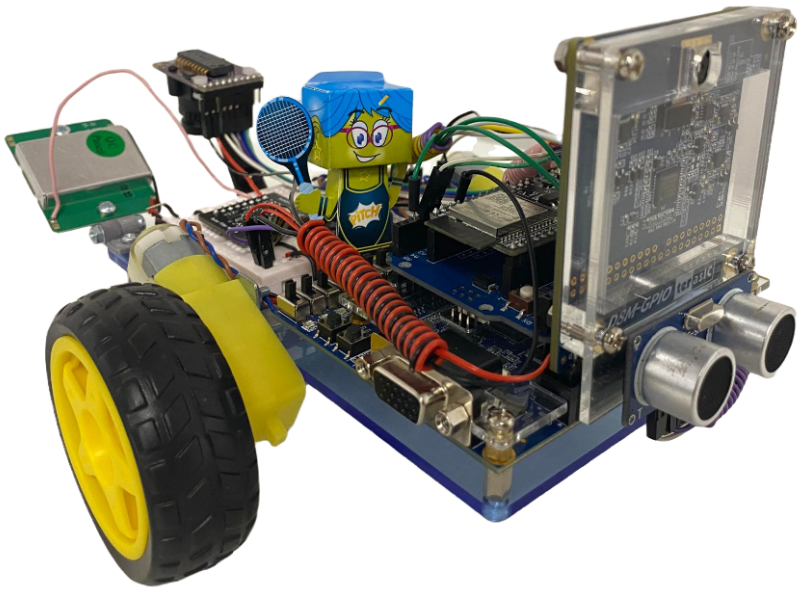
Imperial College London

Instability Mars Rover - Team 15

Petar Barakov, EEE - 01865450 Vladimir Marinov, EEE – 01956293
Alexandra Neagu, EIE – 01843748 Ishaan Reni, EIE – 01906148
Eleftheria Safarika, EEE – 01873841 Joachim Sand, EIE – 01876217
Scott Vandenberghe, EIE – 01858847

June 2022

Word Count: 5624



1 Abstract

This project's goal was to build a Mars rover with the aim of mapping an undiscovered arena, identifying and locating the aliens, obstacles, and underground structures in its course. The rover must be able to avoid these obstacles and get back on track without getting derailed from its course. Live information regarding the position and status of the rover needs to be transmitted securely, while all data can be visualised on a website. This web-page also allows the users to manually send movement commands for the rover to complete. The project also involved a separate power station that the rover returns to when the battery requires charging.

Command Github
Vision Github
ESP Github
Radar Filter PCB Github
Energy Github

Contents

2 Introduction

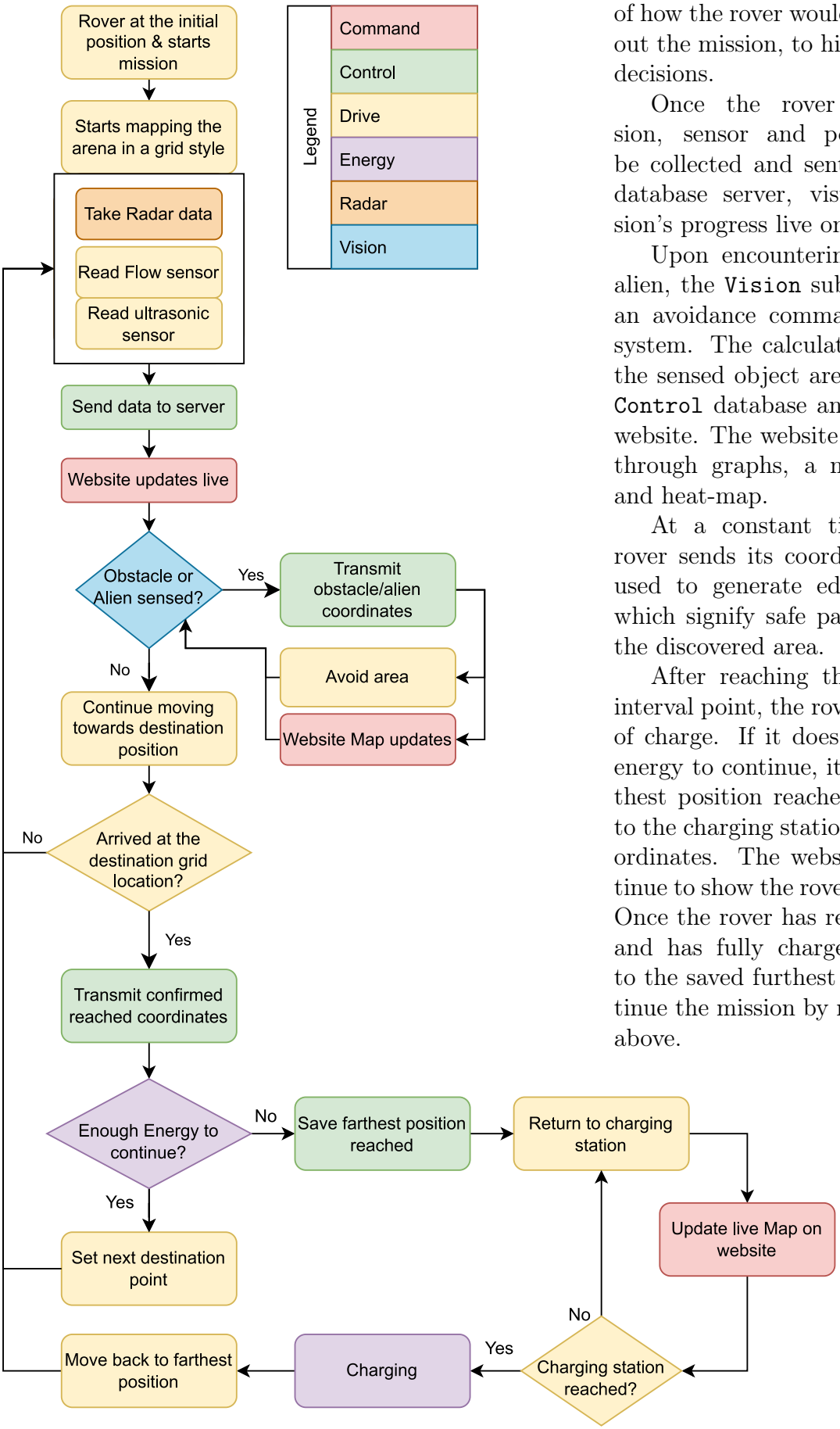


Figure 1: High Level Behaviour Flowchart

The adjacent figure presents a trial run of how the rover would behave throughout the mission, to highlight the design decisions.

Once the rover starts its mission, sensor and position data will be collected and sent regularly to the database server, visualising the mission's progress live on the website.

Upon encountering an obstacle or alien, the `Vision` subsystem transmits an avoidance command to the `Drive` system. The calculated coordinates of the sensed object are forwarded to the `Control` database and to the `Command` website. The website presents the data through graphs, a map, camera feed and heat-map.

At a constant time interval, the rover sends its coordinates, which are used to generate edges on the map, which signify safe paths for traversing the discovered area.

After reaching the aimed furthest interval point, the rover checks its state of charge. If it does not have enough energy to continue, it will save the furthest position reached and will return to the charging station at the initial coordinates. The website map will continue to show the rover's live behaviour. Once the rover has reached the station and has fully charged, it will return to the saved furthest position and continue the mission by repeating all steps above.

3 Team management and Integration

The figure below presents how the project sections were divided and integrated. Also included are the main technologies and programming languages used for each division, hence presenting the complexity of integration.

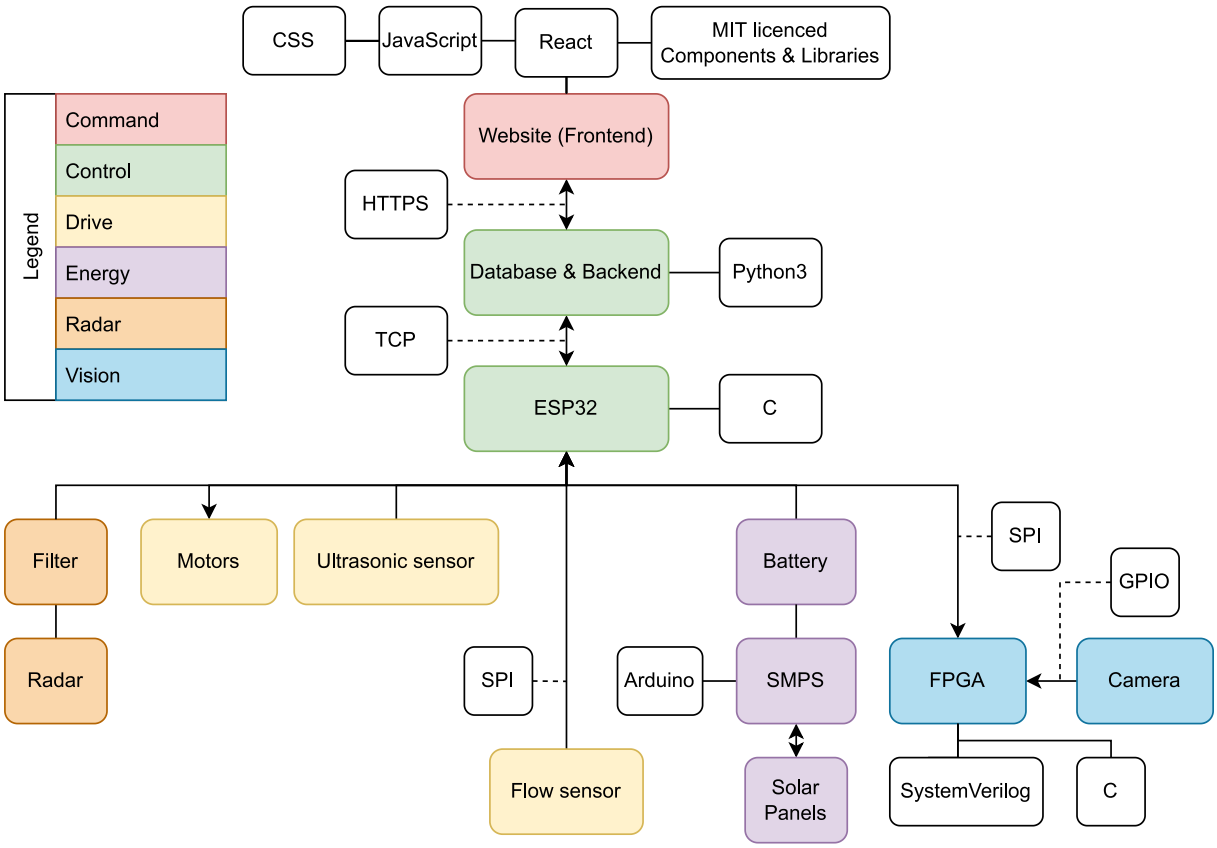


Figure 2: Integration UML Diagram

Considering the wide range of skills needed to complete the project, roles were distributed to maximise efficiency. So, some members worked on multiple sub-systems to simplify the complexity of the final integration.

