Imperial College London

Second Year Group Project

Imperial College London

Department of Electrical and Electronic Engineering

Engineering Design Project
BalanceBug by MazeMaster

Authors:
Johan Jino (02014376)
Sohailul Alvi (02145509)
Ana Dimoska (02061746)
Arihant Singh (02047964)
Gavin Vasandani (01860884)
Shanelle Kulkarni (02027282)

Summer 2023

Abstract

The 2023 Engineering Design Project for Year 2 is centered on designing and fabricating an autonomous, two-wheeled robot capable of navigating through a maze while mapping its layout. The robot utilizes an FPGA-based camera system along with an accelerometer, gyroscope, compass and wheel encoders to autonomously traverse the maze, ensuring not to cross any illuminated lines, that form the maze walls. Additionally, the robot interacts with three illuminated beacons positioned around the maze, which it uses as visual data points to assist in mapping and navigation.

The processing of the maze layout data and navigation instructions are handled by an onboard ESP32 micro-controller in communication with a cloud server. We use optical detection techniques to locate the robot within the maze, ensuring an efficient mapping and navigation. The robot's codebase was optimized for usability, testability, maintainability, and scalability. A critical element of the design was the power management system, which needed to be efficient in handling varying energy levels from the PV-array emulator. A supercapacitor was used for energy storage and an energy system has been devised to ensure uninterrupted operation of the beacons.

The robot successfully navigates the maze while creating a comprehensive map of its layout, and finally highlights the shortest possible path from its start point to the finish point. The system demonstrates quick mapping capabilities and effective navigation while maintaining balance on two wheels. This successful execution shows the competence of our design decisions and the integration of various subsystems, including robot mechanics and dynamic control, robot vision, server communication, beacon management, and navigation. This project has melded our practical engineering skills with teamwork and complex system management, along with taking us through the entire product development journey.

Contents

Introduction and Background

This report will discuss the group's progress throughout the planning, design, and development of the BalanceBug, to meet the requirements of the project. This project follows the outline and criterion provided by the Department of Electrical and Electronic Engineering, as of May 2023.

The brief of this project can be summarised as: *"Build an autonomous bot, balanced on two wheels, that will traverse through an artificial maze formed using white LED strips. The bot shall use an FPGA-based camera module, to detect the LED strips and the three beacons placed on the maze to aid its autonomous navigation process. A live map of the maze has to be displayed on a web application as the bot moves. The bot has to identify any junctions along its way, and at the end find the shortest path from its starting point to the finishing point, which lies on the opposite corner of the maze."*

Hence, to successfully accomplish this project, it can be broken down into six objectives:

1. Design and implement the control system to stabilize the bot on two wheels
2. Prepare the FPGA-Camera hardware and software to perform the necessary detections
3. Set up the server to receive, process, store data, and send the next turn algorithm decisions
4. Host a web application that relays live mapping and highlights the shortest path discovered
5. Build the power grid and management system to light up the three beacons using PV power
6. Integrate the system and establish control communication between the bot, server, and web-application

Along with these six objectives, there were other minor specifications to meet as well. The bot had to be lightweight for easy space transport, cost-effective, and reliable. On top of that, the bot must complete the entire mapping with little to no human interaction.

As a team, we engaged in collaborative brainstorming sessions to explore various methods for achieving our objectives. Once roles were assigned, we proceeded to carefully design and build the different modules, as outlined in the 'Planning and Design' section 2 below.

Planning and Design

2.1 Schedule and Task Distribution

In order to ensure an organized and efficient project tenure, we created a projected timeline for the various components within the project, as depicted in the Gantt chart in Figure 2.1. The overlaps in the bars allow for schedule flexibility. Our team met regularly to discuss the robot's development and the project's sub-components.

A significant portion of time was dedicated to the FPGA vision and component testing phase. The FPGA vision phase was vital for understanding the data fed into the ESP32 for processing. The component testing phase was multifaceted and crucial, involving the testing of various aspects of the project and integrating them for group testing. Attention was also given to the physical design of the robot, particularly for the balancing.

We choose aspects of the project to work on initially. To ensure communication and coordination, a daily log of tasks was maintained on an online project-tracker platform called Monday.com (1), keeping the entire team informed. We swiftly transitioned to new tasks upon completion of previous ones. This approach fostered a dynamic and collaborative task distribution and tracking within the team. Through this structured and collaborative approach, we were able to efficiently allocate resources and manage timelines. Additionally, our cohesive effort and communication greatly contributed to solving complex challenges and ensuring the successful execution of the project.

Figure 2.1: Gantt Chart

Since 22nd May, we consistently organized regular meetings. These meetings involved a range of agendas including engaging in discussions, exchanging ideas, and collaboratively working on the distinct components of the project. The meeting log shown in Figure 2.2 below provides an overview of our collective activities throughout the duration of the project:

Date	Venue	Agenda
22/05	Queen's Lawn	Discuss project requirements and approach
23/05	Study Room	Plan for balance control system and web app
24/05	EE Lab	Start building the web app
27/05	Robotics Lab	Work on the balance control system
29/05	EE Lab	Work on the FPGA and camera vision
31/05	EE Lab	Devise the node identification algorithm
01/06	EE Lab	Plan mapping mechanism and autonomous system
06/06	Robotics Lab	Design and implement the new chassis
08/06	Central Library	Discuss and start writing report
10/06	Robotics Lab	Test and implement the control system
13/06	EE Lab	Add compass for navigation aid
15/06	MS Teams	Test the completed power grid and beacon system
17/06	EE Lab	Clean up circuits and robot fittings
18/06	EE Lab	Debugging software and test running the robot
19/06	Central Library	Test server integration and process execution
20/06	MS Teams	Final rundown and checks on the report

2.2 Design and Development

After we broke down the project into various sub-components as outlined in section 1, it was necessary to identify a design criteria that the overall rover must satisfy. These design requirements had to be clearly defined as well as measurable and testable. These requirements have be clearly listed in table 2.1.

The first step for the team was to discuss the criteria for the rover outlined in the table below and then formulate a strategy to begin the work. The control system for the rover was an initial priority as the rover was required to move while balancing in order to carry out the other functions. So, it was decided to first work on MATLAB in order to initially design the control system and simulate this control system along with the rover. After this was done, the control system was tested with the actual rover and it was realised that it did not work despite the results from SIMULINK. Therefore, several iterations of the control system were considered for the balancing of the rover.

Similarly, the design of the chassis also went through several iterations before reaching an optimal design. We started with the chassis we were provided but ended up designing the chassis in a modular way to enable easy testing and improvement of the design.

The FPGA had to detect different colours and lanes. Different methods were used to implement this in the FPGA hardware. MATLAB was used to determine the thresholds of RGB values to detect each colour independently. On top of this area calculation is done for identifying each node.

The work on the power system first involved the characterisation of the PV panel so that a PSU could be utilised for emulation and testing. Work was done in subsequent stages as the different types of SMPS were recieved in subsequent stages. The boost SMPS connected to the PV panel was considered first and once the work on this was finished, work was done on the buck SMPS for the LED driver.

Criterion	Description	Method of Testing
1. Rover Balancing and Stable Movement	A control system should stabilize the rover on two wheels. The rover should be able to balance in place as well as steer and move while remaining balanced and stable.	Observe whether the rover can maneuver through junctions in a balanced and stable manner.
	The rover should be able to maintain balance when subjected to external forces.	Observe whether the rover can traverse the entire maze without falling and within a 10-minute time limit.
	The control system must be highly responsive, exhibit minimal oscillations, and highly dampened overshoots to keep the FPGA camera module stable.	Observe whether the FPGA camera module is stable enough to track the LED lights and beacons.
2. Autonomous Navigation	The FPGA-Camera module should detect the white LED strips and the three beacons placed on the maze and use them to autonomously navigate.	Pre-set the maze with white LED strips and three beacons, then compare the actual positions with the positions detected by the FPGA camera module.
	The rover should be capable of identifying junctions to map the maze and not retrace already visited paths. This is necessary to eventually determine the shortest path through the maze.	Verify that the rover successfully identifies junctions and correctly maps the maze by comparing the generated map with the actual maze.
3. Real-Time Mapping	A web application must provide a live map of the maze that is updating in real-time as the rover traverses through it.	Use traceroute to measure the latency of the live mapping in the web application to ensure the latency is within 50ms.
4. Reliable Data Transmission	The server should receive, process, and store data from the rover, which is incoming from multiple different sources such as sensors and other microcontrollers.	Verify that the server is correctly receiving and storing the data from the rover's sensors and microcontrollers by comparing the transmitted data and stored data.
	Using this data, the server must send the next movement decision to the rover and receive status updates.	Measure the success rate of data transmission to the server and ensure it is around 95%. Repeat transmission of critical messages such as next movement decisions if necessary.
5. Power Management	A power grid and management system must be built to efficiently and consistently illuminate the three LED beacons using PV (Photovoltaic) power for the duration of the maze traversal.	Monitor that the beacons stay sufficiently bright when the input current limit is varied.
6. System Integration and Control	All components of the system, including the rover, server, and web app, should be tested individually to ensure they're working as expected and then integrated with each other.	Conduct individual component testing by verifying the functionality and performance of each component through unit tests and simulation testing.
	Communication between the rover, server, and web app should be established to enable reliable operation of the entire system.	Conduct integration testing to ensure communication methods used between sensors, microcontrollers, and the server work as expected and data is synchronized.

Table 2.1: Design Criteria for the Overall Rover

Balance Control System

3.1 Overview

When developing a control system for a large-scale autonomous rover, it is crucial to establish criteria that define the characteristics of an effective control system and outline the necessary requirements. Additionally, a clear plan should be outlined for testing and validating these requirements as mentioned in table 2.1.

3.2 Criteria for Controller Design

In Place Stability: The control system for a two-wheeled rover should effectively maintain the rover's position, even when subjected to external forces, while minimizing overshoot. However, it is important to ensure that excessive vibrations resulting from the control system do not interfere with the stabilization of the FPGA cameras. The stability requirement can be broken down into 3 control system characteristics:

1. Swift Response Time: The control system should enable the rover to stabilize rapidly when faced with external forces or disturbances, ensuring prompt corrective action.

2. Effective Damping: The controller design should incorporate high damping to reduce oscillations around the desired set point (given tilt angle) during stabilization. This will help attenuate any excessive or prolonged deviations from the intended position.

3. Minimal Steady-State Error: The rover should exhibit negligible steady-state error, particularly with regard to horizontal velocity. This ensures that the rover remains consistently in position without any noticeable deviation over extended periods of operation.

Steering and Movement Stability: As the rover is expected to maneuver through a maze, it must be capable of steering and moving while maintaining stability.

Sensor Data Accuracy: The response of the control system heavily relies on precise data from the MPU6050 sensor, particularly regarding tilt angle and acceleration in the x, y, and z axes. To ensure accurate measurements, it is important to apply offsets, calibrate the MPU6050 during setup, and utilize a complementary filter that combines accelerometer and gyroscope data (2). This approach enhances stability by considering both short-term and long-term factors when determining the current tilt angle of the rover.

Responsiveness of the control system: The controller must be highly responsive to changes in the rover's tilt angle. The rover should be able to instantaneously change the stepper motor behaviour based on the latest MPU6050 readings. However, the motor behaviour should remain smooth without jitter, preventing the rover from tipping over and ensuring overall stability.

3.2.1 Methods to Test Criteria Requirements

Simulation: To determine the appropriate number of PID controllers and their arrangement (Cascaded or Connected in parallel), as well as the required PID values for tuning all the controllers, the control system can be integrated into a simulation environment. This simulation replicates the rover's characteristics such as weight and center of mass, enabling stability testing. This approach

provides valuable insights about the PID controllers before their physical implementation and allows for highly repeatable tests in the early stages of development (3).

Analyzing sensor data: The readings from the MPU6050 can be outputted to a serial monitor to display the exact measurements that will be passed to the rover. This allows us to observe if there's a latency between moving the MPU6050 in real time and the values printed on the monitor, whether data is being outputted at a high enough frequency, if the data is calibrated by applying an offset determined during the rover setup.

Running benchmarks on Controller Code: Considering that the controller will be implemented in C++ code on a microcontroller, it is crucial to conduct benchmarks to determine the time required for a loop iteration or a data call from the MPU6050. This testing is necessary to ensure efficient code implementation. If the code processing time or data retrieval time is too long, it may result in significant delays that are large enough such that the robot could tilt over and become unstable.

Physical Testing: The rover's ability to maintain stability can be evaluated by subjecting it to external forces or disturbances, such as pushing or shaking. Additionally, a series of test cases or scenarios can be designed, including actions like rapid acceleration and sudden stops, to observe and analyze the rover's behavior. By comparing its performance against the established controller design criteria, we can determine if the rover functions as expected and meets the desired stability objectives.

3.3 Controller Arrangement

The choice of controller arrangement, such as cascaded or parallel, is crucial in designing an effective controller for balancing our two-wheeled rover. Initially, our plan involved employing a single PID controller to regulate the rover's tilt angle. The input to the controller is the measured tilt angle of the rover, obtained from a gyroscope. This measured angle is compared to a desired setpoint from which the required motor acceleration is computed to reach this setpoint.

Although this controller design successfully maintained angle stability, our simulations revealed the presence of a steady-state horizontal velocity. This indicates that the rover was able to achieve the desired tilt angle but struggled to maintain balance in a stationary position. Furthermore, according to the design criteria, it is essential for the rover to maintain balance while being capable of stable steering and movement. To initiate movement, we can adjust the desired angle setpoint for the angle PID controller, prompting the rover to move or steer in order to regain angle stability. Consequently, the angle setpoint becomes a dynamic variable that necessitates an additional controller to modify it based on a given input velocity(4). Therefore, a velocity PID controller that feeds into an angle PID controller was the chosen controller arrangement, as shown below.

