PID Temperature Controller Manual

The PID Temperature Controller consists of hardware and software components. The hardware, referred to as the PIDTC, acts as a controller for a heating/cooling element, and the software is used to interact with the PIDTC and change the control parameters. Once the parameters are tuned, the device will be able to monitor and control the temperature of some small system (a Fabry-Perot interferometer, for example). It is controlled by an Arduino MegaADK powered via USB, and contains some additional circuitry (op-amps included) that is powered by a +/- 5V power supply.

The PIDTC inputs include the +/- 5V supply, the USB for the Arduino, and a TH10K thermistor used to measure the temperature of the system that will monitored. The only output is the power going to the heating/cooling element (a TEC, or thermoelectric-cooler).

The program is used to change the control parameters (P [proportional] parameter, I [integral] parameter, D [derivative] parameter, and the set-temperature) and get a readout of the temperature of the system as it is measured by the device. Once the parameters are properly tuned, the PIDTC can be disconnected from the computer. When it is powered, it will work to match the temperature measured via the TH10K with the set-temperature entered by the user.

Contents

Getting Started……………………………………………………………………………3
Installing Pythics…………………………………………………………………3
Connecting to the PIDTC………………………………………………………3
Powering the Hardware…………………………………………………………4
Heat-sinking the TEC……………………………………………………………4
Using the PID Interface…………………………………………………………………5
Connecting to the PIDTC's Arduino……………………………………………5
Changing Control Parameters……………………………………………………6
Saving Control Parameters………………………………………………………6
Displaying the System Temperature……………………………………………7
Disconnecting the PIDTC…………………………………………………………7
Further Reading/Troubleshooting………………………………………………………8
Python Code………………………………………………………………………8
Arduino Sketch……………………………………………………………………9
Circuit………………………………………………………………………………9
Appendix…………………………………………………………………………………11
Arduino Sketch……………………………………………………………………11
PID Interface Python Code………………………………………………………14
PID Interface Pythics HTML Code………………………………………………23
Circuit Diagram……………………………………………………………………28

Getting Started

Installing Pythics

The program is written in Pythics and it must be opened using Pythics. If Pythics has not been installed, an install can be found at https://code.google.com/p/pythics/downloads/list. It is recommended to download version number 5.0 since it is guaranteed that this program will work with Pythics 5.0.

A Python package used for serial communication, `serial`, is needed to use this program. To check if the package is present in the Python directory, either:

A) Find the `Python27` folder. Go to `Python27 > Lib > site-packages` and look for a folder named `serial`. If the folder is there, the package is installed.

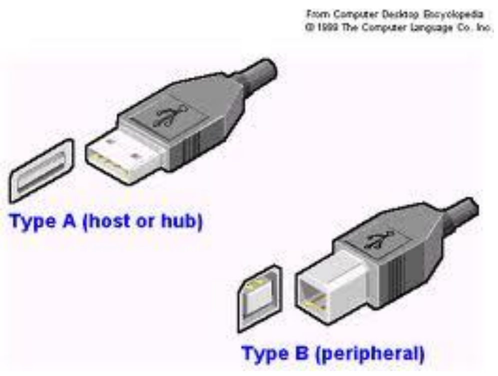
B) Run the Python interpreter by opening the Command prompt. Type in `python` and hit Enter. The Python interpreter will open. Enter `import serial`. If no error messages come up, then the package is installed.

If the `serial` package is not installed it can found here: http://sourceforge.net/projects/pyserial/files/pyserial/2.5/. Download and run the `.exe` file to install the package.

Once Pythics is installed and the `serial` library is in the Python directory, click the **Pythics** shortcut that should have been created on the desktop by the install. A black command prompt screen will open as well as a grey Pythics window. In the Pythics window click **File > Open** and select `PID_interface.html` and open the file. The program is now loaded.

Connecting to the PIDTC

Connect the PIDTC to the computer using a USB cable. The end of the USB cable going into the PIDTC is a Type B USB connection:


From Computer Desktop Encyclopedia
© 1999 The Computer Language Co. Inc.
Type A (host or hub)
Type B (peripheral)

Powering the Hardware

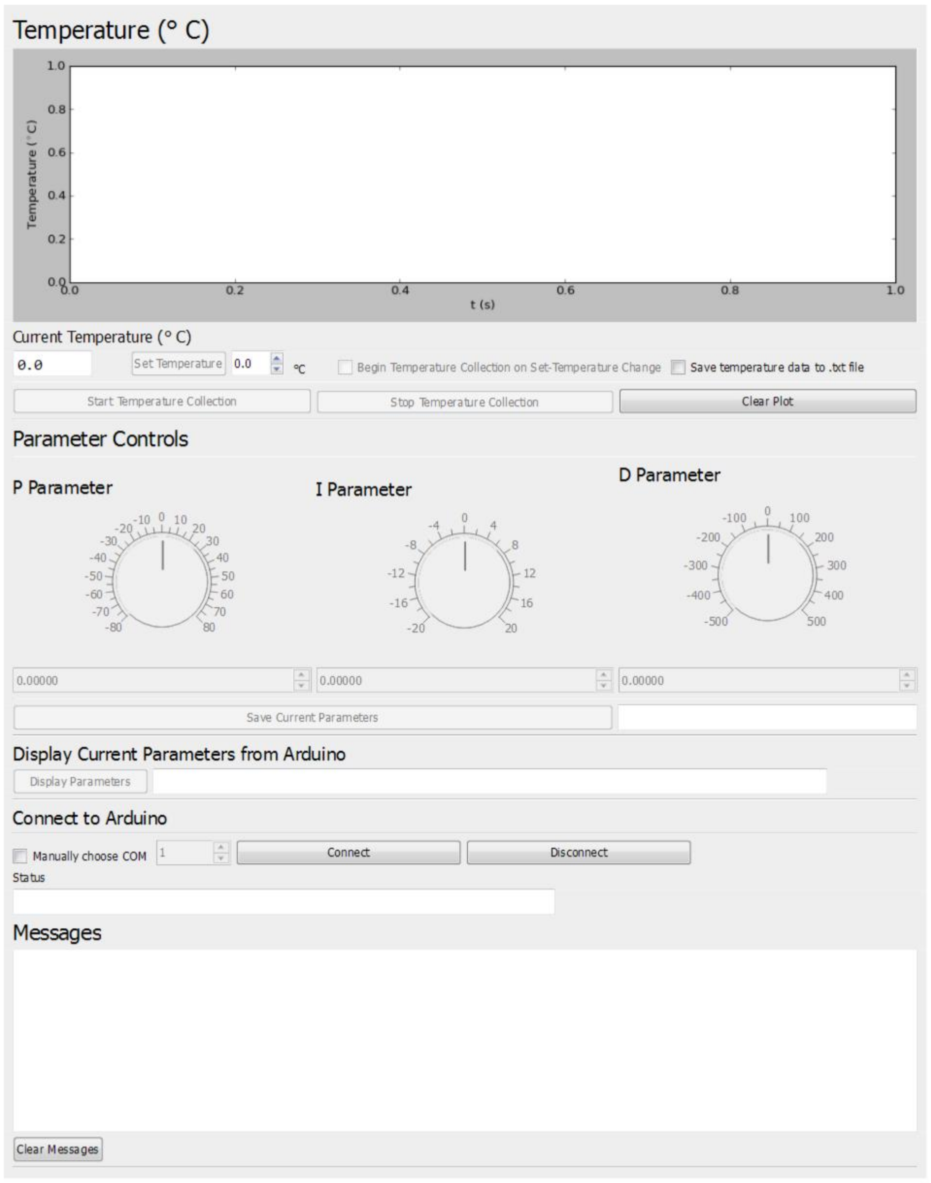
The Arduino is sufficiently powered by the USB connection. The circuit needs a **+/- 5 V** power supply that should be able to provide a maximum current of **2.7 A.** In practice, the circuit pulls around **1 A** when heating or cooling at full power.

