

ALFETTA
SELF BALANCING AND LINE FOLLOWING
ROBOT

Project Report

Eklavya Mentorship Programme

At

SOCIETY OF ROBOTICS AND AUTOMATION, VEERMATA JIJABAI
TECHNOLOGICAL INSTITUTE, MUMBAI

AUGUST-SEPTEMBER 2022

Acknowledgement

We're extremely grateful to our mentors Mateen Shah, Marck Koothoor and Chinmay Lonkar for their support and guidance throughout the duration of the project. We would also like to thank all the members of SRA VJTI for their timely support as well as for organizing Eklavya and giving us a chance to work on this project.

Raghav Agarwal

raghavagarwal3050@gmail.com

Dhruv Sapra

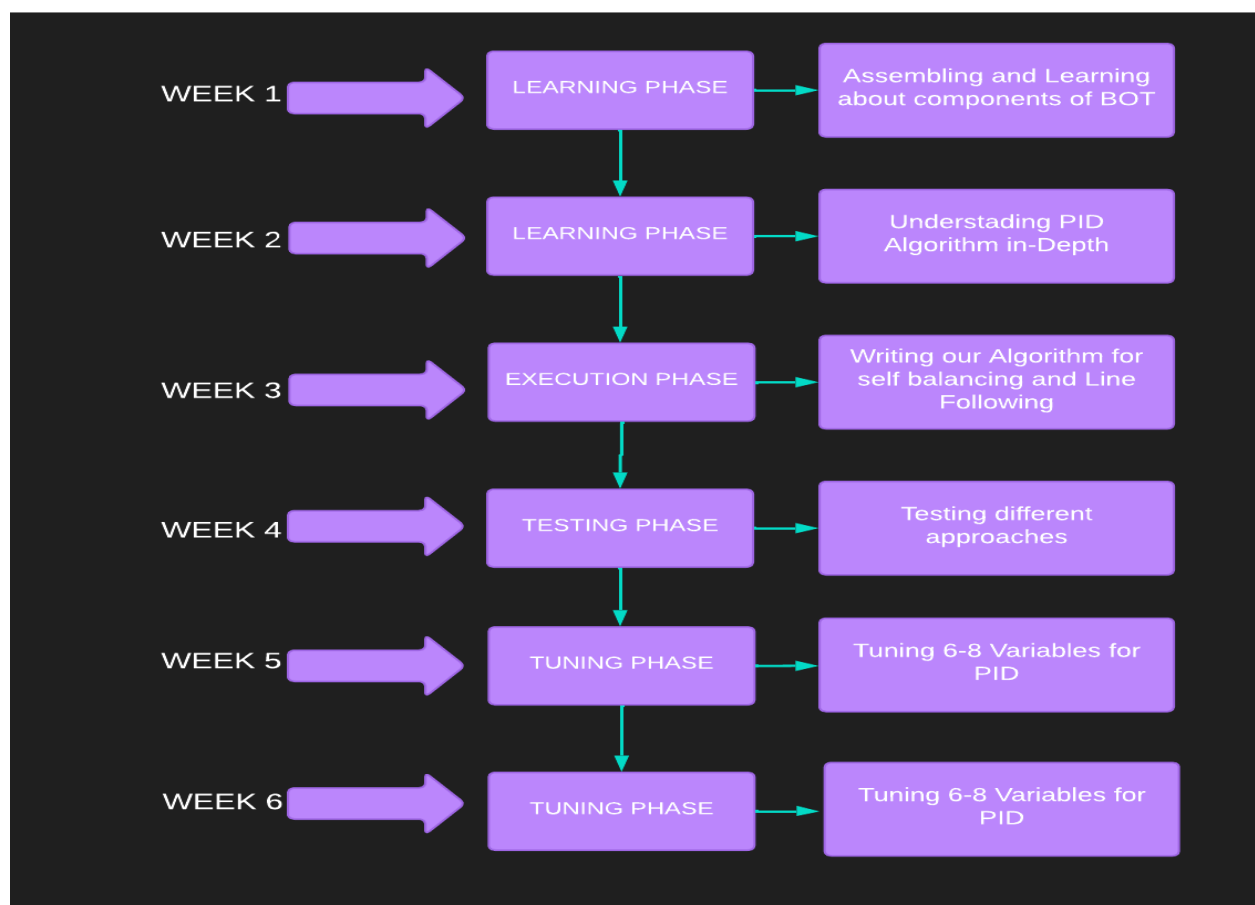
dhruvsapra88@gmail.com

TABLE OF CONTENTS

Project Overview	4
1. Introduction	5
1.1 Goal	
1.2 Introduction to Embedded C	
1.3 Introduction to ESP-IDF Framework	
2. In-Depth Analysis	6
2.1. Hardware	
2.1.1 Sensors	
2.1.1.1 Motion Processing Unit	
2.1.1.2 Line Sensor Array	
2.1.2 Motor and Motor Drivers	
2.1.3 SRA Board, Wheels, Microcontroller	
2.1.4 Other Components	
2.2 Software	8
2.2.1 Control Loops	
2.2.2 PID Controller	9
2.2.3 FreeRTOS	
2.2.3.1 What is FreeRTOS?	11
2.2.3.2 FreeRTOS Concepts	12
2.2.3.3 FreeRTOS Functions	
2.2.4 Roll , Pitch and Yaw	13
2.2.5 Complementary Filter	14
3. Implementation	15
3.1 Line Following Algorithm	
3.2 Self Balancing Algorithm	
3.3 Self Balancing and Line Following Algorithm	
4. Challenges Faced and The Solutions Applied	
4.1 Challenges and Solutions	
5. Applications	
5.1 Industrial Use of Line Follower Bots	
5.2 Used in Restaurants	
5.3 Advantages of Self Balancing Bot	
6. Conclusion and future work	
7. References	
7.1 Useful Links and Research Papers	

PROJECT OVERVIEW

- Line following robots may be used in various industrial and domestic applications such as to carry goods, floor cleaning, delivery services and transportation.
- Self-balancing two wheeled robot is advantageous over traditional four wheeled robots as it helps in taking sharp turns and navigating through tighter areas .
- In this project, we will implement a self balancing and line following robot .



INTRODUCTION

GOAL: To implement an efficient self-balancing and line-following algorithm.

EMBEDDED C :

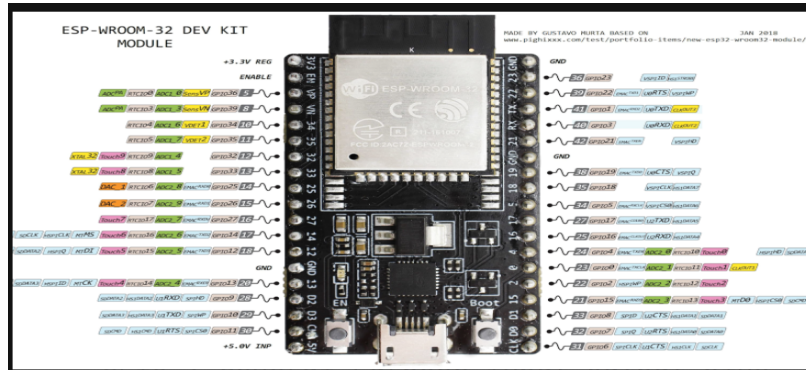
Embedded C programming typically requires nonstandard extensions to the C language in order to support enhanced microprocessor features such as fixed-point arithmetic, multiple distinct memory banks, and basic I/O operations. For our project , we have used the esp-idf framework.

