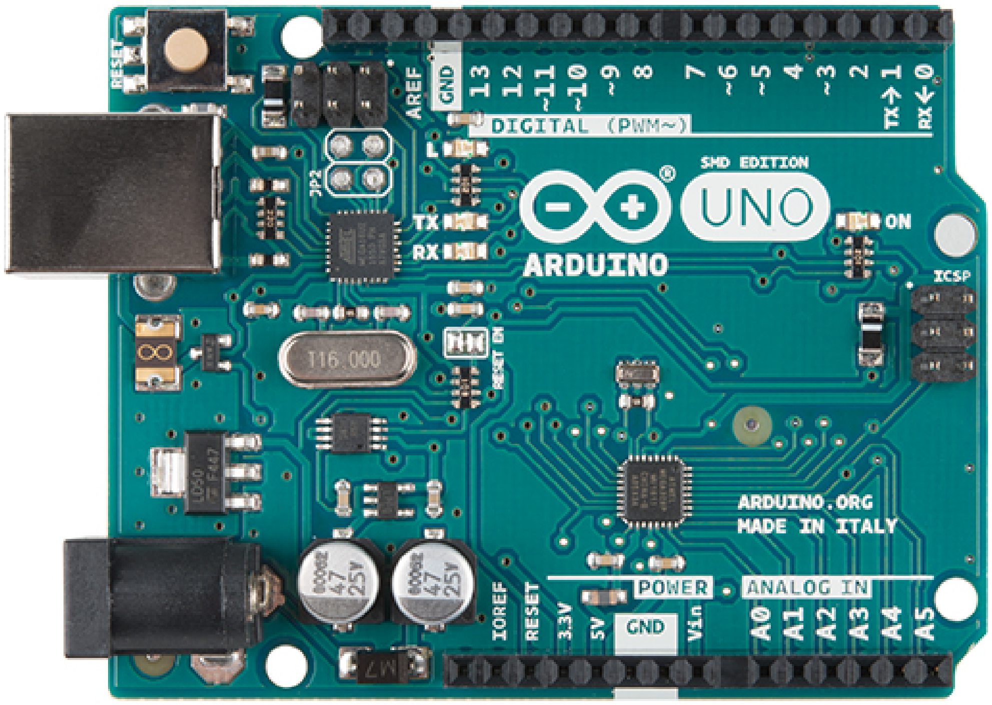
PID CONTROL WITH AN ARDUINO

Julian Stapleton and Michael Goodman | PID control | 8/1/18



Abstract:

Arduino control using PID theory was studied both experimentally and theoretically. Code was used to document the temperature readout over time, and it directly saved the readouts to the SD card. The pulse width was altered using PID to activate the heating element to a fraction of the maximum voltage and obtain a target temperature. We found that using the Zeigler-Nichols PID theory led to indefinite periods of oscillation, but the Skogestad PID theory led to temperature change with a large overshoot above the temperature. The Skogestad theory also showed behavior comparable to a critically damped system as the temperature settled with a few oscillations in half the time.

Introduction:

Proportional Integral Derivative (PID) control uses those three determined values to choose the Pulse Width Modulation (PWM) value that keeps the heater on at certain intervals. PWM is used to give dynamic adjustments to voltage in a system by turning the power source on and off rapidly. The ratio of on to off in a given period is what determines the average voltage in that period. PID uses determined values in a system to dynamically choose the conditions for PWM. The difference in temperature is the proportional term. The “Integral” term is determined through the sum of the proportional terms (or error) thus far. The “Derivative” term is determined by the average slope of the temperature. The Zeigler-Nichols theory gives values received from tuning that are different in magnitude than the Skogestad theory.

The Arduino has built in PWM controls that can pulse a light to give it a dimmed effect. The code for that is default and can be found in its own examples. However, the pulsing is too fast for relay we were using, so we opted for the Timer One library which offers PWM at a period we can control. The shortest pulse we can give the relay, and have it still respond somewhat accurately, is about 25 milliseconds. The ethernet shield allows for an SD card to be written to. With this, the Arduino can function and record autonomously. The code would just have to be edited to remove all calls to the Serial.

Materials and Methods:

For the experimental setup, an Arduino Uno with an Ethernet shield was wired to a temperature sensor, and a solid-state relay. A heating element was wired to the relay and powered by a variable transformer to give it a lower maximum voltage, as 120V was too hot for the heating element. A given pin in the Arduino code would be assigned for the heater, and another for a cooling element (a fan). The fan used was non-adjustable and has an on and off state. The fan was connected to a separate relay. The libraries needed for this experiment are the Timer One library and the PlayingWithFusion_MAX31855 library. Timer One allows for easier PWM control with fewer lines, the other library for the thermocouple breakout board (SEN-30003MAX31855 J-type 4 Channel non-isolated from playingWithFusion). The thermocouple board is used to give the temperature sensor a way to transmit numerical data to the Arduino. The Arduino needs to have a few pins free for the SD card (on the ethernet shield) and the timer one library. Pins that were kept clear: 0, 1, 4, 10, 11. Code was used to document the temperature read out over time, and directly saved it to the SD card. The pulse width was altered using PID to activate the heating element to a certain voltage and obtain a target temperature.

The over all goal was to get feedback that was both quick to rise in temperature initially, and then behaved as if it were critically damped. We found that using the Zeigler-Nichols PID (Astrom, 1995) theory led to long periods of oscillation, due to its overly aggressive K values, where the temperature rarely settled at all. The Skogestad PID theory (Skogestad, 2001) led to aggressive temperature change with a large overshoot, and showed behavior comparable to a critically damped system, as the temperature settled with a few oscillations in half the time. In order to get rid of excess heat from the initial heating period, a fan was applied to try and cool off the system. A fan that is on during certain conditions only and then turned off, forced the system to oscillate indefinitely. But a fan turned on at the target temperature, and then left on, decreases the amplitude of the oscillations in the system. Our ideal critically damped system will never come to be, because the heat capacity (a sort of heat inertia) keeps the system from settling within one oscillation. Unless we observed the data and changed the code after the fact to adjust for the overshoot.

Theoretical:

K(Proportional+(1/Ti*Integral)+(Td*Derivative))

Zeigler-Nichols PID Theory.

Skogestad thinks he’s so cool. (and its true)

Experimental:

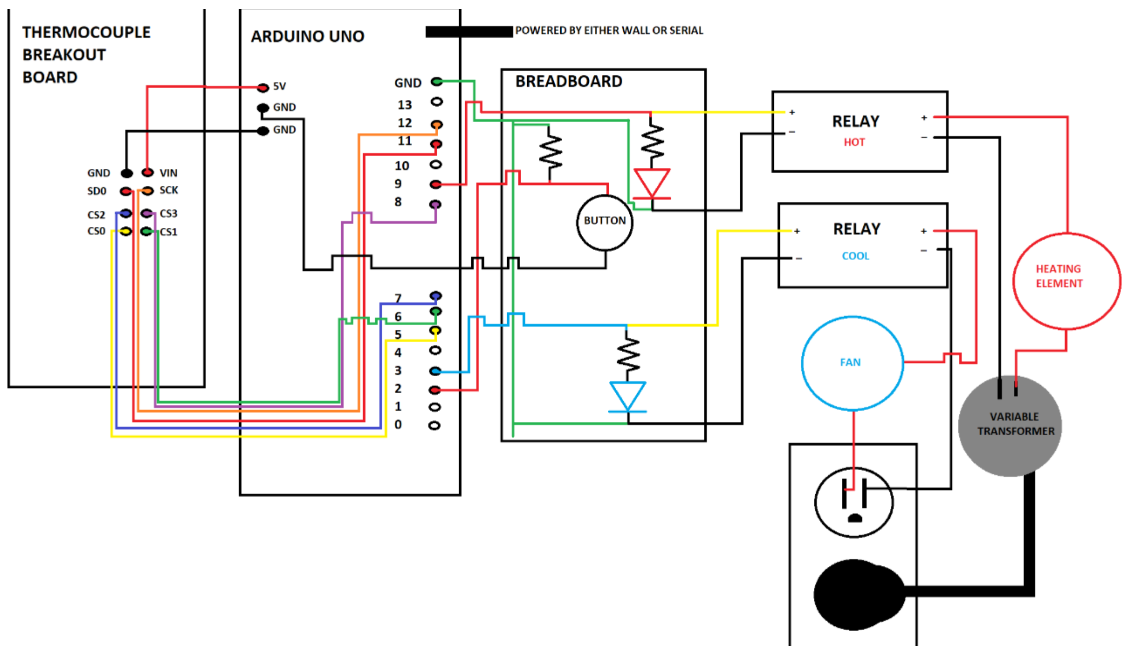


Figure 1-wiring diagram.

Fig. 1 Shows how the Arduino, heating element, cooling element, relay, variable transformer, and thermocouple board are all wired together. The variable transformer allowed for a maximum voltage of 50V and later 75V. The code (appendix 1) uses the difference in current temperature and target temperature to find the proportional value. The proportional term deals with the instantaneous error, however, using only the proportional term leads to permanent oscillations in all cases (Sontag, 1998). The integral value is the sum of the proportional values starting from time = 0. The role of the integral value is to slowly accumulate and eventually force the steady state error to zero. The derivative value is determined by taking the last 30 datapoints and finding the linear regression between them to find the slope. The first 30 seconds are mitigated by a portion of code that tells the linear regression algorithm that all time before t = 0 has the same temperature. The derivative suppresses the short time oscillations by giving extra dampening as the target temperature is approached. Running the code (appendix 1) with a serial connected will prompt the user to input variables such as maximum voltage, resistance of the heating element, and other important information. The system will not start heating until the button is pressed (physical button on the breadboard). After the button is pressed, temperature and time readouts are produced. The error will diminish and then turn negative as the temperature overshoots its target.

