Engineering 58-Control Theory and Design

Final Project-Ball Balancing Beam

Kyle Knapp, David Nahmias &

Daniel Kowalyshyn

Introduction:

For our project, we controlled a ball and beam system, in which the beam balances the ball at two different positions that alternate via a square wave input. To do this, we began with a very simple beam system, which we then altered to constrain the motion of the beam to only one direction. We added an infrared sensor to measure the position of the ball on the beam which we then calibrated to our system. Using an Arduino board, we programmed the system to alternate between the two positions and implemented a Proportional Derivative (PD) controller to control the ball's position by altering the angle of the beam. The beam was connected to a servo motor which altered the angle of the beam to control the ball. We created three different PD controllers, one was underdamped, one was optimally damped ($\zeta \approx .7$), and one was overdamped.

Theory:

In order to design a controller for the ball and beam system, a transfer function for the system was derived. Figure 1 depicts a free body diagram of the ball and beam system.

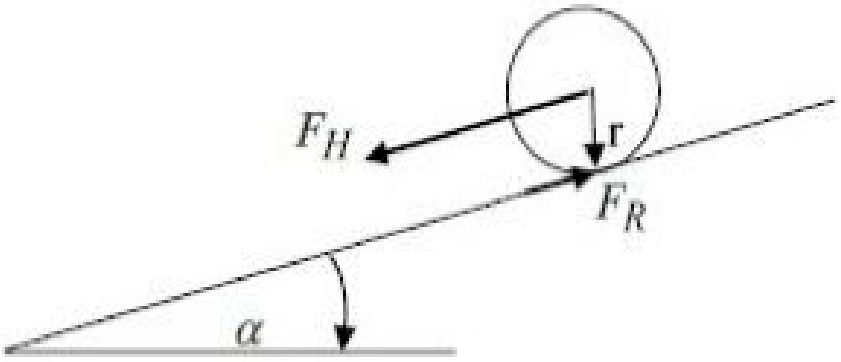

Figure 1. Free Body Diagram of Ball and Beam System.

Based on Figure 1, there are two forces affecting the motion of the ball on the ramp. The forces are: a gravitational force, F_H, and a rotational force, F_R, from the ball rolling down the ramp. Summing the forces acting on the ball in Figure 1 produces Equation 1.

$$m\ddot{x} = F_H - F_R \qquad \textit{(Eq. 1)}$$

Note that the variable *m* represents the mass of the ball and the variable *x* represents the position of the ball on the beam. The gravitational force has components that are parallel to the ramp and perpendicular to the ramp. The parallel component is the one affecting the position of the ball. Using trigonometry, we produce this parallel component shown in Equation 2.

$$F_H = mg \sin \alpha \qquad \textit{(Eq. 2)}$$

However, in this system, the angle of the beam α will always be relatively small, say less than fifteen degrees, because the system would be difficult to control otherwise. A small angle approximation is therefore applied to Equation 2 to produce Equation 3.

$$F_H = mg\alpha \qquad \textit{(Eq. 3)}$$

As to the rotational force F_R, it can be determined from calculating the torque produced by the ball as it rolls, shown in Equation 4.

$$F_R = \frac{T}{r} \qquad \textit{(Eq. 4)}$$

Note that in Equation 4, the variable *T* represents the torque produced by the ball, and the variable *r* represents the radius of the ball. The torque produced by the ball is defined in Equation 5.

$$T = J\dot{\omega} \qquad \textit{(Eq. 5)}$$

As per the notation in Equation 5, the variable *J* refers to the moment of inertia of the ball, and ω refers to the angular velocity of the ball. Equation 6 gives the moment of inertia of the ball.

$$J = \frac{2}{5}mr^2 \qquad \textit{(Eq. 6)}$$

The angular velocity in Equation 5 can be related back to longitudinal velocity as seen in Equation 7.

$$\omega = \frac{\dot{x}}{r} \qquad \textit{(Eq. 7)}$$

Substituting Equation 6 and Equation 7 into Equation 5 produces Equation 8.

$$T = \frac{2}{5}mr\ddot{x} \qquad \textit{(Eq. 8)}$$

Substituting Equation 8 into Equation 4 produces the final form of the rotational force, Equation 9.

$$F_R = \frac{2}{5}m\ddot{x} \qquad \textit{(Eq. 9)}$$

Substituting the gravitational force, Equation 3, and the rotational force, Equation 9, into Equation 1 produces the differential equation of the system, Equation 10.

$$\ddot{x} = \frac{5}{7}g\alpha \qquad \textit{(Eq. 10)}$$

Applying a Laplace transform on Equation 10 produces Equation 11.

$$s^2X(s) = \frac{5}{7}g\Theta(s) \qquad \textit{(Eq. 11)}$$

As per the notation in Equation 11, the variable Θ is the angle of the ramp, which is the input of the system. From Equation 11, the transfer function of the system is determined in Equation 12.

$$G_p(s) = \frac{X(s)}{\Theta(s)} = \frac{\frac{5}{7}g}{s^2} = \frac{K}{s^2} \qquad \textit{(Eq. 12)}$$

The acceleration due to gravity and the *5/7* term are replaced with a single constant *K* in Equation 12. It is more appropriate to consider the plant's transfer function with the constant *K*

because the gain will probably be significantly different from *(5/7)*g* since there are factors that are very hard to model like friction and irregularities on the beam. Therefore, the gain of the transfer function will need to be experimentally determined.

To determine the experimental gain, the response to a step input can be fitted. Multiplying the transfer function in Equation 12 by a step function, *1/s*, produces the Laplace of the step response in Equation 13.

$$X(s) = \frac{K}{s^3} \qquad \textit{(Eq. 13)}$$

Applying an inverse Laplace transform to Equation 13 produces the step response of the system in Equation 14.

$$x(t) = \frac{K}{2}t^2 \qquad \textit{(Eq. 14)}$$

With this derivation, the step response can be plotted and fitted using cftool in MATLAB. Equation 15 is used to fit the response.

$$x(t) = at^2 \qquad \textit{(Eq. 15)}$$

Using the fitted value of *a*, the gain of the system is equal to Equation 16.

$$K = 2a \qquad \textit{(Eq. 16)}$$

Using Equation 12, characteristics of the closed loop system can be determined. By observation, the system is type two because the transfer function has a lone s^2 term in its denominator. The root locus of the uncompensated system is plotted in Figure 2.

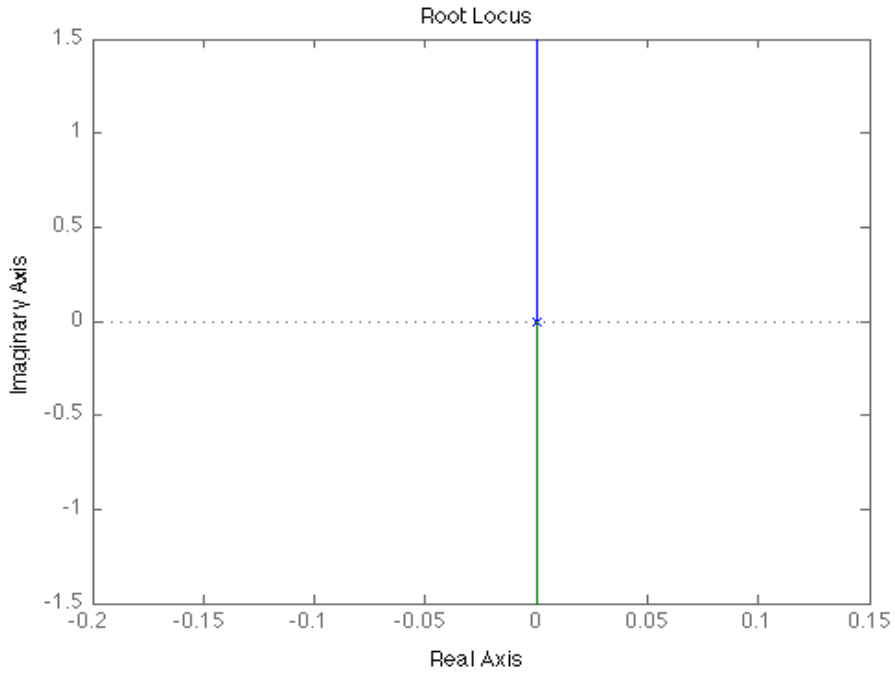

Figure 2. Root Locus of Uncompensated System.

