Feedback Control

Inverted Pendulum
BaeBot
Balancing Automatic Energetic Bot

Phiet Do
Luis Perea
Alexander Matthews
Aneek James
Jay Patel

Contents

Milestone 1

Team Members

Phiet Do, Luis Perea, Alexander Matthews, Aneek James, Jay Patel

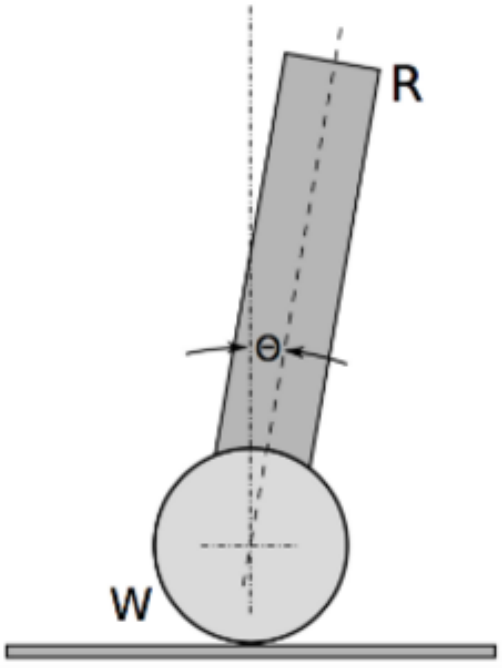
Team Leader

Phiet Do

Model

Inverting Pendulum

Figure 1: The Inverted Pendulum



Milestone 2

The purpose of this milestone is to develop a completed robot body assembly. This robot is to serve as an inverted pendulum, in that the robot body will cause the structure to topple over in itself without an external force to keep it upright. The robot will consist of the robot body, motors, and wheels, along with other necessary components.

Initial Design

Our initial design is to use a 3D printed model for the robot body, as the 3D printing service is free for students and relatively quick at the University of Georgia. Using a pre-created 3D model saved us time on designing a robot and saved money on our budget. The problem with the 3D model we chose, was that the design used wheels made from 3D printed material, which looked similar to gears. This would grant the robot much needed friction, but was out of the question due to our design requirements needing standard rubber wheels. Instead, we used wheels and motors supplied by one of our team members.

The components we have considered for use so far are: The Arduino Uno, DC motors, Motor Driver L298N, IMU (contains gyroscope and accelerometer), and the basic 3D printed structure.

Figure 2: Picture of the components on the structure

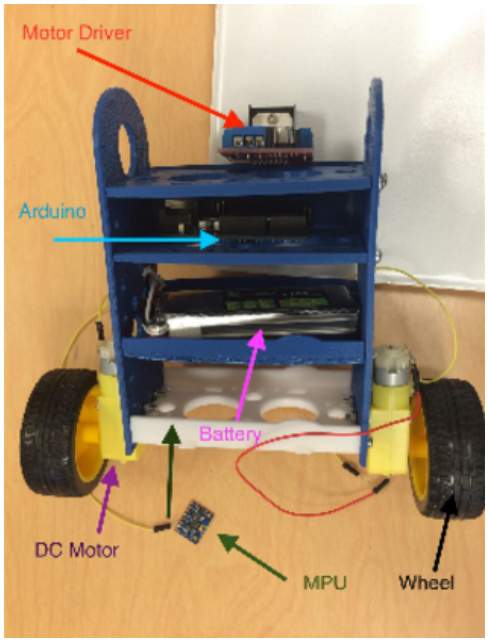


Figure 3: The robot falls down on its own

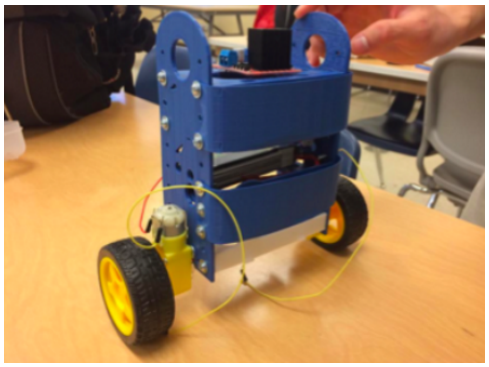


Figure 4: Testing the motors' voltage output

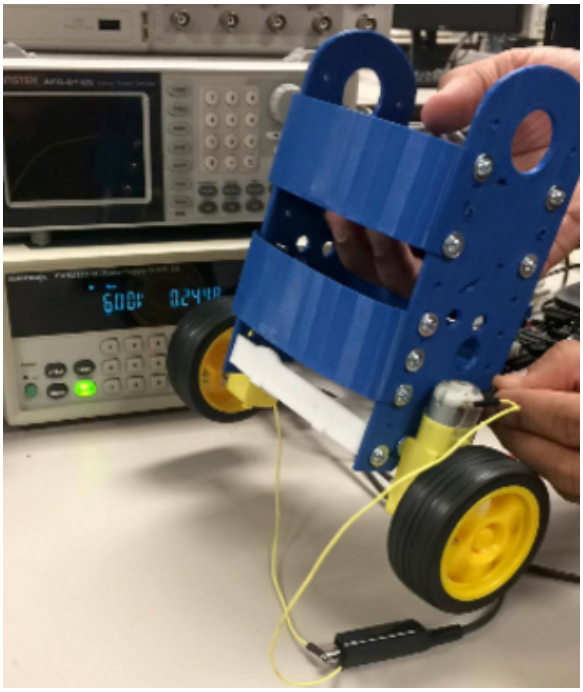
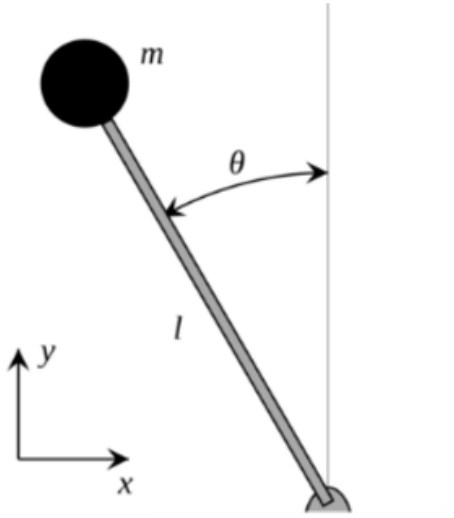


Figure 4 shows the testing of the motors to make sure they would spin the wheels at an acceptable angular velocity. We tested the voltage range from 4 volts to 11 volts. We believe that at least 7 volts should be enough to account for the extra weight of the robot and components for when we use the motors to correct the angular displacement of the robot. For the voltage source, we plan on using either standard AA lithium-ion batteries connected in series, a 9V lithium-ion, or a lithium-polymer battery such as those used in remote controlled cars.

General Equation and Algorithms

Figure 5: A Simple free body diagram



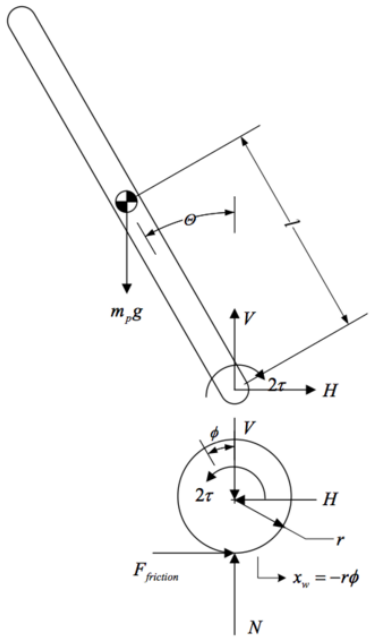
We believe our general equation to be

$$\tau = l * \ddot{\theta}$$

Where l is the moment of inertia, and $\ddot{\theta}$ is the angular acceleration and τ is the torque.

