Balancing Robot

MIE438 – Project Report

Group 38

Jaden Stanshall (1008792905)

Jaehah Shin (1008851137)

Ian Wu (1009063816)

1. Introduction

This project aims to apply our learned skills in embedded systems to build a two-wheeled balancing robot. This report will cover the project plan, outcome, systems design, and applied course concepts. The links to the project demonstration and code repository are listed below for quick reference:

- **Demonstration:** https://play.library.utoronto.ca/watch/da4d44c41fde8f642a34c4a10b491613
  - The robot is following pre-planned forward trajectory.
- **Code Repository:** https://github.com/JadenStanshall/esp-idf-template
  - For the main code, please refer to the **good_jawn.c**

2. Plan and Approach

2.1. Proposed Design

Our proposal was to build a two-wheeled robot with dynamic stabilization as well as a wireless connection to a custom hand-held control that could call motor commands that allowed control of the robot. The intended design was to have a baseline PID control loop that performed the idle control necessary to balance when no active motor commands were being sent. When motor commands were sent, they would be executed instead (higher priority), allowing a person to control the robot. The system would be on the ESP32 WROOM (wireless capable MCU) and have peripheral IMU/Gyro (for angle measurement), and stepper motors (for the wheels). The wireless communication would have been done through Bluetooth Low Energy which is integrated into the MCU.

2.2. Changes to Design

We removed the wireless and remote-control motor commands from the project but otherwise stayed faithful to the original proposal. This was primarily because it took longer than planned to build the baseline idle balancing system as well as to tune the control of the system such that it was able to idle. Knowing this now, we would anticipate more challenges when trying to integrate peripherals with an MCU as well as for designing, testing, and tuning control systems.

3. Project Outcome

3.1. Overall Design

The new scope of the project (only idle balancing, no wireless) was largely successful on the embedded system design; however the control system performance is not ideal as the system has high oscillation and is not robust. The microcontroller plays a central role in reading measurements from the IMU, processing the measurements into a control signal, and outputting the control signal to the stepper motors. A figure presenting the overall system architecture is below. More photos of the physical system are at the end of the appendix.

