective execution of instructions, avoiding ambiguity as instructions should be executed one at a time. In addition, the inclusion of variable `instruction_done` ensured a minimum of 5 repetitions of the main loop to confirm that the instruction was indeed executed. Furthermore, an interrupt pin from Vision was attached to trigger whenever an object was detected in Phase 1. The rover would stop immediately and its angle relative to its initial reset position would be sent to Control. (See Section 2.1.3.2)

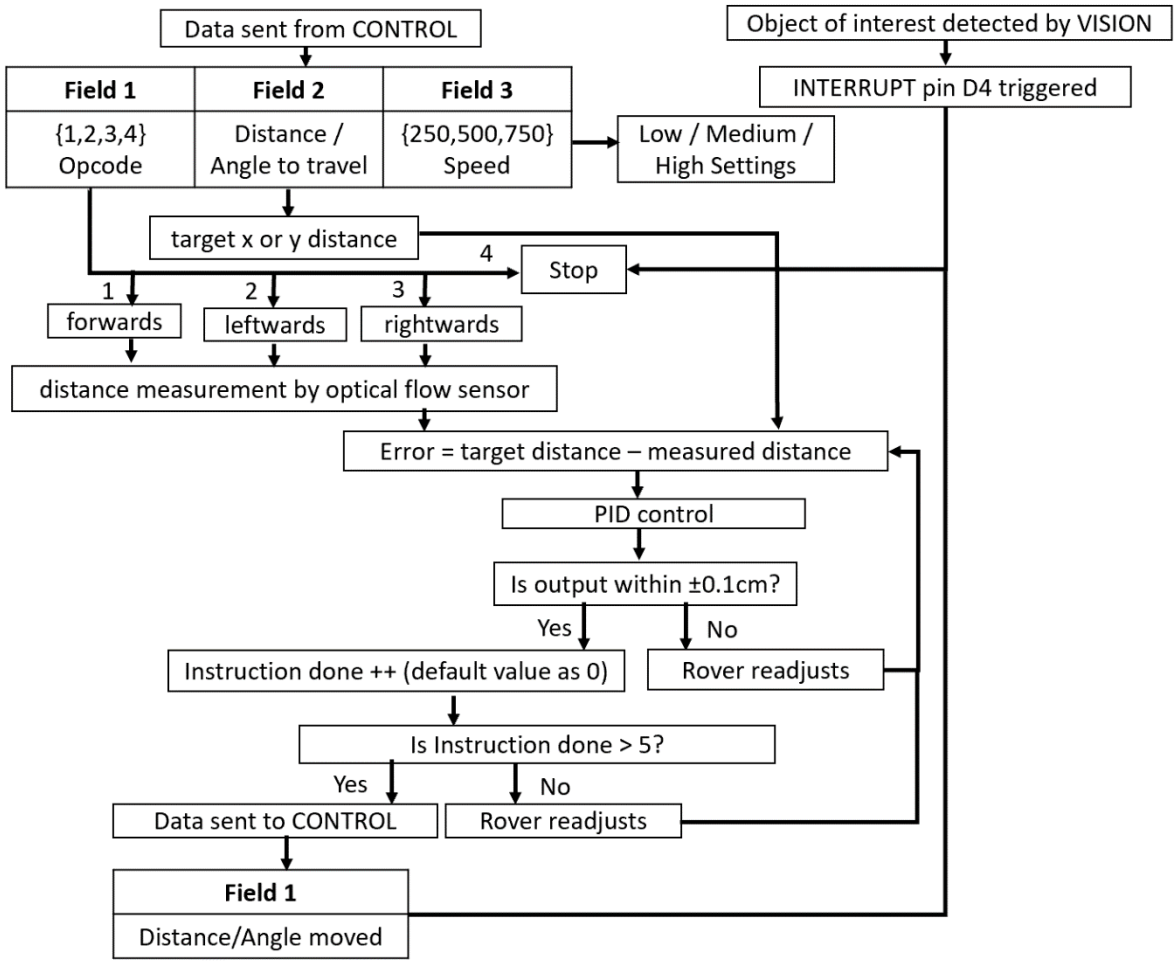


Figure 16: Software design of drive subsystem

2.3.2 Implementation of Subsystem

2.3.2.1 Data Format

All data from Control was sent as integers to Drive in 3 fields. (See Figure 16) The first two fields related to the rover's movement, while the third field related to the rover's speed. (See Section 2.3.2.2)

The first field took the form 1,2,3,4, representing forward, left, right and stop instructions, respectively. The second field represented a distance to be travelled in centimetres for the forward instruction or an angle from 0 to 180 degrees for the left and right instruction.

Backward instructions were not used since Vision's camera was mounted to the front of the rover and was used in Phase 2 to update the location of new potential obstacles that might appear. However, the rover could still move backwards to effect position control when it slightly exceeded its setpoint.

After completion of each instruction, the distance (forward instruction) or angle moved (left or right instruction) as an integer was sent back to Control. (See Section 2.1.3.2)

2.3.2.2 Speed Control

The rover was designed to have three main speed settings, low, medium, and high. Different speeds could be achieved by changing the duty cycle of the PWM wave sent directly to both left and right wheels. It could be set by Command via Control, which could send a PWM value of either 255, 500 or 700 respectively using the third field to determine how fast the wheels turn.

`pwmr` and `pwml` values between 255 and 700 were recommended. During testing of any speed lower than PWM value of 255, the rover moved very slowly or not at all since there was insufficient torque from the motors. Conversely, testing a high speed larger than 700 caused large positional errors due to the time delay between actual and measured distance. After readjustment, the rover's high speed led to over-correction, hence causing high oscillations and poor stability performance.

The default speed settings were set as low for better angle accuracy when the rover was turning, especially during Phase 1. This allowed for an accurate angle facing an object when Vision interrupts to stop the rover. The rover could increase its speed accordingly, such as when travelling longer distances and traversing through steeper or rougher terrain in Phase 2.

To avoid the closed-loop control of the Buck Switch-mode power supply (SMPS) voltage from causing an initial non-constant output voltage with overshoots present, the voltage setpoint of the voltage controller in the SMPS was constant at 5 volts. This avoided the rover from accelerating and decelerating before settling at a constant speed.

2.3.2.3 Turning Method

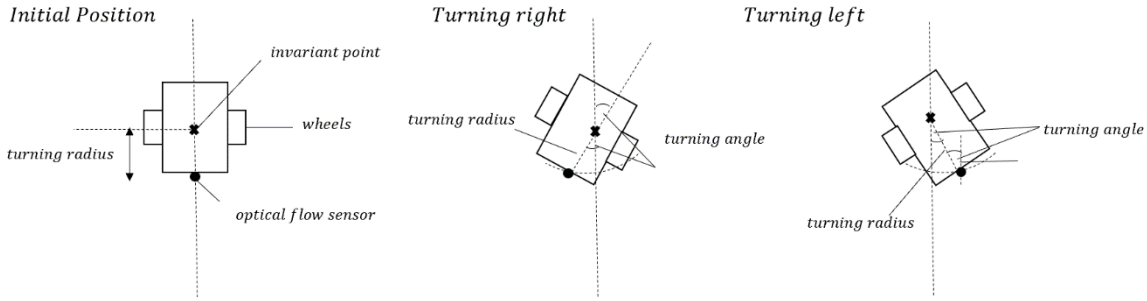


Figure 17: Turning method of rover

The turning method was devised based on how much the rover has travelled in the x direction with respect to its original position. As the rover rotated about an invariant point, the distance between this point and the optical flow sensor, also known as the turning radius of its circular path, was measured to be 15.8cm. The x distance recorded by the optical flow sensor was the total circumference the rover has swept, and not its absolute coordinates. Hence, the target turning angle θ was translated to a target x distance (`temp_x`) of $\pm \frac{2\pi * 15.8 * \theta}{360}$, depending if the rover was turning rightwards (negative) or leftwards (positive).

2.3.2.4 Closed-Loop Position Control

Closed loop position control (Marco Forgione, 2019)(See Figure 18) is required to produce reliable and precise distance measurements, enhance system robustness to external disturbances, thereby

attaining the actual position and improving system stability. The proportional, integral and derivative terms of the PID controller minimises fluctuations, reduces forced oscillations, decreases steady state error and ensures an acceptable reaction time of the controller (Marco Forgione, 2019).

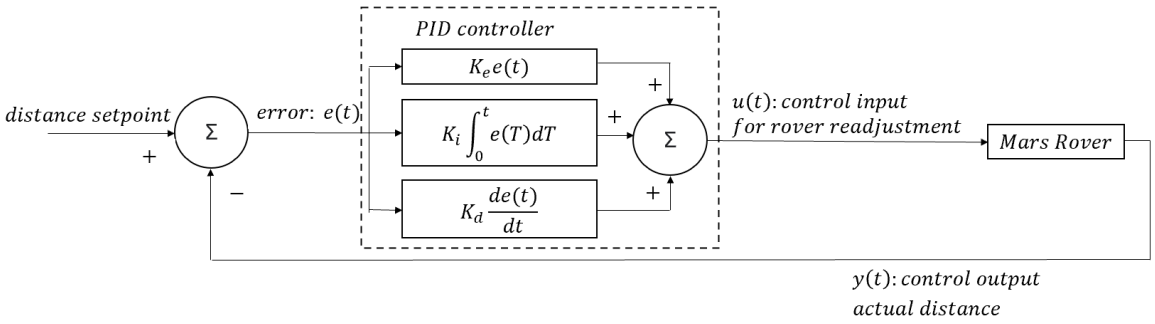


Figure 18: Closed loop position control

The steps and pseudo code for closed loop position control implementation of the rover is as shown.

1. The error term $e(t)$ (`x_error` or `y_error`) was the discrepancy between the target distance (`temp_x` or `temp_y`) and the actual distance travelled (`total_x` or `total_y`).
2. A PID controller (indicated by `pidp` function, adapted from the SMPS voltage controller) calculated $e(t)$ and applied a correction based on the K_p, K_d and K_i terms. The `pidp` function (Refer to Figure 19) used incremental PID programming, avoiding integrations. Its sampling time was 0.008s, or a frequency of 1.25kHz. A fixed sampling time was needed to ensure consistent behaviour of the closed loop.
3. Depending on the PID output, the rover readjusted accordingly to further reduce $e(t)$ and stops if $e(t)$ is small enough.
4. The PID output was set to a threshold of ±0.1cm (`x_threshold/y_threshold`), pegged to the accuracy of the optical flow sensor measuring the actual rover's position.

```
float pidp(float pid_input){
  float e_integration;
  e0p = pid_input;
  e_integration=e0p;

  delta_up = kpp*(e0p-elp) + kip*Ts*e_integration + kdp/Ts*(e0p-2*elp+e2p); //incremental PID programming avoids integrations.
  u0p = ulp + delta_up;  //this time's control output

  ulp = u0p; //update last time's control output
  e2p = elp; //update last last time's error
  elp = e0p; // update last time's error
  return u0p;
}
```

Figure 19: Arduino code for pidp function

Right or Left instruction (Turning)	Forward instruction

`if (right or left)`
`x_error = temp_x - total_x`
`x_error_pid = pidp(x_error)`
`  if(x_error_pid>x_threshold)`
`  go_left()`
`  else if(x_error_pid<-x_threshold)`
`  go_right()`
`  else`
`  stop_rover()`
	`if (forward)`
	`y_error = temp_y - total_y`
	`y_error_pid = pidp(y_error)`
	`  if(y_error_pid>y_threshold)`
	`  go_forward()`
	`  else if(y_error_pid<-y_threshold)`
	`  go_backward()`
	`  else`
	`  stop_rover()`

Table 12: Pseudo code for closed loop position control implementation

When turning, the rover did not gain any y distance during testing. Hence, the closed-loop position control was only applied in the x direction. When moving forwards during testing, the wheels do not start at the same time due to their physical limitations of not being exactly equal. This caused a slight

veering leftwards and rightwards, leading to inaccuracies. The below code was added to stop either the right or left wheel momentarily until the PID output of the error (`x_ferror_pid`) is within the threshold of ±0.1cm (`x_errorcorrection`). The x error (`x_ferror`) is the difference between the current (`total_x`) and previous x distance (`last_x`) detected by the optical sensor, which should be zero ideally as the rover is moving forwards.

```
if (forward)
x_ferror = total_x-last_x
x_ferror_pid = pidp(x_ferror)
    if(x_ferror_pid >x_errorcorrection)
      analogWrite(pwmr, 0)
    else if(x_ferror_pid<- x_errorcorrection)
      analogWrite(pwml, 0)
```

2.3.3 Testing of Subsystem

2.3.3.1 PID Tuning

K_p, K_d and K_i terms were tuned manually in accordance with the following steps to further optimise the closed loop position control (Omega, 2021).

1. K_p, K_d and K_i were first set as zero.
2. A proportional controller was first designed by increasing K_p until convergence of setpoint occurs relatively quickly, without much overshoot. Further halve K_p if there was inaccurate convergence to setpoint.
3. Increase K_i till the process rose quickly enough and oscillated about the setpoint.

Considering the function of each term and design requirements, a PI controller was adopted. Priority was given to minimising any steady-state error, due to interference affecting the rover's ability to travel to its desired position, such as unequal weight distribution and asymmetry of the rover and how the wheels might not start at the exact same time due to slight signal delays sent to the motors. K_d was set as zero since the oscillations observed died out rapidly. A fast response time of the system was deemed less important, as the time needed to send this signal to the rover in Mars from Earth is in the order of minutes.

The same set of tuning parameters ($K_p = 0.072$, $K_i = 0.002$) were used for both distance and angle control as the rover responded well to these values.

2.3.3.2 Distance and Angle Measurement

During testing of 15 sample instructions covering a wide array of distances and angles, a tape measure was used to measure targeted y distances for forward instructions and targeted x distances for left and right instructions (See Section 2.3.2.3). An enlarged protractor print-out was also used to further confirm that the target angle was correct.

Left/Right (degrees)	10	45	90	140	180
Target x distance (cm)	$\pm$2.76	$\pm$12.40	$\pm$24.81	$\pm$38.61	$\pm$49.63
Forward (cm)	10	30	50	70	90
Target y distance (cm)					

Table 13: List of testing instructions

An error of around ±0.1cm and ±0.5 degrees was still present due to inaccuracies of the optical flow sensor in detecting small distances. This was also within the threshold region range of the PID output. Testing also yielded the following observations.

1. Limited detectable distance by optical flow sensor of around ±206 cm due to overflow.
2. Rounding errors between actual x distance (`total_x`) and angle (Refer to Section 2.3.2.3).

3. Rounding errors from counts per inch to centimetres resulted in inaccurate conversions between `total_x1` and `total_x`.
4. Slight time lag in sensor reporting distance travelled by rover.

2.3.4 Evaluation of Subsystem

2.3.4.1 Evaluation of testing results

From the observations in Section 2.3.3.2, sources of error were detected, and solutions were implemented. (See Table 14) The functional requirements were all met once these errors were corrected, although more vigorous testing of sample instructions could have been carried out.