Based on the root locus in Figure 2, the poles of the system begin at the origin and remain on the imaginary axis as the gain is increased. This means the uncompensated system will be marginally stable at best, and a controller needs to be designed such that the poles of the system are pulled into the left hand plane. This can be accomplished by designing a controller consisting of only a zero on the negative region of the real axis. By adding the zero into the open loop transfer function, the two poles will move off the origin because as the gain increases, one of the poles will have to go off to negative infinity and the other pole will have to go to the zero. Figure 3 illustrates this root locus.

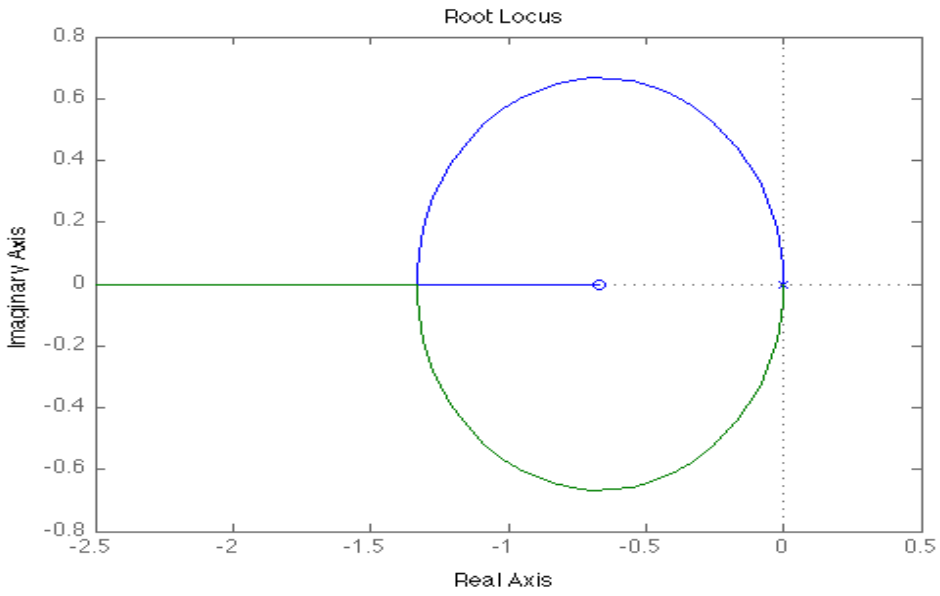

Figure 3. Root Locus with Zero on Negative Real Axis.

In Figure 3, the zero was arbitrarily chosen to be -0.667. As seen in Figure 3, this sort of compensation makes the system stable and allows for control over system's characteristics like the time constant and damping ratio.

The best type of controller to accomplish this task is a proportional derivative (PD) controller. The transfer function for a PD controller is seen in Equation 17.

$$G_c(s) = K_P + K_d s \qquad \textit{(Eq. 17)}$$

By observing Equation 17, the PD controller only places a zero on the negative real axis in the open loop transfer function, as desired. The PD controller is the best controller for this system for multiple reasons. The PD controller makes the system stable and can be easily controlled. The controller does not require an integral term because the system is already a type two system, which theoretically eliminates steady state error for step and ramp inputs. Furthermore, it is not difficult to program a PD controller onto an Arduino. The only possible draw back in this implementation of a PD controller, as seen in Equation 17, is that it has a very

large gain at high frequencies. This means that high frequency noise may be accentuated. This can be avoided by placing a large negative pole, *N*, in the denominator of the derivative term. This would cap the gain of the transfer function at high frequencies. Its transfer function is shown in Equation 18.

$$G_c(s) = K_P + K_d \frac{s}{\frac{s}{N}+1} \qquad \textit{(Eq. 18)}$$

Ideally, the controller in Equation 18 should be used to control the system. However, in the end, it was not used to control the system because it really did not improve the response of the actual system, and it made programming and design techniques a little more complicated because of the extra pole in the denominator of the derivative term.

With our controller from Equation 17, we then only needed to decide the coefficient values to give us pole locations as we desired them. Because of the systems' transfer function, we should have no steady state error regardless of what our proportional gain is, therefore we need not worry about making our proportional gain large to reduce error. Increasing the proportional gain will only increase the speed of the system. Because our system has limits to how much the servo can move the beam, we chose our proportional gain based upon what gain would saturate our system. Then, by means of root locus techniques, we chose our derivative coefficient to give us the damping ratio we desired. The open loop transfer function for the system with the controller is then

$$\left(K_p + K_D s\right)\frac{K}{s^2} \qquad \textit{(Eq. 19)}$$

To use the root locus technique, we must get the equation in the form

$$1 + K_D G(s) H(s) \qquad \textit{(Eq. 20)}$$

To do so, we first find the characteristic equation of the closed loop system, which is given by one plus the open loop function, from Equation 19. We then multiply out the

denominator and divide by every term that does not include the derivative coefficient. After we have done this, we get the following equation, which we can then use to create the root locus diagram.

$$1 + K_D \frac{Ks}{s^2 + KK_p} \qquad \textit{(Eq. 21)}$$

In this equation, K_p is the proportional gain, K is the constant from the plant transfer function, and K_D is the derivative gain, which we are solving for.

Procedure:

The system used in this project was adapted from a previous senior design project. The previous design had several flaws and was significantly improved both physically and with the use of more appropriate sensors and software integration. The first thing that was improved on the system was that the beam at first could slide in two directions on the fulcrum. This made the system not as a stable and applied unnecessary torque to the joints of the arm. Thus, two supports were added to the sides of the fulcrum to limit the beam to one direction of motion. The second physical improvement made on the system was that the there was originally a three-member arm connecting the servo motor to the beam. This design made it so that if the motor were to go over a certain threshold the joints on the arm would buckle. Therefore, in order to avoid this, we eliminated one of the members of the arm. Furthermore, when attaching the arm to the motor, we replaced the plastic parts with more reliable metal parts, as the plastic parts would consistently strip and fail. A picture of the apparatus can be seen in Figure 4.

