Self Balancing Bike

Rauhul Varma
Shray Chevli
Spring 2017

ECE 395: Advanced Digital Projects Laboratory
University of Illinois at Urbana-Champaign

Abstract

This semester, we originally attempted to make an autonomous bicycle, but due to unforeseen circumstances had to modify our project from a fully autonomous bike to a self-balancing bike. We designed a flywheel system to be attached to the bike that would be spun to counteract the fall of the bike. We used a gyroscope to calculate the angle and angular velocity of fall which would then be fed into a PID loop to control the spin of the flywheel.

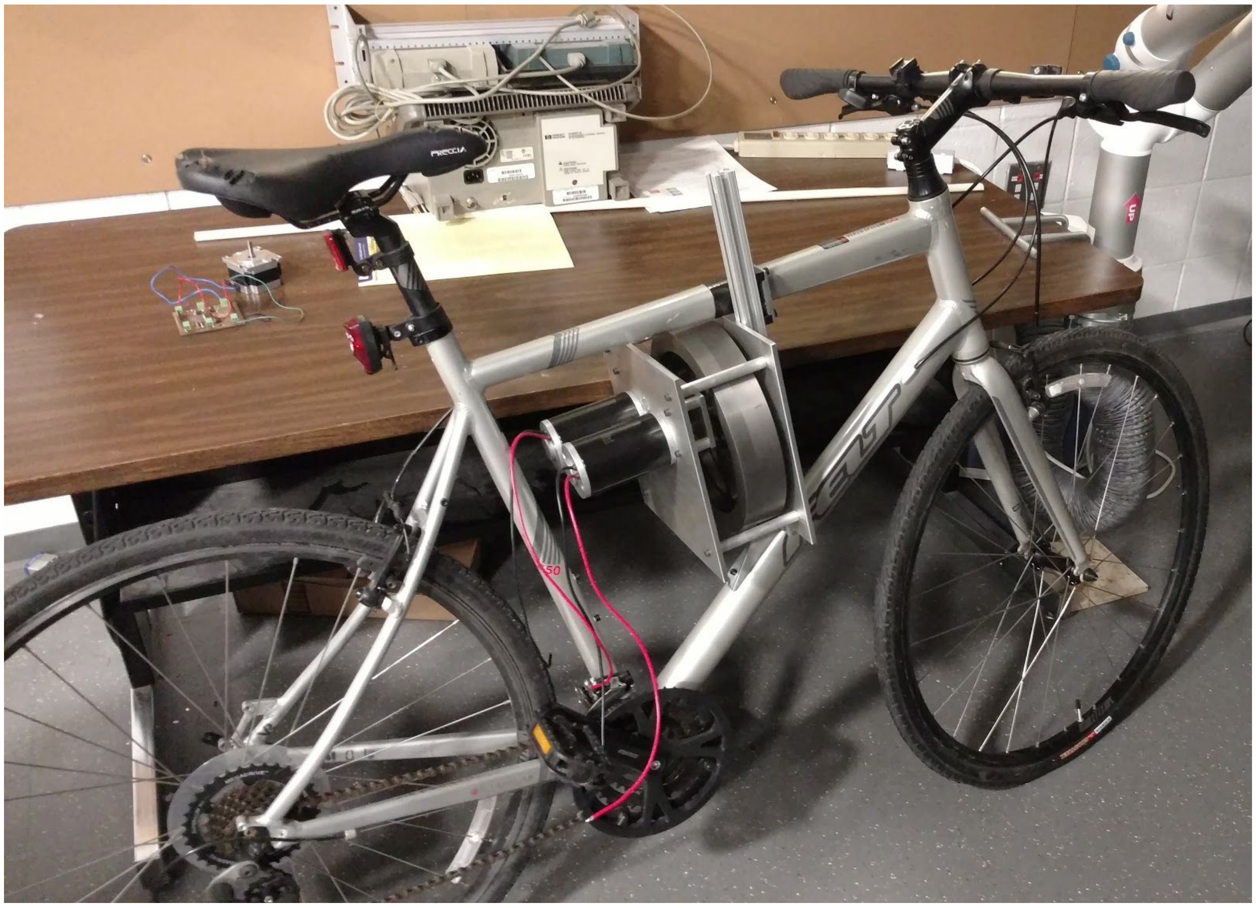


Table of Contents

Abstract **2**

Table of Contents **3**

MPU6050 **4**

- Overview 4
- The Circuit 5
- Data Collection 6
  - Determining the DMP’s Usefulness 6
  - Accessing the DMP on the LPC1114 6
  - Visualizing the DMP’s Output Data 7

Physical Design **8**

- Flywheel vs Shifting Weight 8
- Choosing The System Parameters 8
  - Unstable System 9
  - Stable System 10
- Designing a Flywheel and Gearbox 11

Combining the MPU6050 and Flywheel **12**

- Motor Hardware 12
- Microcontroller Control 13

Conclusion **14**

- Results 14
- Future Work 14

Appendix **15**

- Main.c 15
- MPU6050.h 17
- MPU6050.c 18
- PIDController.h 21
- PIDController.c 23