Control (Database) and Command – Alexandra
Remote Control Algorithms – Ishaan, Vladimir
Energy – Eleftheria, Petar, Vladimir
Radar – Eleftheria, Vladimir
Drive – Vladimir
Vision – Ishaan, Joachim, Scott
Control (ESP32 setup) – Joachim, Vladimir

A Gantt chart of the schedule management and timing goals can be found in the Appendix.

The integration of subsystems was done simultaneously with the development of programs. This reduced the risk of unforeseen issues occurring during the final stages of development. When integrating two subsystems together, a bottom-up testing approach was employed.

4 Vision

The goal of the vision subsystem is to identify the location of aliens (coloured balls) and obstacles (Buren stripes of black and white) while the rover is navigating the arena. Raw pixel data of the Terasic D8M-GPIO camera is fed into a streaming video pipeline on the MAX10 FPGA where pixel-by-pixel processing is applied. Due to the significant throughput of camera data (0.44 GBps)

most of this processing must be performed with logic synthesized from Verilog on the fabric of the FPGA, although some post-processing can be performed on the ESP32 micro-controller.

Creating such processing logic can be cumbersome, as compiling takes about 5 minutes. It was, as a result, decided to initially prototype processing methods in C rather than Verilog.

The processing pipeline was written in SystemVerilog. The choice of SystemVerilog over Verilog was driven by three factors: it is safer to work with, more familiar to the team, and it allows passing multi-dimensional arrays from one module to another. This was particularly useful in the implementation of some modules described later.

4.1 Recognising Aliens

To perform more sophisticated image processing, it is useful to modify the value of a pixel, not only as a function of its own value, but also that of its neighbours. The standard way to do this is with a kernel; a matrix of odd height and width, such that pixels of the output image are computed based on the values of neighbouring pixels in the original image.

Implementing kernel operations requires buffering pixels. The original design choice was to use a single ring buffer which could buffer all the rows required for kernel operations. Unfortunately, the logic required to index such a large unpacked array proved area-inefficient. To resolve this issue, two types of shift registers were implemented. The first is a first-in, first-out(FIFO) queue whose contents are not accessible (this requires significantly less wiring than always having access to all buffered pixels). Pixels that are not needed in the current kernel are buffered but not accessed in the current cycle. Needed pixels are buffered in an accessible shift register.

A preliminary processing step applied to the image's RGB data is a Gaussian Blur kernel which mitigates the effect of noise. Using a 5x5 Gaussian Blur approximate kernel was originally planned. However, while this kernel was implemented successfully, given area challenges, it was decided to use a 1x9 kernel instead, as shown below. This performed similarly to 5x5 Gaussian Blur (located in appendix).

$$G = \frac{1}{256}\begin{bmatrix} 1 & 8 & 28 & 56 & 70 & 56 & 28 & 8 & 1 \end{bmatrix}$$

To attenuate shadows and reflections in object detection, it was desirable to convert each pixel from the RGB to hue saturation value (HSV). Working exclusively with hue should eliminate a large amount of noise in the image and leave balls with the same hue value on their surface. In practice, both saturation and value proved invaluable when tuning the thresholds.

To minimise the system's critical path, it was decided to pipeline this conversion step into a 16-stage system. The equations of RGB to HSV conversion are the following:[1]

$$\Delta = max(r,g,b) - min(r,g,b) \qquad denom = \Delta \times 6 \text{ if } \Delta > 0 \qquad denom = 6 \text{ if } \Delta <= 6 \quad (1)$$

$$H := 255 \times \frac{g-b}{denom} \qquad H := 255 \times \frac{b-r}{denom} + 85 \qquad H := 255 \times \frac{r-g}{denom} + 170 \quad (2)$$

$$\text{if } r \text{ is } max \qquad \text{if } g \text{ is } max \qquad \text{if } b \text{ is } max \quad (3)$$

Once RGB data was converted to HSV, pixels that matched the colour of a ball (thresholds obtained through manual tuning) were identified. Pixels that did not fall into any of the thresholds were set to black.

The HSV conversion and subsequent thresholding was found to generate patches of colour not belonging to aliens. However, these patches were significantly smaller than the aliens. A custom kernel titled 'Modal Kernel' was designed and prototyped in C then implemented in Verilog.

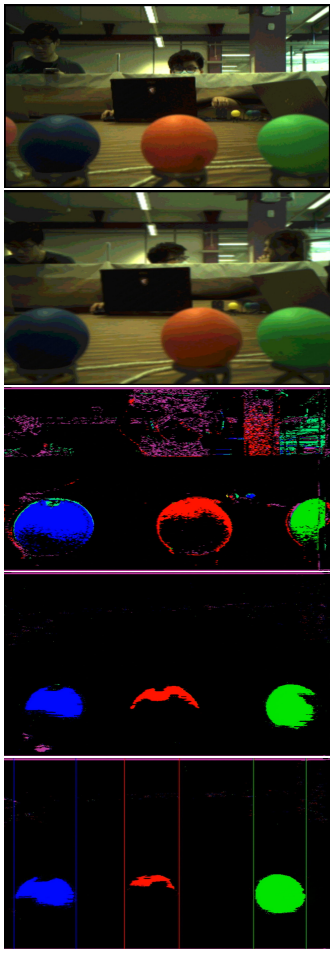


Figure 3: Cumulative from the top: unfiltered, Gaussian blur, HSV thresholding, Modal filter, bounding columns.

The Modal Kernel is a 5x16 kernel that sets each pixel's value to the most common hue in the surrounding 79 pixels, provided at least 40 of them have the same hue. Using the thresholded image data, undesirable small elements of colour were eliminated. Figure 3 illustrates the impressive improvements the Modal Kernel imposes on an image littered with noise and undesirable patches of colour.

The last step in the image processing pipeline for the detection of aliens is drawing bounding lines. Extreme x-axis positions are defined for each ball colour in each frame. The distance between these pixels of identical colour can be assumed to be the size of the ball on the screen and the distance can be inferred. Currently, distance is estimated using a linear relationship to object width but further investigation is needed. Distance and size estimation for balls and obstacles as well as the calculation of Cartesian coordinates are handled on the ESP32.

4.2 Recognising Obstacles

Obstacles on the arena are characterised by uniformly-spaced black and white Buren stripes. This means they have lines of higher contrast than any other object in the arena.

After the image data is converted to grayscale, it is passed through a 1x3 Sobel filter which highlights edges. This kernel returns the absolute value of the difference between the pixels on the left and right of the current centre of the kernel.

Once edge detection has been performed, the image data is thresholded. All pixels with a grayscale value of less than 20 are discarded while the remaining pixels are considered. Post edge detection, bounding boxes can be drawn and used to give rough idea of where the obstacle lies on the map (as can be seen by the red lines on Figure 4).

To identify Buren lines on the obstacle, the image is parsed through on a single row in the middle of the image. Edge pixels found on this row and preceded by 4 non-edge pixels mark the beginning of a new column on the obstacle; this avoids detecting the same edge multiple times. Locations of these new columns are marked with green lines in Figure 4. By performing further processing on the ESP32, the edge of an obstacle can be detected by an increase in the spatial frequency of Buren lines. Distance to the obstacle is estimated by observing the greatest distance between two subsequent Buren lines (assumed to be closest to the camera) on the obstacle.

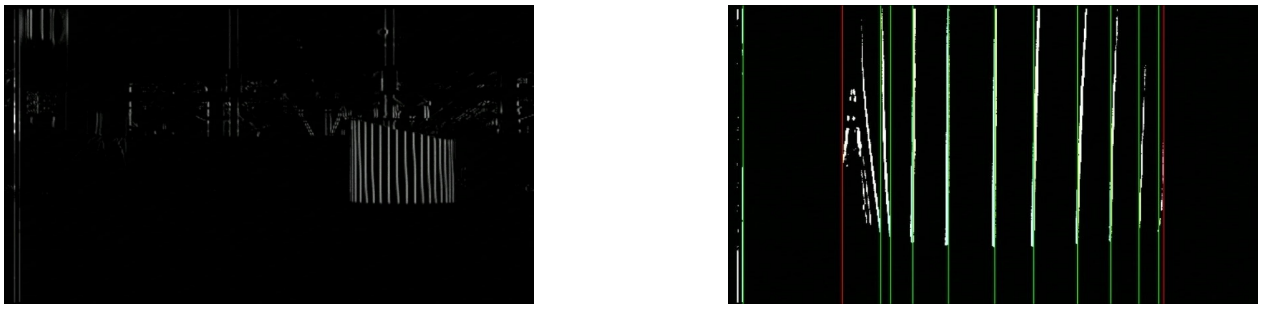


Figure 4: Leftmost image shows the result of Sobel filtering, while the image on the right shows the final result after Sobel filtering, thresholding and spatial frequency treatment.

4.3 Final Pipeline

Pipelining resulted in a strong timing performance of the Vision system. Quartus timing analysis (at 1.2V and 85°C) computed the restricted maximum frequency for the object identification section of the vision pipeline to be 102.93 MHz (faster than the bottleneck clock of 74.67 MHz of the MIPI Serial Interface). In terms of area, the Vision pipeline takes up 16206 logic elements (33% of the MAX10 FPGA), 9121 registers and 168 kB of M9K memory (80%).

