Phys 39 Lab Report: PID Temperature Control

Mason Price

January 29, 2025

Abstract

Many systems require closed-loop control in which some input parameter is continually adjusted based on measurements so that a target value of an output parameter is reached in a reasonable amount of time. Here, we use Proportional-Integral-Derivative (PID) control for the temperature of a peltier. We find that for certain values of PID gain, target temperatures both above and below room temperature can be reached with little fluctuations after a period of 200 seconds or less. This system could be improved by incorporating a filter to smooth out noisy data in order to improve the reliability of the derivative term. An experimental setup like this can be used in applications requiring the control of temperature, such as gel electrophoresis in which the precise control of temperature for the gel is important. More generally, PID control is useful in many other fields, such as to steer a telescope and control the orientation of a rocket.

1 Introduction

PID is the most common algorithm used for control of many systems [1]. The idea is that if some input parameter is related to an output parameter, then we can repeatedly adjust the input until the output reaches a user-specified set value. Let u represent the *control* signal, i.e. the input parameter, and y be a related output parameter. Let r be the reference or *set* value. Then, the control *error* is defined [1] as

$$e := r - y. \tag{1}$$

Our goal is to minimize $|e|$. To accomplish this, we update the current value of the input to

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau)d\tau + k_d \frac{de}{dt}. \tag{2}$$

This equation includes three terms. First, the *proportional* term is

$$P = k_p e(t),$$

where k_p is the *proportional gain*. The gain determines how aggressive any changes are, and this term means we adjust the input by the current error up to some proportionality constant. In this way, if $y < r$ then $e > 0$ and so we increase the input u. Likewise, if $y > r$ then $e < 0$ and so we decrease the input u. Crucially, if e is very small, then our changes are very small. This term tells us the current information about the error [1].

Second, the *integral* term in Equation 2 is

$$I = k_i \int_0^t e(\tau)d\tau,$$

where k_i is the *integral gain*. This term includes the history of the error [1], because it is integrating the accumulated error over time. This term should remove any steady-state errors left by the P term alone, because any such steady-state errors will accumulate area between the set value r and the current value y.

The third term in Equation 2 is the derivative term,

$$D = k_d \frac{de}{dt},$$

where k_d is the *derivative gain*. This is predicting the future trajectory of the error by assuming that it will be approximately linear for small step sizes in t [1]. This term also crucially approaches zero as the error curve flattens out near zero.

In experiment, the output parameter y is the temperature, and our input parameter u is the average voltage.

2 Methods

2.1 Experimental setup

To control our circuit, we use an Arduino. The Arduino is connected to a thermistor in order to measure temperature, and a peltier thermo-electric cooler (TEC) to actively heat and cool. The Arduino is not powerful enough to run the TEC, so we use an H-bridge and power supply to boost the voltage being supplied to the TEC and to control its polarity (i.e. either cooling or heating). The full circuit is shown in Figure 1, and the individual components are described in the following subsections.

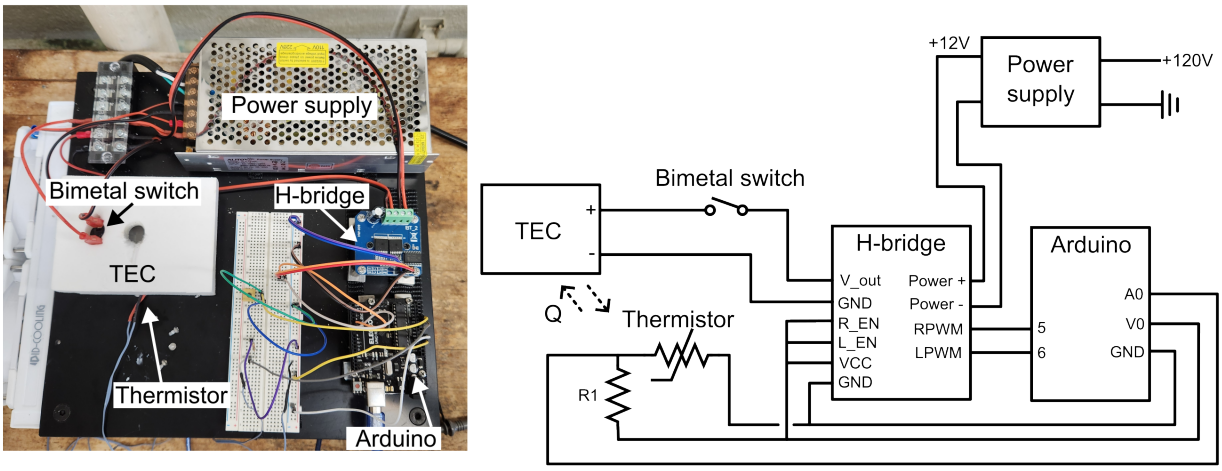

Figure 1: Full circuit diagram. (Left) Picture of experimental setup. (Right) Circuit diagram of experimental setup.

In the Arduino, we use digital pins 5 and 6 to connect to the H-bridge, and analog pin A0 to measure the voltage across the thermistor (Fig. 1). The power supply is plugged into a 120V outlet, and outputs a maximum voltage of 12V at a power of 120W. The TEC is put in thermal contact with the themistor using an aluminum block and thermal paste, and we try to minimize heat transferred to the environment using a block of styrofoam (Fig. 1). To prevent the TEC from overheating, we use a bimetal safety switch, which is electrically in serial with the TEC, and physically in thermal contact with the TEC (Fig. 1). The bimetal safety switch will turn off if the temperature goes too high (about 320K). Finally, the Arduino talks to a MATLAB app using the serial port, which is how the user specifies the target temperature and values of gain to use.

2.2 Thermistor

We use a thermistor to measure temperature. A thermistor is a variable resistor whose value of resistance depends on the temperature of its environment. To calculate this, we first measure the voltage across the thermistor, V_T, for a given reference resistor $R_1 = 3.2k\Omega$ which is put in serial with the thermistor. We derive a formula for the resistance of the thermistor R_T as follows.

