Imperial College London

PROJECT REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

Mars Rover

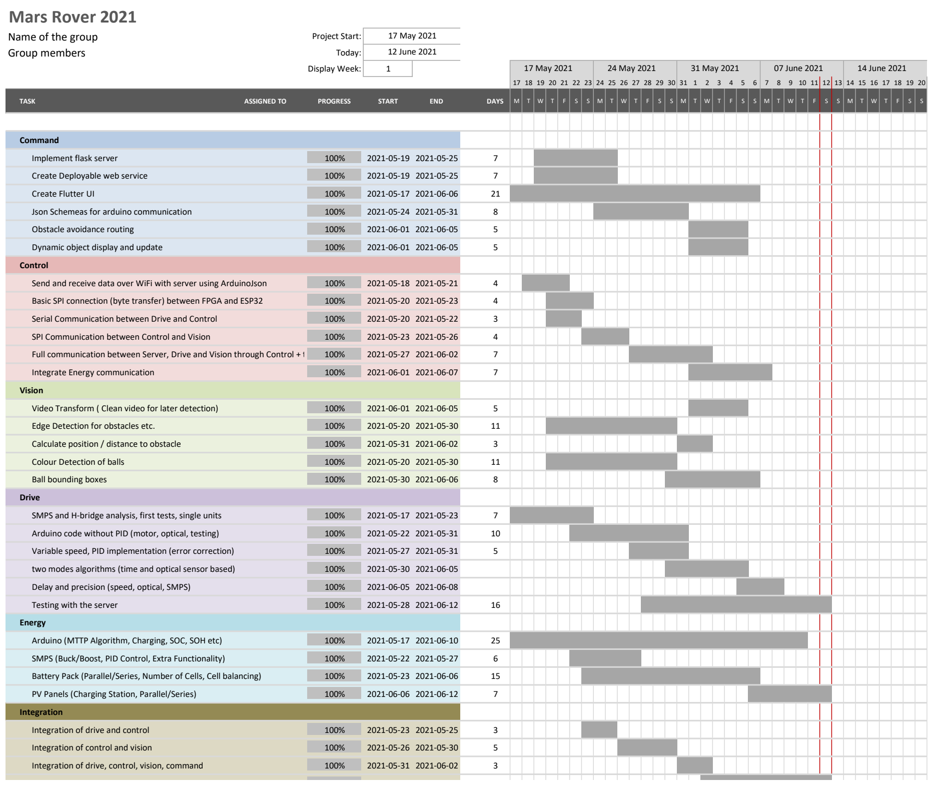
Authors:
Sam Taylor - 01705109
Martin Prusa - 01713176
Leonardo Garofalo - 01726454
Katherine Zhang - 01504365
Maximus Wickham - 01717673
Matilde Piccoli - 01764158

Date: July 13, 2022

Contents

1 Project Management

The given timeline for the project was 5 weeks. The project was split into 5 phases. The first week was used to analyse the project brief and break it down into requirements for each module. The following Gantt chart was used to set the tasks to be completed for each module and approximate deadlines for each task to be completed.



Mars Rover 2021

Name of the group

Group members

Project Start:	17 May 2021
Today:	12 June 2021
Display Week:	1

TASK	ASSIGNED TO	PROGRESS	START	END	DAYS
Command					
Implement flask server		100%	2021-05-19	2021-05-25	7
Create Deployable web service		100%	2021-05-19	2021-05-25	7
Create Flutter UI		100%	2021-05-17	2021-06-06	21
Json Schemeas for arduino communication		100%	2021-05-24	2021-05-31	8
Obstacle avoidance routing		100%	2021-06-01	2021-06-05	5
Dynamic object display and update		100%	2021-06-01	2021-06-05	5
Control					
Send and receive data over WiFi with server using ArduinoJson		100%	2021-05-18	2021-05-21	4
Basic SPI connection (byte transfer) between FPGA and ESP32		100%	2021-05-20	2021-05-23	4
Serial Communication between Drive and Control		100%	2021-05-20	2021-05-22	3
SPI Communication between Control and Vision		100%	2021-05-23	2021-05-26	4
Full communication between Server, Drive and Vision through Control +		100%	2021-05-27	2021-06-02	7
Integrate Energy communication		100%	2021-06-01	2021-06-07	7
Vision					
Video Transform (Clean video for later detection)		100%	2021-06-01	2021-06-05	5
Edge Detection for obstacles etc.		100%	2021-05-20	2021-05-30	11
Calculate position / distance to obstacle		100%	2021-05-31	2021-06-02	3
Colour Detection of balls		100%	2021-05-20	2021-05-30	11
Ball bounding boxes		100%	2021-05-30	2021-06-06	8
Drive					
SMPS and H-bridge analysis, first tests, single units		100%	2021-05-17	2021-05-23	7
Arduino code without PID (motor, optical, testing)		100%	2021-05-22	2021-05-31	10
Variable speed, PID implementation (error correction)		100%	2021-05-27	2021-05-31	5
two modes algorithms (time and optical sensor based)		100%	2021-05-30	2021-06-05	
Delay and precision (speed, optical, SMPS)		100%	2021-06-05	2021-06-08	
Testing with the server		100%	2021-05-28	2021-06-12	16
Energy					
Arduino (MTTP Algorithm, Charging, SOC, SOH etc)		100%	2021-05-17	2021-06-10	25
SMPS (Buck/Boost, PID Control, Extra Functionality)		100%	2021-05-22	2021-05-27	6
Battery Pack (Parallel/Series, Number of Cells, Cell balancing)		100%	2021-05-23	2021-06-06	15
PV Panels (Charging Station, Parallel/Series)		100%	2021-06-06	2021-06-12	7
Integration					
Integration of drive and control		100%	2021-05-23	2021-05-25	3
Integration of control and vision		100%	2021-05-26	2021-05-30	5
Integration of drive, control, vision, command		100%	2021-05-31	2021-06-02	3

Figure 1: Gantt Chart

The Gantt chart shows project progress from the second week onwards. In week 2, it was planned to work on the basic functionalities of each module e.g. the server for the command module, communication between ESP32 and the server, control of the motor based on data from the optical sensor. In week 3, the basic functionalities were completed and more sophisticated functionalities were considered. By the end of week 4, the design of each module was finalised. The focus of week 5 was testing of the full rover with the final versions of each module.

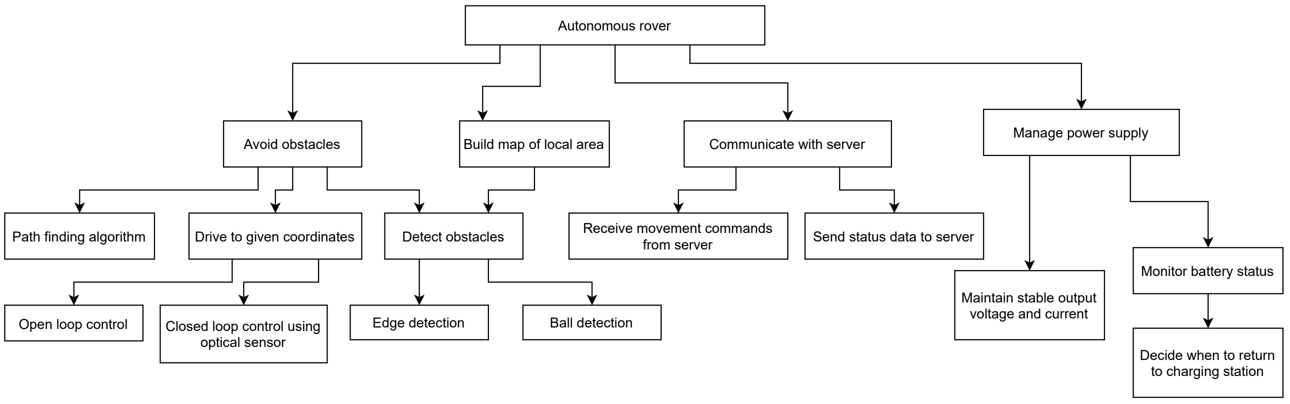
From the second week on-wards, bi-weekly meetings were held. At the beginning of each week, we held a 45-minute meeting. Each member would take 5 minutes to update the group on their progress on the targets set for the previous week, discuss any issues they found and set new targets for the week. Midway through the week, we had a brief 20-minute meeting to check in on everyone's progress and give the opportunity for the members to bring up any problems or difficulties they might have run into and adjust plans accordingly.

The productivity software Notion was also used for each member to make notes of their initial design process, development and testing. It provided an easy method to share progress with the rest of the team. The shared calendar was used to set dates of meetings and mark internal deadlines. A GitHub organisation was created and used for version control for all modules.

2 Design Process

2.1 Structural and Functional Design

Based on the brief given, the following requirement diagram was created. The diagram details the high level requirements of the rover system and the low level functionalities required to achieve them.



Autonomous rover
Avoid obstacles
Build map of local area
Communicate with server
Manage power supply
Path finding algorithm
Drive to given coordinates
Detect obstacles
Receive movement commands from server
Send status data to server
Maintain stable output voltage and current
Monitor battery status
Open loop control
Closed loop control using optical sensor
Edge detection
Ball detection
Decide when to return to charging station

Figure 2: Requirement diagram

The structural design for the full rover system is shown below. The subsystems of the rover system are shown with their internal components and the interactions between different components of different subsystems are drawn out.

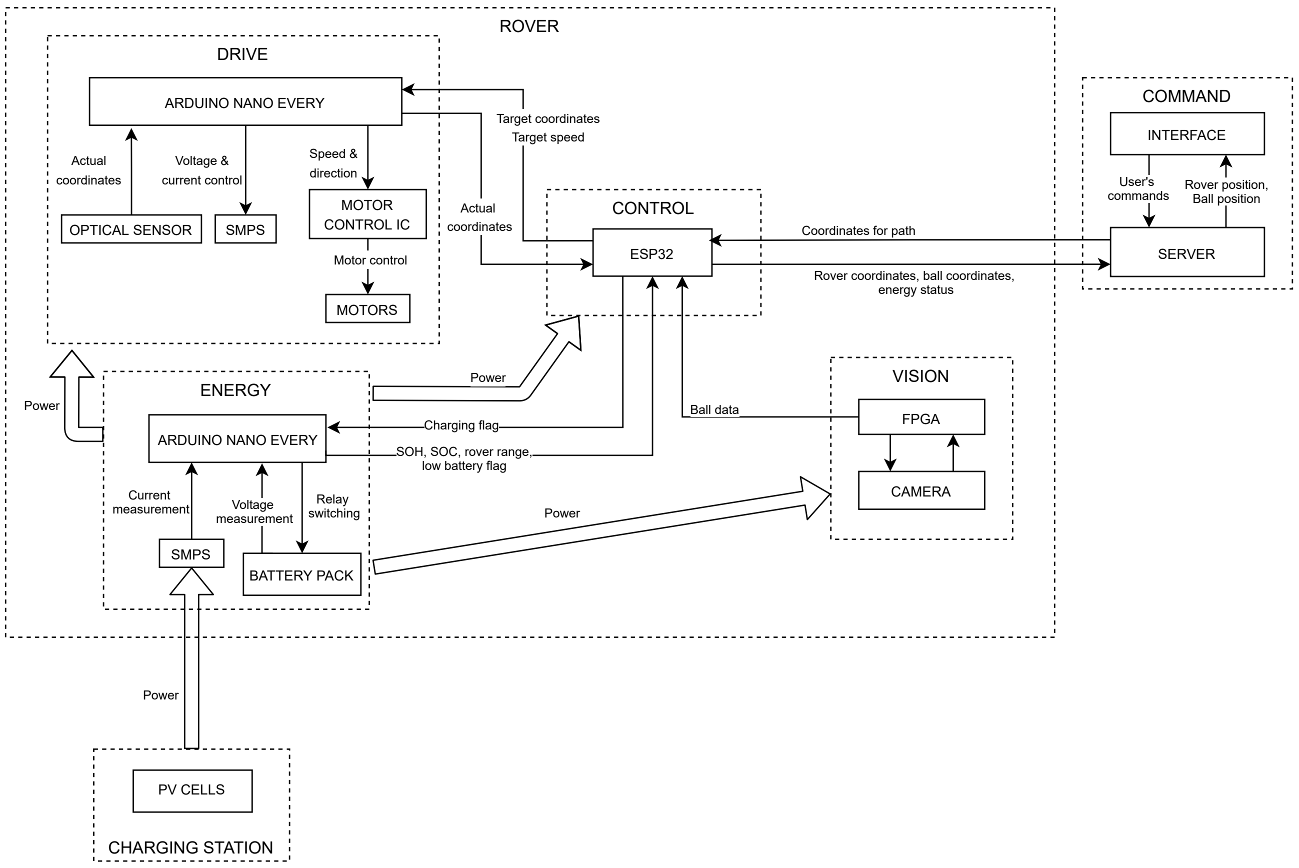


Figure 3: Block diagram showing structural design of rover system

The interactions between the rover system and external actors are shown in the diagram below.

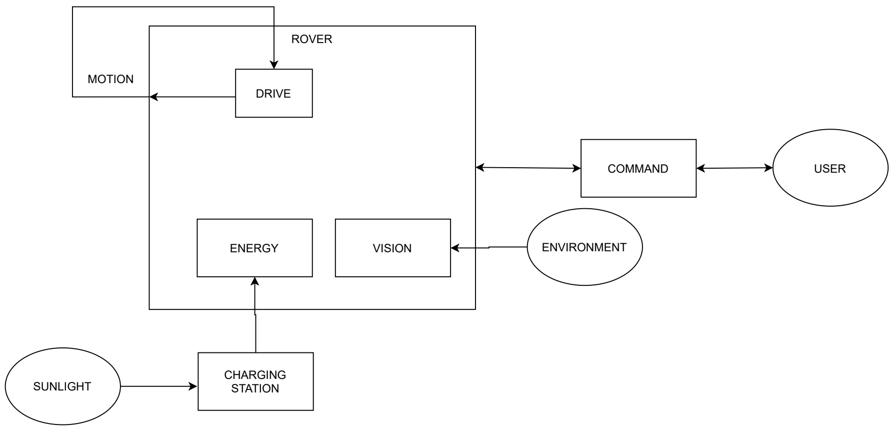


Figure 4: Diagram showing interactions between rover system and external actors

2.2 Problem Definition and Design Criteria

2.2.1 Drive

For the drive subsystem, the following were identified as the functional requirements to be implemented:

1. Translate the commands from the control unit into movement instructions for the two motors, in terms of angle and distance to be computed
2. Move the rover to the specified point in the coordinates system, controlling the speed and direction of the motors via the setting of the SMPS and H-bridge parameters in the Arduino code
3. Track the movements computed and thus deduce the rover location using the data provided by the optical sensor
4. Send the location, speed, and orientation data back to the server, communicating the end of the executed instruction

In accordance with these requirements, the design criteria on the base of which our design was evaluated were the following:

1. correct functionality with respect to the requirements
2. optimal moving algorithm to compute the fastest route in the smallest amount of time
3. precision of the movements with a 10% accepted error in the tacking of the location, due to the optical sensor limits

The main structure of the drive subsystem could be divided into two main elements, the two motors and the optical sensor.

Motors

The main elements we had to focus on were the functionality of the SMPS and H-bridge, and the design of a control algorithm to be implemented in the Arduino. For the first part, an analysis of the relevant material and specifications was pursued and then it was decided with the energy unit to use a reference voltage of 5V for the output of the SMPS (input of the H-bridge). After some testing with a maximum duty cycle for the SMPS, it was concluded that this reference voltage would lead to a maximum speed of the wheels of 60 rpm, a satisfactory speed for the rover to use and yet small enough for the optical sensor data to be reliable.

Optical sensor

The optical data had to be analysed via the Arduino for different surfaces, after adjusting the lens of the optical sensor, we managed to achieve an increase in the quality of the distance measurements. However, after some testing, we concluded that the sensor would give reliable data only for specific surfaces and an alternative way of computing the rover's change in location had to be implemented for surface too uniform or reflective.

In general, the drive unit development consisted in the coding of Arduino programs comprehensive of all the functional requirements of the subsystem, alongside measurements of the SMPS parameters and wheels' movements. Due to a large number of Arduino code, Github was found to be a useful tool to keep track of the process and provide the integration unit with updated code for the parallel testing of the whole rover.

2.2.2 Energy

The energy subsystem can be split into 4 top-level sections which all combine to provide battery power and charging to the rover. These can then be broken down into smaller subsections which are more manageable and easier to test individually (Table 1). This also allows for more unique features to be added later on in the process.

Section	Criteria	Description
Arduino	Data Collection	The current and voltage should be recorded continuously for safety
	Charge Strategy	Try to get near maximum possible capacity out of cells
	SOC Estimation	Real time percentage capacity of the battery via coulomb counting
	SOH Maintenance	Provide balancing of the cells and an estimation of the SOH
	PV MPPT algorithm	Get the maximum amount of power out of the PV Panels
	Communication	Transmit and receive data from Control
	Cell Safety	$I_{In}^{Max} = 250mA$, $I_{Out}^{Max} = 500mA$, $V_{Bat}^{Max} = 3.7V$, $V_{Bat}^{Min} = 2.3V$
	Rover Range	Work out Rover range from Drive current and speed
SMPS	Charging Buck/Boost	PV voltage stepped down to battery voltage
	Discharging Buck/Boost	Battery voltage stepped down to motor control voltage
	Drive Current Required	I_{Drive}^{Max} = 550mA
Battery Pack	Number of Cells	2
	Configuration	Series to allow to better current limitation ($V_{Nominal}$ = 6.4V)
PV Panels	Configuration	2x2 (Two in series then in parallel)
	Usage	A charging station setup in a sunlit area

Table 1: Energy Specification

Design Philosophy

The energy subsection required a lot of initial research into lithium-based cells (LiFePO cells were provided) as these have never been covered as a module. A document was made which collated the various amounts of research into cell charging/discharging, safety and PV panels. Then a process of simple testing ensued to characterise some of the cells and PV panels. After this was finished, the direction to take became clearer and work based upon the Gantt diagram (1) began. The aim was to investigate the variety of options available which allowed me to pick and choose the best. This did lead to a number of dead-ends but in many cases a lot was learned from these and subsequently improved on the overall design.

Due to the large amounts of code required for testing, a number of files were made which would be uploaded to Github. A final Arduino program was made towards the end of the project which combined all the functions from the various files. These will have been individually tested to ensure they work. Ideally, a final test would also be performed but due to possible defects in the cell supply, this was never realised.

2.2.3 Vision

The function of the vision system was broken into three general objectives, which the team agreed the system should achieve (See objectives below). We agreed aspects like rover position and pathfinding would be better tackled by different subsystems. The following diagram shows the three main functions of Vision.

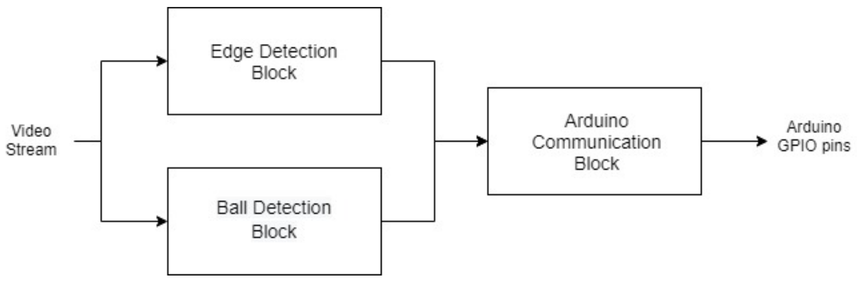


Figure 5: High Level Block diagram

After this high-level design, each objective was broken down into a series of smaller targets with several criteria which the subsystem would have to meet to successfully accomplish its high-level functionality.

Ball detection:

The diagram outlines a basic strategy for approaching the detection of balls through a process of highlighting, filtering and calculating. Each ball occupies its own spectrum of colours and in varying brightness conditions this spectrum can be highlighted so that the maximum and minimum pixels of balls can be determined.