Errors	Solutions
16-bit int declarations of intermediate variable counter (`total_x1`)	Replaced `total_x1` with a 32-bit float to increase detectable distance range.
16-bit int declarations of `total_x`	Replaced `total_x` with a 32-bit float to increase resolution.
400 counts per inch incorrectly listed as 157 counts per cm and instead of 157.48	Data was processed with at least 2 decimal points before rounding off to the nearest degree or centimetre to be sent to Control. A higher accuracy was not required from the rover's movement perspective in consideration of its physical dimensions.
Optical sensor hardware constraints, `delay()` code segments and serial prints	The `delay()` code segments and serial prints used for debugging were removed and the default rover speed was set as slow for ample sensor reaction time (See Section 2.3.2.2).

Table 14: Sources of error and proposed solutions

2.3.4.2 Future Extensions

Future extensions include remote control of the lens of the optical flow sensor from Earth for accurate distance measurement. This is especially so when how well the lens is focused depends on the surface, texture, and light. The optimum lens height may differ slightly especially so if the arena is not flat, which is more representative of Mars.

Alternatively, rotary encoders could be employed as an additional source of odometry sensing. Rotary encoders are typically capable of higher precision than optical flow sensors, and thus might be able to better estimate the position of the rover (Chao-Yang Lu & Shao-Kang Hung, 2019). However, rotary encoders are not robust to wheel-spin, which may be common in the rough terrain of Mars. Hence, a combination of optical flow and rotary encoder sensors could be used for precise and robust tracking of robot motion, with values fused with a Kalman Filter (Zarchan & Musoff, 2015).

Adaptive tuning of the tuning parameters can be considered. A more conservative set of tuning parameters can be adopted when the rover is near the position setpoint while more aggressive tuning parameters can be used when it is further away for a more robust position control setup.

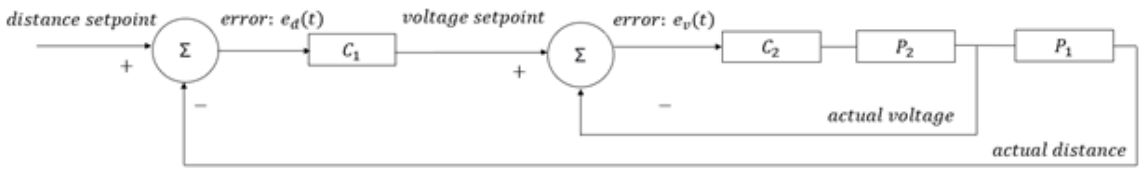


Figure 20: Cascaded control

Another possible extension is the use of cascaded controllers. The inner loop represents the closed-loop buck SMPS, while the outer loop links the position to be controlled with the rover's speed. The input of the first PID controller is the error in distance while its output is the voltage setpoint, determining the speed of the rover. Ideally, when the error is distance is decreases to 0, the corresponding voltage setpoint also decreases to 0 to stop the rover.

2.4 Command Subsystem

2.4.1 Design Considerations

2.4.1.1 Type of Map

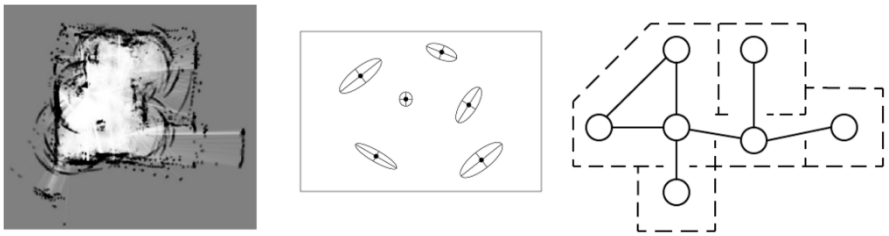


Figure 21: (Starting from the left) Occupancy grid map, feature-based map, topological map

Three types of maps largely used are occupancy grid maps, feature-based maps, and topological maps (Yu Shuien, et al, 2020), shown in Figure 21. Feature-based maps rely on the main features of the environment, like buildings and walls, to build up the map. While this removes redundant data between the larger features of the environment, smaller data points or obstacles could easily be overlooked, resulting in collisions. Topological maps represent the environment in terms of the connectivity between places, similar to a subway train map, but whilst the relationship between points is maintained, scale and distance may vary. Hence, an occupancy grid map was selected, as it maps all points seen in the environment with respect to their actual distance and position, which is essential for navigation tasks, such as path planning with obstacle deviation and position estimation.

As Vision is monocular, only two-dimensional readings are available. Hence, a two-dimensional occupancy grid map was used to represent the information received from Vision and Drive. (André M.Santana, et al, 2011)

2.4.1.2 Pathfinding Algorithm

A* is a heuristic algorithm that makes use of informed estimations of the shortest path to ignore irrelevant nodes. Conversely, Dijkstra's algorithm traverses through all neighbouring nodes, consuming unnecessary memory and processing power. As the map gets larger, A* pathfinding algorithm will have better performance than other conventional algorithms like Dijkstra's algorithm, as it has better time complexity. Hence, A* algorithm was used in generating the path for rover to navigate towards objects of interest.

2.4.1.3 Communication Protocols

The web app is a back-end web server that stores and processes data, and a front-end web browser that renders the web display that the user interacts with. For the display of the frequently updating map, WebSockets are used instead of the traditional HTTP requests used on other static pages of the web page. This allows the map data to be updated only when there is a modification, where it sends a message to the WebSocket channel. Plotly's Dash python analytics application framework (Plotly, 2021) was initially used with its live update functionality to update the map data over a pre-defined interval. However, WebSocket was deemed to be a better choice since it updates the map only when necessary, instead of updating it at fixed intervals which wastes bandwidth and consumes extra overhead.

Communication with Control to receive information about objects and obstacles in Phase 1 and updating of the rover's position in Phase 2 was done through WebSocket messages. WebSocket communication was tested to be much faster, as mentioned in Section 2.1.3.1.3, hence it was used.

2.4.1.4 Background Tasks

Most of the computation is run on a databased-backed work queue, instead of executing under the main script process when loading the web page.

This allows the user to navigate the webpage, whilst calculations occur, rather than having to wait for calculations to be completed. Since some calculations like path finding algorithm can take some time, this allows for a more user-friendly interface.

2.4.2 Implementation of Subsystem

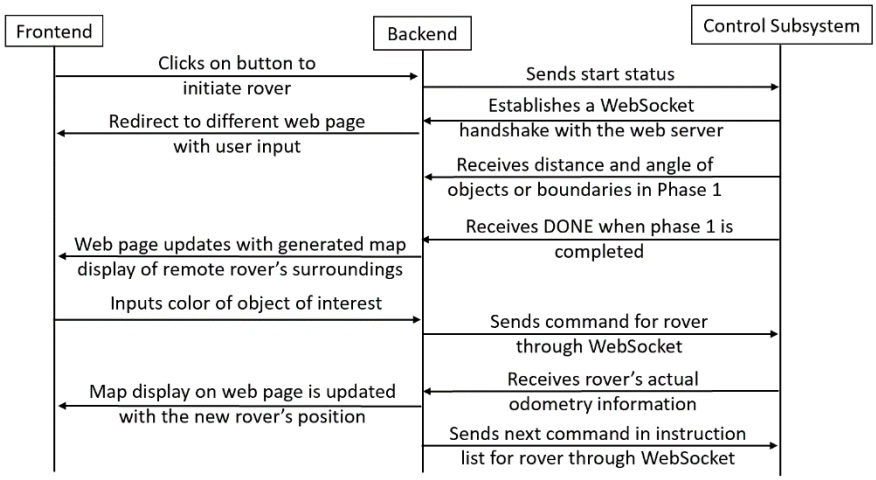


Figure 22: Overall data flow among front-end, back-end server and Control

For a more detailed view of data flow within Command, refer to Appendix 9.

2.4.2.1 Web Framework

Django (Andrew Mccarthy, 2019) is a Python web framework that was used to build the web application. The Channels and Celery libraries were used to support the Django framework. An additional backend server, Redis (Salvatore Sanfillipo, 2021), was also included for datastore and to act as a message broker for both Channels and Celery processes.

2.4.2.1.1 Channels with Django

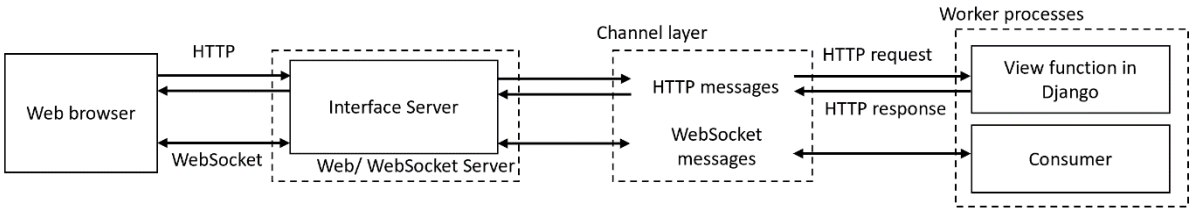


Figure 23: Django Channels data flow and processes

By default, Django operates in request-response mode, to cater to HTTP messages widely used in web browsers. With Channels (Andrew Godwin & Carlton Gibson, 2020), Django operates instead in worker mode, which enables the server to listen on all channels (seen from the channel layer in Figure 23) and execute the corresponding consumer function when a message is detected on the channel (seen from the worker processes in Figure 23). This makes the communication event-based rather than request-response, making it apt for frequent updates between server and client.

Channels allows the web server to handle both traditional HTTP requests and WebSocket messages, where the WebSocket communication helps to decrease bandwidth and unnecessary overhead as previously explained in Section 2.1.3.1.3.

2.4.2.1.2 Celery with Django

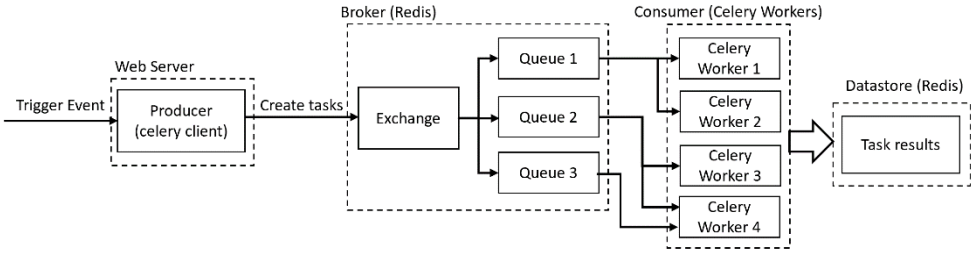


Figure 24: Celery Django data flow when background tasks are triggered

A task or job queue, Celery, (Django Software Foundation, 2019) is used in conjunction with the Django server to queue and execute tasks in the background. As seen from Figure 24, when the background task is triggered, the celery client sends a message to the broker, which is the message server. The broker creates an exchange that routes the messages to the corresponding queues. The broker then routes these messages to the consumer, celery workers, to process the tasks and store results in the Redis datastore. This process is offloaded from the web server, allowing tasks to run in the background, for processes like updating the map and generating the shortest path for the rover's navigation, as highlighted in Section 2.4.1.4.

2.4.2.2 Mapping of the Rover's Surroundings

2.4.2.2.1 Occupancy Grid Mapping

The server uses two-dimensional Occupancy Grid Mapping to generate a discretised map of the rover's surroundings. A 2D matrix holds the occupancy value of the corresponding index of the grid map. The initial map was filled with an occupancy value of 1.0 for all cells, representing unknown map regions, which were coloured on the map in light grey as seen in Figure 25 below.

From Phase 1, Control returns the distance and angle of obstacles or objects. This distance received would be from the camera to the object, where the camera is mounted at the front of the rover. To account for the change in position of the front of rover as it undergoes rotation, the distance from front of the rover to its invariant point is added to the distance received. The angles received are from the rover's bearings, which are converted to degrees from the horizontal x-axis on the server.

These values are processed to get the corresponding coordinates of the points using trigonometry.

The coordinates were converted into grid indexes considering the resolution of a single grid square and factoring in a bias in the x and y coordinates to make grid indexes positive, in order for the grid indexes to be indexable by the matrix.

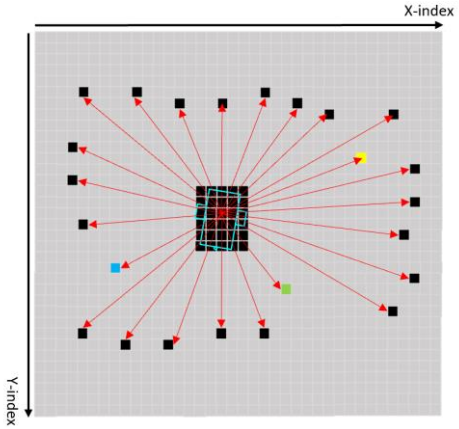


Figure 25: Example of an initial 2D occupancy grid map with obstacle points set in black

These points were coloured black (Figure 25), representing known boundary points.

2.4.2.2.2 Bresenham's Algorithm

The known boundary points on the grid map would not be continuous since the obstacle or object distance were only measured at 10° intervals in Phase 1. Bresenham's algorithm (Kwang-Il Kim, Jung Sik Jeong & Gyei-Kark Park, 2014) was hence used to extrapolate a boundary line between adjacent points.

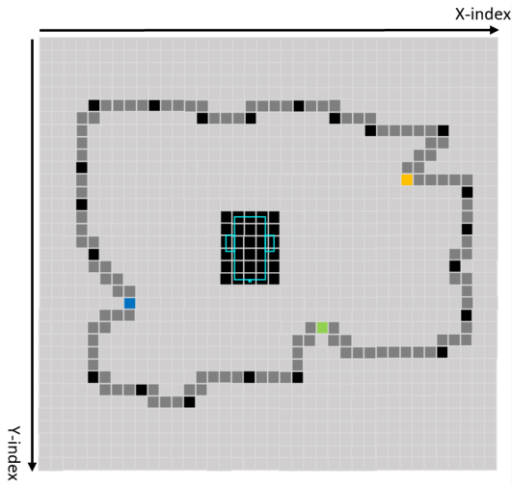


Figure 26: Occupancy Grid Map after boundary extrapolation using Bresenham's algorithm)

These points are coloured dark grey (Figure 26) since it is likely they form a boundary but are not confirmed to be actually occupied.

2.4.2.2.3 A Pathfinding Algorithm*

The A* pathfinding algorithm estimates the shortest path from a start node to a target node, while avoiding obstacles. The algorithm makes intelligent estimates based on heuristics, drastically decreasing the number of nodes traversed through, increasing its efficiency.

