ELEC50008 - Engineering Design Project 2

Group: Name Pending

Vincenzo Nannetti 02099447 | Felipe Mendes 02041106 | Ismail Sajid 01924979 | Jamie Turner 02017907

Kridhay Mahesh 02107811 | Hanbo Xie 02017205 | Jongmin Choi 01956764

June 2023

Abstract

This report outlines the design and development of a self-navigating BalanceBot that is capable of traversing and mapping out the layout of a maze as well as finding the shortest path through the maze. The "BalanceBot" or "Rover" (as referred to in this report) uses a control system to balance vertically on two wheels. The final design utilises an FPGA based camera system and an ESP32 microcontroller. A server hosted by AWS is used to communicate over Wi-Fi with the rover and the user's computer.

Contents

1 Introduction

1.1 Outline

This report provides a detailed account of the design process, hardware integration, software implementation, experimental results, and analysis of the development of our BalanceBot. By developing the BalanceBot, we intend to demonstrate the potential of robotics in solving complex tasks such as maze traversal and path optimisation.

1.2 Project Management and Planning

Time Management

We completed a Gantt Chart, to ensure that each member has a clear outline of the tasks that needed to be completed and by when. Both the Gantt chart and the specific team contributions are outlined on Github Readme. This was created at our first team meeting and we began by following this as closely as possible. Each member was also allocated a role based on their strengths and experiences.

However, as time progressed, we found that the project required a lot of integration between subsystems and coordination between team members. Problems also arose since members would be left without any work since they would have to wait for another team-member to complete a task before they could start their work. We therefore shifted away from the initial plan and we started following a more iterative development cycle. There was more teamwork and we worked together to try and work through the more harder and important subsystems of the project. The iterative style suited us since we would rapidly prototype a system, and continually improve on it until no more changes needed to be made.

Collaboration

To ensure that all of the team could effectively collaborate and each members contribution to the project could be recorded, a GitHub repository was setup.
https://github.com/hanboxie/balance-bug

Budget

We completed the project within the budget of £60. A detailed breakdown of our expenses as a team can be found in Appendix B

1.3 Subsytem Design

We divided the project into appropriate subsystems. This allowed for more effective teamwork as each member could focus on a specific area before integrating the modules at the end.

Figure 1 depicts the overall decomposed system. To facilitate task assignment within the team, the subsystems have been categorised into separate sections. The energy system primarily encompasses the subsystems within the blue box, while the application-related subsystems are contained within the red box. The green box denotes the subsystems that primarily relate to the vision aspect of the project, and the yellow box represents the subsystem that focuses on control. The unboxed section, comprising the main controller (ESP32), motor driver, and motors, offers various functionalities and therefore does not fall under a specific group.

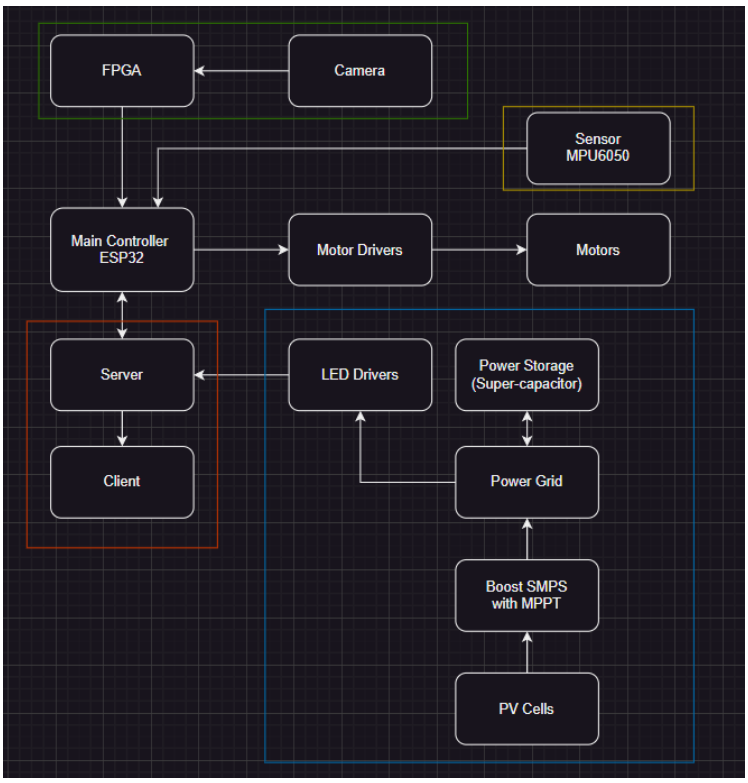


Figure 1: Top Level System Decomposition

2 Hardware

2.1 Energy System

The energy system is concerned with the harvesting of sunlight to supply power to 3 LED beacons. These can be used by the BalanceBot for position and orientation tracking. Due to the unpredictable nature of sunlight and operating conditions, we must account for redundancy when powering the beacons.

We decided to design a grid topology and connect each component to the grid - which provides the power. The photovoltaic (PV) cells supply the grid via a Switch Mode Power Supply (SMPS), which employs a Maximum PowerPoint Tracking (MPPT) algorithm.

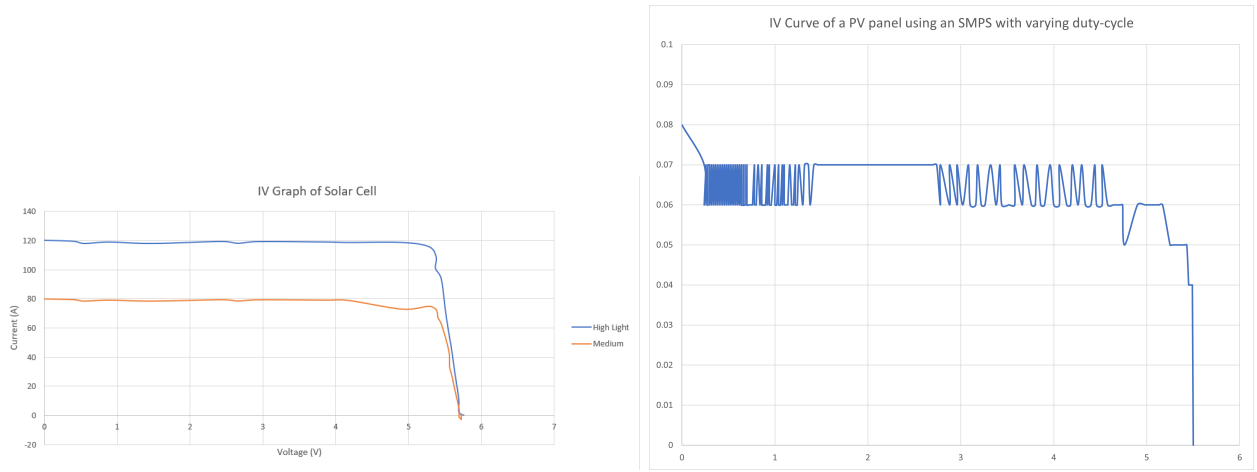
Solar Panels

To characterise the given panels, we took them outside at different times of the day to ensure that they were subject to irradiation by different light intensities and connected them to different value loads. Initially, we used the SMPS as an intermediary to the load and adjusted the duty cycle to draw different values of current. This, however, led to undesirable results and thus we opted to connect the solar panel directly to different value loads.

Results using the SMPS and solely discrete values can be shown in Fig. 2a and Fig. 2b respectively.

After observing these results, we concluded that the open circuit voltage (V_{oc}) under high light intensity is approximately 5.76V The short circuit current (I_{sc}) under the same conditions was seen to be 120mA. We also found that under medium light intensity, V_{oc} was 5.73V and I_{sc} was 80mA. Due to the weather conditions during the period in which we were testing, we were unable to get accurate results for low light intensity - unless the time of day was at night, but we did not feel as though this was an accurate representation of low light conditions.

These results also allowed us to find the fill factor of the solar cell which we can use to estimate the



(a) IV Curve using Resistor Values

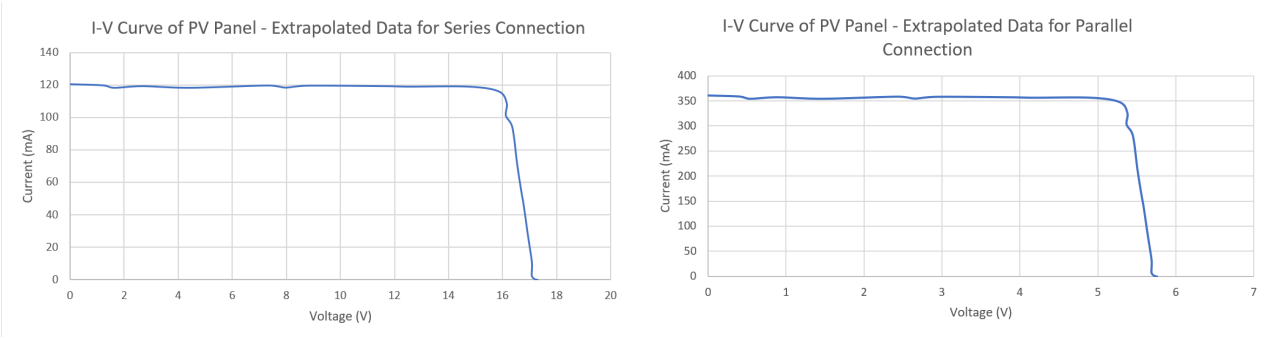
(b) IV Curve using SMPS

Figure 2: IV Curve of PV Panel

efficiency of the solar cell, by calculating the ratio of maximum power to the product of I_{sc} and V_{oc} (O'Kane, Mary 2023).

For maximum light intensity, the maximum power is: $5.28 * 0.116 = 0.61248$. Thus the fill factor can be seen to be $\frac{0.61248}{(5.76 * 0.120)} = 0.886$. This gives us an approximate efficiency of 89%. Similarly, for medium light intensity the fill factor can be found to be 86%. This concludes that our solar panels are adequately efficient and thus suitable to be used within our energy system.

This was completed with one solar panel, however, we will be using 3 for the project and thus the graph must be changed in order to account for this. Depending on the way the solar panels are organised, the graphs will be different. These results can be shown in Fig. 3, whereby a series connection led to the addition of the voltage and the same current, and a parallel connection will have increased current but the voltage will stay the same (Nature's Generator 2023).



(a) Series Connection

(b) Parallel Connection

Figure 3: Extrapolated IV Curve

In each case, there can be seen to be a point of maximum power. This can clearly be seen after plotting Power vs Voltage (Fig. 4). This is for a single PV cell, in actuality the power will be approximately three times these readings.

After researching the different MPPT algorithms, we compared Incremental Conductance and the Perturb & Observe (PO) method.

Incremental Conductance is a method of maximum power point tracking whereby the incremental conductance and instantaneous conductance are measured and compared to determine the direction of the output voltage. If these measured values are equal then the maximum power point has been

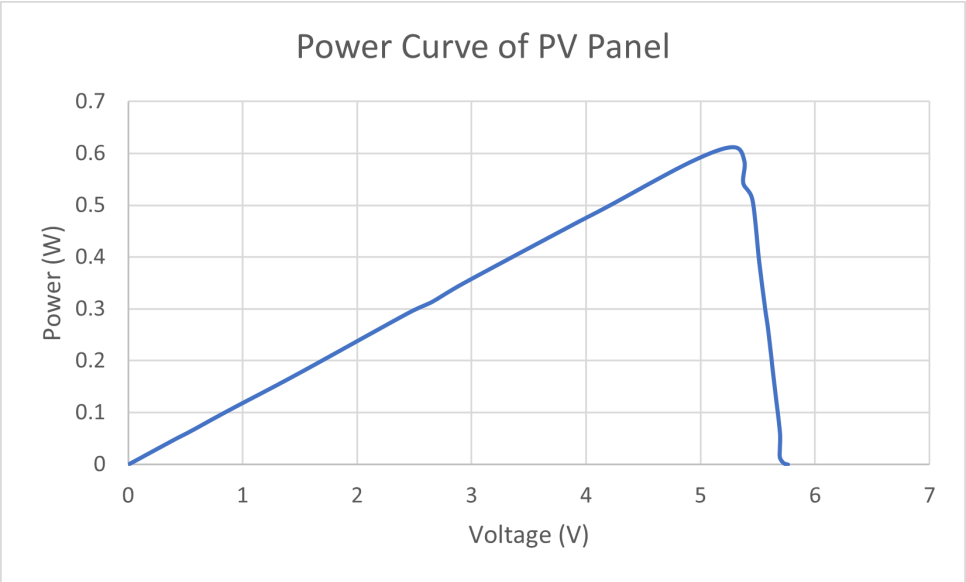


Figure 4: Power Curve of Single PV Cell Under High Light Intensity

reached (Selvan et al. 2016). The general explanation for this algorithm is that it takes the gradient of the Power vs Voltage curve and continuously checks to see if it is increasing, decreasing or zero. Zero is desired as it represented a maximum point.

Perturb & Observe is another method of maximum power point tracking whereby we systematically decrease or increase (perturb) the operating voltage of the solar cell and observe the power yield (Mahdi et al. 2019). If we observe an increase in power yield then we perturb further in this direction.

The MPPT method of Perturb & Observe is simple to implement in digital circuits (Hlaaili & Mechergui 2016), which is beneficial for our application where we will be using an Arduino Nano to implement this algorithm. The Incremental Conductance method, on the other hand, is difficult to implement and requires more complex control algorithms (Hlaaili & Mechergui 2016). We concluded that the Perturb & Observe method would work the best for our requirements. This is due to the SMPS that we are using to employ this algorithm having the necessary sensors, allowing us to calculate the power and adjust the voltage accordingly, by adjusting the duty-cycle.

