Senior Engineering Design Project: Assistive Self-Stabilizing Robot

Omar Escareno, Khem Holden, Andrew Lei, Jeffrey Ng, Jornell Quiambao, Diego Reyes

June 5, 2020

Executive Summary

The report aims to discuss the design of the Assistive Self-Stabilizing Robot project and its process. This project illustrates the collaboration of different hardware and software components to design a final product that is tested in a simulated environment. The design of this project is first formed when consulting the needs and goals of users and clients, personas. The design for an Assistive Robot fulfills the needs of helping elderly and disabled persons with carrying personal items. The robot must be able to avoid obstacles, balance itself on two wheels and track its user. The two-wheel design of the robot was chosen for its mobility, providing a 0^o turn, and for its simplicity, as it requires fewer components to move.

The hardware design of this project consists of the physical model and electronic components including sensors, actuators and microcontrollers. The physical design was created as a CAD model which follows the preliminary design of a two-wheeled rover. The electronic components were chosen to meet conditions for tasks that the robot needs to accomplish. The sensors used were a ping sensor to detect objects, a gyro sensor to detect the robot's orientation, and a camera to give the robot vision to track objects. The robot uses motors and wheels for its main form of movement. The robot is controlled by microcontrollers to utilize the data given by the sensors and to distribute commands to the actuators.

The software design consists of accomplishing the tasks of this project. The robot is primarily controlled through a state machine. The state machine controls the robot through a reactive programming system of taking events gathered from sensor data and servicing the events to other states, which control the actuators of the robot. Running alongside the main state machine is a controller that will keep the robot stabilized. The controller uses the physics of an inverted pendulum and controls the motors through a feedback loop to keep the pendulum balanced. To track its user, the design implements a computer vision system that outputs the location of its user to the state machine. The computer vision aspect is implemented on its own microcontroller to account for its resource usage and high specification requirements.

With the hardware and software aspects of the project, the designs can be integrated a single simulation. The robot is simulated in a Gazebo environment and controlled by ROS. The simulation prototype includes the aesthetic model of the robot taken from the CAD design, simulated sensors that the robot will use physically, the state machine to control the system, computer vision tracking with a real object, and self-stabilization.

Contents

1 Introduction

1.1 Needs and Goals

Persons with certain disabilities may find it more difficult to transport various items than the average human, such as elderly and disabled persons. Our objective was to develop some form of external assistance to alleviate this difficulty.

1.2 Personas

The personas we have created are examples of target end-users who would benefit from assistive technology. The device must be easily accessible and understandable as to not interfere with the daily tasks of the user, but to support them.

1. **Eugene Oldsen**
   Eugene is a retired military veteran who received an arm injury, which has resulted in him being unable to carry any items heavier than 2 pounds. In order to walk extended distances, he requires the use of a walker, which keeps his hands occupied, even if he were even capable of carrying anything anyway.

2. **Kris Kripple**
   Kris was born with a case of osteoporosis, and therefore is confined to a wheelchair for a majority of his transportation. It is difficult for him to carry items, especially heavy ones, on his own without risk of injury.

3. **Jamie Normalman**
   A user who represents an average human, capable of carrying items. External forms of assistance are not necessary for this user, but he can still use our product out of convenience.

1.3 Design Objectives

In order to satisfy the need of an external form of assistance, we set out to develop an autonomous storage bot, capable of following its user. This robot was meant to be very affordable, with a price range less than $200, while also being a decent size, with a storage capacity of at least a 20x20x20-inch cube. It will be made of a lightweight material to account for the added weight. For its battery capacity, it would ideally last for at least an hour. The robot will have two wheels for easy mobility while following its user and avoiding obstacles. The tracking will have an error margin of 5% and the self-stabilizing system will have an error margin of 1%.

2 Design

2.1 CAD Design

The second half of our project implementation was going to involve the fabrication and testing of our design on a physical model. Unfortunately, this quarter presented us with limited resources since our fabrication lab access was revoked due to a virus outbreak. Because of this, it was up to us to figure out a new way to test out our different modules with the main focus being on simulating the PID controller which was an important part of our design. For this reason, it was imperative that our CAD model was as accurate as possible since it would be the only visual representation of our robot. Additionally, the design was expected to be aesthetically pleasing enough for someone to use, meaning that the rover would carry out its intended capabilities while still maintaining a modern look.

2.1.1 Initial Design

Before starting on the CAD model, it was necessary for an initial design to be sketched out in order to get an idea of what the robot would look like. This was primarily chosen by our decision table which narrowed down our options to a drone, a four-wheeled rover, or a two-wheeled rover. In the end, the 2-wheeled rover design obtained the most amount of points, meaning that it was the optimal design to carry out our objectives appropriately and efficiently.

Since it was decided that the system would be a two-wheeled differential drive robot, it was possible to start deciding on the overall shape and dimensions of the robot. The main purpose of the rover would be to carry as much weight as possible for the user while still being able balance itself both in and out of motion. This essentially meant that our system would need a rather large base to increase carrying capacity as well as large wheels to allow for weight to be distributed evenly. The act of self-stabilizing would rely mainly on the type of actuators used, since they would need the strength to balance out the robot's weight in distinct scenarios. In addition, space would need to be provided for the sensors and other electrical components involved in the system.

Taking all this into account, a total of three different designs were created which followed closely to the minimum specifications set out for the rover. The following pictures show the sketches made for each design along with short descriptions on the reasoning behind them.

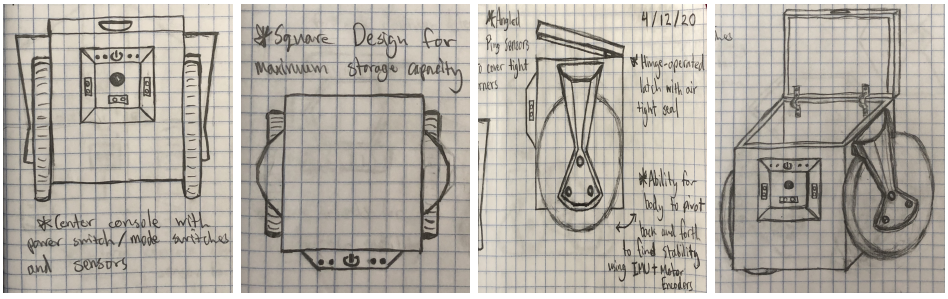


Figure 2: *Design 1 which has a cube-like base with latched lid.*

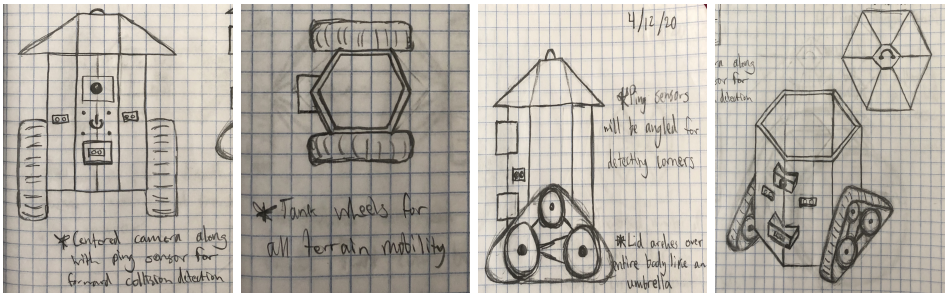


Figure 3: *Design 2 which has a hexagon-like base with umbrella lid.*

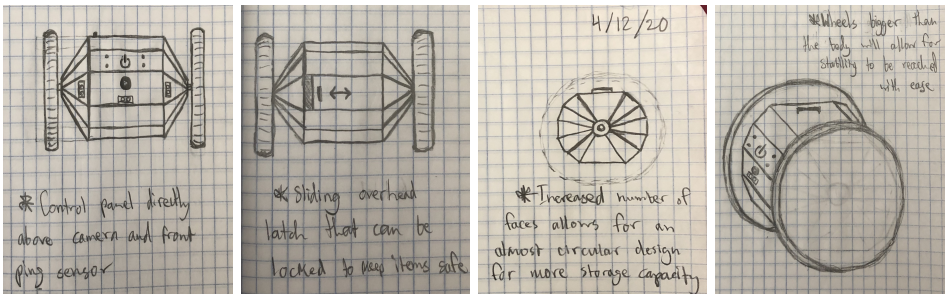


Figure 4: *Design 3 which has a multi-sided base with a sliding lid.*

In the end, it was decided by majority vote that we would go with the first design since it had the largest storage capacity and its simplicity was much more appealing to the eye. Also, the incorporation of wheel mounts would allow for our PID controller to do its job appropriately since these parts would essentially be capable of swinging back and forth when necessary. With a basic outline in hand, it was finally possible to begin our CAD model.

2.1.2 Final Design Specifications

The CAD model was created using Autodesk Inventor Professional 2021 which allows for the creation of 3D models with great attention to their constraints and dimensions. The units used to create the individual parts was inches with all initial creations being started on the bottom plane. This was important since the assembly of the robot was made a lot easier with all parts sharing the same reference frames. The biggest part was the main base of the robot which was a large cube that would also include the center console in charge of initializing the system's various autonomous features. Originally, it was decided that the robot would be created using multiple tab and slot pieces and bolts to fasten them together, but due to the laser cutter being inaccessible, it was possible to depict the robot as one single unit.

The overall dimensions of the two-wheeled rover were determined based on how big we wanted the robot to be in comparison to the user. Additionally, it had to be noted that the robot's main purpose was to act as a storage compartment, ultimately insinuating that its total volume had to be reasonable. It was decided that the overall base of the rover would stand at 24in x 24in x 24in since it seemed like a reasonable height in comparison to the average human. Attaching the wheels would

increase the height to about 27-28in, which was necessary since the entire base would be capable of swinging, meaning that an overshoot could easily damage the robot upon impact with the ground. Below are a couple of images depicting our robot in comparison to a scaled human model, allowing for a better understanding of what our system would look like in a real-life situation. Following that is an annotated document showing the overall robot dimensions.

