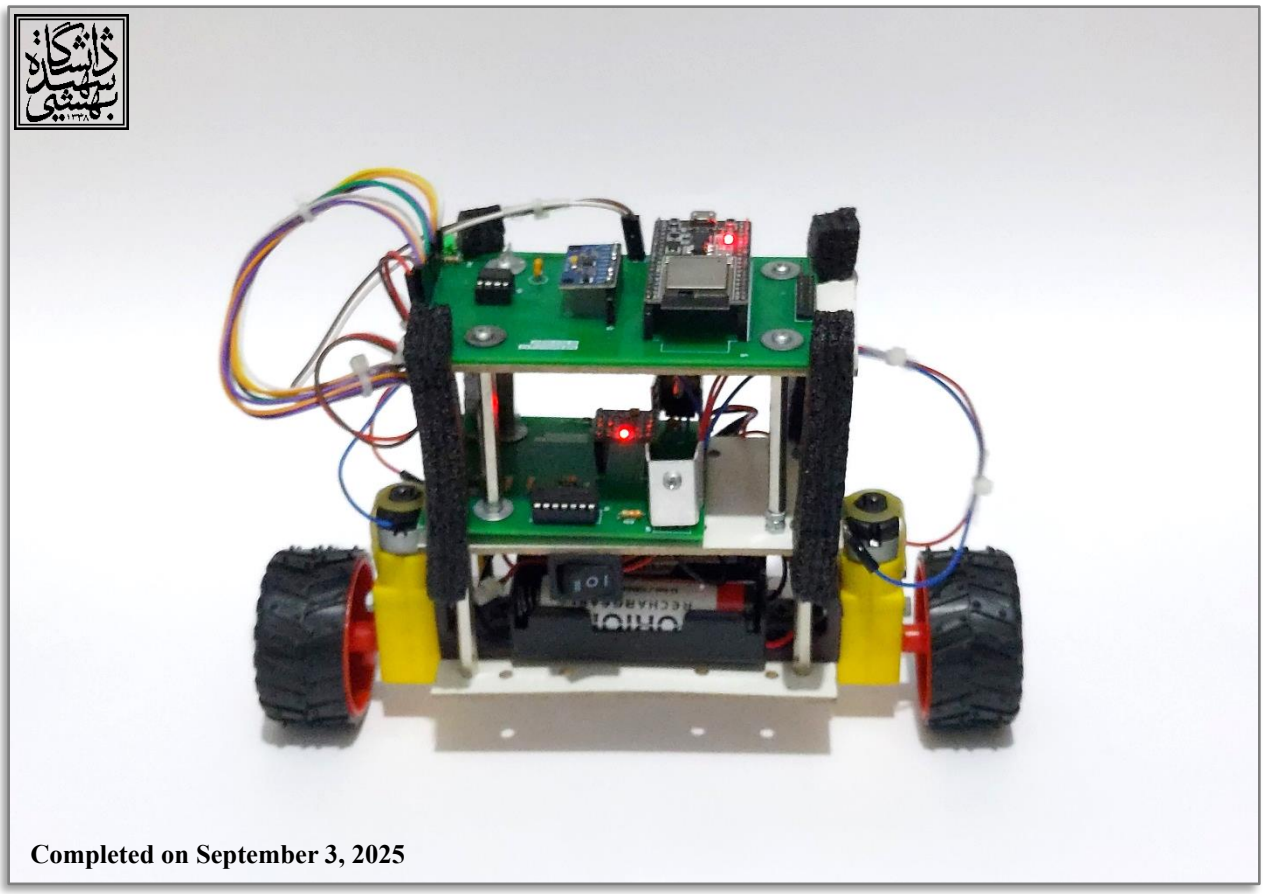


Completed on September 3, 2025

2 Wheel Self Balancing Robot

Designed and Implemented by

Mohammad Sina Majidi

Under the Supervision of

Dr. Mostafa Abedi

Faculty of Electrical Engineering

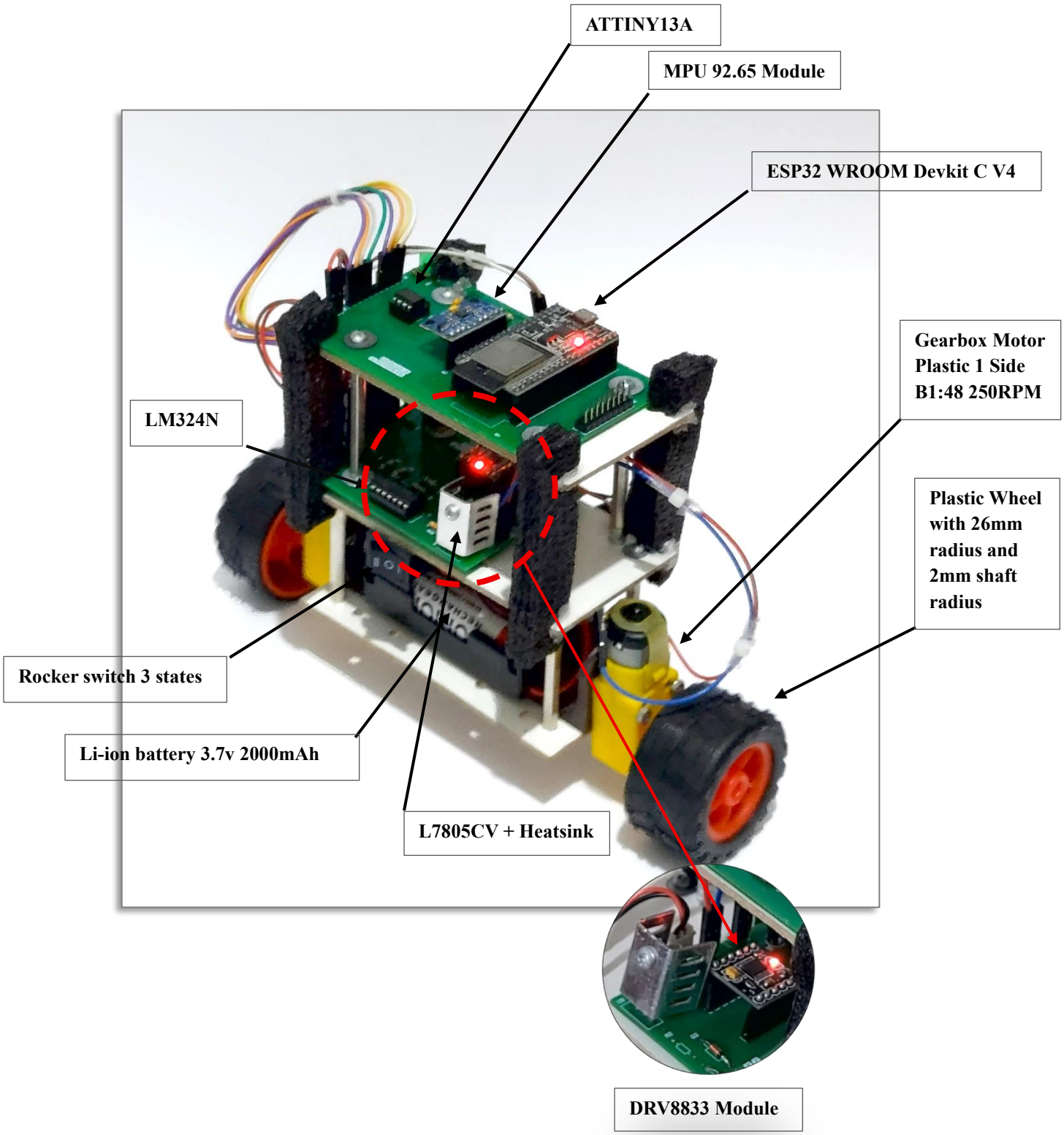
Shahid Beheshti University

Introduction

The project, as indicated by its title, involves the complete design and implementation of a two-wheeled self-balancing robotic car. The primary aim is to implement a PID (Proportional-Integral-Derivative) algorithm to maintain the robot's balance. Additionally, this project will provide valuable experience in robotics and demonstrate how simple control systems can achieve physical stability for a robot.