We then implemented this algorithm within C and uploaded it onto the micro-controller within the SMPS. The implemented program can be seen in the GitHub, however the main algorithm can be seen in Appendix 9.1 (Listing. 1).

Due to the uncertainty of the conditions, as mentioned above, we opted to create an emulation of the solar panel which will allow us to use the power supply unit (PSU) within the lab to test our power grid.

To do this, we set the voltage of the PSU to the open circuit voltage and the current limit to the short circuit current. To get the characteristic curve observed in Fig. 2, resistors needed to be added. The value of these resistors were chosen in accordance with the gradient of the IV graph. To reproduce the left side of the graph a resistance of 1320Ω was added in parallel with the PSU and to reproduce the right side of the graph a resistance of 4.14Ω was added in series.

These resistances were calculated as the gradient of the IV chart is $\frac{I}{V} = \frac{1}{R}$ and thus the resistance is the reciprocal of the gradient.

We then compared the emulation with the results from the testing carried outside (Fig. 5 - Fig. 6) and we concluded that this was an adequate emulation that we can use for the remainder of the testing.

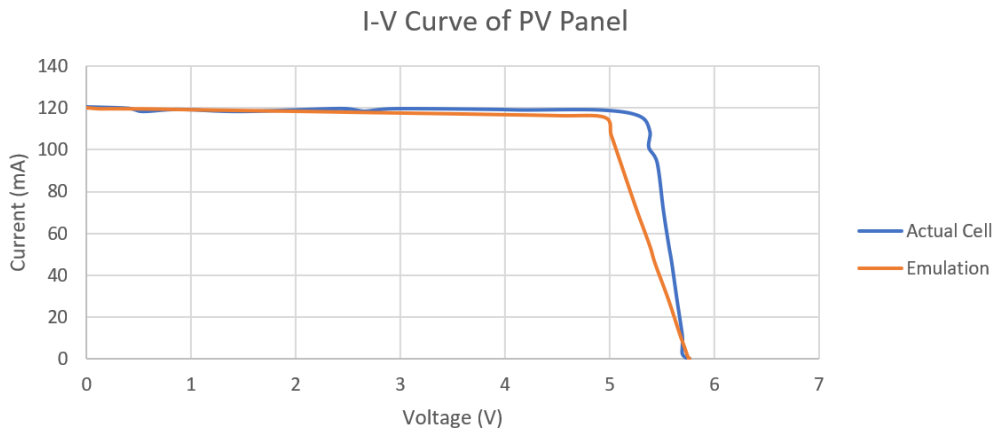


Figure 5: IV Curve of PV Panel including Emulation Results

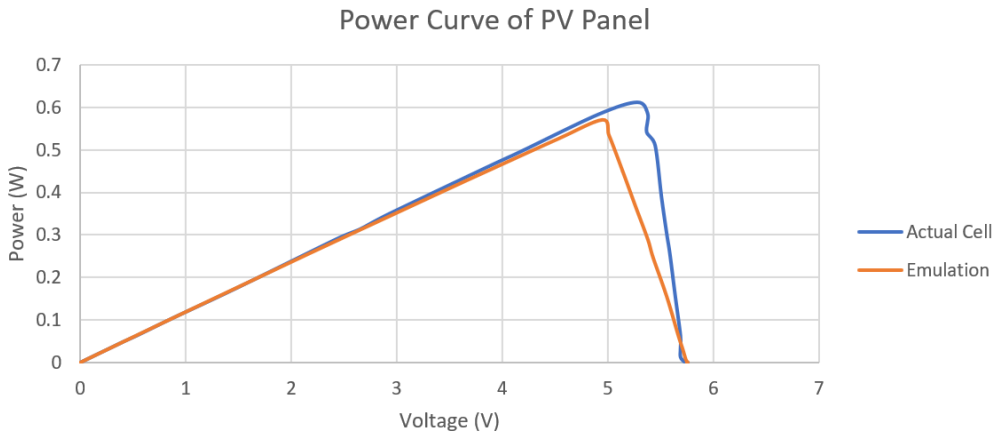


Figure 6: Power Curve of PV Panel including Emulation Results

LED Drivers

The other section that was worked on involved the powering of LEDs (beacons) which are connected to the grid. The beacons have a maximum voltage of 3V and current of 300mA. The grid exceeds these ratings, so, instead of directly connecting the beacons to the grid, they were instead connected to the grid via an LED driver which is rated for a minimum of 7V.

The LED drivers act as a Buck SMPS, which takes a voltage as input and then outputs a voltage which is decreased in magnitude. This is done by varying the duty-cycle (δ) of the SMPS and in continuous mode, which is the mode of operation, the input and output voltage are related by $\frac{V_o}{V_i} = \delta$.

We concluded that the most efficient way to control the output current of these drivers was by using a PI controller which would compare the output current of the LED driver to a given reference and adjust the duty cycle accordingly to match this value (Fig. 7).

To set the reference values of the PI controller, we first needed to characterise the LEDs used for our beacons. This was done by finding the IV curve of each LED and can be shown in Fig. 8.

Observing Fig. 8, we can see that the Blue beacon has the greatest forward voltage of approximately 2.5V, and colours Red and Yellow have approximately the same at 1.75V.

We collaborated with the team members responsible for the vision section to determine the appropriate current range for accurate colour detection by the camera. These ranges were found to be:

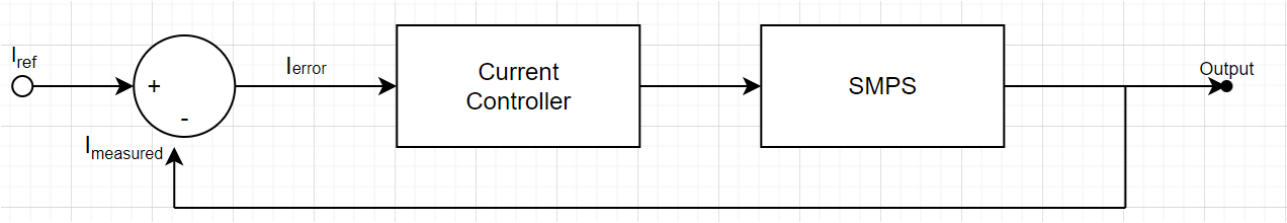
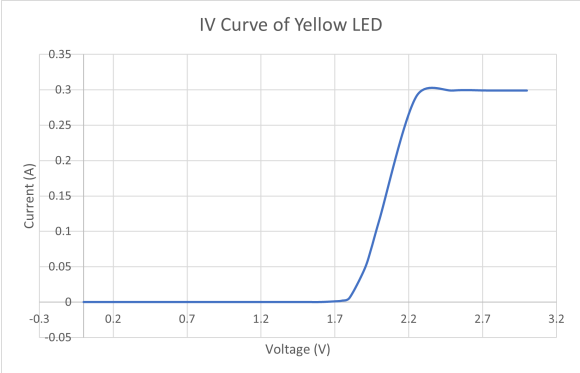
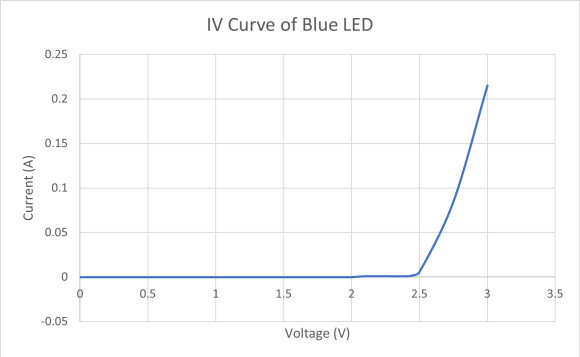


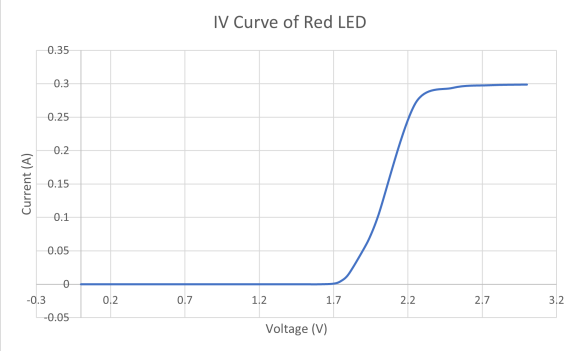
Figure 7: Structure of Current Controller



(a) Yellow



(b) Blue



(c) Red

Figure 8: IV Curve of LEDs

- Red: 15mA - 20mA
- Yellow: 40mA - 45mA
- Blue: 25mA - 30mA

These findings allowed us to establish the minimum voltage and current ranges required for the LED drivers to operate effectively, depending on the specific colour. Consequently, we were able to set appropriate thresholds and fine-tune the PI coefficients accordingly.

These coefficients will be used in a simple PI controller which will control the output current. This works by first calculating the instantaneous current error and integral error and then summing the product of these errors with their respective gains (Samak & Samak 2018). This gives us the overall PI output which is subsequently used as a reference for the pulse width modulation (PWM) signal, which controls the duty cycle of the driver.

As the error becomes more positive, the controller acts in a way to increase the duty cycle of the driver which will increase the output current and thus decrease the error. The opposite is true when the error becomes more negative.

We also wanted to link our LED drivers to the website which would allow us to observe how much

power is being provided to the LEDs - giving us some indication as to how the grid is operating.

This was done by using the capabilities of the Raspberry Pi Pico W having WiFi connectivity - allowing us to communicate to the website.

Supercapacitor

As mentioned previously, relying on solar power alone is not ideal and thus a storage device is added to the grid to provide additional power in cases of low light. We incorporated a supercapacitor as a storage device in the grid. This supercapacitor is connected to the grid through a bidirectional SMPS, which enables both charging and discharging of the capacitor at different voltage levels. The SMPS is equipped with sensors to monitor the system.

The decision to switch between modes is based on the sensor readings from the SMPS. If the grid voltage drops below 7V and the capacitor is sufficiently charged, it will be utilised to provide additional power to the grid, raising the voltage above 7V and ensuring the proper functioning of the LED drivers. Conversely, if the grid voltage is above a certain threshold and the capacitor is not adequately charged or fully charged, the SMPS will be configured to supply power from the grid to charge the supercapacitor, preparing it for future grid fluctuations.

While our design incorporates these principles, the absence of the bidirectional SMPS prevented us from verifying the system's performance firsthand. Therefore, we were unable to identify and address any potential flaws in our logic.

2.2 Control System

The BalanceBot, being an inverted pendulum by design, exhibits instability in open-loop conditions, resulting in an inability to maintain an upright position. However, by implementing a closed-loop controller within the system, we can achieve stabilisation of the BalanceBot. This controller utilises the data collected by the onboard sensors to send suitable signals to the individual motors, enabling the balancing of the centre of mass above the axis of the wheel rotation

The first step was to determine the transfer function of our inverted pendulum system which was done by taking the Laplace Transforms of the motion equations, which were found from Lundgren, Ludvig. and Ahnlund, David. (2022) to be:

$$\ddot{x} = g\theta - l\ddot{\theta}$$
$$F = mg\theta + Mg\theta - Ml\ddot{\theta}$$

This resulted in a transfer function of:

$$\frac{\theta(S)}{X(S)} = \frac{s^2}{g - s^2 l}$$

where l is the length between masses and g is the gravitational field strength. This transfer function is an approximation and is used to check the stability of the rover when connected in closed loop conditions. This function is not actively used in the current design iterations, however will be considered and built upon in the future iterations - such as a more complex controller (Lundberg, Kent 2002).

We decided to use a PID controller, as the fast response and stability provided by PID control enable our balancing bot to swiftly correct its position and maintain stability in the face of disturbances. The proportional component responds promptly to errors, while the derivative component anticipates and dampens sudden changes. Furthermore, the control loop could run at high frequencies due to the real-time implementation, enabling rapid changes and continuous balance maintenance. This feature ensures the timely response necessary for our application, providing effective balance control for our self-balancing robot(Kocur et al. 2014)

In order to implement our controller, we had to feed the sensor data (from the MPU6050 which includes an accelerometer and a gyroscope) into the PID controller. However, we could not just use the raw data from the sensors. In order to get a more accurate picture of the current state, we should use a filter. In order to combine the values from both of the sensors, minimise the errors from each sensor, and to eliminate noise, we must use appropriate filtering of the sensors before feeding the data to the controller.

The Complementary filter is an algorithm that uses a weighted average of accelerometer and gyroscope measurements to accurately estimate the tilt angle. The gyroscope is more heavily weighted for short-term stability while the accelerometer is more heavily weighted for long-term stability in the specified arrangement of the weights. This method of merging the readings enables the complementary filter to produce a more accurate tilt angle calculation while compensating the limitations of each sensor. (Albaghdadi & Ali 2019)

The Kalman filter, on the other hand, accounts for the uncertainty in the sensor measurements and the behaviour of the system. Since the filter takes a probabilistic approach and models the system as a series of equations that explain the dynamics and connections between the state variables and the measurements, it is suitable for systems that are continually changing. The Kalman filter reduces the effect of measurement noise and offers an ideal estimate of the system state by repeatedly updating the state estimate based on fresh measurements and forecasts.(Gui et al. 2015)

Given the real-time constraints of our self-balancing robot, complementary filter emerged as a feasible option, offering adequate accuracy and noise rejection while assuring realistic hardware implementation.

The complementary filter was thus implemented into the control algorithm (Appendix 9.1 - Listing. 3). It first calibrates the sensor by taking 2000 data readings (from a position where the readings should all be 0) and then averaging them. These averages are then stored in variables which will be known as the "offset values", which will be subtracted from each reading thereafter to ensure any offset is removed.

