Kepros Diagnostic Posture Training Device – Final Report

Group: DEC1605
Client: Ted Kepros & KeprosPT
Group Advisor: Jeremias Sauceda

DEC1605 - Team Members and Roles:
Anthony Branson: Team Leader
Benjamin Engh: Team Communication Leader
Samual Eue: Key Concept Holder
Jiahui Quan: Team Webmaster

Website: http://dec1605.sd.ece.iastate.edu

Introduction

Problem:

There does not yet exist an efficient and accurate way to measure someone's posture over a period of time, recording values indicative of poor posture relative to one's homeostatic lumbar levels. Physical therapists and clinicians have to physically touch a patient's back muscles to understand what positions are causing unwanted tension. Because of this, physical therapists and clinicians rely almost entirely on either patient testimony, or snapshot observations of patient posture. One can obviously see how these testimonies may be unreliable, not fit for practical medical use. Athletes and other clients rely on their physical therapists to diagnose pain and optimize the use of their bodies. Ted Kepros and KeprosPT realized this problem and called upon the engineering students of Iowa State University to come up with a solution.

Solution:

Ted Kepros and KeprosPT proposed a wearable device to measure patient posture. This device would ideally be able to measure the posture of a patient over the course of a day, without seriously impeding their usual routines. Also, it was proposed that the device be able to measure a snapshot of a patient (typically an athlete) performing some kind of athletic motion. The eventual device should be able to record the data and display it in way that is not only readable by the therapist, but the patient as well.

Revised Project Design

Project Plan

Preface: Ted Kepros' solution of a wearable diagnostic and posture training device was conceived and shaped by an earlier Iowa State University senior design team. Kepros initially work with this design team in the Spring of 2015 and Fall of 2015. This team was able to formulate an initial iteration of this device, but not a working prototype, prior to our introduction to the device.

System Description

The system is composed of three core components. These components are the device host, the device hardware, and the device software. These three components work together to demonstrate recordable action, gather data, and process data respectively. While this general description is sufficient for a high level view of the system, more detailed explanations of the components are necessary for full understanding.

First, the device host is simply made up of the user and the shirt on which the device is anchored. This component serves as the subject of observation for the rest of the system. Second, the device hardware serves as a data collector for the system.

Second, the device hardware is made up of a power source (Two 9V batteries), four gyroscope and accelerometer MPU sensors, a three pad EMG sensor, and the system brain (Adafruit Feather M0 Adalogger Board). The two 9V batteries power the system while it is being hosted on the user. The MPU sensors collect valuable angular data to determine posture. The EMG sensor collects data pertaining to muscular tension or stress. Lastly, the Adalogger board initializes all of these sensors and retrieves their data using the programming of the device software.

Lastly, the device software is made up of two sub-components. These components are the Java code-base (for data processing) and the Arduino code-base (for device initialization and data retrieval). These aspects of the system allow the user to observe the data points on the user, intended for isolation.

Requirements

Hardware

1. The wearable must:
    a. Fit a medium build
    b. Not hinder the wearer's movement
    c. Be hosted discretely by the shirt
2. The wearable will be outfitted with two 9V batteries, 4 gyroscopes/accelerometer MPU sensors, a three-pad EMG sensor, and an Adalogger board.
    a. The EMGs may be attached to the body to get optimal readings.
3. The wires and electronics need to be arranged in a durable fashion to permit movement

Software

1. The program needs to be able to read data for times ranging from 10 minutes to 24 hours
2. Data from gyroscope and accelerometer must be programmatically combined to determine good or poor posture.
    a. We will allow for a 10-15% deviation from the standard posture
3. The program must record a user's homeostasis back position to calibrate the define good posture for each user
4. The data must be able to tell the reader when the user is leaning, twisting, or slouching, as well as other actions not yet determined

Deliverables

A large part of this project is testing and reading data in order to make the program more streamlined. We must deliver a working prototype. In order to accomplish this, we plan to have all major changes to the shirt and hardware done completed prior to semester II. We will also write software that is able to record and store raw data from the MPUs and EMG on a computer. Also, it is a goal that Ted will be able to use this hardware and software to run tests to get a large amount of data for future testing and modifications.

