# HORNO PID

# HORNO PID

## Electrónica para Ciencias

**Juan Barbosa**
Departamento de Física
Universidad de los Andes

**Luisa Rodríguez**
Departamento de Física
Universidad de los Andes

# CONTENIDOS

**1 Controlador PID** **1**

1.1 Calibración de la temperatura 3
1.2 Caracterización del actuador 3
1.3 Sistema de alimentación 4
1.4 Controlador 5
1.5 Digitalización 6

Referencias 6

**2 Algoritmo** **9**

2.1 peltier.py 10
2.2 TCP.py 13
2.3 controller.py 15

Referencias 18

**3 Mejoras** **19**

# CAPÍTULO 1

# CONTROLADOR PID

Con el objetivo de controlar la temperatura sobre un actuador, se realiza un circuito con controlador PID. El circuito en su totalidad se compone de 5 partes principales:

1. **Sistema de alimentación:** tiene como objetivo generar los 5 V necesarios para encender la Raspberry Pi, así como fijar el valor máximo para la referencia.

2. **Sensor de temperatura:** transforma la temperatura del actuador en una variable eléctrica que es medida por el controlador y mostrada al usuario en tiempo real.

3. **Valor de referencia:** corresponde con el valor de temperatura al que se desea llevar el accionador.

4. **Controlador PID:** contínuamente evalua la diferencia entre el valor de referencia y la temperatura actual, además de aplicar correción que corresponden con términos proporcionales, integradores y derivativos.

5. **Circuito de accionamiento:** tiene como objetivo proporcionar la potencia necesaria para que el actuador funcione de manera correcta.

Estas partes se pueden observar en la Figura 1.1, donde se dibuja el circuito usado para el horno PID. En ella cada parte se encuentra señalada con un color específico, siendo el sistema de alimentación, rojo; el sensor de temperatura, verde; el valor de referencia, azul; el controlador, naranja; y el circuito de accionamiento en violeta.

