Control System for a Self Stabilizing Bicycle Using a Beaglebone Black

December 17, 2014

Tao Marvin Liu (tl474)
Sophomore, Computer Science
MAE 4900 – 3 Credits

Table of Contents

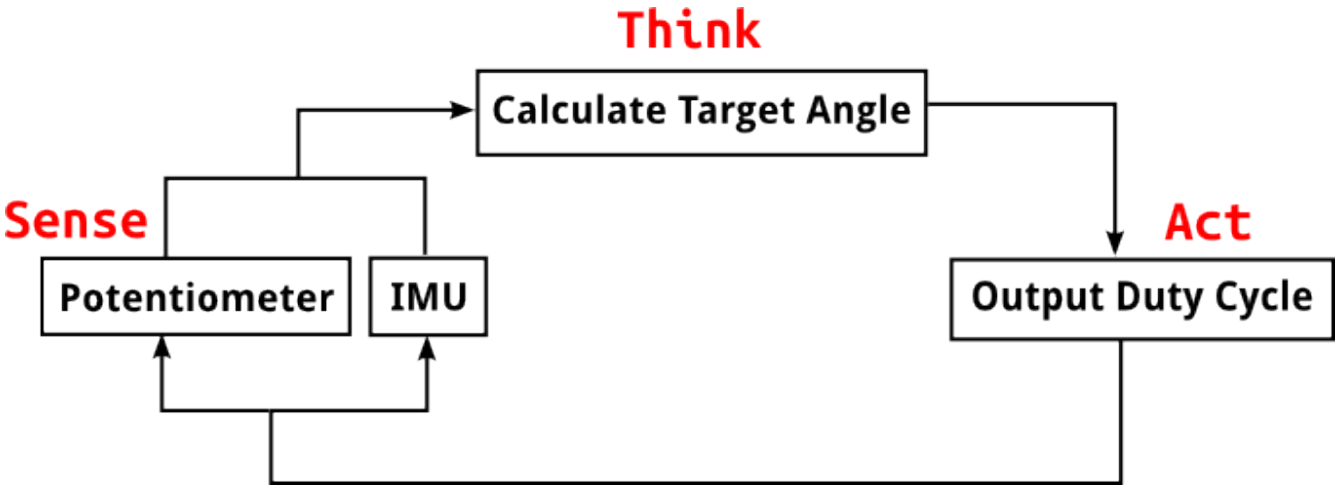
Abstract

The objective of our team is to design and build a self-stabilizing bicycle building off of the work done by a team last semester. My responsibilities for this project is to implement a feedback control system on the Beaglebone Black that takes inputs from an inertial measurement unit (IMU) and a potentiometer and outputs an appropriate voltage gain to power a steering motor. I have built off the work done by Mukund Sudarshan last spring, and Ariam Espinal over the summer, and this paper expands on the “Feedback control loops for balance” section of Mukund Sudarshan's 2014 paper “Using the BeagleBone Black on a robot bicycle”. Using several Python scripts, I have implemented a feedback controller following the6 sense-think-act model, and this paper will cover the setup, implementation, and usage of each of the three parts of the controller.

Motivation

The goal for this project is to understand how humans balance a bicycle while riding and to replicate this using a Beaglebone Black, sensors, and a steering motor to simulate a human balancing a bicycle. The goal is to build a bicycle that can move autonomously without falling over. After we reach this goal, we want to be able to have the bicycle perform a track stand (balance in place while being stationary). In the future, we also want to implement a fly-by wire that will allow human riders to control the bicycle without having to concern themselves with balance. The rides will turn the handlebars like a steering wheel to steer the bicycle, and the on-board control system will stabilize the bicycle for the rider.

Overview of the Control System



In order to keep the bicycle upright, we implemented a feedback control system on the Beaglebone Black that uses the sense-think-act model: it reads the current state of bicycle (lean angle and steer angle), calculates the necessary steering angle to correct the imbalance of the bicycle, and outputs a gain to turn the motor to the correct steering angle.

In the “sense” phase, the controller reads the current lean angle using an inertial measurement unit (IMU) and the current steering angle using a potentiometer. Then, in the “think” phase, these two data points are the inputs to the equation that calculates a target steering velocity. Finally, in the “act phase”, the controller uses a proportional-integral-derivative (PID) controller and the results of the previous two phases to control the steering motor of the bicycle. This loop is run continuously in order to keep the bicycle upright. To accomplish this task, we have chosen to use Python because many Beaglebone specific libraries have already been written for Python, and this simplifies our task greatly. The following sections will expand on each of these phases of the control system.

