Imperial College London

Mars Rover Project

ELEC50003/ELEC50008 COMPUTER ENGINEERING/ENGINEERING DESIGN PROJECT 2020-2021

Group 8

Jianing Li (CID: 01704428)

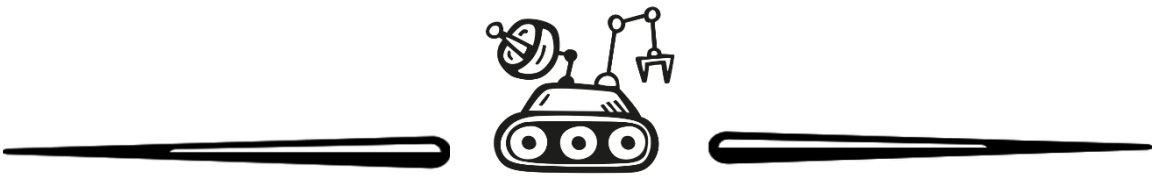
Weng Hou Sio (CID: 01706141)

Wendy Gao-Yin (CID: 01713437)

Kee Leong Koh (CID:01723007)

Jing Lin Loh (CID: 01718534)

Husna Zulkarnain (CID: 01754444)



Markers

Edward Stott
Zohaib Akhtar
Philip Clemow
Muhammad Sharjeel Javaid
Adam Bouchaala
Hakan Merdan
Esther Perea

Date of Submission: 15TH June 2021

Contents

1. Abstract

Autonomous rovers are useful for planet exploration, where humans cannot go. They are more cost effective and can yield valuable information about foreign land, which can pave the way for more exploration in future.

This report follows the development of an autonomous mars rover which has the capability to navigate and map a terrain with obstacles of numerous types. The project was composed of several subsystems that communicate between each other to process and display data. Each subsystem can be created separately and utilised the skills of the team to reach a suitable product.

The final product is a rover that can follow both user input commands and calculated path commands based on orientation, speed, and current location. New obstacles can be identified, and existing obstacles can be avoided, and this information is displayed on a dashboard hosted on the internet. There is also a character-based code to minimise transmission errors.

The report will go through conception based on a brief to design, development and evaluation of the finished product, concluding with recommendations for the future and discussion of the achieved outcomes.

2. Introduction

An autonomous rover can be used for mapping unknown terrain without exposing people to danger whilst having greater flexibility and processing capabilities. It can be used to aid exploration to build up models of land and can go to difficult areas that would otherwise be costly in terms of time and finances.

The rover should be able to send status information, to be stored offsite. The project uses experiences gained throughout the year, where knowledge gained from previous modules are applied to a practical problem, and skills related to project planning and management can continue to be built on.

Using the provided brief, the team generated a set of functional and non-functional requirements based on key features to be implemented. The system is divided into submodules from which module leaders could produce design criteria and focus research on the most pivotal parts of the project.

3. Project Outline

The task involves creating a rover that can use computer vision to detect obstacles of numerous colours(types) and safely traverse a given area. This is used to build up a map of the working area in a cloud database, which can then be displayed via a dashboard, which can also control the rover. A charging station should be created to charge the rover. All the modules are brought together via a central control hub.

4. Design

The product design specification details key system features. Appendix-I/II contains further information.

4.1 Functional Requirements

Drive

Inside the Mars Rover, the drive subsystem enables movement of the rover. The functions of the drive subsystem are as follows. Firstly, it must receive instructions from the control subsystem, interpret the information contained inside the instruction correctly and store that information in an appropriate format as variables. The information includes the speed, direction, and distance that rover to be travelled. Secondly, the rover should use the information stored to make appropriate movements. At this stage, three important functional requirements are set to enable the rover to: 1) Move at the desired speed, 2) Move to the desired position, and 3) Turn to the desired angle. Finally, after the motion has been completed, the drive subsystem must send back the coordinates of the rover so that the control/command subsystem knows the exact location of the rover. At this stage, the functional requirement is 4) Record the real-time coordinate of the rover during movement and send information back to control.

There are also additional requirements for the drive subsystem and these criteria will be used in the evaluation stage to evaluate the rover performance. These requirements are set to such to ensure that 5) The rover remains stable during movements and 6) The rover position is as accurate as possible.

Vision

The high-level goal of the vision subsystem is to prevent the rover colliding with obstacles while it moves around a given area and terrain, by using a D8M camera and a DE-10 FPGA board for obstacle identification. This is achieved by implementing various intermediate components in the image processor. In the given circumstances, obstacles are well-defined, as they are ping-pong balls with different colours, namely red, blue, orange, green, and pink. One key requirement in the design is that the image processor is capable of uniquely identifying each obstacle, in which the colour of the obstruction can be indicated. The vision submodule is also required to send appropriate commands to the motor control such that the rover can avoid obstacles as soon as they are detected.

Energy

This division must provide sufficient power to motor and ESP32 and configure the solar panels to receive maximum amount of sunlight at different times of the day using a MPPT algorithm. The battery also must be chosen so that it can supply power to ESP32, motor and FPGA either in series or parallel. The best charging strategy must be decided upon and SOH and SOC estimations are needed to maintain batteries' health. The whole division also must communicate with the Drive, Control and Command Module for integration purposes either remotely or physically. It also must send and receive data to the various subsystems to enable the over working efficiently.

Command

The high-level goal of this subsystem is to create a dashboard to control the rover remotely and send and display information regarding energy, vision, and drive subsystems. Key functional requirements include saving rover information in an offsite area and allowing the rover to explore autonomously. Obstacles encountered should be tracked on a map, and real time information should be displayed where possible. The subsystems will communicate via a Control module, where a communication protocol must be established.

Moreover, it is expected that the command module can send instructions to reroute around obstacles. Notifications will alert the user to obstacles and client-side interaction support will focus on features to simplify navigation and enlarge dashboard features to fit on various devices. On the server-side, content needs to be handled carefully with real time updates at a suitable refresh rate. Additionally, server-side management of large data sets is required, to synthesize information from the database effectively as the rover travels further.

Control

A Mars rover has several individual modules that needs to work together for it to form a functional rover. In this project, the modules are the Drive module, the Vision module, and the Energy module. Furthermore, to remotely control the rover, communication protocols between the Command module and the rover is needed. To interconnect each of the modules in the rover and enable them to communicate remotely with command would lead to a very messy design that is a waste of resources, as well as having more full duplex connections which may be physical or wireless or a combination of both, which leads to higher rover weight and inefficient use of bandwidth, respectively. Furthermore, the person designing each module must agree with every other module's designer on how they should inter-communicate and interoperate, which makes the design stage form themselves harder.

To overcome that, the goal here is to design a Control module, which acts as an intermediary between all modules shown in Figure 1, hence providing the rover with a clean and clear-cut design based on centralized control, as shown in Figure 2.

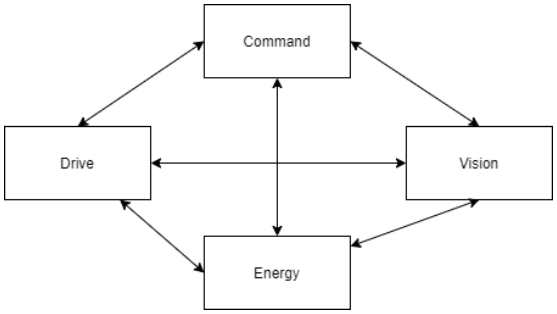


Figure 1: Intercommunication Requirements

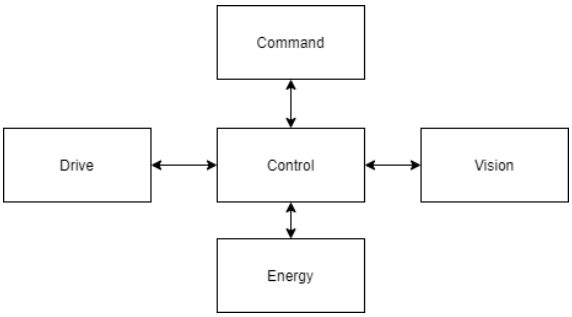


Figure 2: Control Intermediary

In this architecture of the Mars rover, not only did the team cut down on 2 out of the 6 original communication lines, the designers of each of the individual modules only need to focus on interfacing with the Control module. This dramatically simplifies their individual workload, and the necessary scheduling and coordination will be handled by the Control module instead.

Integration

To get a fully working rover, the integration subsystem acts as the aggregation of all subsystems to meet the required functionalities. The integration subsystem is responsible in verifying the functional requirements of each subsystem. Furthermore, the integration subsystem must validate the extent of the conditions that the final rover can perform well in. The requirements here are mainly to manage overall processing and manage communications physically. Figure 3 shows the final form of the rover.

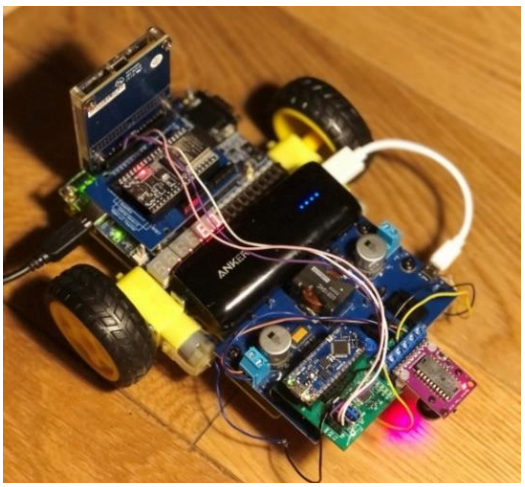


Figure 3: Assembled rover

4.2 Non-Functional Requirements

Reliability

Individual submodules should be debugged prior to connecting. The modules should be operational when joined. Communication between submodules should be consistent and chosen protocols should be reliable.

Performance

Latency between modules should be minimised for real-time display of data. Arduinos have limited storage, and FPGAs have output lags, so calculations should be divided between submodules to maximise performance.

Scalability

The software should allow easy implementation of new features. The hardware should be replicable and thus allow a network of rovers to be created if necessary. The dashboard should be compatible with other devices.

4.3 Initial Structural and Functional Diagrams

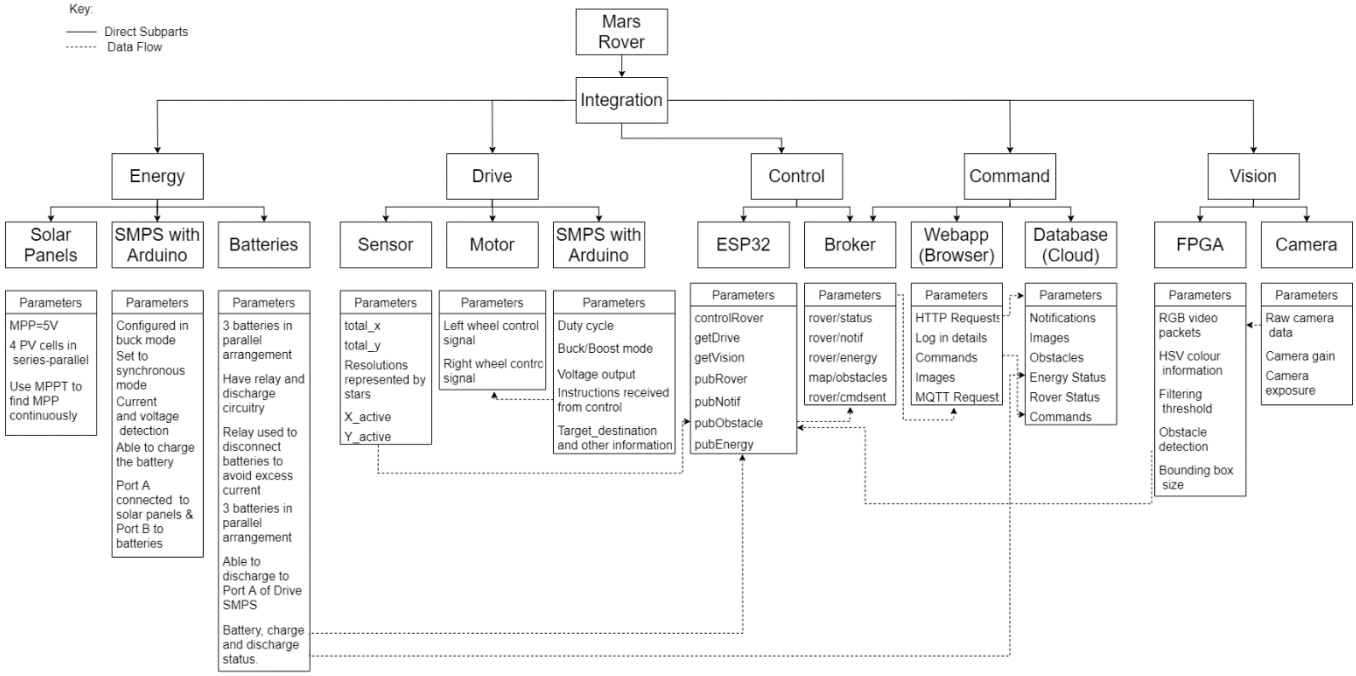


Figure 4: System Structure

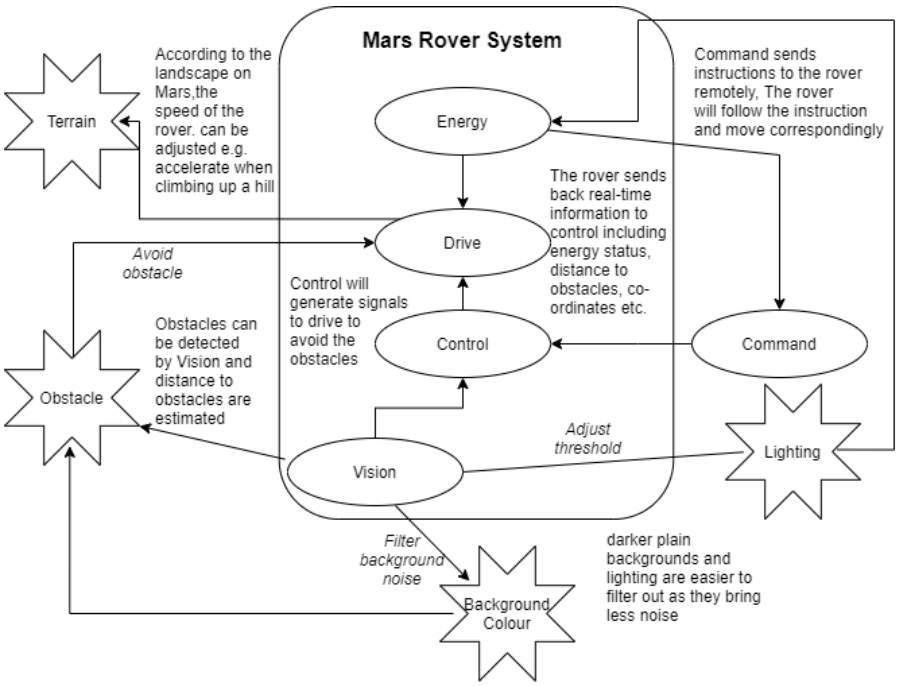


Figure 5: Functional Diagram

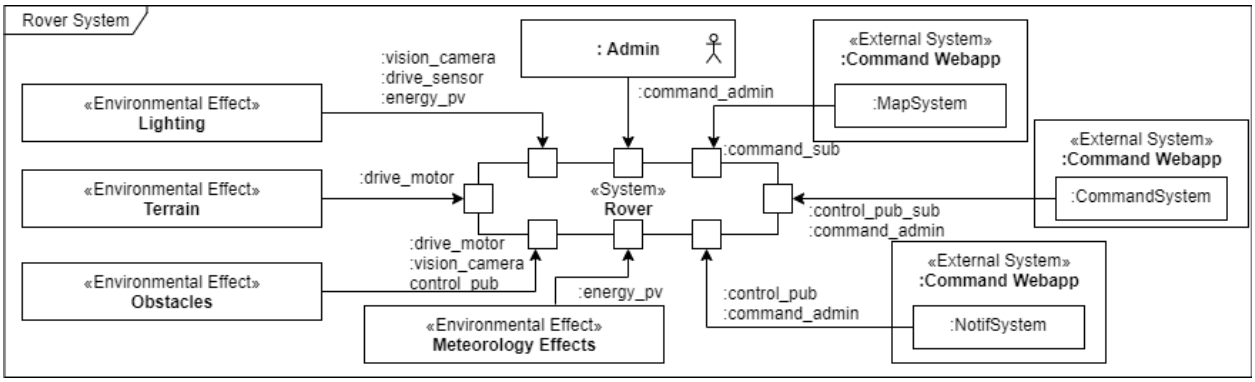


Figure 6: Context Diagram with external interactions

5. Development and Implementation

5.1 Drive Module

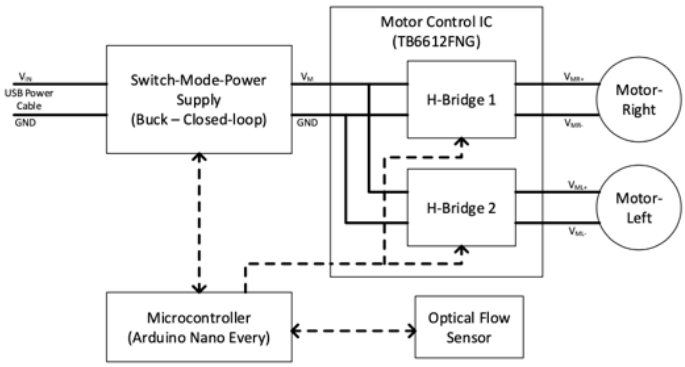


Figure 7: Block Diagram of motor Drive Module

Structural design of the Drive subsystem

The above diagram illustrates how the drive subsystem is designed on a structural level. The Arduino Nano Every is the key microcontroller in the drive subsystem, controlling three main modules. Firstly, with the SMPS in Buck mode, the code utilises a closed-loop control strategy to regulate the voltage output at nearly fixed value, Vm. Vm also determines the rotational speed of the motors. Hence, the speed of the rover can be controlled via controlling Vm.

Secondly, the microcontroller can also control signals sent to the H-bridges on the motor control IC. This will determine the HIGH/LOW state of the motors. Since wheels are attached on the motors, setting the left/right motor state will determine rover movement. When both wheels are rotating clockwise, the rover will move forward. When the left wheel rotates clockwise whilst the right wheel rotates anti-clockwise, the rover will turn left. As a result, the direction of motion can be controlled via setting the states of motors.

Finally, the Arduino also communicates with the optical flow sensor (via a SPI port) to measure the distance moved by the rover. The sensor returns 2 values (`total_x, total_y`) in a coordinate format. It is important to note that these 2 values do NOT represent the exact co-ordinate of the rover. `total_y` records the distance that the rover moved in vertical direction; when the rover is moving forwards/backwards, the value of `total_y` will increase/decrease correspondingly. On the other hand, `total_x` records the circumference that the sensor has swept when the rover is turning. When turning left/right, the value of `total_x` will increase/decrease correspondingly.

The real-time values (`total_x, total_y`) have 2 main uses. Firstly, the values could be used to control the movement of the rover. Secondly, after careful manipulation, the data can be used to calculate the real-time coordinates of the rover during each movement.

Functional design of the drive subsystem (Black box level)

To implement the features described above, the drive subsystem must first agree with the control subsystem on the format of the instruction being sent to drive. After discussions, it was decided that each time an instruction is sent, it will only be one of the 5 types: Moving forward(f), Moving backward(b), Turn left(l), Turn right(r), Stop(s). Each of these instructions comes with a speed and a value. The instruction format will be in '*xdxxx*' , with the first '*x*' as an integer indicates the speed of the current instruction in a scale form 0-9 (9 representing maximum speed). The second character '*d*' could be any of the five types of instructions (f,b,r,l,s). The final *'xxx'* are integers that contain angle/distance values. To illustrate, instruction *'5f010'* = Rover move forward by 10cm in medium speed. *'5r090'* = Rover turn right by 90 degrees in medium speed.

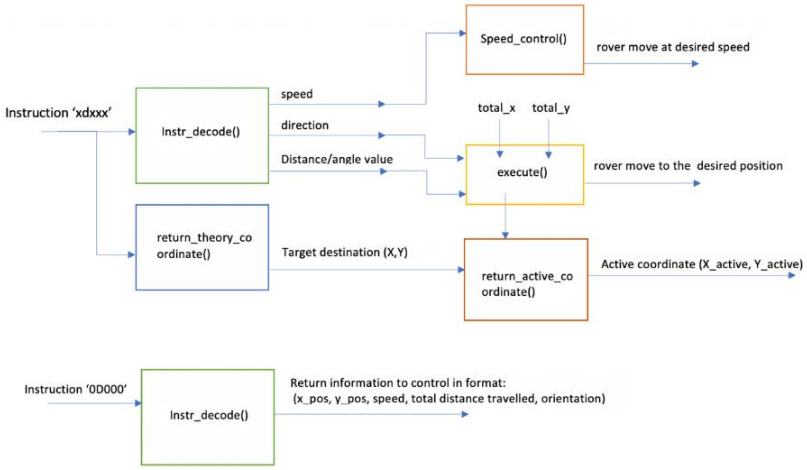


Figure 8: Functional Diagram

The above diagram shows how the rover performs its required tasks on a functional level. After knowing the instruction format, an `instr_decode()` function can be implemented to decode and extract the necessary information. A `return_theory_coordinate()` function is also set up to calculate the target destination of the rover after knowing the instruction to be executed. However, these two functions must only be executed once when a new instruction is sent (not continuously in a loop).

In order to make the rover move in the desired direction and distance, an `execute()` function was implemented. In this function, the rover used the positional/angular error to determine whether the rover has moved to the desired location or turned to the desired angle. If the desired distance/angle is not achieved, the rover will keep moving until reaching that target position/angle. More details of this function will be given in the implementation stage.

A `speed_control(velocity)` function will control the rover speed according to values in the instruction. Controlling the speed of an instruction is necessary since sometimes when the Mars rover is driving through a swamp field or climbing up a hill, it will require greater acceleration and velocity. Similarly, when the rover is turning to another direction, it would be better to move at a lower speed for better angle accuracy.

Finally, the `return_active_coordinate()` function will return the real-time coordinate of the rover when the rover is moving, unlike `return_theory_coordinate()` which would return a theoretical set of target coordinates. This real-time tracking of rover position is necessary for the command subsystem to draw the map of the moving traces of the rover at a later stage. Details will be explained in the implementation stage.

Implementation

Inside the black box:

a) `instr_decode()` function

```
void instr_decode(String instr){

    velocity = (instr.substring(0,1)).toFloat();
    direct = instr[1];
    value = (instr.substring(2)).toFloat();

    if(direct=='f'){

      velocity_ = velocity/10;
      target_distance_y = value;
      temp_y = temp_y + target_distance_y ;
    }

    if(direct=='b'){

      velocity_ = velocity/10;
      target_distance_y = - value;
      temp_y = temp_y + target_distance_y ;

    }

    if(direct=='l'){

      velocity_ = velocity/10;
      angle_turning = value;
      total_angle_turned = total_angle_turned + angle_turning;
      temp_x = temp_x + angle_turning*PI*15.8/180;

    }
    if(direct=='r'){

      velocity_ = velocity/10;
      angle_turning = - value;
      total_angle_turned = total_angle_turned + angle_turning;
      temp_x = temp_x + angle_turning*PI*15.8/180;

    }

    if(direct=='s'){

      velocity_ = 0;
      temp_y = total_y;
      temp_x = total_x;
    }
```

The `instr_decode()` perform two operations. Firstly, it decodes the instruction string and stores it into certain variables. The second operation is the KEY step: based on which of the 5 keywords were sent, the value inside the instruction is accumulated on corresponding 3 variables. These three variables are `temp_x`, `temp_y` and `total_angle_turned`.

`temp_y` is responsible for accumulating the rover movement in the forward/backward direction, `temp_x` is used to accumulate distance that the rover has turned left/right. `total_angle_turned` records the orientation of the rover relative to its starting position.

When the rover is turning right or left, the value $angle\_tuning * PI * 15.8/180$ is incremented/decremented on `temp_x`. This is because the value recorded on `temp_x` is the distance of the circumference that the sensor has swept during the turning. The formula calculates how much `temp_x` has changed based on the angle given.

When a stop is sent in the middle of an instruction, `temp_x`/`temp_y` automatically refreshes to the value of `total_x`/`total_y`. This ensures the distance untraveled in the current instruction is dumped before the sending of the next instruction.

b) `return_theory_coordinate()` function

```
void return_theory_coordinate(){

    if(direct=='f'||direct=='b'){
        Y = Y + target_distance_y*cos(total_angle_turned);
        X = X - target_distance_y*sin(total_angle_turned);
    }
    if(direct=='l' || direct=='r'){
      X = X;
      Y = Y;
    }
    if(direct=='s'){
      X = X_active;
      Y = Y_active;
    }
}
```

The purpose of this function is to calculate the theoretical location of the rover after the current instruction is finished. Note that the coordinate will only change during forward/backward instruction as the center of the rover was set as the origin. Hence when the rover turns left/right, there will not be a change in coordinates.

The calculation of changes on X and Y are standard geometry; `target_distance_y` records the distance for the rover to be moved and the `total_angle_turned` indicates the angle that rover has turned relative to its starting position. As a result, the two pieces of information can be used to calculate theoretical (X,Y) positions.

When a stop instruction is sent, it means that the rover should be stopped immediately during the execution of one instruction. Hence, the `target_coordinate(X,Y)` is set to the actual co-ordinate of the rover, indicated by (`X_active`, `Y_active`). This is a very important feature since it allows refreshing of the theoretical coordinates of the rover if the current instruction is stopped in the middle of action.

c) `execute()` function

```
void execute(){

  if(direct=='f'|| direct=='b'){

    y_error = temp_y - total_y;
    y_error_pid = pid(y_error);

      if(y_error_pid > 0.2){
          go_forward();
          }
        if (y_error_pid < -0.2 ){
          go_backward();
          }
        if (y_error_pid > -0.2 && y_error_pid < 0.2) {
          stop_rover();
          Serial.println("x");

        }
  }

  if(direct=='s'){
       stop_rover();
        Serial.println("x");
  }
```

```
if(direct=='l'||direct=='r'){

      x_error = temp_x - total_x;
      x_error_pid = pid(temp_x - total_x);

      if( x_error_pid > 1){
           go_left();
      }
      if( x_error_pid < -1){
          go_right();
      }
      if( x_error_pid > -1 && x_error_pid < 1){
          stop_rover();
          Serial.println("x");
      }
 }
```

The `execute()` function is the key code segment that controls the rover's motion. The `y_error` is the variable that monitors the distance error in the moving forward/backward stage, and `x_error` is the variable that monitors the `angle_error` in turning left/right stage.

The principal control strategy behind is the combination of a threshold controller and a PI controller. Firstly, the error term is constantly monitored inside the `loop()` function of Arduino. Based on the signs of the error terms, Arduino will send control signals to the servo motors on board that will determine the direction of rover. If the error term finally reduces to zero, the motor will be disabled, and the rover will come to a stop.