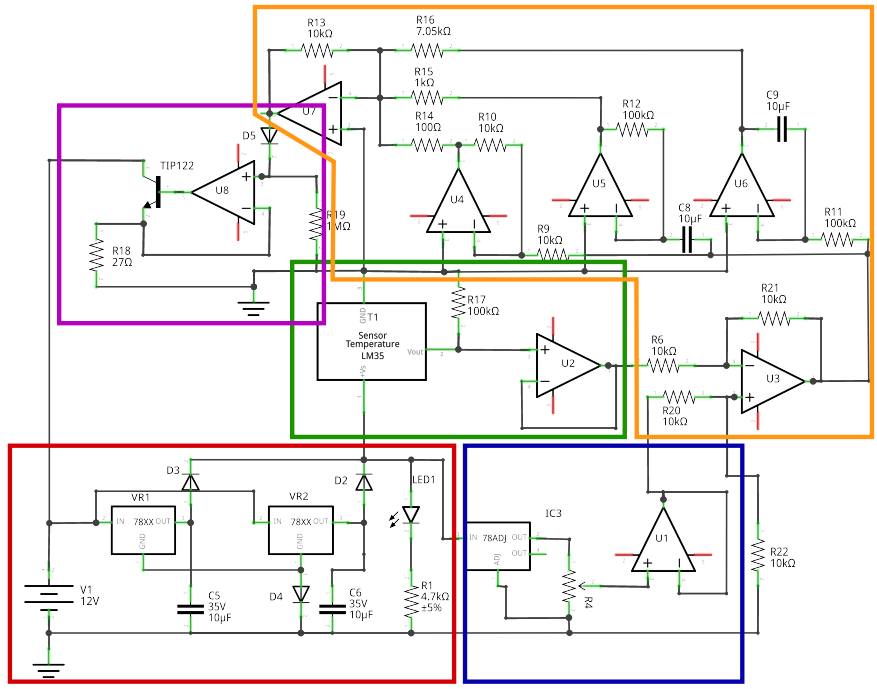



**Figura 1.1** Circuito PID, implementado para el control de la temperatura.

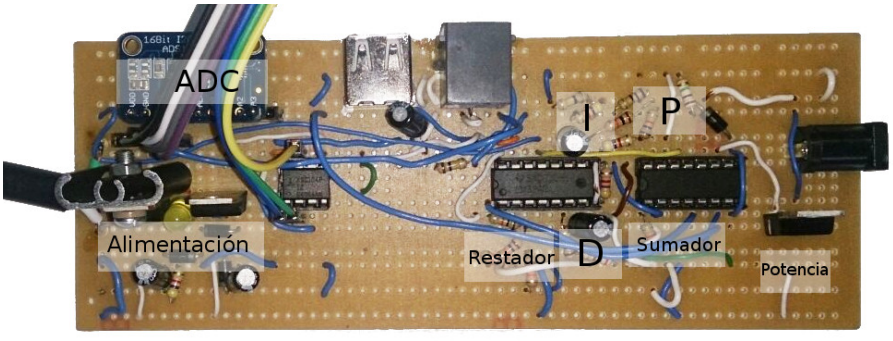



**Figura 1.2** Circuito implementado para el horno.

## 1.1 Calibración de la temperatura

El sensor de temperatura corresponde con un LM35, circuito integrado con respuesta lineal a la temperatura, con un factor de escala de 10 mV/°C y un rango de operación de -55 °C a 150 °C [1]. A pesar de ser un sensor calibrado, se realiza una calibración *in situ* usando una termocupla de NiCr-Ni Phywe.

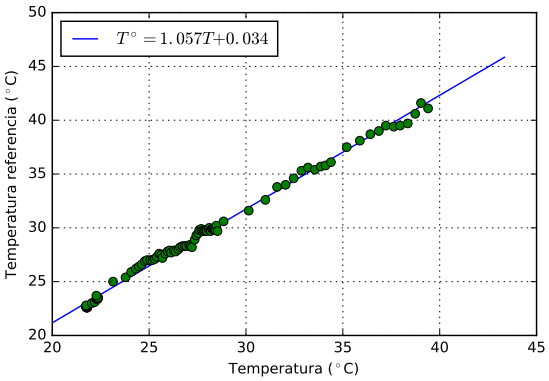



**Figura 1.3** Calibración del sensor.

Los resultados se muestran en la Figura 1.3, de donde se observa la linealidad del sensor usado, tanto la pendiente como el intercepto muestran desfases del valor esperado (1 y 0, correspondientemente) sobre la segúnda cifra decimal, discrepancias menores al 6 %.

## 1.2 Caracterización del actuador

Inicialmente la construcción del horno usaba una celda Peltier, no obstante esta dejó de funcionar. Por este motivo se modifica el actuador a una resistencia de 27 Ω y 5 vatios. Para una resistencia la potencia es proporcional al cuadrado de la diferencia de potencial entre sus terminales:

$$P = VI = V\left(\frac{V}{R}\right) \tag{1.1}$$

Debido a que el sistema se encuentra restringido a voltajes de $\pm 12$ y 0 V, la máxima potencia sobre la resistencia será 5.3 W, valor que se encuentra dentro de la capacidad de la resistencia.

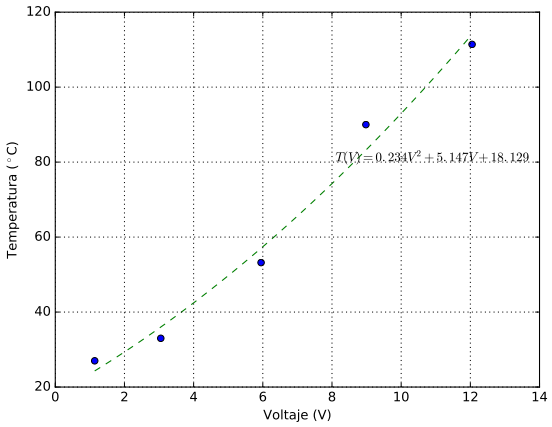



**Figura 1.4** Temperaturas alcanzadas por la resistencia para distintos valores de voltaje.

## 1.3 Sistema de alimentación

El sistema de alimentación está compuesto por dos reguladores de voltaje de 5 V de la familia 7805 [2]. El circuito de alimentación se encuentra acompañado de dos condensadores que actúan como filtros, además se dispone de tres diodos 1N4001 [3] como protección a corriente reversa. El circuito sin carga mantiene un potencial de 5.03 V, y con carga se alcanzan valores de 4.78 V.