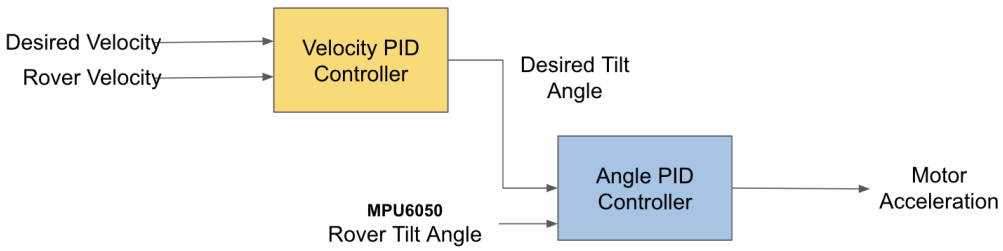


Figure 3.1: Cascaded Controller Arrangement

The following controller operates as follows:

- The user inputs a desired velocity into the velocity PID controller, while simultaneously measuring the rover's current speed. The difference between these two values is then utilized to compute the desired tilt angle required to achieve the desired velocity.

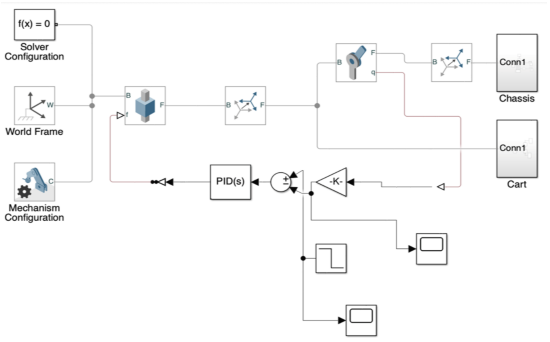
- This desired tilt angle is then fed into the angle PID controller, which takes into account the current tilt angle of the rover. Using these values, the motor acceleration is calculated to attain the desired tilt angle and, consequently, the desired velocity.

- The cascaded PID arrangement provides the user with the flexibility to set the input velocity as 0 to maintain balance in a stationary position or choose a non-zero input velocity to adjust the desired angle and enable stable movement of the rover.

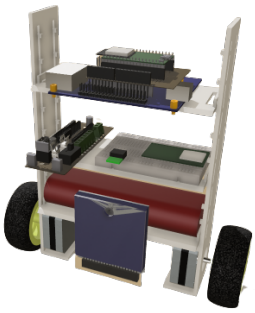
- This controller arrangement can then be simulated to determine whether it satisfies the controller design requirements as well as to tune the velocity and angle controllers' PID values.

3.4 Control System Simulation

The next stage in developing our controller involved recreating the control system in Simulink and introducing a singular PID controller. The control system consists of a 3D model of the rover with precise dimensions, weights, and material choices matching our planned rover.



(a) Simulink Control System Design



(b) 3D Designed Model used for simulations

Figure 3.2: Control System Design and 3D Model

The solver configuration block enables the solution of differential equations that accurately represent the movement dynamics of our rover, effectively replicating its expected real-world behavior (5). A pivotal joint connecting the rover and the world frame facilitates horizontal movement. The rover's tilt angle is measured by a sensor and fed into the angle PID controller, which generates motor acceleration to achieve the desired tilt from the current angle. To evaluate the angle PID controller's performance in stabilizing an unstable control system, a unit step force in the form of a tilt angle is applied. The following video exhibits these simulations we obtained: `https://youtu.be/9gUuJAow9No`

The optimal PID values determined for the angle PID controller were as follows: Kp = 120, Kd = 50, Ki = 0. These values were obtained by generating a desired output response plot in Matlab and utilizing the Simulink solver to automatically tune the PID parameters accordingly. This process served as a valuable starting point for PID tuning. However, we observed a residual horizontal velocity, indicating the presence of a steady-state error. To address this, the Kd parameter was increased and a velocity PID controller was added in a cascaded manner. The desired output response plot used in Matlab is illustrated below.

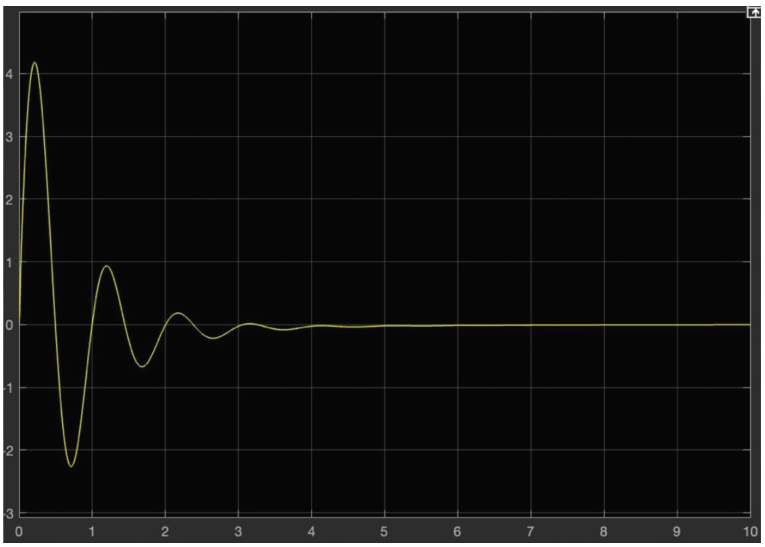


Figure 3.3: Tilt Angle vs. Time

Based on the simulation results, we determined transfer functions for the angle PID and velocity PID controllers that allow for stable movement of the rover as well as the ability to balance in place. We also successfully tuned the PID values for the new, cascaded controller arrangement.

Through observation, we found that the PID values in table 3.1 resulted in desirable outcomes. This configuration provided excellent position stability, fast response times, and effectively dampened oscillations when overshooting occurred.

Table 3.1: Tuned PID Values

Controller	Kp	Kd	Ki
Angle PID	120	30	0.01
Velocity PID	0.004	0	0.0005

3.4.1 Difference in PID values for Angle and Velocity Controllers

It was observed that the PID values for the angle controller were significantly larger than the velocity controller. Upon further research, we determined that as small variations in the rover's tilt angle can significantly impact the rover's balance and stability then higher PID values provides a more responsive controller to maintain balance and stability. However, when it comes to variations in rover velocity, as long as the changes are gradual and smooth, they do not significantly affect the rover's balance. Consequently, lower PID values were selected for the velocity controller, ensuring smoother adjustments in speed without introducing overshooting or instability concerns.

3.5 Measuring Rover Velocity and Tilt Angle

3.5.1 Sensor Requirements

The first step in implementing the simulated controller on the physical rover involves testing the sensors responsible for measuring the rover's current tilt angle and velocity. These sensor readings are inputs for each of the controllers and, therefore, impacts the overall performance and effectiveness of the control system. A suitable sensor satisfies the following core requirements:

1. Accuracy: The sensors should provide reliable and precise measurements of the rover's velocity and tilt angle.
2. Sensitivity: The sensors should be sensitive enough to detect even minor changes in the measured variables.

3. Responsiveness: The sensor should have a high sampling rate and low latency providing real-time and instantaneous readings. This enables the control system to make timely adjustments and respond quickly to changes in velocity and tilt angle.

4. Handling noise and disturbances: The sensor readings shouldn't be susceptible to internal noise inherent to the gyroscope or accelerometer. If such factors are beyond the user's control, then necessary filtering techniques must be used to minimize fluctuations in the sensor's readings.

To determine the rover's velocity an accelerometer is used to measure the rover's current acceleration in 3 domains which is then combined and weighted to determine the rover's acceleration in the horizontal plane. Multiplying the current acceleration with the time difference between now and the previous acceleration measurement, gives the velocity change which is summed with the previous velocity as shown in figure 3.4

```
float dt = float(currentTime - prevTime) * 1e-6;
velocity += accel * dt;
```

Figure 3.4: velocity calculation

To determine the rover's tilt angle a gyroscope was used to detect changes in the angular velocity over 3 different axis as the rover tilts. The angular velocity is then integrated with the time difference between the current and last sensor reading, producing the overall change in angle.

Ultimately, the most suitable sensor that combines the abilities of a 3-axis accelerometer and 3-axis gyroscope is an MPU6050. By using a component with multiple in-built sensors, we were able to fuse readings from the gyroscope and accelerometer when determining the rover's tilt angle. This was achieved using a complementary filter which takes advantage of the gyroscope's responsive, high-frequency data. However, since gyroscope readings are susceptible to drift over time, we utilized the low-frequency accelerometer readings to counteract this drift. The complementary filter effectively mitigated drift by giving priority to the accelerometer data at low frequencies. The complimentary filter transfer function is shown below in figure 3.5.

```
float a = 0.98;  // Complementary filter constant
AngleX = a * gyroAngleX + (1 - a) * accelAngleX;
```

Figure 3.5: angle calculation

3.5.2 Code Development for MPU6050

The Arduino code used to initialize and retrieve data from the MPU6050 sensor was carefully developed to ensure precise and reliable readings. Several measures were taken to enhance the accuracy of the sensor data:

1. High Sampling Rate: The MPU6050 has a sampling rate of 1KHz, as specified in the data sheet. This high sampling rate ensures that the sensor provides instantaneous and real-time data of the rover's current state. The sensor's readings aren't immediately processed by the control algorithm, instead, the readings are stored in a buffer and processed once the buffer is filled. This allows multiple data samples to be collected and averaged over a period of time which is then sent to the control algorithm to be processed. This smoothens out any noise and variations in the data.

2. Calculating Offset: To eliminate any biases or offsets in the sensor readings, a calibration process was implemented. During the first 5 seconds of the program, the sensor samples data and calculates the mean offset. This mean offset is then subtracted from the subsequent tilt angle readings, ensuring accurate measurements.

3. Calibration Functions: The MPU6050 library used in the code includes calibration functions, such as iterative optimization. These functions compare the sensor's readings against known reference values and iteratively adjust the sensor's output to align with the reference values. This iterative optimization further enhances the accuracy and reliability of the sensor readings.

By implementing these measures, the Arduino code maximizes the precision of the MPU6050 sensor.

3.6 Implementing Control Algorithm on Microcontroller

3.6.1 Microcontroller Selection

To implement the control algorithm, we chose a microcontroller with a sufficiently high clock frequency capable of processing the sensor's high sampling rate, light-weight for easier stability and low power consumption so that all components can be powered with the battery. We also prioritized familiarity with the microcontroller and its strong support for interrupt handling.

We opted for an Arduino Nano to implement our control algorithm instead of an ESP32, despite the ESP32's higher clock frequency. This decision was made based on trial and error as well as considering the fact that the ESP32 would be responsible for running the control algorithm as well as for server communication and listening for movement requests. Combining these tasks could strain the processor, potentially causing delays, missed interrupts, and reduced responsiveness in the controller. Such circumstances could compromise the stability of the rover, making it difficult to recover from tipping over. Therefore, a dedicated Arduino Nano to solely implement the control algorithm was the most suitable approach. Moreover, the lower power requirement of the Arduino Nano (6) makes it more suitable for the rover solely powered by the given battery. Additionaly, using Nano we utilize the available processing power to the fullest, hence not waisting resoures as well as making it cost effective.

3.6.2 Using Interrupts in Arduino Nano for Motor Stepping

There are several approaches to implementing motor stepping based on the output of the angle PID controller. Initially, we considered mapping the output of the angle controller, the motor acceleration, to the delay between motor steps. A higher acceleration would correspond to a lower delay resulting in more frequent stepsIn our approach, the motors would run for a fixed number of steps within the program's loop. While this allowed the wheels to rotate in the correct direction to recover the desired tilt angle, it introduced a delay in retrieving the next sensor reading. After each loop iteration, which takes few milliseconds according to our Arduino profiler, the next sensor reading would be obtained. This delay proved to be too long for our requirements.

By the time we retrieved the next sensor reading, the value in the MPU6050 buffer would be outdated and no longer reflective of the current rover state. Consequently, the motors were not responsive enough to make timely adjustments, resulting in the rover tipping over.

In response to these limitations, we opted for an implementation that uses interrupts. Instead of relying on loop iterations to evaluate the rover's tilt angle and execute motor acceleration, we introduced an interrupt mechanism that triggers based on a timer and interrupt value comparison. A timer is initialized and counts based on the clock cycle, when the timer value matches the interrupt value, we pause the current task being executed and we execute a function known as an interrupt service routine (ISR) (7). The timer is then reset to 0 and begins to count up. This implies that a timer is able to run constantly during a program and allows the microcontroller to quickly respond to new sensor readings without being restricted to only checking for them in a continuous loop.

Our code mapped the output of the angle PID controller to frequency of interrupts. A higher motor acceleration meant stepping the motors more frequently. To achieve this, we initialized a timer and modified the interrupt value based on the desired motor acceleration. A higher acceleration would correspond to a lower interrupt value, meaning the timer would more frequently equal the interrupt value and execute an interrupt. Each time an interrupt occurred, the ISR function was called. In our case, the ISR function had a command to step both motors forward. Therefore, by controlling the interrupt value, we could manipulate the frequency and thus speed of motor steps.

This approach led to smooth motor movements, high responsiveness and ultimately led to a balanced in place rover.

A complete breakdown of the code used for interrupts and mapping the output of the PID controller is shown in section 11.3.

3.6.3 PID Controller Implementation in Arduino Nano

The 2 controllers: Velocity PID and Angle PID were implemented in code by defining a PID class with a constructor that takes in PID gain values, desired setpoint and time step. This class helps keep the main program organized and easy to understand, simplifying future debugging. To create the controllers, an instance of the PID class is initialized for both the angle and velocity controllers. Since both controllers have identical transfer functions, this approach avoids code duplication. The PID class also has methods to modify the desired setpoint and calculate the controller output.

For the velocity PID object, the input parameters are the rover's velocity, target velocity and PID gains. After calling the method velocityPID.getControl(); the desired angle setpoint is outputted. This serves as an input parameter for the angle PID controller, alongside the currently measured tilt angle and velocity PID gains. The output of this is the motor acceleration which is fed into a function that maps the acceleration to an interrupt value, modifying the steps per second and consequently the rover speed.

The time step was determined by using the millis() function to measure the time at the current iteration and subtract it from the previous millis() value assigned in the previous iteration. This time step is crucial for determining the rover velocity using its acceleration value, and is used in the integral error in the PID transfer function.

1. Swift Response Time: The control system should enable the rover to stabilize rapidly when faced with external forces or disturbances, ensuring prompt corrective action.

2. Effective Damping: The controller design should incorporate high damping to reduce oscillations around the desired set point (given tilt angle) during stabilization. This will help attenuate any excessive or prolonged deviations from the intended position.

3. Minimal Steady-State Error: The rover should exhibit negligible steady-state error, particularly with regard to horizontal velocity. This ensures that the rover remains consistently in position without any noticeable deviation over extended periods of operation.