The ultimate goal is for the robot to maintain its balance autonomously. Future enhancements being considered include remote control capabilities, the ability to move in various directions, and the installation of a camera module. These features have been taken into account during the PCB design phase for potential future developments.

Hardware Components



Description of Hardware Implementation:

- The system utilizes two Li-ion batteries, which will be recharged using the TP5100 charger module whenever their voltage drops below a minimum threshold. The LM324N is responsible for sensing the battery voltage and shifting the level down to 3.3V, allowing the ESP32's ADC unit to read it safely.

- An ATtiny13A serves as a secondary microcontroller (MCU) to manage critical system functions while the ESP32 is updating firmware over-the-air (OTA). This MCU will also trigger the reset pin once the firmware replacement is complete, allowing the ESP32 to restart.

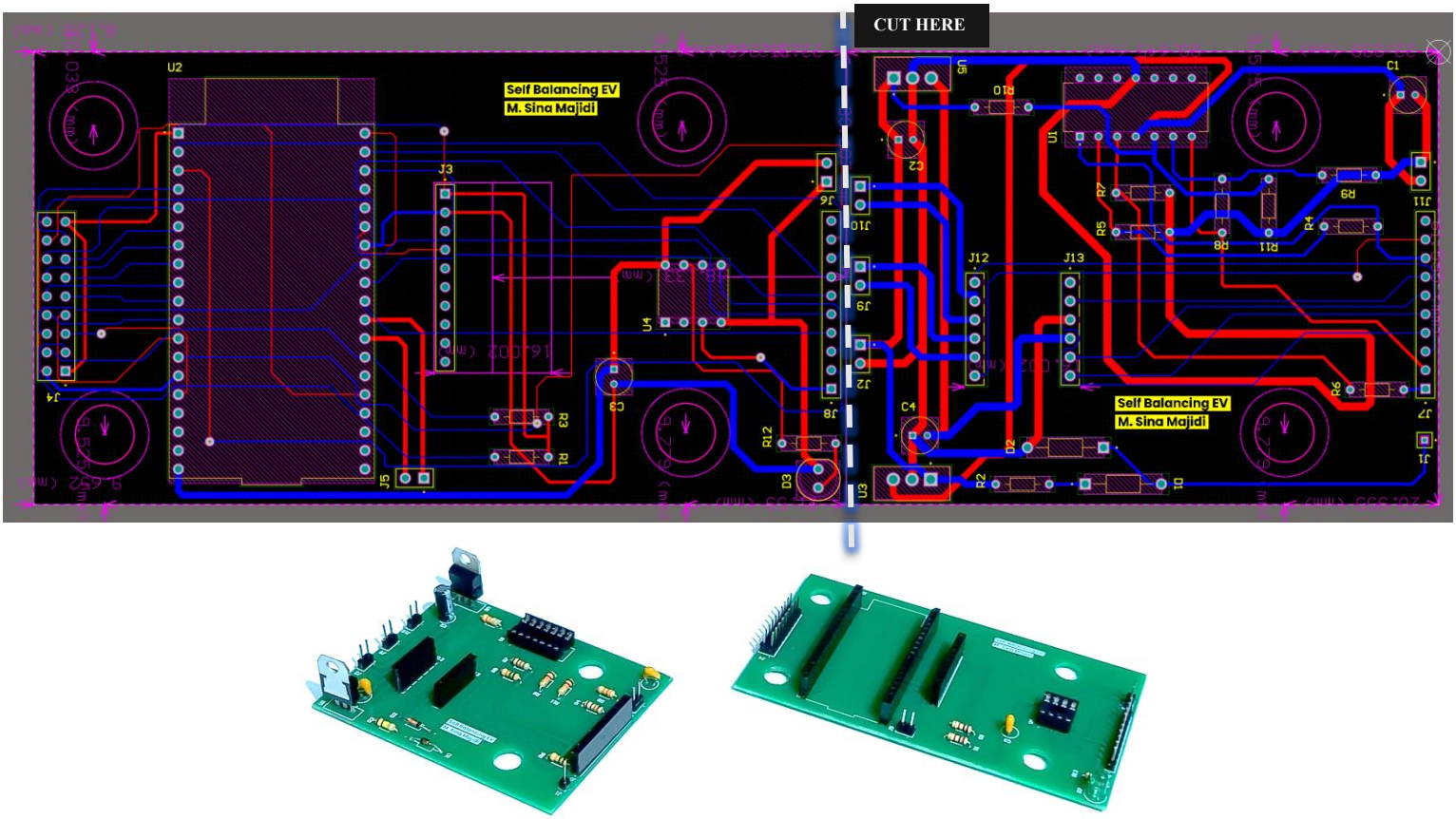
- A green LED indicates specific events, such as each time the ESP32 restarts or successfully completes a firmware update via OTA.

- Two separate 5V regulators, housed in TO-220 packages and equipped with heatsinks for better heat dissipation, are employed. One regulator powers the DC motors, while the other supplies power to integrated circuits (ICs) and other components.

- M3 metal washers, screws, nuts, and M3 metal spacers (50mm) are utilized to secure various parts to the MDF board and the floors.

- Additionally, a servo motor is considered in the PCB design to support the back of the robot while the ESP32 is updating to prevent it from falling. This servo motor is also a part of future development plans.

PCB Overview



Description

The PCB is designed using Altium Designer software and consists of two distinct parts, separated by a line indicated in the image (with "CUT HERE" written above). This line will be lasered during the PCB manufacturing process.