The first step of the filtering is to convert the accelerometer data into data that can be used. We thus divide the incoming data by the "Sensitivity Scale Factor". As we have selected our MPU6050 to be calibrated with a range of +- 2g, the scale factor is 16384 LSB/g - as per the datasheet. After this we use some basic trigonometry and find that the angle in degrees in the x-axis is

$$\arctan\left(\frac{\text{Acc\_Y}}{\sqrt{\text{Acc\_X}^2 + \text{Acc\_Z}^2}}\right)$$

where Acc_X, Acc_Y, Acc_Z are the accelerometer data in the X, Y and Z axis respectively. Similarly, angle in the y-axis is given by

$$\arctan\left(\frac{-\text{Acc\_X}}{\sqrt{\text{Acc\_Y}^2 + \text{Acc\_Z}^2}}\right)$$

In both cases, these values were converted from radians to degrees and the offset found from the calibration was subtracted from them.

Next, we need to process the gyroscope data and to do so we subtracted the offsets from the initial readings and then divided by 131 - the gyroscope sensitivity scale factor relating to the chosen range of +-250°.

After this conversion, we are required to integrate the gyroscopic readings. The reason for this is because the gyroscope measures the rate of rotation (angle) and thus integrating this gives us the angle for all time (Hunter Adams, V 2014).

After we have the correct format of data, we can apply the filtering. We do this by taking a portion of the angle from the gyroscope and the remainder from the accelerometer and then summing them

together. This is called sensor fusion and is used to increase the accuracy of the results (Baker, Bonnie 2018).

The equation of the implemented sensor fusion algorithm is shown below.

$$CurrentAngle = c \times (PreviousAngle + GyroscopeAngle) + (1 - c)(AccelerometerAngle)$$

Where the high-pass filter coefficient, c, is set to 0.98. We decided on this value due to the gyroscope being more accurate than the accelerometer and thus would ensure desired results.

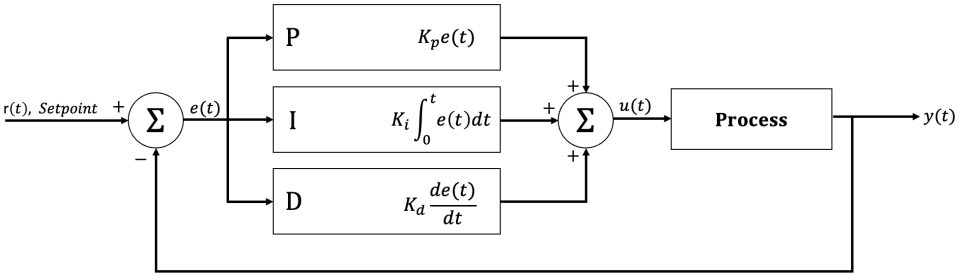


Figure 9: PID controller along with the state feedback (Samak & Samak 2018)

After gathering the data from the sensors, we then employed a PID controller to use these readings and adjust the motors accordingly - based on the angular error with respect to the reference of 0°. The basic structure of the PID controller can be seen in Fig. 9 and the final implemented code can be seen on the Github.

Through the utilisation of Simulink, we conducted a comprehensive simulation to evaluate the performance of our self-balancing robot. The simulation was based on a model from mouad boumediene (2023) whereby our 3D Simscape design (Fig. 11), is perturbed by an impulse signal, allowing us to analyse the outcome. The use of the PID block within Simulink allowed us to automatically tune our PID values.

These values were found to be:

- Kp = 9.56278806508995
- Ki = 76.5047943657588
- Kd = 0.29457713208346
- N = 1642.37346948378

The value N represents the filter coefficient of the PID value that the simulation implemented. This filter coefficient represents some low pass filtering to mitigate the effect of noise in the error signal (https://electronics.stackexchange.com/users/35530/spehro pefhany). We therefore decided to implement this within our own controller.

The results of the pitch angle over a 10 second time period can be seen in Fig. 10. As can be seen, there is a spike at the start which causes the BalanceBot to become too unstable and thus the following methodology is used to aid tuning.

To achieve optimal stability, the PID constants (Kp, Ki, Kd) were iteratively adjusted and fine-tuned; using the PID values found from the simulation as a starting point.

We followed a systematic process of Ziegler-Nicholas tuning method, where we increase the proportional gain of the BalanceBot and find the maximum gain (Bennet, James. and Bhasin, Ajay. and Grant, Jamila. and Chung Lim, Wen. 2023). Following the guidelines of this method, we are currently fine-tuning the controller. The refinement process is expected to be completed before the demonstration.

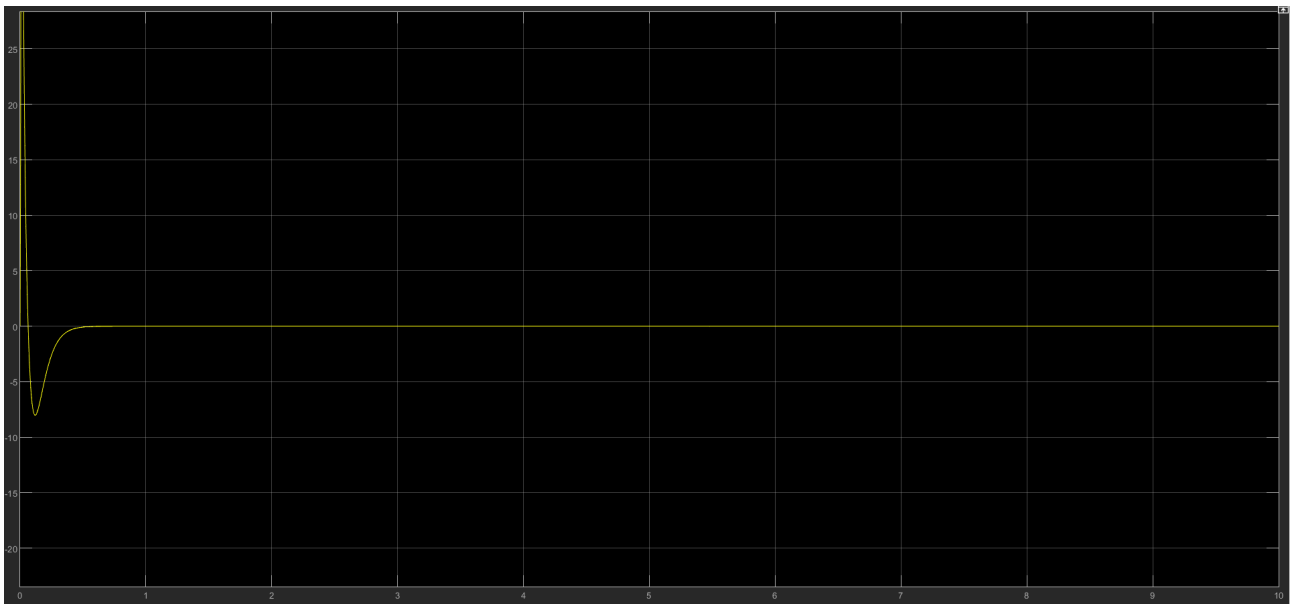


Figure 10: Pitch Angle vs Time

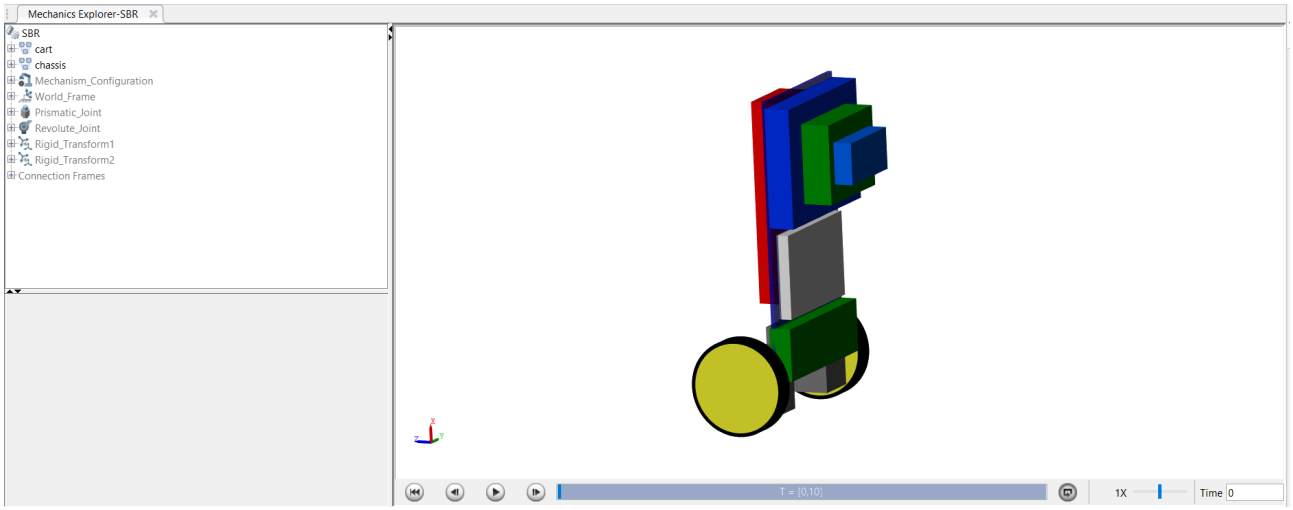


Figure 11: Simscape Model of BalanceBot

2.3 Drive System

Directions

Considering the highly optimised nature of our system and hosting, the latency between the server and the ESP32 was remarkably low. Leveraging the advantages of the server, including its computational power of a cloud server and comprehensive overview of the entire maze rather than just immediate surroundings, we determined it to be the most efficient/accurate approach for the ESP32 to post its data to the server and receive directions.

The ESP32 would communicate with the server by posting rover-related information to the /api/-pathing endpoint, which utilises the maze traversal algorithm. In return, the ESP32 would receive instructions on the next course of action (further details on the data format/endpoints/algorithms are provided in the corresponding section). By constantly updating the rover and its surroundings to the server, the system would continuously self-correct and obtain the most precise readings and instructions.

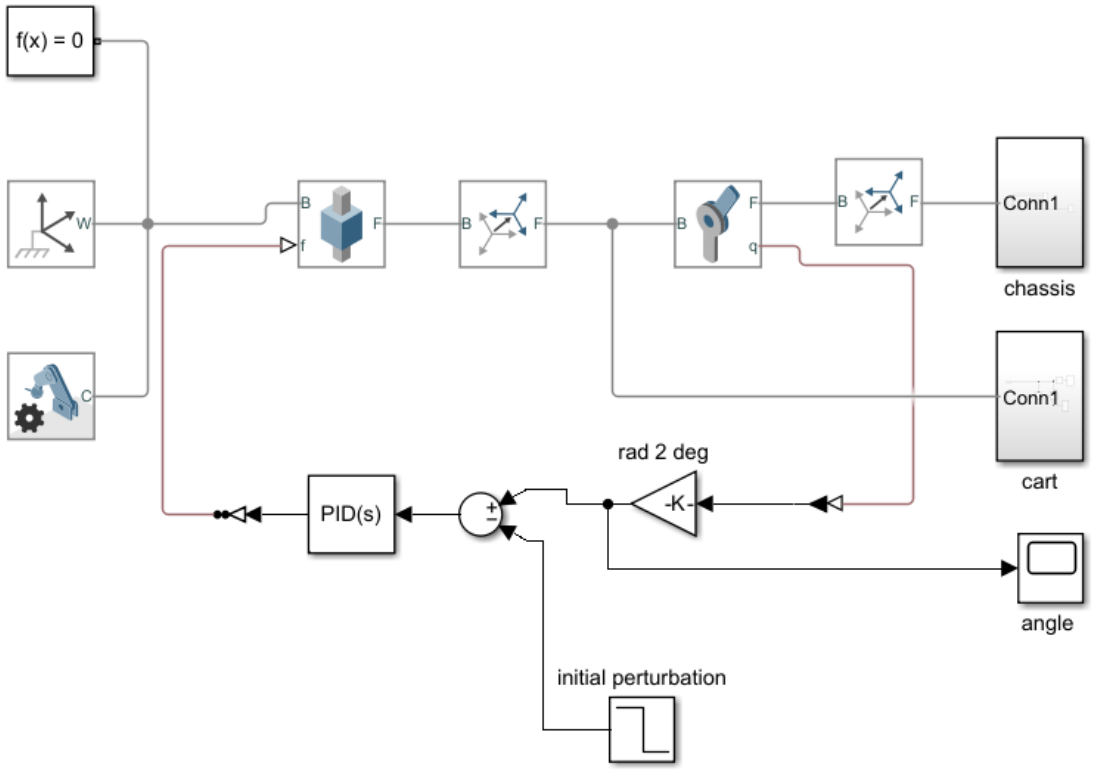


Figure 12: Simulink Model used for finding initial PID constants

Interrupts Motor interrupts play a critical role in enhancing the rover's robustness and performance, specifically in preventing collisions with walls. Among the various interrupts implemented, the most significant one involves pausing the rover immediately upon detecting an illuminated line ahead and requesting new instructions from the server.

A particular challenge arises when the server continuously provides instructions that are not feasible for the rover to execute. To address this issue, the ESP32 module keeps track of the number of consecutive "impossible" instructions received. Once five consecutive invalid instructions are recorded, the rover initiates a clockwise turn until the illuminated line in front is no longer detected. It then proceeds to move forward by approximately 5 cm or until encountering a new obstacle within that distance. (The specific amount of turn angles / consecutive false instructions / forward movement amount will need to be further fined tuned through testing) Subsequently, the rover updates the server to receive fresh instructions and resume its exploration.

Conversion

Once the rover receives an array of directions from the server, it needs to convert these instructions into executable driving directions. Currently, our system for this conversion is relatively basic and is still being refined. The instructions received are in the format of moving forward by a certain distance in centimeters or turning by a specific angle in degrees.

