Experiment Report: PID Motor Control with Arduino and Python GUI

Ziqi Wang*

July 18, 2024

1 Introduction

The purpose of this experiment is to implement a Proportional-Integral-Derivative (PID) controller for motor control using an Arduino board and a Python-based Graphical User Interface (GUI). The PID controller is a widely used feedback control algorithm that adjusts the system's output to reach a desired setpoint by tuning three parameters: proportional, integral, and derivative.

2 Equipment

- Arduino development board
- DC motor
- Precision potentiometer
- Computer with Arduino IDE and Python environment

3 Principle of PID Control

PID control involves three separate parameters: Proportional (P), Integral (I), and Derivative (D). Each parameter influences the controller in a different way:

- **Proportional (P):** This term produces an output value that is proportional to the current error value. The proportional response can be adjusted by multiplying the error by a constant known as the proportional gain (K_p). If the proportional gain is too high, the system can become unstable. The proportional term helps to reduce the rise time.
- **Integral (I):** This term is concerned with the accumulation of past errors. If the error has been present for a long time, the integral term increases the output to eliminate the residual steady-state error. The contribution from the integral term is adjusted by the integral gain (K_i). A high integral gain can lead to overshoot and increased settling time. The integral term helps to eliminate the steady-state error.
- **Derivative (D):** This term predicts system behavior by considering the rate of change of the error. The derivative response can be adjusted by the derivative gain (K_d). A high derivative gain can reduce overshoot, but too high a value can make the system sensitive to noise. The derivative term helps to reduce the overshoot and improve the system stability.

The overall control signal is given by the sum of these three terms:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

where $e(t)$ is the error between the setpoint and the process variable.

*wangzq@shanghaitech.edu.cn

3.1 Adjusting PID Parameters

To achieve optimal control performance, you can adjust the PID parameters:

- **Increasing K_p:** Reduces rise time but may cause overshoot and oscillation.
- **Increasing K_i:** Eliminates steady-state error but may increase overshoot and settling time.
- **Increasing K_d:** Reduces overshoot and improves stability but may make the system sensitive to noise.

When tuning the PID controller:

1. Start with K_p and increase it until the system responds adequately but without significant overshoot.
2. Add K_i to eliminate steady-state error. Increase it gradually while monitoring the system's response.
3. Finally, adjust K_d to reduce overshoot and dampen the oscillations.

4 Experiment Steps

1. Connect the Arduino and motor according to the circuit diagram.
2. Write the Arduino code to implement the PID control algorithm.
3. Develop the Python code to create a GUI for real-time PID parameter adjustment and motor state monitoring.
4. Use the GUI to tune the PID parameters and observe the motor's response.
5. Adjust the parameters as needed to achieve optimal control.

