Redesigned Aerospace Control System Platform for Laboratory Re-purposing

A Senior Project
presented to
the Faculty of the Aerospace Engineering
California Polytechnic State University, San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science

by
Ryan W. Moskaluk
June, 2013

©2013
Redesigned Aerospace Control System Platform for Laboratory Re-purposing

Ryan W. Moskaluk*

California Polytechnic State University San Luis Obispo, San Luis Obispo, California, 93405, United States

This senior project encompasses using an aircraft model in conjunction with the 3' by 4' low speed wind tunnel at Cal Poly San Luis Obispo. The aircraft model is controlled through piezoelectric actuators embedded inside the flexible tail structure via a PID controller. The objective for this project is to demonstrate the effect of tuning the individual PID controller gains with the aircraft model inside the low speed wind tunnel and to create an easy to use graphical user interface for doing so.

I. Introduction

Control systems can be found in many different everyday applications. Some range from controlling the temperature in a home thermostat to controlling the amount of time and heat nuclear reactor control rods require. These systems range in complexity as well. Fundamentally most control systems are used in conjunction with microprocessors but they do not necessarily have to be used with them. The scope of this project is to design an interface for allowing the micro-controller proportional, integral, and derivative gains to be changed easily and creating an easy to use graphical user interface for quick repeatability in experiment testing. The primary objective for the project was to allow for an easy setup and operate platform that could be expanded on in the future for students and teachers to use.

A. Background

One system that can be used in both the analog and digital realm is the Proportional, Integral, Derivative (PID) controller. The PID offers the control system three different “knobs” for controlling the system. The proportional gain decreases rise time while increasing overshoot. The derivative gain reduces the settling time of the system and decreases the overshoot added by the proportional gain. Lastly, the integral gain is used to eliminate the steady state error of a controlled system. In doing so it increases the overshoot and settling time. Using all three of these gains together one can effectively control a system by finding the optimal values for each gain to produce the desired output of the system. To find the optimal gains different computer algorithms have been designed along with using certain techniques. One particular technique that is still widely used is to simulate the linear dynamics of a system via a computer model and tune the gains through simulation to meet specific design characteristics. Using the knowledge of how each gain can affect the system a trial and error technique or intuitive technique can be used. Since the gains are found through using a simplified computer model the physical system might have unmodeled characteristics that will change the gains even farther. The Ziegler-Nichols method is another tuning method for determining the gains of the PID controller. Another design technique is use root locus design methods in the Laplace domain and converting via appropriate transformations to the z domain. Lastly, PID modeling and optimization software can be used for determining the optimal gains of the PID controller. All of these methods have advantages and disadvantages for determining the PID controller gains.

*Student, Aerospace, 1 Grand Avenue
II. Design

A. Initial

1. Software

In the initial stages of this project the primary design was to incorporate an Arduino micro-controller board. This was done to allow for upgrades to be made easily to the project, such as later labs or special project modifications. LabView and less powerful and well know micro-controllers were also looked at. After some initial research it was found from the Arduino family, that an Arduino Uno would be used to control the system and be used as the basis for all other calculations. Upon further investigation the programming language known as Processing, was chosen to be used in conjunction for accessing the Arduino through an easy to use graphical user interface. The Processing language is a sub variant of C which the Arduino IDE is run from so it appeared to be the most logical choice when interfacing directly with the Arduino. Further research showed that an interface library known as controlP5 for Processing was available and would allow for the creation of buttons, knobs, graphs, and much more. One major portion that needed to change for the objective of this project to be met was to find a way to update gains for the control system faster. The original time for the program to be re-flashed onto memory was approximately thirty seconds. Looking at the PID library written for the Arduino a function called \textit{PIDtunings} was found. This function allows the user to change the gains for the PID controller within a much shorter time. This near real time change would allow for quick iterations for PID gains and potentially different control architecture to be implemented in the future. The psuedo code that was developed early on is shown in code block 1. The overall layout was a combination of an Arduino driven program and a front end Processing interface program for the user. After the software portion of this project was started the hardware portion needed to be reconstructed based on the original project from James Bach’s Thesis.\textsuperscript{1}

Code Block 1: Psuedo code for Processing and Arduino programs

```
1  ********************************************************************
2  //Processing Psuedo Code Layout                                  
3  ********************************************************************
4  import processing serial/Ethernet libraries                      
5                                                                 
6  void setup()                                                     
7  {                                                               
8    initialize port and other bookkeeping items                    
9  }                                                               
10                                                                 
11  void draw()                                                     
12  {                                                               
13    all of the GUI pictures/buttons/figures/placement done here    
14  }                                                               
15                                                                 
16  void serialEvent()                                              
17  {                                                               
18    pass inputs to Arduino                                        
19    receive outputs from the Arduino to update the interface      
20  }                                                               
21  ********************************************************************
22  //Arduino Psuedo Code Layout                                    
23  ********************************************************************
24  Initialize any inputs, variables, and PID values                
25                                                                 
26  void setup()                                                   
27  {                                                               
28    attach servos and initialize the Arduino for startup          
29  }                                                               
30                                                                 
31  void loop()                                                    
32  {                                                               
33    read data from the Processing interface                       
34    Change PID value with PIDtunings()                           
35    command servos to specific values                            
36    send data back to the Processing interface to be updated    
37  }
```
2. Hardware

The basis of the paper was to drive the four mounted piezo-electric actuators on the flexible tail structure of the aircraft model. While commanding these to keep the tail rigid and at the same deflection the gyro mounted in the nose of aircraft was used to take measurements of the aircraft for further performance analysis. In the original paper a power amplifier was used to drive the actuators and a program written in C was used to interact with the system. Strain gages are mounted to adjacent to the actuators to measure the bending strain of the tail and feed that back into the program so the actuators can be adjusted and recommended. To amplify the strain gages an amplifier circuit was originally designed using the INA-122 instrumentation amplifier and a simple two pole RC passive filter. The system was designed to provide a gain of 100 to amplify the difference in voltage across the Wheatstone bridge. The Wheatstone bridge was constructed of the four strain gages. The difference in the resistance for each strain gage corresponded to a voltage difference across the Wheatstone bridge and was then amplified by the INA-122 circuit so the appropriate A/D hardware could determine the minute differences. From there the signal was then sent into a 12 bit A/D Zworld XP-8910.\textsuperscript{1} With the Zworld XP-8910 a pulse width modulation signal was sent out from the software implemented PID controller ranging from 0-5.73 volts. This then had to be amplified to -10 to +10 volts as an input signal for the power amplifier being used. This was done using an amplifier circuit and was later found that a higher resolution support board was needed. From reading through the original paper and learning about the support hardware that was used a comparable power amplifier (Piezeo Systems EPA-104) was found at California Polytechnic San Luis Obispo Wind Tunnel laboratory. The power amplifier accepts a signal ranging from -10 to +10 volts and outputs ±200 volts with a max current draw of 200mA. The initial concern was designing the support hardware for amplifying the Arduino signal for interfacing with the power amplifier. A few different filter types were looked at for doing this. These included: simple single pole RC passive filter with an operational amplifier for gain factor, a two pole RC passive filter with an operational amplifier for gain factor, and a 3 pole active Low Pass Buttersworth filter. The 3 pole Low Pass Buttersworth filter was chosen because it uses roughly the same parts as the original project did but is more optimized for better performance, it is an active filter which allows for this better performance than the passive filters, and better filtering was needed for a more accurate output. The 3 pole Low Pass Buttersworth filter can be in fig. 1.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig1.png}
\caption{3 pole Low Pass Buttersworth filter}
\end{figure}

To obtain the feedback needed for the software control loop the strain gages needed to be incorporated. The original project mentioned using an instrumentation amplifier, INA-122, to amplify the signal from the strain gages and feed that into the computer. The design was then to incorporate the INA-122 and a full Wheatstone bridge. The power amplifier would then drive the four piezo-electric actuators and the four strain gages would return the strain in the plexiglass tail structure.
B. Final

1. Software

With a firmer understanding of the project and the hardware/software that was needed to effectively support the project a final design started to emerge. As the software progressed so did the functions that needed to be utilized by the user for convenience and ease in a potential laboratory experiment. The user needed access to storing data from test cases, commanding the servos mounted on the tail and wings, and being able to see in real time what the input, commanded value, and output of the Arduino PID controller was. Figure 2 shows how the Processing graphical user interface is laid out.

![Final graphical user interface for Arduino written in Processing](image)

The highlighted portion A is where the user is able to choose what port the Arduino is on from a list of currently active USB devices on the computer. This was added due to the laboratory utilizing multiple USB devices and different computers. It was also found in testing that this function was necessary when working between two computers as the communications port would change for the Arduino and having to access source code to change the input value was not a desirable method. Section B is where the user is able to enter in specified PID values and then relay those values to the Arduino in near real time. The left most column is where the user actually enters these values. The right column is where the Arduino displays the current PID values and information regarding the controller. The button at the bottom allows the user to send those values through a serial connection to the Arduino to be updated. Section C is where the user can specify a filename for saving different experimental modes of data. When the user is ready for taking data the button labeled START DATA RECORD is clicked then data will be written to internal memory allocated to Processing. Once the user wishes to stop recording data they must label the file and click the STOP DATA RECORD button. The format output of the data is an excel spreadsheet which can be seen in table 1. Section D is for driving the servos located on the aircraft mounted on the tail and both wings. The SET SERVOS button then allows the user to command those servos to change the characteristics of the aircraft while in the wind tunnel during a test scenario. Section E is a graphical display of the Input and Setpoint (commanded value) of the Arduino in real time. The red is the Input signal to the Arduino and the green denotes the Setpoint value of the Arduino. The x-axis measures time while the y-axis is a measure of...
of voltage ranges. Section F shows the output of the Arduino PID controller in real time with the same x and y axis scheme. Both section E and F were designed to allow the user to see what the Arduino is seeing and adapt the PID gains appropriately.

### A. Processing Code

The Processing code is structured as follows and will be talked about in great detail. For the full length of code see the Appendix B. In this section of code the controlP5 library for Processing was used to create the graphical user interface shown in Fig. 2. Lines 1-3 import the following libraries; the serial communication library, the controlP5 graphical user interface library, and a java buffer library for serial communication. Lines 4-10 then initialize the serial communication object and the controlP5 interface objects for use later in the program. Following lines 11-80 initialize all of the variables that are used later on in the program.

#### Code Block 2: Imports and object declarations

```java
1 import processing.serial.*;
2 import controlP5.*;
3 import java.nio.ByteBuffer;
4 Serial myPort;
5 ControlP5 controlP5;
6 controlP5.TextField PropField, IntField, DerField, PropLabel, IntLabel, DerLabel,
7 SetPointField, SetPointLabel, FileField;
8 controlP5.DropdownList dl, dl2, dl3;
9 controlP5.Textlabel DirectionLabel, ModeLabel;
10 controlP5.Knob Servo1_Knob, Servo2_Knob, Servo3_Knob;
11 ControlGroup messageBox, messageBoxData, messageBoxPIDData, messageBoxServoData,
12 messageBoxDataRecord;
```

Starting at line 81 the setup function that Processing calls upon startup initialization of the program is shown. Lines 83-87 declare the size of the window being used, and initialize the serial port connection. Following that lines 88-92 create the drop down list that allows the user to select the communications port for the Arduino. Then placing the remaining items shown in Fig. 2 are done in lines 94-244 and follow the same style as shown by lines 94-100. Lines 245-265 load in the error dialog boxes and set the default values for the PID text boxes on the graphical user interface.