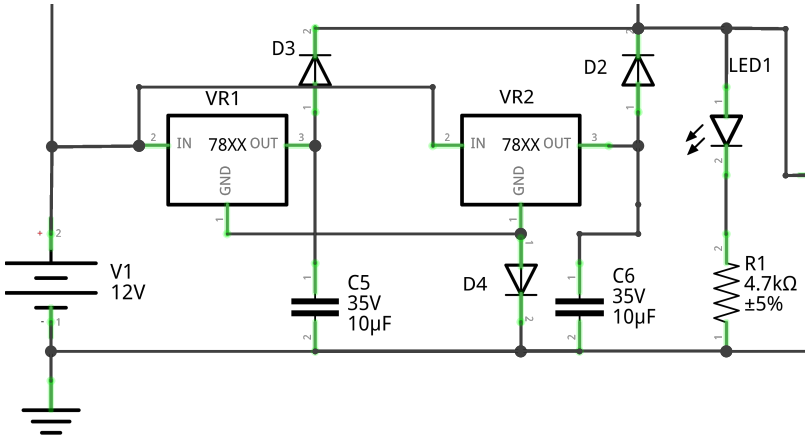



**Figura 1.5** Circuito de alimentación del horno.

## 1.4 Controlador

El controlador está compuesto por tres etapas:

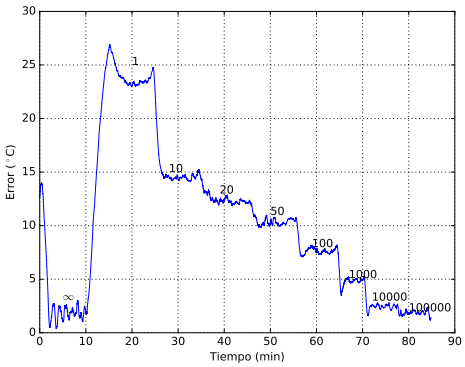
- **Restador:** calcula el error en la temperatura del horno.
- **PID:** todas las partes invierten la salida y mantienen la ganancia en 1.
- **Sumador:** se suma de forma ponderada las distintas componentes del PID. Las constantes usadas se muestran en la Tabla 1.1.

**Tabla 1.1** Parámetros del controlador PID.

| | |
|---|---|
| $K_P$ | 100 |
| $K_I$ | 10 |
| $K_D$ | 1.42 |

Los parámetros del controlador fueron determinados manualmente. En primer lugar se determinó $K_P$ usando 8 valores distintos de $R_{13}$.

$$K_P = \frac{R_{14}}{R_{13}} \tag{1.2}$$




En segundo lugar se agrega la etapa derivativa, para la cual la ganancia $K_D$ corresponde con;

$$K_D = \frac{R_{14}}{R_{15}} \tag{1.3}$$

Finalmente se agrega una etapa integradora para eliminar el error del estado estacionario, los valores de $K_I$ con los que se probó el circuito fueron: 10, 1 y 1.42. El primero, satura permanentemente el actuador, manteniendo siempre al máximo la diferencia de potencial. Con el segundo se alcanza un error del estado estacionario para una temperatura de 67.6 °C en 3 °C, error que se reduce con la última constante a cerca de 1 °C.

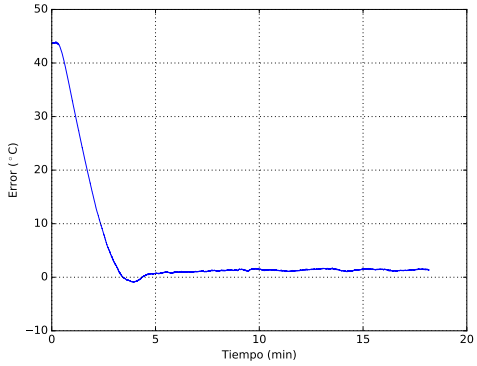



**Figura 1.6** Resultado final del controlador PID.

## 1.5 Digitalización

El valor de referencia es fijado usando un potenciometro digital con referencia X9C103P [4] con resistencia interna de 100 kΩ. Los tres pines lógicos son conectados a los GPIO, los puertos predeterminados son:

19 **IN** (INCREMENT)

20 **U/D** (UP/DOWN)

21 **C/S** (CHIP SELECT)

Sin embargo los mismos pueden ser modificados usando la interfáz gráfica. A continuación se muestran dos curvas del potencial en función de la posición de la aguja del potenciómetro.