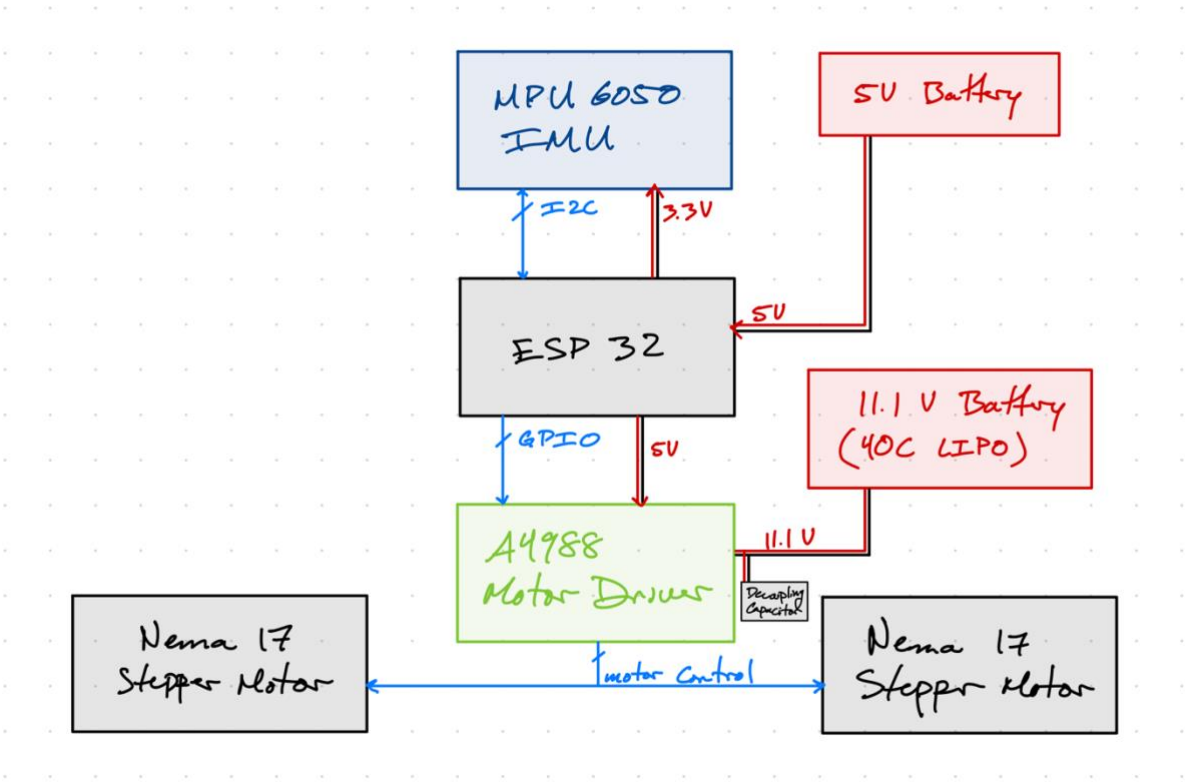


Figure 1: Central role played by the MCU which connects the IMU, processes the measurement and outputs control for actuation.

3.2. Operation of System

Please see the linked demonstration video at the start of this document as well. All mechanical, hardware, and firmware is functioning properly and able to function in the embedded format while battery powered. That is, we can successfully read IMU data, actuate our motors, and send control in a way that attempts to balance the robot. Though the system works correctly, the control outputs are suboptimal as the system experiences significant oscillation and sometimes becomes unstable (resulting in the robot falling). An ideal control system would show close to a critically damped response (little to no oscillation), and stronger disturbance rejection.

Significant causes for the challenges in control are the unfavourable dynamics of the robot, as well as significant noise in the IMU signal. The dynamics are unfavourable since the actuation torque of the stepper motors is very low, meaning it is unable to counteract moderate angle disturbances as the applied torque of gravity is much more than the motors are capable of. This is especially true due to the significant added mass of two battery cells (one for the motors, one for the microcontroller). Secondly, the IMU is quite noisy, and though filtering attempts are made (see Firmware Design, Section 6), the noisy measurements contribute to oscillatory

control.

4. Mechanical and Structural Design

The structure of the robot is simple consisting of a structural square wooden backbone that supports the batteries, breadboard (with MCU, drivers, IMU), and has two stepper motors with wheels attached to the ends. This design is pictured in figure 2. The parts are connected with a metallic tape, because we have iterated our design several times, and the metallic tape worked the best when we changed the mechanical design. First design was with the inverted pendulum shape, which didn't work although we have tried multiple PID values (follows the Ziegler-Nichols method.) There was a lot of noise in IMU reading due to vibrations which lead us to think that we might need to dampen those vibrations to remove the noise in IMU reading. Therefore, after that, we have changed the design to use heavy books to dampen the vibrations, however, as this introduces us the high centre of mass position, it also didn't work. Therefore, we concluded with the design that we have in figure 2. As this can be shown from the figure 2, the battery for motors, and external power bank for MCU are placed under the backbone of the final physical system. This helps to lower the position of the COM so that it can be balancing better become more stable when they are correcting its positions through the PID. The reason for using wide backbone is because, when the wheels are widely separated from each other, lateral stability increases as it handles the disturbances better. From figure 3, this also shows that IMU sensor also centered at the middle of the body as we are reading position data from the IMU sensor and then use that to adjust the PID values accordingly. Therefore, the position of the IMU sensor was also important when we were designing the structure.

Figure 2: The final physical system.

5. Hardware Design

The hardware for the system was fully designed and mounted onto a prototyping breadboard, with the battery power mounted at the bottom of the robot. The following figure shows the system:

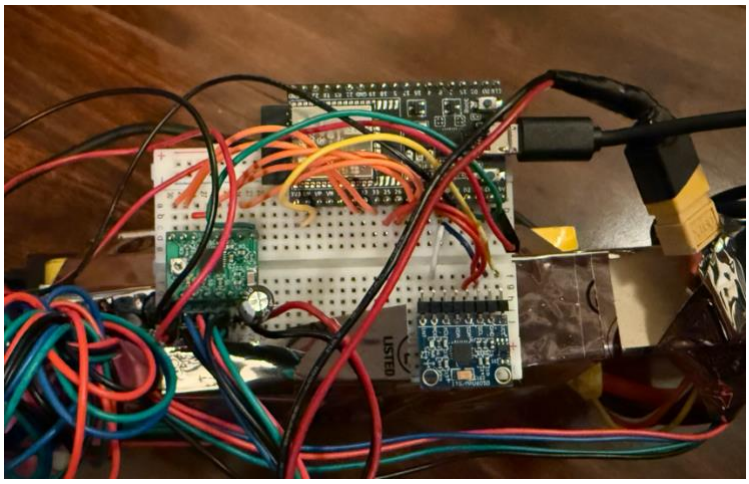


Figure 3: Breadboard system. Three boards pictured, black (MCU), green (motor driver), blue (IMU)

The ESP32 pinout table is also provided below (unconnected pins have been omitted). The reference diagram is in appendix 8.1.