Notice that

$$V_0 = I(R_1 + R_T),$$

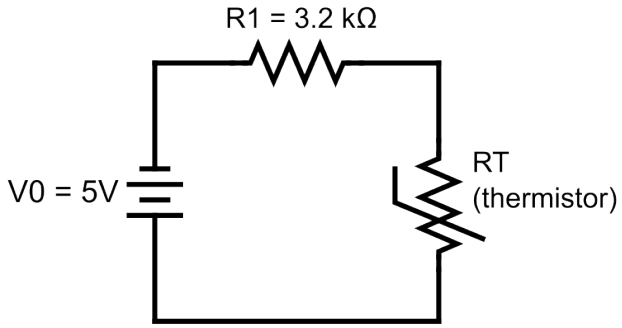

Figure 2: Thermistor circuit. We use an initial voltage of $V_0 = 5V$ from the Arduino and a reference resistance of $R_1 = 3.2\ k\Omega$.

because the current is consistent throughout the circuit. Notice also that

$$I = \frac{V_T}{R_T},$$

which is given by Ohm's Law across the thermistor. Then substituting this expression for I into the equation above and solving for R_T, we see

$$\begin{aligned} V_0 &= \frac{V_T}{R_T}(R_1 + R_T) \\ \iff V_0 &= V_T\left(\frac{R_1}{R_T} + 1\right) \\ \iff \frac{V_0}{V_T} - 1 &= \frac{R_1}{R_T} \\ R_T &= R_1\left(\frac{V_T}{V_0 - V_T}\right) \end{aligned} \tag{3}$$

Therefore, by measuring the voltage V_T across the thermistor, we can calculate its resistance R_T. The voltage V_0 is supplied by the Arduino in our circuit.

Next, to convert the resistance of the thermistor into temperature, we use the Steinhart–Hart equation [4]

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{B}\ln\frac{R_T}{R_0}. \tag{4}$$

Here, T_0 is a reference temperature, R_0 is the default resistance of the thermistor at the temperature T_0, and B is a parameter that depends on the specific thermistor being used. Our thermistor (part number B57861S0104F040) has a default resistance of $R_0 = 100\ k\Omega$ at a temperature of $T_0 = 25°C$, and a value of $B = 4540°C$. Therefore with these values, we can use Equations 3 and 4 to calculate the temperature of the thermistor just by measuring the voltage across the thermistor.

2.3 TEC

A TEC works on the Peltier effect, which is that if a current is passed through two dissimilar electric conductor, then a heat gradient will form between the two materials (i.e. heat will flow from one to the other) [2]. In practice these are made using P- and N-type semiconductors 3, and the direction of the current (i.e. polarity of the voltage) will determine the direction of heat flow (i.e. whether it is heating or cooling a given surface) [2].

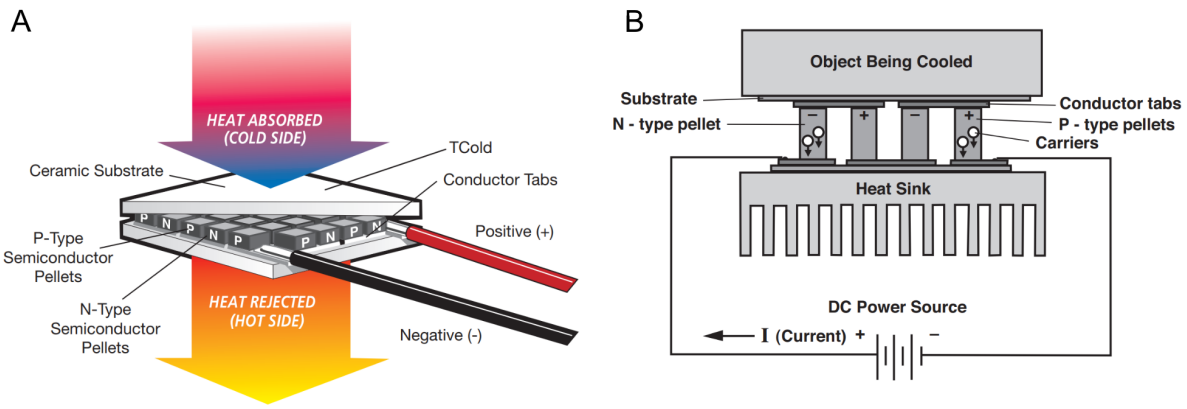

Figure 3: Thermo-electric cooler, figure adapted from [2]. (A) Illustration of heat flow for a given current. Note the arrangement of P- and N-type semiconductors. (B) Illustration of charged pellots travelling as a result of the current.

In our experiment, we use a heat sink and put an aluminum block (with a hole drilled to fit a thermistor) on the top of the TEC, as in Figure 3B.

2.4 H-bridge

An H-bridge is used to control polarity [3]. The idea is that if you open and close switches on either side of an output (say a motor), then you can control the direction in which current will flow. For example, if you close S1 and S4 in Figure 4, the current will flow counter-clockwise and the polarity will be positive, while if you instead only close switches S2 and S3, the current will flow in the opposite direction and the voltage will be negative. This is therefore an effective method of controlling the direction of current in the TEC, and by adding a power supply we can amplify the voltage that is sent to the TEC.

In our experiment, we used an Integrated Circuit H-bridge (Part number IBT_2). This connects to the Arduino digital pins 5 and 6 using pins labeled RPWM and LPWM on the H-bridge, respectively (Fig. 1). Depending on which pin receives a higher voltage, this H-bridge will swap the polarity. If RPWM $>$ LPWM, the TEC will heat, and if RPWM $<$ LPWM the TEC will cool. In practice, it is easier to set LPWM$= 0$ if we are in heating mode, and RPWM $= 0$ if we are in cooling mode.