Edge detection:

By calculating the luminance space of the image one can observe the brightness of each pixel which after the application of a Sobel filter, can be used to produce the edge space. This edge space can then be highlighted using a threshold.

Communication:

This block facilitates communication between the Arduino ESP connected to the FPGA pins. It will utilise an agreed method to transfer data one way to the Control subsystem. All variables calculated in the analysis of the vision data will be relayed to it.

Each block was designed to be testable by certain criteria so its success can be verified to ensure the entire block is functioning.

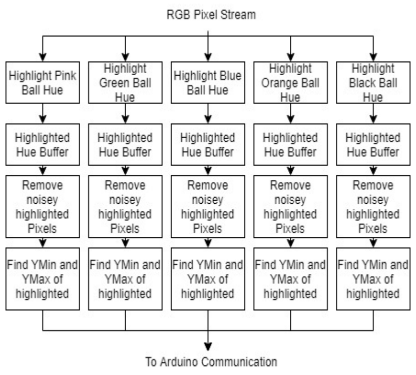


Figure 6: Ball detection overview

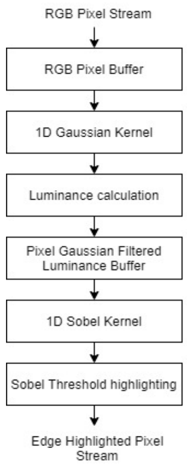

RGB Pixel Stream
RGB Pixel Buffer
1D Gaussian Kernel
Luminance calculation
Pixel Gaussian Filtered Luminance Buffer
1D Sobel Kernel
Sobel Threshold highlighting
Edge Highlighted Pixel Stream

Figure 7: Edge detection overview

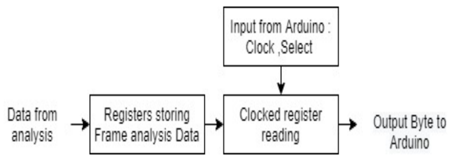

Input from Arduino : Clock ,Select
Data from analysis
Registers storing Frame analysis Data
Clocked register reading
Output Byte to Arduino

Figure 8: Arduino Comms overview

Leaving us with the final design criteria list as such :

1. Detect all five balls independently.
   (a) Consistently highlight pixels belonging to balls in a plain environment.
      i. Highlight the intended colour of the ball and no others.
      ii. Eliminate (Do not highlight) the environment that is not the intended ball.
   (b) Filter/ignore noise data that may exist in the image.
      i. Successfully remove any data that has been highlighted incorrectly when isolating the ball.
   (c) Accurately find the minimum and maximum bounds for balls in the environment.
      i. Ensure associated ymin and ymax are accurate through visual confirmation.
2. Detect obstructions such as unknown visual data.
   (a) Reveal all horizontal edges in an image through highlighting
   (b) Extract meaningful depth data from highlighted edges.
3. Relay information on detected objects to Control subsystem for further processing.
   (a) Instantiate a memory that can store all data from analysis of current frame.
   (b) Relay data gathered and calculated in an agreed format to Control subsystem.
   (c) Check relayed data is accurately received on the Command/Control subsystem.

The development philosophy/process was the following:

We aimed to recursively improve the design and implementation to ensure a robust and flexible subsystem that could adapt to any new criteria, tests or functionalities that the team would agree upon as the project progressed. The development cycle illustrated in figure 9 was followed.

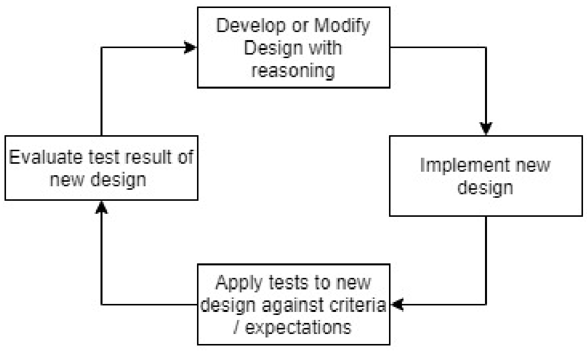

Develop or Modify Design with reasoning
Implement new design
Apply tests to new design against criteria / expectations
Evaluate test result of new design

Figure 9: Subsystem Development Process

The development of our system would all take place upon the DE-10 FPGA and camera provided based on the demo project provided. All development was largely on the EEEImgProc.v and main.c of D8M_Camera_Test file where all Verilog code and C necessary to meet the design requirements was implemented. Any hardware utilised was initialised here or modified in the project QSYS. The goal set was to utilise minimal hardware/implementations to develop the subsystem, meaning that the designs was often implemented in Verilog rather than choosing pre-existing library hardware. This was done to reduce issues when debugging and to ensure not to incorporate functionality that was not necessary.

2.2.4 Command

The rover command module was designed to provide two main functions. Firstly, it had to provide an interface to allow the user to send commands to the rover in different formats and provide information back to the user with regards to the rover's position and sensor data. Secondly, the module had to provide drive commands to the rover based on the user input and on the vision data sent to it from the rover.

The different types of commands that we decided to allow the user to send are as follows:

1. Commands in the form of a coordinate relative to the rovers current position, with the rover deciding the best route to reach the destination.
2. Direct commands such as move forward and turn that the rover follows exactly.
3. Commands to move to a given ball and search for it if its position is unknown.

The information that would be sent back to the user can be broken down as follows:

1. Provide a visual indication of the position of any detected balls relative to the rover position.
2. Provide an indication of the current charge and battery health of the rover.
3. Provide a list of the commands the rover still has to complete.
4. Display the current drive mode that the rover is in.
5. Display depth data for any object in front of the rover.

In order to allow these requirements to be filled a map-based system was decided upon, this would allow the user to easily click on a position that they desired the rover to move to whilst also displaying the current position of any balls within the same format. In order to allow for commands to be entered, a textual input format would also be needed. In addition to ball detection, we also decided to try and find some way of estimating the depth in front of the rover to any obstacle using the difference between two successive frames. It was unclear at the beginning of the project if this would be possible.

When calculating the actual commands to send to the rover, the command module would use three different modes: **Map mode**, where the user clicks on the map to send a coordinate,**command mode** where the user send textual commands and **search mode** where the user can specify a ball to search for.
It was decided that whenever the rover completed a movement sent to it from command, it would send back its change in position and orientation along with any sensor data. This would allow the command module to calculate the new command to send the rover.

2.2.5 Control

The Control subsystem's main task is to allow fast and efficient data exchange between the server and the rover. This process can be broken down into:

- Collect processed data provided by the relevant subsystems
- Send the collected data to the server for analysis
- Receive data back from the server and forward it to the relevant subsystems

The Control subsystem should take advantage of:
Wireless communication Wireless communication is needed to exchange information with the remote server.
Wired communication Local communication is needed to provide local fast information exchange between the various subsystems. Some data can be processed locally and does not need to be sent to the server.

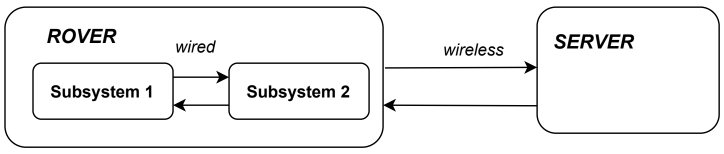


Figure 10: High level rover/Server communication

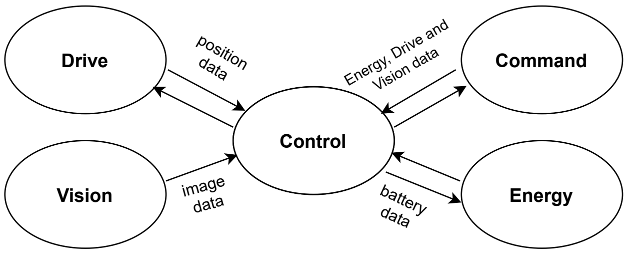


Figure 11: High level subsystem communication

The control subsystem needs to communicate with every other subsystem via a suitable communication protocol. This task can be broken into smaller sub-tasks:

- Communication with the Command subsystem
- Communication with the Drive subsystem
- Communication with the Vision subsystem
- Communication with the Energy subsystem

In particular, the communication between the subsystems should satisfy the following design criteria:

- Communication should be **fast** (i.e. communication delay for transferring data should be less than 100ms)
- Communication should be **reliable** (i.e. corruption of data due to external factors should be minimized)

Furthermore, the communication protocol should be chosen taking into account the amount of information exchanged with other subsystems (payload size) and whether the communication is unidirectional or bidirectional. A summary of these parameters can be found in the following table:

Subsystem	Data exchanged	Payload size	Communication mode
Vision	Processed image data	Large	Unidirectional | Vision $\rightarrow$ Control
Drive	Position data	Small	Bidirectional | Drive $\leftrightarrow$ Control
Energy	Battery data	Small	Bidirectional | Energy $\leftrightarrow$ Control
Command	Energy, Drive and Vision data	Very Large	Bidirectional | Command $\leftrightarrow$ Control

Table 2: Communication Summary

3 Design Implementation

3.1 Drive

Our development and implementation strategy consisted of three main steps:

1. **Functional model:** A simple moving algorithm was coded based on the optical sensor to meet the basic functional requirements. This model was based on a feedback system that could correct the angle error first and the distance error secondly. Both were computed by comparing the optical sensor and the target coordinates. This model was tested in its functionality, and its compatibility with the control unit was checked to have a first general working model.
2. **Optimized models:** More features were progressively added to the main body of the code to optimise the performance, both in terms of time needed to reach a given location and precision of the movement computed and tracked. At this stage, the following additions were considered and tested:
   - Arch trajectory (instead of turning on the spot and moving forward)
   - Time-based model as an alternative to the optical sensor based one (to avoid the error caused by the optical sensor when the quality of the data is low)
   - Variable speed implementation to be decided by the server
   - PI controller for the speed
   - Calculation of the current at the input of the SMPS to compute the power drawn by the drive unit
   - PI controller for the trajectory

   On top of the functional testing, for each further model developed, a series of systematic measurements were conducted to evaluate the actual improvement in performance against the ideal behaviour expected.
3. **Final model decisions:** The different performances were compared against each other in terms of error precision and time needed to compute a given path. Following a general discussion with the other members of the group, the decision on which model to use was based on the priorities agreed in accordance with the needs of the vision and command units.

a. Optical sensor-based model:

This first model was designed with the idea of using the data from the optical sensor to implement a closed-loop controlled system in which the location of the rover is adjusted based on the angle and distance measurement. After the first functional tests of the sensor, we realised that the coordinates sent to the Arduino were represented in polar form: x = angle computed, y = distance computed. We decided to use polar coordinates throughout the program for more clarity. A detail to be noted is that those coordinates represent the movements of the optical sensor, and not of the centre of the rover itself.

Moving algorithm: The moving algorithm implemented consisted of four states (figure 7):

- The OFF state, during which the rover does not move. Here the code is set up and the new distance and angle target are defined. The first time the code will be run, or any other time in which no movement is required, the distance and angle will be both 0. In this case, the state will automatically go to MAX MOVEMENT, waiting for a new instruction.
- The TURN state through which the rover rotates until the angle error between the coordinates of the sensor and the target are smaller than the maximum angle error (set to be 5° for the initial testing). In this state, the motor will turn in opposite directions at a fixed equal speed. The direction of the tuning of each will be in accordance with the sign of the angle to be computed (positive angles the left wheel moves forward and right backwards, vice versa for negative angles). The angle is scaled to be in the range of -180° to 180°.
- The MOVE state, where once the rover is set in the correct orientation, it will start moving forward until the difference between the distance measured by the optical sensor and target distance is smaller than the maximum accepted error (1cm for initial testing)

- The MAX MOVEMENT state: The rover will continue to loop between adjusting the distance and the angle until their respective errors are negligible (ideally 0). At this point, the system will be in the state of maximum movement and both motors will stop. During this, the angle and distance computed (from the optical sensor) are sent to the server. The new instruction is received and scaled to the unit and range of the distance and angle. After the transmission and reception are completed, all the relevant variables are reset, and the state is set to OFF.

State	Motors	Task
OFF	stop	Set up instructions
TURN	Maximum speed, opposite direction	Reach the angle target
MOVE	Maximum speed, same direction	Reach the distance target
MAX MOVEMENT	stop	Send location and receive new instruction

Table 3: Moving Algorithm

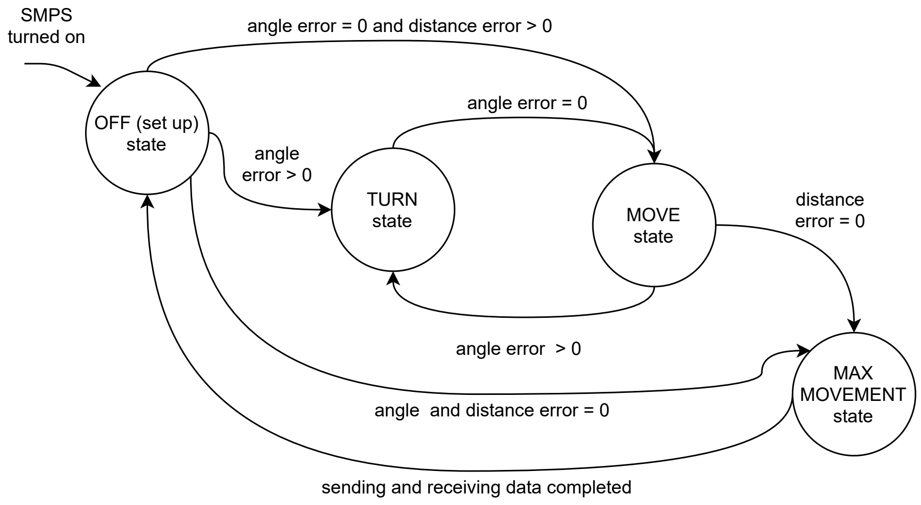


Figure 12: Moving algorithm state machine

Variable speed: The implementation of variable speed seemed to be an important feature that could be used to enhance the performance of the movements. More freedom of choice would be available on the command side of the rover. This was implemented by tuning the value of the PWM (Pulse Width Modulation) input from the Arduino to the gate of the MOSFETs of the H-bridges. Instead of using a digital wire, an analogue wire with values from 0 to 255 was used to control the voltage level perceived by the motor and, consequently, their speed.

Arch trajectory: The initial idea was to implement a different type of trajectory. An arch movement had been implemented to substitute the “turning on the spot and the then moving straight” procedure. Hence the rover could turn while moving. This model was then tested and, although the functionality was correct, the loss in speed due to the slowing of the wheels (against a constant maximum speed for the previous model) did not justify it’s implementation. The original model was already extremely precise and considerably easier to control and track. It was therefore decided not to pursue this model.

PID controller: A model of a PID controller was designed and included in the main code. This was implemented with the idea to better control the precision of the movements when turning or moving forward. The controller for the angle correction compared the angle measured and the target angle. It would then tune the speed of the turn accordingly. The distance controller worked in the similar way, but adjusting the speed used to move forward instead. The expected behaviour of this model was to progressively increase the speed to the maximum and then decelerate when approaching the target to better control the precision of the arriving point. The gains of the controller were progressively tuned, with exception of the differential one, which did not influence the performance significantly during tuning.

Another version of the PI was also considered to control the trajectory instead (making the rover go backwards in case of overshoot). However, this would have not increased the performance of the rover since the feedback given by the optical sensor was observed to be working fast enough to not cause any considerable overshoot. Furthermore, the logic of the optical sensor-based model already worked so that if an error in the angle emerges, it will be automatically be corrected by a turn in the opposite direction.

b. Time-based model:

After comprehensive testing of the optical sensor-based model was concluded, there were certain limitations in the optical sensor that could not be avoided: the quality of the data created by the sensor would decrease beyond an acceptable level when the surfaces on which the rover traveled were too uniform, translucent or of a light colour. This is because the sensor is not able to properly detect the movement computed. Therefore, it was decided that there is need of an alternative model to be used in these edge cases. This model would not use the feedback of the sensor to calculate the distance and angle to be computed. Instead it would calculate a time of execution for the turning and moving forward. These two values are calculated in the code based upon the maximum speed of the wheels measured for the rover (60rpm) and the distance or angle to be computed. Being aware that this a more empirical and imprecise way of proceeding, this algorithm is to be considered as only a back-up model for when the quality of the optical data is extremely low and could lead to major errors in the feedback.

Final model:

In accordance with the results from the testing, a final more optimised model was created integrating all the successful features in a single program. The logic of the code is summarised in figure 8 and includes the following elements:

- Error detection and correction: The optical sensor feedback and PI controller check that the direction is right and adjusts the movement of the wheels accordingly. This leads to a maximum error within a margin of 5° and 5mm. In general the error was within 10%.
- Multiple modes to move depending on the quality of the optical data: The optical sensor feedback, if the quality is above a certain level (15 stars are usually a recommended level at which the data is reliable; however, by measuring the performance, 10 stars had been observed to still lead to good data quality)
- Speed value of the wheels decided by the server within the range of 5-60rpm. By default the maximum speed is used.
- Input current drawn from the energy unit is computed and the value send to the control unit in order to calculate the power consumed. The input current of the SMPS was computed by the measurement of the output current by the current sensing resistor and the formula $I_{in} = \frac{V_{out}I_{out}}{V_{in}}$ (for more see energy unit)
- Sending and receiving section implemented via SPI (see control unit)

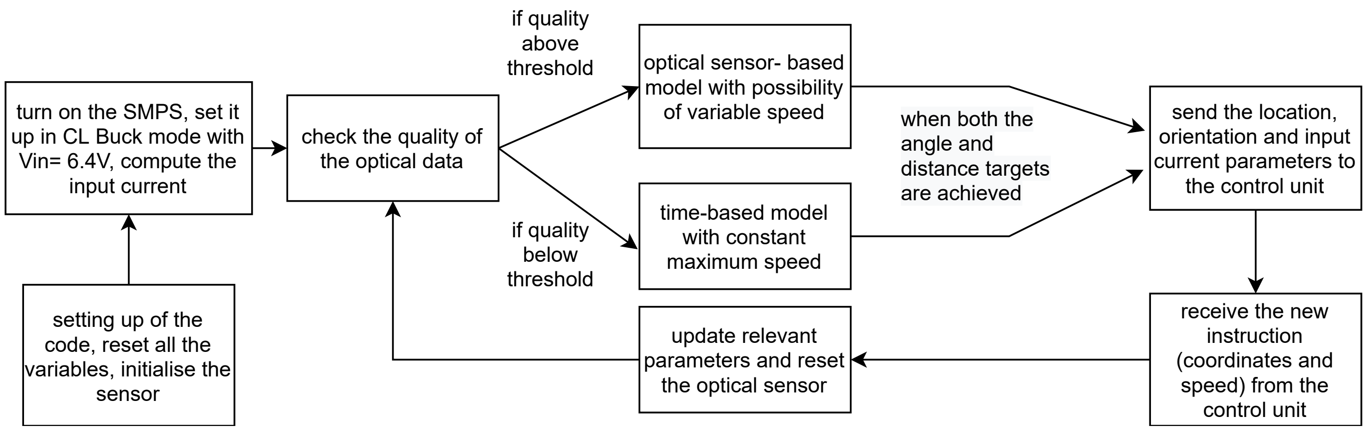

if quality above threshold
optical sensor- based model with possibility of variable speed
turn on the SMPS, set it up in CL Buck mode with Vin= 6.4V, compute the input current
check the quality of the optical data
when both the angle and distance targets are achieved
send the location, orientation and input current parameters to the control unit
time-based model with constant maximum speed
if quality below threshold
setting up of the code, reset all the variables, initialise the sensor
update relevant parameters and reset the optical sensor
receive the new instruction (coordinates and speed) from the control unit

Figure 13: Final logic of the drive unit

Furthermore, a few adjustments were made to enhance the performance of the rover as a whole. Taking into account the need of the vision unit specifically: The idea is that the model has to be designed keeping in mind the highest goal of the Mars Rover itself, i.e. to move around in an unknown environment mapping the surrounding. For this reason, we thought that the quality of the images, movements and the precision of the data from the mapping process had to be prioritised over the smoothness of the movements themselves or the simplified communication feedback. Therefore, we decided make the communication between the control and drive unit more frequent by letting the rover send data every 50 millimetres or 20° computed. This has two main benefits: Firstly, the mapping and the obstacle detection given by the vision unit is more accurate since the server has a frequent feedback on the current location; Secondly, if any sudden change in the surrounding environment occurs or the error from the optical sensor accumulates throughout the instructions, the command has the time to process it and adjust the optimal trajectory immediately. This design decision was confirmed after testing the rover and measuring that the delay given by the more frequent transmission of data is very contained (Under 100ms).

