Imperial College London

Imperial College London

Second Year Design Project

ELEC50003/ELEC50008

# The MARS Rover

*Group 1*

*Georgios Chaimalas*
*Dimitrios Georgakopoulos*
*Edvard J. Skaarberg Holen*
*Hyunjoon Jeon*
*Josiah Mendes*
*Raghav Viswakumar*

Word Count: 18860 Words
June 16, 2021

# Contents

# 1 Introduction

The aim of the Mars Rover Project as outlined by our organisers is as follows: "to design and build an autonomous rover system that could be used in a remote location without direct supervision"[1]. To fulfil this broad goal, the design and development of the rover is separated into six distinct subsystems: **Command, Control, Drive, Energy, Integration and Vision**, each of which will be led by a single team member. These modules will work semi-independently to achieve the functional requirements as defined by the problem and eventually integrate their findings and goals to achieve the goals set out by the specification at hand. We will work under the main consideration that our rover is meant to be a proof of concept for one that could function on Mars. This report contains our design process for the rover, how it was implemented and challenges that were faced.

# 2 Structural Design

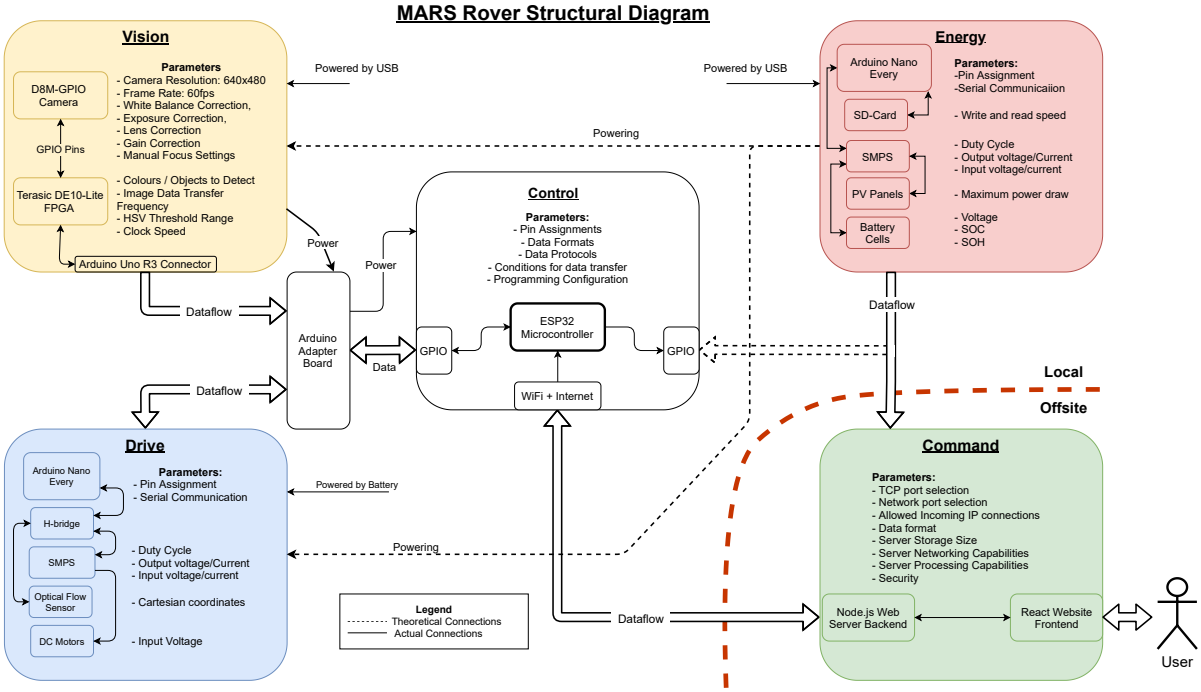


Figure 1: Structural Design Diagram showing Hardware and Parameters of Subsystems for Mars Rover

The structural design of the Rover as shown in Figure 1, shows the 5 main subsystems of the rover together with their main hardware components. This presents an overview of the entire Rover system and how the different components interact with each other. Both power connections and data connections are shown, together with corresponding arrows where directional transfer is necessary. Each subsystem's hardware has a set of configurable parameters that are also shown, these parameters are determined as part of the functional design process and implementation.

# 3 Functional Design

## 3.1 Control

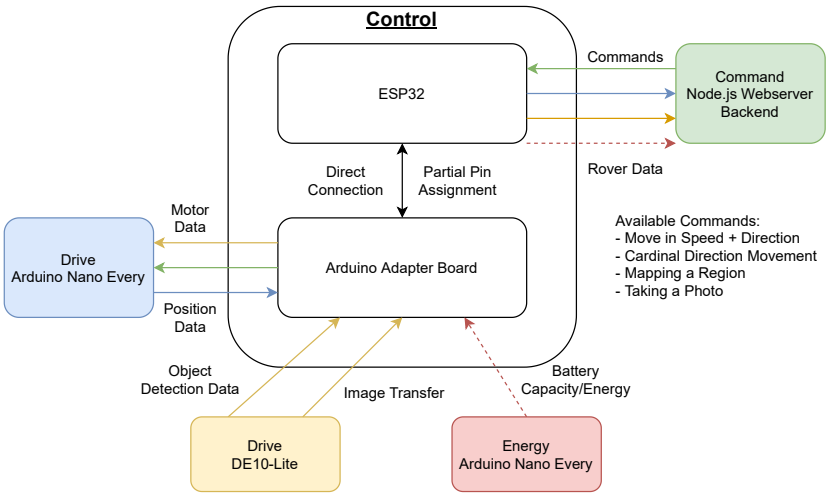


Figure 2: Functional Design Diagram showing Dataflow for Control Subsystem

### 3.1.1 Requirements

The Control module within the context of the rover has one main goal: **to act as the communication hub between all the subsystems, delivering the relevant data where it is required to allow for the rover to integrate as a whole** [1]. These functional requirements can be obtained from the functional design diagram for control, see Figure 2, outlining all the data flow for the Control subsystem. There are a couple of core objectives which must be achieved in order to fulfil this role as outlined below:

1. Act as a WiFi Access Point sitting under a router to receive commands from the Command subsystem and send data through socket-level communication.
2. Use a relevant hardware-level data transmission protocol to send movement commands to the Drive subsystem and receive data for Command.
3. Use a relevant hardware-level data transmission protocol to receive the Vision subsystem data.
4. Connect the Energy subsystem with the rover, sending the relevant data to the Command subsystem.

Although the hardware is fixed, the solutions to the outlined problems above are mostly up to design. It is for this reason that the main challenge stems from understanding what works best for communication to each subsystem and more importantly, how to integrate it such that the transmission of data can be smoothly facilitated through the ESP32.

### 3.1.2 Hardware Organisation

The Control module comprises of two hardware components [2]: the ESP32, a system on chip microcontroller [3], as well as an Arduino adapter board designed to map some of the available GPIO pins on the ESP32 to an Arduino-like board [4].

This hardware was provided by the project organisers and it immediately limits the tools which can be used, however the wide array of functionality which this microcontroller can provide through

programming the chip allows for sufficient freedom and capabilities to complete the required tasks. The GPIO pins are highly configurable when compared to an Arduino-Uno [5], as they are capable of using more data communication through almost any of the pins via software configuration [6]. It is low-cost and low-power, capable of using WiFi capabilities at 2.4GHz and Bluetooth [3] to interface wirelessly with other devices. Using the Arduino Adapter board [4], it can sit directly on top of an Arduino or device with similar pins to directly interface through physical connections like the FPGA in this case.

For wired communications, the ESP32 has limitations for the type of communication peripherals it can use with peer microcontrollers. Unlike Arduinos which have at most 1 SPI, 2 I2C and 1 UART port [7], the ESP32 can support up to 3 SPI, 2 I2C and 3 UART ports [3] which are configurable to most of the GPIO pins available. Considering the ESP32 will have to interface with at most three other subsystems through wired communication, there is a lot of choice with peripherals that must be evaluated.

As the three most common hardware peripherals to use in any engineering project, there is a discussion to be had about the differences between UART, SPI and I2C [8]. UART and I2C utilise 2 ports for connections on each device while SPI uses 4. While UART and SPI are full-duplex, allowing for simultaneous data transfer between devices, this is not the case for I2C. With regards to complexity, UART is the simplest form of transfer that is available. Due to the number of devices in SPI and I2C able to exceed 2, in comparison to UART, they can be much harder to set up as they include additional checks to reflect their multiple device support. While I2C and SPI may be faster methods for data transfer, they limit flexibility in design as I2C for example, requires one device to designate itself as a controller and another as a peripheral, which is not ideal for devices which need to regularly instantiate communication with one another as controllers.

### 3.1.3 Initial Design Choices

There are many options available for programming the ESP32. Due to the relaxed requirements with regards to communication, it was quickly determined that the Arduino IDE and supporting libraries for the ESP32 would be enough for developing what is necessary. As there are stringent practices for the base Arduino Libraries and ESP8266 Libraries, this was extended to a majority of the functionality available to the ESP32 including WiFi, Bluetooth, UART, SPI and I2C [9]. Seeing as the ESP32 through Espressif is programmed through C, most of this can be imported through the Arduino IDE which supports simpler methods for flashing software onto the ESP32.

As there is no requirement imposed upon the data being transferred and displayed, this results in complete freedom for the form in which data is communicated through the ESP32 and whether it should be processed. Seeing as the ESP32 will be handling communication between four other subsystems, it was decided that the amount of processing on the device should be limited as much as possible and offloaded to the other subsystems. This is to ensure that bottlenecks would not appear due to a sizeable processing time in between transmission of data, as this would cause crippling delays for communication. The simplest way to ensure the integrity of the data is to standardise the type for data being transferred. Since initially, the formatting and content of the data was unclear, receiving and sending them as bytes or characters would be the most versatile solution. Internal buffers for the ESP32 would store individual commands or data lines and send them to the required subsystem for processing.

With regards to the communication between Command and Control, there were a couple of options available. As the Command subsystem would be hosted on a website, we could choose from application layer protocols like HTTPs or MQTT, but as we decided our application was more simplistic with regards to the type of data being communicated, we opted for the transport layer protocol, TCP. This is due to the need for consistent transmission of data, retransmitting if packets are lost and ensuring the data does reach its destination. Speed by comparison is a less stringent requirement and as such, UDP was discarded as a choice [10].

For both Drive and Energy, the Arduino Nano Every provided by the project organisers had it's UART ports exposed through the specific hardware and it's connections. As such, there was little need for deliberation over the protocol being used [11]. The peripheral chosen for both these groups was UART. For the Energy submodule specifically, since it can't be integrated into the Rover physically, it was decided that any necessary data would be transmitted directly to Command. This is done through the UART USB connection from the Arduino to the computer, which would then communicate with

Command over TCP. However, to ensure Energy can be integrated if necessary, one UART port for the ESP32 was set aside for Energy.

For Vision, there are two types of data which need to be transferred through the ESP32; the image from the camera and data regarding object detection which could include values like distance to the object. Since the image transfer will likely take a long time to accomplish, a decision was made to dedicate two peripherals between Vision and Control. Considering the ESP32 will sit on top of the DE10-Lite FPGA, the availability of ports is not a concern. As two UART ports are unavailable since one will be used to communicate with Drive and the other is dedicated for Energy, the peripherals chosen instead will be SPI. One SPI connection is dedicated to sending images while the other is dedicated to sending any data between the ESP32 and the FPGA.

## 3.2 Command

### 3.2.1 Requirements

The Command module has three main purposes within the system of the rover:

1. Visualise data from the other subsystems
2. Allow users to command the rover
3. Calculate paths for the rover to follow based on the commands given by the users

In order to achieve the aforementioned goals, it was important to establish connection between the Command module as a server, and other subsystems as clients. The ESP32 of the Control module is the main client that constantly communicates with the server to exchange commands and status of the rover. The energy module is another client and regularly sends data related to the battery of the rover. The received data are presented on a web application where users can check the current status of the rover and command the rover. In order to smoothly handle incoming data and render components that form the web application, the web application was divided into the server-side: backend, and the presentation side: frontend, written in Node js and React respectively.

### 3.2.2 Hardware Organisation

Unlike the other subsystems of the rover, the Command module does not require any hardware components except a laptop on which the web application would be running. Therefore, the main focus was put on choices of tools for building the web application and communication methods between the other submodules and the backend, and between the backend and the frontend of the web application.

### 3.2.3 Initial Design Choices

For building a website, the Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS) would be used. The two technologies are chosen because the HTML allows programmers to describe the structure of the website in the form of document while CSS offers tools to style the individual components of the website. With CSS, it would also be possible to create graphics which may visualise data in a more sophisticated way. React js, the language in which the frontend of the website is written, allows rendering of HTML pages which will form the web application. The website will consist of various graphical user interface in order to enhance the usability.

The backend and the frontend of the web application are written in different languages and are separated. Therefore, a reliable connection between the backend and the frontend during the rover's operation is necessary for the web application. On the web application, users would interact with the frontend only while path calculations and communication with the rover would be done on the backend. For this reason, certain data have to be shared between the server and application side. Since the web application directly interacts with users, an application layer protocol is suitable for communication

between the backend and the frontend. There are multiple application layer protocols available for this purpose such as the Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telecommunication Network (TELNET). Since the HTTP protocol allows clients to send a request to the server and get data as a response, it was considered sufficient for implementing the connection between the backend and the frontend. With the web server that establishes connection with the web browser over the HTTP protocol and with the ESP32 over the TCP protocol, it would be possible for the Control module to know the user's input on the web application.

For the type of commands users can send through the web application, we decided to have 'Automated driving' and 'Manual driving'. With the 'Manual driving', the users would be able to directly command the rover to rotate by a specific angle or move a specific distance. For this command, the web server would need to securely pass the values from the website to the ESP32. With the 'Automated driving', the rover would be driven to find all the obstacles (5 ping pong balls of different colour) without direct supervision. In order to make this happen while preventing the rover from hitting any obstacle, it would be essential for the Command module to have access to data from the Vision module. Therefore, paths of the 'Automated driving' command would need to be calculated on the backend of the web application. It was decided for the website to have a page for commands where users can find text boxes to type in angle or distance value, and buttons to send the values for manual driving or command the rover to explore the area and look for obstacles.

While the rover moves according to the commands sent by the Command module, it would be convenient for users to see visualisation of the rover's movement and the position of any obstacles detected on the way. As a way to visualise these data, it was decided to display a map of the working area on the website. In order to do this, the map would need to be constructed as the rover moves and checks for any obstacles ahead of it. As there are no strict requirements on how to display the map, this functionality is up to design. The map was designed to be a giant canvas which consists of a large number of pixels. Then the rover's movement, the position of the rover and the obstacles would be expressed on the map by painting pixels following the x,y coordinates of the rover's position. Besides, the map can possibly be stored in the form of array which keeps track of pixels to be painted. Therefore, the website would have a page for the map as well.

## 3.3 Vision

#### Core Requirements

Within the context of the rover, the Vision module's main purpose is to **capture object data using the camera and relay the data to the Command module**. This involves taking the raw data from the camera sensor, converting from the Bayer pattern image to a readable RGB format and performing image processing to detect objects of interest and carrying out calculations to determine information about those objects and pass on that data to the Command module via Control. Therefore, this can be broken down into a few core requirements that are listed below:

1. Capture Raw Data from Camera hardware module connected via GPIO and convert it to RGB format for streaming
2. Detect objects of interest within the current frame, and draw a bounding box around them, namely 5 different coloured table tennis balls in a digital hardware system.
3. Send the pixel coordinates of the bounding boxes to a soft core for processing.
4. Calculate the location of objects relative to the FPGA camera based on the pixels of the bounding box.
5. Send the location data to the Control module for forwarding to the Command Module.
6. Send images at regular intervals of the live view of the camera over to Control for forwarding to Command for display on the web client.

### Hardware Organisation

The Vision module is comprised of two main hardware elements: the Terasic DE10-Lite, a cost-effective Intel MAX 10 based FPGA board [12] and the Terasic D8M-GPIO camera package that has a OV8865 image sensor [13] that interfaces with the FPGA through the onboard GPIO connectors.

These hardware choices were made by the project organisers, but are also sufficient and capable of carrying out the tasks at hand. As the FPGA's hardware is configurable, it is more flexible than other embedded systems that are limited to a general purpose processor,and is also able to handle both streaming and processing of high resolution images without significant compromises on framerate or data speed through the use of concurrent operations and dedicated blocks for signal processing applications like multiplication. This particular FPGA is also equipped with a 4-bit VGA output which is useful for debugging object detection live, and also has a connector for an Arduino Uno R3 shield, [12] which can be used to interface with the ESP32 used for Control.

### Initial Design Choices

There are several common approaches to object detection, with the two most prominent being the use of classic Computer Vision algorithms like the Canny edge detection technique and Hough transforms or the more recent use of convolutional neural networks (CNN) and deep learning to perform object detection through a combination of layers that perform convolution and pooling and a fully connected network at the end. According to [14], CNNs have in recent years become the de-facto standard for machine vision applications such as object classification, object detection and object segmentation; but as there is a desire for more performant and power-efficient systems, there has been a shift to using dedicated hardware tailored to the operations used for CNNs. There was a deep interest from the technical lead for the Vision module on this and attempts were made to implement this on the FPGA using various tools such as Intel's High Level Synthesis Compiler [15] and hls4ml - a recently developed python package for machine learning optimised high level synthesis [16] as well as a manual implementation of neurons using multiply-accumulate operations in SystemVerilog. However, there were far too many challenges for implementing a CNN within the current project with key issues being the system is a real-time system that processes video data pixel-by-pixel in a fixed order and in contrast, most CNNs for object detection working with multiple pixels in different locations simultaneously. A lot of previous work on the subject is also on FPGAs being used as accelerators for CNNs running on hard cores rather than standalone embedded systems as seen in [14] , which is the use case here. The field in itself is also still in development and is quite niche, with most work being done and implemented by experienced researchers. The only feasible neural network that could be applied in this period of time would be a non-convolutional fully connected layer network that just acted based on the individual colours of each pixel and possibly the temporal location, but there are other more straightforward methods and equally effective methods for object detection at a pixel level. These challenges are also faced when trying to implement popular traditional computer vision algorithms on the FPGA, as the access of pixels is limited, and the number of pixels that can be stored is also limited by memory constraints on the FPGA.

It was therefore decided that a pixel by pixel approach would be taken, using techniques based on thresholding methods from approaches to image segmentation. This would involve setting thresholds for each desired colour of the ping-pong ball and detecting large areas that met the threshold to differentiate a ball from its surroundings.