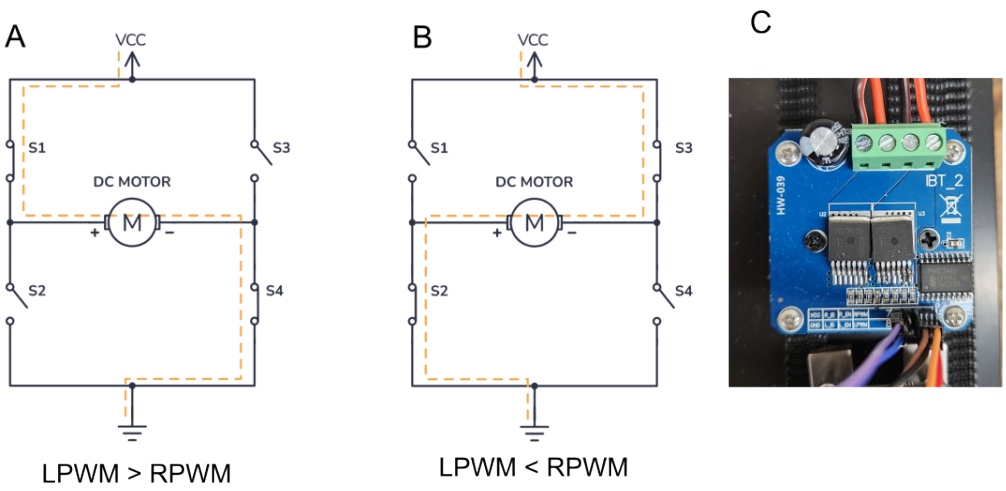

Figure 4: H-bridge. Adapted from [3]. (A & B) The polarity of an H-bridge will swap depending on which switches are open and closed. (A) S1 and S4 are closed, and the voltage is positive with respect to ground. (B) S2 and S3 are closed, and the voltage is negative. (C) Picture of H-bridge used in experiment.

2.5 Pulse-width modulation (PWM)

To vary the effective voltage output from the Arduino, we use Pulse-width modulation. The idea is that instead of continuously varying the value of voltage, as in an analog system, we can effectively vary the *average* power using a digital wave train of pulses whose widths vary (i.e., a sequence of pulses that are either on or off, where "on" corresponds to 5V). One full period for a pulse train is called the *duty cycle*, and by having pulses on for only a percentage of the duty cycle, the average voltage over time can be continuously varied from 0V to 5V. For example, if we use 20% of the duty cycle, then the effective voltage will be $0.2 \cdot 5V = 0.1V$ (Fig. 5).

In Arduino, a PWM value of 255 corresponds to using 100% of the duty cycle, and a PWM value of 0 corresponds to 0% (so that anything in-between 0 and 255 lies between 0 and 100%.)

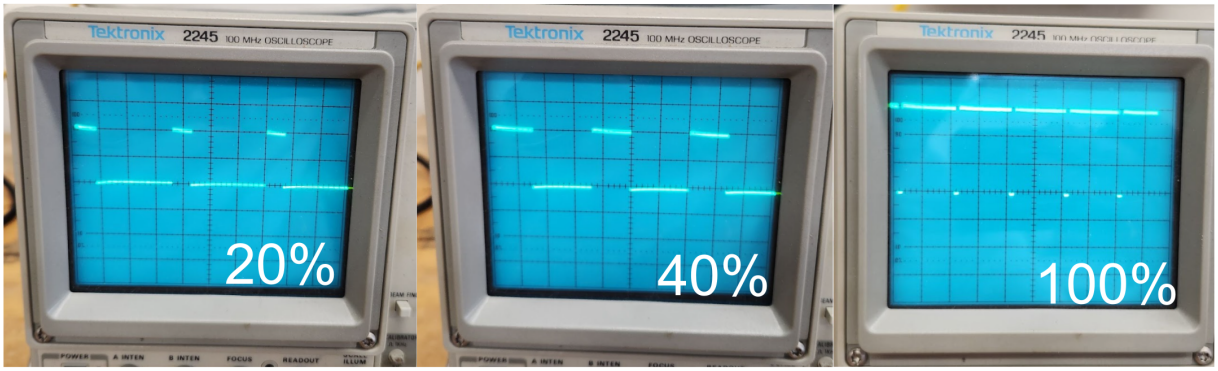

Figure 5: Oscilloscope representation of PWM. The scale is normalized so that 5V is 2 divisions above the zero line. Increasing the percentage of the duty cycle increases the portion that is "on".

3 Results

3.1 Characterizing PWM and temperature

To characterize the relationship between the value of PWM (from 0 to 255, where 255 represents 100% of the duty cycle), and the output temperature of the Peltier in steady-state, we varied the value of PWM in both heating and cooling modes, and waited for the temperature to stabilize (Fig. 6). By fitting a line to our measurements, and recognizing that $dT/d\text{PWM}$ (i.e. the change in temperature per change in PWM) is the slope of this line (call it C), we find that in a cooling mode,

$$C_{\text{cool}} = \frac{dT}{d\text{PWM}_{\text{cool}}} = -0.17. \tag{5}$$

For heating mode, we find

$$C_{\text{heat}} = \frac{dT}{d\text{PWM}_{\text{heat}}} = 0.39. \tag{6}$$

Notably, for a PWM of 0, we measure a room temperature of $285K$.