#### Code Block 3: setup method

```java
81 void setup()
82 {
83 size(1000,600); //width, height
84 smooth();
85 myPort = new Serial(this,"",19200);
86 myPort.bufferUntil('
');
87 controlP5=new ControlP5(this);
88 dl=controlP5.addDropdownList("COM_List")
89 .setPosition(20,40)
90 .setId(1)
91 ;
92 customize(dl);
93 PropField=controlP5.addTextField("Proportional Gain")
94 .setPosition(20,90) //x,y
95 .setSize(100,40)
96 ;
97 customSet(controlP5.FLOAT);
98 .setFont(createFont("arial",20))
99 .setAutoClear(false)
100 .setInputFilter(controlP5.FLOAT);
```

After the setup initialization block of code the different drop down lists that are referenced follow in lines 266-315. At line 316 is where the compiler calls the draw function to create the actual graphical user interface and it also calls the internal plotting user defined plotting method, createPlots(). The method controlEvent on line 321 is the event monitor for when the user pushes a button or selects anything on the interface. From there different case statements are assigned based on the button and drop down list ids assigned to them in the declaration. Case 1 on line 328 handles the choice for which communication port should be used.
and which are detected. Case 3 on line 343 is where the PID values are passed to the respective methods serialPrinterGains and then to floatArrayToByteArray. Error handling of three seconds is wrapped around the method so that the Arduino is flooded with multiple gain values from the user encase the user tries to update within three seconds. Case 4 on line 363 is structured the same way as case 3 but with the servos being sent to the method serialPrinterServos instead. Then for the data recording case 5 starting on line 379 calls the dataRecorder method for collecting and storing the data in the memory allocated by Processing during run time. Lastly, case 6 on line 392 calls the dataSave method for when the user wishes to save the data to an excel formatted file as shown in table 1.

Table 1: Processing output data format table example

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Input</th>
<th>Output</th>
<th>Setpoint</th>
<th>Proportional Gain</th>
<th>Integral Gain</th>
<th>Derivative Gain</th>
<th>Mode</th>
<th>Direction</th>
<th>LH Aileron Servo (degrees)</th>
<th>RH Aileron Servo (degrees)</th>
<th>Tail Servo (degrees)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31969</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>Direct</td>
<td>Automatic</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>32358</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>Direct</td>
<td>Automatic</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>32756</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>Direct</td>
<td>Automatic</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

Code Block 4: controlEvent function

```java
void controlEvent(ControlEvent theEvent) {
  if (theEvent.isGroup()) {
    println("event from group: "+theEvent.getGroup().getValue()+" from "+theEvent.getGroup());
    switch (theEvent.getGroup().getId()) {
      case (1):
        myPort.clear();
        myPort.stop();
        myPort = new Serial(this, Serial.list()[int theEvent.getGroup().getValue()], 19200);
        myPort.bufferUntil(10);
        println(Serial.list()[int theEvent.getGroup().getValue()]);
        break;
      ...
    } else if (theEvent.isController()) {
      println("event from controller: "+theEvent.getController().getValue()+" from "+theEvent.getController());
      switch (theEvent.getController().getId()) {
        case (3):
          delay = millis()–PIDtime;
          if (delay > 3000) {
            println("PID Values");
            println(float PropField.getText());
            println(float IntField.getText());
            println(float DerField.getText());
            println(float SetPointField.getText());
            println(float d12.getValue());
            println(float d13.getValue());
            serialPrinterGains();
            println("Sent package");
            PIDtime = millis();
          } else {
            messageBoxPIDData.show();
          }
        break;
        case (4):
        ...
```
Both the serialPrinterGains and serialPrinterServos methods call the floatArrayToByteArray function for sending the Arduino over the serial communication port floats. The primary reason for this is that the user will most likely enter values that are non-integer values for tuning the PID controller. On line 432 the input to the function is multiplied by a factor of 4 for later use in the conversion to a byte. Line 444 creates a new byte array b of length 4. Then on line 445 the output byte array is created based on the input length to the function. Next the ByteBuffer.wrap function is called for creating a byte buffer for storing the information to later be sent as it is converted into a byte array. Then the for loop on line 437 places the floats into the byte buffer. Lastly on line 441 the output is mapped by the byte array values from b and then returned out from the function to the program's control to be written onto the serial communication port.

Code Block 5: floatArrayToByteArray function

```java
byte [] floatArrayToByteArray(float [] input)
{
    int len=4*input.length;
    int index=0;
    byte [] b=new byte [4];
    byte [] out=new byte [len];
    ByteBuffer buf=ByteBuffer.wrap(b);
    for(int i=0;i<input.length;i++)
    {
        buf.position (0);
        buf.putFloat (input [i]);
        for(int j=0;j<4;j++) out [j+i*4]=b [3-j];
    }
    return out;
}
```
limit of strings in an array is reach, this is set to 10 strings. From there it sorts the strings and parses them into their respectively referenced values for later use. In the communication link between the Arduino and the Processing interface a space separating the strings is used to differentiate between strings. Table 2 shows the structure of the string array that is received from the Arduino in the Processing interface. These strings are separated via spaces and split accordingly with that special character modifier to determine the correct variable to string location. Since the Arduino can send strings that represent floats these values need to be converted into floats for use in graphing them on the Processing interface. Lastly, on line 472 there is an if statement for checking whether this data being received needs to be stored in the memory that Processing has allocated to run in.

Table 2: Processing received Arduino string format example

<table>
<thead>
<tr>
<th>String #1: Proportional</th>
<th>String #2: Integral</th>
<th>String #3: Derivative</th>
<th>String #4: Setpoint</th>
<th>String #5: Direction</th>
<th>String #6: Mode</th>
<th>String #7: LH Servo</th>
<th>String #8: RH Servo</th>
<th>String #9: Tail Servo</th>
<th>String #10: Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>1.00</td>
<td>4.00</td>
<td>50.00</td>
<td>0</td>
<td>1</td>
<td>30.00</td>
<td>25.00</td>
<td>15.00</td>
<td>44.00</td>
</tr>
</tbody>
</table>

---

Combined String="2.00 1.00 4.00 50.00 0 1 30.00 25.00 15.00 44.00"

---

Code Block 6: serialEvent function

```cpp
void serialEvent(Serial myPort) {
    String read=myPort.readStringUntil(10);
    String[] s =split(read, " ");
    PropLabel.setText(s[0]);
    IntLabel.setText(s[1]);
    DerLabel.setText(s[2]);
    SetPointLabel.setText(s[3]);
    Setpoint=Float.parseFloat(s[3]);
    println(s[4]);
    println(s[5]);
    if(s[4].equals("0")) DirectionLabel.setText("Direct");
    else DirectionLabel.setText("Reverse");
    if(s[5].equals("0")) ModeLabel.setText("Automatic");
    else ModeLabel.setText("Manual");
    servoLH=Float.parseFloat(s[6]);
    servoRH=Float.parseFloat(s[7]);
    servoTail=Float.parseFloat(s[8]);
    Input=Float.parseFloat(s[9]);
    Output=Float.parseFloat(s[10]);
    println(s[6]);
    println(s[7]);
    println(s[8]);
    println(s[9]);
    println(s[10]);
    println("Worked");
    if(dataRecording==true) {
        time=millis()−oldTime;
        println("Got here");
        TableRow newRow=table.addRow();
        newRow.setInt("Time (ms)",time);
        newRow.setFloat("Input",Input);
        newRow.setFloat("Output",Output);
        newRow.setFloat("Setpoint",Setpoint);
        newRow.setString("Proportional Gain",s[0]);
        newRow.setString("Integral Gain",s[1]);
        newRow.setString("Derivative Gain",s[2]);
        if(s[4].equals("0")) newRow.setString("Mode","Direct");
        else newRow.setString("Mode","Reverse");
        if(s[5].equals("0")) newRow.setString("Direction","Automatic");
        else newRow.setString("Direction","Manual");
        newRow.setFloat("LH Aileron Servo (degrees)",servoLH);
```

8 of 49

American Institute of Aeronautics and Astronautics
The functionality behind both the START DATA RECORD and STOP DATA RECORD buttons is as follows starting on line 495. The dataRecorder method creates the table object which will be storing all of the data once the START DATA RECORD button is pressed. As seen in code block 6, line 472 the data is then appended onto the already created data table object. On line 512 the data is then saved when the user presses the STOP DATA RECORD button. Line 514 the filename is obtained from the terminal and a .csv file is generated. Then a series of if statements check whether the file already exists in the current working directory. If the file does not exist it proceeds to write the generated data table to the file otherwise it prompts the user with an error explaining that a file with that name already exists in the current working directory. The remaining lines of code for the Processing interface are as follows; lines 531-642 are error prompts and catching to make sure the user does not break the application upon everyday use and lines 644-852 are adapted from another graphical user interface that does real time plots for a PID controller.2

Code Block 7: dataRecord and dataSave functions

```java
void dataRecorder()
{
    table = createTable();
    table.addColumn("Time (ms)");
    table.addColumn("Input");
    table.addColumn("Output");
    table.addColumn("Setpoint");
    table.addColumn("Proportional Gain");
    table.addColumn("Integral Gain");
    table.addColumn("Derivative Gain");
    table.addColumn("Mode");
    table.addColumn("Direction");
    table.addColumn("LH Aileron Servo (degrees) ");
    table.addColumn("RH Aileron Servo (degrees) ");
    table.addColumn("Tail Servo (degrees) ");
}

void dataSave()
{
    String filename = Field.getText()+".csv";
    File f = new File(sketchPath(filename));
    println(sketchPath(filename));
    if (!f.exists())
    {
        println("Here");
        saveTable(table, Field.getText()+".csv");
        myPort.clear();
        dataRecording=false;
    }
    else
    {
        println("File exists");
        MessageBox.show();
    }
}
```

B. Arduino Code

The Arduino code is structured as follows and will be talked about in great detail. For the full length of code see the Appendix C. Lines 6-7 import the essential libraries that the Arduino will be using. The first one is the PID controller library. This library has a few extras associated with it over writing your own PID. A few of the extras include automatic tuning of the gains for the PID controller, changing the sampling period, turning on or off the PID controller for tuning, and output limits. The second library that was used included the servo library for commanding the three servos located on the tail and wings.
Lines 9-41 are where the variables are declared and initialized along with any object variables as well. The Arduino then runs through its setup method upon startup, line 43. Line 45 is where the baud rate is set for serial communication with the Arduino. Here the servo are attached to respective pins digital 9-11 on the Arduino. Since the Arduino was being used to produce approximately a 62 KHz signal the internal timer had to be modified from the default for such a high frequency. The line of code that modified the timer is shown on line 53. This modification maxes out the extent that the Arduino can produce for a pulse width modulation frequency. This frequency was used as the 3 pole Low Pass Buttersworth filter would essentially drop the frequency of the input signal by a factor of 1000. Following that the lines 55 and 56 set the output limits from the PID controller and initialize the PID controller to run.