5 Arduino Code

The following Arduino code implements the PID control algorithm.

```
/*
==== The program will not run until the serial port is initialized! ====

Set PLOTTER to 1 for visualization with the plotter,
set to 0 for manual observation through the serial monitor

Method to adjust PID parameters via serial monitor:
1. Open the serial monitor with a default baud rate of 115200
2. Enter the following commands to adjust parameters:
   - p=1.5: Set P coefficient to 1.5
   - i=0.5: Set I coefficient to 0.5
   - d=0.1: Set D coefficient to 0.1
   - s=100: Set the setpoint to 100
3. After entering the commands, view the current parameter values
*/

#define PLOTTER 1
#define REVERSE 1

#define LED 12
#define STBY 7

#if REVERSE
#define N 8
```

```
#define P 9
#else
#define N 9
#define P 8
#endif

#define PWM 10
#define SetPoint A1
#define Sensor A0

#define delay_time 20

float kp = 3;            // P coefficient
float ki = 0.5;          // I coefficient
float kd = 0.1;          // D coefficient
float e_sum = 0;         // Error sum for integral calculation
float e_last = 0;        // Previous error for derivative calculation
unsigned long last_time; // Last update time

const int MIN_PWM = 68;           // Minimum PWM value for motor rotation
const int POSITION_THRESHOLD = 5; // Position change threshold
const int CONTROL_THRESHOLD = 5;  // Control signal threshold

int set_point = 0;
int prev_set_point = 0;
int prev_pos = 0;
bool serial_control = false;      // Serial control flag

// PID control function
float pid(float pos_error)
{
  unsigned long now = millis();
  float dt = (now - last_time) / 1000.0;
  last_time = now;
  e_sum += pos_error * dt;
  float e_diff = (pos_error - e_last) / dt;
  float output = kp * pos_error + ki * e_sum + kd * e_diff;
  e_last = pos_error;
  return output;
}

void setup()
{
  Serial.begin(115200);
  while (!Serial)
    ;
  pinMode(LED, OUTPUT);
  pinMode(STBY, OUTPUT);
  pinMode(N, OUTPUT);
  pinMode(P, OUTPUT);
  pinMode(PWM, OUTPUT);
  pinMode(SetPoint, INPUT);
  pinMode(Sensor, INPUT);

  last_time = millis(); // Initialize time
}

void loop()
```

```
{
  if (Serial.available() > 0)
  {
    String input = Serial.readStringUntil('\n');
    parseInput(input);
  }

  int curr_pos = analogRead(Sensor); // Read current sensor value

  if (!serial_control)
  {
    // If analog control, update the setpoint
    set_point = analogRead(SetPoint);
  }

  int ready = 0;

  // Check if the setpoint is reached
  if (abs(set_point - curr_pos) < 20)
  {
    ready = 1;
    digitalWrite(LED, 1); // Light up the LED if setpoint is reached
  }
  else
  {
    ready = 0;
    digitalWrite(LED, 0);
  }

  if (PLOTTER)
  {
    Serial.print(float(set_point) / 10.24);
    Serial.print(",");
    Serial.print(float(curr_pos) / 10.24);
    Serial.print(",");
    Serial.print(kp);
    Serial.print(",");
    Serial.print(ki);
    Serial.print(",");
    Serial.print(kd);
    Serial.print(",");
    Serial.print(ready);
    Serial.print("\n");
  }
  else
  {
    Serial.print("Set Point: ");
    Serial.print(set_point / 10.24);

    Serial.print(" Current Position: ");
    Serial.print(curr_pos / 10.24);

    Serial.print(" P: ");
    Serial.print(kp);
    Serial.print(" I: ");
    Serial.print(ki);
    Serial.print(" D: ");
    Serial.print(kd);
```

```
    Serial.print(" Ready: ");
    Serial.print(ready);

    Serial.print("\n");
  }

  float pos_error = set_point - curr_pos;           // Calculate error
  float control_error = set_point - prev_set_point; // Calculate control error
  float control_signal = pid(pos_error);            // Calculate PID control signal

  // Constrain control signal between -255 and 255
  control_signal = constrain(control_signal, -255, 255);

  // Ensure control signal absolute value is not below the minimum PWM value
  if (control_signal > 40)
  {
    control_signal = max(control_signal, MIN_PWM);
  }
  else if (control_signal < -40)
  {
    control_signal = min(control_signal, -MIN_PWM);
  }

  if (abs(pos_error) < POSITION_THRESHOLD && abs(control_error) < CONTROL_THRESHOLD)
  {
    digitalWrite(STBY, 0); // Turn off motor driver
  }
  else
  {
    digitalWrite(STBY, 1); // Enable motor driver

    if (control_signal > 0)
    {
      digitalWrite(N, 0);
      digitalWrite(P, 1);
      analogWrite(PWM, control_signal);
    }
    else
    {
      digitalWrite(N, 1);
      digitalWrite(P, 0);
      analogWrite(PWM, -control_signal);
    }

    prev_set_point = set_point;
    prev_pos = curr_pos;

    delay(delay_time); // Short delay to avoid overly frequent control signal updates
  }
}

void parseInput(String input)
{
  input.trim(); // Remove leading and trailing whitespace

  if (input.startsWith("p="))
  {
```

```
    kp = input.substring(2).toFloat();
    Serial.print("Updated kp to ");
    Serial.println(kp);
  }
  else if (input.startsWith("i="))
  {
    ki = input.substring(2).toFloat();
    Serial.print("Updated ki to ");
    Serial.println(ki);
  }
  else if (input.startsWith("d="))
  {
    kd = input.substring(2).toFloat();
    Serial.print("Updated kd to ");
    Serial.println(kd);
  }
  else if (input.startsWith("s="))
  {
    int sp = input.substring(2).toInt();
    if (sp == -1)
    {
      serial_control = false; // Switch to analog control
      Serial.println("Switched to analog control");
    }
    else if (sp >= 0 && sp <= 100)
    {
      set_point = sp * 10.24; // Map 0-100 value to 0-1023 range
      serial_control = true; // Set serial control flag
      Serial.print("Updated set point to ");
      Serial.println(set_point);
    }
    else
    {
      Serial.println("Invalid set point value");
    }
  }
  else
  {
    Serial.println("Invalid input");
  }
}
```