However, when this strategy is implemented in practice, experiments have shown that the rover will not stop; instead, it will oscillate around the desired position. There are two main reasons. The first reason is that there is a delay between the sensor value returned and the actual position of the rover: this is a hardware problem and cannot be addressed by code. The second reason is there is very little probability that the error terms will eventually settles to zero, if not, the rover will continuously be moving forward/backward and shows oscillating behaviour. This problem can be addressed by giving the error term a threshold region. Once the error term enters that threshold, the rover will be stopped. To ensure the accuracy of the movement, the threshold region should be set quite small. In this case, the threshold is set to +/- 0.2cm, which is accurate enough for the rover as well as preventing the oscillation problem.

The method outline above in nature was a proportional controller with Kp = 1. To further improve the accuracy and reduce the steady state error of the rover, an Integral controller is also implemented. The nature of the integral controller is that it accumulates the error with time. If the rover stopped at a position very close to the desired position, the error is called a steady-state error. Although in practice, if the threshold is set small enough, the error will be very small, but with the integral controller, this error term will become larger and larger with time. As a result, when the error term finally exceeds the thresholds, the rover will autocorrect itself to the desired position. The parameter for the integral controller selected is usually very small compared to the proportional parameter, typically Ki = 0.1. The proportional and the integral controller are integrated together to form a PI controller, which is what was actually implemented in the code.

d) `speed_control()` function

```
void speed_control (float duty){
  analogWrite(6, (int)(255-duty*255));
}
```

The `speed_control()` function is quite easy to implement. The `analogueWrite()` function alters the values on Arduino pin6, which controls the voltage output of the buck SMPS.

e) `return_active_coordinate()` function

```
void return_active_coordinate(){

   if(direct=='f'||direct=='b'){
       Y_active = Y - y_error*cos(total_angle_turned);
       X_active =  X + y_error*sin(total_angle_turned);
    }
    if(direct=='l' || direct=='r'){
      X_active = X_active;
      Y_active = Y_active;
    }
}
```

The `return_active_coordinate()` function actually utilized the theoretical coordinate (X,Y), `y_error` and the `total_angle_turned` term to get the real-time position of the rover. The key concept is that it performs mathematical transformation on the `y_error` term (which is an actively changing variable) to determine the active real-time position of the rover. Similarly, when the rover turns, no change will occur on the (`X_active, Y_active`) coordinate since the origin is defined at the center of the rover.

Inter-module communication:

The Drive subsystem mainly communicate with the control subsystem. On the hardware side, the inter-module communication is achieved via 3-way socket on the motor drive PCB for UART connection. The jumpers can be connected to pins available on the ESP32 via wires. GND is connected to GND. Tx and Rx can be mapped to any pins on ESP32 depend how the connection is set up.

The communication take place in two forms, receiving and transmitting. To begin with, the drive subsystem must receive the instructions from the control subsystems via Rx port. The instruction was then read and decoded by the program and finally executed. The instruction format was previously agreed between the systems. Now how to send those instructions became an important problem. Finally, the team have decided that the instruction should be sent sequentially. When the rover has reached the desired position, the drive will respond to the control by sending an *'x'* character telling the instruction was successfully executed. Only if the control receives this '*x*', it will then send drive the next instruction.

The transmission part was practically very similar to the receiving part. Agreements have been made such that when the control subsystem sends an instruction *'0D000'*, the drive subsystem should return all the necessary information of the rover in a space delimited string. This string contains the information as the format follows: "`x_active y_active speed distance_travelled orientation`".

And in code it was implemented as follows:

```
if(direct == 'D'){
 Serial1.println(String(X_active) + " " + String(Y_active) + " " + String(velocity_) + " " + String(temp_y) + " " + String(total_angle_turned) + ";");
}
```

It is vital for the control subsystem to know the current position and the orientation of the rover. This information will be forwarded to command and command can use the information to draw the moving traces of rover. Furthermore, the control subsystem could avoid obstacles together with the information returned from the vision subsystem. The information on the total distance travelled could also be utilized by the energy subsystem to estimate how much energy the rover consumed and how much further it will travel.

Implementation problem solving (with evidence):

There was a series of problems arises during the implementation stage. In this section, some of the problems arises and method to solve them will be discussed.

To begin with, the optical sensor detection range was limited to around +/-206 mm. (`total_x` and `total_y`). This problem must be fixed otherwise the maximum range for distance measurement will be limited to +/-20cm, if the rover moves beyond 20cm, values will change signs and create problems for the position control. The reason for this limited range is due to `int` inside the variable declaration for `total_x1` and `total_y1`. The `int` inside Arduino is only 16 bits hence the maximum value for it could be around +/-32000, after the scaling down to `total_x` and `total_y`, the range will further reduce. Hence, the `int` was replaced by `long` which is 32bits to increase the detectable distance range.

The delay in the optical sensor also creates problems for position control. This is the delay between the rover movements and the distance value that sensor measured. The sensor values tend to appear slower than the actual rover position. The solution is to remove all the `delay()` code segments in the codes for optical sensor. If one would like to further reduce the delay, he could operate the rover at lower speed so that the sensor has enough time to react.

Before the speed control was implemented, the rover speed was not constant after starting up. There was a significant acceleration followed by a deceleration, before settling at a constant speed. After investigation, the reason was due to the close-loop control of the Buck SMPS voltage, as shown in the diagram below. This is an undesirable effect as it is better for the rover to be moving at a constant speed for better position control accuracy. The solution was to directly modulate the output volage of the buck SMPS for each different instruction.

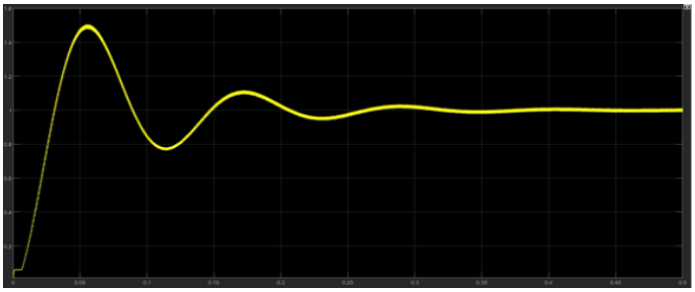


Figure 9: close-loop control of the Buck SMPS voltage

Evaluation on the drive subsystem performance:

After the implementation process, extensive testing was carried out and verified that all the functional requirements for drive subsystem were met. To further evaluate the performance of the drive subsystem, the 3 evaluation matrices are used below:

1) Could the rover move at different speed?
2) Does the rover remain stable during the execution of instruction?
3) Is the final position of the rover accurate enough?

For the first evaluation criteria, a series of instructions was given to the drive subsystem: *5f010, 4r090, 6f010, 3b010, 9f010* and the rover's motion was observed. Once the speed control was implemented successfully, the rover could move at different speed for different instructions. Likewise, when the speed was below 4, the rover will move very slowly since the motor cannot produce enough drive to move the rover. Hence it is advised that all instruction should be performed under speed greater or equal to 4.

The stability of the rover is very important since the Mars rover should not oscillate once it reached the target position. Several tests were carried out and the rover was proved to be stable enough for most instructions with speed lower than 7. If the speed of the rover is above 7, the rover will move significantly faster, while the values returned by the sensor still has a delay. When the control signals were sent to the drive motors, a large positional error was created in a very short period of time, at which point the rover started to auto correct its position, and the fast speed made it correct more than needed. In the end, this process goes on and on until finally an 'oscillation' type of problem was observed on the rover. Hence, it is advised that the speed of rover should smaller or equal to 7 for better stability performance.

The third evaluation criterion is important as well. After careful measurements of the distance travelled by the rover, it can be concluded that the rover is accurate enough to move to the desired position with reasonable speed. The distance error tends to fall in the region between [-0.7, 0.7] cm, which is reasonably acceptable. There are mainly two contributions to this error, the first one being the inaccuracy of optical sensor detection which cannot be further improved with the existing hardware. The second reason is due to the error threshold implemented in the position control logic. However, this threshold is also necessary ensure the rover stability. If the threshold region is set to be very small, the rover position control will become more accurate, but the stability will degrade since the rover will tend to 'oscillate' around the desired position and vice versa. Hence, it is important to find out the optimum threshold region that will maximize both positional accuracy and rover stability.

5.2 Vision Module

Structural Design of the subsystem

The following diagram shows the internal structure of the vision subsystem [1].

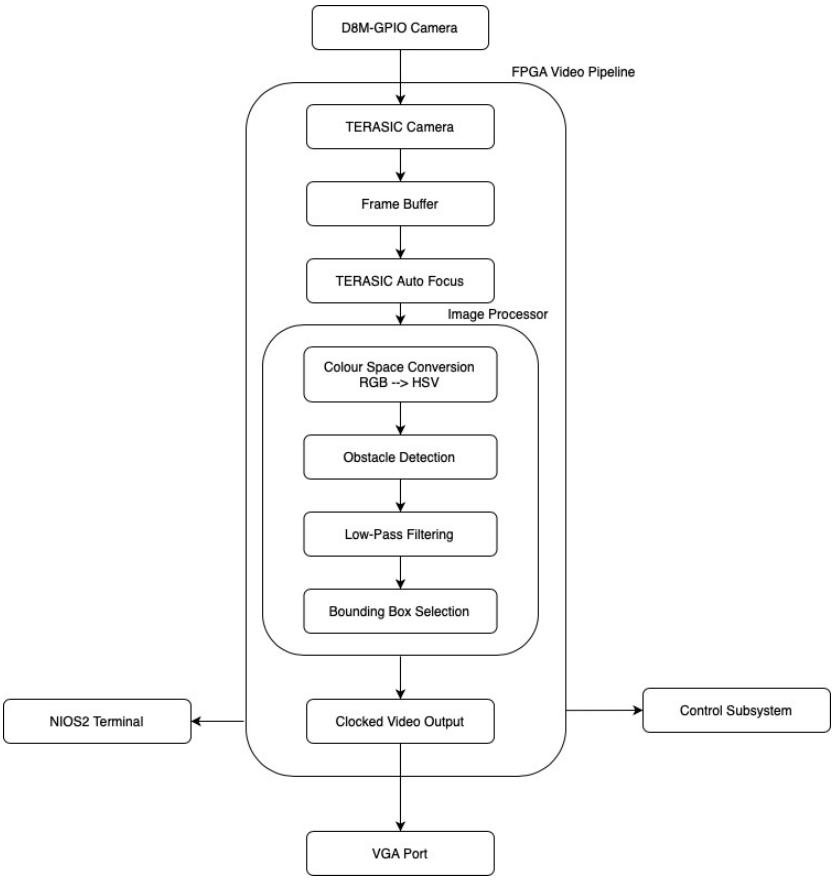


Figure 10: Vision Subsystem Structural Design

Recall from Design, that the high-level goal is achieved by implementing different intermediate components in the image processor. The image processor is composed of four main components, namely the conversion algorithms of colour space, obstacle detectors, mean filter, and the bounding box selector. The main purpose of colour space conversion and filtering is to transform the video such that it is more suitable for obstacle detection and ultimately improve the accuracy of the obstacle detector. The bounding box selector is a tool for choosing the correct obstacle automatically, based on counters within the image processor.

Algorithms for obstacle detection

In the context of image processing, different algorithms can be applied to improve the accuracy and the precision of object detections. However, there are several factors that affect the quality of the output significantly. In particular, image noise is one aspect of electronic noise that arises due to random variation of brightness and colour information in images [2]. Examples of image noise include Gaussian noise, salt-and-pepper noise and shot noise. These noises are inevitable when capturing an image from the camera but there are various algorithms to reduce the effects of such noises.

One well-known approach to suppress the effects of image noise is to apply a mean filter, which is a linear filter, and it works by averaging a specific pixel value with its neighbour values. This has the effect of eliminating pixel values that are unrepresentative of their surroundings and ultimately removing random noises in the environment. Another type of filtering technique is called the median filter and it is a non-linear filter. This type of filter removes parts of the noise by considering each pixel in turns and calculate the median value based on its surrounding pixel values. Other filters are also very useful in removing various noises.

Each image the camera captures contains 307,200 pixels of information and 24 bits of colour information are stored within each individual pixel. In RGB colour space, 3 fundamental colours of light are used to represent

each pixel, namely red, green, and blue. Each colour component is assigned to an 8-bit value, which corresponds to the light intensity of the colour source.

(a) Mean Filter with kernel size 4x4

There are many benefits of using mean filter over other filters. The main advantages are that the mean filter is easy to implement, modify and is effective in removing noises. Choosing a kernel size is also critical since it would determine how much the detected areas are smoothed out. Clearly, a larger kernel size should be avoided as parts of the targeted object will also be suppressed. Similarly, a small kernel size has little effects in removing noises. Therefore, a mean filter with a 4x4 square kernel is used to filter out unwanted noises where the kernel size is neither too large nor too small. Comparisons are made before applying the filter and after the filter is applied, and they are discussed in the next section.

The way it is implemented is slightly modified to allow a more flexible change in the threshold, which ultimately can accommodate colours that are harder to detect, by lowering the threshold in the filter.

The implementation involves the storage of bit values, which indicates the colour detected for each pixel and these bit values are stored in four rows of shift registers. These shift registers allow subsequent bit values to be calculated based on bit values from previous pixels.

The following pseudocode illustrates the implementation of the mean filter.

```
// (x,y) indicates the location of the pixel in an image of size 640*480
// col_detect is a 1-bit value indicating
// row1, row2, row3, row4 are 4 rows of shift registers
if (y == 0):
    row1[x] = col_detect
    output = col_detect
    if(x == 639):
        x = 0
        y = y + 1

else if (y == 1):
    row2[x] = col_detect
    output = col_detect
    if(x == 639):
        x = 0
        y = y + 1

else if (y == 2):
    row3[x] = col_detect
    output = col_detect
    if(x == 639):
        x = 0
        y = y + 1
else:
    row4[x] = col_detect
    if(x <= 3):
        output = col_detect
    else if (x == 639):
        x = 0
        y = y + 1
        row1 = row2
        row2 = row3
        row3 = row4
    else:
        output = ( (col_detect + row4[x-1] + row4[x-2] + row4[x-3] +
                    row3[x] + row3[x-1] + row3[x-2] + row3[x-3] +
                    row2[x] + row2[x-1] + row2[x-2] + row2[x-3] +
                    row1[x] + row1[x-1] + row1[x-2] + row1[x-3]) >= threshold )
```

Figure 11: Implementation of mean filter

The threshold is usually set to 16, in order to remove all possible noise in the image. It is obvious that there is a trade-off between the amount of noise in an image and the amount of detected area of an obstacle. By lowering the threshold, the obstacle would be fully detected while some noise would appear in the surroundings. In contrast, noise would be mostly suppressed, and the detected area of an obstruction would be smaller by setting a higher threshold. In the current applications, the latter is preferred as the bounding box is very sensitive to noise and therefore, noise should be suppressed as much as possible.

(b) Colour Space Conversion

The main goal of colour space conversion is to improve the quality of images by converting one colour space into another colour space. This is vital as the colour space models provide a typical scheme of representing colours in images [3]. In the current applications, HSV colour space is more suitable than RGB colour space and there are many arguments for supporting this statement.

Firstly, HSV colour space is more suitable for distinction of colours than RGB colour space and hence, HSV colour space is widely used in applications such as facial recognition and road traffic sign recognition [3]. This directly links to obstacle detections in the current application since colour detection is the key goal in recognising different obstacles.

Secondly, the R (Red), G (Green), and B (Blue) components describing an object in the RGB colour space are all correlated with the amount of light hitting the object in the image, and hence, image descriptions in terms of these components make object discrimination difficult [4]. On the other hand, image intensity and colour information are separated in the HSV colour space. This suggests that the performance of obstacle detection is preserved under bright or dark conditions in the HSV colour space.

Although the computational complexity of converting the RGB colour space to HSV colour space is large, the entire conversion can be done in hardware instead of software, since a lot of time will be taken if this is calculated by software [3]. Therefore, the conversion is part of the image processor in the FPGA and enables the detection of obstacles based on the H, S, and V values.

The following equations illustrate how to convert the values R, G, and B in RGB colour space to the values H, S, and V in HSV colour space.

The ranges of H, S, and V values are [0,359], [0,255] and [0,255] respectively.

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

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

$$H = \begin{cases} 0 & if\ max = min \\ 60 * \frac{G - B}{max - min} & if\ max = R, G \geq B \\ 60 * \frac{G - B}{max - min} + 360 & if\ max = R, G < B \\ 60 * \frac{B - R}{max - min} + 120 & if\ max = G \\ 60 * \frac{R - G}{max - min} + 240 & if\ max = B \end{cases}$$

$$S = \begin{cases} 0 & if\ max = 0 \\ 255 * \frac{max - \min}{max} & otherwise \end{cases}$$

$$V = max$$

The actual implementation in the Verilog code follows the equations above very closely and allows ranges of H (Hue), S (Saturation), and V (Value) values to be selected for different obstacles, as shown in the next sub-section. Different range of values have been tested to find out the optimal ranges for each of the obstacle, found in the Appendix.

(c) Range of values used to detect each obstacle & Unique identification

Figure 12 provides details on the ranges of H, S, and V values to uniquely identify each of the obstacles.

	H	S	V
Red	0-19	200-360	200-360
Green	80-130	100-360	100-360
Blue	135-165	100-200	100-200
Orange	40-55	160-360	200-360
Pink	0-20	150-200	150-200

Figure 12: HSV colour Ranges

The image processor is capable in identifying any unique obstacle defined in the specification, namely the red, green, orange, blue, and pink ping-pong balls. This is achieved by implementing five individual counters, each represents a unique colour. As each pixel of the image is either "none detected" or associated with one colour, its corresponding counter counts up by one. As a result, the final output is based upon the counter with the largest value and returns the bounding box of the obstruction with the largest counter value.

Output of the vision subsystem

The output of the vision subsystem is mainly about the obstacle information, which contains two 11-bit values corresponding to the width and height of the bounding box, alongside with a 3-bit value indicating the type of obstruction.

The following table shows the mapping between the 3-bit number value and the indicated obstacle colour.

Colour	Detected value
Red	3'b001
Blue	3'b010
Orange	3'b011
Green	3'b100
Pink	3'b101

Segment of code on the message output:

```
always@(*) begin    //Write words to FIFO as state machine advances
    case(msg_state)
        2'b00: begin
            msg_buf_in = 32'b0;
            msg_buf_wr = 1'b0;
        end
        2'b01: begin
            msg_buf_in = `RED_BOX_MSG_ID;   //Message ID
            msg_buf_wr = 1'b1;
        end
        2'b10: begin
            msg_buf_in = {5'b0, width, 5'b0, height};   //Width and Height of bounding box
            msg_buf_wr = 1'b1;
        end
        2'b11: begin
            msg_buf_in = {29'b0,colour}; //colour of the obstacle
            msg_buf_wr = 1'b1;
        end
    endcase
end
```

Figure 13: message output code

The following image shows the output in the nios2 terminal.

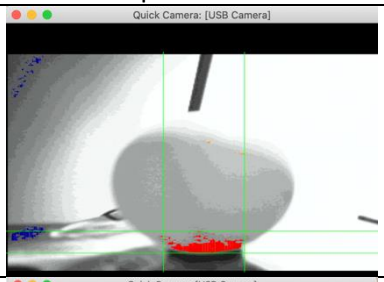
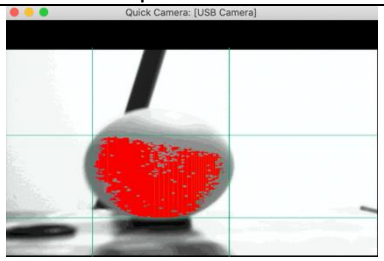
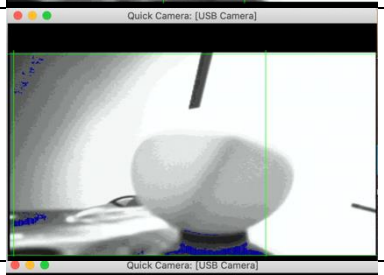
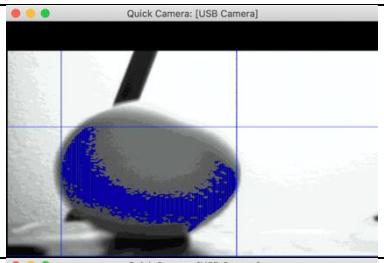
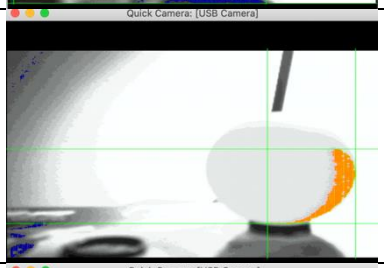
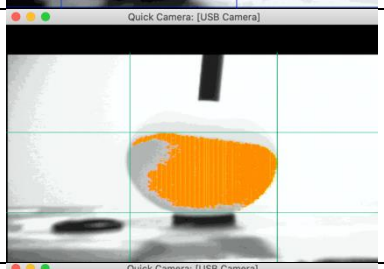
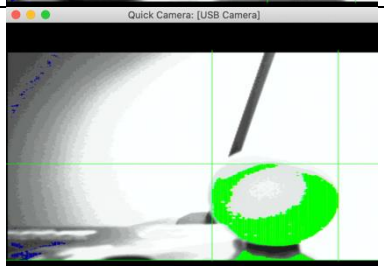
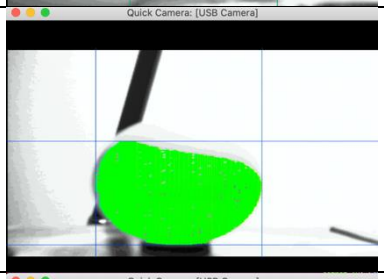
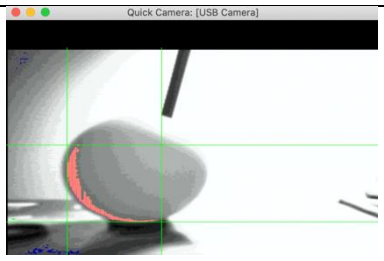
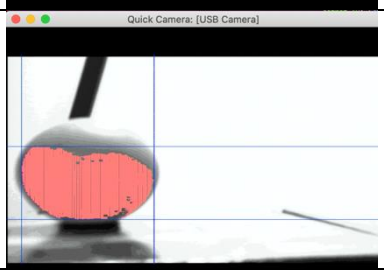
```
00524242 05810621 00000004
00524242 0064001c 00000002
00524242 005d001c 00000002
00524242 006d0014 00000002
00524242 001e0013 00000002
```

Figure 14: nios2 terminal output

The first column is a constant message ID, and this should not be changed as data gets updated. The second column is more important as it contains information regarding the width and height of the bounding box, which is critical in calculating the size and the distance between the obstacle and the rover. Finally, the third column contains a 3-bit value indicating the colour of the obstruction, with zero-extended to keep the consistency of 32-bit messages.

Performance of obstacle detection and Testing

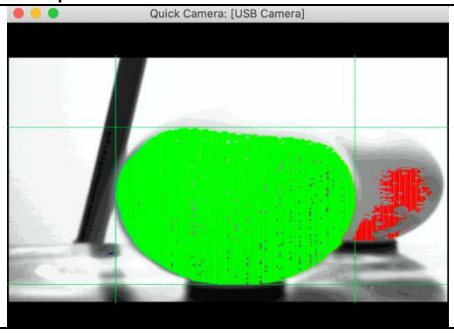
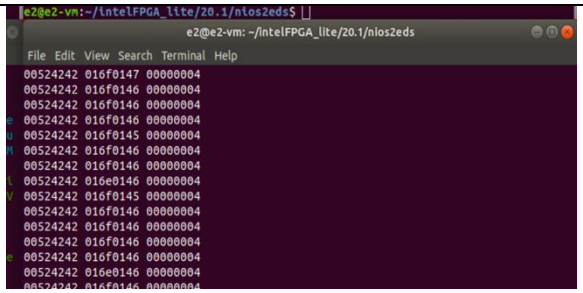
Part 1: Individual Test (HSV vs RGB colour spaces)

Colour	RGB colour space	HSV colour space
Red		
Blue		
Orange		
Green		
Pink		

As shown in the table, obstacle detection has much better performance in the HSV colour space than in the RGB colour space. The detection in the RGB space is very unstable as the light source changes over time and hence, most of the time the bounding box information is inaccurate and valid obstacles are ignored.

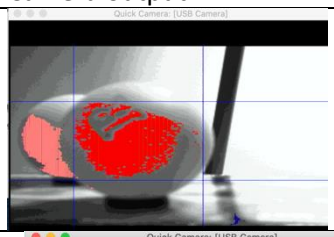
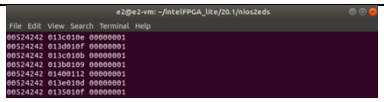
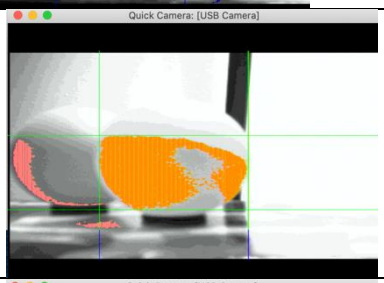
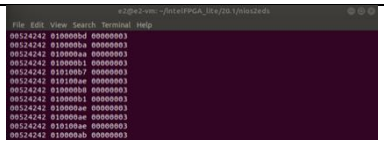
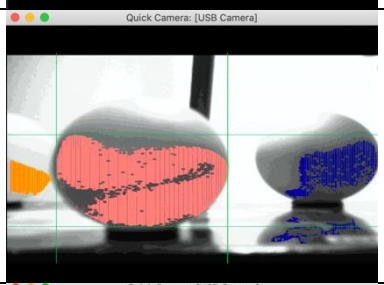
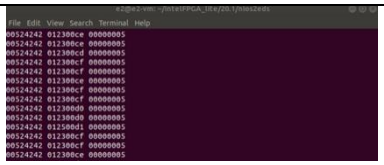
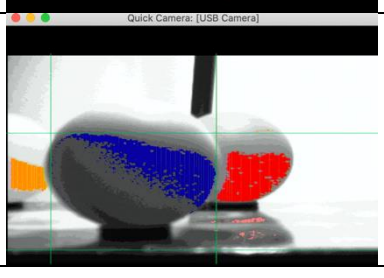
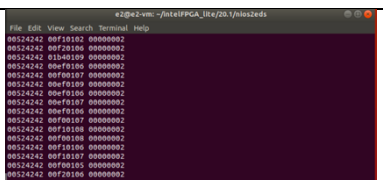
Part 2: Testing on more than 1 obstacles simultaneously

Case 1 - Green obstacle closer to the camera, Red obstacle is further away

Camera output	NIOS2 terminal
	

This test provides evidence that the bounding box correctly identifies the closest obstacle and not the one further away. Firstly, both obstacles are detected in different colours, namely green and red. Secondly, there is no overlapping between the detected green area and red area, which suggests the requirement for unique identification is satisfied. Moreover, the bounding box information shown in the NIOS2 terminal indicates a stable output in terms of width and height of the bounding box. In this example, the width is 16'h016f and the height is 16'h0146. The percentage errors here are 0.27% and 0.31% respectively, suggesting the bounding box information is substantially stable. Finally, the colour of the obstacle indicated is correct; detection of a green obstacle is expected, and the number 4 corresponds to green, as defined in the next section.

