6.1311 Final Project Report - Self Balancing Longboard with Dual Brushless Drive

Fischer Moseley

December 11, 2019

1 Motivation

In 2014, Future Motion Incorporated released the Onewheel, a single-wheeled skateboard with a self-contained battery, inverter and motor designed for personal transport. These have gained popularity on college campuses as a result of their high speed and small size. Several can be seen ferrying students around MIT's campus, and their unique portability and maneuverability provides a uniquely interesting vehicular platform.

However, this method of locomotion presents two unique engineering challenges: brushless motor control, and inverted pendulum dynamics. Most devices in this category use brushless permanent magnet motors in their drivetrains, due to their higher power density, torque, and efficiency over brushed DC motors. These perks come at the expense of an inverter, which commutates the internal magnetic field, as opposed to the mechanical commutation performed by brushed DC motors. Such an inverter is controlled by an algorithm using IMU data to keep the longboard (and its rider) upright.

2 General Description

To accomplish this, a system containing a mechanical assembly, motor drive electronics, control electronics, and a control algorithm was fabricated. The mechanical assembly consists of a pair of wooden footpanels supported by a welded steel tube frame, upon which the motor mounts and the rider stands. This frame also houses the motor controller, consisting of drive electronics that use MOSFETs to drive motor phase currents, and control electronics that perform the real-time motor commutation and run the position control algorithm, outputting data to the drive electronics.

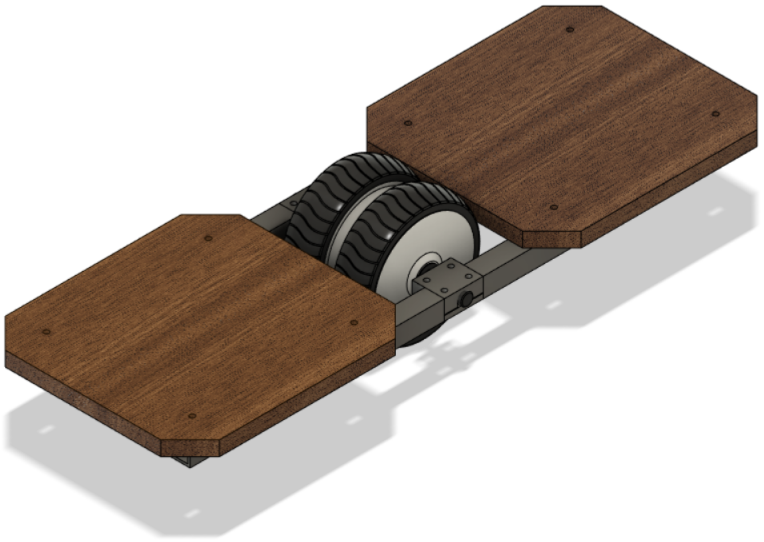
This system allows for the user to control their locomotion by gently leaning along the longitudinal axis, and the control algorithm will drive the motors such that the user's fall is prevented while the board translates in the direction the user leans. The user is responsible for stabilizing themselves in the transverse axis as the longboard does not exhibit a degree of freedom in that axis. This is a shortcoming of both this project and the Onewheel, but the skill required is minimal and not unreasonable to expect of the average rider.

Commercially available Onewheels use a single motor integrated into the wheel hub, mounted in the center of the board. It was decided to slightly depart from this design and use two center-mounted motors, such that the device would be able to turn in place instead of requiring the user to lean while moving in order to turn.