ESP-IDF FRAMEWORK:

ESP32 is a system on a chip that integrates the following features:

- Wi-Fi (2.4 GHz band)
- Bluetooth
- Dual high performance Xtensa® 32-bit LX6 CPU cores
- Ultra Low Power co-processor
- Multiple peripherals

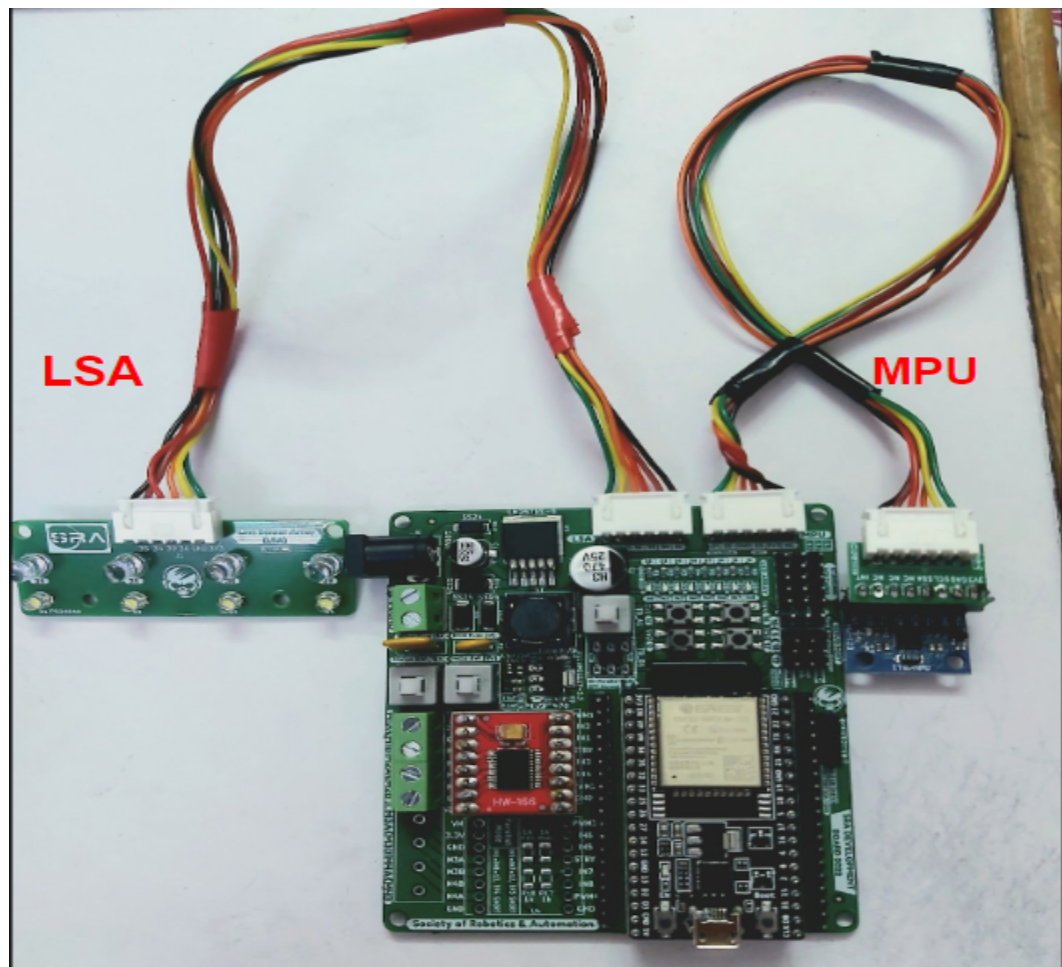
Powered by 40 nm technology, ESP32 provides a robust, highly integrated platform, which helps meet the continuous demands for efficient power usage, compact design, security, high performance, and reliability.



IN DEPTH ANALYSIS

2.1 HARDWARE

2.1.1 Sensors



- *Motion Processing Unit*

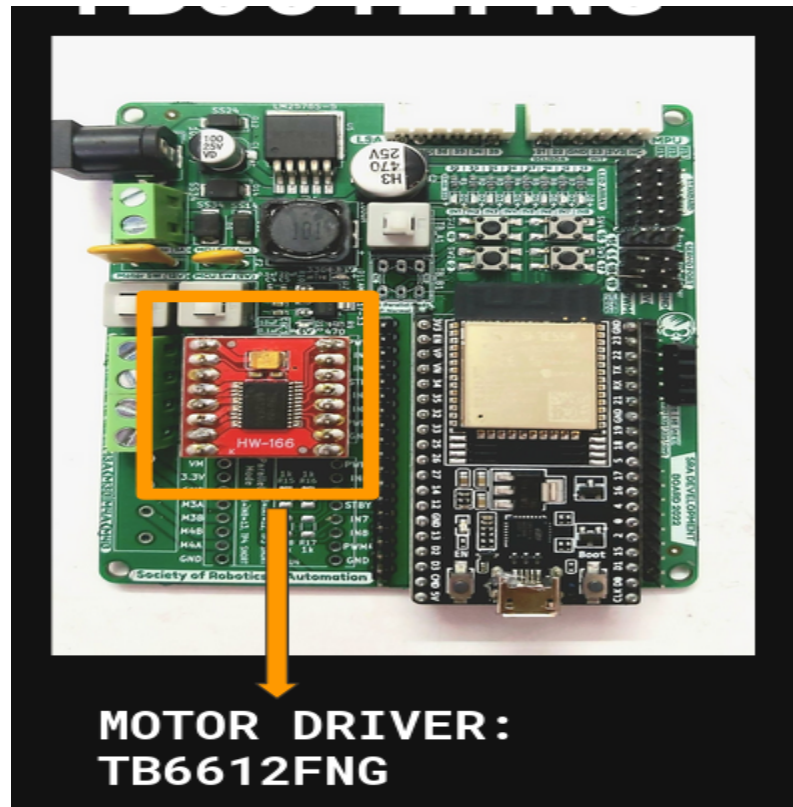
We use MPU6050 for this project to give us the pitch angle using which the bot can balance using PID controller .

MPU6050 : The MPU-6050 devices combine a 3-axis gyroscope and a 3-axis accelerometer on the same silicon die, together with an onboard Digital Motion Processor™ (DMP™), which processes complex 6-axis MotionFusion algorithms.