3.6.4 Stable Movement and Steering

According to our design criteria, the rover must be capable of balancing in place as well as moving forward and steering stably.

1. Balancing: The variable targetVelocity is set to 0 if we want the rover to stay in place, however, when the targetVelocity is set to a non-zero value, it modifies the desired tilt angle. This modifies the motor acceleration to reach this desired targetVelocity. The targetVelocity is multiplied by a factor of 0.99 causing the speed to gradually decrease to 0 and return to a stable state. The following excerpt in figure 3.6 shows how the velocity PID output feeds to the angle controller.

```
float targetAngle = -velocityPID.getControl(velocity, dt);
anglePID.setTarget(targetAngle);
float motorAccel = anglePID.getControl(tiltAngle, dt);
```

Figure 3.6: Cascaded Controller Implementation

2. Steering: A steering variable is added to the left motor velocity and subtracted from the right motor velocity. As a result, one motor has a higher step rate than the other, thus having more speed and causing turning. The steering variable then decreases overtime bringing the steering motion to a halt. (8)

3.7 Stepper Motors And Microstepping

The movement of the stepper motors is controlled through the A4988 Driver and the microcontroller controls the STEP and DIR pins for each driver. The signal at the DIR pin sets the direction for the stepper motor (9) and the motor moves a single step on the rising edge of the signal sent to the STEP pin. For the given stepper motors, each step is equivalent to rotation by 1.8°(10). The precision of the movement of the stepper motors can be improved by implementing microstepping, a function which is supported by the A4988 Driver. For the implementation of the design of the rover and the control system that we have, we configured the A4988 Driver to operate in half-step mode. This allows us to have a more precise control of the position and movement of the stepper motors relative to the full-step mode. The microstepping may be increased to quarter-step mode but we observed that the rotational speed of the motors decreased with an increase in the microstepping. Since the response of the motors and their movement are required to be relatively quick, half-step mode seemed an optimal configuration. Based on the datasheet for A4988, the pins MS1, MS2, and MS3 of the driver should be connected in the configuration outlined in the table below for operation in half-step mode.

MS1	MS2	MS3
HIGH	LOW	LOW

The MS1 pin on the driver already has a pull-down resistor as mentioned in the datasheet so external resistors were not connected and the MS1 pin for both the drivers was directly connected to 3.3V and the other pins to ground. The final schematic for the balance controller is provided in Figure 3.7. All the schematics in this report are made using the EasyEDA software (11).

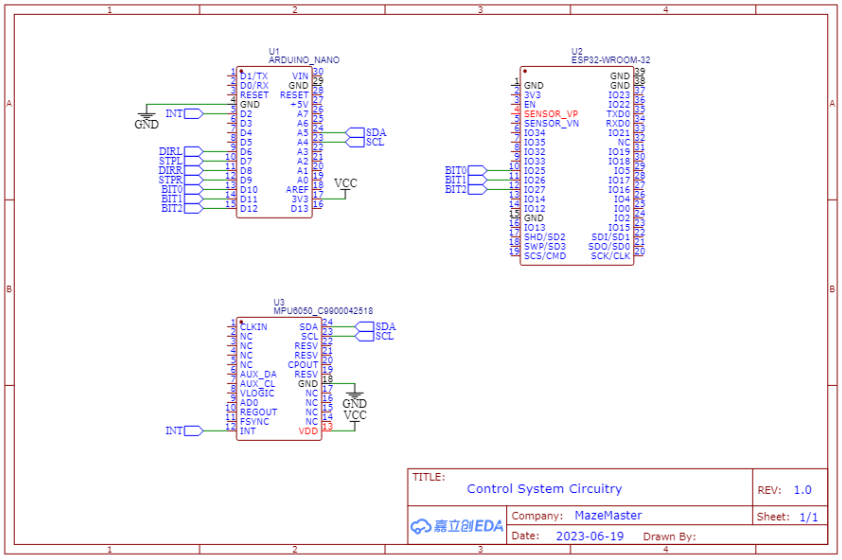


Figure 3.7: Schematic For The Balance System

3.8 Testing Control Algorithm Implementation

After conducting component testing on the MPU6050, stepper motors and testing the control algorithms through simulations, we must now verify the interaction and proper functioning of the different components when integrated together.

3.8.1 Building a Button Controller

For the testing of the steering and movement of the rover along with the control system, external signals were sent to the microcontroller to simulate signals from the FPGA system. This was implemented by having two pushbuttons corresponding to steering and straight movement and sending a HIGH signal to the microcontroller when one of the pushbuttons was pressed. The schematic for

this circuit is provided in Figure 3.8.

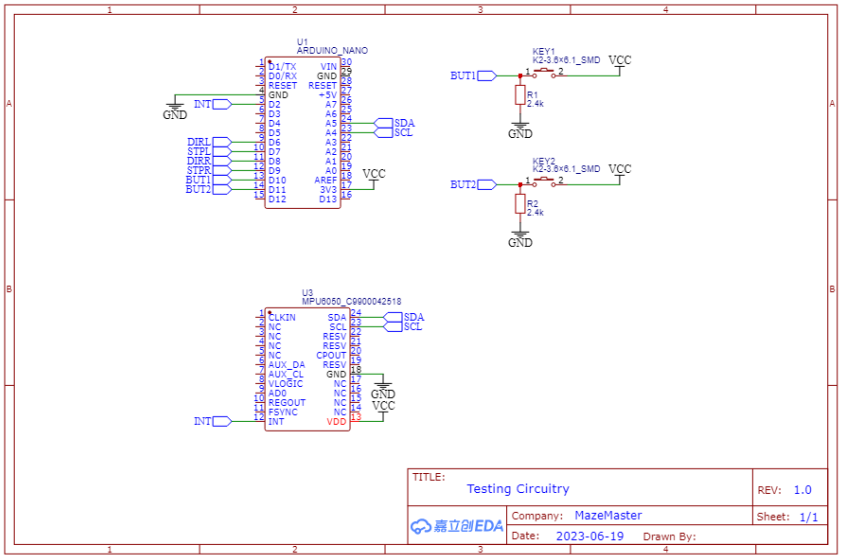


Figure 3.8: Schematic For The Testing System

A pull-down resistor is added to each button system so that the signal is LOW when the pushbutton is unpressed. By using these signals alongside conditional statements in the microcontroller's code we can communicate to the rover whether to perform steering, to move straight, or to stay idle.

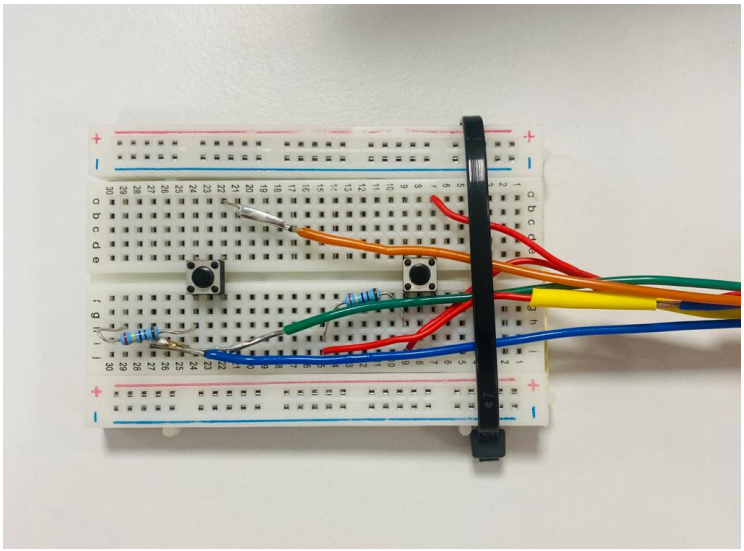


Figure 3.9: Button Controller Used in Testing

3.8.2 Tuning PID Values and Acceleration Constraints

During the testing of the entire control system, we noticed two main issues. First, the rover overshot its target by a considerable amount. Second, it experienced excessive oscillations without proper damping. Although the Simulink simulation helped us determine initial PID values for our velocity and angle controllers, we had to fine-tune these values through trial and error. This was necessary to achieve a more stable and controlled system. The final PID values were:

When mapping the controller output, which determines the motor acceleration, to an interrupt value, it was observed that the stepper motors made a high-frequency whining sound when operated at high motor acceleration values. This was a result of overpowering the stepper motor. To address this, a constrain method was added which limits the motor acceleration to a range. This range

Table 3.2: Final PID Values

Controller	Kp	Kd	Ki
Angle PID	180	20	0
Velocity PID	0.007	0	0.0005

was determined by trial and error, where we balanced the need for sufficient torque to balance and catch the rover from tipping over without causing any significant whining noises. The motor acceleration range chosen was [-200, 200].

3.8.3 Results of Controller Testing

During the testing of the rover, we ensured the test setup was on a flat and stable surface, free of any obstacles. The objective of controller testing is to evaluate the rover's balance capabilities and determine whether the controller design criteria in section 3.2 is satisfied. The breakdown of all the test scenarios and the results are shown below:

Test Scenario	Description	Result
Time for initial Stabilization	Measure the time (ms) taken for the control system to stabilize the rover upon switching on the rover.	$1500\,\text{ms} \pm 75\,\text{ms}$
Response Time Test	Measure the time (ms) taken for the control system to stabilize the rover after applying an external force, pushing the rover with a rod from 5cm back.	$6360\,\text{ms} \pm 200\,\text{ms}$
Steady-state error Test	Observe whether horizontal steady-state velocity is present when the rover is stable and in place over an extended period of time, 60 seconds. Measure the deviation (in degrees) of the rover's tilt angle from the desired set point over a long period of time. Serial monitor outputs the tilt angle.	No steady-state horizontal velocity observed ± 0.2 degrees
Steering and Movement Stability	Conduct turning maneuvers: left turn, right turn, and measure the angle change in the rover's position. Conduct forward movement and measure distance change in the rover's position.	14 degrees angle change per turning maneuver $22.5\,\text{cm} \pm 2\,\text{cm}$

Table 3.3: Test results for rover stability evaluation.

We thus developed a sophisticated control system for the rover's balancing mechanism. In the next sections, we refer to the chassis development which was done simultaneously with controller design since they both were correlated to each other. The balancing robot can be seen here: `https://www.youtube.com/shorts/24yufq_tlpA`

Chassis and Manoeuvrability

The structural design of the chassis of the robot was an integral part in the development of a fully functional robot, as it serves as the foundation upon which all other systems are built. The efficacy of the entire control system and the balancing of the rover are directly dependent on the chassis design. For this robot, the chassis not only needs to support the physical hardware but also needs to ease the integration of various components such as the sensors like the compass and gyroscope, the stepper motors, the FPGA, Camera, ESP32, the Arduino Nano, and the circuitry while maintaining stability and maneuverability. During the development of the rover, the team considered three potential chassis designs, as discussed below.

4.1 Simple Horizontal Configuration

The horizontal configuration as in Figure 4.1 below was our first consideration as it intuitively seemed easier to balance. However, it was observed through testing that this configuration was relatively difficult to balance on the middle point because the motors had to have a high acceleration and needed to react very quickly to changes in the angle relative to the set-point in the gyroscope. Moreover, the room we had for errors were very limited due to the low height design.

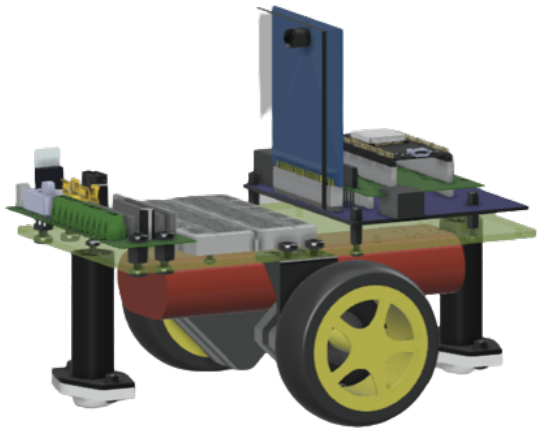


Figure 4.1: Horizontal Configuration (12)

4.2 Simple Vertical Configuration

As a result, the vertical configuration in Figure 4.2 below was considered, and it was reasoned amongst us that it would take longer for the rover to tip over if it was vertical because the height is greater, and the weight is distributed along the vertical axis. There would be sufficient time and acceleration present for the motors to react to variations in the angle relative to the setpoint. Futhermore, the motors are now well offset from the pivot to tilt for stabilisation, which leads to longer radius from axis of rotation allowing greater torque on the rover with smaller force from motors. This is based on the equation as shown in equation: $\tau = rFsin\theta$.

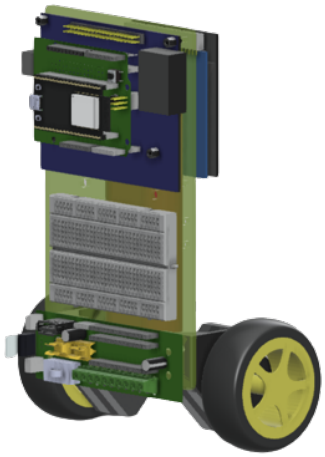


Figure 4.2: Vertical Configuration (13)

The weight distribution of the given chassis was not perfectly symmetrical when the chassis base is kept perfectly vertical, and the natural balancing position of the rover involved a slight tilt in the chassis. Due to the alignment of the motors and their connection to the chassis, it was possible to correct the angle relative to the setpoint when the rover tipped over in the forward direction. However, the asymmetrical weight distribution made it relatively difficult to correct the angle if the rover tipped over a similar amount in the backward direction.

4.3 Final Configuration

Considering the problems encountered with both the above two chassis designs, it was decided that we would go ahead with a redesign of the chassis so that the weight distribution is symmetrical along the vertical axis of the rover. The design process involved designing a modular chassis to enable easy movement of the weight distribution along the vertical axis. Hence, this chassis has been 3D printed in a way so that each of the horizontal stacks/plates of this design can be easily shifted up and down along the height of the rover. We discussed that if the weight distribution was higher at the top end of the rover, then it would be easier to stabilize the rover relative to a design that has more weight distribution at the lower end. However, even though it would be easier to stabilize, the reliability of the stable state is less relative to the other variation. So an optimal weight distribution was required, which is something our new modular design offers. The final chassis design provided in Figure 4.3 below is modular and allows us to test different configurations of the weight distributions to find the optimal configuration to go along with the control system, as each of the stacks can be adjusted.