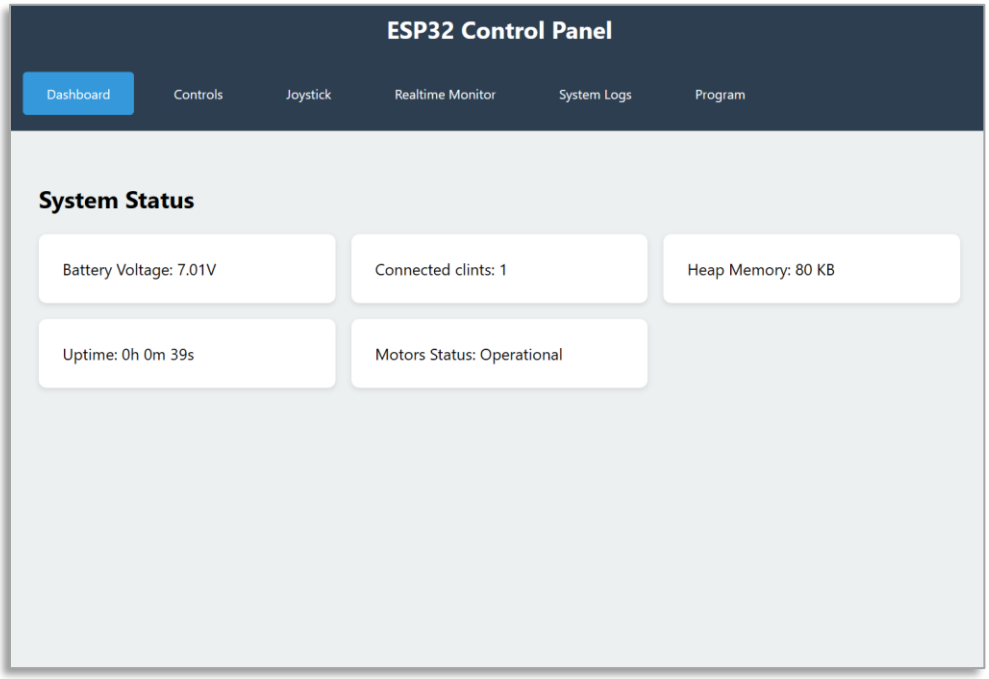
PCB Specifications: It is a 2-layer PCB where the J4 connector is intended for camera mounting and is designed for live streaming with the OV7670 module and a VGA module. As mentioned, the design also allows for future developments.

Mounting holes are provided for securely fastening the two parts of the PCB, which are on different layers, using screws, spacers, nuts, and washers.

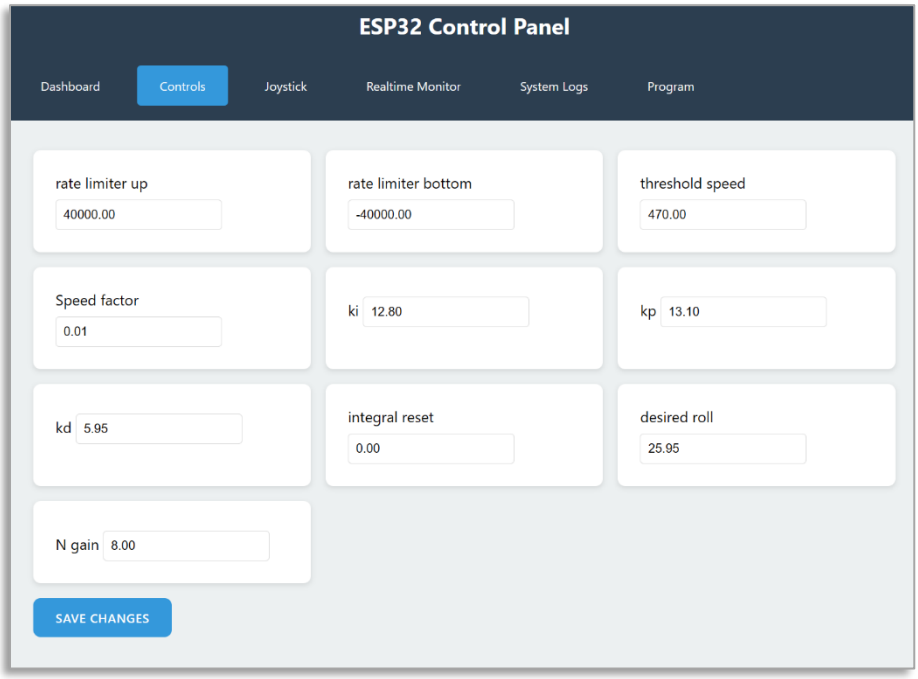
- For this step, I used MATLAB to design the control system. After generating the control system C code using the Embedded Coder in the Simulink app, I integrated it into a C++ program.
- The complete C++ program, which includes the generated code from Simulink, utilized 14.3% of the RAM and 72.8% of the flash memory.
- Since the ESP32 needs to run multiple tasks (five tasks) and host several web pages asynchronously (approximately nine web pages on different paths) over Wi-Fi, I employed the ESPAsyncWebServer and FreeRTOS libraries.

Regarding the online control panel:

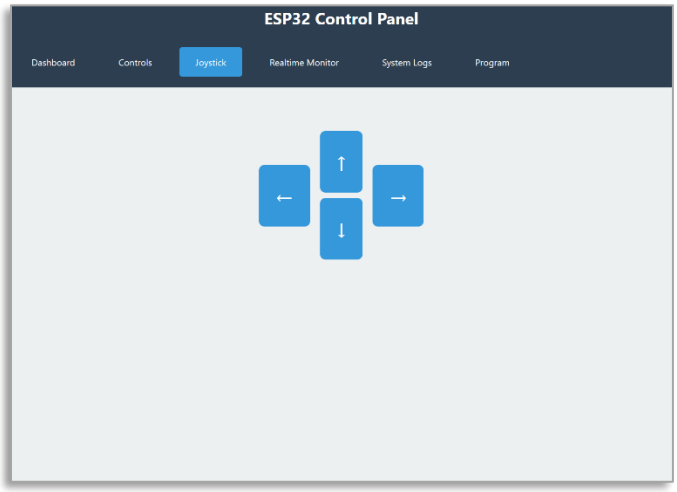
By entering the IP address 192.168.123.100 in the browser, the admin can access the "dashboard" tab. This dashboard updates every second and provides the admin with an overview of the system's general status.



In the “Controls” tab, you can access and carefully change important parameters. These parameters will be explained in the “Control system design” section of this report.



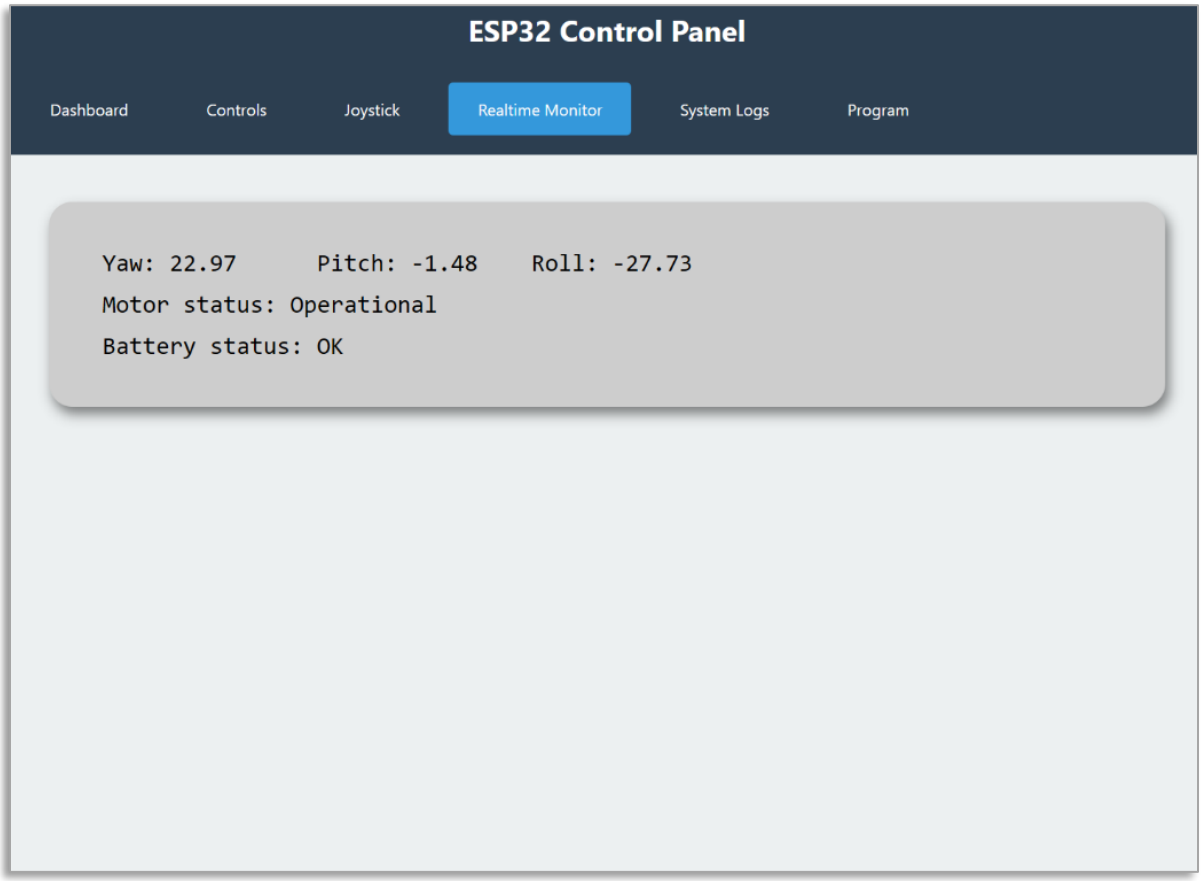
The joystick tab features four buttons, each corresponding to a different direction for moving the robot. At present, only the forward and backward buttons are operational, but support for the other directions will be included in future updates. The code required to enable live camera streaming has been written and compiled as a standalone application. Once the montage and wiring are completed, the live stream will be added above these buttons.