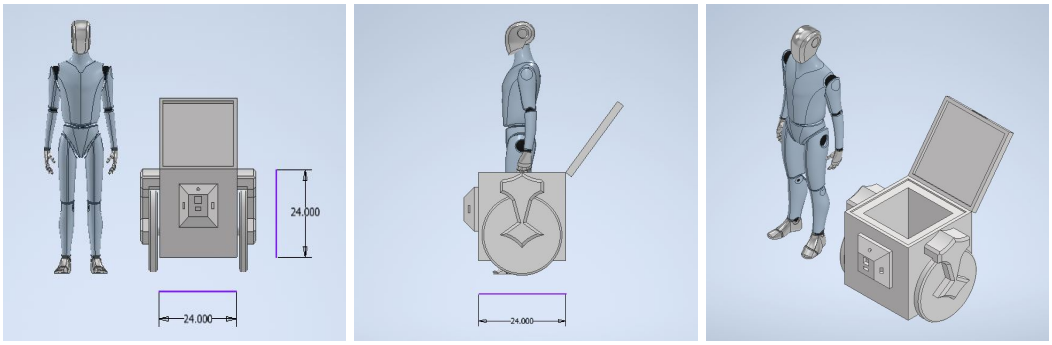


Figure 5: *Front, top, and isometric perspectives of the robot next to a 5ft scaled human model.*

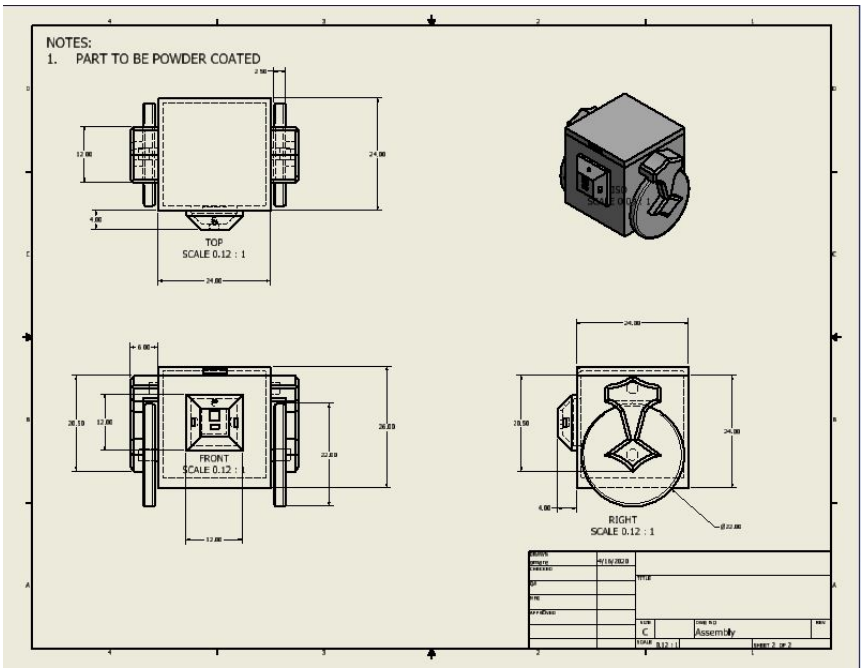


Figure 6: *Drawing annotations of the overall rover structure.*

Another important feature that had to be taken into account when creating the parts was the integration of hardware components. Our system would include two main micro controllers, two

actuators with an H-bridge, three ultrasonic sensors, one center-facing camera, and an inertial measurement unit. Because of this, the necessary amount of space had to be allocated while also making sure that these components could not be directly tampered with by the user. This was done by creating a separate smaller base within the system that would act as the actual user storage compartment. With the inner base having a size of about 22in x 22in x 22in, it was made possible to line the inner edges of the overall base with electronics, wires and other back-end components needed to make the system work. The following are depictions of this inner base and how it would fit within the overall robot as well as a mapping diagram showing the general sensor placements,

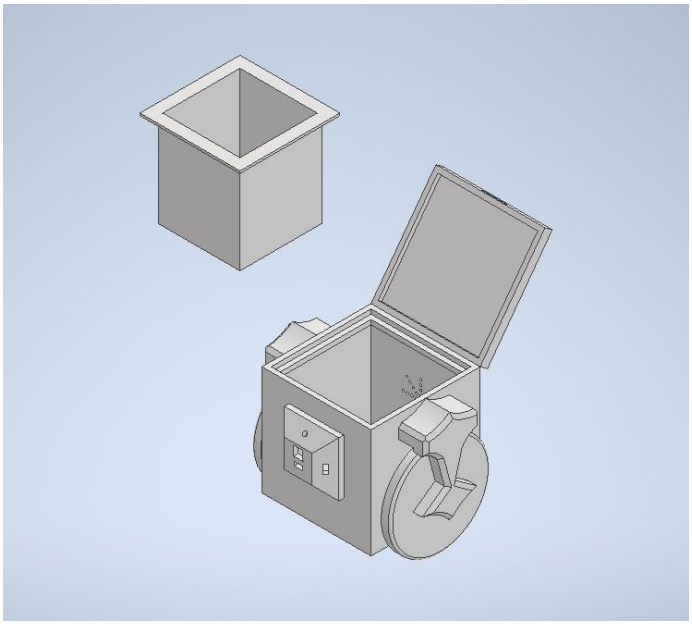


Figure 7: *Deconstructed inner base assembly showing allocated space for electronics.*

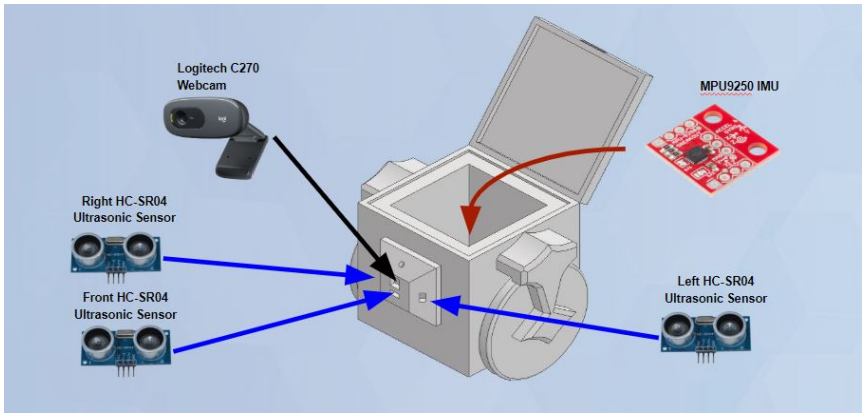


Figure 8: *Mapped out sensor placements on the robot.*

It can be seen that the rover would incorporate the three ultrasonic sensors in such a way that all horizontal angles would be covered. This was intentionally done to avoid collision which will be described later on. Additionally, the camera was placed in the center most part of the robot since this would be the main source of conducting computer vision tracking. With all the electronics being taken care of, it was required that vents be created in order for heat to dissipate properly. They were placed in the back side of the robot and would more than likely work alongside fans to avoid the overheating of our electronics. Before beginning the assembly process, a material had to be set which resulted in a PC/ABS plastic. This material was chosen since it allowed for the robot to be sturdy while also being lightweight, allowing for an easier implementation of the self-balancing feature.

The final assembly of the robot had to be done carefully since it was necessary that the appropriate constraints be placed in the right places. For example, the lid would have an angled constraint since it would be required to flip open while the wheels would have a concentric constraint, allowing them to rotate. Doing so also made it easier to animate the robot's movement and give a live demo of the distinct autonomous parts involved. Once all parts were properly mated, it was possible to treat the system as one whole unit and begin using the model for simulation purposes. Below are different perspective of the final CAD model.

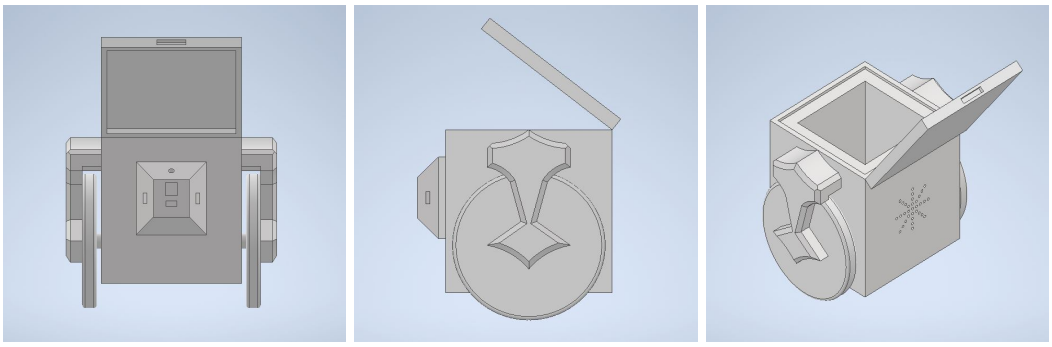


Figure 9: *Front, side, and back perspectives of the final CAD model.*

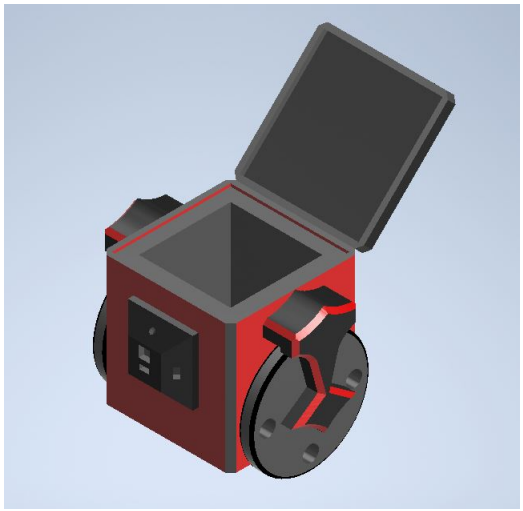


Figure 10: *Isometric perspective of the finished robot model with color.*

2.2 Sensors and Actuators

The sensors and actuators are the electronics components that interact with the environment. The microcontrollers gather input from the sensors' data and then output these data onto the actuators. The robot needs to avoid objects, self-balance, and track its target. Ping sensors are useful for calculating distances in front of the sensor. An IMU is useful for calculating the orientation of an object. A camera allows for the use of computer vision. Since the robot will be a 2-wheeled rover, motors are used to control the wheels.

2.2.1 Sensors

The sensors of the robot are used for the distance tracking, object tracking, object collision and self-balancing. The robot uses 3 HC-SR04 Ultrasonic or "Ping" Sensors (11), 1 MPU9250 IMU (12) and 1 Logitech C270 webcam (13). The ping sensors are placed on the sensor mount in the front face of the robot, with 1 in the front and 2 along the side panels. The MPU9250 IMU is placed inside the base of the robot along the center axis of the wheels, so that the gyro sensor and accelerometer will get more accurate readings. The camera is placed in the front face on the front panel of the sensor mount and angled towards the user it will be tracking.