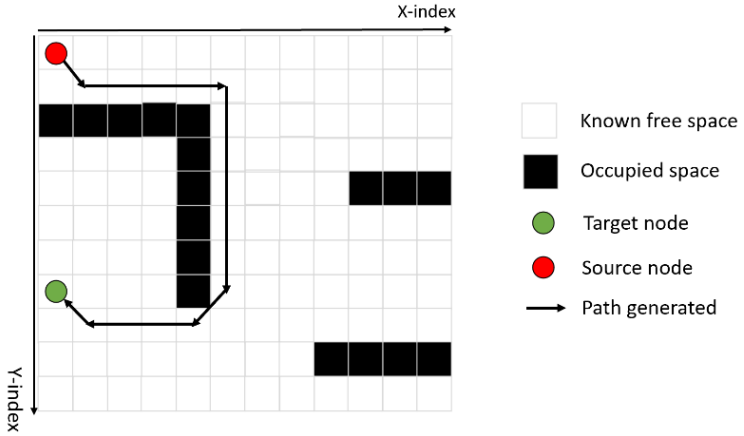


Figure 27: Example of a path generated by the A* algorithm

Starting from the source node, the algorithm traverses its neighbouring 8 nodes, and selects the node with the lowest cost as the next node. The cost of a node is $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of moving from the starting point to the node, and $h(n)$ is the estimated cost of moving from the node to the destination.

$g(n)$ is simply the distance travelled so far. However, calculating $h(n)$ is more time-consuming and is usually approximated using heuristics. In the web server, the Euclidean Distance Heuristic was used, which computes distances between current cell and goal cell using Pythagoras Theorem.

This was chosen over Manhattan and Diagonal distance Heuristics. The Manhattan distance is the sum of absolute differences in source and target nodes' x and y coordinates which is more relevant for movement that is restricted only in four directions (top, bottom, left, right). The Diagonal distance is the maximum of the absolute differences between source and target nodes' x and y coordinates, which is more relevant when rover's movement is restricted in 8 directions (neighbouring 8 grid cells). Since the rover is unrestricted by the grid, the Euclidean Distance Heuristic was implemented.

Dynamic programming was used to optimise the algorithm. The nodes visited were marked with the cost and stored such that in further iterations of selecting neighbours of nodes in path, the same nodes are not chosen or traversed through again.

2.4.2.2.4 Extensions to the A Pathfinding Algorithm*

The conventional A* pathfinding algorithm tracks path and does obstacle avoidance based on a single grid cell. To adapt this algorithm into the project, the pathfinding algorithm was modified to consider the entire region occupied by the rover, along with some extra space to account for errors in the rover's odometry information.

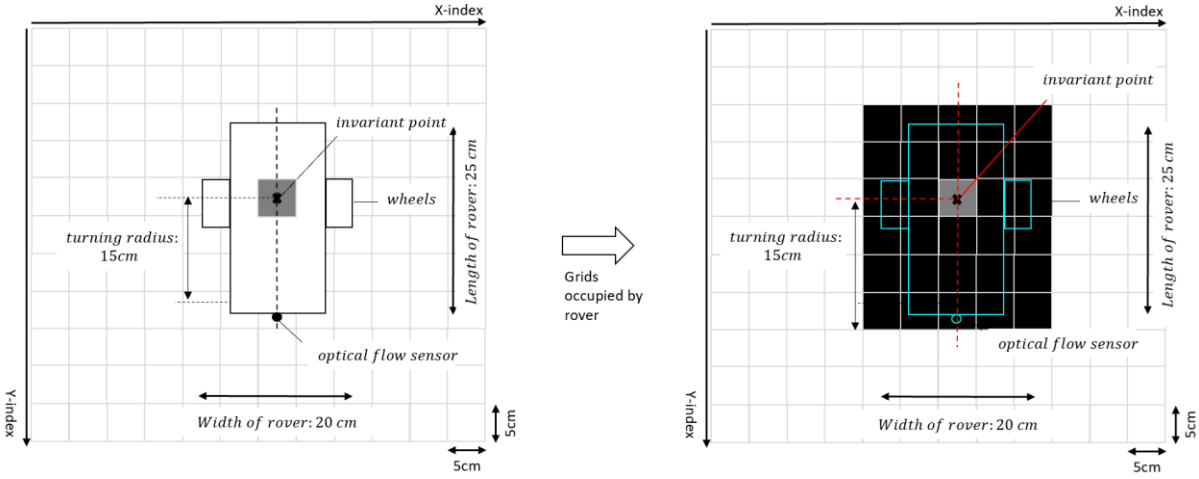


Figure 28: (left) Rover's outline on the grid map; (right) Grid cells marked occupied by the rover

The grids occupied by the rover when at its starting position is shown in Figure 28.

As the rover rotates, its position is mapped by maintaining its invariant point on the same cell. The grid cells occupied by the rover is obtained by first calculating the index of the four corners of the rover. Bresenham's algorithm is used to find the indexes of the grids corresponding to the rover's boundary. The indexes between minimum and maximum indexes of each row or column are then calculated correspondingly.

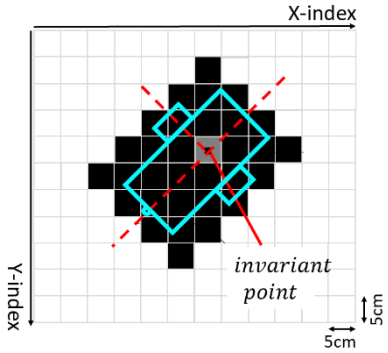


Figure 29: Example of grid cells occupied by rover as its direction changes to a bearing of 45°

2.4.2.2.5 Updating the Map with actual Odometry information from the Rover

After the path to target object is generated, the rover' movement on the map display will mimic the first instruction sent to Control. The actual rover's movement is received from Control after rover finishes executing the instruction. In the case of a discrepancy between rover's actual position and calculated position, the position of the rover on the map would be updated accordingly, and the A* path finding algorithm would be executed once again from the new position of the rover.

2.4.2.3 User Interface

Figure 30 below shows the home page view of the web page, where the user starts up the rover by clicking on "Initiate Rover". Other tabs on the home screen can also be toggled to find out more about the team, and project brief.

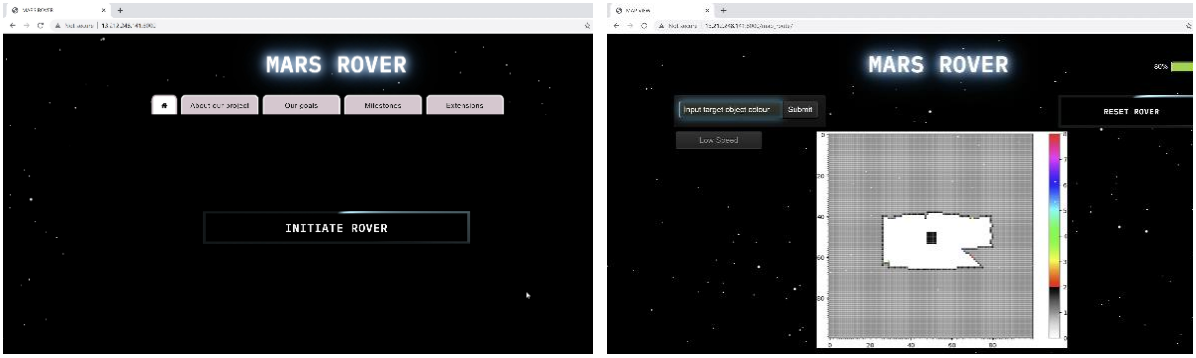


Figure 30: (left) home page with button for user to initiate rover; (right) map display and user input for color of target object, speed and reset of rover

As the user clicks on "Initiate Rover", they will be redirected to another webpage where the map would appear after Phase 1 is completed. The user can input the colour of object of interest in the text box provided and select the speed of the rover from the dropdown speed menu, for speed control as explained in Section 2.3.2.2.

2.4.2.4 Database

The web server uses SQlite (Andrew Mccarthy, 2019) as a database management system to store information about the current odometry of the rover, map environment and instructions to be sent to rover.

This allows the web server to keep track of the rover's position and trajectory in relation to the map.

2.4.3 Testing of Subsystem

2.4.3.1 Map display and A* Pathfinding Algorithm

Different map conditions were simulated to test out the map algorithm's ability to calculate the path to the target object, while still avoiding obstacles.

A map with regular-shaped boundary was tested initially, to ensure that basic map finding algorithms were computed accurately. More convoluted boundaries were then used to test the algorithm's ability to avoid collisions with these boundaries, where the results of the A* algorithm was rendered on the map. Screenshots and details of these tests can be found in Appendix 10.

2.4.3.2 Command Instructions

Next, the conversion of this path into command instructions sent to Control was checked by entering the colour of the target object into the web browser and accessing the database storage of instructions generated to manually confirm the accuracy of the conversion.

2.4.3.3 Rendering Map Display that Updates Automatically

Testing of the map display that updates the rover's position on the map with the estimated and actual rover's odometry data was first tested locally using the matplotlib window. As the code was adapted onto the web server, the web display was checked for appropriate updates of the map display.

2.4.3.4 User Interaction with Web Browser

Clicking buttons and tabs on the page intended for user activity were done to confirm the functionality of the web server to accurately listen and respond to events that happened on the web browser.

2.4.4 Evaluation of Subsystem

The update of the map was found to be rather slow and inefficient using the initial method of using Plotly's dash framework to update the map display as a fixed interval, as explained in Section 2.4.1.3. The server was using up additional bandwidth that caused the connection to the remote AWS server to crash or lag rather frequently, even when map data has not been modified, due to the re-rendering of webpage and map at the pre-specified interval.

Hence, the update of the map was changed to using WebSocket, to send messages on the channel only when map data has been modified. The webpage will only update itself when these messages are received on the channel, reducing consumption of redundant resources.

2.4.4.1 Future Extensions

Simultaneous Localisation and Mapping (SLAM) can be explored in the mapping methodology, to update the map data with new environmental data points simultaneously as the rover moves around the arena.

2.5 Energy Subsystem

2.5.1 Design Considerations

The high-level purpose of the Energy subsystem is to charge a battery using solar panels, which could then be used to drive the motors and ideally power external hardware circuitry of the rover.

The Energy subsystem is designed as a mobile charging station which can be broken down into two main modes: charging and discharging. Figure 31 shows a high-level flowchart of the operation of the energy module:

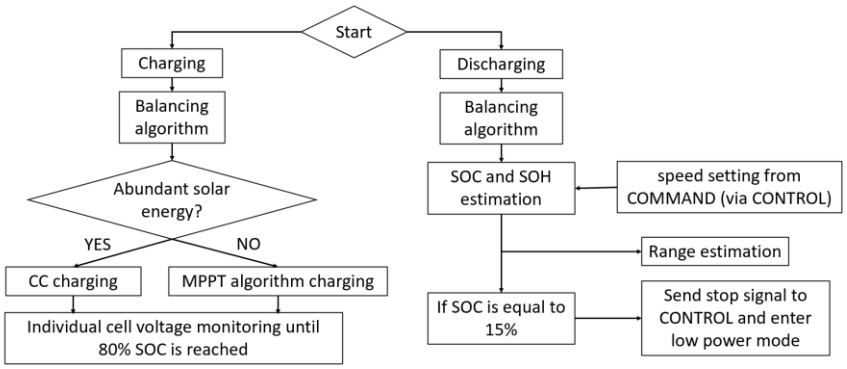


Figure 31: Energy subsystem flowchart

2.5.2 Implementation and Testing of Subsystem

2.5.2.1 PV Panels and Cells Arrangement

There are 8 different ways to configure the solar panels and cells; the opted configuration is 4 solar panels connected in parallel at port A of the SMPS, with 3 cells connected in parallel at port B. In this setup, the SMPS acts as a Buck circuit. According to the datasheet for a single PV panel, under optimal conditions, the specified voltage is 5V and the specified current is 230mA. If the panels are connected in parallel at port A, the maximum input voltage is 5V, which obeys the constraint on the voltage at port A to be no more than 8V due to the V_{GS} rating of the MOSFET. A more exhaustive explanation is provided in Appendix 11.

Cells connected in series increase the total nominal voltage, since the nominal voltage of each cell is summed. This configuration would be useful when powering electronics that require larger voltages. The drawback is that it does not affect the overall capacity of the formed battery; it just controls the

amount of power it can output at a time. On the other hand, by connecting the cells in parallel, the voltage across the batteries is the same as the voltage across a single cell with higher total battery capacity, allowing for greater running time. This would be useful for power electronics that use lower voltages but are used for extended periods of time. Energy prioritises greater capacity over greater voltage, hence, the batteries were setup in parallel to increase the running time of the rover.

2.5.2.2 Solar Panel Characterisation

In order to characterise the solar panels, a single panel was connected to port B and a 120Ω resistor was connected to port A such that the SMPS operates as a boost. The lamp's light was concentrated on a single panel and a duty cycle sweep was done, in which the panel voltage and the current from the panel was recorded to enable solar cell characterisation, seen in Figure 32.

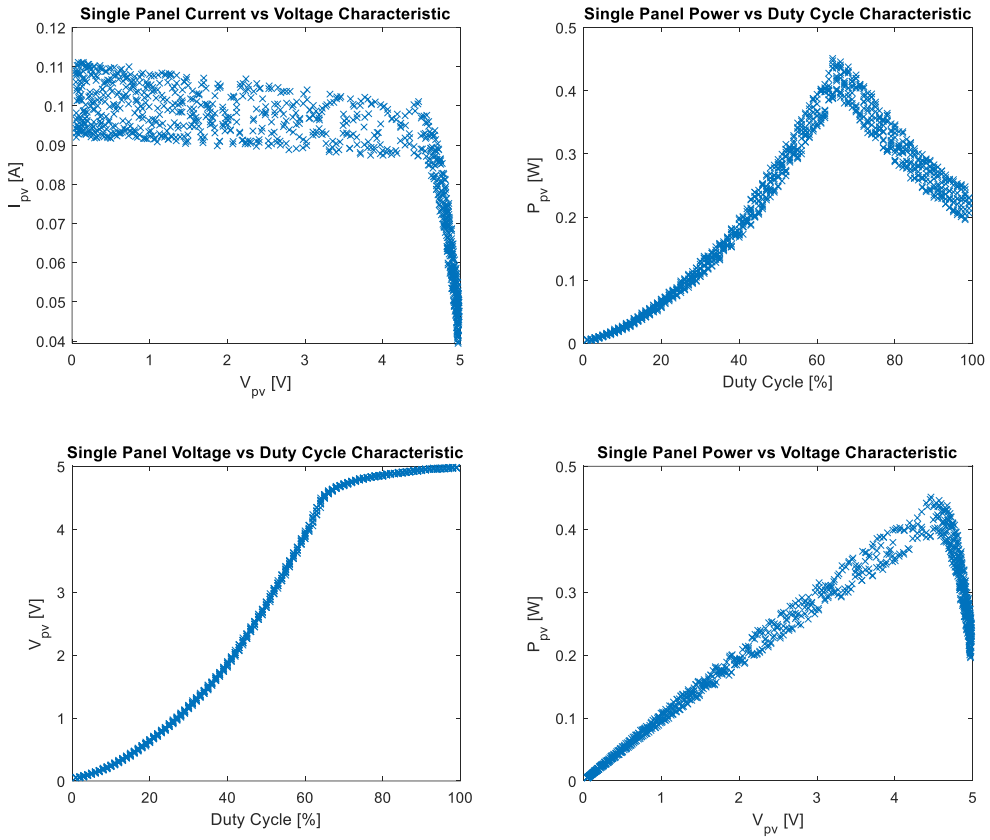


Figure 32: Single PV Panel Characterisation

From the IV characteristic curve, the panel currents were relatively constant within a given margin, from 0V to approximately 4.5V, and drops steeply after that voltage. The short circuit current for this irradiance was approximately 100mA and the open circuit voltage is 5V as expected from the rating of the panel. The PV characteristic curve tells us the maximum power point was approximately around 4.5V and came to about 0.4W for this irradiance. Finally, the duty cycle giving the maximum power point was in the range of 60-70%. Since the ideal configuration used to charge the cells using the PV panels was determined to be 4 parallel panels at port A, with 3 parallel cells at port B, the circuitry would be a buck. Assuming continuous conduction mode, the output voltage is approximately the input voltage multiplied by the duty cycle. Under the circumstance when the MPPT algorithm would be enforced, the MPPT duty cycle and MPPT voltage can be approximated as 65% duty cycle and 4.5V panel voltage. Thus, the output voltage of 2.9V lies between the maximum (3.6V) and minimum (2.5V) voltage of the cells configured in parallel. This MPPT voltage would provide the cells with the most power from the PV panels and thus the charging current would be rather large at this charging voltage.