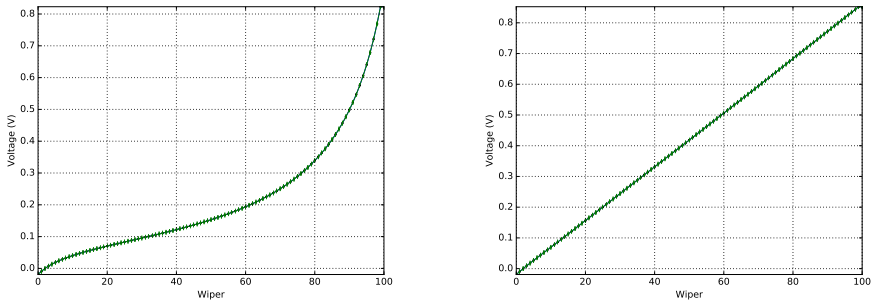



**Figura 1.7** Voltaje de salida del potenciómetro acoplado al restador del PID. A la izquierda con carga, en el derecho con seguidor.

En las figuras anteriores se puede apreciar el efecto de la carga sobre el divisor de voltaje. Una modificación final en la etapa de digitalización permitió lograr un rango de potenciales de 0 a 3.3 V, conservando la linealidad, pero reduciendo la selección de temperaturas a cada 3.3 °C.

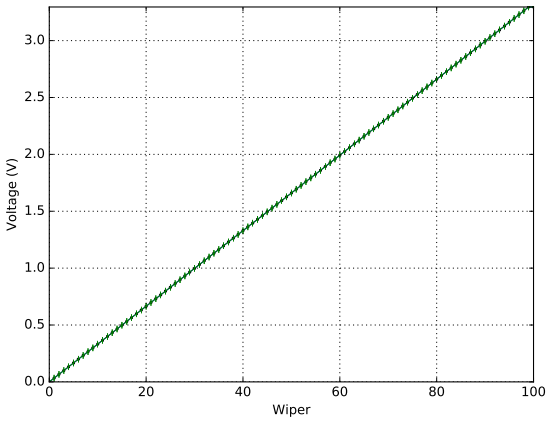



**Figura 1.8** Calibración final del potenciometro digital.

## REFERENCIAS

1. Texas Instruments, “LM35 Precision Centigrade Temperature Sensors” LM35 datasheet, Aug. 1999 [Revised Aug. 2016].
2. Spark fun “Positive-Voltage Regulators” $\mu$A7805C datasheet, May. 1976 [Revised May. 2003].
3. Diodes, “Diffused Junction 1N4001 - 1N4007” 1N4001 datasheet, Sep. 2004.
4. Digitally Controlled Potentiometer “X9C103” datasheet, Jul 2009.

# CAPÍTULO 2

# ALGORITMO

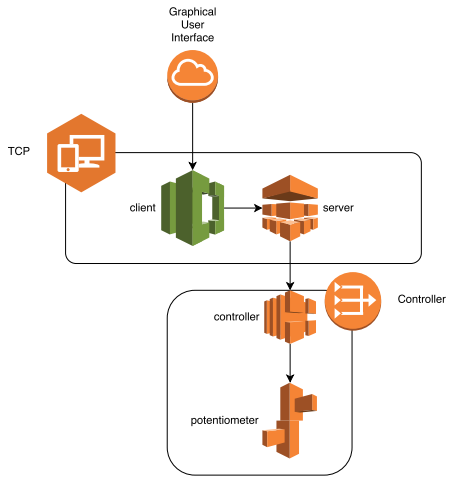



**Figura 2.1** Jerarquía de las clases usadas por la interfaz gráfica.

La parte computacional del proyecto está escrita en su totalidad en Python y usa una interfaz gráfica en Qt. Usando un paradigma de programación por objetos fueron creadas 5 clases que siguen la jerarquía que se muestra en la Figura 2.1.