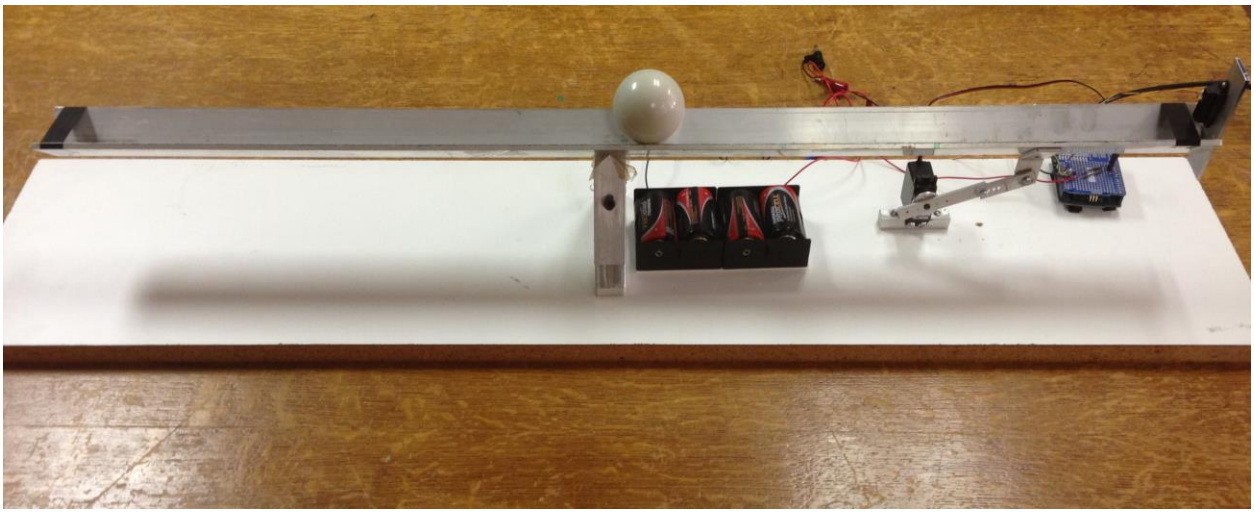

Figure 4. Full apparatus after assembly.

To improve the sensing of the ball we replaced the ultrasonic sensor with a short-range infrared (IR) sensor. The problem with the ultrasonic sensor was that it had too wide of a range, meaning it would pick up obstacles other than the ball rolling down the beam. The IR sensor has a much more narrow range allowing us to precisely track the ball without interference from other objects. To calibrate the IR sensor to our system, we took voltage readings from the IR data using a handheld voltmeter at various positions. We entered this data to MATLAB and made use of the curve-fitting tool in MATLAB to fit the curve. From the IR sensor manufacturer we knew that the curve should be an exponential function. The calibration curve from the manufacturer's website can be seen in Figure 5. The curve we fit, in Figure 6, gives us a relationship between the distances measured and the voltage outputted from the IR sensor. We then solved the equation we got to give us an equation that gives us the distance based off of the voltage reading. As we can see, our curve fits nicely with the data and matches closely to the expected curve from the manufacturer. However, our curve starts at the high point of the curve and drops to zero, so this equation only holds for distances greater than 5 cm. Therefore, when

we pick the locations for the ball to balance at, we must be careful to pick locations that will not have the ball move within 5 cm of the IR sensor, as then our system will not be able to accurately locate the ball. Also, we see that our curve drops to a voltage reading of about zero at distances upwards of 60 cm. Also, because the curve flattens out, there is not a large difference in the readings from 50 cm up. Therefore, when we pick the balancing positions, we choose positions in the range where the voltage readings will not be so similar.

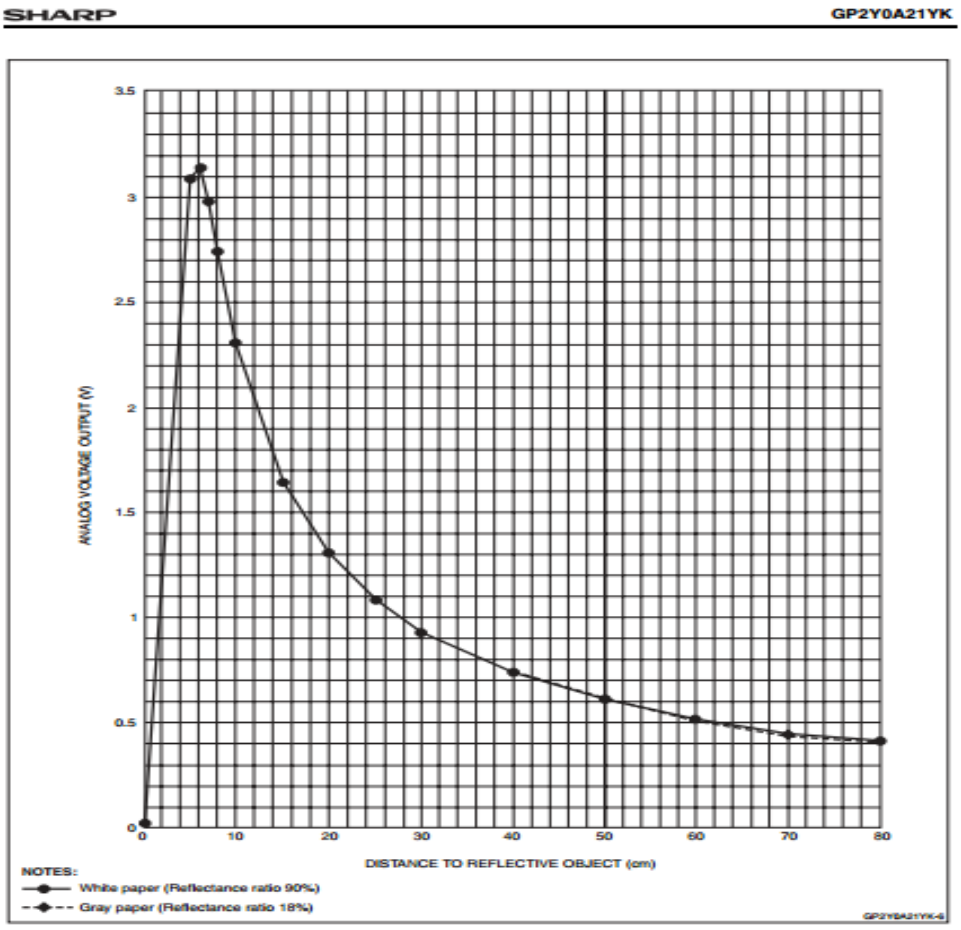

Figure 5. Example Calibration Curve from Manufacturer[1]

[1] SHARP Corporation GP2Y0A21YK Optoelectric Device. http://www.sharpsma.com/webfm_send/1208.