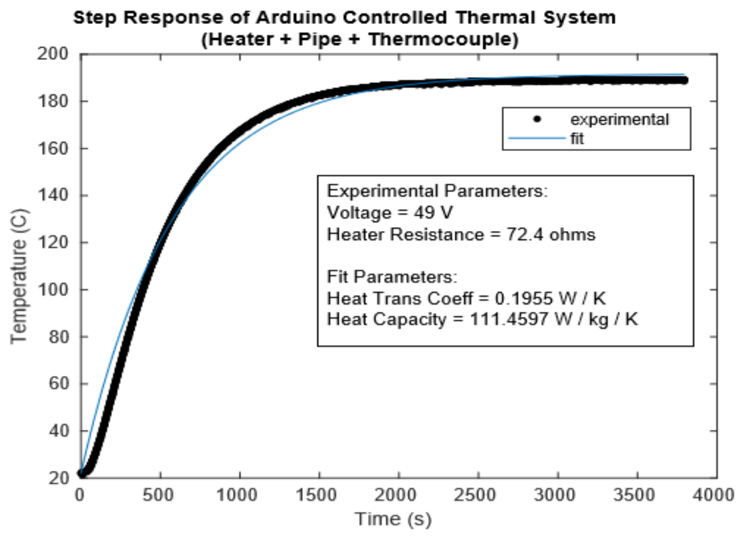


Figure 2 – Step Response curve of Arduino controlled thermal system.

Fig.2 shows the step response when the full voltage is applied for all time. When recording the step response data, tuning values can be procured using code written for MATLAB which enables efficient use of the PID. The step response data is fitted in the program, and the step response curve fit serves as an input to both PID theories to determine “K”, “Ti”, and “Td” values. These values are entered as tuning values into the Arduino code to get a PID response.

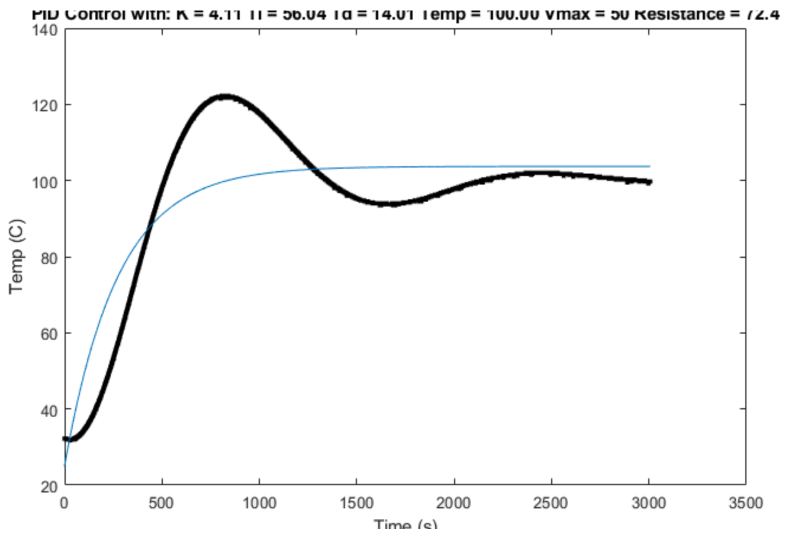


Figure 3

Using the Zeigler-Nichols PID tuning parameters the desired temperature is able to level, but it takes a long time. An error in application of the theory was made, causing the Zeigler-Nichols “K” value to be a lot smaller than it should have been (Fig. 3). Multiplying the K value by the resistance forced the system to become unstable (Fig. 4). A higher K value leads to a more extreme duty cycle, which in turn increases the amplitude of oscillations. The result of the Zeigler-Nichols tuning parameters when used properly are an over aggressive “K” value that leads to permanent oscillation.

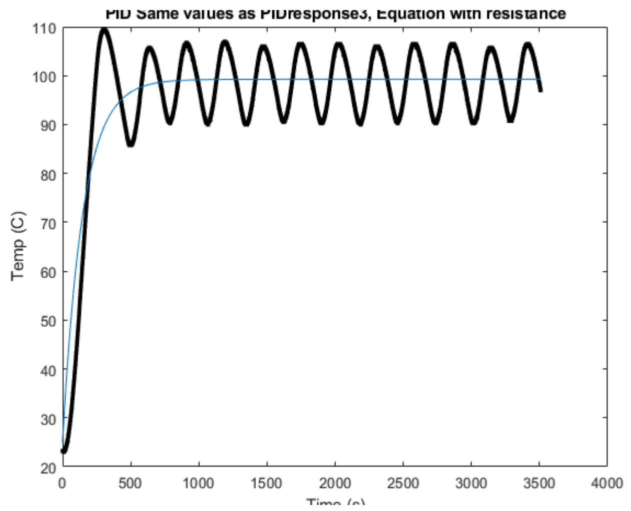


Figure 4

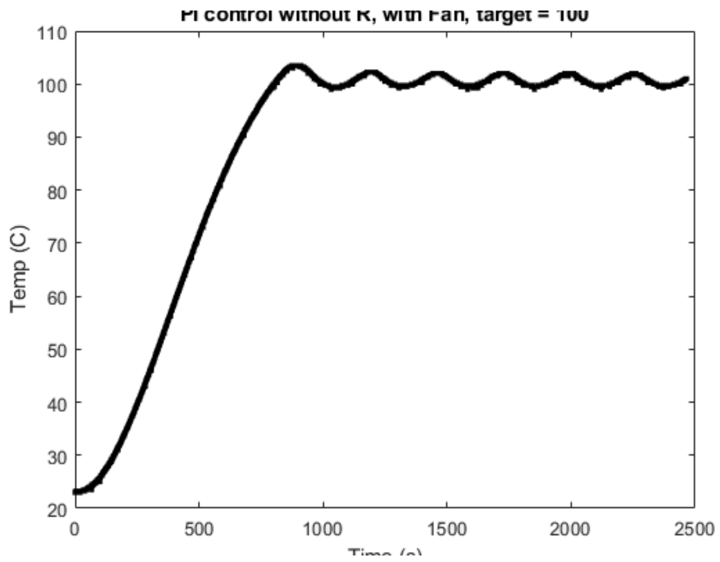


Figure 5

A fan was introduced to the system, and the results were satisfying. Its purpose was to remove excess heat from the system. The fan was triggered when the system surpassed a certain temperature. Which meant it was turning on and off, which leads to the steady oscillations in Fig. 5. To fix the error with the miscalculated tuning values we tried using just PI (proportional integral) control to get more stable results (Fig. 6.1).

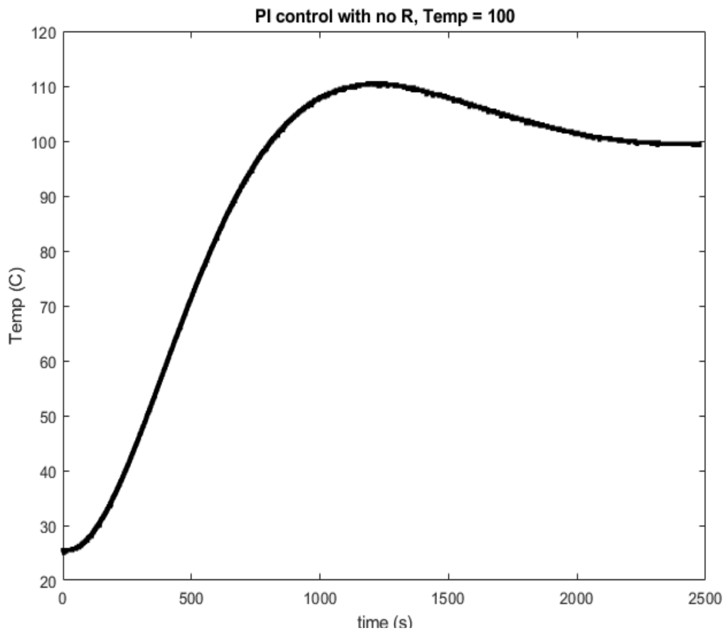


figure 6.1

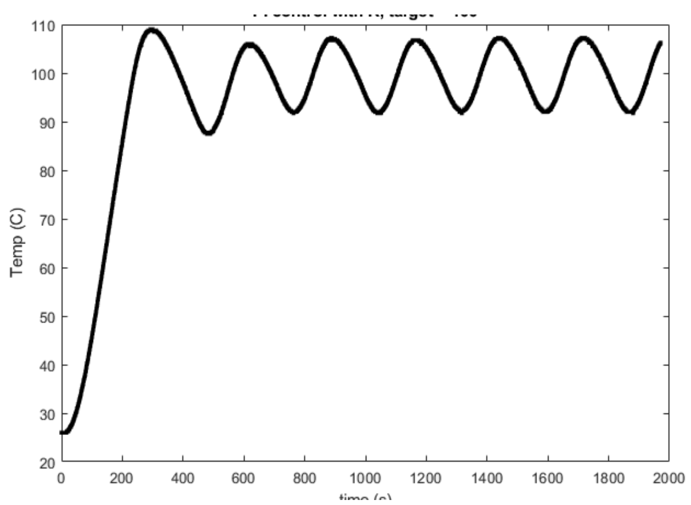


Figure 6.2

Multiplying by the R value only saw larger amplitude in oscillations and the temperature never settled (Fig. 6.2). The next step was to use a more aggressive tuning method provided by Skogestad (Skogestad, 2001) which gave a slightly higher overshoot, but a much more efficient change over all because it only takes a maximum of twenty minutes (Fig.7).