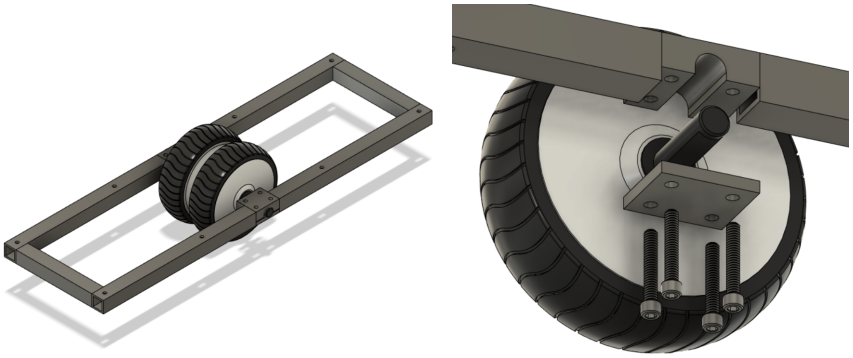
3 Mechanical Assembly

The core of the mechanical assembly is a welded steel tube frame made from 1" square tubing. The frame also contains two steel brackets that mount the hub motors. These were milled to approximate size before being welded, and then milled to final dimension after being welded to ensure that the left and right motors were concentric. Two small steel plates were cut out of 1/4" steel stock on the waterjet, and these clamp the motors into the brackets. All of the steel parts were sandblasted after welding and wire brushed to produce a pleasing surface finish, and then coated with a spray-on clear coat to protect against corrosion.

On top of this rest two wooden foot pads, which were made on a CNC router and then stained and clear coated before being bolted onto the undercarriage. The drive electronics mount to the bottom of the footpad inside the frame, but for the purposes of debugging and demonstration the motor controller was left on the top of the footpad. The final CAD model is shown below:



The steel undercarriage is also shown below, and the motor mounts (expanded) have the following geometry:



4 Drive Electronics

To drive the pair of permanent magnet brushless motors, the three-phase motor controller circuitry that was presented in class was modified and reused. Each motor requires one of the totem boards, each of which contains six half-bridges. Each half bridge consists of a IRF740 N-channel MOSFET and its associated IR2125 gate driver, resulting in a total of twelve MOSFETs and gate drivers across the entire system. A series of LEDs were added to the gate driver enable pins, allowing the user to see which MOSFETs were being activated as the motor rotates, providing the user with a visual interface to the motor commutation logic. Another set of LEDs was connected to the A, B, and C phase outputs which illuminated when that phase's high side MOSFET was active, which allowed for system validation and debugging without a motor connected.

Each of the gate driver enable signals was routed to the ribbon cable, which connects the control board to each of the totem boards. The high-current VDD and VSS connections are connected to a 36V 300W power supply via a pair of ring lugs.

Although the rotor speed sets the commutation speed of the motor controller, the high side FETs switch at the frequency on the PWM input on the PSoC, which is defined by the Teensy. The Teensy supports a relatively arbitrary PWM frequency, meaning that it can be chosen to minimize switching losses. Ideally, the PWM frequency would be as low as possible to reduce losses while also being both higher than the motor commutation frequency and high enough to provide the gate driver's bootstrap capacitor enough time to

recharge such that it never runs out of voltage. With the component values recommended for the totem boards given in class, the IRF2125s need their bootstrap capacitors recharged approximately once every millisecond, setting an absolute lower limit on the gate drive frequency of 1kHz. However, to mitigate coil whine the frequency should be above 20kHz, and this frequency satisfies the previous two conditions. Therefore, the switching losses associated with this frequency can be calculated thusly:

$$P_{sw} = f_{sw} \cdot \left(E_{diss(on)} + E_{diss(off)}\right) \approx 2 \cdot f_{sw} \cdot E_{diss}$$

Where E_{diss} is the energy delivered to the MOSFET gate, and is given by:

$$E_{diss} = \frac{1}{2} \cdot V_{dd} \cdot I_d \cdot \Delta t \approx \frac{1}{2} \cdot V_{dd} \cdot I_d \cdot \frac{\Delta Q_{gate}}{I}$$

Solving for E_{diss} and substituting values:

$$P_{sw} \approx f_{sw} \cdot V_{dd} \cdot I_d \cdot \frac{\Delta Q_{gate}}{I_{drive}} = \left(20kHz \cdot 30V \cdot 5A \cdot \frac{200nC}{1A}\right) = 600mW$$

It's also worth noting that the maximum PWM duty cycle cannot be 100%, as this would produce a situation in which the high side gate driver bootstrap capacitor cannot recharge if the motor is stalled and the PWM duty cycle is at 100%.

