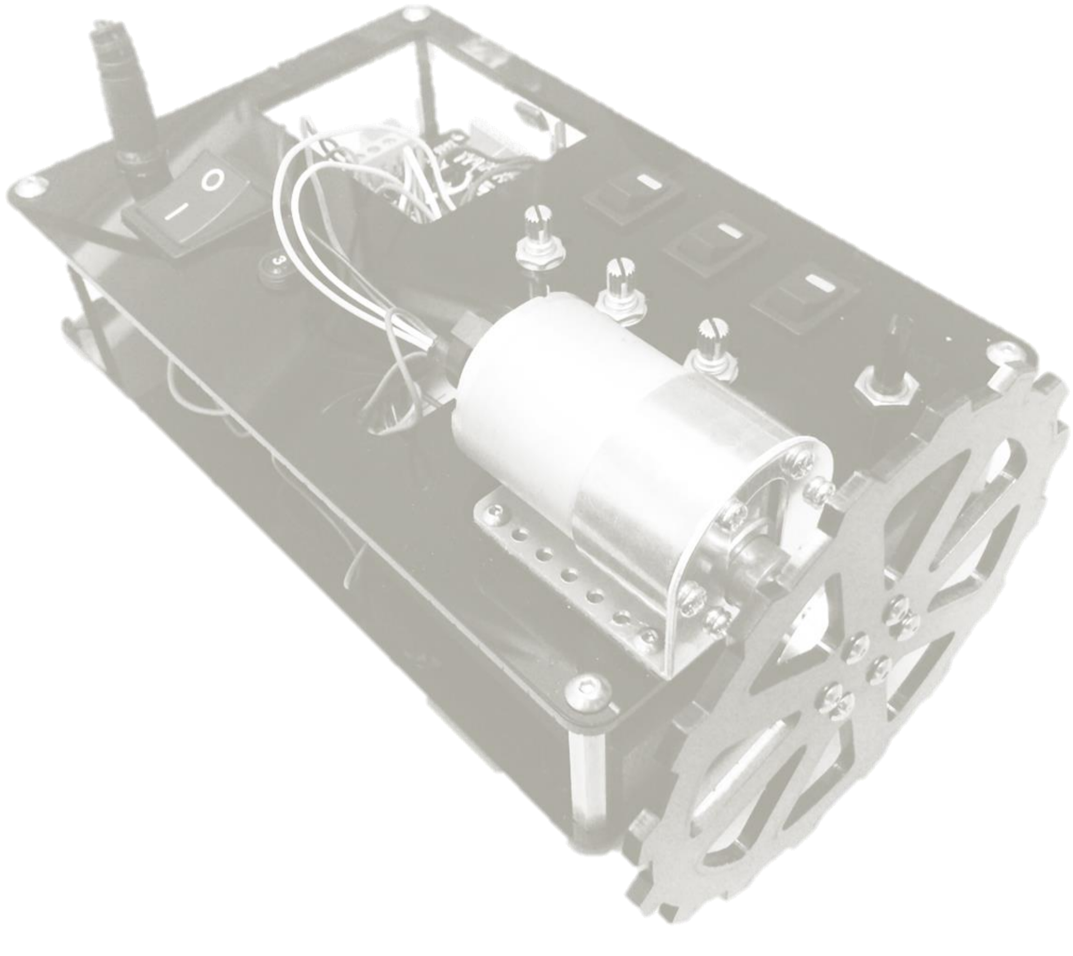
PID Demonstration Platform

Joshua Gafford

03-01-2013



Objective

For the purpose of demonstrating high-level principles of proportional-integral-derivative (PID) control to a multidisciplinary medical device design class (ES227: Medical Device Design at Harvard University), a demonstration box was built allowing a user to vary PID gains in real-time and observe the resulting dynamic response on a plant (a DC motor with a laser-cut acrylic inertial mass and hall-effect quadrature encoding for position feedback). This document summarizes the basic electromechanical design of the hardware, control theory and digital implementation of the controller, as well as some example step responses for various PID gain combinations.