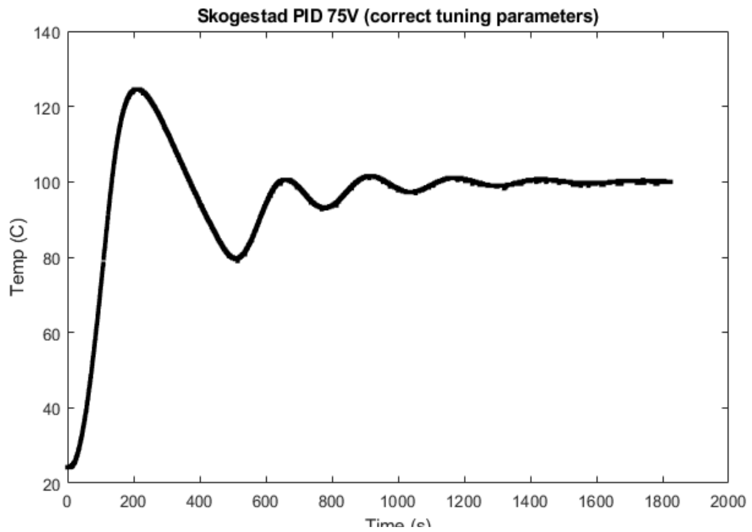


Figure 7

With the new values we could determine the system's behavior in response to a setpoint temperature upset (Fig. 8.1) or environmental upset (Fig 8.2). It responded well; the controller was able to adjust to the new target in the same speed it took to get to its initial target and was able to supply more heat due to heat loss the fan created.

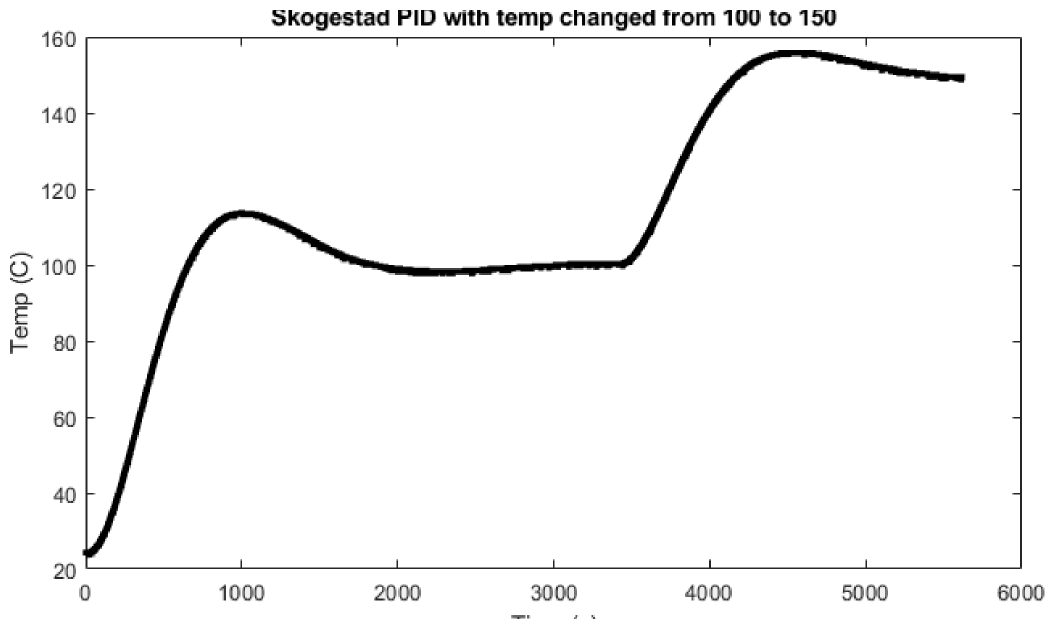

Skogestad PID with temp changed from 100 to 150
Temp (C)
160
140
120
100
80
60
40
20
0
1000
2000
3000
4000
5000
6000

Figure 8.1

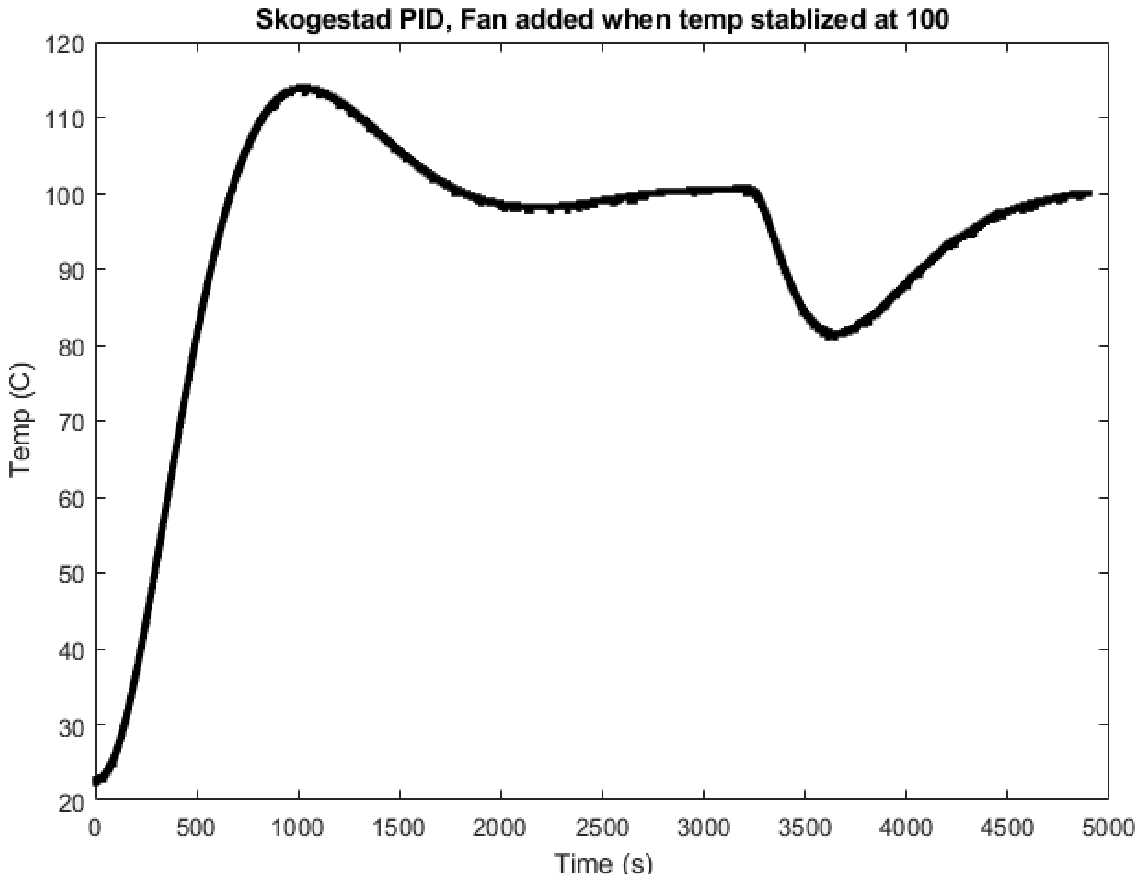

Skogestad PID, Fan added when temp stablized at 100
Temp (C)
Time (s)
120
110
100
90
80
70
60
50
40
30
20
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

Figure 8.2

The final step was adding a fan after a certain interval. The fan would stay on and cool the system while the heater was still on. When tuning with the fan on the whole time, we received new tuning values that can be used to accurately adjust the PWM while the fan is running. This could be considered gain scheduling, as the K value is the only one that changes. With the fan on, the result is comparable to the test without the fan, but with less amplitude. We tested with multiple cases as to when the fan engages and over all, the earlier it engaged the better it performed, even if it was only by a little bit. The wavelengths were slightly shorter as the temperature it engaged at decreased (figure 9).

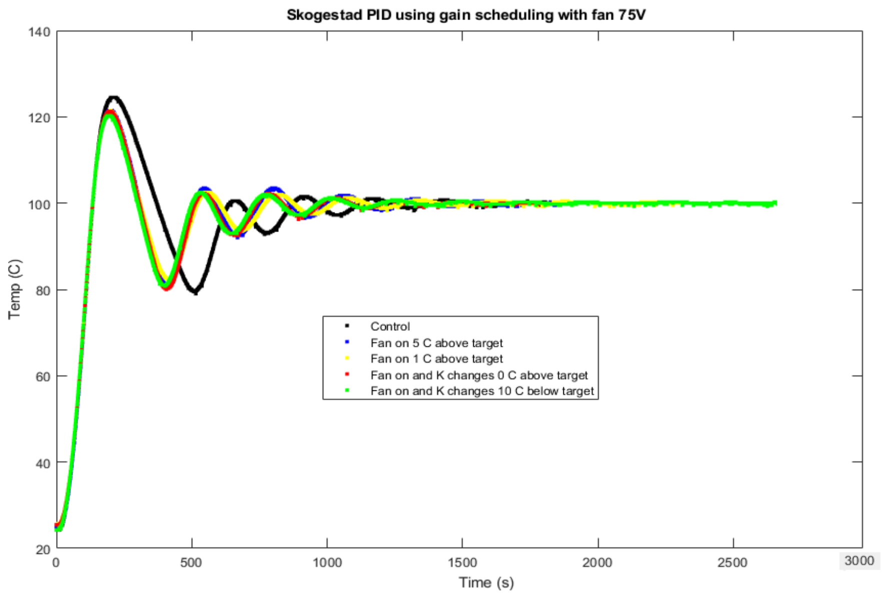


Figure 9

Conclusion:

It can be said that the data shows that the fan can create instability, but if used correctly alongside effective tuning values, a fan can reduce the wavelength and effectively decrease the time it takes for a heating element to get to its set point. Skogestad PID tuning was a bit more effective than Zeigler-Nichols, because it was less aggressive. The more aggressive the tuning values were, the higher the overshoot was.