Heat-sinking the TEC

The TEC is a two-sided device that functions in the following way: when current is passed through the TEC in one direction (call it the positive direction) Side A will heat and Side B will cool, and when current is passed in the negative direction Side A will cool and Side B will heat. The TEC is most efficient when the heat on the hotter side is sufficiently removed, so a heat sink becomes necessary; otherwise, the heat will begin to leak towards the colder side and the colder side loses its cooling efficiency.

In practice, one side of the TEC will be touching the system that is being temperature-controlled and the other side will be open. This open side should be put in thermal contact with the heat sink.

Using the PID Interface



Current Temperature (° C): 0.0

Set Temperature: 0.0 °C

P Parameter: 0.00000

I Parameter: 0.00000

D Parameter: 0.00000

Manually choose COM: 1

Connecting to the PIDTC's Arduino

With the PIDTC connected to the computer via USB, click '**Connect**'. If '**Manually choose COM**' is unselected, the program will search all devices connected to the computer and automatically connect to the one that identifies itself as the PIDTC. Alternatively, if the COM port is known, the user can manually choose which COM the program will connect to. In this case, the program still checks to make sure the device connected to the chosen COM port is the PIDTC. The programs attempts to connect to the PIDTC will be shown in the **Messages** box.

When the PID Interface connects to the PIDTC, a successful connection message will be output to the **Messages** box and the controls will become enabled. The parameters that were last saved the PIDTC will be loaded onto the interface. The user now has control over the PIDTC and can either change the control parameters or monitor the temperature of the system.

Changing Control Parameters*

When any parameter is changed, there is a confirmation carried out between the PIDTC and the PID Interface:

1) the PID Interface sends the new parameter to the PIDTC

2) the PIDTC sends the value it received back to the PID Interface

3) the PID Interface checks to make sure that the value it sent to the PIDTC matches the value that it got back from the PIDTC.

If the confirmation is successful, a message indicating this fact will be displayed in **Messages** and the value indicated in the data field or control knob will change to the new one; otherwise, if communication was unsuccessful, the value will not appear to change on the PID Interface.

The set-temperature is changed by entering the desired value into the box next to the '**Set Temperature'** button and then clicking the '**Set Temperature**' button.

The P, I, and D parameters can be changed by using either the control knobs or the data input fields below the knobs.

Note*: Parameters *cannot* be changed while temperature collection is taking place. It is difficult to interact with the PIDTC when it is continuously sending the temperature information to the PID Interface. For this reason, most controls are disabled when temperature collection is taking place.

Saving Control Parameters

The control parameters can be saved to the PIDTC's internal memory. When the PID Interface is connected to the PIDTC and a parameter is changed, the PIDTC always uses the newest set of control parameters while it is currently running. However, when the PIDTC loses power, as is the case when the USB is disconnected, any changes to the parameters will be lost unless they have been saved. The PIDTC will load the previously saved parameters whenever it is powered up from an inactive state or reset (i.e., the USB is disconnected or the PID Interface connects to the PIDTC, respectively). The parameters saved on the PIDTC's memory can be viewed by clicking the '**Display Current Parameters from Arduino**' button.

Displaying the System Temperature

The temperature measured by the PIDTC can be displayed on the PID Interface by clicking the '**Start Temperature Collection**' button. The temperature will be plotted as a function of time (the time on the *x*-axis is the time elapsed since the PID Interface connected to the PIDTC). The temperature will also be displayed in the **Current Temperature** box. Temperature collection is stopped by clicking the '**Stop Temperature Collection**' button.

There is an optional feature to automatically begin displaying the temperature of the system when the set-temperature is changed, and it is enabled by checking the very explicit '**Begin Temperature Collection on Set-Temperature Change**' check-box.

Additionally, the PID Interface can save the temperature *v.s.* time data points to a `.txt` file whenever temperature collection is started by checking the '**Save temperature data to a .txt file**' checkbox. The data file will be saved to a folder named `Data` located in the directory containing the `PID_interface.html` file, and the file name will consist of the date and time of the recording. The graph of the recorded temperature *v.s.* time can also be saved as a `.png` file by right-clicking the graph. A save-dialog box will come up.

Disconnecting the PIDTC

Once the control parameters are tuned and the set-temperature is at the desired value, the PID Interface can disconnect from the PIDTC. To do this, click the '**Disconnect**' button. The **Messages** box will indicate whether or not the disconnection was successful. The PIDTC's Arduino must be powered via USB, so the USB cable must be connected to either a USB hub or a computer in order for the PIDTC to be operational.

It should be noted that connecting the USB to a computer does not mean the PID Interface has been connected to the PIDTC's Arduino; at this point, the computer and the Arduino are not interacting, and instead the computer is simply providing power to the Arduino. For this reason, the PIDTC can be left connected to a computer and the computer will simply act as a power source for the Arduino. Still, the PIDTC should be disconnected from the PID Interface once the parameters are at the users' desired values.

Further Reading/Troubleshooting

The following sections contain a detailed explanation of how the PID Interface, the Arduino, and the circuit in the PIDTC interact to control the heating/cooling element. These sections should explain all the small ins and outs of the program and the device, so they should be sufficient for troubleshooting any problems that may occur.

Python Code

The commands made through the interface are carried out in Python code. The program connects to the Arduino using the pySerial library. If no manual COM choice is chosen, the program will cycle through COMs 1 through 100 until a connection is made. If a connection is made and the device identifies itself as PID temperature controller, initialization begins. Initialization includes two steps: interface controls are enabled, and the last saved parameters are loaded and the parameters are changed to these values. Any errors in this process will be printed to the 'Messages' box. Once initialization is finished, the user is free to change the PID parameters or to start temperature collection.

The program and the Arduino interact with simple character commands – the user interacts with the interface to begin a command, the program sends a character to the Arduino, and the Arduino, continuously checking the serial port, reads the character and executes a command based on that character. A simple example: the user clicks the "Start Temperature Collection" button, the program sends the character 't' to the Arduino, the Arduino begins writing the recorded temperature to the serial, and this temperature is plotted.

PID parameters are changed by turning the appropriate knob and this change is reflected in the text-box below the knob, or by entering the desired value directly into the text-box, in which case the knob setting is changed to reflect this value. The program handles this command by writing the appropriate character (program writes 'p' to change the P parameter) followed by the new value of this parameter to the Arduino.