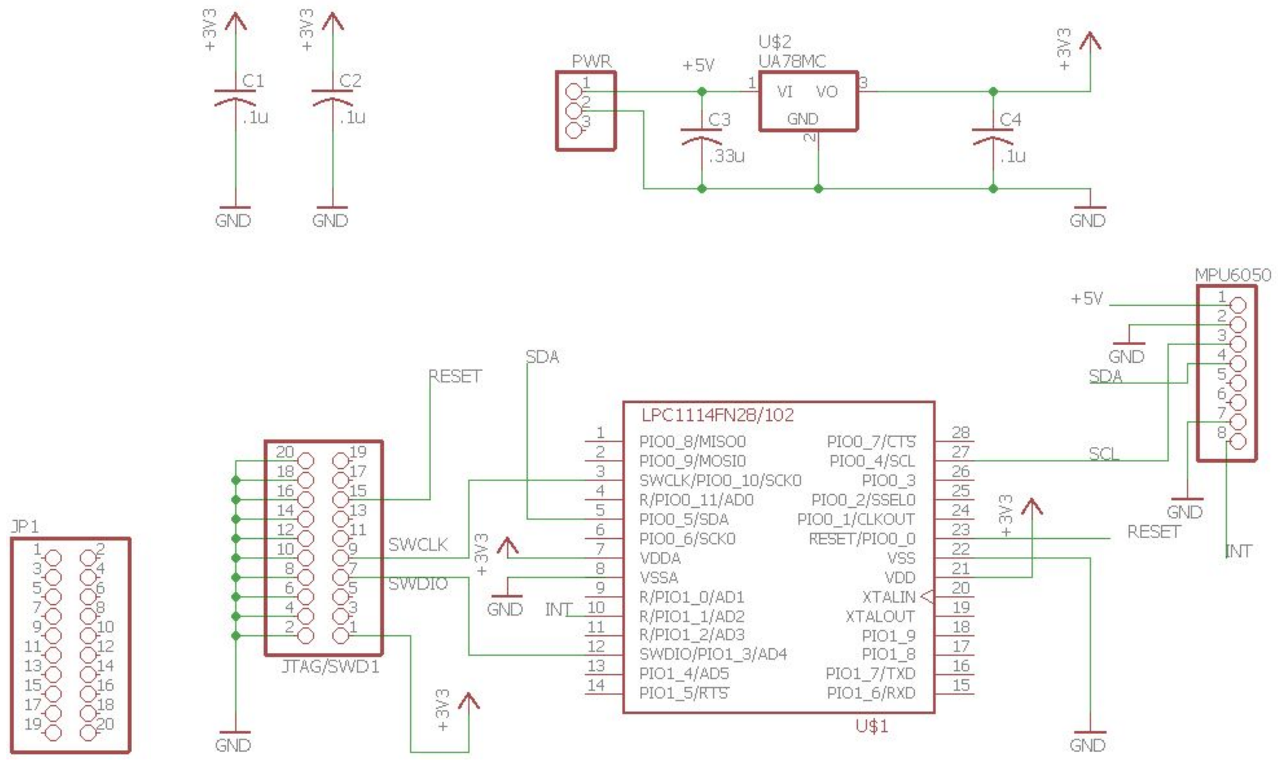
MPU6050

Overview

The MPU6050 is a sensor that combines a 3-axis gyroscope and 3-axis accelerometer into a single package. Additionally the chip has an embedded digital motion processor (DMP) that fuses the raw gyroscope and accelerometer data. The DMP provides this fused data in many forms, such as: a quaternion, gravity adjusted acceleration, and more. The DMP uses sensor fusion to account for drift of the sensor over time as well as calibration. For our design, we used the ARM Cortex M0 based chip, the LPC1114, to communicate with the sensor through the I2C communication protocol.

The Circuit

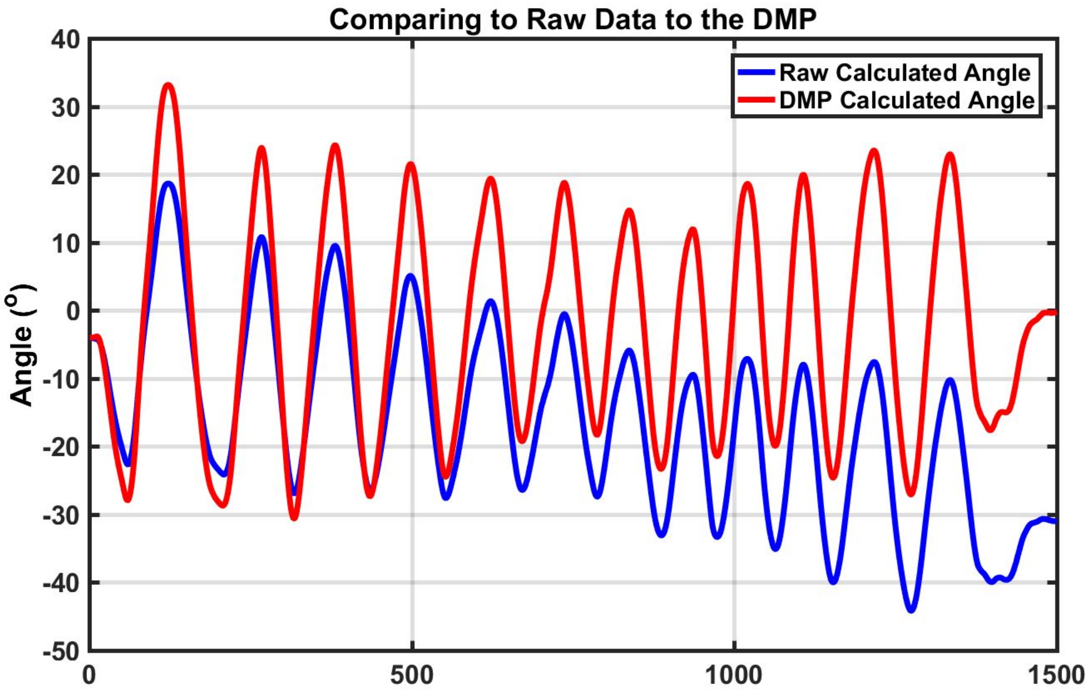
We had initially designed a PCB to integrate the MPU6050 and the LPC1114 onto a single easy to use package, but when adding the motor controller and it circuit protection components, we felt we should put this on hold until we got all the other parts working. This was our circuit design for using the MPU6050 with the LPC1114 along with the serial printer.