Our goal by the end of semester II is to deliver a working wearable device and program as close to commercialization as possible. With this in mind, we must study the data collected and isolate important values indicated by the client. It is a goal to generate graphs showing user's posture and actions. A similar graph must be made for the EMG data to determine muscular fatigue. We will then compare the data from these graphs to the data that is desired for proper posture to determine when the user has correct and incorrect posture. Another functionality that our new software will have is that it will convert the raw data to data that is easier to understand, such as angles instead of random numbers. After all functionality is finished, we will continue to run tests to smooth out the program.

Work Plan

Semester I - Spring 2016

(Insert Table With Work Plan Breakdown)

Semester I - Spring 2016

(Insert Table With Work Plan Breakdown)

Risks & Retrospective Thoughts

The risks for this project are mostly software related. We have a large enough budget to purchase the hardware desired, and Kepros is readily available to us even though he works in Cedar Rapids. Kepros will be able to answer any physical therapy questions we have, but there might be a time when our advisors aren't able to answer all of our software questions. Kothari specializes more in security and Sauceda may have to move for a government project.

Also, the gyroscopes, accelerometer, and EMGs may not be able to get accurate readings while connected to the shirt. In addition, our program and hardware may not be fast and reliable enough to determine quick changes in posture over long periods of time.

One challenge we met for the project is to get understandable readings for users. What we use to get angle reading is MPU6000 device, which combines a 3-axis gyroscope with a 3-axis accelerometer. Gyroscope gives us values of angular velocity, and accelerometer measures acceleration components caused by device motion and acceleration due to gravity. After several test with MPU6000 device, we found that angles occurred in xz-plane and yz-plane can be represented easily from AcX, AcY and AcZ values which are data from accelerometer.

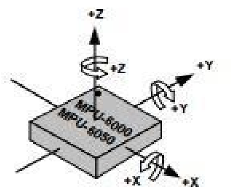

Figure 1: 3-axis Layout of MPU 6000 device

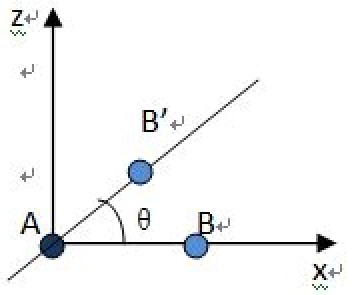

Figure 2: Angle occurs on xz-plane (point A and B are MPU 6000 device)

The formula we are using for the angle calculation is shown below:

$$\theta = \tan^{-1}(AcX_B / AcZ_B)$$

But we cannot directly get angles in xy-plane because there is no change in gravity when you rotate xy-plane. At this point, another set of data GyX, GyY and GyZ collecting from gyroscope will be helpful, which represents the angular velocity, to calculate angles occurred in xy=plane.

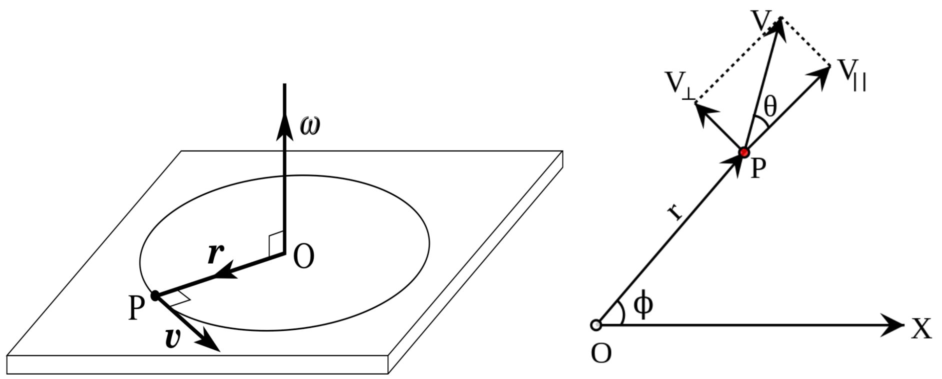

Figure 4: Angular velocity w with its magnitude of v and direction pointing upwards.