Sensing

Inertial Measurement Unit

Setup

For this project, we are using a MicroStrain Inertia-Link IMU with a BeagleBone Black RS-232 Serial Micro-Cape to connect the IMU to the Beaglebone. It is important to use the RS-232 cape to connect the IMU, because the Beaglebone's USB port does not support serial communication. Computer USB ports do support serial communication, but we have discovered that using serial to USB yields some sort of noise bytes that interfere with the data. Thus, for future work using the IMU, it is important to connect the IMU through the RS-232 cape. It is also important to note that the default serial port for the cape is ttyO4 and not ttyO1.

Implementation

The imu.py file contains one function that will be called by users and several helper functions for this main function. The main function, get_roll_angle_ang_rate, takes in the serial port where the IMU is connected to, and returns a tuple containing the lean angle and the angular rates. This function sends a command byte (in our case CF) to the IMU, waits for the response packet, makes sure the checksum of the packet is valid, and returns the relevant data as floating point numbers. If for some reason the checksum is invalid, the function will repeat the process.[1]

The IMU can be used in two modes: polled and continuous. We chose to poll the data with each function call rather then set the IMU into continuous data mode for two reasons:

1. It is more difficult to turn set the IMU into continuous data mode and even harder to turn it off.
2. If we continuously gather data at the maximum speed, then we would need to implement a buffer to hold the information. While this itself is not overly problematic, the problem is that our total control cycle takes longer than the time it takes to gather one packet of information. If we are using continuous mode, by the time we reach our next control cycle, the next data packet in the buffer will be outdated. We could solve this by adjusting the period to be the same as the length of our control cycle, but doing so would defeat the purpose of using continuous mode and it would be simpler and more effective to poll the IMU for data in each cycle.

If future groups decide to switch to continuous mode, instructions can be found in the IMU Communications Protocol[2].

Usage

Here is an example use of the IMU in our main file control.py (Note: The serial port settings are given in the IMU Communications Protocol and should not be changed)

```
import imu
import Adafruit_BBIO.UART as UART

UART.setup("UART4")
imu_serial_port = serial.Serial(port = "/dev/ttyO4", baudrate=115200,
      parity=serial.PARITY_NONE, stopbits=serial.STOPBITS_ONE,
      bytesize=serial.EIGHTBITS, timeout=0, xonxoff=0, rtscts=0)
imu_data = imu.get_roll_angle_ang_rate(imu_serial_port)
      lean_angle = imu_data[0]
```

1 Full documentation of the functions can be found in Appendix B

2 http://files.microstrain.com/dcp/Inertia-Link-3DM-GX2-data-communications-protocol.pdf

Potentiometer

Setup

For this project, we use a standard potentiometer with three connections. Ground the first terminal of the potentiometer. The second terminal should be wired to the output of the Beaglebone, which is P9_32 (VDD_ADC). This provides 1.8V to the potentiometer. Finally, connect the third terminal to any of the Beaglebone's analog input pins. We chose P9_40 specifically because it is the farthest one from where the RS-232 serial cape would be connected to. Additionally, we use a differentiator to calculate the change in voltage, allowing us to determine the velocity at which the potentiometer is turning. We chose P9_39 as the pin on the Beaglebone for the output of the differentiator.[3] The full explanation and set up for the differentiator and the potentiometer can be found in Ariam Espinal's 2014 paper.

Implementation

We defined a Potentiometer class and use this to instantiate potentiometer objects within our main controller code. The reason we do this is that each potentiometer stores two input pins, a smoothing factor, and a current data point. Also, by making it object-oriented, the logic and interaction between the different components of the controller are much clearer.

The potentiometer sends a voltage between 0V and 1.8V to a Beaglebone AIN pin. We then use the analog to digital converter library to read this voltage. In this process, the data we read is scaled to between 0 and 1, so to convert the data into angles or radians, we multiply by 360 or pi respectively.