6 Python GUI Code

The following Python code creates a GUI using Tkinter for real-time PID parameter adjustment and motor state monitoring.

```
import tkinter as tk
from tkinter import ttk, scrolledtext
import serial
import threading
import time
import serial.tools.list_ports

class PIDControllerApp:
    def __init__(self, root):
        self.root = root
        self.root.title("PID Controller")
```

```
        self.serial_port_manager = SerialPortManager(self)
        self.running = False

        self.create_widgets()
        self.refresh_ports()

    def create_widgets(self):
        self.port_label = tk.Label(self.root, text="Select Port:")
        self.port_label.grid(row=0, column=0)

        self.port_combobox = ttk.Combobox(self.root)
        self.port_combobox.grid(row=0, column=1)

        self.connect_button = tk.Button(self.root, text="Connect", command=self.connect_serial)
        self.connect_button.grid(row=0, column=2)

        self.connection_status = tk.Label(self.root, text="Not Connected", bg="grey")
        self.connection_status.grid(row=0, column=3)

        self.kp_label = tk.Label(self.root, text="Kp")
        self.kp_label.grid(row=1, column=0)
        self.kp_scale = tk.Scale(self.root, from_=0, to=10, resolution=0.1, orient=tk.HORIZONTAL)
        self.kp_scale.grid(row=1, column=1)
        self.kp_button = tk.Button(self.root, text="Set Kp", command=self.update_kp)
        self.kp_button.grid(row=1, column=2)

        self.ki_label = tk.Label(self.root, text="Ki")
        self.ki_label.grid(row=2, column=0)
        self.ki_scale = tk.Scale(self.root, from_=0, to=10, resolution=0.1, orient=tk.HORIZONTAL)
        self.ki_scale.grid(row=2, column=1)
        self.ki_button = tk.Button(self.root, text="Set Ki", command=self.update_ki)
        self.ki_button.grid(row=2, column=2)

        self.kd_label = tk.Label(self.root, text="Kd")
        self.kd_label.grid(row=3, column=0)
        self.kd_scale = tk.Scale(self.root, from_=0, to=10, resolution=0.1, orient=tk.HORIZONTAL)
        self.kd_scale.grid(row=3, column=1)
        self.kd_button = tk.Button(self.root, text="Set Kd", command=self.update_kd)
        self.kd_button.grid(row=3, column=2)

        self.setpoint_label = tk.Label(self.root, text="Set Point")
        self.setpoint_label.grid(row=4, column=0)
        self.setpoint_entry = tk.Entry(self.root)
        self.setpoint_entry.grid(row=4, column=1)
        self.setpoint_button = tk.Button(self.root, text="Set Set Point", \
            command=self.update_setpoint)
        self.setpoint_button.grid(row=4, column=2)

        self.current_position_label = tk.Label(self.root, text="Current Position:")
        self.current_position_label.grid(row=5, column=0)
        self.current_position_value = tk.Label(self.root, text="0%")
        self.current_position_value.grid(row=5, column=1)

        self.setpoint_display_label = tk.Label(self.root, text="Set Point Display:")
        self.setpoint_display_label.grid(row=6, column=0)
        self.setpoint_display_value = tk.Label(self.root, text="0%")
        self.setpoint_display_value.grid(row=6, column=1)
```

```
        self.kp_display_label = tk.Label(self.root, text="Kp Display:")
        self.kp_display_label.grid(row=7, column=0)
        self.kp_display_value = tk.Label(self.root, text="0")
        self.kp_display_value.grid(row=7, column=1)

        self.ki_display_label = tk.Label(self.root, text="Ki Display:")
        self.ki_display_label.grid(row=8, column=0)
        self.ki_display_value = tk.Label(self.root, text="0")
        self.ki_display_value.grid(row=8, column=1)

        self.kd_display_label = tk.Label(self.root, text="Kd Display:")
        self.kd_display_label.grid(row=9, column=0)
        self.kd_display_value = tk.Label(self.root, text="0")
        self.kd_display_value.grid(row=9, column=1)

        self.led_label = tk.Label(self.root, text="LED Status:")
        self.led_label.grid(row=10, column=0)
        self.led_status = tk.Label(self.root, text="TUNING", bg="grey")
        self.led_status.grid(row=10, column=1)

        self.canvas = tk.Canvas(self.root, width=200, height=200, bg="white")
        self.canvas.grid(row=0, column=4, rowspan=11)
        self.arc = self.canvas.create_arc(50, 50, 150, 150,  \
            start=90, extent=0, outline="blue", width=2)