1 Theory and Implementation of PID Control

In continuous time, the basic form of the PID controller is given by the following:

$$u(t) = K_p e(t) + K_d \frac{d}{dt} e(t) + K_i \int_{t_0}^{t} e(t)dt \tag{1}$$

where $u(t)$ is the controller effort, $e(t)$ is the error between setpoint and output, and K_p, K_d and K_i are the proportional, integral, and derivative gains (respectively). Converting this controller into the Laplace domain, we can compute the transfer function $G(s)$ of the controller :

$$\frac{U(s)}{E(s)} = K_p + K_d s + \frac{K_i}{s} = K_p \left(\frac{T_i T_d s^2 + T_i s + 1}{T_i s} \right) = G(s) \tag{2}$$

$$T_i = \frac{K_p}{K_i} \tag{3}$$

$$T_d = \frac{K_d}{K_p} \tag{4}$$

where we have introduced T_i and T_d which are integral and derivative time constants, respectively. When implementing such a controller into a digital microcontroller, it is necessary to discretize the system. A straightforward way of doing this is to use a backwards finite difference approximation for the derivative term and a Riemann integral approximation for the integral term:

$$u_k = K_p e_k + \frac{K_d}{T_s}(e_k - e_{k-1}) + K_i T_s \sum_{k=0}^{k} e_k \tag{5}$$

A more exact approach, especially given a variable sample time, is to compute the zero-order hold (ZOH) approximation of the continuous controller transfer function, which exactly captures the sample-and-hold behavior of A/D converters. The ZOH approximation for a continuous transfer function $G(s)$ is given by the following:

$$G(z) = \frac{z-1}{z} Z\left\{ \frac{G(s)}{s} \right\} \tag{6}$$

Leaving out much of the derivation for brevity, the ZOH PID controller is given by:

$$G(z) = \frac{U(z)}{E(z)} = K_p + K_i \frac{T_s(z+1)}{2(z-1)} + K_d \frac{z-1}{z T_s} \tag{7}$$

where T_s is the loop sample rate. Digitizing this controller will give us a form that can be implemented on-board our microcontroller:

$$u_k = u_{k-1} + ae_k + be_{k-1} + ce_{k-2} \tag{8}$$

$$a = K_p + K_i \frac{T_s}{2} + \frac{K_d}{T_s} \tag{9}$$

$$b = -K_p + K_i \frac{T_s}{2} - \frac{2K_d}{T_s} \tag{10}$$

$$c = \frac{K_d}{T_s} \tag{11}$$

Given that the time constants associated with the dynamic behavior of the motor and inertial mass are significantly longer than the sample rate of the system, it really doesn't matter which form of the PID we implement (finite difference, Riemann sum vs. ZOH). But just to be pedantic, we will implement the ZOH form given in Equation 6 as it is generally more robust for variable sample rates.

2 Hardware

An image of the test setup is shown in Figure 1. An Arduino microcontroller handles all of the analog and digital communication (inputs from switches, potentiometers and encoders), as well as PWM generation for the motor. The motor is driven by a separate 12-Volt power supply via 2A FET-based motor shield with flyback and shoot-through protection, PWM input and direction select via a digital signal. Power is taken from the grid via a 120VAC-12VDC transformer/rectifier and regulated via 12VDC-5VDC voltage regulator IC, and a 3A circuit breaker protects the system from large currents (as the motor can draw up to 5A at stall). A complete list of the bill of materials is given below:

- **(1x)** Arduino UNO microcontroller
- **(1x)** 2A FET-based Arduino Motor Shield
- **(1x)** Polulu 12V DC Motor with 26:1 Gearbox and Hall-Effect Quadrature Encoder
- **(4x)** Potentiometers (3 for PID gains, 1 for position setpoint)
- **(3x)** Three LED-Illuminated toggle switches for PID on/off
- **(1x)** One master toggle switch for power
- **(1x)** 12VDC power transformer for motor power
- **(1x)** 5VDC voltage regulator for Arduino/Switch/Potentiometer/Hall-Effect power
- **(1x)** 3A circuit breaker
- **(3x)** Laser-cut mounting hardware
- **(4x)** Two-inch hex standoffs with 8-32 threads on both ends
- **(8x)** 8-32, ¾" length SHCS
- **(1x)** Machined steel block to act as a weight/potential heatsink for regulator
- **(2x)** 1 μF caps for hall effect encoder debouncing
- **(2x)** LED's for power status (1 for 12VDC, 1 for 5VDC)
- **(2x)** Current-Limiting Resistors (1 for each LED, 200Ω for 5VDC and 470Ω for 12VDC)
- Various jumper wires