The conduction losses are also calculated below for the IRF3415. This analysis assumes the worst case operating temperature of 100°C, and an associated $R_{ds(on)}$ of 71.4mΩ. Since the motor will require the most current at zero speed, we know that the worst possible conduction condition is:

$$P_{worst} = I_{max}^2 \cdot R_{ds(on)} = (5A)^2 \cdot 0.0714\Omega = 1.79W$$

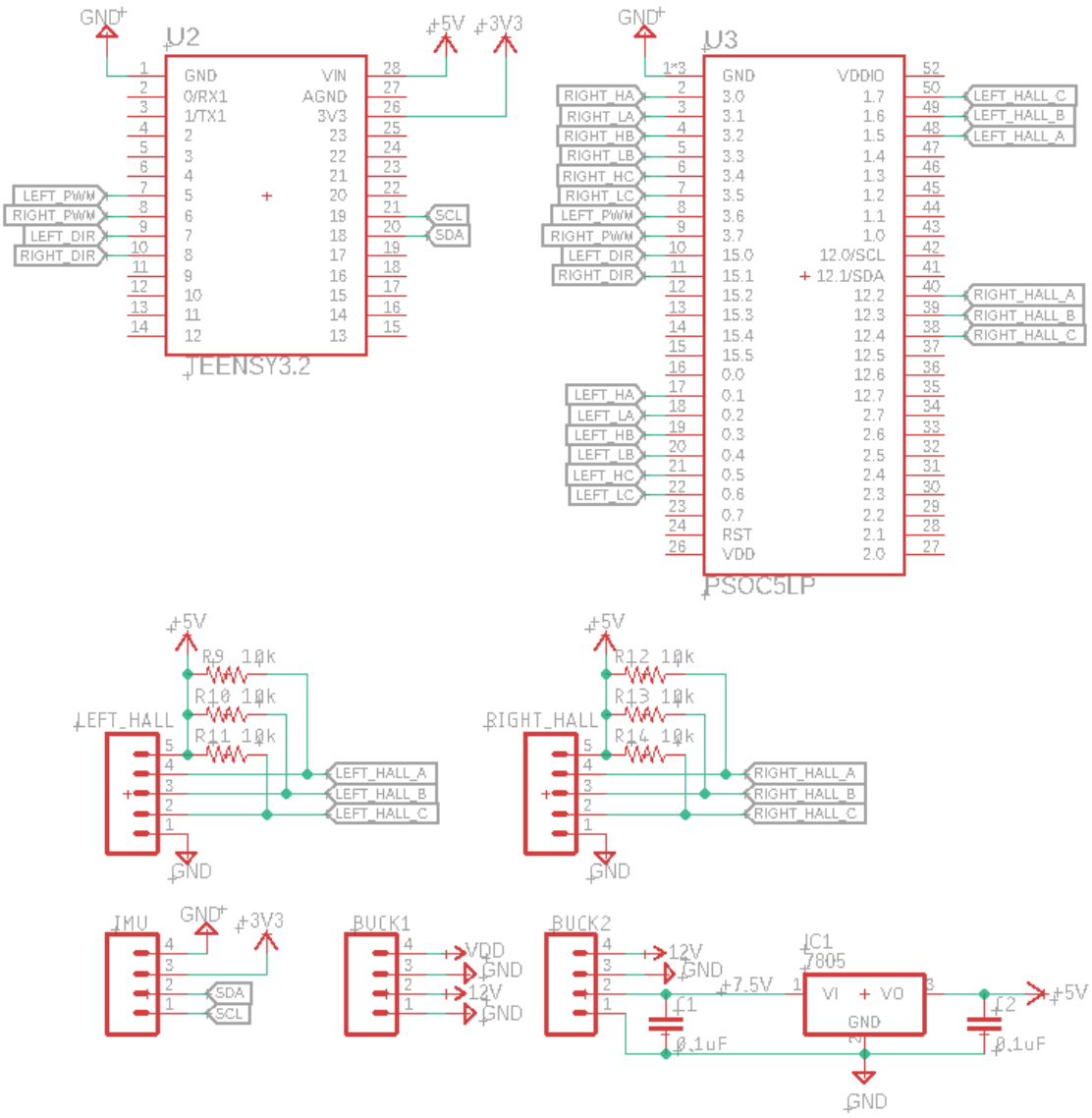
Finally, the gate driver losses are computed, which are given by the expression:

$$P_{gate} = Q_g \cdot V_g \cdot f_{sw} = 200nC \cdot 12V \cdot 20kHz = 48mW$$

The total loss per MOSFET is equal to the sum of these losses, or 2.44W. The datasheet specifies a junction-to-ambient thermal resistance of 62 K/W, which would produce a package temperature of 176.3°C in a 25°C environment. This assumes the worst possible conditions, where the motor is in a stall condition and the FETs are being driven at a high duty cycle.

5 Control Electronics

While the drive electronics are responsible for controlling phase current, the control electronics are responsible for motor commutation and control. The control circuitry consists of a Teensy 3.2 microcontroller that communicates with the IMU and runs the control algorithm, as well as a PSoC 5LP that reads the state of the Hall Effect sensors built into the motor, and performs real-time motor commutation in response to the rotor position. The gate drive signals are then sent over the ribbon connector to the totem boards, which provide the actual phase current. The overall schematic is shown here, omitting the FETs and gate drivers on the totem boards.

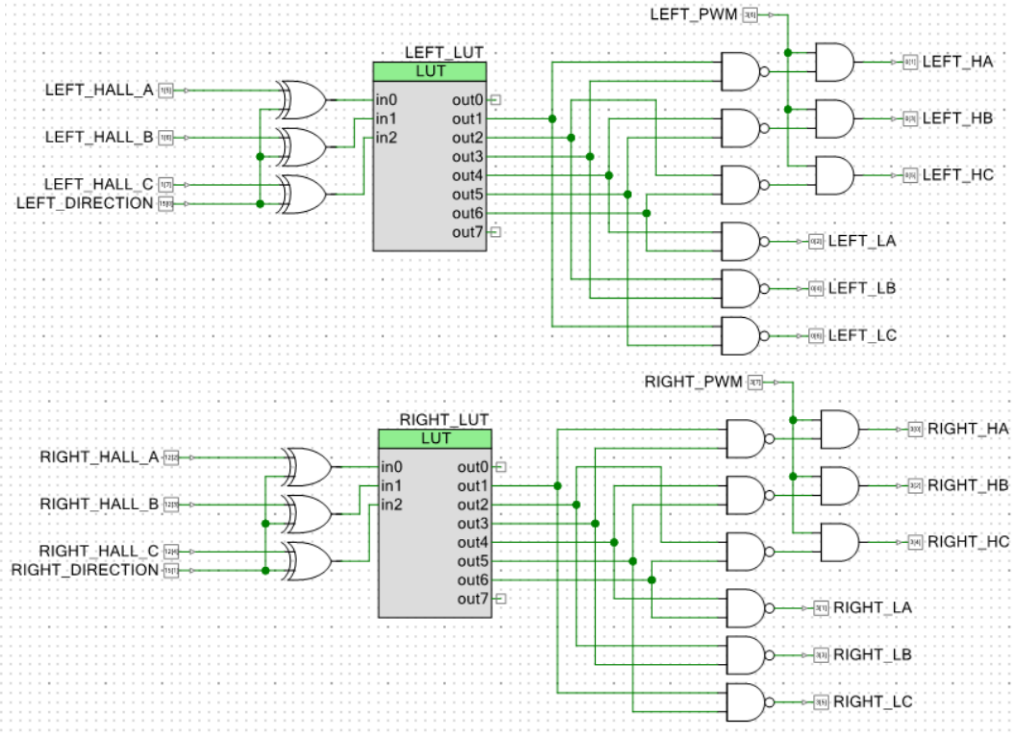


The advantage of this two-processor setup is twofold. Because the motor commutation logic is entirely in the FPGA fabric, no microcontroller interrupts are required to commutate the motors. Therefore, there is no time delay associated with a microcontroller entering an ISR, polling the I/O pins corresponding to the hall effect sensors and control inputs, and writing the correct FET drive signals to I/O. This time delay is on the order of microseconds, but by doing it in real time it ensures that there's no time where the motor controller is outputting the improper FET drive signals, ensuring maximum torque, even at high speeds where the motor phase current switches often.

The other advantage is that since the commutation state needs to be updated whenever

any of the hall effect sensors change state, the I/O pins connected to the hall effect sensors would all need to be interrupt-enabled. Although the Teensy 3.2 allows for interrupts on any I/O pin, if this design was implemented with another microcontroller (such as an ATMega328P found on Arduino Uno or Nano) there may be only a finite number of I/O pins capable of triggering interrupts.

