UNIVERSIDADE FEDERAL DE UBERLÂNDIA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

EXPERIMENTAL DE SISTEMAS DE CONTROLE REALIMENTADO

Davi de Mélo Cordeiro - 12211EAU004
Davi Iwanow - 12211EAU023
Letícia Sanches Guimarães de Sousa - 12021EBI002

Projeto Final

Controle de Temperatura e Vazão

UBERLÂNDIA – MG
2025

DAVI DE MÉLO CORDEIRO
DAVI IWANOW
LETÍCIA SANCHES GUIMARÃES DE SOUSA

EXPERIMENTAL DE SISTEMAS DE CONTROLE REALIMENTADO
CONTROLE DE TEMPERATURA

Trabalho final entregue à Professora Doutora Gabriela Vieira Lima sobre a disciplina de Experimental de Sistemas de Controle Realimentado do curso de Engenharia Elétrica, sob orientação dela.

UBERLÂNDIA
2025

SUMÁRIO

Índice de ilustrações

1. Objetivos

O projeto final da disciplina de Experimental de Sistemas de Controle Realimentado consistiu no desenvolvimento de um sistema de controle capaz de regular a temperatura e vazão de um ambiente. Para isso, primeiramente foi necessário a montagem de um ambiente físico para o sistema. Em seguida, foram realizadas as etapas de modelagem, projeto, implementação e, por fim, montagem de um controlador que atendesse aos requisitos de estabilidade e desempenho exigidos para o controle simultâneo dessas variáveis. Ao longo do projeto, buscou-se integrar os conhecimentos teóricos adquiridos ao longo do curso com a prática experimental em laboratório, promovendo a compreensão dos conceitos estudados em sala de aula.

2. Introdução

Como dito acima, o projeto final consistiu em desenvolver um sistema de controle de temperatura e vazão, conforme solicitado. Primeiramente, para o início da compreensão, pode-se dizer que um sistema é uma disposição, conjunto ou coleção de partes conectadas ou relacionadas de tal maneira a formarem um todo, e o controle estuda como agir sobre esse dado sistema de modo a obter um resultado arbitrariamente especificado [1]. Nesse sentido, um sistema de controle consiste em subsistemas e processos construídos com o objetivo de se obter uma saída desejada com um desempenho desejado, dada uma entrada especificada.

Em primeiro plano, vale complementar que em sistemas de controle, os controladores PD (Proporcional Derivativo), PI (Proporcional Integral) e PID (Proporcional Integral Derivativo) são de extrema importância e podem projetados por meio de ferramentas disponibilizadas pelo MATLAB, principalmente a ferramenta Control System Designer (Sisotool), pois ela oferece muitos recursos que facilitam a análise e o design de controladores. Além disso, também é utilizado o Simulink, responsável pela simulação e modelagem de sistemas por meio de blocos gráficos - ambas ferramentas foram utilizadas para a realização desse projeto.

Em complemento para a compreensão do projeto, pode-se afirmar que o controlador PD combina uma ação proporcional, que reage diretamente ao erro atual, com uma ação derivativa, que antecipa o comportamento futuro do erro com base em sua taxa de variação. Isso o torna útil para melhorar a resposta dinâmica do sistema, reduzindo o tempo de acomodação e aumentando a estabilidade. Já o controlador PI atua corrigindo o erro atual pela ação proporcional e eliminando o erro em regime permanente através da ação integral, que acumula o erro ao longo do tempo e ajusta o controle de forma contínua; é muito utilizado em sistemas onde a precisão final é mais importante que a velocidade de resposta [2].

O controlador PID combina as três ações, proporcional, integral e derivativa, proporcionando uma resposta balanceada, com rápida correção do erro, eliminação do erro permanente e previsão de tendências futuras. O PID é amplamente adotado por sua versatilidade e por oferecer um desempenho robusto em uma grande variedade de sistemas.

Além disso, é importante complementar que para a implementação digital destes controladores é necessário realizar a discretização das equações de controle encontradas. Isso porque, microcontroladores, como ESP 32 e Arduino, operam de forma discreta, ou seja, eles processam e coletam informações em intervalos de tempo pré-definidos. Dessa maneira, as

equações projetadas no domínio contínuo de Laplace devem ser adaptadas para o domínio discreto, utilizando equações de diferenças.

Assim, este relatório tem como finalidade descrever de forma detalhada todo o processo de desenvolvimento do sistema de controle de temperatura e vazão. Para isso, apresenta desde a elaboração do esquema elétrico e do layout até a montagem da Placa de Circuito Impresso. Além disso, contempla a identificação das plantas no domínio contínuo, o projeto dos controladores correspondentes e a posterior implementação digital do sistema.