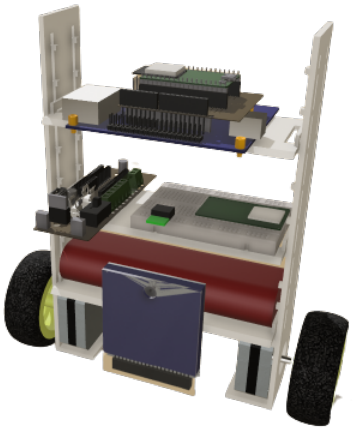


Figure 4.3: Final Configuration

FPGA and Camera Vision

5.1 Overview

The FPGA is mainly used to interface with the camera module and process images frame by frame. Thus, the FPGA acts like an image processing module and the transferring the data to the ESP32. The few key components of the image processing module are colour detection and lane detection. Each of which is implemented within the EEE_IMGPROC module which was the initially layout used to build on top of. The pipeline for image processing is provided in Figure 5.1.

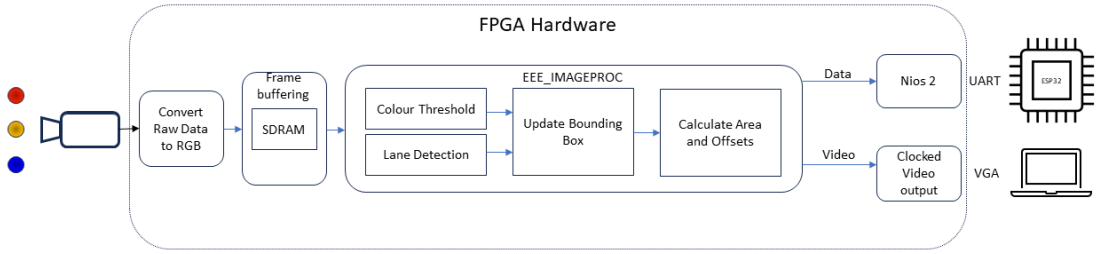


Figure 5.1: FPGA Image Processing Pipeline

5.2 Colour Detection

Colour detection is required to detected different beacons in the maze which is then used for locating itself in the maze, which is the then used to differentiate between nodes in the maze, which is used in the searching algorithm in the server. Using the bit matching strategy as given in the EEEBalanceBug repository (14), we faced issues which colour seperation. Most colours such as red and orange was present in both the beacons red as well as yellow. Hence more intuitive approach was required. We debated on using a RGB to HSV conversion in hardware to then use HSV values to identify pixels, but soon this was neglected given the fact that the precison in the conversions needed to be very high (due to the requirement of dividers in hardware), as well as not all RGB values could be correctly mapped to HSV. Therefore, we used a different approach to implement thresholding. The methodology used involves capturing an image from an FPGA, performing image tuning and thresholding using MATLAB's color thresholding techniques, identifying appropriate RGB ranges, and subsequently implementing the color detection algorithm within the FPGA.

5.2.1 Color Thresholding using MATLAB

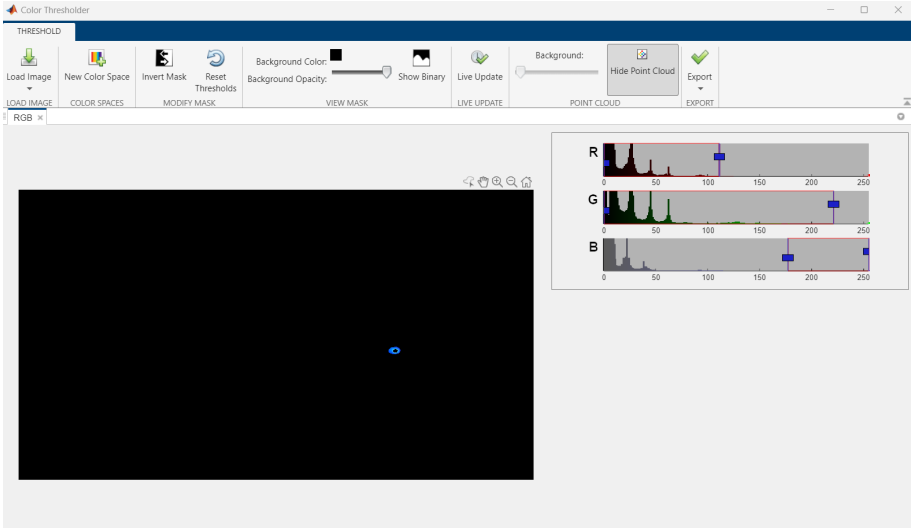
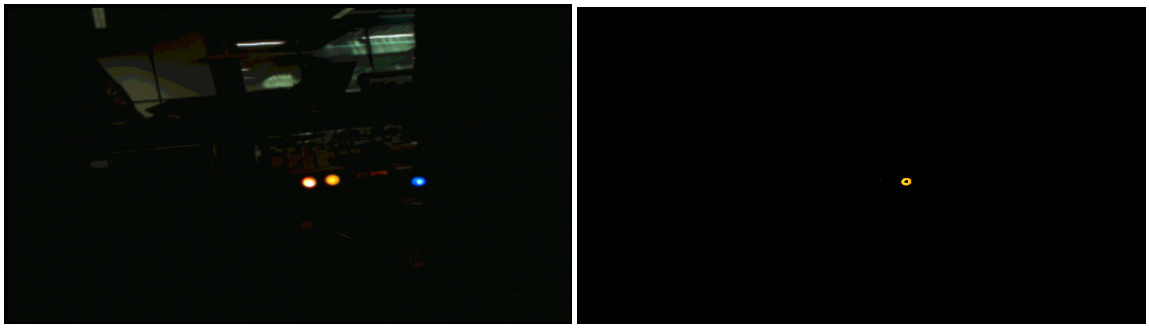


Figure 5.2: Matlab Image Thresholding

An image is captured directly from the FPGA using the provided video capture component. This image is saved in the PC which is then used by MATLAB. MATLAB is employed for color thresholding, the app for which is shown in Figure 5.2.



(a) Before thresholding **(b)** Yellow thresholding

Figure 5.3: Colour Thresholding

Through extensive experimentation and analysis, appropriate RGB ranges are determined for detecting specific colors of interest. These ranges are then used to process images within an FPGA.

5.2.2 FPGA Implementation

The determined RGB ranges from the previous step are translated into ranges for pixel detection within the FPGA. This is used to find minimum and maximum values of pixels at which a colour is located and then used to detect beacons and draw bounding boxes within it. This color detection algorithm is designed to efficiently analyze image data in real-time by utilizing the FPGA's hardware to accelerate image analysis with relatively good performance. Only note-able latency is the 1 frame delay to produce bounding box. This is due to fact that the entire image must be analysed before producing the bounding box which then evaluated and displayed in the next frame. This latency does not affect any other components of the rover, since the rover's movement per frame is negligible. Overall hardware schematic can be seen in Figure 5.1

The converted SystemVerilog module for identifying thresholds has been shown in Figure 5.4

```
    module COLOUR_DETECT (
      input logic [7:0] red,
      input logic [7:0] green,
      input logic [7:0] blue,
      output logic red_detect,
      output logic yellow_detect,
      output logic blue_detect
    );

      always_comb begin
        red_detect      = red>8'd160 & green<8'd70 & blue<8'd70;
        yellow_detect   = red>8'd70 & green>8'd220 & blue<8'd70;
        blue_detect     = red<8'd20 & green>8'd30 & blue>8'd80;
      end

    endmodule
```

Figure 5.4: Colour Detection System Verilog Implementation

```
module AREA (
  input logic [10:0] x_min,
  input logic [10:0] x_max,
  input logic [10:0] y_min,
  input logic [10:0] y_max,
  output logic [31:0] area
);

  always_comb begin
    area = (x_max - x_min + 1) * (y_max - y_min + 1);
  end

endmodule
```

Figure 5.5: Area Calculation System Verilog Implementation

Another feature that had been added to the FPGA was a colour detection limit. We introduced a dynamic colour detection limit. This ensures that coloured pixels are not detected beyond this dynamic limit. This limit is calculated based on the highest y-axis value of white pixel detected. This was implemented to counter act the issue regarding white pixel interference. White pixels also frequently included colours hence the filtering made it harder and more difficult to detect beacons. Since we know the beacons are always above the white led strip, this limit can be used to enhance the detection algorithm as well as give a certain leeway with the ranges of RGB values.

5.3 Lane Detection

Several methods were proposed to implement lane detection. Since this involved shapes rather than colours, ideal implementation would require edge detection. This requires storage pixels stored in a buffer and filtering out data by comparing data with multiple other pixels. This would be computationally expensive to store data and pipeline the process for pixel comparisons. Hence our strategy was to implement a faster yet weaker algorithm that would covey data regarding turns and edges of the lanes.

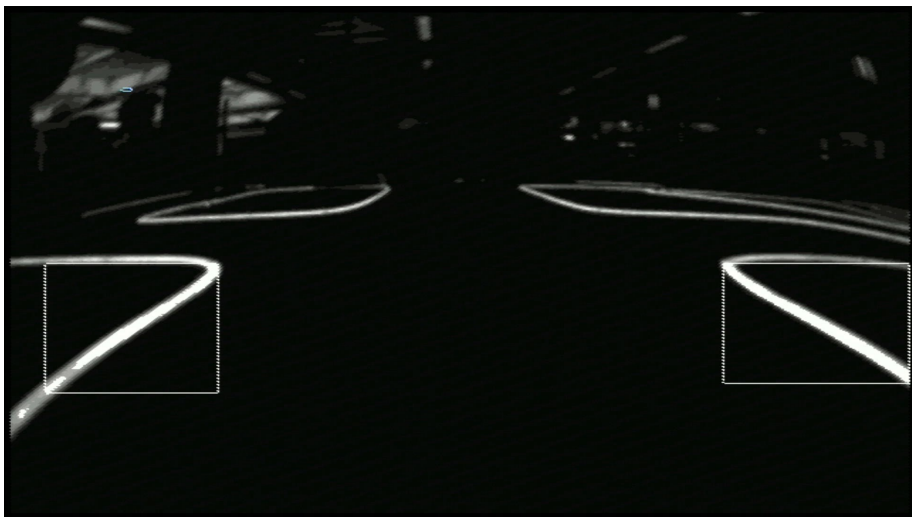


Figure 5.6: Lane Detection

A bounding box can be drawn to the white pixels found below a certain threshold and on both sides of the image. This allows us to determine width and height of each lane, allowing calculation of our position with respect to lanes, end of track as well as missing lanes indicating left and right turns.

- If the right and left boxes cover then entire width of image, we have an end of road ahead,
- If the right box is wider and longer than the left box, it means we are closer to right and hence need to move left. This can be proportionally controlled based on this difference.
- If the height of the right or left boxes are too low or are missing, it would indicate a right or left turn respectively.

The converted SystemVerilog module for lane detection has been shown in Figure 5.7

```
module LANE (
  input logic [10:0] r_x_min,
  input logic [10:0] r_x_max,
  input logic [10:0] r_y_min,
  input logic [10:0] r_y_max,
  input logic [10:0] l_x_min,
  input logic [10:0] l_x_max,
  input logic [10:0] l_y_min,
  input logic [10:0] l_y_max,
  output logic [31:0] offset,
  output logic [31:0] left,
  output logic [31:0] right,
  output logic [31:0] forward
);

  always_comb begin
    offset = (r_x_min - 32'd320) - (32'd320 - l_x_max);
    forward = (r_x_min - l_x_max) < 11'd5;
    left =  (l_y_min > 11'd360);
    right = (r_y_min > 11'd360);
  end

endmodule
```

Figure 5.7: Lane Detection Implementation in System Verilog

All of the System Verilog modules have been initialised and used in the EEE_IMGPROC higher level module schematic which can be found in the GitHub repository mentioned in Section 11.4.

5.4 FPGA ESP32 UART Communication

The FPGA transmits data to the ESP32 micro controller through UART communication which is setup in both the FPGA and ESP32 side. The schematic for the same, given to us from the EEEBalanceBug github repo (14), can be seen in Section 5.8

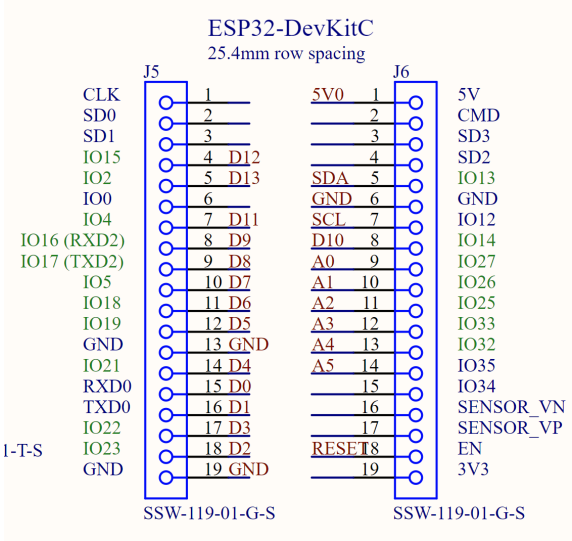


Figure 5.8: ESP32 FPGA Bed

We use pin 16 and 17 which are the UART RX2 and TX2 pins in the ESP32. In the FPGA, these are 8 and 9 pins respectively. Any printed data from the NIOS2 software is passed through these UART pins, which is then used to by the ESP32 for detecting node and storing data. Some issues faced where that the UART pins were blocked or lagging after transmission in very clock cycle because the ESP32 does not read the data every cycle. Further since the data sent is 32 bits, which UART is preset to 8 bit length, it causes data out of sync issues and missing packet issues. To solve these issues, we add use an power intense yet reliable way of reading data from the UART serial. In ESP32, we create a multi-core operation, to keep reading data from the serial with a small delay of 1 millisecond, and store it directly into a memory location defined by the user. This ensures that the main program always has access to the latest data which is stored in the same memory address. Several steps have been taken to ensure errors do not creep in. To ensure race conditions do not occur we use portCRITICAL() functions which ensure other cores wait before a memory is accessed if it is being written at the same time, essentially locking memory locations during use. The implementation of the same is beyond the scope of the report and can be seen in out GitHub repository mentioned in Section 11.4.