The Realtime Monitor can serve two distinct purposes:

1. **Data Sampling**: As explained later in this report, you can utilize web drivers like Selenium to sample data, such as Roll, with a specified sampling time. This data can then be used to perform system identification.
2. **Real-Time Data Monitoring**: The monitor allows you to view important parameters in roughly real-time, with a sampling and refresh frequency of approximately 200 Hz.

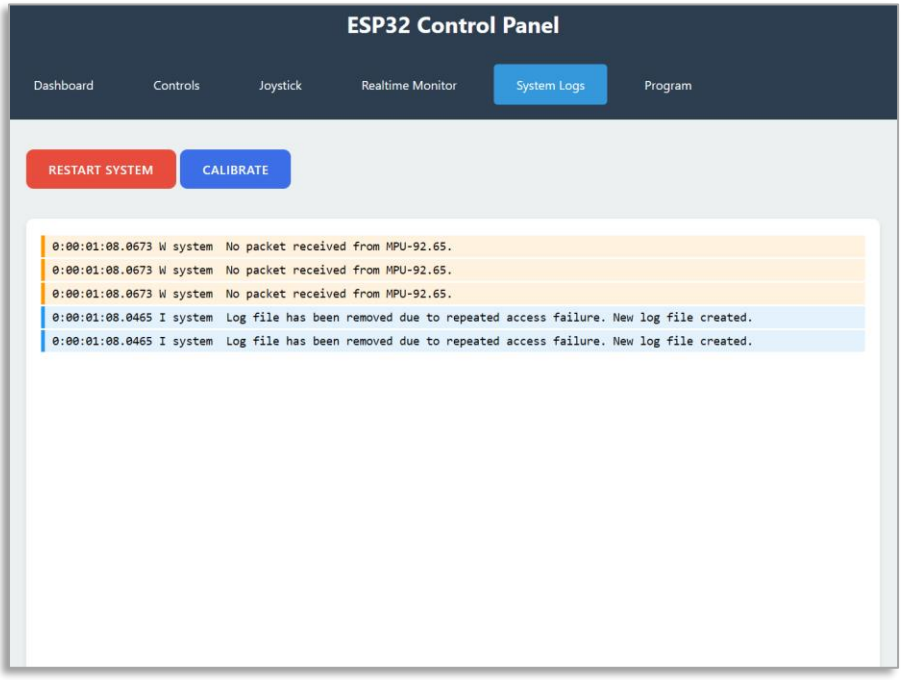
In the near future, there will be a feature that enables real-time plotting of data within the same tab, utilizing the client browser. This will allow users to visualize data over time by employing either an internal or external JavaScript framework.



The System Logs tab records important system events. These logs are non-volatile, meaning that each time a new event occurs, it is written to the internal SPI Flash using SPIFFS and the Logging library. To maintain a constant total log size, the oldest entries are removed when the number of logs approaches the maximum size limit.

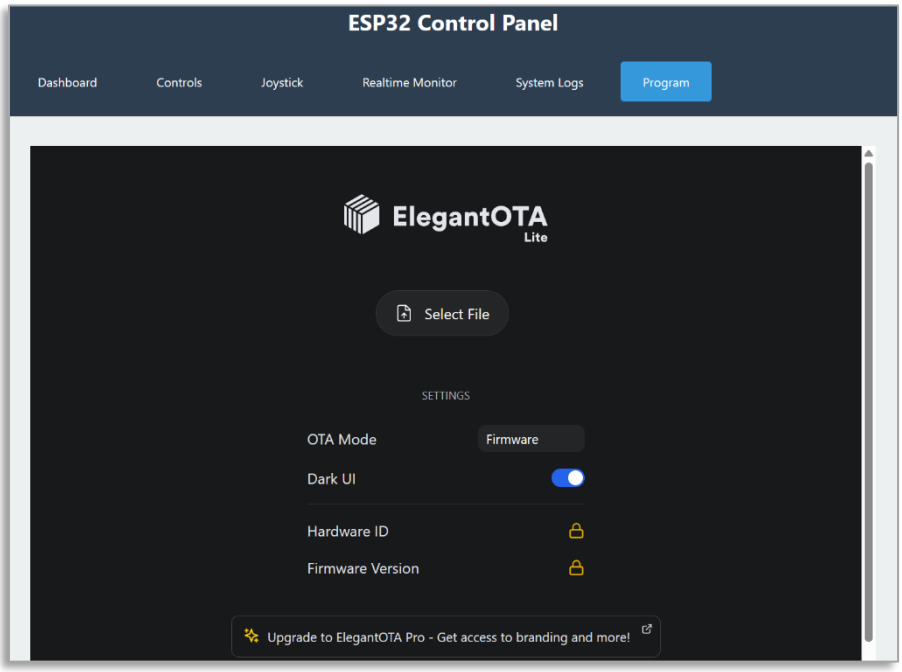
If a problem occurs or if the chip is erased and the program is re-uploaded, accessing the logs may be problematic as they might be erased or not created yet. In this case, the ESP32 will create and overwrite the existing log file.

Additionally, there are two buttons at the top of the tab. The “Restart System” button restarts the ESP32 with the help of the ATTINY13A, while the “Calibrate” button recalibrates the MPU module and rewrites the configuration file. It is important to keep the robot in its zero roll position during the calibration process.



The final tab, labeled “Program,” allows you to upload a new firmware binary file generated by the compiler after compiling the program. It utilizes ElegantOTA to securely download the new firmware and replace the existing one.