The HC-SR04 Ultrasonic or "Ping" Sensors use ultrasonic sound waves to determine distance. The sound wave pulse is emitted by a trigger and it receives the same pulse with its echo. This allows the ping sensors to signal object detection with collision. This also allows the robot to track its user from a distance. Our target range for the robot's following distance is 2 meters or 6 feet. This ping sensor supports up to maximum range of 4 meters. The software implementation of the ping sensors allows a microcontroller for parallel processing 3 of these sensors.

The MPU9250 IMU is needed for its built-in gyro sensor and accelerometer. An IMU is an inertial measurement unit, and this will help with determining the orientation of the robot, specifically the angle in which it tilts. The gyro sensor of the IMU outputs the angular velocity of the robot when it tilts about the wheel axis. Using the angular velocity from the gyro sensor, we derive

the angle of tilt to be used in the self-balancing controller. Since there is noise within the IMU, the accelerometer is used to remove the noise through Euler angles and Integration, which will be explained in Section 3.1.1.

The Logitech C270 camera is a webcam, which mainly enables the use of computer vision for the tracking system of the robot. The webcam supports a video resolution of 720p and frame rate of 30 FPS. It is connected to the Jetson Nano via USB.

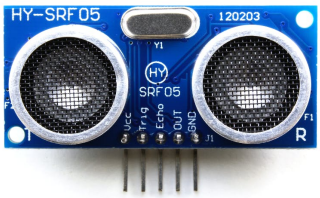


Figure 11: *HC-SR04 Ultrasonic or "Ping" Sensor [1]*

Figure 12: *MPU9250 IMU for the Accelerometer and Gyro Sensor [11]*

Figure 13: *Logitech C270 Webcam [4]*

2.2.2 Actuators

The actuators of the robot are used for physically moving the robot. The robot only uses 2 DC motors (14), which are both controlled by a single L298N Motor Drive Controller H-Bridge (15). The motors are placed at the center of the wheels on each side of the robot. The H-bridge is placed at the base of the robot at the electronics compartment, at an equal distance from the 2 motors.

The DC motors of the robot needed to be able to move the wheels so that it can carry the weight of the robot. The maximum wheel speed of the robot when self-balancing is $0.5\frac{m}{s}$. To achieve that speed, we would need a motor with angular speed of about 50RPM. To account for faster moving speed of the robot and extra weight that will placed in its compartment, we chose a motor that will provide more torque, which is a DC 12V 30RPM motor. The standard voltage input for DC motors is 12V. A 12V power supply and ground is needed to be connected to the positive and negative terminals of the motor to fully power it. Lowering the voltage supply to the motor will reduce its speed output.

To control the DC motor speed through software, an H-bridge is needed. An H-bridge is a circuit configuration of switches in the form of an "H". This configuration allows the H-bridge to open and close switches to correct terminals, allowing the inputs to the motors to receive a reversed polarity signal. The reversed polarity signal allows the DC motors to drive in reverse. The H-bridge driver also allows the robot to change the speed of the motors because the terminals of the DC motors are not directly connected to the power supply. The H-bridge input terminals are from PWM signals that increase and decrease the DC motor speed by changing increasing or decreasing the duty cycle of the PWM signal. A single H-bridge can output to 2 separate DC motors.

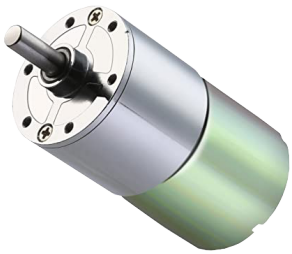


Figure 14: *DC 12V 30RPM Gear Motor [3]*

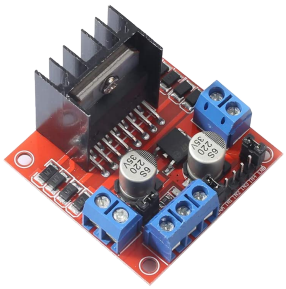


Figure 15: *L298N Motor Drive Controller H-Bridge [10]*

2.3 Microcontrollers

The overall system incorporates two separate microcontrollers: the Raspberry Pi 3 B+ (16) and the NVIDIA Jetson Nano (17). The two microcontrollers divide tasks with the Jetson Nano handling computer vision, while the Raspberry Pi handles everything else. Since computer vision requires a lot of resources from its microcontroller, the Jetson nano is solely dedicated to this job because of the better hardware specifications of the two microcontrollers. The two microcontrollers communicate data through GPIO pins to utilize the sensors and actuators and to be used in tandem with each other.

2.3.1 Raspberry Pi 3 B+

The Raspberry Pi is the main microcontroller of the system, as it handles all tasks except for computer vision. The Raspberry Pi handles the main control through the state machine, implements the self-balancing controller, gathers data from the IMU and ping sensor outputs and drives the H-bridge to control the DC motors. The microcontroller connects to all hardware electronic components through its GPIO pins. For the ping sensor, 2 GPIO pins were set to digital input and outputs to trigger the ultrasonic pulse and receive the echo signal. For the H-bridge, 2 GPIO pins were set as PWM signals to control the motors. The communication between the Raspberry Pi and the IMU was a master-slave communication through I^2C. The I^2C connection allows the IMU to send more data than the GPIO pins, which is useful for sending data from both gyro sensor and accelerometer.

The Raspberry Pi takes in a 5V input from a micro-USB to power the microcontroller. The Raspberry Pi has both 3.3V and 5V GPIO outputs to power the IMU and ping sensors, respectively. Using NOOBs and Raspbian OS, the Raspberry Pi can be directly configured without the need for external systems.

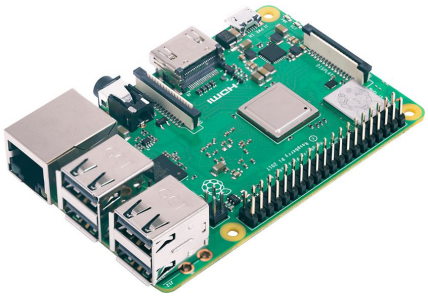


Figure 16: *Raspberry Pi 3 B+ Microcontroller [9]*

2.3.2 Jetson Nano

The Jetson Nano was used solely to process the computer vision algorithm and data of the system. This was due to the fact that it possesses its own 128-core Maxwell Graphics Processing Unit [8], making it far better-suited to handle tasks that were graphically heavy. As discussed in Section 2.2.1, the camera we used for this project receives images with a resolution of 720 vertical pixels at a rate of 30 frames per second. Each pixel on each frame contains a dedicated data value which indicates its particular color, with a total of 921,600 pixels per frame.

Figure 17: *NVIDIA Jetson Nano Microcontroller [8]*

2.4 Wiring Diagram

The wiring schematic was separated into for main sections: Power, Microcontrollers, Sensors, and Actuators. Our primary power source was a 12-volt battery, which was connected to a power distribution board, allowing us to use both a 12-volt power level, which was needed for the actuators, and a 5-volt power level for all other components. The camera was wired to the Jetson Nano via a USB connection. After computing the location of a tracked target, the Jetson would then send these data to the Raspberry Pi across two GPIO pins. The ping sensors are powered with 5V and transfer data through GPIO pins. The IMU is powered by 3.3V and transfers data through an I^2C connection. The H-bridge is powered with 12V from the distribution board and receives data through PWM signals from GPIO pins from the Raspberry Pi.

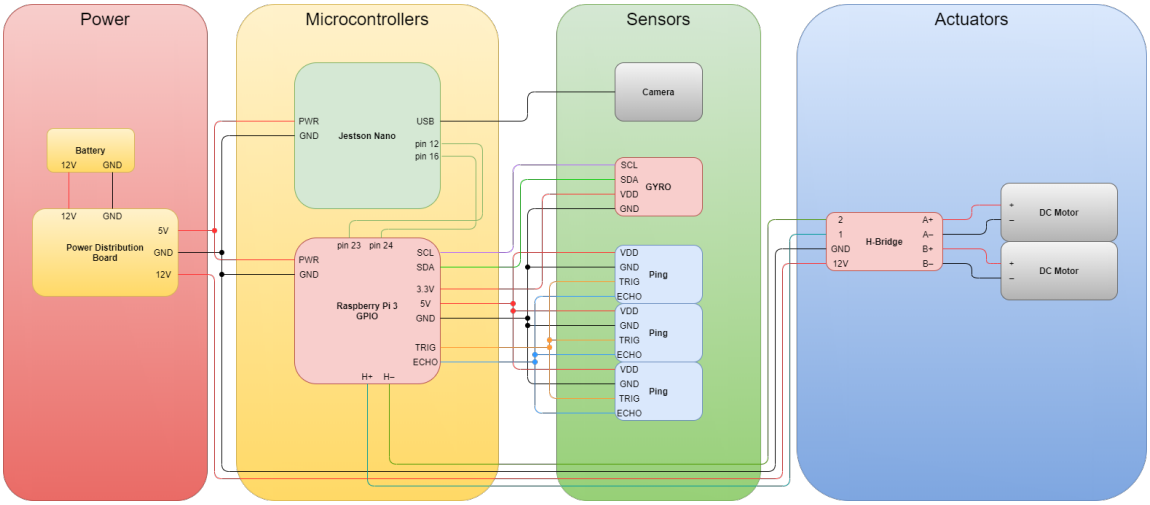


Figure 18: *Wiring Diagram of full system with all of electronic components [2]*

2.5 State Machine

2.5.1 Overview

The autonomous nature of the robot required a state machine to control it. A hierarchical model was used to handle various sub-events, as shown in Figure 19. There are three source files overall that control the system's primary automation: *PingAlt.py*, *event_checker.py*, and *state_machines.py*. Firstly, *PingAlt.py* is used to establish the GPIO pins on the Raspberry Pi to which the actual sensors will be wired. It will then be set into a constant loop in which it will read the values of each sensor and send the data to *event_checker.py*. The source file *event_checker.py* holds several data values which contain the values of each sensor. Every 500 milliseconds, the event checker will receive new data from *PingAlt.py* and compare them to the values currently held in these data values. If the new values have passed over certain thresholds in comparison to the old ones, the event checker will then modify the state machine to act correspondingly. Finally, the event checker will overwrite the old data with the new values and wait another 500 milliseconds for the next batch of data from *PingAlt.py*.