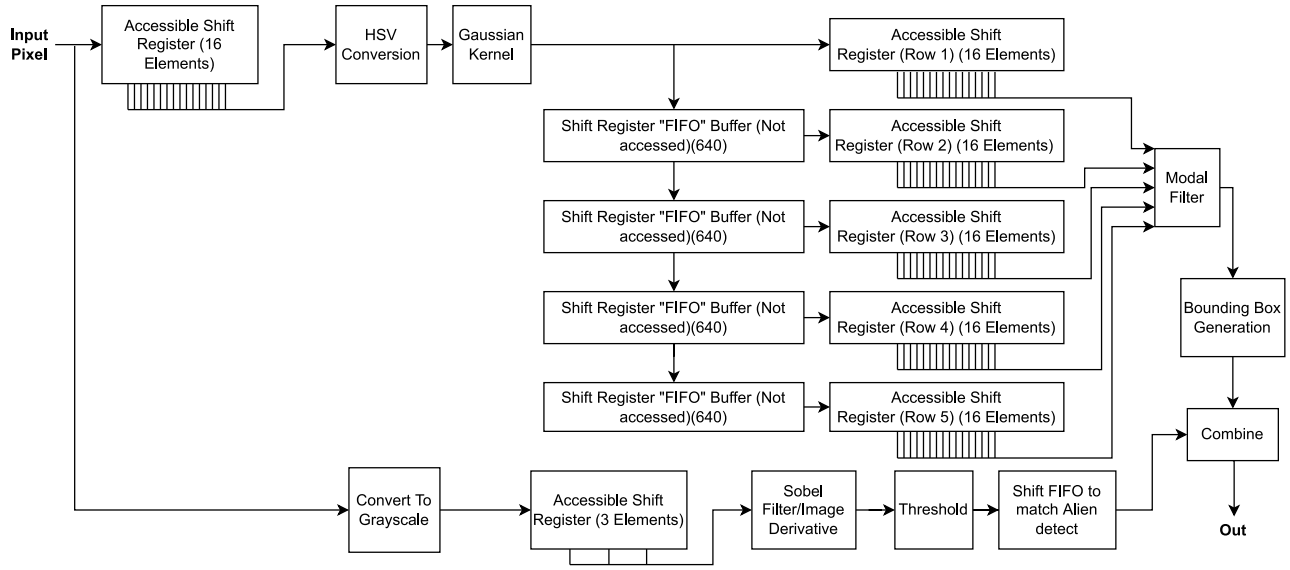


Figure 5: Kernel Buffer Diagram

The communication with the ESP32 is done entirely via SPI, directly from the video processing pipeline in `SPI_SLAVE` module. This was a carefully considered choice - by bypassing the Nios II processor entirely, much greater speeds could be attained (roughly 15 mb/s with `SCLK` at 25 MHz). These speeds are enough to send over a downsampled live camera feed from the FPGA to the front-end website. Initially, it was attempted to add a frame buffer after the `EEE_IMGPROC.v` module such that image frames could be buffered for transmission to the ESP32 without interrupting the processing pipeline. However, due to lacking M9K memory (the Frame Buffer II IP Core requires 4 M9K blocks [2]), this was not possible. Instead, it was chosen to add a backpressure signal from the SPI controller to the input of `EEE_IMGPROC.v`, such that the entire pipeline can be stalled until the ESP32 has read the newest pixel. From the ESP32, the live camera feed is sent to the website through a persistent HTTP connection.

A state machine on both ends of the communication channel is used to control and manage the SPI connection.

5 Drive

The drive subsystem consists of two main components - motors and motor driver hardware, and an optical flow sensor. In combination, these two components allow the rover to move, while also maintaining an accurate estimate of its position.

Since wiring the hardware for the motors, motor drivers, and optical flow sensor was mostly trivial (following their respective datasheets), most work on this subsystem was software-related.

Very early in the project the decision was taken not to use the Arduino framework and environment on the ESP-32. Instead ESP-IDF was used - a lower level framework provided by Espressif. This meant relying on any of the sample code provided for both the motor driver and

the optical flow sensor was not an option. However, this allowed for much more low-level control over the software running on the microcontroller, which was preferred.

Thus, most of the work on this subsystem was writing software drivers for the motor driver and optical flow sensor.

5.1 Motor Driver

The hardware for moving the rover consists of two DC motors mounted on each side of the rover, and a motor driver PCB based on the TB6612FNG IC (a classic H-Bridge driver IC for DC motors) from Toshiba.

The communication between the ESP-32 and motor driver IC is very simplistic and requires just 3 signals per motor - two signals (IN1, IN2) are used to control the state of the motor (stop, rotate clockwise, rotate counter-clockwise), while a third PWM signal is used to control the speed of the motor.

On the ESP-32 side, the ESP-IDF framework was used to access the built-in GPIO and PWM hardware in the ESP-32. GPIO was used for the IN1 and IN2 signals, while the PWM hardware (LEDC in ESP-IDF) was used for motor speed control. Thus, by using built-in hardware capabilities of the ESP-32, the code remained short and straight-forward (mostly providing higher-level control functions, eg. `init_motor_driver()`, `move_forward()`, `rotate()` etc.).

5.2 Optical flow sensor

The optical flow sensor PCB is based on the ADNS-3080 IC from Avago. As for the motor driver, custom low-level code was written for reading the sensor, instead of relying on the sample Arduino-based libraries. The optical flow sensor uses a standard implementation of the SPI protocol, so interfacing it to the ESP-32 was relatively simple. Some unconventional timing constraints required slight adjustment of the way that the ESP-32 SPI hardware is used, but overall the code remained short and (arguably) much neater than the provided example library.

Again, higher-level functions were provided (`init_optical_flow()`, `read_optical_flow()`, etc) to ease the integration with the Control subsystem.

5.3 Ultrasonic sensor

An HC-SR04 ultrasonic sensor was added at the front of the rover. This is used as an emergency stop - if the sensor detects an object very close to the rover, the rover will come to a complete stop, and initiate its object avoidance routine. This will happen even if the object was not detected by the Vision system. The ultrasonic sensor communicates with the ESP-32 via the built-in RMT peripheral in the ESP. The software is thus kept relatively simple, mostly being ESP-IDF setup code for the RMT hardware.

5.4 PID Controller

A PID (proportional–integral–derivative) controller was implemented on the ESP-32 to allow for a repeatable and robust manoeuvring of the rover on the arena. The PID controller controls the speed and directions of the two motors independently, based on negative feedback from the optical flow sensor. This allows the rover to follow a straight line and recover from deviations (due to terrain or motor imperfections). The PID controller can also be used to accurately move the rover a specified distance forward and accurately rotate a specified amount of degrees. With a combination of the three types (lateral, forward, and rotational) of controllers mentioned above, the rover can move anywhere on the arena, and avoid any obstacle or aliens in its path.

The software implementation of the PID controller on the ESP-32 is based on the Backwards Euler discretization method (an industry standard approach to implementing a PID controller on a discrete-time system like a microcontroller). Two common variations of the PID algorithm, that help create a reliable system were also implemented:

Integrator anti-windup When a change in set-point (controller target) occurs, the integral term of the PID controller can accumulate a significantly large value, potentially causing the output to be saturated at its maximum value (100% duty cycle PWM in this case). This is called integrator wind-up and can lead to significant overshoots of the target.

This can be mitigated easily by saturating the integral term at well bellow the maximum controller output. The exact value of integrator saturation is determined empirically during testing.

Derivative on measurement In a discrete system, a set-point change occurs instantly (in one time-sample) which leads to a large discontinuity in the error term. This discontinuity has a very large derivative, which can cause the derivative term to temporarily become abnormally large. This can cause issues in the controller performance, and must be mitigated. The simplest solution is taking the derivative of the measurement (from the optical flow sensor in this case) rather than the derivative of the error term. This does not change the stability properties of the controller.

5.5 Higher Level Control Functionality

Code was written that combines the PID Controllers and Motor drivers to provide higher level Command and Control functionality, such as moving to a specific coordinate on the arena. For simple cases (e.g moving along the x-axis or y-axis), the rover calculates the angle it needs to rotate to and then simply moves the specified amount of distance in that direction. In most cases an easy approach as this is sufficient. In case an obstacle is detected during this process a simple three-point avoidance path is attempted automatically. In the unlikely event of encountering an obstacle, while avoiding another obstacle, the rover automatically switches its pathfinding algorithm from the simple one outlined above to A*, which is guaranteed to find a safe and short path to the target coordinates. Switching between those two modes happens automatically based on the conditions the rover encounters, and the amount of the arena it has already explored - if it has explored the full arena, the rover will only use A*.

6 Control (ESP32 setup)

6.1 WiFi and connection with back-end

One of the key design requirements for the communication with the back-end was that it should be asynchronous and non-blocking. If this was not the case, the communication could potentially prevent proper operation of other critical codepaths on the ESP32.

The connection to a local WiFi network is initialized with the `init_wifi()` function. The function initialises the ESP32 in station mode (STA), and once successfully connected initializes a TCP server that can accept requests from the back-end. The TCP server functions as a FreeRTOS task - it will asynchronously call `server_receive()` whenever a new TCP message is received. This is useful if the user wishes to override the default behaviour of the rover, e.g. for manual control. Additionally, the ESP32 can initiate a TCP message to the back-end by initiating the `tcp_client_task` FreeRTOS task which will send data to the back-end in a non-blocking fashion.

7 Control (Database setup) & Command

The rover was constructed to map the arena by itself by storing all necessary nodes (position, alien, obstacle) internally. However, for a successful simulation of a Mars Rover project, allowing the user to live-monitor and intervene when necessary. Therefore, the Control (Database setup) was combined with the Command sections, due to how strongly related they are for a full-stack development approach.

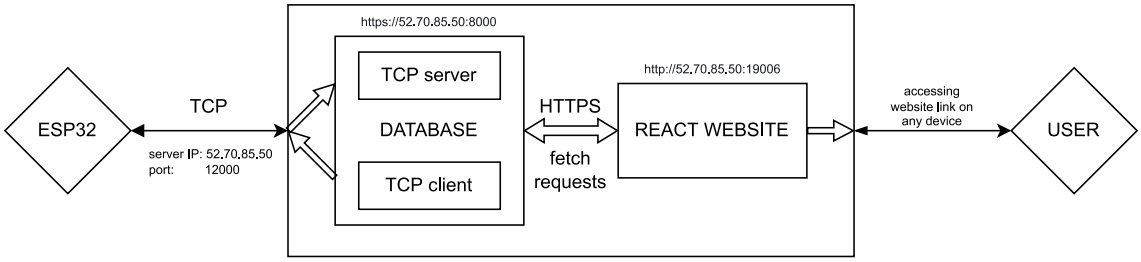


Figure 6: The User Interface and its related transmission protocols, accompanied by example IPs and links

7.1 Control (Database setup) - Back-end:

The Back-end development for the project includes the transmission and manipulation of data by the ESP32. All data mentioned in this section has a JSON format with specific default types explained later on.