3. Materiais e Sistema Físico

3.1. Materiais

Inicialmente, para a construção do projeto, foi necessário desenvolver circuitos eletrônicos que proporcionassem as condições adequadas para o funcionamento do sistema proposto. O primeiro passo consistiu na definição dos componentes eletrônicos responsáveis pela implementação das plantas de temperatura e vazão, bem como na elaboração dos respectivos circuitos de alimentação etc.

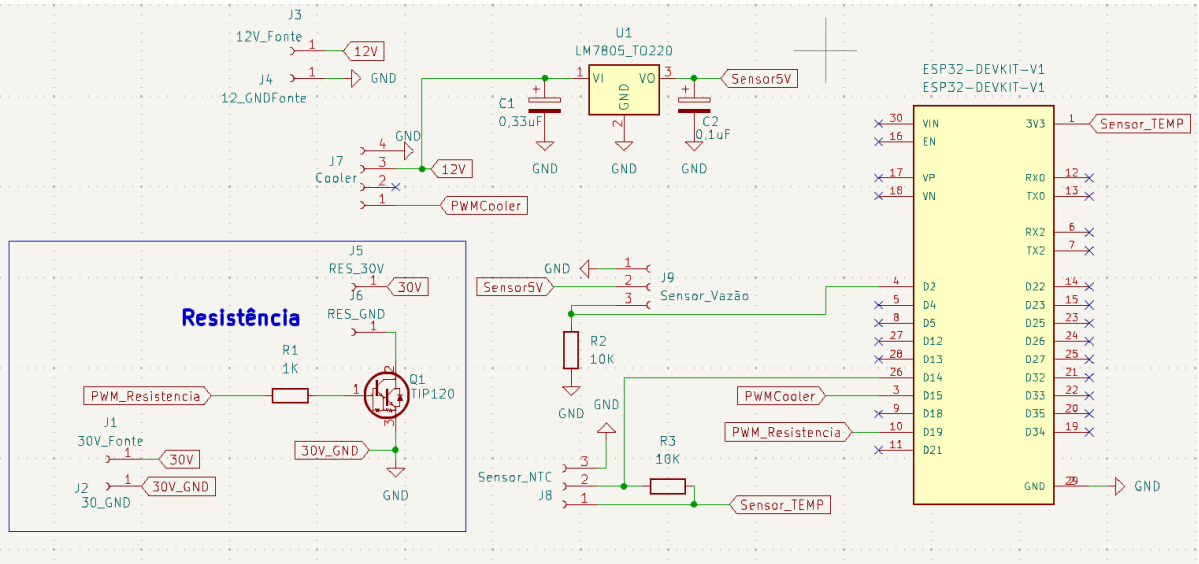
A lista completa dos componentes eletrônicos utilizados no projeto está apresentada a seguir:

• 1 ESP32 de 30 pinos;
• 1 Resistência de Chuveiro;
• 1 Cooler de 4 fios;
• 1 Sensor de Vazão YF-201;
• 1 Sensor de Temperatura de DS18B20 de três fios;
• 1 Regulador de tensão LM7805;
• 1 Capacitor de 0,33uF;
• 1 Capacitor de 0,1uF;
• 1 Conector pin-header de 4 pinos;
• 1 Conector pin-header de 3 pinos;
• 1 Conector borne KRE de 3 vias;
• 1 TIP120;
• 2 Resistores de 10KΩ;
• 1 Resistor de 1KΩ;
• 1 Caixa feita em impressora 3D.

3.2. Esquemático e PCB

Para o desenvolvimento do circuito eletrônico do sistema de controle de temperatura e vazão foi utilizado o Software KICAD. Primeiramente foi feito o esquemático a seguir:

Figura 1: Esquemático completo do circuito eletrônico.



Inicialmente, é importante destacar que a etapa referente à seleção de materiais e ao desenvolvimento do sistema físico foi realizada em conjunto com o grupo composto por Felipe Grossi, João Gabriel e Pedro Uga. Essa colaboração entre os grupos foi devidamente comunicada à Professora Gabriela. Com isso, o projeto desenvolvido conta com duas entradas de alimentação independentes para fontes externas: uma de 12V e outra de 30V. A entrada de 12V estão disponíveis nos conectores J3 e J4, enquanto as de 30V estão nos conectores J1 e J2. A fonte de 12V é usada para fornecer energia ao sensor de vazão YF-S201 e alimentar o cooler. Como esse sensor opera com tensões entre 5V e 24V, a tensão de 12V é reduzida para 5V utilizando o regulador LM7805.