        self.set_point_arc = self.canvas.create_arc(50, 50,  \
            150, 150, start=90, extent=0, outline="red", width=2)

        # Add scrolled text box for serial output
        self.text_box = scrolledtext.ScrolledText(self.root, width=50, height=10, state='disabled')
        self.text_box.grid(row=11, column=0, columnspan=5, padx=10, pady=10)

    def refresh_ports(self):
        ports = self.get_serial_ports()
        self.port_combobox['values'] = ports
        self.root.after(1000, self.refresh_ports)  # Refresh the port list every 1 second

    def get_serial_ports(self):
        ports = serial.tools.list_ports.comports()
        return [port.device for port in ports]

    def connect_serial(self):
        if self.serial_port_manager.is_running:
            self.disconnect_serial()
        else:
            selected_port = self.port_combobox.get()
            if selected_port:
                try:
                    self.serial_port_manager.set_name(selected_port)
                    self.serial_port_manager.set_baud(115200)
                    self.serial_port_manager.start()
                    self.connect_button.config(text="Disconnect")
                    self.connection_status.config(text="Connected", bg="green")
                    self.running = True
                    self.recursive_update_textbox()
                    self.read_initial_data()
                except serial.SerialException:
                    self.connection_status.config(text="Connection Failed", bg="red")
```

```
            else:
                self.connection_status.config(text="No Port Selected", bg="red")

    def disconnect_serial(self):
        self.serial_port_manager.stop()
        self.connect_button.config(text="Connect")
        self.connection_status.config(text="Not Connected", bg="grey")
        self.running = False

    def update_kp(self):
        if self.serial_port_manager.is_running:
            value = self.kp_scale.get()
            command = f"p={value}\n"
            print(f"Sending command: {command}")
            self.serial_port_manager.write(command.encode())

    def update_ki(self):
        if self.serial_port_manager.is_running:
            value = self.ki_scale.get()
            command = f"i={value}\n"
            print(f"Sending command: {command}")
            self.serial_port_manager.write(command.encode())

    def update_kd(self):
        if self.serial_port_manager.is_running:
            value = self.kd_scale.get()
            command = f"d={value}\n"
            print(f"Sending command: {command}")
            self.serial_port_manager.write(command.encode())

    def update_setpoint(self):
        if self.serial_port_manager.is_running:
            value = self.setpoint_entry.get()
            try:
                setpoint = float(value)
                if 0 <= setpoint <= 100:
                    scaled_value = int(setpoint)
                    command = f"s={scaled_value}\n"
                    print(f"Sending command: {command}")
                    self.serial_port_manager.write(command.encode())
                else:
                    print("Setpoint out of range (0-100)")
            except ValueError:
                print("Invalid setpoint value")

    def read_initial_data(self):
        line = self.serial_port_manager.read_line().decode('utf-8').strip()
        if line:
            print(f"Initial data: {line}")  # Debug print
            data = line.split(",")
            if len(data) == 6:
                set_point, curr_pos, kp, ki, kd, ready = data
                self.setpoint_display_value.config(text=f"{set_point}%")
                self.current_position_value.config(text=f"{curr_pos}%")
                self.kp_display_value.config(text=kp)
                self.ki_display_value.config(text=ki)
                self.kd_display_value.config(text=kd)
                if ready == "1":
```

```
                    self.led_status.config(text="READY", bg="green")
                else:
                    self.led_status.config(text="TUNING", bg="yellow")
                self.update_arc(set_point, curr_pos)

    def update_arc(self, set_point, curr_pos):
        # Update red arc
        set_point_extent = (float(set_point) / 100) * 360