To transmit the data between the rover and the database server, a TCP transmission was set up. This ensured a reliable transfer of data packets, as a persistent channel was maintained open throughout the period of the rover's mission. The TCP server was developed and tested so that it could be run on either a personal device or on an EC2 cloud device. The only requirement for changing the environment is to update the public IP address of the server in the ESP32 code to complete the TCP connection.

The server listening to the ESP32's requests only accepts some specific data formats, to avoid data mingling during storing. The accepted formats are:

1. JSON representing the start (left) and end (right) position nodes of the straight-line path just completed by the rover

   ```
   {"position":{"x":0,"y":0}, "position":{"x":100,"y":100}}
   ```

2. The alien coordinates sensed by the rover's camera

   ```
   a {"position":{"x":0,"y":0}}
   ```

3. Live coordinates for an accurate representation of the path completed by the rover, alongside live status data from all sensors and the motors' speeds

   ```
   l {"position":{"x":0,"y":0}} | { "squal":0, "motor_left":0,
   "motor_right":0, "orientation":90,"radar":0, ...}
   ```

4. Any ESP32 debugging messages that do not follow the expected JSON format and should not be forwarded to the web-server

   ```
   d {"debug message"}
   ```

Before storing the data received through TCP in the database, its values are converted from the given scale to the centimetre scale, preserving the accuracy as a floating-point value.

The received data has its structure reshaped in order to respect the format expected by the front-end interface. This includes introducing specific IDs for the coordinates, maintaining an order of nodes, but also generating other necessary data with the received information, such as creating edges connecting the position nodes.

With all data stored, the database makes its data valid for fetching. An HTTPS web-server was implemented to read all data JSON files and simply print it at specific URLs. For instance, the URL https://52.70.85.50:8000/motors presents the data related to the motor speeds throughout the whole mission.

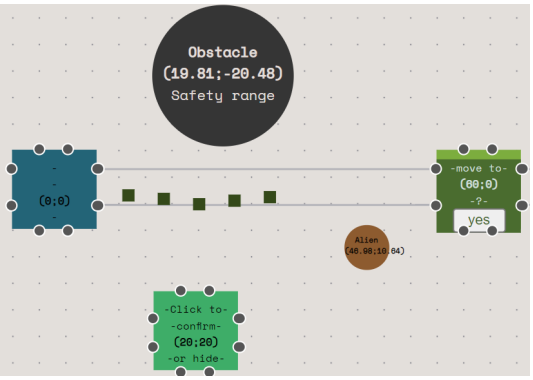
The decision to set up an HTTPS server was due to the fact that the React AJAX library demands some level of security, alongside enabling the Cross-Origin Resource Sharing (CORS) header in the transmission. Therefore, the team needed to generate and sign the keys and certificates.

```
alien.json
currentNode.json
edges.json
joystick.json
motors.json
moveto.json
nodes.json
obstacle.json
pathNode.json
radar.json
squal.json
status.json
ultrasonic.json
```

Figure 7: Data stored by the Database with respective URLs for the HTTPS webserver

7.2 Command - Front-end:

For the website, REACT was chosen as the main open-source JavaScript library, while the open-source platform EXPO was used to set up the web-app. This enabled versatility in programming, as there are a multitude of licensed libraries and components ready to be adapted to new projects on different platforms (Android, iOS, and the web). For front-end development, this versatility is crucial for a fast product delivery.




```
Colour legend:
    current node: blue,
    position Node: green,
    obstacle: black,
    alien: orange,
    virtual node: aqua,
    live path: dark green
```

Figure 8: All possible nodes found on the map screen

The website has three main tabs: the `Map Screen`, the `Control Panel`, and the `Status Panel`.

The `Map Screen` presents the positions of the rover and obstacles found while completing the discovery mission. The map contains position nodes, aliens, obstacles, live path nodes, and edges on the centimetre scale. For supervising and optimising the controller's behaviour, the map presents the live path of the rover, alongside the equally spaced position nodes. This allows us to visualise the actual path taken by the rover and optimise the drive controller accordingly.

On this tab, the user can interact with all components on the page, such as zooming in/out or moving the nodes around to get a better understanding of the area discovered. The options in the corner also allow the user to hide/show extra information on the map, such as connection edges and live paths. To remotely command the rover, the user can select a previously reached position node and request to move to that location, or can add a new virtual node with specific coordinates. In both situations the rover will follow the shortest path to that location by moving between previously visited nodes for as long as possible. It is advantageous to keep the rover moving between the discovered nodes because these areas are known to be safe. In the situation where the rover finished mapping the arena or when the user selected a node to go to, A* is the algorithm of choice to move between two predetermined nodes. A* is ideal for this task, as a breadth-first search would be excessive for plotting a path between two nodes where the mapping has already been completed.

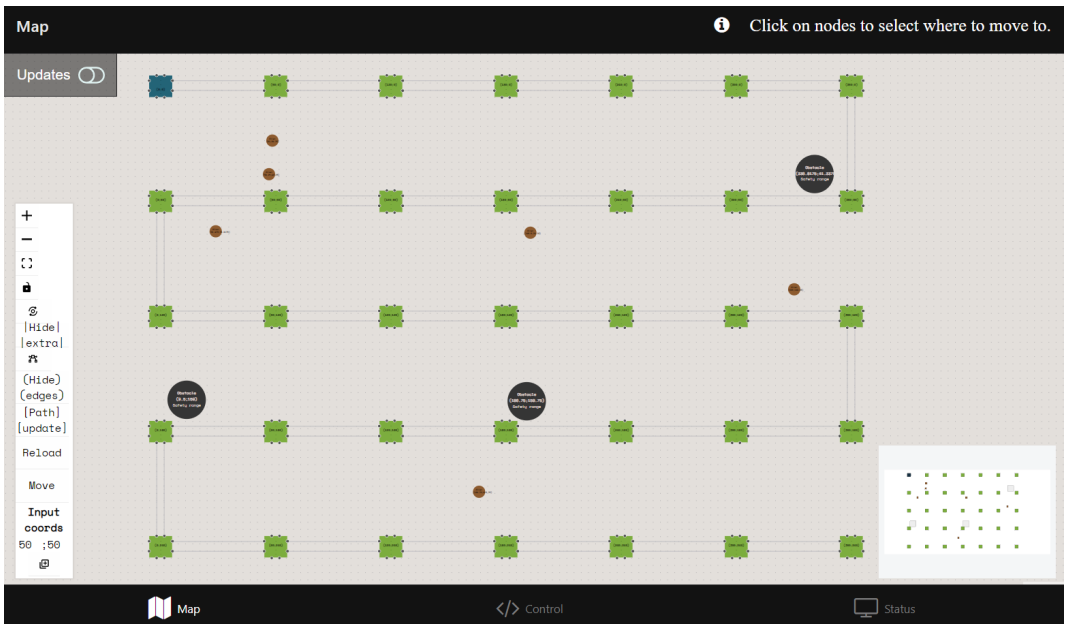


Figure 9: Map screen presenting position, alien and obstacle nodes

The `Control Screen` contains visuals of the rover's orientation, the motors' velocities, and allows the user to use the arrow keys for manual drive control. The user can interact with the graphs by zooming in/out or selecting the specific line graph to focus on. The manual drive is meant to be used only in emergencies when the rover requires calibration for its position or orientation. The screen also presents a navigation button to a modular tab that contains a live visualisation of the processed video feed generated by the FPGA and sent over by the ESP32. The live video feed is displayed using OpenGL in a purpose-built WebAssembly module compiled from C++ using emscripten [3].

The `Status Screen` presents the state of the WiFi connection of the ESP32, both graphs of the ultrasonic sensor and of the SQUAL (signal quality) of the flow sensor, and a heat-map of the scanned arena presenting the radar's readings to find the metallic fans.

Screenshots of each screen can be found in the Appendix.

7.3 Challenges and Design Decisions

Firstly, to have a smooth user experience, all JSON data is stored in the same environment with the running scripts, and the website fetches the wanted data from the database server that presents these JSON files. This avoids application refreshes every time updated data arrives from the rover.

Secondly, all graphs and the map are updated automatically once the floating switch in the top left corner is turned on. This way the user can control when to request all the fetches completed by the website. This is to update the local data in the website with the cached data on the database web server. Therefore, undesired infinite loops, which affected the response time of the website are avoided. Due to memory overloads, this could also, in the worst case, cause crashes and freezes.

8 Radar

The radar module is used to detect underground fan structures, by using the Doppler effect. It includes a source which generates a signal at 10.5GHz, which reflects from the rotating fan blades with a frequency shift (of around 350Hz). This shift is isolated by the radar mixer as Δf.

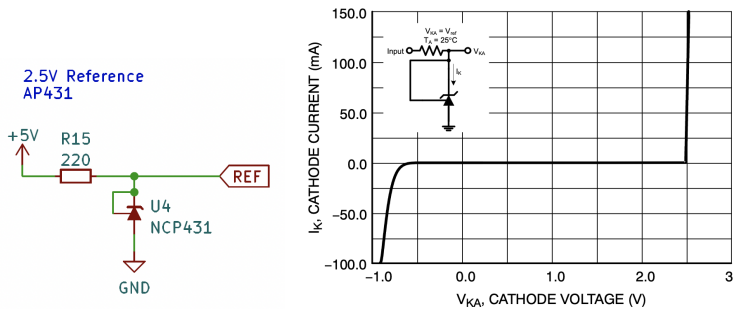
8.1 Initial Testing

Basic identification of Δf was needed. For this, the radar was simply placed right next to a fan, generating a signal at 350Hz. The oscilloscope's FFT tool showed harmonics with smaller amplitudes. The amplitude of the received signal was around 10mV. As the radar moved away, it decreased to smaller amplitudes, until it became negligible. When the fan was turned off, the signal dropped to a constant DC voltage. These observations indicated that a circuit performing two fundamental functions was needed: filtering, such that Δf would be isolated from noise and other reflections from the environment, and amplification, such that the signal would be large enough to detect the presence of a fan with certainty.

8.2 Design and Implementation

In order to transform the initial signal received from the radar into data that the ESP32 can tolerate and allow it to locate the fan, a circuit with five stages was designed: an initial amplification stage, a filtering stage, a second amplification stage, a rectification stage and finally a peak detection stage.

8.2.1 Voltage reference circuit



(a) (b)

Figure 10: Voltage reference circuit with NCP431 and IV characteristic

The voltage reference circuit consists of an NCP431 three-terminal programmable shunt regulator diode, similar to the AP431i seen in the Circuits and Systems module. It allows for a stable reference source from a 5V supply, minimising noise. It also provides equal leg and head room between the rails. The 220 resistor draws approximately 11mA of current, which, as seen in Figure 10, is within the region of correct biasing at 2.5V.