Appendix 1

Theoretical Considerations

0.1 Basic PID Control Theory

Let the *process variable* $x \in C^2(\mathbb{R})$, and let the *state* $y = \left[\begin{smallmatrix} x' \\ x \end{smallmatrix}\right] \in C^1(\mathbb{R}, \mathbb{R}^2)$, where primes denote differentiation. Let $U = \left[\begin{smallmatrix} \mu \\ \nu \end{smallmatrix}\right] \in C^1(\mathbb{R}, \mathbb{R}^2)$ be the *control variable*, and suppose that y satisfies the ordinary differential equation (ODE) $y' = f(y, U)$, $f \in C^1(\mathbb{R}^2 \times \mathbb{R}^2, \mathbb{R}^2)$, with initial conditions (IC) $y(0) = y_0 = \left[\begin{smallmatrix} x_0' \\ x_0 \end{smallmatrix}\right]$ and $U(0) = U_0 = \left[\begin{smallmatrix} \mu_0 \\ \nu_0 \end{smallmatrix}\right]$.

In the experimental portion of this study, x is the temperature of a band heater thermocouple, and μ (or more precisely, u from equation (6) below) can be either the band heater's average voltage, average power, or the duty cycle of the Arduino's PWM output to the band heater SSR. Duty cycle was found to be the most convenient choice for the control variable: it is proportional to the heater power, which appears in the governing ODE; it is independent of the power supply voltage and heater resistance; and it is the internal variable actually used by the Arduino to specify the PWM output. The Arduino uses a 10 bit unsigned integer for the duty cycle; hence, the maximum value of u is 1023.

We want to "control" y to a given *steady state* y_∞, i.e. choose U such that

$$\lim_{t \to \infty} y(t) = y_\infty = \begin{bmatrix} 0 \\ x_\infty \end{bmatrix}.$$

To this end, we decompose f into the sum of a linear and nonlinear part; in particular, by Taylor's theorem,

$$y'(y, U) = \partial_y f(y_0, U_0)(y - y_0) + \partial_U f(y_0, U_0)(U - U_0) + R,$$

where $U_0 = U(0)$ and R is the remainder. Suppose that the control is *additive*, i.e. $f(y, U) = g(y) + U$ for some $g \in C^1(\mathbb{R}^2, \mathbb{R}^2)$. Then

$$\partial_y f(y_0, U_0) = \begin{bmatrix} -a & -b \\ 1 & 0 \end{bmatrix}, \ a, b \in \mathbb{R},$$

(note that x' is not a function of x), and

$$\partial_U f(y_0, U_0) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Therefore,

$$\begin{bmatrix} x' \\ x \end{bmatrix}' = \begin{bmatrix} -a & -b \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x' - x'_0 \\ x - x_0 \end{bmatrix} + \begin{bmatrix} \mu - \mu_0 \\ \nu - \nu_0 \end{bmatrix} + R,$$

which gives

$$x'' + a(x' - x'_0) + b(x - x_0) \approx \mu - \mu_0.$$

Assuming equality in the above relation ($R = 0$) and taking $\mu_0 = 0$, we have

$$x'' + ax' + bx = \mu + ax'_0 + bx_0.$$

Choosing $\mu = -\alpha x' - \beta x - (ax'_0 + bx_0),\ \alpha, \beta \in \mathbb{R}$, then yields

$$x'' + ax' + bx = -\alpha x' - \beta x,\ x(0) = x_0,\ x'(0) = x'_0, \tag{1}$$

which is the usual starting point in treatments of *proportional-differential* (PD) control.

If α and β are chosen such that the roots of the corresponding characteristic equation, $r^2 + (a+\alpha)r + b + \beta = 0$, are negative, then the solution of (1) will decay exponentially to 0 as $t \to \infty$. However, since it is required that $\lim_{t\to\infty} x(t) = x_\infty \neq 0$, we instead take

$$\mu = -\alpha x' - \beta(x - x_\infty) - (ax'_0 + b(x_0 - x_\infty)), \tag{2}$$

so that equation (1) is replaced by

$$x'' + ax' + bx = -\alpha x' - \beta(x - x_\infty) + bx_\infty,$$

or, in terms of the *error*, $e = x_\infty - x$,

$$x'' + ax' + bx = \alpha e' + \beta e + bx_\infty,\ x(0) = x_0,\ x'(0) = x'_0. \tag{3}$$

Provided α and β are properly chosen, Laplace transformation yields a solution of (3) possessing the desired properties. The control law (2) is therefore an exact solution to the $R = 0$ approximation of our original control problem.

In practice, the coefficients a and b of the ODE derive from the physical nature of the dynamical system to be controlled and are known only to within a given accuracy. Under these circumstances, equation (3) may be written

$$x'' + ax' + bx = \alpha e' + \beta e + bx_\infty + \epsilon,\ x(0) = x_0,\ x'(0) = x'_0,$$

where ϵ is the remainder of (2), modulo the aforementioned error. With the change of variable, $\varphi = x - x_\infty$, we then have

$$\varphi'' + a\varphi' + b\varphi = u + \epsilon,\ \varphi(0) = x_0 - x_\infty,\ \varphi'(0) = x_0', \tag{4}$$

where u, the control variable for φ, takes the simple form

$$u = -\alpha\varphi' - \beta\varphi = \alpha e' + \beta e. \tag{5}$$

Solutions of $(4, 5)$ are known (Sontag, p. 19) to approach the *steady-state error* ϵ as $t \to \infty$. To ensure that $\lim_{t\to\infty} \varphi(t) = \lim_{t\to\infty} \varphi'(t) = 0$, it suffices (Sontag, p. 19) to add an integrated error term to (5), thereby arriving at the *proportional-integral-derivative* (PID) control law:

$$u(t) = \alpha e'(t) + \beta e(t) + \gamma \int_0^t e(\tau)\, d\tau. \tag{6}$$

In the engineering literature, it is customary to absorb bx_∞ into ϵ, so that, instead of (4), we use

$$x'' + ax' + bx = u + \epsilon,\ x(0) = x_0,\ x'(0) = x_0'. \tag{7}$$

The ODE (7), together with the control law (6), are the basic ingredients of PID control theory. From the foregoing development, it is clear that, since (6) and (7) result from linearization of a fairly general ODE, we expect that PID control should perform well, at least *locally*, i.e. close to the IC of (7), for a broad class of physical systems. This expectation is borne out in practice.

0.2 PID Tuning

PID *tuning* is the process by which the coefficients α, β, and γ of (6) are selected for a given control system. Proper tuning is necessary for "good" performance, e.g. small steady state error, fast response to changes in setpoint, minimal oscillation about the setpoint, ability to correct for environmental upsets, and robustness against model uncertainty. We will not discuss those aspects of classical feedback control (Skogestad and Postlethwaite, ch. 2) dealing with closed-loop stability and performance, controller design, loop and transfer function shaping, etc., but will instead describe, in some detail, the actual implementation of the two tuning rules used in this project: Ziegler–Nichols (1942) and Skogestad (2001).

To this end, let the *controller gain*, *integral time*, and *derivative time* be $K_c = \beta$, $\tau_I = \beta/\gamma$, and $\tau_D = \alpha/\beta$, respectively. In terms of these parameters, the PID control law (6) takes the so-called *standard form*,

$$u(t) = K_c \left(e(t) + \frac{1}{\tau_I} \int_0^t e(\tau)\, d\tau + \tau_D e'(t) \right). \tag{8}$$

The Ziegler–Nichols and Skogestad tuning procedures are applied to a given system by first fitting a specific model function to the experimentally determined *step response*, i.e. the response of the system (initially at steady state) to a "step" in the control output, $u_{\max} H_0(t)$, where $u_{\max}$ is the maximum control value, and H_0 is the Heaviside step function with jump discontinuity at $t = 0$. The fit parameters of the respective model function are then input into formulae giving the desired tuning parameters.

The Ziegler–Nichols method assumes (Astrom and Hagglund, p. 14, 135) a system transfer function of the form

$$G(s) = \frac{a}{Ls} \exp(-Ls),\ \ a, L \in \mathbb{R}_+.$$

The corresponding Laplace domain step response is

$$\begin{aligned} X(s) &= G(s)\mathcal{L}[u_{\max} H_0(t)](s) \\ &= \frac{a u_{\max}}{L s^2} \exp(-Ls), \end{aligned}$$

which, upon taking the inverse Laplace transform, gives

$$x(t) = \frac{a u_{\max}}{L} (t - L) H_L(t - L) \tag{9}$$

for the time domain step response. Since the experimental step response curve is sigmoidal, the tangent line at the point of maximum slope is a good approximation to (9) for $t \geq L$. This tangent line was found by differentiating a sixth degree polynomial fit to the data with Matlab's `polyfit` function. The parameters a and L are the absolute values of the tangent line's x and t intercepts. According to Ziegler–Nichols, the PID tuning parameters are then given by $K_c = 1.2/a$, $\tau_I = 2L$, and $\tau_D = L/2$. A Matlab program (Appendix 3a) was written to automate the foregoing data analysis.