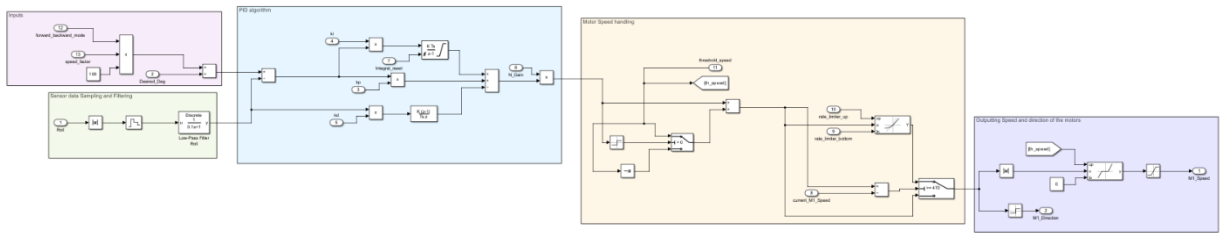
Note: Occasionally, the update process may be interrupted, either due to running motors or other specific issues, resulting in a halt during the update, such as at 80%. However, there is no safety concern, as the OTA framework will not replace the existing firmware until the entire new firmware binary file is received and its hash code has been verified for data integrity. If an interruption occurs, simply retry the upload.



- As you may have noticed, the entire C++ program will not be included in this report because it is still under development. Additionally, it would be cumbersome to include hundreds of lines of code in the main.cpp file, along with multiple custom header files, in this technical report.

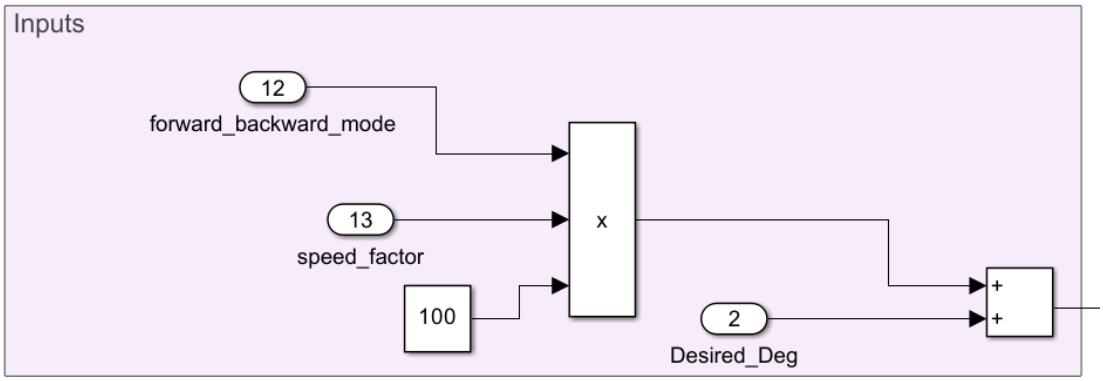
Control System Design

Below is an overview of the different parts of the control system designed in Simulink, and I will explain each part.



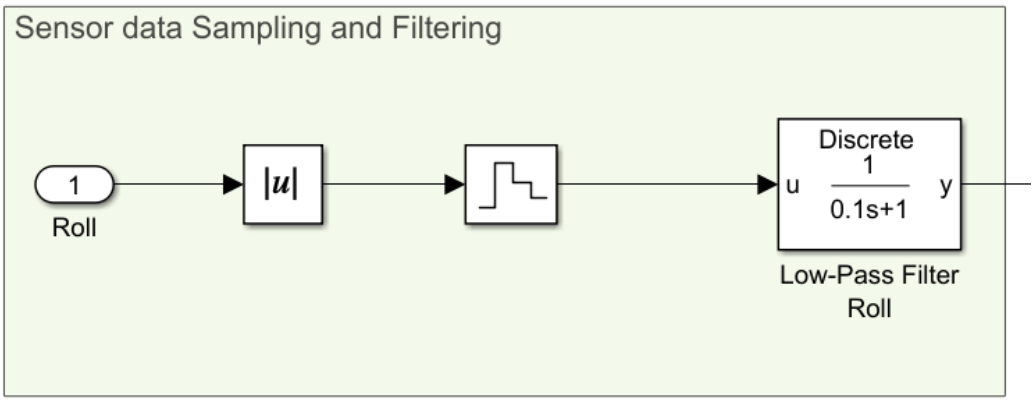
- Inputs:

  The control system's set point is provided through the "Desired_Deg" parameter. By selecting ±1 for the "forward_backward_mode" using the program, which can be adjusted by clicking the arrow buttons, this mode will be established. Additionally, the "speed_factor" determines the movement speed, as indicated by its name, and can be configured in the "Controls" tab of the web control panel.



- Sensor Data Sampling and Filtering

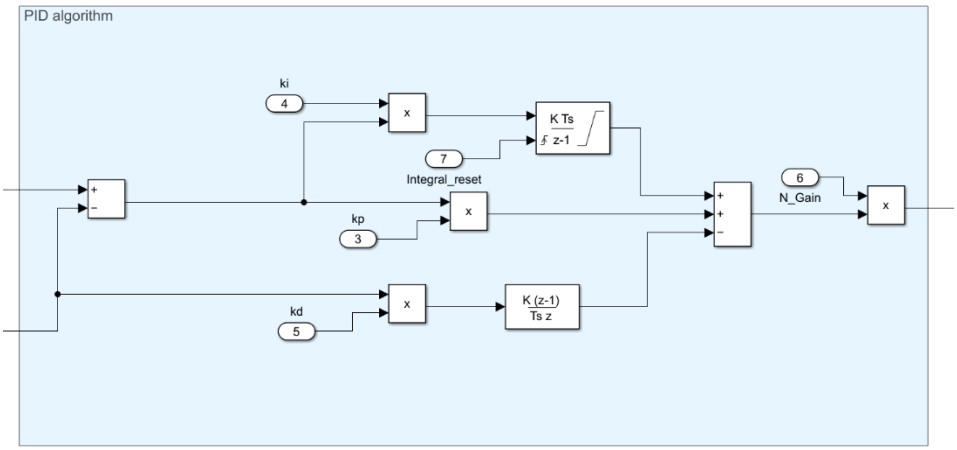
  In this section, we focus on processing the sensor roll data. To obtain the unsigned value of the roll, we take the absolute value. This approach aids in filtering and results in a smoother output for the controller. After sampling the data using Zero-Order Hold (ZOH), we apply a low-pass filter with a time constant of T = 0.1 seconds. The resulting signal is then used as feedback for our system.



- PID Algorithm:

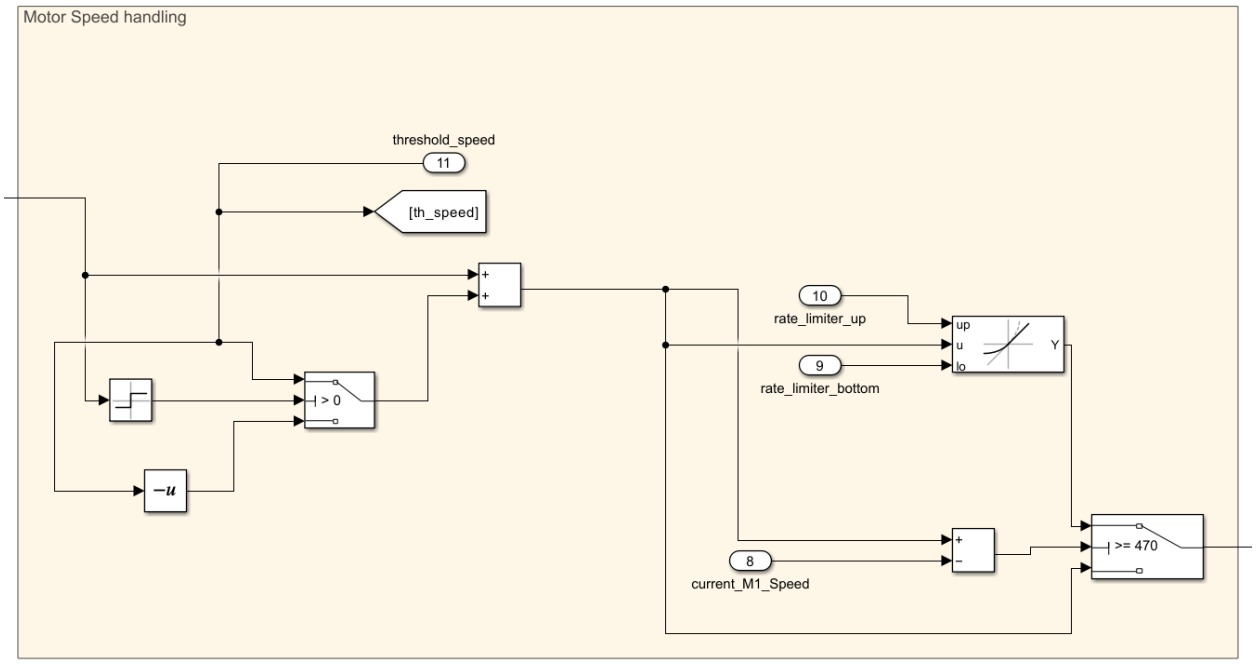
  Although we could easily implement a digital PID block, I opted to design the controller using simpler blocks to allow for greater design flexibility and modification. The integrator includes a limiter to prevent unbounded output, and the feedback signal from the deviator is used to avoid an impulsive output to the adder when a step input is applied.

  I chose a positional PID configuration after testing both incremental and positional methods. I found that the positional PID offers some advantages, even though the incremental PID effectively addresses the integrator saturation problem.



- Motor Speed Handling:

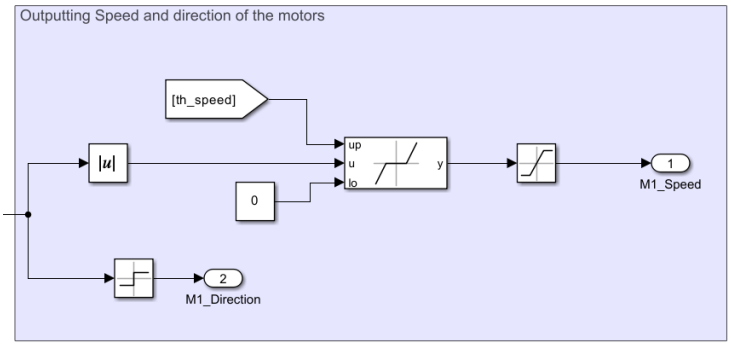
  This section addresses the adjustment of motor speed based on the controller output. Here's what it adjusts:
  Feeding the PID controller output directly to the motor can lead to several issues. The output units are expressed in bits for PWM, which operates with 10-bit resolution at a frequency of 20 kHz.
  Through extensive experimentation, it was found that applying a value greater than 470 at once (either as a differential change from the last signal or as an absolute change) to the DC motors can cause one or both motors to stall, resulting in a stall current draw. This issue arises from the gearbox and the friction between the gears, which necessitates a smooth start to avoid stalling. This behavior holds true for both motors and applies in both forward and backward directions.
  To mitigate this, a simple control logic has been designed so that the PWM signal to the motors increases or decreases gradually. If the change is within the safe range, the signal can be fed directly to the motors without causing them to stall.



- Outputting Speed and Direction of the Motors:

  Based on experimentation, separating absolute speed and direction using a signed number will be beneficial in the main C++ program. This process involves two lines of output signals. The first line is labeled as M1_Direction, which will be applied to both motors. The signed input will be processed through a sign function that will output either +1 or -1 accordingly.

  The second line of output consists of M1_Speed, also applicable to both motors. The signal will be unsigned and will utilize the absolute function. To handle the motor's dead zone, the Simulink Dead Zone Dynamic block will be employed before outputting the signal.

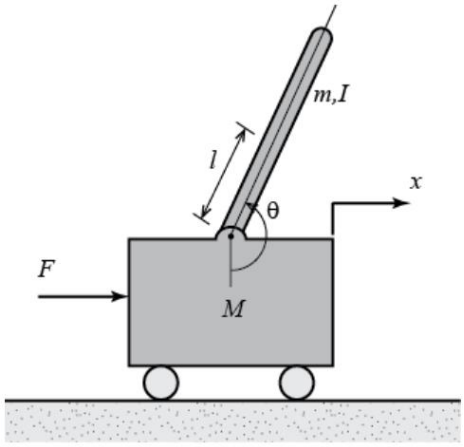


In this section of the report, I will explain the methods I used to derive the transfer function of the robot:

Modeling Based on Physical Principles (I):

I referenced a tutorial on system modeling for an inverted pendulum provided by the University of Michigan[1]. Using this tutorial, I attempted to model the robot and derive its transfer function.

For this analysis, I considered a simplified model of an inverted pendulum placed on a cart.



By having (or calculating) the following parameters we can calculate the transfer function we needed:

```
(M)       mass of the cart

(m)       mass of the pendulum

(b)       coefficient of friction for cart

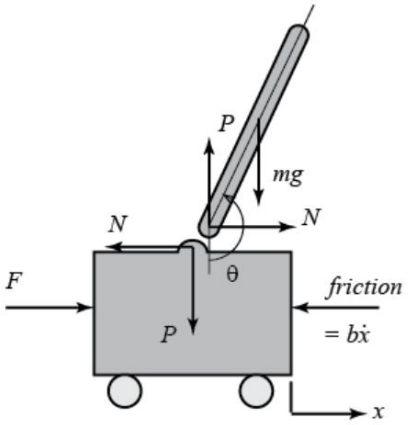
(l)       length to pendulum center of mass

(I)       mass moment of inertia of the pendulum

(F)       force applied to the cart

(x)       cart position coordinate

(theta)   pendulum angle from vertical (down)
```



$$P_{cart}(s) = \frac{X(s)}{U(s)} = \frac{\frac{(I+ml^2)s^2 - gml}{q}}{s^4 + \frac{b(I+ml^2)}{q}s^3 - \frac{(M+m)mgl}{q}s^2 - \frac{bmgl}{q}s} \quad [\frac{m}{N}]$$

For the parameters, I measured some and used default values since it was difficult to experiment and calculate, and the datasheet for the DC motors was not available:

M = 0.2442 Kg

m = 0.1392 Kg

b = 0.1 N/m/sec

L = 12 x 10^-2 m

I= 0.006kg.m^2

<---------->

T stall = 8 x 10^-3 kg-m

r = 3 x 10^-2 m

wnl = 26.1799 rad/sec

After calculating the other parameters from the main ones, the following transfer function has been calculated in MATLAB:

```
G =

                  -0.0001701 s
  ------------------------------------------------
  0.00279 s^3 + 0.0008004 s^2 - 0.06276 s - 0.01637
```

There was a problem when I implemented a closed-loop system and used a PID digital controller in the forward path. After tuning it in MATLAB with the PID Tuner app, I found that the estimated gains were excessively large, and in some cases (when selecting different phase margins and gain margins), they even had negative signs. After thoroughly reviewing each step multiple times, I determined that the modeled transfer function would only work for systems with greater height. Consequently, when the height of the inverted pendulum was set below a certain threshold, the equations produced an incorrect transfer function.