Para a conexão do sensor de vazão, foi utilizado um conector do tipo pin-header, identificado no esquemático como J9. Neste conector, o pino 1 é a terra (GND), o pino 2 fornece a alimentação de 5V e o pino 3 é o sinal de saída do sensor. O sinal enviado pelo sensor passa por um divisor de tensão, composto pelo resistor R2 de 10kΩ, e é ativado por um transistor. Por fim, esse sinal é conectado ao pino D2 da ESP32, que realiza a leitura dos dados enviados pelo sensor.

Já a conexão do cooler a placa foi realizada por meio do conector pin-header J7, conforme indicado no esquema da Figura 1. O pino 1 desse conector é responsável pelo controle de velocidade do cooler por meio de sinal PWM, e por isso foi ligado ao pino D15 da ESP32. O pino 3 fornece os 12V necessários para alimentar o cooler, enquanto o pino 4 é destinado ao aterramento do componente.

A alimentação de 30V foi aplicada diretamente à resistência de chuveiro utilizada para o aquecimento do sistema. Essa resistência foi conectada em série com o transistor TIP120, que realiza a modulação da tensão média aplicada à carga por meio de controle PWM. A conexão da resistência e do transistor foi feita através dos conectores jack banana J1, J2, J5 e J6, conforme representado no esquemático.

Além disso, a conexão do sensor de temperatura foi realizada por meio de um conector Borne KRE, identificado no esquemático como Sensor_NTC em J8. O sensor foi alimentado com uma tensão de 3,3V fornecida diretamente pela própria ESP32. Para possibilitar a leitura do sinal, foi implementado um divisor de tensão utilizando o resistor R3 de 10kΩ, também indicado no esquemático. A saída desse divisor foi conectada ao pino D14 da ESP32, responsável pela leitura analógica da temperatura.

Por fim, a Figura 2 ilustra os *footprints* dos componentes utilizados, facilitando a compreensão da disposição física dos elementos na placa de circuito impresso (PCB).

Figura 2: Footprints dos componentes utilizados.

Símbolo: Atribuições do footprint

1	C1	0,33uF	Capacitor_THT:CP_Radial_D5.0mm_P2.00mm
2	C2	0,1uF	Capacitor_THT:CP_Radial_D5.0mm_P2.00mm
3	ESP32-DEVKIT-V1	ESP32-DEVKIT-V1	Libraryesp32:ESP32_DEVKIT_V1
4	J1	30V_Fonte	Connector:Banana_Jack_1Pin
5	J2	30_GND	Connector:Banana_Jack_1Pin
6	J3	12V_Fonte	Connector:Banana_Jack_1Pin
7	J4	12_GNDFonte	Connector:Banana_Jack_1Pin
8	J5	RES_30V	Connector:Banana_Jack_1Pin
9	J6	RES_GND	Connector:Banana_Jack_1Pin
10	J7	Cooler	Connector_PinHeader_2.54mm:PinHeader_1x04_P2.54mm_Vertical
11	J8	Sensor_NTC	TerminalBlock_Phoenix:TerminalBlock_Phoenix_MKDS-1,5-3-5.08_1x03_P5.08mm_Horizontal
12	J9	Sensor_Vazão	Connector_PinHeader_2.54mm:PinHeader_1x03_P2.54mm_Vertical
13	Q1	TIP120	Package_TO_SOT_THT:TO-220-3_Vertical
14	R1	1K	Resistor_THT:R_Axial_DIN0207_L6.3mm_D2.5mm_P15.24mm_Horizontal
15	R2	10K	Resistor_THT:R_Axial_DIN0207_L6.3mm_D2.5mm_P15.24mm_Horizontal
16	R3	10K	Resistor_THT:R_Axial_DIN0207_L6.3mm_D2.5mm_P15.24mm_Horizontal
17	U1	LM7805_TO220	Package_TO_SOT_THT:TO-220-3_Vertical

Em seguida, foi realizado o projeto da Placa de Circuito Impresso (PCB) utilizando o próprio software KiCad. As Figuras 3, 4 e 5 apresentam, respectivamente, o layout final da placa e sua visualização em modelo tridimensional (3D), permitindo uma melhor compreensão da disposição dos componentes e do roteamento das trilhas.

Figura 3: Layout final da placa.