The following table provides a complete test on different combinations of colour balls and thus, testing under different scenarios is completed.

Case	Camera output	NIOS2 terminal
- Red obstacle closer to the camera - Pink is further away		
- Orange obstacle closer to the camera - Pink is further away		
- Pink obstacle closer to the camera - Blue and orange are further away		
- Blue obstacle closer to the camera - Red and orange are further away		

Connection with the control subsystem

The module outputs a row of data six times per second but will only transmit the data to the control subsystem if the control subsystem makes a query i.e., send a character via the UART. Error handling is done in the control subsystem by filtering out possible erroneous results from the vision submodule. This is because the area of the bounding box is calculated in the control submodule, rather than in the vision part. These errors normally arise in a pattern that the size of the bounding box is either too small or too large, and they should be filtered out. A few constraints have been set for removing erroneous results and they are listed below:

1) The size of the bounding box must be greater than 10,000 pixels^2.
2) The size of the bounding box must be smaller than 100,000 pixels^2.
3) The width and height of the bounding box must not differ by more than 300 pixels.

These constraints are reasonable based on several manual inspections of erroneous results and compared with correct results. The third constraint is derived from the shape of the given obstacles, since they are spheres and the image captured by the camera should indicate that the width and the height of the bounding box are roughly the same.

The distance between the rover and the obstacles can be estimated mainly based on the size of the bounding box, as long as the bounding box is valid. The distance between the rover and the obstacles is inversely proportional to the size of the bounding box. An equation can be derived based on a few reference points and the following method shows the estimation.

A curve fitting technique was used to convert the area of the bounding box for the ball detected by the Vision module into a measure of how far the ball is from the rover. For an area that is too big, or an area that is too small, an object is considered as not detected and ignored, reducing the number of false positives by about 40%. The ideal area, in terms of number of pixels, is between 10000 and 100000 pixels^2, corresponding to a square with side lengths between 100 and 316. The team also used curve fitting on a graph of the actual distance between the rover and the ball plotted against the area of the bounding box of the ball to identify the relationship between them. The team can then use this information to infer distance, and hence location of the obstacles. Here, distance is measured in centimetres and area of bounding box is measured in pixels^2.

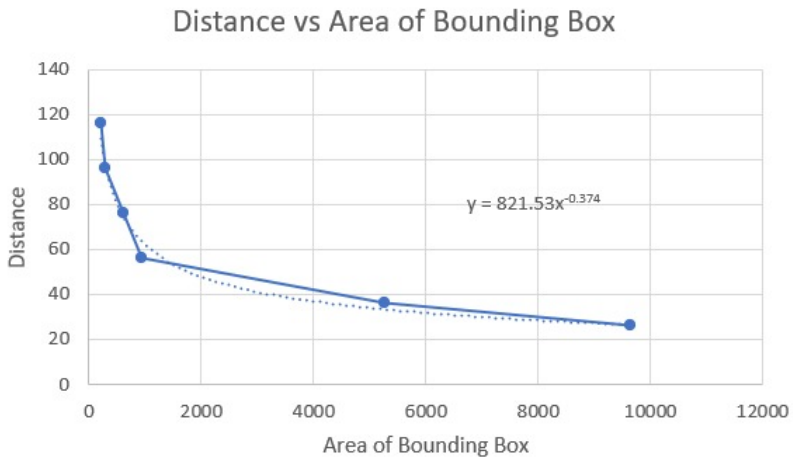


Figure 15: Conversion of Vision bounding box area to distance of object

5.3 Energy Module

Charging the batteries

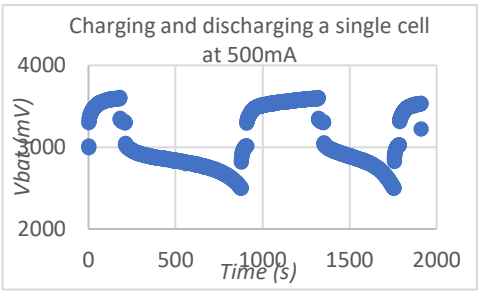


Figure 16: Single Cell charging and discharging

Constant Current Charging is charging when the current supplied by the charger is kept constant. The magnitude of the current will determine time taken to charge the batteries [5]. The higher the current, the faster the charging. However, there is a problem of overheating within the battery that can cause damage. Constant current is also proven to have a nonlinear charging and discharging curve as experimented and shown to the left (Figure 16). Constant current charging also causes a problem of capacity estimation. This is because a higher current will cause the capacity estimation to stop at a lower charge. For example, when a current of 500mA is used, the capacity is 350mAh. On the contrary, charging at 250mA gives a capacity of 500mAh for the same battery.

State	Voltage(V_Bat)	Vref	Current measured
1	3598.91	3.6	932.8
1	3598.91	3.6	883.4
1	3602.89	3.6	881.8
2	3598.91	0	850.9

Constant Voltage Charging charging allows the battery to charge until a preset voltage by full current of the charger. After reaching preset Voltage, the current will decrease to a minimum value [6], reducing the probability of overheating. However, this causes a disadvantage where it takes a long time to charge. Also, if there is a huge difference between charger and battery, there will be a huge spike in initial current produced. Also overcharging can happen and harmful effects can occur to devices charged due to grid corrosion. [5]

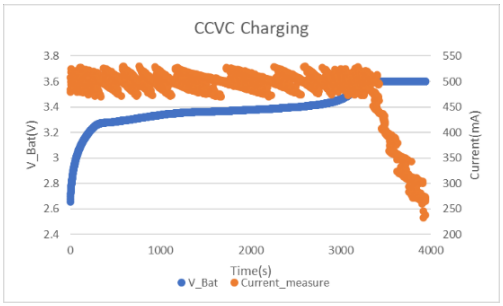


Figure 17: CCVC Charging

There are 3 phases in **Constant-Current Constant-Voltage (CCCV) Scheme.** The first phase is done by applying a small constant current ($1/10^{th}$ of the maximum current) charging method and is performed to make sure that battery is operating normally. The battery voltage's slope is then observed. The second phase involves the same constant current charging method, but it has a higher current value. This will continue up until around 70% of battery capacity. The third phase is the constant voltage phase. Constant voltage is applied across battery causing the current to decrease and stopped when current has drop to $1/10^{th}$ of full charging current. This method is widely used to avoid damage as batteries are very sensitive to voltage and current levels. [6]. However, there is a risk of overheating at the beginning of the charging process. Prolonged charging and subjecting battery to a constant voltage charging will cause constant voltage and current to gradually decrease (Figure 17). As the team is using LIFEPO4 battery, CCCV charging was chosen to prolong battery life and to maintain voltage.

Series vs Parallel

In parallel, current is divided depending on battery internal resistances and capacity is the sum of capacity of all cells. There is variation of voltage in parallel battery that can be measured using relay. Current is the same in all batteries in series, but voltage heavily vary. The capacity of cells depends on the cell with lowest capacity.

In parallel the battery cannot be assumed to have same amount of capacity, properties, and lifetime. Balancing is still needed, and each battery voltage should be considered to balance the batteries. This is so no high current can destroy and hurt the batteries in the long run. Not balancing batteries in parallel can lead to existence of voltage sources that have low internal impedance and causes looping current. High internal current that leads to wastage of energy. [7] In series, one cell can be fully charged before others and lead to faulty issues due to overcharging. It must be balanced by continuously monitoring its SOC. The capacity will be governed by cell of lowest capacity. Balancing for series takes a longer time in this experiment as cells do not naturally balance themselves.

Balancing

Cell degradation could arise from failure to balance the multi cells. With more exposure to higher voltages, the faster the cells degrade, and the capacity becomes poorer. [8] As it is important for balancing, charging, and discharging the battery efficiently, the team tested the time for three series and parallel batteries to balance themselves with fixed power supply when one battery is 2.8 V, another was 2.9 V, and another was 3 V for all of them to reach 3.1 V. Below are the results:

	Time(hrs)	
Series 3 Batteries	>1 hours	>1 hours
Parallel 3 Batteries	0.8 hours	0.77 hours

Parallel circuitry also has the advantage of having a higher capacity compared to series connected cells. In the long run, this technique will cause a problem as battery capacity decreases over time unevenly and the worst-capacity cell in the string will affect other cells as well. In terms of connection to the load, there should also be a power converter as the Port A input of the second SMPS that is used for discharging only has voltage limit of 8V approximately. Also, any fault to one cell will affect other cells in series connection. For all the reasons stated above, the team decided to use parallel circuitry in this experiment and constantly measure the current into each cell in parallel arrangement. Current was measured at Port B and potential divider formula is used to calculate current in each branch of parallel battery to avoid any current surge to battery with lower internal resistance.

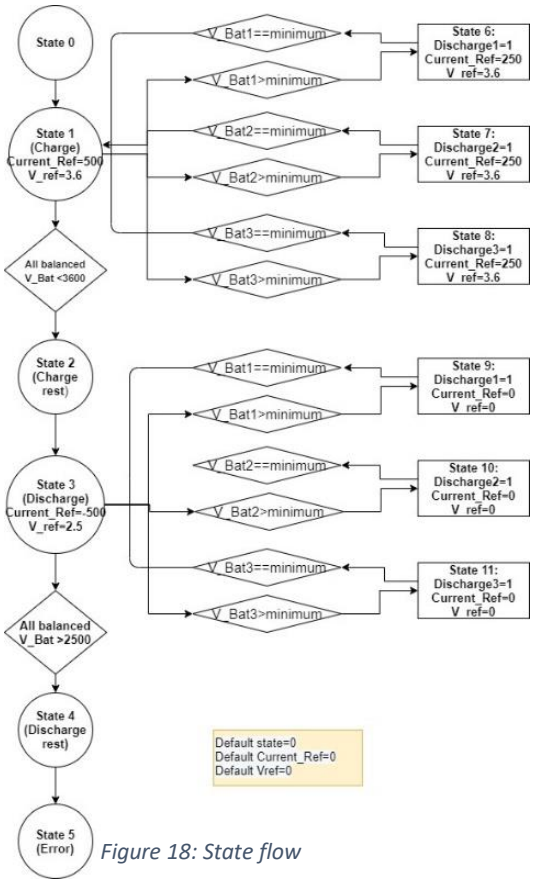


Figure 18: State flow

As voltage level increase when SOC rises, in parallel, current will flow from cells with high SOC to cells with lower SOC. The most charged cell will discharge to the other cells and active cell balancing will occur in the circuit. This method is known to be highly efficient. Nevertheless, this causes issues as it is uncontrollable. From balancing, when the cells are connected, "their terminal will jump to average value and stays there. Their OCV will approach terminal voltage asymptotically and current will approach 0". [9]

[1]To fix the issue, discharging states must be built, where batteries will decrease their individual voltage when one battery voltage is higher than the others. The problem with these states is discharging takes a long time and loses a lot of energy due to passive balancing. As balancing for parallel is faster than series, this brings an advantage. It is still slow, hence it important to combine balancing and charging continuously.

The team tried to combine 2 discharge signals into 1, however due to current limit (200mA for Arduino), this process cannot be done.

Additionally, some charging should happen during the balancing period to avoid the voltage dropping during balancing as seen in Figure 19. This is done by applying a small current reference in the balancing states. A current ref of 250mA is used to avoid voltage drop. Demonstrated in Figure 18.

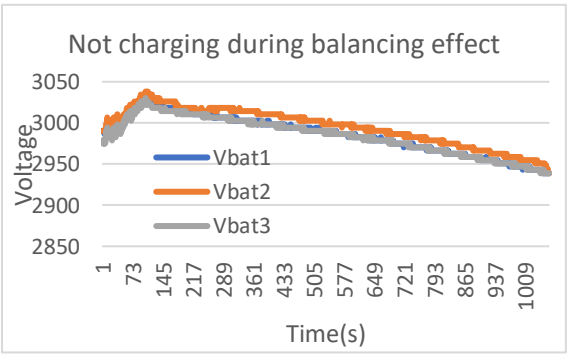


Figure 19: Balancing effect(without charging)

[1] Because of battery malfunction notice, did not manage to test the whole circuitry.

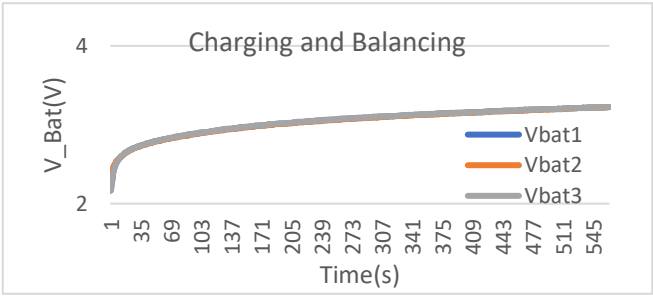


Figure 20: Charging and Balancing

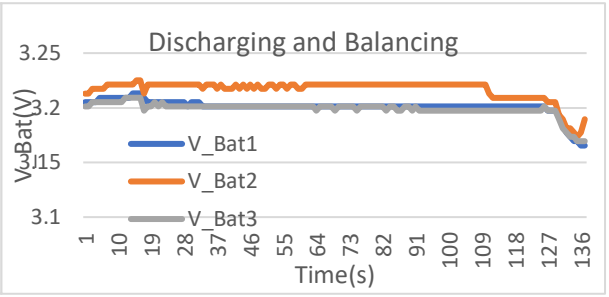


Figure 21: Discharging and Balancing

State of health

State of health is the result of accessing battery performance over time. There are numerous ways to access state of health and it is subjective depending on individual definition. The team can evaluate it by the battery capacity, the battery internal resistance and battery impedance. Capacity determines the battery ability to store energy and internal resistance and impedance reflects its power capability. The team can compare the initial reading of those components to the aging one. The capacity as example will plummet and the internal resistance increases.

Change of internal resistance	Change of capacity
$\mathrm{R} = \frac{\mathrm{OCV(SOC,T)} - V_{bat}\mathrm{(SOC,T)}}{I}$	$\frac{C_{releasable}}{C_{rated}}$x100%=SOH

As example, a LifeP04 battery internal resistance initially is 0.085 ohm based on Figure 22, however the value of resistance increases over time, Figure 23.

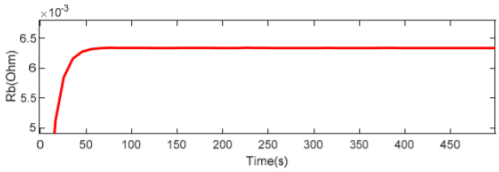


Figure 22,[35]

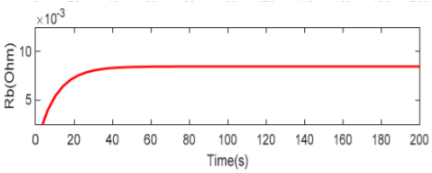


Figure 23, [35]

```
current_measure = current_measure;
Capacity +=current_measure/counter;
```

For this experiment, code to calculate the battery capacity over time during charging and discharging was created to monitor state of health by using the current measurement. To ensure calculating SOH of each battery pack was accurate, the current of each battery pack was calculated by accurately measuring the V_Bat in the fast loop and current using a current division formula $Battery\ current = \frac{Battery\ Voltage}{Total\ Battery\ Voltage} * Current\ measure * 0.94$. Even though the current agrees with the multimeter reading, there is still some precision lost, where it is off by approximately 0.02A.

State of charge

Coulomb counting

In the long run, there would be integration errors and it requires knowledge of the initial battery capacity. To use this method, the team needs to know the initial SOC. The accuracy is also dependent on the measuring instruments. [10] The battery history, temperature and cycle life also influence the accuracy of estimation [11]. In this experiment coulomb counting can be used by $SOC(t) = \frac{I(t)\Delta t}{Q0}$.In a series connection, this method is the easiest to use as the current is the same in each cell. As the team are doing a parallel circuit, it is important to make sure the cells are balanced to use this method. This is because the currents cannot be assumed to be distributed evenly due to their individual resistances. After balancing, the team proceeded to measure the current and monitor it continuously using a multimeter and can observe that current is almost similar in each branch using method

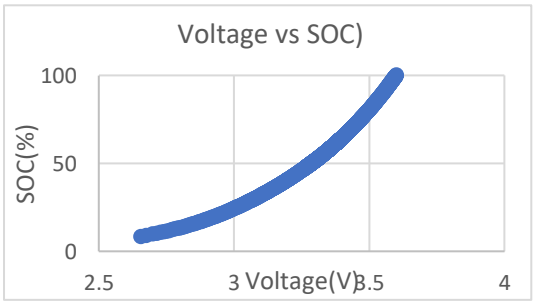


Figure 24: Voltage Look up table

outlined above. However, as they are period in between charging and discharging where cells are not balanced perfectly, current may not divide evenly making this method inaccurate and unreliable.

Voltage lookup

This method calculates the potential difference of anode and cathode of cell at steady state. Steady state means a situation where no current is flowing through the cell and sufficient resting time has been given to the cells. [11] This is noisy and requires disconnection of the battery from the whole circuitry which can be implemented by relays. As long rest periods are needed and the team need to notice the voltage during the constant current charging and discharging period, it is highly inaccurate. LiFeP04 has a flat charge and discharge curve making it hard to estimate SOC by voltage lookup alone. A polynomial equation was fit to the voltage using MATLAB, python and the received formula.

$$\mathrm{SOC(OCV)} = 183.3444145V_{Bat}^{8} - 4697.488753V_{Bat}^{7} + 52385.63044V_{Bat}^{6} - 332113.3296V_{Bat}^{5} + 1309213.756V_{Bat}^{4} - 3286181.343V_{Bat}^{3} + 5129076.929V_{Bat}^{2} - 4551441.973V_{Bat}^{1} + 1758139.254$$

Kalman Filter Method

This method is the most reliable and widely used in industries. Using current and voltage measurement, accurate SOC can be estimated. However, the computational cost is huge and cannot be implemented by Arduino alone. It comprises of 5 major steps and it is recursive.

From Figure 25, one can observe that value of SOC by coulomb counting is different than Voltage lookup. Coulomb counting might be higher because of the wrong values from the start that accumulates over time. This highlights the disadvantage of the method. As coulomb counting is riddled with uncertainty due to the parallel connection of the batteries, the team decide to use the voltage lookup to calculate the SOC but in future work, it is recognised that Kalman Filtering can be implemented using python programming and can use both Voltage lookup and Coulomb Counting to balance both methods' tradeoff.

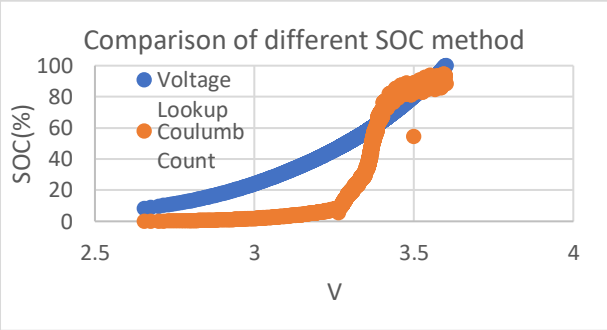


Figure 25: Comparison of SOC

Solar panels Configuration

The team observed the IV and MPPT panel characteristic curve. (Figure 26) From the graph, the team can conclude that there is existence of maxima in the power. Hence this shows that the team needs to incorporate maximum power point tracking (MPPT) in the code. As PV Panels at an unknown location where irradiance and temperature are tolerant to changes, a feedback system is constructed in the fast loop. To operate this condition, the team use a trial-and-error iteration known as perturb and observe. A small perturbation is made and observe the power output. If the power increase, the team continue in the supposed direction. If it decreases, head towards the opposite direction. Thus, the results will show that the output voltage will oscillate about the maximum power point.

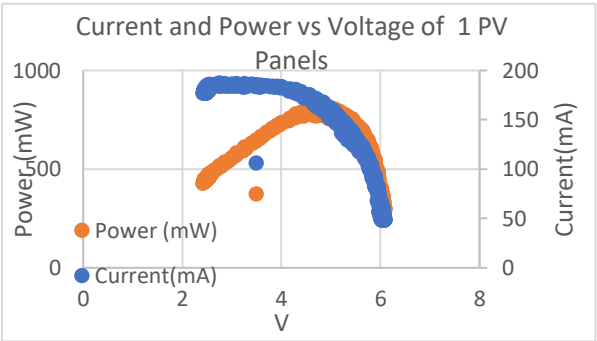


Figure 26: 1 PV Panels IV and MPP graph

To devise the algorithm, a flow chart is needed to incorporate it in Arduino. It is important to ensure there are no hidden states in fast loop to ensure the timer(`int_count`) is continually counting. As the team are also implementing a CCCV charging method, there will be 4 '`if` statements' in the fast loop. Each corresponding to situations as described in the flow chart. (Figure 27)

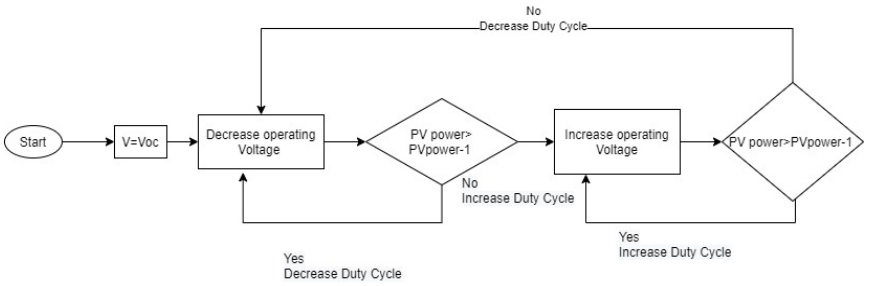


Figure 27: State flowchart

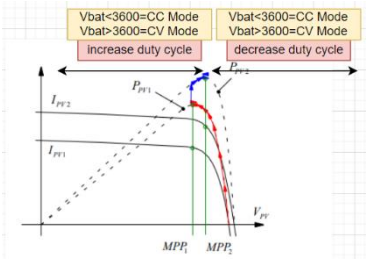


Figure 28: Expected Result

Series vs parallel solar panels

Series Arrangement has the advantage of higher voltage produced. 4 series solar panel with rating 5V can produce approximately 20V. However, partial shading effect the whole circuitry. As they are forced to have the same current, there will be a reverse biased situation involving shaded module that can lead to heat dissipation and loss energy [12]. PV cells will settle on a common current that will cause some mismatching loss.

Parallel Arrangement produces higher power than series. [13] It is also easier for MPPT tracking as only has one peak. The downside is lower voltage and higher current is produced which is not suitable for PV Panels operation. One parallel shading will affect other panel wired. This is due to current imbalance. This will reduce the voltage of the whole solar panel arrangement. voltage mismatch can occur due to the different level of shading of each solar panel. This will cause difficulty in tracking the maximum power point. [12]. PV cells will compromise on the level of common voltage due to and there will be mismatch losses.

Series-Parallel Arrangement (2 series string in 2 parallel arrangement) balances the trade-off of series and parallel circuitry [13] but have lower power yield than a parallel solar panel. [13]

Implementation problem solving (with evidence):

• **Prevent explosion and melt:** Batteries need to be monitored in terms of health and age. The team monitor the capacity level continuously to make sure the batteries have same SOH.

• **Prevent explosion and melt:** A simple diode connection or MOSFET in between batteries can be used as to correct any variation of Voltage in case a healthy battery is connected to a faulty battery. The diode will drop some voltage unlike the MOSFET. However, as the team are working with limited resources, the team made a protection circuit where whenever the current exceeds 500mA, relay will be activated for the branch to prevent any current surge to a particular parallel branch.

```
if (int_count<500&&int_count>450){V_Bat1= vbatindividual();}
else if (int_count<600&&int_count>550){   V_Bat2 = vbatindividual2();}
else if  (int_count<700&&int_count>650){(V_Bat3 = vbatindividual3());}
```

Figure 29: Relay Circuitry

• **Prevent explosion and melt:** To measure individual battery voltage to conduct a balancing mechanism, the team used relay that are turned on at different times within the fast loop. To check individual battery voltage, the team also programmed a relay. As relay consumes 50mA of 200mA of the Arduino current limit, it is important to not turn on multiple relays at the same time as it is incapable of drawing much current The relay was turned on different times as shown in the code. After checking with the multimeter, the voltage reading is accurate.

• As the team is designing a mounted design, the team can use the INA219 on the drive side to calculate the current that is discharged from the battery to power the motor, FPGA and ESP32 for coulomb counting. This is assuming that all the current comes from the battery in the completed design. The code needs to be implemented in the Drive side.

• It is also important the team uses voltage detection of input of drive SMPS to avoid the motor keep drawing current when the parallel battery pack is below 2.5 V to avoid battery damage and degradation. For extra protection, the team also need to implement if statements that say if the input Voltage from battery Is less than 2.5V, discharging will stop from the battery and duty cycle of Drive SMPS will be 0. Fortunately, as discharging is very slow, this is unlikely to happen.

•One thing to note is the as the team are connecting the PV Panels to Port A in buck condition, there is no clear way to calculate current of PV panels. Hence, the team decided on 2 ways, either using Current Shunt resistor or taking Pout/Vout and factoring some SMPS losses (approximately 0.038W). Current shunt has a

problem whereas the current is too small, it is not properly read by the Arduino as example at some instance the current shunt reads current as 0, hence the team decided to use the second method.
• As Port A has a voltage limit, hence we need to use a voltage divider at Port A to measure the voltage. The Vpd also has a very low resistance and does not work well. Hence, we used the potential divider provided to make a potential divider and calculate voltage at Port A.

Final Implementation

For the battery's maintenance, the team calculated the SOC by voltage lookup and coulomb counting even though the former is more preferred. Capacity and maintain balancing determine SOH during charging and discharging. The team also CCCV to maintain the voltage of the battery and not lead to overcharging the battery to avoid any explosion.

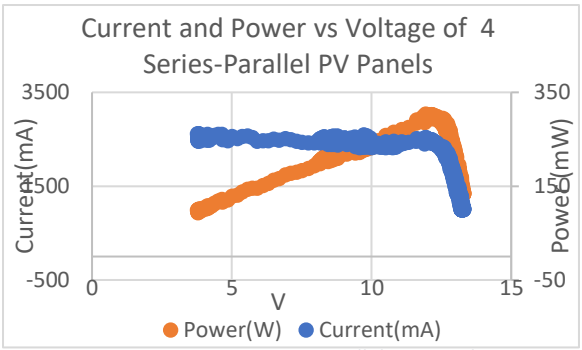


Figure 30: 4 Series-Parallel PV Panels

As the solar panel is said to be dusty in winter on Mars [14], it important to generate high enough voltage for the circuit to function if it operates in buck mode. To capture the most sunlight and operate regardless of condition, the team use a parallel-series configuration. As even at lowest irradiance (mostly cloudy) the voltage will be 4.14 V, so it ensures that the cell will be charging due to the input voltage is more than output voltage and some current is flowing. With limited hardware, the team decided on the use of 3 batteries to supply the Mars Rover which according to calculation is sufficient to supply the Mars Rover with enough power to Buck configuration and parallel battery configuration is also used. Buck mode is more efficient than Boost and does not contribute to any voltage limitation problem posed by the SMPS given that the input is controlled. Even though parallel configuration has the disadvantage of a lower voltage, there is a benefit of higher capacity combined.