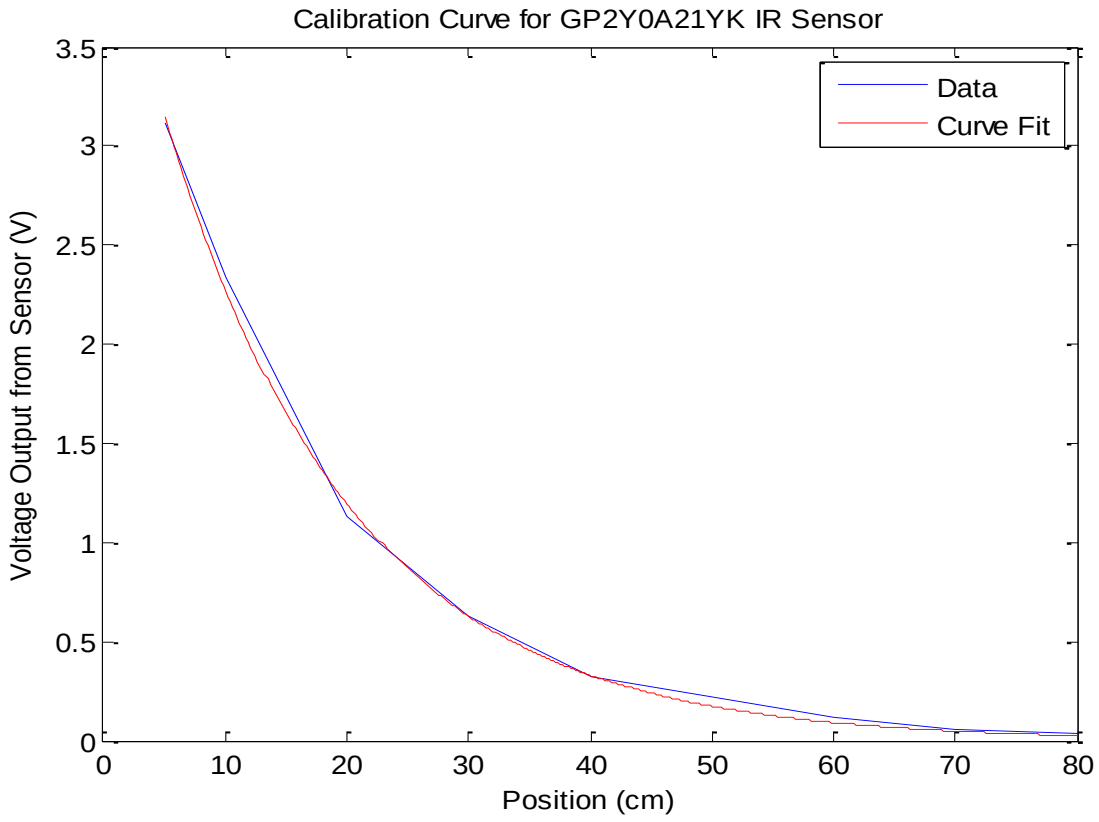

Figure 6. MATLAB Plot of Experimental Calibration and Curve Fit for IR Sensor

Finally, we implemented the controller via an Arduino board. The design and implementation of the code was done via the C and C++ programming language. A program was written and uploaded onto the Arduino in order to control the system. The programs used can be referenced in Appendices A through C. The program essentially consisted of a main while loop that repeated a series of steps. The first step was that it read in a value from the range 0 to 1023 from an analog input, the infrared sensor, on the Arduino board. This value was then converted to a voltage by applying the fact that lowest value zero meant that the Arduino was reading zero volts and these values linearly increased until it reached its highest value of 1023, which meant the Arduino was reading in five volts. Using the experimentally designed calibration curve in Figure 5, the voltage was converted to a distance. However, the IR sensor

did not produce steady measurements even when the object being detected was stationary. So, a median filter of size three was utilized to ensure a more accurate reading. The code for the median filter can be referenced in Appendix C.

Once the distance was determined, the desired position is calculated. This is determined by calculating, based on the current time, where the input should be in its cycle relative to the period. Then, the desired position and the current position are sent to the PID controller, whose code can be referenced in Appendix B. The PID calculates its output, which is in terms of the degrees the servo should be relative to zero degrees. The output is then sent to the servo to move the beam accordingly. Then this whole process is repeated again.

After uploading the code onto the Arduino, we tested the system on a breadboard. After confirming our set up's functionality, we soldered our design onto an Arduino Shield. Finally, in order to implement different controllers, that we designed, we varied the proportional gain and differential gain values in the code for the PID controller while keeping the integral gain value zero to create the PD controller.

Results:

First, the transfer function was experimentally determined. A step function with a size of eight degrees was applied to the system. This was accomplished by commanding the servo to move the angle of the beam from zero degrees to eight degrees. This is considered a step, even though this change in angle is not instantaneous, because the servo moves significantly faster than the ball. Once the step was applied, the distance of the ball was read by the infrared sensor and printed to the terminal window of Arduino. The distances were then plotted in MATLAB and fitted with cftool using Equation 15 as shown in Figure 7.

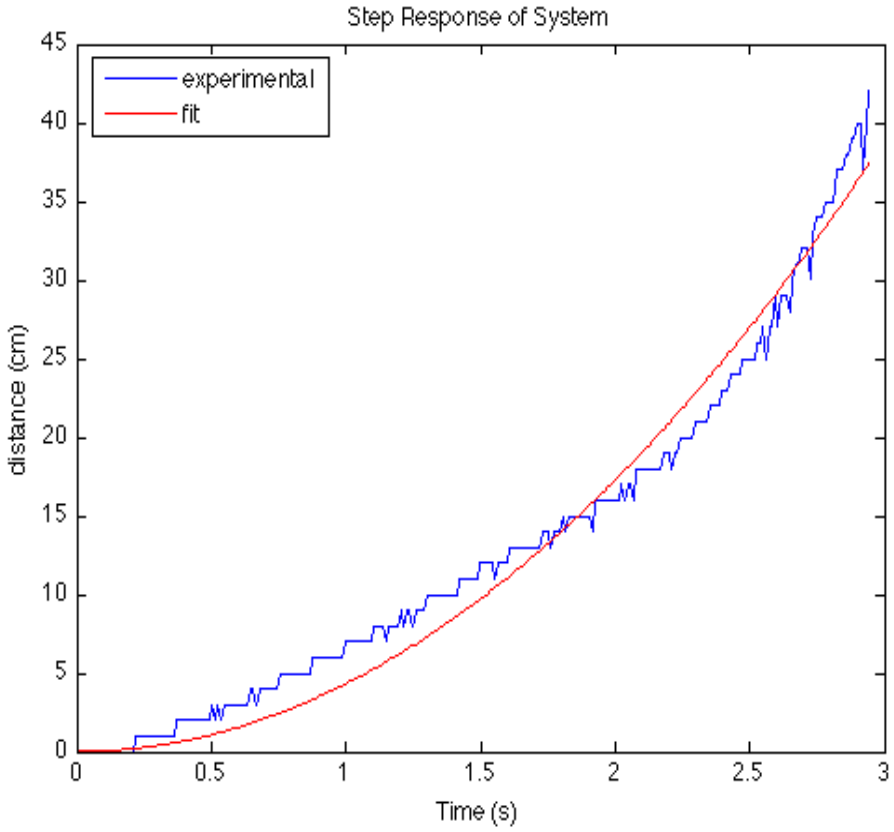

Figure 7. Plot of Experimental Step Response with Fit.

Originally, the experimental data in Figure 7 began at twenty centimeters. So, twenty centimeters was subtracted off the data to begin the response at zero. The components pertinent to the fit are tabulated in Table 1.

Component	**Value**
a	4.319 cm/s^2
Step Input	8 degrees or 0.140 radians
K	0.617 m/radians

Table 1. Important Components from Fitted Step Response.

To calculate the value of the *K*, *a* is first converted to meters. Equation 16 is applied to the value of *a* and is then divided by the step input of 0.140 radians to find the gain of the system, *K*.

Using the design process from the theory section, different proportional gains and derivative gains were calculated in order to produce specific characteristic responses. These gains are listed in Table 2 along with their closed loop poles and damping ratios.

Type of Response	**Proportional Gain (K_p)**	**Derivative Gain (K_d)**	**Poles**	**Damping Ratio (ζ)**
Ideal	0.6	1.4	-0.432 ± 0.429	0.71
Underdamped	0.8	1.3	-0.401 ± 0.577	0.57
Overdamped	0.3	2.5	-.131, -1.14	$\zeta \geq 1.0$

Table 2. PD Designs with Theoretical Pole Locations.

The calculated gains from Table 2 for the ideal response were coded into the controller, and uploaded onto the Arduino. The actual response was then recorded using the infrared sensor. The experimental response, the theoretical response, and the input for the ideal response were then plotted in Figure 8. The code used to produce this plot, and other plots, is located in Appendix D.