Data Collection

Determining the DMP’s Usefulness

We initially set about to determine if the raw data from the MPU6050 would suffice for the project or whether using the DMP would be necessary. Since the MPU6050 has large arduino community support we did our testing on an Arduino UNO. We integrated the raw gyroscope measurements and compared them to angles computed by the DMP over about 10 seconds while moving the sensor. This gave us a baseline of how the raw sensor values behaved compared to the DMP. Our results are below:



From various tests like this as well as additional research, we decided that using the DMP would provide us much clearer measurements than the raw gyroscope data filtered would. As you can see from the graph, the drift is very obvious from the raw data in that the oscillation angle is drifting downwards.

Accessing the DMP on the LPC1114

While there are many well established libraries for using the MPU6050 and the integrated DMP on the Arduino platform this is not true for the LPC1114. This left us with a single course of action: porting the Arduino library to the LPC1114, however, soon after we began we realized the scope of the undertaking. The library we attempted to port had seemingly unending dependencies on other Arduino libraries making porting unfeasible.

Coincidentally this setback led us to a much better solution. The Arduino solutions had all been reverse engineering through trial and error as Invensense had keep the memory maps regarding the DMP features private. To our luck, Invensense recently released a library for accessing all these features as well as a porting guide for getting it to work on any platform.

The porting guide was a little vague at time so we decided to first use port the code to the Arduino, so we could test in a more familiar and forgiving environment than the LPC1114. We were able to port all of the code except that related to the DMP to the Arduino UNO, and all the code to an Arduino with larger unboard storage. Next we began porting the LPC1114.

Porting to the LPC1114 was a much larger challenge due to our unfamiliarity with embedded C and the LPC1114 in general. After a lot of research and fixing some extremely unfortunate *Off-By-One* errors we did manage to get the DMP and the MPU6050 running on the LPC1114 flawlessly.

Visualizing the DMP's Output Data

In order to get a visual sense of the data and ensure that we were getting the correct output from the DMP, we created a MATLAB script to plot the angle of the chip in real time. This script is found under the folder *Matlab DMP Demo* and can be run by calling the function `simulate(port)` where `port` is a string that describes the filepath of the COM port where the LPC1114 is connected. Example usage:

```
>> simulate('/dev/tty.usbserial-A50285BI')
```

The LPC1114 should be running before running this command and should be running the provided code with the compiler flag: `MATLAB_DEMO`.

Physical Design

Flywheel vs Shifting Weight

After doing research on how various self-balancing systems work we narrowed them down to two options. The first method used a very heavy flywheel mounted along the axis the bike's wheels make with the ground. As the bike falls the flywheel spins up transferring that angular momentum from the bike into the wheel, counting the fall. The second method similarly used a large weight, however instead of modifying the angular momentum of the system it moves the weight to alter the center of gravity.

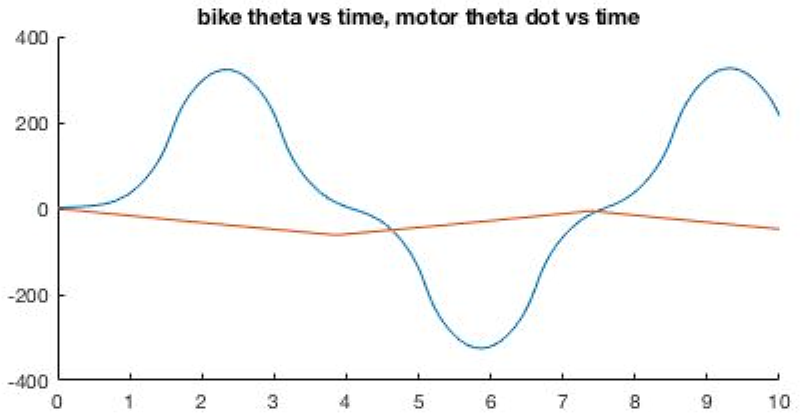
We opted for the first method given our familiarity with flywheels and more generally rotational systems.