The benefit of parallel battery arrangement is that the voltage will not be too high(around 3.6 V) for motor, FPGA and ESP32. Also, as the discharging curve is flat and the capacity of parallel battery is high, the time takes to discharge to 1/3 of capacity to avoid overheating and not cause damage to battery will be 2.37 hours compared to series battery connection that will take 0.79 hours to discharge with a current consumption of 632mA (Reported Current Consumption from motor, ESP32 and FPGA=0.122+0.260+0.250). to discharge to 80% DOD. Also, as the minimum operating voltage for motor and ESP32 is 2 V and 2.3 V respectively, and the FPGA MAX 10 has lowest voltage consumption of 2.5V, [15] the rover can work up until the minimum voltage input supplied by the battery. This means that assuming the battery starts at fully charge at night on Mars, it can continue operating for 2.37 hours during Mars night-time. This makes parallel battery capable of supplying the require power for its continued operation better than series.

The series-parallel arrangement for PV panels and parallel battery gives the edge of the margin between the input voltage and output voltage to be large and allows the MPPT to work effectively. For example, in the datasheet, as irradiance changes, the duty cycle(pwm_out) can be seen to vary continuously to operate at the most efficient voltage. Undeniably, even though series-parallel arrangement is used, the team could have done the design with series battery configuration. However, the team tested that configuration as well, and a problem arise when the voltage combined at the output is very similar to the input voltage. This causes the duty cycle to be saturated to maintain it in the buck mode. As an example, when the input voltage is approximately 9.8 and the PV Panel is 10V, the duty cycle is saturated at 0.99 V, and this causes a problem where this will cause inefficient MPPT algorithm collection for high battery voltage value.

There is an option of making both PV panels and battery as parallel. The problem arises when the margin between input voltage and output voltage becoming too close. As the team are aiming to send it to Mars which has a very dusty and low lighting environment, it is important to make sure that the voltage margin of the PV panels and the battery pack be far enough so that it can continue operating even at low light condition. Also, parallel circuitry of PV panel is not ideal as stated above.

Structural (Physical) design of the Energy subsystem

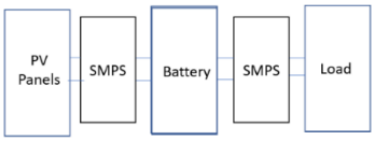


Figure 31: Mounted charging station

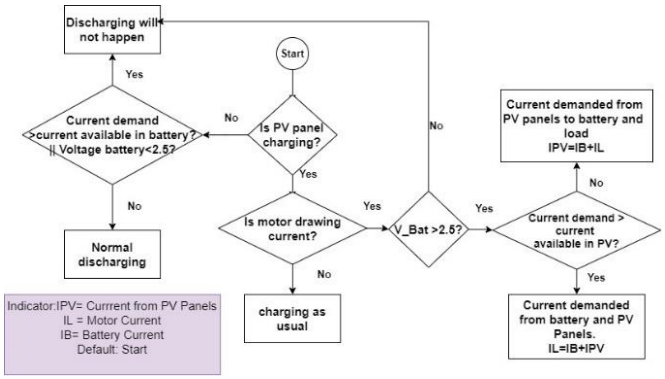


Figure 32: Charging and Discharging Flow

Static Charging Station is easier to implement and does not require the algorithm of charging and discharging simultaneously. However, it takes more power and energy to make the rover to have a charging station separate from its build in case the rover needs to move continuously. Although having a charging station on the Mars Rover may take up power due to its weight, it is still applicable as the rover is needed to move from one place to another. Having one charging station, it will limit the functionality of the rover itself. As discharging and charging needs to happen at the same time, the team need to build a code to accommodate this. The team also built an integration plan to integrate the rover to the energy module, shown in the next section. For charging and discharging at the same time, the team used two SMPS treating it as a parallel circuitry of PV panels, battery, and load (Figure 31). The whole operation is described in Figure 32.

Rover range estimation

$$Power\ consumed\ by\ rover\ while\ static$$

=Power consumed by motor+ Power consumed by ESP32+ Power consumed by FPGA

$$= 3.3 * 0.122 + 3.3 * 260m + 3.3 * 250m = 1.3431$$

$$3\ 3.2\ volt\ battery\ can\ produce\ 3.2volt, 2Ah$$

$$System\ Energy\ Stored = 4.8\ Wh$$

$$\frac{System\ Energy\ Stored}{Power\ consumed\ by\ rover} = \frac{4.8}{1.3431} = 3.574$$

$$Range = Average\ speed\ of\ rover\ per\ second * Time = 6e-02 * 60 * 60 * 3.574$$

$$= 771m\ in\ total$$

From drive's point of view, the average speed is 6.6 cm/s and from integration's viewpoint the average is 6.2 cm/s. This is maybe due to the weight of the ESP32 and FPGA. The team can approximate that average speed to be 6.0 cm/s when the rover is complete. This is not far off with the actual Mars' Rover which is travelling at 4.4 cm/s due to the rough terrain of the Mars itself. [16] Also, as the team are measuring current consumed when the rover is not moving the current estimated is lower than actual power consumed during move. Hence the actual distance might be 771m in total. As we are projecting Mars Rover to stay there for about 100 days (based on real Mars Rover), and LiFePO4 can survive up until 2000 cycles, we can assume that the Mars Rover can fulfill the requirement. The group can detect voltage and current on Drive's side to estimate the total distance rover can still travel with power available and relate that to duty cycle. This information can be displayed on the dashboard. The formula is the same as shown above.

Intermodule Communications

Integration with the whole rover is carried out using an UART connection. As there are 3 UARTs on the ESP32 and UART0 and UART2 are used to connect the vision and drive subsystems respectively, the energy subsystem will use UART1 to connect. Theoretically, the energy subsystem will relay relevant information such as the voltage and current of the battery, the light intensity on Mars and the battery charge status. This information is subsequently relayed back to the web app through the ESP32 so that they can be displayed on the dashboard. The controls are also tasked with relaying information of whether the rover needs charging based on detection of voltage and current at the output. The information will be sent back to the energy module and discharging and balancing of batteries will happen accordingly based on the energy code. Theoretical code snippets are added on both the control and energy subsystem's Arduino code. As the team must communicate with Drive as well, the team must implement a discharge and circuit for the output side of the drive SMPS.

5.4 Command Module

Research

There are several resources available to create a full stack webapp. Frontend is typically designed with html and CSS, with functionality added by JavaScript. The backend will be calculation heavy and make use of APIs to communicate with a database, typically stored in the cloud. A webapp was chosen for its ease of use and cross device compatibility. Furthermore, it allows communication with the rover at any place, and any time. Using an external API would be suited for this. As it is a software-based part of the project, it has very few restrictions, and has the potential to execute a large range of features implemented in hardware.

Structural Design and Implementations

Development of such a system can be split into three main parts, A) frontend, B) backend, C) storage and D) communication with control (Figure 33); the MERN stack was chosen to facilitate this.

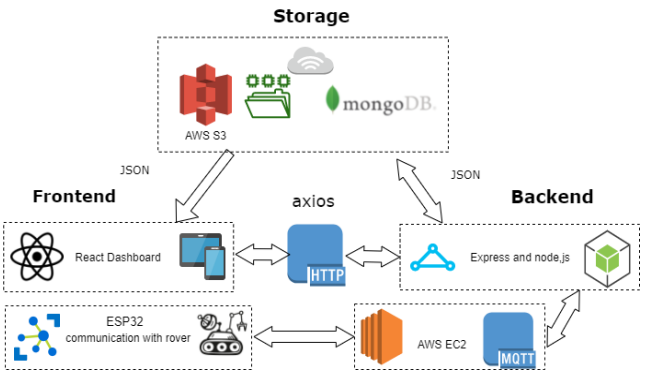


Figure 33:Web App Architecture

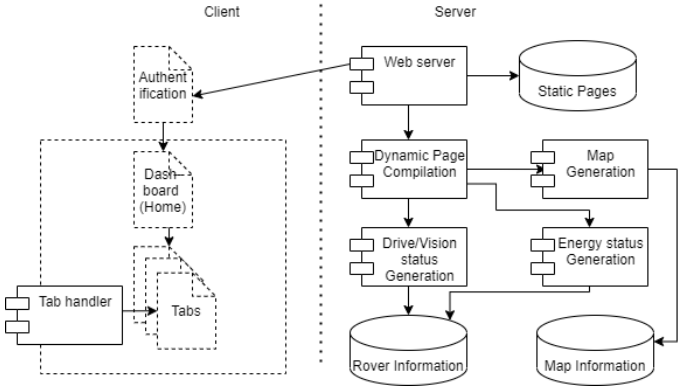


Figure 34: High level Implementation design [17]

A) Frontend

React was chosen to develop the frontend. It was chosen due to being a library that allows compartmentalising parts of a webpage, so components can be reused and embedded within each other. React can also store functionality as states so the library takes care of updating the UI without the developer needing to manage it, particularly useful for people with little experience of web development. React also handles the web and mobile layer, allowing different accesses. A combination of react components [17] as objects and functions were used. Functional components were used for pages due to being easier to understand, leading to more robust code. These used `useEffect()` hooks to keep track of status values. However, when creating more complex objects (e.g. maps), classes were used to retain mounting status and fetch data asynchronously.

```
export default function Dashboard(props) {
    var classes = useStyles();
    var theme = useTheme();
    //var status = [];
    var current = [];
    var voltage = [];
    var dist;
    var battery_status = "good";
    // local
    var [mainChartState, setMainChartState] = useState([]);
    const [status, setStatus] = useState([]);
    useEffect(() => {
        const fetchData = async () => {
            const result = await axios(
                endpturl + 'rover/status');
            setStatus(result.data);
        };
        fetchData();
```

```
export default class MapPlot extends React.Component {
    _isMounted = false;
    state = {
        selectedPointId: null,
        showVoronoi: true,
        obstacles: [],
        rover: []
    };

    componentDidMount() {
        this._isMounted = true;
        axios
            .get(endpturl + 'obstacles')
            .then(res => {
                this.setState({
                    obstacles: res.data
                })
            })
            .catch(err => {
                console.log('Error finding obstacles');
            })
```

Figure 35: Dashboard function and Mapplot object Implementation

The main library chosen was material-UI. It is a set of components built based on Google design guidelines. This means there is a consistent design, and the components can integrate within each other. The components are well documented and there are many example codes online**.** This makes it easier to add on to the website in future iterations, and other webapp developers can add further functionalities with ease.

React-vis [18], a library created by Uber, was chosen to display maps for its good documentation of customisation and adding other functions. Furthermore, the library makes use of Voronoi plots when calculating display states; this is useful for path finding algorithms and helps the user to chart their own path. [19] The default styles and the fast-loading duration(13.3ms) for the library is suited to real time displays. For simpler graphs, recharts was chosen, as it integrated well with material UI and is well maintained. [20]

A dashboard should be visually aesthetic and simple to navigate. As a result of this, mock ups for the design were created in conjunction with usability testing within the group. Webapp sections were created and classes for general layouts were used, to reduce workloads as components could be reused. Given that people who access the dashboard will have full access to rover control, there needs to be authorisation of users. For security purposes, a foundation for a log in system was included, using `sessionStorage` tokens. Once logged in, all pages can be accessed via a menu bar at the side, and a header contains notifications and a search bar. The landing page contains general real time information from energy and drive, however due to the growing size of the map(as more area is covered) it is shown on a separate page, and allows for zooming in.

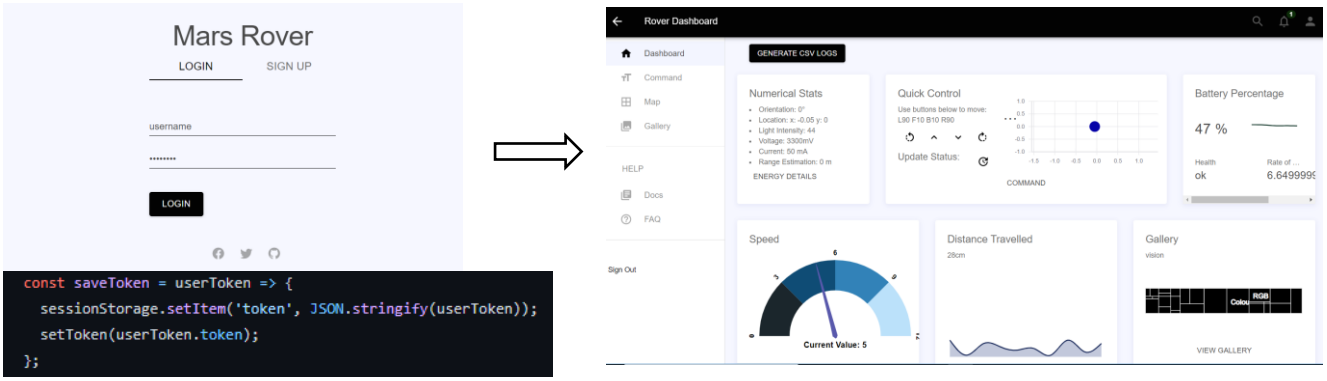


Figure 36: Log in page and landing page, with session storage code

It is important to choose the most useful information from each system to be broadcast due to the rover being on Mars as the time taken for information to travel will be slow. This makes it unfeasible to use a video feed, so an alternative option considered was to send pictures of the obstacles, found in the gallery. Another implication is that updating all data all the time will cause insufficient network resources errors, so the choice was made to take updates at specified intervals with the corresponding object and rover coordinates.

B) Backend

The backend communicates with the frontend using HTTP protocols. The team chose to use Node JS as it matches the language between front and backend and has good performance for real time applications. Express is built on node and manages routes for APIs [21]. *Axios* and *Fetch* are the main libraries available. *Axios* has extensive browser support, whereas *fetch* only works for select browser versions. Therefore, to widen accessibility *Axios* is a better choice. Furthermore, *Axios* automatically converts the messages sent and received to JSON, making parsing information simpler. Requests made also have catch statements and debug outputs to the console; this aids in testing without requiring direct viewing of the database.

Within the backend, the program must also communicate with the ESP32, using a different protocol. Using MQTT, the control submodule can directly send information to the backend, which uses POST requests to update the database. Updating the database then causes some additional overhead before data is available for the webapp to display using a GET request; however, it is certain that the original data sent has arrived.

Figure 37: Typical data flow implementation via MQTT and HTTP

C) Storage

As data should be retained over a long time and accessible, a database should be used. MongoDB is a popular choice as it can be used everywhere and is open source. It is scalable and there are few restrictions on storage structure due to using NoSQL. NoSQL also has high capacities, which is useful in the context of the rover to be able to store details of the area explored, and can be used as an information dump, from which command can synthesize required information to display.

As images will also be stored, a private S3 bucket [22] in AWS was set up. The URLs are stored in mongoDB and details to access these are put into a .env file. Including credentials and security using existing AWS features increases the security of the whole stack, and any sensitive data could be protected. Each item added is automatically allocated an id, which can be used for modification requests such as PUT and DELETE.

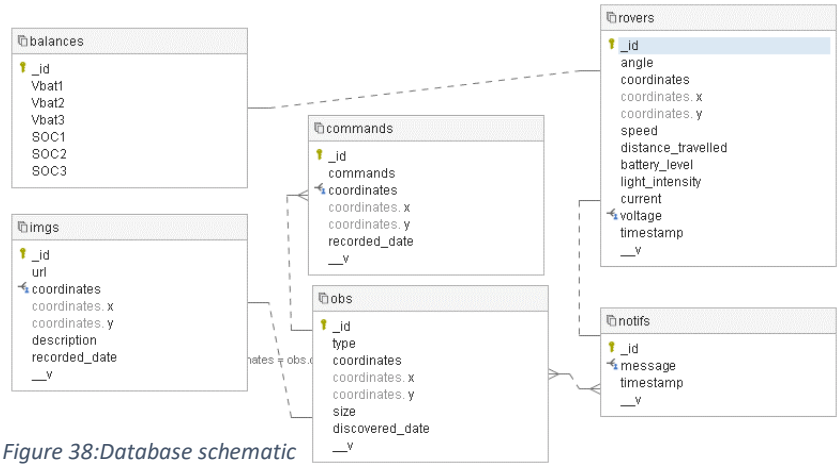


Figure 38:Database schematic

D) Intermodule Communications

MQTT was chosen to communicate with the ESP32. MQTT is a lightweight protocol that can be implemented on constrained hardware such as the Arduino, and it has good quality of service for high latency networks. [23]. A json schema was implemented with a one-to-one mapping between properties of the same name, and instruction formats were predetermined. Repeated fields within HTTP and MQTT requests were kept in the same setup, with the same dependencies. All subsystems have the potential to communicate directly on MQTT channels; given the constraints of the project, the energy subsystem can also use MQTT to communicate.

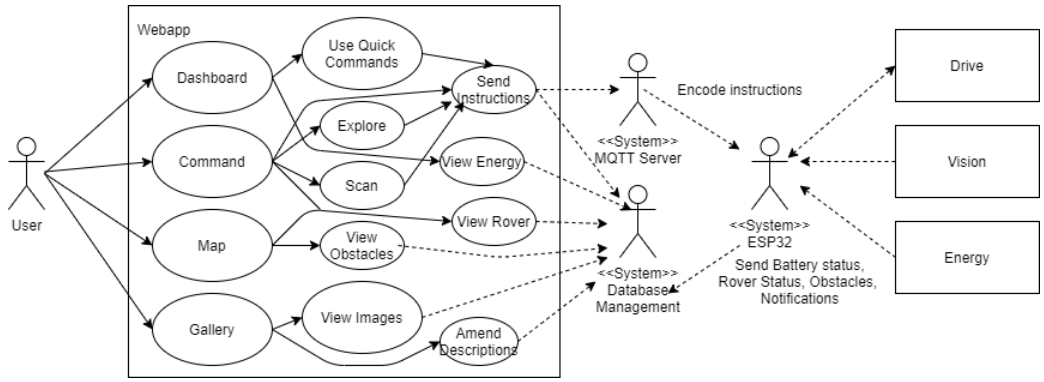


Figure 39: Use Case Diagram

Commands and Pathfinding Algorithms

Design

The simplest commands are found on the homepage, where one button corresponds to one command. This is expanded in the command page, where the user is able to type a string of commands, go to a specific coordinate, explore a five-metre box or choose to scan the area (in a circle). Note that sending commands one at a time will not reroute paths; only when there is a string of commands will it be broken up and rerouted.

Algorithms such as Djikstra, A* can be used to construct a path if a visibility graph can be formed. In the case of the rover, the paths available can be directly mapped to a graph, which means the Voronoi would not be required. Extraction of Voronoi points and attempting to display fetched data results in substantial slowing of the webapp, due to how react-vis is rendered and the requirement to fetch the latest data.

Implementation

Consequently, the A* algorithm was employed, and construct a graph using a separate function [24]. Firstly, a visibility graph was constructed from obstacles (stored in the database). The rover itself has dimensions of around 20cm, accounted for as an avoidance region. From this, a radius around the obstacle is calculated, and a regular polygon is constructed. The coordinates of such a polygon are found from the following equation:

$$\begin{cases} x = \text{obstacle.x} + \text{Rcos(t)} \\ y = \text{obstacle.y} + \text{Rsin(t)} \end{cases} \quad with\ t \in [0, \frac{2\pi}{n}, 2\frac{2\pi}{n}, \ldots, (n-1)\frac{2\pi}{n}]$$

A line is drawn between any two vertices that can be joined without obstacles. A function iterates through all vertices to create a bidirectional path of coordinates. See the Appendix for an in-depth example.

The coordinates are turned into a list of commands containing rotation and forward movements. Bearings are used to calculate the correct angle; after each command, the rover is reoriented to face north. Reorientation allows the location and angle to be tracked more accurately. As the accuracy of turning and moving was increased, the polygon dimensions could be increased. The more sides the polygon has, the better the approximation and the smoother the path; however, there is a limitation on precision due to the Drive module. The commands are encoded according to the predefined structure mentioned in Drive and Control.

The option “Exploration” is also autonomous, aiming to traverse a square area; as soon as an obstacle is detected, a new path is generated to go around it and the existing instructions are allowed to continue execution, albeit with an offset. This obstacle avoidance approach allows a string of commands to complete rather than stopping the rover and idling. In this way, a map of balls in any path traversed will be shown.

```
if (!scan) {
    var speed1 = cmds_arr.substr(0, 1);
    var encoded_turn = encode(speed1 + " R045 R045 F030 L045 L045 ");
    var encoded_back = encode(speed1 + " B030 "); //reverse into position assuming always return to 0
    var arr = encoded_turn + cmds_arr;//+ encoded_back; //not round(but needs to remove)

    client.publish('rover/cmds', JSON.stringify({commands: arr}));
    console.log("encoded arr", arr)
```

```
//encode each command and pad to a 3 digit number
if (isNaN(x) && x !== "S") {
    encoded += speed + x.substring(0, 1).toLowerCase() + pad(x.substring(1), 3) + " ";
} else if (x === "S") {
    encoded += "0s000 ";
}
```

Figure 40: Rerouting and command encoding

Deployment

The backend was deployed on Heroku, and the frontend was deployed on Netlify. As Heroku and Netlify only allows one deployment at a time, the files were split into separate GitHub repositories for hosting purposes. The broker was deployed on AWS, using a private link with hiveMQ. An EC2 instance was used to run a MQTT client to bridge communications between control and command. In practice, it is likely that a private network would be used and so there would be less traffic; this would reduce the latency and increase security.

System Optimisations

The webapp can conduct a lot of processing with little cost and simple debugging. The trade off in this case is one of latency, whereby the computed data must make a round trip, between webapp to control to drive/vision/energy, respectively. Using a lightweight protocol (MQTT) ensures there is little overhead and encourages most of the processing to be carried out on the subscribed clients.

Testing

To enable portability, a docker container was set up, containing all the necessary dependencies on a local environment. This allowed other team members to test the code with the correct image. Management of the webapp could be done via `yarn` or `npm`; `npm` was chosen as the simplest option as it comes with `node` [25] and makes efficient use of the network as it has improved network and cache usage. [26]

Functional testing occurred locally. To avoid having to compile the whole site multiple times, jsfiddle was employed where javascript functions such as user input validation could be tested individually(see Appendix). The function of buttons and user interactions was important for initial tests. Postman was used to check the functionality of API calls and later monitor network statistics. The main priority was to establish a reliable connection; latency and information formats were considered afterwards. Compatibility testing was also conducted; between the team different browsers and operating systems were checked to ensure the display was still reasonable. Prototyping allowed webapp to be flexible and testing individual components(such as `Mapplot.js`) was simple with an agile compartmentalized workflow.

Once the webapp was connected, the time taken for a command to be received was found to be browser dependent. This is likely due to different responses to cross origin requests. When using direct requests via postman, the time take is extremely fast. However, once deployed, the speed of responses from the backend server is substantially slower compared to localhost and slowed down over the course of development due to using free deployments. To reduce the lag, both the database and host were set to the same region. A cache was enabled and mounting status flags were added, improving speeds by almost half, stabilising average time.

Request method		Button command round trip time/s			String command round trip time/s			Coordinate command round trip time/s		
Baseline	Postman	0.222	0.364	0.310	0.399	0.484	0.321	0.352	0.398	0.459
No Flags or Cache, Hosted in EU and US	Chrome(web)	22.14	21.87	22.54	21.04	24.90	21.33	18.30	21.52	22.78
	Chrome(mobile)	0.686	0.904	0.996	0.968	1.946	1.832	1.945	2.834	3.125
	Microsoft edge	3.001	2.783	2.962	4.214	2.902	2.634	2.937	4.241	3.082
Flags and Cache. Both hosted in US	Chrome(web)	13.22	5.039	6.227	11.03	6.463	9.627	7.671	5.584	6.492
	Chrome(mobile)	0.344	0.683	0.504	1.212	1.853	1.346	1.431	1.602	1.832
	Microsoft edge	1.294	1.334	1.302	2.494	2.303	1.990	1.842	1.532	1.788

Figure 41: Results from command and processing trip times

Note: values found from adding on times between the command getting from the website to the MQTT server to the time for the rover to move.

Final implementation

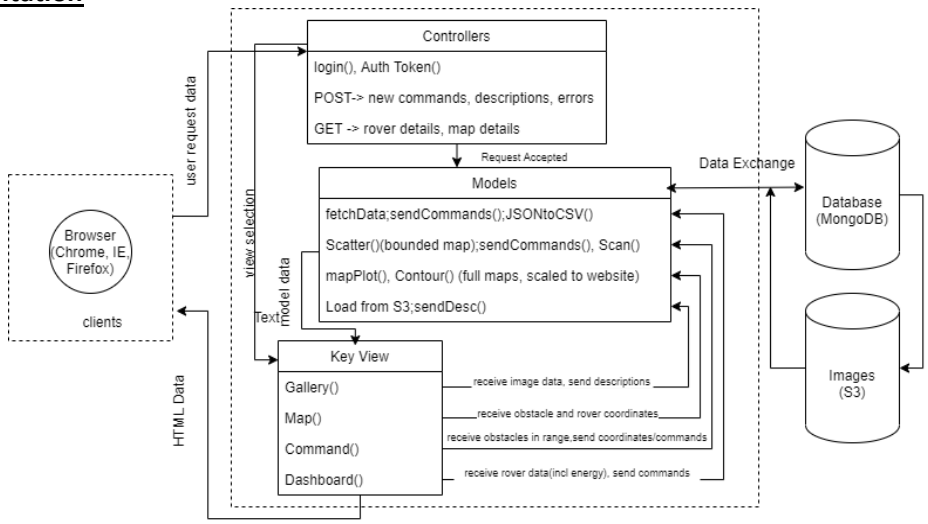


Figure 42:MCV diagram [18]

Functional Design and Implementation

Dashboard -the dashboard contains a `JSONtoCSV()` function to synthesize data in a usable format. Views of items in the dashboard are encapsulated as `Widget` objects, where they can resize according to the screen and are well suited for placement on a grid. `fetchData()` is used continuously to refresh statuses.

```
<div className={classes.widgetWrapper} style={style && {...style}}>
  <Paper className={classes.paper} classes={{ root: classnames(classes.widgetRoot, {
    [classes.noWidgetShadow]: noWidgetShadow
    }) }}>
    <div className={classnames(classes.widgetHeader, {
      [classes.noPadding]: noHeaderPadding,
      [headerClass]: headerClass
```

```
<Grid item lg={3} md={4} sm={6} xs={12}>
    <Widget
        title="Battery Percentage"
        upperTitle
        bodyClass={classes.fullHeightBody}
        className={classes.card}
    >
```

Figure 43:Widget class

Commands -The user has a lot of freedom of choice when deciding how to control the rover. On the main dashboard, there are simple commands to move the rover, which will not cause rerouting. On the command page there are button options to explore a 5x5m space or do a circular scan of the area. More complex features include rerouting and sending a string of commands, or go to a specific coordinate, considering existing obstacles and calculating the best route, using graphs. These are stored in the database. Note that a `RegExp()` function checks for valid inputs using regex, and turns unnumbered commands into 90° turns and travelling forwards/backwards by 10cm. An alert() is sent if it does not match.

```
let re = /([FBRL][0-9]*|S|X|[0-9])$/;
if (re.test(res)) {
    if (res === "L" || res === "R") {
        return res + "90";
    } else if (res === "F" || res === "B") {
        return res + "10";
    }
    return res;
```

Figure 44: Regex expression and overwriting single letters for commands

Maps-Maps are updated in real time as obstacles are observed. There are 2 main maps in place, one to show aggregation of obstacles and one to show obstacles observed in relation to the current position. There is also a map on the commands page, which only displays values within prespecified bounds to reduce load lag. Additionally, the rover position is redisplayed when the status updates, during and after instruction execution.

```
const {crosshairValues, selectedPointId, showVoronoi, lastDrawLocation} = this.state;
const obs = this.state.obstacles
Rover = this.state.rover.map((d, id) => ({...d, id}));
```

Figure 45: Maintaining obstacle and rover states

Notifications- Events of interest(obstacles, battery errors), result in notifications from control. Different messages have different icons. Command will alert the user with messages on the header which work as a FIFO queue. Once seen they are removed from the notifications bar and the database. There is a delay between obtaining data displaying the messages, likely due to constant refresh of the header, resulting in batch notifications when under stress.

```
for (var i = 0; i < notificationsm.length; i++) {
    var notif = notificationsm[i];
    if (notif.message === "obstacle") {
        notifications.push({
            id: i,
            color: "warning",
            type: "report",
            message: "Stopped due to obstacle! Rerouted commands."
        })
    } else if (notif.message !== "success") {
```

```
function clearNotifs() {
    const result = axios.post(
        endpturl + 'notifs/clear');
    console.log(result)

}
```

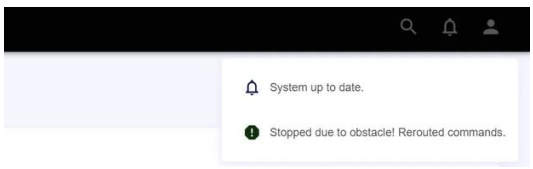


Figure 46: Excerpt from Notification parsing

Gallery-The final decision was to create a gallery linked to a private S3 bucket, pre-set with security configurations. When images are received via MQTT, they are converted from a byte stream and downloaded to the host computer to be uploaded to the bucket. A unique name is generated by UUID and stored in a database with other information from the MQTT request. Users can also modify descriptions via a textbox.