2.5.2.3 Battery Characterisation and State of Charge (SOC) Estimation

As mentioned in Section 2.3.2.2, the rover is designed to have three different speed settings with corresponding discharge rate which can be used to estimate the SOC. For this report, only low speed setting will be considered since this is the default value used when utilizing the rover. Using a current clamp meter, the current measured whilst moving and being idle at low speed averaged to 330mA (this includes current to the FPGA which powers the ESP32). Through testing, it was discovered that the current drawn into each of the batteries during charging were equal. Because the batteries were setup in parallel, the discharge current for each cell would therefore be 110mA. Figure 33 shows how the battery voltage varies with time for a constant charge and discharge rate, and it also shows how the battery voltage varies with state of charge for a discharge rate of 330mA and a charge rate of 750mA.

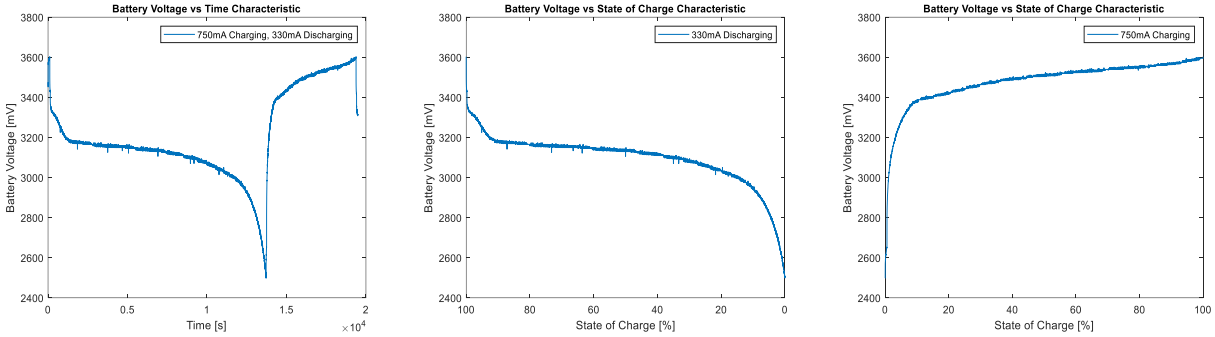


Figure 33: Battery Charging and Discharging Characteristic

The SOC algorithm is stored as function in the Arduino. It employs a voltage lookup table method for determining the SOC, which is attained from the discharge curve obtained from the single cell characterisation for a discharge current of 110mA. From the discharge curve, the voltage range is quantised in such a way that the maximum voltage corresponds to 100% and the minimum voltage corresponds to 0% SOC. Using uniform quantisation, the voltage values from the SOC curve corresponding to multiples of 5% up to 100% SOC and are stored in an array. The function iterates through all the stored voltages from and returns the SOC that corresponds to a particular voltage that is closest to the measured battery voltage. This individual cell SOC is averaged and sent to the ESP32 every second using the UART ports if the energy sub-system was mounted on the rover. This process would be repeated with the discharge rate at medium and high speed such that a 2D lookup table is formed, meaning the SOC would be a function of discharge current and voltage.

2.5.2.4 Battery Balancing Algorithm

The hardware with the battery board allows us to do passive balancing. The purpose of doing balancing is to improve the State of Health (SOH) of the cells. If a cell is weaker than the other cells in the parallel configuration, it would have a lower capacity compared to the other cells and would therefore charge and discharge before the others. When this happens, the SOH of the cell deteriorates, resulting in its capacity diminishing much faster due to over-charging/discharging.

The battery balancing algorithm uses a conditional statement, exploring all possible alterations of the SOC of the three cells. Based on which condition is satisfied, passive balancing is done by switching the digital discharge pins on and off. This algorithm has been placed in the state machine for charging and discharging to maintain the SOC of all of the batteries. Appendix 13 shows the pin connections between the SMPS shield and the battery boards as well as the ESP32 used for communication.

2.5.2.5 Constant Current (CC) Charging and MPPT Algorithm

The charging profiling was compromised by the hardware constraint since the duty cycle of the MOSFET is the only variable that can be controlled. For constant current charging, the current is set by directly hardcoding the reference current in the dual loop PID current and voltage controller whilst the MPPT algorithm requires a duty cycle sweep to obtain IV curves. This was a dilemma when deciding the charging strategy as both methods would require the control of the duty cycle at the same time. However, this problem was solved by implementing both methods but utilizing only one depending on the solar panel input.

The power incident on the solar panels may be large due to abundant solar energy. Hence, it is important to limit the output current from the SMPS such that the maximum charging current into a single cell is 500mA to prevent overcurrent, which is the maximum charging current of one cell according to the datasheet. Because there are 3 cells in parallel, the maximum permissible current at the output of the SMPS is 1500mA. When the current measured by the current sensor is above that threshold, a set of instructions are executed to vary the pwm value (duty cycle) such as to limit the output current to 500mA. In contrast, when there is very little solar energy, the power incident on the solar panels would be scarce. In this case, the MPPT algorithm must be enforced. In the algorithm, the pwm value is varied such as to maximise the power outputted from the SMPS and thus used to charge the cells (Mathworks, 2021). This will enable efficient charging when there is little solar irradiance. By having the MPPT and CC algorithm in the conditional statement, it ensures that the right pwm value is chosen for the respective condition to charge the cells, abiding by safety parameters.

2.5.2.6 Range Estimation Algorithm

The range estimation algorithm takes three parameters as inputs: SOC1, SOC2 and SOC3. Firstly, the average SOC is determined and then the present capacity is determined by multiplying the total capacity of all the cells by the average SOC to get a weighted average. Finally, the remaining range is determined by dividing the present capacity by the discharge current of all 3 cells and then multiplying by the speed in cm/s and by 3600 seconds, which corresponds to an hour to get the remaining range in cm. This value of range would be communicated to the ESP32 every second through the UART ports if it were possible to integrate the energy sub-system to the final working rover.

```
float Range_est(float SOC1, float SOC2, float SOC3, int rover_speed, float SOH) {
    average_SOC = (SOC1+SOC2+SOC3)/3;
    present_capacity = (average_SOC/100)*1700*SOH; // 1700mAh refers to the beginning of life capacity of the three cells in parallel in the battery
    low_battery_capacity = (15/100)*1700*SOH;

    if (rover_speed==low_speed){
      range = ((present_capacity-low_battery_capacity)/discharge_rate)*3600*low_speed;
    }else if (rover_speed==medium_speed){
      range = ((present_capacity-low_battery_capacity)/discharge_rate)*3600*medium_speed;
    }else if (rover_speed==high_speed){
      range = ((present_capacity-low_battery_capacity)/discharge_rate)*3600*high_speed;
    }
    return range;
}
```

Figure 34: Range Estimation Algorithm

2.5.3 Evaluation of Subsystem

2.5.3.1 Charging and Discharging Profiling

Due to unforeseeable circumstances, the code for charging and discharging was not tested with the cells.

The charging phase consists of a conditional statement which chooses between CC charging and MPPT charging based on solar energy abundance. Meanwhile, battery balancing is done to regulate the SOC of all the cells, whilst communicating SOC information simultaneously.

The discharging phase consists of a uniform discharge rate whilst balancing to maintain constant cell SOC. Based on the speed mode indicated by Control, a particular constant discharge rate is chosen and the relevant range and SOC information is regularly communicated to Control. When the SOC falls

to 15%, an interrupt signal is sent to Control, which triggers all the relevant protocols to enter 'Low Power Mode'.

2.5.3.2 Future Extensions

Considering the subsystem was omitted in the final integration test, an obvious extension would be to integrate the energy module with the others to ensure the communication and functions operate as expected.

Improvements could be made on the SOC algorithm that solely relies on the 2D voltage lookup table meaning if the current drawn is not relatively close to currents manually measured for the speed settings, the SOC function will return an inaccurate percentage. Coulomb counting is a more accurate way to continuously monitor SOC, it still relies on a lookup table to measure the initial SOC, however if the initial SOC was forced to be a known value, for instance only after a full charge the rover were allowed to move, than the SOC would be independent of the discharge rate, but more importance would be placed on having accurate capacity of a cell. Also, if possible current sensing resistors should be placed on every battery to ensure accurate current measurements when integrating the current to estimate the charge over a period of time.

3. Testing and Evaluation of Rover

3.1 Testing Setup

To verify that each subsystem integrated successfully, a test arena was devised. The boundaries were defined with black cardboard walls around 50cm tall to remove the possibility of unrelated background elements affecting the rover. See Appendix 12 for photo of the arena.

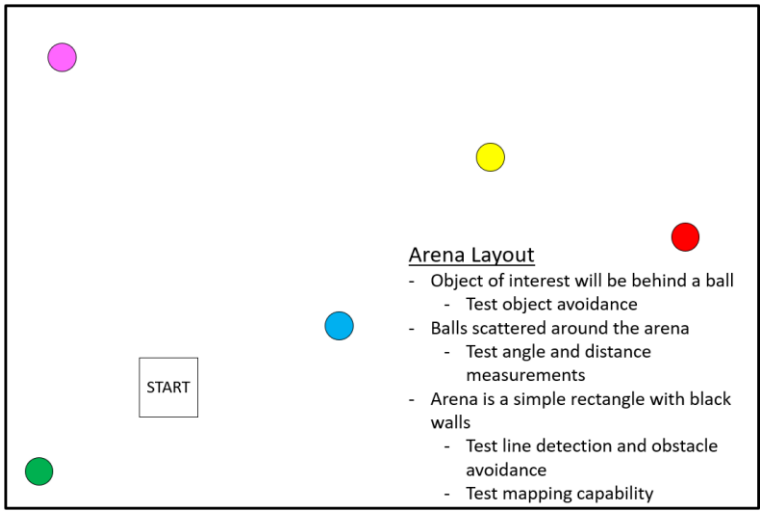


Figure 35: Test Arena Layout

Testing between subsystems was carried out methodically as per the flowchart below. At each point, bugs were fixed before moving on to the next phase of testing.

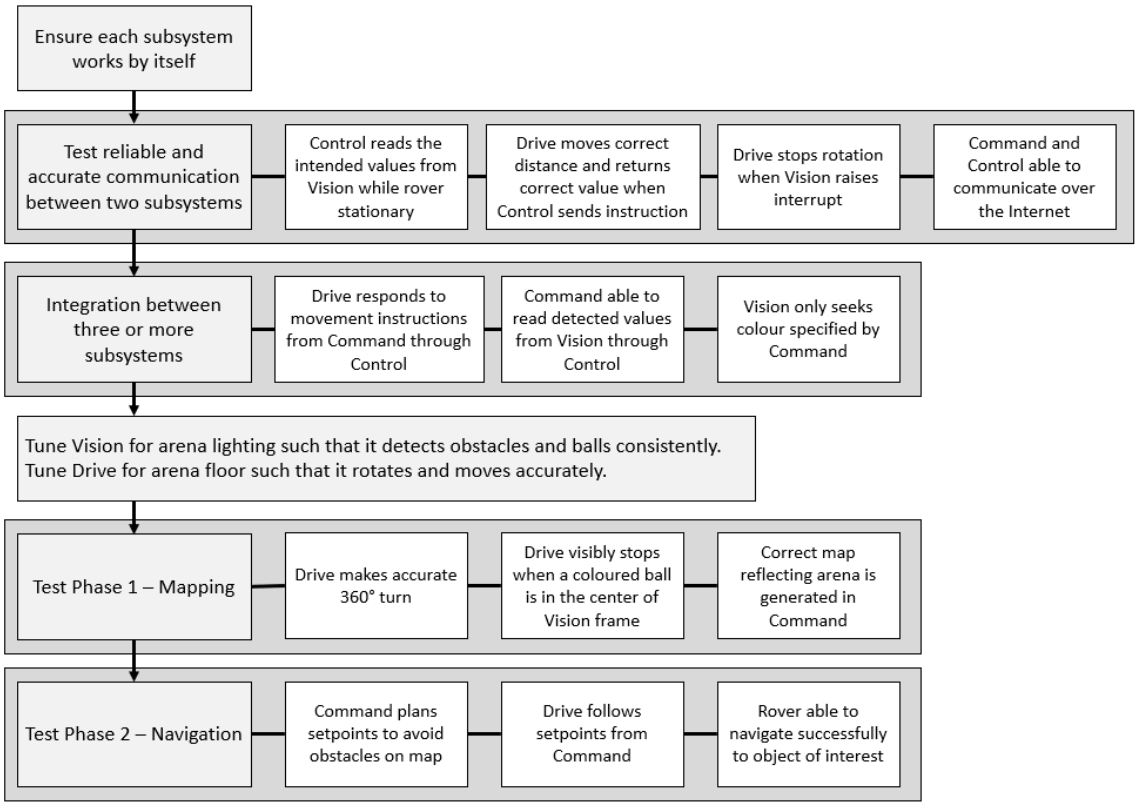


Figure 36: Integration Testing Flow

For testing reproducibility, between test runs on separate days, the layout of the arena and starting position of rover were fixed. In addition, an "idealised" map was generated artificially in Command so as to verify Phase 2 without factoring the error introduced from Phase 1.

To test low-power mode, the code for Control was modified to take in a dummy value from Energy. The rover was then verified to have taken the actions detailed in Section 2.1.1.