Skogestad's method employs the transfer function for a "second-order + time delay" process,

$$G(s) = \frac{k}{(\tau_1 s + 1)(\tau_2 s + 1)} \exp(-\theta s),\ \ k, \tau_1, \tau_2, \theta \in \mathbb{R}_+,$$

which is readily seen to be the Laplace transform of a Heaviside shifted solution to a second-order linear ODE with constant coefficients and real characteristic polynomial roots. Again, the corresponding Laplace domain step response is

$$X(s) = \frac{k u_{\max}}{s(\tau_1 s + 1)(\tau_2 s + 1)} \exp(-\theta s),$$

which, upon inversion, yields the time domain step response

$$x(t) = k u_{\max} \left(1 - \frac{\tau_1}{\tau_1 + \tau_2} \exp\left(-\frac{t-\theta}{\tau_1} \right) - \frac{\tau_2}{\tau_1 + \tau_2} \exp\left(-\frac{t-\theta}{\tau_2} \right) \right). \quad (10)$$

Matlab's `lsqcurvefit` function was used to determine the parameters k, τ_1, τ_2, and θ via a nonlinear least squares fit of (10) to step response data. Following Skogestad, the PID tuning parameters are then given by:

$$\begin{array}{lllll} \tau_1 \le 8\theta : & K_c = \dfrac{1}{2k} \dfrac{\tau_1 + \tau_2}{\theta}, & \tau_I = \tau_1 + \tau_2, & \tau_D = \dfrac{\tau_2}{1 + \frac{\tau_2}{\tau_1}} \\ \tau_1 > 8\theta : & K_c = \dfrac{\tau_1}{2k\theta} \left(1 + \dfrac{\tau_2}{8\theta} \right), & \tau_I = 8\theta + \tau_2, & \tau_D = \dfrac{\tau_2}{1 + \frac{\tau_2}{8\theta}}. \end{array}$$

Appendix 3b shows the Matlab code written to carry out PID tuning via Skogestad's method.