Based on figure 6 by Daniel Miller, we expect our algorithms to be incorporated with the following variables:

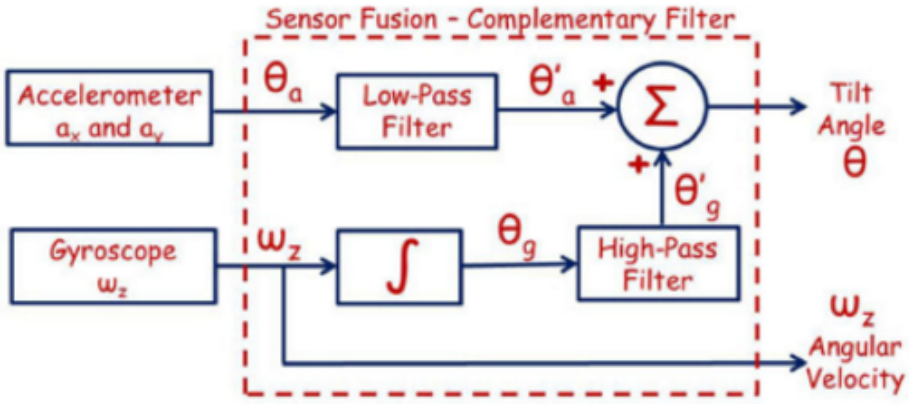
Figure 6: Free body diagram



- l = distance from axle to pendulum CG
- r = radius of wheel
- ϕ = angular displacement of wheel
- x_w = horizontal displacement of wheel
- θ = angular displacement of pendulum
- m_p = mass of pendulum
- V = vertical reaction force at axle
- H = horizontal reaction force at axle
- N = vertical reaction force at wheel
- $F_{friction}$ = friction force at wheel
- τ = torque at axle
- g = gravitational constant

In addition, these pictures provided by Kevin Craig gives us a general idea of what equations we need to implement into our system.

Figure 7: Block diagram for the gyroscope and accelerometer



Milestone 3

In summary, we see that there are three torques that act on the body of the inverted pendulum. Torque due to gravity, inertial torque, and torque due to the motion of the cart.

Gravitational Torque

$$\tau = F \cdot d = F \cdot \frac{l}{2} \cdot \sin(\varphi)$$

Linearization

$$\tau \simeq mg \cdot \frac{L\varphi}{2}$$

Inertial Torque

$$\tau_m = m \cdot \frac{L^2}{4} \cdot \frac{d^2\varphi}{dt^2}$$

Torque due to acceleration in the x-direction

$$\tau_x = m \cdot L \cdot \cos(\varphi) \cdot \frac{d^2x}{dt^2}$$

Linearization

$$\tau_x \simeq m \cdot L \cdot \frac{d^2x}{dt^2}$$

Therefore, the balance of torques equations (for small angles φ):

$$\tau_m = \tau \text{ - } \tau_x$$

$$m \cdot \frac{L^2}{4} \cdot \frac{d^2\varphi(t)}{dt^2} = \frac{mgL}{2}\varphi(t) - mL \cdot \frac{d^2x}{dt^2}$$

- multiply by $\frac{2}{mgL}$

Laplace Domain

$$\frac{L}{2g}s^2\varphi(5) - \varphi(s) = -\frac{2}{g}s^2X(s)$$

Thus:

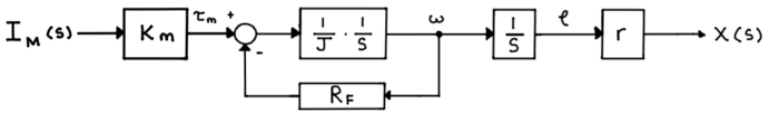
$$G(s) = \frac{\varphi(s)}{X(s)} = -\frac{\frac{2}{g}s^2}{\frac{1}{2g}s^2 - 1}$$

Overall pendulum transfer function

$$G(s) = -\frac{4s}{2s^2 - 2g}$$

Motor/Wheel transfer function

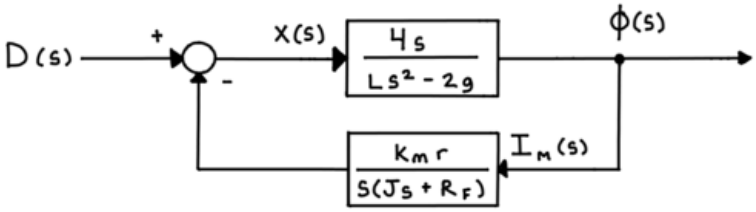
Figure 8: Block Diagram



$$\omega(s) = \frac{k_m r}{s\left(Js + R_F\right)}$$

Overall System

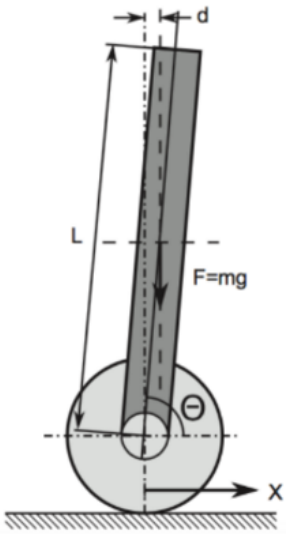
Figure 9: Block Diagram



where D(s) = displacement disturbance

Inverted Pendulum
1 Pole in the right half-plane

Figure 10: Free body diagram of the inverted pendulum



- θ = horizontal orientation
- φ = 90- θ ; angular deviation from the vertical orientation
- $d = \left(\frac{L}{2}\right) \sin(\varphi)$; center of mass shifted from the pivot point
- $F = mg$; gravity on center of mass giving force

$*$ No wheel friction / pivot friction

$$\tau = F \cdot d = F \cdot \frac{L}{2} \cdot \sin(\varphi) \simeq F \cdot \frac{L\varphi}{2} = \frac{mgL}{2}\varphi(t)$$
(linear approximation)

The torque causes an increase of the angular momentum of the mass

$$\tau_m = m \cdot \frac{L^2}{4} \cdot \frac{d^1\varphi}{dt^2}$$

$$torque = (mass)(length)(angularmomentum)$$

Acceleration in the x-direction and torque τ_x acting on the mass:

$$\tau_x = m \cdot L \cdot \cos(\varphi) \cdot \frac{d^2x}{dt^2} \simeq mL \cdot \frac{d^2x}{dt^2}$$

$$\tau_x = (mass)(length)(acceleration)$$

τ_m and τ_x work in opposites for small angles φ

$$\tau = \tau_m + \tau_x$$

Thus:

$$\tau_m = \tau - \tau_x$$

$$\tau = F \cdot d = F \cdot \frac{L}{2} \cdot \varphi(t) = \frac{mgL}{2}\varphi(t)$$

$$\frac{mL^2}{4} \cdot \frac{d^2\varphi(t)}{dt^2} = \frac{mgL}{2}\varphi(t) - mL \cdot \frac{d^2x}{dt^2}$$

Laplace Domain: multiply by $\frac{2}{mgL}$

$$\frac{L}{2g} \cdot s^2\varphi(s) = \varphi(s) - \frac{2}{g}s^2X(s)$$

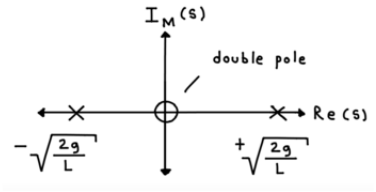
$$G(s) = \frac{\varphi(s)}{X(s)} = \frac{\left(-\frac{2}{g}s^2\right)}{\left(\frac{L}{2g}s^2 - 1\right)}$$

$$s^2 = \frac{2g}{L}$$

$$s = \pm\sqrt[14]{\frac{2g}{L}}$$

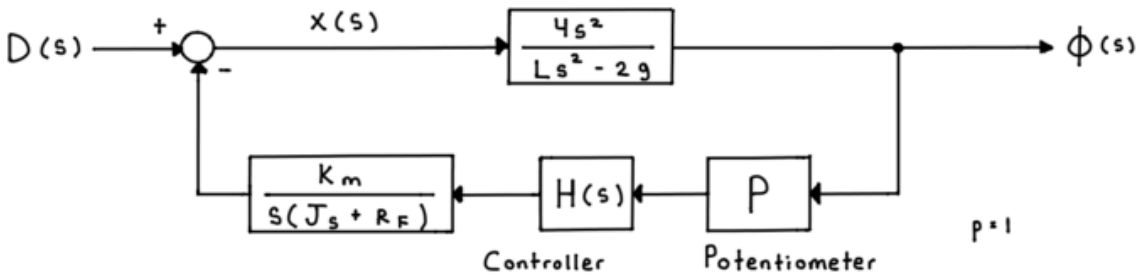
2 real pole and a double zero in the origin

Figure 11: Poles and Zeros Graph



$+\sqrt{\frac{2g}{L}}$, unstable real-valued pole causes angular deviation to exponentially increase.