3.2 Evaluation of Testing Methodology

The test arena was an effective tool in the development of the rover. The ability to perform repeated runs on approximately the same arena and compare results was a crucial factor in debugging and adding features to the rover.

However, there were several limitations to the setup of the arena. Firstly, the lighting of the arena was unable to be held constant. This led to unexpected results from Vision, such as inaccurate bounding boxes, between testing runs, which hampered progress.

Secondly, the requirement to attach cables to Vision and the Drive SMPS affected the motion of the rover, as the provided motors were very weak. It was found that a small tug on the cables connected to the rover could hamper its motion. A group member had to follow the rover around the arena, holding up the cables behind the rover. This introduced additional adverse lighting conditions to the arena, which affected the reproducibility of results from Vision. Furthermore, utmost care was taken to prevent wires from moving under the Optical Flow sensor to ensure accurate distance tracking.

3.3 Evaluation of Rover

Firstly, tests of the communication between Control and each subsystem were done. Communication with Drive was accurate and reliable. Drive could receive the correct instructions from Control and executed the instructions correctly. Control could also receive odometry data from Drive 100% of the time. Communication with Vision was accurate and reliable once error handling was in place. The data received by Control were the same data that Vision had in its buffer, but occasionally 0x00 was received due to the FPGA not finished writing fully to the transmit buffer before reads by the ESP32. Hence, error handling was used to re-read data from Vision to receive the accurate values.

HTTP and WebSocket was tested as the communication protocol with Command. The information sent and received were always accurate for both protocols, but communication using HTTP took extremely long due to frequent HTTP timeouts. Although frequent disconnections from the web server was also observed with WebSocket, error handling was added to reconnect with the server which happened in under 5 seconds. The speed of WebSocket was obvious when testing with the rover, as Phase 1 took 10-11 minutes to complete when HTTP was used, while WebSocket took 2.5 minutes to complete.

Tests with 3 subsystems together showed that all the subsystems were working as intended.

Overall, it was found the rover could meet the Functional Requirements set out in Section 1.1, while decisions made had the Non-Functional Requirements in mind. The rover could autonomously generate a map of its surroundings which was then stored on an external database without human input. It could respond to commands to navigate to a destination, avoiding obstacles along the way, despite external disturbances. Lastly, it could respond to its battery level, acting to conserve power and preserve battery lifespan.

However, it was found that the rover was not robust to external changes. Lighting changes affected its ability to detect objects greatly and tuning of Vision parameters needed to be done frequently. In addition, the lack of resolution in line detection from Vision and from position control in Drive hampered its ability to resolve an accurate map of the surroundings. Lastly, changes to the floor

material affected its turning ability, with separate PID parameters for Drive necessary for different testing locations. These could be improved in future iterations for the rover to be more effective in a harsh and uncertain environment.

4. Intellectual Property Write-up

As we progress in our education, drawing closer to employment with its responsibilities, the talk on Intellectual Property was eye-opening. While avoiding plagiarism is important to students, avoiding copyright infringement is equally crucial, due to the potentially ruinous legal and financial implications.

A peek into the process of securing a patent was intriguing, as the process of safeguarding and capitalizing upon our work was a side of our occupation that we had never been exposed to before. While byzantine, it is necessary to ensure fair competition and innovation. The aspect of "novelty" in patents was especially interesting, as it implied a certain degree of confidence from the inventor, for a patent to be filed and the necessary administrative drudgery done – giving us a deeper respect for the boldness of every successful inventor.

In our project, we have created things that can be considered our intellectual property, such as source code and even this report itself. As the authors/designers of these works, we could potentially claim copyright if others were to plagiarize it or use it for their own private gain. However, as these works do not meet the "Novelty" (and arguably "Inventiveness") requirements for a patent, we should work and think harder in the future for successful patent applications.

However, we acknowledge that our project stands on the shoulders of past research and development. We acknowledge the public-domain code that was provided to us, such as the Vision demo code from Terasic, as well as the demo code for Drive, Energy and Vision. Open-source tools, such as Arduino libraries, have been very helpful. While we would love to submit our project with a GPL license (Free Software Foundation, 2021)so that others may reuse our code freely, without explicit permission from the authors of the provided demo code, we will refrain from doing so.

5. Project Management

Each team member's top 2 Belbin roles (Meredith, 2021) were obtained and compared. Their strengths and weaknesses were highlighted and discussed, to capitalise on each member's capabilities. Decisions made were agreed upon collectively after weighing the pros and cons, ensuring diplomatic decision-making.

Next, for effective project planning, the project was split into 3 main phases. Phase 1 was focused on doing research and identifying functional requirements for each subsystem and the overall project. Phase 2 was the design implementation, testing and evaluation of each subsystem. In Phase 3, integration of the rover was carried out with further optimisation. Team members frequently updated a Gantt chart (attached separately) to reinforce structured and efficient progress among the team.

Lastly, a Telegram chat group was set up to facilitate efficient communication practices. Github repository was used for version control. Microsoft Teams meetings were held periodically, where the progress since the last meeting would be reviewed and work to be carried out by the next meeting would be allocated. Meeting minutes were also taken down (appended in the same Gantt chart document) for everyone to be clear of their tasks.

6. Conclusion

6.1 Summary

This implementation of the Mars Rover has met the essential functional requirements defined in the original specification. It can operate autonomously in a remote location, without direct supervision from someone onsite. Its processing unit can receive commands and send status data so that a map of the local working area can be made and used to navigate the local area, avoiding obstacles, towards areas of interest, indicated by Command.

6.2 Future Extensions

For more effective surveillance of the surroundings, the capability to transmit pictures or even video captured by Vision to Command would be useful. To achieve this, compression algorithms such as Lempel-Ziv-Welch (R.H. Greenfield & W. Kinsner, 2006) or the Discrete Cosine Transform DISC (Dave Marshall, 2001) could be implemented on the FPGA to reduce the file size of the images, saving bandwidth, and reducing the probability of errors in transmission.

Given that the ESP32 has many remaining GPIO ports unused, more hardware such as sensors which measure humidity or pressure could be attached to the rover, giving a more comprehensive understanding of Mars.

As mentioned in Section 2.3.1.2, only a 2-D map can be generated as only one camera is used. To better characterize depth information, future designs can utilize two cameras for stereo vision (Miran Gosta & M. Grgic, 2010). This allows the rover to return more detailed information about its surroundings, highlighting potential areas of interest for study and avoiding hazards.

Lastly, the rover could utilize simultaneous localization and mapping. By detecting and avoiding obstacles simultaneously in real-time, Phase 1 and 2 (Section 2.1.1) can be merged, improving exploration efficiency. The time lag between these 2 phases can also be eliminated, leading to greater efficiency. With more data about its surroundings sent to Command, the rover can move more efficiently since instructions sent from the path finding algorithm will be more accurate.

7. References

André M.Santana, Kelson R.T.Aires, Rodrigo M.S.Veras & Adelardo A.D.Medeiros. (2011) *An Approach for 2D Visual Occupancy Grid Map Using Monocular Vision .* Available from: https://www.sciencedirect.com [Accessed 27 May 2021].

Andrew Godwin & Carlton Gibson. (2020) *Django Channels.* Available from: https://channels.readthedocs.io/en/latest/ [Accessed 18 June 2021].

Andrew Mccarthy. (2019) *Django Project Documentation.* [Accessed 11 June 2021].

Arpit Asati. (2019) *What is web socket and how it is different from the HTTP? .* Available from: https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/ [Accessed Dec 2019].

Becki Robins. (2019) *How To Get Accurate Colors In Your Photos.* Available from: https://photographypro.com/white-balance/ [Accessed 27 May 2021].

Chao-Yang Lu & Shao-Kang Hung. (2019) *Transforming an Optical Flow Sensor into an Angular Displacement Sensor .* [Accessed 10 June 2020].

Damien George & Daniel Campora. (2014) *The MicroPython project .* Available from: https://github.com/micropython/micropython/wiki/Performance [Accessed 8 June 2020].

Dave Marshall. (2001) *The Discrete Cosine Transform (DCT) .* Available from: https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node231.html [Accessed 11 June 2021].

Django Software Foundation. (2019) *Celery.* Available from: https://docs.celeryproject.org/en/stable/django/first-steps-with-django.html [Accessed 11 June 2021].

Free Software Foundation. (2021) *GNU Operating System.* Available from: https://www.gnu.org/licenses/ [Accessed 11 June 2021].

H. D. Cheng, X. H. Jiang, Y. Sun & Jing Li Wang. (2017) Color Image Segmentation: Advances & Prospects. Available from: https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=39645E1F5549AE5974610A920DF81EDF?doi=10.1.1.119.2886&rep=rep1&type=pdf.

Haarith Devarajan & Harold Nyikal. (2008) *Image Scaling and Bilateral Filtering .* Available from: http://scien.stanford.edu/pages/labsite/2006/psych221/projects/06/imagescaling/ [Accessed 8 June 2020].

IDA HÜBSCHMANN. (2021) *The Pros and Cons of Using MQTT Protocol in IoT .* Available from: https://www.nabto.com/mqtt-protocol-iot/ .

Irwin Sobel. (2014) An Isotropic 3x3 Image Gradient Operator. 1-2. Available from: https://www.researchgate.net/publication/239398674_An_Isotropic_3x3_Image_Gradient_Operator.

Jamie Ludwig. (2007) *Image Convolution.* Available from: http://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf .

Jarosław Bernacki. (2020) Automatic exposure algorithms for digital photography. Available from: https://www.researchgate.net/journal/Multimedia-Tools-and-Applications-1573-7721.

John Canny. (2017) A Computational Approach to Edge Detection. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.420.3300&rep=rep1&type=pdf.

Kwang-Il Kim, Jung Sik Jeong & Gyei-Kark Park. (2014) Development of grid projection algorithm of vessel trajectories for e-Navigation. Available from: https://ieeexplore-ieee-org.iclibezp1.cc.ic.ac.uk/document/7044879.

Marco Forgione. (2019) *rover 5 position control.* Available from: http://www.marcoforgione.it/rover5.html# [Accessed 11 June 2021].

Mathworks. (2021) *MPPT Algorithm* . Available from: https://www.mathworks.com/solutions/power-electronics-control/mppt-algorithm.html [Accessed 1 June 2021].

Matthew Brett. (2019) *Smoothing.* Available from: https://matthew-brett.github.io/teaching/smoothing_intro.html [Accessed 8 June 2020].

Meredith. (2021) *The Nine Belbin Team Roles* . Available from: https://www.belbin.com/about/belbin-team-roles [Accessed 12 June 2021].

Miran Gosta & M. Grgic. (2010) Accomplishments and challenges of computer stereo vision . 1-5. Available from: https://www.researchgate.net/publication/224183938_Accomplishments_and_challenges_of_computer_stereo_vision.

MQTT. (2021) *The Standard for IoT Messaging.* Available from: https://mqtt.org/ .

National Aeronautics and Space Administration, U. S. A. (2021) *Moving around Mars.* Available from: https://mars.nasa.gov/mer/mission/timeline/surfaceops/navigation/ [Accessed 10 June 2021].

Omega. (2021) *How to tune a PID controller? .* Available from: https://www.omega.co.uk/technical-learning/tuning-a-pid-controller.html [Accessed 10 June 2020].

On Amlendra. (2018) *Difference between I2C and SPI (I2C vs SPI) .* Available from: https://aticleworld.com/difference-between-i2c-and-spi/ [Accessed 8 June 2020].

Plotly. (2021) *Live Updating Components .* Available from: https://dash.plotly.com/live-updates [Accessed 27 May 2021].

Pramoth Thangavel. (2019) *Arduino Interrupt Tutorial.* Available from: https://circuitdigest.com/microcontroller-projects/arduino-interrupt-tutorial-with-examples [Accessed 8 June 2020].

R.H. Greenfield & W. Kinsner. (2006) *The Lempel-Ziv-Welch (LZW) data compression algorithm for packet radio.* Available from: https://ieeexplore.ieee.org/document/160551 [Accessed 10 June 2020].

Rapid Tables. (2021) *RGB to HSV color conversion .* Available from: https://www.rapidtables.com/convert/color/rgb-to-hsv.html [Accessed 11 June 2021].

Salvatore Sanfillipo. (2021) *Redis .* Available from: https://redis.io/ [Accessed 26 May 2021].

Santos Rui & Santos Sara. (2019) *ESP32-CAM Video Streaming Web Server.* Available from: https://randomnerdtutorials.com/esp32-cam-video-streaming-web-server-camera-home-assistant/ [Accessed 26 May 2021].

Shapiro, L. G. & Stockman, G. C. (2011) *Computer Vision .* Prentice Hall, 2001.

Ted Young. (2019) *ESP32 WebSocket for camera live stream.* Available from: http://www.iotsharing.com/2020/03/demo-48-using-websocket-for-camera-live.html [Accessed 27 May 2021].

Terry Huntsberger & Yang Cheng. (2015) Closed loop control for autonomous approach and placement of science instruments by planetary rovers . 2-5. Available from: https://www.researchgate.net/publication/224623181_Closed_loop_control_for_autonomous_approach_and_placement_of_science_instruments_by_planetary_rovers.

W. Kinsner & R.H. Greenfield. (2019) The Lempel-Ziv-Welch (LZW) data compression algorithm for packet radio . 1-12. Available from: https://ieeexplore.ieee.org/document/160551. [Accessed 8 June 2020].

Wilhelm Burger & Mark J. Burge. (2010) *Principles of Digital Image Processing Core Algorithms.* . Springer Science & Business Media.

Xin Xu, Yinglin Wang, Jinshan Tang, Xiaolong Zhang & Xiaoming Liu. (2011) Robust Automatic Focus Algorithm for Low Contrast Images Using a New Contrast Measure. 1-4. Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3231510/.

Yu Shuien, Chunyun Fu, Amirali K. Gostar & Minghui Hu. (2020) *A Review on Map-Merging Methods for Typical Map Types in Multiple-Ground-Robot SLAM Solutions* . Available from: https://www.researchgate.net/publication/347433066_A_Review_on_Map-Merging_Methods_for_Typical_Map_Types_in_Multiple-Ground-Robot_SLAM_Solutions [Accessed 10 June 2020].

Zarchan, P. & Musoff, H. (2015) *Fundamentals of Kalman filtering.* Progress in astronautics and aeronautics. Fourth edition. edition. Reston, VA, American Institute of Aeronautics and Astronautics, Inc. Available from: http://www.books24x7.com/marc.asp?bookid=101531 .