Evaluation of problems and solutions

Unresolved Promises – To get data from the backend, HTTP requests are made with promises. As they are asynchronous, the information may not be returned before it is required. This resulted in no data to parse, causing garbage data outputs. This was a particular issue when making MQTT calls, as the data had to be pre-processed. To counteract this, `.then` and `.finally` were used, and the POST requests were embedded within those functions. This was highly successful, and optimised speed of webapp instantiation.

Overwriting Obstacles – When overwriting obstacles, the previous obstacles need to be deleted. To get all these obstacles without overwriting the new one required waiting for objects to be deleted first, then to add the new object. This was done with a search API, embedded within another axios call. This is suitable for the case where one object is only one colour; however, if, for example, the ball was red on one side and green on the other, it would be overwritten numerous times, when in fact it should be the same obstacle.

Real time operations – The display needed to refresh constantly, to display the rover location. The system lagged when first loaded and would not update frequently. Functions such as `componentdidUpdate` are available but they blocked other concurrent requests. Using `useEffect()` hooks helped to give consistent updates however sometimes components would update before being mounted, causing stalls. A flag for component mounting had to be set, allowing the components to update once they had been rendered. This is not the most efficient usage of network resources, however it does work for the smaller use cases.

5.5 Control Module

Research

By design, the Control module will be physically connected to the Drive, Vision, and Energy modules via wired connections. The Control module is then connected to the Command module using a wireless connection, which, in this project, will be a Wi-Fi connection to the Internet.

The physical wired connection communication protocol which was used in this project is the Universal Asynchronous Receiver-Transmitter, or UART. The team chose UART due to its asynchronous nature, which means the team do not have to agree on the clock speed of the different microcontrollers of the different modules beforehand. Furthermore, as the rover's architecture is not a master/slave architecture, the team do not consider the use of I^2C or SPI, as those protocols have an unnecessary overhead for slave select [27]. Each of the Drive, Vision, and Energy modules just need to agree on one accepted baud rate, and things should work fine.

The team also chose the MQTT protocol over the HTTP protocol for communicating with the Command module over the internet, after referring to [28] [29]. The HTTP protocol has a heavy overhead to set up the communication, and once the data is sent/received, the connection is closed. There is no reason to use such a heavy protocol requiring such a high bandwidth in a Mars rover application, where bandwidth is expected to be limited. Furthermore, the HTTP protocol is a client-server protocol, with the rover Control being the client and the Command module being the server. Hence, to achieve real-time or near real-time performance, the Control module would need to continuously poll the server, which is a stupid design that is very power inefficient. By contrast, the MQTT protocol is event-driven, as is the case with a Mars rover: it runs fine on its own, until it encounters an object (event) in which it will stop and request further instructions from Command.

Design methodology and considerations

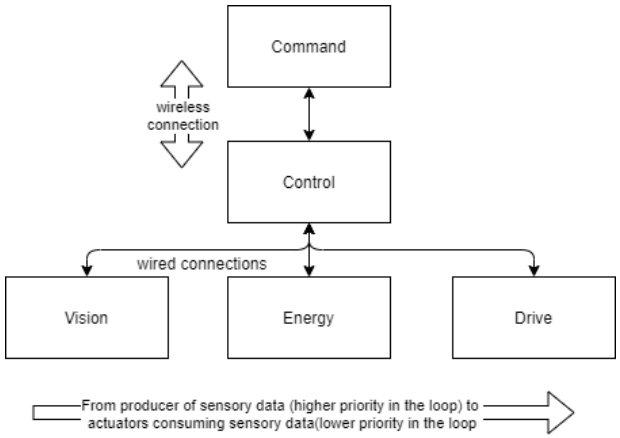


Figure 47: High level structure

The figure above is a high-level structural description of how the modules is planned to be organized, from the Control's point of view. The Command will have the absolute control over how the Control behaves, and Control must be capable of responding to any instructions by command as fast as possible, via interrupts.

After receiving an instruction list from Command, the Control module will then interface with the modules listed in the above figure from left to right. Vision should have the highest priority, as the rover need to ensure that it does not crash into any object. Next comes the Energy module, where the rover needs to ensure that it has enough battery left to return to base before executing any movement that could possibly leave it with a dead battery with no way of returning to base. Finally, the Drive module is the last module the Control will interface with, to carry out the necessary instructions from Command.

1. The Control must be capable of being interrupted by the Command at any time, in which Command will send an instruction string to be executed by the rover. Therefore, interrupts by the Command have the highest priority, and the Control can be interrupted at any time by the Command.

2. An instruction string has many atomic instructions, where an atomic instruction is defined as a fundamental movement instruction and the corresponding magnitude/angle such as move forward, move backward, rotate left, and rotate right. They are single-space delimited, to make the overall coding effort more manageable. The atomic instructions are executed in the order in which they are received, which is from left to right.
3. The Control publishes the current status of the rover to Command via MQTT once each atomic instruction has been successfully executed. This form of regular update allows Command to have a good estimate of the position of the rover after each atomic instruction.
4. The Control module only considers an atomic instruction as successfully executed upon receipt of an explicit completion signal from the Drive module.
5. The Control module polls the Vision module once every second and waits for the reply. Depending on the reply, Control will then control the Drive module, or poll it for relevant information to be sent back to Command. Information from the Energy module is also obtained by polling. In this manner, the Control module only needs to handle interrupts from the Command module and does not need to process interrupts from other modules.
    a. Furthermore, it allows for the Control module to be at liberty to perform scheduling on its own without having to consider more than 1 interrupt (from the Command module), and the 1 second interval provides a buffer for the vision module to ensure that the object is "truly big enough" within the camera frame instead of hovering at the boundary of the threshold for a positive detection.
    b. This design suits hierarchical nature of a command-and-control system, with "universal override" such as the Command module having the highest priority, and the producers of data, i.e. the modules which take in input from sensors will be run first in the iteration of each loop, and then modules which are supposed to respond to those input are then run next.
6. The Control module will stop the rover immediately when an object is detected, and all instructions that have not been completed, together with the rover's status, are reported back to the Command.
7. Suppose that the rover stopped because it detected an object, and to ensure that the rover does not immediately stop again after receiving a fresh instruction string from Command, the rover will ignore everything from the Vision module for 5 seconds after receiving a new instruction string.
8. The implementation of the communication protocol with Drive, Vision and Energy is deliberately designed in such a way that a second request/instruction will not be sent to the modules before the first instruction has been completed and an acknowledgement has been received from the module. This is to ensure that if a module which does not support interrupts are somehow to be connected to the Mars rover in the future, the team can maintain the same program flow and program structure.
9. The Energy module can be connected to the Control module if required through the UART pins 0 and 1. The design has already accommodated that.

Design

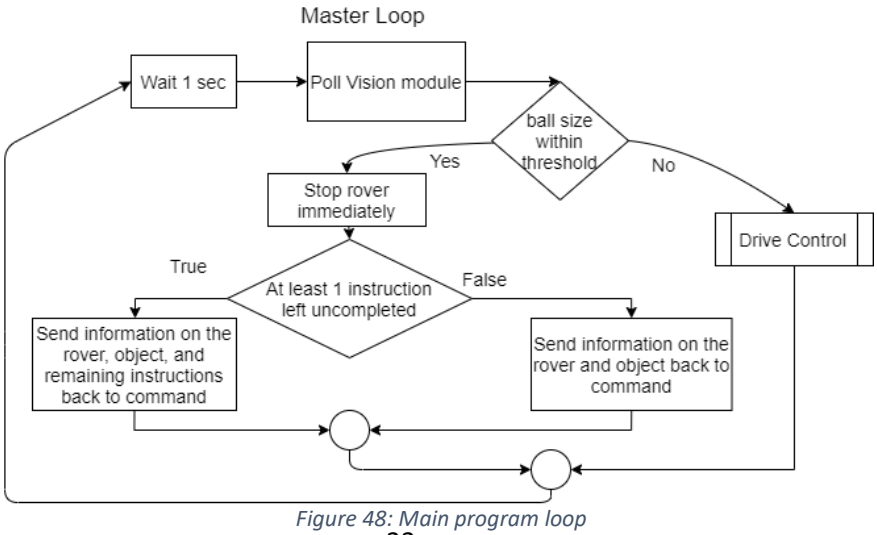


Figure 48: Main program loop

When server sends us a command in the form of a space delimited string, the function `callback()` is called and the instruction string is stored in the variable `remaining_commands_excl_running_command`. Then, all the relevant flags and counters (more on that later) are reset. This happens almost immediately without delay (other than network delay)

Once every second, the control will do the following in its main loop:

1. Query the Vision module to get the size and colour of the object. The query starts with the Control module sending the character 'v' to the Vision module. The Vision module then replies with the horizontal and vertical length (in pixels) of the bounding box for the object detected, and the colour. If the size of the object is within a fixed range (lower bound is set to ignore far objects, and upper bound is set because if the object is too far, the bounding box will suddenly become the whole screen), the `object_detected` flag is set.
    a. There is a grace period of 5 seconds (modifiable in code) that starts counting since the last time a new command was received from Command, and in this time, all objects detected will be ignored (this is because the team stop immediately when the team see an object. This implies that once the team try to start moving again, the team will still see that object, and hence stop again. To avoid that, the team have a 5 second gap of "ignoring" the object)
2. After querying the Vision module, the Control will then decide on what to do for the Drive.
3.

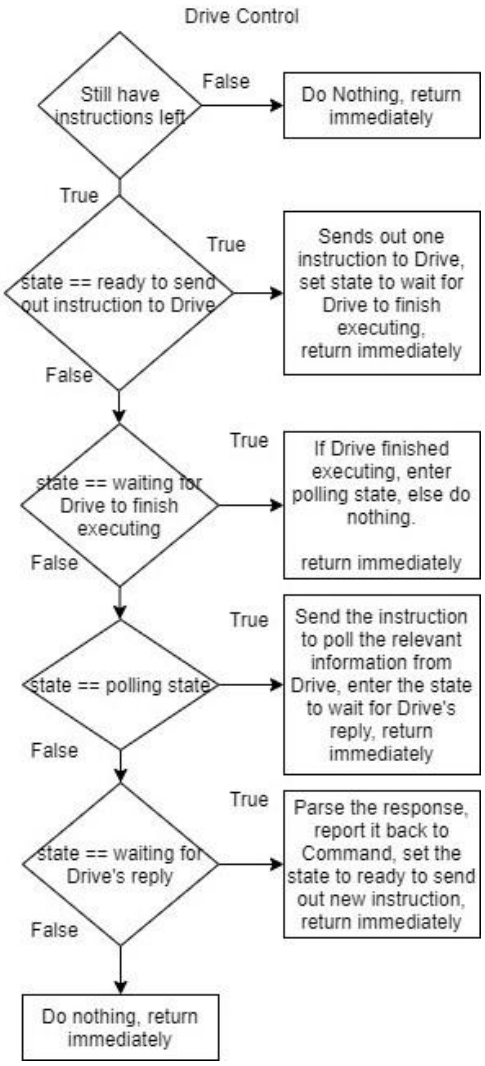


Figure 49:Drive Control Flow

a. IF the `object_detected` flag is set, THEN Control will send "0s000" once every second at Drive, and clear all messages received from Drive from the UART input buffer.

b. ELSE, IF Control is in active mode, THEN Control will send out the nest instruction to Drive and goes into wait mode to wait for the Drive's response. (Note: Control is automatically reset into active mode upon receipt of a fresh instruction list from Command)

c. ELSE, IF Control is in waiting mode, THEN Control will parse every character received from Drive through UART. If 'x' is found anywhere in the string received from Drive, then Control clears all messages received from Drive from the UART input buffer and goes into polling mode.

d. ELSE, IF Control is in polling mode, THEN Control will poll the rover status with 0D000, requesting for the `x_pos`, `y_pos`, `speed`, `distance_travelled`, and `angle`. The Control then enters reply mode.

e. ELSE, IF Control is in reply mode, THEN Control will wait at most 10 seconds for the rover to respond with the status. Once the status (`x_pos`, `y_pos`, `speed`, `angle` and `distance_travelled`) are received, it is published to *rover/status*. The data for `battery_level`, `light_intensity`, `current`, and `voltage` are self-generated.

4. A random notification with a random message may be published to *rover/notif* at random.

5. IF the `object_detected` flag (step 1) is set and there is no grace period left, THEN

a. IF there are remaining instructions to be executed, THEN Control will publish the relevant information to *rover/notif*, only once. After this, only step 3 is executed once every second, until a new set of instructions is received from Command.

b. IF there are no remaining instructions to be executed, THEN Control will publish the relevant information to *map/obstacles*, only once. After this, only step 3 is executed once every second, until a new set of instructions is received from Command.

From the functional description above, it may be noticed that the interface with the Vision module the team glossed over and the Energy module is not mentioned. This is because the interface with the Vision module is simple: Control requests for the Vision module to provide information on whatever it sees, and the Vision module replies. Furthermore, due to the lack of UART ports on the ESP32, the team did not include the Energy module as a direct connection, though it is demonstrated that the design can be extended to include an arbitrary number of modules operating sequentially. The final implementation therefore bypasses control and directly uses a MQTT connection to display values on command; due to the asynchronous nature of MQTT, it does not impact the existing system.

Testing and Implementation

Producers of data will always write the data to a global variable in the current design, hence in the event any testing needs to be done, it can be as simple as just printing data out from the UART serial terminal.
In debugging and narrowing down bugs, a binary search procedure is employed, in which the program is divided into half based on the control flow presented in Figure 1. The whole program is run with some code inserted to print data out of the UART serial terminal at points of interest for the first half of the code, and if some unexpected outputs are obtained, then the search region can be further narrowed down by splitting the region into two again with more print statements. This approach to debugging is very effective as it makes large code fast and easy to debug. The overall time complexity of a binary search is just log n. This implies that if there are N lines of code, the team will not take more than log N tries before isolating the offending code.

Due to the remote nature of this project, it is difficult to test out module interoperability reliably. This was overcome in two primary ways. Firstly, the team used a USB to UART converter to simulate another module's UART output when developing the code for the Control module. The team then send "responses" to the ESP32 from a computer and use the USB to UART converter to double check and confirm what the other module will "see". This allows us to ensure there are no timing bugs or repeated instructions sent. The team also wrote a simple Python script and connected to a free public MQTT broker to simulate the Command module.

In the worst-case scenario in which an unknown error happens during integration, the team solve it by first identifying the lines of code that is most likely the offending piece of code (using binary search again) and do a dump of all the relevant variables involved before and after it. Next, the team trace the results by hand, converting the numbers into binary or hex if necessary, and referring to the table of ASCII codes if necessary. This is primarily to identify synchronisation and endianness issues.

Testing and Results

After testing a few different baud rates, it was found that using a baud rate of 115200 baud was most suitable. Slower baud rates lead to the communication taking too long, hence the communication never ends even after it should end, as the data that should be complexly transmitted in the current session encroaches onto the data to be transmitted in the next session. Having a baud rate that is too high makes it hard for the ESP32 to reliably communicate with other modules without running into any errors. Key results shown below:

Bauds	Bits/s	Bit Duration/ µs	Speed bytes/s	Actual Speed	Byte Duration/ms	Observations
9600	9600	104.167	1200	960 bytes/s	1.042	Able to get full data
115200	115200	8.681	14400	11520	86.806	Data obtained faster
921600	921600	1.085	155200	92160	10.851	Serial print blocking

Figure 50: Baud rates and observations [30]

The team also noticed that switching from HTTP to MQTT allows us to reduce the network latency from about 250 ms to just around 100 ms. This is more than a 50% improvement, illustrating the suitability of MQTT as an IoT communication protocol. Furthermore, MQTT services requires less traffic than HTTP, and when sending

JSON data using MQTT it is especially efficient for receiving and sending vast amounts of data. [31]. From the following table it can be surmised that MQTT can be up to 20 times faster:

1K messages	Bytes transmitted	Number of packets	Time/s
MQTT: 1 pub per message, QoS = 1	283,743	265	5.911
HTTPS: 1 POST per message, keep-alive connection	15,474,263	12,079	115.669
HTTPS:1 POST with 1000 messages	20,515	27	0.307

Figure 51: MQTT vs HTTPS telemetry tests [31]

Discussion/Generalisability of the proposed model

Referring to the main loop of the current model, any extra modules can be added to the current program flow without causing a major disruption to the existing structure. This is because the modules operate sequentially, in a sense, as each loop runs sequentially and modules which act as producers of information can be placed earlier in the loop and modules which act as consumers of information are placed later in the loop. Variables read from producers are stored in global variables for the consumers to read from.

The ability for Command to reset the whole loop and start fresh is also a unique feature, where it can act as a remote controller that is in full control should there be a human operator present. This is enabled by the 5 second cooldown function, where the Command module can constantly send instructions that will override existing instruction executed by the rover. If the rover is left in an unsupervised mode, the list of instructions it was left with will execute until it either an obstacle is detected, or all instructions are completed. Upon detection of an obstacle, it can stop immediately and send a notification to the Command module, which will then alert a human operator who will then send back fresh instructions for the Control to execute.

Another benefit of the design is the regularity of the communication mechanisms between each of the modules. Regular instruction design allows for a clean method of communication between modules, and more complicated instructions can be written as a combination of these basic instructions. If more complicated instructions were required (such as move forward and turn right in one single instruction), then it would need to be broken down into two instructions before sending to drive, and an extra symbol to represent it would be necessary, increasing code size and reduces maintainability. Furthermore, shorter, simpler instructions favour UART transmissions, as UART is asynchronous, and depends on the internal oscillators of the communicating microcontrollers as a timing source. There is no such thing as two oscillators oscillating at the exact same frequency, hence the longer the instruction, the more likely the transmitter and receiver will desynchronise. Shorter instructions make this less likely.

The team also enforced that there can be only one interrupt-based communication with the Control module. Furthermore, the team enforced that when such an interrupt happens, it counts as a full reset. This not only simplifies the control flow, but it also simplifies any subsequent debugging efforts. For one thing, the team enforce that any new instruction coming from Command can only change one thing: the list of instruction to be executed. Hence, when testing, the team just need to check if the change was made correctly and everything else should be unchanged. Furthermore, after the interrupt, every other flag/variable needs to be reset, hence there will be no dependency problems leftover from the previous iterations. Communication with the Drive and Vision modules are also purposely designed to be stateless, which means that errors can be reliably reproduced and taken care of.

In the MQTT communication with the Command module, the team used a JSON format to send data and receive data. The JSON format allows adding of new fields, deleting of existing ones, and recursively adding more attributes to existing members, which allows for the possibility of future expansion.

5.6 Integration Module

Structure and Process

The connection between subsystems is tested and validated using point-to-point integration and the star integration methods [32].The process of integrating the whole system is as follows:

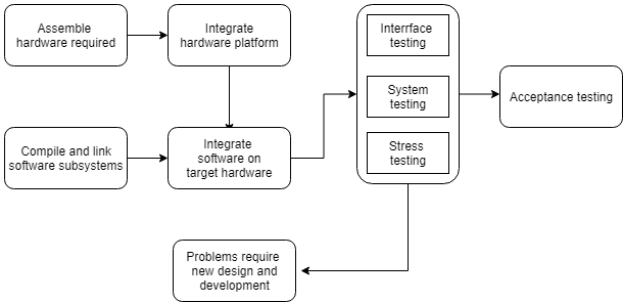


Figure 52:Integration process [32]

Requirements gathering

Frequent communication with each subsystem's leader is held to understand what to expect of the future system. To fully understand interactions between the subsystems, a high-level block diagram is created.

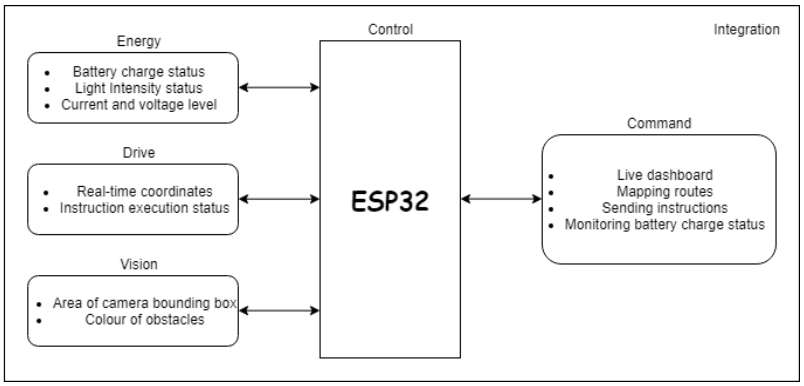


Figure 53: High level block diagram

Validation and testing

Throughout the design process, validation testing is carried out frequently for each of the subsystem and while connecting each of the subsystems. The process and methods used to validate the functionalities the subsystems are described as follow, as well as some quantifying data.

Battery charge status (Energy subsystem):

The connection between the energy subsystem and the ESP32 is setup theoretically due to the limitations on the hardware given. Connection between the energy subsystem and the command subsystem is tested by verifying the transmitting of information through the MQTT server.

Real-time coordinates/Instruction execution status (Drive subsystem):

Stress testing is carried out on the drive subsystem to determine the verify the subsystem's behavior under extreme conditions. The durability of the optical sensor is tested by performing the test cases under different environmental conditions. The team found out that the optical sensor works optimally under gentle lighting and on non-patterned surfaces as the lens can detect the red light more accurately. Under strong lighting, the performance of the optical sensor is compromised.

Further stress testing on the ability to execute instructions was also tested. Test cases instructing the rover to execute the same command continuously with a short delay between each instruction were carried out. These tests proved that the rover could execute the same instructions continuously until the power supply was cut.

Instructions	Time Delay/s	Repeat Duration/s	Dist from expected Position	Behavioural Notes
6f010	0.5	600	1cm	Mostly accurate except for

				oscillations if strong light on red light
5r090	0.5	600	2°	Generally accurate

Figure 54: Excerpt from Integrated Drive tests, Appendix-X

When connecting to the ESP32 and sending real-time coordinates, the accuracy of the rover executing instructions were optimised by setting a suitable amount of delay (figure 47) between each instruction to allow the rover enough time to adjust its position. Connectivity of the drive subsystem to the web app is verified by stress testing the connection points and sending wrong instructions and monitoring the behaviour of the rover.

Obstacle detection (Vision subsystem):

As the test environment for the vision subsystem and the integration subsystem is slightly different, some adjustments were made to the camera such as increasing the gain to obtain a better detection. The vision subsystem is subjected to testing under different conditions, i.e indoors, outdoors, messy space, clean space, different background colours, different lightings. Key findings are detailed below:

Lighting	Gain	Successful Detection rate (/5 balls)
Outside, natural light	0x080	1 (green)
Indoors, lamp	0xFFF	5 (pink, red, orange, green, blue)

Figure 55: Key Integrated Vision tests, Appendix-X

The team found out that the camera works optimally in darker conditions as there is a higher probability of detecting surrounding objects or background as noise. A darker and plain background is also much preferred to a light background as it brings less noise. To further increase accuracy, the team also fine-tuned the object detection algorithm to remove as much noise as possible by adjusting the threshold for detecting obstacles.

The speed of the camera detecting different obstacles while moving around is also monitored on the web app.

Remote control (Command subsystem):

The webapp is subjected to stress testing by identifying the possible scenarios where the web app can break. The team identified that the most likely cause of the web app breaking is when there is a high amount of information being sent to the web app concurrently for a continuous amount of time. Another pain point that the team identified is that some data might be lost while transferring due to a slow network connection. On the other hand, load testing is also carried out to verify the performance of the rover under an expected load.

Final testing

A USB lead is used to power the rover. The FPGA on the rover requires a physical connection to the host computer due to licensing requirements. The final tests were carried out in terms of the following scenarios:

1. Indoors (Wooden floor, indoor lighting, and table lamp)
2. Outdoors (Tiled floor, strong sunlight)

For each case, three scenarios were carried out. Scenarios are as follows:

1. A) 360° scan for obstacles to get fixed obstacles' position on the map, followed by B) repositioning where a route to a coordinate is calculated for the rover whilst avoiding the known obstacles.
2. Unknown obstacles so the rover is initiated with a string of random commands and will self-navigate around the obstacles.

Results

The rover behaves as expected indoors for all three scenarios, with the sensor and camera giving optimal performances. There was some display lag and there are slight oscillations on some occasions while executing instructions but that is due to the direct lighting from the table lamp which is needed to detect the balls compromising the performance of the sensor. Some extra noises are seen by the camera such as from the wooden floor and the white cupboards when light is directly shined on it. A detailed demonstration of the 1st test case is shown in the video. However, the rover did not behave as expected outdoors for all three scenarios

as the lens on the sensor is unable to detect the red light, hence the rover oscillates on the spot most of the time. Furthermore, the strong sunlight is too bright for the Terasic camera to accurately detect the obstacles. Further comments on the 2nd test case will be discussed in the future work section.