The formula we are using for the angle calculation is shown below:

$$\emptyset = \int_{t1}^{t2} \omega \, dt$$

To get accurate angle readings, we have to take several groups of angular velocity readings, get average and put it into the formula. It takes more time to calculate the angle with gyroscope data than accelerometer. As a result, we are not using the gyroscope data all the time.

Testing Processes & Testing Results

Preface: Much of our testing during the Spring of 2016 was intended to determine if particular aspects of the system were working. We were not provided with handoff materials and thus had to try to assemble our system according to the previous team's design. Therefore, much of the testing done during our first semester contributed to a revision of system design.

Hardware Testing

Semester One

Much of the hardware testing done during our first semester contributed to a revision of system design. This included testing each sensor individually, wearing an attempting to move within the original device hosting shirt, and sorting through the wires and setup left behind by the original design team. This testing was done on a weekly basis until we ordered new parts to satisfy our revision near the midway point in the semester.

The new part of our hardware design is that we use a voltage regulator to generate a 3.8V from a 9V battery to power our microcontroller. It is pretty much go with a LM317 regulator, resistors and capacitors shown in the figure below:

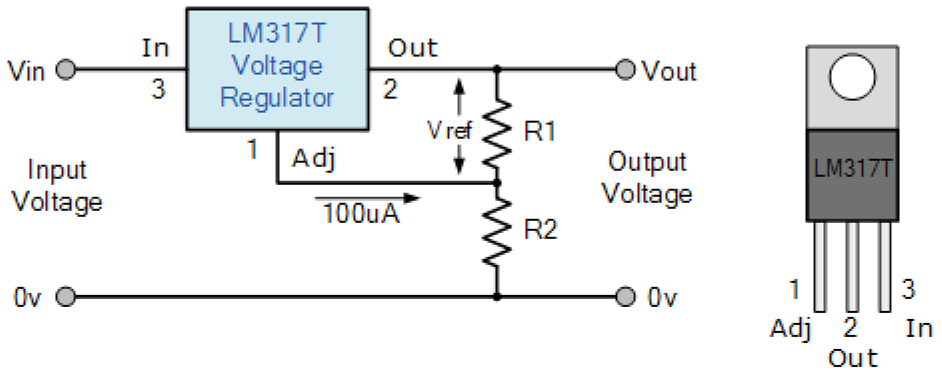

Figure 5: regulator circuit

Semester Two

After settling on the hardware chosen for the second iteration of the device early in the semester, it became time to assemble and test the hardware components as a single entity. Early tests indicated that multiple sensors had stopped responding to the board. Not knowing what caused this issue, we ordered two more sensors of the same type (MPU). This allowed us to get all aspects of the system working at once. However, though we knew how to assemble the device, its ability to retrieve data was shown to be inconsistent.

Software Testing

Knowing the expected results from the previous team and documentation for the Adafruit board helped testing a lot. Equations for calculating the the angles of the sensors could be changed by looking at the sensors current angle and seeing if the software printed out the correct information. Currently, we are testing multiple sensors at once to detect when the user is leaning in a certain direction. For the test, we can simply work in reverse and record ourselves leaning in a certain direction to see if the data from our software matches.
The harder part of the testing came up during the second semester. All of a sudden, the gyroprint program stopped displaying information from our sensors. We ended up using the program I2C_Scanner to look for all of the addresses on the I2C. None of the sensor's addresses were being recognized, so we knew the sensors were the problem. We ordered new sensors and the problem was fixed for some time. Lately, this problem is occurring very often. We know to look for the addresses on each of the sensors and separate the ones that are working from the ones that aren't. However, most times only two or three sensors are working and we are stuck restarting the board and reconnecting wires until the sensor is recognized by the board. In this way, software and hardware testing meshed together during much of the project. See Appendix III for further thoughts on this issue.