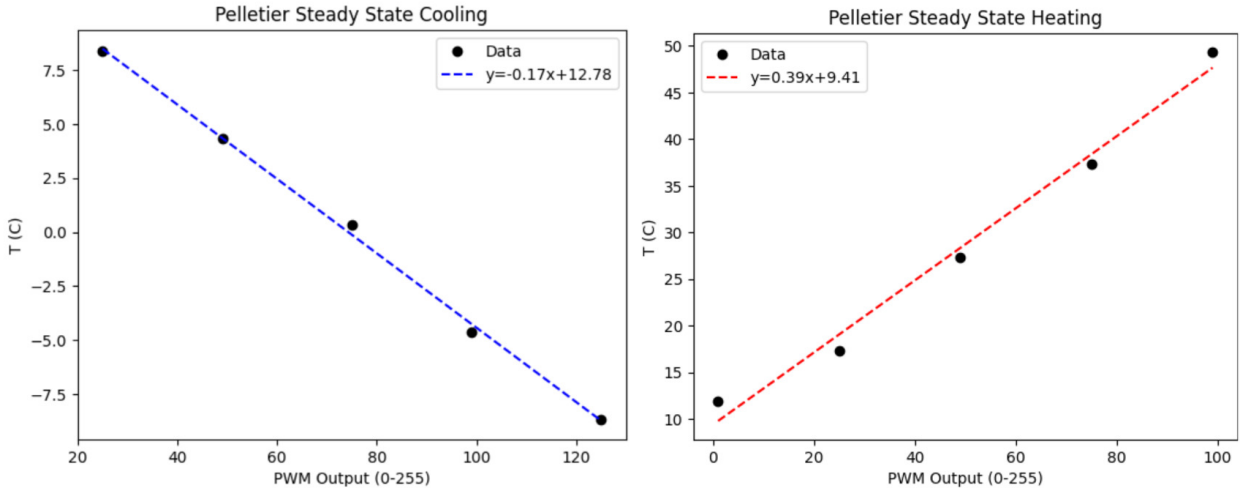

Figure 6: Steady state temperatures of Peltier for different PWM values in both cooling and heating modes.

3.2 MATLAB app

To allow for a user-specified set temperature, we created an app in MATLAB. This app communicates to the Arduino via the serial port. The graphical user interface (GUI) for this app is shown in Figure 7. One sets the target temperature using a knob from 260 to 320 K, and enters values of gain for the P, I and D terms (for the PID control.) Then, the Arduino will print to MATLAB the current temperature (as calculated in section 2.2). Based on this, the MATLAB code will implement the PID control algorithm (section 1), and it will calculate a PWM value from -255 to 255. For heating, we set the PWM to be from 0 to 255, and for cooling we use a PWM from 0 to -255 (so that -255 is 100% of the duty cycle in the cooling mode).

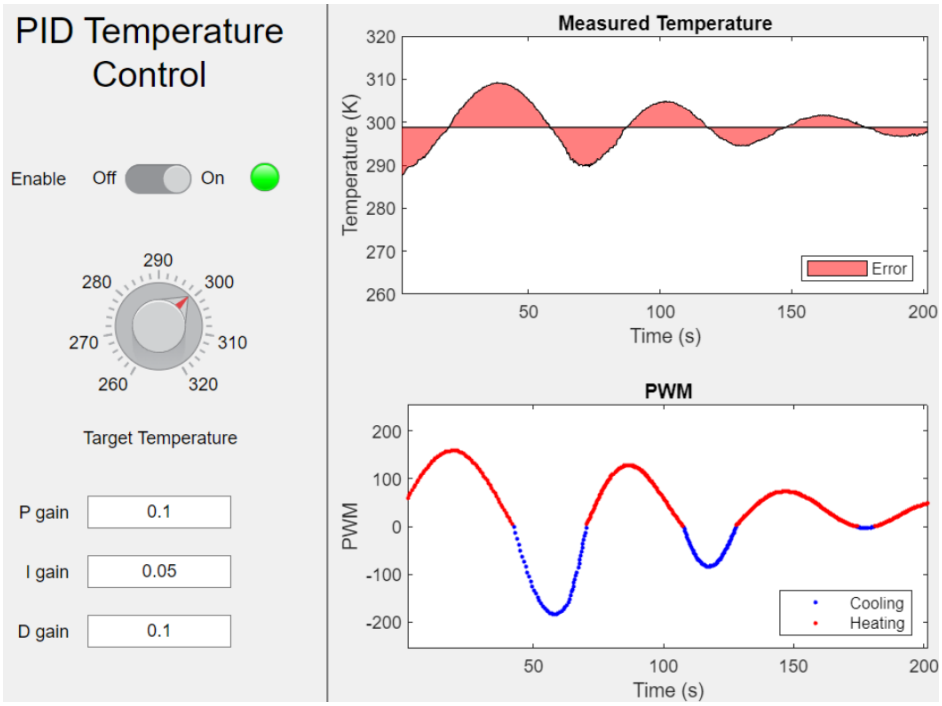

Figure 7: MATLAB app GUI. User specified values are on the left-column. The top right plot shows the measured temperature, with the area between the target temperature and mesaured temperature shaded red. The bottom right plot shows the PWM as calculated using PID. Whenever the PWM is negative, the system is cooling, and when the PWM is positive the system is heating. Note that the PWM curves for heating and cooling look different because we use different conversion factors (i.e. we are more aggresive for cooling because the TEC is slower at cooling).

3.3 PID temperature control

Before we can directly implement PID, we must convert from an error in terms of temperature into something in terms of PWM. For this, since we are considering *changes* in temperature and PWM, we can use the conversion factors given by equations 5 and 6, so that

$$C = \frac{\Delta T}{\Delta \text{PWM}}$$

where C is either $C_{\text{cool}} = 0.17$ or $C_{\text{heat}} = 0.39$ depending on the mode that we are in (where the sign is handled based on this mode).

At $t = 0$, we set the PWM value to be that which corresponds to the target temperature in steady state (e.g. if the target temperature is greater than room temperature, then we solve the inverse of $y = 0.39x + 9.4$, convert it into PWM and start there). Then, once we start measuring, we set

$$\text{PWM} = \frac{u_T}{C}$$

where

$$u_T = k_p e(t) + k_i \int_0^t e(\tau) d\tau + k_d \frac{de}{dt} \tag{7}$$

Here, we calculate the integral numerically by summing over the values of $e(t)$ multiplied by the corresponding time steps dt. Similarly, we calculate the derivative numerically as the slope between consecutive data points.