## 2.1 peltier.py

Contiene la información relacionada con la interfaz gráfica, configura las señales y puertos necesarios por los objetos propios de Qt.

```
import sys
import time
import numpy as np
from TCP import client
import matplotlib.pyplot as plt
from matplotlib.backends.backend_qt4agg import (FigureCanvasQTAgg as
    FigureCanvas)
import csv
from PyQt4 import QtCore, QtGui, uic

form_class = uic.loadUiType("mainwindow.ui")[0]

class MyWindowClass(QtGui.QMainWindow, form_class):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.file = "Output.csv"
        self.connect_button.clicked.connect(self.start_client)
        self.startbutton.clicked.connect(self.start_stop)
        self.clear_button.clicked.connect(self.remplt)
        self.temperature_dial.sliderMoved.connect(self.
            temperature_change_1)
        self.desired_line.editingFinished.connect(self.
            temperature_change_2)

        self.addplt()
        self.start = False

        self.init_time = 0

        self.plt_time = np.zeros(0)
        self.plt_temp = np.zeros(0)
        self.plt_error = np.zeros(0)

        self.client = client()

        self.client_label.setText(self.client.ADDRESS)

        self.desired = float(self.desired_line.text())

        self.dataUpdate(["Time (min)", "Real", "Ref", "Error"])

    def addplt(self):
        self.figure, self.ax1 = plt.subplots(1,1, figsize = (8, 4.5))
```

```
        self.figure.subplots_adjust(bottom = 0.3, left = 0.13, right
            = 0.87)
        self.figure.set_facecolor('none')

        self.ax1.set_xlabel("Time (min)")
        self.ax1.set_ylabel("Temperature (C)", color = "r")
        self.ax1.set_xlim(0, 1)
        self.ax1.set_ylim(0, 20)
        self.ax1.grid()

        self.ax2 = self.ax1.twinx()
        self.ax2.set_ylabel("Error (C)", color = "grey")
        self.ax2.set_xlim(0, 1)
        self.ax2.set_ylim(0, 20)

        self.ax1.hold(True)
        self.ax2.hold(True)

        self.ax1_temperatures = self.ax1.plot([], [], "-", color = "r
            ")[0]
        self.ax2_errors = self.ax2.plot([], [], "-", color = "grey")
            [0]

        self.canvas = FigureCanvas(self.figure)
        self.pltvl.addWidget(self.canvas)
        self.canvas.draw()

    def remplt(self):
        self.pltvl.removeWidget(self.canvas)
        self.canvas.close()
        self.plt_error = np.zeros(0)
        self.plt_temp = np.zeros(0)
        self.plt_time = np.zeros(0)
        self.figure = None

    def plotter(self):
        current_time = (time.time() - self.init_time)/60

        data = self.client.send_data("%.1f"%self.desired)
        current_temp, current_ref = eval(data)
        current_error = current_ref - current_temp

        self.dataUpdate([current_time, current_temp, current_ref,
            current_error])

        self.plt_time = np.append(self.plt_time, [current_time])
        self.plt_temp = np.append(self.plt_temp, [current_temp])
        self.plt_error = np.append(self.plt_error, [current_error])

        minx, maxx = self.ax1.get_xlim()
        miny, maxy = self.ax1.get_ylim()

        if self.plt_time[-1] >= maxx:
            self.ax1.set_xlim(minx, 3*maxx/2)
        if self.plt_temp[-1] >= maxy:
            self.ax1.set_ylim(miny, 3*maxy/2)
```

```
        miny, maxy = self.ax2.get_ylim()
        if self.plt_error[-1] >= maxy:
            self.ax2.set_ylim(min(self.plt_error), 3*maxy/2)
        elif self.plt_error[-1] <= miny:
            self.ax2.set_ylim(min(self.plt_error), maxy)

        self.ax1_temperatures.set_data(self.plt_time, self.plt_temp)
        self.ax2_errors.set_data(self.plt_time, self.plt_error)
        self.canvas.draw()

        self.current_line.setText("%.3f"%current_temp)
        self.available_label.setText("%.1f"%current_ref)

    def start_stop(self):
        self.start = not self.start
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.plotter)

        if self.start:
            if self.init_time == 0 or self.figure == None:
                self.init_time = time.time()

                pot_info = self.IC_spinBox.value(), self.UD_spinBox.
                    value(), self.CS_spinBox.value()
                data = "I(%s,␣%s,␣%s)"%(pot_info)

                self.client.send_data(data)

            if self.figure == None:
                self.addplt()

            self.desired = float(self.desired_line.text())

            try:
                self.desired = float(self.desired)
            except Exception as e:
                print(e)

            self.startbutton.setStyleSheet("background-color:␣rgb(38,
                ␣229,␣57)")
            self.timer.start(200)
        else:
            self.startbutton.setStyleSheet("background-color:␣rgb
                (229,␣38,␣38)")
            self.timer.stop()

    def temperature_change_1(self, value):
        self.desired = float(value)
        self.desired_line.setText(str(self.desired))

    def temperature_change_2(self):
        value = float(self.desired_line.text())
        self.desired = value
        self.desired_line.setText(str(value))
        self.temperature_dial.setValue(value)
```

```
    def start_client(self):
        host = self.host_line.text()
        port = self.port_line.text()
        try:
            port = int(port)
            ans = self.client.start_client(host, port)

            if ans != None:
                raise Exception(ans)

            else:
                self.status_label.setStyleSheet("background-color: 
                    rgb(38, 229, 57)")

        except Exception as E:
            self.showdialog(E)

    def showdialog(self, error):
        msg = QtGui.QMessageBox()
        msg.setIcon(QtGui.QMessageBox.Critical)

        msg.setText("Something went wrong:")
        msg.setInformativeText(str(error))
        msg.setWindowTitle("TCP Error")

        retval = msg.exec_()

    def dataUpdate(self, data):
        f = open(self.file, "a")
        writer = csv.writer(f)
        writer.writerow(data)
        f.close()


app = QtGui.QApplication(sys.argv)
MyWindow = MyWindowClass(None)
MyWindow.show()
app.exec_()
MyWindow.client.close_socket()
```