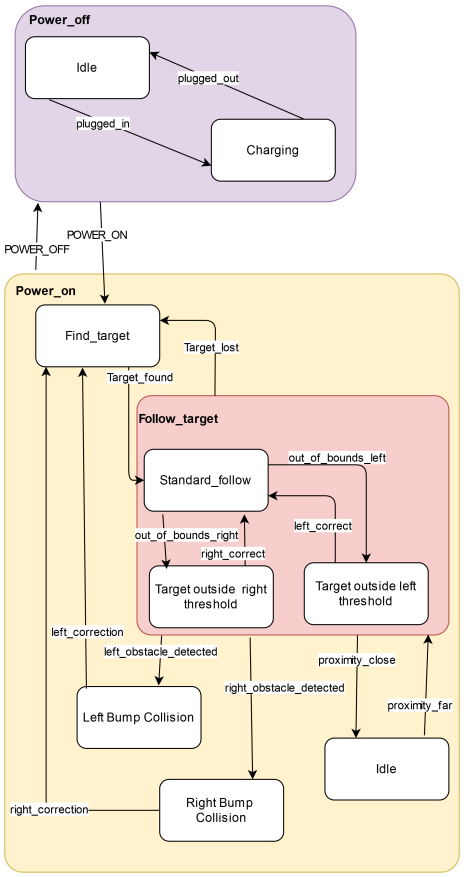


Figure 19: *Overview of complete hierarchical state machine [2]*

2.5.2 PingAlt.py

PingAlt.py interfaces the ping sensors to obtain a digital value related to the distance of the object in front of the ping sensor. In this case, we used three ping sensors placed facing the center, left, and right directions in front of the robot.

The ping sensors communicate with the state machine through the Raspberry Pi GPIO pins. When PingAlt.py is initialized by the event checker, it assigns an input pin and an output pin to each of the ping sensors. The initialization of PingAlt.py requires a list parameter that it will regularly update according to the set time interval. At the end of the initialization, PingAlt will create a timer-based thread for each of the ping sensors, sending out an output, 1, to set the trigger pin of each ping sensor to high. This thread is set to run for 10 microseconds. This high time of the pin will cause the ping sensor to send out ultrasonic pulse in front of it. At this point the echo pin of the ping sensor will go high. PingAlt.py will take note of this time and wait for the echo pin to go low, signalling that the ultrasonic pulse sent out earlier had been received back by the ping sensor. The time difference between sending out and receiving the ultrasonic pulse is the number that will then be translated into the distance between the ping sensor and its target. The values of the array passed through the PingAlt initialization are then updated to the new values. Afterwards, each of the threads are reinitialized and the cycle autonomously continues.

2.5.3 event_checker.py

The event checker begins by initializing a child thread to check input data every 500 milliseconds, as well as establishing parameters which will be used to check for events. As mentioned above, events are registered by comparing new batches of input data to old sets of data stored in arrays. There are five inputs which can cause events in the event checker: the power switch, the computer vision system, and the three ultrasonic sensors. The system is initialized on startup to be powered off, with the computer vision's target not found, and no objects within range of any of the ultrasonic sensors.

The power switch, by nature, is only capable of being in two states, and therefore its value is only set to either 0 for OFF or 1 for ON. As expected, the events for this input check to see if the current value does not match the previous one and to activate a "powered_on" or "powered_off" event accordingly.

The state machine takes in the computer vision's output through two digital GPIO pins. This output is then translated into one of four computer vision states: "lost", "center", "left", or "right". As this input is initialized to the "lost" state, the user must first move to the center range of the computer vision in order for a "target_found" event to occur and the system to begin actively tracking the target. Once tracking has begun, the system can then trigger the "too_far_left" or "too_far_right" events as the target is registered to move into those respective thresholds from the computer vision. Similarly, as the target moves from the left or right sides back to the center of the computer vision, the "return_from_left" or "return_from_right" events are respectively triggered. At any point in these states, if the target is said to be "lost", a "target_lost" event is triggered and the system will wait for the target to move back to the center of the computer vision. It should be noted that in this case, the thresholds for "left", "right" and "center" are predefined in the Jetson Nano itself, and not the Raspberry Pi in which the state machine algorithm is run. This is in comparison to the ultrasonic sensor values which are measured directly through the GPIO of the Raspberry Pi.

Among the three ultrasonic sensors, the center one serves a different purpose from the two on

the sides. The purpose of the central sensor is to measure the distance between the robot and the user, making sure that the two maintain an optimal distance. If the user is measured to be more than a certain distance away, the event checker will activate an event, "proximity_far", indicating that the robot should move forward and close the distance. If the target is measured to be too close, however, the event checker will then trigger the event "proximity_close", indicating that the robot should stop so that it does not collide with the user. Alternatively, sensors at the sides of the robot are meant to check for potential collisions with incoming obstacles. If an object is detected within a certain distance of either of these sensors, a "left_bump" or "right_bump" event is respectively triggered, indicating that the robot should modify its course to avoid the incoming object. Once the object is no longer detected within the range of the sensor, a "left_correction" or "right_correction" signal will be triggered and the system will resume following the user as normal.

It is worth noting that the ultrasonic sensors use an internal unit of measurement, with values ranging from 0 to 1023, with a 0 corresponding to a very close distance and 1023 corresponding to a distance of approximately 4 meters. Based on this system, the central ultrasonic sensor will follow the user until the user is within a range of 500 units. Once the user moves beyond 800 units away, the robot will resume following them. This means that at any point in time, the robot should maintain a distance between approximately 2 and 3 meters from the user. For the left and right sensors, an incoming object will not be detected until it comes within 200 units of the sensor, and the object will be considered cleared once it moves beyond 400 units of the sensor. This corresponds to distances of 0.75 and 1.5 meters, respectively. It should also be noted that upon initialization, the starting values of all of the sensors are set to 1000, which is very close to their maximum values. This was done deliberately to ensure that events could be properly read by each sensor.

2.5.4 state_machines.py

As mentioned above, the state machine receives events from the event checker and transitions between states accordingly. The highest level of hierarchy in the state machine is the Power_on and Power_off states. These, intuitively, correspond to whether the device has been powered on or off by the user. Within the Power_off state, the system checks to see if the battery is currently plugged in to charge. If so, it transitions to a Charging state, and otherwise waits in the Idle state.

Once powered on, the system will begin by searching for its specified target. Once the target has been found by moving to the center of the camera's field of view, the robot will transition into the Follow_target state, where it will follow the user as specified above. All events discussed in Section 2.5.3 are properly accounted for, as seen in Figure 19. It should be noted that upon detecting an incoming object from either side and taking action to avoid that object, the system must once again find its target.

2.5.5 test_bench.py

In order to make sure that the state machine was properly checking for events and transitioning between states, we created test_bench.py. The goal was to be able to interact with the state machine using only manual inputs. As altering the behavior of the state machine to accommodate this task would defeat the purpose of the test bench, it had to have minimal invasion within state_machines.py. Ultimately we had to add functions within state_machines.py that would properly communicate the current state within the state machine. We also had to alter the initialization of the event checker to signal that the test bench was in use and to use the mock sensors rather than real ones, and we had to create a function within event_checker.py that would allow test_bench.py to modify said

mock sensors. Within the test_bench.py file, the main function would run a loop ending only with a sensor input 'STOP'. Within the loop, the function would take in a sensor name and a value to be assigned to the sensor. It would then use the Alter_Sensor function from event_checker.py to change the mock sensor value. After, the function would sleep for 0.5 seconds to ensure that a full cycle of the event checker time interval passed, allowing for the changes to be registered. It would then run the state machine. If an event occurred, the state within the state machine will have changed from the previous loop.

2.6 Self-Balance Controller

The goal of the self-balance controller is to stabilize the orientation of the robot to an upright position as it is stationary, moves forward, or turns. The implementation of the self-balance controller is using a feedback controller that controls the motors of the robot given an angle input calculated by the gyro sensor. The principles of a self-balancing system uses the physics of an inverted pendulum system.

2.6.1 Inverted Pendulum System

The inverted pendulum system for self-balancing includes a cart and pendulum. When the pendulum is offset from being upright, a force is applied to the cart to offset the angle from the upright position. In Figure 20, the inverted pendulum system is modeled with the masses, lengths, forces, and inertia. The model of the system is a cart mass with wheels and a pendulum mass attached to the cart mass by a joint. Applying motion to the cart mass will swing the pendulum mass as well.

The equations of the system can be modeled as transfer functions of the cart and pendulum [6]. The transfer functions of the system both take a force impulse on the system as an input. The transfer function of the pendulum system outputs the new angle of the pendulum mass as a result of the applied force. The transfer function of the cart system outputs the new horizontal position of the cart mass as a result of the applied force.

M: mass of cart
m: mass of pendulum
I: moment of inertia of pendulum
l: length of pendulum
b: coefficient of friction of cart
g: acceleration due to gravity

$$P_{pend}(s) = \frac{\Theta(s)}{U(s)} = \frac{\frac{ml}{[(M+m)(I+ml^2)-(ml)^2]}s}{s^3 + \frac{b(I+ml^2)}{[(M+m)(I+ml^2)-(ml)^2]}s^2 - \frac{(M+m)mgl}{[(M+m)(I+ml^2)-(ml)^2]}s - \frac{bmgl}{[(M+m)(I+ml^2)-(ml)^2]}} \quad [\frac{\text{rads}}{\text{N}}]$$

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

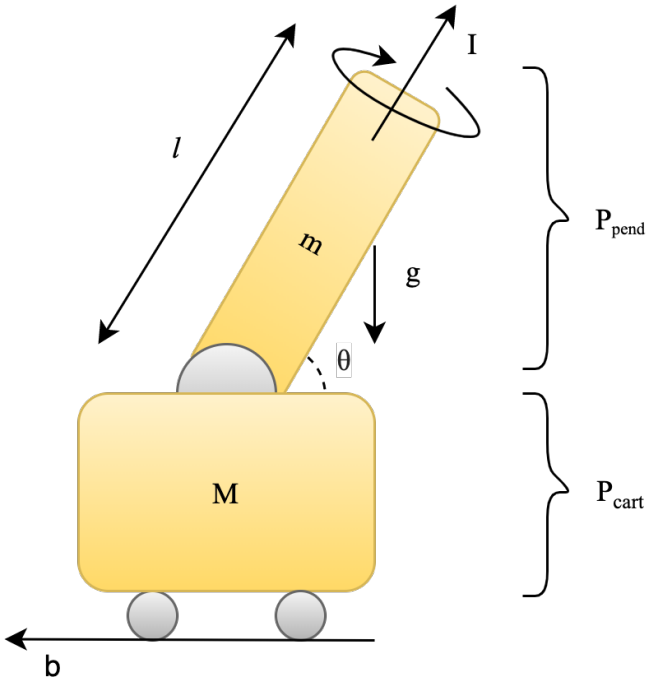


Figure 20: *Inverted Pendulum System of mass cart and mass pendulum [2]*

2.6.2 PID Controller

The feedback controller that is implemented is a Proportional Integral Derivative (PID) controller because of its easy implementation, familiarity and simplicity. The proportional gain of the controller uses only a gain; and increasing this gain would result in an increase in overshoot and decrease in rise time and steady-state error. The integral gain of the controller uses a gain and an integrator; and increasing this gain would result in an increase in overshoot and settling time and decrease in rise time and steady-state error. The derivative gain of the controller uses a gain and a differentiator; and increasing the gain would result in a decrease in overshoot and settling time.