8.2.2 First amplification stage

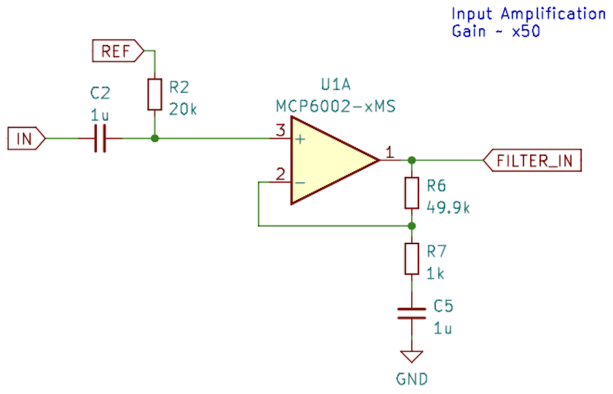


Figure 11: Input amplification stage with x50 gain

This stage is needed to increase the amplitude of the signal entering the filter, to avoid large amounts of distortion. It is a non-inverting stage with a gain of around 50. This will increase a signal IN of 10mV to 0.5V. Capacitor C2 blocks the DC component of the radar. REF biases the positive pin to 2.5V, which, through negative feedback, is roughly duplicated in the negative pin. At DC, C5 presents high impedance and looks like an open circuit, hence the op-amp acts as a voltage follower for the DC component. Overall, the output FILTER _IN has a signal amplified by a factor of 50, centred around 2.5V.

8.2.3 Filtering stage

In order to create the filter, "Analog Filter Wizard" by Analog Devices was used. The criteria given were the following: a band-pass filter with centre frequency of 350 Hz, 0 dB gain in the passband, which spans 200Hz, a topology of 4th order Butterworth, to utilise minimum number of stages, and finally a supply of 5V and 0V. The op-amps were referenced at 2.5V, to stop the signal from going below the rails. The tool yielded the following design:

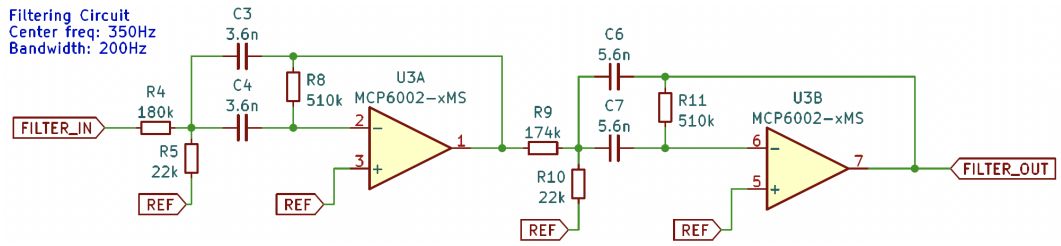


Figure 12: 4th order Butterworth filter with centre frequency 350Hz and 200Hz-wide passband, with 0dB gain

8.2.4 Second amplification stage

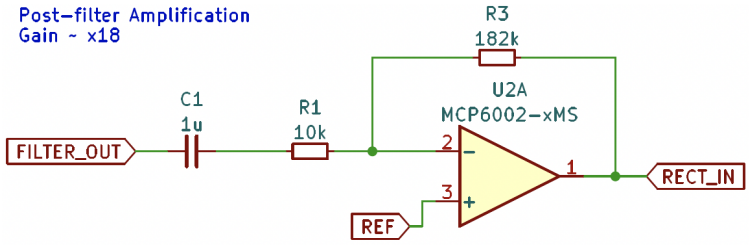


Figure 13: Second amplification stage with x18 gain

This stage is needed for further amplification. This time, an inverting amplifier with a gain of roughly 18 is used. C1 blocks the DC component of the input signal, which should not be amplified. RECT _IN is now a signal that has been appropriately filtered and amplified and is ready to enter the rectification and peak detection stage.

8.2.5 Rectification and Peak detection

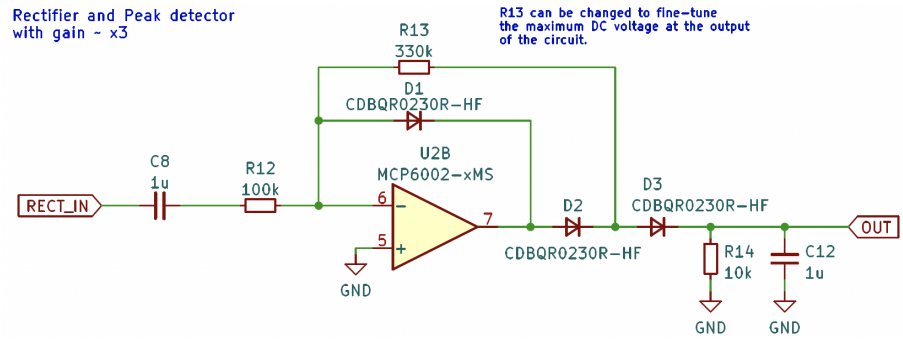


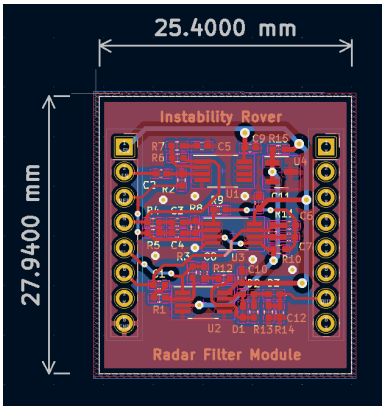
Figure 14: Rectification and peak detection stage with x3 gain

The signal at RECT _IN is still sinusoidal. The ESP32 uses 12 bits to represent a voltage at its Analog to Digital Converter (ADC) input, accepting voltages in the range of 100mV-2.45V. Therefore, the final rectified signal should have at most an amplitude of 3.3V. The amplifier is biased at ground this time, to more effectively utilise the desired range. A similar circuit was seen in the Circuits and Systems course. The rectifier also performs a final amplification.

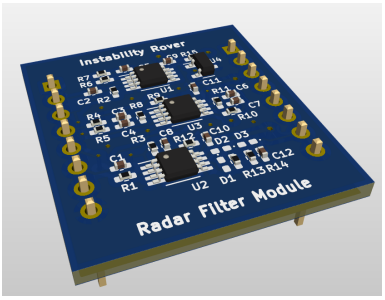
Finally, peak detection requires a parallel RC circuit, with a large enough time constant compared to the period of the signal. Knowing that the signal's period is roughly 3ms, a much larger $\tau = RC$ is needed. In this case, $\tau = 10ms$, which gives a stable enough voltage for the ESP32.

8.3 PCB Design and Final Testing

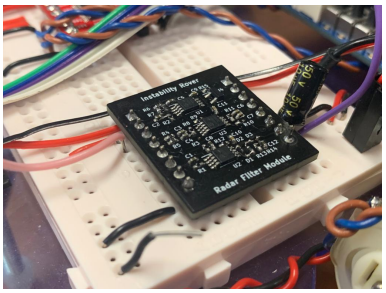
After verifying that our breadboard filter circuit works as expected, a Printed Circuit Board (PCB) was designed in KiCAD to accommodate the exact same circuit design, but on a much smaller scale. All the components on the PCB are SMD (0402 size for the passive components, and MSOP-8 for the op-amps) - this reduces the size of the PCB to about 2.5cm by 2.7cm (which is significantly smaller than the equivalent design on a breadboard). After the PCB was designed in CAD, it was submitted for manufacturing, and components were ordered. After the PCB and components arrived they were assembled (soldered) by hand. The PCB was then tested and seen to work equivalently to the breadboard design, just as expected.



(a) PCB Design in KiCAD

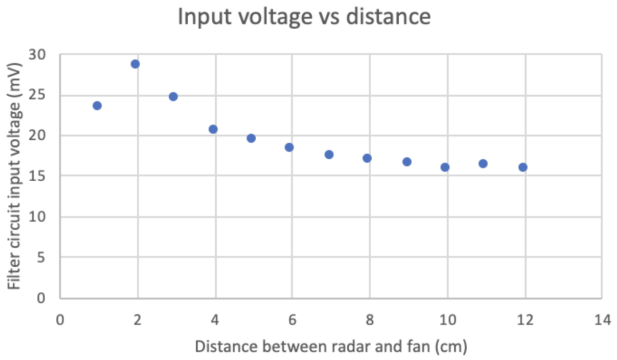


(b) 3D render of PCB exported from KiCAD

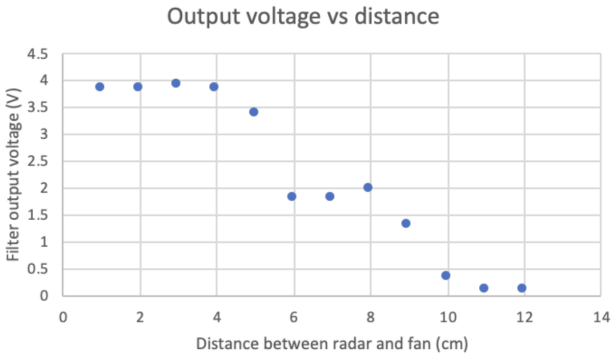


(c) Assembled PCB integrated on rover

Testing the filter module in the arena and checking the input voltage (the radar signal), and the output voltage of the filter, yielded the following plots:



(a) Plot of filter input voltage against distance between radar and fan



(b) Plot of filter output voltage against distance between radar and fan

The filter output signal can be taken through a potential divider into the ESP-32, which will then transmit the radar data to the server, producing the heatmap in the website. Locating the fan is done by identifying the area with the highest radar reading on the heatmap.

9 Energy

The aim of the energy subsystem of the Mars Rover is to efficiently charge batteries from the provided four photovoltaic (PV) cells.

9.1 PV Panels and Battery Characterisation

The PV panels can be approximated as current sources that operate at a current dependent on the solar irradiance. Increasing the amount of solar energy collected by the panels increases the panel current and open circuit voltage, as well as the voltage needed for maximum power production.

