Georgia Institute of Technology
Dr. Sorensen and Dr. Adams
2 December 2019

ME 6404 Final Report – Team 2

Yuen Bak Ching, Clay Dodson, Ethan Kee, Alex Sun

Introduction

The purpose of this project is to create a reaction wheel pendulum that uses a set of flywheels to stabilize itself through adjustments in the angular acceleration of the flywheels. The success of the project will be measured in terms of the pendulum's ability to start and remain upright while resisting mild disturbances and variations in mass balance. The following sections detail the development of the system model in order to determine the dynamic equations of motion, the corresponding hardware selection based on required specifications, the control law development, and the implementation on a physical system.

Part 1 – System Model and Equations of Motion

The first task in the design of the reaction wheel pendulum was to establish a free body diagram of the system. This is shown in Figure 1. below where the states and parameters are labeled as they appear in the diagram.

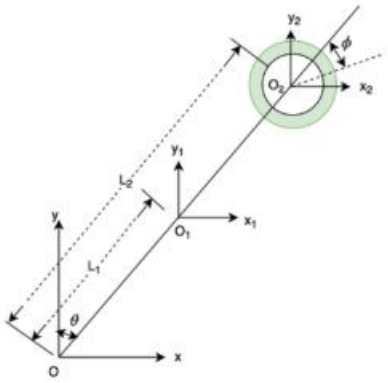

Figure 1. Inverted Pendulum with Reaction Wheel Model [1]

In this case, the two axes of the system were considered independently of one another so the following equations of motion are derived for a single axis but can be applied to either axis. Lagrange's energy method is used as given in the following equations. [1]

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}_i}\right) - \frac{\partial \mathcal{L}}{\partial q_i} = \tau_i \quad where \quad \mathcal{L} = KE - PE$$

Here the generalized coordinates, q_i, are the angles of the pendulum and reaction wheel of the system, θ and ϕ and τ_i is the total torque acting at each of these points. The kinetic and potential energies of the system can then be found as shown below.

$$KE = \frac{1}{2}(m_1 L_1^2 + m_2 L_2^2 + I_1 + I_2)\dot{\theta}^2 + I_2\dot{\theta}\dot{\phi} + \frac{1}{2}I_2\dot{\phi}^2$$

$$PE = (m_1 L_1 + m_2 L_2) g \cos\theta$$

The kinetic and potential energies can then be plugged in to the Lagrange operator to find the equations of motion for the system. These can be restructured into a state space representation as shown below where the variables that make up the A matrix are defined.

$$\begin{bmatrix}\dot{\theta}\\ \ddot{\theta}\\ \ddot{\phi}\end{bmatrix} = \begin{bmatrix}0 & 1 & 0\\ a_{21} & 0 & a_{23}\\ a_{31} & 0 & a_{33}\end{bmatrix}\begin{bmatrix}\theta\\ \dot{\theta}\\ \dot{\phi}\end{bmatrix} + \begin{bmatrix}0\\ b_2\\ b_3\end{bmatrix} \qquad y = \begin{bmatrix}1 & 0 & 0\end{bmatrix}\begin{bmatrix}\theta\\ \dot{\theta}\\ \dot{\phi}\end{bmatrix}$$

$$a_{21} = \frac{b}{a} \qquad a_{23} = \frac{K_t K_e N_g^2}{aR_m}$$

$$a_{31} = -\frac{b}{a} \qquad a_{33} = \left(\frac{K_t K_e N_g^2}{R_m}\right)\left(\frac{a + I_2}{aI_2}\right) \qquad a = m_1 L_1^2 + m_2 L_2^2 + 1$$

$$b = (m_1 L_1 + m_2 L_2) g$$

$$b_2 = -\frac{K_t N_g}{aR_m} \qquad b_3 = \left(\frac{K_t N_g}{R_m}\right)\left(\frac{a + I_2}{aI_2}\right)$$

This state space form is a more meaningful representation of the system as a whole and allows for easier simulation using MATLAB models as will be discussed further in the following sections.

Part 2 – Hardware Selection and Design

The design of the system was created with guidance from Turkmen's paper [1]. The system includes a needle base with motor and flywheel mounts to control the system and a sensor attached to acquire system states. The initial CAD model is shown in Figure 2.

Figure 2. Initial CAD Model

Selecting hardware for the machine involves special consideration for mass and torque. A lower mass results in a system that is more maneuverable while a higher torque increases the ability to move the system. With these key points in mind, two UxCell DC Encoder Gear Motors and an MPU-6050 IMU were chosen to actuate and measure the state of the system. The motor was chosen for its high torque (12 g-cm), its RPM (350 RPM), and its built-in encoder. As stated before, the high torque increases the ability for the system to move itself while the higher RPM allows the motor to accelerate longer before motor saturation which is crucial for generating torque. Motor parameters for the system model are shown in the Table 1. The encoder allows the MCU to collect motor angular velocity which is later used to control the system. The IMU was chosen for its combined gyroscope and accelerometer package which reduced complexity and weight of the system while still providing the needed pendulum angle and pendulum angular

velocity. The actual flywheels need high inertial properties while having low mass. The high inertial properties give the motors more angular resistance therefore generating more torque while the low mass reduces the flywheel's encumbrance on the system. As a result, the flywheel was designed to be a thin wheel made of lightweight aluminum 6061.