The PID feedback controller works by finding the change in angle compared to the set point (for our system, it is 0^o) and multiplying it with the sum of the PID gains to use in the inverted pendulum system and finally output to the motors. For our system, we implemented the PID controller using first MATLAB and Simulink, then Python and ROS. The PID gain values chosen differ for the two different simulations

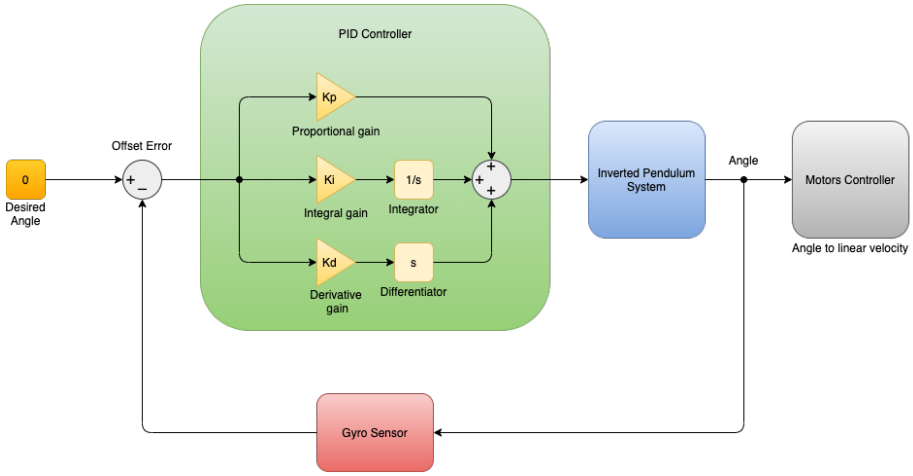


Figure 21: *Block Diagram of PID controller with the inverted pendulum system for feedback self-balancing control [2]*

2.6.3 MATLAB and Simulink

The simulation and testing of the inverted pendulum system and PID controller were processed through MATLAB and Simulink. For this simulation, the values and variables for the system are assigned in a MATLAB workspace, while the physics for the inverted pendulum system and the feedback controller are set up in a Simulink model. The MATLAB workspace allows for easy access to all variables and numbers of the system, while the Simulink model allows for analyzing the feedback control with a visualization of the system.

The feedback loop for the system, in Figure 22 ,is made similar to the block diagram of the PID controller in Figure 21, except a for a force push input that will simulate an applied force to offset the balance of the system. A model of the inverted pendulum system is also made in Simulink to visualize the model of the system (Figure 23). The model is made from Simulink's physics, joints and shapes to emulate a basic form of the robot.

The MATLAB and Simulink models were given test values for the physical system of of $M = 0.5$kg, $m = 0.2$kg and $I = 0.006$kg·m^2. This allowed for easy testing and tuning of the full self-balancing system for this model. The first test for the feedback system is to test the PID values of the controller to be 1. The results from this test show that the system is unstable without tuning the PID controller. The second test for the feedback system is to test the PID controller, but using only the proportional gain. The results from this test show that the system stabilizes close to the desired point, but it has significant amount of overshoot and a slow settling time. The last test for the feedback system is to test the PID controller after tuning, which now uses a combination of gains. The combination of gains that we found useful is a large proportional gain, a small derivative gain and an integral gain of 1. The results from this test show that the system stabilizes closer to the set point with less overshoot and shorter settling time. The test results for the Simulink model are shown in Figure 24.

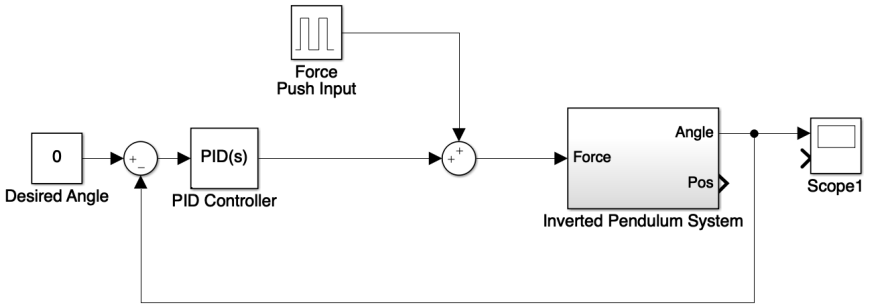


Figure 22: *Simulink model of PID controller and Inverted Pendulum System [6]*

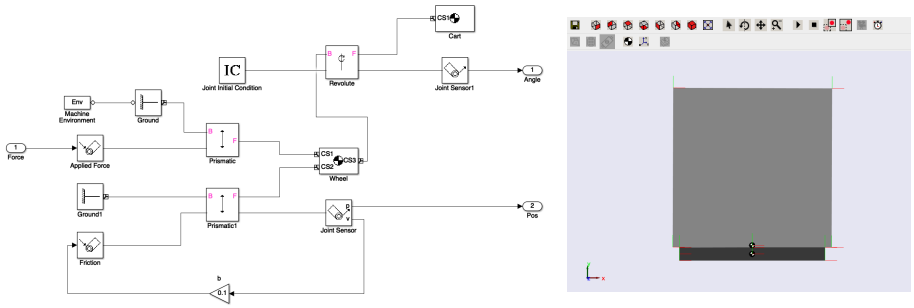
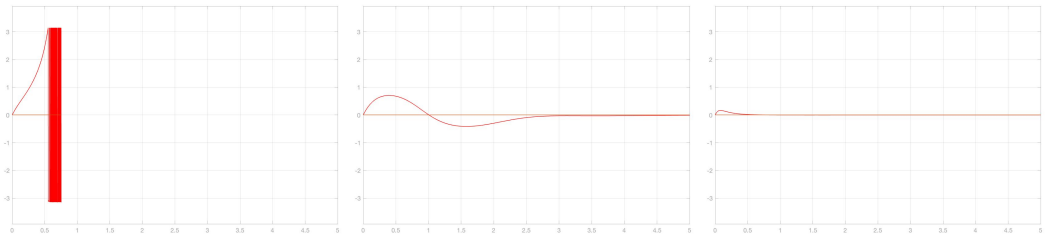


Figure 23: *Simulink model to visualize inverted pendulum system [6]*



(a) *PID controller not active; System is unstable* (b) *Only Proportial gain active; System is now stable* (c) *PID controller tuned; Low overshoot and fast settling time*

Figure 24: *Simulink steady-state response of the angle of the system*

2.7 Computer Vision

The Computer Vision aspect of system was done on the Jetson Nano. When it comes to graphical data, a GPU processes these data sets efficiently. Since the Jetson Nano is a micro controller with a GPU, feeding video and applying object tracking wouldn't be an issue for the Nano.

Programming the Jetson was done via Python and OpenCV. Instead of creating a neural network that was trained to respond to a human, the object tracking was done by following a color that the user was wearing. This process of color tracking is done by creating foreground masking primarily and can be refined with background masking.

When creating any sort of color masking, selecting a color is can done through track bars25. Track bars can be manipulated with a mouse click and drag, and each of these values differs the color wanted and the intensity of color.

Once the track bar values are selected, the program continuously reads one frame at a time and creates a gray scale of the raw frame. The mask is applied and causes the color selected to become white and all other colors to become black - this is called a binary frame 26a. With this binary frame, the Jetson Nano has an easier time to create an exterior region, which is an outline around the highlighted object.

By applying this exterior region, OpenCV calculates the area in the outline, and a specific area can be used to filter out any noise of the same color. For example, if a green leaf started to float next to a user's green jacket, the green jacket would have an area greater than area X while the leaf would be smaller than area X. With this filter, creating confident Region of Interests (ROIs) can be done simply by replacing the exterior region with a contour (highlighted box). Thus demonstrating object tracking is done by applying the contour onto the raw frame 26b.

The second problem is how to communicate the position of the user. A hysteresis and two independent digital out pins have the capability. By creating a hysteresis, a person can be either be in three different positions relevant to the frame. The person can be in the left, right or middle regions of the frame. The two digital pins can be lowered or raised to signify a certain position.