Temperature collection is started by pressing the "Start Temperature Collection" button. When this button is pressed, the Arduino is commanded to write the current recorded temperature to the serial port. To speed up plotting, the program collects the data from the serial port for 0.5 seconds and compiles this data (temperature and time-stamp) into an array, and then plots the data.

Some user-friendliness has been added to the program and interface. Normally, problems will arise when a serial connection is made and then unexpectedly broken. The COM port can get stuck open and will be blocked to further interaction. The interface has a "Disconnect" button to close the Arduino, but the program will also properly close the serial port if the user exits the program. If parameters are changed on the interface, the “Save Current Parameter” button will become enabled. When it is clicked, the parameters are saved and the button becomes disabled. If the parameters have changed and have not been saved and the user chooses to exit the program, the interface will prompt the user to save the parameters or continue exiting the program.

Arduino Sketch

The Arduino sketch is a continuously-run loop. This is the control flow:

- Check the serial port to see if any data is available.
- If data is available for reading from the serial port, read it. The only thing written to the serial port should be a character sent by the Python program
- Based on the character, execute a certain command (if ‘p’, read serial again for float value and change P to this value, if ‘s’, read serial again for float value and change the set-temperature to this value). Here are some special cases:
    - User clicks “Start Temperature Collection”; change Boolean variable **python_wants_data** to true. Clicking “Stop Temperature Collection” will set **python_wants_data** to false.
    - User clicks “Save Current Parameters”; save the P, I, D, set-temperature variables to the Arduino EEPROM.
- Calculate the temperature using the resistance across the thermistor
- Based on the most recent error (current_temperature – set_temperature), the last error, and the error prior to the last one, calculate the appropriate output using the PID algorithm. The result is *PID_output*.
- Map *PID_output* to an appropriate range for PWM output (80 to 220, for example), and write *PID_output* to the appropriate Arduino pin to drive the TEC
- If **python_wants_data** is true, write the current temperature to the serial port.

Circuit

Here is an overview of how the circuit works with the Arduino:

1) The Arduino measures the resistance across the thermistor (a TH10K), and the temperature is calculated from this value. The temperature determines the output of the PID algorithm.

2) The Arduino outputs a PWM signal whose duty cycle is based on the output of the PID algorithm. A PWM signal is a 5V peak-to-peak signal whose duty cycle depends on the value written to the specific Arduino pin (values range from 0, resulting in 0% duty cycle, to 255, resulting in 100% duty-cycle).

3) The PWM signal is passed through a low-pass filter to convert it to a DC signal whose amplitude is directly proportional to the duty cycle, with a range of 0 to 5V.

4) The PWM is divided by 10 using a voltage divider.

5) An op-amp subtractor serves to subtract some voltage from this DC signal. Because this process is not exactly linear, the result is usually an output range of -.3V to +.3V

6) This DC signal, now taking on negative and positive values, is amplified by a factor of 10 using an op-amp to a range of -3V to 3V. This signal powers the heating/cooling element, nominally a thermo-electric cooler (TEC).

Appendix

Arduino Sketch

```
#include <EEPROM.h>
#include <EEPROMAnything.h>

struct parameters {  // a structure containing all the PID parameters
    float P;
    float I;
    float D;
    float set_temp;  // the set-temperature of the PID
}
params;

double Pterm = 0;
double Iterm = 0;
double Dterm = 0;
int controlPin = 2;        // sends PWM signal towards the TEC
int tempPin = A2;          // measures the thermistor's voltage
int totalPin = A4;         // measures the total jvoltage input
int highPin = 22;          // powers the TH10K

double error;              // the difference between the set temp and the real temp
double error_sum = 0;           // the accumulated error
double last_error = 0;      // the previous error
double delta_error;         // the change in error over one time step of the temperature collection
unsigned long last_time;           // the previous time stamp
double delta_time;          // the time step of the temperature collection
double PID_output = 0;

double temp = 0;            // the real temperature measured from thermistor
double last_temp = 0;       // the previous temperature measured from thermistor
double delta_temp;          // the change in temperature over one time step of the temperature collection
double v_therm,v25;         //  v_therm is voltage drop across the TH10K, v25vt is voltage drop adross R25, so voltage going into
TH10k is v25-vt
double log_ratio_Rt_R25;    // log of the ratio between the current resistance of TH10K and resistance of TH10K at
298.15K(which is 10KOhm according to the instruction)
float a = 3.354e-3;        // parameters for obtaining temperature from TH10K datasheet
float b = 2.562e-4;
float c = 2.14e-6;
float d = -7.241e-8;
boolean ready = false;
boolean python_wants_data = false;
String ID = "PID Temperature Controller";

void setup() {
  Serial.begin(9600);
  pinMode(highPin,OUTPUT);
  EEPROM_readAnything(0, params); //read stored parameters from the EEPROM and store them in params
}

void loop() {
  // confirm (one time only) that Serial is ready to send/receive data
  if (Serial && ready==false) {
      Serial.println("Serial ready");
      ready=true;
  }
  else if (!Serial) {
    Serial.println("Serial not ready");
    ready=false;
  }

  // check for incoming commands from PID Controller
  if (Serial && Serial.available() > 0) {
      char response = Serial.read();
      // 'response' is which parameter you wish to change
      switch (response) {
        case 'p':
          // change P
          params.P = Serial.parseFloat();
          // send the command character and the parameter value back to Python side for confirmation
          Serial.print(response);Serial.print(",");Serial.println(params.P,5);
          break;
        case 'i':
          // change I
          params.I = Serial.parseFloat();
          Serial.print(response);Serial.print(",");Serial.println(params.I,5);
          break;
        case 'd':
          // change D
          params.D = Serial.parseFloat();
          Serial.print(response);Serial.print(",");Serial.println(params.D,5);
          break;
        case 's':
          // change set-temperature
          params.set_temp = Serial.parseFloat();
          Serial.print(response);Serial.print(",");Serial.println(params.set_temp,5);
          break;
        case 't': //'t'emperature wanted
          python_wants_data = true;
          Serial.println(response);
          break;
        case 'h': //'h'alt data sending
          python_wants_data = false;
          Serial.println(response);
          break;
        case 'g': //'g'ive current parameters to python
          Serial.println(response);
          Serial.print(params.P,5); Serial.print(",");
          Serial.print(params.I,5); Serial.print(",");
          Serial.print(params.D,5); Serial.print(",");
          Serial.println(params.set_temp,5);
          break;
        case 'w': // 'w'rite params to eeprom, i.e. the params are good and will be used by the arduino until they are changed by the
user.
          EEPROM_writeAnything(0, params);
          Serial.println(response);
          break;
        case 'r': // 'r'eveal Arduino ID:
          Serial.println(ID);
          break;
        Serial.flush();
      }
  }
  else if (!Serial) {
    Serial.println("Nothing sent/Serial not ready");
  }
  unsigned long time_now = millis();
  digitalWrite(highPin,HIGH);
  v25 = analogRead(totalPin);   //------------------ the voltage going into the TH10K, in binary range from 0 to 1023.
  v_therm = analogRead(tempPin);   //----------------- the voltage across the TH10K.
  // Rt,R25: the resistance of TH10K at temp T and resistance of TH10K at 25 degC (10kohm)
  // Given by formula R_therm/R25 = V/Vin,  R_therm = R25*vt/(v25-vt) --> R_therm/R25 = v_therm/(v25-v_therm)
  log_ratio_Rt_R25 = log(v_therm/(v25-v_therm));
  // formula for temperature, from TH10K datasheet
  temp = double(1.0/(a + b*log_ratio_Rt_R25 +
c*(log_ratio_Rt_R25*log_ratio_Rt_R25)+d*(log_ratio_Rt_R25*log_ratio_Rt_R25*log_ratio_Rt_R25)) - 273.0);
  if (python_wants_data) {
     Serial.print(time_now/1000.0); Serial.print(",");Serial.println(temp);
  }
  error = params.set_temp - temp;
  //No need to worry about rollover, it is taken care of by using unsigned variables.
  delta_time = (double)(time_now - last_time);
  delta_temp = temp - last_temp;
  Pterm = params.P*error;
  if (PID_output > 81 && PID_output < 219)
    Iterm += params.I*error*delta_time;
  if (delta_time == 0.0)
    Dterm = 0.0;
  else
    Dterm = params.D*delta_temp/delta_time;
// FOR DEBUGGING
//  Serial.print("error");Serial.println(error);
//  Serial.print("Pterm");Serial.println(Pterm);
//  Serial.print("Iterm");Serial.println(Iterm);
//  Serial.print("Dterm");Serial.println(Dterm);
//  Serial.print("delta_error");Serial.println(delta_error);

  //The value written to the PWM pin (which is the signal that will later drive the TEC) is forced to take on a value between 80
and 220
  //in order for the output to be symmetric.
//  PID_output = constrain(previous_output + params.P*(error - last_error) + (params.I*params.It)*(error) +
(params.D/params.Dt)*(lastlast_error - 2*last_error + error), 80, 220); // typical PID statement
  //PWM: 150 is the 'midpoint' of the output, so assuming temperature is perfectly stable, then Pterm, Iterm, and Dterm equal 0
and output should be 150
  PID_output = constrain(Pterm + Iterm - Dterm + 150, 80, 220);
//  Serial.print("PID output");Serial.println(PID_output);
  last_error = error;   //--------------------------------- update error and time
  last_time = time_now;
  last_temp = temp;
  analogWrite(controlPin, int(PID_output)); //-------------------- drive the TEC with this PWM signal

  delay(100);  //--------------------------------------------------- (in milliseconds)
}
```