Figure 12: Block Diagram

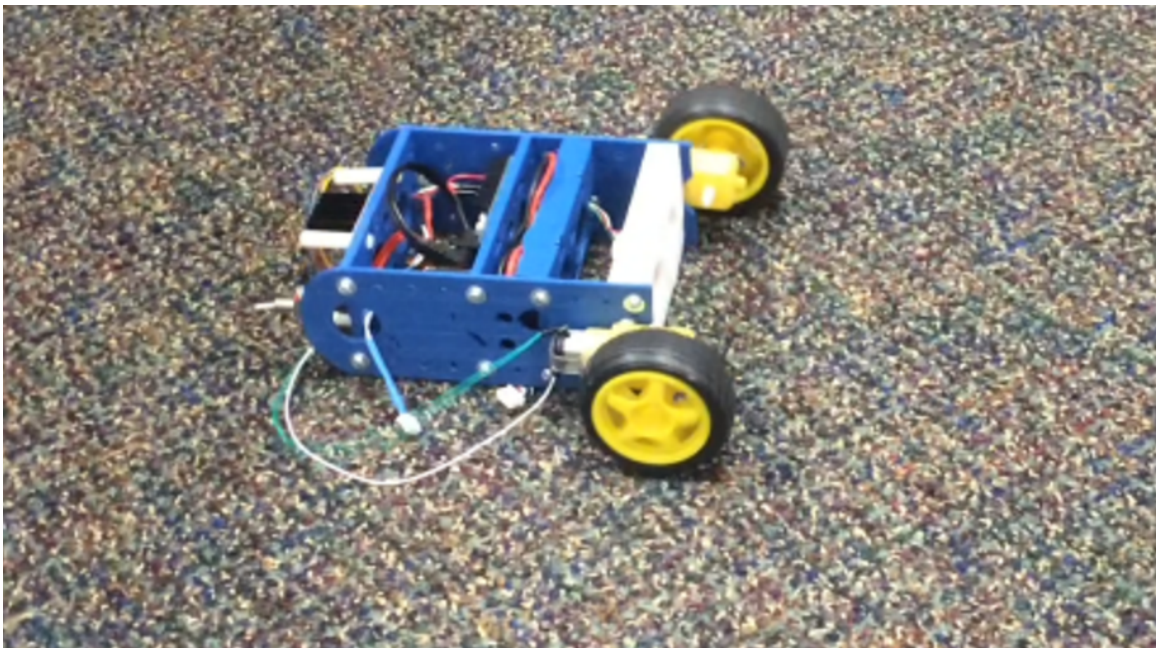


$D\,(s)$ = small shift of the wheels in the negative x-direction.

$$\varphi\,(s) = \frac{4s^2\left(Js+R_F\right)}{LJs^3+LR_Fs^2+(4k_mH(s)-2gJ)s-2gR_F} \cdot D\,(s)$$

We are still working on finding more precise coefficients but at this moment, these block diagrams provide a good explanation of what we have to deal with.

Figure 13: Result of unstable values



Milestone 4

In this milestone, we needed a sensor that could determine the angle of the upright robot. One obvious choice was to include an accelerometer to determine the angle and a gyroscope to determine the angular velocity. Picking these two sensors meant we would have to go digital. Below, we list the specifications and a brief summary of the sensors.

IMU SENSOR

An accelerometer is a device that measures magnitude and direction of proper acceleration. A gyroscope is a device that is useful for measures or maintaining orientation. The MPU 6050 is a 6 DOF (Degrees of Freedom) or a six axis IMU sensor, which means that it gives six values as output. Three values from the accelerometer and three from the gyroscope. The MPU 6050 is a sensor based on MEMS (Micro Electro Mechanical Systems) technology. Both the accelerometer and the gyroscope is embedded inside a single chip. This chip uses I2C (Inter Integrated Circuit) protocol for communication.

The scale of each depends on the sensitivity settings chosen, which can be one of 2, 4, 8, or 16g for the accelerometer and one of 250, 500, 1000, or 2000 deg/sec for the gyroscope. The accelerometer produces data in units of acceleration (distance over time2), and the gyroscope produces data in units of rotational velocity (rotation distance over time).

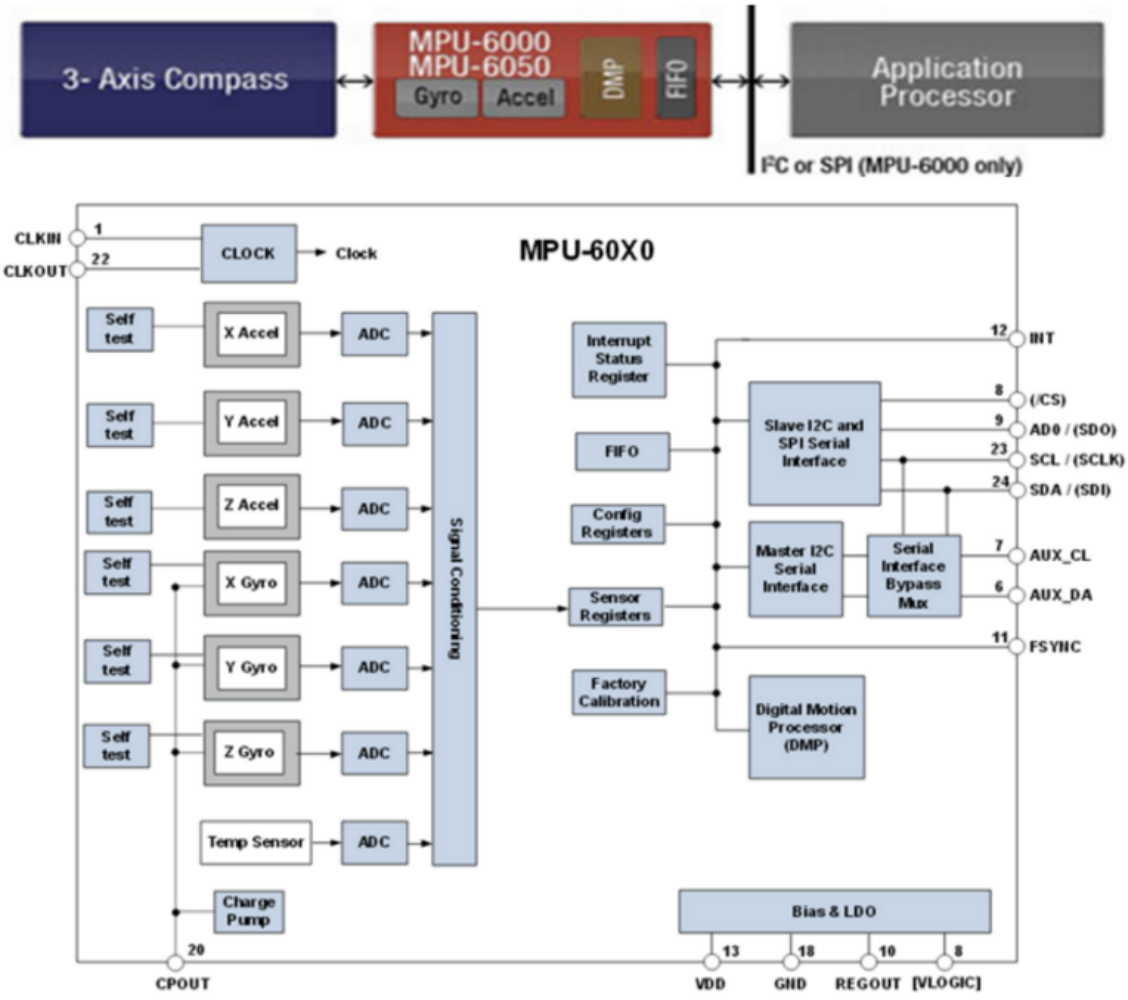
The output scale for any setting is [-32768, +32767] for each of the six axes. The default setting in the I2Cdevlib class is 2g for the accel and 250 deg/sec for the gyro. If the device is perfectly level and not moving, then:

- X/Y accel axes should read 0
- Z accel axis should read 1g, which is +16384 at a sensitivity of 2g
- X/Y/Z gyro axes should read 0

Specifications

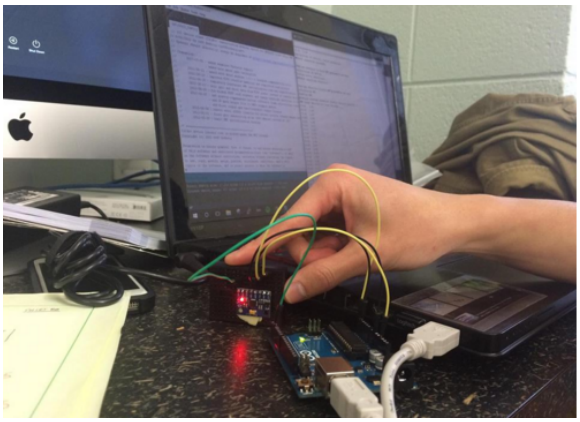
- I2C Digital-output of 6 or 9-axis MotionFusion data in rotation matrix, quaternion, Euler Angle, or raw data format.
- Input Voltage: 3 - 5V.
- Tri-Axis angular rate sensor (gyro) with a sensitivity up to 131 LSBs/dps and a full-scale range of 250, 500, 1000, and 2000dps.
- Tri-Axis accelerometer with a programmable full scale range of 2g, 4g, 8g and 16g.
- Digital Motion Processing (DMP) engine offloads complex MotionFusion, sensor timing synchronization and gesture detection.
- Embedded algorithms for run-time bias and compass calibration. No user intervention required.
- Digital-output temperature sensor.
- Dimensions: 20 * 15 * 1.6mm.

Figure 14: Outline of the MPU internal design



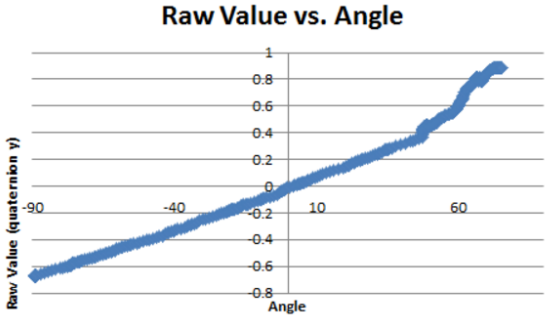
With an I2C library and an MPU6050 library provided by Jeff Rowberg, we were able to simulate our MPU. Unfortunately, we were not able to receive the bit readings of the sensor but we were able to calculate the angle pitch and also the quaternion y value. We started our procedure by placing the mpu sensor -90 degrees from the stationary position and we slowly moved to 90 degrees in the other direction in order to measure the pitch.

Figure 15: Acquiring the angle pitch



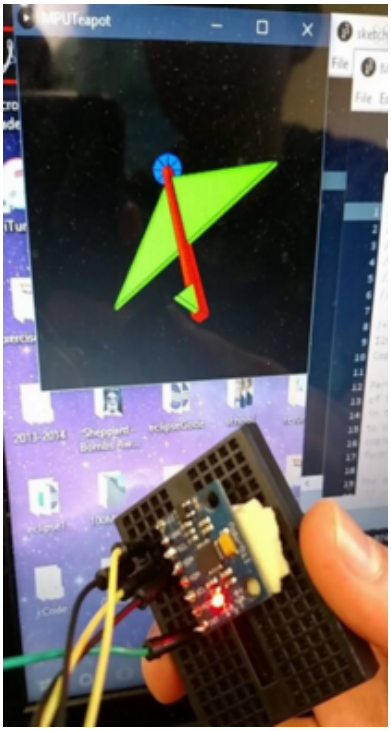
As shown in the figure below, we were able to see a linear relationship between the two based on 400 samples we monitored.

Figure 16: Graph of the sample



We then measured the noise to be 0.160065 based on taking the standard deviation of 250 samples. In addition we were able to simulate the MPU by using the Processing program. In the figure below the simulation shows, a plane adjusting to our sensor.

Figure 17: Program Simulation



Milestone 5

For this milestone, we needed to decide on a motor driver to control our motors speed and direction. Since we already decided on using a Arduino Uno as our microcontroller, we decided on a digital solution. The digital solution utilizes Pulse Width Modulation which is the means of controlling the duration of an electronic pulse. As a side note to our learning, we learned that the longer the pulses the faster the motor wheel will turn, and the shorter the pulses the slower the motor wheel will turn. Often motors will last longer and be more reliable when PWM is of consideration.

We then searched for our digital solution and found a simple and cheap option: the L298N motor driver. The specifications show that the motor driver is a double H bridge drive chip that has a drive voltage from 5v to 35v. It outputs a max power of 25 watts and provides a logical current from 0 mA to 36 mA. In addition, the motor driver has a built in 5 volt power supply when the driving voltage is 7v - 35v. This is option is ideal for our system as in provides a stable and reliable output to drive our two DC motors. The only concern is the controlling aspect is within the software programming now.

Figure 18: Outline of the motor driver

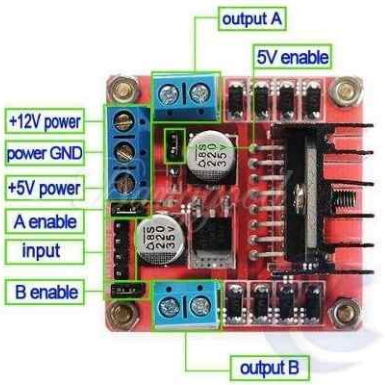
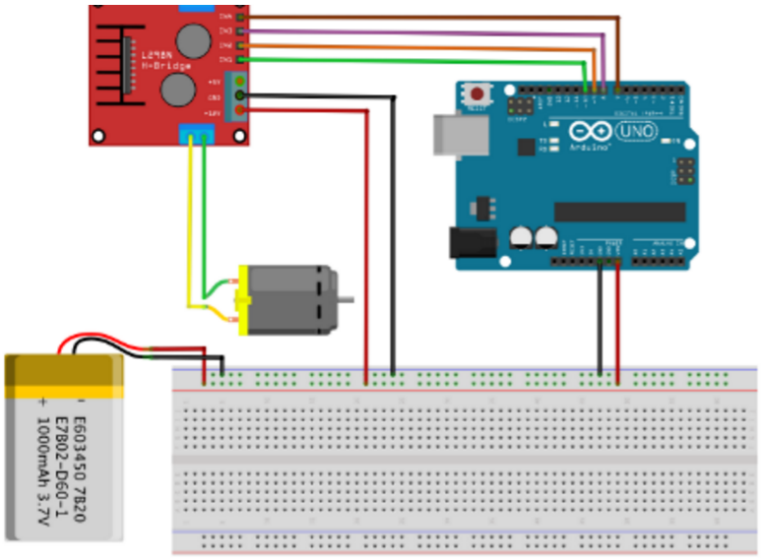


Figure 19: Schematic of the motor driver and Arduino



Milestone 6

The theory of the upright robot can be broken down into several steps. We work towards the open loop (Figure 19), a desired close loop, and our controller values

First, we wanted to examine the open loop poles of the process. This required us to examine the transfer function of the pendulum.