		Distance from expected pos(including re-routing)/cm			Average command and display (Edge) time delay/s			Total Travel Time/s		
		T 1	T 2	T3	T 1	T 2	T3	T 1	T 2	T3
Layout 1	1A	N/A	N/A	N/A	1.492	1.092	1.337	101.267	99.595	99.8
	1B	1.04	0.86	0.53	1.499	1.492	1.531	30.2	32.5	37.4
	2	1.06	0.99	0.90	1.572	1.631	1.957	204.5	201.7	208.3

Figure 56: Excerpt of results from Scenario tests, Appendix-X

6. Intellectual Property

According to WIPO, intellectual property refers to 'creations of the mind: inventions, literary, and artistic works, and symbols, names, images and designs used in commerce [33] Engineers are the frontline of innovation; hence it is vital for engineers to know why and how to protect their inventions.

For the Mars Rover Project, the open-ended design methodology that is encouraged for each subsystem opens room for innovation. Theoretically, certain parts of the rover that provides functionalities can be patented, and the look of the rover can be trademarked. To qualify for a patent, the design must be novel and industry applicable. In the rover, the parts which should be protected are the designs of each of the subsystem. For example, the energy subsystem's way of providing energy and the command subsystem's remote dashboard. If one files for a patent, one should also teach how to repeat the invention so that future extensions on the rover is possible. However, before the patent is filled, the rover must not be made public.

Throughout the design phase, IP is constantly taken into consideration. Actions are taken to determine what is required to acquire enforceable designs and devise measures to avoid patent infringement. Throughout the ideation process and writing the report, searches on copyright, patent and trademark search are frequently carried out on websites such as USPTO. By reading through existing inventions, it also provides better insight into the problem at hand. Most importantly, a thorough breakdown of the Mars Rover design process is also explained in the report to enable future work done by other people.

With patent disclosures, this will undoubtedly help with the advancement of innovative and sustainable designs in the future. As such, incorporating IP discussions into the engineering curriculum at an undergraduate level is paramount.

7. Project Management

The team met biweekly, keeping minutes, and used messaging apps for instant communication. Each module leader had full ownership of their tasks and used help sessions to guide overall progress. This informed the PDS and SRS documents, which also directed the teams' efforts towards tasks to prioritise. Moreover, A Gantt Chart was created to keep track of work and to show the development of tasks over the given timeline. Further management details can be found in Appendix I-IV.

8. Future Work and Improvements

For Drive, although the functional requirements were all met, there is still room for further improvement. The current position control strategy uses a combination of threshold region and PI controller. Both input and output of the controller are the position error.

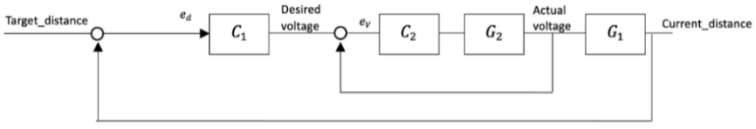


Figure 57: Control diagram

One may also link the speed-control with the position control with a dual-loop controller. The inner loop is the closed-loop buck SMPS implemented previously. The outer loop links the distance to be controlled with the rover speed. $C1$ is a PID controller with parameters to be tuned. The input of $C1$ is the distance_error and the output can be a voltage that will determine the rover speed. Generally, $C1$ should perform the following function: When the `distance_error` is large, the output voltage should be large so that rover could move faster. When the `distance_error` is decreasing, the corresponding output voltage should decrease to make the rover slow down. When the `distance_error` became 0, the speed should also be 0 to stop the rover.

The main advantage of integrating position control and speed control is that it can increase the efficiency of the rover. For example, if the rover is to be moved forward by 100m, in the old controlling scheme the rover can only move at constant velocity and the travel may take a long time. If this new scheme is implemented, rover could accelerate at first and start decelerating when approaching the target destination. The total time taken for travel can be decreased if the parameters can be tuned correctly.

However, there are also drawbacks in this approach. If during the movement, the vision subsystem detects some obstacles, control will send stop signal to drive immediately. If the velocity is at maximum, stopping the rover immediately will be difficult due to inertia. Moreover, rover running at variable speed makes harder for the energy subsystems to track the energy consumed. If there are other on-board vision cameras to take photos of the surroundings, high speed at the beginning of travel would also create blurry images.

To protect the electrical appliances of the rover while it is on Mars, a temperature and dust protection should be developed. As Mars can reach up to -90 degrees, it can damage onboard batteries. Also, as Mars is dusty unlike Earth, the team cannot assume that solar panel will be continuously working through the dust with the equipment's provided. Also, there should be a better electricity generation technique such as nuclear power to provide power during night-time as this is a common practice for actual Mars Rover. LifePO4 battery can also be substituted with actual Lithium-Ion battery for better energy density and temperature management.

As found in the frontend, Voronoi diagrams can be employed to interact with graphs using react-vis. By displaying the Voronoi diagram on the plane the user is given a means to plot their own path. However, using this algorithmically for path finding can be done from Delaunay triangulation [34], enhancing the system further though pre-processing in this case takes O(m+nlogn).

Furthermore, the notification system could be improved, with further details such as image attachments to help diagnose the issues that caused the notification. They could also be cached for later access rather than deleted. In a similar way, the stored commands could be used to load command history, if there is a particular set of commands that is used often. As a method to manually control the rover without rerouting only exists for single commands, an option to switch obstacle detection off may also be useful.

Hardware wise, the rover should be modified further to cope with the rough terrain on MARS instead of a flat surface. Due to the uneven condition on Mars, it is also recommended to decrease speed of the rover to avoid accidents. Better quality sensors would likely improve the accuracy.

9. Conclusion

Overall, the task to create an autonomous rover was achieved effectively. Users can send commands to the rover and receive information back. The rover can traverse a variety of terrains and identify obstacles in each path. Where there are obstacles and commands left, the rover can reroute around the obstacle. The modules were able to interface with each other, and cross module communication was successful. A portable charging system was designed, allowing further exploration, as the rover does not have to return to a station. The modularity of individual sections meant the team could work in parallel, and constant communication meant that intermodular dependencies could be prioritised. The team improved technical understanding and interpersonal management skills. To summarise, the project met all the functional requirements and improved on the existing features in the allotted time to create a successfully working rover.

10.References

[1] E. Stott, “FPGA Starter System Documentation,” 2021. [Online]. Available: https://github.com/edstott/EEE2Rover/blob/master/doc/FPGA-system.md. [Accessed 6 June 2021].

[2] Wikipedia, “Image noise,” [Online]. Available: https://en.wikipedia.org/wiki/Image_noise. [Accessed 6 June 2021].

[3] G. Y. a. S. N. G. Saravanan, “Real Time Implementation of RGB to HSV/HSI/HSL and Its Reverse Color Space Models,” in *2016 International Conference on Communication and Signal Processing*, 2016.

[4] Wikipedia, “HSL and HSV,” [Online]. Available: https://en.wikipedia.org/HSL_and_HSV#Use_in_image_analysis. [Accessed 6 June 2021].

[5] H. A. S. a. E. M. Ahmed, “Effect of the different charging techniques on battery life-time,” *2018 International Conference on Innovative Trends in Computer Engineering (ITCE),* no. 10.1109/ITCE.2018.8316661, pp. 421-426, 2018.

[6] B. T. a. A. Fayed, “An overview of the fundamentals of battery chargers,” *2016 IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS),* no. 10.1109/MWSCAS.2016.7870048., pp. 1-4, 2016.

[7] A. Manfredi, “Re: How to make a modeling of multi- cell batteries in series or parallel?.,” 2016. [Online]. Available: https://www.researchgate.net/post/How-to-make-a-modeling-of-multi-cell-batteries-in-series-or-parallel. [Accessed 5 June 2021].

[8] Y. Barsukov, “Battery Cell Balancing: What to Balance and How,” in *Texas Instruments*.

[9] D. Andrea, “Li-Ion BMS,” 10 February 2012. [Online]. Available: http://liionbms.com/php/wp_parallel_balance.php#:~:text=The%20absolute%20best%20way%20to,the%20cells%20to%20100%20%25%20SOC.. [Accessed 4 June 2021].

[10] S. J. A. M. Ines Baccouche, “Implementation of an Improved Coulomb-Counting,” *INTERNATIONAL JOURNAL of RENEWABLE ENERGY RESEARCH ,* p. 10, 2017.

[11] E. S. S. Lab, “State of Charge Estimation Problem,” UT Dallas, [Online]. Available: https://labs.utdallas.edu/essl/projects/state-of-charge-estimation-problem/. [Accessed 4 June 2021].

[12] H. H. C. G. M. A. R. M. I. R. S. H. Mohammad Reza Maghami, “Power loss due to soiling on solar panel,” *A review, Renewable and Sustainable Energy Reviews,,* vol. 59, no. 1364-0321, pp. 1307-1316, 2016.

[13] J. A. S. K. a. Y. L. T. C. T. K. Kho, “A comprehensive review on PV configurations to maximize power under partial shading,” in *TENCON 2017 - 2017 IEEE Region 10 Conference*, 2017.

[14] A. Rayl, “The Planetary Society,” 2 October 2017. [Online]. Available: https://www.planetary.org/articles/09-mer-update-opportunity-braves-winter. [Accessed 4 June 2021].

[15] T. Inc, “DE10-Lite_User_Manual,” 5 June 2020. [Online]. Available: file:///C:/Users/user/Downloads/DE10-Lite_User_Manual.pdf. [Accessed 2021 June 15].

[16] N. Potter, “NASA’s Mars Perseverance Rover Should Leave Past Space Probes in the Dust,” IEEE, 16 February 2021. [Online]. Available: https://spectrum.ieee.org/tech-talk/aerospace/space-flight/nasa-mars-perseverance-rover-should-leave-past-space-probes-in-dust. [Accessed 4 June 2021].

[17] B. Bhandari, “React Functional Components VS Class Components,” 21 Feb 2020. [Online]. Available: https://medium.com/wesionary-team/react-functional-components-vs-class-components-86a2d2821a22. [Accessed 20 May 2021].

[18] uber, “react-vis,” [Online]. Available: https://uber.github.io/react-vis/. [Accessed 18 May 2021].

[19] M. E. Ercan. [Online]. Available: https://medium.com/capital-one-tech/a-comparison-of-data-visualization-libraries-for-react-27f8fe409934 . [Accessed 18 May 2021].

[20] NPMCompare, “Comparing react-vis vs. recharts vs. spectacle vs. victory,” [Online]. Available: https://npmcompare.com/compare/react-vis,recharts,spectacle,victory#:~:text=When%20comparing%20those%20packages%20you,on%20Github%20and%20more%20forks.. [Accessed 18 May 2021].

[21] R. Barger, “How to Create a React App with a Node Backend: The Complete Guide,” freecodecamp.org, 3 February 2021. [Online]. Available: https://www.freecodecamp.org/news/how-to-create-a-react-app-with-a-node-backend-the-complete-guide/. [Accessed 15 May 2021].

[22] shresths, “aws-s3-bucket-upload,” 19 November 2018. [Online]. Available: https://github.com/shresths/upload-files-aws-s3-bucket/blob/master/aws-s3-bucket-upload.html. [Accessed May 2021].

[23] M. Yuan, “Getting to know MQTT,” IBM, 7 January 2020. [Online]. Available: https://developer.ibm.com/technologies/messaging/articles/iot-mqtt-why-good-for-iot/. [Accessed 22 May 2021].

[24] [CG]Maxime, “Finding Shortest Path in the Plane with Obstacles,” Codeingame, [Online]. Available: https://www.codingame.com/playgrounds/39380/finding-shortest-path-in-the-plane-with-obstacles#graph. [Accessed 20 May 2021].

[25] G. Bar-Gil, “NPM vs. Yarn: Which Package Manager Should You Choose?,” 12 August 2020. [Online]. Available: https://www.whitesourcesoftware.com/free-developer-tools/blog/npm-vs-yarn-which-should-you-choose/. [Accessed 14 May 2021].

[26] msanford, “When to use Yarn over NPM? What are the differences?,” 20 March 2018. [Online]. Available: https://stackoverflow.com/questions/40027819/when-to-use-yarn-over-npm-what-are-the-differences. [Accessed 13 May 2021].

[27] Yida, “UART vs I2C vs SPI – Communication Protocols and Uses,” Seed Studio, 25 September 2019. [Online]. Available: https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/. [Accessed 10 June 2021].

[28] Guru, “MQTT vs HTTP: why you should use MQTT to control your actuators ?,” 21 April 2020. [Online]. Available: https://blog.asksensors.com/mqtt-vs-http-why-you-should-use-mqtt-to-control-your-actuators/.

[29] C. Wang, “HTTP vs. MQTT: A tale of two IoT protocols,” Google Cloud, 27 November 2018. [Online]. Available: https://cloud.google.com/blog/products/iot-devices/http-vs-mqtt-a-tale-of-two-iot-protocols. [Accessed 10 June 2021].

[30] Lulu, “Most common baud rates table,” [Online]. Available: https://lucidar.me/en/serialib/most-used-baud-rates-table/. [Accessed May 2021].

[31] J. Bartinksi, “HTTP vs MQTT performance tests,” 23 January 2018. [Online]. Available: https://flespi.com/blog/http-vs-mqtt-performance-tests. [Accessed 7 June 2021].

[32] Folio3 Microsoft Dynamics Partner, “folio3,” [Online]. Available: https://dynamics.folio3.com/blog/system-integration/. [Accessed 7 June 2021].

[33] World Intellectual Property Organization , “World Intellectual Property Organization (WIPO),” [Online]. Available: https://www.wipo.int/about-ip/en/. [Accessed 8 June 2021].

[34] O'Rourke, “Voronoi Diagrams and Delaunay Triangulations,” February 2016. [Online]. Available: https://www.cs.jhu.edu/~misha/Spring16/11.pdf. [Accessed 19 May 2021].

[35] L. B. a. S. J. Nassim Noura, “A Review of Battery State of Health Estimation,” *World of Electric Vehicle,* p. 20, 2020.

[36] BatteryGuy, “Connecting batteries in parallel,” 13 September 2019. [Online]. Available: https://batteryguy.com/kb/knowledge-base/connecting-batteries-in-parallel/. [Accessed 5 June 2021].

[37] California State University Northridge, “Functional Design,” [Online]. Available: http://www.csun.edu/~twang/595WEB/Slides/Week9.pdf. [Accessed 09 June 2021].

[38] wikipedia, “Model–view–controller,” 03 June 2021. [Online]. Available: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller. [Accessed 03 June 2021].

[39] Pete Houser, “Northrop Grumman Corporation,” November 2011. [Online]. Available: https://ndiastorage.blob.core.usgovcloudapi.net/ndia/2011/system/13007_HouserThursday.pdf. [Accessed 9 June 2021].

11. Appendix

Appendix-I

11.1 Product Design Specification

Date: June 11, 2021

[Revised June 15,2021]

This product design specification is a statement of what this Mars Rover is intended to do. The aim of this document is to ensure that the subsequent design and development of the product meets the requirements of the client.

Background Introduction

The Mars Rover is an electronic design project that is conducted under Imperial College London to second year students in Electrical and Electronic Engineering. It is designed to operate under autonomous conditions and to be resilient under Mars' condition.

Basic Operation – The Mars Rover has a lifecycle of charging, discharging , driving the motors, observing the target, and performing operations assigned through the webapp. All divisions should communicate with each other using ESP32, FPGA and Arduino

Scope

This specification covers the general operation characteristics of the product and provides an overview of the requirements for the electronics of the finished article. This is a response to the criteria laid out by the client.

Product Design and Performance

1. Performance

   The rover should achieve all the target performances and have an autonomous operation. It should be able to communicate and operate multiple operations at the same time.

2. Maintenance

   As the materials are affordable and long-lasting, it is expected that the project should be easy to maintain and requires little cost for any maintenance. The project should be easily maintained by the webapp, and data should be sent continuously.

3. Target Product Cost

   It will cost about £200.

4. Quantity

   Produce one Mars Rover for testing. The codes for multiple division should be uploaded to GitHub so that it can be shared and built if required.

5. Customer

   It is not intended for commercial use. The target market would be researchers who are keen on studying Mars atmosphere, or students who want to use the system and build on it to make hardware/software upgrades. Private organisations can use it to explore other terrains and identify objects and hobbyists can use individual modules to create IoT systems

6. Size

   Length: 24cm, width:20.5cm , height:12.5 cm

7. Weight

   The finished integrated project has a weight of 0.75kg.

8. Materials

   Materials can be easily purchased and substituted in case it is not available. However, the substituted materials should have the same function such as the SMPS. It can be improved by buying better materials depending on use.

9. Aesthetics, Appearance and Finish

   The final product should be equipped with solar panels on the sides and the top. The wires should be carefully laid out to avoid any problems and the batteries should be properly arranged and wired to respective ports on Arduino to avoid short circuits. The provided chassis can be modified.

10. Ergonomics

    The project should be made in a suitable duration and should complement the strength and weaknesses of all team members. It should be adjusted to fit all team members time zones.

11. Standards and Specifications

    It should perform all the functions stated in the requirements. There should be no sharp edges or poisonous substances and conform to legislation set by EU.

12. Quality and Reliability

    The rover should be of high quality and reliable for use autonomously over a long duration of time. All divisions must be reliable as standalone modules, allowing for use in further projects. Once integrated, communication should be consistent between each division.

13. Testing

    The testing is done separately according to division first and then integrated to make sure the software and hardware are compatible with each other and not cause damage in the long run.

14. Processes

    The initial process should be researching the individual division. After compiling the findings, a group discussion should ensue to make sure all the division are compatible with one another. Consultations and questions are attended by all team members to make sure every question is answered. All team members should cooperate to produce the deliverables and the full schematic.

15. Time Scale

    The project should take a month approximately to research, assemble, produce, and document. A Gantt Chart is created to make sure all progresses are tracked.

16. Safety

    The connections should be safe for use and handling. The project should use a reliable connection between components. The documentation and files of the project should also be kept in a safe folder and shared privately between team members. Batteries should be maintained well and no possibility of explosion.

17. Company Constraints

    The report and video are due on the 15th of June 2021 and must be submitted via Blackboard. The project should be finished by that date, together with project management.

18. Patents, Literature and Product Data

    The project should avoid any accusation of plagiarism by referencing the right sources in the documentation. The project's documentation should be shared privately. There should not be any plagiarism between peers as well.

19. Installation

    Full code and installation steps can be found on GitHub. Hardware set up steps can be found via Teams.

20. Documentation

    The full documentation must be submitted to the company, Imperial College London.

Appendix-II

11.2 Software Requirements Specification

1. Introduction

1.1. Purpose

The purpose of this document is to present a description of the software implemented within each of the subsystems of the Mars Rover. The features and constraints under which it must operate will be presented.

1.2. Intended Audience

- Students will be able to use the system and build on it to make hardware/software upgrades
- Private organisations can use it to explore other terrains and identify objects
- Hobbyists can use individual modules to create IoT systems

1.3. Intended Use

The Mars Rover is comprised of several subsystems, where each subsystem is intended to be used as standalone modules and integrated together to create an autonomous mapping system.

1.4. Scope

1.5. Definitions and Acronyms

SRS: Software Requirements Specification
JSON: JavaScript Object Notation
NPM: Node Package Manager
CSV: Comma Separated Values
HTTP: Hypertext Transfer Protocol
MQTT: Message Queuing Telemetry Transport
AWS: Amazon Web Services
UI: User Interface

2. Overall Description

2.1. Product Perspective

The Mars Rover will be made up of the following subsystems:
A) Drive: The rover should be able to move in all directions and varying speeds. The position should also be tracked, and it should be able to give updates on its current location and angle, by sending information to Control to be forwarded to Command.
B) Vision: The camera should be able to detect different colours(Initially ping pong balls). Colours detected should be used to determine the existence of obstacles; any such information should be sent to the Control Module. This module should choose the closest obstacle to be sent(the one with the most danger).
C) Energy: The rover will have a portable charging method to power the various types of hardware required for the modules above. The system should keep the rover charged where possible and supply the required voltages and current to maintain functionality during the day.
D) Command: A webapp will be used to keep users updated with rover statuses by displaying the data in a digestible format. There will be various options to move the rover to explore some terrain and will ultimately display a map of the working area.
E) Control: This brings all the other modules together. It will use an ESP32 to facilitate communication between all the modules, ensuring command gets the information needed and calculating the distance of the rover from the next obstacle, and stopping in time.

2.2. Assumptions and Dependencies

Software packages for Drive, Energy and Control will be developed with C++ on Arduinos, and external tools for testing may involve the use of bash scripting. Terminal knowledge is assumed, to be able to navigate files. Vision will be developed using Quartus and Verilog, using the NIOS terminal to install relevant software. Command will make use of external libraries which can be obtained using NPM; once deployed however, there will be no requirements other than a compatible web browser.

2.3. Operating Environment

Operating System: N/A
Browser Requirements: IE>11
Languages: C++, Javascript, Python
Working Area: The rover can navigate a range of terrains; any area up to $5m^2$ is ample for testing purposes.

3. System Features and Requirements

3.1. System Description and Priority

The system will be able to move a Rover in a desired direction with a user specified speed. The Rover will be able to identify objects and navigate around them or alert the user. The User should be able to access a webapp with full control of the rover and understanding of the status updates. The Rover should be powered via batteries charged using solar panels, and it should maintain healthy charge where possible. It is necessary to be able to host the webapp and connect to the database using HTTP requests; MQTT protocols are used to connect with the rest of the subsystems, which can be hosted on a range of services such as AWS. Control can be used directly to communicate with drive and see the obstacles; in this way, it is possible to run the whole system locally.

3.2. Functional Requirements

Drive

Initial Requirements

- Enable the rover to move at the desired speed.
- Enable the rover to move to the desired position.
- Enable the rover to turn to the desired angle.
- Record the real-time co-ordinate of the rover during movement.
- Send back necessary information back to the control subsystem.

Additional Requirements

- Ensure that the rover remain stable during movements.
- Ensure the rover position is as accurate as possible.
- Ensure the rover communicate with control effectively.

Vision

Initial Requirements

- Implement intermediate components in the image processor
- Identify well defined obstacles(Ping pong balls)
- Send data(bounding box, colour) about obstacles to Control

Additional Requirements

- Indicate the colour of the closest obstacle
- Ensure the camera sees aggregated pixels as colours
- Ensure the data stream is reliable

Energy

Initial Requirements

- Provide sufficient power to the motor and ESP32 to operate at different times of the day.
- Determine configuration of the solar panels.
- Devise MPPT Strategy to get maximum power point from the solar panels depending on irradiance.
- Choose optimum battery configuration and number of cells for powering motor.
- Decide on best charging strategy to power the LiFePo4 battery.
- Maintain a SOH and SOC estimation method.
- Communicate with Drive, Control and Command Module to integrate the energy to module to the rover remotely and physically.

Additional Requirements

- Able to track MPPT continuously in the fast loop during different irradiance condition.
- Track individual battery voltage and current to prolong battery life.
- Charge, discharge, and balance simultaneously using two SMPS and two Arduinos for mounted battery use on rover.
-

Command

Initial Requirements

- Receive and display information regarding energy, vision and drive subsystems
- Save rover information in an offsite area
- Control the rover remotely
- Alert the user to obstacles
- Keep track of obstacles on a map

Additional Requirements

- Be able to reroute around obstacles
- Client-side interaction support – ease of navigation, map zoom
- Server-side content handling- real time updates
- Server-side management of large data sets – synthesizing information from the database effectively

Control

Initial Requirements

- Determine protocols to be used in communications
- Forward Information from the various other submodules to Command.
- Parse data into JSON format for Command
- Forward instructions from Command to Drive
- Stop the Rover when an obstacle is detected by Vision

Additional Requirements

- Implement a code for communication between Drive and Command.
- Allow interruption of instructions from Command at any time.
- Infer distance from the obstacles via the Vision Bounding Box to calculate when to stop.

Following these requirements being met, the Integration module is expected to be able to bring all the subsystems together and create a working Rover from the hardware provided.

3.3. Non-Functional Requirements

Usability

Each of the subsystems has a terminal interface available to view data directly. Once the system is completely set up, the user should only require the UI provided by Command to understand the other subsystems.

Reliability

Individual submodules should be debugged prior to connecting. Once connected, modules should be fully operational and communication between submodules should be consistent, with reliable protocols chosen. In the event no communications are available, the Rover is expected to continue attempting to send information; the MQTT broker chosen allows for a buffer of sent messages, which will be sent at the next available instance.

Performance

Latency between modules should be minimised to allow for close to real time processing of data. Arduinos have limited storage, and the FPGA has latency in outputs, so calculations should be divided between the submodules to maximise performance.

Scalability

The software should allow easy implementation of new features. The hardware should be replicable and thus allow a network of rovers to be created if necessary. The rover should be able to transmit data from a range of distances, and the dashboard/control should be compatible to communicate with other devices.

4. External Interface Requirements

4.1. User Interface

To command the whole subsystem, the user will be required to use a browser to access the webapp. To see the camera output, a VGA to USB converter can be used. Individual modules can be utilised using the standard Arduino IDE.

4.2. Hardware Interface

Modules: • 1 Optical flow sensor • 1 SMPS add-on PCB with H-bridge • 1 Terasic Technologies D8M-GPIO Camera • 1 VGA to USB converter • 1 ESP32 Shield Module Wires and cables: • 4 Stranded wire 120mm • 3 Solid core wire 250mm • 1 USB Micro cable 3.0m • 1 USB Micro cable 1.5m	Mechanical Equipment: • 1 Chassis • 1 Chassis fastener kit (angle brackets, M3 screws and nuts, M4 screw and nuts) • 1 FPGA fastener kit (4× M3 screws and spacers)1 Optical flow sensor fastener kit (2× long M3 screws and nuts) • 2 Wheel • 2 Motor gearbox • 2 Sticky pad for terminal block Note: Communications will be via UART

Software Interface

The final product will require the user to have a stable internet connection and a capable browser. Should they wish to create their own broker, they are advised to create a new AWS instance. All other software can be downloaded onto items in the hardware list to be used as individual modules and interconnected together.