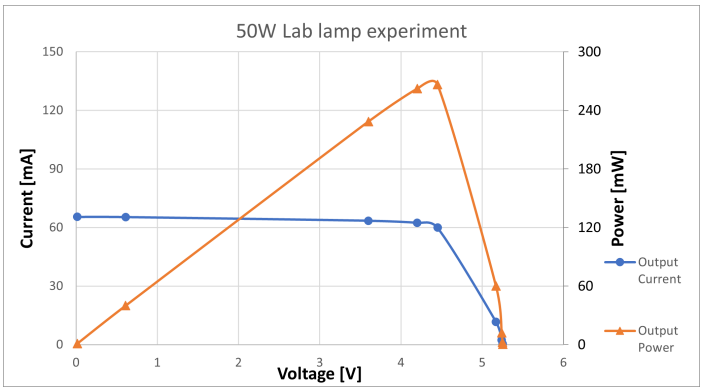


Figure 17: PV and IV characterization of 4 parallel panels

The arrangement of the panels chosen for the design is four in parallel. This is the only possible choice after considering the limitations of the SMPS circuits used, which cannot reliably handle more than 8V.

The battery also has an important input current to input voltage characteristic that shows the maximum current that the battery can draw at a given voltage. This curve is approximately linear in the operational range of the battery and its slope changes with the charge of the battery.

9.2 SMPS Design

The energy station designed consists of four parallel PV cells, a first Boost stage with MPP tracking, a second Buck stage that provides an appropriate output voltage for the battery, a relay that disconnects the battery if the rest of the circuit can damage it, and the charging battery.

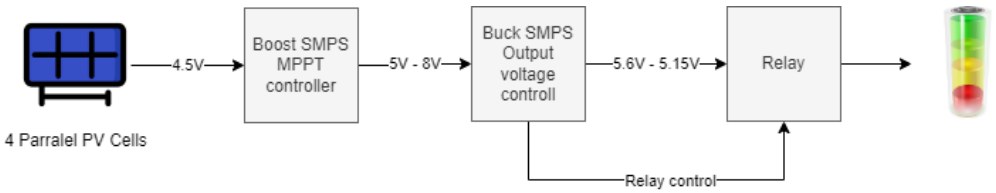


Figure 18: Energy subsystem block diagram

9.3 SMPS Implementation

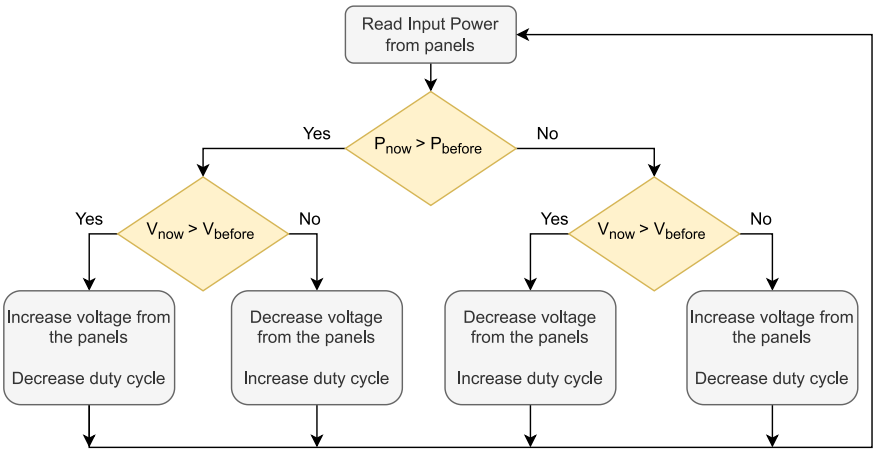


Figure 19: MPPT algorithm flow chart

The first SMPS ensures that the PV panels work at their maximum power point (MPPT). The MPP is tracked by changing the Boost's duty cycle with increments of 3%, according to the power and voltage change between two consecutive samples. The Boost also constrains its output voltage below 8V, which can occur if the battery is disconnected, and the output is open circuit. Failure to stabilise the voltages in this range will prevent the battery from reconnecting.

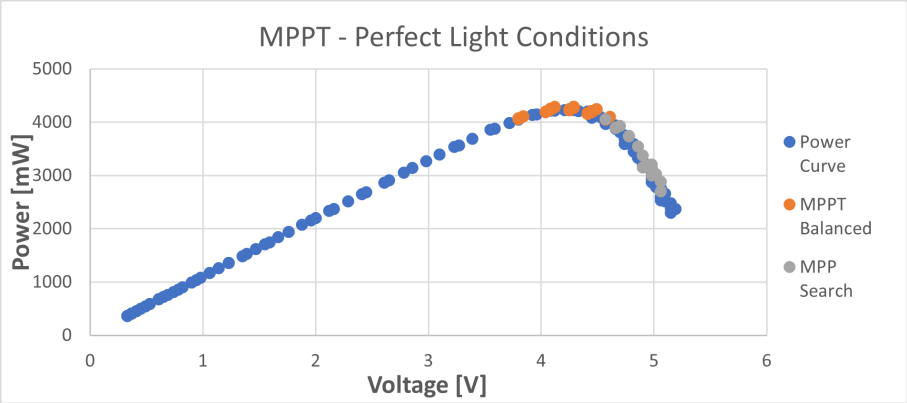


Figure 20: MPPT controller superimposed on expected voltage to power relationship for a cloudy environment

The second stage of the design is a Buck converter that steps down the voltage to around 5V, for safe and efficient battery charging. The controller of this stage is also responsible for the behaviour of the relay. The battery is connected to the normally open ports of the relay, so the Arduino connects the battery to the charger only when the voltage is safe for the battery (between 4.6 and 5.15V).

During the implementation of the relay controller, an issue was observed, which caused the relay to oscillate in some cases and not to set on open or closed operation. This was because the output capacitor of the Buck SMPS was not discharging fast enough, which was resulting in inaccurate readings from the Arduino. The solution was to connect a 1kΩ resistor to the output, allowing for quick discharge of the capacito

9.4 Efficiency

This charging station design requires relatively large input power from the panels to operate. In normal conditions (more than 5.2V input voltage), experiments showed that the input power should be more than 2W for the MPPT to correctly produce more than 5V at the output of the Boost (so that it can be scaled down in the Buck to a useful voltage for the battery).

The overall efficiency is around 77.65% (tested with direct sunlight). Improvements to the design can be made by implementing synchronous SMPS circuits, decreasing the power losses.

9.5 Battery State of Charge

The battery state while charging can be tracked by the power flow into the battery. This method, however, relies on the knowledge of the exact battery percentage at the start of a charge cycle and therefore results in large inaccuracy.

The opposite can be implemented on the rover – by knowing the energy that all components on the rover use for a given time, the discharge power of the battery can be calculated. However, a more precise method is to measure the voltages and currents out of the battery with the use of current sensors. This way the total power use is always known.

10 Final Testing

Since most integration and testing was done continuously during the entire project timeline, the final testing stage was relatively short. However certain minor deficiencies were found and dealt with, and the final system parameters (PID controller gains, Vision HSV thresholds, live transmission timings,etc.) were tuned with extensive testing on the arena.

In conclusion, we found our rover to perform well and conform to the initial requirements that were set out in the project.