        self.canvas.itemconfig(self.set_point_arc, extent=set_point_extent)

        # Update blue arc
        current_pos_extent = (float(curr_pos) / 100) * 360
        self.canvas.itemconfig(self.arc, extent=current_pos_extent)

    def recursive_update_textbox(self):
        serial_port_buffer = self.serial_port_manager.read_buffer()
        if serial_port_buffer:  # Ensure there's something to decode and insert
            self.text_box.config(state='normal')
            self.text_box.insert(tk.END, serial_port_buffer.decode("ascii"))
            self.text_box.see(tk.END)
            self.text_box.config(state='disabled')
        if self.serial_port_manager.is_running:
            self.root.after(100, self.recursive_update_textbox)

    def on_closing(self):
        self.running = False
        if self.serial_port_manager.is_running:
            self.serial_port_manager.stop()
        self.root.destroy()


class SerialPortManager:
    def __init__(self, app):
        self.is_running = False
        self.serial_port_name = None
        self.serial_port_baud = 9600
        self.serial_port = serial.Serial()
        self.serial_port_buffer = bytearray()
        self.line_buffer = ""
        self.app = app

    def set_name(self, serial_port_name):
        self.serial_port_name = serial_port_name

    def set_baud(self, serial_port_baud):
        self.serial_port_baud = serial_port_baud

    def start(self):
        self.is_running = True
        self.serial_port_thread = threading.Thread(target=self.thread_handler)
        self.serial_port_thread.start()

    def stop(self):
        self.is_running = False

    def thread_handler(self):
        while self.is_running:
            try:
```

```
            if not self.serial_port.isOpen():
                self.serial_port = serial.Serial(
                    port=self.serial_port_name,
                    baudrate=self.serial_port_baud,
                    bytesize=8,
                    timeout=2,
                    stopbits=serial.STOPBITS_ONE,
                )
            else:
                while self.serial_port.in_waiting > 0:
                    serial_port_byte = self.serial_port.read(1)
                    self.serial_port_buffer.append(int.from_bytes(serial_port_byte, \
                        byteorder='big'))
                    self.line_buffer += serial_port_byte.decode('utf-8')

                    if '\n' in self.line_buffer:
                        lines = self.line_buffer.split('\n')
                        for line in lines[:-1]:
                            self.update_app(line.strip())
                        self.line_buffer = lines[-1]
        except serial.SerialException as e:
            print(f"Serial error: {e}")
            self.app.disconnect_serial()

    if self.serial_port.isOpen():
        self.serial_port.close()

def update_app(self, data_line):
    if data_line:
        print(f"Update app with data: {data_line}")  # Debug print
        data = data_line.split(",")
        if len(data) == 6:
            set_point, curr_pos, kp, ki, kd, ready = data
            self.app.setpoint_display_value.config(text=f"{set_point}%")
            self.app.current_position_value.config(text=f"{curr_pos}%")
            self.app.kp_display_value.config(text=kp)
            self.app.ki_display_value.config(text=ki)
            self.app.kd_display_value.config(text=kd)
            if ready == "1":
                self.app.led_status.config(text="READY", bg="green")
            else:
                self.app.led_status.config(text="TUNING", bg="yellow")
            self.app.update_arc(set_point, curr_pos)

def read_buffer(self):
    buffer = self.serial_port_buffer
    self.serial_port_buffer = bytearray()
    return buffer

def write(self, data):
    if self.serial_port.isOpen():
        self.serial_port.write(data)

def read_line(self):
    if self.serial_port.isOpen():
        return self.serial_port.readline()
    return b''
```

```
    def main_process(self, input_byte):
        try:
            character = input_byte.decode("ascii")
        except UnicodeDecodeError:
            pass
        else:
            print(character, end="")


if __name__ == "__main__":
    root = tk.Tk()
    app = PIDControllerApp(root)
    root.protocol("WM_DELETE_WINDOW", app.on_closing)
    root.mainloop()
```