To execute these instructions, we calculate the amount of rotation required for each wheel based on the known radius of the wheel. By converting the desired moving distance or angle into the corresponding wheel rotation, we can control the rover's movement accordingly. However, it is important to note that we are still in the process of finding the exact mapping between the distance or angle and the corresponding wheel rotation. This refinement is one of our immediate next steps to improve the accuracy and precision of the rover's movements.

To address the issue of jagged rover movement caused by slanted lines in the maze, the team plans to implement a "look-ahead" algorithm for better direction mapping. Currently, the discrete nature of the maze mapping can lead to jagged paths, resulting in short and abrupt instructions from the server. However, with the look-ahead algorithm, the rover will consider a weighted average of the next few instructions to provide more suitable instructions that smooth out the potential jaggedness.

By taking into account the upcoming instructions and applying a weighted average, the look-ahead algorithm aims to mitigate the jagged movement of the rover and improve its accuracy in longer distances. This approach allows the rover to plan its path more effectively, reducing abrupt turns and ensuring smoother traversal through slanted lines in the maze.

Challenges

The drive system presented three primary challenges. Firstly, testing the synchronisation of motor instructions with the directions proved to be complex. Secondly, the motor system had to manage interrupts, execute server instructions, and maintain rover balance simultaneously. Lastly, the conversion of directions into tangible motor instructions posed a significant hurdle.

These challenges shared similar characteristics, as their verification and fine-tuning could only be accomplished once all other components were operational. Additionally, ensuring the proper functionality of each of these aspects was a time-consuming process that required meticulous attention to detail.

3 Software

3.1 Backend

The backend component of our website was implemented using Node.js. Its primary responsibilities encompassed mapping the maze layout, mapping the discovery progress, determining the shortest path within the fully mapped maze, database management, and aiding motor instructions. In the subsequent section, we will delve into the features and methods employed in detail.

Framework Choice

Initially, we opted for a separate Flask backend, for its simplicity and lightweight nature. However, our subsequent transition to a Node.js backend with the Express framework was motivated by its seamless integration with our React frontend. Running both frontend and backend with `npm run build` and `node server.js` proved easier than the Flask and React combination.

We did not extensively invest time in discovering the optimal framework. Instead, our focus was on optimising our algorithms to achieve greater efficiency, ultimately saving more time and reducing latency.

Furthermore, it only required a single Docker container to build the entire application (as seen in the Dockerfile in our GitHub repo). In contrast, the more intricate process of building individual Docker containers and linking them through Docker Compose for a robust Flask backend and React frontend deployment proved time-consuming. Furthermore, ECS deployment with Docker Compose presented challenges and additional costs, while deploying the simplified single container to Lightsail was cost-effective and hassle-free.

Maze Mapping

To facilitate maze mapping, we opted to represent the maze using a two-dimensional array structure, where each element in the array corresponds to a specific small area within the maze, denoting it as

either a wall or an empty space. Given the precise dimensions of the maze (2.4m by 3.6m), we chose to map it with 240 arrays, each with a length of 360. As a result, each value in the array signifies the "status" of a 1 cm^2 square in the maze. This division of the maze into small squares enables the establishment of a coordinate system, allowing us to designate a specific corner of the maze as the origin (0,0). Consequently, we have a standardised approach to determine the rover's location and orientation within the maze.

Discovery Mapping Initially, our approach involved solely mapping the maze using a single array structure. However, upon recognising the need for improved tracking of discovery progress, we implemented two separate array structures: one for maze layout and another for tracking the progress of discovery. Through testing, we discovered that due to camera inaccuracies and variations in camera position, the actual discovery radius extended slightly beyond the exact area mapped by the rover. We deemed this larger radius beneficial to minimise unnecessary trips, reducing the risk of the rover traversing already confirmed empty spaces that were not explicitly mapped. Utilising a separate discovery mapping not only enhances pathing efficiency but also provides an intuitive visual display for the user. While employing a slightly larger radius introduces the possibility of falsely classifying the maze, we mitigate this risk through a continuous value assignment.

To quantify the amount of discovery progress, we employ a continuous scale ranging from 0 to 100. This quantifies the varying nature of the maze squares, reflecting our confidence levels in their classification. Initially, all squares start with a value of 0. However, if a square falls within the rover's position or the area of the camera, it is immediately assigned a value of 100, indicating certainty in its mapping. Additionally squares within a 2cm radius of the immediate discovery area are assigned a value of 30. With each subsequent pass over these squares their values are incremented by 30. The maze traversal algorithm treats squares with a value strictly greater than 50 as discovered.

The choice of a 2cm distance, a value of 30, and the incremental increase of 30 were selected based on our initial considerations. However, these values will be refined through testing and integration to strike the optimal balance between accuracy and redundancy removal.

Maze Array Value Assignment

Initial Iteration We considered using a single array with characters representing different elements of the maze, such as "W" for walls, "S" for spaces, "R" for the rover, and "P" for paths. However, we realised that this approach was not optimal due to the binary nature of the mapping, which was highly susceptible to camera inaccuracies. There was a significant risk of falsely classifying spaces because the most recent update could overwrite previous mappings.

Final Iteration In order to mitigate the impact of false readings, we implemented a matching system for the discovery array, utilising a continuous spectrum for value assignments. Similarly, the maze mapping array employs a value scale ranging from 0 to 100, with 0 representing an empty space and 100 indicating a wall.

To determine the values, we initially employed a linear increment system for simplicity and testing purposes. Squares initially discovered as empty spaces were assigned a value of 30, while squares identified as walls received a value of 70. Subsequent discoveries amended the assigned value by increments of 10. An undiscovered position is assigned with a constant value of -1, and the rover and path squares have constant values of 200 and 300 respectively.

However, we are currently exploring the potential benefits of an exponential function approach. Under this approach, each discovery would be initialised to a value of 50, with subsequent readings exponentially converging towards values of 0 or 100. Through testing, we aim to evaluate the benefits of both algorithms to determine the most effective approach.

"Filling" Algorithm

To align the inference method used for discovery mapping with maze mapping values, we employ a bridging process. If a square is "discovered" in the discovery mapping but lacks a value status, it is assigned the average value of its four adjacent squares.

We are exploring improvements for inferring values, taking into consideration the surrounding context. Our rationale is that if there are walls in the slightly larger vicinity (within two squares in the same direction, such as above and below instead of directly adjacent), there is an increased likelihood that the square in question may also be a wall. This is important because adjacent squares may exhibit false discoveries, where a value of 30 is incrementally adjusted to 40, yet still classified as an empty space due to being below the threshold of 50.

One potential algorithmic implementation uses a weighted average with a larger radius, adjusting the weighting of the adjacent squares based on nearby walls. Extensive testing is needed to determine its effectiveness in improving mapping accuracy.

Endpoints

Our system currently offers several public endpoints with diverse functionalities. We have standardised our data payloads to have json formatting for easy processing. In Appendices C and D, we have provided an extensive overview and examples of the current endpoints and formats. Note that due to the structure of our backend and data formats, it makes it really easy to include new data and make changes to endpoints.

Database

We have two databases in our system, namely "developer_details" and "payloads". The "developer_details" database stores developer information, utilising the unique primary key "dev_id" and its associated unique "_api_key" for valid payload submissions. On the other hand, the "payloads" database stores previous mappings for maze mapping and discovery progress, including timestamps for updates. Its schema consists of a primary key "payload_id" generated from a hash of "developer_id" and "timestamp" to ensure uniqueness. Additionally, it includes attributes such as "timestamp", "maze", "discovery", and a foreign key "dev_id" referencing the corresponding "dev_id" in the "developer_details" table.

We were going to use the database as a fallback to the latest copy of the maze if something broke in our system. However, this fallback feature was instead implemented directly in our backend. Therefore we can use the payloads table for additional features - e.g. something like a replay feature, or for us to look at past sessions.

Shortest Path

In our backend system, we have implemented a straightforward Breadth-First Search (BFS) algorithm that determines the shortest path from point A to point B. This serves multiple purposes, primarily allowing us to overlay the website with an optimised maze route. Additionally, it aids the rover in identifying the nearest unexplored square, optimising its exploration strategy.

Maze Traversal Algorithm

Initial Iterations The initial challenge involved determining the optimal method to guide the rover through the maze. We researched standard algorithms such as left-wall following, Pledge, and Floodfill. However, these algorithms proved inadequate for our maze due to its loops and curved paths. As a result, we developed a modified Depth First Search approach using a graph data structure. Each

junction and dead end were represented as nodes in this structure, storing distance information from the beacons. While this method allowed the rover to traverse a theoretical graph, the detection and classification of nodes posed a significant challenge due to the limited data received from the FPGA.

Final Iteration The final iteration of our algorithm employed a refined version of the depth-first search (DFS) algorithm. The fundamental principle behind our approach involved the rover continually examining the immediate adjacent squares to identify the closest undiscovered area that is both unexplored and determined to be devoid of any obstacles. This recursive process enables the rover to efficiently navigate through the maze.

Initially, the rover starts in a random location and moves forwards in a random direction without obstacles. From then on, at each point in time, the FPGA camera analyses the immediate surroundings of the rover and provides updates on the position of the rover and the status of obstacles using the "lines" array in the update payload. Based on this information, the rover updates its mapping of the maze. With the updated mapping, the server identifies the closest undiscovered square and determines the shortest path to reach it using the shortest path algorithm.

This process is repeated iteratively, with the rover continuously updating its location, maze mapping and the server having an overview of the rover's surroundings. By iterating through this algorithm, the rover explores and discovers new squares in the maze until eventually, all the squares have been discovered.

This iterative process ensures that the rover systematically explores the maze and maximises its coverage, gradually uncovering the entire maze layout.

Security

Since the website is publicly hosted, this means that anybody can post the endpoint and update our methods. Therefore we added security by only parsing data from payloads with valid dev-id and x-api-keys (this is stored in a table in our database). Furthermore, all valid payloads are hashed with a salt with SHA256 and checked.

3.2 Client

Overview

The client-facing website was developed using a stack consisting of React with TypeScript for the frontend and Node.js for the backend. The final website is packaged into a Docker container and deployed as an instance on Amazon Web Services (AWS).

Iterations

Throughout the development process, multiple iterations of the website layout were implemented, as depicted in Fig. 13 - Fig. 14 showcasing the initial and final versions. Initially, the BalanceBot homepage solely encompassed fundamental features, such as displaying the maze layout. However, substantial improvements were made subsequently, leading to the incorporation of a set of features outlined below.

Display Features

- Current timestamp
- Last updated timestamp: Indicates the timestamp when the website was last updated, allowing users to track the recency of the data presented (useful for noting if there is a long delay between