To test the effectiveness of the P term alone vs P and I (PI) vs P, I and D (PID), we did three tests. First, the integral and derivative gains were set to zero, ($k_i = k_d = 0$). Second, only the derivative gain was set to zero ($k_d = 0$), while the integral gain was fine-tuned through trial-and-error (keeping k_p fixed.) Third,

we performed a test in which all three values of gain are non-zero, and k_d is fine-tuned as well. The results of these three tests are shown in Figure 8. As expected, in P alone there is a steady-state error between the measured temperature and set temperature, because there is no guarantee that the P term $P = k_p e$ will be the input that corresponds to the target temperature. Notably however, with PI this steady-state error goes away, because we accumulate area between the target and measure temperatures (shaded region in Fig. 8), which then offsets the input until the error converges to zero. Additionally, the full PID test seems to have the best results, as it converged in only 70 seconds.

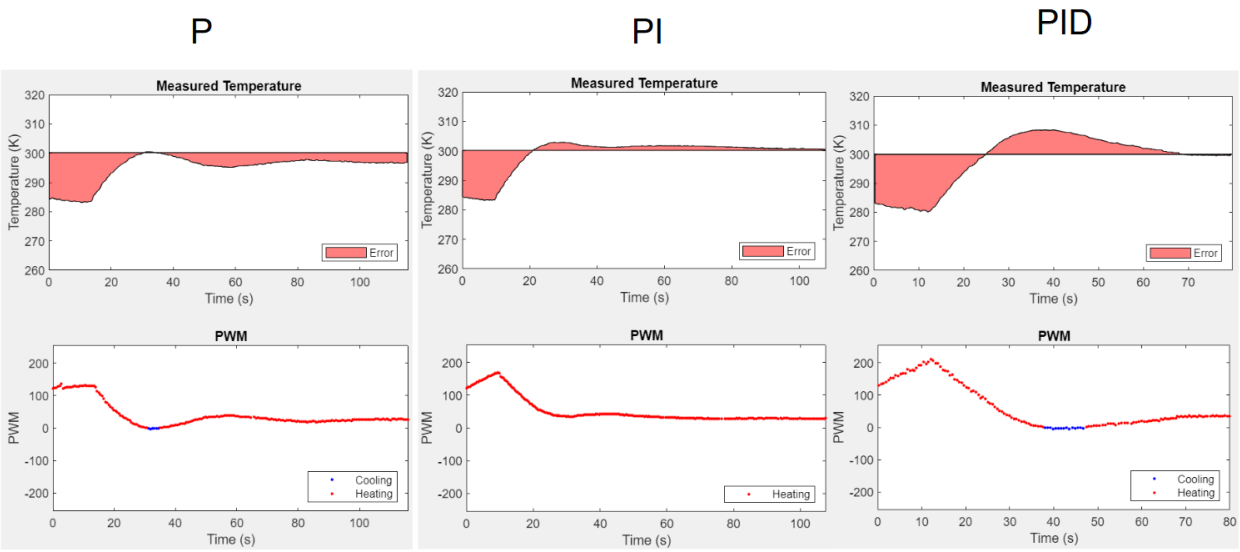

Figure 8: Test of PID terms. The red shaded region is the area between the measured temperature and target temperature. For P, the gains were $k_p = 3$, $k_i = 0$, $k_d = 0$. For PI, the gains were $k_p = 3$ $k_i = 0.1$ and $k_d = 0$. For PID, the gains were $k_p = 3$, $k_i = 0.1$ and $k_i = 1$.

4 Discussion

Although we were able to reach target temperatures within a reasonable amount of time using PID here (on the order of 100 seconds), there are several ways in which this approach could be improved. First, the main problem I see is that we do the PID calculations in Matlab, so any other computations or processes being run locally on the computer will affect how quickly it can send the new value of PWM to the Arduino. For example, if we run the app and then do something else on the computer, the data points will become significantly spaced out, and the effectiveness of the PID is notably changed. This is also true for additional processes in the Matlab code, such as fancy formatting in the plots or expensive tasks such as manipulating the array of time values. With our setup, (as in Fig. 8), we observed the update time step dt is roughly 0.1 s. Therefore a much better solution would be to perform all PID calculations on the Arduino itself (and only send the user-specified target temperature from the Matlab app), so that the updates are faster and decoupled from other computations on the computer running Matlab. This would explain why the PID was very slow in experiment.

Secondly, including a filter before calculating the derivative to smooth out the data would significantly improve the reliability of the D term (and could allow one to increase the D gain). For example, if one has very noisy data, then even if the overall trend of the data is upwards, there will be many examples in which the data happens to go downwards a little and then since we take the slope to calculate the derivative, many of the contributions from the D term will be in the wrong direction [1]. Therefore, using a filter to smooth the data would help. For example, the Sovitsky-Golay filter, in which one fits an m^{th} order polynomial to $m + 1$ data points could be an effective solution.

5 Conclusion

Here, we implemented a PID algorithm to control temperature in an experimental setup using a Peltier, Arduino and Matlab app. The thermistor circuit allowed us to measure temperature by measuring only the voltage across the thermistor. To reach a target temperature, we calculate a PWM in Matlab and send it to the Arduino. Then, the Arduino uses an H-bridge and power supply to run a Peltier, which is in thermal contact with the thermistor. This technique effectively reached a user-prescribed temperature over a period of about 100 seconds, which could be improved even further by running the PID calculations on the Arduino instead of Matlab and filtering the data. This system could be directly used in many experiments where the precise control of temperature is necessary. For example, gel electrophoresis requires precise control of temperature for both the gel itself and the buffer surrounding the gel. One could therefore combine two PID controls as shown here to run an automated gel electrophoresis experiment for an indefinite amount of time (in principle). This same idea of using PID as a control algorithm could also be extended to completely different applications, such as pointing a telescope so that it tracks a certain celestial object, or to control the orientation of a rocket on descent to make it land upright. All of these cases require the precise control and closed-loop feedback from sensor to output, which are precisely the principles learned here.

References

[1] Karl Johan Åström and Richard Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2021.