3.2 Energy

3.2.1 Arduino

Data Collection

Upon initial testing of the Arduino, it became clear that the analogue inputs fluctuate significantly. To mitigate this for both the current and voltage, a 'moving average' filter was implemented which took the 1000 values before the slow loop and summed them in a cumulative variable. This was then divided by 1000 to get the average.

Charge Strategy

To get the most capacity out of a single cell, a CC/CV (Constant Current/ Constant Voltage) charge strategy was decided upon. This involves charging a cell with a steady current until the maximum voltage is reached. Both the charging current (250mA) and maximum voltage (3.6V) are dictated by the data sheet of the cell in use (Appendix A.1). The cell is then charged using a constant voltage whereby the current slowly decreases as seen during testing in figure 41. This ensures that the maximum cell voltage is not surpassed while still trickle charging the cell to near maximum. The current cut-off was set at 0.1C (50mA) so that the cell would not take an excessive amount of time trickle charging. A PID was used to control both the voltage and current. This changed to a combination of MPPT and voltage PID when the charging station was involved.

SOC

Measuring the state of charge is difficult in real time due to a range of factors, including fluctuations in temperature and the degradation of the cell as the number of charge-discharge cycles increases. The computation power of the Arduino is also another factor to consider and therefore an Enhanced Coulomb Counting method [1] was decided upon. This will work out the amount of charge going in/out of the cell and thereby the change in the SOC. TO enable this an initial value of the SOC will also be needed. Given the direct relationship between the SOC and OCV (Open Circuit Voltage), a lookup table was extracted from early cell characterisation seen in figure 43. This is also tightly integrated with SOH Maintenance to allow for auto calibration and integration with the passive balancing algorithm. (Figure 14).

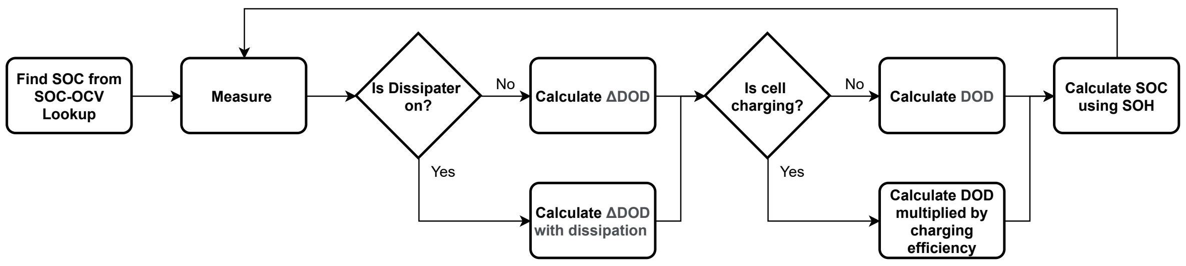

Find SOC from SOC-OCV Lookup
Measure
Is Dissipater on?
No
Yes
Calculate ΔDOD
Calculate ΔDOD with dissipation
Is cell charging?
No
Yes
Calculate DOD
Calculate DOD multiplied by charging efficiency
Calculate SOC using SOH

Figure 14: Top level SOC finding algorithm

$$SOC = 1 - DOD \qquad (1) \qquad SOH = \frac{Q^{Max}_{Discharge}}{Q_{Rated}} \qquad (2) \qquad \eta_c = \frac{Q^{Max}_{Discharge}}{Q^{Max}_{Charge}} \qquad (3)$$

In (1), the state of charge can be represented as a decimal or percentage. It is a measure of the current capacity of the cell. The state of health (2) is defined as the maximum discharge capacity compared to the rated capacity of the cell. It can also be represented as a percentage of the rated capacity. The charge efficiency (3) is used to create a more accurate Coulomb counting algorithm whereby the increased capacity needed during charging is accounted for as seen in figure 14.

$$\Delta DOD = \frac{I_{Bat}}{3600 * SOH * Q_{Rated}} \qquad (4) \qquad\qquad DOD_{(k+1)} = DOD_{(k)} - \eta \Delta DOD \qquad (5)$$

In discrete one second intervals, ΔDOD (4) is seen to be the amount of amp-hours in the sampling period compared to the maximum discharge capacity which has been expanded to include the SOH. During charging, the next value of depth of discharge (5) decreases with an $\eta = \eta_c$. Conversely, the DOD increases during negative current (Out of cell) with $\eta = 1$.

SOH Maintanance

The state of health decreases as the number of charge-discharge cycles begins to accumulate. Hence it is important to track ensuring an accurate Coulomb Counting algorithm. This can be done by calibrating the SOH and η_c during a full charge/discharge (15). This was done during the characterisation of each cell. Ideally this would happen after 1-2 full charge-discharge cycles which would allow the coulomb count to zero itself. To prolong the life of a cell in a battery pack, an SOH maintenance strategy must be employed. This is needed to prevent a single, smaller cell, from restricting the larger cells from charging to maximum capacity. The smaller cell would then degrade at a faster rate than the others leading to a lower overall battery capacity. This strategy involves developing a system to keep the various SOCs at around the same level. There are a range of balancing strategies open to us with the simplest and most effective being passive cell balancing. This involves using dedicated dissipation resistors through which the individual cells can discharge to lower their respective SOC to that of the whole battery pack. This seemed like the most obvious option given the already implemented dissipation resistors in the battery PCBs. The balancing algorithm was then idealized shown by the flowchart in figure 16. It should be noted that the cells should stop dissipating when they are within 3% of each other due to the SOC only being an estimation.

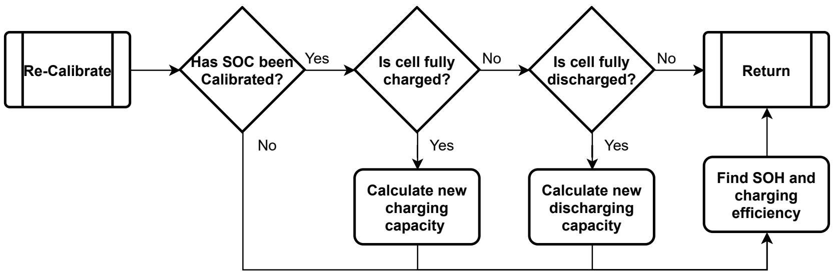

Re-Calibrate
Has SOC been Calibrated?
Yes
No
Is cell fully charged?
No
Yes
Is cell fully discharged?
No
Yes
Return
Calculate new charging capacity
Calculate new discharging capacity
Find SOH and charging efficiency

Figure 15: Re-calibration algorithm

$$Q^{Max}_{Charge_{(k+1)}} = \frac{SOC - SOC_{Init}}{DOD_{Init}} Q^{Max}_{Charge_{(k)}} \qquad (6) \qquad\qquad Q^{Max}_{Discharge_{(k+1)}} = \frac{SOC_{Init} - SOC}{SOC_{Init}} Q^{Max}_{Discharge_{(k)}} \qquad (7)$$

Equation 6 states that the new value of the maximum charge capacity is the ratio of the actual DOD at full capacity compared to the initial estimated DOD multiplied by the old value for the maximum capacity. Hence, if the real DOD was lower than estimated, a smaller value of charge capacity would be calculated. This can then be used to find a new η_c (3). Similarly, (7) calculates a new discharge capacity by using the ratio of the actual DOD and the initial SOC. This is then applied to the previous value of the discharge capacity. Through this, a re-calibration of the SOH (2) and η_c (3) can be processed.

PV MPPT Algorithm

PV panels produce a non-linear voltage-current curve which can vary significantly with temperature and luminosity. This multiplied by the changing efficiency of the SMPS leads to a non-linear, dynamic power curve (47) which needs to be taken advantage of in order to get the maximum power output. It was decided that Incremental Conductance would be used in the MPPT algorithm due to low fluctuations at the peak. The strategy works by making a linear estimation of the gradient using the past sample value as seen in (8) and following the flowchart seen in figure 17. A positive gradient would mean that the system is on the left hand side of the peak and vice versa for a negative gradient. The function would then change the duty cycle accordingly to change the impedance of the power supply as

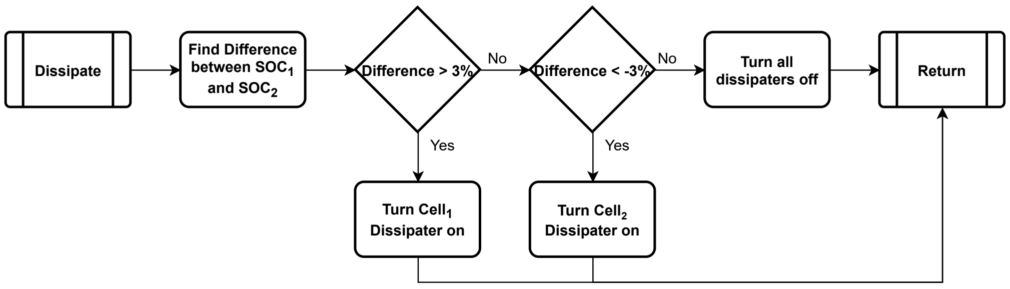

Dissipate
Find Difference between SOC_1 and SOC_2
Difference > 3%
No
Difference < -3%
No
Turn all dissipaters off
Return
Yes
Yes
Turn $Cell_1$ Dissipater on
Turn $Cell_2$ Dissipater on

Figure 16: Dissipation algorithm

seen by the PV panel. These duty cycle 'steps' would be larger for steeper gradients. The MPPT program also tracks if the irradience has changed which would result in no change in voltage but a change in current.

$$\frac{dP}{dV} = \frac{d(VI)}{dV} = I + \frac{dI}{dV} \approx I + \frac{\Delta I}{\Delta V} \tag{8}$$

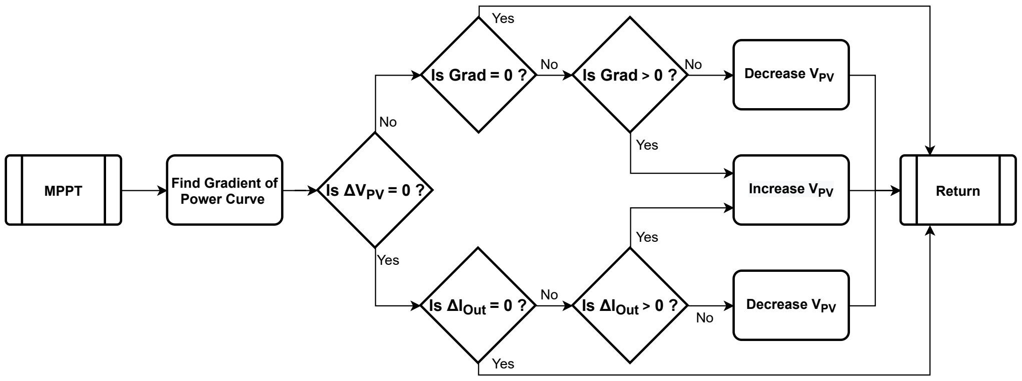


Figure 17: Incremental Conductance Algorithm Source: Adapted from [2]

Communication

Communication between Energy and the rest of the subsystems is important in the reliability of the rover. The SOC, SOH, Rover Range and Low Battery Flag will be sent to control which will relay these to Command. The SOC will be that of the lowest cell and the SOH will be the average of both cells. To get an accurate SOC and Rover Range, the overall current drawn and speed of the rover must be received by Energy. Lastly, Command must send a Charge Flag to indicate that the rover is connected to the solar charging station.

Cell Safety

The use of LiFePO cells requires stringent safety protocols. Based upon the cell specification (Appendix A.1) and the maximum values in table 1, the energy subsystem will completely shut down and refuse to give power unless the problem is solved. This will be done with the use of relays which will disengage the cells.

Rover Range

The rover range is found using (9). This will provide an estimate given the speed, SOC and current draw. The equation will be adjusted according to the SOH; keeping the estimate accurate as the cycle count of the battery pack increases.

$$Range = \left(\frac{Q^{Max}_{Discharge}}{I_{Out}} 3600 \right) v_{rover} = \left(\frac{SOC\ SOH\ Q_{Rated}}{I_{Out}} 3600 \right) v_{rover} \tag{9}$$

3.2.2 SMPS

Initial Top Level Design

How the Energy subsystem is integrated within the design of the rover is more of a hypothetical problem. The processing on board requires a reliable 5V supply which would require the use of another SMPS to step down the voltage of the cells. In our case, the battery is only used to power the motor control through a connection to Port A of the Drive SMPS as seen in figure 29. The drive current is needed to work out the actual current going through the battery for the SOC and range calculations.

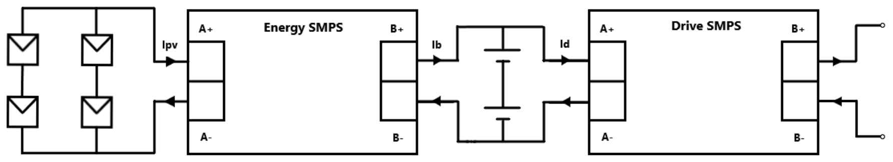


Figure 18: Preliminary design for connection to drive subsystem

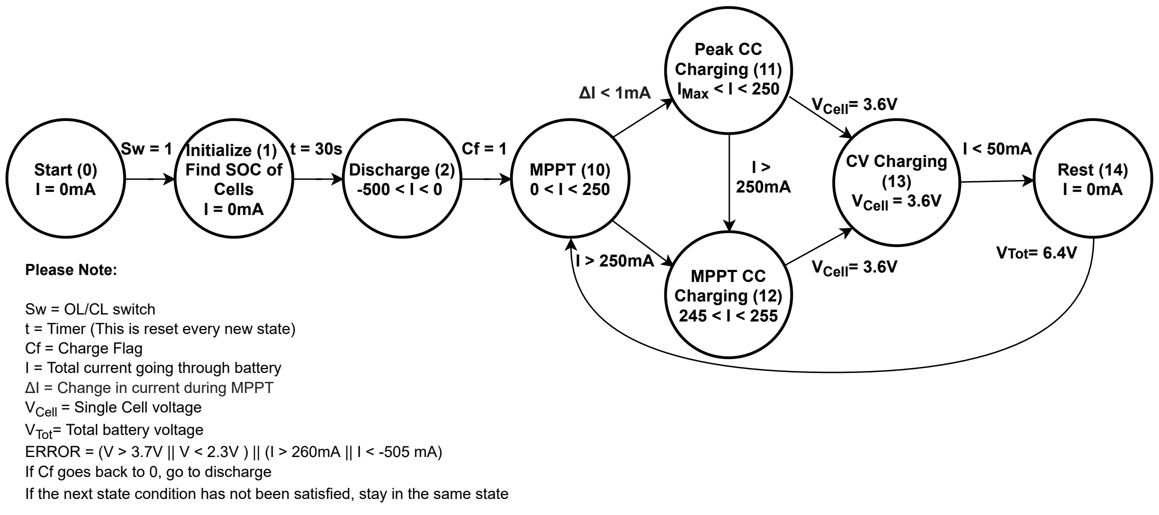
3.2.3 Battery Pack

It was decided that safety and good cell maintenance would be a priority when deciding on the configuration of the cells. This led to the series design seen in figure 29. The series battery pack has the advantage that one can always monitor the current going through each cell which can then be used to ensure the battery is not damaged through over charging/discharging. Unfortunately, more than two cells in series is not possible with the SMPS in synchronous mode due to a $\approx 8V$ limit on port A. This is a limitation of the PMOS MOSFET's Gate threshold voltage and more cells cannot be added as the Drive SMPS requires a buck configuration (Hence the PMOS cannot be set as a diode). During series operation, constant voltage mode would be set up on each cell individually to make sure that no cell is overcharged.

3.2.4 PV Panels

One can see the classic characteristic PV curves on figure 47. The PV specification (Appendix A.2) states that PV panel operates at approximately 5V. As seen in figure 29, a 2x2 design ensures that the maximum power point voltage will be above that of the battery pack at around 10V. Like before, the input voltage limit of the SMPS will affect this arrangement. During testing it worked but a series configuration with 4 panels is impossible as the Buck mode is required from the SMPS.

3.2.5 Final State Machine


Start (0) I = 0mA
Sw = 1
Initialize (1) Find SOC of Cells I = 0mA
t = 30s
Discharge (2) -500 < I < 0
Cf = 1
MPPT (10) 0 < I < 250
ΔI < 1mA
Peak CC Charging (11) I_{Max} < I < 250
I > 250mA
I > 250mA
MPPT CC Charging (12) 245 < I < 255
V_{Cell}= 3.6V
V_{Cell}= 3.6V
CV Charging (13) V_{Cell} = 3.6V
I < 50mA
Rest (14) I = 0mA
V_{Tot}= 6.4V

Please Note:

Sw = OL/CL switch
t = Timer (This is reset every new state)
Cf = Charge Flag
I = Total current going through battery
ΔI = Change in current during MPPT
V_{Cell} = Single Cell voltage
V_{Tot}= Total battery voltage
ERROR = (V > 3.7V || V < 2.3V) || (I > 260mA || I < -505 mA)
If Cf goes back to 0, go to discharge
If the next state condition has not been satisfied, stay in the same state

Figure 19: Final State Machine

The final state diagram can be seen in figure 19. It should be noted that the various functions providing special capabilities (Dissipation, Communication, etc) stated above run every cycle and so have not been included. Note the dual charging stage, if the MPPT reaches 250mA while trying to get to the peak, it stops and keeps the current approximately at that value by using MPPT and inverse MPPT. Otherwise, the peak of the MPPT is used meaning that it is in operation constantly. It then comes together with the voltage PID. This can be done as the current curve leading up to the 250mA will be linear if the peak is higher than the charging current. Similarly, this will also work if the peak is lower than 250mA as the PID will always decrease. A final circuit diagram can bee seen in Appendix A.1

3.3 Vision

3.3.1 Edge Detection Implementation:

Edge detection is used to provide more advanced data for obstacle avoidance and route planning. It is also used to corroborate the hue analysis that occurs in ball detection. This provides us with a more detailed image of the environment.

Video data is received from the camera as a continuous stream of RGB pixels or frame packet information. The edge detection processes the video stream to highlight the object outline. Each pixel received is in an RGB format where R,G and B are represented by a byte having a value between 0 and 255.

The first step in the edge detection process is to calculate the luminance of each pixel from its RGB values. Luminance or Luma is an important photo-metric measure describing the brightness of a pixel. It is often used to produce an achromatic image giving us a more meaningful visual space to extract information from. An approximation for the digital CCIR 601 [3] video standard was used. This value will be between 0 and 255 and be a weighted sum of the R, G, B values of the pixel.

$$Y_{601} = 0.299R + 0.587G + 0.114B \qquad (10) \qquad\qquad Y_{Approx} = 0.375R + 0.5G + 0.125B \qquad (11)$$

An approximation is used as representing specific fractions in Verilog introduces a large amount of complexity. Therefore an approach using a series of additions and bit shifts in Verilog was used . It was decided that the small reduction in accuracy was acceptable for our application as later a threshold would be applied which would highlight low and high luminances.