Table 1. Motor Constants

Back EMF Constant (K_e)	0.001 V-s/rad
Motor Torque Constant (K_t)	0.0095 N-m/A
Armature Resistance (R_a)	2.2 Ω
Gear Ratio (N_g)	34:1

Part 3 – Control Law Development and Simulation

LQR control was utilized to enable the pendulum to stabilize in an upright orientation and drive the rotational velocity of the reaction wheels to zero while being robust enough to handle reasonable external disturbances. The R value is chosen to be 1/100 and Q matrix of the LQR controller is chosen by penalizing each different state and the values are as follows:

q_1 = 1/500 (Pendulum ω) • q_2 = 1/500 (Flywheel angle) • q_3 = 1/500 (Flywheel angle velocity)

$$Q = \begin{bmatrix} q_1 & 0 & 0 & 0 \\ 0 & q_2 & 0 & 0 \\ 0 & 0 & q_3 & 0 \\ 0 & 0 & 0 & q_4 \end{bmatrix}, R = 1/100$$

Using the lqr function in MATLAB with the A and B matrix from the state-space model and the Q and R parameters defined earlier, the gains for the LQR controller is calculated to be:

K = [0.8597, 0.1342, 0.0432]

A simulation environment was developed using MATLAB in order to analyze the stability of the system and impact of the controller. The simulation result with the calculated LQR gains is displayed in Figure 3.

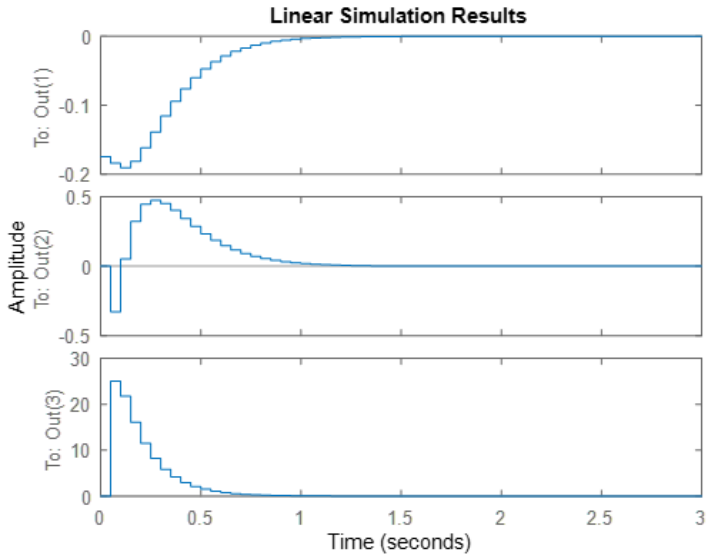

Figure 3. Simulation Results from with LQR gain

Part 4 – Fabrication and Implementation

The self-balancing stick consisted of mainly three parts: the actuator with flywheels, the sensors and the micro-controller.

Starting off with the actuator and flywheel part, the flywheel is initially a water-jetted aluminum wheel of 110 mm diameter and weighs roughly 81 grams. After several tests, the moment of inertia of the flywheel is determined to be insufficient and another steel flywheel with 203.2 mm diameter and 174 grams is water-jetted to replace the original flywheel. The actuators are the two UxCell DC Encoder Gear Motors selected and it is fastened onto the pendulum by a water-jetted aluminum bracket. The flywheel is secured on the motor by a setscrew. The sensor is the IMU (MPU-6050 accelerometer and gyroscope) which is housed on top of pendulum to effectively measure the angle of the pendulum. The micro-controller (MSP-432) is connected to the motors by a dual H-bridge driver and the motor is powered by a 12-V power supply. The sensor, on the other hand, is powered by a 5-V power and is connected directly to the controller using I2C communication protocol. The initial setup, final CAD design and setup is shown below in Figure 4.

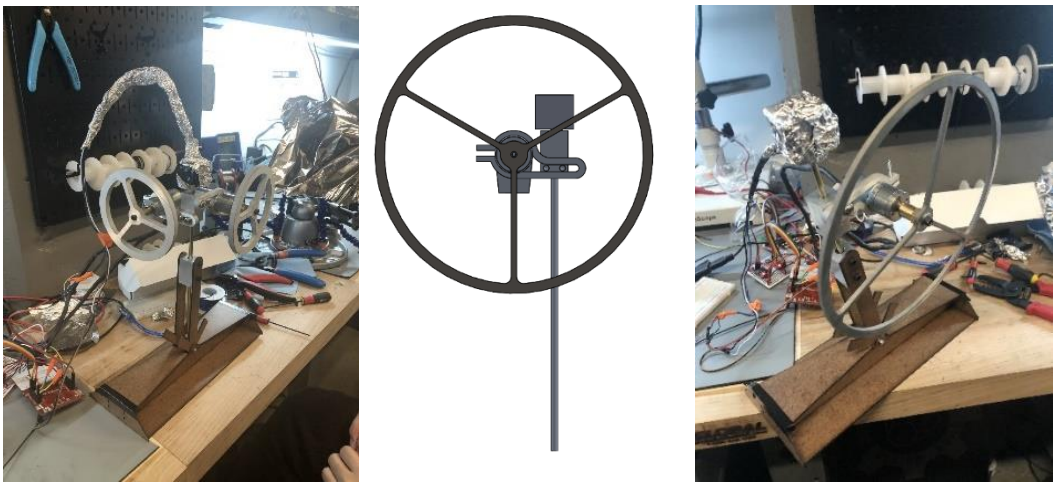

Figure 4. Initial Test Setup (Left) Final CAD design (Middle) Final Setup (Right)