[2] Tellurex Corporation. "An Introduction to Thermoelectrics". In: (2006).

[3] Øyvind Nydal Dahl. *What Is an H-Bridge?* 2018. URL: `https://www.build-electronic-circuits.com/h-bridge/`.

[4] John S Steinhart and Stanley R Hart. "Calibration curves for thermistors". In: *Deep sea research and oceanographic abstracts*. Vol. 15. 4. Elsevier. 1968, pp. 497–503.

6 Supplementary Information

6.1 MATLAB code

```
classdef appPID < matlab.apps.AppBase

    % Properties that correspond to app components
    properties (Access = public)
        UIFigure                    matlab.ui.Figure
        GridLayout                  matlab.ui.container.GridLayout
        LeftPanel                   matlab.ui.container.Panel
        DgainEditField              matlab.ui.control.EditField
        DgainEditFieldLabel         matlab.ui.control.Label
        IgainEditField              matlab.ui.control.EditField
        IgainEditFieldLabel         matlab.ui.control.Label
        PgainEditField              matlab.ui.control.EditField
        PgainEditFieldLabel         matlab.ui.control.Label
        TargetTemperatureKnob       matlab.ui.control.Knob
        TargetTemperatureKnobLabel  matlab.ui.control.Label
        Lamp                        matlab.ui.control.Lamp
        EnableSwitch                matlab.ui.control.Switch
        EnableLabel                 matlab.ui.control.Label
        PIDTemperatureControlLabel  matlab.ui.control.Label
        RightPanel                  matlab.ui.container.Panel
        UIAxes2                     matlab.ui.control.UIAxes
        UIAxes                      matlab.ui.control.UIAxes
    end

    % Properties that correspond to apps with auto-reflow
    properties (Access = private)
        onePanelWidth = 576;
    end

    properties (Access = private)
        arduino % Arduino object for the app to speak to the Arduino over the serial port
        PWMvalues = []; % list to store PWM values
        tempValues = []; % list of temperature values
        times = []; % list of times
        PWM = 0; % current PWM value
        errValues = []; % store all error values
        roomTemp = 285; % define the ambient room temperature

    end
    methods (Access = private)

        function updatePlots(app)
            % Receive the temperature value from the comport connected to Arduino
            tempValue = fscanf(app.arduino, '%f'); % Read the temperature data as a float
            targetTemp = app.TargetTemperatureKnob.Value; % Retrieve value of target
    temperature
            err = targetTemp-tempValue; % Find the error
            app.errValues = [app.errValues, err]; % update error vector
            app.times = [app.times, toc]; % update time vector

            % Adjust the conversion factor we're using based on whether we
            % are cooling or heating (use most recent value).
            if app.PWM > 0
                conversionFactor = 0.39;
            else
                conversionFactor = 0.17;
            end

            % Retrieve the gain values from the UI interface
            Pgain = str2double(app.PgainEditField.Value);
            Igain = str2double(app.IgainEditField.Value);
            Dgain = str2double(app.DgainEditField.Value);

```

```
            % Calculate the P term
            Pterm = err*Pgain/conversionFactor;

            % Calculate the I term
            numTerms = 5;
            y = app.errValues(find(app.errValues, numTerms, 'last'));
            dx = [0, diff(app.times(find(app.times, numTerms, 'last')))];
            disp("y")
            disp(y)
            disp("dx")
            disp(dx)
            integral = sum(y.*dx); % Take the integral over the errors vs time.
            Iterm = integral*Igain/conversionFactor;

            % Calculate the D term
            recentErrs = app.errValues(find(app.errValues, 2, 'last')); % Find the last two
    error values
            recentTimes = app.times(find(app.times, 2, 'last')); % Find the times of these
    values
            %if length(app.times) > 2 % Don't take the derivative if the list is too short.
            if toc > 2
                derivative = (recentErrs(2) - recentErrs(1))/(recentTimes(2) - recentTimes
    (1));
                Dterm = derivative*Dgain/conversionFactor;
            else
                Dterm = 0;
            end


            % Calculate the u term, i.e. the change in PWM = P + I + D
            dPWM = Pterm + Iterm + Dterm;
            PWMvalue = app.PWM + dPWM; % Add this change in PWM to the current value

            % Handle saturation edge-case
            if PWMvalue > 255
                PWMvalue = 255;
            elseif PWMvalue < -255
                PWMvalue = -255;
            end

            % Update the current app's value of PWM and list of temperatures
            app.PWM = PWMvalue;
            app.tempValues = [app.tempValues, tempValue];

            %% Display relevent information in the terminal. This useful for  trouble-
    shooting
            disp("Temperature:")
            disp(tempValue)
            disp("dPWM")
            disp(dPWM)
            disp("Final PWM")
            disp(PWMvalue)
            disp("P term:")
            disp(Pterm)
            disp("I term:")
            disp(Iterm)
            disp("D term")
            disp(Dterm)

            %% Plot the PWM
            fprintf(app.arduino, int2str(int32(PWMvalue)));
            app.PWMvalues = [app.PWMvalues, PWMvalue];
            app.UIAxes2;

            % Clear UIAxes2 to avoid overlapping issues
            cla(app.UIAxes2);
            hold(app.UIAxes2, 'on');