Modeling Based on Physical Principles (II):

After the previous approach failed, I searched through various articles and found one that was also based on physical modeling. After reading the document, I began to use it to derive the transfer function once again.

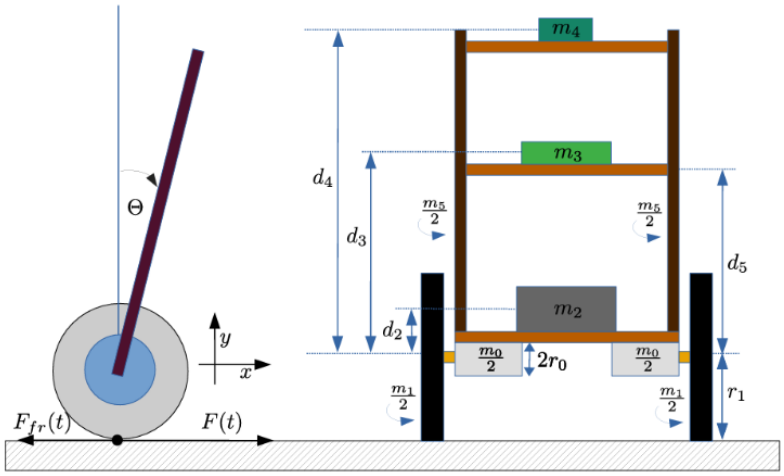


Figure 4. Model of two-wheeled and self-balancing robot.

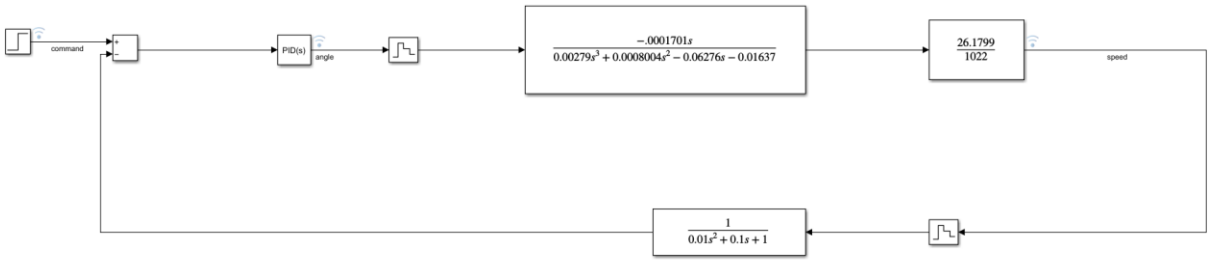
This article discusses a two-wheel self-balancing robot. By using simpler parameters, we can derive a transfer function like this:

$$G(s) = \frac{\Theta(s)}{\Omega_m(s)} = \frac{-s^2 K_3 I_1}{s^3 H_1 r_1 + s^2 H_2 r_1 + s H_3 r_1 - f_{fr} K r_1} \tag{16}$$

After following the steps, the resulting transfer function is as follows:

$$G(s) = \frac{81270180272377250000000\, s^2}{-16162653903355275718269 9\, s^3 + 144889769562601828810500 0\, s^2 + 7985714780667125760000 00\, s + 27491740292138649255936000 0}$$

After dividing the coefficients by a large number, the final transfer function improved significantly. Then, after configuring it similarly in Simulink, I attempted to tune the PID gains.



The results I obtained were once again unsatisfactory. I experimented with various gain margins and phase margins as tuning frequency parameter goals, but the outcomes consistently led to significant, and sometimes negative, gains, resulting in an unstable system in most cases. Therefore, I decided to change my approach and explore a different method for deriving the transfer function, which I will explain next.

System identification was performed using sampled roll data collected over 120 seconds at a sampling frequency of 200 Hz in MATLAB:

```
from selenium import webdriver
from subprocess import check_output
from time import sleep, perf_counter
from selenium.webdriver.common.by import By


def SSID_match(target_ssid: str) -> bool:
    wifi_interface = check_output("netsh wlan show interfaces").decode("utf-
8").splitlines()
    for line in wifi_interface:
        if "SSID" in line and "BSSID" not in line:
            current_ssid = line.split(":")[1].strip()
            return current_ssid.lower() == target_ssid.lower()

    return False


def selenium_driver(sampling_frequency=10,duration = 1, file_name = "./Data.txt"):
    global rt_list
    rt_list = []


    driver = webdriver.Chrome()

    driver.get("http://192.168.123.100/")
    sleep(5)
    driver.find_element(By.XPATH, "/html/body/div[2]/button[4]").click()
    sleep(5)

    sample_time = 1/sampling_frequency
    sample_count = int(duration/sample_time)

    for sample in range(sample_count):
        tik = perf_counter()
        retry = 1
        while retry==1:

            try:
                rt_data = str(driver.find_elements(By.CLASS_NAME, "log-
line")[0].text)
                retry = 0
            except:
                continue
```

```
        rt_data = float(rt_data.split("Roll:")[1].strip())
        rt_list.append(str(rt_data))
        while (perf_counter() - tik < sample_time):
            temp = 1

    driver.quit()

    with open(file_name, "w") as f:
        f.write(f"<----------- Sampled IMU Roll ----------->\n sampling frequency:
{sampling_frequency}Hz, sampling duration: {duration}s, sample time: {sample_time}s,
sample count: {sample_count}\n writing sample data as MATLAB list:\n")
        rt_list_matlab = ",".join(rt_list)
        f.write(f"sample_data = [{rt_list_matlab}];")
        print("file is written.")

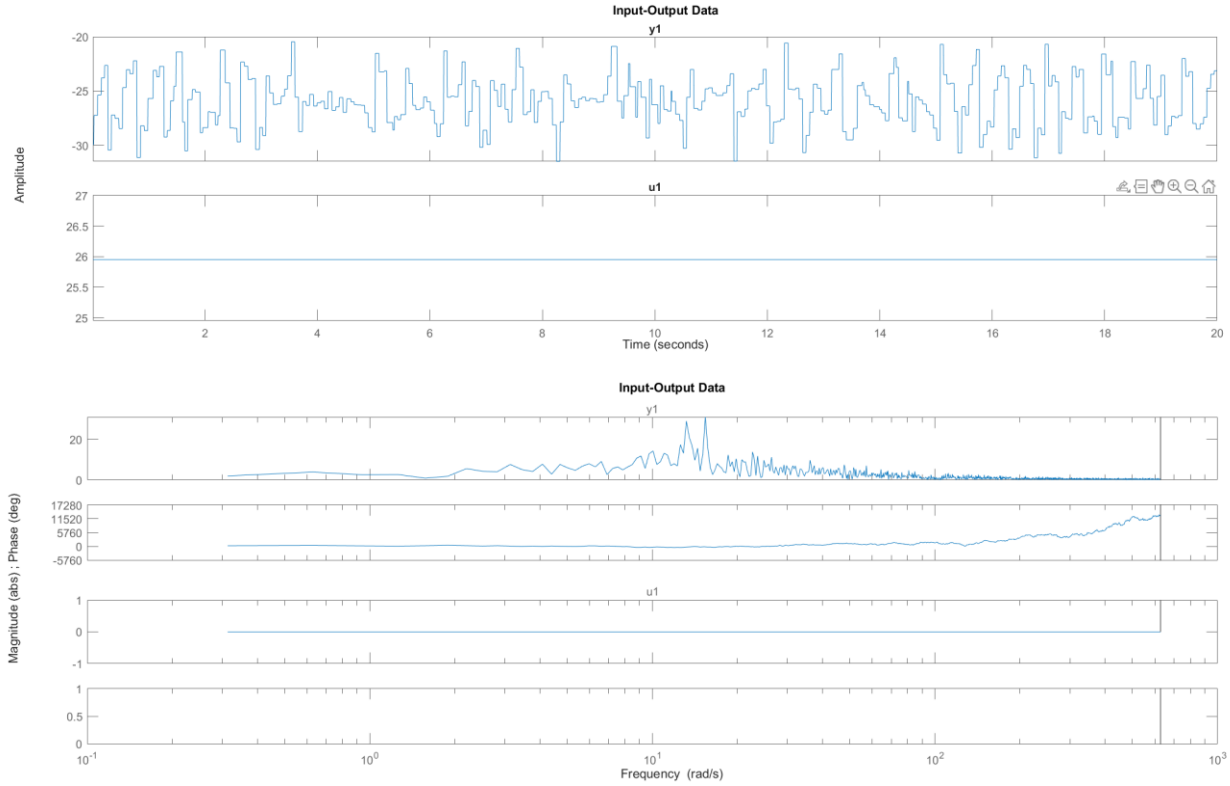


def main():
    Robot_wifi_ssid = "SelfBalancingEV"
    if not SSID_match(Robot_wifi_ssid):
        print(f"Not connected to {Robot_wifi_ssid}. Please connect and try again.")
        return False

    selenium_driver(sampling_frequency=200,duration = 120, file_name =
"./Data_config.txt")


if __name__ == "__main__":
    if not main():
        RuntimeError("Failed to complete.")
        exit(1)
```