PID Interface Python Code

```
# Created by Connor McPartland
# University of Pittsburgh 2013


#   This program interacts with an Arduino that is acting as a PID temperature controller.
#   An Arduino object is created in order to send commands and receive data from the Arduino (the temperature controller).
#   Functionality includes:
#       - plotting the recorded temperature over time
#       - changing the PID parameters (P, I, D)
#       - changing the set-temperature of the controller
#       - saving the parameters to the memory of the Arduino for future use
#       - commands sent to temperature controller are sent back to Python side to make sure
#           no command data was lost in transmission
#   User-friendliness:
#       - disconnects the Arduino if the user shuts down the program
#       - warns the user if parameters have been changed but not saved
#
import math
import serial
import numpy as np
import time
import os
from Arduino import Arduino

class TimeoutException(Exception):
    pass

def initialize(P_knob, I_knob, D_knob,
                P_result_box, I_result_box,  D_result_box,
                com_choice, save_params_button, set_temp_button, display_params_button,
                start_data, stop_data, current_temp, auto_collect, messages, plot, **kwargs):
    global params_changed
    params_changed = False
    P_knob.enabled=False
    I_knob.enabled=False
    D_knob.enabled=False
    P_result_box.enabled=False
    I_result_box.enabled=False
    D_result_box.enabled=False
    com_choice.enabled=False
    save_params_button.enabled=False
    display_params_button.enabled=False
    set_temp_button.enabled=False
    start_data.enabled=False
    stop_data.enabled=False
    auto_collect.enabled=False
    current_temp.value=0.00

    clear_plot(plot,**kwargs)
    clear_messages(messages, **kwargs)

    # Create Data folder if it hasn't already been created.
    global data_directory
    data_directory = os.getcwd() + '\\Data'
    if not os.path.exists(data_directory):
        os.makedirs(data_directory)

def terminate(**kwargs):
    try:
        global arduino
        if arduino.is_connected:
            arduino.disconnect()
            print 'Arduino closed.'
    except NameError:
        pass

def clear_plot(plot, **kwargs):
    plot.clear(rescale=True)
    plot.set_plot_properties(
            x_label='t (s)',
            y_label='Temperature ($^\circ$C)',
            x_scale='linear',
            y_scale='linear',
            aspect_ratio='auto')
    plot.new_curve('temperature', memory='growable', length= 100000, animated=False,
                    line_style='-', line_width=1, line_color='red')
    plot.new_curve('set_temperature', memory='growable', length=100000, animated=False,
                    line_style='-', line_width=.5, line_color='black')

def clear_messages(messages, **kwargs):
    messages.clear()

def activate_com_choice(com_choice, manual_com_choice, **kwargs):
    if manual_com_choice.value:
        com_choice.enabled=True
    else:
        com_choice.enabled=False

# Connect to arduino via serial communication.
def connect_arduino(P_knob, I_knob, D_knob,
                    P_result_box, I_result_box,  D_result_box,
                    connected_result, com_choice, connect_button, disconnect_button, set_temp_box, manual_com_choice,
                    start_data, stop_data, save_params_button, display_params_button, params_box, set_temp_button, auto_collect,
current_temp,
                    plot, messages, **kwargs):
    global parameters
    global arduino
    arduino = Arduino()
    parameters = np.zeros(6)
    found_PID_controller = False
    found_arduino = False
    if manual_com_choice.value:
        arduino = Arduino('COM%s'%com_choice.value, 9600, 1.0)
        if arduino.connect():
            if arduino.ID == 'PID Temperature Controller':
                found_PID_controller = True
                found_arduino = True
                connected_result.value = 'Connected on COM%s'%(com_choice.value)
                messages.write('Successfully connected to Arduino on COM%s. '%com_choice.value)
            else:
                messages.write('Successfully connected to Arduino on %s but ID was not correct.\nID: %s\n' %(com_choice.value,
arduino.ID))
                found_arduino = True
                arduino.disconnect()
        else:
            connected_result.value = 'Failed to connect on COM%s' %(com_choice.value)
    else:
```

```
    coms = []

    for x in map(str, range(1,101)): coms.append('COM'+x)
    for com in coms:
        arduino = Arduino(com, 9600, 1.0)
        if arduino.connect():
            if arduino.ID == 'PID Temperature Controller':
                found_PID_controller = True
                found_arduino = True
                connected_result.value = 'Connected on %s' %(com)
                messages.write('Successfully connected to Arduino on %s. '%com)
                print arduino.ID
            else:
                messages.write('Successfully connected to Arduino on %s but ID was not correct.\nID: %s\n' %(com, arduino.ID))
                found_arduino = True
                arduino.disconnect()

        else:
            connected_result.value = 'Failed to connect on %s' %(com)
        if found_PID_controller: break
    if not found_arduino and not found_PID_controller:
        messages.write('Failed to connect to Arduino on COMs 1-100\n')
if found_arduino:
    if found_PID_controller:
        messages.write('PID Temperature Controller found.\n')
        if arduino.is_connected and arduino.is_ready:
            time.sleep(.2)
            messages.write('Arduino is open for communication.\n')
            P_knob.enabled=True
            I_knob.enabled=True
            D_knob.enabled=True
            P_result_box.enabled=True
            I_result_box.enabled=True
            D_result_box.enabled=True
            connect_button.enabled = False
            disconnect_button.enabled = True
            save_params_button.enabled=False
            display_params_button.enabled=True
            params_box.value=''
            set_temp_button.enabled=True
            start_data.enabled=True
            stop_data.enabled=True
            auto_collect.enabled=True

            initialize_parameters(P_knob, I_knob, D_knob,
                P_result_box, I_result_box,
                D_result_box, set_temp_box, start_data, stop_data,
                current_temp, auto_collect, messages, plot, params_box, **kwargs)
        else:
            messages.write('Arduino connected but not yet ready for serial communication.\n')
    else:
        messages.write('Failed to find PID Temperature Controller, but Arduino boards detected.\n')
else:
    arduino.disconnect()
    P_knob.enabled=False
    I_knob.enabled=False
    D_knob.enabled=False
    P_result_box.enabled=False
    I_result_box.enabled=False
    D_result_box.enabled=False
    disconnect_button.enabled=False
    connect_button.enabled=True
```

```
        set_temp_button.enabled=False
        display_params_button.enabled=False
        start_data.enabled=False
        stop_data.enabled=False


# Initialize parametrs. Upon opening the serial connection, arduino is forced to reset, erasing its current parameters.
# To reset the arduino to its state before opening serial connection, the most recently used parameters have been saved
#    from the last time the arduino was closed. Load previously saved parameters and resend them to arduino.
def initialize_parameters(P_knob, I_knob,  D_knob,
                          P_result_box, I_result_box,
                          D_result_box,  set_temp_box,
                          start_data, stop_data, current_temp, auto_collect,
                          messages, plot,  params_box, **kwargs):