The provided ping-pong balls came in 5 different colours - pink, orange, light blue, grey and green. As these colours cover a wide range of the colour spectrum and even overlap with each other (orange and pink, blue and grey), it was thought to be necessary to do some preprocessing on the incoming image from the camera to ensure optimal colour detection for object detection. As the channel spectrum overlap in the RGB spectrum for different colours was significant, the decision was made to apply a transform from the RGB (Red, Green, Blue) colour space to the HSV (Hue, Saturation, Value) colour space. HSV has been shown to be a theoretically better colour space for image processing tasks like colour segmentation in road sign detection for autonomous driving as it is invariant to illumination changes unlike the RGB space.[17]

In order to perform general purpose operations like calculating the location of the detected objects, configure camera settings, and to provide a debugging interface, the decision was made to instantiate

a Nios® II/f "Fast" soft core on the FPGA. This would also allow for a quicker development cycle as the compilation of Quartus projects is upwards of 6 minutes, but C files can be changed and compiled in seconds. But as the Nios® II is limited in processing power and also in available memory, it is not capable of performing the tasks for the actual detection of the objects. Alternatively, to implement more advanced image processing algorithms or to reduce other hardware components in the system like the multiple Arduinos, a FPGA with a hard core connected via PCIe, known as a FPGA System-On-Chip (FPGA SoC) [18] could be used, which would provide the advantages of having reconfigurable hardware, and a more capable general purpose processor and a higher bandwidth limit for communications between the general purpose software core and the FPGA's hardware.

## 3.4 Drive

The purpose of the Driving module is the following:

1. Make the rover able to move forward, backward, rotate clockwise and anticlockwise.
2. Measure the distance that the rover has covered.
3. Program the rover to move or rotate for a specific distance or angle, respectively.
4. Regulate the voltage and the current that the SMPS generates to the DC motors in order to control the speed of the rover.

### 3.4.1 Initial Design

First and foremost, the most important was to make the rover able to move. Thus, one of the main tasks was to find a way to regulate the voltages that are being delivered to the DC motors. In other words, find a way to control the speed of the rover. Initially, the idea of controlling the speed using numbers from 1-10 came up. This was controversial, because for low and high speeds this would work perfectly, but for intermediate speeds this might have been confusing. For example, if 3/10 was sent to the rover as a speed, then the user would not exactly know how fast the rover would move. One other, idea was to use km/h or mph but for the low speeds that the rover is moving is again confusing. That is why the idea of the 5 speed levels was found, that can be seen in the implementation part analytically.

Moreover, one more potential problem was how to make the rover rotate clockwise and anticlockwise accurately for a specific angle. One potential solution was to convert the cartesian coordinates that the optical flow sensor measures, to polar coordinates. The issue with that solution was that some angles had the same tangent, or it did not exist at all.

In addition, the need for voltage regulation was important. Specifically, the rover has to "generate" the desired voltage for every command (forward, backward, rotation) and also keep the voltage stable in order to maintain the speed and the direction that the rover is having. For that reason, the PID controller had to be tuned perfectly in order to achieve the accuracy of the rover.

Finally, there was a thought of designing a program that could assist the rover if it diverged from its path, but by having a very good voltage regulation and by programming the optical flow sensor to measure the distance covered, precisely, it was not needed.

## 3.5 Energy

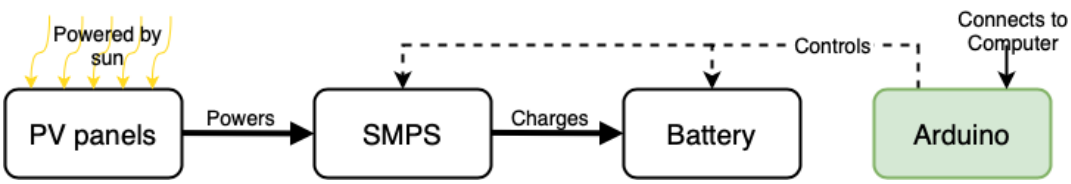


Figure 3: Functional diagram of the energy submodule.

The energy submodule must:

1. Design a battery pack and charge it using solar power.
2. Track SOC data and use it to estimate the range of the rover.
3. Track SOH data and complete necessary SOH maintenance.
4. Integrate into the main data mailbox.
5. Consider how to physically integrate with the rest of the rover.

### 3.5.1 Characterising Components

The energy system consists of three main components: the battery cells, the PV panels and the SMPS. To ensure that appropriate design choices are made, it is necessary to determine the behaviour and limitations of these components. As the characteristics of the SMPS have already been thoroughly studied in the Power lab[19], they will not be reexamined here.

#### Battery Cells

To determine their behaviour, each cell was tracked through a full charge/discharge cycle using the provided "Battery_Charge_Cycle_Logged_V1.1.ino" code[20]. Plotting the obtained data, all cells were found to have similar graphs for voltage versus time. Figure 4 shows the voltage evolution of a battery cell for a full discharge/charge cycle.

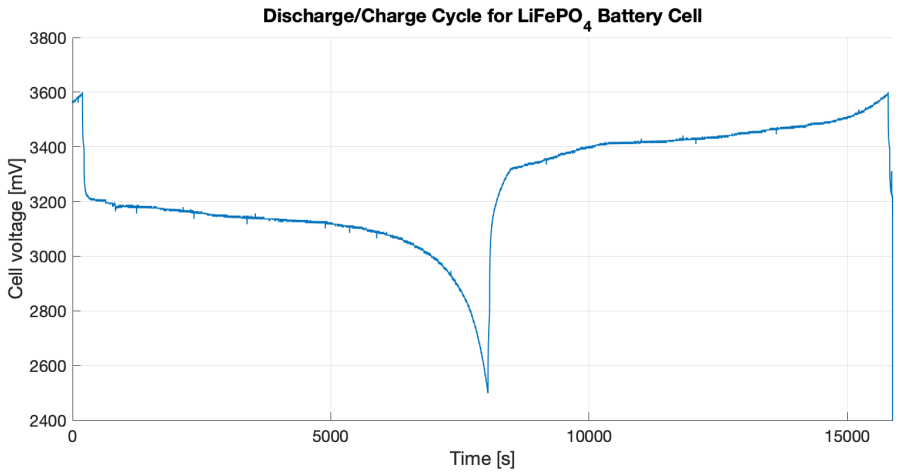


Figure 4: The voltage evolution of a battery cell through a full discharge/charge cycle. The cell is fully charged at 200s and 16000s and fully discharged at 8000s.

In addition to logging the cell voltage, the provided charging algorithm also logs the charging current. By integrating said current for a full charge or discharge section we can determine a cell's capacity in mAh. The results from performing this analysis on each cell are shown in table 1:

| Cell Number | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Capacity (mAh) | 542.7 | 526.1 | 519.5 | 530.1 | 543.7 |

Table 1: Cell capacities

As expected from the datasheet[21] all cells have a capacity of about 500 mAh. However, some cells have a higher capacity than others, which may have implications for the performance for certain battery configurations.

### PV panels

The PV panels are rated for a maximum power of 1.15 W at a voltage of 5.0 V and current 230 mA. Away from the maximum power point, the performance of the panels can be determined from their IV curves. To find the IV curves, each panel was connected on the B-side of the SMPS operating in non-synchronous boost. They were then lit by the lamp and the duty cycle of the SMPS was swept while measurements of panel current and voltage were taken. The resulting data was processed and is plotted in Figure 5.

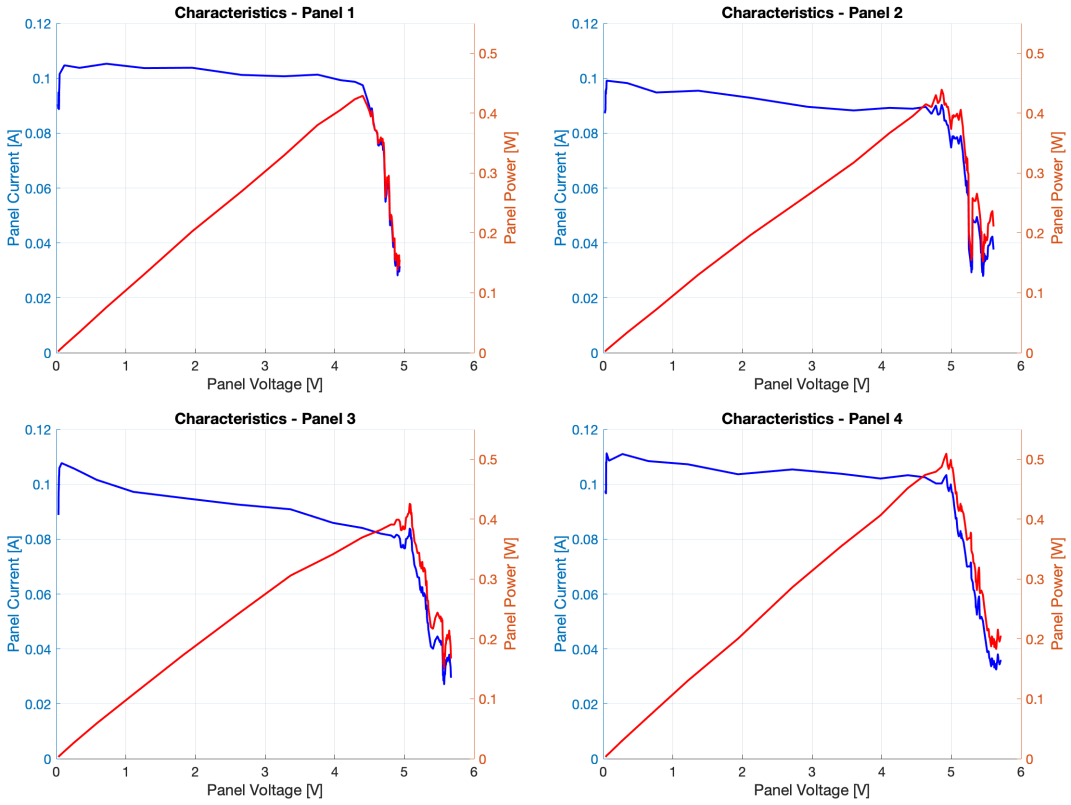


Figure 5: IV curves for the PV panels.

Even though the data is noisy, it is clear that all panels exhibit the standard IV characteristics of a PV cell. That is, they behave as non-ideal current sources with a nearly constant current at low voltages and a rapid current reduction at high voltages[22]. Moreover, it is clear that the provided lamp activates the panels poorly as the peak power for each of the panels is only ∼0.5 W.

### 3.5.2 Initial Design

#### Battery Configuration

When designing the battery pack there are two principal choices that need to be made. Firstly, how many cells should be in the battery pack and secondly, in what manner should these cells be connected?

The optimal number of cells is in large part set by the power and energy needs of other submodules. During testing the total power draw of the rover was found to be about 2 W. Each battery cell has a nominal voltage of 3.2 V and a maximum peak discharge current of 1 A [21], giving a maximum power draw of 3.2 W per cell. Thus, even with only a single cell the power requirements of the rover are met. However, each cell can only store about $3.2\,\mathrm{V} * 0.5\,\mathrm{A} * 3600\,\mathrm{s} \approx 5760\,\mathrm{J}$, which would only power the rover for 48 minutes. It is desirable to have the rover be operational for as many hours as possible each day. Assuming that 12 hours a day are completely without sunlight it is clear that the rover cannot work through the night even if all 5 available battery cells are used. To give the rover the most operational hours we would then want to use all 5 battery cells. However, connecting 5 battery boards to the

Arduino would use at least 11 of the 12 free Arduino pins, leaving only one pin for all other purposes. As such it might be wise to not use all of the available cells in the battery pack.

Now consider the PV array. Each PV panel is rated for 1.15 W, meaning that the array as a whole should have a peak power of 4.6 W. The peak efficiency of the SMPS in boost mode is about 80% [19] giving a maximum usable power on the SMPS output of $0.8 * 4.6\,\mathrm{W} \approx 3.7\,\mathrm{W}$. Assuming 12 hours of sunlight in a day, this means that the PV array produces less energy each day than the rover uses in 24 hours. It is therefore of high priority to capture as much of the solar power as possible. The battery cells have a standard charging current of 250 mA [21]. The power needed to charge the battery pack at this current based on the number of cells is shown in table 2.

| Number of cells | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Nominal charging power (W) | 0.8 | 1.6 | 2.4 | 3.2 | 4.0 |
| Peak charging power (W) | 0.9 | 1.8 | 2.7 | 3.6 | 4.5 |

Table 2: Power needed to charge at standard charging current at nominal cell voltage of 3.2 V and peak cell voltage of 3.6 V

From the table we see that a battery pack using 4 cells has the highest power consumption without going over the limit of 3.7 W. Moreover, 4 cells is a good compromise between having enough free Arduino pins and providing enough energy for long operation. As such, a 4 cell battery pack seems like a good choice. Moreover, as cells 1, 2, 4 and 5 were found to have the highest capacity, these are the cells that will be used.

There are several ways to connect the 4 cells into a power pack, however the design brief advised against mixing parallel and series connections[23]. As such only fully series and parallel battery packs will be considered.

A series battery pack has some obvious disadvantages compared to a parallel battery pack. Firstly, for a series battery pack the battery capacity is limited by the weakest cell. This is not the case for a parallel battery pack as it is possible to draw different currents from each cell. As a consequence, a series battery pack is able to store less usable energy than an equivalent parallel battery pack. Moreover, to check the OCV of each cell they need to be switched out of circuit using the battery board relay. For a series battery pack this leads to an open circuit and any charging/discharging must halt while the voltage is measured. For a parallel battery pack on the other hand, switching the relay of a cell only takes that one cell out of circuit and it is still possible to charge/discharge while measuring cell voltages. Finally, a parallel battery pack is self-balancing[24], while series cells need to be manually balanced using, for example, balancing resistors. This not only is more complex, but also leads to energy being lost as heat during balancing.

There is however a major weakness to a parallel battery pack: it is very hard to track the current into individual cells. The current sensor on the SMPS can only measure the current flowing into the battery pack as a whole and there is no way to know how the current splits between cells. To prevent over-current one must therefore operate with the assumption that all the current can flow into a single cell. Each cell is only rated for a max charging current of 500 mA[21], which is therefore the maximum charging current. At 500 mA the nominal charge power is only $3.2\,\mathrm{V} * 0.5\,\mathrm{A} = 1.6\,\mathrm{W}$, which is less than half the available solar power and with a parallel battery pack a lot of power will therefore go unused. As previously determined, energy is a precious resource and only being able to use half of the available solar power is such a large drawback, that despite the disadvantages of a series battery pack it is still deemed the best option.

### PV Array Configuration

There are four ways in which the four PV panels can feasibly be connected. The possible arrangements are shown in Figure 6.

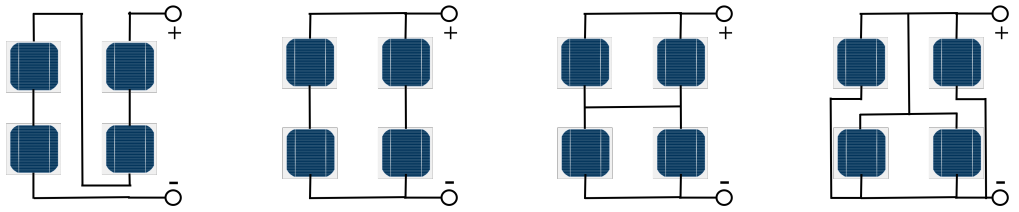

Figure 6: From left to right: Series (S), Series-Parallel (SP), Total-Cross-Tied (TCT), Parallel (P)

Consider first a pure series connection. During characterisation each PV panel was found to have a max voltage of ∼5.5 V. In a series connection the array voltage will then be 20+ V. The nominal voltage of the series battery pack is $4 * 3.2\,\mathrm{V} = 12.8\,\mathrm{V}$. As the array voltage is higher than the battery voltage, the SMPS must be used in the buck configuration. However, the maximum buck input voltage is only 7 V[25] and therefore a series connected PV array cannot be used. Similarly, for the Series-Parallel and Total-Cross-Tied arrangements the maximum array voltage will be about 11 V. However, as the battery voltage can swing between 10 V and 14.4 in a charge cycle, neither a buck nor a boost configuration will be able be able to provide the necessary voltage range with an 11 V input voltages. Thus it is not possible to use either the Series-Parallel or Total-Cross-Tied configuration. This leaves a purely parallel connected PV array as the only viable option, which is why it has been chosen.

### Maximum Power Point Tracking

The energy submodule only has access to a single SMPS device. At any time it will therefore only be possible to either perform MPPT or have the PV power be outputted at the correct current/voltage. This is not a problem, as the goal of the PV array is not to output the maximum amount of power, but simply to provide the power demanded by the charging algorithm. As such, the system does not need conventional MPPT. However, if the PV panels cannot provide the demanded power some sort of power tracking must be used.

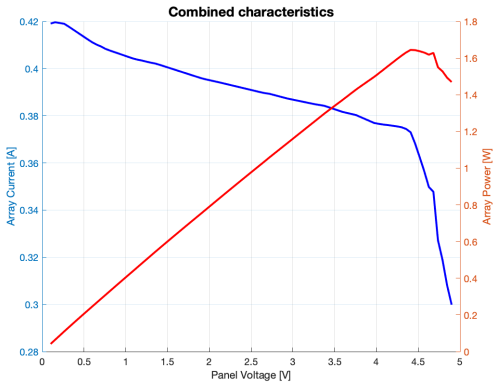


Figure 7: IV-curve (blue) and power curve (red) of the parallel configured PV array lit by a lamp.

Consider an SMPS being used to charge a battery pack at a set current. If the actual current on the output is lower than the setpoint, one would attempt to increase the output current by increasing the output voltage. This is achieved by increasing the duty cycle. Similarly, if the output current is too high one would attempt to lower the output current by lowering the duty cycle. From these considerations we see that increasing and decreasing the duty cycle is associated with higher and lower output power respectively. Now compare this with the IV and power characteristics of the parallel PV array shown in Figure 7. For an SMPS, increasing the duty cycle will lower the input resistance, causing the input current to increase. As increasing the duty cycle increases the output power so too must increased input current lead to increased input power for an equilibrium to exist. However, increased input current only gives increased output current if the PV panels are operating in the region to the right of the maximum power point. Thus this is the region one would want the panels to operate in.