In order to get accurate results from the accelerometer and gyroscope, the readings obtained from them is integrated by a complementary filter as depicted below.

$$Pitch = 0.98 * \left(Pitch_{prev} + Ygyro * dT\right) + (0.02) * (\text{atan}\left(\frac{Y_{accel}}{\sqrt{{X_{accel}}^2 + {Z_{accel}}^2}}\right))$$

$$Roll = 0.98 * \left(Roll_{prev} + Xgyro * dT\right) + (0.02) * (\text{atan}\left(\frac{X_{accel}}{\sqrt{{Y_{accel}}^2 + {Z_{accel}}^2}}\right))$$

The IMU calculated the current angle of the pendulum and with the desired angle along with the LQR gain, the PWM signal is calculated and sent to the motor. The motor should output just enough power to correct the pendulum and hold it in the desired position and attempt to drive all the states to zero.

The self-balancing stick worked with extremely small disturbances (~1-2 N) which is lower than the goal and there were several problems that still needs to be tackled which will be discussed in the following section.

Part 5 – Challenges and Solutions

Due to a variety of nonlinearities in the system as well as some unanticipated physical and electronic limitations, several challenges were encountered in the implementation and testing of the system. The first of these was interference in sensor output caused by the electromagnetic field radiating from the DC motors at high RPM. This interference would corrupt the readings from the IMU and in some cases cause it to drop out completely. This would result in failure of the system as the flywheel would be unable to correct for changes in pendulum angle as the angle of the system is no longer accurate and updating. The solution for this problem was to add electromagnetic shielding. This was done by wrapping the sensor and lead wires with a layer of velostat and aluminum foil to block any potential interference in the signal. This quick fix provided drastic improvement in the IMU's ability to track changes in pendulum even at high motor RPM. The IMU will still drop out from time to time, and to eliminate the effect of

electromagnetic interferences entirely, shielded cables should be used, and grounded metal sheets should wrap around the IMU.

Another issue very similar to the first one is the ground jitter/noise. Since the ground reference is having a lot of noise, it will cause the IMU measurements to have a lot of noise as well. The solution is to use a good power supply with an isolated ground.

The third issue encountered was inadequate reactionary force provided by the flywheel. One potential solution was to simply make a heavier flywheel, but this would potentially hinder the motor ability to provide angular accelerations required by the control law due to the limited torque available. The second solution was to make a larger flywheel with a greater moment of inertia. This would provide greater reactionary forces without causing the motor to saturate at lower RPM.

Once the physical limitations of the design had been overcome, the nonlinearity effects of the motor had to be considered. The first two were saturation and rate-limit. Rate limiting was encountered when the motor was not physically able to accelerate as fast as required by the control law. Figure 5 below shows the actual motor RPM over time for maximum acceleration of the flywheel. The time of 0.05 seconds to reach the maximum RPM for the physical system can be compared to the simulated system in Figure 3 where the control law takes the system to maximum RPM in only 0.02 seconds. This slower response time can result in instability of the system and buildup of error over time.

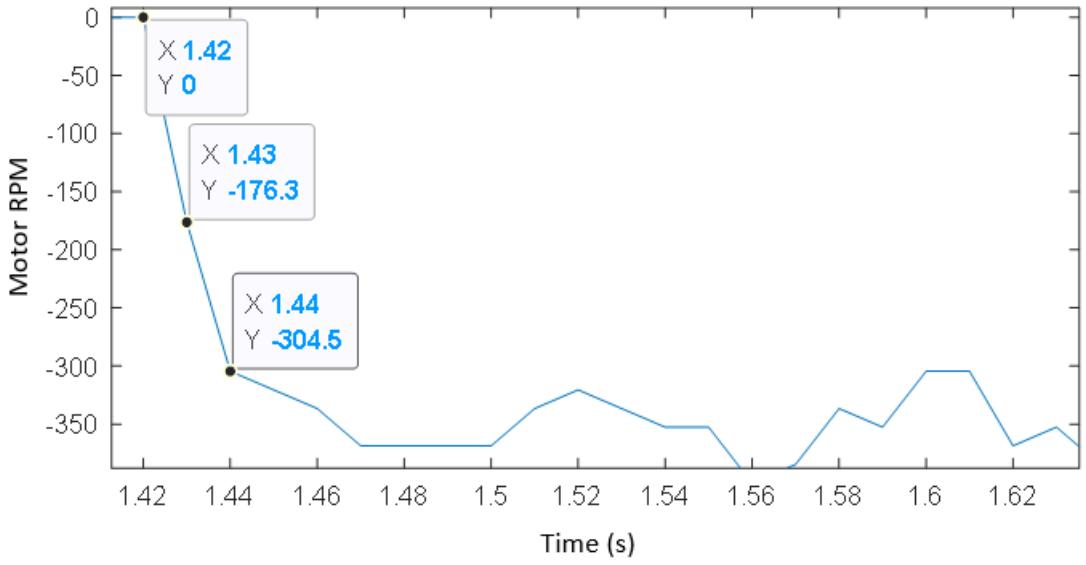

Figure 5. Motor RPM over time to show impacts of nonlinearities on system

The nonlinearity effects caused by saturation can also be seen in the figure above. While the motor is only capable of reaching a maximum angular velocity of about 350 RPM, the control law could easily require more than this in order to compensate for external disturbances. When rate-limiting logic is applied to the control law, these saturation limits were frequently exceeded resulting in failure of the system to maintain an upright inverted orientation.