Code Block 8: setup method

```
void setup()
{
  Serial.begin(19200);
  pinMode(led,OUTPUT);
  myservoRH.attach(9);
  myservoLH.attach(10);
  myservotail.attach(11,600,2400);
  myservoRH.write(95);
  myservoLH.write(80);
  myservotail.write(90);
  TCCR0B = TCCR0B & 0b11111000 | 0x01; // to change time from default multiple everything by 64
  timer=millis();
  myPID.SetOutputLimits(-100,100); // min/max voltages
  myPID.SetMode(AUTOMATIC);
}
```

After the setup block has initialized everything on the Arduino the loop block is entered. Inside this loop block starting at line 61, the Arduino computes an exponentially weighted moving average of the zero voltage reference across the Wheatstone half bridge. This is done for 20 seconds upon startup and only done once per each startup. The value is stored and subtracted out of the system later on. The exponentially weighted moving average used an $\alpha$ value of 0.5 and an added voltage offset of $15 mV$ was added into reduce the error when calculating the initial zero reference voltage. This is shown in lines 66 to 80 in the code block. Eq. (1) was used to compute the exponentially weight moving average where $E_i$ is the exponential average at time $i$, $\alpha$ is the exponential scale factor, $E_{i-1}$ is the previous exponential average, and $P_i$ is the current measurement from the Arduino.

$$E_i = (1-\alpha)E_{i-1} + \alpha P_i$$  \hspace{1cm} (1)

Code Block 9: 20 second exponentially weighted moving average

```
void loop()
{
  timer=millis();
  if(timer<1280000)// 20 seconds converted to timer0 multiply 20000*64
  {
    if(counter>=1)
    {
      sensorValue=analogRead(analogPin);
      voltage_measured=((5.0*sensorValue)/1023)+v_offset;
      value=(1-alpha)*value_prev+(alpha*voltage_measured);
      counter+=1;
      value_prev=value;
    }
    if(counter<1)
    {
      sensorValue=analogRead(analogPin);
      voltage_measured=((5.0*sensorValue)/1023)+v_offset;
      value_prev=voltage_measured;
    }
  }
}
Once the exponentially weighted moving average was taken and recorded then the Arduino proceeds to calculate the initial zero reference voltage value. From there the Arduino then takes a measurement of the circuit off of analog pin zero and records that value. Through the equation for a Wheatstone half bridge configuration the voltage is calculated as shown in Eq. (2).

\[ V_r = \left( \frac{V_{\text{measured}}}{V_{\text{external}}} \right)_{\text{Strained}} - \left( \frac{V_{\text{measured}}}{V_{\text{external}}} \right)_{\text{Unstrained}} \]  

(2)

Where \( V_{\text{measured}} \) is the measured voltage across the Wheatstone half bridge and \( V_{\text{external}} \) is the external voltage source to the Wheatstone half bridge. With this voltage strain is then calculated using Eq. (3).

\[ \varepsilon = -\frac{2V_r}{GF} \left( 1 + \frac{R_1}{R_g} \right) \]  

(3)

Where \( \varepsilon \) is the strain measured in microstrain. Then using a quadratic fitted curve based on the ACX piezo-electric actuators the excitation voltage is found for the piezo-electric actuators and stored as the Input to the system. Eq. (4) shows the fig. 3 digitized using a MATLAB digitization function and then fitted using \texttt{polyfit()}.\textsuperscript{3}

\[ \text{voltage} = -0.00028994\varepsilon^2 + 0.524083569\varepsilon + 2.973830292 \]  

(4)

![Figure 3: Piezo-electric actuator strain versus excitation voltage](image)

Line 106 then computes the output for the PID controller. This output is then mapped to a zero to ten volt range as that is what would be coming out of the Arduino to be sent into the power amplifier input port. Line 110 then remaps the value to the corresponding duty cycle for the pulse width modulation output on digital pin five and it is then written out of digital pin five to the power amplifier. Lastly, the Arduino the call the method \texttt{packageReceive} to listen for anything from the Processing interface. If nothing is being sent to the Arduino then the Arduino proceeds to call the \texttt{packageSend} method to send updated input, output and PID values to the Processing interface.

Code Block 10: Arduino loop after 20 second initialization

```c
if(timer>1280000) // 20 seconds converted to timer0 multiply 20000*64
{
    if(calc_initial_flag==true)
    {
        voltage_measured_initial=value+v_offset;
        //Serial.print("Initial averaged voltage value=");
        //Serial.println(voltage_measured_initial);
        calc_initial_flag=false;
    }
}
```
if (millis()) > serialTime)
{
    sensorValue = analogRead(analogPin);
    voltage_measured_strained = (5.0 * sensorValue) / 1023 + v_offset;
    Vr = (voltage_measured_strained / Vex) - (voltage_measured_initial / Vex) / 100; // divide by circuit gain factor of 100
    strain = Vr * (1 + (100 / 120)) * 1000000; // to convert to micro strain
    Input = a * pow(strain, 2) + b * strain + d; // fitted equation for strain to excitation voltage
    Serial.print("Input= ");
    Serial.print(Input);
    Serial.print("Output= ");
    Serial.print(Output);
    Serial.println(Setpoint);
    myPID.Compute(); // compute the PID Output value
    intermappedValue = map(Output, -100, 100, 0, 10); // map PID output to signal generated range of 0–10V
    mappedValue = map(intermappedValue, 0, 10, 0, 255);
    analogWrite(5, mappedValue); // write 62kHz signal to power amplifier
    packageReceive();
    packageSend();
    serialTime += 25600; // 400 microseconds converted to modified timer0 400*64
}

Just before the packageReceive method the smaller code block from line 119 to 123 is used to store the values that are received by the Arduino from Processing as the byte arrays and then join those values in memory in a union. A union is a data type that allows the Arduino to access the same portion of memory as two different data types. In this case we store it as bytes and then read it as a float. Inside the packageReceive method on line 128 the Arduino is listening to the serial communication port for anything to have been sent. If something has been sent over the serial port then it enters a while loop for listening continuously. Inside the while loop there is a system of structured if statements that allow the same code to be used for multiple scenarios. As shown in table 3 the package can come in either two scenarios. The first scenario is that the user wants to update the gain values and the other functions on the PID controller. The second scenario is that the user wants to update the servo values. In the first scenario a pre identifier string is sent letting the Arduino know what the values it is receiving are. Lines 132 to 137 read in the first five characters to then be combined into that pre identifier string for later comparison. If scenario one is passed to the Arduino then the Arduino moves into the nested if statement on line 138. From there it continues to read the values in the buffer and allocate them to the correct interpreted variables. Lines 140 to 152 read in single values to determine the direction that the PID controller should control about and the mode that the PID controller should be in respectively. If scenario two were to occur then the Arduino would jump to line 153 and would set the index to 7 to break out of the if statement and proceed with reading the values sent to it. Line 160 is where the Arduino continues reading in values and assigning them in memory to be read out as floats later on. Upon completing the reading of all the values the serial buffer is flushed of all values and is ready for the next package to be sent from the Arduino upon completing its task. As the initial scenario message was stored for later use the two defining if statements on line 165 and line 198 now determine how the values in memory should be assigned. In scenario one with the gains the first three values are determined to be the proportional, integral, and derivative gains. Following that the setpoint value is the fourth value read from memory. Then on line 172 the PID gains are set for the controller to use later. Next the mode and direction are checked and set accordingly for the PID controller. Finally for testing purposes the Arduino LED flashes on then off for three seconds to verify that the package was received and executed. If scenario two occurred instead then the first three values would have been the servo angles. They are then mapped to the corresponding output angles that are written to the pulse width modulation pins on the Arduino, which are pins 9, 10, and 11 respectively. A delay of five seconds is added in for testing purposes to verify that the Arduino received and executed the package correctly. Lastly, the initial scenario message is cleared, the mode message is cleared, and the direction message is cleared. Finally the serial is flushed once again just encase anything was still in it.
Table 3: Arduino packageReceive method choices

<table>
<thead>
<tr>
<th>Pre Identifier</th>
<th>Mode</th>
<th>Direction</th>
<th>Proportional</th>
<th>Integral</th>
<th>Derivative</th>
<th>Setpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gains</td>
<td>0</td>
<td>1</td>
<td>2.00</td>
<td>0.00</td>
<td>1.00</td>
<td>32.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pre Identifier</th>
<th>RH Servo</th>
<th>LH Servo</th>
<th>Tail Servo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servos</td>
<td>15.00</td>
<td>25.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Code Block 11: packageReceive method

```cpp
union {
    byte asBytes[12];
    float asFloats[3];
} foo;

void packageReceive ()
{
    int index=0;
    while(Serial.available())
    {
        if(index<7)
        {
            if(index<5)
            {
                char c=Serial.read();
                msg.concat(c);
                index++;
            }
            if(msg=="Gains")
            {
                if(index==5)
                {
                    char h=Serial.read();
                    Direct.concat(h);
                    index++;
                }
                if(index==6)
                {
                    char b=Serial.read();
                    Mode.concat(b);
                    index++;
                }
            }
            if(msg=="Servo")
            {
                index=7;
            }
        }
        else
        {
            foo.asBytes[index-7]=Serial.read();
            index++;
        }
    }
    Serial.flush();
    if(msg=="Gains")
    {
```

American Institute of Aeronautics and Astronautics

13 of 49
double p, i, d;
p = double(foo.asFloats[0]);
i = double(foo.asFloats[1]);
d = double(foo.asFloats[2]);
Setpoint = double(foo.asFloats[3]);
myPID.SetTunings(p, i, d);
if (Mode == "0")
    {
        myPID.SetMode(AUTOMATIC);
    }
else if (Mode == "1")
    {
        myPID.SetMode(MANUAL);
    }
if (Direct == "0")
    {
        myPID.SetControllerDirection(DIRECT);
    }
else if (Direct == "1")
    {
        myPID.SetControllerDirection(REVERSE);
    }
digitalWrite(led, HIGH);
delay(64000);
digitalWrite(led, LOW);
delay(64000);
digitalWrite(led, HIGH);
delay(64000);
digitalWrite(led, LOW);
delay(640);
if (msg == "Servo")
    {
        double RH, LH, tail;
        RH = double(foo.asFloats[0]);
        LH = double(foo.asFloats[1]);
        tail = double(foo.asFloats[2]);
        RH = map(RH, -35, 35, 60, 130);
        LH = map(LH, -35, 35, 115, 45);
        tail = map(tail, -20, 20, 70, 110);
        myservoRH.write(RH);
        myservoLH.write(LH);
        myservotail.write(tail);
        digitalWrite(led, HIGH);
        delay(320000);
        digitalWrite(led, LOW);
    }
    msg = "";
    Mode = "";
    Direct = "";
    Serial.flush();
}

Inside the packageSend method the structure of data that the Arduino send the Processing interface can be seen and is shown in table 2. These values are sent through the serial communication port and then read in and updated accordingly by the Processing interface that the user sees in near real time.

Code Block 12: packageSend method

```c
void packageSend()
{
    Serial.print(myPID.GetKp());
    Serial.print(" ");
    Serial.print(myPID.GetKi());
    Serial.print(" ");
    Serial.print(myPID.GetKd());
    Serial.print(" ");
    Serial.print(Setpoint);
    Serial.print(" ");
}```
2. Hardware

After deciding on a 3 pole Low Pass Buttersworth filter the sizing of the resistors and capacitors was conducted. Table 4 shows the theoretical values and the physical values used in the filter. Figure 1 shows the 3 pole Low Pass Buttersworth filter layout. It can be noted that the reasoning for not choosing the two pole RC filter was because the bandwidth of operational amplifier comes into play across the feedback capacitor. The feedback capacitor only really works if it is operating within the bandwidth of the operational amplifier.

<table>
<thead>
<tr>
<th>Theory</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 (kΩ)</td>
<td>R2 (kΩ)</td>
</tr>
<tr>
<td>39.5</td>
<td>37.1</td>
</tr>
<tr>
<td>38</td>
<td>35</td>
</tr>
</tbody>
</table>

After deciding on a 3 pole Low Pass Butterworth filter with a gain of two the capacitors and resistors had to be sized accordingly with a cutoff frequency, \( F_c \), equal to 62 Hz. The primary driver that needed to be taken into account for designing the 3 pole Low Pass Butterworth filter was being able to buy off the shelf resistor and capacitor parts. A sizing script written in MATALB\(^4\) was used to determine the resistor values based on set capacitor values. The initial capacitor values were chosen to be 0.01 µF for all C1, C2, and C3 as shown in table 4. Using the relationship for cutoff frequency and testing demonstrated from an online source in designing Buttersworth filters the following equation was used to solve for a scalar x value.\(^5\)

\[
F_c = \frac{1}{2\pi xy}
\]

\[
x = \frac{1}{2\pi F_c y}
\]

Where \( F_c \) is the cutoff frequency in Hz, \( x \) is the scaling factor for the resistor values, and \( y \) is the scaling factor for the capacitor values. Since the capacitor values were all equal to each other the value for \( y \) was 0.001 based on table 5 and the initial values we wanted for the capacitors.

\[
y = \frac{C_{desired}}{C_{table}}
\]

Where \( C_{desired} \) is the values that were chosen initially for the capacitors and \( C_{table} \) are the values in the table for the appropriate gain factor. In the design of the filter a gain factor of 2 or 6dB was chosen because the output signal from the Arduino ranges from 0 to 5V and a 0 to +10V signal was needed so the Arduino signal needed to be doubled to obtain the magnitude.