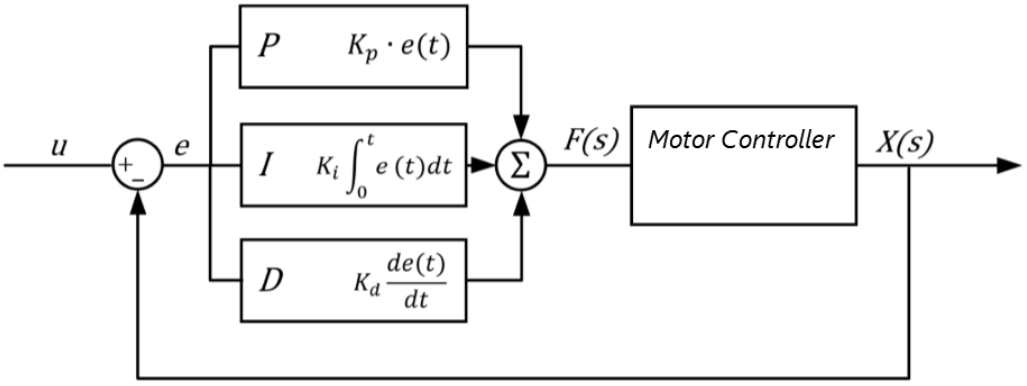
However, as a result of using a two-processor architecture, there is some interface required between to communicate the control signals from the Teensy to the PSoC. This interface uses two signals, one for the intended direction of the motor, and one for the PWM signal that sets the average voltage across the motor winding. A low logic level on the direction pin corresponds to a forward rotation on the motor, and a high logic level represents a reverse rotation. Likewise for the PWM pin, a duty cycle of 0% corresponds to an average phase voltage of 0% of the input voltage, and a duty cycle of 100% will put the entire input voltage on each motor phase. Each motor has both a PWM and a direction output going from the Teensy to the PSoC, and by modulating these outputs in response to the IMU's rotation vector the Teensy can keep the rider upright. The final configuration of the PSoC's FPGA fabric is as shown below:



The Teensy gets its rotation vector from a BNO080 IMU that outputs its rotation vector in quaternion format over I^2C every 10ms. Upon receiving the vector, the Teensy converts it to euler angles (pitch, yaw, roll) and then feeds it to the PID algorithim.

6 Control Algorithm

The skateboard uses an implementation of the PID algorithim to prevent the user from falling over, the diagram of which is shown below:



The control algorithm takes in a pitch angle and a setpoint and returns a single output - the duty cycle of the PWM on the high side FETs. For its simplicity and ease of implementation, a PID controller was used and implemented in software. A LQR controller was also considered, but four independent parameters would have been necessary to determine the weights on the state space variables $\left(x, \dot{x}, \theta, \dot{\theta}\right)$. Although deriving a K matrix from the weights is simple, tuning the weights is not a straightforward problem and becomes a four-dimensional tuning problem instead of a three-dimensional tuning problem in the case of PID.