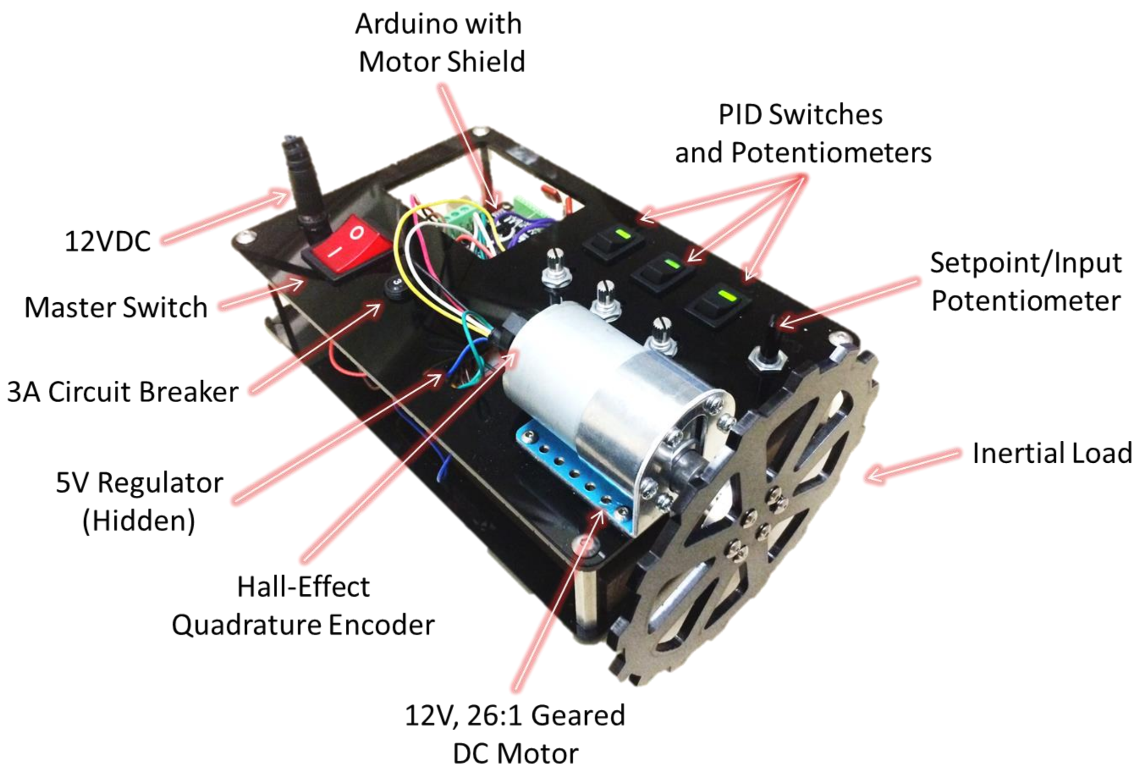

Arduino with Motor Shield
PID Switches and Potentiometers
12VDC
Setpoint/Input Potentiometer
Master Switch
3A Circuit Breaker
Inertial Load
5V Regulator (Hidden)
Hall-Effect Quadrature Encoder
12V, 26:1 Geared DC Motor

Figure 1. Image of demonstration box hardware

A wiring diagram of the system is shown in Figure 2. Encoder inputs are debounced via 1μF capacitors. Status LEDs are current limited by 200Ω resistors. Note that the motor PWM and Enable are internally wired in the motor shield itself so these are not shown.