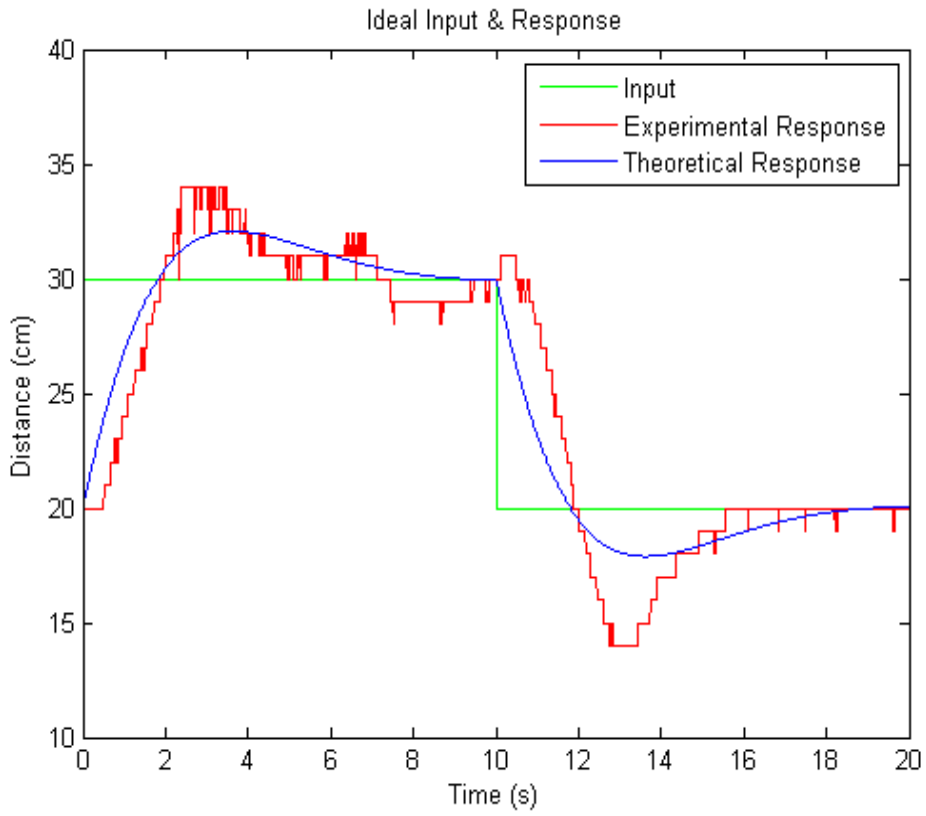

Figure 8. MATLAB Plot of Ideal Response with Input

In Figure 8, the input to the feedback system is a square wave such that the ball starts at twenty centimeters at time equaling zero seconds and moves to thirty centimeters. Then, it moves back to twenty centimeters. The overall period of this square wave is twenty seconds. In order to plot the theoretical response in Figure 8, take note that the input was shifted to zero first. Then, the theoretical response was produced via the *lsim()* function in MATLAB, and the response was then shifted back by the same amount to accurately represent the experimental response. The objective of the ideal design was to produce a response that is slightly underdamped such that there is only a small amount of oscillations. Based on Figure 8, both the theoretical and the experimental response are a little underdamped, as both responses overshoot and return to the steady state value with hardly any oscillations.

The calculated gains from Table 2 for the underdamped response were coded into the controller and then uploaded to the Arduino. The actual response was then recorded. The experimental response, the theoretical response, and the input for the ideal response were then plotted in Figure 9.

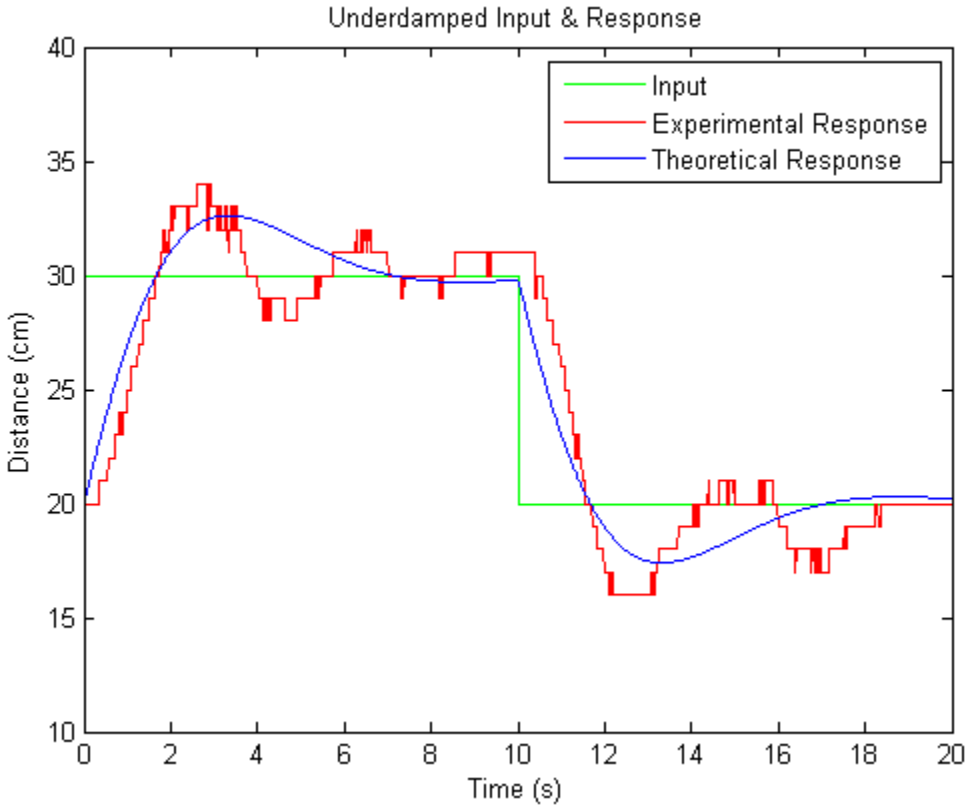

Figure 9. MATLAB Plot of Underdamped Response with Input

The input into the response in Figure 8 is identical to that of Figure 9. In addition, the same modifications used to produce the theoretical response in Figure 8 were used again to produce the theoretical response in Figure 9. The objective of this design was to produce a system response that had a significant amount of oscillations. Based on Figure 9, the experimental response has a few oscillations before it settles at the steady values of thirty and

twenty centimeters. The theoretical response in Figure 9 has fewer oscillations than the experimental response, but it is noticeably more underdamped than the ideal response.

The calculated gains from Table 2 for the overdamped response were coded into the controller and then uploaded to the Arduino. The actual response was then recorded. The experimental response, the theoretical response, and the input for the ideal response were then plotted in Figure 10.