ESP32 Pin	**Target Peripheral**	**Connection Purpose**
5V	A4988 Driver	Power (Logic only)
GND	A4988 Driver	Power
GPIO13	A4988 Driver	Enable
GPIO12	A4988 Driver	MS1 – Stepping mode config
GPIO14	A4988 Driver	MS2 – Stepping mode config
GPIO27	A4988 Driver	MS3 – Stepping mode config
GPIO33 – PWM	A4988 Driver	Step – PWM control
GPIO32	A4988 Driver	Direction
GPIO25	IMU	I2C – SCL
GPIO26	IMU	I2C – SDA
GND	IMU	ADC – Prevent Floating Pin
3V3	IMU	Power
GND	IMU	Power

5.1. MCU (ESP32 WROOM 32E)

When we were flashing the MCU, we used a laptop to flash and power the MUC. However, during the demonstration and testing, the MCU is powered separate from the motors using a phone power bank and has the left pins connected to the board and exposed for making connections to the peripherals.

5.2. IMU (MPU6050)

The IMU is situated on the breadboard and powered by the 3.3V pin from MCU. It is connected to the MCU via I2C for data communication. Note that IMU is at centre, as the readings from

the IMU are used for the PID so that the robot itself can balance based on the current tipping angle.

5.3. Stepper Motor (NEMA 17) and Driver (A4988)

The stepper motor driver connected to the two motors with one set of four H bridge outputs mapping to both stepper motors (ie. they share the same input nodes). The driver also receives ground and 5V from the MCU for the logical voltage, as well as the 11.4V motor drive voltage and ground from the LIPO cell.

5.4. Motor Power Supply (LIPO 11.1V, 4500mAh, 1.5A) and Phone Power Bank (MCU)

The two power supplies are attached underneath the robot. As mentioned the LIPO powers the motor and the Phone Bank powers the MCU (and attached IMU/Driver Logic). The LIPO is attached in parallel with a smoothing capacitor of 100uF to account for large instantaneous current draws from the motors.

6. Firmware Design

The firmware system is programmed using the ESP-IDF (provided by the manufacturer Espressif) and we developed it through the VSCode version of the IDE, using UART for the flashing and serial monitor link to the device during development. Programming was done in C. Specific firmware details are discussed below, please refer to the code linked in the repository at the start of this document. Source files imu.c and motor.c contain critical functions of each task, and they are called through main (good_jawn.c) where the tasks run in their respective task functions (pid_task, stepper_task) which are infinite loops running on separate cores.

6.1. Overall Firmware System

The firmware system primarily contains two separate tasks, one for motor control (constantly outputting PWM for the desired control), one for the IMU reading and PID update (which updates the desired control). The diagram of this is pictured below.

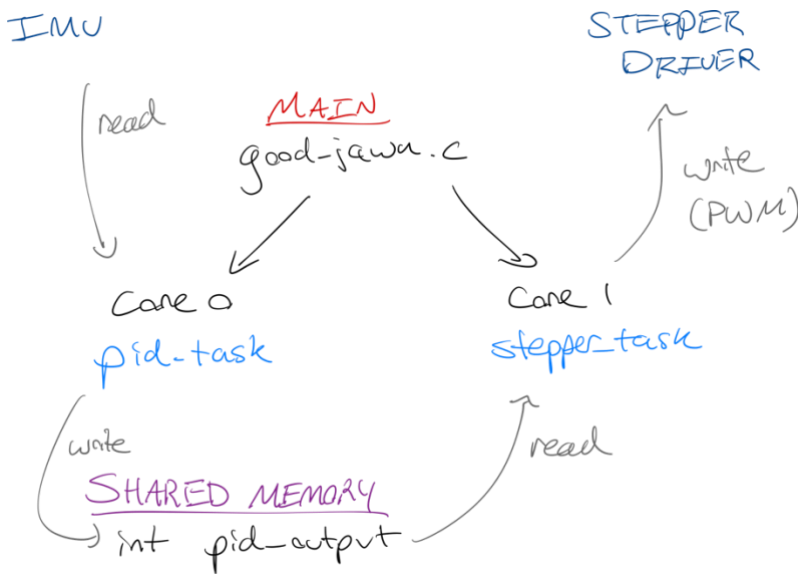


Figure 4: Firmware system diagram. The two cores execute their respective tasks indefinitely until the program is stopped.