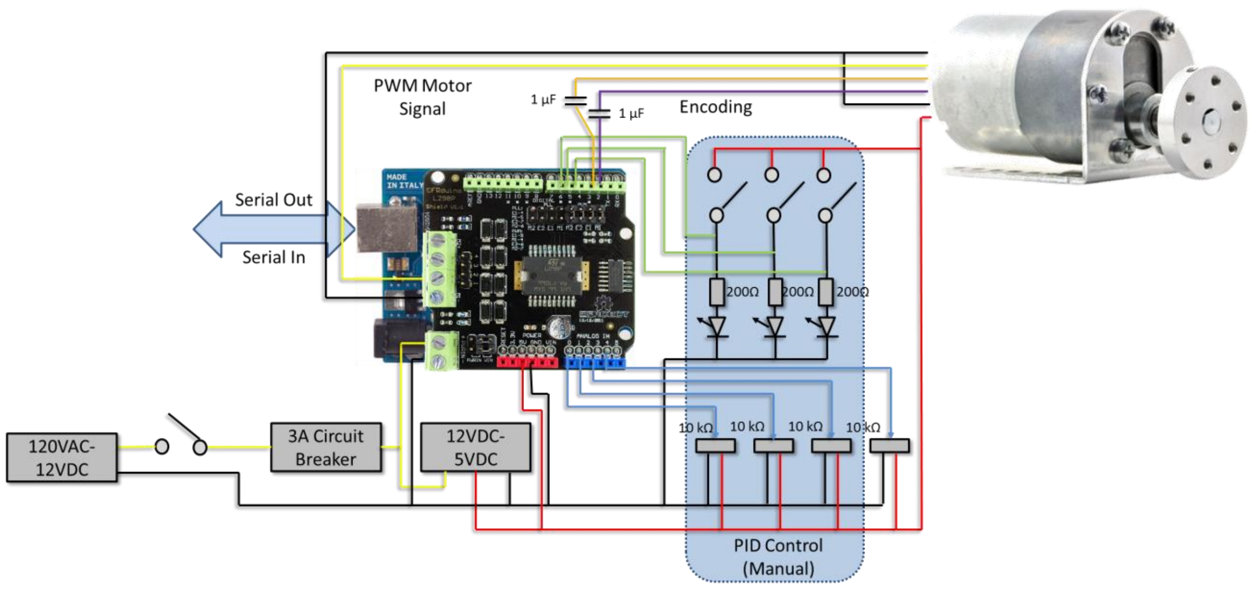


Figure 2. Wiring diagram of the entire system

3 Software Implementation Considerations

3.1 *Motor Encoding*

Arduino's built in `digitalRead()` function has considerable computational overhead which can slow down processing and cause encoder counts to be missed, even when implemented in the form of a program interrupt. As such, an interrupt-based encoding function that utilizes the `digitalReadFast()` library was implemented, allowing the encoders to be read roughly 50 times faster than those based on `digitalRead()`.

3.2 *Sampling Time*

Arduino's sampling capabilities are notoriously volatile and unreliable. Using the `delay()` function is a common (hacky) way of 'forcing' a semi-reliable sampling rate, but this has the drawback of halting code execution for the specified delay interval which is bad for any system that requires continuous sensory input, such as the encoding. In addition, the ZOH approximation requires a relatively constant sample time T_s to be accurate. As such, the basic PID code has been modified to constrain the controller to correct only at user-specified sampling intervals.

3.3 *Continuously-Tunable Controller Implementation*

The digital controller was built in Arduino's text-based interactive design environment (IDE). A high-level block diagram of the controller is shown below. Encoder parsing takes place using an interrupt routine to inform the controller of the current motor position. PID gains are selected/adjusted in real-time via toggle switches and potentiometers. Motor direction is determined by the sign of the PID correction term. Saturation limiting ensures the PWM output is within 8-bit levels (0-255).