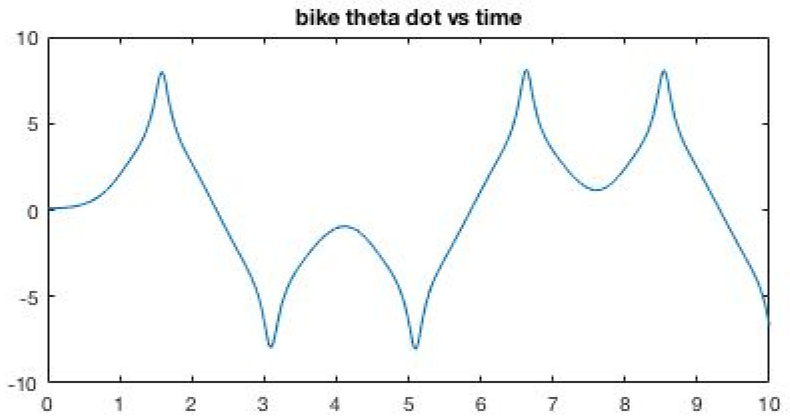
Choosing The System Parameters

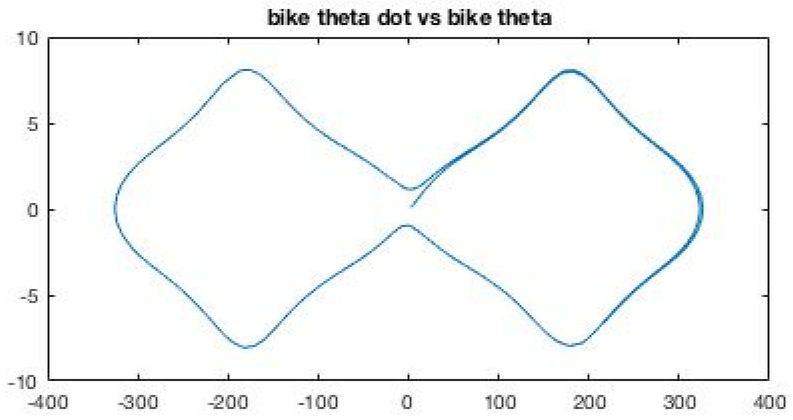
After we decided to use a flywheel we needed to develop a mathematical model for the system. We first estimated of the center of mass as well as the weight of the bike. We then chose boundary conditions for our stabilizing system; for our system to stabilize the bike, it must be no more than 10 degrees from normal and the tangential velocity of the center of mass must not be more than 2 feet per second.

With boundary conditions chosen and the bike parameters known, we created a model describing the way the bike falls. We then added a generic flywheel and PID controller and adjusted the parameters until we had a stable system. The following two graphs represent an unstable system and a stable system, respectively. These simulations can be rerun by using the MATLAB script in *Simulation* and calling `simulate_bike_stability().`