## 2.2 TCP.py

Contiene las clases `client` y `server` que funcionan como cliente y servidor TCP (Protocolo de Control de Transmisión) los cuales permanentemente intercambian los datos de temperatura deseada, actual y referencia. El uso de un servidor TCP permite interactuar con el horno a través de una red LAN.

```
import socket
from controller import controller

class client:
```

```
    def __init__(self, TCP_IP = "10.42.0.207", TCP_PORT = 12345,
        BUFFER_SIZE = 1024):
        self.TCP_IP = TCP_IP
        self.TCP_PORT = TCP_PORT
        self.BUFFER_SIZE = BUFFER_SIZE
        self.ADDRESS = socket.gethostname()

    def start_client(self, TCP_IP, TCP_PORT):
        self.TCP_IP = TCP_IP
        self.TCP_PORT = TCP_PORT
        try:
            self.socket = socket.socket(socket.AF_INET, socket.
                SOCK_STREAM)
            self.socket.connect((self.TCP_IP, self.TCP_PORT))
        except Exception as e:
            return e

    def send_data(self, message):
        message = message.encode(encoding='UTF-8')
        self.socket.sendall(message)
        data = self.socket.recv(self.BUFFER_SIZE).decode(encoding='
            UTF-8')
        return data

    def close_socket(self):
        self.socket.close()


class server:
    def __init__(self, host, port):
        self.host = host
        self.port = port

        self.socket = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM)
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR
            , 1)
        self.socket.bind((self.host, self.port))
        print(host , port)

        self.socket.listen(1)
        self.conn, addr = self.socket.accept()

    def attention(self):
        while True:
            try:
                data = self.conn.recv(1024).decode(encoding='UTF-8')

                if data:
                    try:
                        data = float(data)

                        temp, ref = self.controller.read()
                        self.controller.set_desired(data)
                        answer = ("(%.3f, %.3f)"%(temp, ref)).encode(
                            encoding='UTF-8')
```

```
                    self.conn.sendall(answer)
                except ValueError:
                    if data[0] == "I":
                        self.conn.sendall("Fine".encode(encoding=
                            'UTF-8'))
                        pot_info = eval(data[1:])
                        self.controller = controller(pot_info)

                    elif data == "close":
                        break

            else:
                self.socket.listen(1)
                self.conn, addr = self.socket.accept()

        except socket.error:
            print("Error Occured.")
            break
    self.conn.close()
```