                          #It_knob,It_result_box, Dt_knob, Dt_result_box):
    global parameters
    parameters = get_params(params_box, messages, **kwargs) #np.loadtxt('parameters.txt',delimiter=',', dtype='float')
    if not parameters == '':
        P_knob.value = float(parameters[0])
        P_result_box.value = float(parameters[0])
        I_knob.value = float(parameters[1])
        I_result_box.value = float(parameters[1])
        D_knob.value = float(parameters[2])
        set_temp_box.value = float(parameters[3])

# Change the set-temperature.
def set_temp(set_temp_box, start_data, stop_data, plot, current_temp, auto_collect, save_temp_data, set_temp_button,
             P_knob, I_knob, D_knob, P_result_box, I_result_box, D_result_box, disconnect_button,
             display_params_button, save_params_button, messages, **kwargs):
    global parameters
    global params_changed
    global arduino
    set_temp = float(set_temp_box.value)
    arduino.write('s')
    arduino.write(set_temp)
    time.sleep(.2)
    response = arduino.read().split(',')
    messages.write('Message %s was received by Arduino\n'%response)
    if response[0] == 's' and np.around(float(response[1]),7) == np.around(set_temp,7):
        parameters[3] = set_temp
        messages.write('Set Temeprature set to %f\n' %set_temp)
        params_changed = True
        save_params_button.enabled = True
        if auto_collect.value:
            start_data_collection(start_data, stop_data, P_knob, I_knob, D_knob, P_result_box, I_result_box, D_result_box,
                          disconnect_button, save_params_button, display_params_button,
                          plot, current_temp, auto_collect, save_temp_data, set_temp_button, set_temp_box, messages, **kwargs)
    else:
        set_temp_box.value = parameters[3]
        messages.write('Parameter not properly changed due to error in serial communication.\n')

# If P_knob is used, match the value of P_result_box below the knob to this value.
# If P_result_box is used, match the value of P_knob to this value, then send the value to arduino.
def set_P_box(P_result_box, P_knob, messages, save_params_button, **kwargs):
    P_result_box.value=P_knob.value
    set_P_parameter(P_result_box, P_knob, messages, save_params_button, **kwargs)
def set_P_parameter(P_result_box, P_knob, messages, save_params_button, **kwargs):
    global parameters
    global params_changed
    global arduino

    P = float(P_result_box.value)
    if arduino.is_connected:
        arduino.write('p')
        arduino.write(P)
        time.sleep(.2)
        response = arduino.read().split(',')
        messages.write('Message %s was received by Arduino\n'%response)
        if response[0] == 'p' and np.around(float(response[1]),7) == np.around(P,7):
            parameters[0] = P
            params_changed = True
            save_params_button.enabled = True
            messages.write('P set to %f\n' %P)
            P_knob.value=P_result_box.value
        else:
            P_result_box.value = float(parameters[0])
            P_knob.value = float(parameters[0])
            messages.write('Parameter not properly changed due to error in serial communication.\n')

# Repeat the procedure used for P parameter for the remaining parameters:
def set_I_box(I_result_box, I_knob, messages, save_params_button, **kwargs):
    I_result_box.value=I_knob.value
    set_I_parameter(I_result_box, I_knob, messages, save_params_button, **kwargs)
def set_I_parameter(I_result_box, I_knob, messages, save_params_button, **kwargs):
    global parameters
    global params_changed
    global arduino

    I = float(I_result_box.value)

    if arduino.is_connected:
        arduino.write('i')
        arduino.write(I)
        time.sleep(.2)
        response = arduino.read().split(',')
        messages.write('Message %s was received by Arduino\n'%response)
        if response[0] == 'i' and np.around(float(response[1]),7) == np.around(I,7):
            parameters[1] = I
            params_changed = True
            save_params_button.enabled = True
            messages.write('I set to %f\n' %I)
            I_knob.value=I_result_box.value
        else:
            I_result_box.value = float(parameters[1])
            I_knob.value = float(parameters[1])
            print parameters[1]
            messages.write('Parameter not properly changed due to error in serial communication.\n')

def set_D_box(D_result_box, D_knob, messages, save_params_button, **kwargs):
    D_result_box.value=D_knob.value
    set_D_parameter(D_result_box, D_knob, messages, save_params_button, **kwargs)
def set_D_parameter(D_result_box, D_knob,  messages, save_params_button, **kwargs):
    global parameters
    global params_changed
    global arduino

    D = float(D_result_box.value)

    if arduino.is_connected:
        arduino.write('d')
        arduino.write(D)
        time.sleep(.2)
        response = arduino.read().split(',')
        messages.write('Message %s was received by Arduino\n'%response)
        if response[0] == 'd' and np.around(float(response[1]),7) == np.around(D,7):
            parameters[3] = D
            params_changed = True
            save_params_button.enabled = True
            messages.write('D set to %f\n' %D)
            D_knob.value=D_result_box.value
        else:
            D_result_box.value = float(parameters[3])
            D_knob.value = float(parameters[3])
            messages.write('Parameter not properly changed due to error in serial communication.\n')

# Begin temperature data collection.
def start_data_collection(start_data, stop_data, P_knob, I_knob, D_knob, P_result_box, I_result_box, D_result_box,
                          disconnect_button, save_params_button, display_params_button,
                          plot, current_temp, auto_collect, save_temp_data, set_temp_button, set_temp_box, messages, **kwargs):
    global arduino
    global data_directory
    plot.clear_data('temperature')
    plot.clear_data('set_temperature')
    arduino.write('t')              # ask arduino to send temperature data
    response = arduino.read()
    if response == 't':
        set_temp_button.enabled=False
        start_data.enabled=False
        P_knob.enabled=False
        I_knob.enabled=False
        D_knob.enabled=False
        P_result_box.enabled=False
        I_result_box.enabled=False
        D_result_box.enabled=False
        save_params_button.enabled=False
        display_params_button.enabled=False
        disconnect_button.enabled=False
        if save_temp_data.value:
            try:
                dirs = data_directory+'\\'+time.strftime("\%d%b%Y_%Hh%Mm", time.localtime())
                os.makedirs(dirs)
                data_file = open(dirs+time.strftime("\Temperature_Data_%d%b%Y_%Hh%Mm.txt", time.localtime()), 'w')
                data_file.write('Time (s) \tTemperature (deg C)\n')
                params_file = open(dirs+time.strftime("\Params_Data_%d%b%Y_%Hh%Mm.txt", time.localtime()), 'w')
                params_file.write('Parameters:\n')
                params_file.write('P:\t%s\n' %P_result_box.value)
                params_file.write('I:\t%s\n' %I_result_box.value)
                params_file.write('D:\t%s\n' %D_result_box.value)
                params_file.write('Set temperature:\t%s deg C\n' %set_temp_box.value)
                params_file.close()
            except:
                message.write('Error creating data file. Data not saved!')
                save_temp_data.value = False
        messages.write('Collecting data...\n')
        while not stop_data.value:
            data = []
            current_time = time.time()
            temp = 0.0
            #Sample data for 0.8 seconds, then plot.
            while time.time() < current_time+0.8:
                response = arduino.read().split(",")
                time_from_arduino = float(response[0])
                temp = float(response[1])
                if save_temp_data.value:
                    data_file.write(str(time_from_arduino) + '\t' + str(temp) + '\n')
                data.append((time_from_arduino, temp))
            current_temp.value = temp
            plot.append_data('set_temperature', np.column_stack((np.linspace(data[0][0], data[-1][0], len(data)),
    np.ones(len(data))*float(set_temp_box.value))))
            #plot.append_data('set_temperature', np.column_stack((np.linspace(plot.plot_properties['ylimits'][0],
    plot.plot_properties['ylimits'][1], len(data)), np.ones(len(data))*float(set_temp_box.value))))
            plot.append_data('temperature', np.array(data))
        if save_temp_data.value:
            data_file.close()
        stop_data_collection(start_data, stop_data, set_temp_button,
                        P_knob, I_knob, D_knob, P_result_box, I_result_box, D_result_box,
                        disconnect_button, save_params_button, display_params_button, messages, **kwargs)
    else:
        messages.write('Temperature collection failed to start due to error in serial communication.\n')

    def stop_data_collection(start_data, stop_data, set_temp_button,
                        P_knob, I_knob, D_knob, P_result_box, I_result_box, D_result_box,
                        disconnect_button, save_params_button, display_params_button, messages, **kwargs):
    global arduino
    global params_changed
    arduino.write('h')
    time.sleep(.2)

    # It's possible that temperature data sent by the Arduino is stuck on the buffer when command 'h' is sent.
    # Read from Arduino to clear the buffer. Stop when the proper response is heard. If proper response is not heard
    #   in 3 seconds, assume communication has failed.
    end_time = time.time() + 3
    while time.time() < end_time:
        response = arduino.read()
        if response=='h':
            break
    if response=='h':
        stop_data.value=False
        start_data.enabled=True
        set_temp_button.enabled=True
        P_knob.enabled=True
        I_knob.enabled=True
        D_knob.enabled=True
        P_result_box.enabled=True
        I_result_box.enabled=True
        D_result_box.enabled=True
        disconnect_button.enabled=True
        if params_changed:
            save_params_button.enabled=True
        display_params_button.enabled=True
        messages.write('Data collection stopped.\n')
    else:
        messages.write('Temperature collection failed to stop due to error in serial communication.\n')
        disconnect_button.enabled=True


    # Generic function to send a message to arduino.
    #def write_to_arduino(message, error_message):
    #   global arduino
    #   try:
    #     arduino.write(message)
    #     arduino.flushInput()
    #     print 'Message "',message,'" sent to arduino'
    #   except:
    #     error_message.open()
    #     print 'Failed to send.'
    ## Generic function to read incoming message from arduino.
```

```
#def read_from_arduino(error_message):
#    global arduino
#    arduino.flush()
#    try:
#        response = arduino.readline().strip()
#        return response
#    except:
#        error_message.open()
#        print 'No response found'

# Save the current parameters to a file.
def save_parameters(save_status, messages, save_params_button, **kwargs):
    # This method saves the parameters to a text file.
    #try:
    #    np.savetxt('parameters.txt', parameters, delimiter=',', fmt='%g')
    #    save_status.value=''
    #    save_status.value='Parameters succesfully saved.'
    #    messages.write('Parameters saved to parameters.txt\n')
    #except IOError:
    #    save_status.value='Failed to save parameters!'
    #    messages.write('Error writing to parameters.txt!\n')

    # Tell arduino to write current parameters to eeprom
    global params_changed
    global arduino
    arduino.write('w')
    time.sleep(.2)
    response = arduino.read()
    if response == 'w':
        save_status.value='Parameters saved to Arduino.'
        params_changed=False
        save_params_button.enabled=False
        messages.write('Parameters saved to Arduino EEPROM.\n')
    else:
        save_status.value='Error in saving parameters.'
        save_params_button.enabled=True
        messages.write('Parameters not saved due to error in serial communication.\n')


# Get the current PID parameters being used by the arduino from the arduino and print them
def get_params(params_box, messages, **kwargs):
    global arduino
    arduino.write('g')
    time.sleep(.2)
    response = arduino.read()
    if response == 'g':
        params = arduino.read()
        try:
            params = params.split(',')
            params_box.value= 'P: %s\tI: %s\tD: %s\tSet Temp: %s' % (tuple(params))

            messages.write("Parameters succesfully loaded from Arduino's EEPROM.\n")
            return params
        except:
            messages.write('Arduino sent invalid data, parameters failed to load\n')
            return ''
    else:
        messages.write('Parameters could not be loaded due to error in serial communication.\n')
        return ''

# Close the arduino. Save the current parameters.
def disconnect(P_knob, I_knob,  D_knob,
                    P_result_box, I_result_box,  D_result_box,
                    connect_button, disconnect_button, connected_result, save_status, save_params_button, display_params_button,
params_box,
                    current_temp, start_data, stop_data, set_temp_button, messages, save_params_message, **kwargs):
    global arduino
    global params_changed
    if arduino.is_connected:
        if params_changed:
            if save_params_message.open():
                save_parameters(save_status, messages, save_params_button, **kwargs)
                time.sleep(.5)
                disconnect(P_knob, I_knob, D_knob,
                            P_result_box, I_result_box,  D_result_box,
                            connect_button, disconnect_button, connected_result, save_status, save_params_button,
display_params_button, params_box,
                            current_temp, start_data, stop_data, set_temp_button, messages, save_params_message, **kwargs)
            else:
                params_changed=False
                disconnect(P_knob, I_knob,  D_knob,
                        P_result_box, I_result_box, D_result_box,
                        connect_button, disconnect_button, connected_result, save_status, save_params_button, display_params_button,
params_box,
                        current_temp, start_data, stop_data, set_temp_button, messages, save_params_message, **kwargs)
        else:
            if arduino.disconnect():
                connected_result.value = 'Disconnected'
                messages.write('Arduino disconnected.\n')
                P_knob.enabled=False
                I_knob.enabled=False
                D_knob.enabled=False
                P_result_box.enabled=False
                I_result_box.enabled=False
                D_result_box.enabled=False
                display_params_button.enabled=False
                connected=False
                connect_button.enabled=True
                disconnect_button.enabled=False
                save_params_button.enabled=False
                save_status.value=''
                set_temp_button.enabled=False
                start_data.enabled=False
                stop_data.enabled=False
                params_box.value=''
                current_temp.value=0.0
                time.sleep(1)
            else:
                connected_result.value = 'Failed to disconnect'
                messages.write('Arduino failed to disconnect.\n')
```