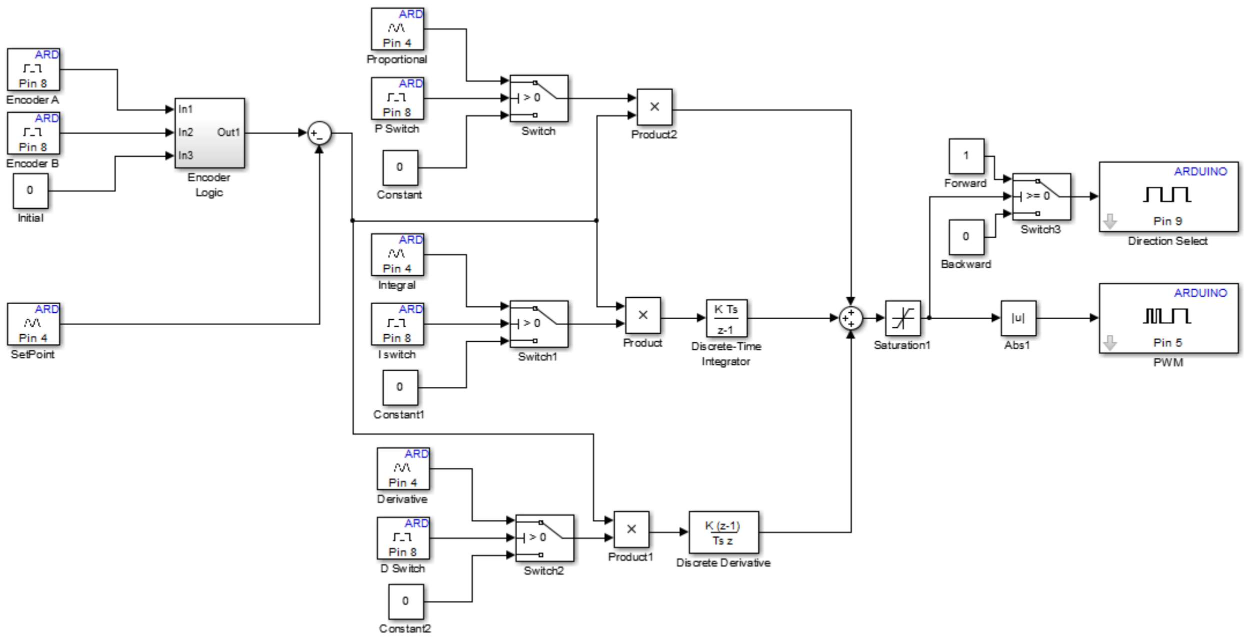


Figure 3. Block diagram illustration of embedded controller with continuously-tunable PID gains. Note that this is purely representational.

4 Example Results

4.1 Proportional Control

A characteristic curve for a (somewhat aggressive) proportional controller in response to step inputs is shown in Figure 4. Observe significant overshoot as the controller essentially acts as a linear spring. In addition, the lack of an integral term results in a non-negligible steady-state error, as the control signal generated by the proportional term isn't enough to overcome the motor dead-band (caused by friction within the system).

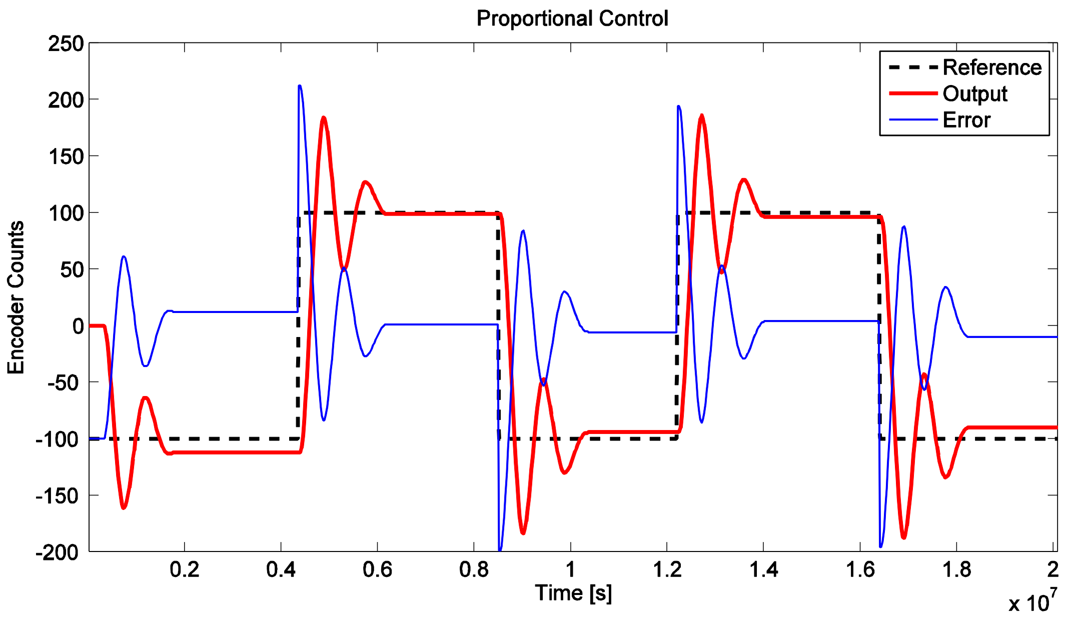


Figure 4. Response of a proportional controller to step inputs

4.2 Proportional-Integral-Derivative Control

The PID implementation, shown in Figure 5, improves upon the proportional controller in several ways. Of note, the derivative term adds damping to the system which speeds up convergence and limits overshoot. Additionally, the integrator ensures that steady-state error goes to zero (we can see the 'dithering' effect about steady state as the integrator attempts to overcome backlash within the system).