## 2.3 controller.py

Contiene las clases `controller` y `potentiometer`, la primera determina los puertos lógicos con los que la Raspberry controla el potenciometro digital, además, determina la posición de la aguja del potenciometro para la cual se obtiene el voltaje más cercano a la temperatura deseada. La clase `potentiometer` está en capacidad de modificar la posición de la aguja del potenciometro cada vez que `controller` lo considere necesario.

```
try:
    import RPi.GPIO as GPIO
    import Adafruit_ADS1x15
    GPIO.setmode(GPIO.BCM)
except Exception as e:
    print("Test ", e)

from time import sleep
import numpy as np

MAXIMUM = 32767.0

class tester:
    def __init__(self):
        pass

    def read_adc(self, channel, gain = 1):
        return np.random.rand()*MAXIMUM

    def change(self, value):
        self.value = value

class controller:
    def __init__(self, pot_pins):
```

```
        self.maximum = MAXIMUM
        self.gain_voltage = 4.096
        self.GAIN = 1

        try:
            self.pot = potentiometer(pot_pins)
        except:
            self.pot = tester()

        try:
            self.ADC = Adafruit_ADS1x15.ADS1115()
        except NameError:
            self.ADC = tester()

        self.temperature, self.reference = self.from_bits_to_value(0)
            , self.from_bits_to_value(1)

        self.pot.change(100)
        self.pot.change(0)

        self.v_min = 0
        self.v_max = 3.3

        self.desired_temperature = 0

    def from_bits_to_value(self, channel):
        temp = self.ADC.read_adc(channel, gain=self.GAIN)
        return (temp*self.gain_voltage*100)/self.maximum

    def read(self):
        self.temperature = self.from_bits_to_value(0)
        self.reference = self.from_bits_to_value(1)
        print(self.temperature, self.reference)
        return self.temperature, self.reference

    def change_pot(self, value):
        self.pot.change(value)

    def set_desired(self, value):
        value = value/100
        if value != self.desired_temperature:
            print(value, 'here')
            self.desired_temperature = value

            m = 100*(self.desired_temperature-self.v_min)/(self.v_max
                -self.v_min)
            m = int(np.around(m))
            self.change_pot(m)

    def close(self):
        GPIO.cleanup()

class potentiometer:
    def __init__(self, pins):
        self.pins = {"IC": pins[0], "UD": pins[1], "CS": pins[2]}
        self.value = 0
```

```
        self.time = 0.001

        for pin in self.pins:
            GPIO.setup(self.pins[pin], GPIO.OUT)
            GPIO.output(self.pins[pin], GPIO.LOW)

        GPIO.output(self.pins["IC"], GPIO.HIGH)

    def increase(self, value):
        GPIO.output(self.pins["UD"], GPIO.HIGH)
        while self.value < value:
            GPIO.output(self.pins["IC"], GPIO.LOW)
            sleep(self.time)
            GPIO.output(self.pins["IC"], GPIO.HIGH)
            sleep(self.time)
            self.value += 1

    def decrease(self, value):
        GPIO.output(self.pins["UD"], GPIO.LOW)
        while self.value > value:
            GPIO.output(self.pins["IC"], GPIO.LOW)
            sleep(self.time)
            GPIO.output(self.pins["IC"], GPIO.HIGH)
            sleep(self.time)
            self.value -= 1

    def change(self, value):
        if self.value > value:
            self.decrease(value)
        else:
            self.increase(value)
```

Los códigos del proyecto así como los datos obtenidos para las curvas de calibración se encuentran publicados en GitHub, y hacen parte del repositorio `supreme-pi` como parte de un proyecto de *physical computing* [1].

## REFERENCIAS

1. B. Juan, *supreme-pi*, GitHub repository, https://github.com/jsbarbosa/supreme-pi

## CAPÍTULO 3

# MEJORAS

Las mejoras al sistema de control se centran en los parámetros del PID. En ese sentido se considera que la mayor mejora se encuentra en la aplicación de algoritmos que permitan determinar los valores ideales midiendo iterativamente la respuesta del sistema para distintos valores de $K_P$, $K_I$ y $K_D$.

Es necesario tener en cuenta que la familia de potenciometros usados X9Cxxx solo pueden tener una diferencia de potencial entre $\pm 5$ V y 0 V entre las terminales de la resistencia, razón por la cual es necesario recurrir a otras referencias de potenciometros digitales que no tengan esta limitante.