From this table and Eq. (6) the value for \( x \) was determined to be 2.5263. This then scaled the values read from table 5 and the finalized resistor values were determined and can be seen in table 4. The initial design of the Buttersworth filter as shown in fig. 1 would require values for resistors that are not possible.
Table 5: Online Buttersworth sizing table

<table>
<thead>
<tr>
<th>M(Kac)</th>
<th>0dB</th>
<th>6dB</th>
<th>12dB</th>
<th>18dB</th>
<th>24dB</th>
<th>30dB</th>
<th>36dB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>1292</td>
<td>15652</td>
<td>1624</td>
<td>4305</td>
<td>3246</td>
<td>1437</td>
<td>3234</td>
</tr>
<tr>
<td>$R_2$</td>
<td>2093</td>
<td>14694</td>
<td>4067</td>
<td>1750</td>
<td>2134</td>
<td>16260</td>
<td>7198</td>
</tr>
<tr>
<td>$R_3$</td>
<td>3698</td>
<td>4348</td>
<td>15144</td>
<td>13276</td>
<td>1444</td>
<td>42794</td>
<td>42950</td>
</tr>
<tr>
<td>$R_4$</td>
<td>0</td>
<td>10000</td>
<td>30000</td>
<td>70000</td>
<td>15000</td>
<td>31000</td>
<td>63000</td>
</tr>
<tr>
<td>$R_5$</td>
<td>infinite</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>$C_1$</td>
<td>1.E-3</td>
<td>100.E-6</td>
<td>1.E-3</td>
<td>1.E-3</td>
<td>1.E-3</td>
<td>1.E-3</td>
<td>1.E-3</td>
</tr>
<tr>
<td>$C_2$</td>
<td>1.E-3</td>
<td>100.E-6</td>
<td>100.E-6</td>
<td>100.E-6</td>
<td>100.E-6</td>
<td>10.E-6</td>
<td>10.E-6</td>
</tr>
<tr>
<td>$C_3$</td>
<td>100.E-6</td>
<td>100.E-6</td>
<td>100.E-6</td>
<td>100.E-6</td>
<td>1.E-3</td>
<td>100.E-6</td>
<td>100.E-6</td>
</tr>
</tbody>
</table>

To obtain off the shelf easily. To fix this problem the value were rounded to the nearest resistor value and a potentiometer was added into the circuit design to adjust the output voltage. This new design can be seen in fig. 4 where the potentiometer is connected to the negative terminal of the operational amplifier and in between $R_4$ and $R_5$. The final resistor values can be seen in table 4.

![3 pole Low Pass Buttersworth filter with potentiometer](image)

Figure 4: 3 pole Low Pass Buttersworth filter with potentiometer

The next piece of hardware that was designed was the amplification circuit and Wheatstone bridge that amplifies the signal from the strain gages. The initial idea was to reuse the instrumentation amplifier that was used in the original paper but it might be possible to amplify the signal enough and have it be of decent quality from using a regular operational amplifier. Since the LN158-N operational amplifier chip was already purchased and had two operational amplifiers on the chip this option was pursued with a back up of the instrumentation amplifier encase it did not work. The gain for the circuit was first sized at 100 from the original paper followed by the cutoff frequency for the circuit. To not interfere with the frequency of the 3 pole Low Pass Buttersworth filter and the natural frequency of the mechanical system the cutoff frequency for the amplifying circuit was chosen to about 600 Hz, an order of magnitude higher than the filter. To obtain a gain of 100 as used in the original project the following equation was used.
\[ V_{out} = (V_2 - V_1) \frac{R_3}{R_1} \]  \hspace{1cm} (8)

Where \( V_{out} \) is the output voltage to the system in volts, \( V_2 \) is the negative terminal voltage in volts, \( V_1 \) is the positive terminal voltage in volts, \( R_3 \) is the resistor in the feedback loop and off the negative terminal in \( \Omega \), and \( R_1 \) is the resistor at both the ends where the voltage difference is measured in \( \Omega \) or as depicted in fig. 5 by \( V_1 \) and \( V_2 \). Next, the size of the resistors for the amplifying circuit the following equation was used.

\[ F_c = \frac{1}{2 \pi R_3 C} \]  \hspace{1cm} (9)

Where \( F_c \) is the cutoff frequency in \( Hz \), \( R_3 \) is the resistor in \( \Omega \) in the feedback loop and off the negative terminal, and \( C \) is the capacitor in \( pF \) in the feedback loop. With initial calculations the value for \( C \) was 265 \( pF \) which is not an easily obtainable capacitor value. So a redesign of the circuit was performed to obtain an off the shelf parts list. Table 6 shows these new values. This changed the value for the cutoff frequency to roughly half the theoretical, which is still much higher than the other frequencies in the system. The new value of the cutoff frequency is approximately 338 \( Hz \).

<table>
<thead>
<tr>
<th>Theory</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1(\Omega) )</td>
<td>( 10 )</td>
</tr>
</tbody>
</table>

Table 6: Strain gage amplifier circuit values
With the strain gage amplifier circuit sized a voltage reference was added to the circuit. This was done because the original operational amplifier that was bought was unipolar and could only stay within the positive voltage range. The voltage reference added a DC bias to the circuit of approximately +2.5V. With this added the strain gages could have a negative voltage difference as the DC bias offset this and a measurement could be taken. The key portion to choosing a MAX6225BCPA voltage reference chip was that it was insensitive to temperature and the power supply which made it a static constant in the circuit. The accuracy for the voltage reference chip was not as important as the software is designed to remove this value through using a 20 second exponentially weighted moving average.

The strain gages were then configured in a half bridge configuration due to the power amplifier potentially not having enough current to drive all four piezo-electric actuators at the same time. Figure 5 shows this configuration. When designing the Wheatstone bridge the resistors were sized based on the resistance values of the strain gages. Measuring the strain gages the resistant values were approximately 120 Ω. Off the shelf resistors that were close to that value were found, they are sized at 100 Ω and with a power rating of one watt. This power rating could have been dropped as the calculation that was performed via Eq. (10) shows that the resulting power dissipated was only 0.1225 watts.

\[
P_{\text{dissipated}} = \frac{V_1^2}{R}
\]  

Where \( V_1 \) is the voltage in between the resistors on the Wheatstone half bridge, which is 3.5V approxi-
mately, and $R$ is the resistor value in the Wheatstone half bridge, which is $100 \, \Omega$. An overview of the full system can be seen in fig. 6. In an effort to not electrically load the strain gage bridge the resistor, $R_1$, was sized to have a much higher impedance that the strain gages and the $100 \, \Omega$ resistors in the strain gage bridge. With the resistors in the strain gage amplifying circuit being two order of magnitudes higher than the $100 \, \Omega$ resistors, the bridge did not experience a larger electrical load then if they were closer to the $100 \, \Omega$ resistors. With the circuit being designed around a higher input impedance the signal amplitude was unaffected which resulted in a clearer signal.

![Figure 6: Overall schematic of the system](image)

**III. Apparatus**

This project comprised of two major portions of support equipment; the hardware and the software. Both finalized designs were discussed in the final design section of this paper. The support hardware is divided into four major components. These components include; the model, the power amplifier, the Arduino, and the protoboard 3 pole Low Pass Buttersworth filter and Wheatstone half bridge. Each of these components will be discussed in some detail with diagrams and pictures.

**A. Model**

The model was originally from another project and is approximately 12 years old, dating back from 2001. It comprises of a gyro mounted in the nose section of the aircraft, three servos, four strain gages, and four piezo-electric actuators. The strain gages are mounted parallel to the piezo-electric actuators to measure the bending strain. The piezo-electric actuators are mounted in a weakened cross section on the model’s fiberglass tail structure. There is a servo in each wing to drive the ailerons and one servo mounted to the tail to drive the elevator. The gyro is mounted inside a foam cutout which resides in the front of the nose section. Figure 7 shows the overall model size and layout of everything while fig. 8 shows the piezo-electric actuators and strain gages with the tail mounted servo. Lastly, fig. 9 shows the servos in the wings and the gyro three pin connector mounted in the nose. The wing is made from foam with a heavy duty coating while the nose is made from hardened plastic.
B. Power Amplifier

The power amplifier is made by Piezo Systems and is model EPA-104. It has an input BNC port, a gain knob, a DC voltage offset knob, a polarity knob, and output banana ports. The device is capable of $\pm 200\text{V}$ range for driving the piezo-electric actuators and has a max current draw of 200mA. The max gain is 20 with a max DC voltage offset of 200V in either positive or negative direction. Figure 10 shows the power amplifier and the interface panel.

C. Arduino

The Arduino is made by Arduino and is an Arduino Uno R3 model. It has a 16 $MHz$ clock speed with 14 digital I/O pins (6 for pulse width modulation), 6 analog pins, 32KB of flash memory, 2 KB of SRAM, 1 KB of EEPROM, and an on board pulse width modulation of 8 bits, and and A/D converter of 10 bits.6 The
Arduino is a fairly robust platform with many options for upgrading or improving the existing system. The Arduino is programmed in the Arduino language which is based on Processing language which is C. Figure 11 shows what the Arduino looks like. It is approximately 2.7 by 2.1 inches with some overhang of the USB and power connectors. It has a height of approximately 0.5 inches tall.

![Arduino Uno R3 pin layout](image)

**Figure 11: Arduino Uno R3 pin layout**

D. Protoboard

The protoboard is comprised of many different components. Starting with the 3 pole Low Pass Butterworth filter it comprises of 11 kΩ, 27 kΩ, 35 kΩ, and 38 kΩ resistors. It also has 0.01 µF and 4.7 µF capacitors. The operational amplifier chip that is being used is the LM158-N made by Texas Instruments. This circuit utilizes the operational amplifier on pins 5-8. The Wheatstone half bridge circuit composes of 100 Ω, 10 kΩ, and 1 MΩ resistors. The capacitors that are used include: 470 pF and a 0.01 µF. The operational amplifier that is used is the LM158-N pins 1-4. The voltage reference chip that is used is the MAX6225BCPA+ND made by Maxim Integrated. There are also two 2.5mm breadboard mounted DC jacks mounted to the protoboard. One supplies the power for the operational amplifier while the other supplies power to the servo block. Figure 12 and fig. 13 shows the layout of the protoboard.

![Protoboard layout](image)

**Figure 12: Protoboard layout**

![Protoboard layout continued](image)

**Figure 13: Protoboard layout continued**

IV. Learning Milestones

The scope of this project was to create a working apparatus for students and teachers to be able to use and expand upon in the future. Although the results from the control system were not desirable the hardware and software to operate this lab were set in place. The initial stages of the project started with a model that was more than 10 years old with little to no support hardware associated with it. To then
becoming a fully functional model with support hardware and software that can be expanded upon. In the process a 3 pole Low Pass Buttersworth filter was designed and built, a Wheatstone half bridge with a gain factor of 100 was designed, and a support graphical user interface was constructed. With this foundation implemented the project was deemed a huge success as it shows that the model can be improved upon but also be used to some extent in the current condition. Throughout this project many milestones of learning were achieved and many realizations for how projects are managed in time and money was also realized.

From working on this project many different milestones were reached. Some of these milestones included learning how to design more advanced filters and converting floats to byte arrays. This happened to be one of the very first milestones as when investigating the software portion of this project the Processing language had no way of sending a float data type over the serial communication port to the Arduino so it had to be down converter to a byte array and sent that way. Scouring the Internet resources it was finally found as to how to even do that. It was quite the learning experience as there are many factors to consider when dealing with device to device communication and data type configuration is an important one to know. Another milestone that was reached early on was that of designing a 3 pole Low Pass Buttersworth filter. Understanding how filters work to a deeper level and how different type can give different results as well was achieved. The other main portion was the difference between the active and passive filtering which led to an improved filter design from the original paper using the same parts. These initial milestones opened the door for even more milestones on the second half of this project during spring quarter.