The role of each error term is seen more clearly in Figure 6. Note that the step behavior is shown for reference but does not share the same units or scaling as the error terms. The proportional term shown by the red line is a linear function of the error scaled by a gain K_p. The derivative term, given by the blue line, is a linear function of the change-in-error scaled by a gain K_d. The integrator term shown by the green line integrates the error (which we can see in steady-state) proportional to a gain K_i. It is wind-up limited to +/-40.

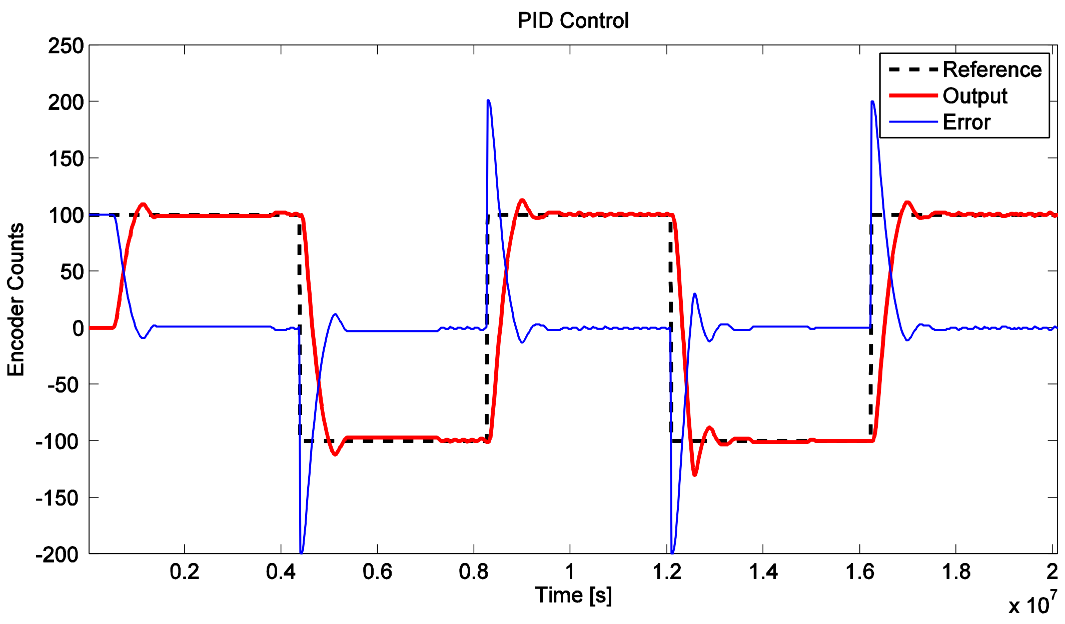


Figure 5. Response of a PID controller to step inputs.