Appendix-III

11.3 Meeting Minutes

Meeting Date	Discussions
13/05	• General Set up • Team communications • Research and understanding of provided resources
18/05	• Clarifications following consultations • Dividing tasks, researching specific requirements • Information modules will communicate
21/05	• General Updates • How modules will communicate • Levels of implementation • Plan for the week
24/05	• Integration • Troubleshooting • Test scripts/methods
28/05	• Command-Control-drive • MQTT debugging • Updates on information to be sent
2/06	• Energy structure • Communications with Command • Updates on precision and accuracy
04/06	• Command-control-vision • Camera output • Using UART
06/06	• Debugging whole system • Report plan • Creating video scenarios

Appendix-IV

11.4 Gantt Chart Tasks

TASK	ASSIGNED TO	START	END
Command			
Research Webapp technologies(tutorials etc)	Command	2021-05-11	2021-05-16
Front end UX design	All	2021-05-14	2021-05-17
Rover Control widget(Frontend)	Command	2021-05-17	2021-05-19
Energy display and parsing(Frontend)	Command, Energy	2021-05-20	2021-05-22
Database Schema (Storage, Data transfer)	Command, Control	2021-05-17	2021-05-22
Backend API (Node JS, Express)	Command	2021-05-21	2021-05-27
Communication Protocols Testing	Command, Control	2021-05-24	2021-05-28
Commands translation	Command, Control	2021-05-29	2021-05-31
Website docs, faq, information(Frontend)	All	2021-06-01	2021-06-06
Implementing Map views and updates (Frontend)	Command	2021-05-27	2021-05-31
Path finding, obstacle avoidance algorithm research	Command, Control, Drive, Vision	2021-05-24	2021-05-31
MQTT implementation, Data transfer format	Command, Control, Integration	2021-05-27	2021-06-06
Integrating with the Rover	Command, Integration	2021-06-03	2021-06-13
Video storyboarding and filming	All	2021-06-07	2021-06-13
Report	All	2021-05-31	2021-06-13
Control			
Research different type of protocols (HTTP, MQTT, TCP)	Control	2021-05-12	2021-05-14
Connect ESP32 to the internet and make HTTP requests	Control	2021-05-14	2021-05-17
Make MQTT publications and subscriptions, Serial port	Control	2021-05-14	2021-05-17
Set up local HTTP server to test connetions	Control	2021-05-17	2021-05-18
Implement agreed MQTT interfaces with Command	Control, Command	2021-05-18	2021-05-21
study and familiarize with MQTT structure and protocol	Control	2021-05-19	2021-05-23
Handle Communication with Command, create test scripts	Control, Command, Integration	2021-05-24	2021-05-27
Handle Communications with Drive using USB to UART converter	Control, Drive, Integration	2021-05-26	2021-05-30
Interface with Vision using UART	Control, Vision, Integration	2021-05-31	2021-06-04
Video Help	Control	2021-06-05	2021-06-09
Report Writing	Control	2021-06-08	2021-06-13
Vision			
Research object detection algorithm	Vision	2021-05-12	2021-05-16
Build the starter project	Vision	2021-05-14	2021-05-16
Understand initial version of image processor	Vision	2021-05-15	2021-05-19
Research methods to remove noise	Vision	2021-05-17	2021-05-25
Implement mean filter	Vision	2021-05-18	2021-05-23
Research colour space conversion	Vision	2021-05-22	2021-05-24
Implement coversion from RGB to HSV in Verilog	Vision	2021-05-22	2021-05-28
Find range of values for other colour balls i.e. orange, green, blue and pink	Vision	2021-05-26	2021-06-05
Testing video output and message in nios2 terminal	Vision, Integration	2021-06-03	2021-06-06
UART communication from vision to control	Vision, Control, Integration	2021-06-01	2021-06-06
Integrate with rover	All	2021-06-07	2021-06-10
Report writing	All	2021-06-06	2021-06-11
Drive			
Research position control algorithm	Drive	2021-05-12	2021-05-18
Understand and link PCB layout to the design	Drive	2021-05-13	2021-05-19
Assenble the rover hardware	Drive	2021-05-19	2021-05-19
Testing&Understanding of how rover is moved	Drive	2021-05-20	2021-05-21
Method to receive instruction from control/ Instrution format	Drive, Control, Integration	2021-05-21	2021-05-23
Design & Implementationof position control	Drive	2021-05-24	2021-05-27
Design & Implementationof turning/direction control	Drive	2021-05-25	2021-05-29
Design & Implementation of speed control	Drive	2021-05-28	2021-05-30
Improving rover stability/accuracy performance	Drive	2021-05-29	2021-06-01
Design & Implementation of co-ordinate turn method	Drive	2021-06-01	2021-06-07
Returning information to Control subsystem	Drive, Control, Integration	2021-06-07	2021-06-08
Integrating drive subsystem onto rover and testing	Drive, Integration	2021-06-04	2021-06-13
Report writing	All	2021-06-07	2021-06-13

Energy			
Research battery and charger implementation	Energy	2021-05-12	2021-05-15
Research charging method	Energy	2021-05-15	2021-05-18
Determine ways to Test SOC and SOH by battery capacity	Energy	2021-05-19	2021-05-21
Research and implement parallel and series battery arrangement	Energy	2021-05-21	2021-05-28
Address safety concerns in battery	Energy	2021-05-27	2021-05-28
Tested PV panels with lamp and outdoor condition	Energy	2021-05-29	2021-05-31
Determine bestt PV Panels Configuration for MPPT and irradiance	Energy	2021-06-01	2021-06-04
Integrated all energy components into one using buck	Energy	2021-06-05	2021-06-08
Implementation of discharging in Drive	Drive, Energy	2021-06-07	2021-06-10
Testing all charging and discharging	Drive, Energy	2021-06-10	2021-06-11
Integrating all components with control, command , drive	rive, Control, Integration, Energy, Commar	2021-06-07	2021-06-10
Report	All	2021-06-07	2021-06-13
Integration			
Research integration methods	Integration	2021-05-12	2021-05-15
Setup required hardware and software platforms	Integration	2021-05-14	2021-05-15
Setup UART connection between Arduino Nano Every and ESP32	Control, Integration	2021-05-19	2021-05-20
Setup communication between drive and control	Control, Drive, Integration	2021-05-21	2021-05-23
Tested communication with command using different test scripts	Control, Command, Integration	2021-05-24	2021-05-27
Tested communication with drive	Control, Drive, Integration	2021-05-25	2021-05-29
Setup and tested UART connection between control and vision	Control, Vision, Integration	2021-05-31	2021-06-04
Testing video output and terminal messages	Control, Vision, Integration	2021-06-02	2021-06-06
Tested new drive code with command and control	Control, Command, Drive, Integration	2021-06-06	2021-06-08
Integrated all subsystems onto the rover	All	2021-06-09	2021-06-10
Tested the whole rover	All	2021-06-09	2021-06-13
Video storyboarding, filming and editing	All	2021-06-12	2021-06-13
Report	All	2021-06-09	2021-06-13

The team set out some high-level goals that we wanted to achieve by the end of the project. These were then divided into subcomponents that were flexible in terms of the timeframe. We met twice a week on teams to give progress updates and plan next steps. This Gantt chart was used to keep track of progress and provide an initial framework for tasks that needed to be done. Inter module tasks were left for later in the timeline so that the basic outline of each submodule was in place, and testing occurred iteratively throughout. Furthermore, consultation sessions were utilised, and the relevant team members were encouraged to attend; if there were any further clarifications for the team we would use direct messages to communicate and confirm what other submodules had to do.

Appendix-V

11.5 Section of code on the RGB-->HSV conversion

```
91  //RGB to HSV conversion
92  wire rg, rb, gb;
93  reg [7:0] maxc, minc, diff, val;
94  wire [13:0] val_60;
95  wire [7:0] division;
96  reg [8:0] hsv_h;
97  reg [7:0] hsv_s, hsv_v;
98
99  assign rg = (red > green);
100 assign rb = (red > blue);
101 assign gb = (green > blue);
102
103 always@(posedge clk) begin
104     if(~rg & ~rb & ~gb) begin
105         maxc[7:0] <= blue[7:0];
106         minc[7:0] <= red[7:0];
107         diff[7:0] <= blue[7:0] - red[7:0];
108         val[7:0] <= green[7:0] - red[7:0];
109     end
110     else if(~rg & ~rb & gb) begin
111         maxc[7:0] <= green[7:0];
112         minc[7:0] <= red[7:0];
113         diff[7:0] <= green[7:0] - red[7:0];
114         val[7:0] <= blue[7:0] - red[7:0];
115     end
116     else if(~rg & rb & gb) begin
117         maxc[7:0] <= green[7:0];
118         minc[7:0] <= blue[7:0];
119         diff[7:0] <= green[7:0] - blue[7:0];
120         val[7:0] <= red[7:0] - blue[7:0];
121     end
122     else if(rg & ~rb & ~gb) begin
123         maxc[7:0] <= blue[7:0];
124         minc[7:0] <= green[7:0];
125         diff[7:0] <= blue[7:0] - green[7:0];
126         val[7:0] <= red[7:0] - green[7:0];
127     end
128     else if(rg & rb & ~gb) begin
129         maxc[7:0] <= red[7:0];
130         minc[7:0] <= green[7:0];
131         diff[7:0] <= red[7:0] - green[7:0];
132         val[7:0] <= blue[7:0] - green[7:0];
133     end
134     else if(rg & rb & gb) begin
135         maxc[7:0] <= red[7:0];
136         minc[7:0] <= blue[7:0];
137         diff[7:0] <= red[7:0] - blue[7:0];
138         val[7:0] <= green[7:0] - blue[7:0];
139     end
140 end
141
142 assign val_60 = {val,6'b000000} - {val,2'b00};
143 assign division = (diff > 0) ? (val_60/diff) : 8'd240;
144
145 always@(posedge clk) begin
146     if(~rg & ~rb & ~gb) begin
147         hsv_h <= 9'd240 - division;
148     end
149     else if(~rg & ~rb & gb) begin
150         hsv_h <= 9'd120 + division;
151     end
152     else if(~rg & rb & gb) begin
153         hsv_h <= 9'd120 - division;
154     end
155     else if(rg & ~rb & ~gb) begin
156         hsv_h <= 9'd240 + division;
157     end
158     else if(rg & rb & ~gb) begin
159         hsv_h <= 9'd360 - division;
160     end
161     else if(rg & rb & gb) begin
162         hsv_h <= division;
163     end
164
165     hsv_s <= (maxc > 8'd0) ? {diff[7:0],8'b00000000} / maxc : 8'd0;
166     hsv_v <= maxc;
167 end
```

Appendix-VI

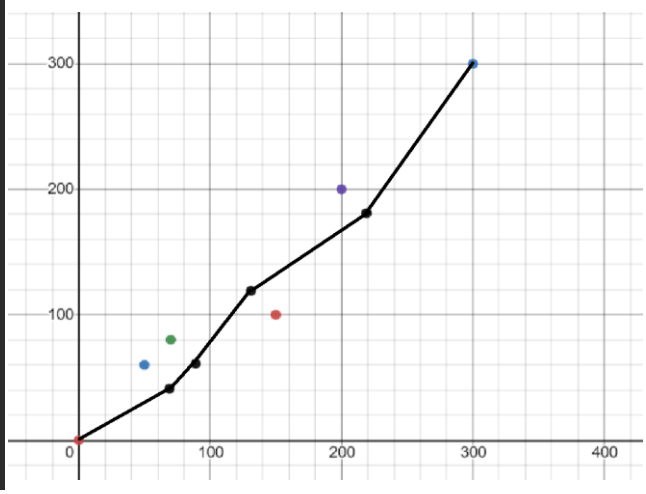
11.6 Path Finding

Paths can only be formed once polygons have been constructed. This snippet of code occurs before commands are sent via MQTT, and aims to convert coordinates into viable instructions. The centre of the rovers and obstacles are used to calculate safe paths in graph constructions, and a map is used to give each obstacle coordinate a polygon boundary. These become a series of coordinates that can be used in the shortest path function, where a q is created and each coordinate is checked for the optimal distance to the destination, if such a path exists. If a new path found is shorter than an old path, the previous data gets overwritten, until an optimal set of coordinates is found.

```
function shortestPath(graph, srcNode, dstNode, co) {
    var q = [{x: srcNode.x, y: srcNode.y, path: [srcNode], currentLength: 0, heuristic: 0}]
    var visited = {}
    while (q.length > 0) {
        q.sort((a, b) => a.heuristic - b.heuristic) // this should be a queue
        const el = q.shift()

        if (directPath(dstNode, el,co)) {
            return [...el.path, dstNode]
        }
        visited[`${el.x} ${el.y}`] = true

        const successors = graph[`${el.x} ${el.y}`]
        if (!successors) {
            continue
        }


        successors
            .filter(succ => !visited[`${succ.x} ${succ.y}`])
            .forEach(succ => {
                //console.log("succ")
                const distToSucc = distance(succ, el)
                const newPath = el.path.slice()
                const currentLength = el.currentLength + distToSucc
                const heuristic = currentLength + distance(succ, dstNode)
                newPath.push({x: succ.x, y: succ.y})
                q.push({x: succ.x, y: succ.y, path: newPath, currentLength, heuristic})
            })
    }
```

```
function buildGraph(coords) {
    // Add edge from each vertex to all visible vertex
    const allVertices = coords
        .map(dst => polygon(dst))
        .reduce((a, b) => [...a, ...b])

    const graph = {}
    coords.forEach(src => {
        const srcPoly = polygon(src)

        // Centers can also reach any visible vertices
        srcPoly.push(src)

        srcPoly.forEach(srcP => {
            allVertices
                .filter(c => c.x !== srcP.x || c.y !== srcP.y)
                .forEach(c => {
                    if (directPath(srcP, c, coords)) {
                        const key = `${srcP.x} ${srcP.y}`
                        if (graph[key] == null) {
                            graph[key] = []
                        }
                        graph[key].push(c)
                    }
                })
        })
    })
    return graph;
}
```

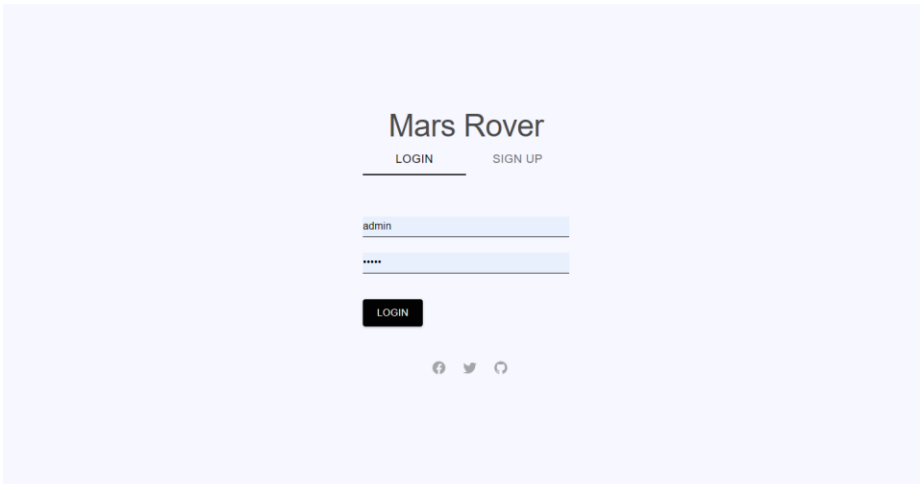
```
{ x: 0, y: 0, radius: 10 },
{ x: 300, y: 300, radius: 1 },
{ x: 300, y: 500, radius: 2 },
{ x: 150, y: 100, radius: 4 },
{ x: 50, y: 60, radius: 4 },
{ x: 70, y: 80, radius: 4 },
{ x: 200, y: 200, radius: 4 }
```

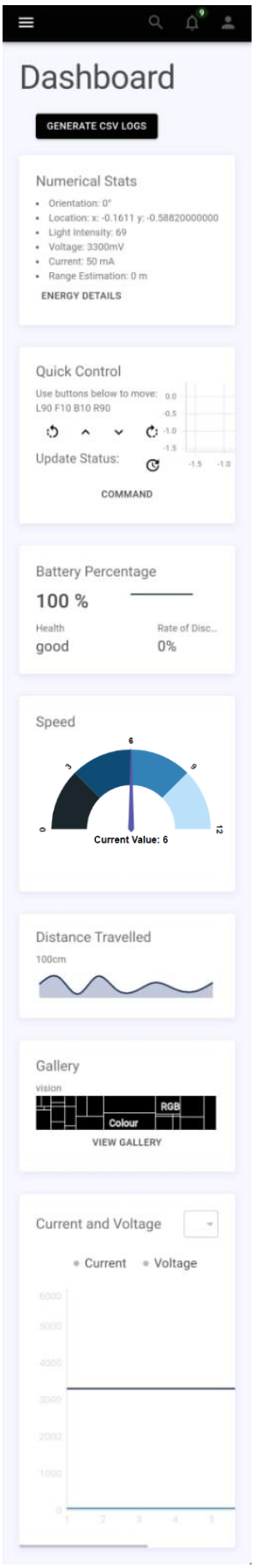


Appendix-VII

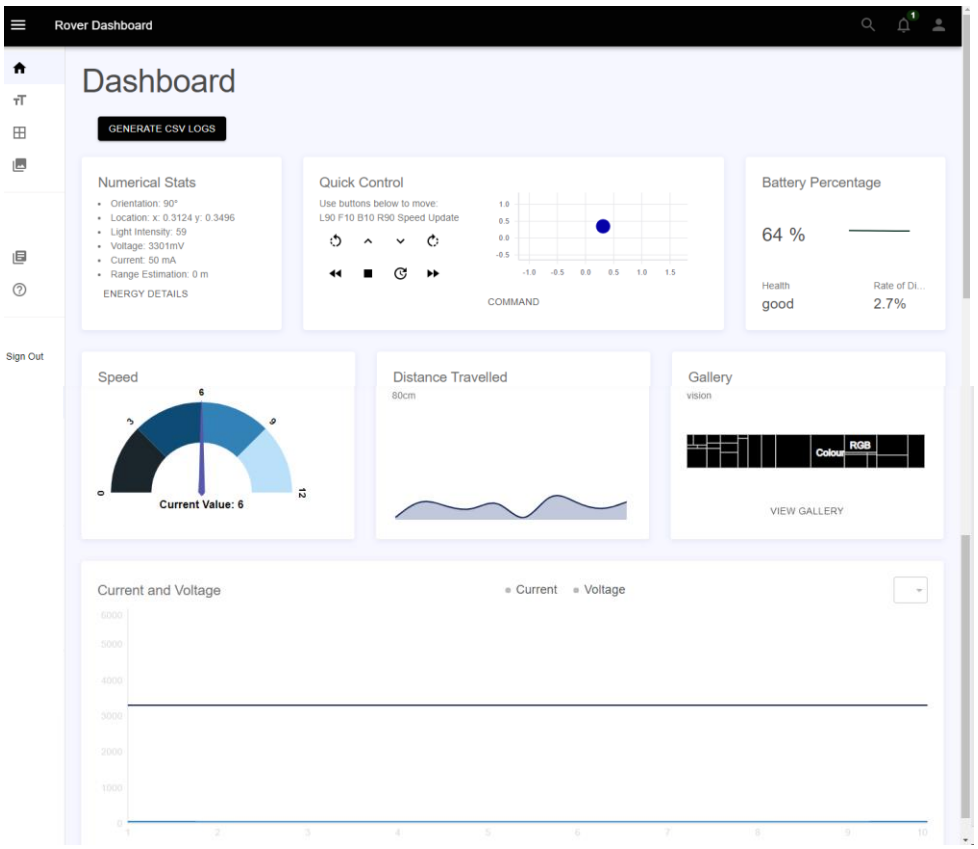
11.7 Key UX Views

Following login, the user is presented with the **dashboard.** It contains battery and rover data. It is organised into blocks so that the webapp can expand for phone use with some changes to the view, as shown on the left. All functions are as normal, including downloading of CSV files.

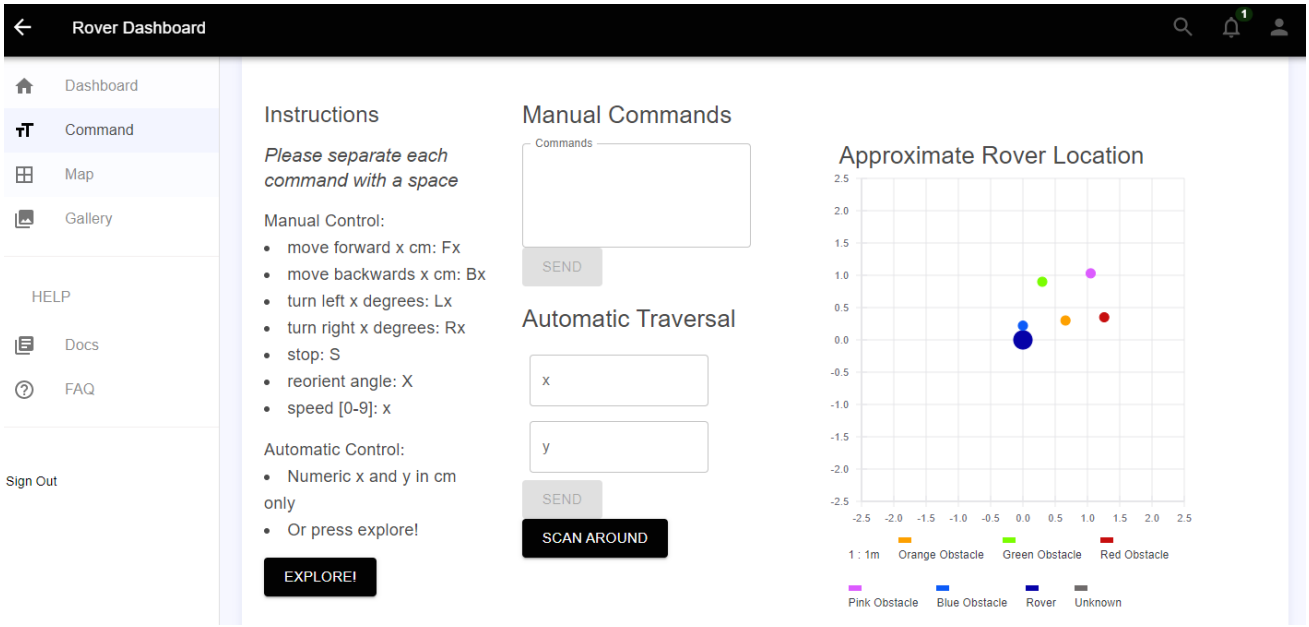


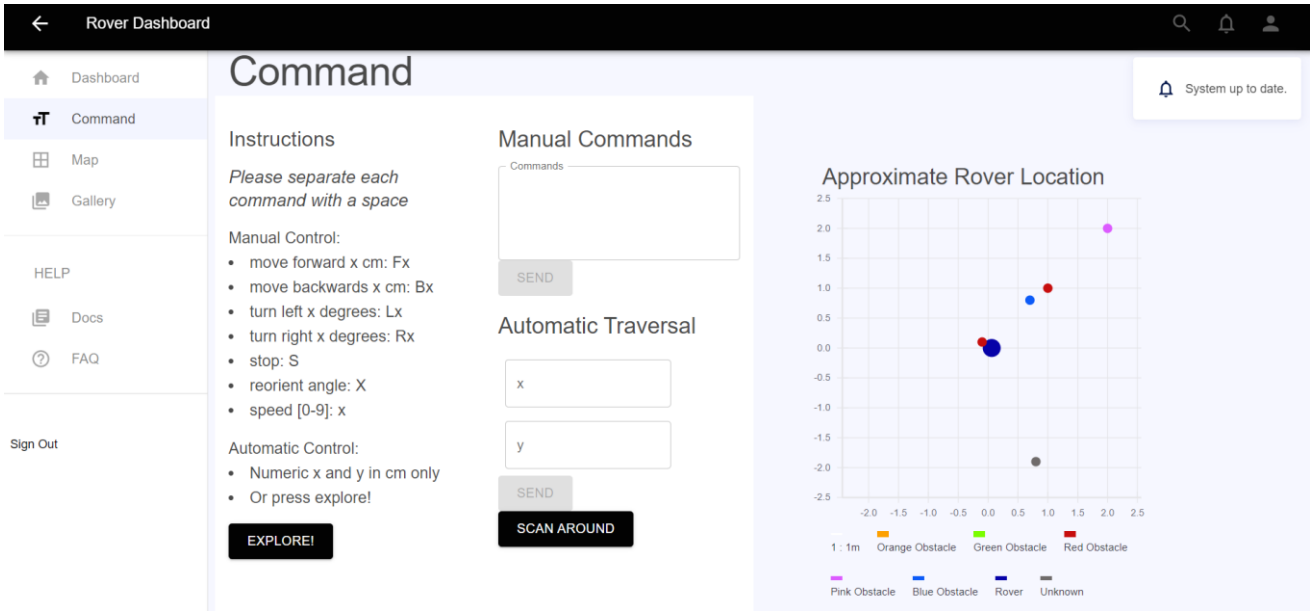


As shown in the subsequent views, the header is always in view and the side bar is always accessible. Although not yet implemented, the search function could be used to get to specific features. Notifications regarding rover status and obstacles come from the ESP32.

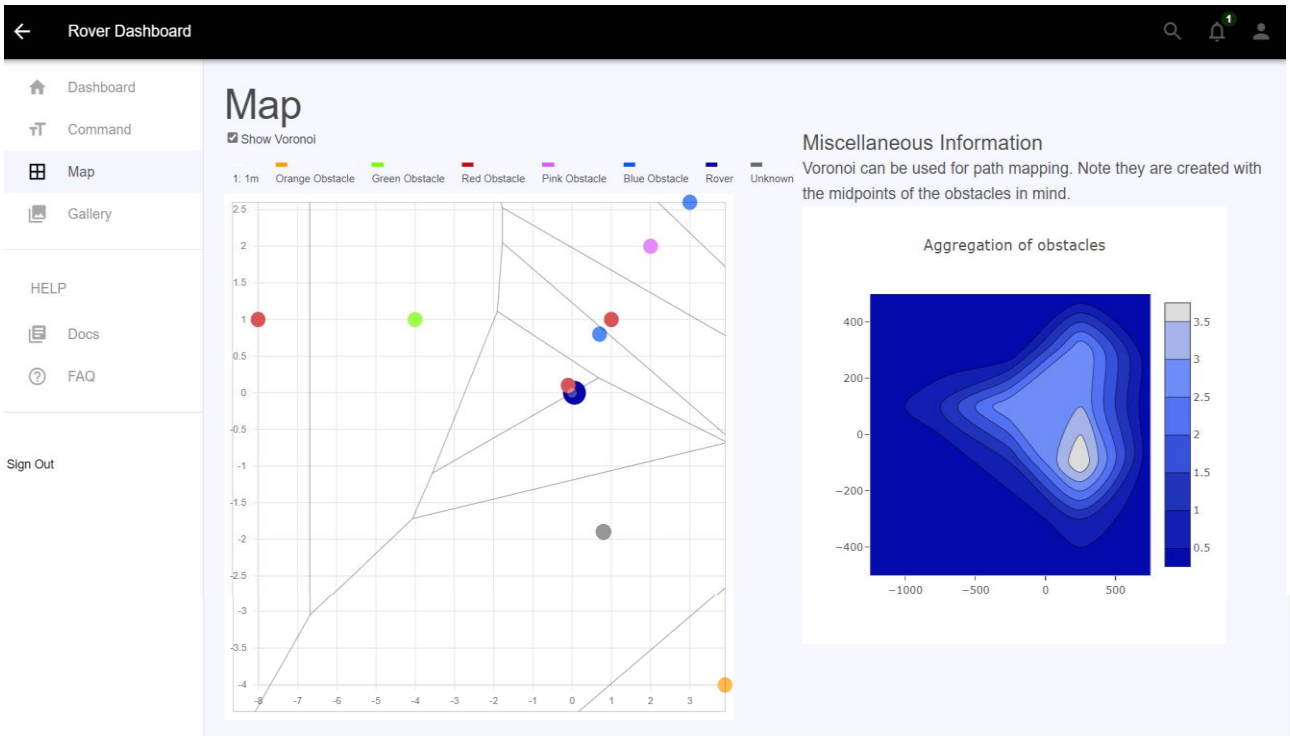


The **command** page shows instructions and rover location within a current square. The rover is centred in the map and real time movement and obstacles encountered are shown. It can be seen that on different browsers the widget renders differently; the first one is in Chrome, with poorer latency, and the second image comes from Microsoft edge. Again, the dashboard and sidebar are always accessible.