The biggest milestone both quarters was understanding the original paper and what was done before fully proceeding. This partially hindered the project development and direction that it could have been taken. In any project there is the possibility of not fully understanding what was done previously due to bad documentation, lack of understanding, or time. It should be noted that taking the time to initially better understand what was done and why it was done is greatly beneficial and will save a lot of time in the end especially when under the gun to meet deadlines. In the project it was neglected that the voltage range from the computer to the power amplifier swept a range of -10 to +10v which is coupled with the piezo-electric actuators as they either deflect up or down depending on the voltage polarity. Another was that the instrumentation amplifier was used to amplify the signal coming from the full Wheatstone bridge and send it to the computer. Luckily for this it was found that the signal could be amplified with a regular op-amp that was collocated in the same chip as the 3 pole Low Pass Buttersworth filter. As the project neared completion these initial errors caused expenses to increase and time to be lost due to redesigns and workarounds for the project. Another major learning milestone was how a Wheatstone bridge works in conjunction with strain gages. Having only used strain gages in classroom environment before and not being too familiar with them it was a huge milestone in understanding what exactly they are doing and how the work together to produce voltage differences to measure the bending strain. The other milestone with the Wheatstone bridge was how the different configurations can be used to measure strain and when they might be useful for using to take measurements.

As half of this project was hardware and the other half being software there were many milestones in regards to the software side of the project. One of them in particular was learning a new programming language and how it functioned which was the Processing language. Like the Arduino language which is based on C a broader understanding was developed throughout the project when creating the graphical user interface and communicating with the Arduino over the serial communication port. Another was expanding my knowledge of what the Arduino is capable of. With any new or unfamiliar language most of the battle to learn it is fought with what can it do and is there a function that can help me do what I want to do. This project really helped with improving my understanding and obtaining a better grasp of the Arduino and Processing languages for later use in hobbies or project development.

Overall this project entailed many milestones for learning and self realization. It showed me that time management and understand play a crucial role in any project and should never be neglected. Also, that the last 20% of the work takes 80% of the time to complete in almost any project. It also showed me that there are of course many ways to solve a problem and each has its own pros and cons associated with it. It is unfortunate that the Senior Project is being removed in the Aerospace department as I felt with the right project and advisor the potential for going above and beyond in self discovery and improvement is limitless.
V. Future Considerations and Improvements

Looking back at this project there were many places that could have vastly been improved upon to create a better project and platform for teachers and students in the future. A major one would be to rewire and rework the harnesses for the model. Originally the model had a single wire harness that was not easily detachable at the mounting point for the sting mount. This proved to be a good design for routing the wires but it was poorly labeled and a quick disconnect would have been more beneficial for when the model was needing to be mounted in the wind tunnel. The iterated design was one that used more wire bundles which could not as easily be routed through the model and still did not have quick disconnect points for when mounting the model to the sting mount. It was however easier to determine what was connected to what on the model and made connecting the supporting hardware easier than before. The major improvement in the future would be to re-solder wires to the strain gages and piezo-electric actuators and then create a quick disconnect for them near where the sting mount attaches. Then bundling these together and labeling them or color coordinating the wires for easy reference for connecting them to the appropriate hardware would be ideal. If this were to be done it would improve the model greatly in terms of ease of use in a laboratory for teachers and students to assemble and disassemble.

Another consideration for this project with a possible improvement would be to switch out the resistors in the circuits with resistors that have a 1% tolerance on them from the 5% tolerance resistors used to construct the circuitry. The main reason behind this is the amount of extra error attributed to the circuitry. Calculations show that 2.5V should be theoretically going in to the operational amplifier for the Wheatstone bridge instead of the 3.1 to 3.3V that. With any system noise and error are always being driven out and this fix is an easy one that could vastly improve the performance and data of the project. The instrumentation amplifier could also be used to improve the measurements as it is specifically designed to detect minute differences with little noise and error. In designing and redesigning the original project and as mentioned in section IV portions of this project were overlooked and neglected in different stages to then be reevaluated and redesigned at the end. A major portion of this was the setup of only using two strain gages and two piezo-electric actuators. In the original project all four of both the strain gages and the piezo-electric actuators were used to drive the tail structure. A future improvement would be to get all four strain gages connected in a full Wheatstone bridge and connect all four piezo-electric actuators together to drive the system with them. In order to do this the circuitry needs to be modified from a half bridge to a full bridge. Also, the supporting power amplifier needs to be tested with all four piezo-electric actuators running at once to see if it can power them.

In order to obtain the full -200 to +200V range that the piezo-electric actuators can go through the 3 pole Low Pass Buttersworth filter needs to be redesigned with a dual power supply operational amplifier. For simplicity of the design a single power supply was used to power the operational-amplifier. Currently the DC offset on the power amplifier is being used to correct for this problem of getting both polarities of voltage to drive the actuators. It does hinder the project in that a full voltage range is not utilized though the actuators still flex in both directions just at lower strain values. From a software aspect a few things can be expanded and improved upon from what they currently are. One would be to rework the coding so that different control laws can be loaded in for different experiments. This would involve creating a function loaded for users to write their code blocks for their control law and then splicing that into the Arduino code to run. From my understanding it would be possible to modify the code and allow the user to upload their function from the Processing interface and then pass it to the Arduino code to be used. A rework of how the interface acts as well would need to be done as there are many different controller types so the original PID inputs boxes would need to be redone. Overall with these improvements the future laboratory experiments can be replicated, adapted, modified, or run effectively and efficiently.
A. Apparatus Setup Procedure

If the apparatus is not setup then follow these steps in order to get it setup correctly and safely. Note that you will be handling equipment and operating equipment that is running at high voltage so take the necessary precautions when proceeding.

High Voltage

1. Model Setup

Make sure all the wires on the model are properly routed and taped as so they do not flap around or get caught on unexpected objects. If the wires are not properly routed or taped route and or tape them as necessary. There is a set screw and washer that connects the wing to the fuselage. Make sure this is attached and secured tightly. Too much torque will cause the foam to compress and break the coating and weaken the mounting face section. Exhume caution when tightening the set screw.

Figure 14: Wind tunnel mounted model
Don’t over tighten the set screw

After everything is routed, taped and the set screw is tighten the model can now be placed on the sting in the wind tunnel. Mount the sting mount to the model at the base of the model as shown by fig. 15. Tighten until a snug fit is achieved. Next make sure no other projects are in the wind tunnel and that the previous sting mount is removed from the sting. Tighten the sting mount to the sting carefully as to not damage the sting.

![Figure 15: Sting mount connected to model](image)

Carefully mount the sting mount to the sting

Once the model is mounted to the sting carefully route the wire harnesses along the sting mount down the sting arm and through the hole towards the ground. Make sure to zip tie any loose harnesses as to reduce the flapping of them in the wind tunnel for safety reasons and better sting results.

2. Protoboard and Arduino Setup
   
   A. Strain Gages

   Disconnect all power connections before starting

   When connecting the protoboard to the model make sure that all power connector are disconnected before proceeding with any connection. Figure 16 shows each section that will be connected. Starting with the strain gages the list below shows which wire is connected to which strain gage in the strain gage wire harness. Depending on your experiment the use of more or less of these may vary. For this experiment the half bridge was utilized with the back two strain gages. Do not forget the signs for hooking up the
strain gages as if they are not hooked up in this order then compression will be positive and tension will be negative. For compression to be negative and tension to be positive connect the wires in this configuration shown in fig. 17:

1. Positive solid brown to to pin 1
2. Negative white brown to pin 4
3. Positive solid green to pin 2
4. Negative white green to pin 3

Once the strain gages are connected to the protoboard proceed with connecting the strain gage amplifying circuit to the Arduino. Figure 18 shows the signal wire in white which needs to be connected to the Arduino’s analog pin 0 as shown in fig. 19. Now connect one of the grounding wires to the Arduino ground pin above the analog pins.
b. 3 Pole Low Pass Buttersworth Filter

To connect the 3 pole Low Pass Buttersworth filter take the input signal wire, as shown in fig. 20, and connect it to digital pin 5 on the Arduino as shown in fig. 21. Take the second grounding wire from the protoboard and proceed to connect it to the Arduino ground pin. There should now be two grounds going into the Arduino from the protoboard.

c. Servos

Do not forget to connect the servo common ground to the Arduino

For setting up the servos obtain the servos harnesses coming from the model. The following list shows the harness and the color associated with the end connector. Harness number two has a longer end connector to reach the right hand aileron servo. Harness number one connects the tail and left hand servos. Figure 22 shows the pins. The servo connectors should be connected in the following order otherwise the Arduino will
have mapped the wrong servo to the wrong control interface knob. The right hand aileron servo is mapped
to digital pin 9, the left hand aileron servo is mapped to digital pin 10, and the tail servo is mapped to
digital pin 11.

Figure 22: Protoboard servo connector pins

1. Harness #2 Negative solid blue to pin 1
2. Harness #2 Positive orange white to pin 2
3. Harness #2 Signal solid orange to pin 3
4. Arduino digital pin 9 to pin 3
5. Harness #1 Negative blue white to pin 4
6. Harness #1 Positive solid blue to pin 5
7. Harness #1 Signal orange white to pin 6
8. Arduino digital pin 10 to pin 3
9. Harness #1 Negative solid green to pin 7
10. Harness #1 Positive green white to pin 8
11. Harness #1 Signal solid orange to pin 9
12. Arduino digital pin 11 to pin 3
13. Arduino common ground to pin 10

D. Piezo-electric actuators

High voltage be careful when operating the power amplifier

Before setting up the power amplifier it needs to be configured. To configure the power amplifier follow
the steps listed below:

1. Connect a multimeter to the positive and negative outputs of the power amplifier
2. Plug in the power amplifier
3. Turn on the power amplifier
4. Adjust the polarity knob to negative
5. While measuring voltage with the multimeter adjust the DC offset until it reaches -100V
6. Turn the gain knob to x20
7. Turn off the power amplifier
Connect the output of the 3 pole Low Pass Buttersworth filter to the input using the BNC to grabber cable. Connect the ground of the filter to the ground on the BNC cable as well. Now connect the piezo-electric actuator power junction to the positive and negative outputs of the power amplifier as shown in fig. 23.