The transfer function for the process is shown here

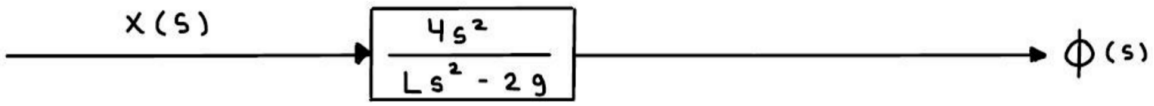
$$\varphi(s) = \frac{4s^2}{Ls^2 - 2g}$$

We then calculated our poles.

$$P_1 = +\frac{\sqrt{2g}}{\sqrt{L}} \qquad P_2 = -\frac{\sqrt{2g}}{\sqrt{L}}$$

This states that we would have one left and one right pole which makes a system unstable at the moment.

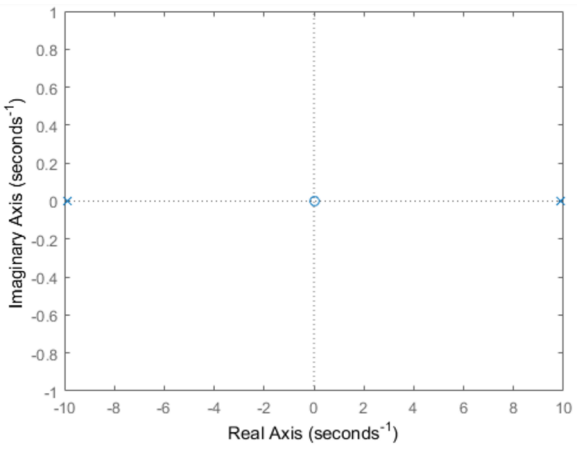
Figure 20: Open Loop Diagram



Using this example values, we find exact values for our poles:

$L = 0.2m \qquad g = 9.81\frac{m}{s^2}$

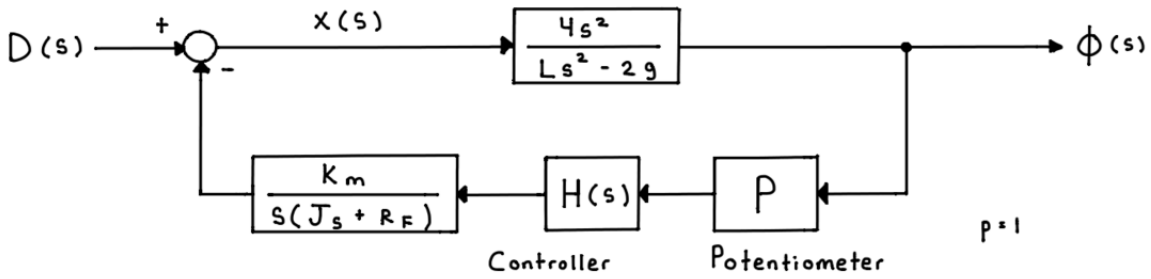
Figure 21: Pole-Zero Map



$P_1 = +9.9045 \qquad P_2 = -9.9045$

The open loop process is unstable because there is a pole to the right side of Imaginary axis. To make the process stable we first start by adding a feedback with a controller.

Figure 22: Closed Loop Diagram



First we decided to test with PI controller:

$H(s) = k_p + \frac{k_I}{s}$

Making our transfer function:

$$\varphi(s) = \frac{4s^2\left(Js+R_F\right)}{LJs^3+LR_Fs^2+\left(k_p-2gJ\right)s+k_I-2gR_F}$$

Routh Array

s^3	JL	$k_p - 2gJ$
s^2	R	$k_1 - 2gJR_F$
s^1	β	0
s^0	$k_1 - 2gJR_F$	0

$$\beta = \frac{R_FL\left(k_p-2gJ\right)-JL\left(k_1-2gR_F\right)}{R_FL}$$

For the system to be stable we need

- $k_p > 2gJ$
- $k_I > 2gR_F$
- $k_p R_F > k_I J$

Using values of

$L = 0.2m \qquad g = 9.81\frac{m}{s^2} \qquad R_F = 0.5Nm \qquad k_p = 2 \qquad k_I = 10$

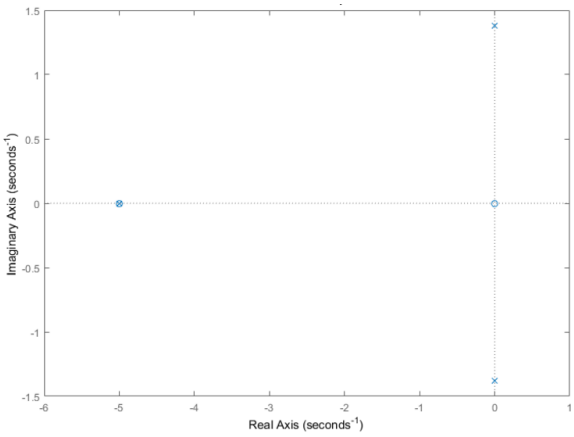
$$\varphi(s) = \frac{0.4s^3+2s^2}{0.02s^3+0.1s^2+0.038s+0.19}$$

Now zeros and poles of this system are:

Zeros: $Z_1 = 0 \qquad Z_2 = 0 \qquad Z_3 = -5$

Poles: $P_1 = -5 \qquad P_2 = +1.3784i \qquad P_3 = -1.3784i$

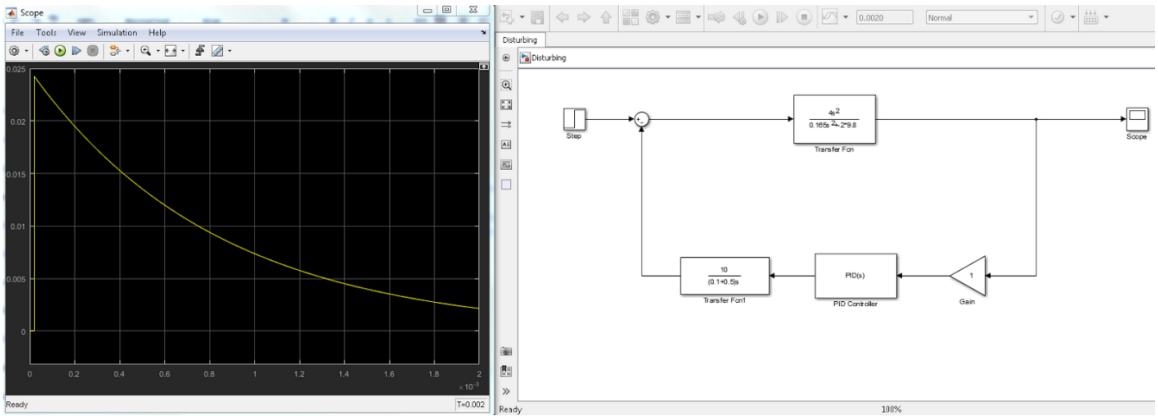
Figure 23: Pole-Zero Map



At this point, our system was marginally stable but still unstable.

Our next step was to implement a PID controller. We then used MATLAB to simulate the PID controller effectiveness.