### State of Charge

The state of charge (SOC) of a battery is defined as the remaining usable charge given as a percentage of the battery's total charge capacity[26]. There are many methods for estimating the SOC of a battery, the most common of which rely on measurements of the voltage and/or current of the battery[27]. The perhaps simplest SOC estimation method is to measure the open circuit voltage (OCV) of the battery and then calculate the SOC using a formula or a lookup table. For many types of battery this is a good estimation method. An example is lead-acid batteries, for which the OCV varies approximately linearly with the SOC. However, as was shown figure 4, this is not the case for LiFePO4 batteries. For LiFePO4

cells the voltage is nearly constant for a majority of each charge/discharge cycle. Any measurement error or change in OCV due to current recently flowing through the battery, would therefore produce large SOC estimation errors. This holds true for all SOC methods relying purely on voltage measurements and they are therefore not good alternatives.

An alternative SOC estimation method is Coulomb counting, where the current flowing through the battery is integrated to find the net charge that has left or entered the battery. Seeing as charge is a conserved quantity nearly all charge put into the battery will be available during discharge. As such, the SOC will vary nearly perfectly linearly with the integrated current. The sources of error for Coulomb counting are mainly the Coulombic efficiency of the batteries and current measurement errors. However, using correction methods the error can be kept small, on the order of 1-2%[28]. Given the simplicity of the estimation method, this is a very small error. Other estimation methods, such as Kalman filters and neural networks, are claimed to give higher estimation accuracies[27]. However given the already high accuracy of Coulomb counting the improvement is marginal. Moreover, they are far more complex both computationally and in implementation. As such, Coulomb counting was deemed the best option for SOC estimation.

### State of Health

The state of health of a battery is a measure of its current condition and performance compared to when it was new[29]. Indicators of a battery's state of health include battery charge capacity, energy capacity, cell voltage balance, and the number of completed charge/discharge cycles[30]. Over the course of its lifetime the SOH of a battery will naturally degrade. However, through SOH maintenance the degradation can be slowed significantly. Most importantly for a series battery pack is to keep the battery cells balanced, as unbalanced battery cells lead to lower capacity and faster cell degradation[31]. To facilitate balancing, each of the provided battery boards have mounted resistors which through a MOSFET can be connected to the battery cell. During operation one must decide when to switch said resistors on and off to keep the cells balanced. Usually, balancing is only done towards the end of a charge cycle. There are several reasons for this. Firstly, passive balancing requires energy to be expended and will therefore reduce the total amount of usable energy in a battery if done during discharging. Secondly, differences in impedance and charge curves between cells might make it look as though a cell is charged more than others, but the voltage difference might disappear naturally as the battery is charged more or as charge current is reduced towards the end of a charge cycle.

### Integration of the Energy Submodule

It is not necessary to physically integrate the energy module with the full rover. However, if it were, the energy module could either be integrated as a charging station or be mounted directly on the rover itself. The advantage of a charging station is that the battery is only ever charged or discharged at a given time. This makes it easier to track current and power and leads to simpler charging and SOC algorithms. A drawback of using a charging station is a reduced range, as the rover always needs to get back to the charging station before its battery is depleted. Moreover, if the rover is detached it is difficult for the energy module to track the cell voltages and SOC during discharging. This could be fixed by using the microcontroller of another subsystem to track the battery while the rover is not connected to the charging station. However, a likely simpler solution is to mount the energy module on the rover. This would increase range, but does require the battery to charge and discharge at the same time. To be able to track the battery current, it would then be necessary to collect current and power data from other submodules. This data could be relayed through control and read in on the energy Arduino through UART.

The rover has four separate voltage regions. The battery and PV array have already been discussed. In addition there is a 5 V node used to power the FPGA and microcontrollers, and a variable voltage node used to power the motors. This power must originate in the battery and as such the 5 V and motor voltages must be connected to the battery through voltage converters. The obvious implementation is to use two switch mode power supplies in buck mode, one to provide 5 V and one to power the motors. One problem exists however with this solution: The 10.0 – 14.4 V of the battery is higher than the SMPS maximum input voltage[19]. A way to solve this would be to exchange the PMOS on the SMPS board for another NMOS, which would increase the maximum input voltage.

# 4 Implementation

## 4.1 Control

### 4.1.1 Command Communication

The first connection that had to be established was the one between Command and Control. From the requirements set out, it was clear that the ESP32 had to use it's WiFi capabilities to interface with Command. The two options available to the ESP32 are as follows: to act as either it's own router through the Access Point mode or it's own station under an existing router through the Stationary mode [32]. As the Access Point mode would mean that the rover is not connected to the internet, since it would create a private inaccessible network, the stationary mode was chosen instead. Having the capability to communicate with the rover through an existing router allows for communication from anywhere in the world through WiFi. This would be much more flexible and representative of communicating with a rover on Mars. A working prototype for communication was built using TCP. In order to test whether it was functioning, since Command was still in very early stages, a python-based TCP server was created. When running, it would take the text put into the terminal and send it to the ESP32 while displaying the messages incoming from the ESP32. The ESP32 was configured to echo back all messages. Through this simple system, the connection was successfully implemented by debugging the connection and the messages being sent back and forth. Once the Command web server was nearing completion, this process was tested, with commands being sent from the server and data being sent from the ESP32. As prior individual testing on either side was extremely effective, there was little to no extra fixes required for this connection to function as intended.

### 4.1.2 Drive Communication

As the first physical connection between the ESP32 and a peer microcontroller, it was decided that the best course of action would be to completely test communication with the simplest of devices, the Arduino Nano Every. Since the Arduino has an extensive and well-documented library for UART communication, Drive was the best subsystem to test out initial communication. When testing out the UART communication, two methods documented through the Arduino IDE were used. SoftwareSerial [33], which uses code to imitate a UART connection allowing you to map the peripheral to any GPIO pin, or directly assigning the pins through hardware assignment. Initially the former was utilised, but it was quickly realised that the multiplexing feature of the ESP32 chip allows for hardware configuration to most of the GPIO pins of your own choice [6]. However, an interesting interaction that was recorded was that the 3 UART connections that are available to the ESP32 aren't completely unused [34]. One of the UART connections is used to communicate with the laptop when connected via USB. This UART connection by default is assigned to pins 0_RX and 1_TX on the Arduino Adapter Board. As a result, mapping a secondary UART on these same pins and serially writing to your laptop would also write to the peripheral connected to 0_RX and 1_TX. Once this was determined, these two pins were discarded as options for communication. UART between Drive and Control was tested with some movement functionality on Drive and test code on the ESP32 allowing for commands to be sent by the terminal of the Serial Monitor through the UART connection between the laptop and the ESP32. This was then forwarded to Drive. This allowed for imitating the entire chain of communication from Command to Drive, but by only using the Control subsystem and ESP32 as a testing device.

### 4.1.3 Vision Communication

As described in the functional requirements for Control, the initial design was to set up two SPI connections with the FPGA due to the overabundance of pins between the two devices. SPI as a peripheral was superior as well since it would take less overhead for the FPGA and would be a faster method for communication when compared to UART [8]. Alongside this, due to the ESP32 being dual-core , a task scheduler [35] could be used to simultaneously support image transfer while standard data was being transferred. Initially, to test out the SPI capabilities of the ESP32, the ESP32 was set up as an SPI controller, while an Arduino was set up as an SPI peripheral. Through SPI.h, a library designed for the ESP32 and ESP8266 [9], this was accomplished and communication between the two devices was

successful. It was discovered that although the Nios® II processor supported acting as an SPI controller, acting as an SPI peripheral was not well-supported. This proved problematic when instantiating an SPI peripheral using the Nios® II, even if the FPGA itself supported it. Although there is an immense amount of support for the ESP32 on the Arduino, when it comes to acting as an SPI peripheral, this has yet to be implemented from the Espressif documentation in C. Although SPI peripheral libraries have been created for the older ESP8266, this unfortunately has not extended to the ESP32. There have been some projects which attempted to develop and use their own SPI peripheral functionality. This is done by translating and porting over the libraries from the original Espressif documentation into Arduino, setting them up as Arduino libraries. After attempting to rework these libraries [36] [37]so that they could support the functionality we needed, it was eventually discarded as an option. Even if the ESP32 could function successfully as an SPI controller, it wouldn't matter since the FPGA had the same limitation as it only had SPI controller functionality. Eventually it was agreed that this would have to change to UART for standard data transmission. As for image transfer, since this was meant to be done through SPI, a framework for the ESP32 acting as an SPI controller to transfer these images was created. However, as it proved too difficult to use the FPGA as a functioning SPI peripheral, this also had to be scrapped. In the end the only connection which remained between Vision and Control was the singular UART port designed for collecting Vision data to send to Command for mapping and detecting obstacles.

### 4.1.4 Integrating Communication

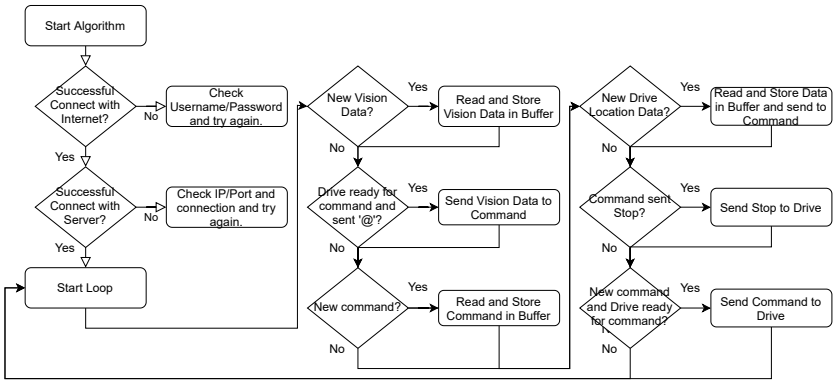


Figure 8: Flowchart describing ESP32 Data Transmission Algorithm

Although the communication between peer devices was successfully set up and tested, there was still a small challenge. To be able to incorporate all communication such that it could occur simultaneously and effectively, with as limited of a delay as possible. In order to achieve this, it was noted that the UART and TCP communication between devices are always held within internal buffers and as such there is no need to immediately pull out the data as it arrives. A secondary set of buffers were created, except these were created to store the next line of data or next command depending on the data stream. To understand what was considered a line of data or command, there was discussion between the groups. This allowed us to dedicate certain characters as "start" and "end" characters to differentiate between data sets. Also, certain subsystems would send special characters to the ESP32 to designate their status. For example, Drive would send an '@' if it was prepared for its next command. Command would send a 'S' if it required the Drive subsystem to stop it's current command. Unlike a complete path, which had the program collect the data from one device and immediately send it to another, this would allow the program for a better scheduling of tasks to be able to simultaneously keep up data collection. Once a certain buffer had the required contents to be sent, a boolean would evaluate to true, activating the sending loop to transmit the contents of the buffer and then clear it. In the end, this would allow for staggered sending and receiving between subsystems, giving the illusion that communication is simultaneous even when all subsystems are transmitting across the ESP32. For ease of understanding in what order the processes were occurring, the UART dedicated to Energy was used as a debugging tool, sending data to the Serial Monitor to understand how the data was flowing to and from the ESP32.

## 4.2 Command

### 4.2.1 Communication with the ESP32

For communication with the ESP32, the backend of the web application acts as a host which the ESP32 can connect to via a TCP connection. The connection was implemented using the "net" Javascript module [38]. Similarly, the Energy module connected to the Command module via the TCP protocol and sent data.

### 4.2.2 Communication between the Frontend and the Backend

Initially the HTTP protocol was decided to be the communication method between the backend and the frontend of the web application. However, some drawbacks of the HTTP protocol were identified during the implementation of the web application. The HTTP protocol is uni-directional. [39] This means that the communication can only be initialised from one side. And the other side is not able to communicate as long as it does not receive connection incoming. In case of the HTTP protocol, clients, which is the web browser in this project, may establish connection to the web server and initialise communication. Then the web server receives the request coming from the web browser and sends data back as a response. In the initial design of the web application, the frontend could send a HTTP request to the web server when the website is opened by a user. And the web server could send an HTML document back for the frontend to show on the website. Another use of the HTTP protocol was delivering user's input to the backend. When a button is pressed or a text box is filled by a user, the frontend could notify the backend the value of the user's input which could lead the web server to send command to the Control module via TCP protocol. However, for the frontend to visualise data from the other subsystems it had to gain access to the data which were available on the backend of the web application. The HTTP protocol was not the best communication method for this case because the server could not send these data without a request from the frontend. Also forwarding specific variables only and not the entire HTML document was very demanding with the HTTP protocol.

Therefore, it was decided to use WebSocket instead. The biggest advantages of using WebSocket were that the connection between the server and the client was persistent and both server and client could initialise communication.

```
var website;
const io = require('socket.io')(server);
io.on('connection', (socket) => {
    console.log('website client connected');
    website = socket; // storing website client
    socket.once('disconnect', () => {
        console.log('website client disconnected');
    });
    socket.on('Angle', data => {
        console.log("Manual driving. Angle received on server: %s", data);
        if(esp32 !== "undefined") {
            esp32.write("![R" + data);
        }
        socket.emit('AngleDistance', [data,0]);
    });
```

```
import io from 'socket.io-client';

const Socket = io.connect('http://localhost:3000', {});

export default Socket;
```

Figure 9: WebSocket code with Server Side on Top and Client Side Below

Figure 9 shows how WebSocket was implemented on the server side by using the "socket-io"

Javascript module. [40] The server function is triggered when a client connects to the server with a parameter "socket" which contains the socket where the server and the connecting client can communicate. Then the socket parameter could trigger functions which only react to specific event name. Figure 9 shows how the server handles the event "Angle" incoming from the frontend with data. The server understands that the incoming data is a command telling the rover to rotate by the given angle and sends the ESP32 of the Control module the corresponding rotation command that the Drive module understands. The variable "esp32" contains the TCP socket used to communicate with the Control module. As there is just one web browser connecting to the server, the socket was stored in a global variable "website" and allowed other parts of the server code to use it to send data to the frontend. On the frontend, it was important to ensure that it is securely connected to the backend through WebSocket throughout the entire usage time of the web application and that it reacts to incoming data from the server instantly and updates the webpages. Therefore, a const variable "Socket" which connects to the backend was exported to App which was a function rendered to show the website. The client side WebSocket implementation is shown in Figure 9. App decided which components are shown on each webpage so as the frontend receives data through WebSocket, App could pass the received data to a webpage where the data was visualised.

### 4.2.3 Website

The general structure of the website including the design of home page, navigation bar, and button were taken from a React Website Tutorial [41] as they were considered to be suitable for building the rover's command system. The home page of the website is shown in Figure 10. The navigation bar which can always be found at the top of the website is convenient for the users to move to another page on the website. When one of the buttons on the navigation bar is pressed, the frontend shows the corresponding page. There are four pages on the website which are "Home", "Power", "View" and "Command". Additionally, the "Get started" button directs the Command page when it is pressed.

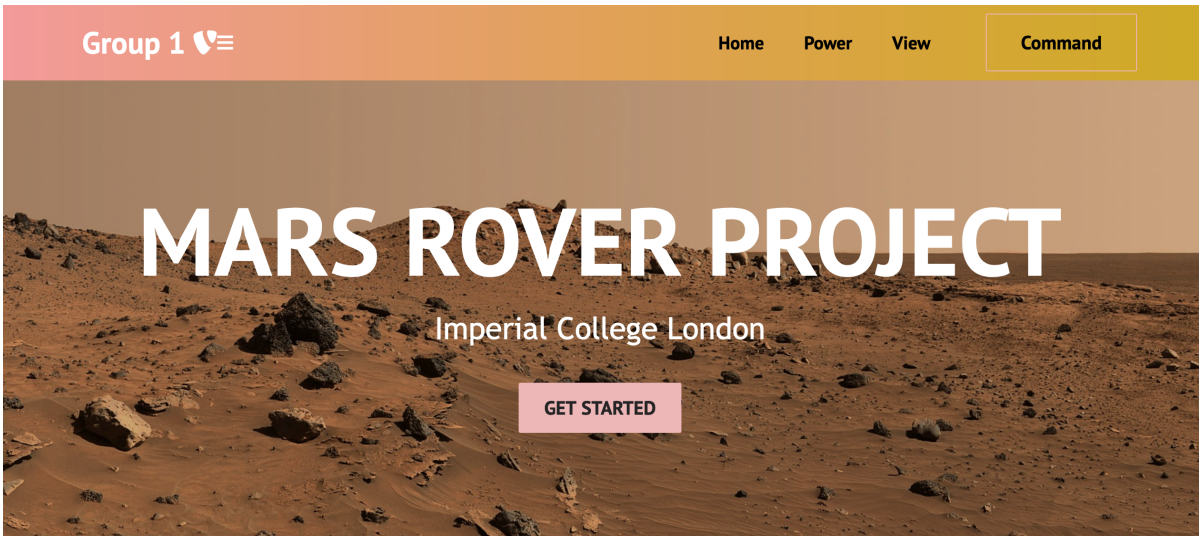


Figure 10: Home Page on the website

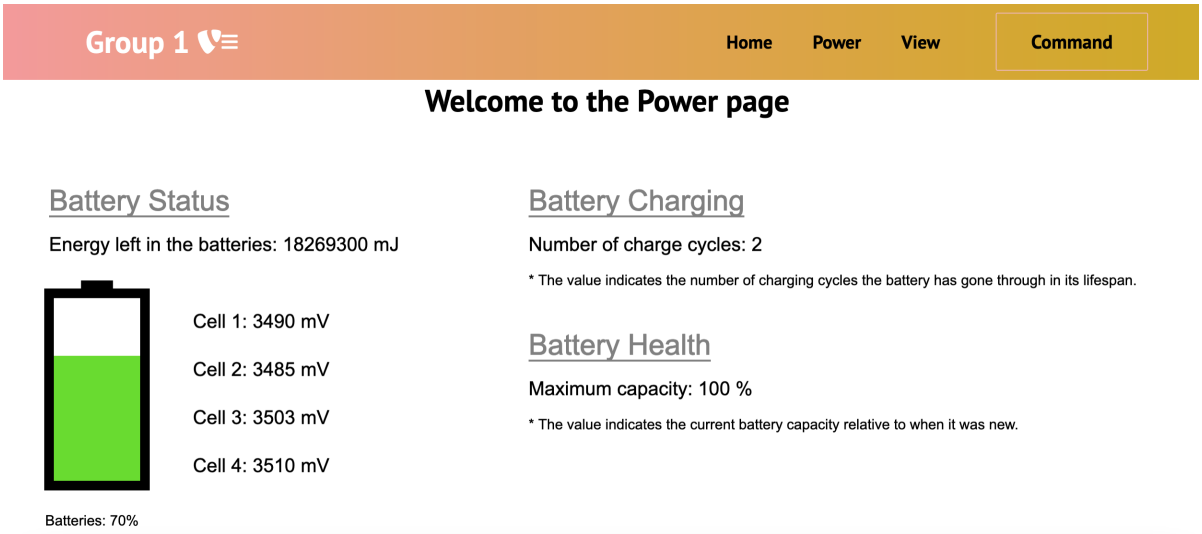


Figure 11: Power Page on the website

The Power page shows the rover's energy status as shown in Figure 11. There are three subsections on this page which show the status, charging and health of the batteries relatively. The data represented are received from the Energy module which regularly sends data to the backend via TCP protocol. Then the data were sent to the frontend via WebSocket.[40] To enhance the readability of the data represented, a graphic in the form of battery was created. It shows how much energy is left in the batteries with respect to the amount of energy when batteries are fully charged. Additionally for the battery health, the ratio of the current battery pack capacity to the maximum battery pack capacity is shown. The Power page was designed using the "react-konva" Javascript module [42] which will be further discussed in the following explanation of the View page.