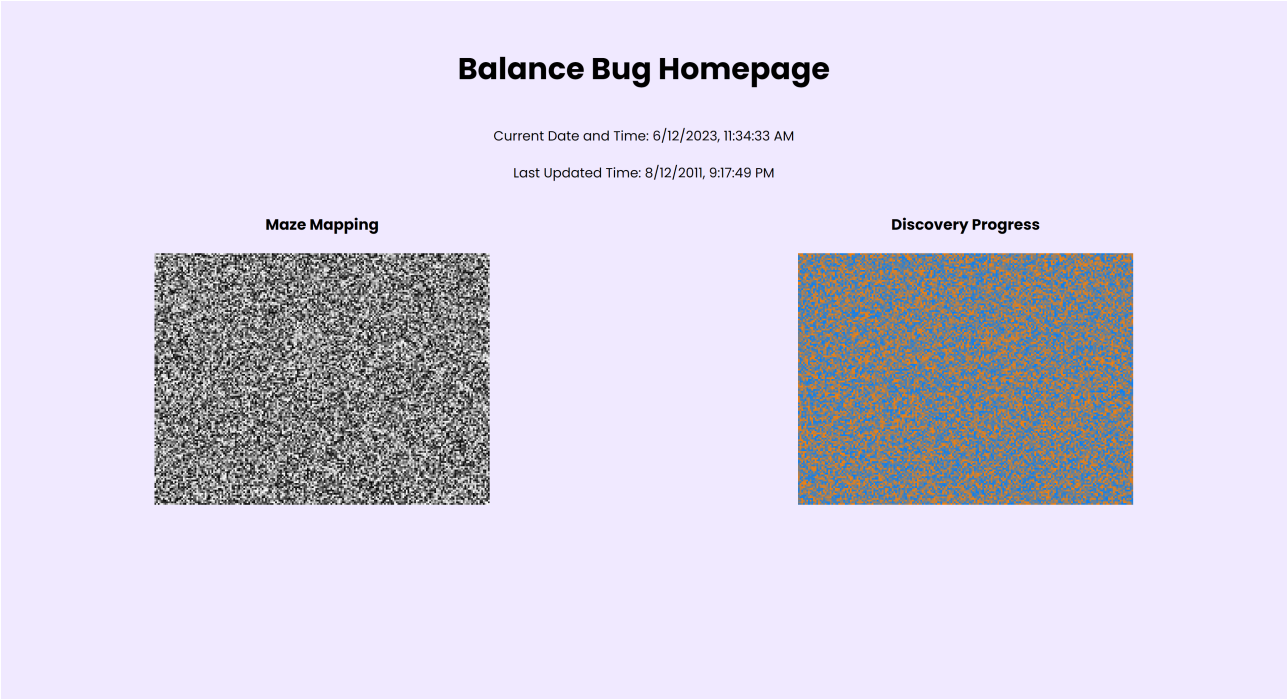


Figure 13: Initial Iteration

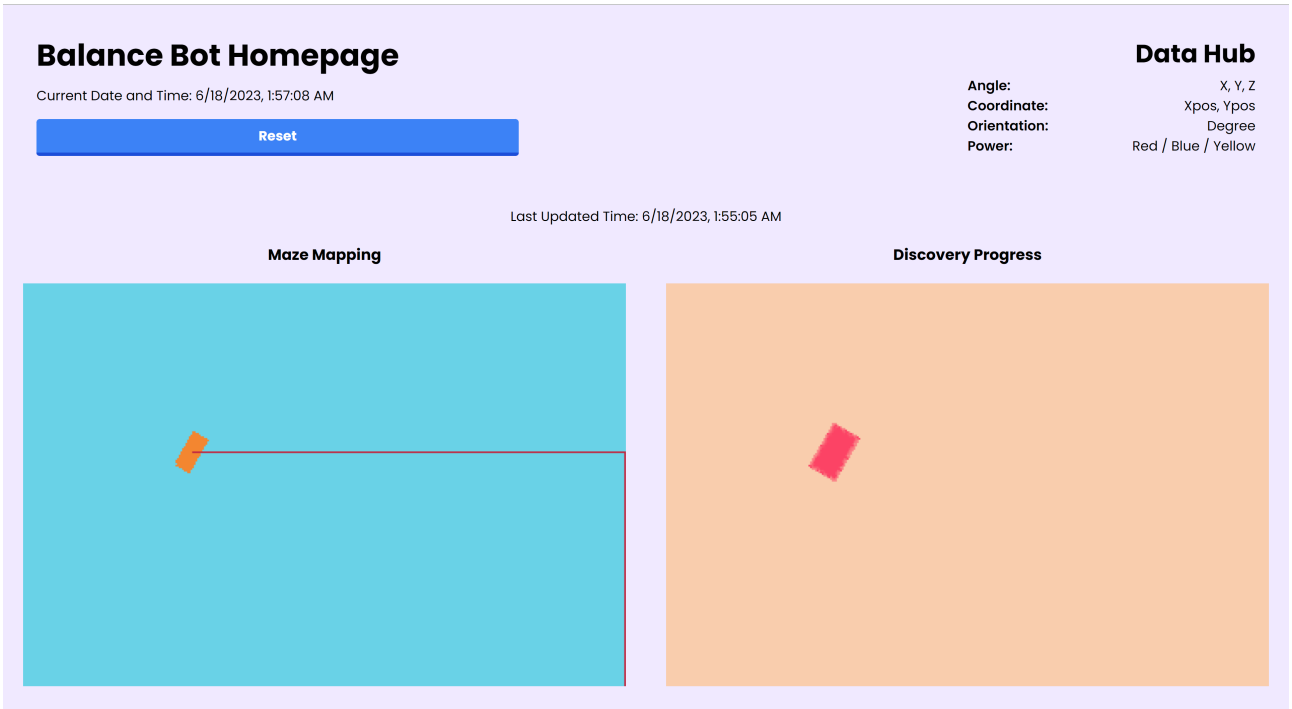


Figure 14: Final Iteration

updates - helps detect possible errors)

- Reset button - resets mapping and datahub, connected to the /api/reset endpoint.
- Rendered display of maze mapping overlaid with shortest path and discovery progress: Visualises the maze layout, highlighting the shortest path and indicating the progress of exploration.
- Datahub: Various readings from the rover (e.g. power, position etc.)

Design Decisions

Rendering The maze is displayed on the website using an array of arrays, where each value represents a colour. The array values are mapped to a desired colour spectrum, creating a visually informative representation of the maze. Although slower than displaying raw values, the visual representation is more intuitive. To optimise rendering speed, a 2D HTML canvas element is used to render each pixel.

Update Frequency Initially, continuously listening for updates through a WebSocket on the backend was considered but posed challenges due to high update frequencies and the array's large size. This resulted in slower frontend rendering times (approximately 1 second). As a compromise, the frontend now fetches data from the backend every second. Although the update frequency may seem low, it meets our requirements, providing near real-time updates while optimising website performance and rendering efficiency.

Debouncing Additionally, a debouncing technique is employed to optimise the frequency of data fetching, ensuring an optimal rate of updates. This approach avoids excessive re-rendering on the frontend and only triggers a re-render when the current display is different to the newest data received. The debouncing combined with rate limiting provides for a much better balance between performance and experience.

Error Catching To ensure robustness, the Node.js backend maintains a backup of the last updated maze version, ensuring uninterrupted functionality in case of issues. The website will continue to display the most recent and accurate maze mapping available.

Additionally, the website actively updates the corresponding AWS database row to indicate the successful processing of the maze mapping. This approach enables us to keep track of the status and progress of the mapping process, serving as a valuable troubleshooting and monitoring tool.

Furthermore, both the frontend and backend incorporates extensive error handling mechanisms. Error handling is implemented with every function, including rendering functions, frontend fetching methods, database update methods and data parsing. By catching and handling errors at different stages of the application, we ensure that any encountered errors are gracefully handled and returned as appropriate responses, preventing them from disrupting the overall functionality of the website.

Through these robustness measures, we aim to provide a reliable and resilient user experience, mitigating potential issues and minimising their impact on the functionality and performance of our client application.

Logging We make sure to log all errors, updates and payloads we receive with their respective timestamps in the backend. This drastically helps with debugging and keeping track of things both locally and on production. This was also utilised a lot in the testing process for the website and backend to make sure the behaviour was expected.

Hosting We opted to host the Docker container on an AWS Lightsail server in the EUW 2 (London) region. Choosing the closest AWS server minimises latency and ensures a smooth user experience. We selected AWS Lightsail due to its simplicity in the deployment process. The logging capabilities provided by AWS Lightsail also proved helpful during the development and testing phases. Additionally, since the Docker container is lightweight and required only basic capabilities, we found that the extensive features offered by an EC2 instance were not necessary.

Another major reason we chose AWS Lightsail was because of its cost-effectiveness. As a small-scale project, we were able to take advantage of the Micro (1 GB RAM, 0.25 vCPUs) $\times 1$ node, which

falls under the free tier offering. This allowed us to host our Docker container without incurring any additional charges for the server resources.

3.3 ESP to FPGA and Server

WiFi The ESP32-WROOM module that we are using has built in WiFi capabilities. This allows it to act as an Access Point (i.e. a router of its own network) or be in Station Point Mode (allows the ESP32 to be a client of a network). The project requires the ESP to send data over the internet from the FPGA to an AWS server as to map the maze. Using the arduino library "WiFi" the WiFi mode was set to STA, and the SSID and password of the network used was that of a team member's personal hot-spot. This was due to the fact that both Imperial-WPA and eduroam required username and password to connect, which is not supported by the `WiFi.begin()` function.

AWS Once the ESP32 has established a WiFi connection it is then ready to be able to send JSON data to our server in the format shown in Data Formatting. To do this we used the HTTP protocol so we could make use of POST and GET requests which had already been set up in the web app design. We would post to the endpoint /api/update, and get from (NEED TO SET UP ROVER INSTRUCTION GET REQUESTS) using the functions `void httpPost()` and `void httpGet()` respectively.

FPGA Getting the ESP32 to receive data from the FPGA proved to be a challenge. We agreed to use UART communication as there was dedicated pins for this, and serial communication could be established between these two devices with this protocol. The idea is that the FPGA will send eight 32 bit words to the ESP32, which will then decode this information and forward it to the server in the required format.

4 Vision System

4.1 Design

Core Requirements

The primary function of the Vision module revolves around capturing object information through the camera and transmitting it to the server via the ESP-32 for maze mapping and path finding. This entails the process of taking raw data from the camera sensor, converting it from the Bayer pattern image to a comprehensible RGB format, and conducting image processing to identify relevant objects. Consequently, we can break down these tasks into a set of fundamental requirements as outlined below:

1. Capture Raw Data from Camera hardware module connected via GPIO and convert it to RGB.

2. Detect objects of interest within the current frame, and draw a bounding box around them, in our case, the 3 beacons and illuminated white walls.

3. Send the pixel coordinates of the bounding boxes to a soft core processor(NIOS-ll) for processing.

4. Calculate the location of objects relative to the FPGA camera based on the pixels of the bounding box.

5. Send the location data to the ESP-32 to be forwarded to the backend server.

Hardware Organisation

The Vision module is made up of 2 hardware components which were provided by the department:

- Terasic DE10-Lite, an Intel MAX 10 based FPGA board.
- Terasic D8M-GPIO, a camera package with an OV8865 image sensor that interfaces with the FPGA through it's GPIO connectors.

The FPGA's configurability grants it greater flexibility compared to other embedded systems that rely on general-purpose processors. Additionally, the FPGA efficiently handles both the streaming and processing of high-resolution images, ensuring minimal compromises in terms of frame rate and data speed. This is achieved through the utilisation of concurrent operations and dedicated signal processing blocks, such as multiplication. The FPGA also includes a 4-bit VGA output, which is handy for live debugging of object detection. It features a connector designed specifically for an Arduino Uno R3 shield, enabling seamless interfacing with the ESP-32.

Initial Design Choices

The project specifications entailed handling various colours and objects that required different processing approaches. One specific shape was a hemisphere representing the beacons, while the walls appeared as thin cylinder shapes in 2D when captured by the D8M camera. The beacons had a fixed size, allowing for straightforward distance calculation between their top and bottom or left and right edges in pixels. This pixel distance could then be converted into a known physical distance. Since there would only be one instance of each colour (red, blue, and yellow), determining this distance was relatively straightforward, as no two objects of the same colour would appear in the same frame.

Detecting the walls presented a different challenge as they could have diverse shapes due to the maze's nature, and multiple instances of the same colour could be present. The team considered using the Connected Component Labeling (CCL) algorithm (The MathWorks, Inc. 2023), an application of graph theory, to calculate the positions of the walls. CCL assigns unique labels to subsets of connected components based on a given heuristic. However, none of the team members had prior experience with this complex algorithm. As a result, a decision was made to employ simpler, logical algorithms to identify the presence of walls. One idea involved dividing the image into distinct "zones." By detecting white pixels in specific areas of the image, the presence of a wall in that corresponding region could be determined. The image was partitioned into left, middle, and right sections, allowing for individual analysis of each area to ascertain nearby walls and provide appropriate instructions to the BalanceBot via ESP-32 to avoid turning or proceeding in specific directions.

The positioning of the camera relative to the BalanceBot was a crucial design consideration. The OV8865 image sensor, with a field-of-view (FOV) of 68 degrees, exhibited difficulty in capturing nearby objects during camera testing. For instance, when the camera was placed 20cm above the ground, parallel to the maze floor, it failed to detect a horizontal white line located approximately 40cm ahead. To address this limitation, several options were explored, including placing the camera closer to the ground or employing mirrors or a fish-eye lens to enhance visibility and prevent missing nearby walls. During the design stage, all options were deemed feasible, and the team deferred the final decision until the implementation stage to determine the most accurate and effective solution for wall detection.

The came in three different colours: red, yellow, and blue. Preprocessing of the captured camera image was deemed necessary to optimise colour detection for object identification. To achieve this, a transformation from the RGB colour space to the HSV (Hue, Saturation, Value) colour space was chosen. The HSV colour space has demonstrated superiority in image processing tasks, such as colour segmentation in road sign detection for autonomous driving, due to its ability to handle illumination variations, unlike RGB (Mohd Ali et al. 2013). For instance, in the RGB space, a continuously brighter blue beacon would involve changes in red, green, and blue values until reaching #FFFFFF. In contrast, in the HSV space, only the value component, which represents brightness, needs to be adjusted as the colour becomes brighter. This simplifies the process of classifying colours within specific ranges by the image processor, facilitating effective colour-based object classification.

4.2 Implementation

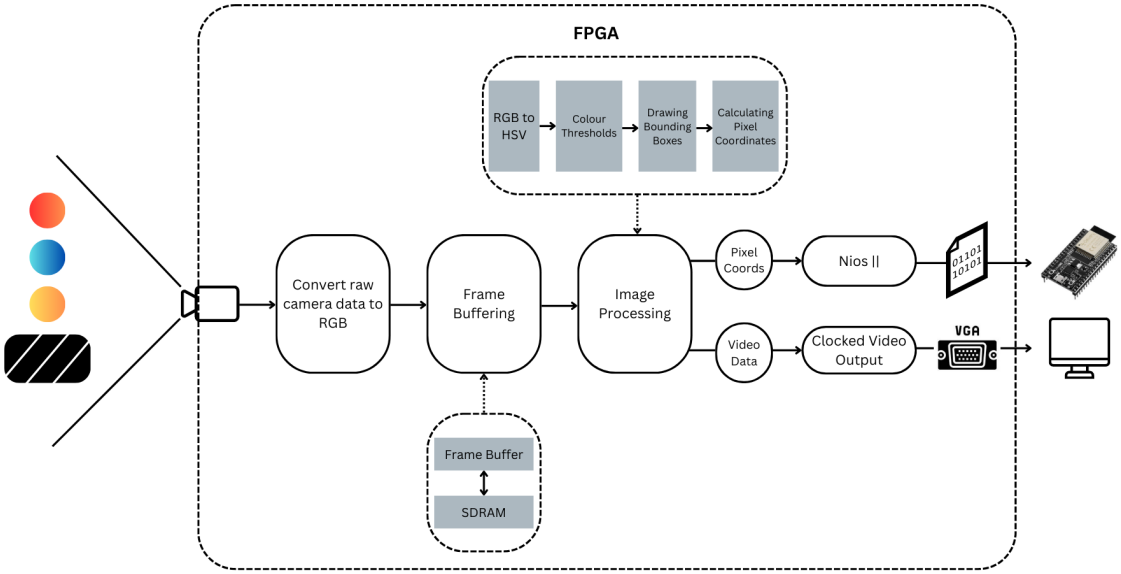


Figure 15: Image Processing Pipeline

Starter Project – Image Capture and Processing

The first part to getting the Robot Vision to work, was understanding how the starter project worked in finding and bounding red pixels that were detected by the camera. Raw data from the camera was taken in Bayer filter form, transformed into RGB video packets, where the frames were buffered to allow asynchronous processing of the camera and output data. The image data was then processed and converted to serial data for output over VGA as well as through the NIOS-ll soft processor. The VGA output allowed us to see what the camera was detecting, and how it would place the bounding box around the red colours in the frame when SWITCH0 was toggled. The NIOS-ll terminal outputted the pixel coordinates in a 32-bit hexadecimal format, whereby 11 bits was assigned to the minimum and maximum x and y coordinates for a total of 4 coordinates. The message FIFO was inputted with 3 32-bit values, the first being the RED colour identifier, and the following two words containing the 4 coordinates.