Appendix I: Operation Manual

Components & Corresponding Terms

Format: (- Component | Term)

- **MPU6000 Accelerometer & Gyroscope |MPU:**
  These four sensors allow us to retrieve angle data using basic calculus.
- **Muscle Sensor V3 | EMG (Electromyography):**
  This sensor provides a tension level of the muscle using electromyography

- **Adafruit Feather M0 Adalogger | Board:**
  This serves as the brain, connecting all sensors with Arduino Software

- **Two 9V Batteries | Power Source**

- **Flexible Athletic Fitting Shirt | Wearable Host**

- **Arduino Code Base | Board Programming:**
  Initializes the sensors and retrieves data

- **Java Code Base | Data Processing:**
  Processes the sensor data and outputs human readable values over a given time period.

Design: (intended)

The center of the design is the Feather M0 board, Attached via headers is the proto wing with the extent of the shirt specific hardware soldered on. If necessary the micro controller can be swapped with any of the others in the Feather line without much effort. Attached to the wing directly is out 9v to 3.4v regulator so we can run the entire system on 2 9v batteries. All sensors are soldered onto flexible wire out for mounting to the shirt.

Demonstration: (intended)

Make sure the wing and feather are firmly attached together, install both 9v batteries flip put on the shirt so that the conductive fabric is on the inside back, attach the MPUs to the velcro points on the shoulders and hips, lie subject on stomach and switch both batteries on. The system is now on and feeding data to the on board sd card, alternatively attach a micro usb to the board and the data will also be fed to the attached computer.

Appendix II: Iteration One & And Design Decisions

What was changed for iteration two?

We ended up making several changes due to our initial view of the first iteration; the microcontroller, wiring harness and data management

Why were these decisions made?

The first iteration of the board had a rat's nest of wiring that was impossible to trace or troubleshoot, we decided to cut the existing sensors from the harness to work with them to test the code from the first iteration. We also decided to change the microcontroller because we found one that was faster and was able to support 2 hardware I2C buses. Due to the state of the previous iteration's code we decided it would be easier to store data on an SD on the board itself rather than try to write the code for both the hardware and a side application to read and manipulate the data.

Appendix III: Anecdotal Notes and Takeaways

Over the year, we have run into many strange inconsistencies between our hardware and software. Previously, the second I2C was displaying no information to serial. At times the sensor's address wasn't recognized and EMGs gave values in the hundreds at rest when we expected values around 0. We have also had difficulty communicating between the Arduino and Java programs.

We solved the I2C and Arduino communicating with Java problems by scouring the datasheet and internet for information on our board and finding the sercom mux table and other people with the same issues. The EMG values became much more readable after using more reliable monitoring electrodes. We also tested some of the sensors and concluded they were broken, so we ordered new ones and began seeing results again.

The occurrence of an issue always felt random. Ultimately, there was a solution for every problem even though we didn't see it at the time. At the moment, we have run into another random occurrence and hope that our information can help future teams get more reliable readings from the sensors.

Currently, only two of the sensors are reading data consistently. They are two of our older sensors and they are of addresses 104 and 105. Sometimes, the newer sensors are also working and the recordings are done flawlessly. However, if the power source to the sensors is disconnected and reconnected, the sensors' address will stop being recognized. There are reports of similar issues on the web, but none of the solutions have worked for us. The documentation shows that we are connecting the sensors correctly, and we know this because the same method has worked for months. We have also tried running old versions of code that were working perfectly in the past with different wire configurations, boards, and sensors. In the end, the issue seems random to us, but we know there is a way to fix it and we hope that a future team is able to solve it.