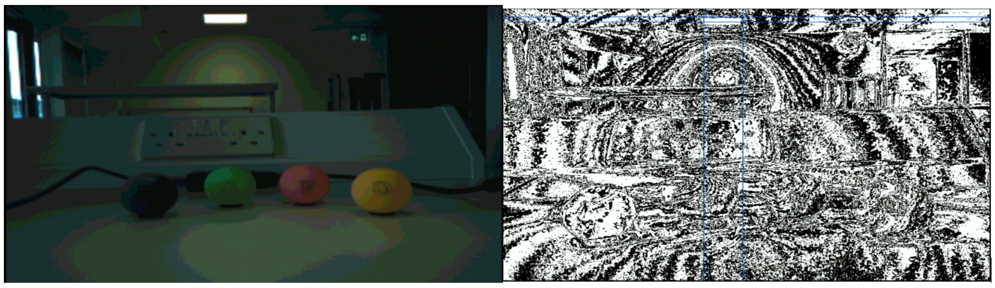


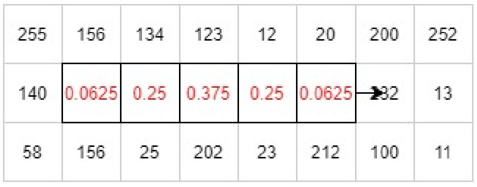
Figure 20: Example Luminance image

Following the review of the luminance space image, a large amount of speckled noise-like data was observed. Gaussian filtering was used to remove this data to give a smoothed luminance space. This helped in eliminating erratic brightness data that may be present in the luminance space. The Gaussian filtering was accomplished through the convolution of a 1D Gaussian kernel with a series of horizontal pixels and returning a Gaussian value which is the convolution product for that specific pixel. See reference [4]

$$\begin{bmatrix} 0.006 & 0.061 & 0.242 & 0.383 & 0.242 & 0.061 & 0.006 \end{bmatrix} \quad (12) \quad \begin{bmatrix} 0.0625 & 0.25 & 0.375 & 0.25 & 0.0625 \end{bmatrix} \quad (13)$$

Figure 21: 1D x component Gaussian Kernel (12) and approximated version (13)

To accomplish convolution the current and previous four values of pixel luminances are stored in a buffer called *Gaussian_buffer*. After storing these values, the sum of weighted values is calculated in the buffer. Weighting is accomplished through bit shifting each luminance value according to its position. The weighting values are again approximated as shown above to an acceptable degree of accuracy. Thus the kernel moves through the image data horizontally. The sum of weighted luminances is then returned as a variable named *Gaussian* in the new stream.



255	156	134	123	12	20	200	252
140	0.0625	0.25	0.375	0.25	0.0625	→32	13
58	156	25	202	23	212	100	11

Figure 22: Gaussian buffer example

After this convolution we now have a Gaussian filtered luminance video stream which is ready for edge detection.

Two methods were attempted when implementing edge detection on the Gaussian filtered luminance video stream. The first being a 3x3 kernel applied across and the second a 1X5 horizontal Sobel filter. Both methods had their benefits and drawbacks which will be discussed.

Method 1 - 3X3 Kernel - Implementation in Full_Sobel branch:

To apply the following kernel it is necessary to have on access the values under the matrix, this stretches across three rows of pixels and three columns. These values are not accessible from the stream so we must create a buffer which stores the previous pixels of the stream, making them accessible for the matrix convolution. An attempt was made to implement this buffer with a bank of registers defined as such:

```
reg [7:0] luminance_buffer[2:0][(IMAGE_W-1):0]
```

This stores the RGB values for all pixels in the three rows necessary. However this implementation took exceedingly long to compile. (A version using a shift register IP block in the QSYS was then attempted. However before the shift register was fully implemented, method two was chosen to implement the edge detection instead.)

Thus the matrix convolution is then calculated as such:

```
assign GXSOBEL = (1 * ImageMatrix[2][0] ) + (-1 * ImageMatrix[2][2] ) +
(2 * ImageMatrix[1][0] ) + (-2 * ImageMatrix[1][2] ) +
(1 * ImageMatrix[0][0] ) + (-1 * ImageMatrix[0][2] )

assign GYSOBEL = (-1 * ImageMatrix[2][0] ) + (-2 * ImageMatrix[2][1] ) +
(-1 * ImageMatrix[2][2] ) + (1 * ImageMatrix[0][0] ) + (2 * ImageMatrix[0][1] ) +
(1 * ImageMatrix[0][2] )
```

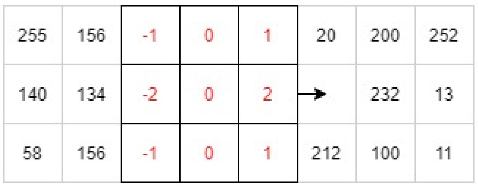
Finding the square root of the summation is not immediately possible in verilog so a SQRT function was imported from the web, referenced here [5]. (However this way is not used in the final implementation). The value GSOBEL was then calculated from GXSOBEL and GYSOBEL in the following way:

```
assign GSOBELSQUARED = ((GYSOBEL)*(GXSOBEL)) + ((GYSOBEL) * (GYSOBEL))
assign GSOBEL = sqrt(GSOBELSQUARED)
```

$$G_X = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad (14) \qquad G_Y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (15) \qquad G = \sqrt{G_X^2 + G_Y^2} \quad (16)$$

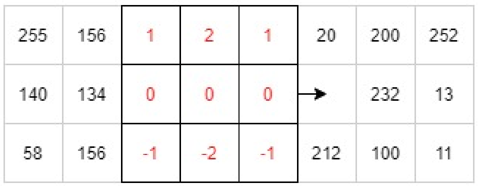
Figure 23: 3X3 Sobel Kernel for horizontal and vertical and final

This method was not chosen due to a few issues. Firstly the implementation reveals both horizontal and vertical edges in the image, however it was discovered only horizontal edges are of concern to our rover for calculating obstacle proximity. This is because horizontal edges highlight the approaching bottom or top edge of obstacles which our avoidance algorithm uses to calculate avoidance (Further detail of why this is can be found in Command). As such it was unnecessary to calculate GX and GSOBEL as they would not be used in analysis. Therefore, to reduce the complexity of design and any usage of unnecessary FPGA hardware method 2 was followed. Further to this, the memory buffer/shift register used was very large in the design which led to considerable delay in compilation and was not an optimised implementation.



255	156	-1	0	1	20	200	252
140	134	-2	0	2	→	232	13
58	156	-1	0	1	212	100	11

Figure 24: Horizontal Sobel filter example



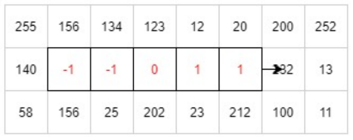
255	156	1	2	1	20	200	252
140	134	0	0	0	→	232	13
58	156	-1	-2	-1	212	100	11

Figure 25: Vertical Sobel filter example

Method 2- 1X5 Horizontal Kernel - Implementation is final master.

The second method is relatively simpler than the first as a 1D horizontal kernel is used which means that only the current and four preceding pixel RGB values must be stored for the convolution calculation. This method requires the camera to be orientated horizontally but results in far a simpler buffer and convolution, implemented as such (where Gaussian is the filtered stream pixel):

```
reg [10:0] luminance_buffer[4:0];
always@(posedge clk) begin
    luminance_buffer[4] <= {3'd000,Gaussian};
        luminance_buffer[3] <= luminance_buffer[4];
        luminance_buffer[2] <= luminance_buffer[3];
        luminance_buffer[1] <= luminance_buffer[2];
        luminance_buffer[0] <= luminance_buffer[1];
end
assign GSOBEL = luminance_buffer[0] + luminance_buffer[1]
           + ~luminance_buffer[3] + ~luminance_buffer[4];
```



255	156	134	123	12	20	200	252
140	-1	-1	0	1	1	→232	13
58	156	25	202	23	212	100	11

Figure 26: 1D Sobel kernel visualisation

Horizontal Camera Orientation and Highlighting

Due to the operation of the horizontal Sobel filter the subsystem detects vertical edges in the image. This is because it is moving through the image horizontally and highlighting an edge that may occur in its horizontal buffer which is always a vertical edge in the image. However in our analysis we are only interested in horizontal edges. A simple solution is to rotate the camera 90° so that the buffer moves through the image vertically (respective to the video) and highlights horizontal edges as desired.

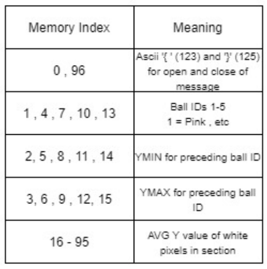
Analysis of Highlighted Edge Video Stream

The edge video stream is then analysed in the following way: First the height of the image was divided into 80 vertical sections, each section has a sum and count which is reset when a new section is entered. When a white pixel is detected in this vertical section, the vertical position of the white pixel is added to the sum and the count is incremented. At the end of the section the sum is divided by the count and the result is inputted into the memory array for communication to the Command subsystem. When the final section is complete the sum and count are reset for the next frame. This process is done for only part of the image excluding data at the top of the camera as it is visually too far from the rover.

3.3.2 Communication to Control Subsystem:

To communicate with the Arduino, the system is connected via 3 GPIO pins which are used to implement a basic SPI communication protocol. The pins are select, MISO and clkArduino. When select is low the Arduino begins to request data from the FPGA; when select is high is is not requesting. The MISO line is the avenue which the bit stream of data from the memory array will flow. The memory array is a bank of 98 registers storing 1 byte each. The first memory register is loaded with an open message symbol '{' in ascii 123 and the 98th register is a close message symbol '}' in ascii 125. The memory bank is read from one register at a time from start to finish where on each block pulse the next bit is read. The byte and bit addresses are updated as we move through the registers and these counts are reset whenever the select line goes high. The implementation can be seen below and is relatively simple:

```
always @(posedge clkArduino) begin
if ( selectArduino == 0)begin
  miso  <= (counterByteAddress < 7'd100)
  ? mem[counterByteAddress][counterBitAddress] : 0 ;
  counterBitAddress <= counterBitAddress + 3'b1 ;
  counterByteAddress <= counterByteAddress
  + (counterBitAddress == 3'b111);
end
if ( selectArduino == 1)begin
  miso <= 0 ;
  counterBitAddress <= 3'b0 ;
  counterByteAddress <= 7'b0;
end end
```



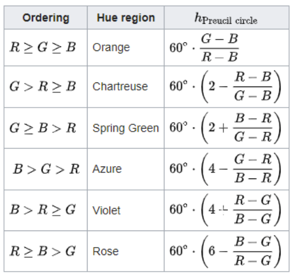
Memory Index	Meaning
0 , 96	Ascii '{ ' (123) and '}' (125) for open and close of message
1 , 4 , 7 , 10 , 13	Ball IDs 1-5 1 = Pink , etc
2, 5 , 8 , 11 , 14	YMIN for preceding ball ID
3, 6 , 9 , 12, 15	YMAX for preceding ball ID
16 - 95	AVG Y value of white pixels in section

Figure 27: Memory map

3.3.3 Ball Detection Implementation:

Each ball is detected using the same process using modified thresholds. The process begins by converting the RGB space into a highlighted HUE space representation. According to the HUE regions we assign either a white or black value to the pixel. For example, if the RGB values of the pixel lie in the orange HUE region we set the HUE_Orange value to 0 for black. If it does not lie in this region we set it to 255 for white. This process is implemented as such:

```
reg  [7:0]HUE_Orange ;
always@(posedge clk) begin
if(R >= G)begin
        if(R >=  B)begin  // R > G  && R > B, R IS MAX
                if (G >= B )begin // R MAX B MIN
                        HUE_Orange <=  8'd0;
                end
                else begin // R MAX G MIN
                        HUE_Orange <=  8'd255;
                end
        end
end
//Implementation is cut here as too long but can be
            //found in appendix
```



Ordering	Hue region	$h_{\text{Preucil circle}}$
$R \geq G \geq B$	Orange	$60^\circ \cdot \frac{G-B}{R-B}$
$G > R \geq B$	Chartreuse	$60^\circ \cdot \left(2 - \frac{R-B}{G-B}\right)$
$G \geq B > R$	Spring Green	$60^\circ \cdot \left(2 + \frac{B-R}{G-R}\right)$
$B > G > R$	Azure	$60^\circ \cdot \left(4 - \frac{G-R}{B-R}\right)$
$B > R \geq G$	Violet	$60^\circ \cdot \left(4 + \frac{R-G}{B-G}\right)$
$R \geq B > G$	Rose	$60^\circ \cdot \left(6 - \frac{B-G}{R-G}\right)$

Figure 28: HUE region table [6]

However some noise/unexpected RGB values exist which disrupts the bounding that occurs later. Thus to filter stray highlighted pixels that speckle across the image, a very simple filtering process is used. A buffer is used to store the current and previous 4 highlighted HUE values of the pixels (using the buffer process shown below). The top bit of these values is then examined: if it is 0, it will be highlighted and if it is 1, it will not. Therefore, if all stored pixel HUE values are 0 then all are highlighted and it is accepted that it is not noise but a consistent region of orange/black/pink hue. As such the pixel is re-highlighted as white (implying non noise) and black if it is noise. This leaves us with a image like in figure 29:

```
wire  Orange_detect  ;
assign Orange_detect = ~HUE_Orange_buffer[4][7]
                    && ~HUE_Orange_buffer[3][7]
                    && ~HUE_Orange_buffer[2][7]
                    && ~HUE_Orange_buffer[1][7]
                    && ~HUE_Orange_buffer[0][7]   ;
wire [23:0] HUE_Out_Orange  ;
assign HUE_Out_Orange = {Orange_detect, ... ,Orange_detect} ;
```

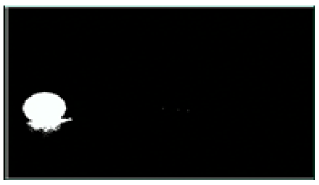


Figure 29: Highlighted detected image (Orange)

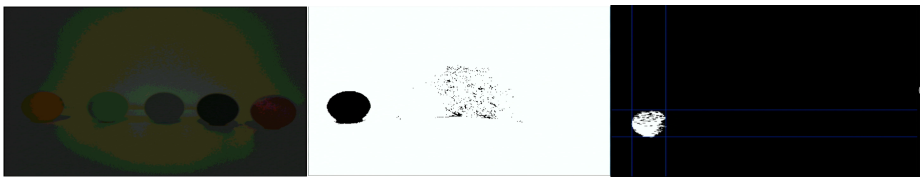


Figure 30: Highlighted Hue space before and after noise filtering

The simple bounding box process is now applied on this filtered highlighted hue space. The values of Ymin, Ymax, Xmin, Xmax for each ball are updated at the eop for each frame. This method works relatively robustly but can be affected by high contrast conditions. As such the camera is kept in a low contrast/low brightness mode when bounding the boxes which seems is a preferable environment.

3.4 Command

3.4.1 Overview of Design

The overall design of the command module has it split into two main components: the front end that the user interacts with in their browser and the back end that runs on a server in the cloud. The server is responsible for making all of the decisions for the rover as it can be given as much computing power as is required, which means more complex decisions can be taken quicker. The rover itself will send information about its change in polar coordinates after every movement it completes to the server and it will then use the information it has to send a new movement to the rover. The user interface will concurrently request data from the server about the rover to present to the user and will send user commands to the server to allow it to make movement decisions.

The user interface allows for three different methods for controlling the rover. The first is the most simple and consists of sending direct commands, such as move forward *x* steps, turn *x* degrees or stop. When the rover is executing these commands it will not avoid any obstacles in its way. The second means of control is the sending of coordinates to the rover. This is done through the user clicking on a map of the rover's surroundings to give it a desired location resulting in the rover then moving to this location avoiding any obstacles on the way. The final method of control allows the user to specify a ball that the rover should find. It will head to this ball if its location is known else it will search its surrounding until it does. In this mode as well as in the coordinate mode the rover will avoid obstacles and will automatically head back to the charging station if low on power. The flow diagram in figure 31 demonstrates the process the command system uses to communicate with the rover.

The server was built using Python and used flask library for handling http request. This decision was made as Python has extensive libraries for mathematics meaning calculating the commands for the rover would be easier. The flask library was used as it is very lightweight and does not come with a plethora of extra features that would not be used. The user interface was made using flutter - this allowed the app to be compiled as either a desktop app or as a website without changing any code. The python server and the static files for the interface were deployed on the Digital Ocean App Platform. This means of deployment was chosen as it is extremely user friendly which has led to the shortening of development time. It also deals with services such as SSL certificates automatically.

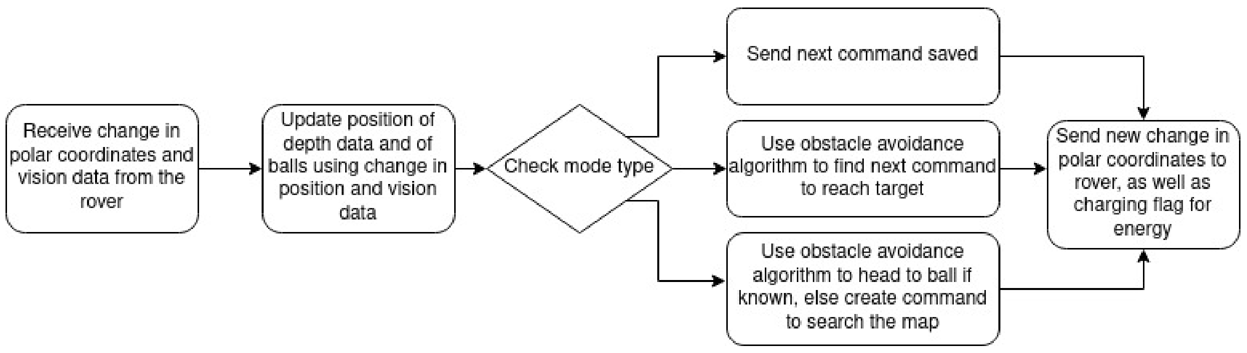

Receive change in polar coordinates and vision data from the rover
Update position of depth data and of balls using change in position and vision data
Check mode type
Send next command saved
Use obstacle avoidance algorithm to find next command to reach target
Use obstacle avoidance algorithm to head to ball if known, else create command to search the map
Send new change in polar coordinates to rover, as well as charging flag for energy

Figure 31: Top level overview of the command system

3.4.2 Receiving data to and from the rover

The data received from the rover is in the form of a Json file. These Json files used for each communication direction have a schema that was carefully designed with the control system so as to ensure that both systems could be worked on independently as the input and outputs between each system were well defined. The schemas themselves went under many revisions as the design was changed over time. The rover itself communicated with the server using a single POST request, whilst the user interface communicated using many different GET and POST requests. The interface would continuously poll the server for new information to always stay up to date.

3.4.3 Updating Map Information

Once a movement was received from the rover the server's current knowledge of the map needed to be updated using this new information. The three main changes that needed to be made were updates to the depth information, ball information and the target location.

3.4.4 Updating Ball Information

The position of any balls known to the server were stored in polar coordinates. These coordinates would always be relative to the rover's forward direction. This meant that the first step in updating their positions was to convert their positions to Cartesian coordinates. Then the polar changes received from control for the last movement could also be converted to Cartesian and the resulting dx and dy be applied to the ball positions, before finally converting back to polar. These new polar coordinates would also have to be rotated in order to be with respect to the new forward direction of the rover. All polar angles were taken from the clockwise forward direction of the rover. The block diagram for this logic is shown in figure 32. In order to test this logic in python, the library "matplotlib" was used. This meant that the balls could easily be plotted before and after a movement in order to see that the resultant change in position is the expected one. The decision to use polar coordinates for storage was used as it makes it more intuitive to imagine the change in the rover's axis as it is simpler to imagine rotations. It is also closer to the manner in which the balls are perceived by the rover.