PID Interface Pythics HTML Code

```
<!-- Created by Connor McPartland
University of Pittsburgh 2013
Contact: cmcpartland91@gmail.com-->

<html>
<head>
<title>PID Temperature Controller</title>
<style type="text/css">
body {background-color: #eeeeee; margin: 10px; padding: 5px}
a {align: left; color: black; font-size: 8pt; font-family: default; font-style: normal; font-weight: normal}
p {align: left; color: black; font-size: 8pt; font-family: default; font-style: normal; font-weight: normal}
h1 {align: center; font-size: 18pt; font-family: default; font-style: normal; font-weight: bold}
h2 {align: left; font-size: 16pt; font-family: default; font-style: normal; font-weight: normal}
h3 {align: left; font-size: 14pt; font-family: default; font-style: normal; font-weight: normal}
h4 {align: left; font-size: 12pt; font-family: default; font-style: normal; font-weight: normal}
h5 {align: left; font-size: 10pt; font-family: default; font-style: normal; font-weight: normal}
h6 {align: left; font-size: 8pt; font-family: default; font-style: normal; font-weight: normal}
object {align: left}
table {align: center}
</style>
</head>

<body>
<h2>Temperature (&#176; C)</h2>

<object classid='mpl.Plot2D' id='plot' width='100%' height='300'>
</object>
<br/>
<h5>Current Temperature (&#176; C)</h5>
<object classid='TextBox' id='current_temp' width='25%'>
        <param name='read_only' value='True'/>
</object>
<p>      </p>
<object classid='Button' id='set_temp_button' width='25%'>
        <param name='label' value='Set Temperature'/>
        <param name='action' value='PID_interface.set_temp'/>
</object>
<object classid='NumBox' id='set_temp_box' width='15%'>
        <param name='maximum' value='500'/>
        <param name='minimum' value='-273'/>
        <param name='label' value='Set Temp'/>
</object>
<p> &#176;C       </p>
<object classid='CheckBox' id='auto_collect'>
        <param name='label' value='Begin Temperature Collection on Set-Temperature Change'/>
</object>

<object classid='CheckBox' id='save_temp_data'>
        <param name='label' value='Save temperature data to .txt file'/>
</object>
<hr/>
<object classid='Button' id='start_data' width='33.33%'>
   <param name='label' value='Start Temperature Collection'/>
   <param name='action' value='PID_interface.start_data_collection'/>
```