5.5 Performance Evaluation

The effectiveness of the FPGA-based color detection system is evaluated through comprehensive testing and benchmarking. Metrics such as detection accuracy, and resource utilization (See 11.2 for FPGA resource utilization) are considered to assess the system's performance and compare it against existing software-based approaches. FPGA's colour detection algorithm runs in at least 60 frames per second. The only bottleneck is in the rate at which camera transfers data to the FPGA along with the one frame delay for calculation. A bigger data bus between the camera and FPGA would significantly increase performance, as well as make it easier to process edges since multiple nearby pixels can be accessed together. This would then be similar to a high level implementation in Python.

Yet, the FPGA allows better frame rates since it is hardware implemented rather than software. Though a implementation similar to our algorithm would perform similarly in Python-OpenCV (considering a average spec PC with no GPU), great performance can be achieved if pipelining and comparing neighbouring pixels is done. This is similar to how a GPU accelerates parallel processing using SIMD (Single Instruction Multiple Data). It must also be noted that our comparison is between a DE-10 lite terrasic FPGA versus a modern powerful PC, which is in no way similar

to price and amount of hardware. Power consumption is also an enormous difference between a GPU/CPU and a FPGA (15) This clearly points out the advantage of creating specialised image processing hardware.

Autonomous Navigation System

6.1 Mapping Algorithm

We had decided to solve the maze with a Depth first search (DFS) approach. The reason for a DFS traversal was the following:

- Reach the end of maze fastest given that start and end are exactly opposite each other.
- This helps find at least a single path from start to end even if time runs out.
- Simple to implement and easier from rover to move through maze with making least number of turns before reaching end.

The rover is partially controlled from the server where the DFS is implemented, hence decisions are taken remotely, on a powerful machine. This is the preferred and ideal way to implement this due the lack of memory and powerful computation for error approximations and searching algorithms. Further, the server has access to a database which can be used to efficiently store and retrieve path information.

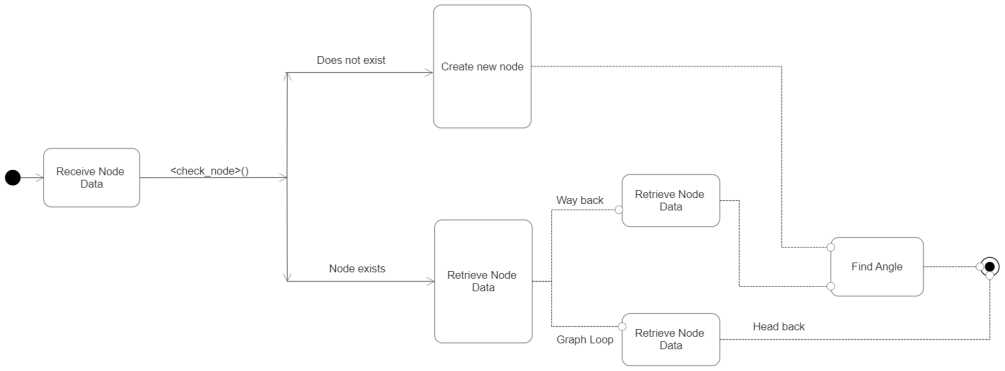


Figure 6.1: Autonomous Server Algorithm

Details of the autonomous control procedure is as follows:

1. The rover connects to the server from the ESP32 micro-controller. This is done using a HTTP request made to the servers endpoint. The data send will be a nested array, which includes data such as Red/Yellow/Blue Beacon Area as well compass readings for each turns. The Server returns a angle value to take. A Sample data is shown in Figure 6.2. An array of $[0, 0, 0, 0]$ means turn n does not exist.

Turn 1	Turn 2	Turn 3	Turn 4
$[100, 25, 90, 320]$	$[10, 115, 0, 150]$	$[0, 50, 0, 10]$	$[0, 0, 0, 0]$

Figure 6.2: Sample Dataset Sent from Rover to Server

2. The rover takes from the starting location considering it as a starting node, transmits the data to the server and the direction to take is received.

3. The rover then turns and moves to that angle and proceeds to move forward until a new node with turns is identified. Then procedure is repeated from the rover side.

4. From the server side, the decision to take an angle depends on if the node is new, or visited or if a loop is encountered. Data from the rover is tried to match any of the existing found nodes in the database, with a certain confidence using a probabilistic matching functions as mentioned in section 6.2. Using a threshold of confidence, it matches a node and decides the movement accordingly based on DFS.

6.2 Weighted Euclidean Matching

To be able to match node data when visited again, we need an algorithm that can be well used to give a relative confidence of the current node being one of the visited. The reason to do this is as follows:

- Errors in colour detection by FPGA either because of fluctuating power supply to beacons or varying external lighting

- Slightly off compass readings due to laboratory environment with slightly changing magnetic field.

Hence, we specifically use a Weighted Euclidean Matching algorithm for this purpose. The euclidean distance between current node and previous visited nodes is taken. The calculated distance is weighted because the amount of error varies for each sensor data (3 from FPGA area calculation, 1 magnetometer reading). Once this distance is calculated, we use a tuned exponential decay to return a confidence value between 1 and 0, which is then converted to a percentage confidence of matching. The formula for Weighted Euclidean Matching and Confidence Exponential Decay is shown in 6.1 and 6.2 respectively.

$$D = \sqrt{\sum_{n=1}^{4} (i_n - j_n)^2 w_n} \tag{6.1}$$

$$C = 100e^{\frac{-x}{\tau}} \tag{6.2}$$

The confidence is the compared with a threshold above which we can confirm the nodes are matched successfully. This methodology allows us to filter out errors as well make our algorithm more reliable.

6.3 Algorithm implementation

The algorithm, as mentioned in section 6.1, is implemented on the Server since it is more powerful as well as it has access to a Database. Python server is wrapped using the Flask framework (16). Everytime the rover encounters a node, i.e. a junction, it makes a POST request to the server which then processes the data using the algorithm and returns the angle for the rover to take. For implementing this with robustness, we created our own class object to identify nodes and keep track of them, which can then later be stored in the database. The blueprint for the same can be seen in 6.3. We have added several attributes and functions for efficiently updating and creating nodes.

```python
# Graph representing the map
class Graph:
    def __init__(self, id, property):
        self.id = id
        self.property = property  # Node property (image data of node)
        self.edge_list = []  # Edges list (angles of each edge)
        self.edge_map = {}  # Edge mappings (maps where each angle leads to)

    # Function to add an edge to the graph
    def add_edges(self, edge_list):
        self.edge_list = edge_list

    # Function to map each edge to node
    def update_map(self, angle, id):
        self.edge_map[angle] = id
```

Figure 6.3: Graph Class

In depth error checking algorithms mentioned above had been implemented as functions wrapped on top of the Graph class blueprint shown in Figure 6.3.

The local client running on the ESP32 also has a part to play in the autonomous navigation. A FSM logic as shown in fig **??** has been implemented on the ESP32 to facilitate the entire autonomous behaviour.

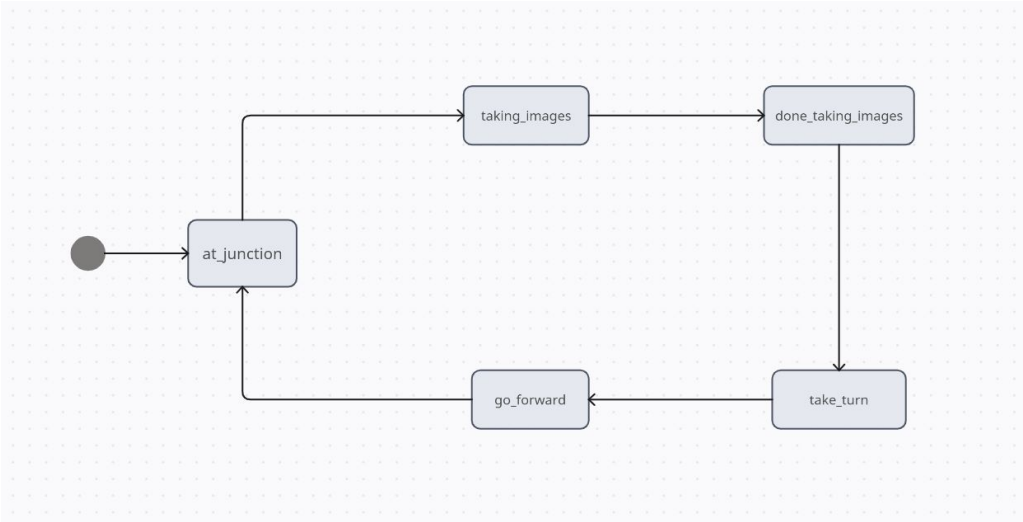


Figure 6.4: Autonomous ESP32 FSM

This completes the autonomous behaviour of the Rover. For further reference on exact implementation of each autonomous systems, see Section Appendix 11.4 as well as 7.

Web Application and Server

7.1 Overview

In this project, the bot used an ESP32 microcontroller to navigate the maze and collect data at junctions using the FPGA-Camera system as discussed above. This data is sent via Wi-Fi to a Python server on AWS EC2. The server processes the data, checks against a DynamoDB database for previously visited nodes, and sends back navigation decisions. Concurrently, the server communicates the robot's movements to the web application hosted on the same EC2 instance but on a different port. This web app, with a React frontend and Node.js back-end, visually displays the robot's real-time location and path on the map. The system integrates robotics, cloud computing, and web technologies for an interactive navigation experience. Figure 7.1 below portrays the communication architecture layout involved.

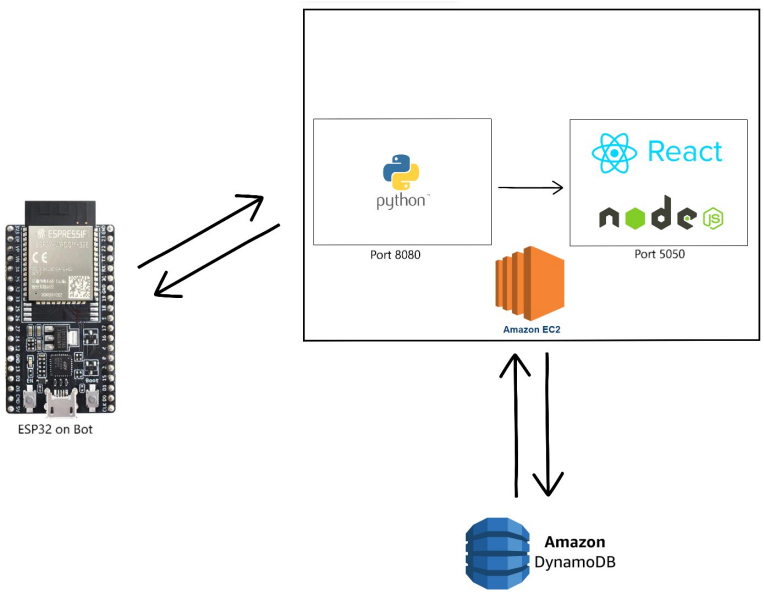


Figure 7.1: Robot-Server Communication Layout

7.2 Server and Communication

The communication and data flow between the ESP32 micro-controller on the bot, the intermediate Python server based on Flask, and the React/Node.js web application have been carefully managed in this project. Figure 7.1 shown above forms the base of the discussion in this section. Codes for each of the elements here are available on GitHub repository found in the Appendix 11.4.

We start with the ESP32 micro-controller, which functions as the client communicating with the Python server. The ESP32 is built with an integrated Wi-Fi module, exploited to set up a TCP/IP link. Figure 7.2 below shows the part of the code residing on the ESP32 responsible for connecting the robot to the Wi-Fi network. The complete code is available on the team's GitHub repository.

```
const char* ssid = "maze_master";
const char* password = "password";

const char* serverURL = "http://192.168.43.199:8080/process";

void connect_WiFi() {
  WiFi.begin(ssid, password);
      ...
      ...#Complete code can be found in the GitHub Repository
      ...
}

String send_and_receive(float Node_Data[4][4]) {
  HTTPClient http;
  http.begin(serverURL);
  http.addHeader("Content-Type", "application/json");
      ...
      ...#Complete code can be found in the GitHub Repository
      ...
  http.end();
}
```