I've plotted the first 20 seconds of the sampled data and its FFT in MATLAB. Here are the plots:



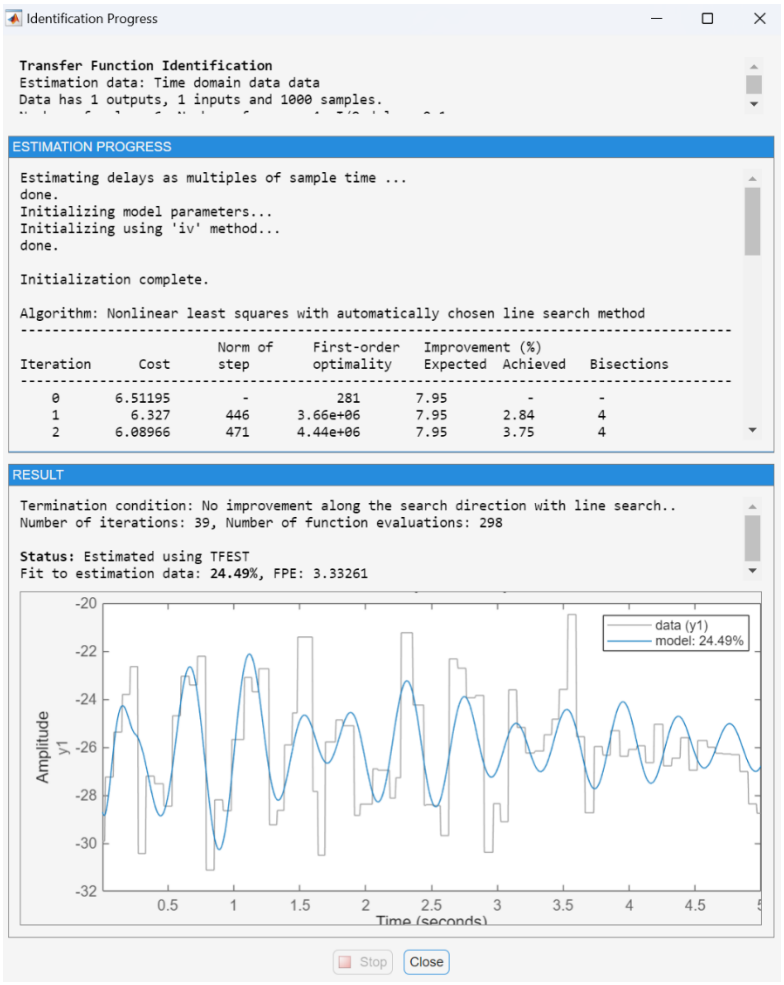
Using the code below, I created IDDATA for both the sampled data and its FFT:

```
clc; clear; close all;


setpoint_roll = 25.95;
sample_data = […];
Ts = 0.005;
% we'll just keep 20 secs of it:
setpoint_arr = repmat([setpoint_roll],1,length(sample_data))
data = iddata(transpose(sample_data(1:20/Ts)), transpose(setpoint_arr(1:20/Ts)), Ts);
data_fft = fft(data);


plot(data)
plot(data_fft)
```

I used the System Identification app in MATLAB to estimate the transfer function of the robot. I experimented with various configurations, including different numbers of zeros and poles, both continuous and discrete forms of the transfer function, and various sampling durations (120 seconds, 60 seconds, 20 seconds, and 5 seconds). Ultimately, I found that the best results were achieved with a model match of 24.49% in the final configuration. Here are the identification results:



Additionally:

```
SelfBalancingEV_tf =

  From input "u1" to output "y1":
                           -31.91 s^4 - 527.6 s^3 - 37.89 s^2 + 1.411e04 s - 6.627e05
  exp(-0.245*s) * ------------------------------------------------------------------------
-
                  s^6 + 4.439 s^5 + 390.8 s^4 + 1446 s^3 + 3.895e04 s^2 + 1.007e05 s +
6.633e05

Name: SelfBalancingEV_tf
Continuous-time identified transfer function.

Parameterization:
   Number of poles: 6   Number of zeros: 4
   Number of free coefficients: 11
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "data".
Fit to estimation data: 24.49% (stability enforced)
FPE: 3.333, MSE: 3.221
```

Since the model similarity match was so low, I decided to stop here and not continue with system modeling and identification, as there might be some problems I hadn’t noticed in those aspects or within the system itself.

Conclusion

The PID parameters for the robot control system have been tuned through experiments using web control panels, resulting in a steady state error of 0.3 degrees.

This project encompassed numerous fascinating aspects that required several months of my time. It provided me with a wealth of theoretical knowledge and practical experience, inspiring me to embark on future projects with greater interest and a more efficient approach.

Bibliography

[1] University of Michigan, “Control Tutorials for MATLAB and Simulink - Inverted Pendulum: System Modeling,” *ctms.engin.umich.edu*. https://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum§ion=SystemModeling

[2] J. Velagić, I. Kovač, A. Panjević and A. Osmanović, "Design and Control of Two-Wheeled and Self-Balancing Mobile Robot," 2021 International Symposium ELMAR, Zadar, Croatia, 2021, pp. 77-82, doi: 10.1109/ELMAR52657.2021.9550938. keywords: {Robot motion;Tuners;Software packages;Dynamics;Transfer functions;Position measurement;Mathematical models;Self balancing robot;Equations of motion;PD empirical controller;Lagrangian method;ZOH discretization},