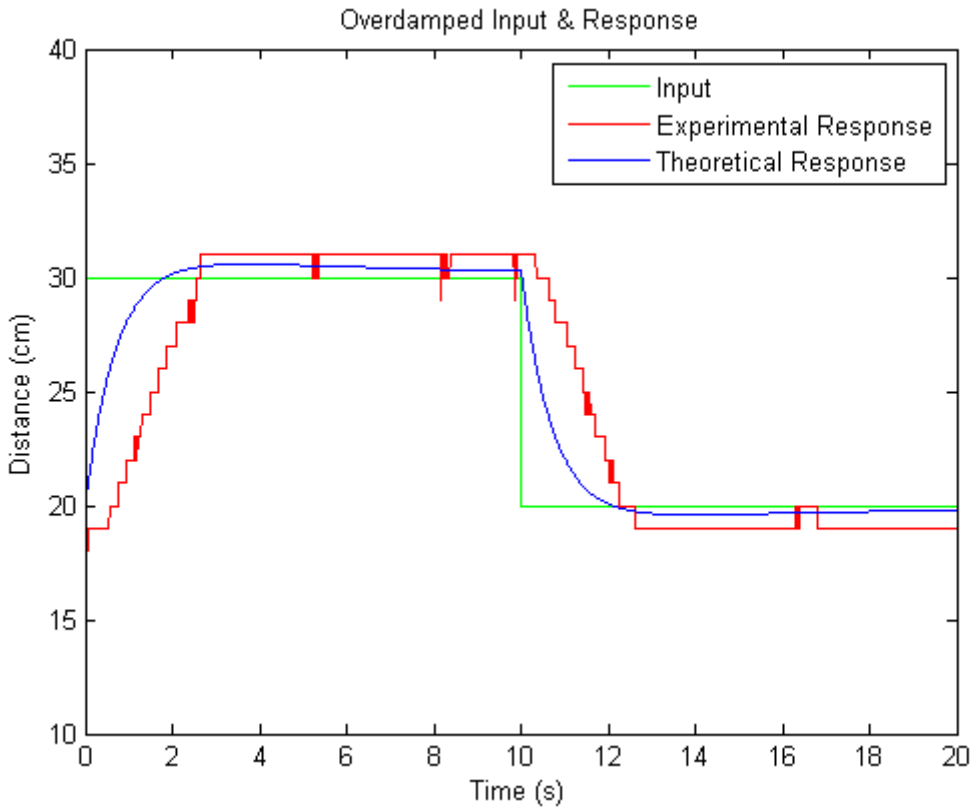

Figure 10. MATLAB Plot of Overdamped Response with Input

The input into the response in Figure 8 is identical to that of Figure 10. In addition, the same modifications used to produce the theoretical response in Figure 8 were used again to produce the theoretical response in Figure 10. The objective of this design was to produce a

response that had little to no overshoot and no oscillations. Based on Figure 10, the theoretical response has a small amount of overshoot, but it asymptotically returns back to the steady sate values of thirty and twenty centimeters. The experimental response is similar to the theoretical response. However, the experimental response does not return to the steady state values.

Discussion:

Based on Figures 8, 9, and 10 from the results section, the experimental responses deviate by a certain degree from the theoretical responses. By observing Figure 8, there is only a very small deviation between the theoretical response and the experimental response. The main deviation is that the experimental response overshoots a small amount more and experiences more overshoot than the theoretical response. The most probable reason why this happens is that there may be some delay in the system because the infrared sensor cannot read distances continuously. The sensor requires a small setup time of a few milliseconds in order to produce accurate readings. As a result, it will cause the system to become more underdamped than desired because the controller is adjusting the plant based off readings that are slightly delayed.

For the underdamped case, the discrepancy is much more noticeable. Similar to the ideal response, the experimental response is more underdamped than the theoretical response. Some of the discrepancy can be accredited to the delay in the system. However, there are probably other factors causing the experimental response to be overly underdamped. One factor is that the plant motion is not very smooth. Because the proportional gain is high for this controller, the servo will move the beam drastically, and these changes in position are not smooth. As a result, it is very difficult for the ball to come to rest at the steady state values of twenty and thirty centimeters because the movement is not smooth. This issue is compounded by the fact that the

infrared sensor is not entirely accurate. For example, from experience, the ball could be stationary, but the infrared sensor may not be reading a constant distance because of noise. So, even if the ball reached its steady state value, the system at times would believe that there was still some error present in the system. This error would cause the high proportional gain and jerkiness of the system to significantly move the ball off the steady state position, and the controller would have to move the ball back. This produced more oscillations than desired in the response.

For the overdamped case in Figure 10, the discrepancy between the theoretical and the experimental model is different from the previous two cases. The theoretical response slightly overshoots but asymptotically approaches the steady state values. The experimental response overshoots the desired position similar to the theoretical response, but it does not return back to the steady state positions. This happens because the controller is trying to slowly return the ball to the final position. This requires the controller to send a very small output to the servo motor. In this case, the controller would send about one degree of change to the servo motor. However, the servo would not change its position at all because it could not overcome the internal friction of the servo. Thus, the controller was never able to get the ball to return to the desired position.

Conclusion:

In conclusion we saw that we were able to control the system relatively well using the infrared sensor and the Arduino board. The physical improvements, from restricting the range of motion of the beam too repairing the servo motor, seemed to help with controllability of the system. We demonstrated control over system by being able to control the ball between twenty centimeters and thirty centimeters for ideal, underdamped and overdamped cases. Furthermore, we saw that our experimental results matched up with our theoretical results rather well meaning

that our theoretical model of the system was relatively accurate and our exclusion of some variables such as friction were acceptable. For future experimentation, this system could be improved upon with a better infrared sensor that has a larger area of accurate sensing. We were limited to the 20cm to 30cm region because of the accuracy of the infrared sensor. A better sensor would allow for a wider range of motion for the system. Finally, the motor and the arm linking the motor to the beam could be improved to both be more resilient and durable. The motor occasionally had difficulty raising or lowering the beam, and the arm, made of two parts though both straight, could buckle if the parts come loose.

Acknowledgements:

We would like to thank Professor Erik Cheever, Grant Smith (Smitty) and Ed Jaoudi for all their advice and aid in completing this project along with the Swarthmore College Engineering department.

Appendices:

Appendix A (Arduino Code – Main Loop):

```
//Authors: Kyle Knapp, David Nahmias, Dan Kowalyshyn

#include <Servo.h>
#include <Wire.h> //I2C Arduino Library
#include "PID_v1.h"
#include "RunningMedian.h"

#include <math.h>
#include <stdio.h>

Servo myservo; //Servo
static int delayValue=10; //amount to delay by
int currentPosition; //keeps track whenever the input changes

double Setpoint, Input, Output;  //some global variable for PD controller
PID myPID(&Input, &Output, &Setpoint, 0.5,0, 1.0  , DIRECT);  //the order of the constants is
PID

RunningMedian myMedianFilter = RunningMedian(3); //create median filter

void setup(){
  Serial.begin(9600);
  myservo.attach(9);
  Wire.begin();
  myPID.SetMode(AUTOMATIC);
  myPID.SetOutputLimits(-20,20);  //set output limits of PID controller
  myPID.SetSampleTime(30); //set the sample time
  currentPosition = 0; //initialize the current position
}


float convertToDistance2(int AI){
 //This implements our experimentally created curve fit
 float distance;
 float voltage;

 voltage = 5.0*(AI/1023.0);  //This is the voltage read in
 distance = log(voltage/4.336)*(-1/0.06439); //the distance based on voltage

 return distance;
```

```
}


int sampleInput(unsigned long currentTime, unsigned int period, int lowPos, int highPos){
  //This creates a square wave to use as an input

  int setPos;
  int periodTime;

  periodTime = currentTime % period; //time relative to place in period

  if (periodTime<period/2){ //at low position at the first half of the period
    setPos = lowPos;
  }
  else{ //at high position at the second half of the period
    setPos = highPos;
  }
  return setPos;

}




void loop(){ //continually loop

  static unsigned long currentTime;

  int desiredPos;
  static int prevPos;

  int sensorAI;
  int distance;

  currentTime=millis(); //obtain current time

  sensorAI = analogRead(0); //read infared sensor data
  distance = convertToDistance2(sensorAI); //convert it to distance

  myMedianFilter.add(distance); //insert the distance into the median filter
  Input = myMedianFilter.getMedian(); //determine median of three distances

  //oscillate position between 20 and 30 cm every ten seconds
```

```
 desiredPos = sampleInput(currentTime, 20000, 20, 30);

 if (desiredPos!=currentPosition){//notify a change in input
   Serial.println("changing positions");
   currentPosition = desiredPos; //set the current position
 }

 Setpoint = desiredPos; //make the setpoint the desired input

 myPID.Compute();  //calculates what output to apply to the plant

 //Determine where the servo needs to go.
 //we add 59 because we make that the zero angle
 //this may need changing depending on.  Output is negative
 //because the servo reacts such that positive is clockwise instead
 //counter clockwise
 int servoAngle = -1*Output+59;

 Serial.println(distance);   //print the distance out
 myservo.write(servoAngle);  //change the servo angle


 delay(delayValue);
}
```