RGB to HSV conversion

HSV (hue, saturation and value) is expressed as 3 values where:

- Hue – 0-360 degrees
- Saturation – 0-100
- Value – 0-100

Conversion from RGB involves many different calculated values, including the maximum and minimum RGB colours present, the difference between them, the difference between two RGB values etc. The mathematical operations for conversions can be found in abundance online and below is the calculations we used for conversion(https://www.rapidtables.com/convert/color/rgb-to-hsv.html):

$$R' = R/255$$

$$G' = G/255$$

$$B' = B/255$$

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

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

$$\Delta = Cmax - Cmin$$

$$H = \begin{cases} 0^\circ & \Delta = 0 \\ 60^\circ \times \left(\frac{G'-B'}{\Delta} mod 6\right) & , C_{max} = R' \\ 60^\circ \times \left(\frac{B'-R'}{\Delta} + 2\right) & , C_{max} = G' \\ 60^\circ \times \left(\frac{R'-G'}{\Delta} + 4\right) & , C_{max} = B' \end{cases}$$

$$S = \begin{cases} 0 & , C_{max} = 0 \\ \frac{\Delta}{C_{max}} & , C_{max} \neq 0 \end{cases}$$

$$V = Cmax$$

Figure 16: RGB to HSV conversion formulae

Floating point calculations in System Verilog are computationally expensive. Our compilation times almost doubled from 215s to 348s when we had only implemented the RGB to HSV conversion. To reduce some of the complexity, we removed the need to classify an RGB value between 0 and 1, and instead left it in the default RGB format between hex 0-FF. We also removed the need to classify saturation and value between 0 and 100% as this would also require a floating-point operation, so our S and V values remained between 0 and 255. There were still some operations that needed to be included for the conversion to remain accurate, such as the division that is required when calculating hue or saturation.

System Verilog Implementation

To start, we needed to work out Cmax and Cmin. This was done with a nested conditional to find the maximum and minimum RGB values. If green was greater than red, green would be compared to blue, the outcome of which would be green if true and blue if false. The same was done for Cmin with inverse operators.

Previously in the starter project, red was detected by using the most significant bit of the RGB value for red, and seeing if it was high or not. This was then logical ANDed with the inverse MSB's of green and blue. We could now use the HSV for red, assigning red if the Hue was less than 10 degrees, and saturation/value was greater than 128. Similar principles were applied to blue, yellow and white, which can be found in Appendix A.

Filtering

When detecting our colour thresholds, we did not want the camera to pick up single instances of red, blue, yellow or white. This was because in our testing, which did not involve using the maze but in the open laboratory, the scattering of white light from the ceiling would cause the camera to aggressively place different colours at different coordinates for each frame. An example of this is shown in Fig. 17 - Fig. 18.

To correct the unpredictability of scattered light and how the camera detects it, we implemented a filtering algorithm that would only assign an object as red, if and only if the previous 3 pixels were also red. As the image is processed row by row, this was trivial to add and involved the use of 3 shift registers to store each pixel's colour as the image data was being processed. And example of the effects after filtering is shown below, along with code in Appendix A.

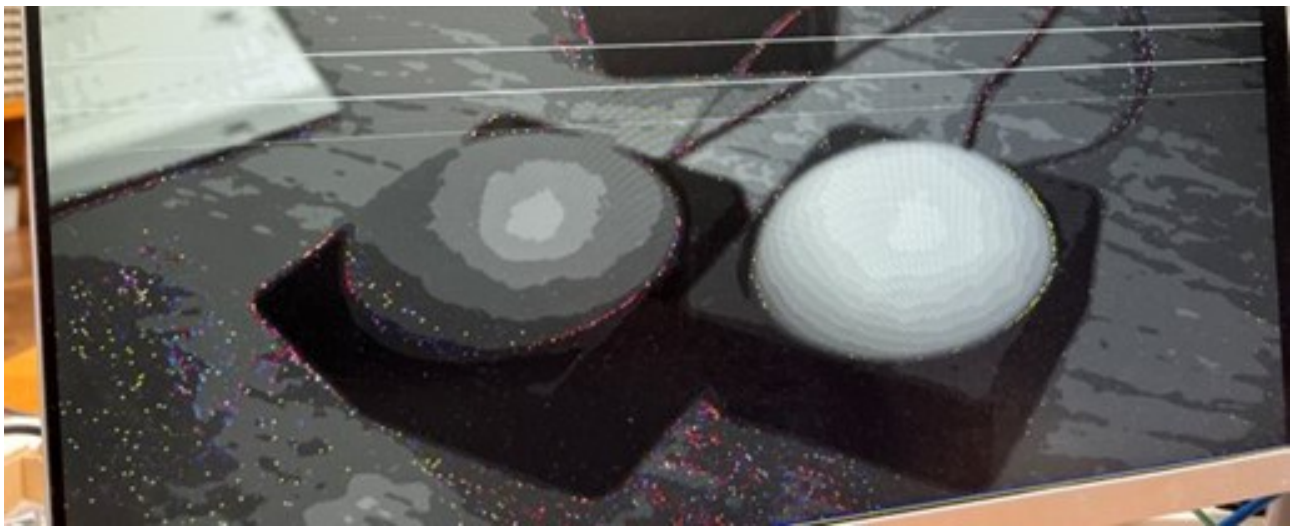


Figure 17: Before Filtering

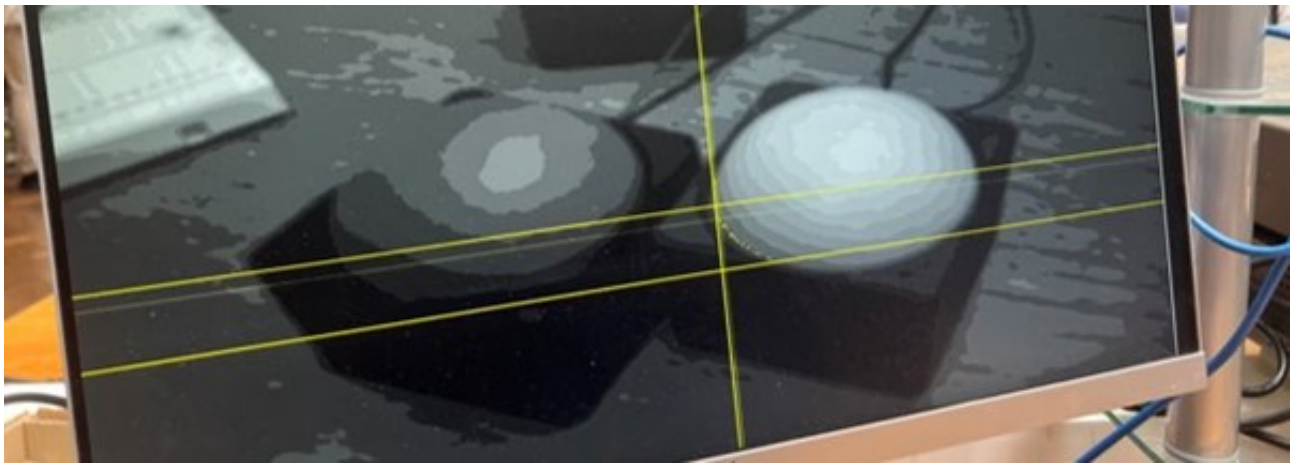


Figure 18: After Filtering

Wall Detection

Initially, the objective was to enable the camera to capture and map the entire maze. However, despite adjusting the gain and exposure of the OV8865 sensor, the camera struggled to recognize the white walls at the maze's edges, resulting in their appearance as yellow. Consequently, a decision was made to focus the camera on processing the immediate surroundings of the Rover.

To effectively detect white walls, the image resolution was divided into three sections. Two 100x100-pixel sections were placed in the bottom corners, while a long, narrow rectangle occupied the central region. The top half of the screen was disregarded since the camera would never detect white pixels in that area when facing parallel to the maze floor. Extensive testing was conducted under various conditions, including clear paths (walls on the left and right), dead-ends (walls on all sides), and no walls (dark room).

The primary objective was to determine the presence of walls rather than obtaining precise pixel coordinates. Therefore, the identification of a single white pixel coordinate served as an indicator of wall presence.

Another challenge arose regarding determining the direction in which the walls were positioned, with only three designated "zones." For example, a wall on the left might run parallel to the left edge of the screen, or a wall ahead might be at an angle of 20 degrees relative to the Rover's front. To address this, each corner was divided into two zones, resulting in a total of six zones. These zones were

diagonally connected so that the presence of a white pixel in both connected zones would indicate a wall immediately adjacent to the Rover's path, running parallel to it.

We utilised trial and error to tweak the white zones to ensure the Rover can determine with certainty the location of a wall. The middle zone was moved closer to the bottom, with a thinner height and length to make sure it did not interfere with the side walls.

On Eclipse, the message FIFO sends 32-bit words, each containing two 11-bit coordinates, the first and second zones of a corner. We used logical AND to check whether both coordinates were different from 0, if they were, a signal "1" would be placed in the MSB of a 3-bit word. Bits [2:0] represents [left, middle, right] where if the output was 111, this means the robot has detected a wall on both sides and in front, i.e., a dead-end.

Distance Calculation to the Beacons

This part of the problem was relinquished to the software (Eclipse) of the NIOS-ll. As mentioned before, the singular presence of each beacon meant we would not run into the same issue we had with walls. We used various setpoints to see how many pixels the beacons took up at certain ranges. For example, at 30cm away (relative to the camera), they occupied 80 pixels of height and width, and at 15cm away they used 160 pixels. From the inverse proportionality between the pixel height and distance, we calculated our constant of proportionality, k, as 2400. Due to the shape of the beacon being a hemisphere, the diameter we calculate using pixels does not exclusively require the x or y maximum or minimum coordinates, either can be used. We settled with the difference between the y values.

An issue arose with regards to whether the bounding box placed around the beacons used to calculate the coordinates, and hence the distance from the camera, is accurate enough. In varying lighting conditions, the camera failed to detect the beacons, resulting in unknown distance measurements. Additionally, when the beacon was powered by a very small current, creating a non-uniform colour distribution across their surface, only a very small section of the beacons would be correctly detected. Consequently, the calculated distance from our algorithm showed the beacon was very far away.

To address this issue, we chose certain currents to power each beacon colour, so that the circumference of each would fall within the HSV colour thresholds we had set. This ensured that no matter the lighting conditions, if the beacons received the correct current, their distances can always be accurately measured within an error of +-5cm.

The output data to the ESP-32 was configured as a 32-bit hexadecimal, the top 8 bits representing the hexadecimal value of the colour, and the bottom 8 bits representing the distance in centimetres. For example, a red beacon detected 80cm away would be outputted as 0x72000050. There were a total of 3 words outputted to the Nios-ll terminal, each representing a beacon, followed by a 3-bit binary word for the walls.

Evaluation of our Vision system

Throughout the entirety of working on the Robot Vision, there were many issues we faced which are listed below:

- Difficulty connecting JTAG server between Programmer modules on local host and ee-mill servers. Thus, we were unable to progress with flashing the FPGA for standalone operation. This somewhat affected our progress as we had to leave flashing for the last step, which constrained our time.
- Various communication issues between the flashed FPGA and NIOS-ll software. This made it very hard to obtain consistent results, as we would randomly face unexpected results, such as green and pink camera streams, shifting image resolutions where the entire camera resolution

was split in the middle, high gains and exposure when downloading the .elf file even though we had set the gain and exposure parameters to our desired values, zoomed in/out of focus frames.

- Long compilation times. When it came to narrowing down the thresholds for the HSV ranges of each beacon, it took a lot of trial and error to get the perfect ranges and to ensure none of the thresholds overlapped. This meant the entire top-level module would have to be recompiled whenever for example, the HSV of red needed its Hue reduced by 10 degrees. Compilation times were generally above 6 minutes, which was frustrating.
- Elf file working only on the first build. When building and running the project in Eclipse, the Nios-ll terminal would only output meaningful data (32-bit coordinates) on the first build, whereby if we wanted to make a change in eclipse to process the coordinates from the FIFO, and test again, the Nios-ll terminal would not work the second time. This meant we had to reflash the FPGA, and then build again. However, even this step would not work, and the camera would behave in the manner described in point 2 above. This also made it very hard to test our image processor consistently for accuracy.

The algorithms used were simple, but adequate for our intended use cases, and for the purpose of the Vision system. With our use of filtering, strict HSV ranges and multiple zones for wall detection, there is a low false positive rate outputted by the FPGA and camera. If there was more time, we would try to use a more rigorous and robust algorithm for calculated the distance of the beacons from the camera, rather than a constant term for proportionality. Our algorithm for wall detection works very well and could perhaps be made more accurate with more zones. We would also like to incorporate the use of mirrors/fish-eye lens to better allow the Rover to see nearby walls, and to not be so dependant on it's height or the mechanical design.

5 Mechanical Design

We collectively came to the decision that designing our own product would meet the non-functional requirements of being robust, as well as efficiently constructed.

It was decided that an inverted pendulum approach was desirable as it would be the most efficient way and least complex way to balance the product (Kim & Kwon 2015). Once this was established, initial designs were sketched as illustrated in Fig. 19.