```

```
            plot(app.UIAxes2, app.times(app.PWMvalues<0), app.PWMvalues(app.PWMvalues<0), 'b.', 'DisplayName', 'Cooling')
            plot(app.UIAxes2, app.times(app.PWMvalues>0), app.PWMvalues(app.PWMvalues>0), 'r.', 'DisplayName', 'Heating')
            % Adjust the axis scales
            if toc < 300
                xlim(app.UIAxes2, [0, toc])
            else
                xlim(app.UIAxes2, [toc-300, toc])
            end
            ylim(app.UIAxes2, [-255, 255])

            % Add a legend
            legend(app.UIAxes2, 'Location','southeast');

            % Draw the plot
            drawnow;

            %% Plot the temperature
            app.UIAxes;

            % Plot measured temperature data points
%             plot(app.UIAxes, app.times, app.tempValues, 'DisplayName', 'Data')

            % Shade the area between the target and measure temperature
            if size(app.times, 2) > 2
                % Define y-coordinates of the target temperature line
                y2 = targetTemp*ones(1, size(app.errValues, 2));
                x2 = [app.times, fliplr(app.times)];
                inBetween = [app.tempValues, fliplr(y2)];
                fill(app.UIAxes, x2, inBetween, 'r', 'FaceAlpha', 0.5, 'DisplayName', 'Error');
            end

            % Adjust the axis scales
            if toc < 300
                xlim(app.UIAxes, [0, toc])
            else
                xlim(app.UIAxes, [toc-300, toc])
            end
            ylim(app.UIAxes, [260, 320])

            % Add a legend
            legend(app.UIAxes, 'Location','southeast');

            % Draw the plot
            drawnow;

        end
    end

    % Callbacks that handle component events
    methods (Access = private)

        % Code that executes after component creation
        function startupFcn(app)
            app.arduino = serial('COM4', 'BaudRate', 9600);
            fopen(app.arduino);  % Open the serial port
        end

        % Value changed function: EnableSwitch
        function EnableSwitchValueChanged(app, event)
            value = app.EnableSwitch.Value;
            if value == "On"
                %fprintf(app.arduino, "On\n")
                app.Lamp.Color = "green";
                app.PWMvalues = [];
                app.times = [];
```

```
                app.tempValues = [];
                tic
            else
                %fprintf(app.arduino, "Off\n")
                app.Lamp.Color = [0.50,0.50,0.50];
            end
            while value == "On"
                updatePlots(app);
                value = app.EnableSwitch.Value;
                %pause(1.5);
            end
        end

        % Callback function
        function SwitchModeValueChanged(app, event)

        end

        % Close request function: UIFigure
        function UIFigureCloseRequest(app, event)
            fclose(app.arduino); % Close the serial port
            delete(app.arduino); % Delete the serial object
            clear app.arduino;   % Clear the variable
            delete(app)

        end

        % Value changed function: TargetTemperatureKnob
        function TargetTemperatureKnobValueChanged(app, event)
            value = app.TargetTemperatureKnob.Value;
            % If the temp knob is adjusted, reset everything to 0/empty
            app.PWMvalues = []; % list to store PWM values
            app.tempValues = []; % list of temperature values
            app.times = []; % list of times
            app.PWM = 0; % current PWM value
            app.errValues = []; % store all error values

            targetTemp = app.TargetTemperatureKnob.Value;
            if targetTemp > app.roomTemp
                % Set initial PWM corresponding to target temp
                app.PWM = (targetTemp-282.56)/0.39;
            else
                % Set initial PWM corresponding to target temp
                app.PWM = -(targetTemp-285.93)/0.17;
            end

            % and restart the timer
            tic;
        end

        % Changes arrangement of the app based on UIFigure width
        function updateAppLayout(app, event)
            currentFigureWidth = app.UIFigure.Position(3);
            if(currentFigureWidth <= app.onePanelWidth)
                % Change to a 2x1 grid
                app.GridLayout.RowHeight = {480, 480};
                app.GridLayout.ColumnWidth = {'1x'};
                app.RightPanel.Layout.Row = 2;
                app.RightPanel.Layout.Column = 1;
            else
                % Change to a 1x2 grid
                app.GridLayout.RowHeight = {'1x'};
                app.GridLayout.ColumnWidth = {224, '1x'};
                app.RightPanel.Layout.Row = 1;
                app.RightPanel.Layout.Column = 2;
            end
        end
    end
```

```
    % Component initialization
    methods (Access = private)

        % Create UIFigure and components
        function createComponents(app)

            % Create UIFigure and hide until all components are created
            app.UIFigure = uifigure('Visible', 'off');
            app.UIFigure.AutoResizeChildren = 'off';
            app.UIFigure.Position = [100 100 640 480];
            app.UIFigure.Name = 'MATLAB App';
            app.UIFigure.CloseRequestFcn = createCallbackFcn(app, @UIFigureCloseRequest,
    true);
            app.UIFigure.SizeChangedFcn = createCallbackFcn(app, @updateAppLayout, true);

            % Create GridLayout
            app.GridLayout = uigridlayout(app.UIFigure);
            app.GridLayout.ColumnWidth = {224, '1x'};
            app.GridLayout.RowHeight = {'1x'};
            app.GridLayout.ColumnSpacing = 0;
            app.GridLayout.RowSpacing = 0;
            app.GridLayout.Padding = [0 0 0 0];
            app.GridLayout.Scrollable = 'on';

            % Create LeftPanel
            app.LeftPanel = uipanel(app.GridLayout);
            app.LeftPanel.Layout.Row = 1;
            app.LeftPanel.Layout.Column = 1;

            % Create PIDTemperatureControlLabel
            app.PIDTemperatureControlLabel = uilabel(app.LeftPanel);
            app.PIDTemperatureControlLabel.HorizontalAlignment = 'center';
            app.PIDTemperatureControlLabel.FontSize = 24;
            app.PIDTemperatureControlLabel.Position = [20 412 186 59];
            app.PIDTemperatureControlLabel.Text = {'PID Temperature'; 'Control'};

            % Create EnableLabel
            app.EnableLabel = uilabel(app.LeftPanel);
            app.EnableLabel.Position = [17 346 42 22];
            app.EnableLabel.Text = 'Enable';

            % Create EnableSwitch
            app.EnableSwitch = uiswitch(app.LeftPanel, 'slider');
            app.EnableSwitch.ValueChangedFcn = createCallbackFcn(app,
    @EnableSwitchValueChanged, true);
            app.EnableSwitch.Position = [101 347 45 20];

            % Create Lamp
            app.Lamp = uilamp(app.LeftPanel);
            app.Lamp.Position = [190 348 20 20];
            app.Lamp.Color = [0.502 0.502 0.502];

            % Create TargetTemperatureKnobLabel
            app.TargetTemperatureKnobLabel = uilabel(app.LeftPanel);
            app.TargetTemperatureKnobLabel.HorizontalAlignment = 'center';
            app.TargetTemperatureKnobLabel.Position = [71 172 109 22];
            app.TargetTemperatureKnobLabel.Text = 'Target Temperature';

            % Create TargetTemperatureKnob
            app.TargetTemperatureKnob = uiknob(app.LeftPanel, 'continuous');
            app.TargetTemperatureKnob.Limits = [260 320];
            app.TargetTemperatureKnob.ValueChangedFcn = createCallbackFcn(app,
    @TargetTemperatureKnobValueChanged, true);
            app.TargetTemperatureKnob.Position = [94 228 60 60];
            app.TargetTemperatureKnob.Value = 285;

            % Create PgainEditFieldLabel
```

```
            app.PgainEditFieldLabel = uilabel(app.LeftPanel);
            app.PgainEditFieldLabel.HorizontalAlignment = 'right';
            app.PgainEditFieldLabel.Position = [20 122 39 22];
            app.PgainEditFieldLabel.Text = 'P gain';

            % Create PgainEditField
            app.PgainEditField = uieditfield(app.LeftPanel, 'text');
            app.PgainEditField.HorizontalAlignment = 'center';
            app.PgainEditField.Position = [74 122 100 22];
            app.PgainEditField.Value = '0.1';

            % Create IgainEditFieldLabel
            app.IgainEditFieldLabel = uilabel(app.LeftPanel);
            app.IgainEditFieldLabel.HorizontalAlignment = 'right';
            app.IgainEditFieldLabel.Position = [25 82 34 22];
            app.IgainEditFieldLabel.Text = 'I gain';

            % Create IgainEditField
            app.IgainEditField = uieditfield(app.LeftPanel, 'text');
            app.IgainEditField.HorizontalAlignment = 'center';
            app.IgainEditField.Position = [74 82 100 22];
            app.IgainEditField.Value = '0.05';

            % Create DgainEditFieldLabel
            app.DgainEditFieldLabel = uilabel(app.LeftPanel);
            app.DgainEditFieldLabel.HorizontalAlignment = 'right';
            app.DgainEditFieldLabel.Position = [19 42 40 22];
            app.DgainEditFieldLabel.Text = 'D gain';

            % Create DgainEditField
            app.DgainEditField = uieditfield(app.LeftPanel, 'text');
            app.DgainEditField.HorizontalAlignment = 'center';
            app.DgainEditField.Position = [74 42 100 22];
            app.DgainEditField.Value = '0.1';

            % Create RightPanel
            app.RightPanel = uipanel(app.GridLayout);
            app.RightPanel.Layout.Row = 1;
            app.RightPanel.Layout.Column = 2;

            % Create UIAxes
            app.UIAxes = uiaxes(app.RightPanel);
            title(app.UIAxes, 'Measured Temperature')
            xlabel(app.UIAxes, 'Time (s)')
            ylabel(app.UIAxes, 'Temperature (K)')
            zlabel(app.UIAxes, 'Z')
            app.UIAxes.Box = 'on';
            app.UIAxes.Position = [6 244 405 228];

            % Create UIAxes2
            app.UIAxes2 = uiaxes(app.RightPanel);
            title(app.UIAxes2, 'PWM')
            xlabel(app.UIAxes2, 'Time (s)')
            ylabel(app.UIAxes2, 'PWM')
            zlabel(app.UIAxes2, 'Z')
            app.UIAxes2.Box = 'on';
            app.UIAxes2.Position = [6 6 405 219];

            % Show the figure after all components are created
            app.UIFigure.Visible = 'on';
        end
    end

    % App creation and deletion
    methods (Access = public)

        % Construct app
        function app = appPID
```

```
            % Create UIFigure and components
            createComponents(app)

            % Register the app with App Designer
            registerApp(app, app.UIFigure)

            % Execute the startup function
            runStartupFcn(app, @startupFcn)

            if nargout == 0
                clear app
            end
        end

        % Code that executes before app deletion
        function delete(app)

            % Delete UIFigure when app is deleted
            delete(app.UIFigure)
        end
    end
end
```