- *Line-Sensor Array*

Line Sensor Array which has set of LED's which emit light and after reflection the light is absorbed by photo-diode. The reflection percentage are not same for white and black surfaces and this becomes the differentiating factor and the sensor helps in distinguishing the paths for the bot .

2.1.2 MOTOR AND MOTOR DRIVERS



It listens to the low voltage (3.3V) from the microcontroller /processor and controls an actual motor which needs high input voltage(12V). The motor driver IC controls the direction of the motor based on the commands or instructions it receives from the controller.

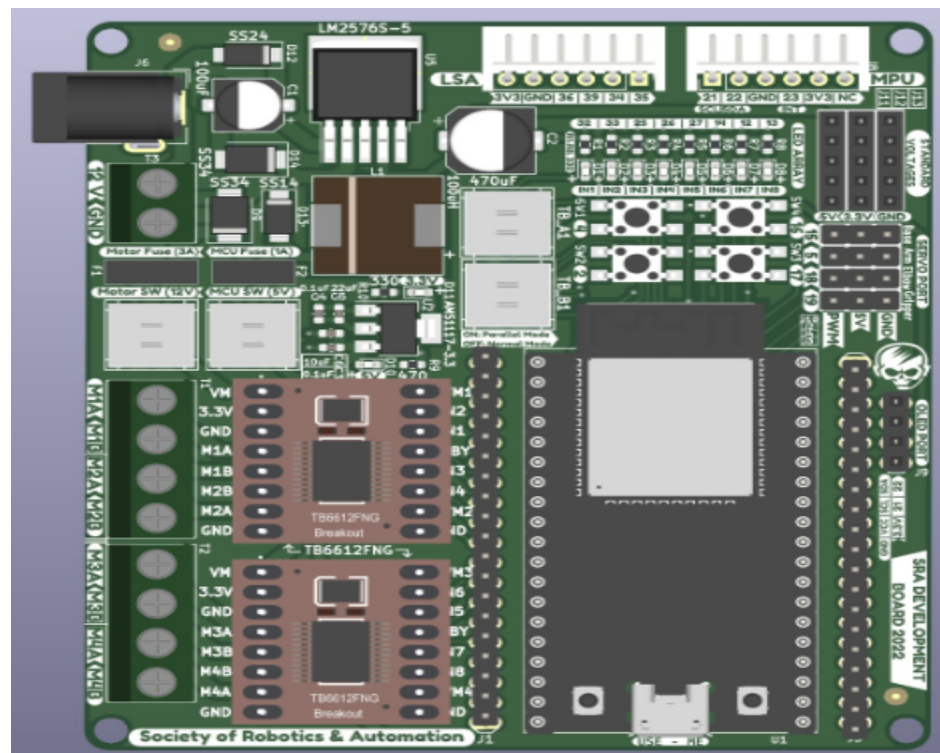
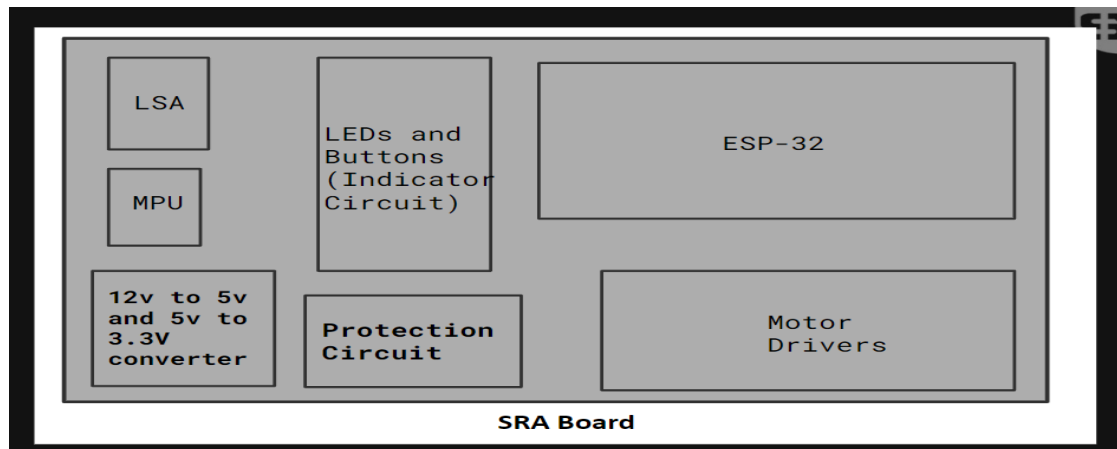


Above is a picture of BO motor used for this project . BO motors are low density ,lightweight and low cost motors running at 300 rpm at 12

V.

2.1.3 SRA BOARD and WHEELS

The PCB Board we used in our project is named SRA Board , given by our mentors . It has the following components :



WHEELS



The wheels used for the bot are BO motor wheels , made with high strength plastic and rubber.

2.1.4 Other Components Used

List of other small components used :

1. LM2596 Buck Converter (12v to 5v)
2. VOLTAGE REGULATOR: AMS1117 (5v to 3.3v)
3. Connecting Wires
4. Female to Female Connector
5. Sensor Ports
6. Switches and Leds
7. Fuses

2.2 SOFTWARE

2.2.1 Control Loops :

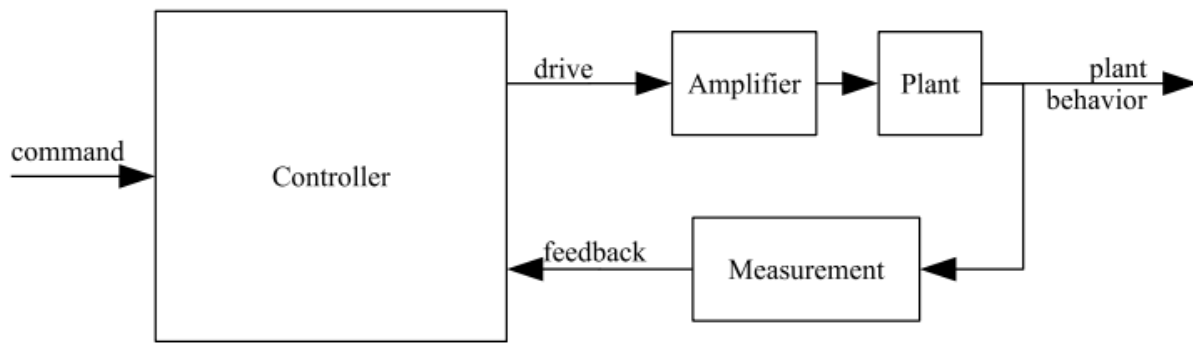


Figure 1: Anatomy of a control loop.

Plant : The object we want to control . In our case it is the Walle bot.