The function get_angle simply takes one reading of the potentiometer and returns the angle that the potentiometer is currently at. Because of the nature of analog to digital conversions as well as background noise in transmitting the data, the angle readings have a range of 0.8 degrees when the potentiometer is stationary[4]. To deal with this inaccuracy, we have implemented a simple exponential filter. The formula for the exponential filter is

$$data = SMOOTH * (new\_data) + (1 - SMOOTH) * current\_data$$

where SMOOTH is between 0 and 1 inclusive. A smoothing factor closer to 1 means less smoothing. We calculate the initial data point for the exponential filter by averaging the first four data points. It is because of the exponential filter that we created potentiometer objects rather that just use a function that reads the AIN pin once. To use the exponential filter, use the function get_steer_angle instead of get_angle.[5]

The function get_velocity reads the given velocity pin, scales the voltage from the differentiator, and returns the instantaneous angular velocity of the potentiometer.

Usage

Here is an example use of the potentiometer without exponential smoothing in our main file control.py :

```
import potentiometer as Pot
# Potentiometer pins on BeagleBone Black
VELOCITY_PIN ='P9_39'
```

3 The Beaglebone pin out diagram can be found in Appendix A.

4 Graphs of the potentiometer data with and without smoothing and capacitors can be found in Appendix C.

5 Full documentation of the functions can be found in Appendix B.

```
POS_PIN ='P9_40'

# Constructs a new potentiometer object
pot = Pot.Potentiometer(POS_PIN, VELOCITY_PIN)

steer_angle = pot.get_angle()
```

Possible Future Changes

Currently, our controller implementation does not make use of exponential filtering. The tests for the potentiometer and exponential filter can be found in Appendix C. Our data still varies within a range of a degree, even though the points are more concentrated towards the center. In the future, we may want to construct a low pass filter to filter out the high frequency noise as well as use a more sophisticated filter like the Butterworth filter to process our signals from the potentiometer. The code for the filter has already been written, but testing still needs to be done with the potentiometer and the filter. One concern about using the Butterworth filter is the increase in the number of computations per feedback loop. We will need to measure the time it takes for the signal to be processed using the Butterworth filter to determine the exact trade off between speed and accuracy.

Thinking

Once the control system has gathered the lean angle, lean velocity, and the steering angle from the IMU and the potentiometer, it calculates the target velocity that the steering motor must be at to correct the balance of the bicycle. To do so, the main controller invokes a function called calculate, which takes in the gathered data as three parameters and returns a target velocity. The equation that we are using right now is developed by Shihao Wang, and is as follows:

$$\dot{\delta} = \begin{bmatrix} -4.8174 & -1.0151 & 5.5671 \end{bmatrix} \begin{bmatrix} \phi \\ \dot{\phi} \\ \delta \end{bmatrix}$$

$$\phi = lean\ angle$$
$$\dot{\phi} = lean\ velocity$$
$$\delta = steer\ angle$$
$$\dot{\delta} = targetvelocity$$

In code, the controller would be:

```
def calculate(lean_angle, lean_velocity, steer_angle):
      k1 = -4.8174
      k2 = -1.0151
      k3 = 5.5671
      return -(k1*lean_angle + k2*lean_velocity + k3*steer_angle)
```

By encapsulating the logic within a separate function, the programmer of the control system will have a much easier time reasoning about the logic of the program and, if necessary, changing the mathematical model in the future. For example, in the main function of the control system, we only have to write two lines of code to control the motor.

```
target_velocity = calculate(lean_angle, lean_velocity, steer_angle)
motor.control_motor(target_velocity, time.time())
```

If the mathematical model has to be changed in the future, the only code that will have to be changed will be the body of calculate().

Acting: Proportional-Integral-Derivative Control

Setup

For the setup of the motor and related circuitry, refer to Ariam Espinal's 2014 paper.

Implementation

In order to control the steering motor, we created a class called MotorController to stores all the associated constants and data needed for our PID controller. We are using PWM signals sent from the Beaglebone Black through pulse width modulation (PWM) pins to control the direction and speed of the motor. P8_19 is used to control the direction of the motor, and P8_13 is used to send the appropriate duty cycle to control the velocity of the motor.

Initializing the motor controller

The init function of MotorController requires a Potentiometer object defined in our potentiometer.py file. This is the potentiometer that the PID controller will use to get the current velocity of the steering motor, which is necessary to calculate the error between our target velocity and our current velocity.

The function also requires the PWM output pins for direction and duty cycle for the motor, as well as the KP, KI, and KD tuning in the PID controller model. The value QMAX represents the maximum value that we allow the integral term of the controller to be. This helps prevent windup in the controller, where the motor rapidly increase in speed when we don't want it to.