Appendix 2a

```
/* PIDcontrol
   Julian Stapleton
   7/2/18
   Thermocouple temps written to an SD card when physical button is
pressed
   samples every second, simultaneous with pwm changes, sampling does
add delay
   start/pause measuring with hard button
   write to new file with reset button
   can ask for target temp, ambient temp, heat transfer coefficient,
and resistance in a circuit, and maximum voltage
   determines target voltage and chooses pwm frequency accordingly to
get desired temperature
*/
#include <SPI.h>
#include <SD.h>
#include"TimerOne.h"
#include"PlayingWithFusion_MAX31855_1CH.h"
#include"PlayingWithFusion_MAX31855_STRUCT.h"
boolean scronch = true;
File root;
File tempsFile;
float ambientTemp = 0;
float count = 0;
float cronch = true;
float currentTemp = 0;
String data = "";
int dutyCycle = 1;
unsigned long delayy;
long egg;
float initTemp;
float Kfinal;
float k;
unsigned long lastCheck = 0;
unsigned long lastChange = 0;
boolean leg = false;
boolean limOn = true;
float lrCoef[] = {0, 0};
float maxVolt = 0;
```

```
long period;
double resist;
float targetTemp = 0;
float Ti;
float T2;
float Td;
float theta;
float tempsList[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; //try different number of zeros
float timeList[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int tempsSize = (sizeof(tempsList)/sizeof(float));
unsigned int startTime;
float slope;
float x = -3;

float error = (targetTemp-currentTemp);
float duty=1;//((K*(error+(count/Ti)+(T2*slope))/sq(maxVolt))*1023);
int button_read_pin = 2;
int SSR_pin = 9;
boolean SSR_state = LOW;
boolean beenPressed = false;
int8_t CS3_PIN =  8;
int8_t CS2_PIN =  7;
int8_t CS1_PIN =  6;
int8_t CS0_PIN =  5;
const int sdPin = 4;
int8_t fanPin = 3;
PWFusion_MAX31855_TC thermocouple0(CS0_PIN);
PWFusion_MAX31855_TC thermocouple1(CS1_PIN);
PWFusion_MAX31855_TC thermocouple2(CS2_PIN);
PWFusion_MAX31855_TC thermocouple3(CS3_PIN);
void printDirectory(File dir, int numTabs){
  while (true){
    x++;//determines new file name (one more than the last)
    //Serial.println(x);
    File entry =  dir.openNextFile();
    if(! entry){
      // no more files
      break;
    }
    for (int i = 0; i < numTabs; i++){
      Serial.print('\t');
    }
   Serial.print(entry.name());
   if(entry.isDirectory()){
     Serial.println(F("/"));
     printDirectory(entry, numTabs+1);
    }else{
      // files have sizes, directories do not
     Serial.print(F("\t\t"));
     Serial.println(entry.size(), DEC);
    }
   entry.close();
  }
}
void copy(float arrayOriginal[], float arrayCopy[], int arraySize){
  while (arraySize > 0){
   arrayCopy[arraySize] = arrayOriginal[arraySize];
    arraySize--;
  }
}
float* progress(float newTemp, float list[]){
  for (int i = 0; i <= tempsSize-2 ; i++){
    list[i] = list[i+1];
  }
  list[tempsSize-1] = newTemp;
  return list;
}
float* simpLinReg(float* x, float* y, float* lrCoef, int n){
  // pass x and y arrays (pointers), lrCoef pointer, and n.  The lrCoef
array is comprised of the slope=lrCoef[0] and intercept=lrCoef[1].  n
is length of the x and y arrays.
//http://en.wikipedia.org/wiki/Simple_linear_regression

  // initialize variables
  float xbar = 0;
  float ybar = 0;
  float xybar = 0;
  float xsqbar = 0;

  // calculations required for linear regression
  for (int i = 0; i < n; i++){
    xbar = xbar+x[i];
    ybar = ybar+y[i];
    xybar = xybar+x[i]*y[i];
    xsqbar = xsqbar+x[i]*x[i];
  }
  xbar = xbar/n;
  ybar = ybar/n;
  xybar = xybar/n;
  xsqbar = xsqbar/n;

  // simple linear regression algorithm
 lrCoef[0] = (xybar-xbar*ybar)/(xsqbar-xbar*xbar);
 lrCoef[1] = ybar-lrCoef[0]*xbar;
 return (lrCoef);
}
booleandebounced_button_press(intbutton_read_pin){
  int debounce_delay = 10;
 boolean reference_button_state = digitalRead(button_read_pin);
 delay(debounce_delay);
 if((digitalRead(button_read_pin) == reference_button_state) &&
(reference_button_state == HIGH)){
    return true;
  }
 else{
    return false;
  }
}
void oof(){
  egg = millis();
  data = egg;
  data += ",";
  static struct var_max31855 TC_CH0;
  static struct var_max31855 TC_CH1;
  static struct var_max31855 TC_CH2;
 static struct var_max31855 TC_CH3;
  double tmp;
  // update TC0
 struct var_max31855 *tc_ptr;
  tc_ptr = &TC_CH0;
  thermocouple0.MAX31855_update(tc_ptr);      // Update MAX31855
readings
  // update TC1
  tc_ptr = &TC_CH1;
  thermocouple1.MAX31855_update(tc_ptr);      // Update MAX31855
readings
  // update TC2
  tc_ptr = &TC_CH2;
  thermocouple2.MAX31855_update(tc_ptr);      // Update MAX31855
readings
  // update TC3
  tc_ptr = &TC_CH3;
  thermocouple3.MAX31855_update(tc_ptr);      // Update MAX31855
readings
  // TC0
  //Serial.println(F("Thermocouple 0:"));          // Print TC0 header
  // MAX31855 Internal Temp
  //  tmp = (double)TC_CH0.ref_jcn_temp*0.0625;  // convert fixed pt #
to double
  //  Serial.print(F("Tint = z                        // print internal
temp heading
  //  if((-100 > tmp) || (150 < tmp)){Serial.println(F("unknown fault");
}
 // else{Serial.println(tmp);data+=tmp;}
  // MAX31855 External (thermocouple) Temp
  tmp = (double)TC_CH0.value*0.25;           // convert fixed pt # to
double
 //data+=",";
  //Serial.print(F("TC Temp = "));                   // print TC temp
heading
 if(0x00 == TC_CH0.status){
   //Serial.println(tmp);
  }//data+=tmp;}
 else if(0x01 == TC_CH0.status){
   Serial.println(F("OPEN"));
  }
 else if(0x02 == TC_CH0.status){
   Serial.println(F("SHORT TO GND"));
  }
 else if(0x04 == TC_CH0.status){
   Serial.println(F("SHORT TO Vcc"));
  }
 else{
   Serial.println(F("unknown fault"));
  }
 //data+=",";
  //  // TC1
  //Serial.println(F("Thermocouple 1:"));           // Print TC0 header
  //  // MAX31855 Internal Temp
  //  tmp = (double)TC_CH1.ref_jcn_temp*0.0625;  // convert fixed pt #
to double
  //  Serial.print(F("Tint = ");                         // print internal
temp heading
  //  if((-100 > tmp) || (150 < tmp)){Serial.println(F("unknown fault");
}
 // else{Serial.println(tmp);data+=tmp;}
  //  // MAX31855 External (thermocouple) Temp
  tmp = (double)TC_CH1.value*0.25;           // convert fixed pt # to
double
 //data+=",";
  //Serial.print(F("TC Temp = "));                    // print TC temp
heading
 if(0x00 == TC_CH1.status){
   //Serial.println(tmp);
    data += tmp;
  }
 else if(0x01 == TC_CH1.status){
   Serial.println(F("OPEN"));
  }
 else if(0x02 == TC_CH1.status){
   Serial.println(F("SHORT TO GND"));
  }
 else if(0x04 == TC_CH1.status){
   Serial.println(F("SHORT TO Vcc"));
  }
 else{
   Serial.println(F("unknown fault"));
  }
  data += ",";

  //  // TC2
  //Serial.println(F("Thermocouple 2:"));           // Print TC0 header
  //  // MAX31855 Internal Temp
  //  tmp = (double)TC_CH2.ref_jcn_temp*0.0625;  // convert fixed pt #
to double
  //  Serial.print(F("Tint = ");                          // print internal
temp heading
  //  if((-100 > tmp) || (150 < tmp)){Serial.println(F("unknown fault");
}
 // else{Serial.println(tmp);data+=tmp;}
  //  // MAX31855 External (thermocouple) Temp
  tmp = (double)TC_CH2.value*0.25;          // convert fixed pt # to
double
  //  data+=",";
  //Serial.print(F("TC Temp = "));                     // print TC temp
heading
  if(0x00 == TC_CH2.status){
   //Serial.println(tmp);
    data += tmp;
  }
 else if(0x01 == TC_CH2.status){
   Serial.println(F("OPEN"));
  }
 else if(0x02 == TC_CH2.status){
   Serial.println(F("SHORT TO GND"));
  }
 else if(0x04 == TC_CH2.status){
   Serial.println(F("SHORT TO Vcc"));
  }
 else{
   Serial.println(F("unknown fault"));
  }
  data += ",";
  // TC3
  // Serial.println(F("Thermocouple 3:"));          // Print TC0
header
  // MAX31855 Internal Temp
  //  tmp = (double)TC_CH3.ref_jcn_temp*0.0625;  // convert fixed pt #
to double
  //  Serial.print(F("Tint = ");                    // print internal
temp heading
  //  if((-100 > tmp) || (150 < tmp)){Serial.println(F("unknown fault");
}
 // else{Serial.println(tmp);data+=tmp;}
  // MAX31855 External (thermocouple) Temp
  tmp = (double)TC_CH3.value*0.25;          // convert fixed pt # to
double
  //  data+=",";
  //Serial.print(F("TC Temp = "));                 // print TC temp
heading
  if(0x00 == TC_CH3.status){
   //Serial.println(tmp);
    data += tmp; currentTemp = tmp;
    if(!leg){
      initTemp = currentTemp; leg = true;
      for (int i = 0; i < tempsSize; i++){
       tempsList[i] = initTemp/sq(maxVolt);
      }
    }
   copy(tempsList, progress(tmp/sq(maxVolt), tempsList),
tempsSize);                                  //tempchange
   error = (targetTemp-currentTemp);
  }
 else if(0x01 == TC_CH3.status){
   Serial.println(F("OPEN"));
  }
 else if(0x02 == TC_CH3.status){
   Serial.println(F("SHORT TO GND"));
  }
 else if(0x04 == TC_CH3.status){
   Serial.println(F("SHORT TO Vcc"));
  }
 else{
   Serial.println(F("unknown fault"));
  }
  data += ","+(String)duty;
 tempsFile = SD.open("temps"+(String)((int)x+1)+".txt", FILE_WRITE);
  if(tempsFile){
   tempsFile.println(data);
    tempsFile.close();
    // print to the serial port too:
   Serial.println(data);
  }
  // ifthe file isn't open, pop up an error:
 else{
   Serial.println(F("error opening temps"));
Serial.print((String)x+1+".txt");
  }
}
float ask(String s){
  float f = 0;
 Serial.print(F("What is the ")); Serial.println(s);
  while (Serial.available() == 0){}
 f = (Serial.readString()).toFloat();
 Serial.println(f);
  return f;
}
void
setup(){
//SETUP SETUP SETUP
 Serial.begin(9600);
 pinMode(button_read_pin, INPUT);
 pinMode(SSR_pin, OUTPUT);
 pinMode(fanPin, OUTPUT);
 digitalWrite(SSR_pin, SSR_state);
 digitalWrite(fanPin, LOW);
  // Open serial communications and wait for port to open:
 Serial.begin(9600);
  while (!Serial){
    ; // wait for serial port to connect. Needed for native USB port
```

```
only
  }
 Serial.print(F("Initializing SD card..."));
 if(!SD.begin(sdPin)){
   Serial.println(F("initialization failed!"));
    while (1);
  }
 Serial.println(F("initialization done."));
  root = SD.open("/");
  printDirectory(root, 0);
 Serial.println(F("done!"));
  SPI.begin();                          // begin SPI
  SPI.setDataMode(SPI_MODE1);           // MAX31865 is a Mode 1 device
  //    --> clock starts low, read on rising edge
  // initalize the chip select pins
  pinMode(CS0_PIN, OUTPUT);
  pinMode(CS1_PIN, OUTPUT);
  pinMode(CS2_PIN, OUTPUT);
  pinMode(CS3_PIN, OUTPUT);

 tempsFile = SD.open("temps"+(String)((int)x+1)+".txt", FILE_WRITE);
  if(tempsFile){
   //Serial.println(F("Printing temperatures"));
    tempsFile.close();
   //Serial.println(F("Done."));
  }else{
   //Serial.println(F("Error opening file in setup."));
   }
 //Serial.println(F("Playing With Fusion: MAX31855-4CH, SEN-30002"));
 targetTemp = ask(F("TARGET TEMPERATURE?"));
 ambientTemp = ask(F("AMBIENT TEMPERATURE?"));
  resist = ask(F("RESISTANCE?"));
  Kfinal = ask(F("K?"));
  Ti = ask(F("Ti?"));
  Td = ask(F("Td?"));
 //ask(F("THETA (delay before heating in seconds)"));
  //calculations nevermind use matlab for these
 //Kfinal = ((.5*Ti)/(k*theta))*(1+(T2/(8*theta)));
  //Ti = 8*theta+T2;
 //Td = T2/(1+(T2/(8*theta)));

  delayy = ask(F("DELAY in seconds(controls when the PWM is
calculated)"));
 delayy = delayy*1000;//scale
 maxVolt = ask(F("MAXIMUM VOLTAGE"));
 long period = (long)ask(F("PERIOD in miliseconds?"));
 period = period*1000;//more scales
  dutyCycle = 0;//default off
  duty = 0; //techinally doesnt mean anything
  count = 0; // integral starts at zero
 //Serial.print(F("Kfinal is ")); Serial.println(Kfinal); //debug
 Serial.println(F("press to start: "));
 //Serial.print(period);//DEBUG
  startTime = (int) millis();
 Timer1.initialize(period);
}
void loop(){
 if(debounced_button_press(button_read_pin) == true){
   while (debounced_button_press(button_read_pin) == true){
     }
   //Serial.print(F("pressed: "));
   beenPressed = !beenPressed;
   if(beenPressed){
     Serial.println(F("on"));
     Timer1.pwm(SSR_pin, dutyCycle, period);
    }else{
     Serial.println(F("off"));
     Timer1.pwm(SSR_pin, 0, 100);
      SSR_state = LOW;
     digitalWrite(fanPin, LOW);
    }
  }
 if(beenPressed){
   unsigned long now = millis()-startTime;
   if(now-lastChange >= delayy){ //changes dutyCycle every (user
determined) seconds
     if(currentTemp >= targetTemp -10 && scronch){//ten is arbitrary
but it worked well
```

```
      //digitalWrite(fanPin, HIGH);
       //give it tuning values with the fan on
      scronch = false;//only lets this happen once
     }else{
      //digitalWrite(fanPin, LOW);//NEVER USE THIS PART IT WILL LEAD
TO PERMANENT OSCILLATIONS
     }
     if(cronch){
     duty = (long)(Kfinal*(error+(count/Ti)+(Td*slope)));
      if((duty > 1023 && error > 0)){//for integrator wind up
        //otherwise count is constant for this cycle
       }else{
        count = count+(error*(now-lastChange)/1000);
       }
     duty = (long)(Kfinal*(error+(count/Ti)+(Td*slope)));
       lastChange = now;
       slope = lrCoef[0];
       if(duty >= 1023){
         dutyCycle = 1023;
      }else{if(duty < 0){dutyCycle = 0;}else{
           dutyCycle = duty;
          }
       }
      Timer1.pwm(SSR_pin, dutyCycle, period);
     }else{//always on until current = target set cronch to false if
desired
       duty = 1023;
       dutyCycle = duty;
      Timer1.pwm(SSR_pin, dutyCycle, period);
      if(currentTemp >= targetTemp){
         cronch = true;
       }
     }
   }
   if(now-lastCheck >= 1000){ //checks once per second
     oof();
     lastCheck = now;
    copy(timeList, progress(lastCheck/1000, timeList), tempsSize);
//this just shifts the array and destroys the first value adding data
```

```
to the last
    //Serial.print(currentTemp);Serial.print(" ");Serial.
println(duty);
    Serial.print(F("currentTemp: ")); Serial.print(currentTemp);
Serial.print(F("dutyCycle:"));Serial.print(dutyCycle);Serial.
print(F(" error:")); Serial.print(error, 2); Serial.print(F(" Slope:
"));Serial.println(slope,7);
    Serial.print(F("time elapsed: ")); Serial.print((now/60000));
Serial.print(F(" minutes and ")); Serial.
print(((now/1000)-((now/60000)*60))); Serial.print(F(" seconds"));
Serial.print(F("cronch="));Serial.println((String)cronch);
      if( now > 2000)
       copy(lrCoef, simpLinReg(timeList, tempsList, lrCoef,
tempsSize), 2);
    }
  }
}
```