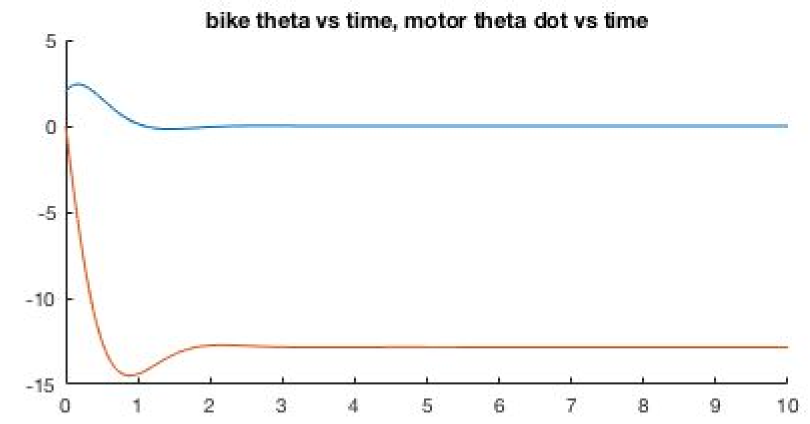
Unstable System

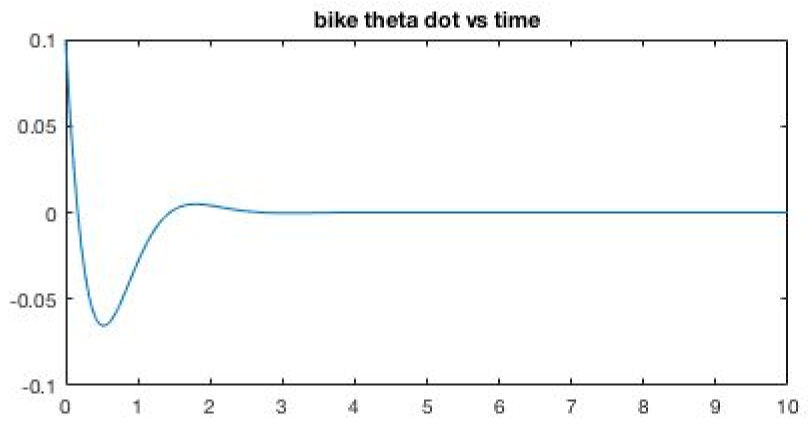


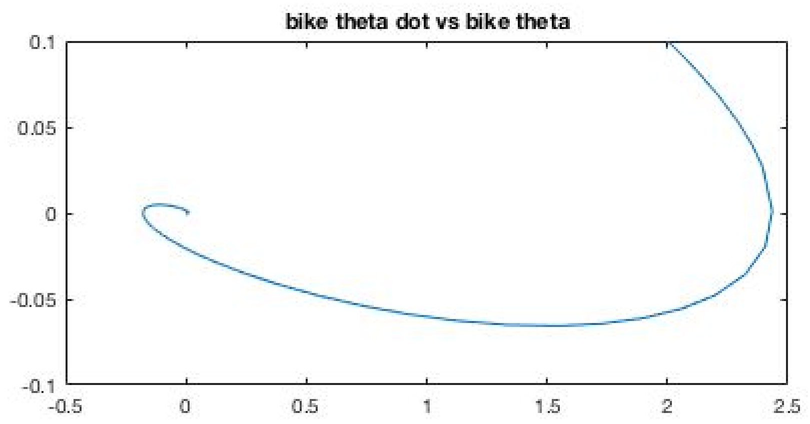




Stable System







Designing a Flywheel and Gearbox

After we had found the required moment of inertia of the flywheel, the motor stall torque and free speed we were ready to design a mechanism that fit those criteria. We used a pair of brushed motors through a 16.67 to 1 ratio, this achieved a spec close enough to our design. We verified these motors would work by re-running our stability tests in MATLAB with these motors additionally accounting for projected efficiency loss. We then designed a flywheel and ensured we had a sturdy mounting system so the gearbox would stay on the bike.

Re-using the gearbox should be fairly simply, if you want to use it on the same project, just power the motors via a motor controller. If you want to use the gearbox on another project: unmount the gearbox from the bike by removing the two ¼-20 bolts that connect the gearbox plate closest to the flywheel to the 80-20 rod on the bike.

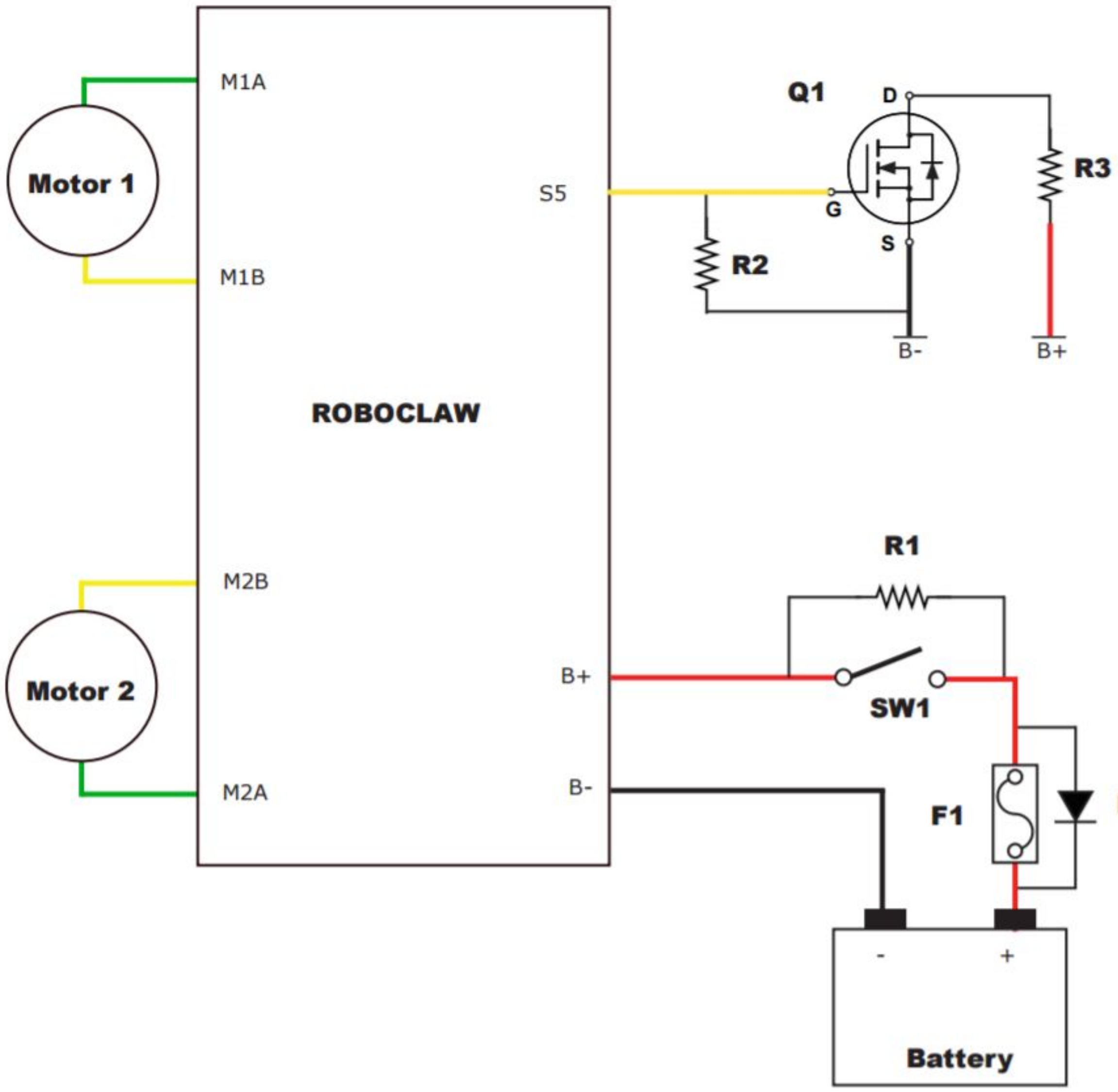
Combining the MPU6050 and Flywheel

Motor Hardware

For our design, we chose to use 2 DC brushed motors for our flywheel. Along with the motors, we decided that for this application and the speed we going to be running the motors at, it would be practical to obtain a motor controller to control the speed and direction. For that, we found the Roboclaw 2x15a to be a good fit to our system.