Pin 12	Pin 16	Output/region
0	0	00/missing
0	1	01/right
1	0	10/left
1	1	11/middle

Table 1: *Digital Signal Combinations*

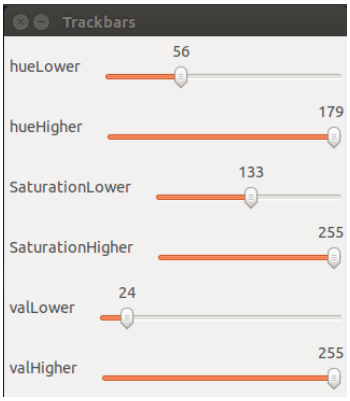
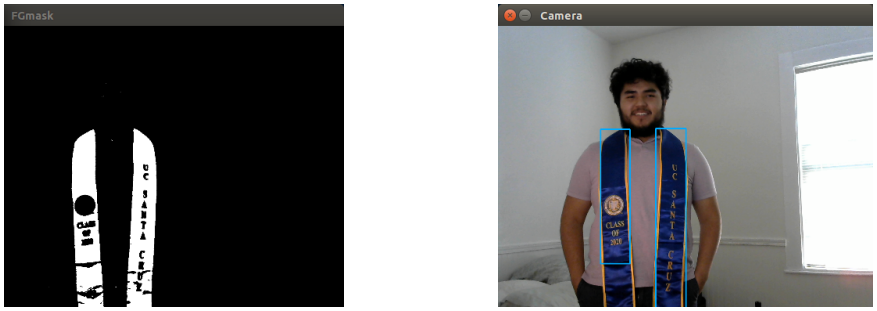


Figure 25: *A method of color selection and intensity[5]*



(a) *A color mask that filters out any color that didn't fit in the selection [5]*

(b) *Use ROI to surround the highlighted from mask and apply to raw frame*

Figure 26: *Computer Vision masking and Final Product[5]*

3 Evaluation

3.1 Functional Prototype

The functional prototype was created through the use of ROS and Gazebo. The Robot Operating System (ROS) is a set of libraries and tools that allows its users to create a system that can simulate a robot through nodes that act as inputs or outputs. Gazebo is the application that allowed the robot to be visualized in addition to simulating noise in the sensors' outputs.

3.1.1 ROS

The programming of the robot was done using Python as well as the rospy library. In order to make our simulation as similar as possible to a real world robot, there is a main node that takes in IMU data as well as computer vision data and outputs the velocity to a linear velocity controller. Figure 27 shows that the CV node outputs the computer vision data and the IMU node publishes its data to the balance main node. This node then outputs to cmd_vel which controls the differential drive controller.

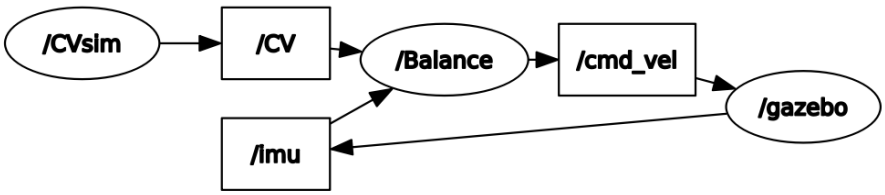


Figure 27: *The full ROS node chart*

To simulate the computer vision sending data to the robot as close as possible, the output of the computer vision was used. First the output data from the Jetson Nano that would be sent to the Raspberry Pi is saved into a .csv file. In ROS, the file is then read and each row is printed on a loop so that the robot will receive the computer data allowing the simulation to run forever without crashing.

3.1.2 Gazebo

Gazebo is the application which allows the robot to be placed and viewed in an empty world as shown in figure 28. Using URDF files, we can control the variables in the environment and for the robot. Gazebo allows the differential drive controller to have a specific torque in order to simulate the torque our real motors would have. We can also control the update rates of the IMU and differential drive controller as well as friction and mass of the wheels and chassis.

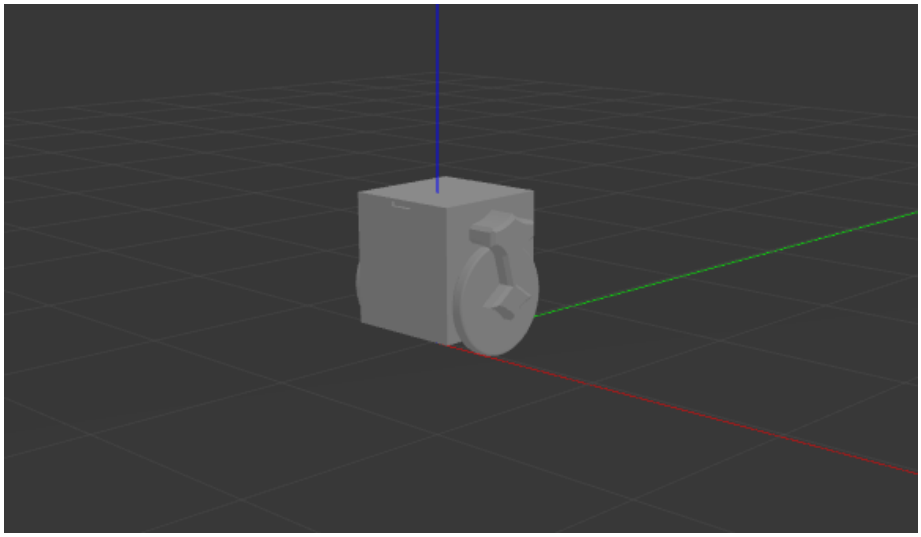


Figure 28: *The robot in the empty simulation world*

The IMU was an in-built plugin through Gazebo that gave us the angular velocity as well as the specific force on the robot, like an IMU. The IMU has a natural noise to it through Gazebo, but it is also possible to add Gaussian noise and change the update rate of the sensor. As we can see from figure 29 the IMU is quite noisy and has a standard deviation of 1.8708. The way that the noise was dealt with was through the methods used in ECE-167 for IMU open loop integration. The idea behind the code is that we can use the accelerometer on the IMU to correct for noise in the gyroscope which would cause the Euler angles on the system to drift away from the current position. We use a feedback controller to perform closed loop integration which deals with the accelerometer noise and gyro noise before using the direction cosine matrix to calculate the Euler angles.

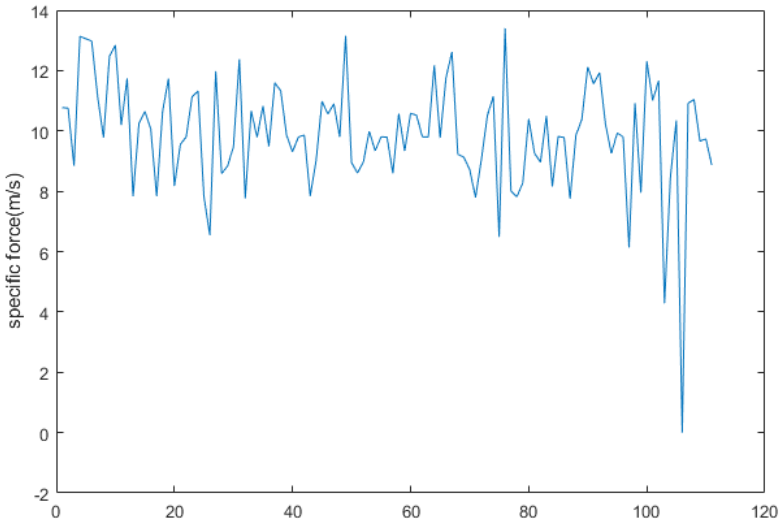


Figure 29: *IMU Z-axis measurements of a still object*

The control of the wheels was done through the differential drive plugin for ROS. This plugin allows us to easily control the linear and angular velocity of the robot. This allows us to take the output of the feedback controller and convert to linear velocity of the wheels as a whole as opposed to manually controlling both joints that connect the wheel to the robot. We can also set the angular velocity which lets us turn the robot easily. Having the differential drive controller control the angular velocity of the wheel also allows us to set the linear and angular velocity so that the robot can turn and balance at the same time without needing to calculate the effect on the individual motors. When the main code for the actual robot is created, the angular and the linear velocities will have to be translated to a PWM output for the actual motors.

3.2 Testing

The testing of the robot in ROS started with testing the balancing on its own. The controller lasted around 20 seconds without turning. As we can see from Figure 30, the controller starts off working well but past a certain point it looks like the controller balances around values other than 0. We deemed that this length for the controller was good enough to test the implementation of the computer vision and turning.

For testing the self-balancing system and tuning the PID controller, we initially used the gain values from the MATLAB and Simulink model of $P = 100, I = 1, D = 20$, but immediately found that the system wasn't stable. With high overshoot, we decreased the values of the controller to $P = 30, I = 1, D = 1$, where the system was stable, but had a drift in its linear velocity. To reduce the linear velocity drift, we applied two different methods: implement a time difference to the PID controller, and implement a second PID controller to control linear velocity [7]. The addition of a time difference improved the accuracy of the output value from the controller, while the second PID controller only complicated the tuning process. Tuning the PID gains was the method that

produced the most successful results. A PID controller with gain values $P = 20, I = 0.1, D = 0.1$ resulted in the most usable and stable system for the prototype.

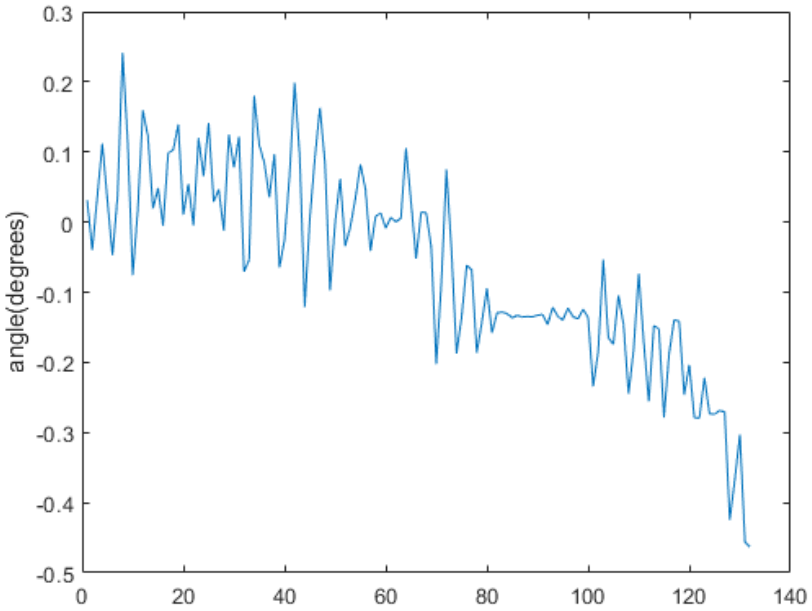


Figure 30: *The angle over time of the ROS robot*

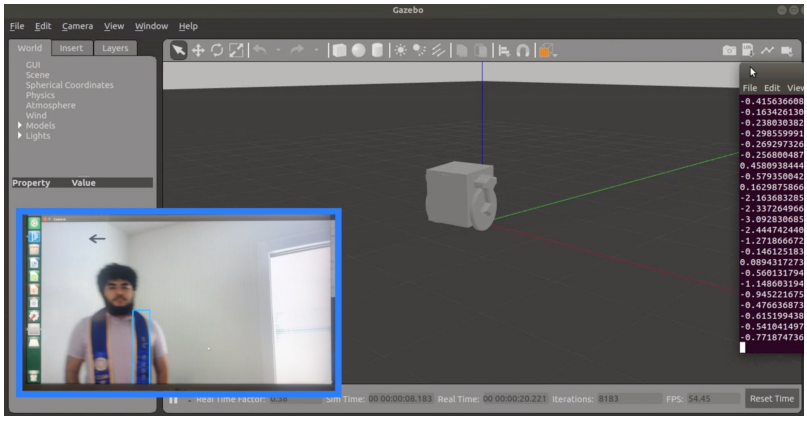


Figure 31: *The simulation of the robot turning according the outputs of computer vision tracking data (Arrows in openCV mark the direction for turning).*

4 Conclusion

The completion of a ROS prototype displays the collaborative design of the different parts that were implemented. The ROS model is able to visualize the CAD design of the physical model. The physical model follows the aesthetic shape and physics of a physical model. Along with the aesthetic model, the ROS model is also able to emulate the sensors and actuators needed to run a physical model. The sensors emulated show that the robot is able to react given a real world input. The actuators emulated show that the robot is able to physically move accordingly.