6.2. PID Task

The function pid_task is an infinite loop that reads the angle measurement using the get_angle() function that reads using I2C. Several filters are applied to try to smooth out the sensor readings (median filter and clipping signals beyond a threshold. Once the filtered angle is computed, PID control is used to determine what the control output should be. We have implemented additional PID features such as gain scheduling (depending on the error size), and zero crossing detection to reset the integral error.

6.3. Stepper Task

The stepper task continuously monitors the global variable “pid_output” which is computed by the PID task. It determines the desired direction of the motor based on the sign of the pid output, and sets the DIR_PIN accordingly. The magnitude of the pid output is translated into a step delay using the helper function “pid_to_delay()” with a linear mapping, this decides how fast each step of the motor is executed. The motor is then controlled with the computed delay. We also use a unique microstepping configuration to ensure smooth operation while maintaining the necessary speed to balance our robot; the appropriate microstepping configuration was determined by trial and error, and the configuration is set using the GPIO pins 12, 14, and 27. The stepper task also includes a “dead-band" feature where if the robot’s tipping angle is within a certain range of the setpoint, the robot is effectively balanced and the motor speed is set to zero (by pulling the ENABLE pin HIGH on the A4988 driver).

6.4. Two Core, Simultaneous Operation

Separate cores on the ESP32 were used specifically to handle the PID update and motor control tasks separately. This was critical as fast motor updates were necessary to ensure smooth motor

operation, and fast pid updates were necessary to ensure the robot could react appropriately to its changing position. Evidently, we initially attempted to run both the pid computation on the same core in the same while loop as the motor control, but we were experiencing rough motion on the motors which consequently exacerbated the vibrations impacting the IMU measurements; thus, it was necessary to make the switch to multi-core operation.

7. Applied Course Concepts

7.1. Control Loops

We have used the control loops called PID and filter technique Kalman filter which we have learned in MIE 438 (lecture slide 11). Basically, the PID is used to maintain the robot's balance by continuously adjusting the motor control based on real-time feedback from the IMU sensor. The system aims to keep the robot's angle at a defined setpoint, and we get the angle data from MPU6050 sensor. To make sure the accuracy, the raw data is processed with a median filter to remove "sudden" spikes caused by vibrations that was not removed by introducing the dampen. We refined those with Kalman filter. Then those re-defined sensor input is fed into the PID computation routine, which calculates the control signal needed to adjust the motor's movement.

Within the PID update function, the process starts by calculating the error (diff between setpoint and current filtered angle). The code includes safeguards by clamping the error to moderate abrupt changes so that we can avoid the risk of overcorrecting. The integral of the error will be accumulating over time to address any persistent gap, the derivative of error helps to dampens by responding to the rate of change of the error. We have used two different sets of PID gains, one for when the error is within a defined exceeds that threshold. This gain schedule strategy allows the controller to be both stable near the desired angle and more responsive during larger deviations.

The computed PID output is important as it directly influences the behaviour of the motor and also the position of the robot. A higher PID output results in shorter delays, allowing the motor to move faster to correct imbalances. Conversely, low PID values lengthen the delay to prevent unnecessary motor activation. This integration of sensor feedback, dynamic PID calculation, and motor control maintains real-time balance and stability.

7.2. Subroutines and Multiple Core Scheduling

The code that we wrote is organized well into clear subroutines that captures specific functionalities. For example, initializing interfaces like I2C and the MPU6050 sensor, processing sensor data (with median and Kalman Filters), computing the PID control, and managing the motor control. These subroutines simplify the debugging processes and maintain the code well so that it can be used in other platforms more easily. (reusability of the code) Also, the code leverages FreeRTOS to implement multi-core scheduling, where separate tasks (such as PID calculation and stepper motor control) are pinned to different cores. This concurrent execution

ensure that sensor processing and motor control run in parallel, leading to more efficient and responsive real-time operations.