The final nonlinearity effect encountered was the motor dead zone. Due to the small duty cycle of the selected motor, it was unable to provide adequate precision in terms of angular

position control of the flywheel to accommodate fine adjustments of the pendulum angle. This effect lead to further instability in the system and was difficult to accurately model in the simulated environment.

Part 6 –Conclusions

Despite numerous challenges encountered due to the physical limitations of the hardware used as well as electromagnetic interference from the DC motors during testing, successful balancing of the inverted pendulum for a single degree of freedom was achieved. The final system was robust enough to maintain the inverted stability for extended periods of time as well as accommodate light external disturbances. While the initial goal of achieving stability with multiple degrees of freedom for large disturbances within a wide range of angles was not met, the project was successful in furthering the stated purpose of the course of developing intuition and a greater understanding of the pragmatic implementations of controls.

With a longer timeframe, it is likely that the stated goals of the project could be reached by further optimizing the flywheel weight and inertial properties. Additionally, providing better shielding for the IMU sensor would provide more accurate readings with less noise in the signal caused by electromagnetic interference from proximity to the DC motors.

Works Cited

[1] Turkmen, Abdullah, et al. *Design, Implementation and Control of Dual Axis Self Balancing Inverted Pendulum Using Reaction Wheels*. *Design, Implementation and Control of Dual Axis Self Balancing Inverted Pendulum Using Reaction Wheels*.

Appendix I: Important Feedback

Yuen Bak Ching

- Make sure the data is significant before plotting it out, or else, don't bother to plot it
- Diagrams to accompany state-space representation to make it more understandable
- Create a variety of plots such as bode, bar-chart to help visualize the data instead of just purely displaying the data which can be confusing.

Clay Dodson

- Reducing figure count by summarizing findings in bar graphs/tables.
- Including equations in the body of the report leads to better flow and clarity.
- Describe not only the steady state responses, but also transient/disturbed behavior and call out these features out on the graphs.

Ethan Kee

- Properly identify and label disturbances and response timeframes on system response plots. Be specific about the form of disturbance shown on plots, whether it is an impulse, unit step, etc.
- Make sure that figures and tables are properly discussed in the text and appear in the paper or appendix in the order in which they are discussed
- Include a diagram of the system model with states and parameters labeled and specified to go with calculated equations of motion

Alex Sun

- Don't reference more than one figure at a time.
- If there is a discrepancy then make sure to explain it.
- Always consider various forms of errors