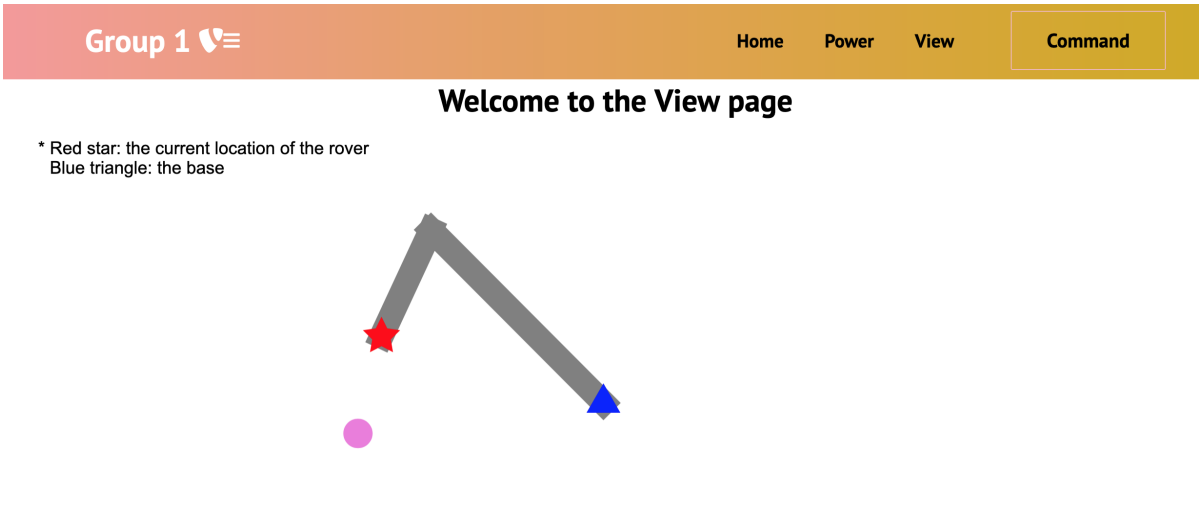


Figure 12: View Page on the website

The View page, shown in Figure 12 contains a map of the rover's working area. The map was initially planned to be drawn with pixels as mentioned in the Initial Design Choices section of this report. However, some drawbacks of this method were identified. A map drawn by painting pixels was not the best at representing rotation of the rover. It is possible to represent rotation of angles which are multiples of 90 degrees but the other angles showed difficulties. Therefore, it was decided to draw the map using the "react-konva" Javascript module instead. [42] In order to draw the map, various data are required. This includes the positions of discovered obstacles from the Vision module and the commands sent to the Drive module. Figure 12 shows an example map where the rover discovered an

obstacle (pink pingpong ball) after moving according to a set of commands given to it. As stated on the top left corner of the View page, the red star indicates the current position of the rover while the blue triangle indicates the position of the base which is the starting point of the rover. The set of commands given to the rover was 1) rotate 135 degrees clockwise 2) move forward 250 mm 3) rotate 110 degrees anti-clockwise 4) move forward 120 mm. After executing the fourth command, the rover discovered a pink pingpong ball and this vision data was sent to the backend of the Command module through the Control module. The backend informs the frontend about the colour and position of the discovered obstacle via WebSocket and the frontend draws the obstacle on the map according to the vision data. The area passed by the rover is represented by drawing grey rectangles on the map.

```
function paths(angleDistance, color) {
  var height = Number(roverWidth) + Number(angleDistance[1]);
  var finalAngle = Number((Number(global_angle)+Number(angleDistance[0]))%360);
  if(angleDistance[1] !== 0) {
    roverPath.push(<Rect x={currPos[0]} y={currPos[1]} width={roverWidth} height={height}
      fill={color} rotation={finalAngle} offsetX={roverWidth/2} offsetY={roverWidth/2}/>);
  }
  var sine = Math.sin(Math.PI/180*finalAngle);
  var cosine = Math.cos(Math.PI/180*finalAngle);
  currPos = [Number(currPos[0]) - Number(angleDistance[1])*sine,
             Number(currPos[1]) + Number(angleDistance[1])*cosine];
  currPos_backend = [Number(currPos_backend[0]) - Number(angleDistance[1])*sine,
             Number(currPos_backend[1]) + Number(angleDistance[1])*cosine];
  global_angle = Number(finalAngle);
}
```

Figure 13: Code for Drawing Paths on the Map

Figure 13 shows the function used to draw the grey rectangles and represent the rover's movement. To accurately represent the rover's movement on the map, several global variables that represent the rover's status were created and updated constantly. Such variables are the current position (an array of length 2 that contains x and y coordinates) of the rover on the map, the angle the rover has rotated relative to the beginning and an array of grey rectangles to be drawn on the map are saved as currPos, global_angle, roverPath respectively. The "paths" function in Figure 13 is called when the frontend receives a new command data that the rover has completed. The first parameter "angleDistance" is an array that contains an angle and a distance command that the rover has completed. The second parameter "color" determines the grey of the rectangle drawn to represent the rover's movement and it's fixed to be "grey" when the function is called. The height of each grey rectangle drawn should accurately represent the distance travelled by the rover. Therefore, it is set to be the sum of the rover's height and the distance travelled. On the map, the rover is treated as a square so the rover's width is used for its height. Then the global_angle is updated by adding the angle command sent to the frontend. Since a rotation of 360 degrees does not make a difference on the map, modulo 360 is done. Then if the distance travelled was not zero, the rectangle is created as shown in Figure 13 with the calculated values and added to "roverPath". The offset values are added to make the point of rotation to be the centre of the rover as it was the top left corner of the shape by default. Finally the current

```
function drawObstacle(obstacle, pos, angle) {
  var color = "black";
  switch(obstacle[0]) {
    case "pink":
      color = "#eb7bdd";
      break
    case "green":
      color = "#97df62";
      break
    case "blue":
      color = "#8bcbd4";
      break
    case "orange":
      color = "fc9031";
      break
    case "grey":
      color = "1c1e1b";
      break
    default:
      //
  };
  var xpos = obstacle[1][0]*Math.cos(angle*Math.PI/180)
             - obstacle[1][1]*Math.sin(angle*Math.PI/180);
  var ypos = obstacle[1][0]*Math.sin(angle*Math.PI/180)
             + obstacle[1][1]*Math.cos(angle*Math.PI/180);
  obstacleSet.push(<Circle x={pos[0]+xpos} y={pos[1]+ypos}
                   radius={15} fill={color}/>);
}
```

Figure 14: Code for Drawing Obstacles on the Map

position of the rover on the map is updated for the next rectangle. Figure 14 shows the function used to draw discovered obstacles on the map. The first parameter "obstacle" is the vision data from the backend. It is an array where the first element is a string representing the colour of the obstacle and the second element is an array that contains the x,y coordinates of the ball relative to the rover's camera. The second parameter "pos" is the current position of the rover which is used to determine the obstacle's position on the map since the coordinates of the obstacle given are relative to the rover's current position. The third parameter "angle" is "global_angle" explained in the function that draws rover's movement. It represents the total angle the rover has rotated since beginning. The function first chooses the correct hex code which determines the colour of the obstacle that would be drawn on the map. Then the function calculates the x,y coordinates of the obstacle with respect to the rover's current position. The difference between these coordinates to the coordinates given in the parameter "obstacle" is that these coordinates take the rotated angle into consideration. This calculation is necessary because even though the obstacle is equally placed 5 cm in front of the rover for example, the position of the ball when the rover is facing south and when the rover is facing east will be different. Finally, with the calculated position, a circle of the chosen colour is added to an array called "obstacleSet". The arrays "roverPath" and "obstacleSet" are sent to the "react-konva" canvas and the added shapes are drawn.

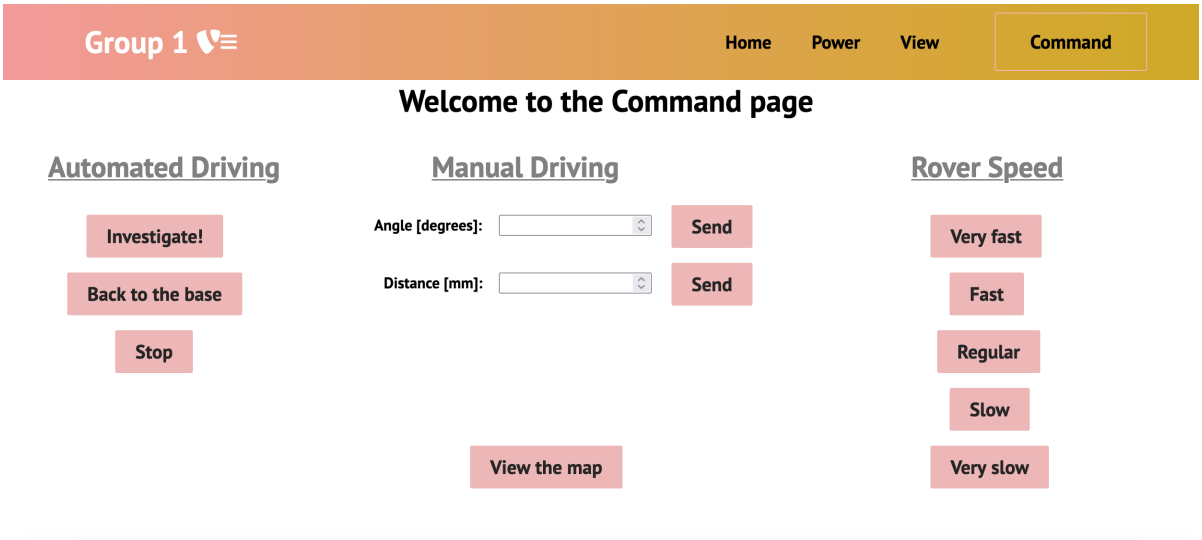


Figure 15: Command Page on the website

The Command page, shown in Figure 15, contains three sections: Automated Driving, Manual Driving and Rover Speed. Under the Manual Driving section, there are two text boxes where the user can input an angle or a distance and the rover would follow the command when the user presses the send button next to the text box. Under the Rover Speed section, there are five buttons which represent the five speed profiles that the rover has. By default, the rover would start with the regular speed but the user can change the speed by pressing these buttons. When the user presses a button, data corresponding to the button will be sent to the backend via WebSocket and the backend will send command to the ESP32 accordingly. Under the Automated Driving section, there are three buttons: "Investigate", "Back to the base" and "Stop". When the user presses the "Investigate" button, the backend will command the rover to drive until it finds all the obstacles. During this operation, the user can command the rover to stop or go back to the base.

### 4.2.4 Automated Driving

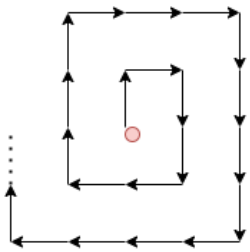

Figure 16: Deterministic Path for Automated Driving

```
var map = createMap(step, vision);
var graph = new Graph(map[0]);
var start = graph.grid[Math.ceil(map[0][0].length/2)-1][0];
var distanceTravelled = 0;
if(map[1]) { // if destination is blocked
    if(distanceToTravelLeft < map[2]) {
        distanceTravelled = map[3]; // distance until just before obstacle
    } else {
        distanceTravelled = map[2]; // distance until right after obstacle
    }
} else {
    distanceTravelled = step;
}
 var end = graph.grid[Math.ceil(map[0][0].length/2)-1][distanceTravelled];
 command = convert2command(astar.astar.search(graph, start, end));
```

Figure 17: Automated Driving code

Generally, the web server would command the rover to follow a deterministic path until it finds all the obstacles. The deterministic path is shown in Figure 16, the red circle indicates the starting position of the rover and the arrows represent the rover's movement. When there are no obstacles on the way, the rover can just follow this path. However, there is a problem when one or more obstacles are blocking the rover's path since the rover should be able to avoid hitting the obstacles. In order to solve this problem, a pathfinding algorithm A* [43] was used to find the shortest path to the destination that the rover can follow without hitting the obstacles. The A* algorithm was chosen because it compares estimated cost of path to the destination at every step to find the shortest path and with the help of a heuristic function of the A* algorithm, the pathfinding process can be done efficiently. [43] The A* algorithm was implemented using the "javascript-astar" Javascript module [44]. In order to use this algorithm, it was necessary to create a graph which can indicate positions where the rover can go and cannot go (blocked by obstacles). Figure 17 shows a part of the function used for automated driving. This function is called repeatedly after the rover moves "one step" indicated by an arrow in Figure 16. The function creates a map required for the A* algorithm considering the distance until the destination, stored in parameter "step", and the position of any obstacle ahead of the rover. The created map is a 2-dimensional array filled with 1's for positions where the rover can go and 0's for positions where the obstacle was blocking. Then the start and end position on the created map were specified. The map was created in a way that the starting position is always the bottom centre on the map. The end position was tricker to determine. If the obstacle is not blocking the destination, which is distance of "step" away from the start point, then the end position can simply be destination. However, if the obstacle is blocking the destination, then a decision should be made whether the rover should stop before the obstacle or pass the obstacle. As shown in Figure 17, it was implemented so that the rover passes the obstacle if the next command is continuously moving forward (indicated by "distanceToTravelLeft" being bigger than the distance required to pass the obstacle. If the rover's next command is rotation of 90 degrees, then the rover would move forward and stop just before the obstacle. Since there is a limitation on hardware that the rover cannot identify obstacles which are ahead of the rover by distance less than rover's width, the rover's width was added to the obstacle's width when the function draws a graph for the A* algorithm. This way, the rover can avoid hitting obstacles that are located near it.

## 4.3 Drive

#### General Specifications

Initially, the main goal of the driving subsystem was to make a movable rover. The specifications and the limitations of the SMPS, Arduino and the DC motors, needed to be known. The first challenge was to find a way to measure the rotations per minute (RPM) of the DC motors, precisely. The solution was to stick a small piece of paper on the rotating axes of the DC motor. Additionally, one screwdriver was placed perpendicularly to the small piece of paper. This was done because each time the DC motor completed one whole rotation the small piece of paper would "trigger" the screwdriver and generate a distinctive sound. Finally, the only unsolved part was how to pick up that sound. So, the microphone of one set of earphones was placed close enough to the DC motor in order to pick up that distinctive sound. Then, the whole process was recorded, leading to the following outstanding outcome. The recorded sound was processed with an audio application and every time the DC motor completed a whole period the waveform of the recorded sound had a spike with a recognisable magnitude. Finally,

the time difference between one spike to the following one was the period (T) of the DC motor. Then only the conversion from time to RPM was left and in order to find the RPM, the following formula was used: $RPM = \frac{60}{T}$

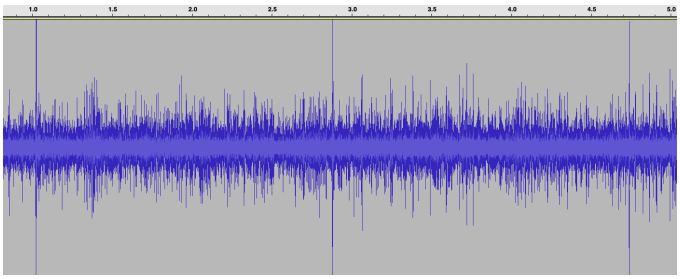

Figure 18: Shows the spikes that correspond to a period(T)

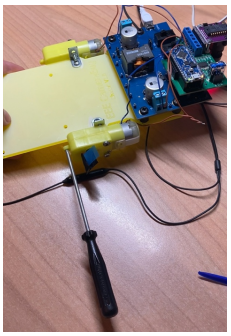

Figure 19: Equipment Setup for Measuring RPM

In the following table some basic specifications have been measured:

| ***Input*** | ***Output*** |
|---|---|
| Voltage ≈ 5V | Minimum Operating Voltage: 1.384V<br>Maximum Arduino's Voltage for accurate conversion: 4.096V |
| Minimum Current: 0.134A<br>Maximum Current: 0.22A | Minimum Operating Current: 0.035A<br>Maximum Arduino's Current for accurate conversion: 0.060A |

**Speed Levels**

One of the main goals was to make the user interface with the rover relatively simple and that is why the idea of introducing 5 speed levels came up. Specifically, the idea behind those 5 speed levels, is to avoid using km/h or mph because the range of km/h is really small (0.17km/h-0.62km/h) and the range for the mph is even smaller (0.11mph-0.38mph) so, it will confuse the user. Thus, one way to simplify the speed range is to set 5 speed levels that can be understood easily and be user friendly as well. Additionally, it is more convenient and straightforward as it does not require further knowledge (voltages, RPM).