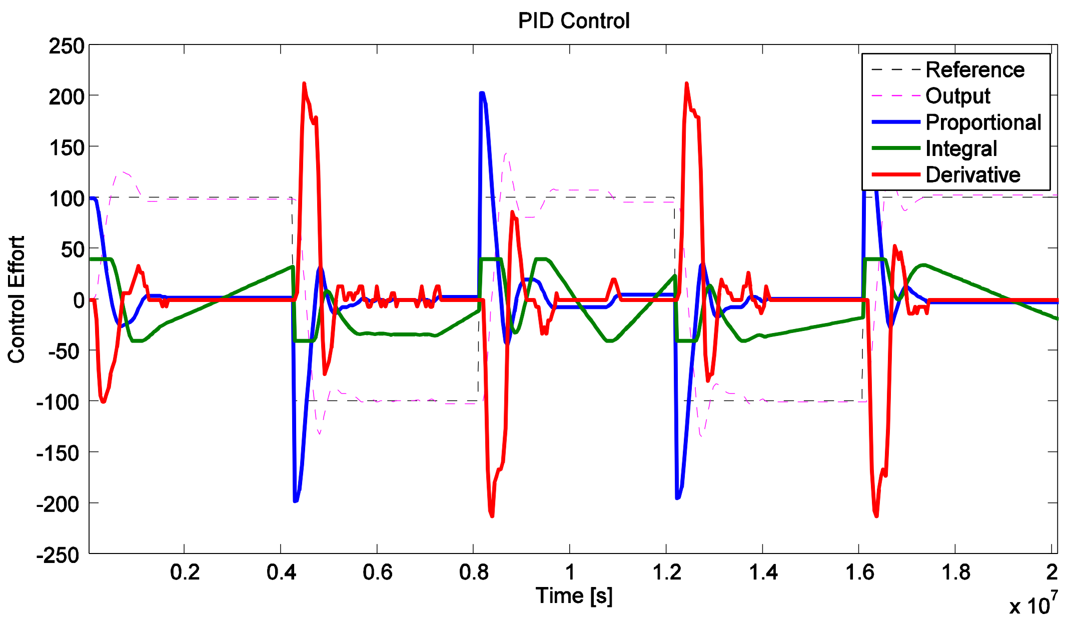


Figure 6. PID error terms

5 Arduino Code

Note: This code requires the `digitalWriteFast()` library which can be downloaded from the internet.

```
/*PID Demonstration Box Demo*/
//
//Written by: Joshua Gafford
```

```
//Date: 02/28/2014
//

///include all relevant libraries
#include "Arduino.h"
#include <digitalWriteFast.h>

/*define pin numbers*/

//Position and PID gain potentiometers
int pospot = A1;
int ppot = A2;
int ipot = A3;
int dpot = A4;

//PID gain switches
int psw = 6;
int dsw = 5;
int isw = 4;

//Motor enable and PWM
int E1 = 10;
int M1 = 12;

//Encoder Initialization
// Quadrature encoders
// Left encoder
#define c_EncoderInterruptA 0
#define c_EncoderInterruptB 1
#define c_EncoderPinA 2
#define c_EncoderPinB 3
#define EncoderIsReversed

// quadrature volatile variables
volatile bool _EncoderASet;
volatile bool _EncoderBSet;
volatile bool _EncoderAPrev;
volatile bool _EncoderBPrev;
volatile long _EncoderTicks = 0;

//Potentiometer values
int posval = 0;
int pval = 0;
int ival = 0;
int dval = 0;

//PID gain switch logic values
int ptog=0;
int itog=0;
int dtog=0;

/*working variables*/
unsigned long lastTime = 0;
unsigned long currTime = 0;
int Input, Output, OutputPrint, Setpoint;
double ITerm, lastInput;
double a, b, c;
```

```
float kp, ki, kd;
double lastU = 0;
double lastE = 0;
double lastlastE = 0;
int SampleTime = 25; //1 sec
double outMin = 0;      //output limiting
double outMax = 255;    //output limiting
double intMin = -40;    //integrator saturation
double intMax = 40;     //integrator saturation

void setup(){

  Serial.begin (9600);
  pinMode(M1,OUTPUT);

  // Quadrature encoders
  // Left encoder
  pinMode(c_EncoderPinA, INPUT);      // sets pin A as input
  digitalWrite(c_EncoderPinA, LOW);  // turn on pullup resistors
  pinMode(c_EncoderPinB, INPUT);      // sets pin B as input
  digitalWrite(c_EncoderPinB, LOW);  // turn on pullup resistors
  attachInterrupt(c_EncoderInterruptA, HandleMotorInterruptA, CHANGE);
  attachInterrupt(c_EncoderInterruptB, HandleMotorInterruptB, CHANGE);
}
void loop(){
  /*Compute PWM based on current position*/
  double Output=Compute(_EncoderTicks/2);
  analogWrite(E1,Output);
}

/*PID output compute routing*/
int Compute(long rotaryCount)
{
  unsigned long now = millis();
  int timeChange = (now - lastTime);
  int currTime=lastTime+timeChange;
  if(timeChange>=SampleTime){
    /*Compute all the working error variables*/
    Setpoint=(1024-analogRead(pospot))-1024/2;
    Input=rotaryCount;

    /*Uncomment for Step Profile based on position potentiometer
    */
    if (Setpoint>0){
      Setpoint=400;
    }
    else{
      Setpoint=-400;
    }


    /*Get PID gains based on potentiometer/switch*/
    kp=GetKP();
    ki=GetKI();
    kd=GetKD();

    /*Compute coefficients for ZOH controller*/
    /*
    a = kp+ki*(timeChange/2)+kd/timeChange;
    b = -kp+ki*(timeChange/2)-2*kd/timeChange;
    c = kd/timeChange;*/

    double error = Setpoint - Input;

    /*Determine motor direction based on error sign*/
    if (error<0){
      digitalWriteFast(M1,HIGH);
    }
    else{
      digitalWriteFast(M1,LOW);
    }

    /*Update integral term*/
    ITerm+= (ki * error);
    ITerm=ITerm;

    /*Saturation Control*/
    if(ITerm > intMax) ITerm= intMax;
    else if(ITerm < intMin) ITerm= -intMax;

    /*Compute derivative term*/
    double dInput = (Input - lastInput);

    /*Compute PID Output*/
    Output = kp * error + ITerm - kd * dInput;

    /*Uncomment for ZOH Implementation*/
    //Output = lastU + a*error + b*lastE + c*lastlastE;

    OutputPrint=Output;
    Output=abs(Output);

    /*Limit Output to 8-bit*/
    if(Output > outMax) Output = outMax;
    else if(Output < outMin) Output = outMin;

    /*Print data to serial*/
    Serial.print(currTime);Serial.print(" ");
    Serial.print(Setpoint);Serial.print(" ");
    Serial.print(Input);Serial.print(" ");
    Serial.print(error);Serial.print(" ");
    Serial.print(OutputPrint);Serial.print(" ");
    Serial.print(kp*error);Serial.print(" ");
    Serial.print(ITerm);Serial.print(" ");
    Serial.println(-kd*dInput);

    /*Remember some variables for next time*/
    lastInput = Input;
    lastTime = now;
    lastlastE = lastE;
    lastE = error;
    lastU = Output;
    return Output;
    }
```

```
}

/*Compute Proportional Gain based on switch/pot input*/
float GetKP(){
  float k_p;
  ptog=digitalRead(psw);
  if(ptog==1){
    pval=analogRead(ppot);
    k_p=(1024-float(pval))/800;    //arbitrary heuristic scaling
  }
  else{
    k_p=0;
  }
  return k_p;
}

/*Compute Integral Gain based on switch/pot input*/
double GetKI(){
  double ki;
  double SampleTimeInSec = ((double)SampleTime)/1000;
  itog=digitalRead(isw);
  if(itog==1){
    ival=analogRead(ipot);
    ki=SampleTimeInSec*(1024-ival)/100;  //arbitrary heuristic scaling
  }
  else{
    ki=0;
  }
  return ki;
}

/*Compute Derivative Gain based on switch/pot input*/
double GetKD(){
  double kd;
  double SampleTimeInSec = ((double)SampleTime)/1000;
  dtog=digitalRead(dsw);
  if(dtog==1){
    dval=analogRead(dpot);
    kd=(1024-dval)/(25000*SampleTimeInSec);   //arbitrary heuristic scaling
  }
  else{
    kd=0;
  }
  return kd;
}

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