The ROS model also implements the main software aspects of the project: the state machine control, self-stabilizing, and tracking. The state machine is the main control of the robot. Alongside the state machine, the self-balancing controller is always running to keep the robot stabilized. For tracking, the robot reacts accordingly given inputs from computer vision data. The ROS prototype shows that these software implementations are functional working together.

Shortcomings or flaws in our project are also prevalent in the prototype. The self-balance controller shows some instability working in tandem with the robot turning. Without a double PID implementation for a linear velocity controller, the prototype is more susceptible to instability and velocity drift. With only a non-physical prototype, hardware compatibility and electronic components power usage and voltage intake is difficult to test. Terrain and uneven ground was untested in the prototype.

If we were to continue this project, we would consider these problems. The next step into this project is to build a physical model of the robot with all hardware and software components included. To address self-balancing issues, a linear velocity controller for the Raspberry Pi is required to be built. Other future features we considered included a charging station for the physical system, and applications for users to configure their own robots, such as changing settings on tracking to force the robot to stay in place or follow at varying distances.

Appendix 1 – Problem Formulation

Conceptualization & Brainstorming

Land Robot	Drone	Impractical
2-Wheeled Rover	Quadcopter	Hover Bot
4-Wheeled Rover	RC Helicopter	Sphere-Gyro Bot
Tank-Treads	Winged Drone	4-Legged Robot Dog
Omni-Wheels		2-Legged Robot
Mecanum Wheels		Soft Robotics
3-Wheeled Rover		
RC Car		

Decision Tables

Robot design

Criteria/Scale	Units	Weight	Design 1:			Design 2:			Design 3:		
			Raw	Scaled	Weighted	Raw	Scaled	Weighted	Raw	Scaled	Weighted
Robot:			**Drone**			**2 Wheel Rover**			**4 Wheeled Rover**		
Cost	$	30	4000	2	60	140	9.72	291.6	160.94	9.67812	290.3436
Strength	lbs	30	17	3.4	102	20	4	120	20	4	120
Size (L x H x W)	cm^3	5	102 x 102 x 42	5.63032	28.1516	25 x 30 x 25	9.8125	49.0625	35 x 27 x 21	9.80155	49.00775
Battery Life	hr	20	0.4167	4.167	83.34	1	10	200	1	10	200
Customizability	personal score	5	5	5	25	10	10	50	10	10	50
Durability	personal score	20	2	2	40	5	5	100	5	5	100
Damage Likelihood	probability	10	0.9	1	10	0.5	5	50	0.6	4	40
Noise	personal score	10	2	2	20	7	7	70	7	7	70
Mobility	personal score	50	10	10	500	5	5	250	3	3	150
Safety	personal score	30	2	2	60	8	8	240	8	8	240
Total Score:					**928.4916**			**1420.6625**			**1309.35135**

Primary Microcontroller

Criteria/Scale	Units	Weight	Design 1:			Design 2:			Design 3:			Design 4		
			Raw	Scaled	Weighted	Raw	Scaled	Weighted	Raw	Scaled	Weighted	Raw	Scaled	Weighted
Microcontroller:			**PSoC**			**UNO32**			**Raspberry pi**			**Arduino**		
Cost	$	10	0	10	100	200	2	20	0	10	100	50	8	80
Programmability	personal score	30	8	8	240	5	5	150	9	9	270	7	7	210
Library Available	personal score	50	9	9	450	2	2	100	9	9	450	8	8	400
Comfortability	personal score	50	9	9	450	5	5	250	9	9	450	2	2	100
Total Score:					**1240**			**520**			**1270**			**790**

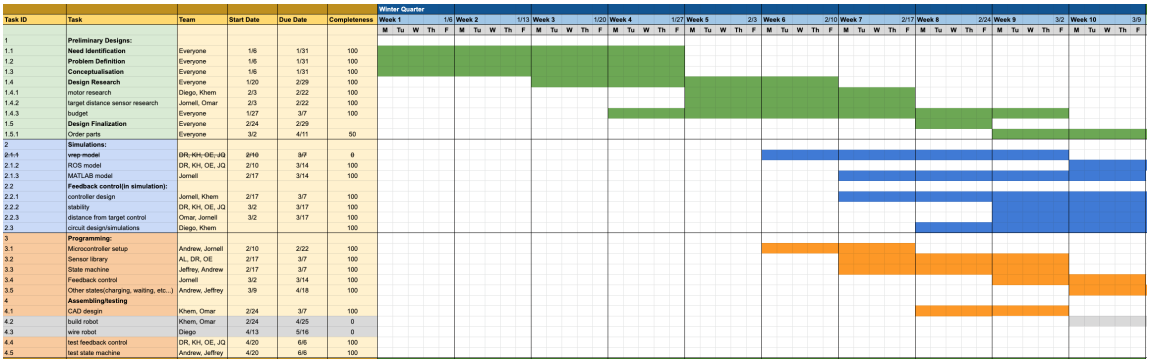
Morphological Chart

Robot Components			Solutions		
Base Shape	Sphere	Square	Hexagon	Octogon	Circular
Compartment Door	Sliding	Hinge	Lid	Open	Hatch
Wheels	2-Wheel	4-Wheel	Tank Treads	Omni	Mecanum

Appendix 2 – Planning

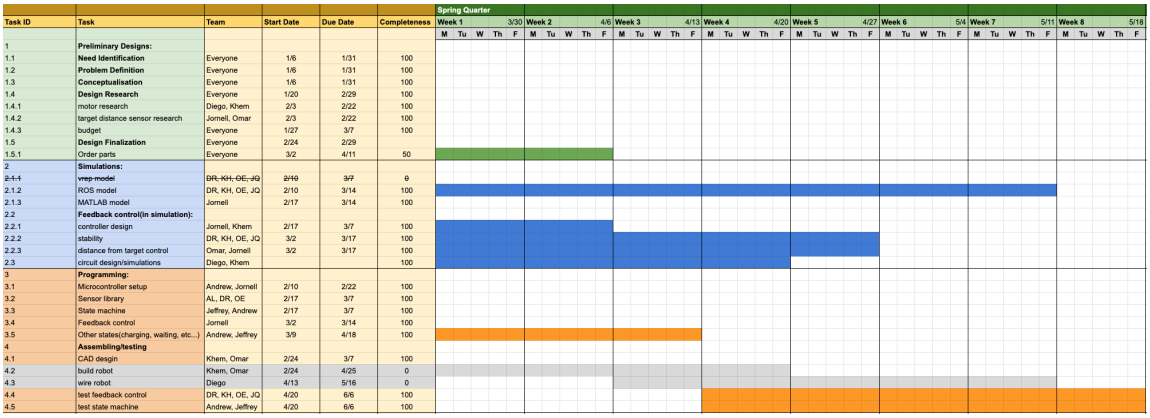
Gantt Chart

Winter Quarter 2020



Task ID	Task	Team	Start Date	Due Date	Completeness
1	**Preliminary Designs:**				
1.1	**Need Identification**	Everyone	1/6	1/31	100
1.2	**Problem Definition**	Everyone	1/6	1/31	100
1.3	**Conceptualisation**	Everyone	1/6	1/31	100
1.4	**Design Research**	Everyone	1/20	2/29	100
1.4.1	motor research	Diego, Khem	2/3	2/22	100
1.4.2	target distance sensor research	Jornell, Omar	2/3	2/22	100
1.4.3	budget	Everyone	1/27	3/7	100
1.5	**Design Finalization**	Everyone	2/24	2/29	
1.5.1	Order parts	Everyone	3/2	4/11	50
2	**Simulations:**				
~~2.1.1~~	~~**vrep model**~~	~~DR, KH, OE, JQ~~	~~2/10~~	~~3/7~~	0
2.1.2	ROS model	DR, KH, OE, JQ	2/10	3/14	100
2.1.3	MATLAB model	Jornell	2/17	3/14	100
2.2	**Feedback control(in simulation):**				
2.2.1	controller design	Jornell, Khem	2/17	3/7	100
2.2.2	stability	DR, KH, OE, JQ	3/2	3/17	100
2.2.3	distance from target control	Omar, Jornell	3/2	3/17	100
2.3	circuit design/simulations	Diego, Khem			100
3	**Programming:**				
3.1	Microcontroller setup	Andrew, Jornell	2/10	2/22	100
3.2	Sensor library	AL, DR, OE	2/17	3/7	100
3.3	State machine	Jeffrey, Andrew	2/17	3/7	100
3.4	Feedback control	Jornell	3/2	3/14	100
3.5	Other states(charging, waiting, etc...)	Andrew, Jeffrey	3/9	4/18	100
4	**Assembling/testing**				
4.1	CAD desgin	Khem, Omar	2/24	3/7	100
4.2	build robot	Khem, Omar	2/24	4/25	0
4.3	wire robot	Diego	4/13	5/16	0
4.4	test feedback control	DR, KH, OE, JQ	4/20	6/6	100
4.5	test state machine	Andrew, Jeffrey	4/20	6/6	100