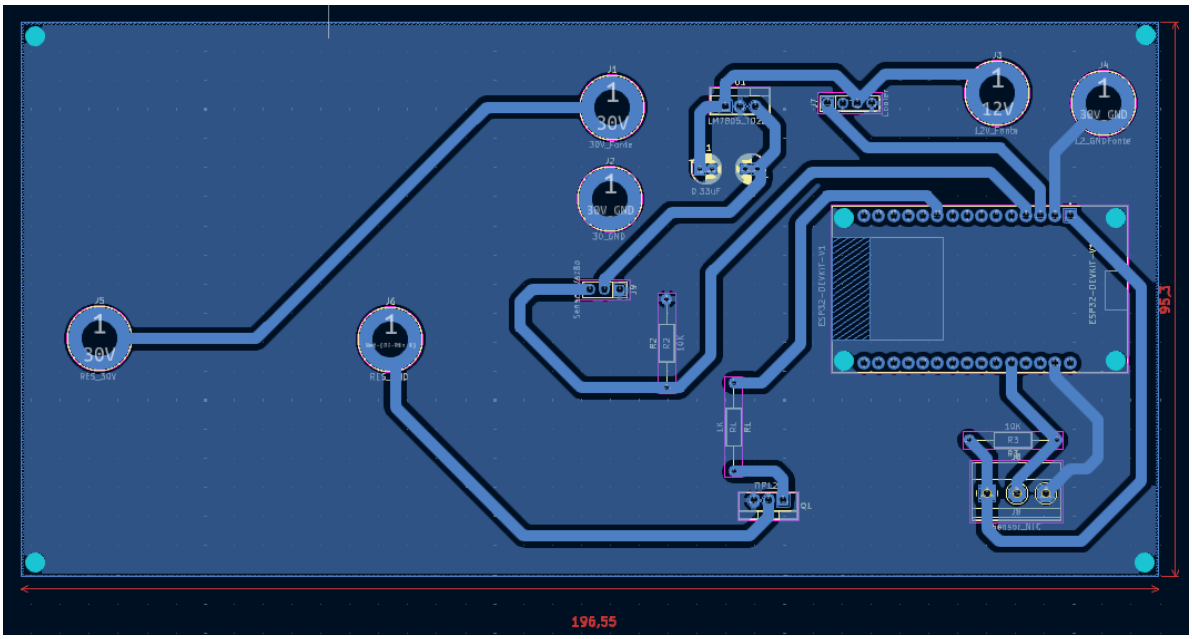


Figura 4: Modelo tridimensional visto de cima.

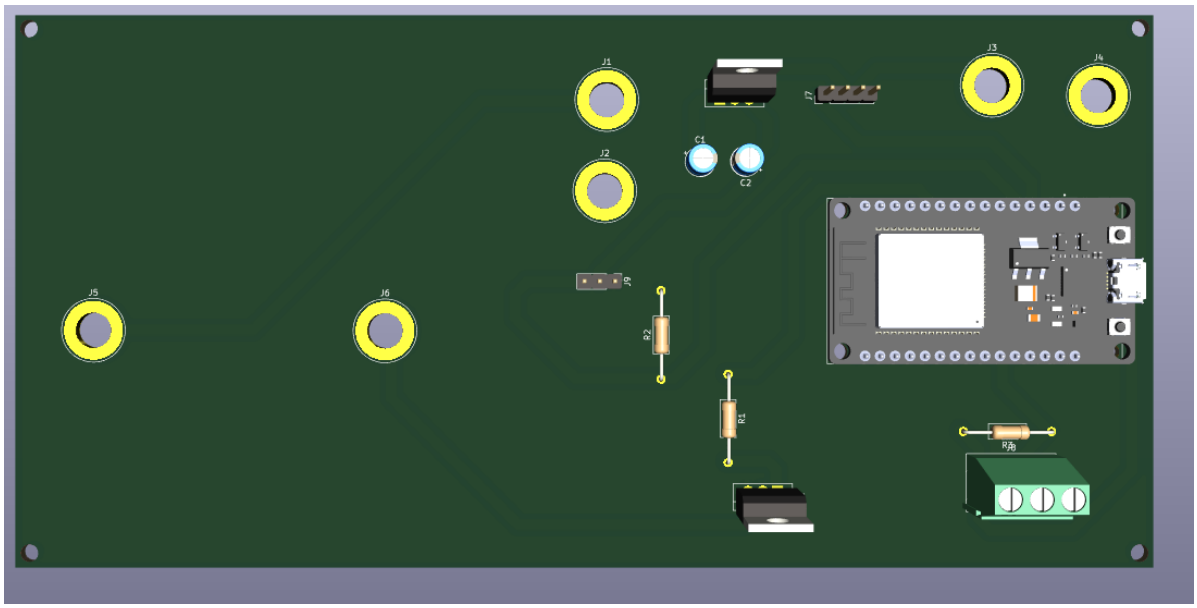
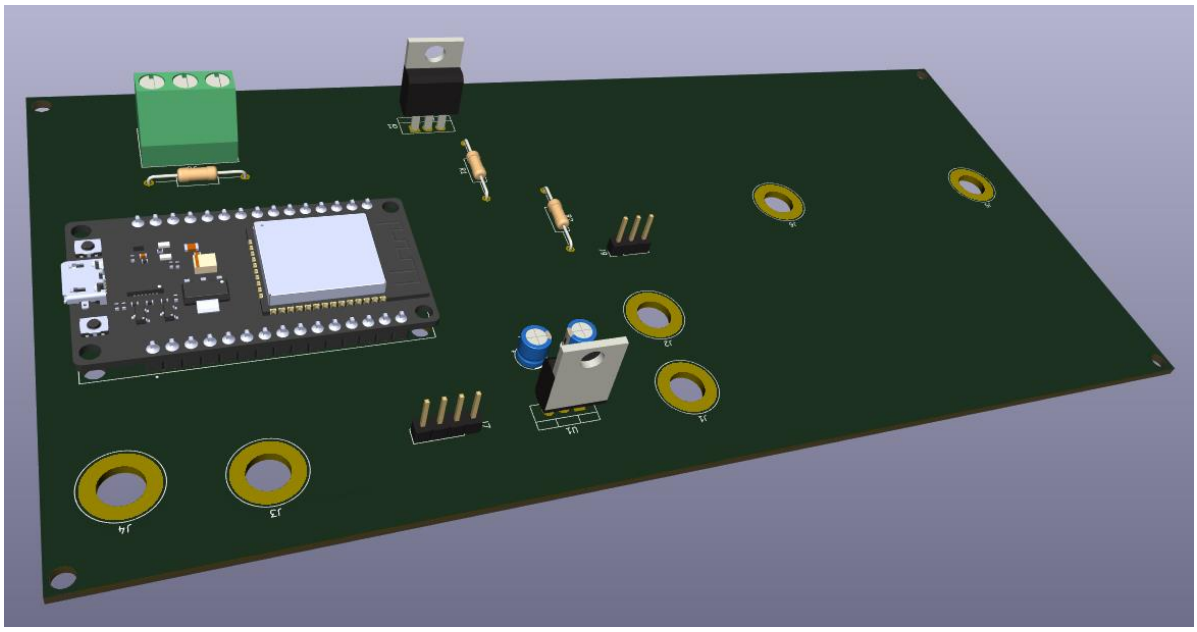