It is also worth noting that in class a nested control loop topology was proposed where a secondary proportional controller was used to discipline the speed of the motor. Here, a primary controller holds a positional setpoint by adjusting motor speed, which itself is disciplined by the secondary loop. The quicker performance this system offers made it tempting to implement for this project, but it was ultimately decided that it would be unnecessary in this case.

This decision motivated by noticing that the motors in this application operate at relatively low speed, such that the back EMF generated can be approximated as zero, and therefore the average current through the motor is roughly equal to $V_{dd} * D/R_{phase}$, where D is the duty cycle of the PWM control signal. Since the average current is known, the average torque can be arrived at with the motor equation, so therefore:

$$\tau = I \cdot \kappa \approx \frac{V_{dd} \cdot D \cdot \kappa}{R_{phase}}$$

Thus at low speeds, the motor behaves as an approximately linear duty-cycle-controlled torque source. The second nested control loop existed to re-introduce this linearity, but for small wheel speeds, it already exists.

The PID constants were found by setting all constants to zero, and slowly increasing the K_p term until the system became oscillatory, and then slightly reducing K_p while bringing up the K_I term, and then finally adding a small amount of K_d until the system behaves as expected.

7 Appendix I: Teensy Control Code

```
#include <PID_v1.h>
#include <Wire.h>
#include "SparkFun_BNO080_Arduino_Library.h"
#include <math.h>
#define FORWARD 0 //the logic level on the direction pin that makes the motor go forwards
#define REVERSE 1 //the logic level on the direction pin that makes the motor go backwards

/*---------------- State Variables -----------------*/
struct eulerAngle{ double yaw = 0; double pitch = 0; double roll = 0; };
eulerAngle rotationVector;
eulerAngle gyroRate;
double pitchOffset = 7.51;


struct stateVector { double left_motor; double right_motor; };
stateVector controlVector;

struct Motor {
  //configuration
  uint8_t stopThresholdTime = 12 ; //the amount of time (in ms) that has to pass where the hall effect sensors don't change to be considered "stopped"
  const int16_t MIN_MOTOR_OUTPUT = -255;
  const uint8_t MAX_MOTOR_OUTPUT = 255;

  //IO pin definitions
  uint8_t HALL_A_PIN = 0;
  uint8_t HALL_B_PIN = 0;
  uint8_t HALL_C_PIN = 0;
  uint8_t PWM_PIN = 0;
  uint8_t DIR_PIN = 0;

  //state variables
  int16_t current_pwm = 0;
  uint8_t lastHallState = 0;
  unsigned long lastHallChange = millis(); //timestamp (in MCU time) of the last time that the hall effect sensor's state changed

  int16_t pwm = 0; // the duty cycle of the PWM on the high side motor FETS, from -255 (full reverse) to 255 (full forwards)

  //set internal variables and configure GPIO
  void init(uint8_t Hall_A, uint8_t Hall_B, uint8_t Hall_C, uint8_t PWM, uint8_t Dir) {
    //set internal variables with pin definitions
```

```
  HALL_A_PIN = Hall_A;
  HALL_B_PIN = Hall_B;
  HALL_C_PIN = Hall_C;

  PWM_PIN = PWM;
  DIR_PIN = Dir;

  //configure IO
  pinMode(HALL_A_PIN, INPUT);
  pinMode(HALL_B_PIN, INPUT);
  pinMode(HALL_C_PIN, INPUT);