Controller Logic

The function that will be called by users outside of pid.py is control_motor. This function takes in a target velocity as well as the time at which the function is called (use time.time() to find this).

The first step is to calculate the velocity error, as this is the basis of PID controllers. To that, we subtract the input target velocity and the current velocity. We also calculate another term, delta_error, which is the difference between our current error and the error from the last loop. This value will be used in the derivative controller part of the PID controller.

Next, we update the integral term q and also use the function anti_windup, which resets the integral term every 3 cycles to prevent it from getting infinitely large.

Finally, we calculate the output duty we want to send to the motor using the function calc_motor_output, which is simply the PID algorithm, and then send it using the set_duty_cycle function from the PWM library. Full documentation of the code can be found in Appendix B.

Important Note for Future Development

We have switched the PID controller to calculate the error in the steering motor's velocity rather than position only in the last few days. Thus, the MotorController object has yet to be fully tested, and must be done at the start of next semester.

Usage

Here is an example use of the PID controller in our main file control.py (**Note**: The constants are purely for testing purposes and have not been tuned to work with the actual bicycle)

```
import pid

VELOCITY_PIN ='P9_39'
POS_PIN ='P9_40'

# Motor Constants
DIR_PIN = "P8_19"
DUTY_PIN = "P8_13"
KP = 0.15
KI = 0
KD = 0.001
QMAX = 982

pot = Pot.Potentiometer(POS_PIN, VELOCITY_PIN)
motor = pid.MotorController(pot, DIR_PIN, DUTY_PIN, KP, KI, KD, QMAX)

# Output to motor
target_velocity = calculate(lean_angle, lean_velocity, steer_angle)
motor.control_motor(target_velocity, time.time())
```

Motor Hub

Setup

For propulsion, we are using a Magic Pie 2X made by Golden Motor Technology[6]. We then connect the motor hub to a PWM pin on the Beaglebone. In our case we used P9_16.

Usage

For now, our motor will be constantly powering the bicycle, as our goal is to balance the bicycle while it is moving at a constant rate. Thus the code to use the motor hub is very simple.

```
import Adafruit_BBIO.PWM as PWM
MOTOR_PIN = "P9_16"
PWM.start(MOTOR_PIN, 50)
```

Running the Control System

Testing

The main controller which interacts with the PID controller, the IMU, and the potentiometer, is located in the file control.py. Thus, all that is needed to run the bicycle control system is to enter into the command line:

```
python control.py
```

To shutdown the PWM signals after ending the control program, we can use the stop and cleanup functions of the PWM library. Here is an example:

```
import Adafruit_BBIO.PWM as PWM

duty_cycle_pin = "P8_13"
direction_pin = "P8_19"

PWM.stop(duty_cycle_pin)
PWM.stop(direction_pin)

PWM.cleanup()
```

Autorun control.py

Since we will not be able to type into the command line when the Beaglebone Black is mounted on the bicycle, we must have control.py be run as soon as the computer starts up. To do that, create a service file on the Beaglebone that automatically runs control.py on startup. The full step by step instructions for how to do this can be found in Mukund Sudarshan's paper[7].

6 http://goldenmotorcz.en.made-in-china.com/product/xBnJsNZSZAci/China-250W-750W-Electric-Bike-Hub-Motor-Magic-Pie-2X-Motor-.html

7 Using the BeagleBone Black on a robot bicycle. *Mukund Sudarshan.*

Conclusion

This paper describes our approach to creating a control system for a self-stabilizing bicycle using a sense-think-act model of robotics and a proportional-integral-derivative controller. We have made a lot of progress this semester: deriving a model for bicycle stability, acquiring a new steering motor that met our specifications, modifying the steering tube to accommodate this new motor, designing the circuitry to connect the various components of the bicycle, and designing and implementing a control system for the bicycle on the Beaglebone Black. Despite the progress, we still have a lot of work to do next semester, mostly in testing the bicycle to tune the parameters of the controller.