The main design point we wanted to ensure was present on our product was the fact that it has an enclosure around the circuitry and components. Due to the nature of the task that the product will be undertaking, an enclosure would ensure that if the rover was to fall then none of the components would be damaged which is desirable as it will allow us to robustly test the product.

These sketches allowed us to get an idea of what we wanted to achieve with our design. The 3D design of this iteration can be seen in Fig. 20.

After constructing and testing (Fig. 20) our target design, it was concluded that the given motors did not provide enough torque for the wheels, ultimately meaning it could not balance or move - which is a main requirement. We suggested that we buy new motors however, this would mean we would have to buy new motor drivers and since a motor driver PCB had already been provided, we decided that we would compromise and modify the chassis that was included in the starter kit.

6 Integration

Integration involved bringing together all of the sub-systems and ensuring that they all work together.

We had to get all of the data from the FPGA to the ESP32. We decided to use UART to send the data to the board. We also had to make sure that our Node backend which was running on our

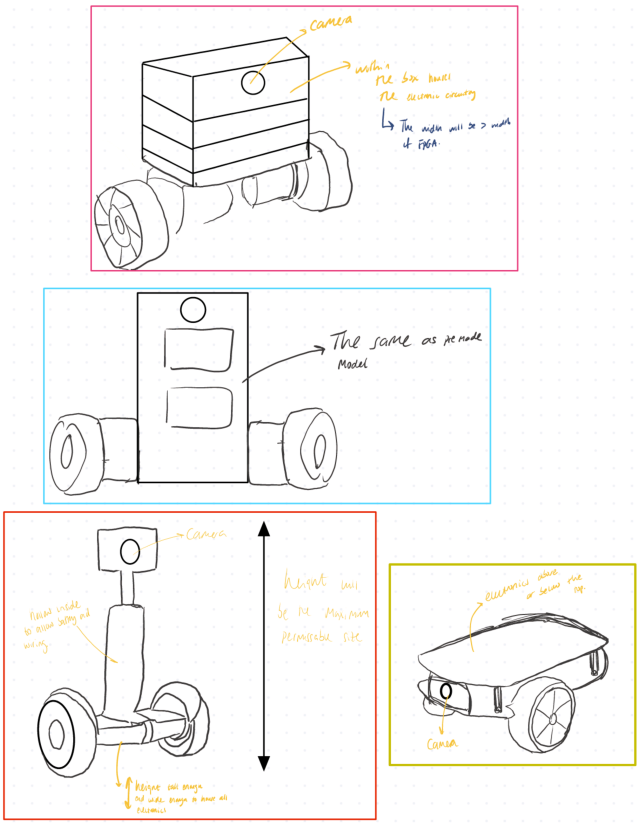


Figure 19: Initial Design Sketches

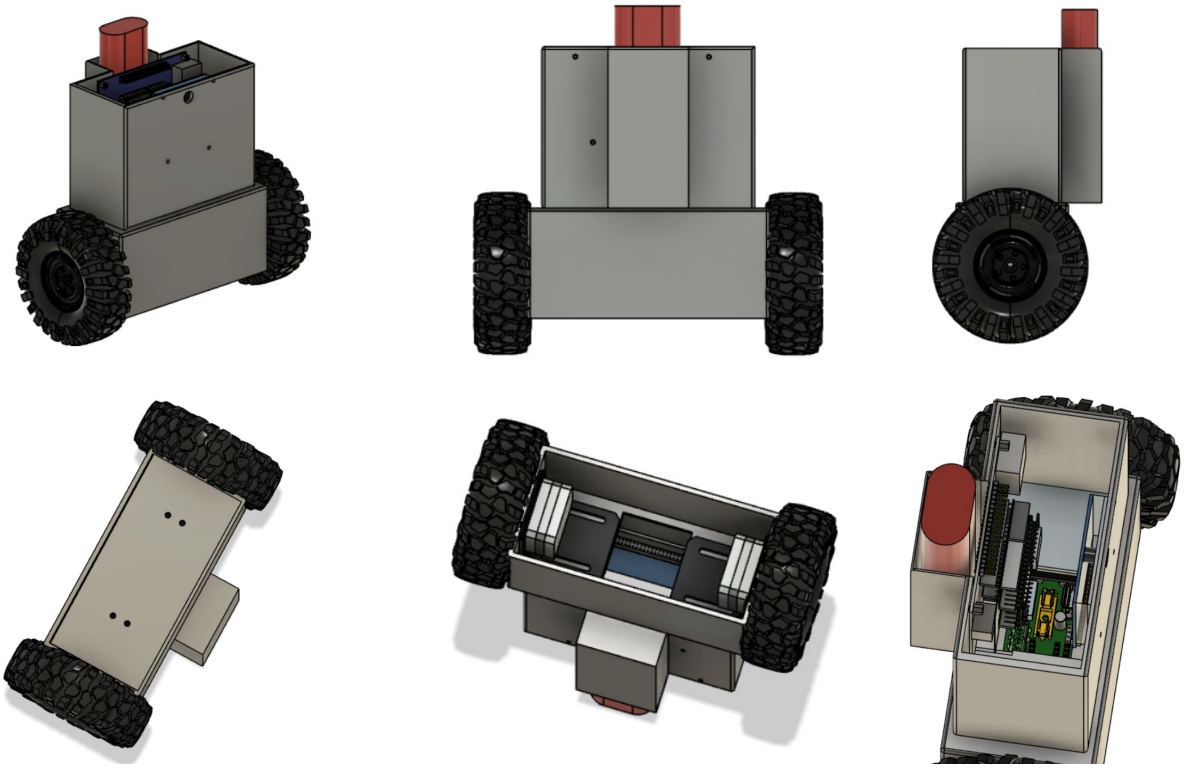


Figure 20: Multiple 3D Perspectives of the Target Design

AWS instance was able to communicate with the ESP-32 and also the user's computer with the React frontend. We also had to assemble all of the hardware components on the physical chassis of the rover and ensure that the control system would still work with everything attached.

This task is still ongoing and we aim to complete this by the demonstration on Friday.

7 Testing

Each subsystem was individually tested prior to integration. Once we complete integration, we plan to thoroughly conduct tests on the final rover.

7.1 Energy System

Throughout the design and construction of the energy system, a continuous process of testing was undertaken. This involved conducting individual tests on each system component, including the LED drivers and MPPT algorithm.

Prior to conducting the performance evaluation of the LED drivers, it was essential to tune the PID controller appropriately. This tuning process involved individually adjusting the three components of the PID controller in order to optimise the system's response. Throughout the testing phase, a specific scenario arose where setting the initial proportional gain (P) value too low resulted in the LED driver's failure to reach the target current. However, by appropriately increasing the P value to 45, the LED driver demonstrated a precise and accurate response, achieving the desired current output. Subsequently, the integral (I) value was adjusted to ensure a rapid and accurate response in attaining the target output current. The value of D was observed to be inconsequential and was set to 0.

After conducting initial testing of the MPPT (Maximum Power Point Tracking) algorithm, we observed unexpected results where the duty-cycle consistently saturated at the upper limit of 0.99. To address this issue, a delay was introduced as a solution.

7.2 Control System

After the completion of the initial designs, an extensive testing process was conducted on the control system. To tune the PID values, we initially identified a point where the system achieved a reasonable level of stabilisation, ensuring that it remained stable for a sufficient duration to gather data points. The data points that were being collected included the pitch angle and time, allowing us to visualise how varying the PID affects the angle of tilt.

The ultimate aim for this method is to find a point at which there are periodic oscillations and then employ the Ziegler-Nicolas method to fine-tune the values. Following the method, PID gains were initially set to zero and the proportional gain Kp was gradually increased until the system exhibits sustained oscillations. The value of gain at the point is known as the ultimate gain Ku at the point. Then, the period of these oscillations Tu were measured and the ultimate period $Pu = 4 \times Tu$ was calculated. The PID gains could be derived from the obtained values.(Bennet, James. and Bhasin, Ajay. and Grant, Jamila. and Chung Lim, Wen. 2023).

- *Proportional gain* $Kp = 0.5 \times Ku$
- *Integral gain* $Ki = 0.45 \times \frac{Ku}{Pu}$
- *Derivative gain* $Kd = 0.125 \times Ku \times Pu$

7.3 Vision System

To ensure we calibrated the HSV Thresholds for our colour detection accurately, we used the a screenshot of the output VGA port image, which was then imported into a MATLAB tool known as the Colour Thresholder App. This application allows colour images to be segmented by changing thresholds in different colour channels, supporting RGB, HSV, YCbCR and L*a*b*. The tool provides an accurate way to select the values for each of the HSV channels, as the binary mode can be used to ensure that the regions of interest are maximised and the other regions are ignored.

7.4 Website and Server

Throughout the development process, the website was consistently tested locally. This was done by using Postman to send sample payloads to the http://localhost:3000 endpoint to see if the desired response is received. One advantage of using Postman to test payloads was that we could see the latency in the responses. It was around 50-100ms for a 200 response from the prod AWS server which was definitely acceptable. To test the database, we used TablePlus to initiate a connection to AWS RDS on Prod and we can track the live contents of the database to see if our features are working.

8 Evaluation

8.1 Performance

As of now, all subsystems of the rover have been successfully implemented and individually tested for performance. The details of each subsystem's performance have been discussed in more detail in their respective sections. Our focus now is on integrating these subsystems and fine-tuning the parameters to ensure optimal functionality. We are working towards completing the integration and parameter adjustments before the upcoming demo.

8.2 Limitations

The rover's performance was largely constrained by equipment limitations. Firstly, the motor's limited torque imposed restrictions on the design options available. Secondly, the constraints imposed on the FPGA/UART system made it highly unlikely to transmit precise rover feeds to the server, thereby limiting the image processing capabilities. Specifically, we were unable to get real-time updates of the entire image/feed to the server and were also discouraged to do so, referencing thread 21 on EdStem. This prevented the utilisation of advanced libraries like LIDAR SLAM or OpenCV, forcing reliance on lower-level image processing methods. Consequently, the accuracy of image mapping was considerably reduced due to this limitation.

8.3 Feedback

The primary challenge faced during the project was the significant time pressure imposed on the team. The successful performance of the final system relied not only on the individual functionality of each subsystem but also on fine-tuning modules such as the drive system. However, due to the necessity of integrating all components before fine-tuning, it became apparent that more time was needed than initially anticipated. Additionally, refining parameters and making necessary adjustments proved to be time-consuming, despite having prior calculated values from theoretical models / created strategies.

If the project were to be attempted again, the team would prioritise better time management from the outset. This would involve starting the integration process earlier, allowing for more extensive testing and refinement. A greater emphasis would be placed on allocating sufficient time for parameter fine-tuning, as this aspect proved to be crucial for achieving optimal performance. By recognising

the importance of early integration and parameter refinement, the team would strive to strike a balance between individual subsystem development and timely integration, ultimately ensuring smoother project execution in future endeavours.

8.4 Future Work

Moving forward, our future work can be divided into two main sections, with the first section focused on fine-tuning parameters and enhancing the performance of our existing systems.

- more advanced conversion from instructions to motor control (need more testing of corresponding values)
- fine-tuning the weights and how many squares to "look-ahead" for the pathing algorithm (discussed in its own section)
- fine-tuning the discovery radius and offset and discovery progress for optimal maze mapping progress tracking (discussed in its own section)

The second section focuses on additional implementable features

- more advanced sensor fusion i.e. Kalman filter
- utilisation of the database feature into session fetching and replay session based on timestamps
- implementing more advanced image processing algorithms or to reduce other hardware components in the system like the multiple Arduinos, a FPGA with a hard core connected via PCIe, known as a FPGA System-On-Chip (FPGA SoC) (Intel Corporation 2023) could be used, which would provide the advantages of having reconfigurable hardware, and a more capable general purpose processor and a higher bandwidth limit for communications between the general purpose software core and the FPGA's hardware

Bibliography

Albaghdadi, A. & Ali, A. (2019), 'An optimized complementary filter for an inertial measurement unit contain mpu6050 sensor', *Iraqi J. Electr. Electron. Eng* **15**(2), 71–77.

Baker, Bonnie (2018), '*Apply Sensor Fusion to Accelerometers and Gyroscopes*'. https://www.digikey.co.uk/en/articles/apply-sensor-fusion-to-accelerometers-and-gyroscopes, [Accessed 18th June 2023].

Bennet, James. and Bhasin, Ajay. and Grant, Jamila. and Chung Lim, Wen. (2023), 'PID Tuning via Classical Methods'. https://eng.libretexts.org/Bookshelves/Industrial_and_Systems_Engineering/Chemical_Process_Dynamics_and_Controls_(Woolf)/09, [Accessed 20th June 2023].

Gui, P., Tang, L. & Mukhopadhyay, S. (2015), Mems based imu for tilting measurement: Comparison of complementary and kalman filter based data fusion, *in* '2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)', pp. 2004–2009.

Hlaaili, M. & Mechergui, H. (2016), 'Comparison of different mppt algorithms with a proposed one using a power estimator for grid connected pv systems', *International Journal of Photoenergy* p. 10.

(https://electronics.stackexchange.com/users/35530/spehro pefhany), S. P. (n.d.), 'What is the relation between filter coefficient (n) of simulink pid block and filter time constant of matlab pid command?', Electrical Engineering Stack Exchange. URL:https://electronics.stackexchange.com/q/637203 (version: 2022-10-03).

Hunter Adams, V (2014), 'Complementary filters'. https://vanhunteradams.com/Pico/ReactionWheel/Complementary_Filters.html#Complementary-filters, [Accessed 18th June 2023].

Intel Corporation (2023), '*Intel® FPGAs and SoC FPGAs*'. https://www.intel.co.uk/content/www/uk/en/products/details/fpga.html, [Accessed 20th June 2023].