Appendix 2b

```
/* Tuning
   v4.0
   7/10/18
   Thermocouple temps written to an SD card when physical button is
pressed
   samples every second, simultaneous with pwm changes, sampling does
add delay
   start/pause measuring with hard button
   write to new file with reset button
*/
#include <SPI.h>
#include <SD.h>
#include"TimerOne.h"
#include"PlayingWithFusion_MAX31855_1CH.h"
#include"PlayingWithFusion_MAX31855_STRUCT.h"

File root;
boolean leg = false;
boolean legg = true;
unsigned int startTime;
unsigned long lastCheck = 0;
unsigned long period;
float maxSlope;
float initTemp;
float finalTemp;
float theta;
float intercept;
unsigned int dutyCycle;
float targetTemp = 0;
float currentTemp;
float maxVolt;
float tempsList[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};//try different number of zeros
float timeList[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int tempsSize = (sizeof(tempsList) / sizeof(float));
float lrCoef[] = {0, 0};
int button_read_pin = 2;
int SSR_pin = 9;
```

```
boolean SSR_state = LOW;
boolean beenPressed = false;
int8_t CS3_PIN =  8;
int8_t CS2_PIN =  7;
int8_t CS1_PIN =  6;
int8_t CS0_PIN =  5;
const int sdPin = 4;
File tempsFile;
String data = "";
long egg;
PWFusion_MAX31855_TCthermocouple0(CS0_PIN);
PWFusion_MAX31855_TCthermocouple1(CS1_PIN);
PWFusion_MAX31855_TCthermocouple2(CS2_PIN);
PWFusion_MAX31855_TCthermocouple3(CS3_PIN);
float x = -3;
void printDirectory(File dir, int numTabs) {
  while (true) {
    x++;
   //Serial.println(x);
    File entry =  dir.openNextFile();
    if (! entry) {
      // no more files
      break;
    }
    for (int i = 0; i < numTabs; i++) {
      Serial.print('\t');
    }
   Serial.print(entry.name());
    if (entry.isDirectory()) {
      Serial.println(F("/"));
      printDirectory(entry, numTabs + 1);
    } else {
      // files have sizes, directories do not
      Serial.print("\t\t");
     Serial.println(entry.size(), DEC);
    }
    entry.close();
  }
}
```

```
void copy(float arrayOriginal[], float arrayCopy[], int arraySize) {
  while (arraySize > 0) {
   arrayCopy[arraySize] = arrayOriginal[arraySize];
    arraySize--;
  }
}
float* progress(float newTemp, float list[]) {
  for (int i = 0; i <= tempsSize - 2 ; i++) {
    list[i] = list[i + 1];
  }
  list[tempsSize - 1] = newTemp;
  return list;
}
booleandebounced_button_press(intbutton_read_pin){
  int debounce_delay = 10;
 boolean reference_button_state = digitalRead(button_read_pin);
 delay(debounce_delay);
 if ((digitalRead(button_read_pin) == reference_button_state) &&
(reference_button_state == HIGH)) {
    return true;
  }
  else {
    return false;
  }
}
void oof() {
  egg = millis();
  data = egg;
  data += ",";
  static struct var_max31855 TC_CH0;
  static struct var_max31855 TC_CH1;
  static struct var_max31855 TC_CH2;
  static struct var_max31855 TC_CH3;
  double tmp;
  // update TC0
  struct var_max31855 *tc_ptr;
  tc_ptr = &TC_CH0;
  thermocouple0.MAX31855_update(tc_ptr);       // Update MAX31855
readings
```

```
 // update TC1
 tc_ptr = &TC_CH1;
 thermocouple1.MAX31855_update(tc_ptr);      // Update MAX31855
readings
 // update TC2
 tc_ptr = &TC_CH2;
 thermocouple2.MAX31855_update(tc_ptr);      // Update MAX31855
readings
 // update TC3
 tc_ptr = &TC_CH3;
 thermocouple3.MAX31855_update(tc_ptr);      // Update MAX31855
readings
 // TC0
 Serial.println(F("Thermocouple 0:"));          // Print TC0 header
 // MAX31855 Internal Temp
 //  tmp = (double)TC_CH0.ref_jcn_temp * 0.0625;  // convert fixed pt
# to double
 //  Serial.print("Tint = ");                    // print internal
temp heading
 //  if((-100 > tmp) || (150 < tmp)){Serial.println("unknown fault");}
 // else{Serial.println(tmp);data+=tmp;}
 // MAX31855 External (thermocouple) Temp
 tmp = (double)TC_CH0.value * 0.25;          // convert fixed pt # to
double
 //data+=",";
 Serial.print(F("TC Temp = "));                 // print TC temp
heading
 if (0x00 == TC_CH0.status) {
  Serial.println(tmp);
 }//data+=tmp;}
 else if (0x01 == TC_CH0.status) {
  Serial.println("OPEN");
  }
 else if (0x02 == TC_CH0.status) {
  Serial.println("SHORT TO GND");
  }
 else if (0x04 == TC_CH0.status) {
  Serial.println("SHORT TO Vcc");
  }
  else {
   Serial.println("unknown fault");
  }
 //data+=",";
  //  // TC1
  Serial.println(F("Thermocouple 1:"));           // Print TC0 header
  //  // MAX31855 Internal Temp
  //  tmp = (double)TC_CH1.ref_jcn_temp * 0.0625;  // convert fixed pt
# to double
  //  Serial.print("Tint = ");                      // print internal
temp heading
  //  if((-100 > tmp) || (150 < tmp)){Serial.println("unknown fault");}
 // else{Serial.println(tmp);data+=tmp;}
  //  // MAX31855 External (thermocouple) Temp
  tmp = (double)TC_CH1.value * 0.25;          // convert fixed pt # to
double
 //data+=",";
  Serial.print(F("TC Temp = "));                  // print TC temp
heading
  if (0x00 == TC_CH1.status) {
   Serial.println(tmp);
    data += tmp;
  }
  else if (0x01 == TC_CH1.status) {
   Serial.println("OPEN");
  }
  else if (0x02 == TC_CH1.status) {
   Serial.println("SHORT TO GND");
  }
  else if (0x04 == TC_CH1.status) {
   Serial.println("SHORT TO Vcc");
  }
  else {
   Serial.println("unknown fault");
  }
  data += ",";

  //  // TC2
  Serial.println("Thermocouple 2:");           // Print TC0 header
  //  // MAX31855 Internal Temp
  //  tmp = (double)TC_CH2.ref_jcn_temp * 0.0625;  // convert fixed pt
# to double
  //  Serial.print("Tint = ");                       // print internal
temp heading
  //  if((-100 > tmp) || (150 < tmp)){Serial.println("unknown fault");}
 // else{Serial.println(tmp);data+=tmp;}
 //  // MAX31855 External (thermocouple) Temp
  tmp = (double)TC_CH2.value * 0.25;          // convert fixed pt # to
double
  //  data+=",";
  Serial.print("TC Temp = ");                  // print TC temp heading
  if (0x00 == TC_CH2.status) {
   Serial.println(tmp);
    data += tmp;
  }
  else if (0x01 == TC_CH2.status) {
   Serial.println("OPEN");
  }
  else if (0x02 == TC_CH2.status) {
   Serial.println("SHORT TO GND");
  }
  else if (0x04 == TC_CH2.status) {
   Serial.println("SHORT TO Vcc");
  }
  else {
   Serial.println("unknown fault");
  }
  data += ",";
  // TC3
  Serial.println("Thermocouple 3:");           // Print TC0 header
  // MAX31855 Internal Temp
  //  tmp = (double)TC_CH3.ref_jcn_temp * 0.0625;  // convert fixed pt
# to double
  //  Serial.print("Tint = ");                       // print internal
temp heading
  //  if((-100 > tmp) || (150 < tmp)){Serial.println("unknown fault");}
 // else{Serial.println(tmp);data+=tmp;}
 // MAX31855 External (thermocouple) Temp
```

```
  tmp = (double)TC_CH3.value * 0.25;           // convert fixed pt # to
double
  //  data+=",";
  Serial.print("TC Temp = ");                   // print TC temp heading
  if (0x00 == TC_CH3.status) {
   Serial.println(tmp);
    data += tmp;
    currentTemp = tmp;
    if (!leg) {
      initTemp = currentTemp;
      leg = true;
      for (int i = 0; i < tempsSize; i++) {
       tempsList[i] = initTemp/maxVolt;
      }
    }
   copy(tempsList, progress(tmp / maxVolt, tempsList),
tempsSize);                                //tempchange
  }
  else if (0x01 == TC_CH3.status) {
   Serial.println("OPEN");
  }
  else if (0x02 == TC_CH3.status) {
   Serial.println("SHORT TO GND");
  }
  else if (0x04 == TC_CH3.status) {
   Serial.println("SHORT TO Vcc");
  }
  else {
   Serial.println("unknown fault");
  }
  data += "," + (String)targetTemp;
  tempsFile = SD.open("temps" + (String)((int)x + 1) + ".txt",
FILE_WRITE);
  if (tempsFile) {
   tempsFile.println(data);
   tempsFile.close();
    // print to the serial port too:
   Serial.println(data);
  }
  // if the file isn't open, pop up an error:
  else {
   Serial.println("error opening temps" + (String)x + 1 + ".txt");
  }
}
float ask(String s) {
  float f = 0;
  Serial.println("What is the " + s);
  while (Serial.available() == 0) {}
 f = (Serial.readString()).toFloat();
 Serial.println(f);
  return f;
}
float* simpLinReg(float* x, float* y, float* lrCoef, int n) {
  // pass x and y arrays (pointers), lrCoef pointer, and n.  The lrCoef
array is comprised of the slope=lrCoef[0] and intercept=lrCoef[1].  n
is length of the x and y arrays.
//http://en.wikipedia.org/wiki/Simple_linear_regression

  // initialize variables
  float xbar = 0;
  float ybar = 0;
  float xybar = 0;
  float xsqbar = 0;

  // calculations required for linear regression
  for (int i = 0; i < n; i++) {
    xbar = xbar + x[i];
    ybar = ybar + y[i];
    xybar = xybar + x[i] * y[i];
    xsqbar = xsqbar + x[i] * x[i];
  }
  xbar = xbar / n;
  ybar = ybar / n;
  xybar = xybar / n;
  xsqbar = xsqbar / n;

  // simple linear regression algorithm
  lrCoef[0] = (xybar - xbar * ybar) / (xsqbar - xbar * xbar);
  lrCoef[1] = ybar - lrCoef[0] * xbar;
  return (lrCoef);
}
void setup()
{                                                              //SETUP
SETUP SETUP
 Serial.begin(9600);
 pinMode(button_read_pin, INPUT);
 pinMode(SSR_pin, OUTPUT);
 digitalWrite(SSR_pin, SSR_state);
  // Open serial communications and wait for port to open:
 Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port
only
  }
 Serial.print(F("Initializing SD card..."));
  if (!SD.begin(4)) {
   Serial.println(F("initialization failed!"));
    while (1);
  }
 Serial.println(F("initialization done."));
  root = SD.open("/");
 printDirectory(root, 0);
 Serial.println(F("done!"));
  SPI.begin();                          // begin SPI
  SPI.setDataMode(SPI_MODE1);           // MAX31865 is a Mode 1 device
  //    --> clock starts low, read on rising edge
  // initalize the chip select pins
 pinMode(CS0_PIN, OUTPUT);
 pinMode(CS1_PIN, OUTPUT);
 pinMode(CS2_PIN, OUTPUT);
 pinMode(CS3_PIN, OUTPUT);

  tempsFile = SD.open("temps" + (String)((int)x + 1) + ".txt",
FILE_WRITE);
  if (tempsFile) {
   Serial.println(F("Printing temperatures"));
    tempsFile.close();
   Serial.println(F("Done."));
  } else {
   Serial.println("Error opening file in setup.");
  }
 Serial.println(F("Playing With Fusion: MAX31855-4CH, SEN-30002"));
 targetTemp = 100;//ask("TARGET TEMPERATURE?");
  maxVolt = ask("MAXIMUM VOLTAGE?");
  long period = 1000;//(long)ask("PERIOD in miliseconds?");
  //D = (RH(T-t))^1/2 / V // muliply by 1024 for dutyCycle
  period = period * 1000;
  dutyCycle = 1024;
  Serial.println(F("press to start: "));
//Serial.print(period);//DEBUG
 Timer1.initialize(period);
}
void loop() {
 if (debounced_button_press(button_read_pin) == true) {
   while (debounced_button_press(button_read_pin) == true) {
    }
   Serial.print(F("pressed: "));
   beenPressed = !beenPressed;
    if (beenPressed) {
     Serial.println(F("on"));
      startTime = (int) millis();
     Timer1.pwm(SSR_pin, 1024, period);
    } else {
      currentTemp = finalTemp;
     Serial.println(F("off"));
     Timer1.pwm(SSR_pin, 0, 100);
      SSR_state = LOW;
   Serial.print(F("K:"));Serial.print((finalTemp-initTemp)/(1023));
Serial.println();
    Serial.print(F("theta:"));Serial.print(theta);


    }
  }
  if (beenPressed == true) {
    unsigned long now = millis() - startTime;
    if (lrCoef[0] > maxSlope && now > 10000) {
      maxSlope = lrCoef[0];
      intercept = lrCoef[1];
    }
    if (now - lastCheck >= 1000) { //checks once per second
      lastCheck = now;
     copy(timeList, progress(lastCheck / 1000, timeList), tempsSize);
      oof();
      if(lrCoef[0] > 0 and legg){
        theta = now;
        legg = false;
      }
    copy(lrCoef, simpLinReg(timeList, tempsList, lrCoef, tempsSize), 
2);

      //for (int i = 0; i < tempsSize; i++)
      //  Serial.print("" + (String)tempsList[i] + " ");
     //Serial.println(F(""));
     Serial.print("dutyCycle: " + (String)dutyCycle + " currentTemp: " 
+ (String)currentTemp + " slope: "); Serial.print(lrCoef[0], 5); Serial.
print(F(" max Slope: ")); Serial.println(maxSlope, 5);
     Serial.println("time elapsed: " + (String)(now / 60000) + " 
minutes and " + (String)((now / 1000) - ((now / 60000) * 60)) + " 
seconds");
    }
  }
}
```