In order to find the true range of voltages the rover's limits needs to be found. The code that is responsible for the voltage range has a reference voltage of 4.096V. The value 4.096, is the Arduino's upper threshold for accurate conversion. On the other hand, the buck SMPS can generate up to 4.8V ( due to the power losses it cannot generate exactly 5V). If the Arduino's reference voltage is increased then if, for example, a 2.5V reference voltage is sent to the rover, it would not "generate" 2.5V as an input for the DC motors. This happens because the Arduino can handle only up to 4.096V and if this value is exceeded then the conversion will not be accurate. On the other hand, the lower limit is 1.46V because everything beyond that will not have enough power to move the rover. To conclude, the voltage range is from 1.46V to 4.096V in order to have a movable and an accurate rover. So, that is why the speed levels are divided like that. Those 5 speed levels can easily be "corresponded" to a specific voltage, and a specific voltage can easily be "corresponded" to a specific velocity. After the the RPM were found then, they were converted to speed values by the following formula:

$$v\left(\frac{km}{hr}\right) = r \times \frac{2\pi}{60} \times N\,(rpm) \times 3.6$$

where r is the radius of the wheel and is equal to 3.4cm and the number 3.6 is used to convert m/s to km/h.

| Speed Level | RPM | km/h | mph | Output Voltage(v) | Input Voltage (V) | Input Current (A) | Pin(W) |
|---|---|---|---|---|---|---|---|
| **Very fast** | 48.58 | 0.6227 | 0.3869 | 4.05 | 4.95 | 0.216 | 1.0692 |
| **Fast** | 41.27 | 0.5289 | 0.3287 | 3.4 | 4.96 | 0.192 | 0.95232 |
| **Regular** | 32.38 | 0.4150 | 0.2579 | 2.75 | 4.97 | 0.162 | 0.80514 |
| **Slow** | 23.92 | 0.3066 | 0.1905 | 2.1 | 4.98 | 0.152 | 0.75696 |
| **Very Slow** | 13.89 | 0.1780 | 0.1106 | 1.46 | 4.99 | 0.134 | 0.66866 |

One interesting observation is that the relationship between the voltage and the RPM is almost linear. This is confirmed in Figure 20

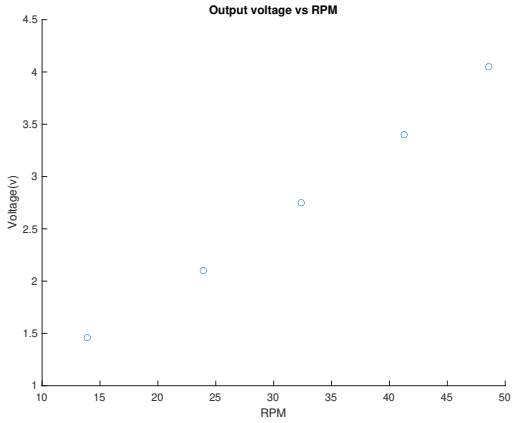


Figure 20: Output Voltage vs RPM

### Commands

After finding the specifications the next step was to make a movable rover. Initially, one crucial task to do was to simplify and "connect" the two sample codes. To do that, there was a need for the use of an object oriented approach which would make it feasible to combine functions of the two sample codes into two different objects. In this case classes were used to implement a rover object and an optical flow sensor object. This allows to use the complicated code of the optical flow sensor in a simple way inside the rover functions.After, making the code simple and creating classes that would make the testing straightforward, 2 major changes were made to the code. Firstly, even though the control knob was really helpful to measure some key data, for the project it was needed to be able to set output voltages independently of the control knob and that is why it was not used. Specifically, the voltage of the DC motors is now being controlled by the user with those 5 speed levels. Secondly, the optical flow sensor was connected directly to the code that is responsible for the movement of the rover. In other words, the rover was able to move backward and forward by just setting inside a while loop the desired distance and comparing it with the measurements of the optical flow sensor. For example, the desired distance is 20cm, the rover will be moving forward or backward until the measurements read from the optical flow sensor for the y axes are approximately equal to 20cm.

```
void Rover:: move_forward(float rover_speed, int distance) {
  digitalWrite(pwmr, HIGH);       //setting the motor on or off
  digitalWrite(pwml, HIGH);       //setting left motor on or off
  total_x = 0; // x axes measured in mm
  total_y = 0;// y axes measured in mm
  total_x1 = 0;// x axes measured in inches
  total_y1 = 0;// y axes measured in inches
  int total_y_int = total_y * 100; // *100, for accuracy
  while (total_y_int < distance * 100) { // As long as total_y_int is less the desired distance*100 the rover is moving forward
    vref = rover_speed; // the rover's voltage is set here
    this->VoltageRegulationStep(); // The PI controller regulates the voltage
    this->SetState(HIGH, LOW); // the clockwise/anticlockwise rotation of the motors is set
```

Figure 21: Forward Command Code

Additionally, to make the rover rotate clockwise and anticlockwise the following process was implemented: Assuming that the centre of the rover (the centre of the imaginary axes that connects the two DC motors) is not rotating and it remains in the same position while the main body is rotating, then it can concluded that the rover is tracing an arc. The imaginary arc with the body of the rover is forming a circular sector. The radius of the imaginary circular sector is the length from the centre of the optical flow sensor to the centre of the imaginary axes that connects the DC motors. The mathematical formula is given as follows **arc length** $= \theta r$, where $\theta$ is the angle in radians and $r$ is the radius. So, to implement this, as seen in Figure 22 a while loop in the code was implemented to make the rover to rotate as long as the traced path of the optical flow sensor is less than the arc length needed to be traced to cover a specific angle. For example, if the desired angle is 90° or $\frac{\pi}{2}$ the rover will be rotating until the measured path of the optical flow sensor (the x value that is being measured from the optical flow sensor) is equal to the angle times the radius. The radius is approximately 168 mm but in the code it has been adjusted in order the rotation command to be accurate.

```
void Rover:: rotate_clockwise(int deg) {
  digitalWrite(pwmr, HIGH); //setting right motor on/off
  digitalWrite(pwml, HIGH); //setting left motor speed on/off
  this->SetState(LOW, LOW);// setting the clockwise/anticlockwise rotation
  float radian = float((deg * 71) / 4068.0);//converting degrees to radians
   total_x = 0; // x axes measured in mm, inital value=0
  total_y = 0;// y axes measured in mm, initial value=0
  total_x1 = 0;// x axes measured in inches, initial value=0
  total_y1 = 0;// y axes measured in inches,initial value=0
  float r = 175 ;// radius=length from the centre of the OFS to the rotating axes
  float arc = radian * r;// calculating the arc length
  float d = 0; // setting distance =0
  float fabs_arc = float(fabs(arc - d));// the desired distance is subtracted from the arc length measured from the OFS
  vref = 2;// is the speed of the rotation
  int l = fabs_arc * 100;// *100 for accuracy
  while (l > 200) { // the rover is rotating as long as l is greater from 2mm(tollerance)*100 (for accuracy)
    float d = abs(total_x);// d is equal to the measuremets of x axes from the OFS
    float fabs_arc = fabs(arc - d);// fabs_arc is being recalculated while the rover is rotating
    l = fabs_arc * 100;//*100 for accuracy
    vref = 1.6;// The rotating speed is being set
    this->VoltageRegulationStep();// The PI controller regulates the voltage
```

Figure 22: Clockwise Command Code

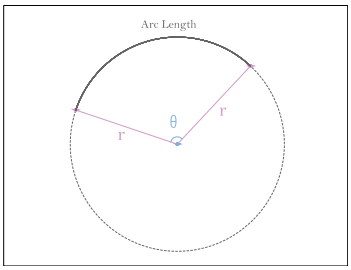


Figure 23: The arc that the optical flow sensor is tracing

### Controller

The voltage controller had some pre-set values. Using the trial and error method, initially, only one of the three variables of the PID controller was changed at a time while the others remained set. Regarding the derivative controller, it made the system unstable because it made it oscillating and inaccurate. So, the best choice was to set the Derivative controller equal to 0. Additionally, the combination of changes in the PI controller values decreased the settling time, the rise time, the overshoot and the steady state error. As it can be seen in the graph bellow, the difference between the initial and the final PID values is significant. The overshoot has been eliminated, the rise and the settling time have been improved dramatically and the steady state error has an acceptable value. Furthermore, it is worth mentioning that if the ratio $\frac{ki}{kp}$ is great enough (larger than 8) then the response of the system becomes almost perfect. To conclude, by using only a PI controller and by setting P=2.6 and I=27 the rover has a stable driving system that does not take a lot of time to stabilise its reference voltage (in other words, its reference speed), and most importantly, it does not overshoot and oscillate before it takes its reference value. These findings can be confirmed by the graph shown in Figure 24

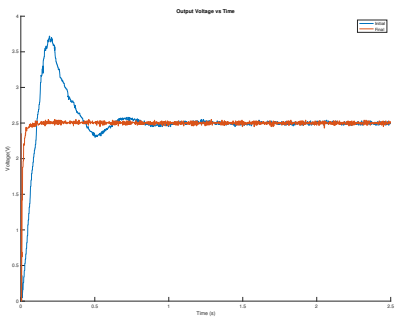

Figure 24: Initial vs Final PI controller

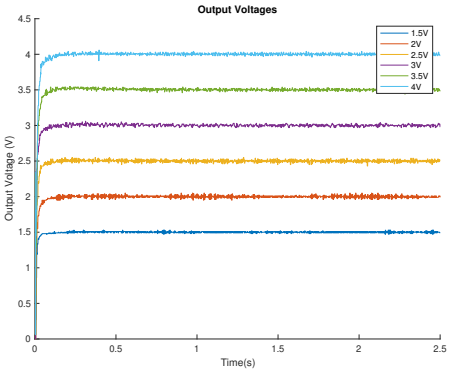


Figure 25: Output Voltages for all Reference Voltages

The difference in the system's response is obvious as the system with the final PI values almost instantly reaches its reference value. The table below is showing the performance of the two PI controllers:

| | Overshoot(%) | Rise time(s) | Settling time(s) | Steady State Error(% ) |
|---|---|---|---|---|
| Initial | 48 | 0.143 | 1.25 | 1.8 |
| Final | - | 0.0136 | 0.12 | 1.6 |

As it can be seen in Figure 24, using the previous PI values the system responds perfectly for all the reference voltages. Also, in the table below are given the steady state error for every voltage value. It is really important that the system after it has reached its reference value, it does not fluctuate.

| Voltages(V) | 1.5 | 2 | 2.5 | 3 | 3.5 | 4 |
|---|---|---|---|---|---|---|
| Steady State Error | 1.3 | 2 | 1.6 | 1 | 1 | 0.75 |

Regarding the Current Controller, it was not tuned because the voltage controller interacts with it. Thus, after further testing, making changes to the PI current controller gains was not necessary. Finally, the current instantly takes the desired value and it does not oscillate.

## 4.4 Vision

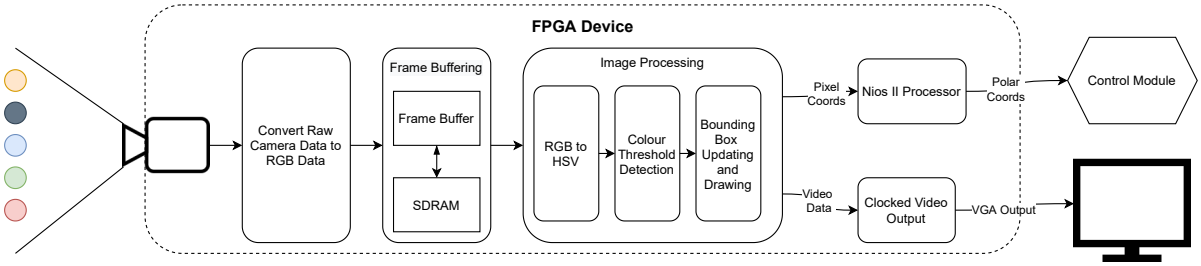


Figure 26: Image Processing Pipeline

The purpose of the Vision module as aforementioned was to find different coloured ping-pong balls and draw bounding boxes around them to find their estimated location, the desired outcome is shown in Figure 27. The following section is organised in order of the video processing stream: first the raw data to RGB data, then the transformation to the HSV colour space, followed by the object detection, then finally the calculation for the estimated distance of each ball. The operation is summarised by the diagram in Figure 26.

Figure 27: Desired Outcome of Image Processing applied in Vision module

### 4.4.1 Image Capture & Processing Stream

The image capture and buffering is based on a starter project provided by Terasic Inc for the D8M Camera module that was modified by Dr Edward Stott [45] and provided to us for this project. The system makes use of several IP components from the Intel Video and Image Processing Suite, namely the Clocked Video Output and Frame Buffer Core. The system is compromised of several modules, that take the raw data from the camera in Bayer filter form[13], transform it into RGB video packets with a resolution of 640*480, buffer the frames to allow camera and output to run at different frame rates, process the image data and convert the video data to serial data for output over VGA.[45] The components are shown in Figure 26. As this system and its corresponding documentation is openly available, the implementation outside of the image processor will not be described in more depth, interested readers can consult the referenced repository and starter project for more information.

### 4.4.2 Image Pre-Processing Module - RGB to HSV Conversion

HSV is traditionally expressed as three values ranging between 0-360° for hue, and two values between 0 and 1 for saturation and value.[46] As floating point calculations in SystemVerilog are computationally expensive, and also introduce a layer of unnecessary complexity, the conversion was adjusted to express all three values in terms of an 8 bit integer ranging from 0-255. The algorithm for conversion from 8-bit RGB integers is presented mathematically:

$$M = \max(R, G, B)$$

$$m = \min(R, G, B)$$

$$C = M - m$$

$$H = \begin{cases} M = R, & 0 + \frac{43 \times (G-B)}{C} \\ M = G, & 85 + \frac{43 \times (B-R)}{C} \\ M = B, & 171 + \frac{43 \times (R-G)}{C} \\ M = 0, & 0 \\ C = 0, & 0 \end{cases}$$

$$S = \begin{cases} M = 0, & 0 \\ C = 0, & 0 \\ \text{else} & \frac{255 \times C}{V} \end{cases}$$

$$V = M$$

Although the inclusion of a divide operation was undesirable as it does not translate well to hardware implementations, it was unavoidable with hue being obtained by dividing the difference of two components by the chroma value. This is a result of HSV being a non-linear transform from HSV.

The YUV colour space was also considered as a option for conversion as it also separates colour into 3 components - one for luma and two chrominance components. This would also make it invariant to illumination changes. The transform from RGB to HSV is also more suited to hardware as it is a linear transform requiring only multiplication. However, as it cannot represent the full RGB spectrum and is limited in what colours it can represent, the decision was made to use the HSV.

#### SystemVerilog Implementation

The division in this case has a 16 bit numerator and an 8 bit denominator, which would negatively affect the critical path if implemented as a single cycle/combinatorial divider, causing timing failures in the image processing stream as the processor would not be able to keep up with the input speed of the camera and the output speed necessary to produce a VGA output. As the provided Intel LPM Division FPGA IP Core has a configurable pipeline latency, the decision was taken to pipeline this division process in order to maintain the clock speed. By gradually increase the pipeline and testing the critical path delay, the decision was made to use a 5 cycle latency for the divider, which allowed the FPGA's image processing module to run at 100MHz. As there are two divide operations for each HSV calculation, one for H and one for S, two dividers were instantiated with different parameters to account for the differences in the calculations as H values require signed values and S values do not.

In order to maintain the video stream and process one pixel per cycle, this conversion was implemented with a 6 stage cycle that took in a new RGB value every cycle and produced a new HSV value every cycle. This included storing the maximum RGB value $M$ and difference values $C$ for each input for 6 cycles, to ensure that the data being used for comparison and output is correct. In order to use the pipeline, 12 separate dividers were instantiated, 6 for each type. Although this used a lot of logic elements and hardware, it comes at the benefit of being able to maintain the clock speed and optimise the processor for speed.

### 4.4.3 Image Object Detection Algorithm

The object detection algorithm was designed using a colour segmentation method, as the processor receives each RGB pixel from the image processing stream, it converts the RGB values to HSV values. After the HSV value is obtained, each of the components are compared with a set of values that describe the HSV colour components of the objects that are being looked for. If a pixel meets the criteria for 1 of the 5 colours, it's location is recorded and marked out as a candidate pixel to be part of the object and it's bounding box.

#### Colour Threshold Determination

In order to determine the colour thresholds for detecting each of the different coloured balls, images from the camera were taken after adjusting the camera settings for gain, exposure and white balance to maximise the difference between the balls and the background as well as the balls and each other.

An image was then obtained from the camera by taking an image from the output VGA port. This image was then imported into a MATLAB tool from the Image Processing toolbox - the Colour Thresholder App. This application allows colour images to be segmented by changing thresholds in different colour channels, supporting RGB, HSV, YCbCR and L*a*b*. An example of the user interface for this tool is shown in Figure 28

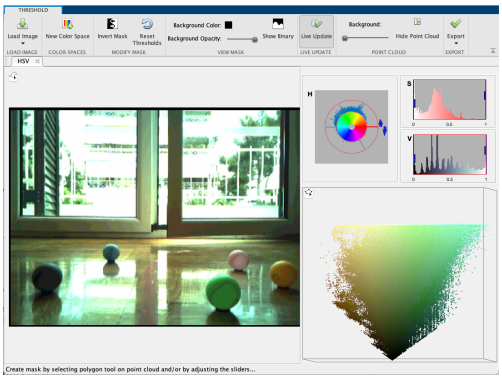

Figure 28: MATLAB Colour Thresholder Tool

Figure 29: Tool Produced Composite for Green Ball

As can be seen in Figure 29, the tool provides an accurate way to select the values for each of the HSV channels, as the binary mode can be used to ensure that the regions of interest are maximised and the other regions are ignored.

However, the accuracy of the colour threshold is not perfect for several reasons. Firstly, even though the HSV colour scheme is used to reduce the effect of illumination, the differences in light as the rover approaches the table-tennis balls from different angles can vary by a large amount, causing differences in the H and S channels, hence multiple pictures have to be used for this calibration step. This is also because the materials used for the construction of the ping pong balls have a specular reflectance that causes non-uniform colour dependent on the angle of the light. Secondly, as the pictures being used for calibration are from the 4-bit VGA output, the output image's colour spectrum is reduced compared to the colour that can be "seen" by the camera is the full 24-bit colour, that has much more depth and a wider range of colours than what is perceived in the output image. Additionally, as the data is obtained from the VGA output and captured using an application on a computer, there may be some processing that is carried out on the image, meaning that the image's data may be different in terms of RGB values from the raw RGB values that the camera is capturing.

#### Use of Pixel Location for Filtering

A purely colour based pixel approach to object detection has significant flaws unless the difference between the environment's colours and the object of interest's colours are significantly different from each other and the background's colour is plain and singular. As the testing environment was not

ideal, there would be other colours in the scene that would conflict with the objects causing inaccurate bounding boxes that were usually too large. In order to filter out these errors and glitches, the location of the pixel was taken into account.

Firstly, it was observed that as the testing environment was a flat surface, the location of the balls and hence the individual pixels would never be smaller than a certain value (since y values are counted from the top down with the top of the frame being the 0 value), so this was taken into account and bounding boxes were only updated on the following conditions: when a colour was detected AND the video packet was valid AND the pixel y-coordinate $> 280$. The value of 280 was chosen as objects near the horizon level at around 240 pixels would be very far away and could be easily mistaken.

This simple adjustment and addition to the object detection algorithm reduced false positives by a significant amount. However, it would not be useful if the testing environment included changes in gradient or if objects of interest were located on top of other objects.

Secondly, as there were still occasional inaccurate bounding boxes, conditions were implemented on the software running on the Nios® II processor to filter out boxes larger or smaller than certain values, and boxes that had a ratio that did not resemble a square.