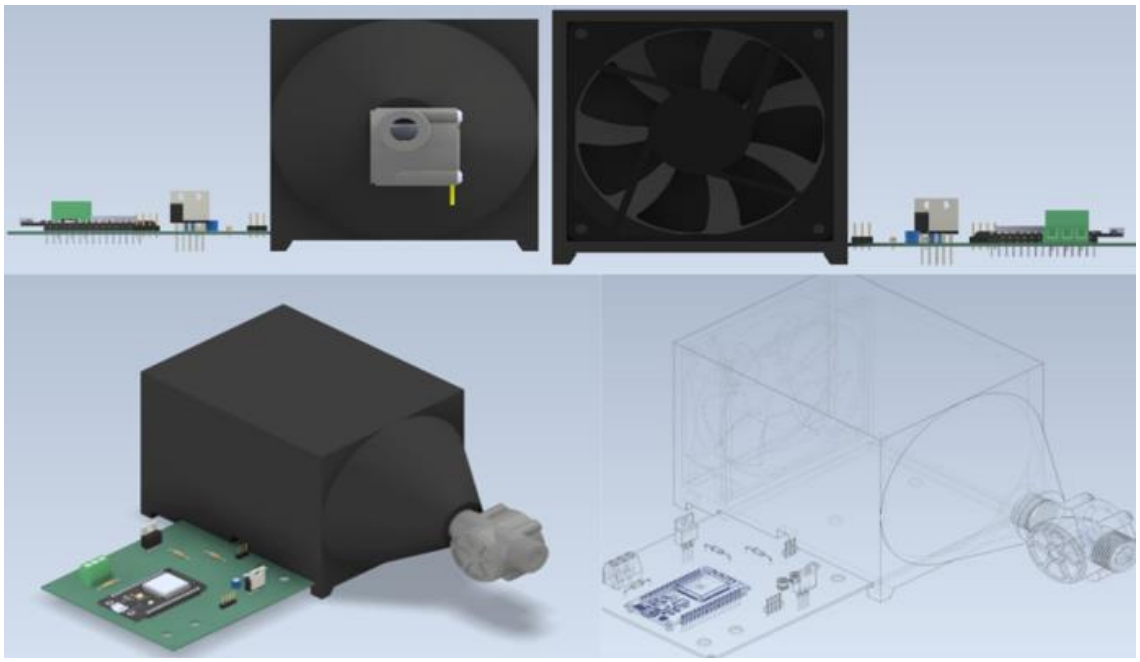
Figura 5: Modelo tridimensional visto de lado.