Appendix IV: Code

- All of our code can be found at our GitHub Repository (INCLUDE LINK)
- **Arduino Code for Detecting Addresses (I2C_Scanner.ino):**

- This Arduino program can be used to discover if the board is able to find the address of the MPU 6000 sensor. If the address is not discoverable, the board will be unable to read data from the sensor.
- To use, one or two sensors must be connected to the SDA and SCL ports of the board. The yellow wire connects to SCL and the green wire connects to SDA. The red wire is connected to power and the black wire is connected to ground.
- Code:

```
void setup() {
  // put your setup code here, to run once:
    Wire.begin();
    Serial.begin(115200);
    Serial.println("\nI2CScanner begins");
}
```

In Arduino, the code inside of setup() is run before the code in loop(). Also, the code in setup is only run once, but the code in loop continues to run. Therefore, setup() is used for initialization while loop() records data.
Wire.begin() initializes Arduino’s Wire library, which is to communicate with the first I2C. This I2C corresponds to ports SCL and SDA on the board. Serial.begin(115200) initializes the Serial with a baud rate of 115,200. Think of the serial as the output. As you can see from the next line of code, print and println can be called from serial to print out information. The baud rate controls how fast the serial transmits data. The higher the number, the more quickly data will be transmitted.

```
void loop() {
    byte error, address;
    int errorNum = 0;
    Serial.println("Beginning Scan.");

    for(address = 0; address <= 127; address++){
      Wire.beginTransmission(address);
      error = Wire.endTransmission();
      if (error == 0){
        Serial.print( "I2C device found at address ");
        Serial.println(address);
      }
      else if (error == 2){
        errorNum++;
      }
      else if( error == 4){
        Serial.print( "unknown error at address ");
        Serial.println(address);
      }
    }
    Serial.println("done"); Serial.print("The number of 2's is : "); Serial.println(errorNum);
    delay(1500);
}
```

For the other larger programs, I'll just include the complicated and interesting parts, but I've included all of the code from I2C_Scanner to give you a better idea of how it all comes together.
The board can read any address from address 0 to address 127. Our sensors have addresses 104 and 105. On the physical sensor, you will see solder on ADR 0 or ADR 1. This corresponds to address 104 and 105 respectively.
As the name suggests, Wire.beginTransmission(address) starts the I2C's transmission for the specific address. Wire.endTransmission(address) ends the transmission and allows a new I2C device to be transmitted. Also, Wire.endTransmission(address) returns either 0, 1, 2, 3, or 4. The following shows what each of those numbers stands for.
0 : The address is found
1 : Too much data is being transmitted
2 : The address is not recognized
3 : The data is not recognized
4: Some other error.

The most common results of Wire.endTransmission are 0 or 2. If no address is found, the program will say that 128 2's were detected. Otherwise, it will say that addresses 104 and/or 105 were found. Changing the address from not recognized to recognized can be very puzzling. Often times, we change nothing, but we reset the board and re-plugged everything back in. When the sensor is working, if the sensor's power source is disconnected, the address will no longer be recognized when the power is reconnected.

- **Getting Sensor Data from the First I2C (Gyroprint.ino):**
    - This program reads data from SCL and SDA. It acts as the first half of gyroprint4sensors.ino. It is used for debugging the first I2C. To run it, connect 2 sensors the same as they were connected for I2C_Scanner.ino. You may want to test these sensors on I2C_Scanner first to ensure that the addresses are being found correctly.

```
Wire.beginTransmission(address);
  Wire.write(0x6B);  // PWR_MGMT_1 register
  Wire.write((byte) 0);     // set to zero (wakes up the MPU-6050)
  int debug1 = Wire.endTransmission();

  Wire.beginTransmission(address);
  Wire.write(0x1C);  // Accel config register
  Wire.write(0x08);     // Setting range to +-4G
  int debug2 = Wire.endTransmission();
  Wire.beginTransmission(address);
  Wire.write(0x1B);  // Gyro config register
  Wire.write(0x08);     // setting range to +-500 degrees per second
  int debug3 = Wire.endTransmission();
```