Figure 24: Matlab Simulation

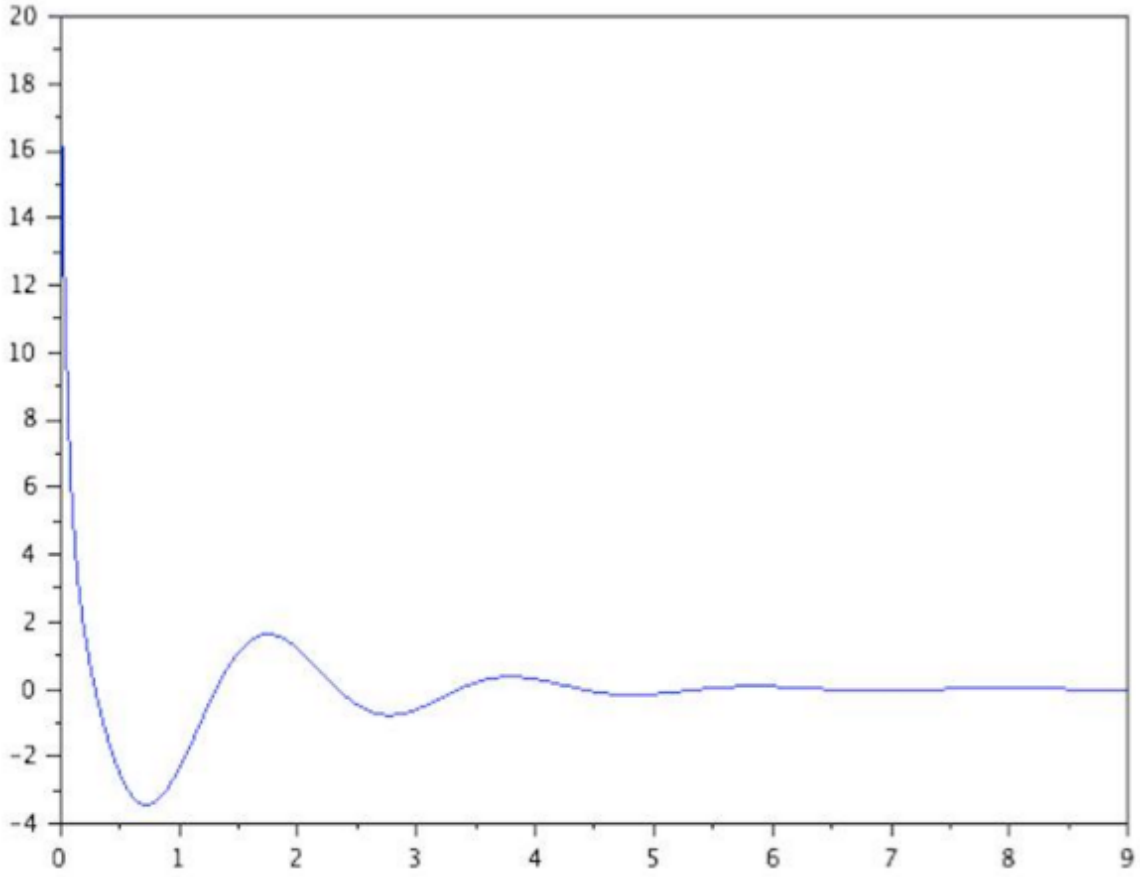


According to the simulation with $P = 2.7$, $I = 13.5$ and $D = 0.3$, we see that the response time is down to 2 ms.

To examine the steady state, we used SciLab to verify that our system would eventually reach equilibrium $\theta = O\,(t \to \infty)$

According to the plot below, our system does indeed reach equilibrium but at a time that we are not proud of. In addition, we have oscillations with undershoots and overshoots that we need to fix.

Figure 25: SciLab Simulation



Hopefully, with more time, we can fine tune the system to be stable and critically damped in the simulation. It is necessary to note that we based some of our constants on reasonable estimates, but in order to provide more accurate measurements in the future we must find the true motor efficiency variables, friction, inertial moments, and other considerations.

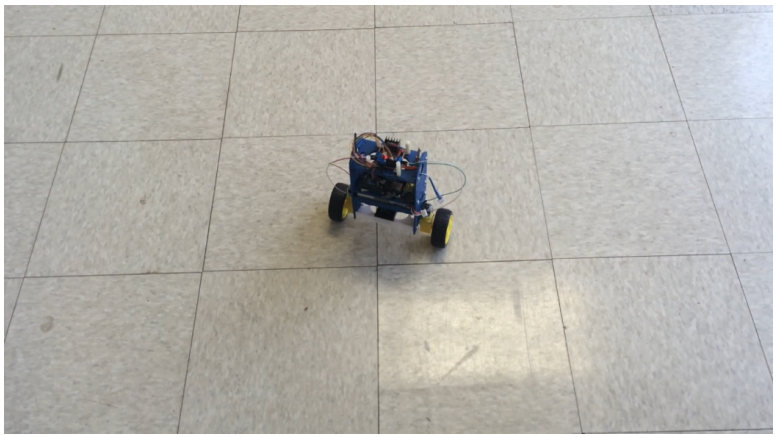
Milestone 7

We had to calibrate the PID values for the robot. Too much KP resulted in quick overshoots. Too much KD resulted in oscillations. Too much KI resulted in Chaos. Since we did not implement potentiometers to tune the PID values and the zero point in real time, we had to resort to constantly connecting the arduino and changing the code every time we wanted to make a change. However, we eventually narrowed down values that we were contempt with.

We also had to change the battery frequently. Not because the battery drained too quickly, but because we found that the amount of voltage charge in the battery heavily affected the performance of the robot. A charge of 9V would cause extreme vibrations and oscillating, but if the battery had drained to 7V, the robot would move slowly and only one wheel would turn. So, we settled that we prefered a battery with a voltage low enough to cause smooth motions, but high enough to power both wheels.

Other aspects that we changed were the wires and the motors. We decided that our first attempt at wiring was not up to our standard, so we resoldered to provide stronger connections and cleaner wiring. The motors we also changed at one point, because we felt our original set of motors did not function correctly as one motor was prone to stalling. It turned out that the problem was actually with the voltage supply and not the motors. ,

In the end, we finished building a self-balancing system.



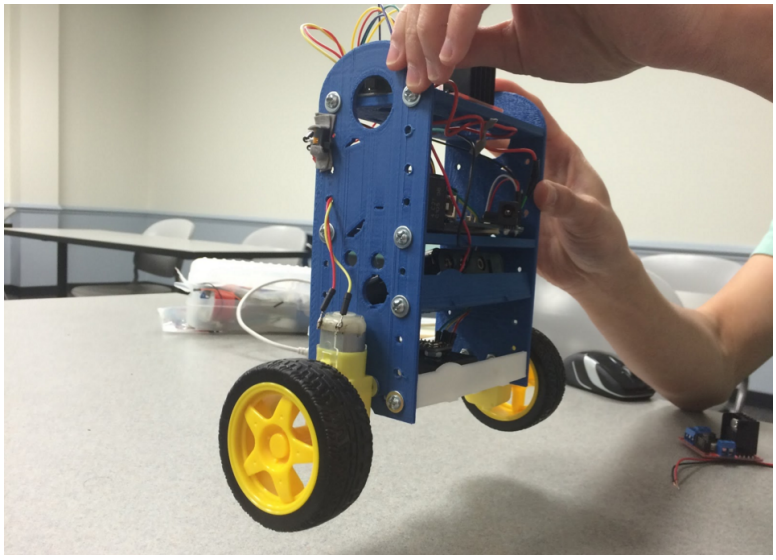
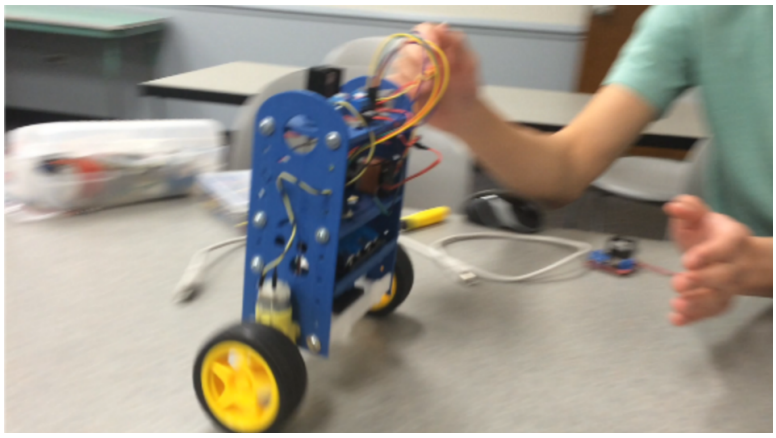


Figure 26: BaeBot in action



Works Cited

Craig, Kevin. "Measurement Systems." Measurement Systems. EDN, 4 Dec. 2013. Web. 11 Feb. 2016. <http://www.edn.com/electronics-blogs/mechatronics-in-design/4425655/Measurement-systems>.