Figure 7.2: ESP32 - Connecting to Wi-Fi

The SSID and password facilitate the authentication of the ESP32 on the network, while the `WiFi.begin()` method commences the connection. The server URL is assigned to `serverURL`, where the IP address and port number direct to the Flask-based Python server hosted on AWS. The `HTTPClient` class is included and engaged to dispatch an HTTP POST request, and headers are incorporated to define the content type. The payload, a 4x4 matrix of floating-point numbers denoting data points, as shown in Figure 6.2 above in the Mapping Algorithm section, is serialized into a JSON format and transmitted as the request body. The function `send_and_receive(float Node_Data[4][4])`, as can be seen in the complete code on GitHub (Appendix 11.4), takes care of both the sending and receiving of data between the ESP32 microcontroller and the Python server. It sends the POST request and captures the subsequent response using `response=http.getString()`, and returns the response.
Next, the Python server, crafted using the Flask framework, serves as a go-between, processing the ESP32 data and interacting with the React/Node.js web application. The code snippet in Figure 7.3 below highlights the key aspects of this Python server in terms of receiving the data sent to it from the ESP32, whenever the `send_and_receive(float Node_Data[4][4])` function is called.

```
from flask import Flask, request, jsonify
import sys

app = Flask(__name__)

@app.route('/process', methods=['POST'])
def process():
    data = request.get_json()
    response = process_data(data)
    return jsonify(response)

def process_data(data):
    #Any processing of the data can be done within this function, such as
    #storing and querying of the DynamoDB database or other computations
    return prcessed_data

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

Figure 7.3: Python Server - Receiving the Data

Note, the `0.0.0.0`, implies the IP to be the same as the hosting computer's IP, in this case the EC2 instance, and utilizes port `8080`. The server is receptive to HTTP POST requests at the `/process` endpoint. The `request.get_json()` method extracts the JSON payload from the request and it undergoes processing via the `process_data(data)` function, which entails our next-turn decision-making algorithm as discussed in Section 6.3 above and interfacing with the DynamoDB database that stores data for all the junctions visited and previously taken turns. The computed response is then sent back to the ESP32 via `return jsonify(response)`, which makes the bot take its next turn based on this response. The entire code residing on this server can be found in our GitHub repository under the 'Server' folder.

This Python server also establishes communication with the React/Node.js application initiating another HTTP POST request from the Python server to the web application. This web application is hosted on port 5050, as of Figure 7.1. This POST request carries the mapping directions, decided by our algorithm running on the Python server, using JSON format.

Coming to the React/Node.js domain, the application is set up to accept the inbound HTTP POST request. The Node.js backend employs the Express.js library for this request handling. Upon reception, the JSON data is extracted and relayed to the React frontend, which employs this data to refresh the state of its components, consequently triggering a re-rendering with the updated data. Details on the web application can be found in section 7.3 below,

To sum it up, the ESP32 collects data and transmits it in JSON format to the Flask-based Python server. The Python server processes this data, liaises with the database as needed, and sends a response back to the ESP32, and a mapping data to the React/Node.js web application. The web application, in turn, renders this data in real-time. This architecture is distinguished by a sequence of data processing and communication, where each component fulfills a particular role.

7.3 Web Application

An immersive web application is developed using React and Node.js that serves as an interactive map, based on a grid system. This web application is running on port 5050 of the EC2 instance on AWS, as shown above in Figure 7.1. The front-end look of the web app can be seen in Figure 7.4 below, with the 'blue' dot marking the starting position of the bot, 'yellow' dots being the junctions identified, and the 'red' dot representing the bot's live position. The white line represents the trail of the bot, creating the maze's map. The key highlight of this web app is the dynamic movement of the 'red' dot, which moves in response to the control signals received from the Python server running on port 8080 of the same EC2 instance. This seamless integration allows for real-time mapping and visualization of the bot's movements on the real-world maze.

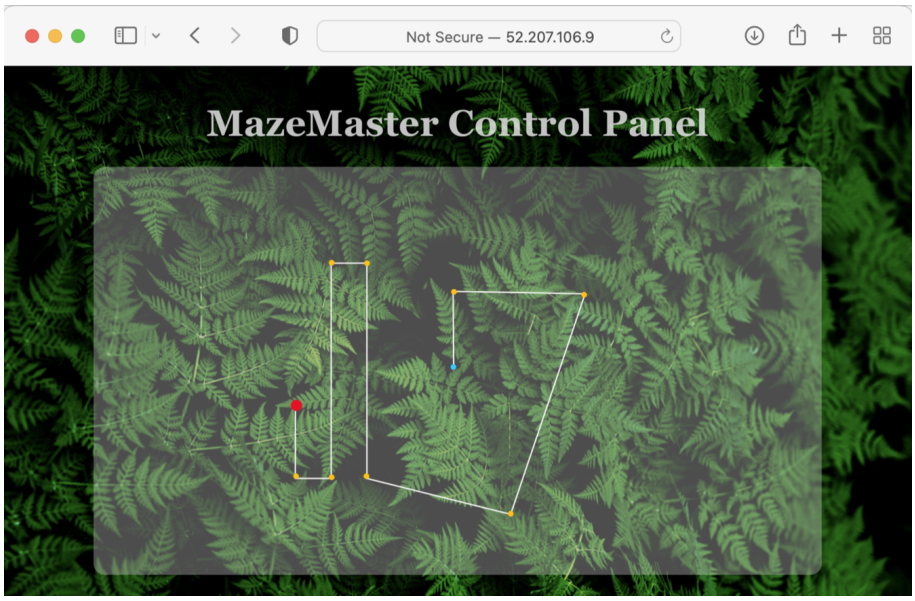


Figure 7.4: Web App Front-End

The grid, with its expansive size of 600 by 432 cells, provides a vast playground for exploration and navigation. The choice of this grid size is based on the compromised 'grid-area:visibility' ratio, providing a good enough grid space, and still maintaining decent visibility on the map. As the bot autonomously moves through the maze, based on decisions sent to the ESP32 from the Python server, the server also sends those decision choices to this web app, instructing the red dot on the map to move in specific directions. The dot dynamically adjusts its position on the grid, reflecting the bot's movements in real-time. This real-time synchronization creates an engaging and interactive experience for users, allowing them to witness the bot's path unfold before their eyes.

One intriguing aspect of this application is the representation of movements at certain angles. By cleverly utilizing the grid system, movements along diagonal angles can be represented by a combination of going up/down and left/right. This technique enables the bot to draw angled lines on the map, seen as tilted white lines in Figure 6.2 above, expanding its range of movement beyond the traditional four directions. Users can observe the precise lines and angles formed by the bot's movements, offering a visual representation of its navigational capabilities.

The Python server efficiently processes the data from the ESP32 and sends it to the Node.js back-end of the web app in the form of discrete integers - 1, 2, 3, 4, and 5, which correspond to the control signals for moving forward, turning right, turning left, moving backward, and node identification, respectively. Notably, the communication between the Python server and the React+Node.js web application is unidirectional. The Python server only sends these values to the web application, which subsequently maps them into visual representations.

The combined operation of React's robust front-end capabilities with the Python server's adept handling of control signals results in a versatile and adaptable system, well-suited for an array of interactive applications. With real-time visualization of the robot's trajectory, and the capacity to depict angled paths within a grid, this web application emerges as an exemplary tool for educational demonstrations, robotics experimentation, and simulation exercises. Additionally, it fulfills all the front-end requirements needed for this project.

7.4 Database

In this project, we use DynamoDB, an Amazon Web Services offering, a highly performant NoSQL database, ideal for handling large amounts of data. In the context of this project, it plays a pivotal role in storing and retrieving information about the nodes or junctions that the robot encounters, including a record of the turns taken. The Figure 7.1 seen in the overview section above highlights the 2-way communication between the EC2 instance (where the Python server resides) and the DynamoDB database.

To start, a table has been set up in DynamoDB, which involved defining a Node_ID as our primary key, to uniquely identify each junction or node. Read and write capacities has been configured so as to allow the Python server to query and write Node_Data into this table. With auto-scaling options enabled, the table expands as new Node_Data arrive into the database. Each row in this table is of the form shown in Figure 7.5 below.

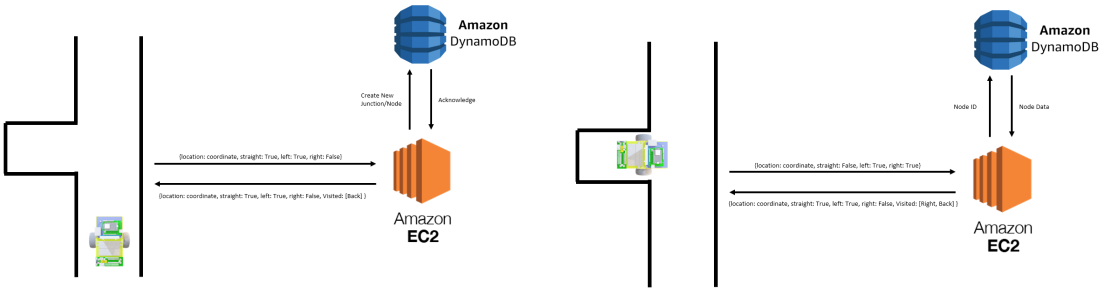
Node_ID	Turn 1	Turn 2	Turn 3	Turn 4	Turns Taken
001	$[100, 25, 90, 320]$	$[10, 115, 0, 150]$	$[0, 50, 0, 10]$	$[0, 0, 0, 0]$	$[0, 0, 1, 0]$

Figure 7.5: Sample Dataset Stored in the DynamoDB Databse

With the table setup complete, the Python server processes the 4x4 array of data received from the ESP32. This array, representing the junction contains information regarding the area of the beacons spotted and the angle of the rover's heading for each possible opening of the junction. The Python server uses Boto3, an AWS SDK for Python, to store this data in the DynamoDB table. Each element in the table is the viewing of the rover for a possible turn, characterized by the data points collected by the FPGA-Camera module (area of the beacons) and the rover's angle of heading from a compass connected to the ESP32.

The retrieval and querying of data are where DynamoDB proves its optimality. As the robot approaches a junction, the Python server must check whether this node has been previously encountered. Through the use of the 'get_item' method, the server queries the DynamoDB by providing a set of data to check against. In the event that the node already exists in the database, the Python server retrieves the history of turns taken. Based on this, the algorithm decides the robot's subsequent movements and updates the record of turns taken using the 'update_item' method. This appends a new turn to the list of turns taken. And, in case this is a newly encountered node, the Python server uses the 'put_item' method to add this new node's data into the table along with the record of the first turn to be taken.

Figures 7.6a and 7.6b below, illustrate the data transfer mechanism from the robot to the server on the cloud and involves the fetching and querying of the various node data on the DynamoDB database, which aids in making the next turn decision and stores data regarding the previously taken turns.



(a) Robot Visiting Junction for First Time **(b)** Robot Visiting Junction for Second Time

Figure 7.6: Transfer of Data between ESP32, EC2 and DynamoDB

In conclusion, DynamoDB is integral to the effective tracking and navigation of the robot through the maze. By storing, retrieving, and updating node information, the Python server can make informed decisions regarding the robot's path. Moreover, the scalability and performance of DynamoDB help us ensure that the system remains responsive and efficient.

Power Grid and Beacons

8.1 Overview

The power grid and beacons system described bellow consists of several components working together to ensure reliable and efficient power supply to the beacons. The PV panels play a crucial role in generating power from sunlight. A boost SMPS was used to extract maximum power from the PV panels. The LED driver acts as a current source to ensure consistent light output from the LEDs. Collaboration and teamwork was needed to determine the desired current to achieve optimal visibility of the LEDs. To address fluctuations in power delivery from the PV panels, a supercapacitor was added. The addition of a bidirectional SMPS further improves the configuration by allowing controlled charge and discharge of the supercapacitor.

Overall, this power grid and beacons system ensures efficient power generation, stable LED current, and reliable power supply to the beacons, enhancing visibility and functionality.

8.2 Circuit Diagram

The voltage and current range for each node in the circuit was depicted in a rough diagram, taking into consideration the limits of our devices. The range for node 1 in Fig. 8.1 was selected as $7-18V$, based on the voltage limit of the capacitor ($18V$) and the lower limit of the LED driver ($7V$). In order to achieve this desired range, the PV panels will be connected in parallel, and a boost SMPS will be utilized. A current range of $0-300mA$ was established for node 2 to ensure consistent light output from multiple LEDs, which was determined by the limits of the LEDs. As the brightness of an LED is directly influenced by the current flowing through it by precisely specifying the desired current level, we ensure uniform and reliable illumination across all the LEDs.

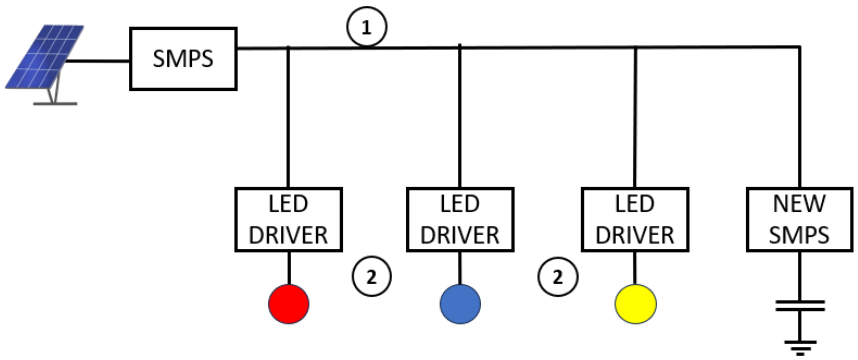
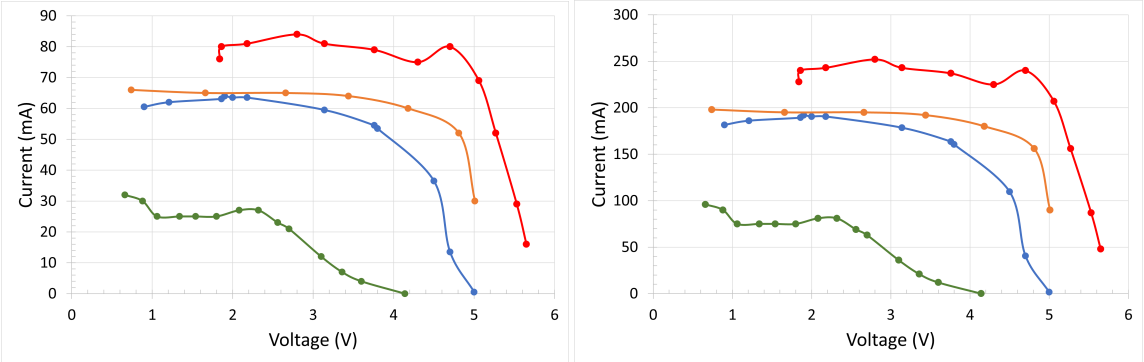


Figure 8.1: Rough Circuit Diagram

8.3 PV Panels

The PV panel was characterized as our first practical task. To gain a deeper understanding of the device's behavior under different light intensities, an IV curve was generated. This curve served as a valuable tool to analyze the relationship between current and voltage. A boost SMPS was

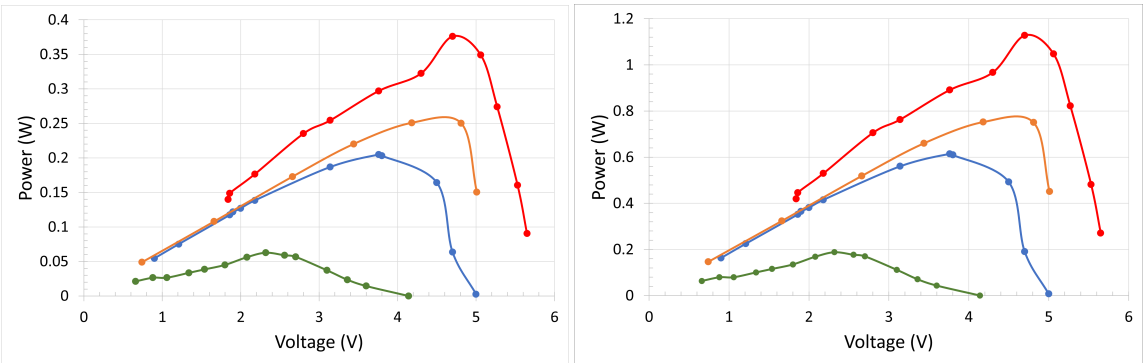
connected to the PV panel to draw various currents from it and measure the voltage across its terminals. The task posed challenges due to the fluctuating light intensity, limiting the number of readings that could be taken for a single curve.



(a) 1 PV Cell **(b)** Extrapolation- 3 PV Cell Connected in Parallel

Figure 8.2: IV Chart

Furthermore, the behavior of a PV panel was emulated using a power supply unit, utilizing the aforementioned graph. By calculating the gradients of the slopes in the curve, two resistors were connected to the power supply - one in parallel and another in series. This method enabled the accurate simulation of the PV panel's characteristics. The values of these resistances were determined to be 11.2Ω for the series resistance and 535.7Ω for the shunt resistance through our analysis.



(a) 1 PV Cell **(b)** Extrapolation- 3 PV Cell Connected in Parallel

Figure 8.3: Power Chart

In the boost configuration, the SMPS connected to the PV panels employs an MPPT perturb and measure algorithm to extract the maximum power from the PV panels.