Appendix II: Controller Code

Main.c

```
#include "msp.h"
#include "imu.h"
#include "driverlib.h"
#include <stdio.h>

#include "imu.h"
#include "motors.h"
#include "serial.h"
#include "timer.h"

/**
 * main.c
 */
float* angles;
float dutyConversion(float u1);

void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer
    // Get microcontroller running as fast as possible

    //make as fast as possible
    FlashCtl_setWaitState( FLASH_BANK0, 2);
    FlashCtl_setWaitState( FLASH_BANK1, 2);
    PCM_setPowerState( PCM_AM_DCDC_VCORE1 );
    CS_setDCOCenteredFrequency( CS_DCO_FREQUENCY_48 ); // S_DCO_FREQUENCY_48
    CS_setDCOFrequency(48000000);//48000000
    CS_initClockSignal(CS_MCLK, CS_DCOCLK_SELECT, 1);
    CS_initClockSignal(CS_SMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_16); //16
    CS->KEY = 0;

    //for floating point calculations
    FPU_enableModule();

    angles = (float*) malloc(sizeof(float) * 4);

    controls();
}

//ACTUAL CODE
void controls()
{
    init_MPU6050();
    initMotors();
    initEncoder();
    init_timer();
    init_uart();
    //Timing and uart buffer
    int prevTime = 0;
    int dataTime = 0;
    int sampleTime = 0;
```

```
char data[20];
//states
float speed1 = 0;
float speed2 = 0;
float stateMat1[3];
float stateMat2[3];

//Gain
float K[3] = {18, 3.42, 1 } ;  //change THIS!!!

float u1 = 0; //inputs
float u2 = 0;
while (true)
{
    int currentTime = micros();

    if (currentTime - sampleTime > 5000) {
        // in
        float dT = ((float)(currentTime - sampleTime)) / 100000;
        speed1 = ((((float) getSpeed1()) / dT)) / 374; // rev/sec
        speed2 = ((((float) getSpeed2()) / dT)) / 374;

        setSpeed1(0);
        setSpeed2(0);
        //
        getAngles(angles, 2548, -3138, -868, -255, -255);
        stateMat1[0] = angles[0] * 3.14 / 180;
        stateMat1[1] = angles[2] * 3.14 / 180;
        stateMat1[2] = speed1 * 2*3.14;
        stateMat2[0] = angles[1] * 3.14 / 180;
        stateMat2[1] = angles[3] * 3.14 / 180;
        stateMat2[2] = speed2 * 2 * 3.14;
        //input calculations
        u1 = K[0] * stateMat1[0] + K[1] * stateMat1[1] + K[2] * stateMat1[2];
        u2 = K[0] * stateMat2[0] + K[1] * stateMat2[1] + K[2] * stateMat2[2];
        //duty cycle conversion
        u1 = dutyConversion(u1);
        u2 = dutyConversion(u2);
        setSpeeds(0, u1); // u1);
        //setSpeeds(u1, u2);
        sampleTime = currentTime;
    }

    if (currentTime - dataTime > 10000)
    {
        //send data every .1 second
        sprintf(data, "angleY: %2.4f", angles[0]);
        sendData(data);
        sprintf(data, "angleX: %2.4f", angles[1]);
        sendData(data);
        sprintf(data, "vY: %2.4f", angles[2]);
        sendData(data);
        sprintf(data, "vX: %2.4f", angles[3]);
        sendData(data);
        sprintf(data, "speed1: %2.4f", speed1 * 60);
```

```
            sendData(data);
            sprintf(data, "speed2: %2.4f", speed2 * 60);
            dataTime = currentTime;
        }
    }
}

float dutyConversion(float u1)
{
    if (u1 < 0) {
        u1 = (-u1 * 0.0776) + 0.1178;
        u1 = -1* u1;
    } else {
        u1 = (u1 * 0.0776) + 0.1178; //convert voltage to duty cycle based on 12V
source
    }

    if (u1 > 1) {
        u1 = 1;
    } else if (u1 < -1) {
        u1 = -1;
    }

    return u1;
}

// This works!!!
void testPolling()
{
    initMotors();
    initEncoder();
    init_timer();
    init_uart();
    int prevTime = 0;
    int dataTime = 0;
    int sampleTime = 0;
    float speed1 = 0;
    float speed2 = 0;
    char data[20];

    setSpeeds(.5, .5);
    int t = 0;
    while (t < 1000000)
    {
        int currentTime = micros();
                    pollingEncoder();
                    if (currentTime - sampleTime > 1000) {
                        // in
                        float dT = ((float)(currentTime - sampleTime)) / 100000;
                        speed1 = ((((float) getSpeed1()) / dT)) / 374; // rev/sec
                        speed2 = ((((float) getSpeed2()) / dT)) / 374;
                        sampleTime = currentTime;
                        setSpeed1(0);
                        setSpeed2(0);
                    }
```

```
                    if (currentTime - dataTime > 10000)