Para a construção do sistema físico, foi utilizada uma estrutura confeccionada em impressora 3D, modelada previamente no software Autodesk Inventor. O design da caixa foi desenvolvido de forma a acomodar todos os componentes do sistema de maneira funcional e organizada. Em uma das extremidades, foi instalado o cooler, responsável pela circulação de ar, enquanto na extremidade oposta foi fixado o sensor de vazão. Na lateral da estrutura, foi criada uma abertura destinada à passagem de parte da PCB, possibilitando o encaixe da resistência de aquecimento em uma configuração tipo "gaveta", facilitando sua remoção e manutenção. Além disso, foi feito um orifício na parte superior da caixa para a inserção do sensor de temperatura no interior do sistema.

A Figuras 6 apresenta, a estrutura final montada e o processo de modelagem da caixa.

Figura 6: Estrutura final montada e o processo de modelagem da caixa.



4. Identificação das Plantas

4.1. Ensaio das Plantas

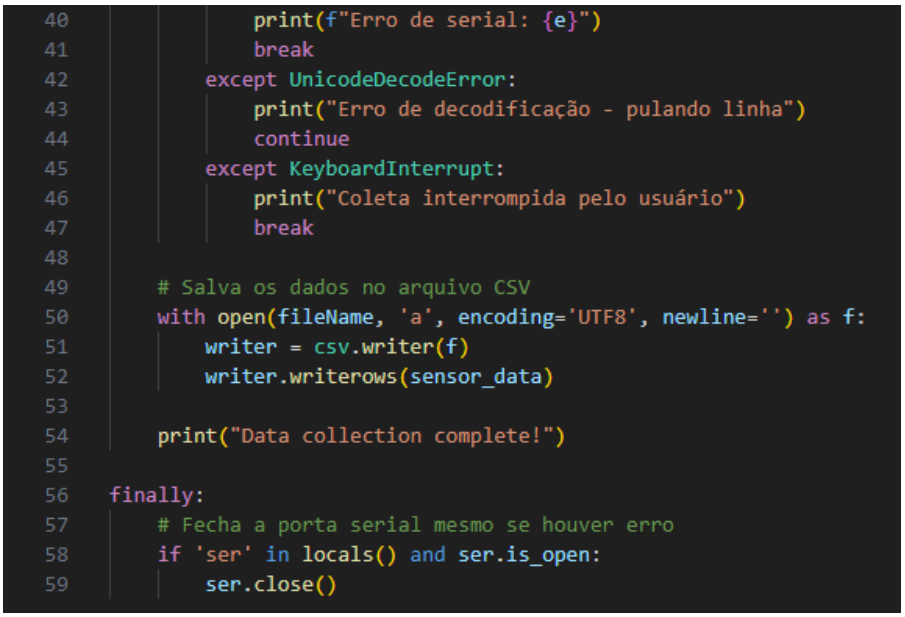
Após impressão e montagem da PCB, inicialmente, foram realizados três diferentes ensaios, a fim de levantar as funções de transferência presentes no sistema, sendo eles: Ensaio de Temperatura, Ensaio de Vazão e o Ensaio de Temperatura com Perturbação de vazão.

Para realizar as leituras dos ensaios, foi desenvolvido um código em Python responsável por capturar os dados exibidos na porta serial e armazená-los em um arquivo .CSV. Essa coleta permite, posteriormente, a análise dos dados para obtenção da função de transferência da planta. O código utilizado para a leitura dos dados está apresentado a seguir.

Figura 7: Printscreen código para a leitura dos dados.

```
leitura > Leitura_Serial.py > ...
import serial
import csv
from serial import SerialException
import time

arduino_port = "COM7"       # Porta serial do Arduino
baud = 230400
fileName = input("Digite o nome do arquivo: ") + ".csv"

# Função para resetar o Arduino via serial
def reset_arduino(ser):
    ser.setDTR(False)  # Desliga DTR
    time.sleep(1)      # Espera 1 segundo
    ser.setDTR(True)   # Liga DTR novamente (isso causa o reset)
    time.sleep(2)      # Espera o Arduino reiniciar completamente

try:
    # Abre a conexão serial
    ser = serial.Serial(arduino_port, baud)

    # Reseta o Arduino antes de começar
    reset_arduino(ser)

    sensor_data = []  # Armazena os dados

    # Limpa o buffer serial (remove dados antigos)
    ser.reset_input_buffer()

    while True:
        try:
            getData = ser.readline()
            dataString = getData.decode('utf-8').strip()  # Remove \r\n
            readings = dataString.split(",")

            if len(readings) > 1:  # Verifica se recebeu dados válidos
                print(readings)
                sensor_data.append(readings)

        except SerialException as e:
```