```
if (Boost_mode){
  if (CL_mode) {
      power = vb*iL;
      if (power > prev_power){
        closed_loop = closed_loop + step;
      }
      else if(power < prev_power){
        closed_loop = closed_loop - step;
      }
      else{
        closed_loop = closed_loop;
      }
      closed_loop=saturation(closed_loop,0.99,0.01);
      pwm_modulate(closed_loop);
      prev_power = power;
  }
```

Figure 8.4: MPPT Algorithm Implementation

The code above is from the code we were provided for our ELEC50012 Power Electronics and Power Systems Labs (17) for the operation of the Boost SMPS and includes additional code within the closed-loop system. Here, the variable "step" is assigned a value of 0.001, "power" stores the current input power to the boost SMPS, and "prev_power" stores the input power to the boost SMPS in the previous iteration. Initially, the variable closed_loop is 0.

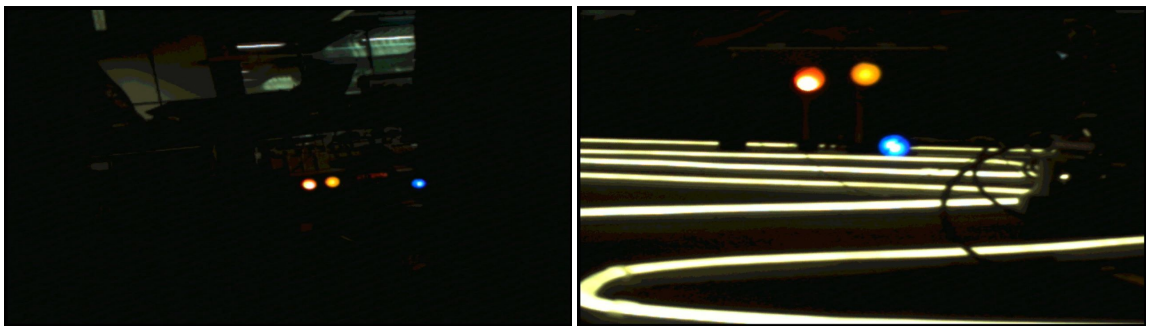
The input power from the PV panel is continuously tracked by the algorithm through the adjustment of the duty cycle. If a higher input power is observed upon increasing the duty cycle, it is further increased. On the other hand, if the input power decreases, the duty cycle is decreased to return to the previous higher input power. This iterative process ensures the stabilization of the input power around its maximum value, optimizing the performance of the system.

8.4 LED Driver

As previously mentioned, the LED driver functions more as a current source. The exact current was determined in collaboration with the camera team, as an overly bright LED would appear white to the camera, while a dim LED might not be visible at all. After conducting tests, a consensus was reached that a current of $0.2A$ would be ideal for the camera, Fig.8.5a. Taking into account the varying tolerances of each LED, the values in Table 8.1 were obtained. As depicted in Fig.8.5b, the colored LED exhibits an appropriate level of brightness within the maze, ensuring visibility without appearing excessively white.

Beacon	Voltage (V)	Current (A)	Power (W)
Blue	3.14	0.203	0.637
Yellow	2.14	0.201	0.430
Red	2.3	0.202	0.465

Table 8.1: LED Values



(a) LEDs **(b)** LEDs in The Maze

Figure 8.5: Visibility of LEDs

A PID controller was implemented to regulate the desired current. Through iterative testing, starting with low currents, we fine-tuned the gains and the wind-up values. Care was taken when setting the integral gain to avoid the accumulation of excessive current error. To address this, limits were established to ensure that the integrator sum remained within an acceptable range.

```
#Current control loop
oc = setpoint - current #Calculate current error

integral += oc * delta
ref = kp * oc + ki * integral
pwm_ref = pwm_ref + int(ref)

pwm_out = saturate(pwm_ref) #Keeps in limits
pwm.duty_u16(pwm_out)

#Stops integral from getting too large, wind-up
if integral > 50:
    integral = 50
elif integral < -50:
    integral = - 50
else:
    integral = integral