### 4.4.4 Object Location Calculations

With the bounding boxes obtained from the Verilog module, the coordinates of the bounding box for each colour is sent to the Nios® II processor using a FIFO buffer. The calculations for how far away the object is and it's relative position in comparison to the camera module was are done on the Nios® II processor.

There are multiple ways to calculate the location of a 3D object in space on a plane with some ways being more precise than others. Some methods rely on using the size of a known object and comparing it's relative size in the camera, but the decision was made not to use this type of measurement as it was not possible to always guarantee that the size of the bounding box would enclose the entire ball, making the location calculation inaccurate. As the testing environment for the rover was ideal with the camera in a fixed position and the testing surface being completely level, the algorithm for the mapping between pixel coordinates and real world x-y coordinates relative to the camera was based on the following form:

$$x = A_x m^2 + B_x n^2 + C_x mn + D_x m + E_x n + F_x$$
$$y = A_y m^2 + B_y n^2 + C_y mn + D_y m + E_y n + F_y$$

This is a relatively simple quadratic mapping between the pixel coordinates and the real world coordinate system which is a good choice for this system as the Nios® II is limited in both processing power and available memory. By taking a picture of multiple balls in different locations and measuring the actual corresponding mapping distance and the pixel locations with a set of at least 6 measurements, the coefficients can be found by solving the matrix equation:

$$\begin{bmatrix} m_1^2 & n_1^2 & m_1 n_1 & m_1 & n_1 & 1 \\ m_2^2 & n_2^2 & m_2 n_2 & m_2 & n_2 & 1 \\ . & . & . & . & . & . \\ m_i^2 & n_i^2 & m_i n_i & m_i & n_i & 1 \\ . & . & . & . & . & . \\ m_N^2 & n_N^2 & m_N n_N & m_N & n_N & 1 \end{bmatrix} \cdot \begin{bmatrix} A_x & A_y \\ B_x & B_y \\ C_x & C_y \\ D_x & D_y \\ E_x & E_y \\ F_x & F_y \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \\ x_2 & x_2 \\ . & . \\ x_i & y_i \\ . & . \\ x_N & y_N \end{bmatrix}$$

where the pixel locations are represented by $(m, n)$ and the physical locations on the plane are represented by $(x, y)$. Initially 6 samples were taken to calculate the coefficients, but as more measurements were taken to verify the method, they were also added to the list of samples to improve the accuracy of the algorithm. It was noted during initial testing that balls at a further distance were located more accurately than balls closer together, the cause of this problem was because the calibration data had more data points at a further away distance, causing the mapping to prioritise accuracy there, this was then fixed by adding more data points for balls that were closer to the camera. These calculations for the coefficients were carried out in MATLAB, and then implemented in C to be run on the Nios® II processor.

The Nios® II processor does not include floating point hardware by default, but as this algorithm requires floating point calculations as some of the coefficients tend to be very small, and this algorithm is

run for each set of colours, it was prudent to include hardware support for single precision floating point adds and multiplications. This was done by adding the Nios® II Custom Floating Point Instruction module from Intel's IP catalogue for custom floating point instructions, which reduces the amount of cycles necessary to carry out a floating point multiplication to 4 cycles using a single instruction and a floating point add to 5 cycles[47]. If this hardware was not used, it would be implemented using software, and hence would take multiple instructions and more hence more cycles.

This method is satisfactory in its performance, with a worst case error of $\pm 5$ cm in difference between calculated position and actual position when the bounding box is accurately detected. For the rover's path finding algorithm, this is a value that is easily taken into account and as multiple measurements of the ball are taken, the actual position can be refined and more accurately determined by taking the average of multiple readings in the Command module in the server. This also solves the problem of when a ball is not fully in the frame of the camera or not fully detected by the object detector, as the rover is moving, all objects should eventually come into frame fully and not be cut off by the sides of the view or receive a better view of the object leading to a better bounding box and hence distance calculation. Additionally, this algorithm's performance is ideal in the current testing environment, but if tested on a surface with changing gradients or elevated objects, this algorithm would be inaccurate due to it's inability to take into account objects on the real world z-axis (i.e. the height of an object relative to the plane).

### 4.4.5 Image Transfer over SPI

To be able to send an image to the Command view for the user to view is an essential part of a Mars Rover's mission, thus it was one of our goals to do this as well. However, due to time limitations and other operations that took priority, there was not enough time to and memory available on the FPGA to implement it fully.

In order to facilitate this function, a custom SystemVerilog module was designed specifically for sending an image via SPI. The image parameters as aforementioned are $640 * 480$ at 60fps with 8 bits of colour for each channel. This requires 921,600 bytes per frame which is just under a 1kb per frame. Usually, for sending image data, before the data transfer happens, compression is applied on the sender's side before compression, but as memory and processing power is limited, it is not possible to implement any sort of encoding onboard the FPGA and the only option is to offload it to either the ESP32 or the web server for encoding and display.

The module's structure is shown in Figure 30. It acts as an SPI peripheral that provides data whenever it is read from by an SPI controller, with pixel values being stored in a FIFO buffer that is written into by the image processing module. 26 bits are stored within the FIFO with the first two bytes used to indicate a start of frame or an empty FIFO, this indicates to the SPI controller the position of the image and also when to stop asserting the sck to indicate that transmission should stop. The FIFO used is a dual-clock FIFO that uses the main clock as the input clock and the sck provided by the SPI controller is used as the clock for the read.

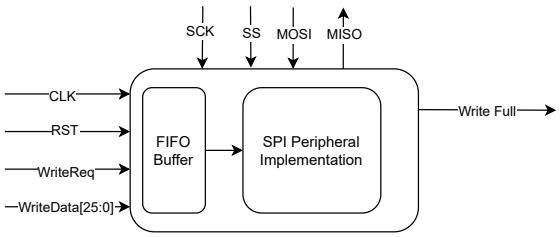


Figure 30: SPI Image Transfer Module Block Diagram

### 4.4.6 Evaluation of Image Processing

The algorithm applied was simple, but adequate for our use. By filtering out poor results, it has a low false positive rate. But as it is purely colour based, glitches do occur. Improvements that could be made include applying a noise filter for better low light performance, or an edge detection filter to combine colour detection with edge detection. Attempts were made to do this using a 3x3 kernel filter and the Filter IP provided by Intel, but due to limited memory constraints (See Appendix 1 & 2), it was not possible. If there was more time, we would try to reduce the memory usage of other components,

such as by switching to SPI communications or to reduce the size of the FIFOs in the design and the frame buffer.

## 4.5 Integration

The purpose of Integration is to ensure and test the work of all the subsystems together. The following section is organised in a way that shows the progress of the system and all the testing and measurements done to allow the delivery of this working system. The good communication and will to work hard that was shared by all the team members was a crucial factor that led to the completion of the tasks.

### 4.5.1 Drive

Having an accurate and reliable driving system was the first important milestone for this project as it is the means of the Rover to navigate its environment. The first test done to ensure that was measuring fixed distances traversed by the Rover and calibrating the optical flow sensor using the sample code. The testing procedure was fairly simple but produces consistent results. By using some fixed tiles on the floor that have a dimension of $25cm \times 25cm$, I could measure how accurate the Rover moves and rotates and also minimizing human errors that would arise if the reference distance and angle was not in place. The first problem that arose was an error in distances greater than $20cm$ where there was a sudden sign change in the measured distance of the sensor.

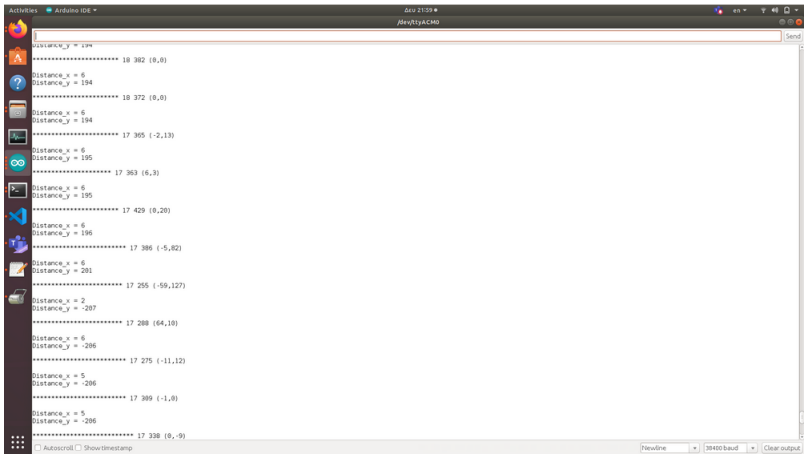


Figure 31: Serial Monitor output showing the overflow of distance.

As it can be seen in Figure 31 the distance measured went from $201mm$ to $-207mm$ which indicated a bug. I communicated my findings with the member that was in charge of Driving and we concluded that it is an overflow caused by the multiplication operations of the code which can be easily fixed by prioritising divisions.

After having received the fixed code and continuing testing, it was concluded that the floor material affects the optical flow sensor's accuracy greatly. Trial and Error showed that the best floor material in the area of testing operations was wood which gave the most reliable readings. Having a proof of concept is crucial when deciding to move to more complex operations for the system and during the initial testing of the Drive module, it was concluded that it had a good accuracy that would be essential for the later stage of mapping the balls.

When trying to measure considerably bigger distances with the Rover, it was identified that the strain from the cables connected could impose some considerable torques on the system which could result in inaccuracies. This could possibly be solved by implementing a PID that ensures no rotation while moving forward and backward, but after consultation with the appropriate team member it was decided that because the purpose of the cables is for debugging, capturing of data and supplying power, in a real-life scenario it would not affect the rover. For the demonstration and all the testing it was also

decided that the cables should be placed in positions that would cause minimal strain in the system.

After ensuring that the navigation of the Rover is accurate enough, the next logical step is to test and implement the communication procedure with the control module.

### 4.5.2 Communications between Drive and Control

Knowing that the team member in charge of Control only has the ESP32 board, it was crucial to establish a close contact so that the feedback from the Integration testing is relayed back and used to improve the Control code. Having a Rover that can navigate is important, but there needs to be a communications framework that will allow it to receive and execute commands. When researching the different communication protocols, it was concluded that UART is a reliable and good hardware device that allows the transmission and reception of data. After consulting with the team member of Control and receiving the available UART pins of the ESP32, the cable connections were made.

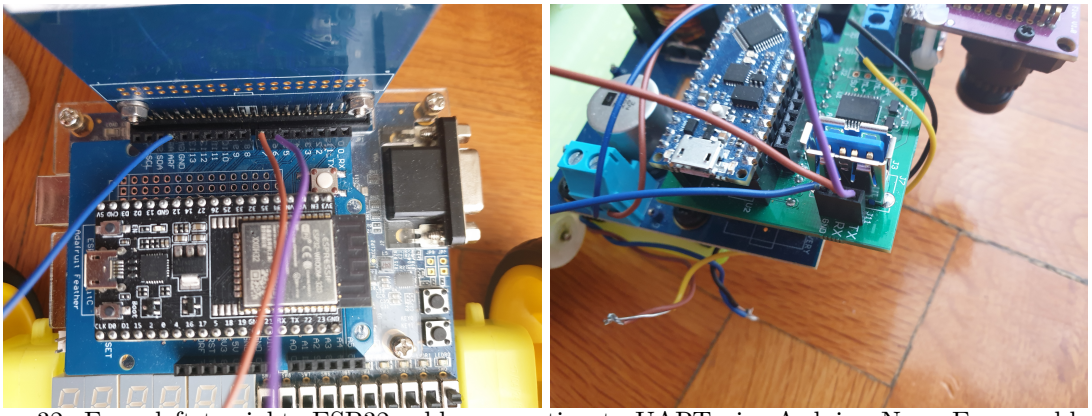

Figure 32: From left to right: ESP32 cable connection to UART pins,Arduino Nano Every cable connection to UART pins.

UART_test

```
void setup() {
  Serial1.begin(115200, SERIAL_8N1);
  Serial.begin(115200);
}

void loop() {

  if(Serial1.available()) {
    char data_rcvd = Serial1.read();
    Serial.println("Driving module received: ");
    Serial.println(data_rcvd);
  }

}
```

Figure 33: Test code to ensure UART functionality

To test the connection, a fairly simple program was made that reads the Serial buffer when it is available. To achieve reliable Integration of the systems, a modular approach is chosen where the testing of even the simplest components is of great importance. This is to confirm functionality and detect faulty hardware without having to debug complex programs that may or may not contain bugs. After ensuring UART functionality, a more complex program was implemented to allow driving to receive commands from Control.

Basic_Commands | optical_sensor.cpp | optical_sensor.h | rover_navigation.cpp | rover_navigation.h

```
//Process Command
int count = 0;
char parameter[32] = ""; //either distance or angle
int count_param = 0;
char command;
char sp;
for (int i = 0; i < 32; ++i) {
 if(i==0)
 {
  command = Command[i];
  if(command == 'M')// this character is for the move_forward and move_backward command
  {
    sp = Command[i+1]; // this character denotes the required speed of the rover (ranges from A-E)
    i++;
  }
  continue;
 }

  if (Command[i] == ',' || i == 31) { // comma seperates commands and i = 31 when we read the last character of the Command buffer
    if (command == 'M')
    {

      Serial.print("Distance is: "); Serial.println(parameter); //example
      float ret_speed = choose_speed(sp);
      if(atoi(parameter) > 0)
        rover.move_forward(ret_speed,atoi(parameter));// if the value is positive then move forward
      else
        rover.move_backward(ret_speed,abs(atoi(parameter)));// if it is negative then backward
    }
    else {
      Serial.print("Angle is: "); Serial.println(parameter);
       if(atoi(parameter) > 0)//if angle is positive rotate clockwise else anticlockwise
        rover.rotate_clockwise(atoi(parameter));
       else
        rover.rotate_anticlockwise(abs(atoi(parameter)));
    }
    count++;
    count_param = 0;
    strcpy(parameter, "");
    command = i!= 31 ? Command[i+1] : ' ';//read next command
    i = i+1;
```

Figure 34: Command decoder

The command structure was fairly simple and not resource heavy. The Rover does a series of rotations and movements. The starting character of a command is "[" which allows the system to identify the start of a command being transmitted by Control. To move the Rover a transmitted character should be $M$ and after that the speed should be identified using a character between "$A-E$" with "$A$" being the fastest available option and "$E$" the slowest. Because rotations are the only other option, an arbitrary character can be used accompanied with the angle of rotation. The code also handles signs denoting direction of rotation and movement. For example, "$[MA300, R-90]$" is a command that when executed by the Rover will result in a forward movement of $300mm$ and an anticlockwise rotation of 90 degrees. Lastly, to allow for flexibility of commands, a stop functionality was added when Control sent the character "$S$" while executing a movement command which stops the motors and transmits the distance covered back to Control.

```
if(Serial1.available()){
  char temp;
  temp = Serial1.read();
  if(temp == 'S')
  {
    digitalWrite(pwmr, LOW);       //stop motors
    digitalWrite(pwml, LOW);       //stop motors
    Serial1.write('Q');Serial1.write(int(total_y));
    break;//stop moving
  }
}
```

Figure 35: Stop functionality

### 4.5.3 Vision

When testing the image processing and the Vision module, the camera had to be recalibrated to match the testing environment. Initially, the only configurable parameters were the HSV thresholds as described in section 4.4.3, the exposure and the gain settings. This resulted in unsatisfactory performance as the testing environment was tinted yellow and there was no way of adjusting for that.

After researching the camera, and going through the OV8865's datasheet, [48] it was discovered that white balance - the gains of the Red, Green and Blue pixels could be individually manipulated and manually calibrated. By testing these values with trial and error in conjunction with the MATLAB Colour Threshold Tool, a set of gains were found that maximised the colour detection and reduced the number of false positives in the bounding box detection. This fixed the problem that had occurred due to the tint of the room.

It was also noted in initial testing that the corners of the camera were darker and dimmer than the rest of the image, leading to difficulties in detecting objects that were in the corner of the frame due to different HSV values. This is a commonly seen phenomenon known as vignetting. Although to the human eye, this may seem like a minor problem, for a computer vision algorithm, it can have highly detrimental effects. [49] While also looking through the camera datasheet, we realised that the OV8865 had a built in lens correction function that was not initially used. The lens correction algorithm on the OV8665 compensates for the vignetting effect by calculating a gain for each pixel and correcting each pixel to compensate for the illumination drop off. [48] By turning this on and using the default settings, the problem was reduced and there was better illumination and colour pickup in the corners of the frame. Due to time constraints, the algorithm was not manually tuned to achieve the best results.

During Integration testing, it was realised that because of the lengthy procedures, the sunlight was affecting the testing environment differently throughout the day. To accommodate for that, many calibrations were done based on how dark or lit the testing environment was.

Figure 36: From left to right: sunlight example, room lighting.

One more important measurement done during Integration testing was an accurate measurement of the distances of balls from the camera. This was done so that these values could be used to tune a model that approximates the $x$ and $y$ distances of the balls relative to the camera. An accurate method of measurement is using a flat surface with a reference straight line which is used to accurately place two pieces of red tape to represent the $x$ and $y$ axis. Then, the rover is placed directly in the middle and the camera is set at a perfect 90 degree angle relative to the floor which is done by using 2 triangles, one to ensure that the y axis is aligned with the camera len's center, and one more to align the camera with the x axis. After ensuring that, the x and y components of the balls' position are projected to the red tapes and then are measured. The measurements combined with the raw image of the camera were given to the Vision module. This procedure was done multiple times and was improved upon throughout the course of the project, and due to the good communication with the Vision team member, an accurate model of distance approximation was made as shown in the video demo.