Appendix A : Hardware Components

A1. Microstrain Inertia-Link
Link to Communications Protocol:
http://files.microstrain.com/dcp/Inertia-Link-3DM-GX2-data-communications-protocol.pdf

Link to Datasheet:
http://files.microstrain.com/Inertia-Link%20datasheet%20-%20v%202.pdf

A2. Beaglebone Black RS-232 Serial Micro-Cape
Link to Product:
http://www.logicsupply.com/components/beaglebone/capes/cbb-ttl-232/

A3. Beaglebone Black
Link to Product:
http://beagleboard.org/black

Pin out diagram:
http://www.alexanderhiam.com/wp-content/uploads/2013/06/beaglebone_pinout.png

A4. Golden Motor Magic Pie 2x Motor
Link to Product:
http://goldenmotorcz.en.made-in-china.com/product/xBnJsNZSZAci/China-250W-750W-Electric-Bike-Hub-Motor-Magic-Pie-2X-Motor-.html

Appendix B : Code Documentation

The full code can be found in this repository: https://github.com/marvinliu19/autonomous_bicycle

Below are the list of public functions that are used from each file.

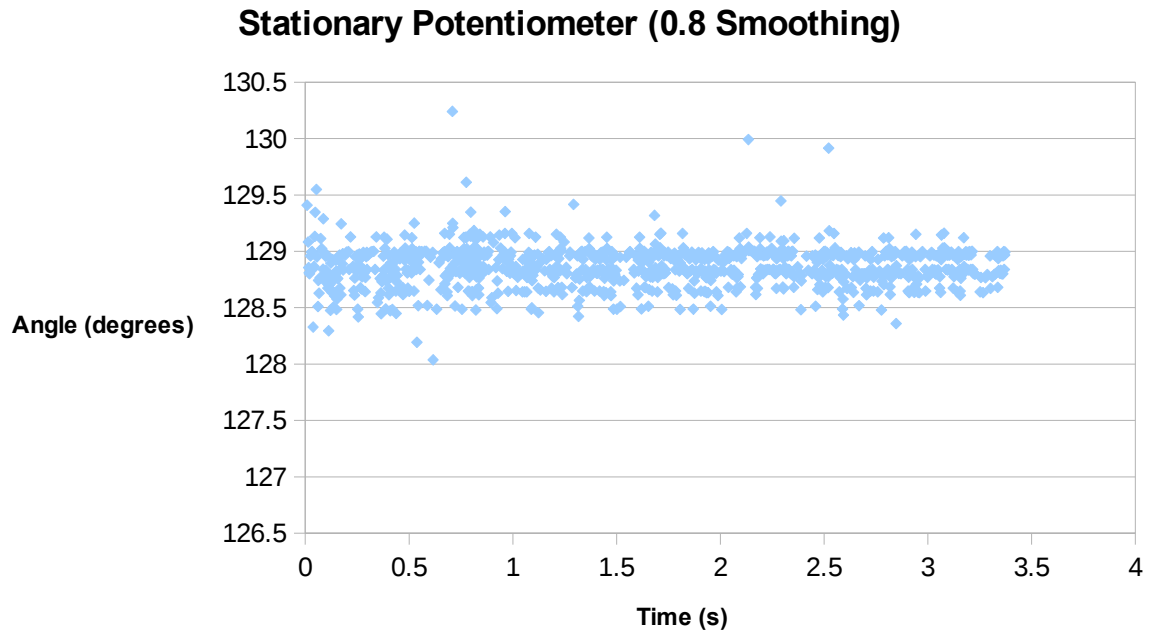
imu.py	
`get_roll_angle_ang_rate(serial_port)`	Gets the current roll angle and angular rates of the IMU **serial_port** - the serial port the IMU is connected to