8. Appendices

Appendix 1: FPGA Resources Used by Vision Subsystem

Flow Summary	
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	17,771 / 49,760 (36%)
Total registers	8980
Total pins	171 / 360 (48%)
Total virtual pins	0
Total memory bits	1,236,890 / 1,677,312 (74%)
Embedded Multiplier 9-bit elements	19 / 288 (7%)
Total PLLs	1 / 4 (25%)
UFM blocks	0 / 1 (0%)
ADC blocks	0 / 2 (0%)

Table 15: Quartus Compilation Flow Summary

Timing from Slow 1200mV 85C Model			
Fmax	Designed frequency	Clock Name	Remarks
39.33 MHz	100.0 MHz	u0|altpll_0|sd1|pll7|clk[2]	Clock for Video Pipeline Elements
68.1 MHz	50.0 MHz	MAX10_CLK1_50	Clock for Nios II processor and other Qsys components
87.28 MHz	25.0 MHz	MIPI_PIXEL_CLK	Clock to Camera components
90.99 MHz	25.0 MHz	u0|altpll_0|sd1|pll7|clk[3]	Clock to Camera components
113.02 MHz	10.0 MHz	altera_reserved_tck	

Table 16: Quartus Timing Analyser Summary

PowerPlay Power Analyser Summary	
Device	10M50DAF484C7G
Power Models	Final
Total Thermal Power Dissipation	539.23 mW
Core Dynamic Thermal Power Dissipation	345.69 mW
Core Static Thermal Power Dissipation	98.25 mW
I/O Thermal Power Dissipation	95.28 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Table 17: Quartus Power Analyser Summary

Appendix 2: RGB to HSV Conversion Implementation

Calculated Subcomponents:

$$R' = \frac{R}{255}, G' = \frac{G}{255}, B' = \frac{B}{255}$$

$$C_{max} = \max(R', G', B')$$

$$C_{min} = \min(R', G', B')$$

$$\Delta = C_{max} - C_{min}$$

As floating-point numbers are difficult to implement in hardware, it was decided to skip the scaling step in the formula and to account for the difference in the range of values elsewhere. Hence, the raw values of R, G, B (from 0-255) are input into the C_{max}, C_{min} and Δ functions.

Value:

Formula: $\boldsymbol{Value} = C_{max}$.

Value would originally be returned as a percentage from 0-100%, as C_{max} is usually represented as a fraction of 255. However, it is returned with a range of 0-255 in hardware.

Saturation:

Formula: $\boldsymbol{Saturation} = \begin{cases} 0, & C_{max} = 0 \\ \frac{\Delta}{C_{max}}, & otherwise \end{cases}$

Modifications:

$$Saturation = \frac{\Delta}{C_{max}} = \frac{C_{max} - C_{min}}{C_{max}} = 1 - \frac{C_{min}}{C_{max}}$$

Following the convention for Value, to represent a value of 100% Saturation as 255, an overall multiplication of 256 should be done to the Saturation term. This is easily performed in hardware with a bit extension by 8 bits.

Hence, the final equation used is $Saturation = 256 - \frac{C_{min} \ll 8}{C_{max}}$.

Hue:

Formula: $\boldsymbol{Hue} = \begin{cases} 0, & \Delta = 0 \\ \left(60 \times \left(\frac{G\prime - B\prime}{\Delta}\right) + 360\right) mod\ 360, & C_{max} = R \\ \left(60 \times \left(\frac{B\prime - R\prime}{\Delta}\right) + 120\right) mod\ 360, & C_{max} = G \\ \left(60 \times \left(\frac{R\prime - G\prime}{\Delta}\right) + 240\right) mod\ 360, & C_{max} = B \end{cases}$

Hue is typically represented as an angle around a circle ranging from 0-360°, with the area around 360-0 representing red, around 120 representing green, and 240 representing blue. A scale factor of 60 applied to the difference is required to obtain the correct hue value.

However, to fit into an 8-bit value, the Hue value is scaled to fit between 0-255 instead. In addition, the scale factor is changed to $60 \div 360 \times 255 \approx 42$.

Ordinarily, calculating Modulo would require the use of an additional division unit to give the remainder term. However, as the value of the will not exceed -256 or +511, adding or subtracting values to keep the result from 0-255 is sufficient.

Modification:

$R_{mult} = 42 \times R$, $G_{mult} = 42 \times G$, $B_{mult} = 42 \times B$

$$SubtRes = \begin{cases} G_{mult} - B_{mult}, & C_{max} = R \\ B_{mult} - R_{mult}, & C_{max} = G \\ R_{mult} - G_{mult}, & C_{max} = B \end{cases}$$

$$DivRes = \begin{cases} 0, & \Delta = 0 \\ \dfrac{SubtRes}{\Delta}, & \Delta \neq 0 \end{cases}$$

$$HueAdd = \begin{cases} 255, & C_{max} = R \\ 85, & C_{max} = G \\ 170, & C_{max} = B \end{cases}$$

$$HueMap = DivRes + HueAdd$$

$$\boldsymbol{Hue} = \begin{cases} HueMap - 255, & HueMap > 255 \\ HueMap + 255, & HueMap < 0 \\ HueMap, & otherwise \end{cases}$$

In order to meet timing, these calculations can be split up into different phases to facilitate pipelining. The calculations were split as such:

Combinatorial	- Find $\max(R, G, B)$ and $\min(R, G, B)$ by comparing their individual sizes. - Perform the subtraction and selection operation for $SubtRes$ - Select the appropriate value for $HueAdd$
Stage 1	- Calculate R_{mult}, G_{mult}, and B_{mult}. - Calculate Δ. - Perform Left Shift for C_{min}.
Stage 2	- Perform division for Saturation ($\frac{C_{min} \ll 8}{C_{max}}$).
Stage 3	- Perform division for Hue ($SubtRes$).
Stage 4	- Compute $HueMap$ by adding $DivRes + HueAdd$
Stage 5	- Perform Hue "modulo" operation, and output result. - Perform Saturation subtraction operation and output result. - Assign Value as C_{max} and output result.

Table 18: Pipelined implementation for RGB to HSV convertor

Appendix 3: Bilateral Filter Implementation

The bilateral filter has equation $I^{filt}(x) = \frac{1}{W_p}\sum_{x_i \in \Omega} I(x_i) f_r(||I(x_i) - I(x)||) g_s(||x_i - x||)$, with normalisation term $W_p = \sum_{x_i \in \Omega} f_r(||I(x_i) - I(x)||) g_s(||x_i - x||)$.

I^{filt} refers to the filtered image, and I refers to the image to be filtered.
x is the coordinates of the current pixel to be filtered.
Ω is the image window centred on x, thus $x_i \in \Omega$ is another pixel in the window.
f_r is the range kernel for smoothing differences in intensities.
g_s is the spatial kernel for smoothing differences in coordinates. This was set to be identical to the coefficients for an approximation of Gaussian smoothing with a 3x3 kernel, shown below.

$\frac{x_{1,1}}{16}$	$\frac{x_{1,2}}{8}$	$\frac{x_{1,3}}{16}$
$\frac{x_{2,1}}{8}$	$\frac{x_{2,2}}{4}$	$\frac{x_{2,3}}{8}$
$\frac{x_{3,1}}{16}$	$\frac{x_{3,2}}{8}$	$\frac{x_{3,3}}{16}$

$x_{1,1} \gg 3$	$x_{1,2} \gg 2$	$x_{1,3} \gg 3$
$x_{2,1} \gg 2$	$x_{2,2} \gg 1$	$x_{2,2} \gg 2$
$x_{3,1} \gg 3$	$x_{3,2} \gg 2$	$x_{3,3} \gg 3$

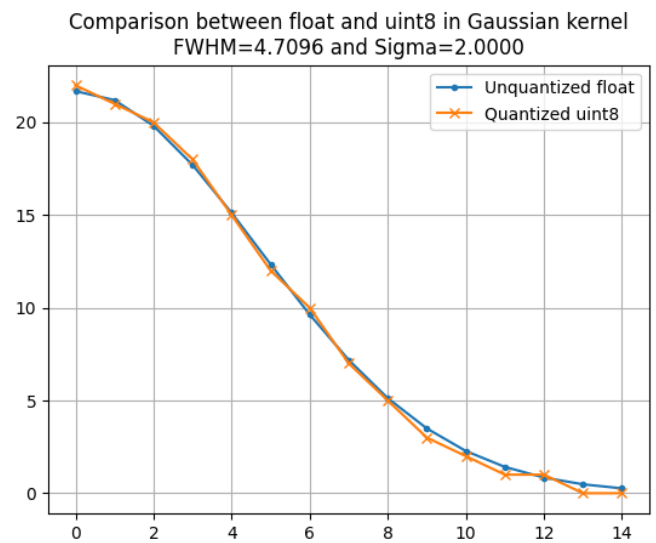
As the coefficients are neatly powers of two, instead of using expensive and slow dividers, it is possible to simply right-shift the coefficients instead.

Figure 37: Coefficients for Gaussian Smoothing

As the possible range of pixel values is from 0-255, the range of values for $|I(x_i) - I(x)|$ are limited to 0-255. This would then be needed to be mapped to a Gaussian distribution centred around 0, with variance as a parameter to be set. To reduce device area required, a submodule containing a lookup table was used. In addition, to reduce overall memory footprint, only the 4 MSBs of the absolute difference were considered, reducing the input range from 256 to 16.

The coefficients for f_r were then designed with help from an online resource (Matthew Brett, 2019). Using NumPy, a Gaussian curve was plotted for values from 0-15, and the variance adjusted till appropriate. The Full Width at Half Maximum (FWHM) measure was used to aid in design of the curve.

Quantization was then done by scaling the entire graph by 256 (8-bits) and then rounding off to the nearest 8-bit integer. The scaling for each value can then be done by multiplying by the numerator factor and then right shifting by 8 bits.



A graph comparing the unquantized (32-bit floating point) and quantised (8-bit unsigned integer) coefficients for the numerator term at each possible value between 0-15.

As can be seen, there is not much quantization error in this step.

Figure 38: Comparison between 8-bit unsigned and floating point values for Gaussian kernel.

After the coefficients for f_r and g_s are computed, then two sums are calculated, the sum of $f_r \times g_s$ (which is W_p) and of $I(x) \times f_r \times g_s$ over the nine pixels in the 3x3 grid.

Lastly, the output is given as the quotient of the second sum and W_p, corresponding to the filtered value of pixel (2,2).

Appendix 4: Sobel Operator Implementation

The imaging kernels for G_x and G_y are below. These were calculated using the convolution machinery detailed in Section 2.2.2.2.1.

The LPM_SQRT megafunction was used to aid in calculating the square root value.

G_x:

$+1$	0	-1
$+2$	0	-2
$+1$	0	-1

G_y:

$+1$	$+2$	$+1$
0	0	0
-1	-2	-1

After obtaining G_x and G_y for each individual pixel, the overall gradient magnitude is computed as

$$G = \sqrt{G_x^2 + G_y^2}$$

Figure 39: Convolution kernels for Sobel Operator

Appendix 5: View from each Detection IP

Below are images captured using the Windows Camera program from the VGA output of the FPGA. The approximate outputs of each IP were multiplexed to the FPGA video output, allowing debugging.

Video artefacts exist on the left edge of the frame, due to the unhandled delay between video pixel input and output. However, they do not affect the correctness of the detection IP.

Colour Detection

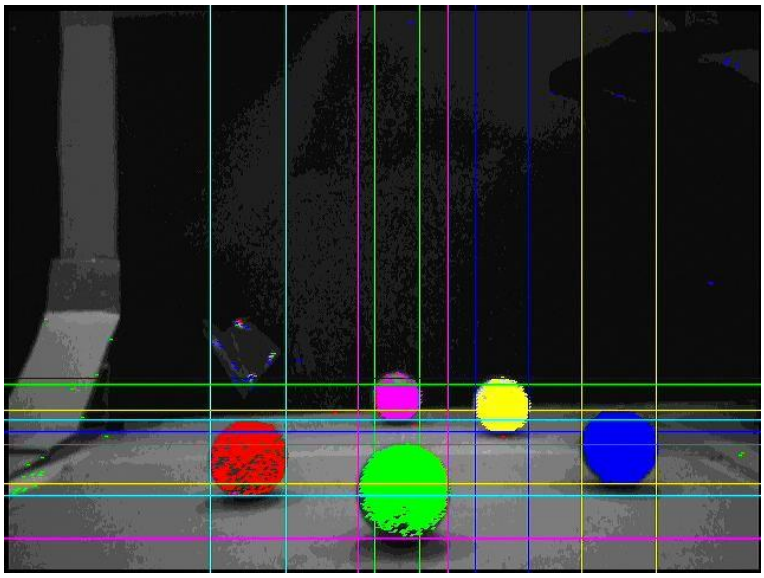


Figure 40: Illustration of colour detection

As can be seen, all colours are detected and distinguishable. It can also be seen that though only parts of the yellow, red, and purple balls are detected, the bounding box approximately encapsulates the area around it, because the pixels are in close proximity.

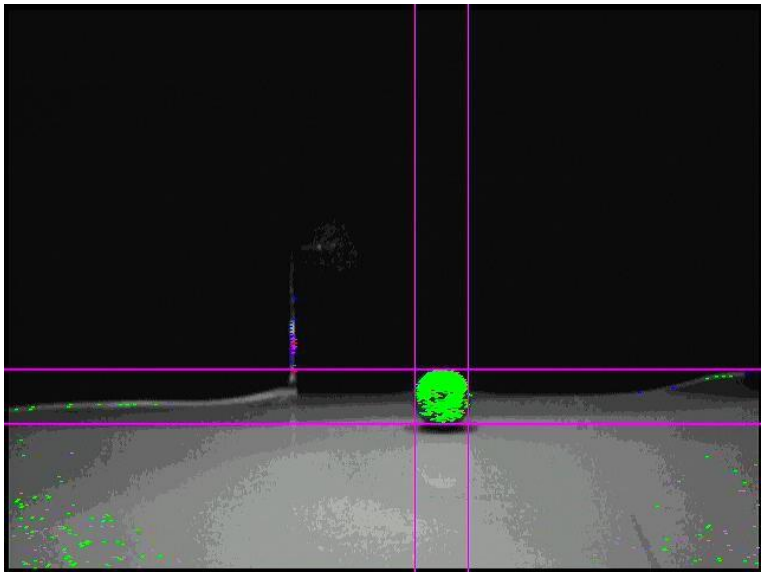


Figure 41: Illustration of volume detection

Even though there are green pixels detected in the image, the bounding box centres around the main green ball. This shows that the volume detection system is working.