Figura 8: Printscreen código para a leitura dos dados.



```
                print(f"Erro de serial: {e}")
                break
            except UnicodeDecodeError:
                print("Erro de decodificação - pulando linha")
                continue
            except KeyboardInterrupt:
                print("Coleta interrompida pelo usuário")
                break

        # Salva os dados no arquivo CSV
        with open(fileName, 'a', encoding='UTF8', newline='') as f:
            writer = csv.writer(f)
            writer.writerows(sensor_data)

        print("Data collection complete!")

    finally:
        # Fecha a porta serial mesmo se houver erro
        if 'ser' in locals() and ser.is_open:
            ser.close()
```

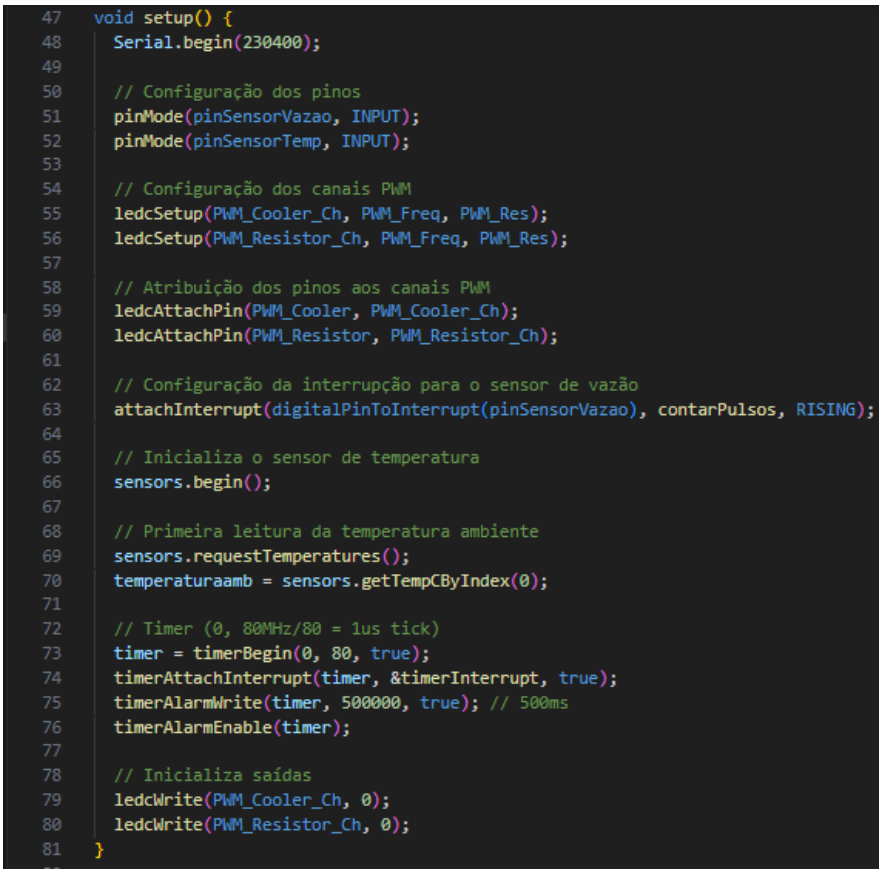
Para o primeiro ensaio, com a caixa em temperatura ambiente e o cooler desligado, foi enviado um sinal de 100% do PWM para a resistência presente na caixa, coletando amostras seguindo um período de amostragem de 1,658 segundos e armazenando os dados em um arquivo do tipo CSV utilizando o código "*Leitura_Serial*" mostrado nas Figuras 7 e 8. Durante todo o processo de aquecimento, foram coletadas amostras da temperatura interna da câmara a cada período de amostragem previamente definido. O experimento prosseguiu até que a temperatura alcançasse o regime permanente, caracterizado pela estabilização dos valores medidos, em nosso caso 81 ºC. Posteriormente, para o processamento dos dados, a temperatura ambiente inicial foi subtraída de todas as amostras coletadas, eliminando, assim, o efeito das condições iniciais e permitindo uma análise mais precisa da resposta do sistema.

O ensaio de vazão, responsável por descrever a relação entre a tensão aplicada ao cooler e a vazão de ar na saída, seguiu metodologia semelhante. O sistema foi iniciado com o cooler desligado, garantindo que a vazão inicial fosse nula. Em seguida, aplicou-se um sinal PWM de 100% ao cooler, e as amostragens da vazão (medição em pulsos) foram realizadas de forma contínua a cada período de amostragem de 0,5 segundos. Note que, devido à natureza mais rápida da dinâmica da vazão em comparação com a temperatura, foi necessário adotar um período de amostragem menor para garantir a captura adequada dos fenômenos transitórios. O ensaio foi mantido até que a vazão atingisse o regime permanente de 34 pulsos.