                    {
                        //send data every .1 second
                        sprintf(data, "speed1: %2.4f", speed1 * 60);
                        sendData(data);
                        sprintf(data, "speed2: %2.4f", speed2 * 60);
                        sendData(data);
                        dataTime = currentTime;
                    }
                    t++;
    }
    setSpeeds(1, 1); //full speed should be 350 rpm
    while(true)
       {
            int currentTime = micros();
            pollingEncoder();

            if (currentTime - sampleTime > 1000) {
                // in
                float dT = ((float)(currentTime - sampleTime)) / 100000;
                speed1 = ((((float) getSpeed1()) / dT)) / 374; // rev/sec
                speed2 = ((((float) getSpeed2()) / dT)) / 374;
                sampleTime = currentTime;
                setSpeed1(0);
                setSpeed2(0);
            }
            if (currentTime - dataTime > 10000)
            {
                //send data every .1 second
                sprintf(data, "speed1: %2.4f", speed1 * 60);
                sendData(data);
                sprintf(data, "speed2: %2.4f", speed2 * 60);
                sendData(data);
                dataTime = currentTime;
            }
       }
}
// TEST FUNCTIONS
void testEncoders()
{
    initMotors();
    initEncoder();
    init_timer();
    init_uart();
    setSpeeds(1, 1); //full speed should be 350 rpm
    int prevTime = 0;
    int dataTime = 0;
    int sampleTime = 0;
    float speed1 = 0;
    float speed2 = 0;
    char data[20];
    while(true)
    {
        int currentTime = micros();
        //float dT = ((float)(currentTime - prevTime)) / 100000;
```

```
        prevTime = currentTime;

        if (currentTime - sampleTime > 10000) {
            // in
            float dT = ((float)(currentTime - sampleTime)) / 100000;
            speed1 = ((((float) getSpeed1()) / dT)) / 374; // rev/sec
            speed2 = ((((float) getSpeed2()) / dT)) / 374;
            sampleTime = currentTime;
        }
        if (currentTime - dataTime > 10000)
        {

            //send data every .1 second
            sprintf(data, "speed1: %2.4f", speed1 * 60);
            sendData(data);
            sprintf(data, "speed2: %2.4f", speed2 * 60);
            sendData(data);
            dataTime = currentTime;
        }
    }
}

void testIMU()
{
    init_MPU6050();
    init_timer();
    init_uart();
    int prevTime = 0;
    int dataTime = 0;
    int sampleTime = 0;
    char data[20];
    while(true)
    {
        int currentTime = micros();
        int dT = currentTime - prevTime;
        prevTime = currentTime;

        //getAngles(angles, 0, 0, 0, 0, 0);
        /**
        angles[0] = totalAngleY;
        angles[1] = totalAngleX;
        angles[2] = yGyro;
        angles[3] = xGyro;
        */
        if (currentTime - sampleTime > 1000) {
            getAngles(angles, 0, 0, 0, 0, 0);
            sampleTime = currentTime;
        }
        if (currentTime - dataTime > 10000)
        {
            //send data every .1 second
            sprintf(data, "angleY: %2.4f", angles[0]);
            sendData(data);
            sprintf(data, "angleX: %2.4f", angles[1]);
            sendData(data);
```

```
            sprintf(data, "vY: %2.4f", angles[2]);
            sendData(data);
            sprintf(data, "vX: %2.4f", angles[3]);
            sendData(data);
            dataTime = currentTime;
        }
    }
}
/**
void calibration() {
    int t = 0;
    int* values;
    calibrations[0] = 0;
    calibrations[1] = 0;
    calibrations[2] = 0;
    calibrations[3] = 0;
    calibrations[4] = 0;
    while(t < 5000) {
        values = getReadings();

        calibrations[0] = values[0] + calibrations[0];
        calibrations[1] = values[1] + calibrations[1];
        calibrations[2] = values[2] + calibrations[2];
        calibrations[3] = values[3] + calibrations[3];
        calibrations[4] = values[4] + calibrations[4];
        t++;
    }
    calibrations[0] = -calibrations[0] / t;
    calibrations[1] = -calibrations[1] / t;
    calibrations[2] = -(calibrations[2] / t) + 16383;
    calibrations[3] = -calibrations[3] / t;
    calibrations[4] = -calibrations[4] / t;
}

*/
```