For each sensor connected to the board, the above code needs to be run in setup to write bytes to the I2C to initialize the sensors. Keep in mind that the address should be 104 for one of the sensors and then 105 for the other. The comments explain specifically what each write initializes.

```
  Serial.print("IMU1 : ");
  //code inside of I2CPrint
  Wire.beginTransmission(address);
  Wire.write(0x3B);  // starting with register 0x3B (ACCEL_XOUT_H)
  Wire.endTransmission(false);
  Wire.requestFrom(address,14,true);  // request a total of 14 registers
  AcX=Wire.read()<<8|Wire.read();  // 0x3B (ACCEL_XOUT_H) & 0x3C
(ACCEL_XOUT_L)
  AcY=Wire.read()<<8|Wire.read();  // 0x3D (ACCEL_YOUT_H) & 0x3E
(ACCEL_YOUT_L)
  AcZ=Wire.read()<<8|Wire.read();  // 0x3F (ACCEL_ZOUT_H) & 0x40
(ACCEL_ZOUT_L)
  //Tmp=Wire.read()<<8|Wire.read();  // 0x41 (TEMP_OUT_H) & 0x42 (TEMP_OUT_L)
  GyX=Wire.read()<<8|Wire.read();  // 0x43 (GYRO_XOUT_H) & 0x44 (GYRO_XOUT_L)
  GyY=Wire.read()<<8|Wire.read();  // 0x45 (GYRO_YOUT_H) & 0x46 (GYRO_YOUT_L)
  GyZ=Wire.read()<<8|Wire.read();  // 0x47 (GYRO_ZOUT_H) & 0x48 (GYRO_ZOUT_L)
  int value = 0;
  value = analogRead(A2);
```

The above code reads in bytes from the I2C at the designated address. This time, Wire.endTransmission(false) is called. The default value is true. True releases the bus after transmission while false allows us to continue using the same bus.
AcX through GyZ are initialized as type int16_t, and the wire.read() functions are reading specific bytes that give us the desired data from the sensors.
Finally, analogRead(A2) reads information connected to the port A2. This is used to read the data from the EMG. The yellow wire from the EMG is connected to A2 while the others are connected to power and ground.

```
  Serial.print("EMG = "); Serial.print(value);
  Serial.print(" | AcX = "); Serial.print((AcX/8192.0)* 90.0);
  Serial.print(" | AcY = "); Serial.print((AcY/8192.0) * 90.0);
  Serial.print(" | AcZ = "); Serial.print((AcZ/8192.0) * 90.0);
  //Serial.print(" | Tmp = "); Serial.print((Tmp/340.00+36.53)*9/5 + 32);  //equation for
temperature in degrees C from datasheet
  Serial.print(" | GyX = "); Serial.print(GyX/65.5);
  Serial.print(" | GyY = "); Serial.print(GyY/65.5);
  Serial.print(" | GyZ = "); Serial.print(GyZ/65.5);
  Serial.println(";");
```

As shown in I2C_Scanner, we can print important information to the serial. That is what is happening here, and the raw data is being altered to make it more readable. The data from the accelerometer gives angles between -90 and 90 degrees. Tmp is currently commented out because it gives the measured temperature which is not needed for our project.

- **Getting Sensor Data from the Second I2C (gyroprintTwoWire.ino):**
  - Similar to gyroprint, except the initialization is different so that we are measuring from a second I2C. Again, use it for debugging the second I2C (the third and fourth sensor). Gyroprint4sensor.ino is the program that reads from all 4 sensors at the same time.
  - Only two sensors can be connected to the I2C (the SCL and SDA pins on the board) at once, because the sensors can only be set to addresses 104 and 105. To connect the other two sensors, a second I2C needs to be made.

```
  #include "wiring_private.h" //pinPeripheral() function
 TwoWire mywire(&sercom1, 11, 13); // (SDA, SCL) (pad0, pad1)
  //inside setup
  mywire.begin();
  pinPeripheral(11, PIO_SERCOM);
  pinPeripheral(13, PIO_SERCOM);
```