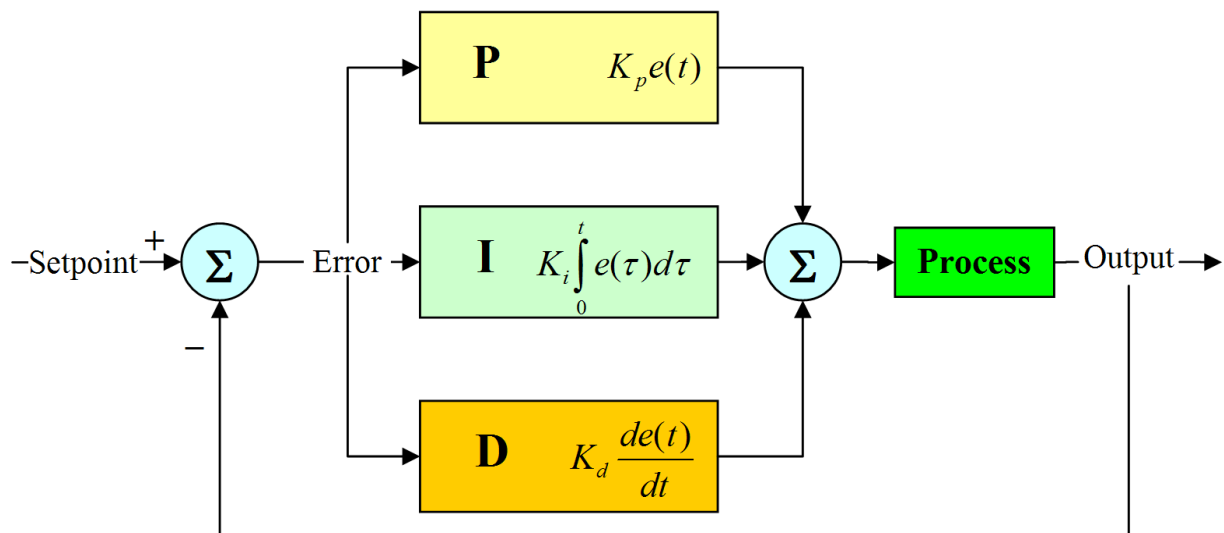
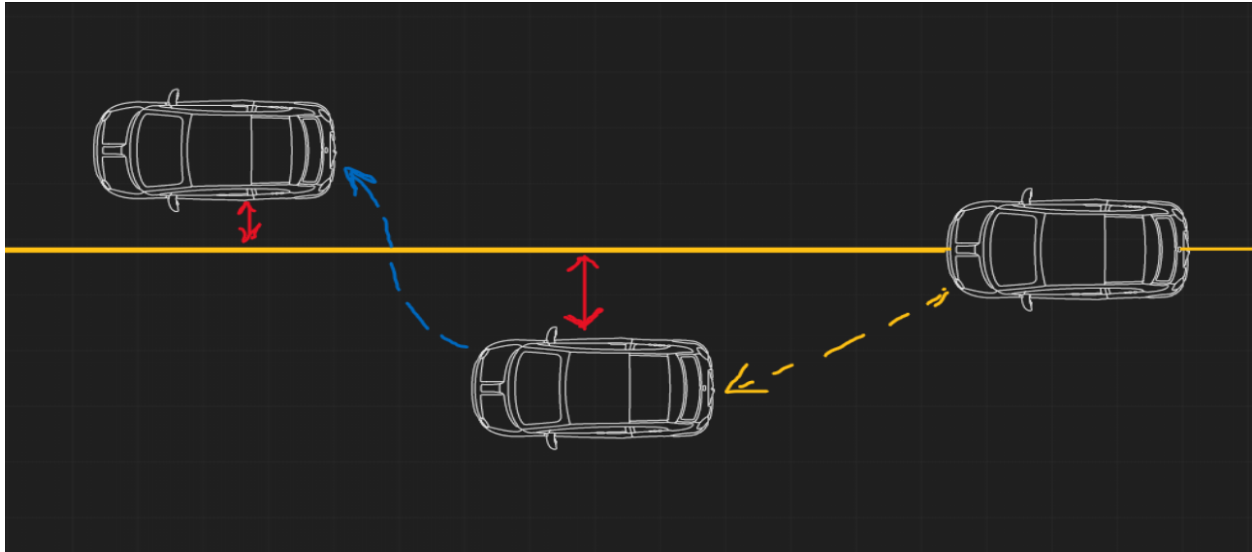
Command:- Task given to controller . In our case it is to align with line/balance itself.

Drive :- Response produced by the controller in order to complete the task . In our case this is the bot moving back/forth to self balance or left/right to follow line.

In closed Loop control systems such as in PID , the plant behaviour is measured and fed to the controller . That is the controller uses both feedback from the plant and the command to generate the drive . Just like in Walle bot we use previous error and current error. (feedback from bot)

2.2.2 PID CONTROLLER :

A PID controller continuously calculates an *error value* $e(t)$ as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms (denoted P , I , and D respectively), hence the name.



A **proportional controller** is just the error signal multiplied by a constant and fed out to the drive.

$$P_term = K_p * e$$

Where K_p = Proportional gain,
 e = error

The **integrator** state "remembers" all that has gone on before, which is what allows the controller to cancel out any long term errors in the output. Due to the limitation of P_term where there always exists an offset between the process variable and setpoint, I_term is needed, which provides necessary action to eliminate the steady-state error.

$$I_term = K_i \int e(t) dt$$

Where **K_i** = Integral gain,
 $\int e(t) dt$ = Cumulative error

Derivative Control predicts the plant behavior using the change in error . Given by $D = (de/dt) * K_d$

$$D_term = K_d (de/dt)$$

Where **K_d** = Derivative gain,
 de/dt = Change in error w.r.t time

Final Error Term :

$$u(t) = K_p e(t) + K_i \int e(t) dt + K_d \frac{de}{dt}$$

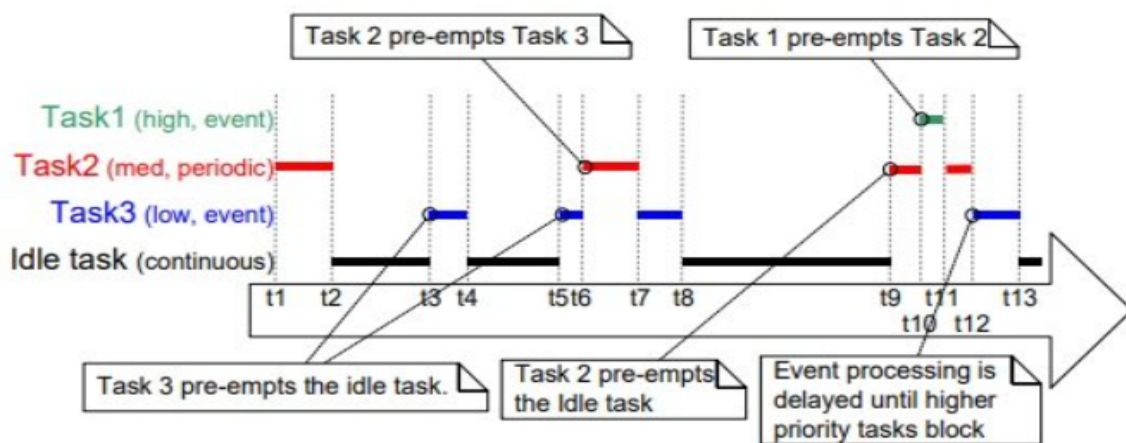
2.2.3 FreeRTOS

2.2.3.1 What is FreeRTOS?

- FreeRTOS is Open Source (MIT + Commercial license) RTOS kernel for embedded devices. It has been ported over to >35 uC platforms.
- Good Documentation , Small Memory Footprint , Modular Libraries and support for 40+ Architectures , etc. are some more features of FreeRTOS.
- Since 2017 Amazon has contributed in many updates to the original kernel apart from AWS freeRTOS
- FreeRTOS kernel supports two types of scheduling policy:

1. Time Slicing Scheduling Policy : This is also known as a round-robin algorithm. In this algorithm, all equal priority tasks get CPU in equal portions of CPU time.
2. Fixed Priority Preemptive Scheduling : This scheduling algorithm selects tasks according to their priority. In other words, a high priority task always gets the CPU before a low priority task. A low priority task gets to execute only when there is no high priority task in the ready state.

FreeRTOS uses the combination of above two Scheduling Policy , so the scheduling policy is FreeRTOS Prioritized Preemptive Scheduling with Time Slicing.



2.2.3.2 FreeRTOS Concepts:

1. **Semaphore** : Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
2. **Binary Semaphore** : A semaphore which can take either 1 or 0 . Also called as mutex lock . Differs from the mutex in the fact that it can be signaled by any thread unlike the mutex which can be released only by the thread that acquired it .
3. **Task Notifications** : Each RTOS task has an array of task notifications. Each task notification has a notification state that can be either 'pending' or 'not pending', and a 32-bit notification value.
4. **Queues** : A queue is a simple FIFO system with atomic reads and writes . Queue may be used to start another task to start doing its work while the primary task continues doing its own work independently.
5. **Mutex** : A mutex is similar to binary semaphore but it differs in the fact that a mutex will only allow ONE task to get past xSemaphoreTake() operation and other tasks will be put to sleep if they reach this function at the same time.

2.2.3.3 FreeRTOS Functions Used

- **void vTaskDelay(const TickType_t xTicksToDelay);**

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

- **SemaphoreHandle_t xSemaphoreCreateBinary(void);**

Creates a binary semaphore, and returns a handle by which the semaphore can be referenced.

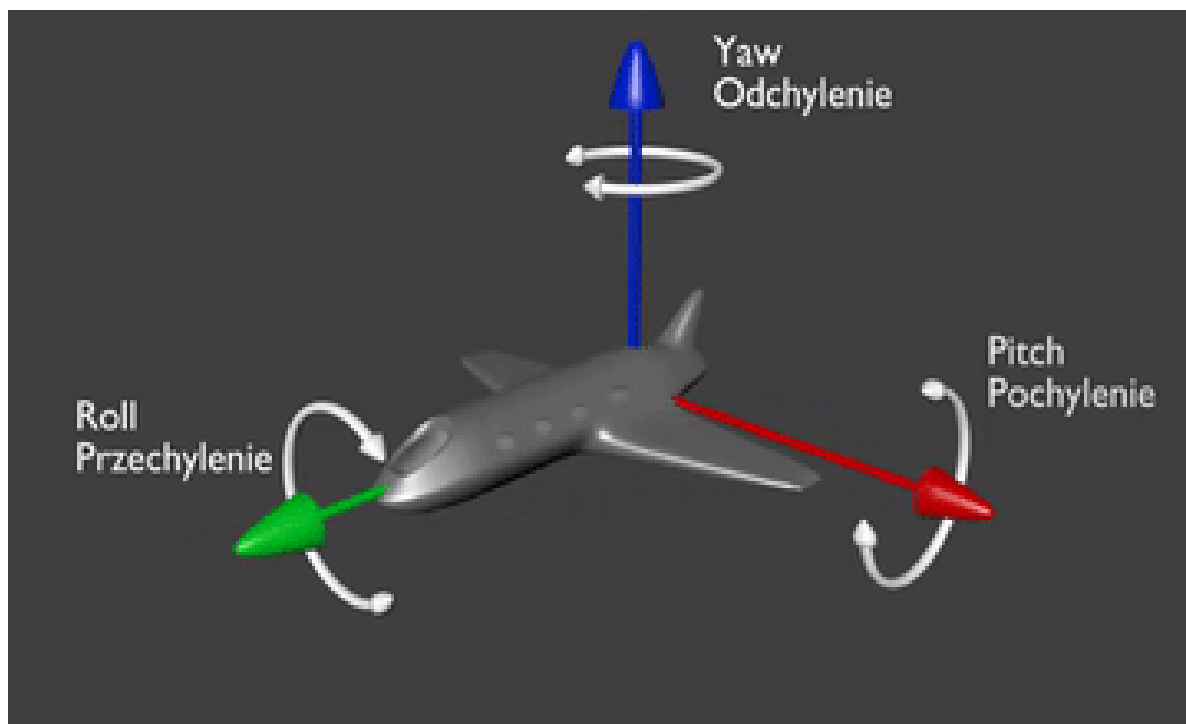
- **xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);**

Macro to obtain a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

- **xSemaphoreGive(SemaphoreHandle_t xSemaphore);**

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

2.2.4 ROLL PITCH AND YAW



Roll: Roll is the angle made by the bot with X-axis.

Pitch: Pitch is the angle made by the bot with Y-axis. We make use of this for balancing the bot .

Yaw : Yaw is the angle made by the bot with Z-axis.

2.2.5 Complementary Filter

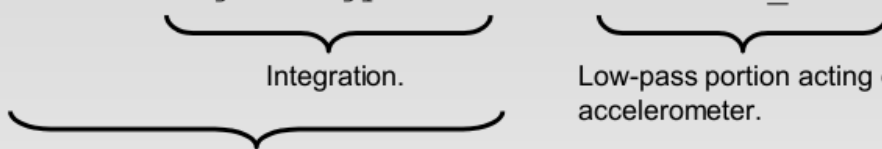
We get the readings of angle from both Gyroscope and Accelerometer. Both sensors can give readings directly.

- Gyroscope has a problem of Gyroscopic Drift which accumulates over time, so it gives reliable readings for a short period of time.
- Whereas Accelerometer is jittery and sensitive, but the readings are based on g-vector which has an absolute value, the readings can be relied on for long term.

To use the benefits of each Sensors we do Sensor fusion:

- Complementary filter takes slow moving signals from gyroscope and fast moving signals from an accelerometer and combines them. Accelerometer data is reliable for long term so we apply a "Low pass" filter to it whereas Gyroscope data is reliable in short term so we use "High pass" filter.