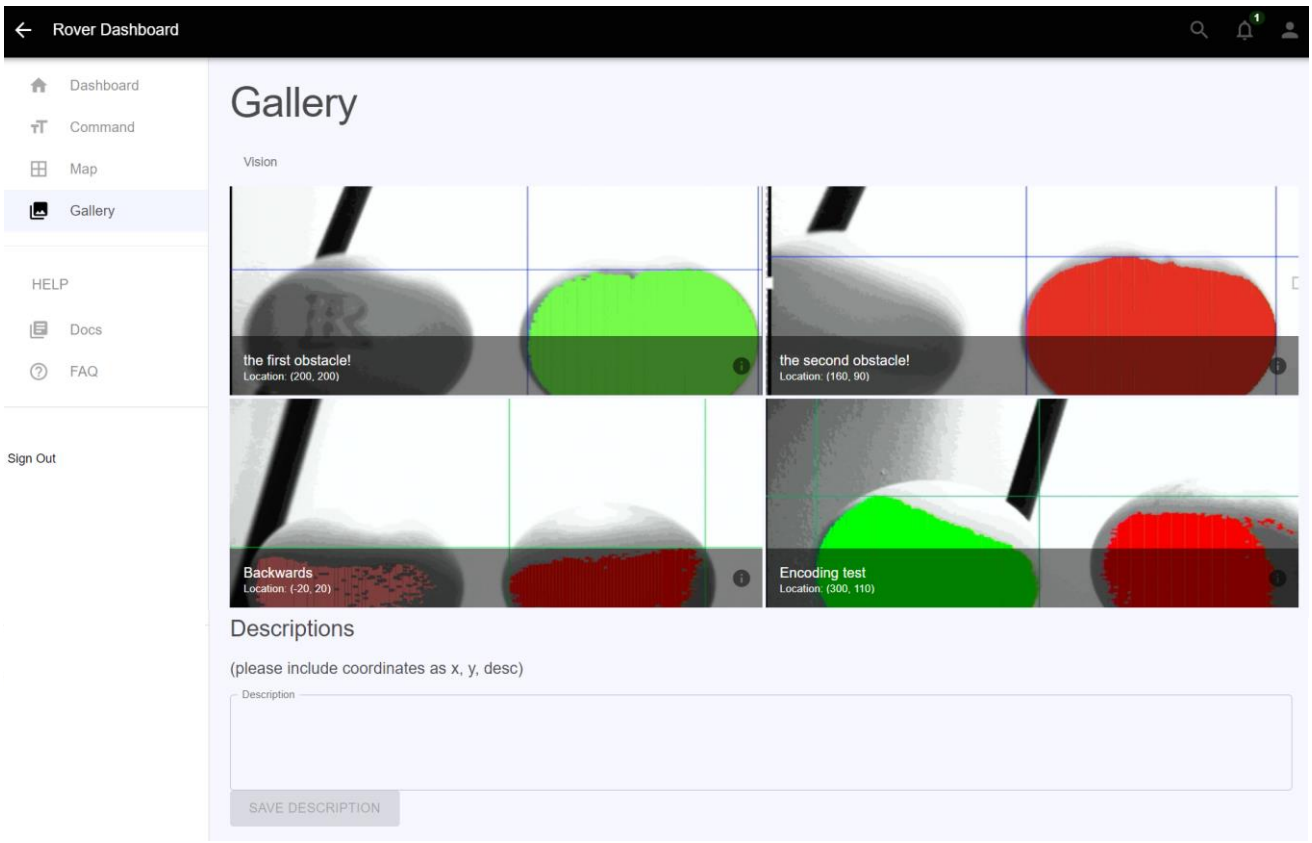




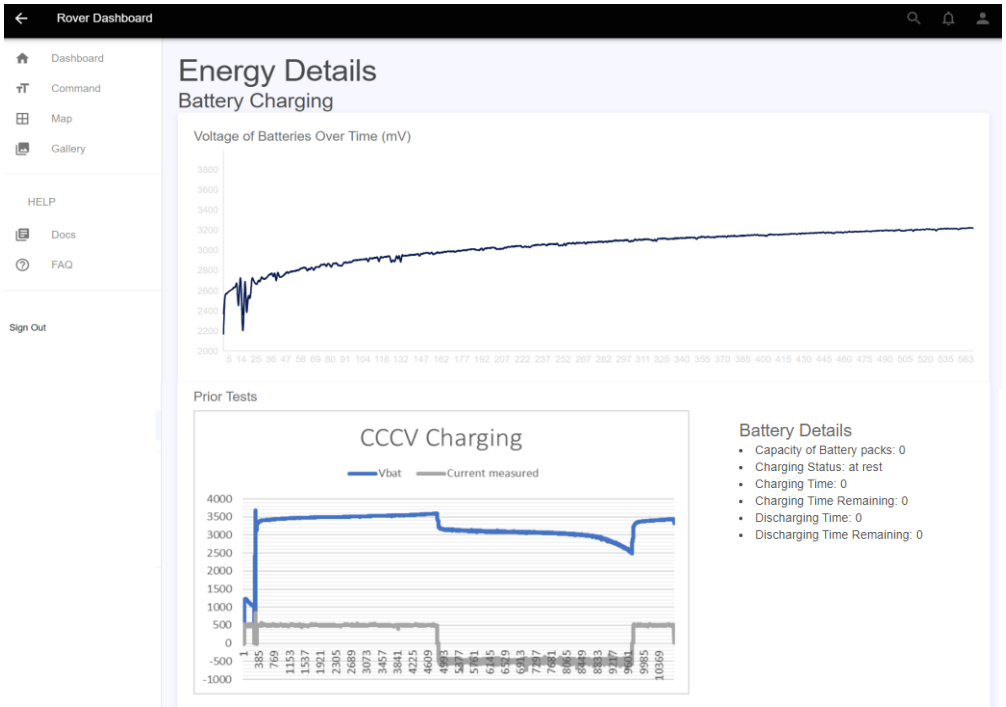
The **map** shows the whole area covered, from the moment the rover turns on. This has greater scope than command and the scale conforms accordingly to fit into the square. There is also a contour plot to show where obstacles are concentrated.



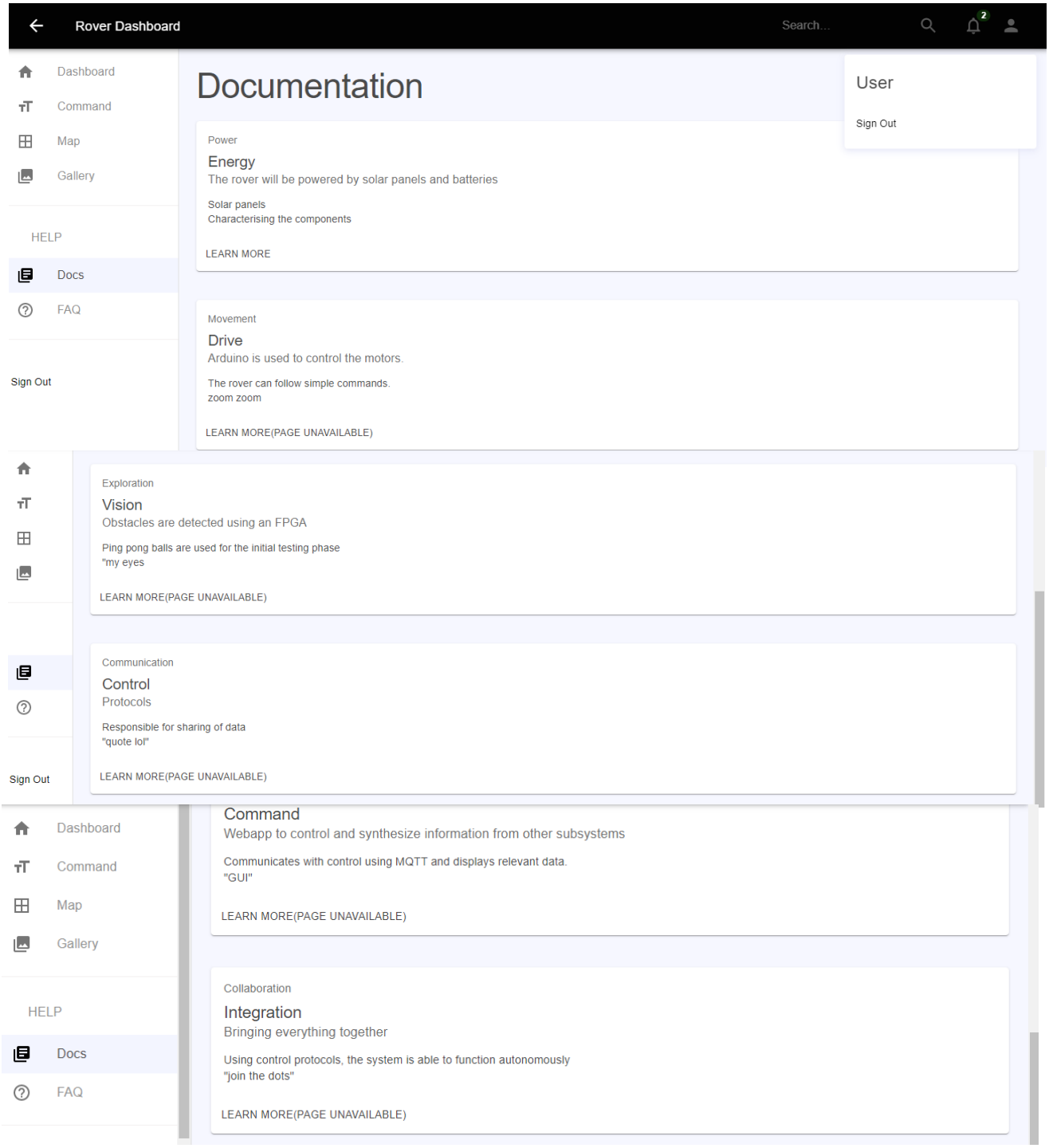
The **gallery** will show snapshots of objects seen whilst exploring. The images would be sent when obstacles are encountered, as a bytestream to be parsed and uploaded to S3, though image recovery has not yet been implemented.



If the user wants a more in depth view of the charging process, they can proceed to the **energy details** page, with a real time graph of the charging process.



The **Header** also allows a user to sign out. The **Documentation** page exists on the deployed website, providing an overview of the project. Again the **sidebar** can reduce. In future, the notifications and user details can be more in depth; t here are libraries available to manage such functions.



To the right are the routes available for the website, using react-router-dom :

```
import {
  Route,
  Switch,
  withRouter,
} from "react-router-dom";
```

```
<Switch>
  <Route path="/app/dashboard" component={Dashboard} />
  <Route path="/app/command" component={Command} />
  <Route path="/app/maps" component={Maps} />
  <Route path="/app/docs" component={Docs} />
  <Route path="/app/FAQ" component={Faq} />
  <Route path="/app/gallery" component={Gallery} />
  <Route path="/app/energy" component={Energy} />
</Switch>
```

Appendix-VIII

12.8 Testing Methods

Serial Monitor: Drive, Energy, Control

esp32_sketch

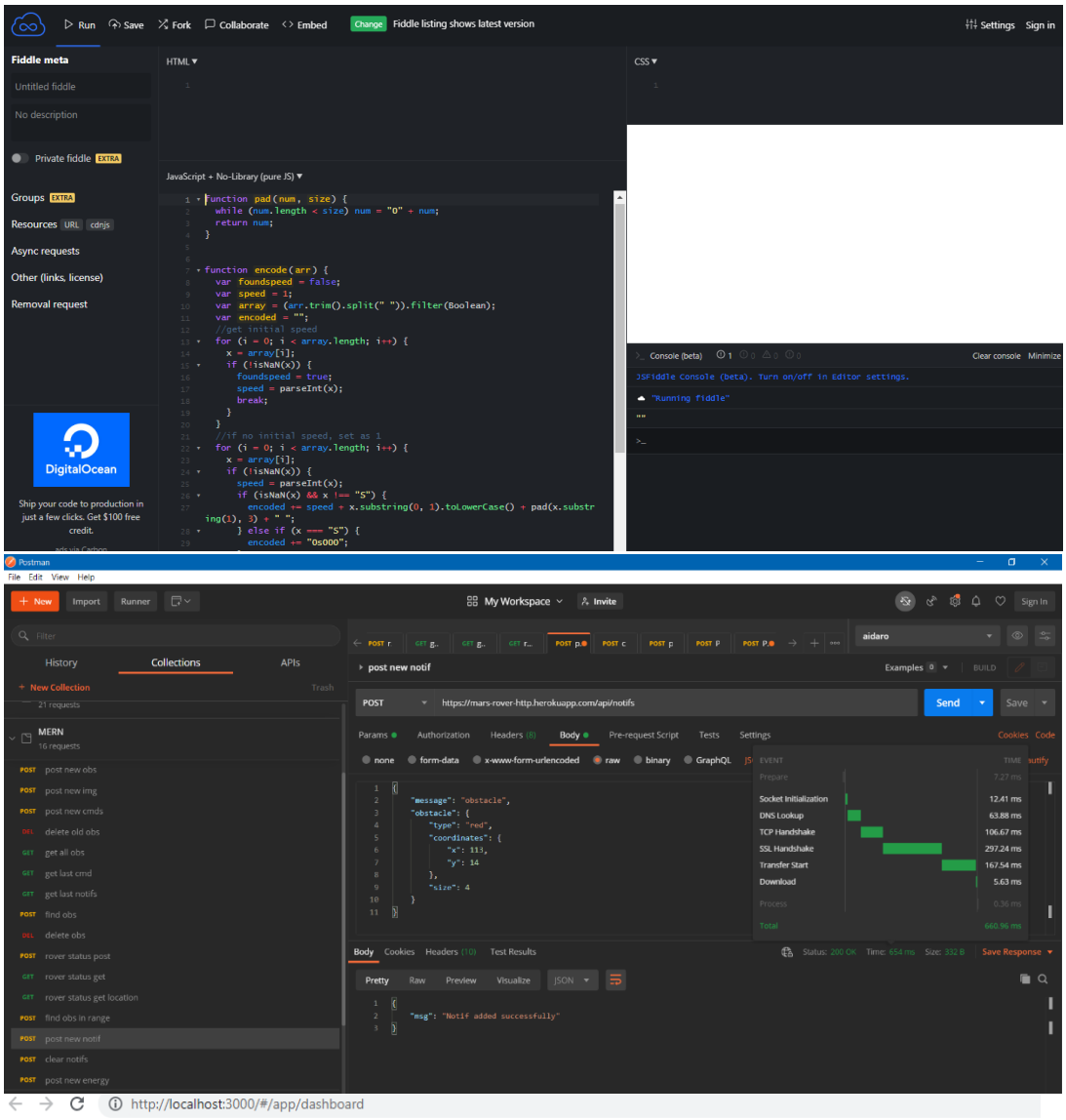
```
#define RXD2 16
#define TXD2 17

void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  Serial2.begin(115200, SERIAL_8N1, RXD2, TXD2);
}

void loop() {
  //Serial.println(Serial2.readString());
  //Serial2.print("000f10");
  delay(5000);
  Serial2.print("7f10");
  delay(5000);
  Serial2.print("7b10");
  delay(5000);
  Serial2.print("6r90");
  delay(5000);
  Serial2.print("7190");
  delay(5000);
}
```

```
instr = 0D000
Distance_x = -39
Distance_y = 1

X = -10
Y = 10
X_active = -10
Y_active = 10
enter instruction
********** 20 391 (0,0)

instr = 0D000
Distance_x = -39
Distance_y = 1

X = -10
Y = 10
X_active = -10
Y_active = 10
enter instruction
********** 20 391 (0,0)
```

Nios2 Terminal: Vision

```
00524242 00d500f6 01e701c4
00524242 00d500f3 01e601c3
00524242 00d600f1 01e601c3
00524242 00d500f1 01e701c3
00524242 00d500f5 01e601c3
00524242 00d500fe 01e701c3
00524242 00d500f4 01e701c3
00524242 00d500fb 01e701c4
00524242 00d600fe 01e701c4
00524242 00d500f7 01e701c3
00524242 00d500f6 01e601c4
00524242 00d500f5 01e701c4
00524242 00d500f3 01e601c3
00524242 00d500fc 01e701c2
00524242 00d500f4 01e701c3
00524242 00d400fa 01e601c2
00524242 00d500ec 01e701c3
00524242 00d500f1 01e701c3
00524242 00d500f4 01e601c3
00524242 00d500ef 01e701c3
00524242 00d500fa 01e601c3
```

JSfiddle, Postman: Command



Appendix-IX

11.9 Control Flow

The following code segment shows the collection of global variables for the rover. The use of global variables is to avoid having to pass variables around in a confusing manner in the various functions.

```
long last_pub = 0;  //last publication time
long buffer_secs_left = COOLDOWN_TIME; // this acts as a "ignore object detected" time if new command comes in. Imagine a
int pub_already = 0;

String remaining_commands_excl_running_command, remaining_commands_incl_running_command;
//wait = control waits while Drive is executing
//active = control will now try to send a instruction to Drive
//poll = control attempts to poll Drive
//reply = control is waiting for Drive's reply
enum CONTROL_DRIVE_STATE {CONTROL_WAIT = 0, CONTROL_ACTIVE = 1, CONTROL_POLL = 2, CONTROL_REPLY = 3};
CONTROL_DRIVE_STATE ctrl_drv_state = CONTROL_ACTIVE;
int poll_timer = 0;

int drive_x = 0;
int drive_y = 0;
float drive_speed = 0;
float drive_distance_travelled = 0;
float drive_angle = 0;

int vis_xlength, vis_ylength; // These are the x and y length of the corners of the rectangle of the ball bounding box
char ballcolour; // The ball colour as a char, can be either r, g, o, p or b
int vision_ball_area; // The area in pixels of the bounding box, calculated in the function getVision()
int object_detected = 0; // A flag that is set in the main loop() if an object is considered to be detected.
int obj_x_coord = 0, obj_y_coord = 0;

int energy_voltage = 3300, energy_current = 50, energy_battery = 100, energy_light = 69;
```

The following is the source code where all the relevant flags and statuses are reset upon receipt of a new instruction from Command:

```
if (strcmp(topic, "rover/cmds") == 0)
{
  const char* cmdlst = jsondoc["commands"];
  remaining_commands_excl_running_command = String(cmdlst);
  remaining_commands_incl_running_command = remaining_commands_excl_running_command;
  // F<number> = forward, <number> units of distance
  // B<number> = backward, <number> units of distance
  // L<angle> = turn left, <angle> degrees
  // R<angle> = turn right, <angle> degrees
  vis_xlength = 0; // reset vision
  vis_ylength = 0; // reset vision
  vision_ball_area = 0; // reset vision
  obj_x_coord = 0; // reset vision
  obj_y_coord = 0; // reset vision
  buffer_secs_left = COOLDOWN_TIME; // reset buffer time
  object_detected = 0; // reset object detection flag
  pub_already = 0; // reset publication flag
  ctrl_drv_state = CONTROL_ACTIVE; // reset control drive state so the controlRover() can do its thing
}
```

The following is code where we unpacked and re-assembled the raw bytes received from the Vision module. Notice that we first send a 'v' character to the Vision module first to query for the reply.

```
void getVision()
{
  Serial1.print("v");
  while (Serial1.available()!=0)
  {
    // change code to handle bitstream instead of chars in unicode.
    unsigned char data[12];
    Serial1.readBytes(data, 12);
    unsigned int temp1 = 0, temp2 = 0;
    temp1 = data[5];
    temp2 = data[4];
    vis_xlength = (temp1 << 8) | temp2; //width
    temp1 = data[7];
    temp2 = data[6];
    vis_ylength = (temp1 << 8) | temp2; //height

    Serial.print("got xlen =");
    Serial.println(vis_xlength);
    Serial.print("got ylen =");
    Serial.println(vis_ylength);
    Serial.print("got colour =");
    Serial.println((int)data[8]);
```

The following is a segment of code used to convert the size of the ball into a distance measure of how far the ball is from the rover. The, based on the x and y coordinates of the rover, as week as the orientation of the rover, it is then possible to determine the location of the object.

```
vision_ball_area = vis_xlength * vis_ylength;

int dist = convArea2Dist(vision_ball_area);

if (-45 < drive_angle && drive_angle <= 45) //facing up, positive y
{
  obj_x_coord = drive_x;
  obj_y_coord = drive_y + dist;
}
else if (45 < drive_angle && drive_angle <= 135) //facing left, negative x
{
  obj_y_coord = drive_y;
  obj_x_coord = drive_x - dist;
}
else if (-135 < drive_angle && drive_angle <= -45) //facing right, positive x
{
  obj_y_coord = drive_y;
  obj_x_coord = drive_x + dist;
}
else if (135 < drive_angle || drive_angle <= -135) //facing down, negative y
{
  obj_x_coord = drive_x;
  obj_y_coord = drive_y - dist;
}
```

An overview of the if/else states of the Drive control is shown in the following figure. The logical flow of the program is just as explained in the flowchart in section 5.4.

```
411 >  if (object_detected) ···
423 >  else if (ctrl_drv_state == CONTROL_ACTIVE) ···
445 >  else if (ctrl_drv_state == CONTROL_WAIT) ···
460 >  else if (ctrl_drv_state == CONTROL_POLL) ···
467 >  else if (ctrl_drv_state == CONTROL_REPLY) ···
```

The following code is the code for the main loop of the program.

```
client.loop(); // calling this function hands control to the MQTT client to do its thing. If this is not called, the MQTT client will not be
able to do anything.
// If there are no incoming messages, this function will return immediately. If got message, then for each message, the callback will be
called once to handle it. i.e. if got 5 messages, callback will be caled 5 times


long now = millis(); // get the "current time". millis gives the number of milliseconds elapsed since the start of the program
if (now - last_pub > 1000)
{
  last_pub = now;
  if(buffer_secs_left > 0) // If buffer_secs_left is positive and nonzero
    buffer_secs_left--; //then decrement it.

  getVision(); // Get the 4 coords, ball colour, and area of ball bounding box in term of pixels, and write into global varaibles
  // if no more buffer time left, AND the ball bounding box area is bigger than the threshold, consider object detected.
  if (buffer_secs_left == 0 &&  OBJECT_SIZE_THRESHOLD_MIN < vision_ball_area && vision_ball_area < OBJECT_SIZE_THRESHOLD_MAX)
    object_detected = 1;
  controlRover(); // Control the rover. This function stops the Drive immediately if object_detected == 1. otherwise, it just runs as normal
  if (object_detected)
  {
    if (remaining_commands_incl_running_command.length() > 0)
      pubNotif();
    else
      pubObstacle();
```

Appendix-X

11.10 Integration Testing

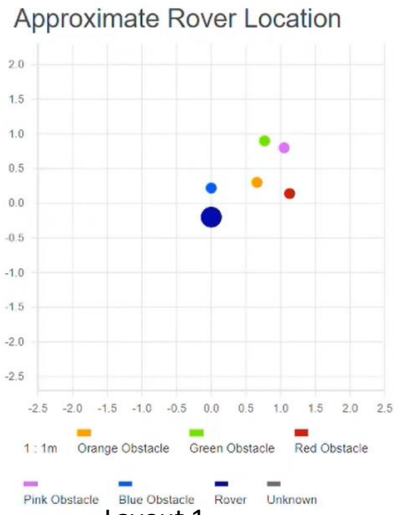
General Commands

Instructions	Time Delay/s	Repeat Duration/s	Dist from expected Position	Behavioural Notes
6f010	0.5	600	1cm	Mostly accurate except for oscillations if strong light on red light
5r090	0.5	600	2°	Generally accurate
7l090	0.5	600	2°	Generally accurate
4b010	0.5	600	0.5cm	Generally accurate
6r045	0.5	600	0.5°	Generally accurate
5l030	0.5	600	2.5°	Generally accurate

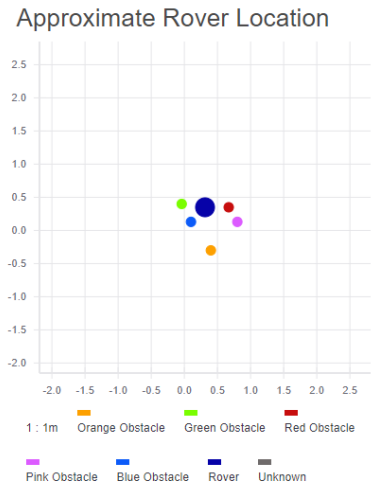
Scenarios

1. A) 360° scan for obstacles to get fixed obstacles' position on the map, followed by B) repositioning where a route to a coordinate is calculated for the rover whilst avoiding the known obstacles.
2. Unknown obstacles so the rover is initiated with a string of random commands and will self-navigate around the obstacles.

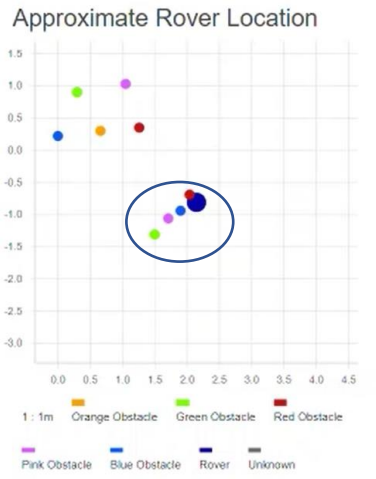
Scenarios		Distance from expected pos (including re-routing)/cm			Average command and display (Edge) time delay/s			Total Travel Time/s		
		T 1	T 2	T3	T 1	T 2	T3	T 1	T 2	T3
Layout 1	1A	N/A	N/A	N/A	1.492	1.092	1.337	101.267	99.595	99.8
	1B	1.04	0.86	0.53	1.499	1.492	1.531	30.2	32.5	37.4
	2	1.06	0.99	0.90	1.572	1.631	1.957	204.5	201.7	208.3
Layout 2	1A	N/A	N/A	N/A	0.991	1.231	1.071	101.3	98.4	99.5
	1B	0.95	1.07	0.98	1.274	1.598	1.567	55.7	52.6	53.4
	2	2.08	1.02	1.00	1.534	1.752	1.856	249.5	251.4	255.1
Layout 3	1A	N/A	N/A	N/A	1.498	1.107	0.985	98.4	97.6	99.7
	1B	0.91	0.92	0.87	1.617	1.546	1.103	70.6	71.4	69.8
	2	0.54	1.07	1.11	1.765	1.819	1.213	150.6	151.4	153.9



Layout 1



Layout 2



Layout 3