7 Python Libraries

The following libraries were used in the Python code:

- **serial:** This library is used for serial communication between the Arduino and the Python GUI. For more details, visit `https://pypi.org/project/pyserial/`.
- **threading:** This library is used to handle serial communication in a separate thread to avoid blocking the GUI. For more details, visit `https://docs.python.org/3/library/threading.html`.
- **Scrolledtext:** This library is used to create a scrollable text box for displaying serial output. For more details, visit `https://docs.python.org/3/library/tkinter.scrolledtext.html`.

8 GUI Implementation and Functionality

The Python GUI was implemented using Tkinter and provides the following functionalities:

- **Serial Port Connection:** Allows the user to select and connect to the appropriate serial port.
- **Parameter Adjustment:** Provides sliders and buttons to adjust the PID parameters (K_p, K_i, K_d) and the setpoint.
- **Real-time Monitoring:** Displays the current position, setpoint, and PID parameters in real-time.
- **LED Status:** Shows the motor's status as either "READY" or "TUNING" based on the motor's position relative to the setpoint.
- **Visualization:** Uses arcs to visualize the setpoint and current position on a canvas.
- **Scrollable Text Box:** Provides a scrollable text box to display the serial output from the Arduino.

9 Packaging Python Program as Executable

In this section, we will discuss how to package the Python GUI program into a standalone executable (.exe) file using *PyInstaller*. This process allows the program to be distributed and run on computers without requiring a separate Python installation.

9.1 Installing PyInstaller

First, you need to install *PyInstaller*. This can be done using the following command:

```
pip install pyinstaller
```

9.2 Creating the Executable

Assuming your Python file is named `PIDControllerApp.py`, you can create an executable using the following command:

```
pyinstaller --onefile --windowed PIDControllerApp.py
```

Explanation of the parameters:

- `--onefile`: Tells *PyInstaller* to bundle everything into a single executable file.
- `--windowed`: Creates a GUI application without a console window (only for Windows).

9.3 Including Additional Data Files and Icons

If your program requires additional data files or a custom icon, you can include them during the packaging process.

```
pyinstaller --onefile --windowed --icon=icon.ico PIDControllerApp.py
```

9.4 Complete Example

Here is a complete example assuming your program file is named `PIDControllerApp.py` and you wish to use a custom icon `icon.ico`:

```
pip install pyinstaller
pyinstaller --onefile --windowed --icon=icon.ico PIDControllerApp.py
```

9.5 Output

The generated executable will be located in the `dist` folder within your project directory. You can run the `PIDControllerApp.exe` file directly to launch your application.

9.6 Troubleshooting

If you encounter any issues during the packaging process, refer to the *PyInstaller* documentation for more detailed guidance: PyInstaller Documentation.

Common troubleshooting steps include:

- Ensuring all necessary Python libraries are installed.
- Checking the log files generated in the `build` folder for any error messages.

10 Conclusion

This experiment successfully demonstrates the implementation of a PID controller for motor control using an Arduino and a Python-based GUI. The system allows real-time adjustment and monitoring of PID parameters, providing a valuable tool for understanding and tuning PID controllers.