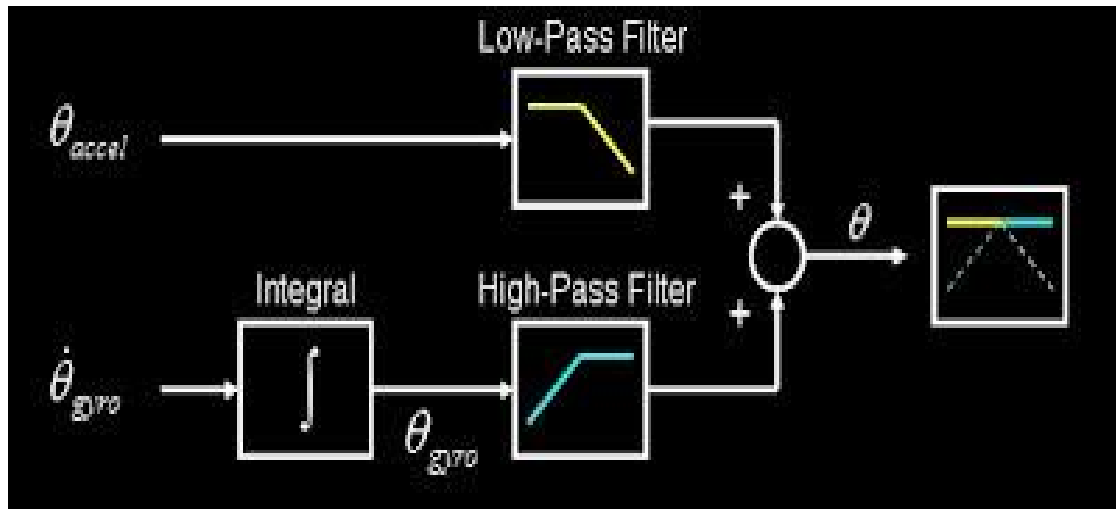
```
angle = (0.98) * (angle + gyro*dt) + (0.02) * (x_acc);
```



Integration.

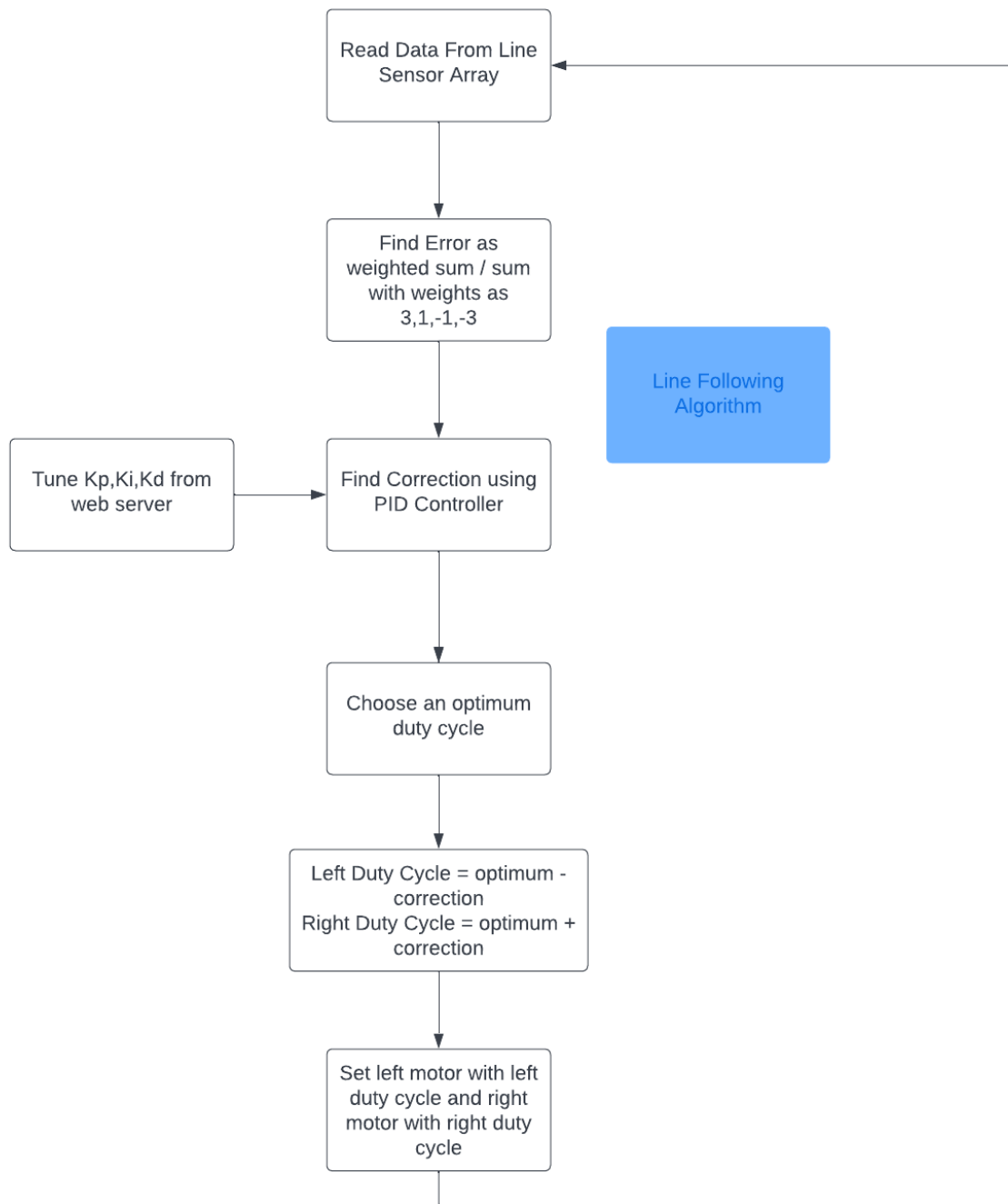
Low-pass portion acting on the accelerometer.

Something resembling a high-pass filter on the integrated gyro angle estimate. It will have approximately the same time constant as the low-pass filter.