11 Appendix

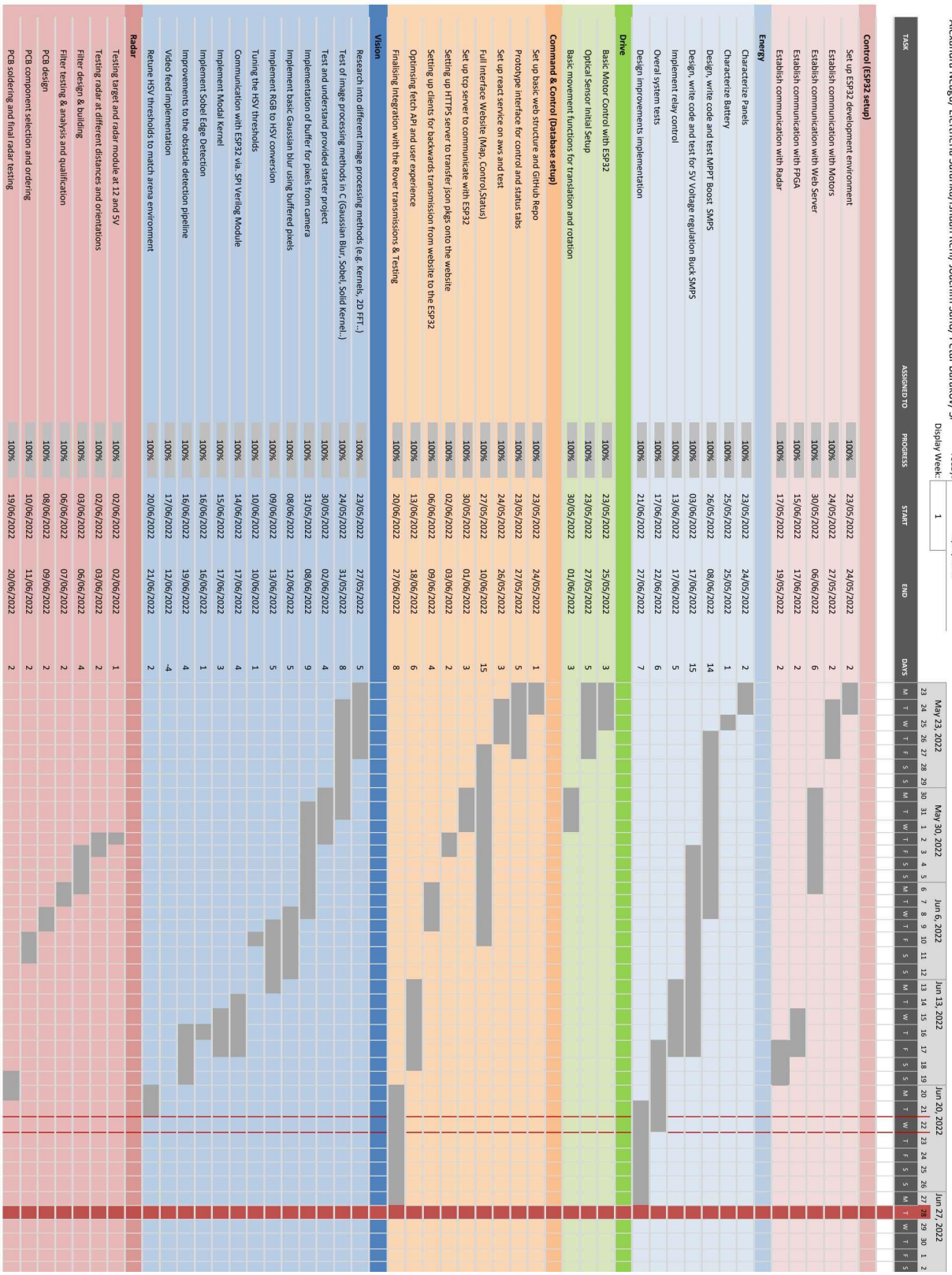


Figure 21: Gantt Chart: time management throughout the project

11.1 Appendix: Vision

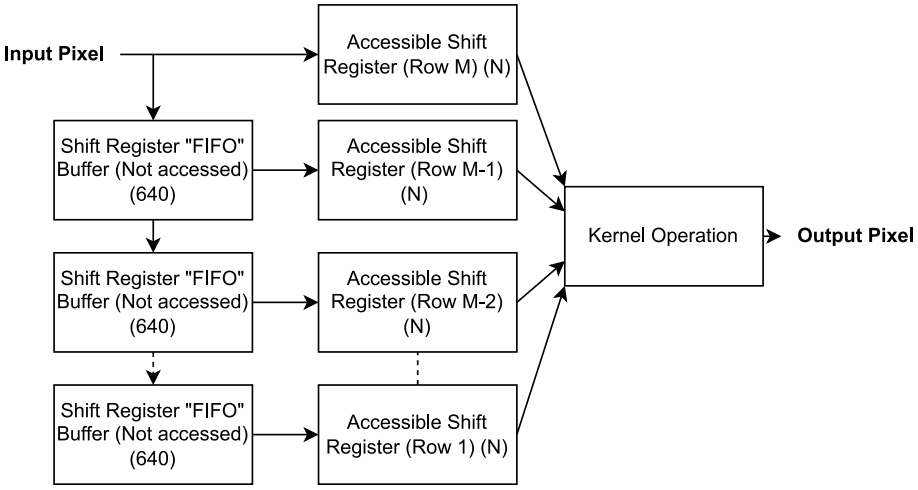


Figure 22: Diagram of the final pixel processing pipeline utilised in the vision subsystem

$$G = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Figure 23: Gaussian Blur Kernel Originally Considered

11.2 Appendix: Radar Filter

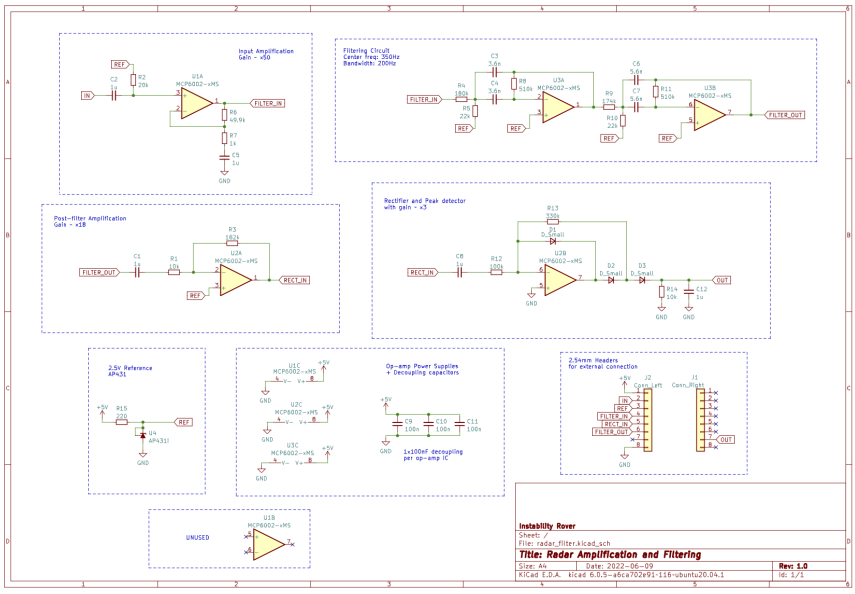


Figure 24: Full schematic of the Radar Filter PCB

11.3 Appendix: Command

Figure 25: Control Screen presenting motor speeds and orientation of rover

Figure 26: Status Screen presenting optical flow sensor's, ultrasonic sensor's and radar's readings

```
PS D:\2_Work\Y2_courseworks\Instability_Rover\Instability\Command> & C:/Python310/python.exe d:/2_Work/Y2_courseworks/Instability_Rover/Instability/Command/backend/serverESP.py
We're in tcp server...
TCP Server running on port  12000
cmsg:   l {"position": {"x": 24,"y": 590}} | {"squal": 45, "motor_left": 164, "motor_right": 136, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
Live msg:   {"position": {"x": 24,"y": 590}} | {"squal": 45, "motor_left": 164, "motor_right": 136, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
     live status:   {"squal": 45, "motor_left": 164, "motor_right": 136, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
{'position': {'x': 2.0425531914893615, 'y': 50.212765957446805}}
cmsg:   l {"position": {"x": 24,"y": 1256}} | {"squal": 22, "motor_left": 164, "motor_right": 136, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
Live msg:   {"position": {"x": 24,"y": 1256}} | {"squal": 22, "motor_left": 164, "motor_right": 136, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
     live status:   {"squal": 22, "motor_left": 164, "motor_right": 136, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
{'position': {'x': 2.0425531914893615, 'y': 106.8936170212766}}
cmsg:   l {"position": {"x": 23,"y": 1569}} | {"squal": 1, "motor_left": 163, "motor_right": 137, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
Live msg:   {"position": {"x": 23,"y": 1569}} | {"squal": 1, "motor_left": 163, "motor_right": 137, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
     live status:   {"squal": 1, "motor_left": 163, "motor_right": 137, "orientation": 45, "ultrasonic": 0.000000, "radar": 0}
{'position': {'x': 1.9574468085106382, 'y': 133.53191489361703}}
```


Figure 27: Messages transmitted over TCP from ESP32 to the database server


```
{
    "data": {
        "left": [
            {
                "x": 0,
                "y": 0
            },
            {
                "x": 1,
                "y": 160
            },
            {
                "x": 2,
                "y": 160
            }
        ],
        "right": [
            {
                "x": 0,
                "y": 0
            },
            {
                "x": 1,
                "y": 140
            },
            {
                "x": 2,
                "y": 140
            }
        ]
    },
    "name": "motors",
    "orientation": 45
}
```


(a) Motor data


```
{
    "data": [
        {
            "x": 0,
            "y": 0
        },
        {
            "x": 1,
            "y": 45
        },
        {
            "x": 2,
            "y": 31
        },
        {
            "x": 3,
            "y": 16
        },
        {
            "x": 4,
            "y": 12
        }
    ],
    "name": "squal"
}
```


(b) Optical sensor data


```
{
    "0": {
        "id": "a_0",
        "position": {
            "x": 182,
            "y": 612
        }
    },
    "7": {
        "id": "o_1",
        "position": {
            "x": 1418,
            "y": 55
        }
    },
    "10": {
        "id": "p_0_1",
        "position": {
            "x": 240,
            "y": 0
        }
    }
}
```