#Current limitter -Keeps current lower than LED limit
if current > current_limit :
    pwm_ref = pwm_ref - 100
```

Figure 8.6: Python code for controlling current and limiting LED current

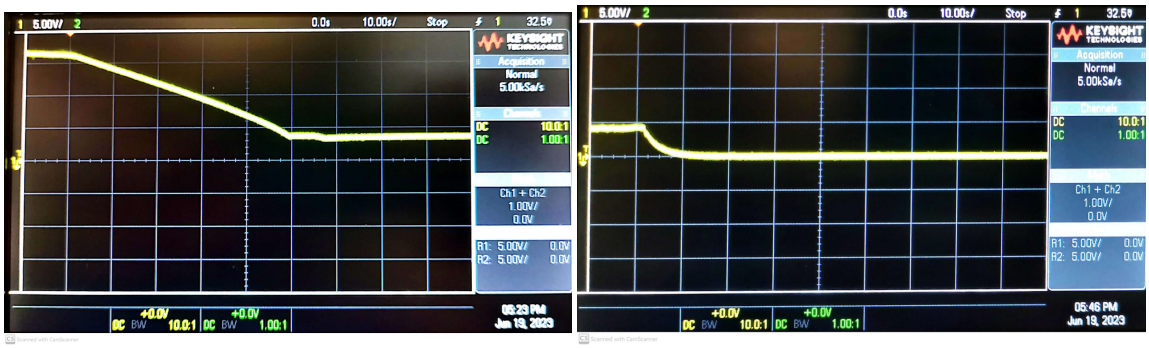
To provide protection against potential damage to the LED, a current limiter was integrated into the code. This preventive measure ensured that the output current did not exceed the LED's limit of $0.3A$. By incorporating an if statement, the duty cycle was reduced if the measured output current exceeded this threshold.

8.5 Bidirectional SMPS and Supercapacitor

The power delivered by the PV panel is not constant and would fluctuate constantly. Even though the LED drivers are configured with a control system to ensure constant output current in the presence of varying input power, it is not desirable to have insufficient power at the input of the LED drivers. To ensure sufficient power flow even in the presence of fluctuations and insufficient input power, a supercapacitor is added in parallel to the output of the boost SMPS. The capacitor charges up in the presence of input power and will store the energy it has gained while charging up. The effect of the supercapacitor can be observed on an oscilloscope in Fig. 8.7a.

(a) With Capacitor **(b)** Without Capacitor

Figure 8.7: Charging Voltage Measurements



(a) With Capacitor **(b)** Without Capacitor

Figure 8.8: Discharging Voltage Measurements

Figure 8.7b shows the absence of the supercapacitor for comparison to Figure 8.7a which shows the presence of the supercapacitor place in parallel to the input of the three LED drivers. The output voltage increases at a slower rate due to the addition of the supercapacitor to the DC grid. If the input power fluctuates due to fluctuations in the input power provided by the PV panel, the output voltage of the SMPS will not immediately fluctuate and will be relatively constant. When the PSU being utilised for emulation was turned off, it was observed in Figure 8.8a that the voltage on the DC grid falls relatively slowly with the presence of the supercapacitor which is beneficial for the LED drivers as they can continue to be powered because the input voltage to the LED drivers is within the operating range for longer. The measurements for 8.8a were made when only two LED drivers were placed parallel to the supercapacitor.

When the output power at the output of the boost SMPS decreases, part of the power to the beacons should be provided by the supercapacitor which would discharge in order to continue the power flow. Therefore, a source of sufficient power would be maintained which would allow the functioning of the beacons in periods of insufficient power.

The configuration of the supercapacitor can be improved by implementing a bidirectional SMPS along with the supercapacitor. The input of the bidirectional SMPS would be connected to the output of the boost SMPS and the supercapacitor would be connected to the output of the bidirectional SMPS. The overall configuration of the DC grid would then be the one provided in Figure 8.1.

When there is sufficient power delivered by the PV panel, the bidirectional SMPS would operate in the configuration of a boost SMPS. A control system can be implemented to adjust the duty-cycle to ensure that the output voltage of the bidirectional SMPS corresponds to the maximum voltage rating of the supercapacitor. The reason behind this is that the energy stored in a capacitor is proportional to the square of the voltage across it and having the maximum possible voltage across it would maximize the energy storage. The SMPS we were provided has a resistor at the input of side of the boost configuration specifically for measuring the current so the input power can be obtained if the bidirectional SMPS also has a similar configuration. When there is insufficient power delivered by the PV panel, the direction of energy flow through the bidirectional SMPS

would be from the supercapacitor to the DC grid. The beacons altogether require a minimum of approximately $1.2W$ to function properly as observed through testing and power below this could be considered insufficient. In this case, the bidirectional SMPS would be in the configuration of a buck SMPS with a control system that regulates the output voltage and also limits the current. The benefit of having the supercapacitor in this configuration along with a bidirectional SMPS would be that it would allow a more controlled charge and discharge of the supercapacitor which would allow controlled and efficient power transfer from the supercapacitor whenever necessary.

8.6 Full Grid Testing

After completing the assembly of the entire beacon and power grid circuit, certain modifications were required due to the direct connection of the capacitor to the DC grid. At the maximum light intensities, the voltage on the DC grid reached levels as high as $20V$, which exceeded the operating limits of our components. To address this issue, a voltage limiter was incorporated into the boost SMPS along with the MPPT algorithm. The limit was set to $17V$, slightly below the capacitor's maximum limit of $18V$, ensuring the protection of the components.

Integration and Testing

Integrating the different components of the rover together is a paramount task within the project. The individual components though working, needs to be working in union to give the desired result. Each individual testing done has been mentioned in the table 9.1.

The FPGA does the image processing and is mounted with the ESP32 microcontroller which controls the entire rover in terms of sending data to server, sending inputs for movements as well as executing the autonomous state machine as mentioned in figure 6.4.

The ESP32 is also integrated with the Arduino Nano using different digital pins allowing us to transmit driving instructions to the controller within the Arduino Nano. The Arduino Nano itself is connected to the stepper motors DIR and STEP pins for driving them and also connected with the MPU6050 for processing incoming gyroscope data.

Outside of the rover, the ESP32 is connected to the Python server wrapped in the flask framework, which accepts POST requests from ESP32 and returns processed information regarding the rovers next movements, based on the DFS search algorithm as mentioned in section 6.1.

Furthermore, the Python server sends positional data to the React WebApp to plot the movements of the rover and allow an appealing visual display and control panel to the users.

Our complete software stack includes:

- Python - for server on AWS
- React/Node.js - for webapp front and back-end
- C - for embedded soft-core processing on FPGA
- C++ - for ESP and Arduino programming
- SystemVerilog/Verilog - for image processing
- Matlab - for preprocessing and research

Complete Structural Diagram:

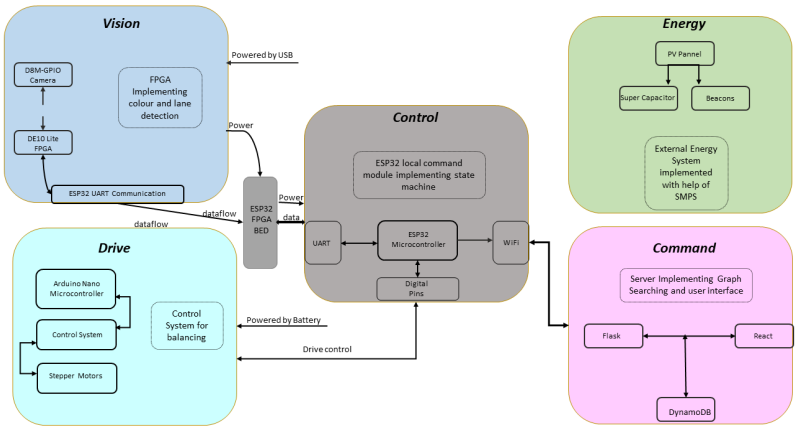


Figure 9.1: Rover Structural Layout

Table 9.1: Results of Controller Testing against Design Criteria

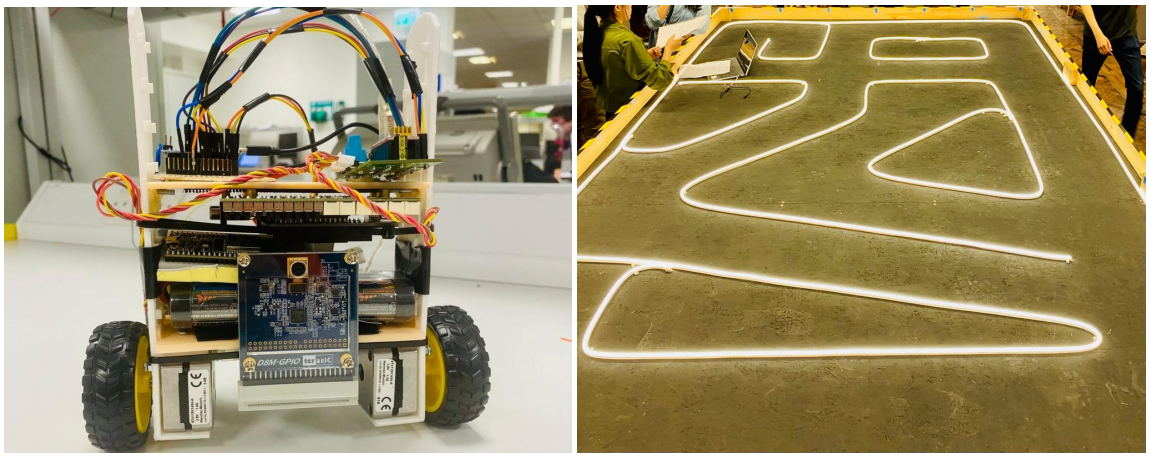
Criteria	Results from Testing Methodology	Steps to Improve
Rover Balancing and Stable Movement	Rover demonstrated satisfactory maneuverability and stability when navigating through most junctions. It encountered problems for junctions located on the maze boundary, where space is limited. The rover had to make multiple turns to complete a full rotation in such situations.	The wheelbase (distance between wheels of the rover) affects its turning capabilities. Our rover has a wide wheelbase, which enhances stability and weight distribution. But, a wider wheelbase leads to a larger turning radius. This trade-off between stability and maneuverability needs to be tuned.
Autonomous Navigation	The rover autonomously navigated by using the FPGA camera to detect the 3 beacons, however, was unable to capture the white LED strips that were in close proximity to the rover.	Due to the FPGA camera module's limited field of view, a solution would be adding a fish-eye lens allowing the camera to detect close-by white LED strips.
Real-time Mapping	Rover was unable to fully map the maze, the parts of the maze that were accurately mapped were correctly shown in the web app, verifying that it works as expected in the overall system. When running a traceroute tool, the time between ESP32 sending the position data and the server receiving it and updating the web app was 20ms. Considerably lower than our anticipated 50ms.	The testing requirements for this criteria were sufficiently met.
Reliable Data Transmission	Server ensured that critical movement directions were reliably sent to the ESP32. Data transmission from ESP32 to the server for maze mapping resulted in some incorrect or lost values. To ensure reliable data transmission between ESP32 and server.	To ensure reliable data transmission between ESP32 and server, a timeout mechanism can be implemented. If an acknowledgement isn't received within a specified time, the data is retransmitted. This mechanism doesn't disrupt the transmission of other data, allowing the system to continue real-time mapping without interruptions.
Power Management	After the implementation of a PID controller for the LED drivers, the MPPT algorithm for the boost SMPS, and the addition of the supercapacitor parallel to the DC grid, the beacons stay sufficiently bright and this brightness stays relatively constant subject to variations in the input power.	The testing requirements for this criteria were sufficiently met.
System Integration and Control	Unit testing was done for each component. Comprehensive system compatibility testing was also conducted, including testing with edge cases. During this testing, a situation was identified where the system failed.	An edge case where the system failed was when the server, and consequently the ESP32, sends no movement commands to the Nano which controls the rover. In this scenario, the Nano continually ran the last received movement. To address this, improvements were made to the rover's movement logic to automatically default to an idle state when no data is sent.

Conclusion and Remarks

10.1 Final Views and Remarks

In conclusion, the robot we made passed through the trials required to meet its core requirements. Moreover, the processes such as positioning of the sensors and individual components were optimised to increase the durability of the stabilization control system. By doing so, the process would not bottleneck by forcing the entire imbalances on the stabilization software and its stepper motors, rather it would get some support from the appropriate stack poisonings. The movement speed of the stepper motors was also adjusted to help us meet the optimal stability states whenever required during the development process.

After the successful integration of the various elements, including the ESP32 for on-board data processing and movement control, the Arduino Nano for the balance control, AWS EC2 for hosting a Python server and a web app, and DynamoDB for data storage, we achieved real-time tracking of the rover's movements, visualized on the web app, and the highlighted shortest path found. Thus, meeting all the requirements of this project as introduced in the 'Introduction and Background' section at the start of the report. Figure 10.1a below presents the final robot built by our team. And, Figure 10.1b shows the sample of a maze our robot is able to traverse through and successfully accomplish all the project targets. This robot is now ready to be placed and tested on any other test grounds as well, comprising a completely different path structure.



(a) EEBalanceBug by MazeMaster **(b)** Sample Test Ground (Maze)

Figure 10.1: Final Robot and Test Ground

This project proved to be an exciting venture into the domain of robotics, cloud computing, and real-time data processing. The blend of diverse technological aspects, combining elements from each and every modules we have studied so far, has paved the way to turn our educational and experimental learnings into practical implementation and to engineering a complete and fully functional product.

10.2 Suggestions for Future Works

Considering the time of submission of this report, further tasks and extensions are still planned on being be undertaken. It is possible to improve the rover in numerous ways. Some of the improvements require the sourcing of some low-cost components, some require structural modifications, and some can be achieved via the additions of new functions in the microcontroller code. Above all, the implementation of these improvements requires additional time, which was very limited for the scope of this project. The possible improvements that are suggested for future work includes, but is not limited to:

1. Use of Fish-eye lens:
   A fisheye lens is an ultra wide-angle lens that produces strong visual distortion intended to create a wide panoramic or hemispherical image (18). This allows us to have a better view and angle of the maze and led strips, helping us to detect turns and junctions right below us. We can use these to ease FPGA vision and lane detection.

2. Motorized arm for re-balancing:
   A 3D printed arm equipped to a small servo motor can be placed midway onto our robot. The arm would quickly react by rotating to the side of fall, and apply force on the ground to make the robot stand-up again, just like how a human would. This mechanism would get triggered whenever the gyroscope's tilt angle goes beyond a certain threshold, that indicates the robot's stability system has failed, and it has fallen to the ground. With this mechanism added, our robot would always stand back up and continue the stability algorithms, no matter how many times it fails or falls, and at the end, successfully complete mapping any possible terrains.

3. IR sensors at the front and back:
   Adding two IR sensors at the front and back of our robot, can help it prevent from hitting the LED strips that form the walls of the maze. It can be placed close to the ground level, so that whenever the robot rolls too close to the LED strips, it can stop itself from moving any farther in that direction, and may try to stabilise by moving to-and-fro within a position. This would reduce the chance of our rover hitting them and falling.

Bibliography

[1] Monday.com, "Project management software," 2023.

[2] GeekMomProjedts, "*MPU-6050 Redux: DMP Data Fusion vs. Complementary Filter* ," 2017.

[3] U. of Michigan, "*Introduction: Simulink Control*," 2011.

[4] O. Saleem, K. Mehmood-ul Hasan, and M. A. Imtiaz, "*Attitude Control and Stabilization of a Two-Wheeled Self-Balancing Robot*," 09 2015.

[5] Mathworks, "*Inertial Reference Frame* ," 2023.

[6] Arduino, "*arduino nano technical specifications*."

[7] J. Joseph, "*ESP32 Timers Timer Interrupts*," 2022.

[8] Ardupilot, "*How Balance Bots work* ," 2023.

[9] A. MicroSystems, "*DMOS Microstepping Driver with Translator And Overcurrent Protection*," 1990.

[10] Stepperonline, "Stepper motor datasheet," 2005.

[11] Easyeda, "*JLCPCB and LCSC Software*," 2017. Version 6.5.23.

[12] H. Merdan, "*EEEBalanceBug Horizontal Design Model*," 2023.

[13] H. Merdan, "*EEEBalanceBug Vertical Design Model*," 2023.

[14] H. Merdan, "*EEEBalanceBug Github repository*," 2023.

[15] J. D. Edgcombe, "*Hardware Acceleration in Image Stitching: GPU vs FPGA*," Master's thesis, Grand Valley State University, August 2021.

[16] A. Ronacher, "*Flask framework*." Version 2.3.3.

[17] D. P. Clemow, "*Boost_ OL*," 2023.

[18] Wikipedia, *Fisheye Lens*.

Appendix

11.1 Cost of Development

Though several iterations of design was undertaken, all were done cautiously not only considering feasibility, scalabilty and reliability but also affordability. Given the constraints several decisions and limitations were in placed to work within the given budget as well as within a reasonable cost. Therefore, few additional materials were used:

- Arduino Nano: An Arduino Nano was used to implement the Control system independently to other components. A Nano was relatively much more simpler, lighter and smaller which made it an ideal choice. Further, the power consumption is significantly lower compared to other microcontrollers (6).

- 3D printed Chassis: A new chassis had been used as mentioned in Section 4. This design was completely 3D printed.

- Magnetometer Sensor: A simple compass module is used to uniquely identify turns within a node.

These adds up on top of the given parts which are the two stepper motors, gyroscope MPU, FPGA and terrasic camera module, motor driver PCB and a ESP32 microcontroller. A few other parts used during the development phase include other 3D printed chassis, laser cut acrylic chassis, IR sensors for proximity, jumper cables, wires, breadboard and strip-board.

11.2 FPGA Resource Utilization

Fitter Resource Usage Summary	
Resource	Usage
Total logic elements	10,931 / 49,760 (22 %)
-- Combinational with no register	4099
-- Register only	1574
-- Combinational with a register	5258
Logic element usage by number of LUT inputs	
-- 4 input functions	4136
-- 3 input functions	3007
-- <=2 input functions	2214
-- Register only	1574
Logic elements by mode	
-- normal mode	7479
-- arithmetic mode	1878
Total registers*	6,899 / 51,509 (13 %)
-- Dedicated logic registers	6,832 / 49,760 (14 %)
-- I/O registers	67 / 1,749 (4 %)
Total LABs: partially or completely used	871 / 3,110 (28 %)
Virtual pins	0
I/O pins	171 / 360 (48 %)
-- Clock pins	3 / 8 (38 %)
-- Dedicated input pins	1 / 1 (100 %)
M9Ks	173 / 182 (95 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)
Total block memory bits	1,341,624 / 1,677,312 (80 %)
Total block memory implementation bits	1,594,368 / 1,677,312 (95 %)
Embedded Multiplier 9-bit elements	12 / 288 (4 %)
PLLs	1 / 4 (25 %)
Global signals	15
-- Global clocks	15 / 20 (75 %)
JTAGs	1 / 1 (100 %)
CRC blocks	0 / 1 (0 %)
Remote update blocks	0 / 1 (0 %)
Oscillator blocks	0 / 1 (0 %)
Impedance control blocks	0 / 1 (0 %)
Average interconnect usage (total/H/V)	7.8% / 7.8% / 7.7%
Peak interconnect usage (total/H/V)	25.3% / 25.8% / 26.1%
Maximum fan-out	3667
Highest non-global fan-out	808
Total fan-out	60204
Average fan-out	3.31

* Register count does not include registers inside RAM blocks or DSP blocks.

Figure 11.1: FPGA Resource useage

11.3 Excerpt of Arduino Nano Code with Interrupts

Initializing timer:

```
// Timer is set to mode where timer resets to 0 when timer equal interrupt value.
TCCR1B |= (1 << TIMER1_WGM12);
// Timer increments once every 8 cycles.
TCCR1B |= (1 << TIMER1_CS11);
// Timer is enabled and shall be compared to interrupt value stored in OCR1A
    register.
TIMSK1 |= (1 << TIMER1_OCIE1A);
\end{verbatim}

Mapping motor acceleration to interrupt value:
```

```
\begin{verbatim}
if (motor_accel == 0) {
    interrupt_value = ZERO_SPEED;
    dir_M1 = 0; // don't move motors in any direction
}
else if (motor_accel > 0) {
    interrupt_value = 2000000 / motor_accel; // Higher the motor_accel, lower the
    interrupt_value
    dir_M1 = 1;
    // Assign motor direction forward.
}
else {
    interrupt_value = 2000000 / abs(motor_accel);
    dir_M1 = -1;
    // Assign motor direction backward.
}
```

Listing 11.1: Initinalizing Timer

Function called when interrupt executed:

```

ISR (TIMER1_OCIE1A) { // ISR for stepper motor 1 interrupt

    // When motor1 ISR called we move motor by 1 step:

    if (dir_M1 == 0) { // If we are not moving, we don't generate a pulse
        return;
    }

    // Move forward by 1 step

    digitalWrite(STEP_PIN_MOTOR1, HIGH);
    delayMicroseconds(1);
    digitalWrite(STEP_PIN_MOTOR1, LOW);
}
```

Listing 11.2: Interrupt Execution

11.4 GitHub Repo - MazeMaster

For further references on the code and applications used see our GitHub repository MazeMaster GitHub at `https://github.com/alvi-codes/MazeMaster`.

11.5 Product Design Specification (PDS)

The PDS below serves as a comprehensive document outlining the requirements, features, and constraints for the development and design of EEEBalanceBug as of 2023:

Table 11.1: Design Brief

Design Brief	**Description**
Autonomous Self-Balancing Rover	An autonomous self-balancing two-wheeled rover will traverse a maze with walls marked by white LED strips. The rover will use an FPGA camera to detect LED strips and beacons, enabling maze mapping and autonomous navigation. A live mapping of the maze will be displayed on a web application, with the shortest path highlighted at the end.
Customer's Requirements (Functional)	- The rover must be self-balancing and capable of traversing and mapping the maze. - The live mapping of the maze must be provided, highlighting the shortest path upon completion.
Customer's Requirements (Non-functional)	- The rover should be lightweight, robust, reliable, cost-effective, and easy to use to ensure stability and successful maze traversal. - The web application should have an aesthetically pleasing design and intuitively display real-time mapping. - The project must be completed within a budget of 60 GBP.
Performance	- The rover should traverse the maze using only two wheels. - The camera should be able to detect LED light strips and beacons as references for navigation and maze mapping.
Power	The power system should ensure the LED beacons function properly even in the presence of power fluctuations or insufficient input power.
Environment	The rover will operate on a flat, smooth surface with walls indicated by LED white strips, effectively acting as obstacles.
Cost	The project budget is limited to 60 pounds.
Size	The rover should be compact to achieve better inherent stability. It should have sufficient space to accommodate all necessary peripherals for successful maze traversal, stability, and control.
Materials	Lightweight, robust, and durable materials that are not easily damaged by impact. Materials can be sourced from labs or purchased online.
Disposal	Breadboards can be used for repeated prototyping, and components such as wires and resistors can be reused in future projects.
Reliability	- Data transmission between ESP32 and server should be as reliable as possible. - The rover should remain balanced during different maneuvers, such as left turns, right turns, and forward movements. - The camera should have a high success rate in framing beacons and detecting LED strips, even on a balanced but potentially jittery self-balancing rover.
Aesthetic, Appearance, and Finish	The rover should present an image of robustness, compactness, and sleekness.
Ergonomics	The rover design should be modular to facilitate different center of mass configurations and weight distributions for stability testing. The final design should feature a stronger plastic chassis with screw inserts for secure fastening and robustness.