Appendix 3a

```
% Michael Goodman
% 11 July 2018
% Ziegler-Nichols PID step response tune w/ polynomial fit

clear all; close all; clc;

n = input('Number of File: ');
V_max = input('Maximum Voltage: ');
filename = strcat('TEMPS',num2str(n),'.TXT');
fileID = fopen(filename);
M = textscan(fileID,'%f %f %f %f %f','Delimiter',',\n');
fclose(fileID);

time = (M{1,1}-M{1,1}(1))/1000;
scaled_temp = M{1,4}/(V_max^2/72.4);

deg = 6;
fit_coeff = polyfit(time,scaled_temp,deg);
Dfit_coeff = fit_coeff(1:end-1).*[deg:-1:1];

fit = polyval(fit_coeff,time);
Dfit = polyval(Dfit_coeff,time);

[m,X] = max(Dfit);
b = fit(X)-m*X;
y_0=scaled_temp(1);
a = y_0-b;
L = a/m;
K=1.2/a
t_i=2*L
t_d=L/2

fit_tangent = m*time(1:round(end/3))+b;

plot(time,scaled_temp,'k.','markersize',8)
hold on
plot(time,fit,'b-');
plot(time(1:round(end/3)),fit_tangent,'r-');
```

Appendix 3b

```
% Michael Goodman
% 26 July 2018
% Skogestad PID step response tune w/ second-order + delay fit

clear all; close all; clc;

n = input('Number of File: ');
filename = strcat('TEMPS',num2str(n),'.TXT');
fileID = fopen(filename);
M = textscan(fileID,'%f %f %f %f %f','Delimiter',',\n');
fclose(fileID);

u_max = 1023;

time = (M{1,1}-M{1,1}(1))/1000;
process_variable = M{1,4};

model=@(parameter,t)u_max*parameter(1)*(1-parameter(3)*exp(-(t-parameter(2))/parameter↙
(3))/(parameter(3)-parameter(4))+parameter(4)*exp(-(t-parameter(2))/parameter(4))/↙
(parameter(3)-parameter(4))).*heaviside(t-parameter(2));

k_initial=(process_variable(end)-process_variable(1))/u_max;
%k_initial=process_variable(end)-process_variable(1);
theta_initial=time(find(process_variable>=(process_variable(1)+1),1,'first'));
tau_1_initial=time(find(process_variable>=0.63*process_variable(end),1,'first'));
tau_2_initial=1;

options=optimset('display','off');
parameter = lsqcurvefit(model,[k_initial,theta_initial,tau_1_initial,tau_2_initial],↙
time,process_variable-process_variable(1),[0,0,0,0],[1000,1000,1000,1000],options);
k = parameter(1)
theta = parameter(2)
tau_1 = parameter(3)
tau_2 = parameter(4)
if tau_1<=8*theta
    K_c=1/(2*k)*(tau_1+tau_2)/theta
    tau_I=tau_1+tau_2
    tau_D=tau_2/(1+tau_2/tau_1)
else
    K_c=1/(2*k)*tau_1/theta*(1+tau_2/(8*theta))
    tau_I=8*theta+tau_2
    tau_D=tau_2/(1+tau_2/(8*theta))
end

fit=model(parameter,time)+process_variable(1);

plot(time,process_variable,'k.','markersize',8)
hold on
plot(time,fit,'b-');
```

Bibliography:

Skogestad, Sigurd. (2001). Probably the best simple PID tuning rules in the world.

Astrom, K. J., & Hagglund, T. (1995). PID Controller: Theory, Design, and Tuning (2nd ed.).

Sontag, E. (1998). Mathematical Control Theory Deterministic Finite Dimensional Systems (2nd ed.).