(c) Position nodes data

Figure 28: Example of Data shown on the HTTPS webserver

11.4 Appendix: Energy

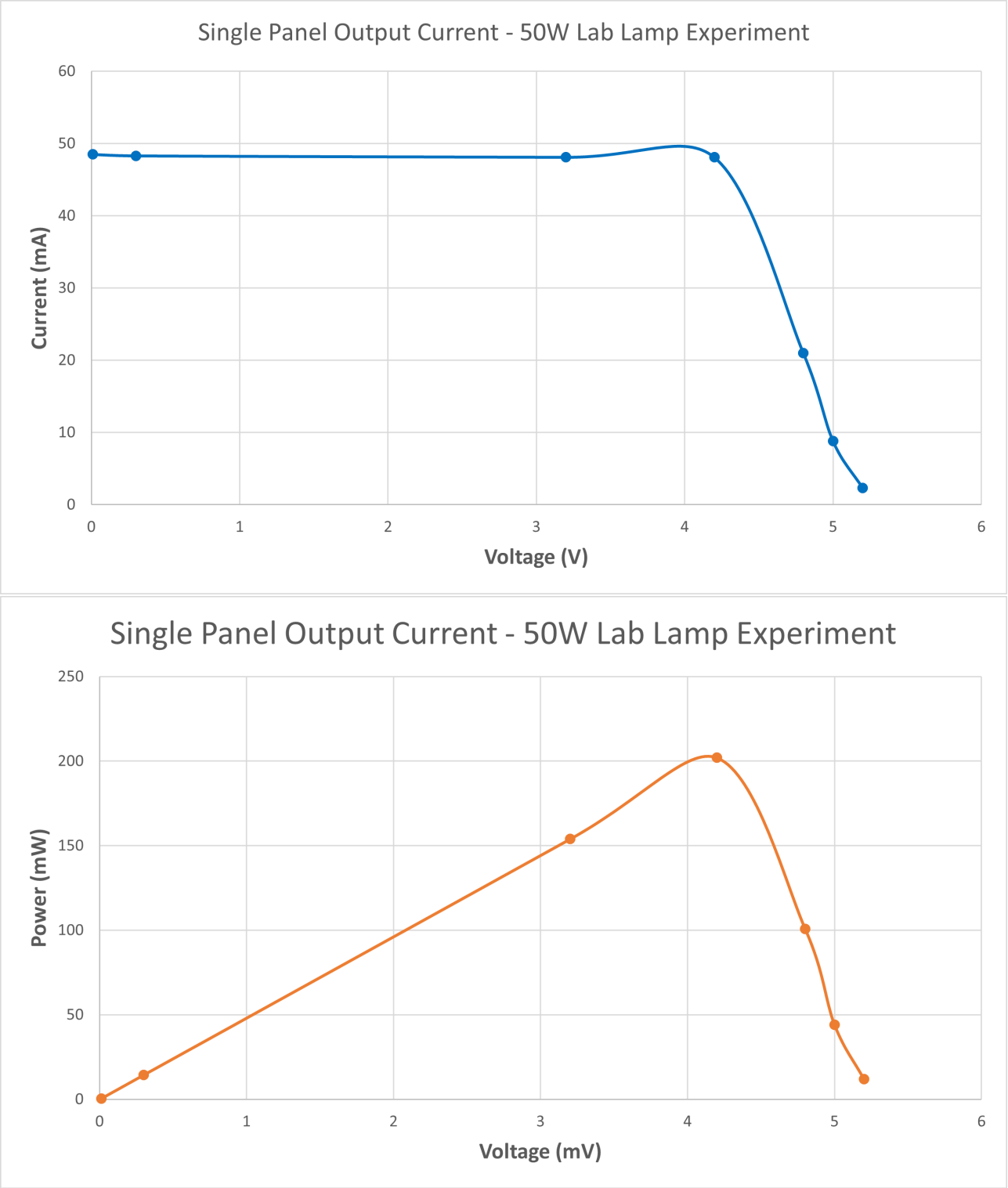


Figure 29: Single Panel characterisation

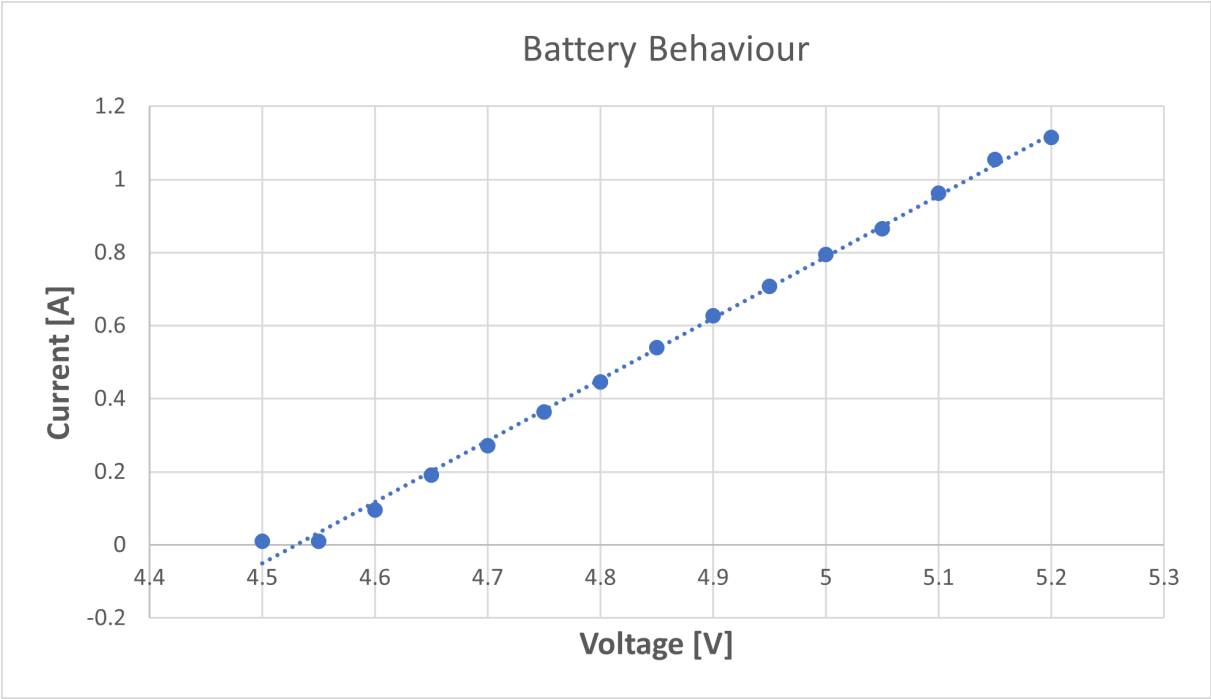


Figure 30: Battery IV behaviour

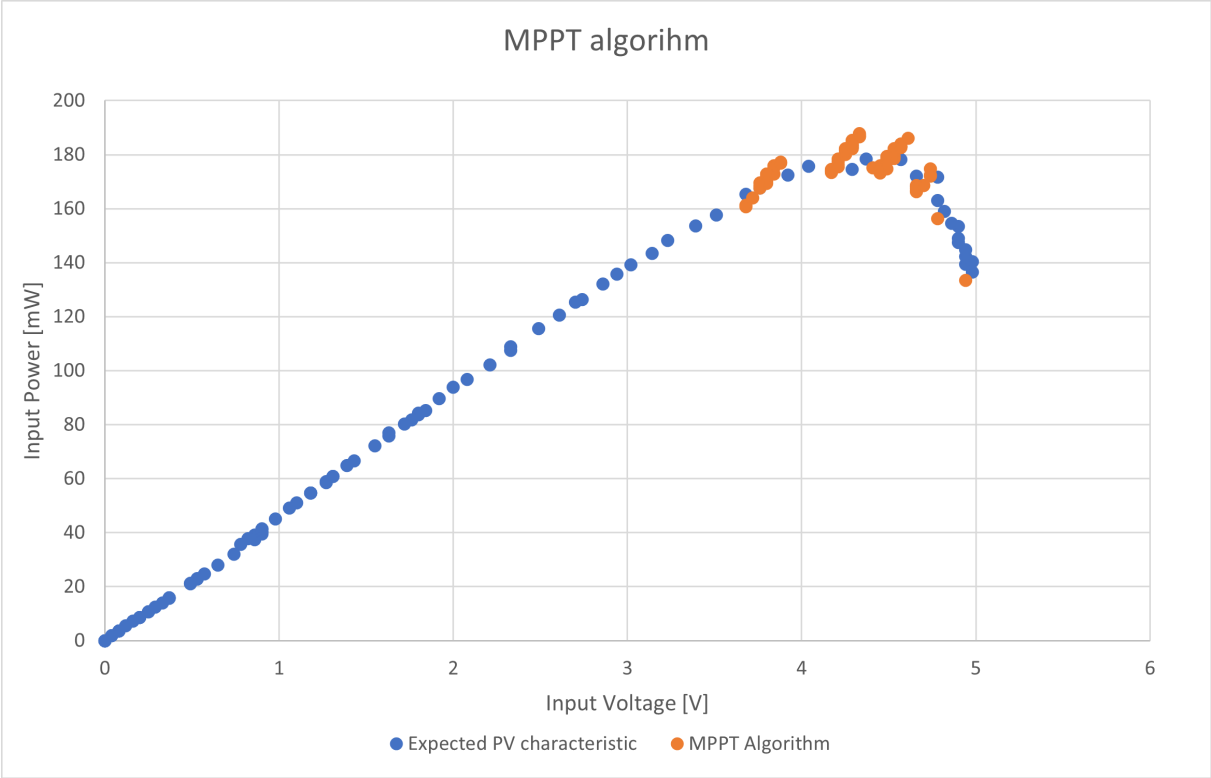


Figure 31: MPPT (cloudy environment)

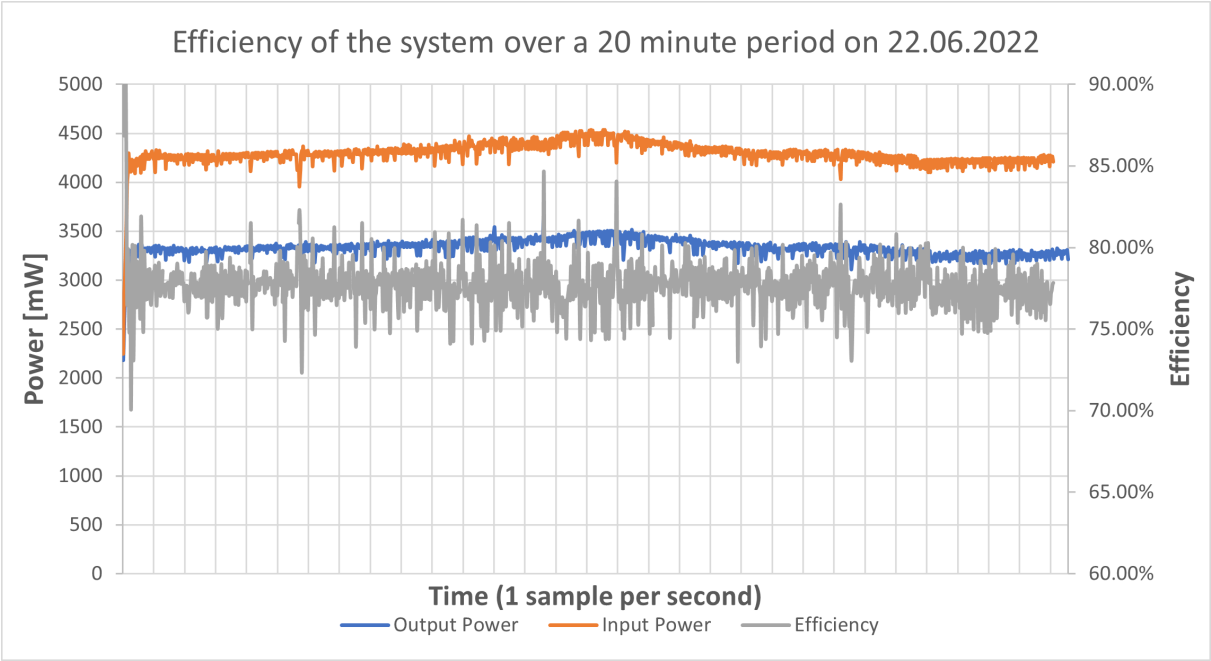


Figure 32: Input and Output Power of the Energy System

References

[1] *MIT_source_hsv*. [Online]. Available: http://web.mit.edu/6.111/www/f2011/tools/tools_2011.html.

[2] *Video and Image Processing Suite User Guide*, Feb. 2014. [Online]. Available: https://www.intel.com/content/dam/support/jp/ja/programmable/support-resources/bulk-container/pdfs/literature/ug/ug-vip.pdf.

[3] *Main — Emscripten 3.1.9-git (dev) documentation*. [Online]. Available: https://emscripten.org/ (visited on 06/21/2022).

[4] *Expo Documentation*, en. [Online]. Available: https://docs.expo.dev/.

[5] *React Navigation | React Navigation*, en. [Online]. Available: https://reactnavigation.org//.

[6] *MUI: The React component library you always wanted*, en. [Online]. Available: https://mui.com/.

[7] *@mui/icons-material*, en. [Online]. Available: https://www.npmjs.com/package/@mui/icons-material.

[8] *Home | React Flow*, en. [Online]. Available: https://reactflow.dev/.

[9] *React-chartjs-2*, en. [Online]. Available: https://www.npmjs.com/package/react-chartjs-2.

[10] *@elsdoerfer/react-arrow*, en. [Online]. Available: https://www.npmjs.com/package/@elsdoerfer/react-arrow.

[11] *React-bulb*, en. [Online]. Available: https://www.npmjs.com/package/react-bulb.

[12] *Fs.promises*, en. [Online]. Available: https://www.npmjs.com/package/fs.promises.

[13] *React-heatmap-grid*, en. [Online]. Available: https://www.npmjs.com/package/react-heatmap-grid.