7.3. I2C

The I2C protocol was used to interface between the IMU and the ESP32. I2C is a two-wire synchronous bus with both a serial line (GPIO25) and a data line (GPIO26), and “fast mode” (400kHz) was selected as the sampling rate to enable quick and frequent control updates. Accelerometer data began at address 0x3B (acceleration in x axis), from which sequential measurements (acceleration in y and z) can be accessed (0x3B + 2 and 0x3B + 4 respectively). To begin each communication, the ESP32 sends to the I2C bus the MPU6050 address (0x68), followed by the particular acceleration data register address (0x3B, where the data begins). The MPU6050 returns on the I2C bus two bytes for each axis representing each measurement as a 16-bit signed integer. We then combine the two-byte pairs for each measurement into signed integers with bitwise operations for use in our program.

7.4. Simulating Pulse Width Modulation

In our code, we simulate PWM-like behavior through software by manually toggling the stepper motor's STEP pin. The function calculates a variable step_delay_us based on the desired RPM, which determines the duration of both the high and low states of the signal—effectively creating a square wave. With the STEP pin high for step_delay_us and low for the same duration, the complete period becomes 2 * step_delay_us, yielding a frequency of 1/(2 * step_delay_us). This square wave drives the stepper motor, where each rising edge represents a step. Therefore, reducing the delay increases the frequency of these pulses and, consequently, the motor's speed, while increasing the delay slows the motor down. This approach, while emulating some aspects of traditional PWM, is implemented entirely in software rather than using dedicated hardware PWM channels.

7.5. Register Addressing and Incrementing

In our code, we define several macros to establish clear and explicit addresses for the MPU6050 sensor’s registers. For example, we set MPU6050_ADDR to 0x68, which specifies the sensor’s I2C address, while MPU6050_PWR_MGMT_1 is defined as 0x6B to target the power management register, enabling us to wake the sensor upon start up. Most notably, MPU6050_ACCEL_XOUT_H is set to 0x3B, designating the base address where the accelerometer data is stored. Since the sensor arranges its data registers sequentially, we can increment this base address to access adding 2 to obtain the Y-axis and by adding 4 for the Z-axis data. This addressing and incrementing strategy simplifies sensor data retrieval by allowing us to efficiently access all necessary measurement registers in a structured and predictable manner.

7.6. Variable Storage and Scoping

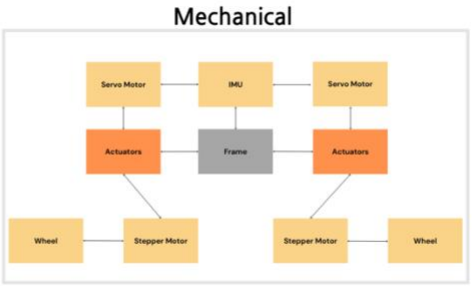
The code employes strategic variables storage to manage both constant values and dynamics

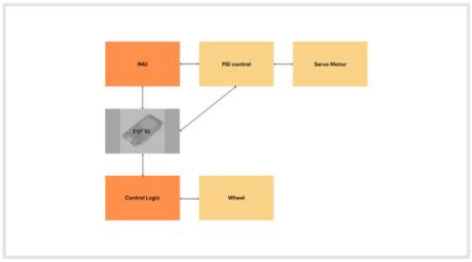
data needed for sensor processing and control. Global macros define immutable addresses for the MPU6050 sensor registers, such as the sensor's I2C address, the power management register, and the starting register for accelerometer data. Also, global variables are used to store PID parameters, facilitating persistent access throughout the program's execution. Variables like "previous_angle", "lastError", and "integral" maintain state between successive PID calculations, ensuring smooth control adjustments over time. The code also allocates memory for filtering purposes, including a median filter bugger and a kalman filter instance, which are important for processing raw sensor data. Altogether, this organized variable storage creates a framework where constants, sensor values, and computed parameters coexist, allowing the system to efficiently perform real-time balance adjustments and motor control operations.