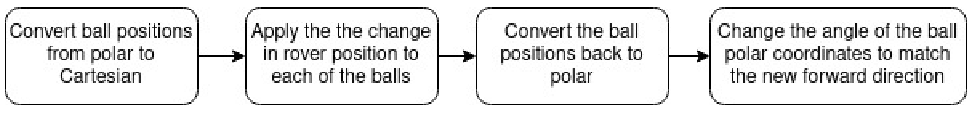

Convert ball positions from polar to Cartesian
Apply the the change in rover position to each of the balls
Convert the ball positions back to polar
Change the angle of the ball polar coordinates to match the new forward direction

Figure 32: Overview of the ball position updating

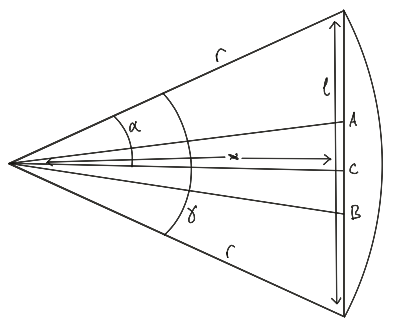
Once all the positions of known balls are updated, the position of any new balls has to be added. The information the rover receives about new balls is in the form of two values from the vision system. These two values represent the horizontal camera position of the two sides of the balls in the camera view. From these two values the polar angle of the centre of the ball horizontally can be calculated. This requires imagining how the flat view of the camera is projected onto a curved view and can be done using the equation below. The angle α is the desired angle. The diagram is shown from which these equations were created, the point C represents the midpoint between the two given points A and B, the value l can be treated as one as it is cancelled when evaluating the final equation.

$$\frac{l}{2r} = sin(\gamma/2) \tag{17}$$

$$r = \frac{l}{2sin(\gamma/2} \tag{18}$$

$$x = \sqrt{r^2 + c^2 - 2rccos(90 - \gamma/2)} \tag{19}$$

$$\alpha = sin^{-1}(\frac{c * sin(90 - \gamma/2}{x}) \tag{20}$$



Once this angle is calculated the distance to the ball can be calculated using the distance between the two values as the diameter of the ball is known. This is achieved using the equation below. Once the coordinates of any new balls are found they are added to the map and replace their old position if they had been previously detected. The angle β used is the angle between the two detected edges calculated using the value α from above. The value d is the diameter of one of the balls and D is the distance to the ball.

$$D = \frac{d}{2tan(\beta/2} \tag{21}$$

3.4.5 Updating Depth Information

Once the position of the balls had been calculated an estimation of the depth to any object, not just including balls, was calculated. An algorithm was developed for this separately as a proof of concept before beginning work on the rover implementation. The basic idea is that it should be possible to detect the distance towards objects using two different images taken at different positions. The first step to be able to do this was to apply an edge detection algorithm to the image. This was done in testing using the python "Open CV" library. The algorithm we then used to detect depth on this image was very simple. The principal is that horizontal edges that are closer to the rover will move more between frames than edges that are further from the user. Therefore, the average position of all the white pixels in every vertical row of the image is calculated to measure this change. Another image is then needed, taken closer to the object, this was done in testing using a video from a mobile phone using every $20th$ frame. The percentage change of the average position of any white pixels within a vertical row is then taken and this is used as the distance estimator. This method is inspired from an extremely simplified version of optical flow, which is used to detect movement in images. The results from testing can be shown plotted with the image in figure 33. The edge detection version of the photo is shown in figure 34, the photo is of two bottles placed at different distances.

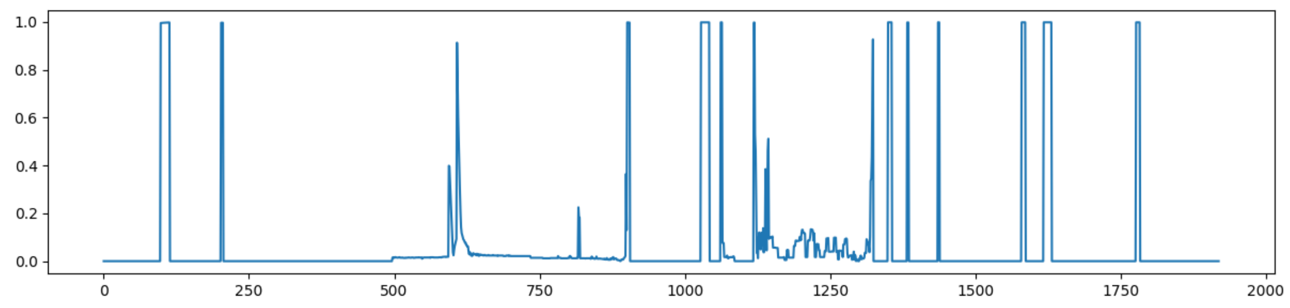


Figure 33: Unfiltered algorithm output

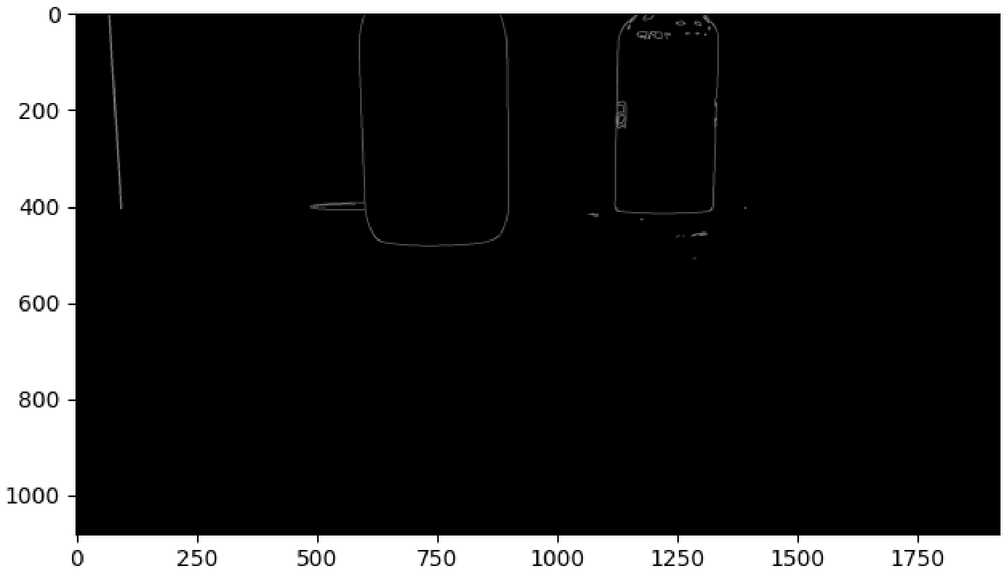


Figure 34: Edge detection on image of two bottles

Since this data is very noisy a moving average filter was taken over the data. This still resulted in several small spikes, and so any spikes that were below a certain width were then removed from the data. This was done by checking each window of the data and seeing the maximum change and maximum fall. If the fall and gain were both over a threshold all the data in the window was set to the minimum of the window. The results from testing using this methodology can be seen in figure 35. As can be seen this does give a good estimator of depth and can certainly be used to notify a user if an object is within the path of the rover. The flow chart for this diagram is shown in figure 36. This was decided to be a good enough proof of concept and so the algorithm was incorporated into the server code with data coming from the rover itself. The server uses data sent after every movement and only uses the new data if it has not changed orientation.

Initially when depth data had been calculated we wanted to have it be remembered if the rover was to turn around, giving a full 360 estimator of depth at all times. In order to accomplish this the depth data was stored in an array, with its index representing its angle and its value representing depth. After every movement this was updated similarly to how the ball data was updated. Firstly, every point had its position converted to Cartesian and was translated by the rover movement. Then each point was converted back to polar coordinates. The data however was needed at specific angles corresponding to indexes of the array and so the average depth of the data either side of the desired angle was taken. After this the data was rotated to represent the change in rover orientation. This could then be sent to

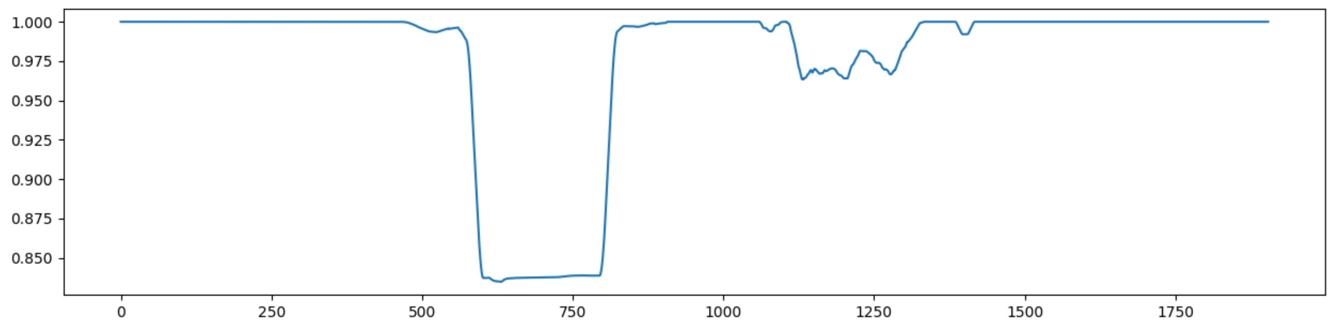


Figure 35: Filtered algorithm output

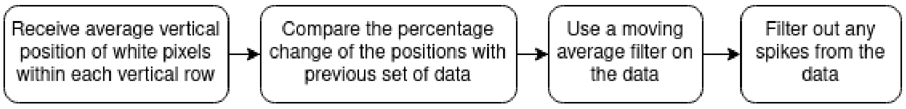

Receive average vertical position of white pixels within each vertical row
Compare the percentage change of the positions with previous set of data
Use a moving average filter on the data
Filter out any spikes from the data

Figure 36: Flow diagram for depth detection algorithm

the user interface and plotted on the map as shown by the circle. It was found, however, that the depth data was too inaccurate to bother with this method and so only the arc representing the rover's current field of view was used so no data had to be translated. As with the balls, ”matplotlib” was used to visualise the translation of the coordinates whilst developing this.

3.4.6 Deciding Next Move

In order to decide upon the next move for the rover the server first checks the current movement mode type and then calls one of three functions in order to create the move. The three movement modes are command, search and map.

3.4.7 Command Move

Creating a command move is the simplest mode. The three types of command the rover can receive are ”forward(x)” where x is the distance in mm, ”turn(y)” where y is the angle in degrees to turn and finally ”stop”, which clears all saved and in-progress commands. Commands are sent by the user through the user interface to the server and are kept in a stack object, being cleared as they are executed. Since the rover only computes small movements before checking in with the Vision section, any command that requires a movement or turn greater than the maximum movement or turn is broken up into sub-commands that add up to the movement. This means that when the rover requests a movement the server first checks if a sub-command exists: if it does it takes it from the sub-command stack and sends it to the rover; if it doesn't it takes the next main command and splits it into sub-commands which are added to the sub-command stack. The first sub-command is then sent. If the server receives a stop command from the user all sub-commands and main commands are deleted.

3.4.8 Map Move

In map mode the user sends a coordinate to the rover that it should complete. A movement command therefore needs to take the user closer to the required target. Firstly, the rover checks if it is low on charge relative to the distance to the charging station, if it is it uses that as the target otherwise it uses the user coordinate as the target. In initial testing the server would just send the target straight to the drive system and the drive would only complete the maximum allowed distance. An obstacle avoidance system was then created to find the optimum path to the target avoiding any balls on the way.

3.4.9 Obstacle Avoidance

In order to make the rover fully autonomous a general obstacle avoidance algorithm was developed. This allows the rover to reach a given destination whilst making intelligent decisions on how to avoid obstacles on the way. The algorithm works in the following way:

- Receive information of the obstacles nearby from the sensors
- Construct a graph with the sensor's information. The graph represents a "map" of the surroundings. Every node on the graph represents a location where the rover is allowed to move. Nodes are removed from the graph where an obstacle is present. A fixed weight of one is assigned to the edge connecting a node to any of its neighbours.
- An algorithm to find the shortest weighted path between the location of the rover and the destination is applied.
- As the rover moves towards the destination, the "map" is updated with new obstacle information. The shortest weighted path to the destination is recomputed and the rover changes direction accordingly

Best path algorithm Dijkstra's algorithm was used to find the shortest path between two points in the "map". This was later improved upon and transformed into an A* algorithm by adding a heuristic function to give priority to nodes closer to the destination. The code for these implementations was adapted from [7],[8] and [9].

3.4.10 Search Move

In search mode the rover is given a specific ball colour to go towards. If the ball's position is unknown the rover must search the surroundings for it. Designing the route the rover should take to search the map was the core part of this mode. It was decided that no particular direction should be prioritised over any other and for this reason a spiral route was chosen, with the center being the charging station. This was chosen as it allows the rover to view the map from many angles and always covers ground closer to the rover before ground further away. Since the distance between spiral turns should remain constant an Archimedes's spiral was used as it has this property. The equation is shown below,

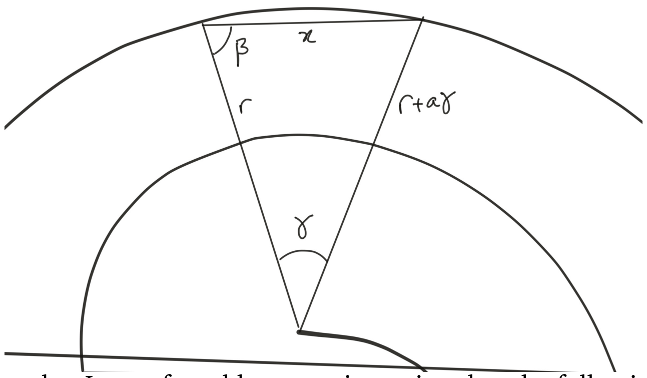
$$r = a\theta$$

and simply makes the radius of rotation in direct proportion to the total angle change. Using this equation is necessary to calculate the movement the rover should take given that it is a given distance from the charging station and is facing 90 degrees to it. The diagram and equations show how to calculate this polar coordinate relative to the rover, the distance x here is the desired depth the rover should move forward and r is the distance to the charging station.

$$\frac{l}{2r} = sin(\gamma/2) \tag{22}$$

$$\frac{x}{sin(\gamma} = \frac{r + a\gamma}{sin(\beta} \tag{23}$$

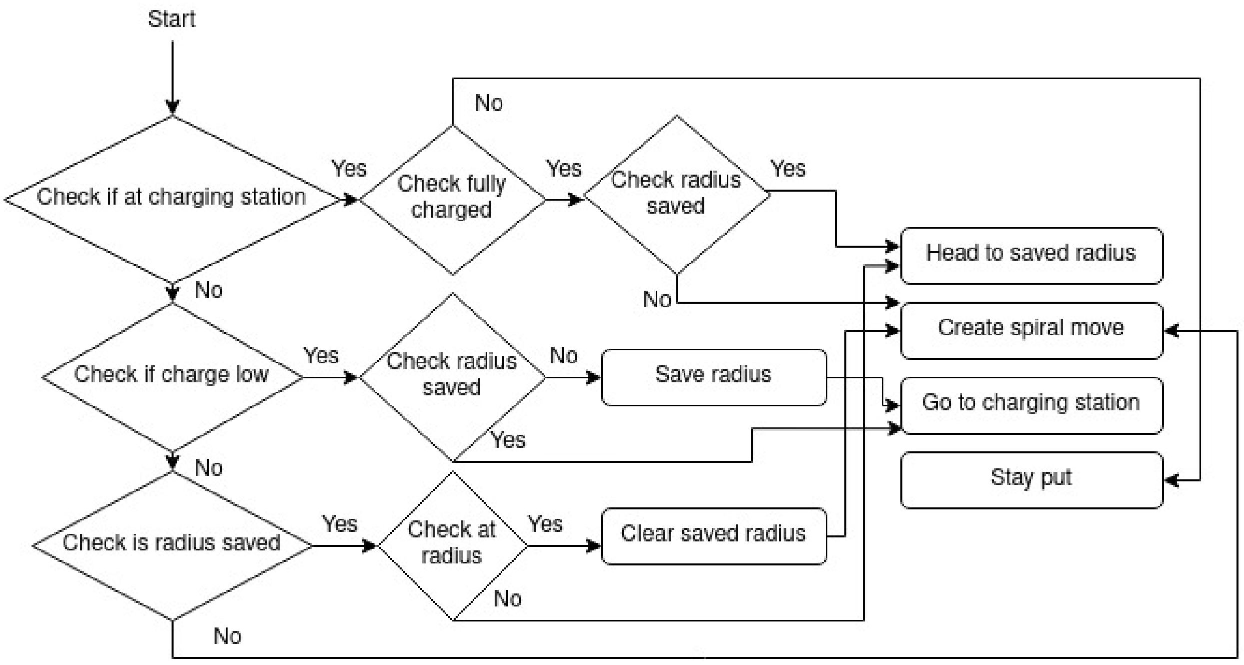
$$\beta = sin^{-1}((r + a\gamma) * sin()) \tag{24}$$



This equation was tested using "matplotlib" and gave good results. It was found however in testing that the following code gave very similar results and is much less computationally expensive and so was used instead, giving the results shown in figure 37.

```
angle = angle_to_charge + 90 //angle that the rover should turn should be 90 degrees to the charging station
angle %= 360
theta = 0.7 - pow(math.e, -1 * distance / 100) * 0.7 //this value is used to keep the distance between spiral turns constant
angle += theta
angle = angle % 360
```

Once this coordinate is given, it is then treated as a target for the map mode command creation which employs the obstacle avoidance, meaning the rover will avoid any other balls it meets on the way. If the rover finds it is low on charge relative to the distance from the charging station, it will head back to the charging station. The rover will then head back to the radius it was at previously once fully charged. The flow diagram for the search mode algorithm is shown in figure 3.4.10.



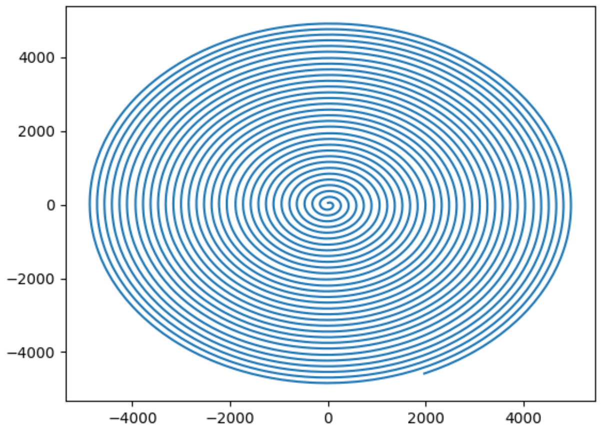


Figure 37: A simulated plot of the spiral movements of the rover

3.4.11 Design of The User Interface

The user interface was built using the flutter toolkit from Google. This meant that a cross platform app could be easily built that could run as a desktop app or in a browser providing more options to the users. The app was designed to present all the information to the user at once, avoiding pagination so that commands can be made as quick as possible. The photos in the appendix show the design of the interface. A panel on the right changes depending on the command mode the user selects. It either shows coordinates of targets and balls for the map mode, a command input and list of commands for command mode or the option to select which ball to search for in search mode. On the left the interface shows a map that indicates the position of any balls relative to the rover and can also display the depth information. The user can also click on this map to select the target coordinate for the rover. Finally along the bottom information such as battery charge and health is displayed along with a speed control option for the rover.

3.5 Control

3.5.1 Message Protocol

In order to communicate with the Drive, Vision and Energy subsystems a general message protocol has been designed to ensure that only valid data is received. The message is composed by:

- Start of Message (SOM) character, used to identify the beginning of the message.
- The payload of the message. The size and structure of the payload is fixed and depends on the specific subsystem considered.
- an End of Message (EOM) character, used to identify the end of the message.

Messages are constructed and sent byte by byte according to the protocol specified above. The transmitter sends a message in the following way:

- Send SOM.
- Send the payload. Each subsystem expects data (characters) to be in a specific order. It is important that this predefined order is respected to communicate successfully.
- Send EOM.

This ensures that the receiver can safely receive a message by:

- Waiting until Start of Message character is received.
- Receiving the payload of the message.
- Stop receiving when the End of Message character is encountered.