Appendix B (Arduino Code – PID Implementation):

```
#ifndef PID_v1_h
#define PID_v1_h
#define LIBRARY_VERSION     1.0.0

class PID
{


  public:

  //Constants used in some of the functions below
  #define AUTOMATIC 1
  #define MANUAL      0
  #define DIRECT  0
  #define REVERSE  1

  //commonly used functions
**************************************************************************
   PID(double*, double*, double*,       // * constructor.  links the PID to the Input, Output, and
      double, double, double, int);     //   Setpoint.  Initial tuning parameters are also set here

   void SetMode(int Mode);              // * sets PID to either Manual (0) or Auto (non-0)

   bool Compute();                      // * performs the PID calculation.  it should be
                                        //   called every time loop() cycles. ON/OFF and
                                        //   calculation frequency can be set using SetMode
                                        //   SetSampleTime respectively

   void SetOutputLimits(double, double); //clamps the output to a specific range. 0-255 by
default, but
                                                                        //it's likely the user
will want to change this depending on
                                                                        //the application



  //available but not commonly used functions
*****************************************************
   void SetTunings(double, double,      // * While most users will set the tunings once in the
                   double);             //   constructor, this function gives the user the option
                                        //   of changing tunings during runtime for Adaptive control
```

```
        void SetControllerDirection(int);        // * Sets the Direction, or "Action" of the
controller. DIRECT
                                                                    //   means the
output will increase when error is positive. REVERSE
                                                                    //   means the
opposite.  it's very unlikely that this will be needed
                                                                    //   once it is set in
the constructor.
   void SetSampleTime(int);          // * sets the frequency, in Milliseconds, with which
                                //   the PID calculation is performed.  default is 100



  //Display functions
*********************************************************************
        double GetKp();                                             // These functions query
the pid for interal values.
        double GetKi();                                             //  they were created
mainly for the pid front-end,
        double GetKd();                                             // where it's important to
know what is actually
        int GetMode();                                              //  inside the PID.
        int GetDirection();                                  //

  private:
        void Initialize();

        double dispKp;                                  // * we'll hold on to the tuning parameters
in user-entered
        double dispKi;                             //   format for display purposes
        double dispKd;                                  //

        double kp;              // * (P)roportional Tuning Parameter
   double ki;              // * (I)ntegral Tuning Parameter
   double kd;              // * (D)erivative Tuning Parameter

        int controllerDirection;

   double *myInput;           // * Pointers to the Input, Output, and Setpoint variables
   double *myOutput;           //   This creates a hard link between the variables and the
   double *mySetpoint;         //   PID, freeing the user from having to constantly tell us
                          //   what these values are.  with pointers we'll just know.

        unsigned long lastTime;
```

```
        double ITerm, lastInput, lastOutput, lastError;

        unsigned long SampleTime;
        double outMin, outMax;
        bool inAuto;
};
#endif


/**********************************************************************
*****************
 * Arduino PID Library - Version 1.0.1
 * by Brett Beauregard <br3ttb@gmail.com> brettbeauregard.com
 *
 * This Library is licensed under a GPLv3 License

**********************************************************************
***************/

#if ARDUINO >= 100
 #include "Arduino.h"
#else
 #include "WProgram.h"
#endif

#include "PID_v1.h"

/*Constructor (...)*********************************************************
 *    The parameters specified here are those for for which we can't set up
 *    reliable defaults, so we need to have the user set them.
 *********************************************************************/
PID::PID(double* Input, double* Output, double* Setpoint,
      double Kp, double Ki, double Kd, int ControllerDirection)
{

   myOutput = Output;
   myInput = Input;
   mySetpoint = Setpoint;
        inAuto = false;

        PID::SetOutputLimits(0, 255);                    //default output limit corresponds
to
                                                                                //the
arduino pwm limits
```

```
  SampleTime = 100;                                                    //default Controller Sample
Time is 0.1 seconds

  PID::SetControllerDirection(ControllerDirection);
  PID::SetTunings(Kp, Ki, Kd);

  lastTime = millis()-SampleTime;
}


/* Compute()
**********************************************************************
 *    This, as they say, is where the magic happens.  this function should be called
 *   every time "void loop()" executes.  the function will decide for itself whether a new
 *   pid Output needs to be computed.  returns true when the output is computed,
 *   false when nothing has been done.

**********************************************************************************
****/
bool PID::Compute()
{
  if(!inAuto) return false;

  float T = (float)SampleTime*0.001;
  float w = 3.1416/(2.0*T);

  unsigned long now = millis();
  unsigned long timeChange = (now - lastTime);
  if(timeChange>=SampleTime)
  {
    /*Compute all the working error variables*/
          double input = *myInput;
    double error = *mySetpoint - input;
    ITerm+= (ki * error);
    if(ITerm > outMax) ITerm= outMax;
    else if(ITerm < outMin) ITerm= outMin;
    double dInput = (input - lastInput)/SampleTime;

    /*Compute PID Output*/
    double output = kp * error + ITerm- kd * dInput;
          if(output > outMax) output = outMax;
    else if(output < outMin) output = outMin;
          *myOutput = output;
```

```
    /*Remember some variables for next time*/
    lastInput = input;
    lastOutput = output;
    lastError = error;
    lastTime = now;
          return true;
  }
  else return false;
}


/* SetTunings(...)*************************************************************
 * This function allows the controller's dynamic performance to be adjusted.
 * it's called automatically from the constructor, but tunings can also
 * be adjusted on the fly during normal operation

******************************************************************************
/
void PID::SetTunings(double Kp, double Ki, double Kd)
{
  if (Kp<0 || Ki<0 || Kd<0) return;

  dispKp = Kp; dispKi = Ki; dispKd = Kd;

  double SampleTimeInSec = ((double)SampleTime)/1000;
  kp = Kp;
  ki = Ki * SampleTimeInSec;
  kd = Kd / SampleTimeInSec;

 if(controllerDirection ==REVERSE)
  {
    kp = (0 - kp);
    ki = (0 - ki);
    kd = (0 - kd);
  }
}

/* SetSampleTime(...) *********************************************************
 * sets the period, in Milliseconds, at which the calculation is performed

******************************************************************************
/
void PID::SetSampleTime(int NewSampleTime)
```

```
{
  if (NewSampleTime > 0)
  {
    double ratio  = (double)NewSampleTime
                / (double)SampleTime;
    ki *= ratio;
    kd /= ratio;
    SampleTime = (unsigned long)NewSampleTime;
  }
}

/* SetOutputLimits(...)****************************************************
 *    This function will be used far more often than SetInputLimits.  while
 *  the input to the controller will generally be in the 0-1023 range (which is
 *  the default already,)  the output will be a little different.  maybe they'll
 *  be doing a time window and will need 0-8000 or something.  or maybe they'll
 *  want to clamp it from 0-125.  who knows.  at any rate, that can all be done
 *  here.
 **************************************************************************/
void PID::SetOutputLimits(double Min, double Max)
{
  if(Min >= Max) return;
  outMin = Min;
  outMax = Max;

  if(inAuto)
  {
          if(*myOutput > outMax) *myOutput = outMax;
          else if(*myOutput < outMin) *myOutput = outMin;

          if(ITerm > outMax) ITerm= outMax;
          else if(ITerm < outMin) ITerm= outMin;
  }
}

/* SetMode(...)****************************************************************
 * Allows the controller Mode to be set to manual (0) or Automatic (non-zero)
 * when the transition from manual to auto occurs, the controller is
 * automatically initialized

******************************************************************************
/
void PID::SetMode(int Mode)
{
```

```
   bool newAuto = (Mode == AUTOMATIC);
   if(newAuto == !inAuto)
   {  /*we just went from manual to auto*/
      PID::Initialize();
   }
   inAuto = newAuto;
}

/* Initialize()****************************************************************
 *       does all the things that need to happen to ensure a bumpless transfer
 *  from manual to automatic mode.

******************************************************************************
/
void PID::Initialize()
{
  ITerm = *myOutput;
  lastOutput = *myOutput;
  lastInput = *myInput;
  lastError = 0.0;
  if(ITerm > outMax) ITerm = outMax;
  else if(ITerm < outMin) ITerm = outMin;
}

/* SetControllerDirection(...)*************************************************
 * The PID will either be connected to a DIRECT acting process (+Output leads
 * to +Input) or a REVERSE acting process(+Output leads to -Input.)  we need to
 * know which one, because otherwise we may increase the output when we should
 * be decreasing.  This is called from the constructor.

******************************************************************************
/
void PID::SetControllerDirection(int Direction)
{
  if(inAuto && Direction !=controllerDirection)
  {
     kp = (0 - kp);
     ki = (0 - ki);
     kd = (0 - kd);
  }
  controllerDirection = Direction;
}

/* Status Funcions*************************************************************
```

```
 * Just because you set the Kp=-1 doesn't mean it actually happened.  these
 * functions query the internal state of the PID.  they're here for display
 * purposes.  this are the functions the PID Front-end uses for example

******************************************************************************
/
double PID::GetKp(){ return  dispKp; }
double PID::GetKi(){ return  dispKi;}
double PID::GetKd(){ return  dispKd;}
int PID::GetMode(){ return  inAuto ? AUTOMATIC : MANUAL;}
int PID::GetDirection(){ return controllerDirection;}
```