Listing 1: MATLAB Code

6.2 Arduino Code

```
int TEMP_PIN = A0; // pin used to read the voltage across the thermistor
long R0 = 100000;
float R1 = 1600;
float T0 = 298;
float V0 = 5;
float B = 4540;

String mode = "Cool";
int PWMvalue = 0;

int PWM_pin_cold = 6;
int PWM_pin_hot = 5;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  while(!Serial){
    ;
  }
}

void loop() {
    float sensorValue = 0;
    for (int i=0; i<100; i++){
       sensorValue += analogRead(TEMP_PIN);
    }
    sensorValue = sensorValue/100;

    // convert to voltage
    float voltageMeasured = sensorValue*5/1023;
    float RT = R1*(V0/(V0-voltageMeasured) - 1);
    float Tinv = 1/T0 + 1/B*log(RT/R0);
    float T = 1/Tinv;
    Serial.println(T);

      if (T<320){
        String readValue = Serial.readStringUntil('\n');
        if (readValue.startsWith("-")) {
```

```
            sscanf(readValue.c_str(), "-%d", &PWMvalue);
            mode = "Cool";
          } else {
            sscanf(readValue.c_str(), "%d", &PWMvalue);
            mode = "Heat";
          }
          if (mode == "Cool") {
            analogWrite(PWM_pin_cold, PWMvalue);
            analogWrite(PWM_pin_hot, 0);
          } else {
            analogWrite(PWM_pin_hot, PWMvalue);
            analogWrite(PWM_pin_cold, 0);
          }

        } else {
          analogWrite(PWM_pin_cold, 0);
          analogWrite(PWM_pin_hot, 0);
        }
}
```

Listing 2: Arduino Code