The diagram in figure 38 shows how data is sent and received.

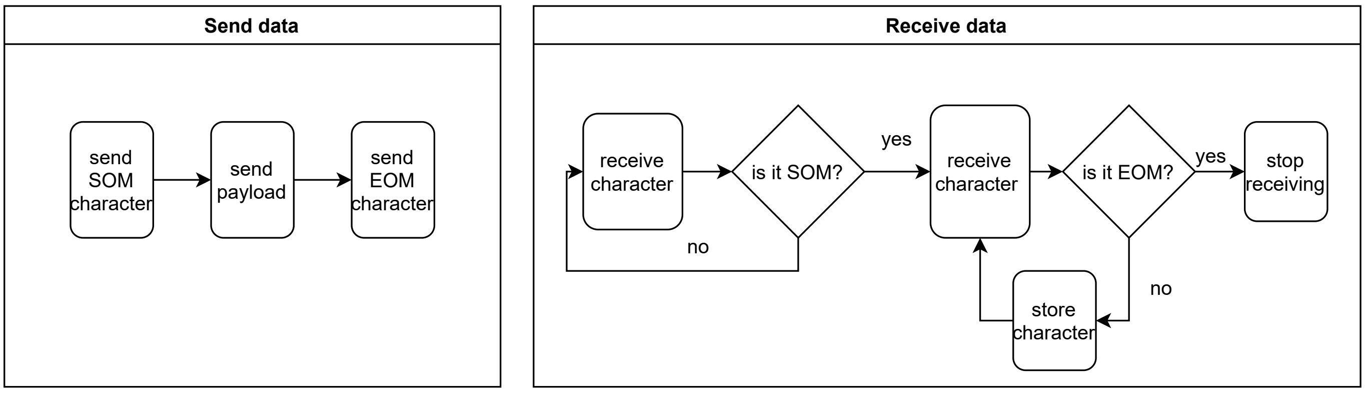


Figure 38: Sending and Receiving data

3.5.2 Data Timing

The ESP32 is what controls and manages the flow of information. It collects data from every subsystem, sends it to the server for processing and finally receives back data to reach the next destination.

Independent communication with each subsystem: Priority was given to implementing communication between Control and each subsystem. A suitable message protocol was designed according to the data exchanged with each subsystem. Tests were carried out making sure that the data sent from one subsystem matched with the data received by the other subsystem.

General communication between subsystems | Version 1 The next step was designing the general data flow between every subsystem. JSON documents were chosen as our file format for data interchange because:

- JSON documents provide an easy way of merging the data collected from each subsystem in a single file
- There are well documented libraries for ESP32 that make the serialization/de-serialization process easy for the Control subsystem and the Command subsystem

The data flow is **sequential** and data is received in order from Drive, Energy and Vision. A JSON document is constructed with the collected data and sent to Command. Finally the response form Command is sent to Drive.

General communication between subsystems | Version 2 Our design specification required fast bidirectional communication between Energy and Control. This has been the biggest challenge in the implementation of the Control subsystem. The Energy arduino code would not function correctly if delays due to receiving data are introduced. The source of this delay are the computations performed on the server and the inherent single delays due to the sequential communication with every other subsystem. The solution that we adopted for this problem was **parallel** communication, which was achieved by exploiting the dual-core nature of the ESP32. In particular the communication flow was split between the two cores in the following way:

- **Core1** manages receiving data from Drive, Energy and Vision. It also manages communication with the server and forwards data to the Drive subsystem to compute the next move
- **Core2** continuously sends data to Energy. This ensures that new data is always ready to be received and hence there are no delays from the point of view of the energy subsystem due to to communication

The flow of information is described in Figure 39.

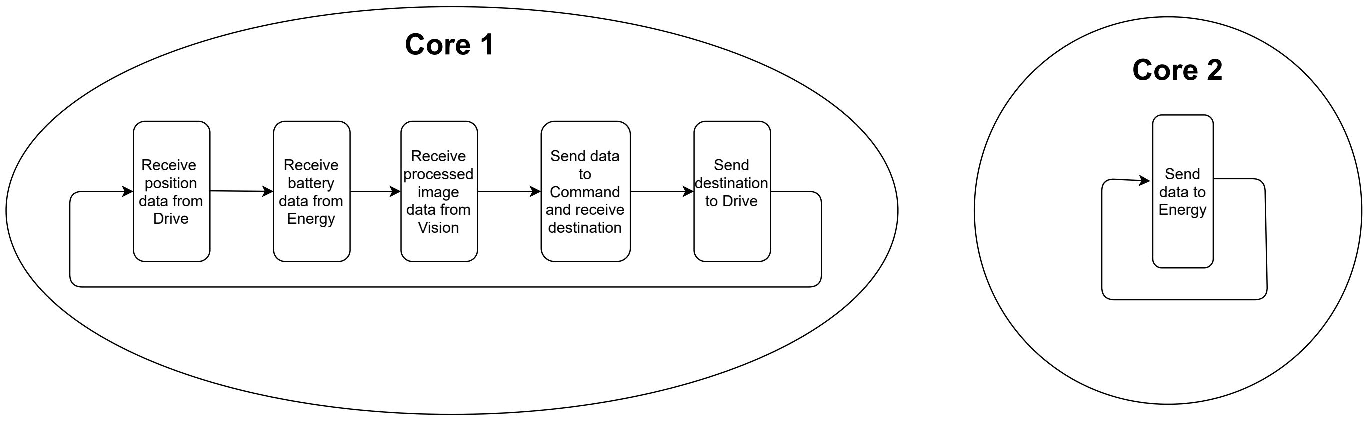

Core 1
Receive position data from Drive
Receive battery data from Energy
Receive processed image data from Vision
Send data to Command and receive destination
Send destination to Drive
Core 2
Send data to Energy

Figure 39: High Level data flow

3.5.3 Communication Protocols

Suitable communication protocols have been chosen in order to meet the design criteria. For each communication protocol speed of communication, reliability, direction of communication and hardware availability were analyzed for a given subsystem. Whenever multiple communication protocols satisfied the design criteria, the simplest option to implement was selected.

UART The UART communication protocol has been chosen for data transmission between Drive and Control and between Energy and Control. In particular UART was chosen because:

- It is fast enough for the application: the Drive and Energy subsystem only need to exchange small amounts of data (small payload size)

- It has in-built error detection and correction
- It allows full-duplex communication. Full-duplex is essential for Control-Energy communication as it allows receiving and sending data at the same time
- It is easier to implement than SPI

SPI The SPI communication protocol has been chosen for data transmission between Vision and Control. In particular SPI was chosen because:

- It allows for high speed data transmission and is faster than UART or I2C. This makes it suitable for transmitting large amount of data such as image data received from Vision (large payload size)
- It provides robust error detection features

WiFi and HTTP The Rover communicates with the remote server using WiFi. WiFi was chosen over Bluetooth because:

- WiFi range is much bigger than Bluetooth range. This means that the rover can explore a wider area of the Mars surface whilst still communicating with the Server
- WiFi provides better wireless security than Bluetooth making unauthorized accesses to the server harder
- WiFi connection is faster than Bluetooth. This allows efficient data exchange between the Rover and the Server

Data is exchanged between Control and Command using HTTPS POST requests. HTTPS was chosen over other standards because:

- It is fast enough for the application: it satisfies the speed design criteria standards
- It provides enhanced security by encrypting the data exchanged
- It is compatible with the JSON documents library used on the ESP32.

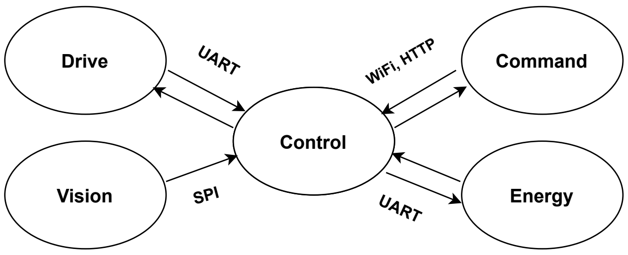


Figure 40: Communication Protocols

4 Testing and Evaluation

4.1 Drive

The tests of the drive subsystem conducted throughout the project were of the following type:

Functional testing

1. **Testing of the single parts of the subsystem:**
    (a) Motor functionality: Turning, moving forward and backwards were tested by hard-coding the distance and angle targets. The motors were found to be accurate in the movements even at maximum speed (for error calculation see the testing of performance), as long as cables were not obstructing the movements and connections were not lose.
    (b) SMPS functionality: A delay was perceived when turning on the SMPS, which was fixed by setting the Buck to closed-loop mode. Since a delay of 5seconds was still perceived because of the large capacitors used, this was later fixed by making the logic of the state machine idle for the first 5sec the SMPS is on (only the first time of the loop). Additionally, even though the SMPS was thought to receive 6.4V at its input from the energy subsystem, this could not be tested (battery defect), so 5V were used at the input (standard power socket), resulting in an output voltage of 5V or less.
    (c) Optical sensor functionality: the sensor was tested on different surfaces and its data checked via the Serial Monitor of the Arduino. The quality of the images (expressed in terms of stars) was so compromised on certain surfaces that it would ruin the whole functionality of the rover. This was solved by creating the alternative, time-based model. However, in the right environment, the optical sensor was observed to be very precise and was the best available way of tracking the movements.
2. **Testing of the models:**
    The code for each component of the drive subsystem was merged creating an auxiliary functions file in the same Arduino sketch to ease the reading of the main program; Data from the optical sensor was used in the feedback loop that would calculate the remaining path to be computed. Then, the functionality of each section was re-checked through the Serial Monitor and, by inspection, confirming that the sensor and motors were working correctly and that the state machine of the moving algorithm would not lead to undefined states. For each further feature added to the main code, a series of functional testing were conducted to check both the original functionality and the expected behaviour from the addition. Tests were made to check that the PID controller and the variable speed of the wheels were implemented correctly, before adding them to the code. It was measured that the minimum speed at which the rover would start moving is above 5 rpm, with a maximum of 60rpm for the highest duty cycle (Vref of approximately 5V). Having found some bugs in the built-in PID controller provided in the relevant Arduino library, we decided to create our own controller, which was tested with various inputs and then integrated to adjust the current speed (when moving forward) and speed difference of the wheels (when turning whilst moving).

Testing on the performance

(These tests were done on the fully integrated rover since its weight and specific characteristics could influence the actual speed and precision of movements)

1. **Arc trajectory model:** Following some research on the optimal path to be computed by the rover for a given location, an alternative method for the movement was tested; using an arc trajectory. However, some successful tests highlighted how this method did not enhance the performance of the rover (neither in terms of speed, nor precision). The method involved some extremely complicated calculations for the movement tracking. Therefore this model was discarded and it was decided that the optimal path was to be computed on the server, which would send small, constantly updated instructions.
2. **Optical sensor-based model:** This version was extensively verified to make sure both turning and moving forward were within the error stated in the requirements (10%). Various paths and different sequences of commands were tested, and the results were the following: for flat non-homogeneous and non-reflective surfaces (e.g. wood) the precision of the movement was measured to be above 90%. This was an extremely good result

given the limitations of the sensor and the imperfections of the motors. The maximum errors measured for the distance and the angle were approximately 5mm and 5°. However, for other surfaces (e.g. white table or carpet) the optical sensor did not provide reliable enough data to be used in the feedback system.

3. **Time-based model:** This back-up model was tested especially on the surfaces for which the optical sensor could not be used. The general tests on the precision were overall satisfactory, with an error within 13% (as expected lower than the optical sensor model because of the lack of a feedback system).

4. **Optical sensor-based model with PI controller:** This model achieved roughly the same level of precision in the movement as the other without a PI controller (within 10% error). However, the overall time spent before reaching the target was greater. As the controller would decrease the speed when approaching the target, instead of using a constant maximum speed. This would have been an important feature if high speed was used; however, an analysis of the performance showed how, in our case, the benefit of a speed control system did not increase the overall quality of the movements. This is because the other model would already result in well-controlled movements and did not produce an overshoot, given that the distances and speed are always contained. Therefore, the model without PI controller was chosen for the final version.

Testing with the control and command unit

For each model, the communication with the control unit was tested making sure the data sent and received was correct and that the functionality of the rover was not compromised. The delay measured between the end of the current instructions and the beginning of the next execution was measured to be below 100ms, meaning that the communication is fast enough to not cause a major delay between movements, allowing the instruction to be short but frequently updated. Various tests were conducted to check that the current location computed from the optical sensor coordinates was correctly updated on the monitor of the website and that the next instruction was calculated without errors.

4.2 Energy

4.2.1 Cell Testing

Testing Energy involved making a variety of programs which confirmed the performance of functions and helped in the further development of others. Figure 41 was used to test the CCCV charging mode. It clearly worked well with the CV mode keeping the cell voltage at 3.6V and the charging ceasing when the current hit 50mA. This was then tested using the solar panels. The effect was less pronounced as the max lamp current is 70mA and therefore the 50mA limit was decreased to 20mA (Figure 46). However, it was good to know that the voltage PID worked well in combination with the MPPT function (Note the fluctuating current). The SOC estimation was tested by having a discharge program which changed the current at 50% as seen in figure 42. The SOC was a linear line that was steeper when the discharge current was 250mA. The linearity was expected due to the constant current output and the change to a shallower gradient during the -125mA confirmed that the function was working as expected. The 2.6% overshoot seems acceptable given that this was a first cycle with no SOH calibration. This also justifies the use of a 3% tolerance with the balancing function.

As stated during the Design Implementation, the initial value of the SOC is found using a lookup table of a pre-made SOC-OCV graph shown in figure 43 (There is a direct relationship between open-circuit voltage and state of charge). Linear interpolation within the Arduino was then used to estimate the value of the starting SOC. Note however the uniform nature of the curve and the hysteresis caused by charging/discharging. This has created a significant amount of error due to an average having to be used. In addition, the quantisation error of the Arduino is ($^4/_{2^{10}} = 3.9mV$), in some cases, larger than the difference in voltage between two 5% intervals. Consequently the decision to zero the Coulomb count at every available moment was made. The Re-Calibration of the SOH is an integral part of the system in order to keep an accurate value of the SOC. This was tested by going through three charge-discharge cycles and allowing the Arduino to calibrate the SOH between each one. Figure 44 shows how well this worked with the error at each end decreasing to 0.03% during charging and 0.3% for the discharge. When testing the balancing, a simple discharge of two cells was implemented (Figure 45) with the gap between the SOCs correctly converging to 3%. Having tested how the SOH re-calibration positively effects the coulomb count, it would make sense to decrease the balancing tolerance to a lower value like 0.3% as the number of complete charge/discharge cycles increases.

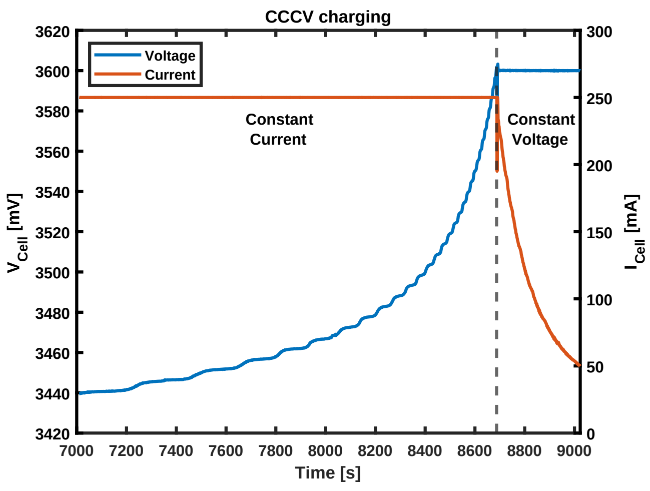


Figure 41: Constant Current - Constant Voltage Charge

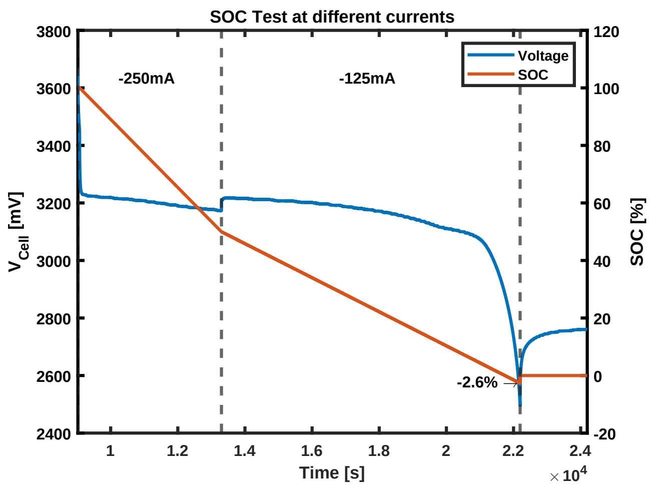


Figure 42: SOC discharge at different currents

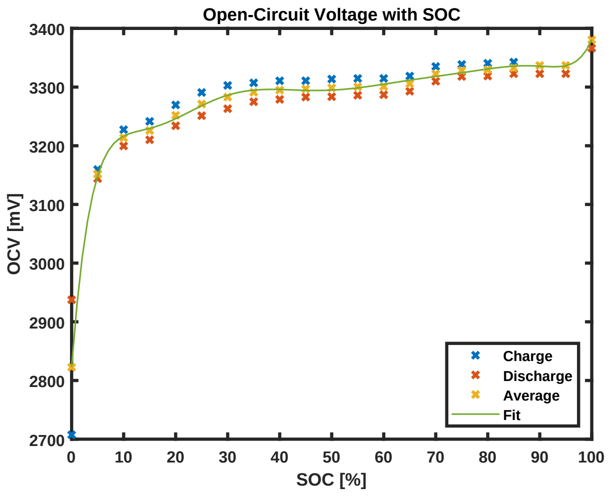


Figure 43: Relationship between SOC and OVC

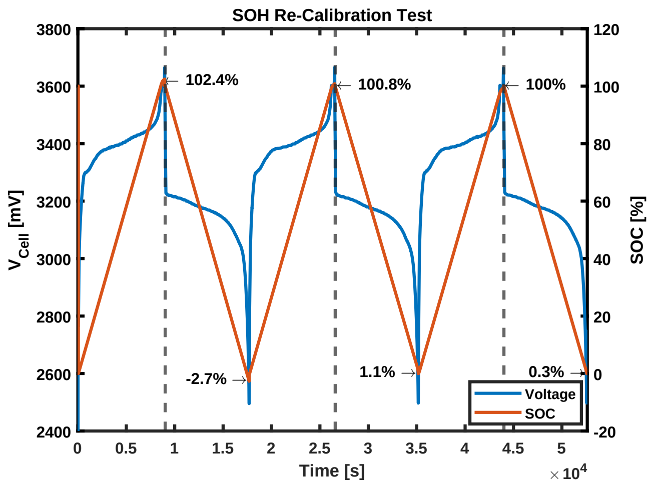


Figure 44: Testing SOH Re-calibration

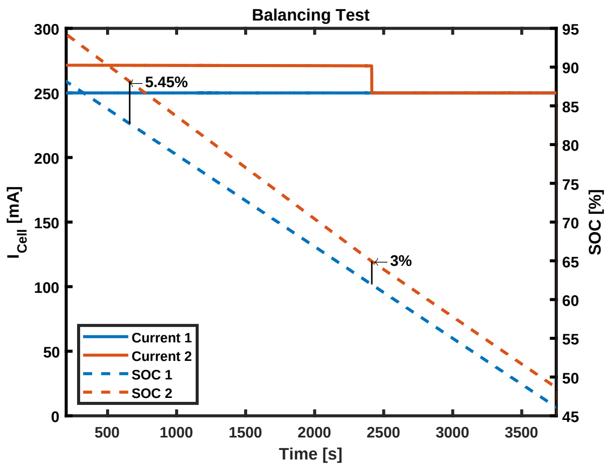


Figure 45: Relationship between SOC and OVC

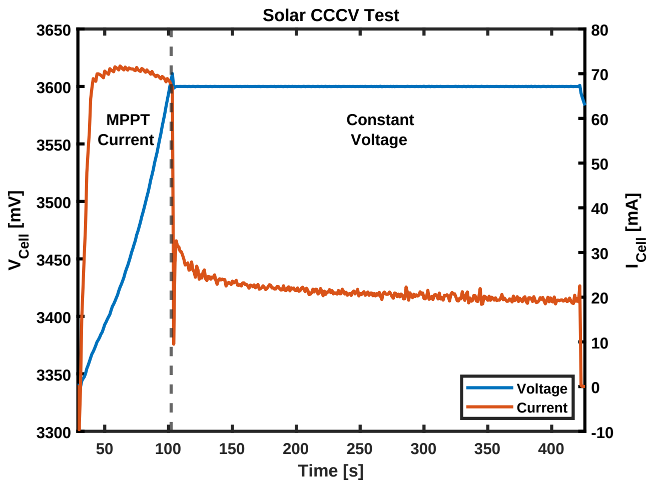


Figure 46: Testing CCCV charging with PV

4.2.2 PV Testing

A single PV panel was first characterised to give a baseline for the MPPT program. In line with the theory, a single panel had a flat current-voltage curve until a breakdown voltage was reached (Figure 47). This breakdown voltage decreased as the PV cell was under the lamp for longer. Clearly, this is because temperature had increased, resulting in a lower peak for the power curve. The MPPT was then tested with a 2x2 panel arrangement in direct sunlight (Figure 48). In this case, the MPPT was looking at the voltages and currents at the output of the buck SMPS because the power losses of the SMPS is likely to effect the shape of the power curve. Left-side tracking worked well with bigger intervals when the system was far away from the peak and smaller ones taken near it (Note the big clump of data points) and resulted in lower fluctuations at the peak. Note that the right-sided tacking made it to the peak but took longer to do so which seems like an extreme case where the gradient turns positive on the right-hand side - it is unlikely that it would ever get this bad but caution is advised. This may be an consequence of the $\sim 8V$ limitation of Port A on the SMPS causing shoot through between the PV Panel and the load.