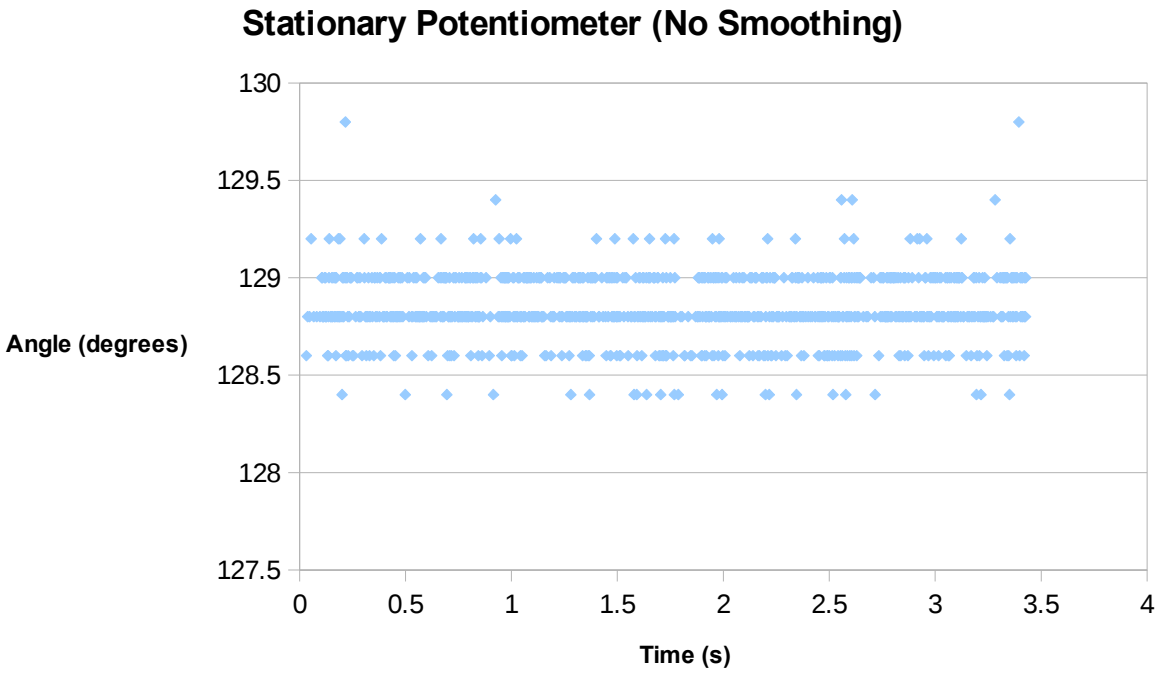
pid.py	
`pid.MotorController(pot, dir_pin, duty_pin, kp, ki, kd, qmax)`	Creates a new MotorController object **pot** - a Potentiometer object used to get the current steer angle **dir_pin** - output pin on BeagleBone that controls motor direction **duty_pin** - output pin on BeagleBone that controls duty cycle **kp** - KP constant in the PID controller **ki** - KI constant in the PID controller **kd** - KD constant in the PID controller **qmax** - the maximum q (wind-up) allowed in the PID controller **return:** MotorController object set in forward direction and duty cycle = 0
`control_motor(target_velocity, start_time)`	Sets the duty cycle and direction of the motor to reach the target velocity **target_velocity** - the velocity the controller is trying to reach **start_time** - the time when the function is called

potentiometer.py	
`potentiometer.Potentiometer(pos_pin, vel_pin)`	Creates a new Potentiometer object **pos_pin** - AIN pin that measures the position of the potentiometer **vel_pin** - AIN pin that measures the velocity of the potentiometer **return:** Potentiometer object that reads from the given pins
`get_angle()`	Reads the current angle of the potentiometer in radians
`get_velocity()`	Reads the current angular velocity of the potentiometer in radians/second
`get_steer_angle()`	Reads the current angle of the potentiometer with exponential filtering Reduces the noise caused by the analog to digital conversion