Appendix C (Arduino Code – Implementation of Median Filter):

```
#ifndef RunningMedian_h
#define RunningMedian_h
//
//    FILE: RunningMedian.h
//  AUTHOR: Rob dot Tillaart at gmail dot com
// PURPOSE: RunningMedian library for Arduino
// VERSION: 0.1.02
//     URL: http://playground.arduino.cc/Main/RunningMedian
// HISTORY: See RunningMedian.cpp
//
// Released to the public domain
//

#if defined(ARDUINO) && ARDUINO >= 100
#include "Arduino.h"
#else
#include "WProgram.h"
#endif

#include <inttypes.h>

#define RUNNINGMEDIANVERSION "0.1.02"
// should at least be 5 to be practical
#define MEDIAN_MIN 1
#define MEDIAN_MAX 19

class RunningMedian
{
      public:
      RunningMedian(uint8_t);
      RunningMedian();
      void clear();
      void add(float);
      float getMedian();

      float getAverage();
      float getHighest();
      float getLowest();

      protected:
      uint8_t _size;
      uint8_t _cnt;
```

```
	uint8_t _idx;
	float _ar[MEDIAN_MAX];
	float _as[MEDIAN_MAX];
	void sort();
};

#endif
// END OF FILE


//
//    FILE: RunningMedian.cpp
//  AUTHOR: Rob dot Tillaart at gmail dot com
// VERSION: 0.1.02
// PURPOSE: RunningMedian library for Arduino
//
// HISTORY:
// 0.1.00 - 2011-02-16 initial version
// 0.1.01 - 2011-02-22 added remarks from CodingBadly
// 0.1.02 - 2012-03-15
//
// Released to the public domain
//

#include "RunningMedian.h"

RunningMedian::RunningMedian(uint8_t size)
{
	_size = constrain(size, MEDIAN_MIN, MEDIAN_MAX);
	clear();
}

RunningMedian::RunningMedian()
{
	_size = 5; // default size
	clear();
}

// resets all counters
void RunningMedian::clear()
{
	_cnt = 0;
	_idx = 0;
}
```

```
// adds a new value to the data-set
// or overwrites the oldest if full.
void RunningMedian::add(float value)
{
	_ar[_idx++] = value;
	if (_idx >= _size) _idx = 0; // wrap around
	if (_cnt < _size) _cnt++;
}

float RunningMedian::getMedian()
{
	if (_cnt > 0)
	{
		sort();
		return _as[_cnt/2];
	}
	return NAN;
}


float RunningMedian::getHighest()
{
	if (_cnt > 0)
	{
		sort();
		return _as[_cnt-1];
	}
	return NAN;
}

float RunningMedian::getLowest()
{
	if (_cnt > 0)
	{	sort();
		return _as[0];
	}
	return NAN;
}

float RunningMedian::getAverage()
{
	if (_cnt > 0)
	{
```

```
            float sum = 0;
            for (uint8_t i=0; i< _cnt; i++) sum += _ar[i];
            return sum / _cnt;

      }
      return NAN;
}

void RunningMedian::sort()
{
      // copy
      for (uint8_t i=0; i< _cnt; i++) _as[i] = _ar[i];

      // sort all
      for (uint8_t i=0; i< _cnt-1; i++)
      {
            uint8_t m = i;
            for (uint8_t j=i+1; j< _cnt; j++)
            {
                  if (_as[j] < _as[m]) m = j;
            }
            if (m != i)
            {
                  long t = _as[m];
                  _as[m] = _as[i];
                  _as[i] = t;
            }
      }
}

// END OF FILE
```

Appendix D (MATLAB Code – For plotting and data analysis):

```
%% Control Theory Final Project-Ball Balancing Beam
%% Kyle Knapp, David Nahmias & Daniel Kowalyshyn
%% Initialize
%%
clear all; close all
%% Load Data
%%
load('IdealContollerData.mat')
load('OverdampedContollerData.mat')
load('UnderdampedContollerData.mat')
```

```
time = linspace(0,20,length(distance3));
%% Ideal Case
%%
input = 20.*(time==0)+30.*((time~=0) & (time<=10)) + 20.*(time>10);
Gp = tf(0.617,[1 0 0]);
Gc = tf([1.4 0.6],1);
sys = feedback(Gp*Gc, 1);

figure
plot(time, input, 'g')
hold on
plot(time, distance,'r')

input2 = 10.*(time<=10);
y = lsim(sys,input2,time);
plot(time,y+20)

xlabel('Time (s)')
ylabel('Distance (cm)')
ylim([10,40])
title('Ideal Input & Response')
legend('Input','Experimental Response','Theoretical Response')
hold off

%% Underdamped Case
%%
input = 20.*(time==0)+30.*((time~=0) & (time<=10)) + 20.*(time>10);
Gp = tf(0.617,[1 0 0]);
Gc = tf([1.3 0.8],1);
sys = feedback(Gp*Gc, 1);

figure
plot(time, input, 'g')
hold on
plot(time, distance2,'r')

input2 = 10.*(time<=10);
y = lsim(sys,input2,time);
plot(time,y+20)

xlabel('Time (s)')
ylabel('Distance (cm)')
ylim([10,40])
title('Underdamped Input & Response')
legend('Input','Experimental Response','Theoretical Response')
hold off

%% Overdamped Case
%%
input = 20.*(time==0)+30.*((time~=0) & (time<=10)) + 20.*(time>10);
Gp = tf(0.617,[1 0 0]);
Gc = tf([2.5 0.3],1);
sys = feedback(Gp*Gc, 1);
```

```
figure
plot(time, input, 'g')
hold on
plot(time, distance3,'r')

input2 = 10.*(time<=10);
y = lsim(sys,input2,time);
plot(time,y+20)

xlabel('Time (s)')
ylabel('Distance (cm)')
ylim([10,40])
title('Overdamped Input & Response')
legend('Input','Experimental Response','Theoretical Response')
hold off
```