Imu.c

```
#include "msp.h"
#include "driverlib.h"
#include "imu.h"
#include "timer.h"
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

/*
 * imu.c
 *
 *  Created on: Nov 28, 2019
 *      Author: clayd
 */
#define IMU_ADDRESS 0x68
#define TEMP_TIMEOUT 50000
```

```
float totalAngleX;
float totalAngleY;
int prevTime;
int16_t* data;
void icInterupt();
void resetI2C();

const eUSCI_I2C_MasterConfig I2C_MasterConfig =
{
    EUSCI_B_I2C_CLOCKSOURCE_SMCLK, //uint_fast8_t selectClockSource;
    3000000, //kHz  uint32_t i2cClk;
    EUSCI_B_I2C_SET_DATA_RATE_400KBPS, //uint32_t dataRate;
    0, //uint_fast8_t byteCounterThreshold;
    EUSCI_B_I2C_NO_AUTO_STOP //uint_fast8_t autoSTOPGeneration;
};



void init_MPU6050()
{
    MAP_Interrupt_disableMaster();
    //scl
    GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P6, GPIO_PIN5,
GPIO_PRIMARY_MODULE_FUNCTION);
    //sda
    GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P6, GPIO_PIN4,
GPIO_PRIMARY_MODULE_FUNCTION);
    MAP_I2C_initMaster(EUSCI_B1_BASE, &I2C_MasterConfig);
    MAP_I2C_setSlaveAddress(EUSCI_B1_BASE, IMU_ADDRESS);
    MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_TRANSMIT_MODE);
    MAP_I2C_enableModule(EUSCI_B1_BASE);
    //MAP_I2C_masterSendStart(EUSCI_B1_BASE);
    //MAP_I2C_masterSendMultiByteNext(EUSCI_B1_BASE, 0x00);
    //MAP_I2C_masterSendMultiByteStop(EUSCI_B1_BASE);
    MAP_I2C_clearInterruptFlag(EUSCI_B1_BASE, EUSCI_B_I2C_RECEIVE_INTERRUPT0);
    I2C_clearInterruptFlag(EUSCI_B1_BASE, EUSCI_B_I2C_TRANSMIT_INTERRUPT0 +
EUSCI_B_I2C_NAK_INTERRUPT);

    MAP_I2C_enableInterrupt(EUSCI_B1_BASE, EUSCI_B_I2C_NAK_INTERRUPT);
    //GPIO_registerInterrupt(EUSCI_B1_BASE, icInterupt);


    // Clear the 'sleep' bit to start the sensor.

    writesBytes(MPU6050_PWR_MGMT_1, 0x00);
    // make sure accelerometer and gyro is enabled
    writesBytes(MPU6050_PWR_MGMT_2, 0x00);
    /* Enabling MASTER interrupts */
    MAP_Interrupt_enableMaster();

    data = (int16_t*) malloc(sizeof(int16_t));
    prevTime = 0;
}
```

```
void getAngles(float* angles, int xAoff, int yAoff, int zAoff,int xGoff, int yGoff)
{
    int currentTime = micros();
    float dT = currentTime - prevTime;
    prevTime = currentTime;
    dT = dT / 100000; // convert to sec

    float xAccel = 0;
    float yAccel = 0;
    float zAccel = 0;
    float xGyro  = 0;
    float yGyro  = 0;
    float zGyro  = 0;
    int16_t val =  0;
    float value =  0;

    readsBytes(0x3B, data); //0x3D should be 0x3B
    //trying to trick it
    readsBytes(0x3D, data); //0x3D should be 0x3B
    val = *data;
    // angles 2g
    value = (float) val + xAoff;
    value = 2*(value / 32767);
    xAccel = value;

    readsBytes(MPU6050_ACCEL_ZOUT_H, data); //0x3F
    val = *data;
    // angles 2g
    value = (float) val + yAoff;
    value = 2*(value / 32767);
    yAccel = value;


    readsBytes(0x41, data); //for whatever reason 0x41 works
    val = *data;
    // angles 2g
    value = (float) val  + zAoff;
    value = 2*(value / 32767);
    zAccel = value;

    readsBytes(MPU6050_GYRO_XOUT_H , data);
    //Gyro
    readsBytes(MPU6050_GYRO_YOUT_H , data);
    val = *data;
    // 250Deg/s is full scale
    value = (float) val  + xGoff;
    value = 250*(value / 32767);
    xGyro = value;


    readsBytes(MPU6050_GYRO_ZOUT_H, data);
    val = *data;
    // angles 2g
    value = (float) val  + yGoff;
    value = 250*(value / 32767);
```

```
    yGyro = value;

    totalAngleY = .98*(totalAngleY + yGyro*dT)  + (180 /
3.141593)*.02*atan(yAccel/sqrt(xAccel*xAccel + zAccel*zAccel));
    totalAngleX = .98*(totalAngleX + xGyro*dT)  + (180 /
3.141593)*.02*atan(xAccel/sqrt(yAccel*yAccel + zAccel*zAccel));
    angles[0] = totalAngleY;
    angles[1] = totalAngleX;
    angles[2] = yGyro;
    angles[3] = xGyro;
}

// i2c protocols

int writesBytes(uint8_t address, uint8_t data)
{
    while(I2C_isBusBusy(EUSCI_B1_BASE)==EUSCI_B_I2C_BUS_BUSY);
    MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_TRANSMIT_MODE);
    bool result = MAP_I2C_masterSendMultiByteStartWithTimeout(EUSCI_B1_BASE, address,
TEMP_TIMEOUT);
    while(I2C_isBusBusy(EUSCI_B1_BASE)==EUSCI_B_I2C_BUS_BUSY);
    //while(UCTXIFG = 0  );

    result = MAP_I2C_masterSendMultiByteNextWithTimeout(EUSCI_B1_BASE, data,
TEMP_TIMEOUT);
    while(I2C_isBusBusy(EUSCI_B1_BASE)==EUSCI_B_I2C_BUS_BUSY);
    result = MAP_I2C_masterSendMultiByteStopWithTimeout(EUSCI_B1_BASE, TEMP_TIMEOUT);
    while(I2C_isBusBusy(EUSCI_B1_BASE)==EUSCI_B_I2C_BUS_BUSY);
    if (!result)
    {
        resetI2C();
    }
    return (int) result;
}


int readsBytes(uint8_t address, int16_t* data)
{
    uint8_t valueA[2];
    while(I2C_isBusBusy(EUSCI_B1_BASE)==EUSCI_B_I2C_BUS_BUSY);
    MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_TRANSMIT_MODE);
    //MAP_Interrupt_disableMaster();
    bool result = MAP_I2C_masterSendSingleByteWithTimeout(EUSCI_B1_BASE, address,
TEMP_TIMEOUT);
    while(I2C_isBusBusy(EUSCI_B1_BASE)==EUSCI_B_I2C_BUS_BUSY);
    int i = 0;
    for(i = 0; i < 5000; i++){}
    if (result == true)
    {
        MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_RECEIVE_MODE);
        MAP_I2C_masterReceiveStart(EUSCI_B1_BASE);
        while(I2C_isBusBusy(EUSCI_B1_BASE)==EUSCI_B_I2C_BUS_BUSY);
        //for(i = 0; i < 1000; i++){}
        valueA[0] =
MAP_I2C_masterReceiveMultiByteNext(EUSCI_B1_BASE);//MultiByteNext(EUSCI_B1_BASE);
```

```
        while(I2C_isBusBusy(EUSCI_B1_BASE)==EUSCI_B_I2C_BUS_BUSY);
        //for(i = 0; i < 1000; i++){}
        result = MAP_I2C_masterReceiveMultiByteFinishWithTimeout(EUSCI_B1_BASE,
&(valueA[1]), TEMP_TIMEOUT);
        if (result == true)
        {
            int16_t High = (int16_t) valueA[0];
            int16_t Low = (int16_t) valueA[1];
           // int sign =
            *data = (High << 8) | Low;
        } else
        {
            int ab = 0;
        }

    }
    return (int) result;
}
    //MAP_Interrupt_enableMaster();
//return result;

void resetI2C()
{


}


void icInterupt()
{
    MAP_I2C_clearInterruptFlag(EUSCI_B1_BASE, EUSCI_B_I2C_NAK_INTERRUPT);
    int t = 0;
}
```