![Piezo-electric power junction](image)

Figure 23: Piezo-electric power junction

Verify everything is properly grounded before turning on the power amplifier

**E. Finalization**

Verify all connections before turning everything on

Once everything is connected and verified proceed to turn the equipment on in the following order:
1. 3 Pole Low Pass Butersworth filter and Strain gage amplification circuit
2. Servos
3. Arduino
4. Power amplifier
B. Processing Code

```java
/*
Ryan Moskaluk
Senior Project 2012–2013
rwmoskaluk@gmail.com
*/

import processing.serial.*;
import controlP5.*;
import java.nio.ByteBuffer;
Serial myPort;
ControlP5 controlP5;
controlP5.TextField PropField, IntField, DerField, PropLabel, IntLabel, DerLabel,
SetPointField, SetPointLabel, FileField;
controlP5.DropdownList dl, dl2, dl3;
controlP5.Textlabel DirectionLabel, ModeLabel;
controlP5.Knob Servo1Knob, Servo2Knob, Servo3Knob;
ControlGroup messagebox, messageboxData, messageboxPIDData, messageboxServoData,
messageboxDataRecord;
Table table;

PrintWriter output;
int time=0;
int oldTime=0;
int PIDtime=0;
int Servotime=0;
int delay=4000;
boolean firstContact = false; // Whether we've heard from the microcontroller
int [] serialInArray = new int[3]; // Where we'll put what we receive
int serialCount = 0; // A count of how many bytes we receive
int xpos, ypos; // Starting position of the ball
int bgcolor; // Background color
int fgcolor; // Fill color
boolean dataRecording = false; // flag for whether to record data or not
float servoLH=0.0;
float servoRH=0.0;
float servoTail=0.0;
float Setpoint=0.0;
float Input=0.0;
float Output=0.0;
color [] colors = new color[1];

int windowWidth = 1000; // set the size of the
int windowHeight = 400; // form
float InScaleMin = -150; // set the Y-Axis Min
float InScaleMax = 150; // and Max for both
float OutScaleMin = -110; // the top and
float OutScaleMax = 110; // bottom trends

int windowSpan = 300000; // number of mS into the past you want to display
int refreshRate = 100; // how often you want the graph to be reDrawn;

//float displayFactor = 1; //display Time as Milliseconds
float displayFactor = 1000; //display Time as Seconds
//float displayFactor = 60000; //display Time as Minutes

int nextRefresh;
int arrayLength = windowSpan / refreshRate+1;
int [] InputData = new int[arrayLength]; //we might not need them this big, but
int [] SetpointData = new int[arrayLength]; // this is worst case
int [] OutputData = new int[arrayLength];

float inputTop = 25;
//float inputHeight = (windowHeight–70)*2/3;
float inputHeight =175;
```
float outputTop = inputHeight + 50;
// float outputHeight = (windowHeight–70)*1/3;
float outputHeight = 175;

float ioLeft = 300, ioWidth = windowWidth – ioLeft – 50;
float ioRight = ioLeft + ioWidth;
float pointWidth = (ioWidth) / float(arrayLength – 1);

int vertCount = 10;
int nPoints = 0;

boolean madeContact = true;

void setup()
{
  size(1000,600); // width, height
  smooth();
  myPort = new Serial(this, "", 19200);
  myPort.bufferUntil(‘\n’);
  controlP5 = new ControlP5(this);
  dl = controlP5.addDropdownList("COM List")
    .setPosition(20,40)
    .setId(1)
    .customPosition(20,40)
  PropField = controlP5.addTextField("Proportional Gain")
    .setPosition(20,90) //x, y
    .setSize(100,40)
    .setFont(createFont("arial", 20))
    .setAutoClear(false)
    .setInputFilter(ControlP5.FLOAT);
  IntField = controlP5.addTextField("Integral Gain")
    .setPosition(20,150)
    .setSize(100,40)
    .setFont(createFont("arial", 20))
    .setAutoClear(false)
    .setInputFilter(ControlP5.FLOAT);
  DerField = controlP5.addTextField("Derivative Gain")
    .setPosition(20,210)
    .setSize(100,40)
    .setFont(createFont("arial", 20))
    .setAutoClear(false)
    .setInputFilter(ControlP5.FLOAT);
  SetPointField = controlP5.addTextField("PID Setpoint")
    .setPosition(20,270)
    .setSize(100,40)
    .setFont(createFont("arial", 20))
    .setAutoClear(false)
    .setInputFilter(ControlP5.FLOAT);
  dl2 = controlP5.addDropdownList("PID Control Direction")
    .setPosition(20,350)
    .setSize(100,40)
    .setFont(createFont( "arial", 20))
    .setAutoClear(false)
    .setInputFilter(ControlP5.FLOAT);
  dl3 = controlP5.addDropdownList("PID Control Mode")
    .setPosition(20,420)
    .setSize(100,40)
    .setFont(createFont( "arial", 20))
    .setAutoClear(false)
    .setInputFilter(ControlP5.FLOAT);
  PropLabel = controlP5.addTextField("Current Proportional Gain")
137  . setPosition (160, 90) // x, y
138  . setSize (100, 40)
139  . setFont (createFont ("arial", 20))
140  . setAutoClear (false)
141  . setInputFilter (ControlP5.FLOAT);
142  
143  IntLabel = controlP5.addTextField ("Current Integral Gain")
144  . setPosition (160, 150)
145  . setSize (100, 40)
146  . setFont (createFont ("arial", 20))
147  . setAutoClear (false)
148  . setInputFilter (ControlP5.FLOAT);
149  
150  DerLabel = controlP5.addTextField ("Current Derivative Gain")
151  . setPosition (160, 210)
152  . setSize (100, 40)
153  . setFont (createFont ("arial", 20))
154  . setAutoClear (false)
155  . setInputFilter (ControlP5.FLOAT);
156  
157  SetPointLabel = controlP5.addTextField ("Current PID Setpoint")
158  . setPosition (160, 270)
159  . setSize (100, 40)
160  . setFont (createFont ("arial", 20))
161  . setAutoClear (false)
162  . setInputFilter (ControlP5.FLOAT);
163  
164  DirectionLabel = controlP5.addTextField ("Direction label")
165  . setText ("")
166  . setPosition (160, 340)
167  . setFont (createFont ("arial", 20))
168  
169  ModeLabel = controlP5.addTextField ("Modelabel")
170  . setText ("")
171  . setPosition (160, 370)
172  . setFont (createFont ("arial", 20))
173  
174  controlP5.addButton ("Set PID")
175  . setPosition (120, 430)
176  . setSize (80, 40)
177  . setId (3)
178  . getCaptionLabel (). align (ControlP5.CENTER, ControlP5.CENTER)
179  
180  Servo1_Knob = controlP5.addKnob ("LH Aileron")
181  . setRange (-35, 35)
182  . setValue (0)
183  . setNumberOfTickMarks (14)
184  . snapToTickMarks (true)
185  . setPosition (380, 420)
186  . setRadius (80)
187  . setDragDirection (Knob.VERTICAL)
188  
189  Servo2_Knob = controlP5.addKnob ("RH Aileron")
190  . setRange (-35, 35)
191  . setValue (0)
192  . setNumberOfTickMarks (14)
193  . snapToTickMarks (true)
194  . setPosition (580, 420)
195  . setRadius (80)
.setDragDirection(Knob.VERTICAL)
;
Servo3.Knob=controlP5.addKnob("Tail")
.setRange(-20,20)
.setValue(0)
.setNumberOfTickMarks(8)
.snapToTickMarks(true)
.setPosition(780,420)
.setRadius(80)
.setDragDirection(Knob.VERTICAL)
;
controlP5.addBang("Set Servos")
.setPosition(270,420)
.setSize(80,40)
.setId(4)
.getCaptionLabel().align(ControlP5.CENTER, ControlP5.CENTER)
;
FileField=controlP5.addTextfield("Output Filename")
.setPosition(20,500) //x,y
.setSize(200,40)
.setFont(createFont(“arial”,20))
.setAutoClear(false);
controlP5.addBang("Start Data Record")
.setPosition(270,480)
.setSize(80,40)
.setId(5)
.getCaptionLabel().align(ControlP5.CENTER, ControlP5.CENTER)
;
controlP5.addBang("Stop Data Record")
.setPosition(270,540)
.setSize(80,40)
.setId(6)
.getCaptionLabel().align(ControlP5.CENTER, ControlP5.CENTER)
;
errorSameFilename();
dataCollectionBegins();
messageBox.hide();
messageBoxData.hide();
errorPIDtimer();
messageBoxPIDData.hide();
errorServotimer();
messageBoxServoData.hide();
errorDataRecordingAlready();
messageBoxDataRecord.hide();

PropTypes.setText("3.0");
IntField.setText("1.0");
DerField.setText("2.0");
SetPointField.setText("0.0");
FileField.setText("ExampleFilename");
}

void customize(DropdownList ddl)
{
ddl.setBackgroundColor(color(190));
ddl.setItemHeight(20);
ddl.setBarHeight(15);
ddl.citationLabel().set("ARDUINO COM PORT");
ddl.citationLabel().style().marginTop=3;
ddl.citationLabel().style().marginLeft=3;
ddl.valueLabel().style().marginTop=3;
String COMPORTS[] = Serial.list();
for(int i=0;i<=COMPORTS.length-1;i++)
{  
  ddl.addItem(COMPORTS[i], i);
}  
ddl.setColorBackground(color(60));
ddl.setColorActive(color(255, 128));
void customize2(DropdownList ddl)
{
  ddl.setBackgroundColor(color(190));
  ddl.setItemHeight(20);
  ddl.setBarHeight(15);
  ddl.captionLabel().set("PID Control Direction");
  ddl.captionLabel().style().marginTop=3;
  ddl.captionLabel().style().marginLeft=3;
  ddl.valueLabel().style().marginTop=3;
  ddl.addItem("Direct", 0);
  ddl.addItem("Reverse", 1);
  ddl.setValue(0);
  ddl.setColorBackground(color(60));
  ddl.setColorActive(color(255, 128));
}
void customize3(DropdownList ddl)
{
  ddl.setBackgroundColor(color(190));
  ddl.setItemHeight(20);
  ddl.setBarHeight(15);
  ddl.captionLabel().set("PID Control Mode");
  ddl.captionLabel().style().marginTop=3;
  ddl.captionLabel().style().marginLeft=3;
  ddl.valueLabel().style().marginTop=3;
  ddl.addltem("Automatic", 0);
  ddl.addltem("Manual", 1);
  ddl.setValue(0);
  ddl.setColorBackground(color(60));
  ddl.setColorActive(color(255, 128));
}
void draw()
{
  background(0);
  createPlots();
}
void controlEvent(ControlEvent theEvent)
{
  if (theEvent.isGroup())
  {
    println("event from group: "+theEvent.getGroup().getValue()+" from "+theEvent.getGroup());
    switch (theEvent.getGroup().getId())
    {
      case (1):
        myPort.clear();
        myPort.stop();
        myPort = new Serial(this, Serial.list()[int(theEvent.getGroup().getValue())], 19200);
        myPort.bufferUntil(10);
        println(Serial.list()[int(theEvent.getGroup().getValue())]);
        break;
    }
  }  
  else if (theEvent.isController())
  {
    println("event from controller: "+theEvent.getController().getValue()+" from "+theEvent.getController());
    switch (theEvent.getController().getId())
    {
      case (3):
        delay=millis()−PIDtime;
if (delay > 3000)
{
    println("PID Values");
    println(float(PropField.getText()));
    println(float(IntField.getText()));
    println(float(DerField.getText()));
    println(float(SetPointField.getText()));
    println(dl2.getValue());
    println(dl3.getValue());
    serialPrinterGains();
    println("Sent package");
    PIDtime=millis();
}
else
{
    messageboxPIDData.show();
}
break;
case(4):
delay=millis()-Servotime;
if (delay > 3000)
{
    println("Servos");
    println(Servo1_Knob.getValue());
    println(Servo2_Knob.getValue());
    println(Servo3_Knob.getValue());
    serialPrinterServos();
    Servotime=millis();
}
else
{
    messageboxServoData.show();
}
break;
case(5):
if (dataRecording==false)
{
    println("Start Data recording");
    messageboxData.show();
    dataRecording=true;
    dataRecorder();
}
else
{
    messageboxDataRecord.show();
}
break;
case(6):
dataSave();
oldTime=time;
println("Stop Data recording");
broadcast;
}
}

void serialPrinterGains()
{
    float[] packetToSend = new float[4];
packetToSend[0] = float(PropField.getText());
packetToSend[1] = float(IntField.getText());
packetToSend[2] = float(DerField.getText());
packetToSend[3] = float(SetPointField.getText());
myPort.write("Gains");
if (dl2.getValue()==0.0) myPort.write("0");
else myPort.write("1");
if (dl3.getValue()==0.0) myPort.write("1");
myPort.write("0");
}

void serialPrinterServos()
{
    float[] packagetosend=new float[3];
    packagetosend[0]=Servo1_Knob.getValue();
    packagetosend[1]=Servo2_Knob.getValue();
    packagetosend[2]=Servo3_Knob.getValue();
    myPort.write("Servo");
    myPort.write(floatArrayToByteArray(packagetosend));
}