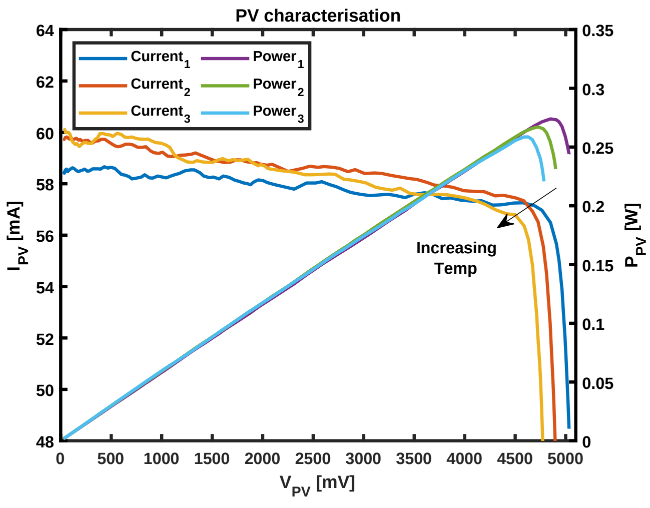


Figure 47: PV Current Characterisation

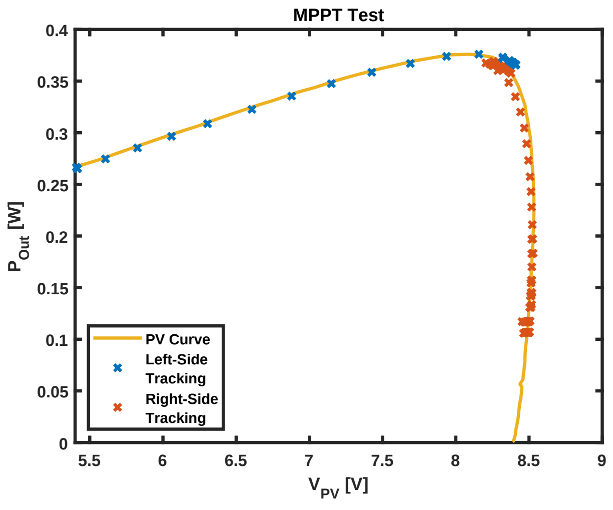


Figure 48: 2x2 PV MPPT Test under normal sunlight

4.3 Vision

The subsystem was tested according to the design criteria laid out in the problem definition and design criteria section. Each criteria was assessed and evaluated to determine whether it has been met.

Ball Detection Testing:

To collect this data, a black background has been used for the coloured balls and a white background for the black ball. Each output stream was examined and a camera gain of 0 was found to be the preferable test setting.

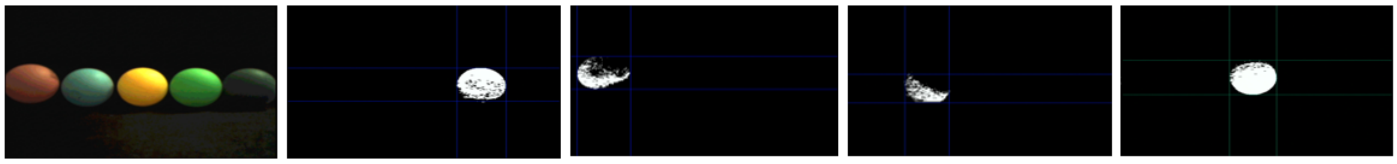


Figure 49: Ball detection : Video Input , Green Detect , Pink Detect , Blue Detect , Orange Detect

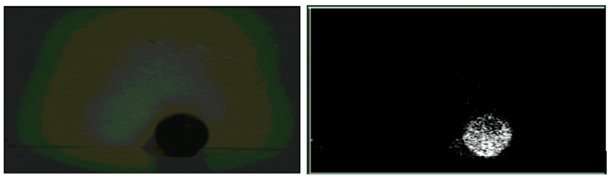


Figure 50: Black ball detection

Criteria One and Criteria Two were shown to be operational for all balls in stated conditions. The highlighting was unique to each ball however it was susceptible to differing brightness conditions which meant the gain of the camera had to be adjusted to a low value. Further to this, without a solid background the possibility for incorrect highlighting of colour or unexpected brightness in the background was greater so often a solid background was required for testing. *Criteria Three* was shown to work in the filter implementation which removed non constant white horizontals; it was found that the image sometimes showed reduced noise occurring. This means that the size of the noise filtering buffer should be increased to further increase the intensity of noise reduction. However, this worked satisfactorily for non-reflective backgrounds. *Criteria Four* was met in the ball detection figure above, where ymin ymax and xmin xmax matchd our ball highlighted pixels. Further testing of the distance calculated was done in Command testing.

Edge Detection Testing:

The aim of this section was to detect obstructions from unknown visual data. This section was tested by observing a range of objects in front of the rover and moving them backwards and forwards in front of the camera, in both the sideways and upright camera mode.

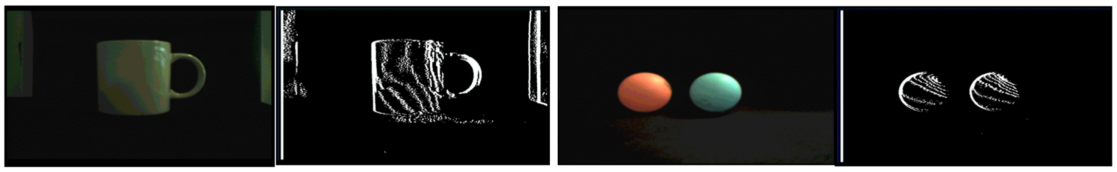


Figure 51: Vertical Camera position analysis

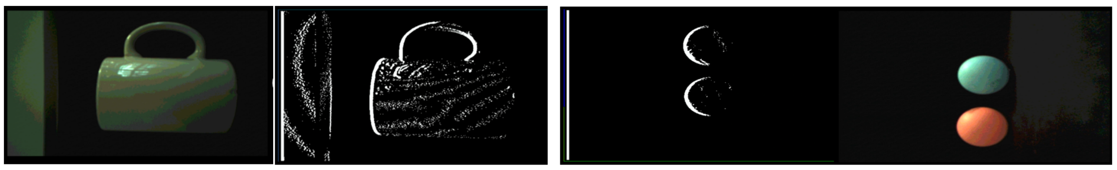


Figure 52: Horizontal oriented camera analysis

Criteria One: depending on the orientation of the camera either vertical or horizontal edges are highlighted. The horizontally oriented camera analysis figure reveals the horizontal edges and was the orientation used in our final design. *Criteria Two* was successfully accomplished through the sectioning and calculation of the average white pixel position. This was further developed in the Command algorithms to find the depth.

Communication Testing:

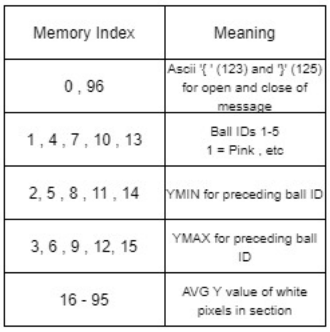
This section's aim was to relay information on detected objects to Control subsystem for further processing. To test its function we printed the received byte values on the serial monitor of the Arduino ESP32 it was connected to. We measured their accuracy and consistency by comparing the expected value to the received value across multiple messages.

Criteria One was proved to be met by the inclusion of our array of registers: `reg [7:0]mem[96:0];`, storing all data from analysis of the current frame. *Criteria Two:* the format can be seen in both the vision and control implementations, but a summary is included in figure 53, alongside an annotated serial monitor output of the Arduino. This shows how this criteria is satisfactorily met. *Criteria Three:* figure 54 reveals our data is received in the intended format with values associated to what we expect. This consistently occurred across messages during testing. Further to this to ensure a robust design we implemented safeguards on the command subsystem which sanity check data before using it. Thus we can safely say the relayed data is accurate.

4.4 Command

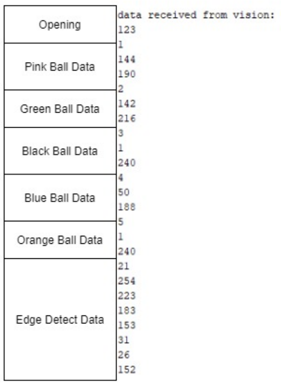
4.4.1 Testing Communication

The communication through the API was tested using the ”postman” app for HTTP requests. This application makes it easy to send POST and GET requests. The results of sending different fake rover movements and the resulting move commands from the server could easily be tested to ensure that the data was as expected. This also made it easier to work on the command system independently from the control system. Each of the three command modes was tested this way to ensure they gave the expected behaviour. The data sent to the interface was also checked for validity in different situations. The actual movement testing is discussed in the drive section.



Memory Index	Meaning
0 , 96	Ascii '{ ' (123) and '}' (125) for open and close of message
1 , 4 , 7 , 10 , 13	Ball IDs 1-5 1 = Pink , etc
2, 5 , 8 , 11 , 14	YMIN for preceding ball ID
3, 6 , 9 , 12, 15	YMAX for preceding ball ID
16 - 95	AVG Y value of white pixels in section

Figure 53: Memory map for communication to Control subsystem



```
data received from vision:
123
1
144
190
2
142
216
3
1
240
4
50
188
5
1
240
21
254
223
183
153
31
26
152
```

Figure 54: Annotated Serial output of Arduino

4.4.2 Testing Ball Positions

As previously discussed, the ”matplotlib” python library was used for testing the map to ensure that the data looked roughly as expected. However, it was also necessary to check the data numerically. After several movements and rotations were sent to the command system, the position of the balls on the server was checked to be numerically correct. The balls were also checked to be tracked as expected in all of the four quadrants of the map, to ensure that the conversion between Cartesian and Polar coordinates was correct. When first testing, considerable errors were found when the ball was near the edge of the screen or too close to the rover. The error was caused by the ball being only half on the screen and so the ball data was decided to be checked against certain conditions to ensure both edges were far enough within the field of view before using the data. We then tested the accuracy of the distance measurements. To test this, the balls were placed on marked positions in front of a solid background and the depth calculation result was read from the interface. We found that when the ball was closer than 12cm to the camera, the ball went over the edges of the camera view resulting in a distorted distance calculation. Due to this, the balls that were calculated to be within this margin are rejected by the server. It was also found that balls less well defined by the hue space had greater errors. This matched the results in vision testing, where we can see the highlighting for blue and pink is less effective. The results of this testing can be seen in table 4.

Ball Colour	Actual Distance (cm)			Calculated Distance (cm)			Error
Blue	15	30	50	19	36	60	21%
Green	15	30	50	16	32	56	9%
Black	15	30	50	14	33	59	11%
Orange	15	30	50	17	35	55	12%
Pink	15	30	50	20	39	62	27%

Table 4: Average Error of Ball Position at Different Distances

4.4.3 Testing Depth Detection

In testing the depth detection algorithm, we were unable to get a smooth estimator of depth in many directions. We were able to detect if an object was moving directly towards the rover, which is useful in notifying the user of a potential collision. An example of this working can be seen in the video at time 1:18.

4.4.4 Testing Speed of Move Generation

The speed of the server to make a movement decision in each of the three modes (not including the time taken to reach control) was tested to ensure that the delay was acceptable. The results were computed by averaging the running time taken for 10000 move creations with random inputs. The results from the testing are in table 5. As it can be seen, the creation of one movement took a time in the order of 100th of a millisecond, which is a negligible amount of time next to the delay in the transfer of information over the internet

Mode	Command	Map	Search
Time (s)	0.06	A 0.24	0.31

Table 5: Total Time to Generate 10000 Moves

4.4.5 Testing Obstacle Avoidance

The obstacle avoidance system was tested with multiple ball s and routes to ensure that the main functionality was working correctly, which was found to be true. The obstacle avoidance was also tested separately from the actual rover using a function for printing the map and the route adapted from the sources used [8][9]. We then decided to test the speed of the two different algorithms for finding the optimal route, Dijkstra and A-Star. Table 5 shows the results from testing the two modes with varying number of balls. As can be seen, there is no real time difference between the two options within this ball range. As the maximum number of balls we have is 5 either option is clearly fine.

Mode	Number of Balls	Time (ms)
Dijkstra	50	0.018
A-Star	50	0.016
Dijkstra	70	0.043
A-Star	70	0.046
Dijkstra	100	0.062
A-Star	100	0.078
Dijkstra	150	0.080
A-Star	150	0.091

Table 6: Time Taken To Find Optimal Route

4.5 Control

Two different testing strategies have been adopted to ensure proper functioning of the Control subsystem. The full communication flow was tested by giving instructions to the rover using the remote UI and observing behaviour.

Qualitative testing Communication between Control and each subsystem has been tested in the following way:

- The receiver subsystem and the transmitter subsystem were connected by the chosen communication protocol. (In the case of bidirectional communication their roles would be interchanged in order to test the communication in the opposite direction)
- The transmitter would send mock data over the communication protocol
- It was checked that the data received by the receiver matched with the mock data sent by the transmitter. Adjustments were made if this was not the case.

Quantitative testing Each communication protocol (UART, SPI, HTTP) has been tested and evaluated according to two parameters

- Error rate, in order to asses the **reliability** of the communication
- Speed of data transmission, in order to asses whether the communication is **fast** enough

The error rate has been measured in the following way:

- A receiver and a transmitter are connected using UART, SPI and HTTP POST
- The transmitter sends an array of known data
- The receiver collects the data and counts the number of times the wrong number has been received

No errors were detected in any of the tests run, which brought us to the conclusion that the communication protocols were reliable enough for the given application.

The speed of data transmission for UART and SPI can be adjusted to meet the design criteria by picking an appropriate baud-rate and selecting a clock divider for UART and SPI communication respectively. The speed of data transmission for HTTP over WiFi has been measured by sending messages with a fixed payload and measuring the average delay for the full data transmission.

Communication Protocol	Speed of data transmission
UART	115200 bits/s
SPI	2000000 bits/s
HTTP	70 ms latency

Table 7: Speed of Transmission

4.6 Integration

During the development of each subsystem, each time a new implementation was completed the code was sent to Integration and uploaded to the full rover for functional testing.

Once the design of all subsystems was finalised, the following tests were carried out to test the functionalities of the rover:

- In command mode, does the rover successfully complete the given movement commands.
- In map mode,
  - does the rover successfully reach the target position indicated on the map.
  - is the map successfully updated with the detected obstacles.
  - does the rover successfully avoid all obstacles.
- In search mode,
  - does the rover successfully take a spiral path to search for a specific coloured ball.
  - is the map successfully updated with the detected obstacles.
  - does the rover successfully avoid all obstacles.
  - does the rover successfully return to the charging station when battery is low.

Once the rover was able to successfully complete those tests; one final test was carried out to test all the functionalities together:

- Is the rover able to detect all 5 coloured balls and other non-ball objects and add them to the map whilst avoiding all obstacles and returning to the charging station when required.

5 Evaluation and Critical analysis

Drive Final evaluation: The drive unit implementation was satisfactory both in terms of functionality and performance: additional features were successfully implemented such as the option to tune the speed and the alternative back-up algorithm for when the optical sensor is unreliable. However, the decision of sending the location information very frequently caused small delays throughout the movements. This could have been improved by speeding up the communication or continuing with the execution of the instruction while communication with the server, although this would have made the location data less accurate. Another feature that could have improved the quality of the movements would have been to implement a mathematical model to compute the optimal trajectory following an arch, in order to make the movements look smoother (e.g. doing the spiral to search for balls). This have been discarded for the huge complexity that would have involved, given that the precision and speed were our priorities. Overall, the performance was within the 10% error required and in the end the only relevant limitations were caused by the hardware components we could not substitute (e.g. optical sensor).

Energy Final evaluation: Though the criteria for the Energy subsystem is clearly defined, there are multiple ways the specification could have been satisfied with varying degrees of success. A main challenge in getting an accurate SOC and SOH estimation has been dealing with the quantization error of the analogue ports. The moving average filter has worked well in mitigating this but a highly accurate system would require a Kalman filter. This was not used as it is computationally extensive, though it could have been offloaded to Control. Having an estimate of the current on the other side of the SMPS is also not ideal and in the future this should be accounted for. An MPPT algorithm based upon Incremental Conductance worked well in most conditions but, as seen during testing, it struggled if the slope was near vertical. A perturb and observe method may have worked better with this extreme case. Overall, the system met the specification set but a main problem would be the capacity as only 2 cells are in use. This unfortunately is a limitation of the SMPS and a possible solution would be to have the cells in a 2x2 configuration. This would however compromise on safety due to unwanted auto balancing between the cells.

Vision Final evaluation: The design criteria of my subsystem were met by the final implementation. The edge detection process successfully allowed for unidentified (Non ball) objects to be detected and displayed, which was a great success. However, we could aim for further noise elimination or a higher tolerance for edge high-lighting as it was very susceptible to differing brightness and shadowing on flat surfaces this could be done by also applying a 3x3 Gaussian filter rather than a one dimensional. Ball detection was robust for colours such as green ,orange, blue, red (All well-defined hue spaces), but adjusting of the camera gain and environment was required to isolate the black ball. Further to this, if a significant amount of constant colour existed, it could affect the detection. To tackle this, a more advanced elimination of non-ball highlighting could be developed. Overall, the vision system achieved its high-level objectives for obstacle avoidance and target identification.

Control Final evaluation: Although the design criteria for the Control subsystem were met, a few improvements could be made to improve the communication design. The obstacle avoidance algorithm could have been implemented locally on the ESP32. In this way the delay introduced by continuously exchanging data with the Server to receive the next move could have been removed. Furthermore, WebSockets could have been used instead of HTTPS to reduce the latency for wireless communication with the Server. In addition, error checking features could have been added in our own Verilog implementation of the SPI communication.