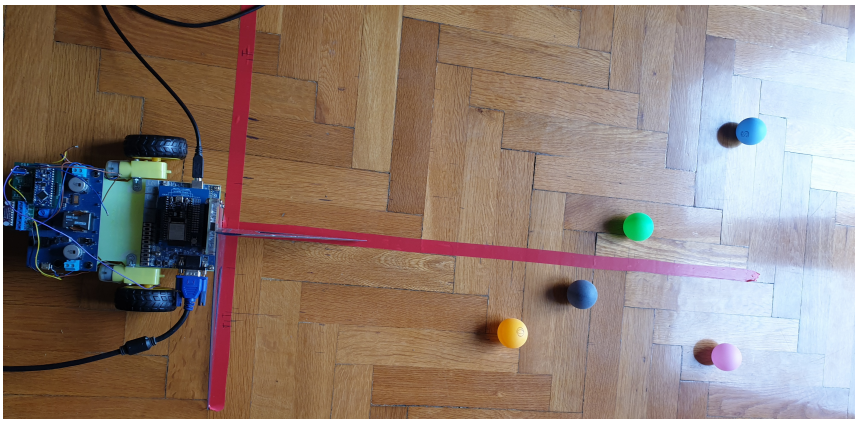

Figure 37: Measurement setup

## 4.6 Energy

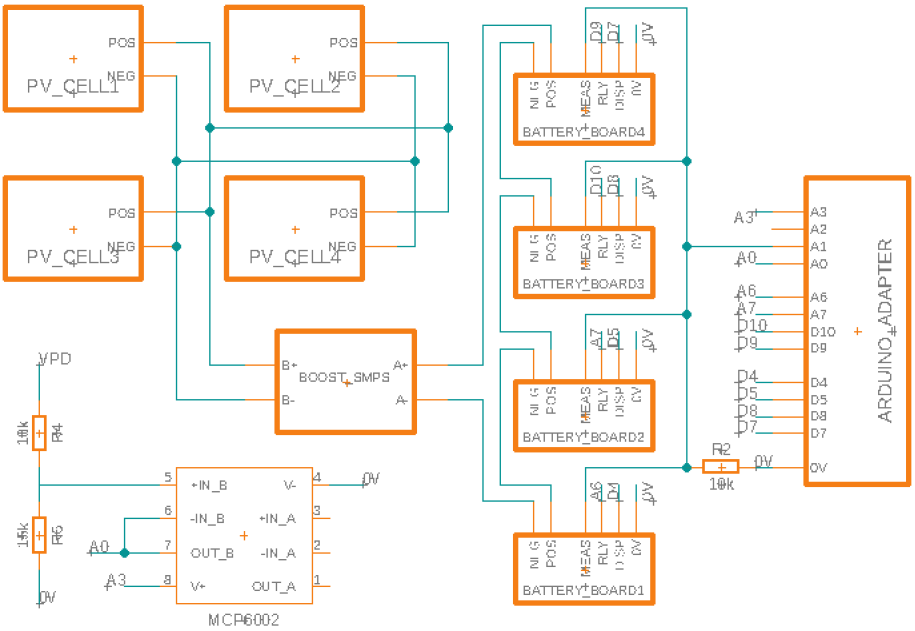


Figure 38: Circuit diagram of the finished energy system.

### SMPS Configuration

The voltage of the series battery pack is higher than the voltage of the parallel connected PV panels. The voltage of the PV array must therefore be stepped up before it can be used for charging. The SMPS is therefore operated in boost mode with the PV panels connected on port B and the battery on port A. Moreover, as the SMPS has a max output voltage of 7 V in synchronous boost mode[19], non-synchronous mode was used. Finally, the on/off switch was set to emphoff such that all power comes from the PV array and not from the USB connection powering the Arduino.

### Measurements

Measurements of the circuit currents and voltages are used to track the operation of the subsystem. During charging, the most important measurements are the output voltage, the cell voltages, and the battery current.

The battery voltage is usually in the range 10.0 to 14.4 V. However, the Arduino can only measure voltages up to 4 V. To solve this issue the output voltage was passed through a voltage divider. The voltage divider consist of two parts. Firstly, a 560 Ω and a 330 Ω resistor create $V_{pd} \approx 0.37 * V_{out}$. $V_{pd}$ is then reduced even further using a 15 $k\Omega$ resistor and a 10 $k\Omega$ resistor to $\sim 0.22 * V_{out}$, which is then passed to the Arduino. With this voltage divider it should be possible to measure output voltages up to 18 V. The second stage uses far large resistors than the first so that the two do not impact one another. However, due to the large resistance values, the current drawn by the Arduino during sampling would cause a significant voltage drop. To prevent this an op-amp from the Circuits lab was used to buffer the voltage before it was connected to the Arduino.

The measurement port on the battery boards is used to measure cell voltages. To use this port the battery board relay is be switched by setting the *RLY* input *HIGH*. While the relay is switched the battery cell will be disconnected from its power terminal. For a series battery pack this means that the cells can only ever be measured or charged/discharged at a given time. To have the smallest impact on operation the measurements are therefore completed as quickly as possible and taken at a low frequency. To measure the voltage of a cell the first step is to set the output current to 0 mA to prevent voltage spikes while the battery is disconnected. The relay is then switched, the voltage is sampled and the relay is switched back. When abiding by the switching times of the relay and giving time for the current/voltage to stabilise this process takes about 40 ms. To minimise the number of

Arduino ports needed for measurements, the cell voltages are not measured at the same time. Instead, one cell is measured every second. As there are 4 battery cells this means that each cell is measured with 4 second intervals. Due to the cell voltages changing very slowly the long update time was not found to be a problem.

The current in and out of the battery needs to be tracked to determine the SOC. However, the current sensor is on the wrong side of the SMPS to measure the current flowing into the batteries. Initially this problem was solved by connecting a 0.5 Ω resistor between the SMPS output and the battery. By measuring the voltage across this resistor the current into the battery could be determined. Initial tests of this solutions gave promising results. However, an excessively complex circuit was needed to ensure the sensor worked for both positive and negative currents. It was determined that a better solution was to simply calculate the output current from the input current using the formula $I_{out} = (1 - \delta) * I_{in}$. A problem with this solution is that it assumes CCM. At the standard charging current of 250 mA this assumption holds, but for smaller current, below about 150 mA, the SMPS might enter DCM. In this case the calculated current will be somewhat higher than its true value, which might impact SOC tracking. However, as will be discussed, the SOC algorithm has methods for correcting itself and this will hopefully mean that the error does not get too big. Unfortunately, it was not possible to test this due to the ban on working with batteries.

### Controlling the SMPS

The dual loop controller from the sample charging code[20] was used as the starting point for the controller design. However, some changes were necessary to accommodate other design choices. As previously discussed, the current needs to be set to 0 for only a couple of milliseconds when cell voltages are being measured. This is not possible using the 1 Hz Moore machine from the original controller. As such the slow loop was changed to a Mealy machine, where the outputs could change with the clock speed of the fast loop. This had the unfortunate side effect of creating timing issues as the resulting code was too complex for the Arduino to successfully run the fast loop at 1 kHz. For this reason the clock speed of the fast loop was reduced to 500 Hz. It should also be noted that the PI-gains from the sample code were changed as they did not work optimally in boost mode. The new gains were found through trial and error.

### Charging Algorithm

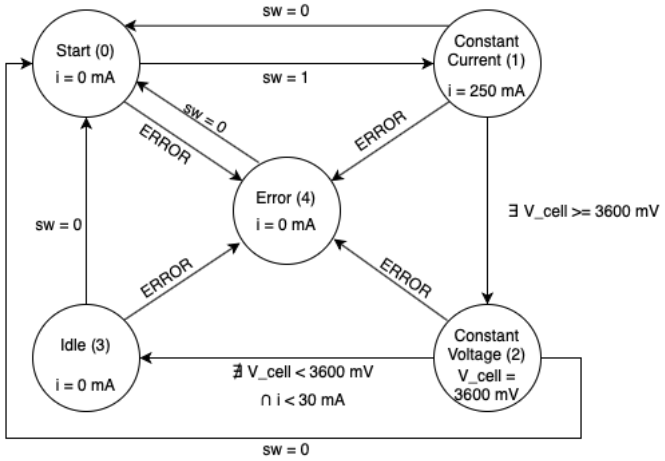


Figure 39: State machine of the charging algorithm.

The battery cells are designed to be charged first with a constant current and then with a constant voltage of 3.6 V[21]. The implemented charging algorithm follows this specification as much as possible. The state machine starts in state 0 where the output current is set to 0. It waits in this state until the OL/CL switch is set to 1, after which it moves to the constant current state. In the constant current state the current is nominally 250 mA. However, as will be discussed later, if the solar panels do not produce enough power to charge at 250 mA, the MPPT algorithm might impose another lower value for the constant current in this state. Moreover, technically the current is not constant as 40 ms of each second is used to measure cell voltages, and for a short period there will be zero current. These short charge breaks were not found to have an impact on the performance of the charging process. In

fact, charging $LiFePO_4$ battery cells with a pulsating current has even been found to severely reduce battery degradation[50], which might mean better performance than truly constant current in the long term. The charging algorithm leaves the constant current state for the constant voltage state once any cell reaches a cell voltage of 3600 mV.

Initially the output voltage was kept constant in the constant voltage state. However, even if the voltage across the battery pack as a whole is constant this does not mean that the voltage across individual cells is constant. Testing revealed that with a constant output voltage, weaker cells could sometimes reach a cell voltage over 3700 mV as current died down. To prevent this, the constant voltage state was redesigned from keeping the voltage across the battery pack constant, to trying to keep the voltage of each individual cell constant. This was done in two steps. Firstly, the battery input current is regulated so as to keep the highest cell voltage in the range 3600 to 3630 mV. Secondly, balancing ensured that cells of lower voltage catch up to the highest voltage. The code used to control the constant voltage state is shown in Figure 40. Initial test of this code gave promising results, but it could not fully be tested due to the ban on working with batteries. However, Figure 42 shows the ability of the code to stabilise all cell voltages around 3600 mV. After all cell voltages have reached 3600 mV and the current has died to below 30 mA, the charging algorithm is done and the state machine enters an idle state.

```
case 2:{ // Constant voltage/Balancing state
  if(input_switch == 0){ //If the switch = 0, then go back to start
    next_state = 0;
  }else if(min_cell < 3600 or I_setpoint > 30){ //If not done balancing stay in balancing state
    next_state = 2;
    balancing(); //Switch on the relevant balancing resistors

    //Adjust current to keep maximum cell voltage in desired range
    if (max_cell > 3630 and I_setpoint > 0 or I_setpoint > max_power / V_out){
      I_setpoint -= 0.5;
    }else if(max_cell < 3600 and I_setpoint < 250 and I_setpoint < max_power / V_out){
      I_setpoint += 0.5;
    }

  }else{ //If done balancing move to charge rest
    next_state = 3; //Go to charge rest
    SOH[1]++;//We have now finished a charge cycle so add 1 to SOH[1]
    SOH[0] = 0; //At this point no charge has been used so reset depth of discharge
    SOH[4] = 0; //At this point no energy has been expended so reset SOH[4]
  }

  break;
}
```

Figure 40: Code used to keep cell voltages constant.

### SOH and Cell Balancing

To monitor the battery's SOH the charge capacity, energy capacity, cell voltage balance, and the number of completed charge/discharge cycles are all tracked. To retain the data between sessions, it was logged in a CSV format to the SD card every 60 seconds. The relevant data was then read in from the SD-card on start up.

To maintain SOH the cell voltages were balanced every charge cycle. As previously explained balancing should only occur at high SOC, which is why balancing is only done in the constant voltage state. The code use for balancing is shown in Figure 41. During CV charging it is desirable that all cells have a voltage of 3600 mV. If any cell has a voltage which is higher than 3600 mV, its voltage is too high and the dissipative resistors are switched on to lower the voltage. When used in conjunction with the constant voltage charging algorithm this code was extremely successful at balancing the cells. The balancing is shown in action in Figure 42. As can be seen, the cell voltages are one by one brought up to and kept at 3600 mV. Even with an initial voltage difference of 200 mV, the balancing only takes about 1000 seconds = 17 minutes.

```
//Balances the cell voltages
void balancing(){
  //If cell voltage is too high, set dis input high
  for(int i = 0; i < 4; i++){
    if(V_cell[i] > 3600){
      digitalWrite(dis_order[i],true); //Turn on dis
    }else{
      digitalWrite(dis_order[i],false); //Turn off dis
    }
  }
}
```

Figure 41: Function called while charging at high SOC to balance out the voltage of the cells.

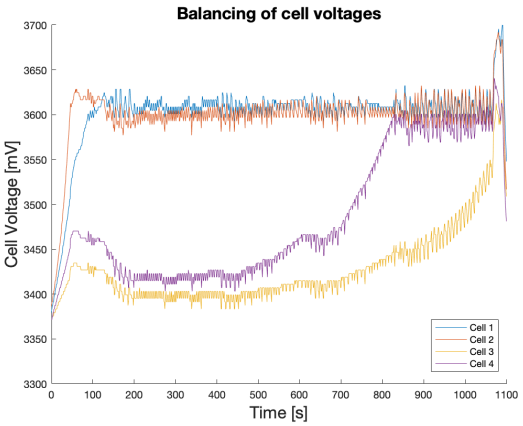


Figure 42: Cell voltages as they are being balanced, ignore voltage spikes at the end as they are not related to the balancing.

## MPPT

As previously discussed, the PV array should be operated in the region where the current is lower than at the maximum power point. To ensure this, the following algorithm was used. First, the duty cycle was set to 0. This makes the input impedance of the SMPS very high, leading to a near 0 PV array current. The duty cycle is then increased until the output current/voltage reaches its desired value. If the desired operating point requires less power than what the PV array can provide, then an equilibrium should be found as the duty cycle is increased. If however the desired operating point requires more power than the PV panels can provide, then no equilibrium can exist and the duty cycle will increase to 1 without ever reaching the desired operating point. Thus, if the duty cycle ever reaches 1 during operation the charging algorithm must be trying to draw more power from the PV array than is available. Whenever the duty cycle reaches 1, the variable *max_power*, is reduced by 100 mW and a new current setpoint is found through the formula $i_{setpoint} = max\_power/V_{out}$. The duty cycle is then reset to 0 and the process repeats until an equilibrium is found. When at an equilibrium, the algorithm increases *max_power* every 60 seconds to see if it is possible to operate at a higher power point than it currently is. If not, the algorithm will simply fall back to the same equilibrium. This algorithm was found to be very successful at tracking the maximum power point of the PV array. It is shown in action in the demo video.

## SOC

Implementing the Coulomb counting was done by integrating the current flowing into the battery. The integral value gives how much current has flowed in or out of the battery. The SOC as a percentage can then be found by dividing the net charge into the battery by the charge capacity of the battery and then multiplying by 100. However, due to effects such as the Coulombic efficiency of the battery, the integral can start to stray when done over many cycles. As such, the SOC is reset to the maximum capacity every time the battery is fully charged. The maximum capacity will of course reduce over time as the battery degrades. This is why the maximum capacity is measured and updated with every discharge cycle. Moreover, as the Arduino is sometimes switched off before a charging/discharging cycle is complete, it is necessary to know the SOC on startup. As such, the current SOC is continually logged to the SD card during operation. When the Arduino is then switched on again, it simply fetches the SOC data from the SD card and continues at the SOC at which it left off.

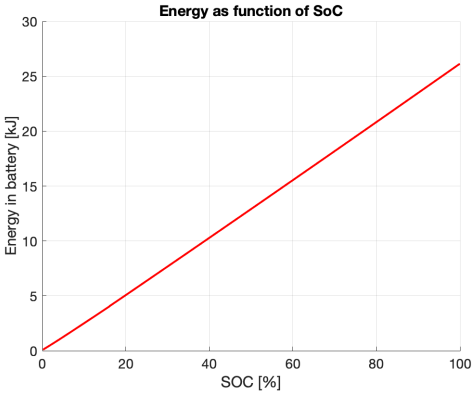


Figure 43: Lookup table data

The SOC should be used to estimate the range of the rover. The obvious way of estimating the range is to divide the remaining energy in battery by the power consumption of the rover and multiply by the speed. However, the SOC does not directly give how much energy is in the batteries at any given time. To find the energy in the battery, the total expended energy was logged alongside SOC data for a full discharge of the battery. From this data a 600 element lookup table was created which related the SOC of the battery to the remaining energy in the battery. The lookup table data is plotted in Figure 43.

#### Discharging

The energy module is also tasked with keeping track of the SOC and SOH during discharging. This was done using the same algorithms as during charging. In addition the battery power was continually tracked as a function of time and SOC. When the battery is done d is charging, this information is used to update the table relating the SOC of the battery to the remaining usable energy. This is done using the formula $New\ Energy\ Array = 0.99 * (Old\ Energy\ Array) + 0.01 * (Measured\ Energy\ Array)$. By having the energy array update itself it is hoped that the energy array will be a good approximation even after many battery cycles as the battery performance starts to deteriorate.

#### Communicating with Other Modules

Though it is not necessary to fully integrate the energy module with the rest of the rover, other submodules, specifically command, needs access data such as the battery SOH, SOC and remaining energy. For communicating with other modules the Arduino shield has a set of UART ports. However, as group members were not in the same location it was not possible to physically connect the energy module to the rover, which is necessary to use UART. As such, an alternative approach was employed. First the Arduino was connected to a computer via USB. On the computer a Python script was run. At the start, the Python script establishes a connection to a server created by running a similar script on the command module. After a connection has been established the Python script starts reading the serial data coming from the Arduino and transmits it using TCP to the command module. Each message coming from the Arduino is in CSV form where the first entry is the message ID, which allows the command script to decode what type of data is being sent.

# 5 Project Management and Organisation

As this project was carried out remotely with contributors located in different countries, it was important to have a good framework for communication and management. For direct communications between the team, WhatsApp was used for short texts, and Microsoft Teams was utilised for calls as its screen-sharing functionality was useful for helping to debug errors.

The main tool used for written communications and management was Git + GitHub. As the codebase was incredibly complex, involving many different libraries and with each submodule being capable as a standalone project, it was vital to have a version control system in place. Being able to keep a history of commits and changes made to a project was useful, especially when trying to track down the origin of a bug and what caused it.

There were different structures for communication during the designing phase and testing phase of the product. During the designing phase, there was a lot of one-on-one communication with other sub teams to try to determine the functional requirements and design aspects which involved more than one subsystem. This was vital to try to limit time spent on researching topics which had been researched by other team members while establishing a framework for the project as a whole. It also allowed for swift documentation and planning for implementation. When it came to the testing phase, in order to effectively communicate improvements between Integration and the individual subsystems, there was clear documentation for feedback delivered through WhatsApp and Microsoft Teams to improve upon project work. Likewise, there were documents maintained for data transmission and formatting to ensure subsystems could communicate effectively with one another.

The team also made use of GitHub Issues to track progress and accountability in the initial design phase. A thread was opened for each submodule to show what the lead for that submodule had been

doing and potential avenues of achieving their goals. This was beneficial both for the leads to keep track of their research, but also allowed other members to contribute to other submodules by adding comments and voicing their thoughts. GitHub Issues were also linked directly to commits in the codebase to allow for a more in-depth explanation and reasoning with context for a commit than what is allowed in the commit message area.

By way of using GitHub, the team also made use of GitHub's Actions, which allow for automated workflows to be run on certain events like a commit or a push for continuous integration. An Action was designed for the writing of this report that compiled the LaTeXdocument and produced a PDF.

Simultaneously, a Gantt chart was maintained to keep track of progress and is available for viewing under the maintained GitHub repository linked in the Appendix.