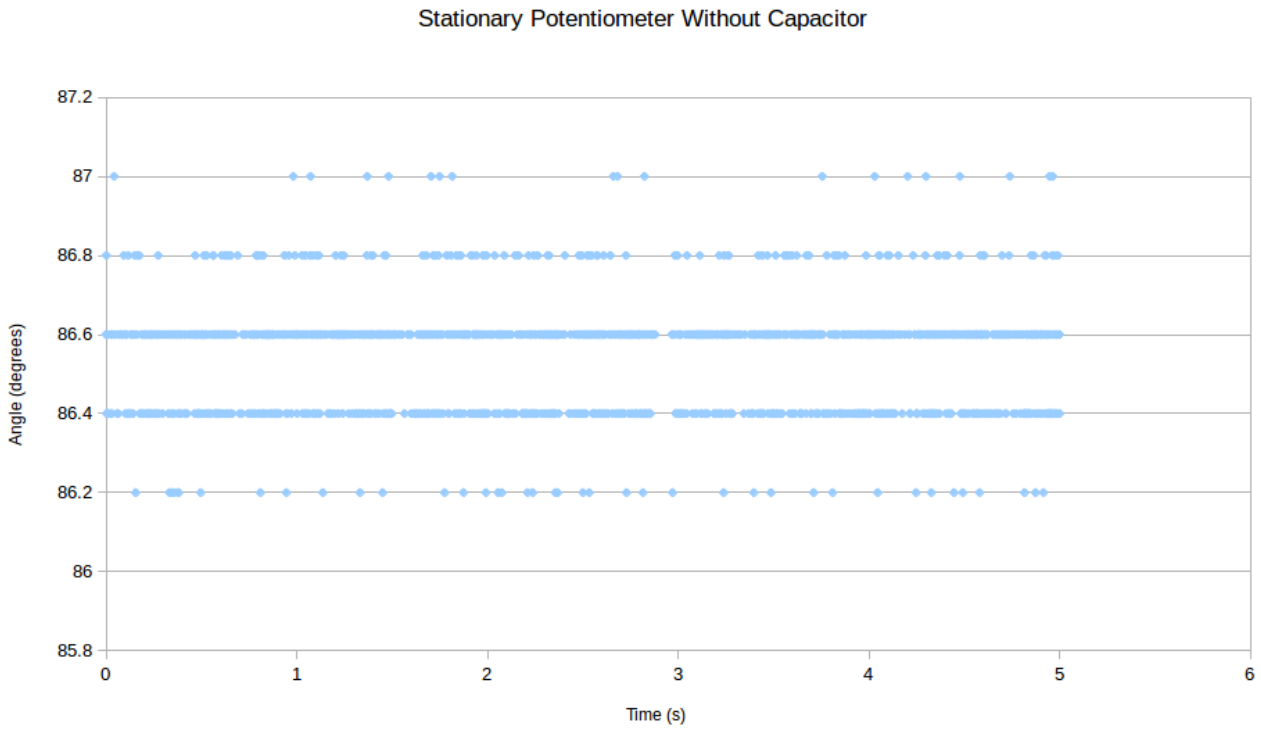
Appendix C : Graphs of Tests

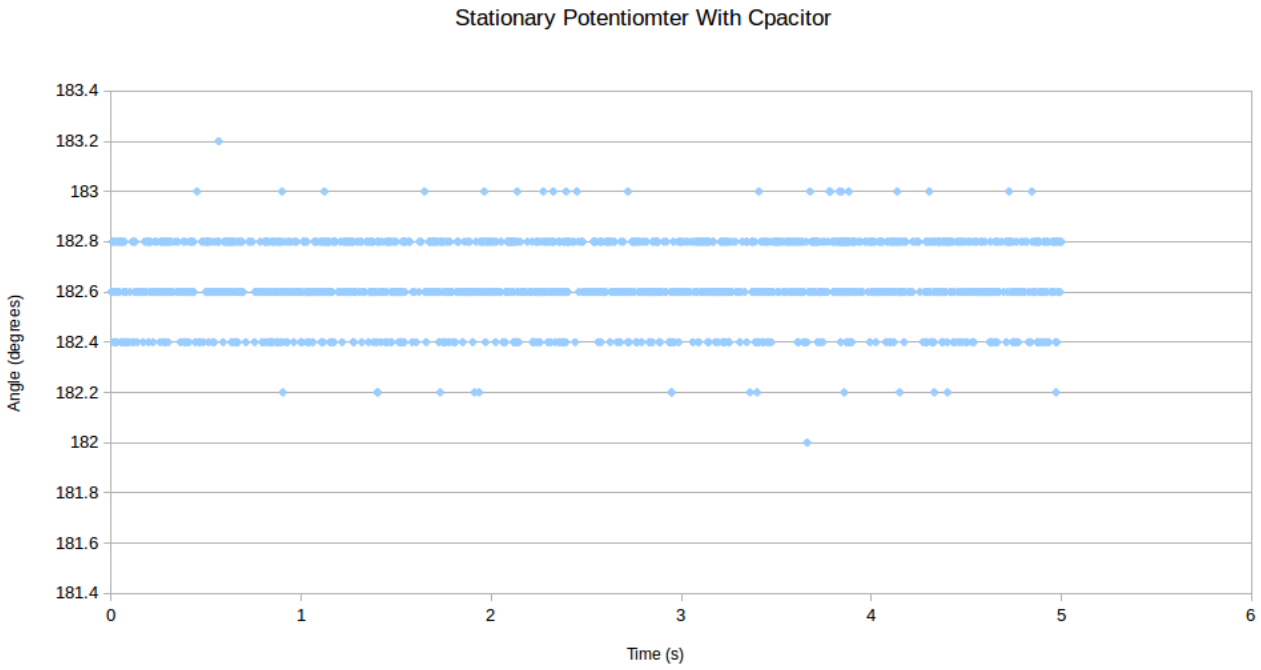
C1. Stationary Potentiometer With and Without Exponential Smoothing