The code above shows the extra initializations needed to create the second I2C. The TwoWire function details the pins that will act as the SDA and SCL pins. Pins 11 and 13 are chosen because they match the desired results from the Sercom Mux table (https://learn.adafruit.com/using-atsamd21-sercom-to-add-more-spi-i2c-serial-ports/muxing-it-up). Other pins can be used if they use pads 0 and 1. Also, sercom1 in TwoWire is used for pins 11 and 13, but a different sercom or alt_sercom is needed if other pins are needed. By default, pins SDA and SCL are used on sercom3.
The pinPeripheral function allows the numbered pin to be used by sercom. If you want to use sercom_alt, use PIO_SERCOM_ALT. Mywire.begin() is the function to start up the second I2C. Contrary to what you may think, pinPeripheral needs to be called after the TwoWire variable begins. Otherwise, TwoWire acts the same way as Wire.

- **Getting Sensor Data from Four Sensors (gyroprint4sensors.ino):**

    - This is the program that records data from all 4 sensors and 2 EMGs. Basically, gyroprint.ino and gyroprintTwoWire.ino are combined into one program. All four sensors need to be initialized and their data is recorded separately. To make this cleaner, functions readDataW1 (I2C 1) and readDataW2 (I2C 2) are created and called in the loop for addresses 104 and 105.
- **User Interface (gyroscope/main.java):**
    - This code can tell the board to run for a certain amount of time. It also records data from the serial recorded over the set time and saves it to a text file.

```
import com.fazecast.jSerialComm.*;
import javax.swing.*;
static SerialPort serialPort;
//get the ports that are recognized on the serial, should be COM7
SerialPort ports[] = SerialPort.getCommPorts();
//use the port that the user selected using the GUI
serialPort = SerialPort.getCommPort(portsBox.getSelectedItem().toString());
serialPort.setComPortTimeouts(SerialPort.TIMEOUT_READ_SEMI_BLOCKING, 0, 0);
if(serialPort.openPort()){
        System.out.println("Opened the port");
} else{
        System.out.println("Port did not open");
        System.exit(0);
 }
BufferedReader data = new BufferedReader(new InputStreamReader(serialPort.getInputStream()));
double timeSoFar = 0;
String line = null;
try {
    PrintWriter pwriter = new PrintWriter("gyroInfo.txt", "UTF-8");
    boolean firstLine = true;
    while(((line = data.readLine()) != null) && (timeSoFar < chooseTime)){ //while(data.hasNextLine() && (timeSoFar <
chooseTime)){
        try{
                //don't print the first line to a file, because it is often an incomplete line
                if(firstLine){
                System.out.println(line);
                TimeUnit.MILLISECONDS.sleep(233);
                timeSoFar = timeSoFar + 233;
                firstLine = false;
        }
        else{
                System.out.println(line);   //data.nextLine()
                pwriter.print("Seconds: " + Math.round((timeSoFar / 1000.0) * 100.0) / 100.0 + " | ");
                pwriter.println(line);
                TimeUnit.MILLISECONDS.sleep(233);
                timeSoFar = timeSoFar + 233;
                }
    }
    catch(Exception e2){}
}
```

The above code shows ways that the imported jSerialComm library can be used to read data from the serial and print it to text file gyroInfo.txt. The time that the data is recorded is also printed on the text file. There is more to the java file, but it is mostly using swing to create the GUI and is relatively easy to follow.

Going forward:

It might be beneficial to look into some of these when going forward in the development of this project.

- Different gyro sensors. Many of the delays we have had with this iteration have been to due the finicky nature of the MPUs, many times we have had a fully working setup then the next day the MPUs were not recognised by the microcontroller despite no changes in code and only slight shifting of the hardware from transport.
- Better attachment / shirt: at the beginning of the year we were looking into a new design on Ted's recommendation however due to hardware we had to scrap the idea for more time to debug the issues cropping up with the MPUs