The Roboclaw 2x15a can provide 2 motors with up to 30 Amps each. Because of the large current draw of up to 60 Amps total for the motors, we also used a few other components in our design to protect the motor and the controller from overuse. First of all, we chose to use use a switch for basic circuit protection, so that we could physically control the voltage going to the motor controller. Along with the switch, the user manual recommended putting a 1K Ohm resistor as a pre-charge resistor parallel to the switch. Next we added a fuse, F1, parallel to reverse biased diode, D1, in between the positive terminal of the battery and one of the terminals of the switches. Based on the specifications of the user manual, we chose to use a fuse rated for 50 Amps, because based on the motors we had, as well as our application, we would only reach max current draw when switching directions so it should theoretically never be reaching that current for a prolonged time, but in the situation that it does, it would break the fuse and stop current flow altogether.

The other circuit we implemented was to protect the motor controller from regenerative voltage. The transistor we chose was the recommended in the user manual and R2 was 10K Ohm resistor to keep the MOSFET off while not in use and R3 is a high wattage resistor suchs as 50 Watts meant to help dissipate the extra voltage but when we implemented it, R3 started smoking so we scrapped that part of the circuit because we did not have enough time to figure out what went wrong and was not completely necessary for controlling the motors. It is possible we had received the wrong component from digikey.



Microcontroller Control

Our original design choice to control the motors was a PWM signal coming from the LPC1114 where the pwm signal would be determined from a PID Loop with the input as the Euler Angle from the sensor. Some of the issues we ran into was that the ARM outputted a 3.3 Volt PWM signal whereas the microcontroller used a filtered PWM signal of 2 Volts as its input. Given this, we did not have enough time to determine how to develop a circuit to accurately reduce the voltage to how the motor controller prefers it.

Conclusion

Results

At the end of the semester, we were not able to complete the project to our satisfaction. We had successfully programmed the sensor to read from the on board DMP to get the Euler Angles. On top of that, we were able to test our design of the flywheel system using the motor controller and the computer interface provided by the creators of the controller. Using that interface, we were able to control the speed and direction of the flywheel and show the flywheel has the power to rotate the bike opposite the fall and stabilize it.

Future Work

At the moment, we have no plans to continue this project next semester through ECE 395 but we have thought about the possibility of continuing this project through one of the Special Interest Groups in ACM by finally merging the sensor circuit with the motor circuit. Another next step would be to design a PCB that for our components as well as slightly modify the gearbox so that we can house some of our bigger needed components on the bike such as the battery or the fuse holder.

Appendix

Main.c

```c
#include <stdio.h>
#include <math.h>
#include <rt_misc.h>
#include "LPC11xx.h"
// MPU6050 Core
#include "MPU6050 Core/driver_config.h"
#include "MPU6050 Core/inv_mpu.h"
#include "MPU6050 Core/inv_mpu_dmp_motion_driver.h"
#include "MPU6050 Core/i2c.h"
#include "MPU6050 Core/timer32.h"

extern volatile uint8_t int_data_ready;
extern void SER_init (void);

#define DEFAULT_MPU_HZ 4

unsigned short gyro_rate, gyro_fsr;
unsigned char accel_fsr;
float gyro_sens;
unsigned short accel_sens;
void setup_MPU_6050() {
        uint8_t errors = 0;
        printf("Setting up\n");

        struct int_param_s int_param = { 1 };
        errors += mpu_init(&int_param);
        errors += mpu_set_sensors(INV_XYZ_GYRO|INV_XYZ_ACCEL);
        errors += mpu_configure_fifo(INV_XYZ_GYRO|INV_XYZ_ACCEL);
        errors += mpu_set_sample_rate(DEFAULT_MPU_HZ);

        errors += mpu_get_sample_rate(&gyro_rate);
        printf("FIFO rate: %d Hz\n", gyro_rate);

        errors += mpu_get_gyro_fsr(&gyro_fsr);
        printf("Gyro FSR: +/- %d DPS\n", gyro_fsr);

        errors += mpu_get_accel_fsr(&accel_fsr);
        printf("Accel FSR: +/- %d G\n", accel_fsr);

        errors += dmp_load_motion_driver_firmware();
        errors += dmp_set_fifo_rate(DEFAULT_MPU_HZ);
        errors += dmp_enable_feature(DMP_FEATURE_GYRO_CAL | DMP_FEATURE_6X_LP_QUAT |
DMP_FEATURE_SEND_RAW_ACCEL | DMP_FEATURE_SEND_CAL_GYRO);
        errors += dmp_set_interrupt_mode(DMP_INT_CONTINUOUS);
        errors += mpu_set_dmp_state(1);

        errors += mpu_get_gyro_sens(&gyro_sens);
        errors += mpu_get_accel_sens(&accel_sens);
        printf("%d errors.\n", errors);
}

short gyro[3], accel[3], sensors;
unsigned char more;
long quat[4];
unsigned long sensor_timestamp;

int main() {
        SER_init();

        // Start I2C
        I2CInit(I2CMASTER);

        init_timer32(0, TIME_INTERVAL);
        enable_timer32(0);

        delay32Ms(0, 100);
        setup_MPU_6050();

        while(1) {
                if (int_data_ready == 1) {

                        if (dmp_read_fifo(gyro, accel, quat, &sensor_timestamp, &sensors, &more))
                                int_data_ready = 0;
                        if (more > 0)
                                int_data_ready = 0;

                        if (int_data_ready == 1) {
                                #if defined DEBUG
                                        printf("FIFO PACKET:\n");
                                        if (sensors & INV_XYZ_GYRO)
                                                printf("Gyro: %f %f %f\n", gyro[0]/gyro_sens,
gyro[1]/gyro_sens, gyro[2]/gyro_sens);
                                        if (sensors & INV_XYZ_ACCEL)
                                                printf("Acce: %f %f %f\n", accel[0]/(float)accel_sens,
accel[1]/(float)accel_sens, accel[2]/(float)accel_sens);
                                        if (sensors & DMP_FEATURE_6X_LP_QUAT)
                                                printf("Quat: %ld %ld %ld %ld\n", quat[0], quat[1],
quat[2], quat[3]);

                                        printf("\n");
                                #elif defined MATLAB_DEMO
                                        printf("%ld %ld %ld %ld\n", quat[0], quat[1], quat[2], quat[3]);
                                #else

                                #endif

                                int_data_ready = 0;
                        }
                }
        }
}
```