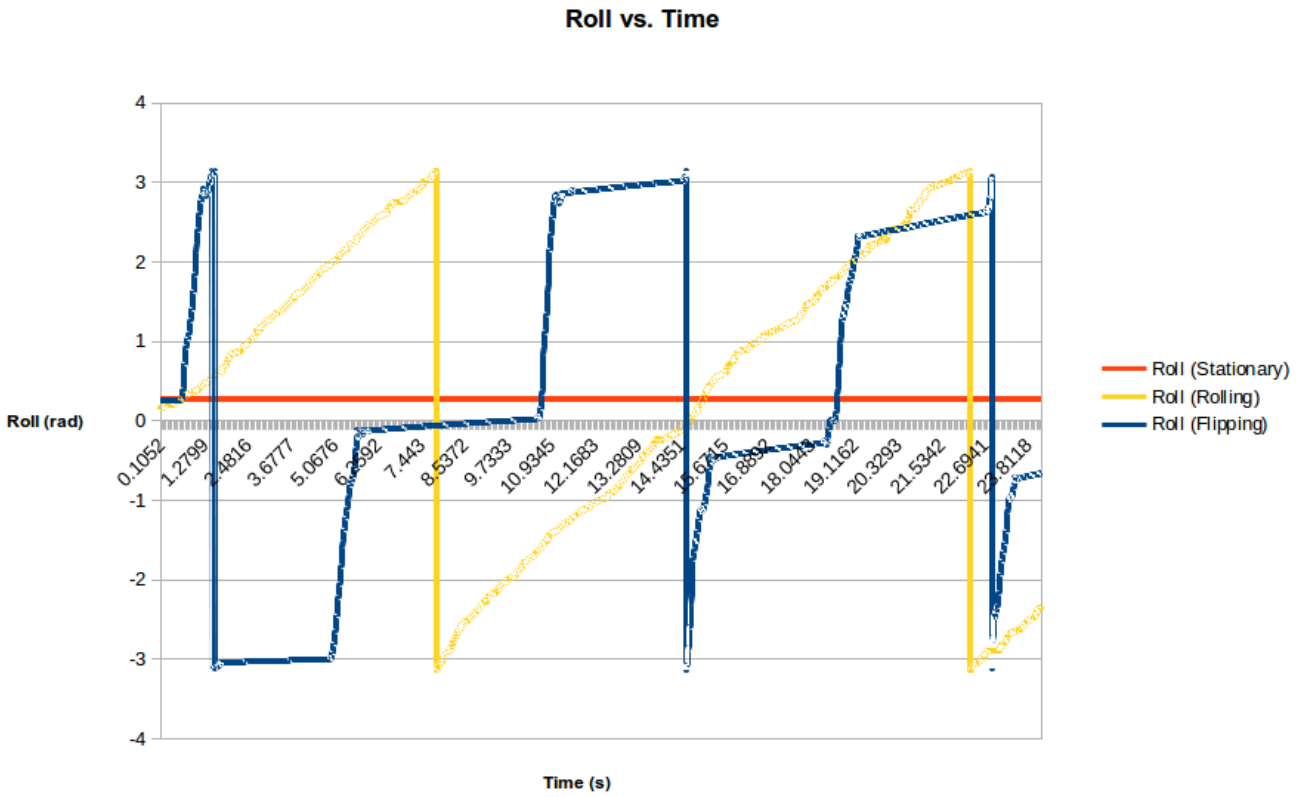


C2. Stationary Potentiometer With and Without Capacitor as Low Pass Filter





C3. IMU Roll Data (Stationary, Rolling, Flipping)



Average Frequency of Data Gathering: 148.19 Hz

Appendix D : References

- Mukund Sudarshan's 2014 paper “Using the BeagleBone Black on a robot bicycle”
  - Extremely helpful with its step by step instructions on how to setup the BeagleBone Black for initial use in this project.

- Ariam Espinal's code done over the summer
  - Helpful in learning the basics of how to communicate with each sensor and provided a good starting point for me in my work this semester

- PySerial Documentation
  - Helpful for serial communication using Python
  - http://pyserial.sourceforge.net/

- MicroStrain Inertia-Link Data Communications Protocol
  - Instructions and commands for interacting with the IMU
  - http://files.microstrain.com/dcp/Inertia-Link-3DM-GX2-data-communications-protocol.pdf