Kim, S. & Kwon, S. (2015), 'Dynamic modeling of a two-wheeled inverted pendulum balancing mobile robot', *International Journal of Control, Automation and Systems* **13**(4), 926–933.

Kocur, M., Kozak, S. & Dvorscak, B. (2014), Design and implementation of fpga - digital based pid controller, *in* 'Proceedings of the 2014 15th International Carpathian Control Conference (ICCC)', pp. 233–236.

Lundberg, Kent (2002), '*The Inverted Pendulum System*'. https://web.mit.edu/klund/www/papers/UNP_pendulum.pdf, [Accessed 15th June 2023].

Lundgren, Ludvig. and Ahnlund, David. (2022), '*Control of self-balancing robot*'. http://kth.diva-portal.org/smash/get/diva2:1737768/FULLTEXT01.pdf, [Accessed 15th June 2023].

Mahdi, A. S., Mahamad, A. K., Saon, S., Tuwoso, T., Elmunsyah, H. & Mudjanarko, S. W. (2019), 'Maximum power point tracking using perturb and observe, fuzzy logic and anfis', *SN Applied Sciences* **2**, 89.

Mohd Ali, N., Md Rashid, N. & Mustafah, Y. (2013), 'Performance comparison between rgb and hsv color segmentations for road signs detection', *SN Applied Sciences* **393**, 550–555.

mouad boumediene (2023), '*two wheeled self balancing robot*'. https://uk.mathworks.com/matlabcentral/fileexchange/88768-two-wheeled-self-balancing-robot?s_tid=srchtitle, [Accessed 10th June 2023].

Nature's Generator (2023), 'SOLAR PANEL SERIES VS PARALLEL'. https://naturesgenerator.com/blogs/news/solar-panel-series-vs-parallel, [Accessed 20th June 2023].

O'Kane, Mary (2023), '*Fill Factor of Solar Cells*'. https://www.ossila.com/pages/fill-factor-solar-cells, [Accessed 17th June 2023].

Samak, C. & Samak, T. (2018), 'Design of a two-wheel self-balancing robot with the implementation of a novel state feedback for pid controller using on-board state estimation algorithm'.

Selvan, S., Nair, P. & Umayal. (2016), 'A review on photo voltaic mppt algorithms', *International Journal of Electrical and Computer Engineering (IJECE)* **6**(2), 567–582.

The MathWorks, Inc. (2023), 'Label and Measure Connected Components in a Binary Image'. https://uk.mathworks.com/help/images/label-and-measure-objects-in-a-binary-image.html, [Accessed 20th June 2023].

9 Appendix

9.1 Appendix A: Code

```
power = iL * vb;
// A SLIGHT DELAY IS ADDED
if (com_count % 20 == 0) {
  if (power > power_prev) {
  // here we need to increase it more towards limit
      if (power > 0 || power_prev > 0) {
        dutycycle = dutycycle + 0.01;
      }
  } else if (power < power_prev) {
    // here we need to move it backwards
    if (power > 0 || power_prev > 0) {
      dutycycle = dutycycle - 0.01;
      }
  } else {
    dutycycle = dutycycle;
  }
  // ensure that the duty cycle doesnt pass the limits
  if (dutycycle > 0.99) {
    dutycycle = 0.99; // upper limit of the duty cycle
  } else if (dutycycle < 0.33) {
    dutycycle = 0.33; // lower limit of the duty cycle maybe we can change
        this value
  }
  dutyref = dutycycle;
  power_prev = power;
}
```

Listing 1: MPPT Algorithm

```
from machine import Pin, ADC, PWM

vret_pin = ADC(Pin(26))
vout_pin = ADC(Pin(28))
vin_pin = ADC(Pin(27))
pwm = PWM(Pin(0))
pwm.freq(100000)
pwm_en = Pin(1, Pin.OUT)

voltage_ratio = 249/1249  # 0.19935949

count = 0
pwm_out = 0
pwm_ref = 0

led = Pin("LED", Pin.OUT)
led.on()

CURRENT_LIMIT = 0.29

# CURRENT PID CONTROL VARS
KPI = 0.0
KDI = 0.0
KII = 0.0

TS  = 0.001
# CURRENT VARS

CURRENT_ERROR = 0.0
CURRENT_DERIVATIVE_ERROR = 0.0
CURRENT_INTEGRAL_ERROR = 0.0
CURRENT_LAST_ERROR = 0.0
CURRENT_PID_OUTPUT = 0.0
PREVIOUS_CURRENT_PID_OUTPUT = 0.0

LAST_OUTPUT_CURRENT = 0.0


TARGET_CURRENT = 0.03

def saturate(duty):
    if duty > 62500:
        duty = 62500
    if duty < 100:
        duty = 100
    return duty


interrupt_flag = False

def interrupt_callback(timer):
    global interrupt_flag
    interrupt_flag = True

#timer = machine.Timer(0)

#timer.init(period=800, mode=Timer.PERIODIC, callback=interrupt_callback)


```

```
while True:

    KPI = 45.0
    KDI = 0.0
    KII = 0.00054

    pwm_en.value(1)
    #if interrupt_flag:
    interrupt_flag = False

    vin   = 5*(vin_pin.read_u16()  * (3.3/65535))   # input voltage
    vout  = 5*(vout_pin.read_u16() * (3.3/65535))   # output voltage
    vret  = 0.7*(vret_pin.read_u16() * (3.3/65535))   # current reference
       pin (1.02ohm)
    current  = vret / 1.02

    # PI(D) CONTROL
    # target current = 0.15A
    CURRENT_ERROR = TARGET_CURRENT - current
    CURRENT_INTEGRAL_ERROR += CURRENT_ERROR * TS
    CURRENT_DERIVATIVE_ERROR = (CURRENT_ERROR - CURRENT_LAST_ERROR) / TS
    CURRENT_LAST_ERROR = CURRENT_ERROR # set the last error to the current
       error (as it will now be the last)

    # antiwindup
    if (LAST_OUTPUT_CURRENT >= CURRENT_LIMIT):
        KII = 0.0
    elif (LAST_OUTPUT_CURRENT <= 0):
        KII = 0.0

    # this output is used to control the duty cycle (adds to the last
       control signal)
    CURRENT_PID_OUTPUT += (KPI*CURRENT_ERROR) + (KDI*
       CURRENT_DERIVATIVE_ERROR) + (KII*CURRENT_INTEGRAL_ERROR)
    LAST_OUTPUT_CURRENT = current

    pwm_ref = int((CURRENT_PID_OUTPUT))
    pwm_out = saturate(pwm_ref)
    pwm.duty_u16(pwm_out)
```

Listing 2: LED driver PID controller

```
    unsigned long current_time = millis();
    dt = (current_time-previous_time) / 1000.0;
    previous_time = current_time;

    // /* Get new sensor events with the readings */
    sensors_event_t a, g, temp;
    mpu.getEvent(&a, &g, &temp);

    float accX = a.acceleration.x / 16384;
    float accY = a.acceleration.y / 16384;
    float accZ = a.acceleration.z / 16384;

    x_angle_acc = (atan2(accY, sqrt(pow(accX,2) + pow(accZ,2))) * 180 / PI)
       - base_x_accel;   // roll
    y_angle_acc = (atan2(-1*accX, sqrt(pow(accY,2) + pow(accZ,2))) * 180 /
       PI) - base_y_accel; // pitch

    float gyro_angle_x = (g.gyro.x - base_x_gyro) / 131;
    float gyro_angle_y = (g.gyro.y - base_y_gyro) / 131;
    float gyro_angle_z = (g.gyro.z - base_z_gyro) / 131;

    gyro_angle_x = gyro_angle_x*dt + last_x_angle;
    gyro_angle_y = gyro_angle_y*dt + last_y_angle;
    gyro_angle_z = gyro_angle_z*dt + last_z_angle;

    float alpha = 0.97;
    float angle_x = alpha*gyro_angle_x + (1.0 - alpha)*x_angle_acc;
    float angle_y = alpha*gyro_angle_y + (1.0 - alpha)*y_angle_acc;
    float angle_z = gyro_angle_z;  //Accelerometer doesn't give z-angle

    last_x_angle = angle_x;
    last_y_angle = angle_y;
    last_z_angle = angle_z;
```

Listing 3: Complimentary Filter in Main Loop

```
  // Filter
reg prev_b, prev_b1, prev_b2;
reg prev_r, prev_r1, prev_r2;
reg prev_y, prev_y1, prev_y2;
reg prev_w, prev_w1, prev_w2;

initial begin
        prev_b<=0;
        prev_b1<=0;
        prev_b2<=0;
        prev_r<=0;
        prev_r1<=0;
        prev_r2<=0;
        prev_y<=0;
        prev_y1<=0;
        prev_y2<=0;
        prev_w<=0;
        prev_w1<=0;
        prev_w2<=0;
end

always@(negedge clk) begin
        prev_b2 = prev_b1;
        prev_b1 = prev_b;
        prev_b = blue_detect;

        prev_r2 = prev_r1;
        prev_r1 = prev_r;
        prev_r = red_detect;

        prev_y2 = prev_y1;
        prev_y1 = prev_y;
        prev_y = yellow_detect;

        prev_w2 = prev_w1;
        prev_w1 = prev_w;
        prev_w = wall_detect;
end
```

Listing 4: Colour Filtering

```
  // HSV Conversion
wire [7:0] hue, saturation, value, cmax, cmin;

assign cmax = (blue > green) ? ((blue > red) ? blue[7:0] : red[7:0]) : (
    green > red) ? green [7:0] : red[7:0];
assign cmin = (blue < green) ? ((blue < red) ? blue[7:0] : red[7:0]) : (
    green < red) ? green [7:0] : red[7:0];
assign hue = (cmax == cmin) ? 0
: (cmax == red) ? ( (green>blue) ? ((((15*((green - blue) / ((cmax - cmin)
    >>2)))>>1)+180)%180) : ((180-((15*((blue - green) / ((cmax - cmin)>>2)))
    >>1))%180) )
: (cmax == green) ? ( (blue>red) ? ((((15*((blue - red) / ((cmax - cmin)>>2)
    ))>>1)+60)%180) : ((60-((15*((red - blue) / ((cmax - cmin)>>2)))>>1))
    %180) )
:               ( (red>green) ? ((((15*((red - green) / ((cmax - cmin)>>2)))
    >>1)+120)%180) : ((120-((15*((green - red) / ((cmax - cmin)>>2)))>>1))
    %180) ); // 0 to 180
assign saturation = (cmax == 0) ? 0 : ((cmax - cmin)* 100 / cmax); // 0 to
    100%
assign value = ((cmax)/255)*100; // 0 to 100%

// Detect areas of interest
wire red_detect, blue_detect, yellow_detect, wall_detect;
assign red_detect = (hue >= 0 && hue  <= 10) && (saturation > 50 &&
    saturation <= 100 && value >= 80 );
assign blue_detect = (hue >= 80 && hue <= 125) && (saturation > 60 &&
    saturation <= 100 && value >= 80 );
assign yellow_detect = (hue >= 25 && hue <= 50) && (saturation > 60 &&
    saturation <= 100 && value >= 80);
assign wall_detect = (hue < 1) && (saturation < 1) && (value > 99);
```

Listing 5: RGB to HSV conversion

9.2 Appendix B: Budget

Description	Unit price	Quantity	Amount
12mm to 4mm bearing for wheels.	£2.97	3	£8.91
Wheels 150cm	£5.10	2	£10.20
ALLEN CAPHEAD SCREWS M6 x 40MM	£0.18	8	£1.44
M2.5 x 20MM screws	£0.03	4	£0.12
		Total	£20.67

9.3 Appendix C: Endpoints

- POST /api/mockupdate - fills the maze with random values from 0 to 100 to mock a random update to the maze by posting to the /api/updatemaze endpoint
- POST /api/reset - resets the maze layout and the discovery mapping to undiscovered status (also accessible through the reset button on the frontend)
- POST /api/update - it takes in raw json data payload from the rover. It is the main method to update the progress of maze mapping and discovery progress.
- POST /api/updatemaze - this is the final endpoint that updates the maze and discovery variable in the backend. It also stores the version of the mappings with its timestamp in the database.
- POST /api/updatemaze - this endpoint uses the maze traversal algorithm to help the motor get its instructions (discussed more in its own separate section)
- POST /api/datahub - it takes in raw sensory and environmental data from the rover (e.g. orientation of the rover, yaw, roll and pitch of the rover, power of the LEDs etc.)
- GET /api/displaymaze - this endpoint is exposed so that the frontend can fetch the relevant mappings from the backend.
- GET /api/displaydata - similar to above. This endpoint is for frontend to display raw data

9.4 Appendix D: Standardised Data Format

Throughout the project, we are looking for ways to standardise our data formats that will be transmitted between subsystems to make collaboration easy. The json payloads listed below are examples of data formats that we are currently using and are subject to change.

```
// data format for maze mapping update
{
    "xpos": xpos,
    "ypos": ypos,
    "orientation": degrees,
    "lines": [0, 0, 0, 1, 1, 0]
}
```

Above is the payload format that we post to the /api/update endpoint. The xpos and ypos are the respective x and y positions of the center of the rover in the maze. The orientation of the rover is

self-explanatory. The "lines" key contains an array of 6 boolean values. The values correspond to whether or not there are illuminated lines in the left bottom, left top, right bottom, right top, front left and front right regions of the rover respectively. They correspond to the detection status in the 6 regions in the FPGA.

```
// data format for maze mapping update
{
    "coordinate": [xcoord, ycoord],
    "angles": [X, Y, Z],
    "orientation": degree,
    "power": [red, blue, yellow]
}
```

This is a sample data payload posting to the /api/datahub endpoint which includes raw readings from the rover. Note that since the raw data comes from different sources, each source only includes the key that is relevant. The backend then processes the payload to upload the frontend with the respective field that has been updated.