byte[] floatArrayToByteArray(float[] input)
{
    int len=4*input.length;
    int index=0;
    byte[] b=new byte[4];
    byte[] out=new byte[len];
    ByteBuffer buf=ByteBuffer.wrap(b);
    for(int i=0; i<input.length; i++)
    {
        buf.position(0);
        buf.putFloat(input[i]);
        for(int j=0; j<4; j++) out[j+i*4]=b[3-j];
    }
    return out;
}

void serialEvent(Serial myPort)
{
    String read=myPort.readStringUntil(10);
    String[] s =split(read, " ");
    PropLabel.setText(s[0]);
    IntLabel.setText(s[1]);
    DerLabel.setText(s[2]);
    SetPointLabel.setText(s[3]);
    Setpoint=Float.parseFloat(s[3]);
    println(s[4]);
    println(s[5]);
    if(s[4].equals("0")) DirectionLabel.setText("Direct");
    else DirectionLabel.setText("Reverse");
    if(s[5].equals("0")) ModeLabel.setText("Automatic");
    else ModeLabel.setText("Manual");
    servoLH=Float.parseFloat(s[6]);
    servoRH=Float.parseFloat(s[7]);
    servoTail=Float.parseFloat(s[8]);
    Input=Float.parseFloat(s[9]);
    Output=Float.parseFloat(s[10]);
    println(s[6]);
    println(s[7]);
    println(s[8]);
    println(s[9]);
    println(s[10]);
    println("Worked");
    if(dataRecording==true)
    {
        time=millis()−oldTime;
        println("Got here");
        TableRow newRow=table.addRow();
        newRow.setInt("Time (ms)",time);
        newRow.setFloat("Input",Input);
        newRow.setFloat("Output",Output);
        newRow.setString("Setpoint",Setpoint);
        newRow.setString("Proportional Gain",s[0]);
        newRow.setString("Integral Gain",s[1]);
        newRow.setString("Derivative Gain",s[2]);
        if(s[4].equals("0")) newRow.setString("Mode","Direct");
else newRow.setString("Mode", "Reverse");
if(s[5].equals("0")) newRow.setString("Direction", "Automatic");
else newRow.setString("Direction", "Manual");
newRow.setFloat("LH Aileron Servo (degress)", servoLH);
newRow.setFloat("RH Aileron Servo (degress)", servoRH);
newRow.setFloat("Tail Servo (degress)", servoTail);
}

} else
void dataRecorder()
{
table=createTable();
table.addColumn("Time (ms)");
table.addColumn("Input");
table.addColumn("Output");
table.addColumn("Setpoint");
table.addColumn("Proportional Gain");
table.addColumn("Integral Gain");
table.addColumn("Derivative Gain");
table.addColumn("Mode");
table.addColumn("Direction");
table.addColumn("LH Aileron Servo (degress)");
table.addColumn("RH Aileron Servo (degress)");
table.addColumn("Tail Servo (degress)");
}

} else
void dataSave()
{
String filename=FilePath.getText()+".csv";
File f =new File(sketchPath(filename));
println(sketchPath(filename));
if(!f.exists())
{
println("Here");
saveTable(table,FilePath.getText()+".csv");
myPort.clear();
dataRecording=false;
}
else
{
println("File exists");
messageBox.show();
}

} else
void errorSameFilename()
{
messageBox = controlP5.addGroup("messageBox",width/2−80,150,250);
messageBox.setBackgroundHeight(120);
messageBox.setBackgroundColor(color(2,52,77));
messageBox.hideBar();
Textlabel errorLabel = controlP5.addTextlabel("messageBoxLabel","Error file exists already
, rename file and resave",20,20);
errorLabel.moveTo(messageBox);
Button b1 = controlP5.addButton("buttonOKError",0,65,80,80,24);
b1.moveTo(messageBox);
b1.setBackgroundColor(color(40));
b1.setColorActive(color(20));
b1.setBroadcast(false);
b1.setValue(1);
b1.setBroadcast(true);
b1.setCaptionLabel("OK");
}

void buttonOKError(int theValue)
{
messageBox.hide();
}
void dataCollectionBegins()
void buttonOKData(int theValue)
{
    messageboxData.hide();
}

void errorPIDtimer()
{
    messageboxPIDData = controlP5.addGroup("messageBoxPIDData", width/2 - 80,150,250);
    messageboxPIDData.setBackgoundHeight(120);
    messageboxPIDData.setBackgoundColor(color(2,52,77));
    messageboxPIDData.hideBar();
    Textlabel DataPIDLabel = controlP5.addTextlabel("messageBoxPIDLabelData","Please wait 3 seconds while the PID updates",20,20);
    DataPIDLabel.moveTo(messageBoxPIDData);
    Button b12 = controlP5.addButton("buttonOKPIDData",0,65,80,80,24);
    b12.moveTo(messageBoxPIDData);
    b12.setColorBackground(color(40));
    b12.setColorActive(color(20));
    b12.setBroadcast(false);
    b12.setValue(1);
    b12.setBroadcast(true);
    b12.setCaptionLabel("OK");
}

void buttonOKPIDData(int theValue)
{
    messageboxPIDData.hide();
}

void errorServotimer()
{
    messageboxServoData = controlP5.addGroup("messageBoxServoData", width/2 - 80,150,250);
    messageboxServoData.setBackgoundHeight(120);
    messageboxServoData.setBackgoundColor(color(2,52,77));
    messageboxServoData.hideBar();
    Textlabel DataServoLabel = controlP5.addTextlabel("messageBoxServoLabelData","Please wait 3 seconds while the Servos update",20,20);
    DataServoLabel.moveTo(messageBoxServoData);
    Button b13 = controlP5.addButton("buttonOKServoData",0,65,80,80,24);
    b13.moveTo(messageBoxServoData);
    b13.setColorBackground(color(40));
    b13.setColorActive(color(20));
    b13.setBroadcast(false);
    b13.setValue(1);
    b13.setBroadcast(true);
    b13.setCaptionLabel("OK");
}

void buttonOKServoData(int theValue)
{
    messageboxServoData.hide();
}
void errorDataRecordingAlready()
{
    messageboxDataRecord = controlP5.addGroup("messageBoxDataRecord", width/2 - 80, 150, 250);
    messageboxDataRecord.setBackgroundHeight(120);
    messageboxDataRecord.setBackgroundColor(color(2, 52, 77));
    messageboxDataRecord.hideBar();
    Textlabel DataRecordLabel = controlP5.addTextlabel("messageBoxLabelDataRecord", "Data is already being recorded, \n press STOP RECORD DATA \n to stop data collection.", 20, 20);
    DataRecordLabel.moveTo(messageboxDataRecord);
    Button b14 = controlP5.addButton("buttonOKDataRecord", 0, 65, 80, 80, 24);
    b14.moveTo(messageboxDataRecord);
    b14.setColorBackground(color(40));
    b14.setColorActive(color(20));
    b14.setBroadcast(false);
    b14.setValue(1);
    b14.setBroadcast(true);
    b14.setCaptionLabel("OK");
}

void buttonOKDataRecord(int theValue)
{
    messageboxDataRecord.hide();
}

void createPlots()
{
    // ****************************************
    // Adapted from Brett Beauregard’s PID Control GUI
    // ****************************************
    // draw Base, gridlines
    stroke(0);
    fill(230);
    rect(iLeft, inputTop, ioWidth-1, inputHeight);
    rect(iLeft, outputTop, ioWidth-1, outputHeight);
    stroke(210);

    // Section Titles
    // textFont (TitleFont);
    fill(255);
    text("PID Input / Setpoint", (int)iLeft+10,(int)inputTop-5);
    text("PID Output", (int)iLeft+10,(int)outputTop-5);

    // GridLines and Titles
    // textFont (AxisFont);

    // Horizontal grid lines
    int interval = (int)inputHeight/5;
    for(int i=0;i<6;i++)
    {
        if(i>0&&i<5) line(iLeft+1,inputTop+i*interval,iRight-2,inputTop+i*interval);
        text(str((InScaleMax-InScaleMin)/5*(float)(5-i)+InScaleMin),iRight+5,inputTop+i*interval+4);
    }

    interval = (int)outputHeight/5;
    for(int i=0;i<6;i++)
    {
        if(i>0&&i<5) line(iLeft+1,outputTop+i*interval,iRight-2,outputTop+i*interval);
        text(str((OutScaleMax-OutScaleMin)/5*(float)(5-i)+OutScaleMin),iRight+5,outputTop+i*interval+4);
    }

    // Vertical grid lines and TimeStamps
    int elapsedTime = millis();
    int interval = (int)ioWidth/vertCount;
    int shift = elapsedTime*(int)ioWidth/windowSpan;
    shift %= interval;

    // ****************************************
    // Draw Base, gridlines
    // ****************************************
    // Section Titles
    // textFont (TitleFont);
    fill(255);
    text("PID Input / Setpoint", (int)iLeft+10,(int)inputTop-5);
    text("PID Output", (int)iLeft+10,(int)outputTop-5);

    // GridLines and Titles
    // textFont (AxisFont);

    // Horizontal grid lines
    int interval = (int)inputHeight/5;
    for(int i=0;i<6;i++)
    {
        if(i>0&&i<5) line(iLeft+1,inputTop+i*interval,iRight-2,inputTop+i*interval);
        text(str((InScaleMax-InScaleMin)/5*(float)(5-i)+InScaleMin),iRight+5,inputTop+i*interval+4);
    }

    interval = (int)outputHeight/5;
    for(int i=0;i<6;i++)
    {
        if(i>0&&i<5) line(iLeft+1,outputTop+i*interval,iRight-2,outputTop+i*interval);
        text(str((OutScaleMax-OutScaleMin)/5*(float)(5-i)+OutScaleMin),iRight+5,outputTop+i*interval+4);
    }

    // Vertical grid lines and TimeStamps
    int elapsedTime = millis();
    int interval = (int)ioWidth/vertCount;
    int shift = elapsedTime*(int)ioWidth/windowSpan;
    shift %= interval;

    // ****************************************
    // Draw Base, gridlines
    // ****************************************
    // Section Titles
    // textFont (TitleFont);
    fill(255);
    text("PID Input / Setpoint", (int)iLeft+10,(int)inputTop-5);
    text("PID Output", (int)iLeft+10,(int)outputTop-5);

    // GridLines and Titles
    // textFont (AxisFont);

    // Horizontal grid lines
    int interval = (int)inputHeight/5;
    for(int i=0;i<6;i++)
    {
        if(i>0&&i<5) line(iLeft+1,inputTop+i*interval,iRight-2,inputTop+i*interval);
        text(str((InScaleMax-InScaleMin)/5*(float)(5-i)+InScaleMin),iRight+5,inputTop+i*interval+4);
    }

    interval = (int)outputHeight/5;
    for(int i=0;i<6;i++)
    {
        if(i>0&&i<5) line(iLeft+1,outputTop+i*interval,iRight-2,outputTop+i*interval);
        text(str((OutScaleMax-OutScaleMin)/5*(float)(5-i)+OutScaleMin),iRight+5,outputTop+i*interval+4);
    }