Haidekker, Mark A., and ScienceDirect (Online service). Linear Feedback Controls: The Essentials. First edition. 2013. Web. 18th February 2016.

Jordan. "Pulse-width Modulation." SparkFun Electronics. SparkFun, n.d. Web. 25 Apr. 2016. <https://learn.sparkfun.com/tutorials/pulse-width-modulation>.

Kleeman, Lindsay. "Wavelet Kalman Filtering." Understanding and Applying Kalman Filtering(n.d.): 164-77. Understanding and Applying Kalman Filtering. Monash University, 1996. Web. 11 Feb. 2016. <http://www.cs.cmu.edu/ motionplanning/paper/sbp_papers/integrated3/kleeman_kalman_basics.pdf>.

Lampatronics. "L298N Motor Driver Board Module for Arduino - Lampa Tronics." Lampa Tronics. Lampa Tronics, n.d. Web. 16 Mar. 2016. <http://lampatronics.com/product/l298n-motor-driver-board-module-for-arduino/>.

Lauszus, Kristian. "TKJElectronics/KalmanFilter." GitHub. N.p., n.d. Web. 30 Mar. 2016.

Lindblom, Jim. "Adafruit/Adafruit_ADXL345." GitHub. Andydoro, n.d. Web. 25 Mar. 2016.

Miller, Daniel. “Self Balancing Robot Control System on Seelio.” Self Balancing Control System. Seelio, 29 Jan. 2016. Web. 11 Feb. 2016. <https://seelio.com/w/1t5k/self-balancing-robot-control-system>.

”MPU-6050 — InvenSense.” MPU-6050 Datasheet. Invensense, n.d. Web. 03 Mar. 2016.

Rowberg, Jeff. ”I2cdevlib.” GitHub. N.p., 30 May 2012. Web. 03 Mar. 2016.