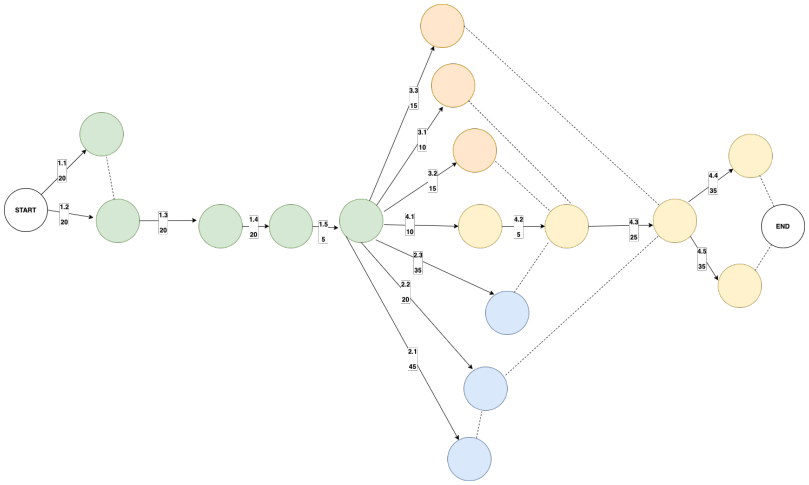
Spring Quarter 2020



Task ID	Task	Team	Start Date	Due Date	Completeness
1	**Preliminary Designs:**				
1.1	**Need Identification**	Everyone	1/6	1/31	100
1.2	**Problem Definition**	Everyone	1/6	1/31	100
1.3	**Conceptualisation**	Everyone	1/6	1/31	100
1.4	**Design Research**	Everyone	1/20	2/29	100
1.4.1	motor research	Diego, Khem	2/3	2/22	100
1.4.2	target distance sensor research	Jornell, Omar	2/3	2/22	100
1.4.3	budget	Everyone	1/27	3/7	100
1.5	**Design Finalization**	Everyone	2/24	2/29	
1.5.1	Order parts	Everyone	3/2	4/11	50
2	**Simulations:**				
~~2.1.1~~	~~**vrep model**~~	~~DR, KH, OE, JQ~~	~~2/10~~	~~3/7~~	0
2.1.2	ROS model	DR, KH, OE, JQ	2/10	3/14	100
2.1.3	MATLAB model	Jornell	2/17	3/14	100
2.2	**Feedback control(in simulation):**				
2.2.1	controller design	Jornell, Khem	2/17	3/7	100
2.2.2	stability	DR, KH, OE, JQ	3/2	3/17	100
2.2.3	distance from target control	Omar, Jornell	3/2	3/17	100
2.3	circuit design/simulations	Diego, Khem			100
3	**Programming:**				
3.1	Microcontroller setup	Andrew, Jornell	2/10	2/22	100
3.2	Sensor library	AL, DR, OE	2/17	3/7	100
3.3	State machine	Jeffrey, Andrew	2/17	3/7	100
3.4	Feedback control	Jornell	3/2	3/14	100
3.5	Other states(charging, waiting, etc...)	Andrew, Jeffrey	3/9	4/18	100
4	**Assembling/testing**				
4.1	CAD desgin	Khem, Omar	2/24	3/7	100
4.2	build robot	Khem, Omar	2/24	4/25	0
4.3	wire robot	Diego	4/13	5/16	0
4.4	test feedback control	DR, KH, OE, JQ	4/20	6/6	100
4.5	test state machine	Andrew, Jeffrey	4/20	6/6	100

PERT & CPM Analysis

1.1	**Need Identification**
1.2	**Problem Definition**
1.3	**Conceptualisation**
1.4	**Design Research**
1.5	**Design Finalization**
2.1	**ROS model simulation**
2.2	**Feedback control (in simulation)**
2.3	circuit design/simulations
3.1	Microcontroller setup
3.2	Sensor library
3.3	State machine
4.1	CAD design
4.2	build robot
4.3	wire robot
4.4	test feedback control (physical)
4.5	test state machine (physical)



Division of Labor

	Omar	Khem	Andrew	Jeffrey	Jornell	Diego
Preliminary	Brainstorming	Brainstorming	Brainstorming	Brainstorming	Brainstorming	Brainstorming
	Needs/Goals	Needs/Goals	Needs/Goals	Needs/Goals	Needs/Goals	Needs/Goals
	Design Research	Design Research	Design Research	Design Research	Design Research	Design Research
Physical Design	Sensors Research	Sensors Research	Sensors Research	Sensors Research	Sensors Research	Sensors Research
	Microcontroller Research	Microcontroller Research	Microcontroller Research	Microcontroller Research	Microcontroller Research	Jetson Nano Research
	Material Research	Material Research			Wiring Diagram	Camera Research
	Design Drawing	Motors Research				
	CAD design					
Software Design		Sensor Testing	State Machine Diagram	State Machine Diagram	Self-Balancing Research	Sensor Testing
			State Machine Code	State Machine Code	PID Testing MATLAB	OpenCV
			State Machine Testing	State Machine Testing	PID Testing Simulink	OpenCV Hystersis
			Sensor Testing	Sensor Testing	PID Code Python	OpenCV Testing
			OpenCV Hystersis			
Prototyping	ROS Tutorial	ROS Tutorial		ROS Tutorial	ROS Tutorial	OpenCV CSV Output
		ROS Model			ROS Self-Balancing	
		ROS Physics				
		ROS Self-Balancing				
		ROS CV Turning				
		Gazebo Simulation				

Collaboration

Tasks were assigned by comparing our relative skills and experiences and placing teammates in roles that they were best suited. We used GitHub and Google Drive to share and collaborate on files. The GitHub stored the software and code. The Google Drive stored documents, designs, diagrams, and videos.

Appendix 3 – Review

Omar Escareno

I was tasked with tackling the design implementation of the robot and making sure that the minimum specifications were being met. Unfortunately, we were unable to build the physical prototype so the majority of my work went towards creating a functional, accurate CAD model to be used within our simulations. The design objectives we had set out for our system were of upmost importance when sketching, creating, and assembling the two-wheeled rover. Because of this, I made sure that all parts were created and constrained appropriately to ensure that our test results would be accurate. Physical fabrication of the robot would have been the final step in the design process and I would have made adjustments based on real life tests.

Khem Holden

I was in charge of the ROS and Gazebo simulation. The ROS went well, however I feel like my lack of knowledge with feedback controllers hindered the progress of ROS because I struggled with tuning the controller. The actual ROS implementation was good and all the parts worked except for the main controller. If I were to make a change I would try and control the linear velocity of the wheels individually to represent the motors as opposed to using a differential drive controller that does it automatically.

Andrew Lei

I mainly worked on the state machine and ping sensor interfacing. I also assisted with the computer vision aspect. The state machine went well, but at times progress was hindered due to the uncertainty of the sensors' outputs. Lack of physical testing definitely impaired this. Because we could not build a physical prototype, it was hard to thoroughly test the robot's behavior and observe how its sensor readings and movement would affect one another. This problem was partially remedied by the test bench, which we used to get see the events triggered and state transitions according to sensor values.

Jeffrey Ng

I assisted with the development and planning of the primary state machine and its integration into the system. This role matched well with my field of expertise, as I am quite confident in my logic design skills, whereas my teammates were more familiar with robotics and electronics. In addition to this project, I have been involved with several engineering-based group projects and, as such, I am largely pleased with what we accomplished during this time. While it is true that real-world circumstances prevented us from making as much progress as we would have liked, such obstacles are just some of many that can happen during the engineering process. Considering our circumstances, I believe we made good progress and have a clear understanding of what still needs to be done should we continue our work.

Jornell Quiambao

I was the project manager of the group. I organized the meetings, job distribution, and documentation of designs and decisions. I was also in charge of the control design for self-balancing. I wrote the code and simulation for the self-balancing in MATLAB/Simulink and Python. I also helped with tuning the PID controller for self-balancing on the ROS model. The self-balancing PID implementation worked well, except small flaws with the balancing in the ROS and gazebo environment, which were difficult to tune or debug. The group did their respective parts well. If I were to repeat this project, I would change our controller implementation from PID to LQR to make the stabilizing more robust.

Diego Reyes

I was in charge of the computer vision aspect of the robot. Computer vision went really despite having limited restrictions of the testing equipment. The hardest part of the computer vision was the research and finding the most suitable way of following a human. As a group we decided to follow a user via their clothes color. This made the task of object tracking a reasonable amount of work. Figuring out how to color mask and manipulate live video was great via Python3 and OpenCV. Overall I think the computer vision went really well. I would really like to implement an app, so that user could select any color.

References

[1] Arduino. *HC-SR04 Ultrasonic Sensor*. URL: https://create.arduino.cc/projecthub/Nicholas_N/distance-measurement-with-an-ultrasonic-sensor-hy-srf05-64554e.

[2] Draw.io. *Diagrams*. URL: https://www.draw.io.

[3] Geartisan. *DC 12V 30RPM Gear Motor*. URL: https://www.amazon.com/Greartisan-Electric-Reduction-Centric-Diameter/dp/B071KFT4P7.

[4] Logitech. *C270 Webcam*. URL: https://www.amazon.com/Logitech-Desktop-Widescreen-Calling-Recording/dp/B004FHO5Y6.

[5] Paul McWhorter. *AI on the Jetson AI*. URL: http://toptechboy.com/.

[6] University of Michigan. *Control Tutorials for MATLAB Simulink*. URL: http://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling (visited on 02/20/2020).

[7] Salatiel Garcia1 Miguel Velazquez David Cruz and Manuel Bandala. "Velocity and Motion Control of a Self-balancing Vehicle based on a Cascade Control Strategy". In: *International Journal of Advanced Robotic Systems* (2016).

[8] NVIDIA. *Jetson Nano*. URL: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[9] Raspberry Pi. *Raspberry Pi 3 B+*. URL: https://www.distrelec.biz/en/raspberry-pi-model-1gb-ram-raspberry-pi-raspberry-pi-3b/p/30109158.

[10] Qunqi. *L298N Motor Drive Controller H-Bridge*. URL: https://www.amazon.com/Qunqi-Controller-Module-Stepper-Arduino/dp/B014KMHSW6/ref=sr_1_8?dchild=1&keywords=h+bridge&qid=1591300748&sr=8-8.

[11] Sparkfun. *MPU9250 IMU*. URL: https://learn.sparkfun.com/tutorials/mpu-9250-hookup-guide/all.