    // Vertical grid lines and TimeStamps
    int elapsedTime = millis();
    int interval = (int)ioWidth/vertCount;
    int shift = elapsedTime*(int)ioWidth/windowSpan;
    shift %= interval;
int iTimeInterval = windowSpan/vertCount;
float firstDisplay = (float)(iTimeInterval*(elapsedTime/iTimeInterval))/displayFactor;
float timeInterval = (float)(iTimeInterval)/displayFactor;
for(int i=0;i<vertCount;i++)
{
    int x = (int)ioRight-shift-2-i*interval;
    line(x,inputTop+1,x,inputTop+inputHeight-1);
    line(x,outputTop+1,x,outputTop+outputHeight-1);
    float t = firstDisplay-(float)i*timeInterval;
    if(t>=0) text(str(t),x,outputTop+outputHeight+10);
}
// add the latest data to the data Arrays. the values need
// to be massaged to get them to graph correctly, they
// need to be scaled to fit where they’re going, and
// because 0, 0 is the top left, we need to flip the values.
// this is easier than having the user stand on their head
// to read the graph.
if(millis() > nextRefresh && madeContact)
{
    nextRefresh += refreshRate;
    for(int i=nPoints-1;i>0;i--)
    {
        InputData[i]=InputData[i-1];
        SetpointData[i]=SetpointData[i-1];
        OutputData[i]=OutputData[i-1];
    }
    if (nPoints < arrayLength) nPoints++;
    InputData[0] = int(inputHeight)-int(inputHeight*(Input-InScaleMin)/(InScaleMax-InScaleMin));
    SetpointData[0] =int(inputHeight)-int(inputHeight*(Setpoint-InScaleMin)/(InScaleMax-InScaleMin));
    OutputData[0] = int(outputHeight)-int(outputHeight*(Output-OutScaleMin)/(OutScaleMax-OutScaleMin));
}
//draw lines for the input, setpoint, and output
strokeWeight(2);
for(int i=0; i<nPoints-2; i++)
{
    int X1 = int(ioRight-2-float(i)*pointWidth);
    int X2 = int(ioRight-2-float(i+1)*pointWidth);
    boolean y1Above, y1Below, y2Above, y2Below;
    //DRAW THE INPUT
    boolean drawLine=true;
    stroke(255,0,0);
    int Y1 = InputData[i];
    int Y2 = InputData[i+1];
    y1Above = (Y1<inputHeight); // if both points are outside
    y1Below = (Y1<0); // the min or max, don’t draw the
    y2Above = (Y2<inputHeight); // line. if only one point is
    y2Below = (Y2<0); // outside constrain it to the limit,
    if(y1Above)
    {
        if(y2Above) drawLine=false; //
        else if(y2Below) {
            Y1 = (int)inputHeight; //
            Y2 = 0; //
        }
    }
    else Y1 = (int)inputHeight;
    }
    else if(y1Below) //
    {
}
if(y2Below) drawLine=false; //
else if(y2Above) {
  Y1 = 0;
  Y2 = (int)inputHeight;
} else Y1 = 0;
}

if(y2Below) Y2 = 0;
else if(y2Above) Y2 = (int)inputHeight;

if(drawLine)
{
  line(X1,Y1+inputTop, X2, Y2+inputTop);
}

//DRAW THE SETPOINT
drawLine=true;
stroke(0,255,0);
Y1 = SetpointData[i];
Y2 = SetpointData[i+1];

if(y1Above) (Y1>(int)inputHeight); // if both points are outside
y1Below = (Y1<0); // the min or max, don't draw the
y2Above = (Y2>(int)inputHeight); // line. if only one point is
y2Below = (Y2<0); // outside constrain it to the limit, and leave the other one untouched.

if(y1Above)
{
  if(y2Above) drawLine=false;
  else if(y2Below) {
    Y1 = (int)(inputHeight);
    Y2 = 0;
  } else Y1 = (int)(inputHeight);
}
else if(y1Below)
{
  if(y2Below) drawLine=false;
  else if(y2Above) {
    Y1 = 0;
    Y2 = (int)(inputHeight);
  } else Y1 = 0;
}
else
{
  if(y2Below) Y2 = 0;
  else if(y2Above) Y2 = (int)(inputHeight);
}

if(drawLine)
{
  line(X1, Y1+inputTop, X2, Y2+inputTop);
}

//DRAW THE OUTPUT
drawLine=true;
stroke(0,0,255);
Y1 = OutputData[i];
Y2 = OutputData[i+1];

y1Above = (Y1>outputHeight); // if both points are outside
y1Below = (Y1<0); // the min or max, don't draw the
y2Above = (Y2>outputHeight); // line. if only one point is
y2Below = (Y2<0); // outside constrain it to the limit, and leave the other one untouched.
if(y1Above)
{
  if(y2Above) drawLine=false;
}
else if (y2Below) {
    Y1 = (int)outputHeight;  //
    Y2 = 0;                  //
} else Y1 = (int)outputHeight;  //
}
else if (y1Below) {
    if(y2Below) drawLine=false;  //
    else if(y2Above) {
        Y1 = 0;                  //
        Y2 = (int)outputHeight;  //
    } else Y1 = 0;                //
} else if (y2Below) Y2 = 0;   //
else if(y2Above) Y2 = (int)outputHeight;  //
}

if(drawLine)
{
    line(X1, outputTop + Y1, X2, outputTop + Y2);
}

strokeWeight(1);

C. Arduino Source Code

/*/  
Ryan Moskaluk  
Senior Project 2012-2013  
rwmoskaluk@gmail.com  
*/
#include <PID_v1.h>  
#include <PWMServo.h>  

PWMServo myservotail, myservoRH, myservoLH;  
int led=13;  
String msg="";  
String Direct="";  
String Mode="";  
char c;  
double Input, Output;  
double Setpoint=50.0;  

int analogPin=A0;  
float strain=0.0;  
int vSource=15;  
float voltage_measured=0.0;  
float voltage_measured_strained=0.0;  
float Vr;  
float Vex=6.78;  
float voltage_measured_initial=0.0;  
int counter=0;  
boolean calc_initial_flag=true;  
int sensorValue=641;  
float alpha=0.5;  // for EWMA  
float value_prev=0.0;  
float value=0.0;  
float v_offset=0.015;  // volts  
float a=-0.00028994;  //coefficient a  
float b=0.524083569;  //coefficient b
float d=2.973830292; //coefficient d
int mappedValue=0;
int intermappedValue=0;
PID myPID(&Input,&Output,&Setpoint,10,2,5,DIRECT);
unsigned long time, time1, timer;
unsigned long serialTime;

void setup()
{
  Serial.begin(19200);
pinMode(led,OUTPUT);
myServoRH.attach(9);
myServoLH.attach(10);
myServoTail.attach(11,600,2400);
myServoRH.write(95);
myServoLH.write(80);
myServoTail.write(90);
TCCR0B = TCCR0B & 0b11111000 | 0x01; // to change time from default multiple everything by 64
timer=millis();
myPID.SetOutputLimits(-100,100); //min/max voltages
myPID.SetMode(AUTOMATIC);
}

void loop()
{
timer=millis();

if(timer<1280000)//20 seconds converted to timer0 multiply 20000*64
{
  if(counter>=1)
  {
    sensorValue=analogRead(analogPin);
    voltage_measured=((5.0*sensorValue)/1023)+v_offset;
    value=(1-alpha)*value_prev+(alpha*voltage_measured);
    counter+=1;
    value_prev=value;
  }
  if(counter<1)
  {
    sensorValue=analogRead(analogPin);
    voltage_measured=((5.0*sensorValue)/1023)+v_offset;
    value_prev=voltage_measured;
    counter+=1;
  }

  if(timer>1280000)//20 seconds converted to timer0 multiply 20000*64
  {
    if(calc_initial_flag==true)
    {
      voltage_measured_initial=value+v_offset;
      //Serial.print("Initial averaged voltage value= ");
      //Serial.println(voltage_measured_initial);
      calc_initial_flag=false;
    }
  }

  if(millis()>serialTime)
  {
    sensorValue=analogRead(analogPin);
    voltage_measured_strained=((5.0*sensorValue)/1023)+v_offset;
    Vr=((voltage_measured_strained/Vex)-(voltage_measured_initial/Vex))/100; //divide by circuit gain factor of 100
    strain=Vr*(1+(100/120))*1000000; //to convert to micro strain
    Input=(a*(pow(strain,2)))+(b*strain)+d; //fitted equation for strain to excitation voltage
    //Serial.print("Input= ");
    //Serial.println(Input);
  }
}
// Serial.print(" Output="); // Serial.println(Output); // Serial.print(" Setpoint="); // Serial.println(Setpoint);

myPID.Compute(); // compute the PID Output value

// Serial.print(" Output Afterwards=");

intermappedValue=map(Output,−100,100,0,10); //map PID output to signal generated range of 0−10V

mappedValue=map(intermappedValue,0,10,0,255);

analogWrite(5,mappedValue); //write 62kHz signal to power amplifier

packageReceive();

packageSend();

serialTime+=25600; //400 microseconds converted to modified timer0 400∗64

union {
byte asBytes[12];
float asFloats[3];
}

foo;

void packageReceive()
{
int index=0;

while(Serial.available())
{
if(index<7)
{
if(index<5)
{
char c=Serial.read();
msg.concat(c);
index++;
}
if(msg=="Gains")
{
if(index==5)
{
char h=Serial.read();
Direct.concat(h);
index++;
}
if(index==6)
{
char b=Serial.read();
Mode.concat(b);
index++;
}
if(msg=="Servo")
{
index=7;
}
}
else
{
foo.asBytes[index−7]=Serial.read();
index++;}
}
Serial.flush();

if(msg=="Gains")
{
double p, i, d;
p=double(foo.asFloats[0]);
i=double(foo.asFloats[1]);
d=double(foo.asFloats[2]);
Setpoint = double(foo.asFloats[3]);
myPID.SetTunings(p, i, d);
if (Mode == "0")
    { myPID.SetMode(AUTOMATIC);
    }
else if (Mode == "1")
    { myPID.SetMode(MANUAL);
    }
if (Direct == "0")
    { myPID.SetControllerDirection(DIRECT);
    }
else if (Direct == "1")
    { myPID.SetControllerDirection(REVERSE);
    }
digitalWrite(led, HIGH);
delay(64000);
digitalWrite(led, LOW);
delay(64000);
digitalWrite(led, HIGH);
delay(64000);
digitalWrite(led, LOW);
delay(640);
}
if (msg == "Servo")
{
    double RH, LH, tail;
    RH = double(foo.asFloats[0]);
    LH = double(foo.asFloats[1]);
    tail = double(foo.asFloats[2]);
    RH = map(RH, -35, 35, 60, 130);
    LH = map(LH, -35, 35, 115, 45);
    tail = map(tail, -20, 20, 70, 110);
    myservoRH.write(RH);
    myservoLH.write(LH);
    myservotail.write(tail);
    digitalWrite(led, HIGH);
    delay(320000);
    digitalWrite(led, LOW);
}
msg = "";
Mode = "";
Direct = "";
Serial.flush();
}
void packageSend()
{
    Serial.print(myPID.GetKp());
    Serial.print(" ");
    Serial.print(myPID.GetKi());
    Serial.print(" ");
    Serial.print(myPID.GetKd());
    Serial.print(" ");
    Serial.print(Setpoint);
    Serial.print(" ");
    Serial.print(myPID.GetDirection());
    Serial.print(" ");
    Serial.print(myPID.GetMode());
    Serial.print(" ");
    Serial.print(myservoLH.read());
    Serial.print(" ");
    Serial.print(myservoRH.read());
    Serial.print(" ");
    Serial.print(myservotail.read());
    Serial.print(" ");
}
Serial.print(Input);
Serial.print(" ");
Serial.print(Output);
Serial.println(" ");
Acknowledgments

I would like to thank and acknowledge my advisor Dan Wait for all the good talks and discussions involving this project and life in general. Thanks for pushing me to do good and improve as a person and engineer during my career at Cal Poly.

I would like to thank and acknowledge my Dad for helping me out on the electrical engineering aspect of this project along with consulting on many other aspects of this project as it developed from an idea to a finished product.

I would like thank and acknowledge my Mom for helping make the poster board and gather supplies when I needed them throughout the two quarters that this project encompassed.

Special thanks to \LaTeX for being such an amazing program.

Lastly, special thanks to Processing and Arduino for being awesome development platforms.

2 Beauregard, B., “Arduino PID Tuning Front End, Version 0.2,” May 2009, Arduino and Processing PID GUI.