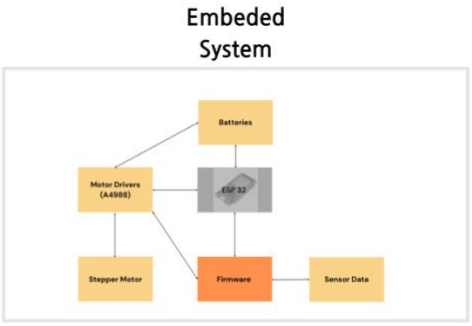
8. Appendix

8.1. Microcontroller Pin Diagram

8.2. Subsystem Diagrams

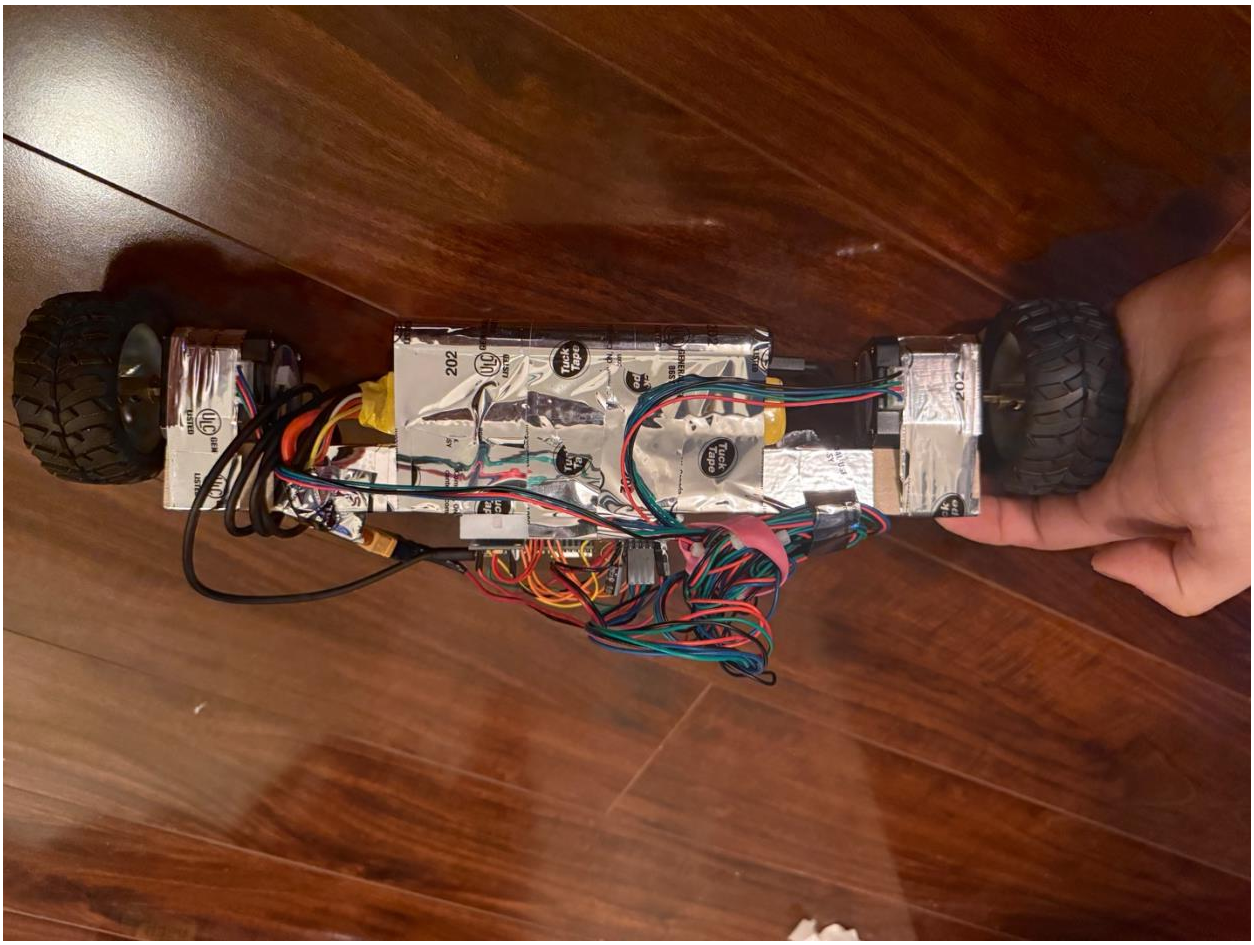






8.3. Supplemental Pictures of Robot


Computer Science and Scientific Computing


202
Duck Tape
Duck Tape
202