Command Final evaluation: The command subsystem met the desired requirements in terms of providing all the information necessary to the user and creating commands to send to the rover. Several features were successfully implemented as well as possible with the data available, such as terrain mapping, depth estimation and the ability to search for balls. The obstacle avoidance system, whilst functioning as expected, could have been improved by adding an estimation of terrain suitability to find a better path. It could also have been improved by changing the way the graph is constructed so as to only contain nodes next to obstacles. This would increase speed of computation and find a more optimal route.

Rover Final evaluation: The final version of the rover system achieved all requirements outlined in the project brief - the rover successfully achieved communication with the server, detected the coloured balls and avoided them. On top of the basic criteria, several additional functions and more sophisticated implementations were developed and implemented in the final rover. This includes the use of edge detection in the Vision module to detect non-ball objects; the spiral motion used to search for the balls in the autonomous mode implemented by Command; the bi-directional communication between the Control and Energy modules.

6 Essay 'Intellectual Property'

Intellectual property (or IP) refers to the ownership of a creation by the person who created it. [10] It is an intangible asset that can be bought or sold. Various types of intellectual property rights exist that allow the owner to stop others from using or copying their work. In the development of a new product, it is extremely important to understand the laws on IP in the countries you plan to publish your creation and to consider applying for IP rights as early on as possible. It is also important to carry out research on the intellectual property held by others in the same sector to avoid IP infringement.

The main types of IP rights are copyright, design rights, trademarks and patents. [11] For copyright, unregistered design rights and unregistered trademarks, no application has to be made as these rights apply automatically. However, in order to report infringement for these rights, proof must be provided of the product, work or branding being used or copied. For registered design rights, registered trademarks and patents, applications must be made in order to obtain the rights. In these cases, proof is not required to report infringements. However, registered rights need capital and time – in particular, obtaining patents can cost thousands of pounds and the application can take several years to obtain.

Copyright protects creative expression and in the context of the Mars Rover, would automatically apply to all original software developed for the project. For code that has been used or adapted from others, we would need to reach out to and come to an agreement with the authors of that code before publishing or using commercially. Design rights protect the appearance of the product and for the Mars Rover, it would refer to the specific arrangement of the rover modules. In this case, the design rights would belong to the department who came up with the original design for the rover. Trademarks protect branding, and would be important if we decided to start a business to commercialise the rover. Patents protect inventions and products and in the case of the rover, could be applied to things such as, if the energy module discovered a method that draws power more efficiently from the solar panels. Patents must be applied for before the invention is published and so it is imperative that an IP attorney are considered at an early stage of the invention.

Appendices

A Energy

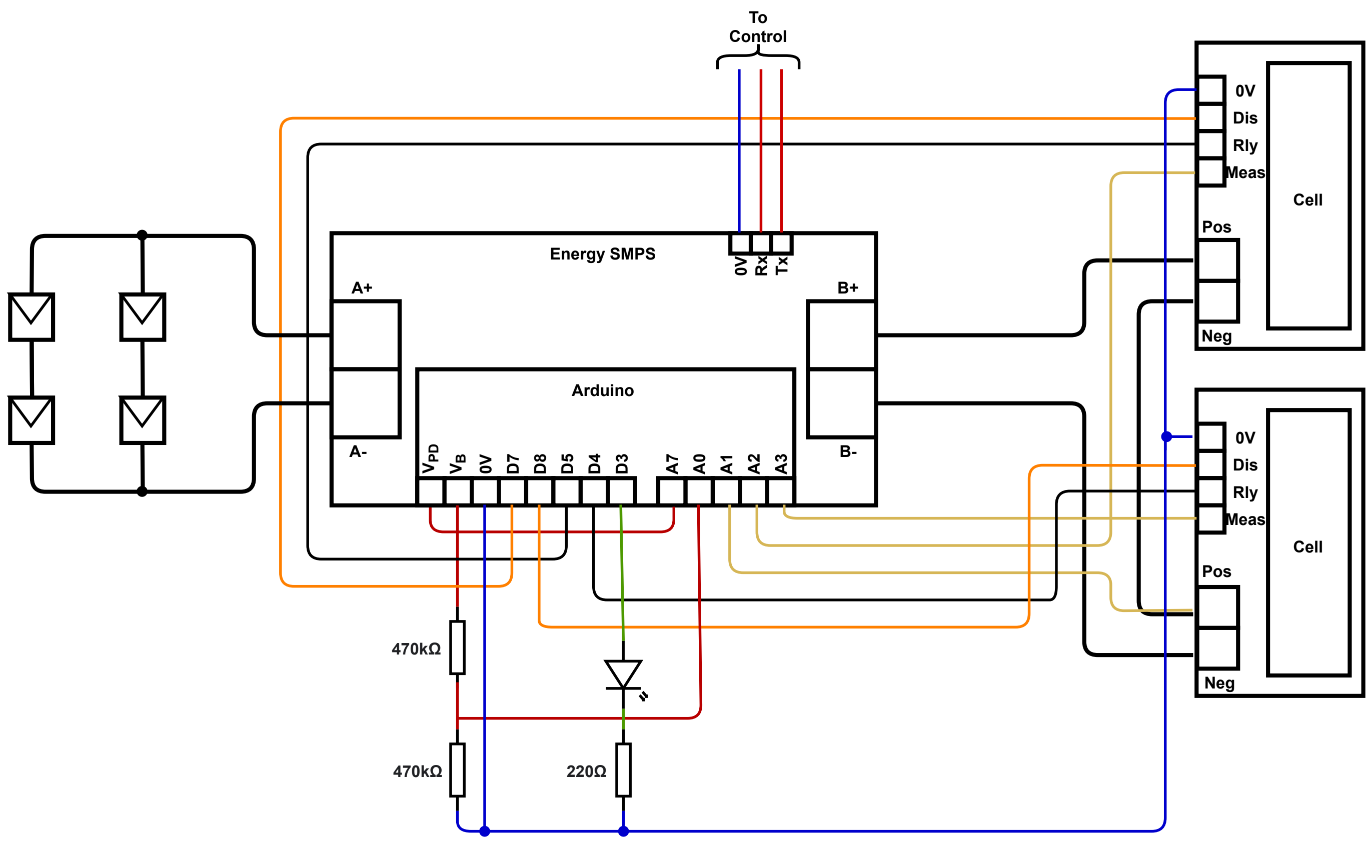


Figure 1: Complete Circuit

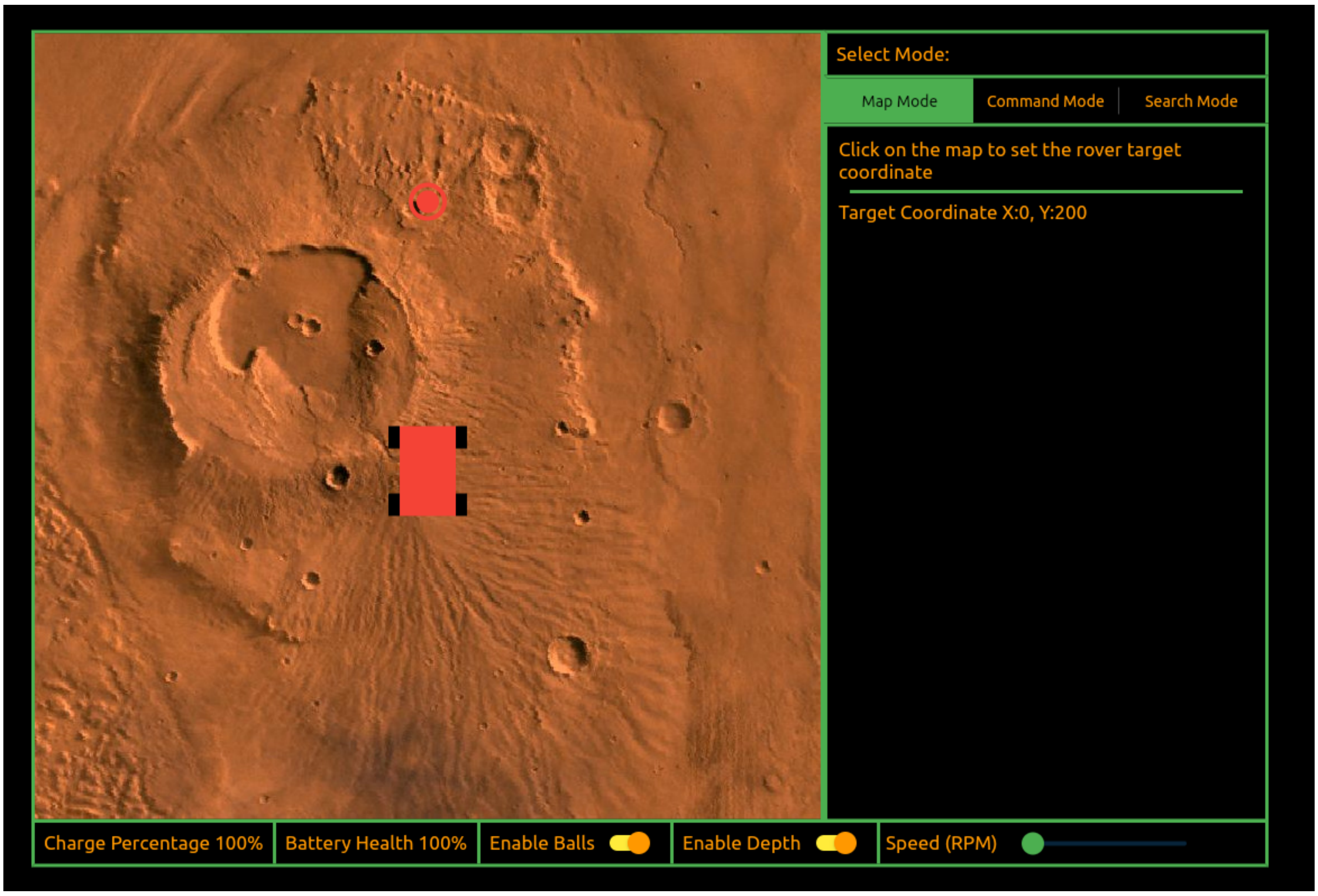
Specification	Value	Unit
Nominal Voltage	3.2	V
Nominal Capacity	500	mAh
Minimum Voltage	2	V
Charging Method	CC/CV 3.6	V
Standard Charging Current CC	250	mA
Continuous Discharge Current	500	mA

Table 1: Cell Specification Source: Adapted from [12]

Specification	Value	Unit
Voltage	5	V
Current	230	mA
Power	1.15	W
Power Tolerance	+/-3	%
Temperature Coefficient	-0.45	$^\circ C$

Table 2: PV Specifiation Source: Adapted from [13]

B Command



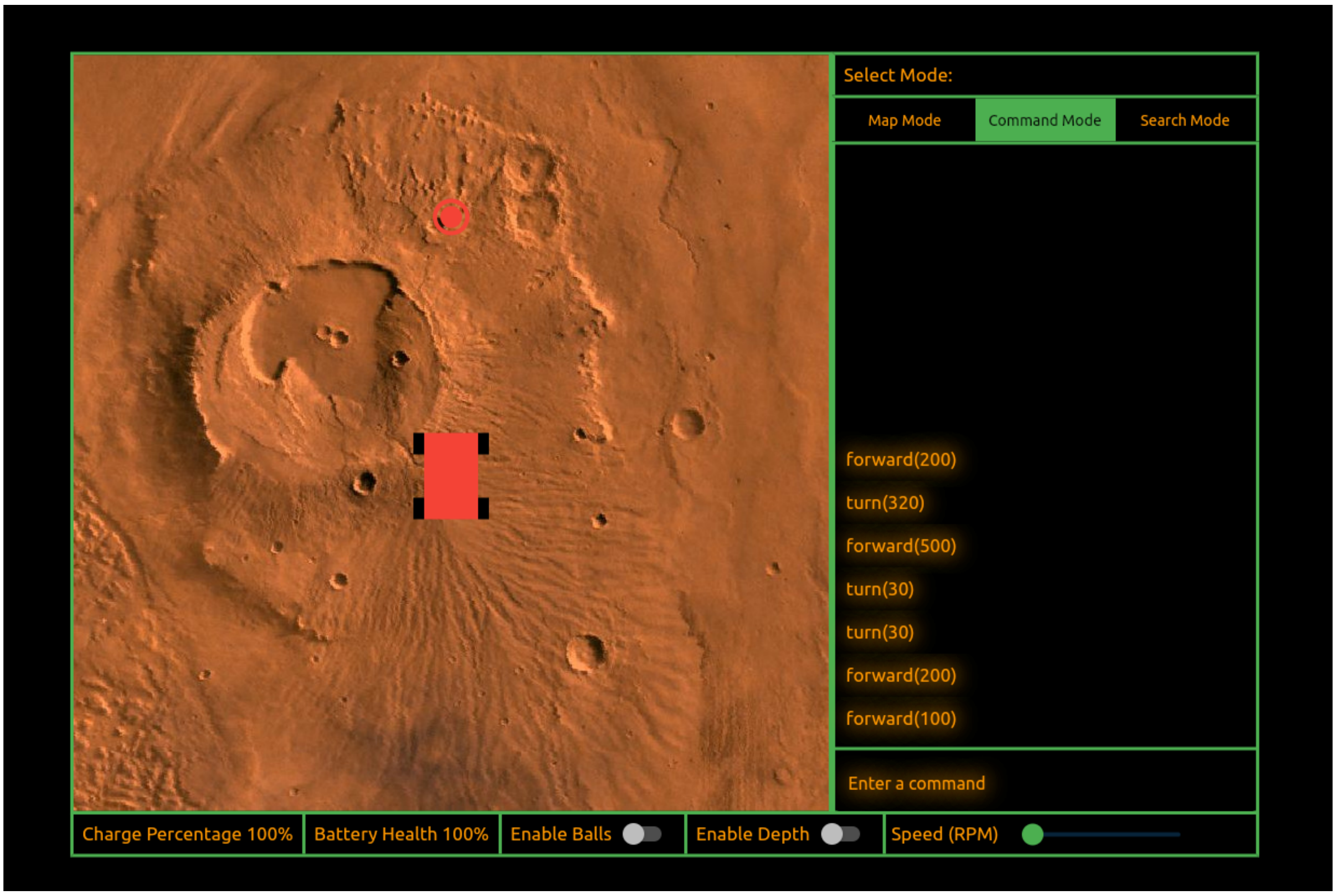
Click on the map to set the rover target coordinate

Target Coordinate X:0, Y:200

Charge Percentage 100%

Battery Health 100%

Figure 1: Interface Map Mode



```
forward(200)
turn(320)
forward(500)
turn(30)
turn(30)
forward(200)
forward(100)
```

Charge Percentage 100%
Battery Health 100%

Figure 2: Interface Command Mode

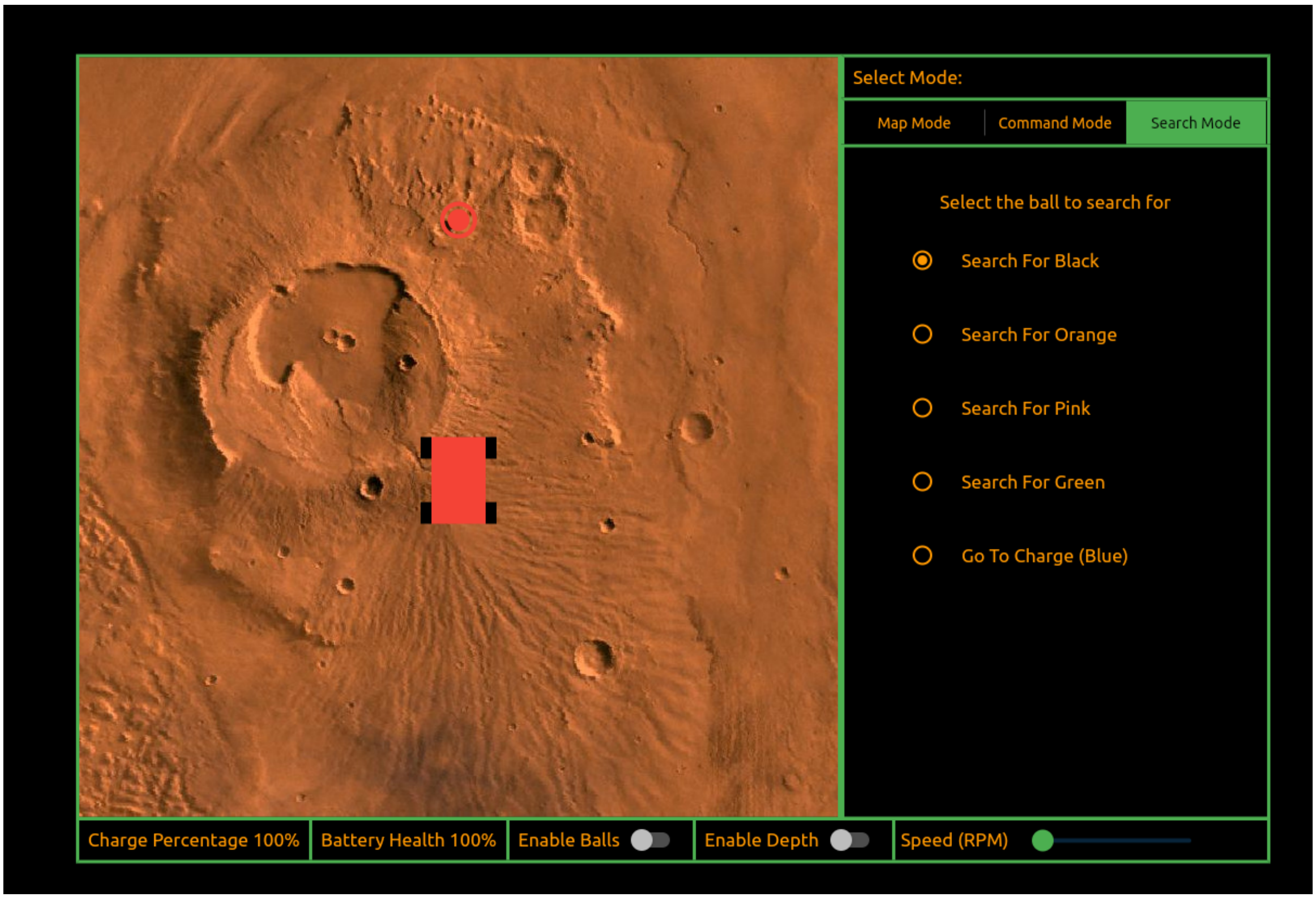


Figure 3: Interface Search Mode

References

[1] 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, 9 2009. pages 14

[2] D. Choudhary and A. Ratna Saxena, "Incremental Conductance MPPT Algorithm for PV System Implemented Using DC-DC Buck and Boost Converter," Tech. Rep. 8, 2014. [Online]. Available: www.ijera.com pages 16

[3] "Luma (video) - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Luma_(video) pages 18

[4] "Spatial Filters - Gaussian Smoothing." [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm pages 18

[5] "Verilog Coding Tips and Tricks: A Verilog Function for finding SQUARE ROOT." [Online]. Available: https://verilogcodes.blogspot.com/2017/11/a-verilog-function-for-finding-square-root.html pages 19

[6] "Hue - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Hue pages 21

[7] "A* search algorithm - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm pages 26

[8] "A* Search Algorithm in Python — A Name Not Yet Taken AB." [Online]. Available: https://www.annytab.com/a-star-search-algorithm-in-python/ pages 26, 37

[9] "Implementation of A*." [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/implementation.html pages 26, 37

[10] "What is Intellectual Property?" [Online]. Available: https://www.wipo.int/about-ip/en/ pages 40

[11] "Intellectual property and your work: Protect your intellectual property - GOV.UK." [Online]. Available: https://www.gov.uk/intellectual-property-an-overview/protect-your-intellectual-property pages 40

[12] "Ampsplus 14500 3.2V 500mAh Battery." [Online]. Available: https://www.ampsplus.co.uk/ampsplus-14500-3-2v-500mah-battery-button pages 41

[13] "(No Title)." [Online]. Available: https://static.rapidonline.com/pdf/502676_v1.pdf pages 42