  pinMode(PWM_PIN, OUTPUT);
  pinMode(DIR_PIN, OUTPUT);

  //for some reason, the PSoC is weird and won't self-start unless
  ↪  there's some I/O change
  //I think this is because the lookup table doesn't always reinitialize
  ↪  to some known value,
  //so by pulsing the direction pin at startup, we can make sure the
  ↪  lookup table is in a known config.

  analogWrite(PWM_PIN, 0); //make sure the motor speed is off
  digitalWrite(DIR_PIN, FORWARD); //pulse the direction pin
  delay(100);
  digitalWrite(DIR_PIN, REVERSE);
  delay(100);
  digitalWrite(DIR_PIN, FORWARD);
}

//returns true if the motor's hall effect sensors haven't changed in the
↪  last 10ms, false otherwise
bool isStopped() {
  //get the current state of the hall effect array
  uint8_t currentState = 0x00;
  bitWrite(currentState, 2, digitalRead(HALL_A_PIN));
  bitWrite(currentState, 1, digitalRead(HALL_B_PIN));
  bitWrite(currentState, 0, digitalRead(HALL_C_PIN));


  if(currentState == lastHallState) { //the motor hasn't changed
  ↪  position, so check if enough time has passed
    if(millis() - lastHallChange > stopThresholdTime) {
      //the motor is stopped because it's halls haven't changed in
      ↪  stopThresholdTime (~10ms)
      //leave both lastHallState and lastHallChange alone, because we
      ↪  have no reason to change them.
      return true;
    }
```

```
    //the motor could be stopped, but we need to wait longer
    //leave both lastHallState and lastHallChange alone, because we'll
    ↪  want to check those later
    //return false, just to be safe
    return false;
  }

  else {
    //the hall positions have changed, so the motor must be turning
    //the last state of the hall effect sensors is no longer valid, so
    ↪  update both it and when it was last changed
    lastHallState = currentState;
    lastHallChange = millis();
    return false;
  }
}

//updates IO, pushes motor state
bool update(double new_pwm){
  //cast the left and right motor controls to int16_t, as they are
  ↪  stored as doubles (required by PID library :)
  int16_t pwm = static_cast<int16_t>(new_pwm);

  //if the new PWM command is outside the allowable range, don't do
  ↪  anything and return with error
  if(pwm > MAX_MOTOR_OUTPUT || pwm < MIN_MOTOR_OUTPUT) return 1;

  //we need to pulse the direction pins when we start up
  if(current_pwm == 0 && pwm != 0) {
    //twiddle direction pins
    digitalWrite(DIR_PIN, FORWARD);
    delay(5);
    digitalWrite(DIR_PIN, REVERSE);
    delay(5);
  }

  //pulse the direction pins when we change directions, but make sure to
  ↪  run through zero speed first
  if( (current_pwm < 0 && pwm > 0) || (current_pwm > 0 && pwm < 0) ){
    //twiddle direction pins
    analogWrite(PWM_PIN, 0);
    digitalWrite(DIR_PIN, FORWARD);
    delay(5);
    digitalWrite(DIR_PIN, REVERSE);
    delay(5);
  }

  if(pwm < 0) digitalWrite(DIR_PIN, REVERSE);
```

```
    if(pwm > 0) digitalWrite(DIR_PIN, FORWARD);

    analogWrite(PWM_PIN, abs(pwm));
    current_pwm = pwm;
    return 0;
  }
};


/*------------- Hardware Configuration ----------- */
//IO pins to use for interfacing with the PSoC controller
#define LEFT_MOTOR_PWM 5
#define RIGHT_MOTOR_PWM 6
#define LEFT_MOTOR_DIR 7