Appendix III: MATLAB Code

Simulation

```
m1 = .330; % mass of everything except flywheel
m2 = .071; % mass of reaction wheel
I1 = .000599596; % moment of inertia of pendulum about COG
I2 = 0.000147; % moment of inertia of flywheel
L1 = .214; % moment arm length to COG of pendulum
L2 = .22; % moment arm length to COG of flywheel
Ke = 0.00963; % back emf
Kt = .006295; % motor torque constant
Rm = 2.2; % resistance
Ng = 34;

% error adjustment

m1 = m1.* 1;
m2 = m2 .* 1;
I1 = I1 .* 1;
I2 = I2 .* 1;
L1 = L1 .* 1;
L2 = L2 .* 1;
Ke = Ke .* 1;
Kt = Kt .* 1;
Rm = Rm .* 1;

Ts = .01; % in sec
delay = 0; % in sec
g = 9.81;

a = m1.*(L1.^2) + m2.*(L2.^2) + I1;
b = (m1.*L1 + m2.*L2).*g;

a21 = b ./ a;
a24 = (Kt.*Ke.*(Ng.^2)) ./ (a.*Rm);
a41 = -(b ./ a);
a44 = ((a + I2) ./ (a.*I2)).*((Kt.*Ke.*(Ng.^2))./Rm);

b2 = -(Kt.*Ng)./(a.*Rm);
b4 = ((a+I2)./(a.*I2)).*((Kt.*Ng)./Rm);



A = [0   1 0   0;
     a21 0 0 a24;
     0   0 0   1;
     a41 0 0 a44];

 B = [0 b2 0 b4]';
 C = eye(4);

 D = [0 0 0 0]';

%  C = [1 0 0 0];
%  D = 0;
```

```
 sysc = ss(A,B,C,D);
 sysX = c2d(sysc, Ts);
 sysX.InputDelay = delay;

 % LQR Control
 q1 = 1./(10);
 q2 = 1./ (500);
 q3 = 1./(100); %dont need to penalize position
 q4 = 1* 1./(5000); % only want angular velocity to go to zero
 Q = [q1 0 0 0;
      0  q2 0 0;
      0  0  q3 0;
      0  0   0 q4];
 R = 1/100;
 Q = Q*.001
 % end LQR control


 %eig(A)

 % pole placement
 p1 = -.13
 p2 = -.1
 p3 = -.05
 p4 = -.06
 K = place(A,B,[p1 p2 p3 p4]);
% LQR
[K,s,p] = dlqr(sysX.A, sysX.B, Q, R);
% K calculated

 sys_cl = ss(sysX.A-sysX.B.*K,sysX.B,sysX.C,sysX.D,Ts);

 t = 0:Ts:100;
 u = ones(1,length(t));
 u = [u.*0]

lsim(sys_cl, u, t, [-5*pi./180 0 0 0]')
```

LQR

```
% CONSTANTS (measured)
Lp = 0;                 % Length to pendulum c.g
Lw = 0;                 % Length to wheel c.g
mp = 0;                 % Mass of pendulum
mw = 1;                 % Mass of wheel
Ip = 0;                 % Pendulum moment of inertia
Iw = 0;                 % Wheel moment of inertia

% CONSTANTS (data sheets)
g  = 9.81;              % Gravity
Ng = 34;                % Gear Ratio
Rm = 0;                 % Armature Coil Resistance (should be quite low <1)
                        % directly measure the leads of motor should be
                        % accurate enough but not 100% accurate
Kt = 0;                 % Motor Torque Constant (Should be ke=kt)
Ke = 0;                 % Motor back e.m.f
```

```
%https://www.precisionmicrodrives.com/content/reading-the-motor-constants-
from-typical-performance-characteristics/

%To simplify
a = mp*Lp^2 + mw*Lw^2 + Ip;
b = (mp*Lp + mw*Lw)*g;

A = [0 1 0 0; b/a 0 0 Kt*Ke*Ng^2/a/Rm; 0 0 0 1; -b/a 0 0
(a+Iw)/a/Iw*Kt*Ke*Ng^2/Rm];
B = [0; -Kt*Ng/a/Rm; 0; (a+Iw)/a/Iw*Kt*Ng/Rm];
R = 1;

q1max = 2*pi/180;
q2max = 1;
q3max = 2*pi/180;
q4max = 1;
Q1 = [1/q1max 0 0 0; 0 1/q2max 0 0; 0 0 1/q3max 0; 0 0 0 1/q4max];

U_lqr = lqr(A,B,Q1,R)*-Q;
```

Real Time Plotting

```
% Code for reading serialport realtime
% change this to whatever port you have
SerialPort = '/dev/cu.Bluetooth-Incoming-Port';


s = serial(SerialPort);
fopen(s);

TimeInterval = 10e-3;
loop = 1e4;
time = now;
angle = 0;
%% Set up the figure
figureHandle = figure('NumberTitle','off', 'Name','Real Time data');

% Set axes
axesHandle = axes('Parent',figureHandle, 'YGrid','on', 'XGrid','on');

hold on;

plotHandle =
plot(axesHandle,time,voltage,'Marker','.','LineWidth',1,'Color',[0 1 0]);

xlim(axesHandle,[min(time) max(time+0.001)]);

% Create xlabel
xlabel('Time','FontWeight','bold','FontSize',14);

% Create ylabel
ylabel('\theta','FontWeight','bold','FontSize',14);

% Create title
```

```
title('Real Time Data','FontSize',15);

%% Initializing variables

angle(1)=0;
time(1)=0;
count = 2;

while ~isequal(count,loop)
    %%Serial data accessing

    angle(count) = fscanf(s,'%f');

    time(count) = count;
    set(plotHandle,'YData', angle, 'XData', time);

    pause(TimeInterval);
    count = count +1;
end

%% Clean up the serial port
fclose(s);
delete(s);
clear s;
```

```
duty   = [0.9 0.8 0.7 0.6 0.5 0.4 0.3];
volts  = [10 8.65 7.5 6.4 5.15 3.7 2.1];
a = fit(duty', volts',  'poly1');

v2 = 11:-1:3;
rpm = [336 304 272 240 208 176 144 112 80]
b = fit(v2', rpm', 'poly1');
```

```
rpm =

   336   304   272   240   208   176   144   112    80
```

motor saturation

```
time = [0 .1 .2]
vel  = [0 176.3 304.5] ./ 60;
b = fit(time', vel', 'poly1');
b.p1
```

```
time =

         0    0.1000    0.2000


ans =

   25.3750
```

Published with MATLAB® R2019b