MPU6050.h

```
#ifndef __M0_MPU_6050__
#define __M0_MPU_6050__
#if defined MPU6050 & defined ARM_CORTEX_M0

#include "inv_mpu.h"
#include <stdint.h>

#define I2C_WRITE_BUFFER_RESERVE 2

uint8_t i2c_write(uint8_t slave_addr, uint8_t reg_addr, uint8_t length, uint8_t *data);
uint8_t i2c_read(uint8_t slave_addr, uint8_t reg_addr, uint8_t length, uint8_t *data);

void i2c_clear_buffers(void);

void delay_ms(unsigned long length);
void get_ms(unsigned long *timestamp);

#define log_i(fmt, ...) printf(fmt, ## __VA_ARGS__)
#define log_e(fmt, ...) printf(fmt, ## __VA_ARGS__)

uint8_t reg_int_cb(struct int_param_s *int_param);

#define labs(x)   (((x) >  0 ) ? (x) : (0 - (x)))
#define min(a, b) (((a) > (b)) ? (b) : (a))
#define __no_operation() __nop()

#endif
#endif /* __M0_MPU_6050__ */
```

MPU6050.c

```c
#include "m0_mpu_6050.h"
#if defined MPU6050 & defined ARM_CORTEX_M0

#include "driver_config.h"
#include "i2c.h"
#include "timer32.h"
#include <stdio.h>

extern volatile uint8_t  I2CMasterBuffer[BUFSIZE], I2CSlaveBuffer[BUFSIZE];
extern volatile uint32_t         I2CReadLength, I2CWriteLength;


uint8_t i2c_write(uint8_t slave_addr, uint8_t reg_addr, uint8_t length, uint8_t *data) {
        uint8_t result;

        i2c_clear_buffers();

        uint8_t write_length = length + I2C_WRITE_BUFFER_RESERVE;
        if (write_length > BUFSIZE)
                return 2;        // Buffer Overflow

        I2CWriteLength           = write_length;
        I2CReadLength            = 0;
        I2CMasterBuffer[0]       = slave_addr;
        I2CMasterBuffer[1]       = reg_addr;

        for (uint8_t i = I2C_WRITE_BUFFER_RESERVE; i < write_length; i++) {
                        I2CMasterBuffer[i] = data[i - I2C_WRITE_BUFFER_RESERVE];
        }

        result = I2CEngine();

        #if defined DEBUG
                printf("addr:%d reg: %d\n", slave_addr, reg_addr);
                for (int i = 0; i < length; i++) {
                        printf("W: %d\n", data[i]);
                }
        #endif

        if(result != I2C_OK)
                return 1; // i2c Engine Failed

        #if defined DEBUG
                if (length == 1) {
                        uint8_t verify = 123;
                        i2c_read(slave_addr, reg_addr, length, &verify);
                        if (verify != data[0]){
                                printf(">> read != write <<\n");
                        }
                }
        #endif

        return 0;
}
```

```
uint8_t i2c_read(uint8_t slave_addr, uint8_t reg_addr, uint8_t length, uint8_t *data) {
        uint8_t result;

        i2c_clear_buffers();

        if (length > BUFSIZE)
                return 2;        // Buffer Overflow

        I2CWriteLength           = 2;
        I2CReadLength            = length;
        I2CMasterBuffer[0]       = slave_addr;
        I2CMasterBuffer[1]       = reg_addr;
        I2CMasterBuffer[2]       = slave_addr | RD_BIT;

        result = I2CEngine();

        if(result != I2C_OK)
                return 1; // i2c Engine Failed

        for(uint8_t i = 0; i < length; i++) {
                data[i] = I2CSlaveBuffer[i];
        }

        #if defined DEBUG
                printf("addr:%d reg: %d\n", slave_addr, reg_addr);
                for (int i = 0; i < length; i++) {
                        printf("R: %d\n", data[i]);
                }
        #endif

        return 0;
}

void i2c_clear_buffers() {
        for (uint8_t i = 0; i < BUFSIZE; i++) {
                I2CMasterBuffer[i] = 0;
                I2CSlaveBuffer[i]  = 0;
        }
}

void delay_ms(unsigned long length) {
        delay32Ms(0, length);
}

void get_ms(unsigned long *timestamp) {
        *timestamp = read_timer32(0);
}

#define INT_PIN  (1<<0)  //select pin of GPIO1

uint8_t reg_int_cb(struct int_param_s *int_param) {
        if (int_param->should_setup_int == 0) {
                return 1;
        }
        printf("Attach Interrupt\n");
        LPC_IOCON->R_PIO1_0 |= 0x01;  //select GPIO
        LPC_GPIO1->DIR &=~INT_PIN;               //0: input
        LPC_GPIO1->IS  &=~INT_PIN;               //0: Edge on corresponding pin is detected.
        LPC_GPIO1->IBE &=~INT_PIN;               //0: Single edge, determined by corresponding bit in
GPIOIEV register.
        LPC_GPIO1->IEV &=~INT_PIN;               //0: Falling edges, or low levels on corresponding pin
trigger interrupts.
        LPC_GPIO1->IE  |= INT_PIN;               //1: Corresponding pin interrupt is masked.
        NVIC_EnableIRQ(EINT1_IRQn);

        return 0;
}
```