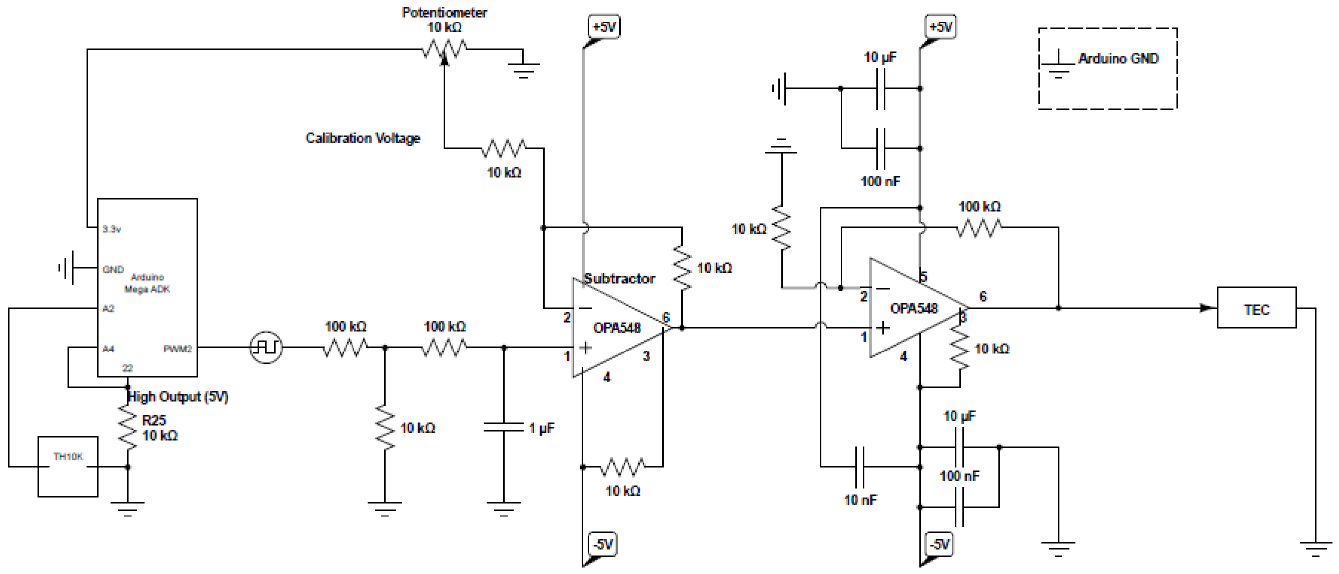
```
</object>
<object classid='Button' id='stop_data' width='33.33%'>
            <param name='toggle' value='True'/>
   <param name='label' value='Stop Temperature Collection'/>
</object>
<object classid='Button' width='33.33%'>
   <param name='label' value='Clear Plot'/>
   <param name='action' value='PID_interface.clear_plot'/>
</object>
<br/>

<hr/>
<h3>Parameter Controls </h3>
<!-- <h5>(Equation Used: PID_output = last_output + P*(error - last_error) + (I*It)*(error) + (D/Dt)*(lastlast_error -
2*last_error + error)</h5> -->

<hr/>
<!-- <h4>P Parameter                    I Parameter                    It Parameter                    D Parameter
Dt Parameter</h4> -->
<!--<h4>            P Parameter                                        I Parameter                                        D
Parameter                   </h4>-->
<table width='100%' border="1">
<tr><td width='25%'>
<h4>P Parameter</h4>
<object classid='qwt.Knob' id='P_knob' width='100%'>
            <param name='action' value='PID_interface.set_P_box'/>
            <param name='border_width' value='1'/>
            <param name='minimum' value='-80'/>
            <param name='maximum' value='80'/>
            <param name='step' value='1'/>
            <param name='scale_step' value='10'/>
            <param name='knob_width' value='100'/>
</object>
</td><td width='25%'>
<h4>I Parameter</h4>
<object classid='qwt.Knob' id='I_knob' width='100%'>
            <param name='action' value='PID_interface.set_I_box'/>
            <param name='tracking' value='False'/>
            <param name='minimum' value='-20'/>
            <param name='maximum' value='20'/>
            <param name='step' value='.5'/>
            <param name='scale_step' value='4'/>
            <param name='knob_width' value='100'/>
</object>
</td><!-- <td width='18%'>  -->
<!-- <object classid='qwt.Knob' id='It_knob' width='100%'>
            <param name='action' value='PID_interface.set_It_box'/>
            <param name='tracking' value='False'/>
            <param name='minimum' value='.1'/>
            <param name='maximum' value='10.0'/>
            <param name='step' value='.5'/>
            <param name='scale_step' value='1'/>
            <param name='knob_width' value='100'/>
</object> -->
<!-- </td> --><td width='25%'>
<h4>D Parameter</h4>
<object classid='qwt.Knob' id='D_knob' width='100%'>
            <param name='action' value='PID_interface.set_D_box'/>
            <param name='tracking' value='False'/>
            <param name='minimum' value='-500'/>
            <param name='maximum' value='500'/>
            <param name='step' value='5'/>
```

```
		<param name='scale_step' value='100'/>
		<param name='knob_width' value='100'/>
</object>
</td><!-- <td width='18%'>
<object classid='qwt.Knob' id='Dt_knob' width='100%'>
		<param name='action' value='PID_interface.set_Dt_box'/>
		<param name='tracking' value='False'/>
		<param name='minimum' value='1'/>
		<param name='maximum' value='10'/>
		<param name='step' value='.5'/>
		<param name='scale_step' value='2'/>
		<param name='knob_width' value='100'/>
</object>
</td> --></tr>
<tr><td width='25%'>
<object classid='NumBox' id='P_result_box' width='100%'>
		<param name='action' value='PID_interface.set_P_parameter'/>
		<param name='digits' value='5'/>
		<param name='maximum' value='80'/>
		<param name='minimum' value='-80'/>
</object>
</td><td width='25%'>
<object classid='NumBox' id='I_result_box' width='100%'>
		<param name='action' value='PID_interface.set_I_parameter'/>
		<param name='digits' value='5'/>
		<param name='maximum' value='20'/>
		<param name='minimum' value='-20'/>
</object>
</td><td width='25%'>
<object classid='NumBox' id='D_result_box' width='100%'>
		<param name='action' value='PID_interface.set_D_parameter'/>
		<param name='digits' value='5'/>
		<param name='maximum' value='500'/>
		<param name='minimum' value='-500'/>
</object>
</td></tr>
</table>
<hr/>



<!-- <object classid='NumBox' id='It_result_box' width='33.33%'>
		<param name='action' value='PID_interface.set_It_parameter'/>
		<param name='digits' value='3'/>
		<param name='maximum' value='30'/>
		<param name='minimum' value='-30'/>
</object> -->

<!-- <object classid='NumBox' id='Dt_result_box' width='33.33%'>
		<param name='action' value='PID_interface.set_Dt_parameter'/>
		<param name='digits' value='3'/>
		<param name='maximum' value='10'/>
		<param name='minimum' value='-10'/>
</object> -->
<br/>
<object classid='Button' id='save_params_button' width='66.6%'>
		<param name='label' value='Save Current Parameters'/>
		<param name='action' value='PID_interface.save_parameters'/>
</object>
<object classid='TextBox' id='save_status' width='33.3%'>
		<param name='read_only' value='True'/>
</object>
```

```
<hr/>
<h4>Display Current Parameters from Arduino</h4>
<object classid='Button' id='display_params_button' width='15%'>
		<param name='label' value='Display Parameters'/>
		<param name='action' value='PID_interface.get_params'/>
</object>
<object classid='TextBox' id='params_box' width='75%'>
		<param name='read_only' value='True'/>
</object>
<hr/>
<h4>Connect to Arduino</h4>
<hr/>
<object classid='CheckBox' id='manual_com_choice'>
		<param name='label' value='Manually choose COM'/>
		<param name='action' value='PID_interface.activate_com_choice'/>
</object>
<object classid='NumBox' id='com_choice' width='10%'>
		<param name='digits' value='0'/>
		<param name='increment' value='1'/>
		<param name='minimum' value='1'/>
		<param name='maximum' value='99'/>
</object>
<object classid='Button' id ='connect_button' width='30%'>
		<param name='label' value='Connect'/>
		<param name='action' value='PID_interface.connect_arduino'/>
</object>
<object classid='Button' id='disconnect_button' width='30%'>
		<param name='label' value='Disconnect'/>
		<param name='action' value='PID_interface.disconnect'/>
</object>
<h6>Status</h6>
<object classid='TextBox' id='connected_result' width='60%'>
		<param name='read_only' value='True'/>
</object>

<h3>Messages</h3>

<object classid='TextIOBox' id='messages' width='100%' height='200'>
</object>
<br/>
<object classid='Button'>
		<param name='label' value='Clear Messages'/>
		<param name='action' value='PID_interface.clear_messages'/>
</object>
<hr/>

<br/>
<object classid='MessageDialog' id='save_params_message' width='100%'>
    <param name='message' value='Parameters not saved! Do you wish to save them?'/>
    <param name='yes_button' value='True'/>
    <param name='no_button' value='True'/>
    <param name='severity' value='critical'/>
		<param name='label' value=''/>
</object>
<object classid='MessageDialog' id='error_message' width='100%'>
		<param name='message' value='An unexpected error has occurred. Recommend disconnect from Arduino, then
reconnect.'/>
		<param name='ok_button' value='True'/>
		<param name='severity' value='critical'/>
		<param name='label' value=''/>
</object>
<object classid='ScriptLoader' width='0%'>
```

```
    <param name='filename' value='PID_interface'/>
            <param name='initialization_action' value='PID_interface.initialize'/>
            <param name='termination_action' value='PID_interface.terminate'/>
            <param name='label' value="/>
</object>

</body>
</html>
```

Circuit Diagram


Potentiometer
10 kΩ
Calibration Voltage
10 kΩ
+5V
+5V
10 µF
100 nF
Arduino GND
3.3v
GND
Arduino Mega ADK
A2
A4
PWM2
22
High Output (5V)
R25
10 kΩ
TH10K
100 kΩ
100 kΩ
10 kΩ
1 µF
Subtractor
OPA548
10 kΩ
10 kΩ
-5V
10 kΩ
100 kΩ
OPA548
10 kΩ
TEC
10 µF
100 nF
10 nF
-5V