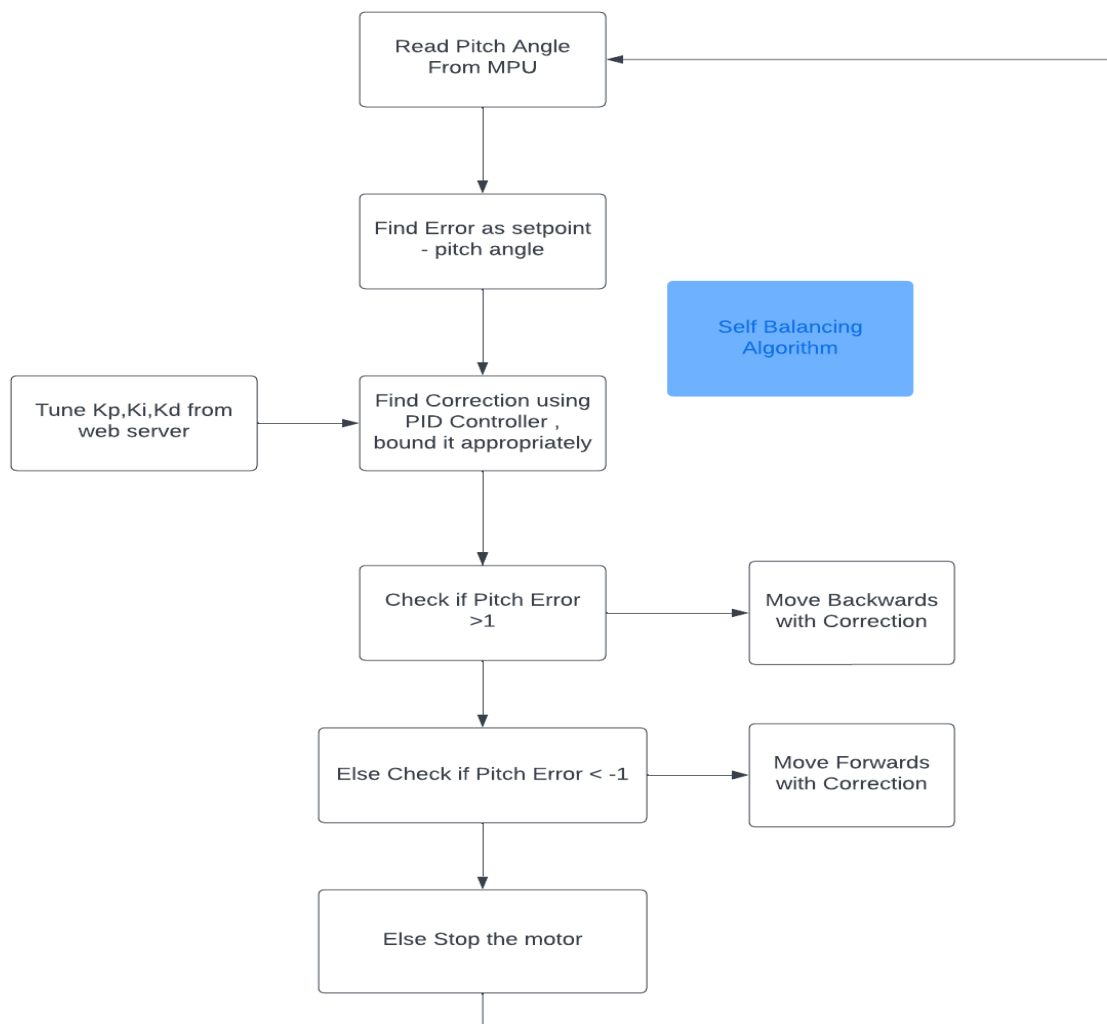
3. Implementation

3.1 Line Following Algorithm



The basic concept used in line-following algorithm is that the left motor and right motor speed should differ to take a turn . The amount by which they should turn is decided by PID which can be tuned .

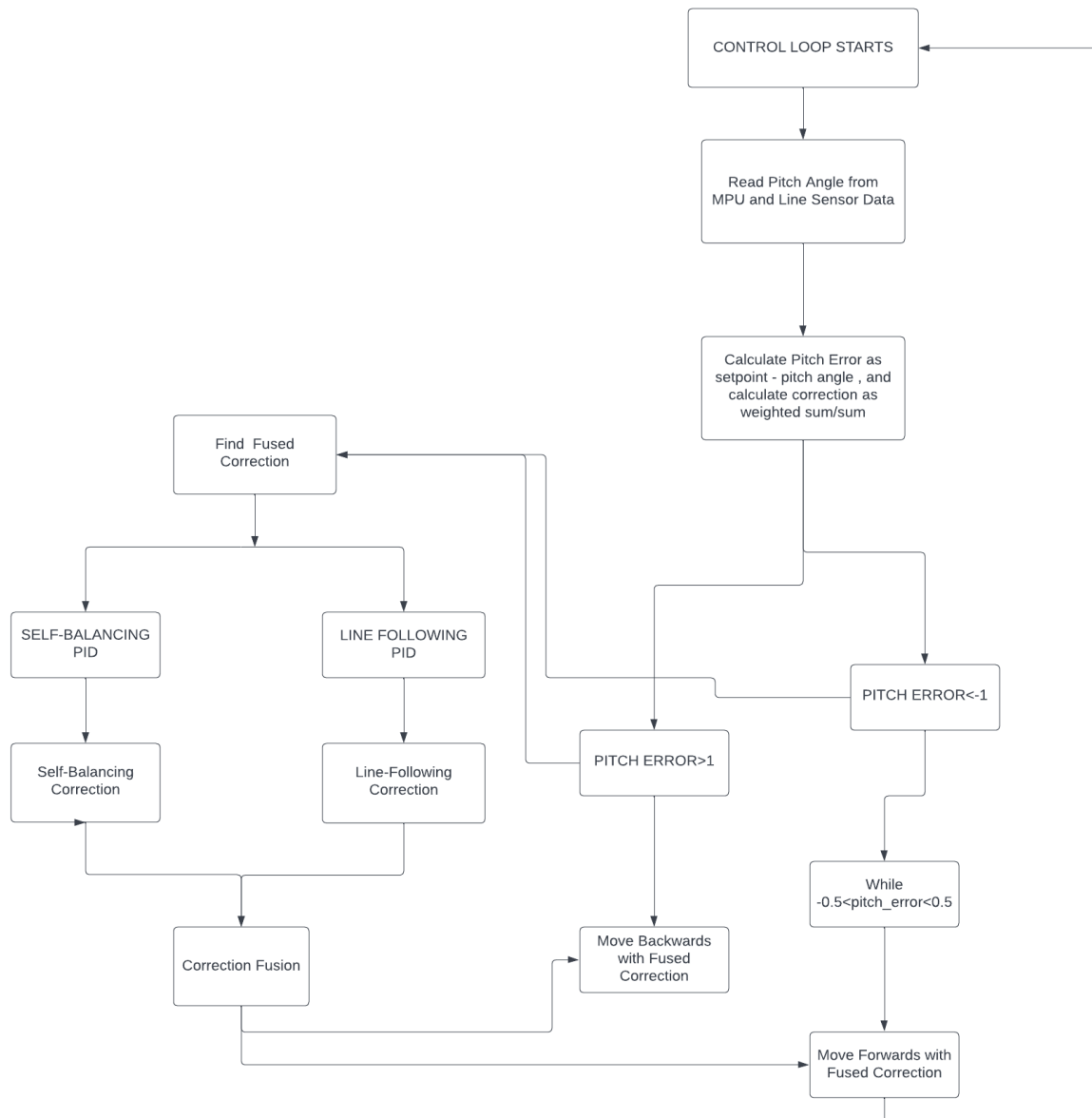
3.2 Self-Balancing Algorithm



The basic concept used in self-balancing algorithm is that when the chassis is below the setpoint the bot should move forward to lift it up

and when it is above the setpoint the bot should move backwards to lift it down using inertia . The speed by which the bot moves is decided by PID which can be tuned .