PIDController.h

```
// include guard
#ifndef __PID_Controller_H__
#define __PID_Controller_H__

// dependencies
#include <stdlib.h>
#include <iostream>
#include <time.h>

/**
    PIDController object abstracts the math required to compute the duty cycle a
        motor should be spinning at inorder to maintain a desired equalibrium
 */
class PIDController {


private:
    /// `Porportional` constant for the PID Loop
    double k_P          const;

    /// `Integral` constant for the PID Loop
    double k_I          const;

    /// `Derivative` constant for the PID Loop
    double k_D          const;

    /// MAX RPM the motor this PIDController is controlling can spin at
    double k_MAX_RPM    const;

    /// MAX TORQUE the motor this PIDController is controlling can apply
    double k_MAX_TORQUE const;

    /// the equilibium point for the PIDController to maintain
    double k_TARGET const;

    /// Integral of the error (k_TARGET - current_position)
    double accumulated_error;
    double previous_position;
    time_t previous_time;

public:
    /**
        Creates a PIDController object with the specified PID constants and max
            rpm/torque
        - parameters:
            - P: `Porportional` constant for the PID Loop
            - I: `Integral` constant for the PID Loop
            - D: `Derivative` constant for the PID Loop
            - MAX_RPM: The maximum RPM the motor this PIDController is
              controlling can spin at
            - MAX_TORQUE: The maximum torque the motor this PIDController is
              controlling can apply
            - TARGET: The equilibium point for the PIDController to maintain
```

```
     */
    PIDController(double P, double I, double D, double MAX_RPM,
        double MAX_TORQUE, double TARGET);

    /// Destroys a PIDController object
    ~PIDController();

    void start(double current_position);

    /**
        returns: the desired dutyCycle of the PWM signal going to the motor
            ranging from -1 to 1
        note: a negtive sign means the motor should spin in the opposite
            direction
     */
    double dutyCycle(double current_position, double motor_omega);
}

#endif // __PID_Controller_H__
```

PIDController.c

```
#include "PIDController.h"

/**
    Creates a PIDController object with the specified PID constants and max
        rpm/torque
    - parameters:
        - P: `Porportional` constant for the PID Loop
        - I: `Integral` constant for the PID Loop
        - D: `Derivative` constant for the PID Loop
        - MAX_RPM: The maximum RPM the motor this PIDController is
            controlling can spin at
        - MAX_TORQUE: The maximum torque the motor this PIDController is
            controlling can apply
        - TARGET: The equilibium point for the PIDController to maintain
 */
PIDController::PIDController(double P, double I, double D, double MAX_RPM,
        double MAX_TORQUE, double TARGET) {
    k_P = P;
    k_I = I;
    k_D = D;
    k_MAX_RPM    = MAX_RPM;
    k_MAX_TORQUE = k_MAX_TORQUE;
    k_TARGET     = TARGET;
}

/**
    Destroys a PIDController object
 */
PIDController::~PIDController() {
    // nothing to do here
}

void PIDController::start(double current_position) {
    previous_position = current_position;
    previous_time = time(NULL);
}

/**
    returns: the desired dutyCycle of the PWM Signal going to the motor
        ranging from -1 to 1
    note: a negtive sign means the motor should spin in the oppiste
        direction
 */
double PIDController::dutyCycle(double current_position, double motor_omega) {
    time_t current_time = time_t(NULL);
    time_t time_diff = current_time - previous_time;
    previous_time = current_time;

    double current_error = k_TARGET - bike_theta;
    accumulated_error += current_error * ((double) time_diff);
    double derivative = (current_position - prev_position) / ((double) time_diff);

    // compute response torque based on system state and PID constants
    double torque = (k_P * current_error) + (k_I * accumulated_error) +
        (k_D * derivative);

    /**
        assumption: the maximum torque the motor can produce at a given angular
            velocity is the same in both rotational directions
     */

    // compute the maximum torque the motor can produce at the current angular
    //      velocity
    double torque_limit = k_MAX_TORQUE * (1 - (abs(motor_omega)/k_MAX_RPM));

    // convert torque to PWM duty cycle
    double duty_cycle = torque/torque_limit;

    // clamp duty cycle range to [-1: 1]
    if (duty_cycle > 1) {
        duty_cycle = 1;
    } else if (duty_cycle < -1) {
        duty_cycle = -1;
    }

    return duty_cycle
}
```