Toni. “Accelerometer Basics.” SparkFun Electronics. SparkFun, n.d. Web. 25 Apr. 2016. <https://learn.sparkfun.com/tutorials/accelerometer-basics>.

```
/*
 *  Upright Robot that solves the problem of the classic inverted
 *  pendulum problem. This program operates with the sensor and motors
 *  included in the system and provides a corrective action while being
 *  able to adjust PID values. The following code utilizes the Kalman
 *  library, I2C library, and the ADXL345_U library. Credit goes to
 *  Kristian Lauszus, Jim Linblom, and pinuct for the simplicity.
 */

#include <Wire.h>
#include <Kalman.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL345_U.h>

/*
 * SETTING UP VARIABLES FOR THE SENSOR, TIMER,
 * AND ARDUINO PINS
 *
 */

//set up the IMU sensor
Adafruit_ADXL345_Unified accel = Adafruit_ADXL345_Unified(12345);
#define CTRL_REG1 0x20
#define CTRL_REG2 0x21
#define CTRL_REG3 0x22
#define CTRL_REG4 0x23
#define CTRL_REG5 0x24

//Routine to repeat part of the code every X milliseconds
#define runEvery(t) for (static long _lasttime;\
                         (uint16_t)((uint16_t)millis() - _lasttime) >= (t);\
                         _lasttime += (t))


// assigns a variable to locate the address
int L3G4200D_Address = 105;


//kalman variables
Kalman kalmanP;
Kalman kalmanR;


//Set up accelerometer variables

float acceloX, acceloY, acceloZ;
double aPitch, aRoll;

//Set up gyroscope variables
float gyroX, gyroY, gyroZ;
float gChangeX, gChangeY, gChangeZ;
float gyroP = 180;
float gyroR = -180;
float gyroYaw = 0;
double gyro_sensitivity = 70; //Data value 2000DPS = 70
int x;
int y;
```

```
int z;
//Set up a timer Variable
uint32_t timer;

// Set up Angle value variables
double InputPitch, InputRoll;

// Set up Initial angle value variables
double pitchInitial,rollInitial;

// Assigning pin variables to motors
int onA = 3;
int onB = 9;
int motorA1 = 5;
int motorA2 = 4;
int motorB1 = 10;
int motorB2 = 11;


void setup() {

  Wire.begin();

  Serial.begin(9600);

  /*
   * Confiure the ADXL345 sensor
   */
  setupL3G4200D(200); // Configure L3G4200  - 250, 500 or 2000 deg/sec
  delay(1500);


  // Detect if the sensor is not on or working
  if(!accel.begin())
  {
    /* check if sensor is working, if not print error message */
    Serial.println("check your sensor wiring");
    while(1);
  }

  // Assign Motorpins to read and set them as outputs
  pinMode(onA, OUTPUT);
  pinMode(onB, OUTPUT);
  pinMode(motorA1, OUTPUT);
  pinMode(motorA2, OUTPUT);
  pinMode(motorB1, OUTPUT);
  pinMode(motorB2, OUTPUT);
  ////////////////////////////////////////////////////////////////////////

  // Set to no direction at the beginning
  digitalWrite(motorA1,HIGH);
  digitalWrite(motorA2,HIGH);
  digitalWrite(motorB1,HIGH);
  digitalWrite(motorB2,HIGH);

  accel.setRange(ADXL345_RANGE_2_G); //configure the accelerometer
```

```
  sensors_event_t event;  // create nother variable to hold the accelerometer
data
  accel.getEvent(&event);

  // Calculate bias for the Gyro i.e. the values it gives when it's not
moving
  // NOTE: You must keep the robot vertical and static for the initial
seconds.
  // This will help the sensor calculate the initial values at the starting
set point.
  for(int i=1; i < 100; i++){

    getGyroValues(); //grabbing the gyro data
    gyroX += (int)x;
    gyroY += (int)y;
    gyroZ += (int)z;

    accel.getEvent(&event); //grabbing the accelerometer data
    acceloX += event.acceleration.x;
    acceloY += event.acceleration.y;
    acceloZ += event.acceleration.z;

    delay(1);
  }

  //final bias values for every axis
  gyroX = gyroX / 100;
  gyroY = gyroY / 100;
  gyroZ = gyroZ / 100;

  acceloX = acceloX / 100;
  acceloY = acceloY / 100;
  acceloZ = acceloZ / 100;


  //Get Starting Pitch and Roll
  accel.getEvent(&event);
  aPitch = (atan2(-event.acceleration.x,-
event.acceleration.z)+PI)*RAD_TO_DEG;
  aRoll = (atan2(event.acceleration.y,-event.acceleration.z)+PI)*RAD_TO_DEG;

  if (aPitch <= 360 & aPitch >= 180){
    aPitch = aPitch - 360;
  }

  if (aRoll <= 360 & aRoll >= 180){
    aRoll = aRoll - 360;
  }


  // Set starting angle for Kalman
  kalmanP.setAngle(aPitch);
  kalmanR.setAngle(aRoll);

  kalmanR.setQangle(0.01);       // 0.001
  kalmanR.setQbias(0.0003);     // 0.003
  kalmanR.setRmeasure(0.01);     // 0.03
```

```
  gyroP = aPitch;
  gyroR = aRoll;

  timer = micros();
  delay(1000);
  ValueInitial();

}

////////////////////////////////////////////////////////////////////////////
/
double Setpoint;
void MotorControl(double out){
  //set the direction of the wheels to go forward
  if (out > 0){
    digitalWrite(motorA1,HIGH);
    digitalWrite(motorA2,LOW);
    digitalWrite(motorB1,LOW);
    digitalWrite(motorB2,HIGH);
  }
  //set the direction of the wheels to go backward
  else{
    digitalWrite(motorA1,LOW);
    digitalWrite(motorA2,HIGH);
    digitalWrite(motorB1,HIGH);
    digitalWrite(motorB2,LOW);
  }
  // get the absolute value of the velocity
  byte vel = abs(out);
  //check to see if the velocity hits the oMax output range
  if (vel<0)
    vel=0;
  if (vel > 255)
    vel=255;

  //PWM to the motors
  analogWrite(onA,(vel/1.1));
  analogWrite(onB,(vel/1.1));
}


/*
 * Function that grabs the initial values from the Accelerometer and
Gyroscope
 */
void ValueInitial(){

  //  Accelerometer calculations start here
  // These steps are similar to finding the starting values at the beginning
  sensors_event_t event;

  accel.getEvent(&event);
  //calculate the pitch and roll changing radians to degrees - pinuct
  aPitch = (atan2(-event.acceleration.x,-
event.acceleration.z)+PI)*RAD_TO_DEG;
  aRoll = (atan2(event.acceleration.y,-event.acceleration.z)+PI)*RAD_TO_DEG;
```

```
  if (aPitch <= 360 & aPitch >= 180){
    aPitch = aPitch - 360;
  }
  if (aRoll <= 360 & aRoll >= 180){
    aRoll = aRoll - 360;
  }


  //  Gyroscope calculations start here
  getGyroValues();
  // read raw angular velocity measurements from device
  gChangeX = ((int)x - gyroX)*.07; //*(.0105);
  gChangeY = -((int)y - gyroY)*.07; //*(.0105);
  gChangeZ = ((int)z - gyroZ)*.07; //*(.0105);

  gyroP += gChangeY * ((double)(micros() - timer)/1000000);
  gyroR += gChangeX * ((double)(micros() - timer)/1000000);
  gyroYaw += gChangeZ * ((double)(micros() - timer)/1000000);

  pitchInitial = kalmanP.getAngle(aPitch, gyroP, (double)(micros()-
timer)/1000000);
  //rollInitial = kalmanR.getAngle(aRoll, gyroR, (double)(micros()-
timer)/1000000);
  timer = micros();
  rollInitial = gyroR;

  Serial.print("Initial Pitch: ");
  Serial.println(pitchInitial);
  Serial.print("Initial Roll: ");
  Serial.println(rollInitial);
  Setpoint = 0;

}
////////////////////////////////////////////////////////////////
/*
 * Function that grabs the continuously grabs values from the Accelerometer
and Gyroscope
 * The code is ran every 10 miliseconds which is 100Hz
 */
int i=0;
double ang =0;
void loop() {

  runEvery(10){

    sensors_event_t event;

    accel.getEvent(&event);
    aPitch = (atan2(-event.acceleration.x,-
event.acceleration.z)+PI)*RAD_TO_DEG;
    aRoll = (atan2(event.acceleration.y,-
event.acceleration.z)+PI)*RAD_TO_DEG;

    if (aPitch <= 360 & aPitch >= 180){
      aPitch = aPitch - 360;
    }
```

```
    if (aRoll <= 360 & aRoll >= 180){
      aRoll = aRoll - 360;
    }
    getGyroValues();

    // get raw angluar velocity values
    gChangeX = -((int)x - gyroX)*.07; //*(.0105);
    gChangeY = -((int)y - gyroY)*.07; //*(.0105);
    gChangeZ = ((int)z - gyroZ)*.07; //*(.0105);

    gyroP += gChangeY * ((double)(micros() - timer)/1000000);
    double timeVar = gChangeX * ((double)(micros() - timer)/1000000);
    gyroR += gChangeX * ((double)(micros() - timer)/1000000);
    gyroYaw += gChangeZ * ((double)(micros() - timer)/1000000);

    InputPitch = kalmanP.getAngle(aPitch, gyroP, (double)(micros()-
timer)/1000000);
    InputRoll = kalmanR.getAngle(aRoll, gyroR, (double)(micros()-
timer)/1000000);
    timer = micros();

    byte a= map(abs(Compute(InputRoll-rollInitial)),0,255,0,124);
    ang = 0.98* (ang + timeVar) + 0.02 * (aRoll);
    MotorControl(Compute(ang-rollInitial));


  }

}

/*
 * The following is to provide the corrective action for the system
 * by utilizing PID values. Note, PID values must be changed depending on
 * the different specs of the upright robot
 */
int oMax = 255;
int oMin = -255;
float lastValue = 0;
double ITerm =0;
double kp = 15;
double ki = 0;
double kd = 2;

double Compute(double input)
{

      double error = Setpoint - input;
      ITerm+= (ki * error);
      if(ITerm > oMax) ITerm= oMax;
      else if(ITerm < oMin) ITerm= oMin;
      double dInput = (input - lastValue);

      /*Compute PID Output*/
      double output = kp * error + ITerm + kd * dInput;

          if(output > oMax) output = oMax;
```

```
      else if(output < oMin) output = oMin;

      /*Remember some variables for next time*/
      lastValue = input;
      return output;
}

/*
 * The following is code from pinuct and Jim Lindblom. This code is used to
 * set up the sensor's registers by using I2C. This helps us read directly
 * from the sensor without using too much wires.
*/
void getGyroValues(){

  byte xMSB = readRegister(L3G4200D_Address, 0x29);
  byte xLSB = readRegister(L3G4200D_Address, 0x28);
  x = ((xMSB << 8) | xLSB);

  byte yMSB = readRegister(L3G4200D_Address, 0x2B);
  byte yLSB = readRegister(L3G4200D_Address, 0x2A);
  y = ((yMSB << 8) | yLSB);

  byte zMSB = readRegister(L3G4200D_Address, 0x2D);
  byte zLSB = readRegister(L3G4200D_Address, 0x2C);
  z = ((zMSB << 8) | zLSB);
}

int setupL3G4200D(int scale){
  //From  Jim Lindblom of code

  // turn on x, y, z
  writeRegister(L3G4200D_Address, CTRL_REG1, 0b00001111);

  writeRegister(L3G4200D_Address, CTRL_REG2, 0b00000000);

  // Configure CTRL_REG3 to generate data ready interrupt on INT2
  writeRegister(L3G4200D_Address, CTRL_REG3, 0b00001000);

  // CTRL_REG4 controls the full-scale range, among other things:

  if(scale == 250){
    writeRegister(L3G4200D_Address, CTRL_REG4, 0b00000000);
  }else if(scale == 500){
    writeRegister(L3G4200D_Address, CTRL_REG4, 0b00010000);
  }else{
    writeRegister(L3G4200D_Address, CTRL_REG4, 0b00110000);
  }
  // CTRL_REG5 controls high-pass filtering of outputs
  writeRegister(L3G4200D_Address, CTRL_REG5, 0b00000000);
}

void writeRegister(int deviceAddress, byte address, byte val) {
    Wire.beginTransmission(deviceAddress); // start transmission to device
    Wire.write(address);       // send register address
    Wire.write(val);         // send value to write
    Wire.endTransmission();     // end transmission
}
```

```
int readRegister(int deviceAddress, byte address){

    int v;
    Wire.beginTransmission(deviceAddress);
    Wire.write(address); // register to read
    Wire.endTransmission();

    Wire.requestFrom(deviceAddress, 1); // read a byte

    while(!Wire.available()) {
        // waiting
    }

    v = Wire.read();
    return v;
}
```