```
void SetOutputLimits(double Min, double Max)
{
  if(Min > Max) return;
  outMin = Min;
  outMax = Max;

  if(Output > outMax) Output = outMax;
  else if(Output < outMin) Output = outMin;

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

void HandleMotorInterruptA(){
  _EncoderBSet = digitalReadFast(c_EncoderPinB);
  _EncoderASet = digitalReadFast(c_EncoderPinA);

  _EncoderTicks+=ParseEncoder();

  _EncoderAPrev = _EncoderASet;
  _EncoderBPrev = _EncoderBSet;
}

// Interrupt service routines for the right motor's quadrature encoder
void HandleMotorInterruptB(){
  // Test transition;
  _EncoderBSet = digitalReadFast(c_EncoderPinB);
  _EncoderASet = digitalReadFast(c_EncoderPinA);

  _EncoderTicks+=ParseEncoder();

  _EncoderAPrev = _EncoderASet;
  _EncoderBPrev = _EncoderBSet;
}

/*Determine position based on quadrature encoder logic*/
int ParseEncoder(){
  if(_EncoderAPrev && _EncoderBPrev){
    if(!_EncoderASet && _EncoderBSet) return 1;
    if(_EncoderASet && !_EncoderBSet) return -1;
  }else if(!_EncoderAPrev && _EncoderBPrev){
    if(!_EncoderASet && !_EncoderBSet) return 1;
    if(_EncoderASet && _EncoderBSet) return -1;
  }else if(!_EncoderAPrev && !_EncoderBPrev){
    if(_EncoderASet && !_EncoderBSet) return 1;
    if(!_EncoderASet && _EncoderBSet) return -1;
  }else if(_EncoderAPrev && !_EncoderBPrev){
    if(_EncoderASet && _EncoderBSet) return 1;
    if(!_EncoderASet && !_EncoderBSet) return -1;
  }
}
```