Bilateral Filter

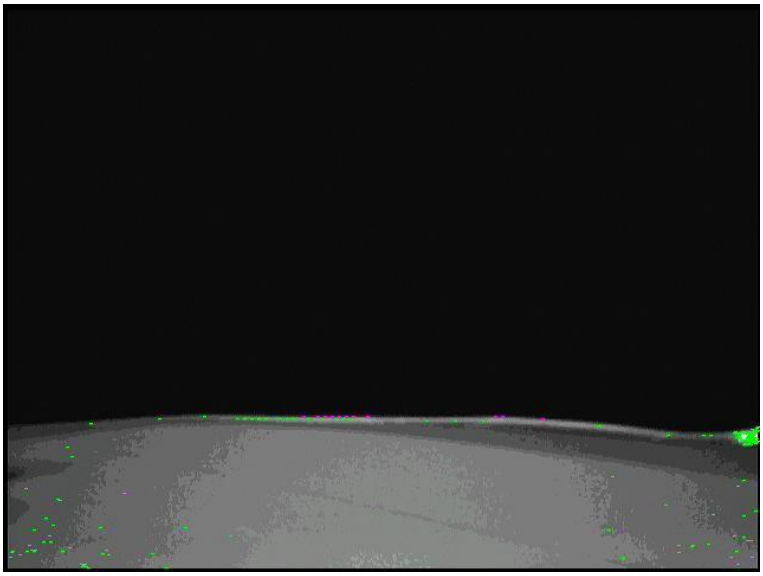


Figure 42: Illustration of the input to the Bilateral Filter

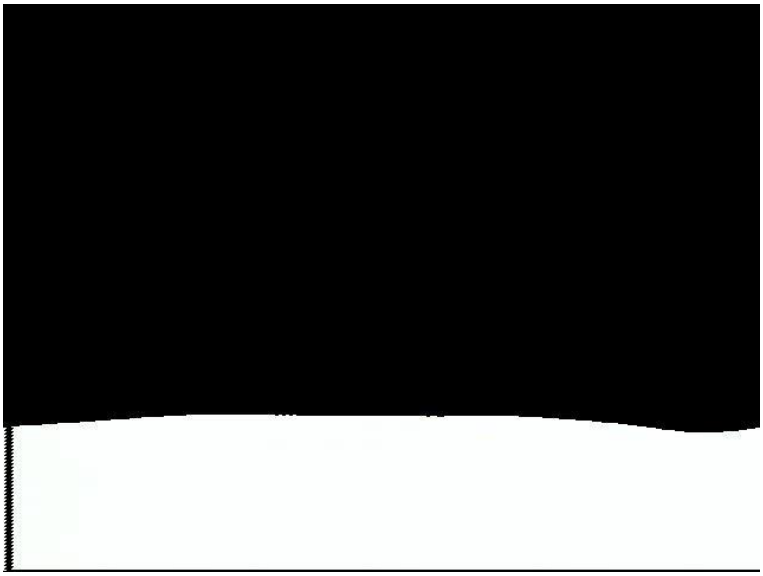


Figure 43: Illustration of the Bilateral Filter output after brightness thresholding

The Bilateral Filter IP, along with its Brightness Thresholding function, clears out visual noise in the image. Though there are shadows and lines in the image that may result in unwanted edges being detected by the subsequent Sobel Edge Detector, the output image presents a clear, unambiguous edge for detection.

Sobel Edge Detector

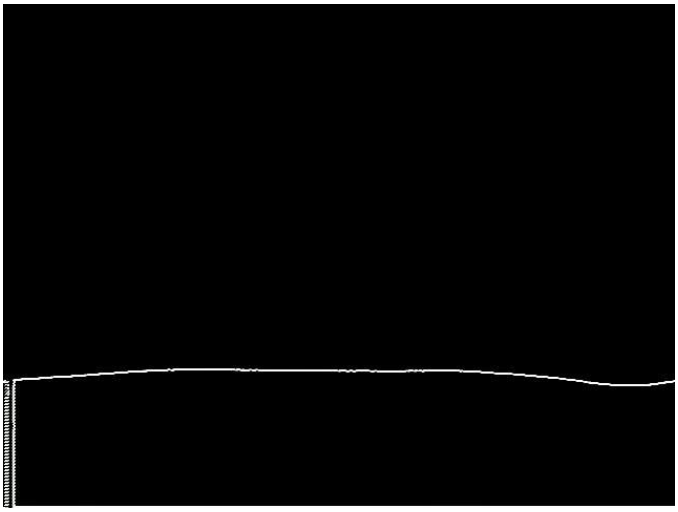


Figure 44: Illustration of output of Sobel Edge Detector

The output of the Sobel operator is shown. The artefacts on the left edge of the image are a result of the x,y pixels of the VGA output not being in sync with the x,y pixels of the Sobel IP output. Nevertheless, a line corresponding to a wall is clearly visible.

As can be seen, there is a very strong edge detected, corresponding to the intensity difference highlighted earlier.

Non-Maximum Suppression and Hough Transform

Figure 45: Illustration of output of Non-Maximum Suppression followed by Hough Transform Overlay

The line detected from the Edge Detection is thinned using Non-Maximum Suppression. In addition, debug output from the Hough Transform IP is displayed on the left and right of the screen. On the left, a representation of the Accumulator is shown for visual comparison with the output of the Python test algorithm. On the right, a representation of the Image Buffer is shown, showing its 20x120 resolution.

Appendix 6: Calculating Distances from Y-coordinate

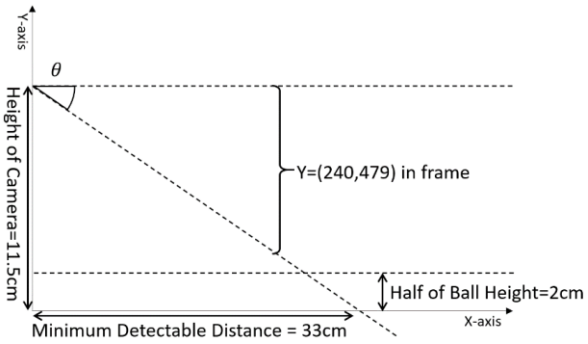


Figure 46: Distance Measurement from Y-Coordinate

Converting y-coordinate values to distances was done by solving the geometric representation of the system shown above. The angle θ between the two dotted lines shows how the lower half of the camera's field of view grows larger as distance away from the camera increases.

The distance shown in the curly brace is shows the physical height that can be viewed at each distance from the camera. This increases with distances away from the camera, but the number of pixels remains the same, 240, corresponding to the lower half of the image frame.

By taking the ratio of the y-coordinate of the centre of the ball, the corresponding distance away from the ball can be computed.

The pixel coordinate at heights 2cm (for ball detection) and 0cm (for wall detection) can be given as the ratio $P - 240 = \frac{240\times(11.5-h)}{\frac{11.5}{33}\times d}$, where h is the height in question and d is the distance from the camera. Inverting the function such that P is the subject of the equation gives the neat equations $d = \frac{7923}{P-240}$ for wall detection and $d = \frac{6542}{P-240}$ for ball detection.

In addition, conversion from r, θ values from the Hough transform to pixel y-coordinates was needed for wall detection. This was done with trigonometric calculations using the following code snippet. As there was insufficient program memory for `math.h` for trigonometric functions on the Nios 2, the r and θ values were passed to Command, where they were further processed.

```
uint8_t wallDistFromRTheta(uint8_t r, uint8_t theta_in) {
  int theta = theta_in * 2 + 50;
  float dist = 0; // Perform operations on this variable

  if (theta == 90) return r;

  if (theta > 90) {
    theta -= 90;
    dist = (float) r * cos((float) theta * pi / 180) + (10.0 - (float) r *
sin((float) theta * pi / 180)) * (tan((float) theta * pi / 180));
  } else {
    dist = (((float) r / cos((float) theta * pi / 180)) - 10.0) *
tan((float)(90 - theta) * pi / 180);
  }
  dist = 240 + dist * 2; // Y-value on screen

  if (dist > 435) return 33;
  if (dist < 265) return 254;

  return (int) 6542 / (dist - 239);
}
```

Appendix 7: Vision Distance Calibration Rig

A distance calibration rig was made out of cardboard. This allowed for efficient validation of distance testing algorithms as it was built with a scale that showed distance from the front of the camera.

The height of the camera on the rover was measured and replicated using PCB standoffs to ensure consistency between the test rig and deployment on the rover.

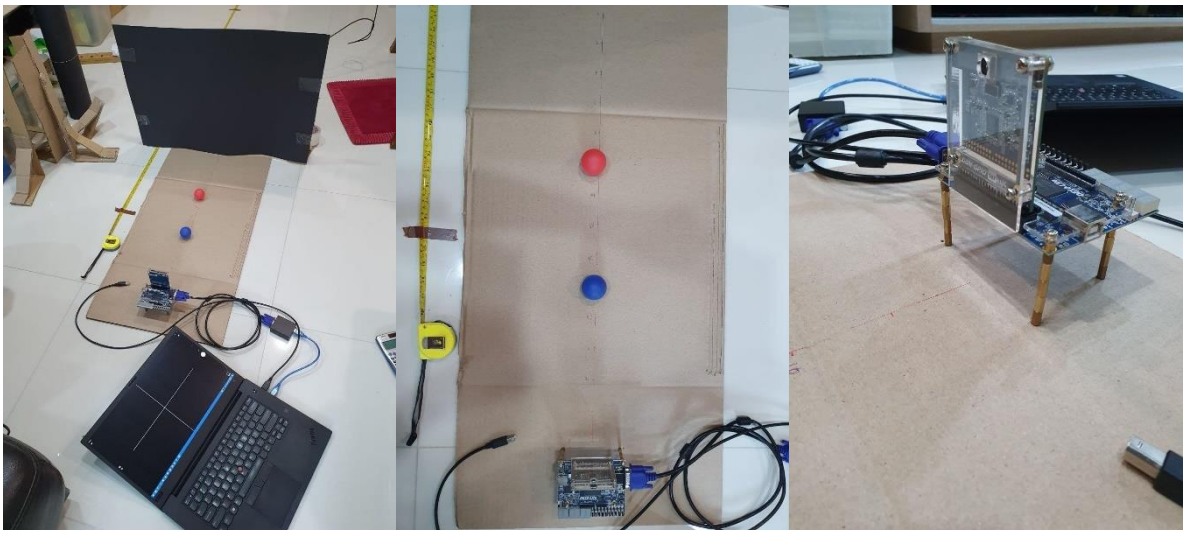


Figure 47 : a) Overall view of Distance Calibration Rig (left) b) Top view of Distance Calibration Rig (centre) c) Standoffs for FPGA height compensation (right)