# 6 Intellectual Property

Intellectual Property (IP) is a complex topic that has financial, technological, and moral significance when developing products or inventions. In short, IP rights are what lets companies, inventors and creators prevent other people from using their work [51]. Many types of IP rights exists including trademarks, copyrights, design rights and patents. In the context of our design project, there are sections within IP which are less applicable than others. These include trademarks, as no brand is being developed, and design rights, as the rover's physical design has been provided by the college and will not be significantly altered by students. Note however, that as the college has created the rover design, it owns the design rights and could stop students from using its appearance if desired. Patents protect new inventions. To be eligible for a patent an invention must be new, useful and not obvious. Unfortunately, most of the technology used in the rover is not new, but has been developed by others. This includes the hardware, communication protocols, vision algorithms. As such, it is unlikely that the project has produced any patent opportunities. That being said, a lot of the code and algorithms written could be considered as original expression, which would grant the group a copyright license. However, as the Arduino and its libraries are under the GNU General Public License [52], an open-source license, users need to follow the same license when using these libraries and we would hence be restricted to an (L)GPL license as well. A more complex situation is the system that is running on the FPGA, although all components are available for testing in what is known as the Intel FPGA IP Evaluation Mode, any commercial use of the IP would require a license to be bought from Intel[53]. There are few individual modules written in SystemVerilog that could be copyrighted, like the Image processing module and the image transfer over SPI module, if the FIFO and divider components were replaced with either open-source/self-designed modules.

# References


[1] A. Bouchaala, *Mars Rover Instructions*, 2021.

[2] ——, *Content of Boxes*, 2021.

[3] Espressif Systems. "ESP32 Series Datasheet." (2021), [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[4] E. Stott, *ESP32 Arduino Adapter*, 2021.

[5] cdaviddav. "Arduino vs ESP8266 vs ESP32 Comparison." (2020), [Online]. Available: https://diyi0t.com/technical-datasheet-microcontroller-comparison/.

[6] S. Santos. "ESP32 Pinout Reference." (2018), [Online]. Available: https://randomnerdtutorials.com/esp32-pinout-reference-gpios/.

[7] R. Mitchell. "Common Communication Peripherals on the Arduino." (2018), [Online]. Available: https://maker.pro/arduino/tutorial/common-communication-peripherals-on-the-arduino-uart-i2c-and-spi.

[8] yida. "UART vs I2C vs SPI – Communication Protocols and Uses." (2019), [Online]. Available: https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/.

[9] O. Source. "Arduino ESP32." (2021), [Online]. Available: https://github.com/espressif/arduino-esp32.

[10] D. Thomas, *ELEC50014 - Software Systems*, 2021.

[11] Arduino. "Serial." (2019), [Online]. Available: https://www.arduino.cc/reference/en/language/functions/communication/serial/.

[12] Terasic Inc. "DE10-Lite Board." (), [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English%5C&No=1021.

[13] ——, "D8M GPIO - 8 Mega Pixel Digital Camera Package with GPIO interface." (), [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English%5C&No=1011.

[14] K. Abdelouahab, M. Pelcat, J. Sérot, and F. Berry, "Accelerating CNN inference on FPGAs: A survey," *CoRR*, vol. abs/1806.01683, 2018. arXiv: 1806.01683. [Online]. Available: http://arxiv.org/abs/1806.01683.

[15] Intel Cooperation. "High-level synthesis compiler." (), [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/hls-compiler.html.

[16] F. Fahim, B. Hawks, C. Herwig, *et al.*, "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices," *CoRR*, vol. abs/2103.05579, 2021. arXiv: 2103.05579. [Online]. Available: https://arxiv.org/abs/2103.05579.

[17] N. M. Ali, N. K. A. M. Rashid, and Y. M. Mustafah, "Performance comparison between RGB and HSV color segmentations for road signs detection," *Applied Mechanics and Materials*, vol. 393, no. 1, pp. 550–555, 2013. DOI: http://dx.doi.org/10.4028/www.scientific.net/AMM.393.550.

[18] Intel Corporation. "Intel® FPGAs & SoC FPGAs." (), [Online]. Available: https://www.intel.co.uk/content/www/uk/en/products/details/fpga.html.

[19] E. J. S. Holen, "Power lab logbook," 2021.

[20] P. Clemow, *Battery_Charge_Cycle_Logged_V1.1.ino*. [Online]. Available: https://imperiallondon.sharepoint.com/sites/msteams%5C_03ef97/Shared%20Documents/General/Energy%5C%20Module/Battery%5C_Charge%5C_Cycle%5C_Logged%5C_V1.1.ino.

[21] *Ampsplus 14500 3.2v 500mAh Battery Button*. [Online]. Available: https://www.ampsplus.co.uk/ampsplus-14500-3-2v-500mah-battery-button.

[22] T. Green. "Photo-Voltaic Energy." (), [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-2060823-dt-content-rid-8486224_1/courses/10435.202020/2%20Notes%20-%20Photovoltaic%20Energy%20-%20ELEC50012%2020-21%281%29.pdf.

[23] P. Clemow. "Introduction to Batteries." (2021), [Online]. Available: https://imperial.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=ae9ef40c-98fa-4698-a8bd-ad2600e15323.

[24] "Battery balancing." (), [Online]. Available: https://en.wikipedia.org/wiki/Battery_balancing.

[25] STMicroelectronics. "P-channel 100 V, 0.136 typ., 10 A STripFET F6Power MOSFET in a DPAK package." (), [Online]. Available: https://www.st.com/en/power-transistors/std10p10f6.html.

[26] E. Dickinson, "Batteries — partial-state-of-charge," in *Encyclopedia of Electrochemical Power Sources*, J. Garche, Ed., Amsterdam: Elsevier, 2009, pp. 452–458, ISBN: 978-0-444-52745-5. DOI: https://doi.org/10.1016/B978-044452745-5.00877-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780444527455008777.

[27] M. Danko, J. Adamec, M. Taraba, and P. Drgona, "Overview of batteries state of charge estimation methods," *Transportation Research Procedia*, vol. 40, pp. 186–192, 2019, TRANSCOM 2019 13th International Scientific Conference on Sustainable, Modern and Safe Transport, ISSN: 2352-1465. DOI: https://doi.org/10.1016/j.trpro.2019.07.029. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352146519301905.

[28] K. S. Ng, C.-S. Moo, Y.-P. Chen, and Y.-C. Hsieh, "Enhanced coulomb counting method for estimating state-of-charge and state-of-health of lithium-ion batteries," *Applied Energy*, vol. 86, no. 9, pp. 1506–1511, 2009, ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2008.11.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306261908003061.

[29] "State of health (soh) determination." (), [Online]. Available: https://www.mpoweruk.com/soh.htm.

[30] L. Ungurean, G. Cârstoiu, M. V. Micea, and V. Groza, "Battery state of health estimation: A structured review of models, methods and commercial devices," *International Journal of Energy Research*, vol. 41, no. 2, pp. 151–181, 2017. DOI: https://doi.org/10.1002/er.3598. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/er.3598. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/er.3598.

[31] Y. Barsukov, "Battery cell balancing: What to balance and how," [Online]. Available: https://www.ti.com/download/trng/docs/seminar/Topic%202%20-%20Battery%20Cell%20Balancing%20-%20What%20to%20Balance%20and%20How.pdf.

[32] S. Santos. "ESP32 Access Point AP Web Server." (2018), [Online]. Available: https://randomnerdtutorials.com/esp32-access-point-ap-web-server/.

[33] D. Kaar. "ESPSoftwareSerial." (2021), [Online]. Available: https://www.arduino.cc/reference/en/libraries/espsoftwareserial/.

[34] M. R. Thakur. "ESPSoftwareSerial." (2018), [Online]. Available: https://circuits4you.com/2018/12/31/esp32-hardware-serial2-example/.

[35] S. Santos. "ESP32 Dual Core Arduino IDE." (2018), [Online]. Available: https://randomnerdtutorials.com/esp32-dual-core-arduino-ide/.

[36] shaielc. "SlaveSPIClass." (2018), [Online]. Available: https://gist.github.com/shaielc/e0937d68978b03b2544474b641328145.

[37] iPAS. "ESP32 Slave SPI." (2019), [Online]. Available: https://github.com/iPAS/esp32-slave-spi.

[38] Nodejs. "Node.js v16.3.0 documentation." (), [Online]. Available: https://nodejs.org/api/net.html#net_net.

[39] Mozilla. "An overview of http." (), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[40] socket.io. "Server api." (), [Online]. Available: https://socket.io/docs/v4/server-api/#socket-on-eventName-callback.

[41] B. Design. "React website tutorial - beginner react js project fully responsive." (2020), [Online]. Available: https://www.youtube.com/watch?v=I2UBjN5ER4s.

[42] KonvaJS. "Getting started with react and canvas via konva." (), [Online]. Available: https://konvajs.org/docs/react/index.html.

[43] Thaddeus Abiy, Hannah Pang, Beakal Tiliksew, Karleigh Moore, and Jimin Khim. "A star search." (), [Online]. Available: https://brilliant.org/wiki/a-star-search/.

[44] B. Grinstead. "A* search algorithm in javascript." (), [Online]. Available: https://briangrinstead.com/blog/astar-search-algorithm-in-javascript/.

[45] edstott, *EE2Rover*, https://github.com/edstott/EEE2Rover, 2021.

[46] A. R. Smith, "Color gamut transform pairs," *SIGGRAPH Comput. Graph.*, vol. 12, no. 3, pp. 12–19, Aug. 1978, ISSN: 0097-8930. DOI: 10.1145/965139.807361. [Online]. Available: https://doi.org/10.1145/965139.807361.

[47] Intel Corporation, *Nios II Custom Instruction User Guide*, 2020.04.27, Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 95054-1549, USA., Apr. 2020. [Online]. Available: https://www.intel.co.uk/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_nios2_custom_instruction.pdf.

[48] *Ov8865 rev1 datasheet product specification*, 2.03, colour CMOS 8 megapixel (3624* 2448) image sensor with OmniBSI technology, OmniVision Technologies, 4275 Burton Drive Santa Clara California, 95054, Feb. 2014.

[49] Y. Zheng, S. Lin, C. Kambhamettu, J. Yu, and S. B. Kang, "Single-image vignetting correction," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 12, pp. 2243–2256, 2008.

[50] G. Garcia, S. Dieckhofer, W. Schuhmann, and E. Ventosa, "Exceeding 6500 cycles for lifepo4/li metal batteries through understanding pulsed charging protocols," *J. Mater. Chem. A*, vol. 6, pp. 4746–4751, 2018.

[51] B. Chapman and B. Hallam, "Intellectual property in engineering," Imperial College London, 2021.

[52] Arduino. "License." (2021), [Online]. Available: https://github.com/arduino/Arduino/blob/master/license.txt.

[53] *AN 320: Using Intel® FPGA IP Evaluation Mode*, Eng, Online, Intel Corporation, Nov. 2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an320.pdf.

# Appendices

## A Appendix 1: FPGA Resource Usage Summary

| | | |
|---|---|---|
| Total Logic Elements | 16,994/49,760 | 34% |
| Total Registers | 8907/51,907 | 17% |
| Total Pins | 171/360 | 48% |
| Total Memory Bits | 1,341,898/ 1,677,312 | 80% |
| Total M9K Blocks | 177/182 | 97% |
| Embedded Multipliers | 10/288 | 3% |
| Total PLLs | 1/4 | 25% |
| Total Power Dissipation | 469.83mW | N/A |
| Dynamic Power Dissipation | 280.97mW | N/A |

## B Appendix 2: FPGA Resource Usage by Entity

| | Logic Cells | Dedicated Logic Registers | Memory Bits | M9Ks | UFM Blocks | DSP Elements | DSP 9x9 | DSP 18x18 | Pins | LUT-Only LCs | Register-Only LCs | LUT/Register LCs |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| \|DE10_LITE_D8M_VIP | 16994 (13) | 8840 (12) | 1341894 | 177 | 1 | 10 | 2 | 4 | 171 | 8154 (2) | 1916 (11) | 6924 (0) |
| \|FpsMonitor:uFps\| | 77 (77) | 44 (44) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 33 (33) | 5 (5) | 39 (39) |
| \|Qsys:u0\| | 16585 (0) | 8620 (0) | 1341894 | 177 | 0 | 10 | 2 | 4 | 0 | 7964 (0) | 1875 (0) | 6746 (0) |
| \|EEE_IMGPROC:eee_imgproc_0\| | 4571 (952) | 1855 (540) | 8462 | 5 | 0 | 0 | 0 | 0 | 0 | 2716 (412) | 177 (33) | 1678 (507) |
| \|MSG_FIFO:MSG_FIFO_inst\| | 64 (0) | 37 (0) | 8192 | 1 | 0 | 0 | 0 | 0 | 0 | 27 (0) | 0 (0) | 37 (0) |
| \|STREAM_REG:in_reg\| | 34 (34) | 28 (28) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 (6) | 1 (1) | 27 (27) |
| \|STREAM_REG:out_reg\| | 84 (84) | 21 (21) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 63 (63) | 0 (0) | 21 (21) |
| \|colour_threshold:ColourDetect\| | 57 (57) | 0 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 57 (57) | 0 (0) | 0 (0) |
| \|rgb_to_hsv:rgb_hsv\| | 3380 (551) | 1229 (125) | 270 | 4 | 0 | 0 | 0 | 0 | 0 | 2151 (400) | 143 (54) | 1086 (229) |
| \|Qsys_alt_vip_vfb_0:alt_vip_vfb_0\| | 1967 (198) | 1516 (109) | 72192 | 9 | 0 | 0 | 0 | 0 | 0 | 450 (88) | 396 (3) | 1121 (114) |
| \|Qsys_altpll_0:altpll_0\| | 11 (7) | 6 (2) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 (3) | 2 (0) | 6 (4) |
| \|Qsys_jtag_uart:jtag_uart\| | 164 (40) | 105 (13) | 1024 | 2 | 0 | 0 | 0 | 0 | 0 | 49 (17) | 25 (5) | 90 (18) |
| \|Qsys_key:key\| | 2 (2) | 2 (2) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 (0) | 0 (0) | 2 (2) |
| \|Qsys_led:led\| | 22 (22) | 10 (10) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 (2) | 10 (10) | 10 (10) |
| \|Qsys_mipi_pwdn_n:mipi_pwdn_n\| | 5 (5) | 1 (1) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 (3) | 0 (0) | 2 (2) |
| \|Qsys_mipi_pwdn_n:mipi_reset_n\| | 3 (3) | 1 (1) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 (1) | 0 (0) | 2 (2) |
| \|Qsys_mm_interconnect_0:mm_interconnect_0\| | 1009 (0) | 632 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 263 (0) | 259 (0) | 487 (0) |
| \|Qsys_nios2_gen2:nios2_gen2\| | 3225 (48) | 1762 (40) | 62720 | 13 | 0 | 6 | 0 | 3 | 0 | 1462 (8) | 373 (7) | 1390 (25) |
| \|Qsys_nios2_gen2_custom_instruction_master_multi_xconnect\| | 73 (73) | 0 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 70 (70) | 0 (0) | 3 (3) |
| \|Qsys_nios_custom_instr_floating_point_2_0:nios_custom_instr_floating_point_2_0\| | 1933 (0) | 496 (0) | 0 | 0 | 0 | 4 | 2 | 1 | 0 | 1402 (0) | 57 (0) | 474 (0) |
| \|Qsys_onchip_memory2_0:onchip_memory2_0\| | 71 (1) | 2 (0) | 1048576 | 128 | 0 | 0 | 0 | 0 | 0 | 69 (1) | 2 (0) | 0 (0) |
| \|Qsys_sdram:sdram\| | 354 (242) | 209 (121) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 145 (141) | 81 (3) | 128 (78) |
| \|Qsys_sw:sw\| | 10 (10) | 10 (10) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 (0) | 0 (0) | 10 (10) |
| \|Qsys_timer:timer\| | 145 (145) | 120 (120) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 24 (24) | 19 (19) | 102 (102) |
| \|Qsys_uart_0:uart_0\| | 137 (0) | 92 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 43 (0) | 12 (0) | 82 (0) |
| \|TERASIC_AUTO_FOCUS:terasic_auto_focus_0\| | 728 (232) | 326 (113) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 401 (125) | 54 (12) | 273 (95) |
| \|TERASIC_CAMERA:terasic_camera_0\| | 563 (71) | 401 (29) | 129464 | 17 | 0 | 0 | 0 | 0 | 0 | 161 (26) | 196 (24) | 206 (22) |
| \|CAMERA_RGB:CAMERA_RGB_inst\| | 340 (0) | 241 (0) | 22968 | 4 | 0 | 0 | 0 | 0 | 0 | 96 (0) | 97 (0) | 147 (0) |
| \|Bayer2RGB:Bayer2RGB_inst\| | 295 (202) | 180 (159) | 22968 | 4 | 0 | 0 | 0 | 0 | 0 | 93 (50) | 81 (81) | 121 (71) |
| \|CAMERA_Bayer:CAMERA_Bayer_inst\| | 65 (65) | 61 (61) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 (3) | 16 (16) | 46 (46) |
| \|rgb_fifo:rgb_fifo_inst\| | 170 (0) | 131 (0) | 106496 | 13 | 0 | 0 | 0 | 0 | 0 | 39 (0) | 75 (0) | 56 (0) |
| \|alt_vipitc131_IS2Vid:alt_vip_itc_0\| | 423 (113) | 262 (54) | 19456 | 3 | 0 | 0 | 0 | 0 | 0 | 161 (59) | 103 (10) | 159 (44) |
| \|altera_reset_controller:rst_controller_001\| | 3 (0) | 3 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 (0) | 2 (0) | 1 (0) |
| \|altera_reset_controller:rst_controller_002\| | 16 (10) | 16 (10) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 (0) | 9 (5) | 7 (5) |
| \|altera_reset_controller:rst_controller\| | 3 (0) | 3 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 (0) | 1 (0) | 2 (0) |
| \|i2c_opencores:i2c_opencores_camera\| | 262 (0) | 129 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 132 (0) | 4 (0) | 126 (0) |
| \|i2c_opencores:i2c_opencores_mipi\| | 279 (0) | 129 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 149 (0) | 6 (0) | 124 (0) |
| \|pzdyqx:nabboc\| | 136 (0) | 73 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 63 (0) | 8 (0) | 65 (0) |
| \|sld_hub:auto_hub\| | 183 (1) | 91 (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 92 (1) | 17 (0) | 74 (0) |

## 7 Appendix 3: GitHub Repository Source Code

`https://github.com/JosiahMendes/Engineering-Design-Project-MARS-Rover`