3.3 Self-Balancing and Line-Following Algorithm



```
if (pitch_error > 0)
```

```
{  
  
    set_motor_speed(MOTOR_A_0, MOTOR_BACKWARD, right_combined_cycle);  
  
    set_motor_speed(MOTOR_A_1, MOTOR_BACKWARD, left_combined_cycle);  
  
}  
  
else if (pitch_error < -1)  
{  
  
    set_motor_speed(MOTOR_A_0, MOTOR_FORWARD, left_combined_cycle);  
  
    set_motor_speed(MOTOR_A_1, MOTOR_FORWARD, right_combined_cycle);  
  
}
```

```
left_duty_cycle=bound(forward_pwm-correction,lower_duty_cycle,higher_duty_cycle);  
  
right_duty_cycle=bound(forward_pwm+correction,lower_duty_cycle,higher_duty_cycle);  
  
set_motor_speed(MOTOR_A_0, MOTOR_FORWARD, left_duty_cycle);
```

```
set_motor_speed(MOTOR_A_1, MOTOR_FORWARD, right_duty_cycle);
```

We use the concept of fused correction that is

Left duty cycle = balancing_correction - line_correction

Right duty cycle = balancing_correction + line_correction

We promote bot to move forwards when pitch error is less by making it move forward for condition when $-1 < \text{pitch error} < 0$

4. CHALLENGES FACED AND THE SOLUTIONS APPLIED :

4.1 CHALLENGES AND SOLUTIONS :

1. CHALLENGE : It was hard to think of a logic to merge self balancing and line following . The resources online were scarce for this .

SOLUTION : Try to understand from whatever is available, build your own logical deductions of what could work and keep on trying yourself.

2. CHALLENGE : The MPU would not give us perfect readings at times .

SOLUTION : We configured the mpu before using it by adding offset to its values which fixed the issue .

3. CHALLENGE : BO motors would not adhere tightly to the chasis even after tightening screws .

SOLUTION : We figured that instead of tightening the screws every time , we could use cello tape to fix them to the chassis.

4. CHALLENGE : Our bot's chassis wouldn't lift up even after providing a huge value of forward velocity.

SOLUTION : Eventually we realized that this might be due the unsymmetrical shape of the chassis at the front and also the chassis being too heavy. Therefore we switched to a new chassis design and tried our code on it . After few days we realized the problem was in the setpoint . Keeping the setpoint a bit high prevents this .

5. CHALLENGE : It was very difficult to tune 6 variables for line following and self balancing . Changing one would often mean that we need to change the other variable too .

SOLUTION : We had to keep on trying to tune our bot for quite a few weeks before it gave decent results. The key is to first initialize k_p, k_d, k_i for line following to 0 and tune only for self balancing. After this we slowly assign k_p, k_i, k_d for line following .

6. CHALLENGE : The combining of the self-balancing and line-following was a bit tough . We were trying to test an algorithm which would do both line following and self balancing .

SOLUTION : We realized that its better to first implement an algorithm on bot which would make it balance and move forward. After this adding line following to the code was relatively simple.

5. APPLICATIONS :

5.1 INDUSTRIAL USE OF LINE FOLLOWING BOTS :

The robots in industries perform a lot of tasks, including the material handling. To automate the process of material handling, the robot has to be guided properly and using a dedicated vision system or coding the coordinates is not feasible. The simplest solution, paint colour strips on the floor for the robot to follow, this deals with the transportation of materials. The material slots can also be colour coded to allow the robot to distinguish. This is the major application of the line following robots.

5.2 USED IN RESTAURANTS :

A line following bot can be used to deliver products from point of production to point of utilisation. It can be used in a restaurant for transporting food items made by a chef to the customers' table.

5.3 ADVANTAGES OF SELF-BALANCING ROBOTS :

A two wheeled self balancing bot can cut quite a lot of the cost in designing the bot. You would require only two wheels, two sensors and a couple of the bot's body parts and you are ready to go. Also, a two wheeled bot has much more accuracy in terms of velocity and during turns than a three or four wheeled robot. We can have more control over a two wheeled robot as compared to other robots.

6 . CONCLUSION AND FUTURE WORK :

6.1 CONCLUSION AND THINGS LEARNT :

To conclude, we would like to say that over the course of the past few weeks working on this project, we have learnt a lot of concepts, lessons and skills not only related to the project but also related to engineering and life in general. We came everyday carrying bags of hardware material and sat on the ground for hours testing and tuning our bot . This gave us the skills of hardwork and patience. Also thanks to the support and guidance of our mentors and seniors, we were able to make progress in this project. We also learned how to research about new topics, how to debug efficiently, how to write algorithms, how to work professionally on a project and much more.

What did we learn from the project?

1. We better understood how to work with embedded Systems .
2. We understood how to maintain hardware properly .
3. We learnt the extremely useful skill of soldering and connecting wires .
4. We got accustomed with programming in C and came across new concepts like macros and ifdef statements .
5. We also learned about algorithms of self-balancing and line-following. We understood how a bot actually balances itself using readings from sensors like imu and how it moves on a line.
6. We understood what PID is, studied about PID controllers, how to implement PID, the uses and importance of PID, etc.
7. In general we got more familiar with control systems

8. We got more familiar with electronic devices like multimeter , 3d printer , wire stripper , solder station etc which are used in a lot of projects .

9. We got accustomed with Real Time Operating Systems and specifically FreeRTOS which would be of great use in future projects in the field of embedded systems .

6.2 FUTURE ASPECTS OF THE PROJECT :

Our self balancing and line following algorithm is still not perfect . The bot still touches the ground at occasions which is not ideal for a self balancing bot .We would like to remove this shortcoming as soon as possible .

The following developments are yet to be achieved which we would like to try in the future :

- Fine Tuning the PID for better results
- Improving the Self-Balancing and Line-Following Algorithm
- Adding Power Source to make bot work wirelessly
- Improving Design of Robot for better self balancing

7. REFERENCES :

7.1 Useful links and Research Papers :

- Our github repository : <https://github.com/Raghav323/Alfetta>
- Official Walle v2.2 repository of SRA-VJTI :
https://github.com/SRA-VJTI/Wall-E_v2.2

- For Learning about PID :
<https://www.wescottdesign.com/articles/pid/pidWithoutAPhd.pdf>
- Other Implementation of our Project which helped us :
https://www.researchgate.net/publication/304294248_Control_or_balancing_line_follower_robot_using_discrete_cascaded_PID_algorithm_on_ADROIT_V1_education_robot
- For Learning about FreeRTOS :
<https://sravjti.in/embedded-systems-study-group/week6/week6.html>
- ESP-IDF Documentation :
<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/>
- FreeRTOS Documentation:
<https://www.freertos.org/features.html>