#define RIGHT_MOTOR_DIR 8

//IO pins to use for interfacing with the hall effect sensors
#define LEFT_HALL_A 2
#define LEFT_HALL_B 3
#define LEFT_HALL_C 4
#define RIGHT_HALL_A 10
#define RIGHT_HALL_B 11
#define RIGHT_HALL_C 12

//Motor configuration
Motor leftMotor, rightMotor;

//IMU configuration
BNO080 IMU;

struct PIDconfigContainer {
  double yawSetpoint = 0;
  double pitchSetpoint = 0;
  double rollSetpoint = 90;
  double K_p = 15;
  double K_i = 2;
  double K_d = 0.3;
} PIDconfig;

//Specify links and initial tuning parameters
PID leftPID(&rotationVector.pitch, &controlVector.left_motor,
↪  &PIDconfig.pitchSetpoint, PIDconfig.K_p, PIDconfig.K_i, PIDconfig.K_d,
↪  REVERSE);
PID rightPID(&rotationVector.pitch, &controlVector.right_motor,
↪  &PIDconfig.pitchSetpoint, PIDconfig.K_p, PIDconfig.K_i, PIDconfig.K_d,
↪  REVERSE);

/*------------- Wrapper Functions ----------- */
void updateRotationVector(struct eulerAngle* rotationVector){
```

```
  if(IMU.dataAvailable()){
    float i = IMU.getQuatI();
    float j = IMU.getQuatJ();
    float k = IMU.getQuatK();
    float r = IMU.getQuatReal();
    float x; float y;

    /* YAW */
    x = 2 * ((i * j) + (r * k));
    y = sq(r) - sq(k) - sq(j) + sq(i);
    rotationVector->yaw = degrees(atan2(y, x));

    /* PITCH */
    rotationVector->pitch = degrees(asin(-2 * (i * k - j * r))) -
    ↪  pitchOffset;

    /* ROLL */
    x = 2 * ((j * k) + (i * r));
    y = sq(r) + sq(k) - sq(j) - sq(i);
    rotationVector->roll = degrees(atan2(y , x));
  }
}

void updateGyroRate(struct eulerAngle* angleVariable){
  if(IMU.dataAvailable()){
    angleVariable->yaw    =  IMU.getGyroX();
    angleVariable->pitch  =  IMU.getGyroY();
    angleVariable->roll   =  IMU.getGyroZ();
  }
}

void printStateVector(struct stateVector* controlVector){
  //prints to the serial port the current contents of controlVector.

  Serial.print("ControlVector: left_motor: ");
  Serial.print(controlVector->left_motor);
  Serial.print("  right_motor: ");
  Serial.print(controlVector->right_motor);
  Serial.println();
}

void setup() {
  //start up all them interfaces
  Serial.begin(115200);
  Wire.begin();
  Wire.setClock(400000);
  IMU.begin();
  IMU.enableRotationVector(10); //enable the rotation vector to be updated
  ↪  every 10ms
```

```
  //configure the motor GPIO
  leftMotor.init(LEFT_HALL_A, LEFT_HALL_B, LEFT_HALL_C, LEFT_MOTOR_PWM,
  ↪  LEFT_MOTOR_DIR);
  rightMotor.init(RIGHT_HALL_A, RIGHT_HALL_B, RIGHT_HALL_C,
  ↪  RIGHT_MOTOR_PWM, RIGHT_MOTOR_DIR);

  //turn on PID
  leftPID.SetOutputLimits(leftMotor.MIN_MOTOR_OUTPUT,
  ↪  leftMotor.MAX_MOTOR_OUTPUT); //set the output limits of the PID
  ↪  controllers
  rightPID.SetOutputLimits(rightMotor.MIN_MOTOR_OUTPUT,
  ↪  rightMotor.MAX_MOTOR_OUTPUT);
  leftPID.SetMode(AUTOMATIC);
  rightPID.SetMode(AUTOMATIC);
}

void loop() {
  updateRotationVector(&rotationVector); //get new rotation vector data
  leftPID.Compute(); //recompute motor PIDs
  rightPID.Compute();
  leftMotor.update(controlVector.left_motor);
  rightMotor.update(controlVector.right_motor);
  Serial.print("pitch: ");
  Serial.print(rotationVector.pitch);
  Serial.print("    pitchOffset: ");
  Serial.println(pitchOffset);
  printStateVector(&controlVector);
}
```