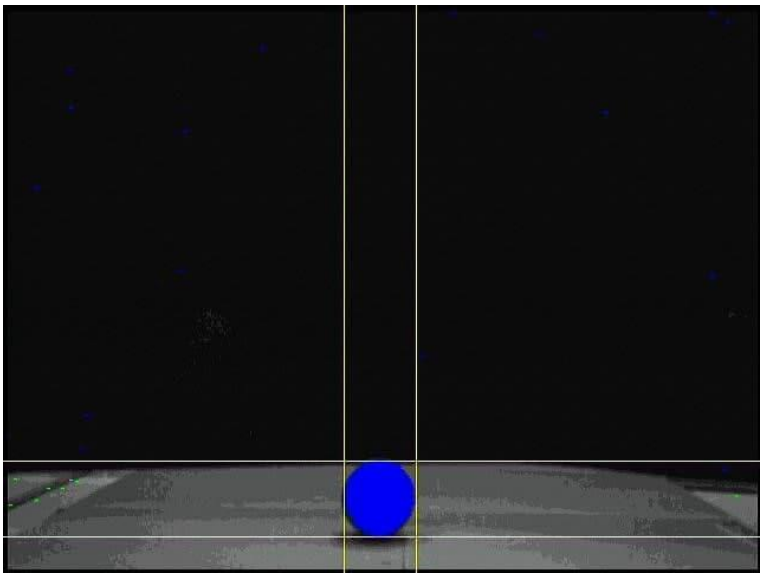


Figure 48: View from the Distance Calibration Rig

Appendix 8: Drive Hardware Design and PCB

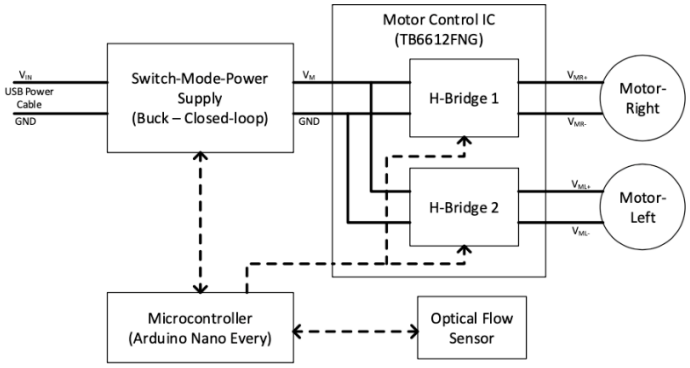


Figure 49: Drive Hardware Design

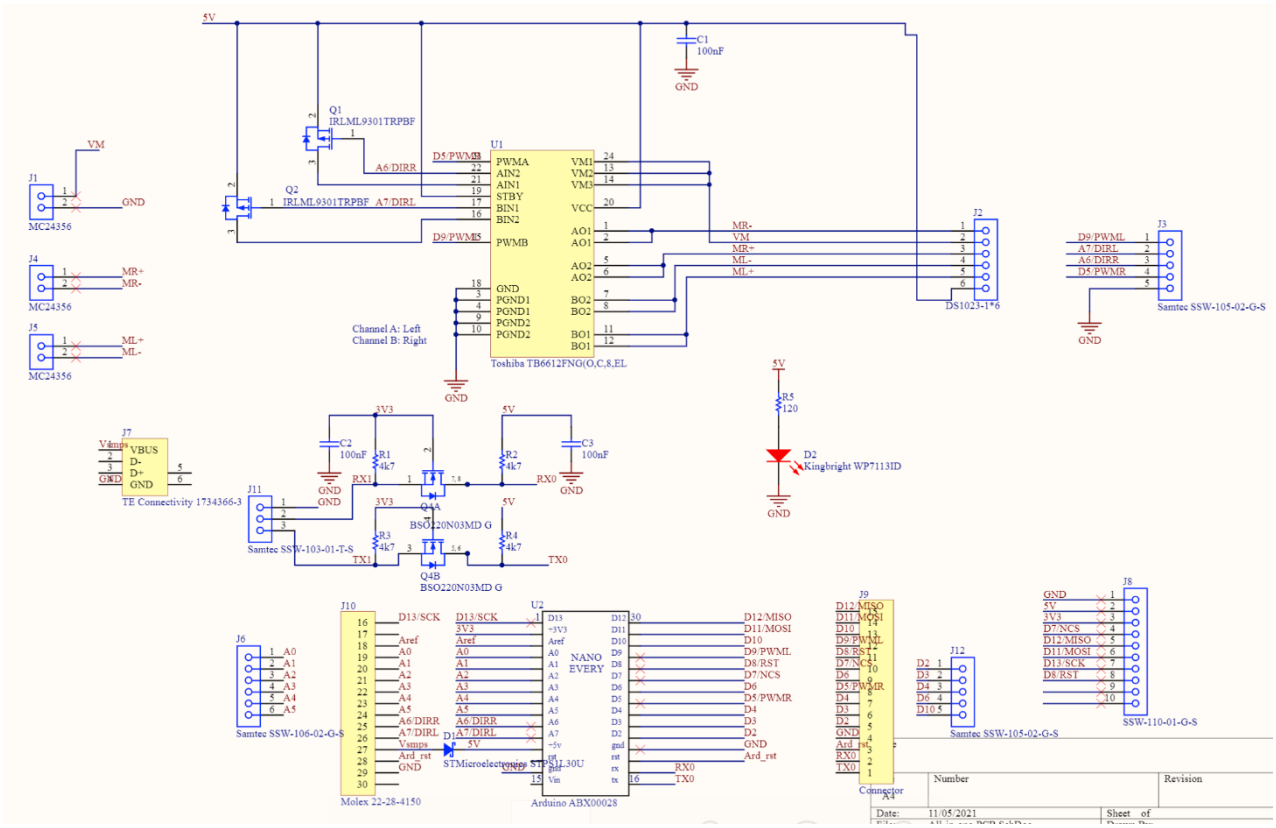


Figure 50: Drive PCB Connections

Appendix 9: Command data flow

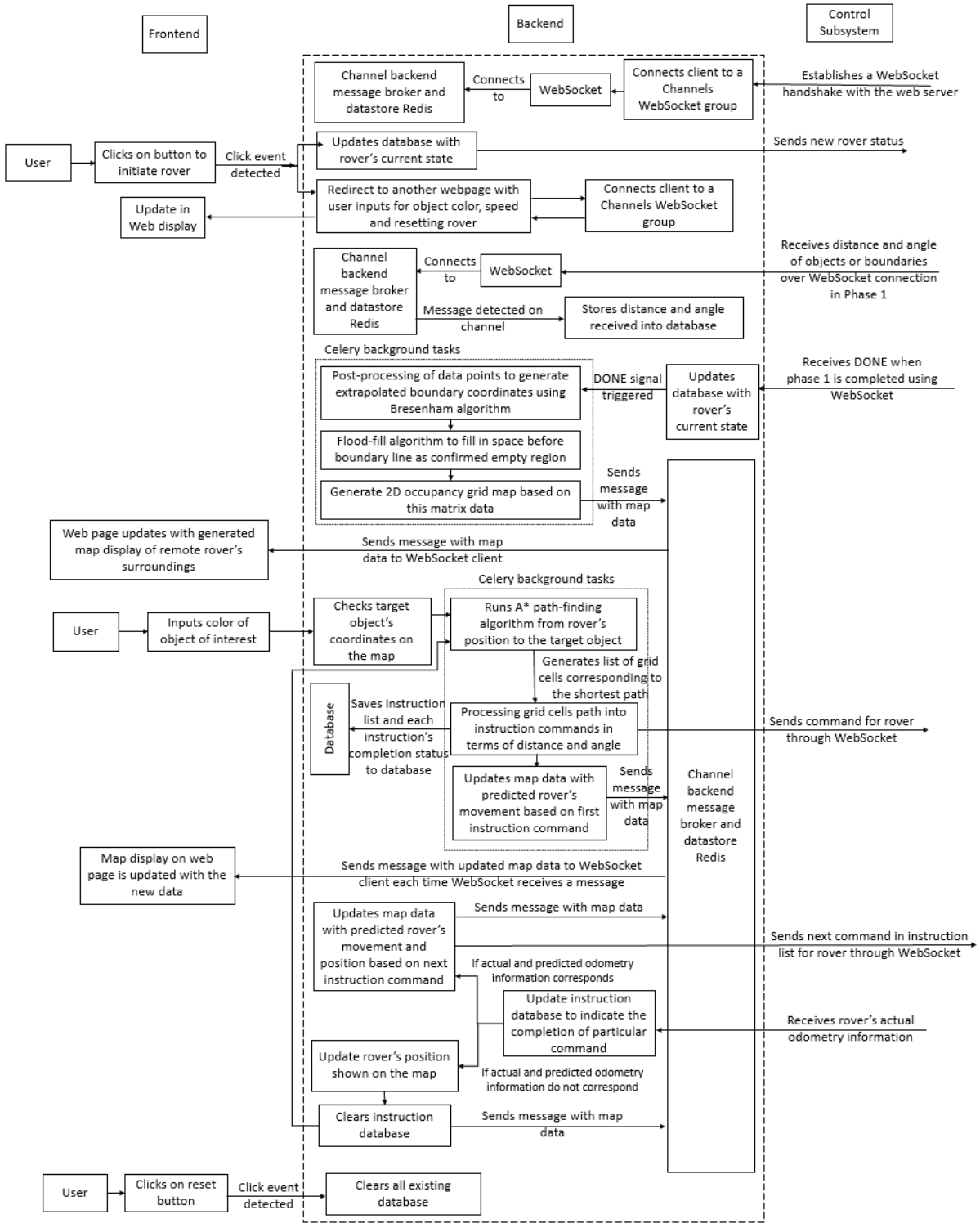


Figure 51: Detailed view of data flow within Command Web App and to and from Control

Appendix 10: Command Testing

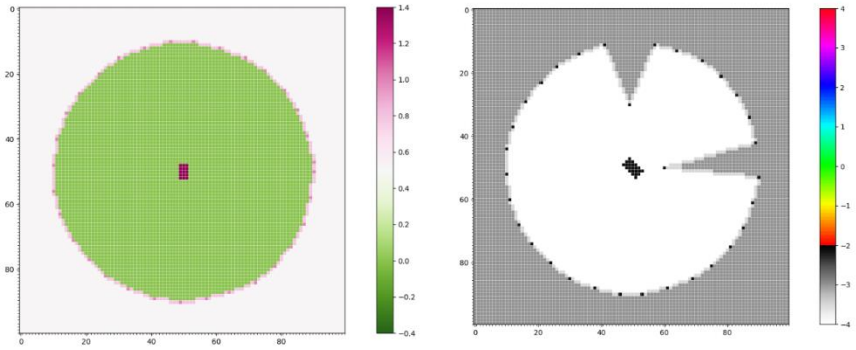


Figure 52: Map generation with equal distanced obstacle points

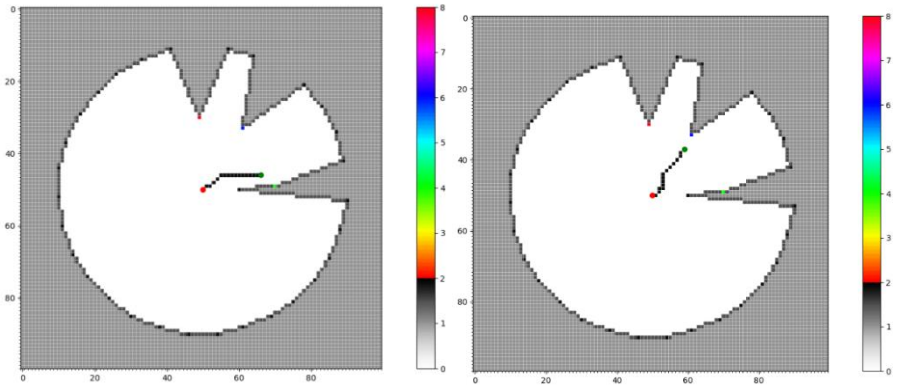


Figure 53: Map generation with unequal distanced obstacle points, and path generated by A* algorithm marked by the grid cells coloured black

Appendix 11: Energy Hardware Setup Consideration

Configuration (Port A)	Configuration (Port B)	Buck/ Boost	Suitability and Reasoning
3 series cells	4 series PV panels	Boost	The port A voltage is limited to 8V. 4 series PV panels under optimal conditions have a combined nominal voltage of 20V. As input to the boost circuit, it violates the port A voltage restriction.
3 series cells	4 parallel PV panels	Boost	The port A voltage is limited to 8V. The maximum voltage of a single cell is 3.6V. The minimum voltage of a single cell is 2.5V. The nominal voltage of a single cell is 3.2V. 3 cells in series would have a combined nominal voltage of 9.6V, which violates the 8V port A limit. If the cells were to be connected in series, in order to abide by the limit, they cannot be charged to anywhere near full capacity so it is a sub-optimal configuration.
3 parallel cells	4 series PV panels	Boost	The port A voltage is limited to 8V. 4 series PV panels under optimal conditions have a combined nominal voltage of 20V. As input to the boost circuit, it violates the port A voltage restriction.
3 parallel cells	4 parallel PV panels	Boost	The cells would get over-voltage. The maximum voltage of each cell is 3.6V. Under optimal conditions, the voltage at port B is 5V so the Boost circuit would provide much greater voltage than 3.6V required at port A for the parallel combination of cells.
4 series PV panels	3 series cells	Buck	Under optimal conditions, the voltage of a single panel is 5V. Therefore, the nominal voltage of 4 solar panels connected in series would be 20V, which is much larger than the 8V limitation enforced on port A.
4 series PV panels	3 parallel cells	Buck	Under optimal conditions, the voltage of a single panel is 5V. Therefore, the nominal voltage of 4 solar panels connected in series would be 20V, which is much larger than the 8V limitation enforced on port A.
4 parallel PV panels	3 series cells	Buck	Under optimal conditions, the maximum voltage at port A would be 5V. If the voltage of each cell is 3.2V, the output nominal voltage of the 3 cells in series is 9.6V. This isn't possible with a buck configuration
4 parallel PV panels	3 parallel cells	Buck	This is the most suitable configuration as the voltage at port A is below the 8V limit on port A. Since the cells are in parallel, the output voltage is going to be in the range of 2.5V to 3.6V, which is lower than the input voltage which is typically around 5V so the buck setup allows the best charging of the cells in this configuration.

Table 19: Energy Hardware Setup Considerations

Appendix 12: Arena Setup

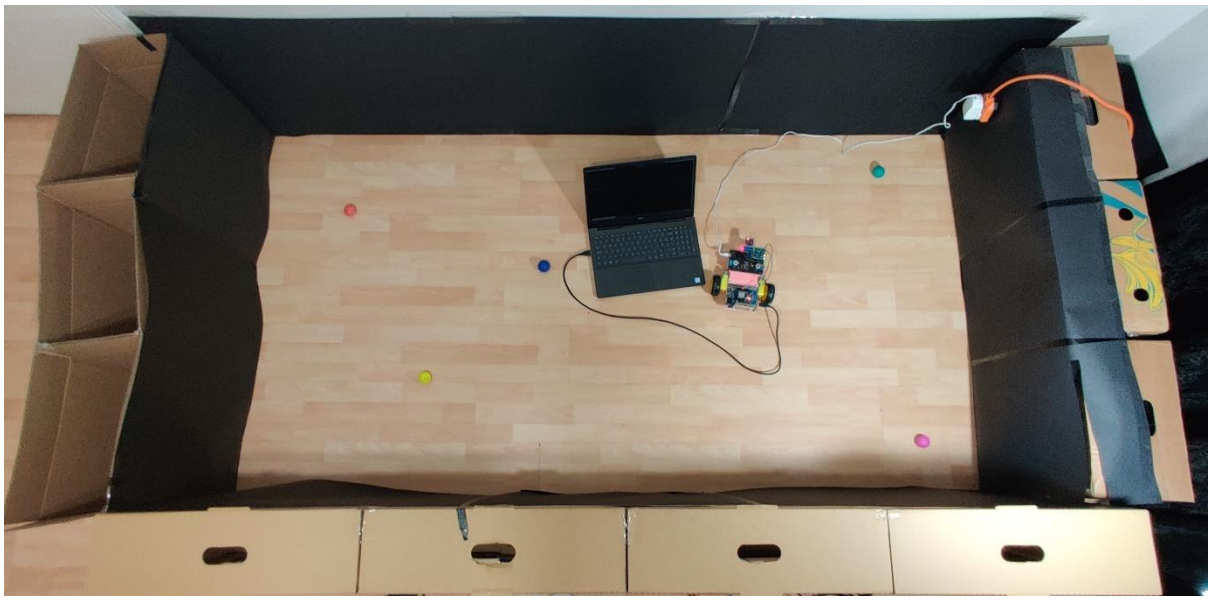


Figure 54: Arena Set-up

The arena was a 2.5 by 1.5m rectangle bordered with black cardboard walls about 50cm high. This ensured that the background seen by the camera would be a known constant.

The floor of the room was left unchanged, ensuring that a high image quality detected by the optical flow sensor.

Extension cords were used to connect the local power supply unit to the rover. It is envisioned for battery packs to be used instead to ensure greater mobility of rover and prevent wire interference.

Difficulty was encountered in ensuring uniform lighting for the entire arena as the walls introduced shadows to the arena edges, meaning that balls were more difficult to detect then.

Appendix 13: Pin Connections to ESP32 for Communication and to SMPS Shield for Balancing and Cell Voltage Measurement

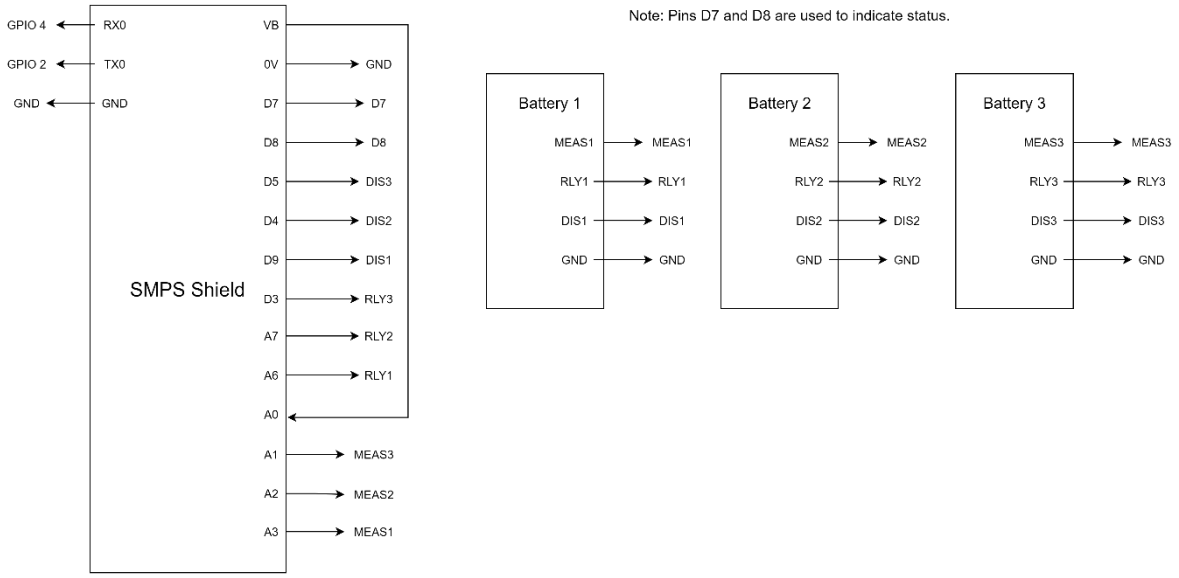


Figure 55: Pin Connections to Energy Subsystem Hardware