A seguir, é apresentada a lógica utilizada para enviar o valor máximo de PWM ao cooler, assim como a estruturação dos dados para exibição na porta serial. Para o ensaio de temperatura e temperatura com perturbação, foi aplicada a mesma lógica, com a diferença de

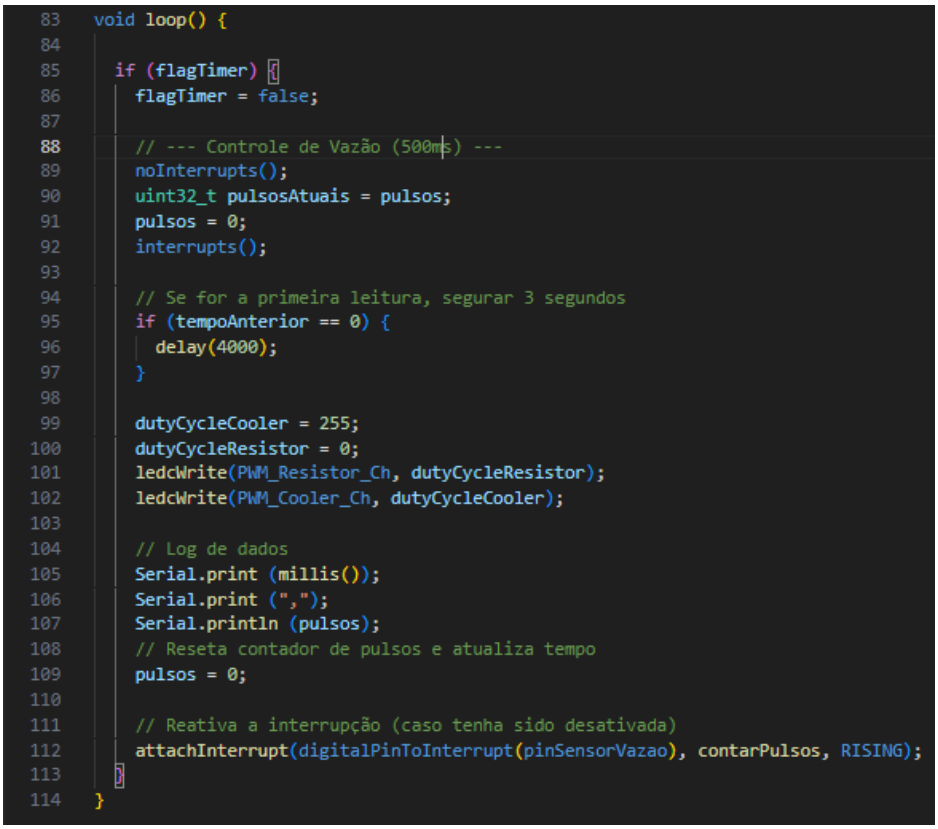
que, por se tratar de um sistema significativamente mais lento, o período de amostragem adotado é maior.

Figura 9: Printscreen código valor máximo de PWM ao cooler.



```
47  void setup() {
48    Serial.begin(230400);
49
50    // Configuração dos pinos
51    pinMode(pinSensorVazao, INPUT);
52    pinMode(pinSensorTemp, INPUT);
53
54    // Configuração dos canais PWM
55    ledcSetup(PWM_Cooler_Ch, PWM_Freq, PWM_Res);
56    ledcSetup(PWM_Resistor_Ch, PWM_Freq, PWM_Res);
57
58    // Atribuição dos pinos aos canais PWM
59    ledcAttachPin(PWM_Cooler, PWM_Cooler_Ch);
60    ledcAttachPin(PWM_Resistor, PWM_Resistor_Ch);
61
62    // Configuração da interrupção para o sensor de vazão
63    attachInterrupt(digitalPinToInterrupt(pinSensorVazao), contarPulsos, RISING);
64
65    // Inicializa o sensor de temperatura
66    sensors.begin();
67
68    // Primeira leitura da temperatura ambiente
69    sensors.requestTemperatures();
70    temperaturaamb = sensors.getTempCByIndex(0);
71
72    // Timer (0, 80MHz/80 = 1us tick)
73    timer = timerBegin(0, 80, true);
74    timerAttachInterrupt(timer, &timerInterrupt, true);
75    timerAlarmWrite(timer, 500000, true); // 500ms
76    timerAlarmEnable(timer);
77
78    // Inicializa saídas
79    ledcWrite(PWM_Cooler_Ch, 0);
80    ledcWrite(PWM_Resistor_Ch, 0);
81  }
```

Figura 10: Printscreen código valor máximo de PWM ao cooler.



```
83  void loop() {
84
85    if (flagTimer) {
86      flagTimer = false;
87
88      // --- Controle de Vazão (500ms) ---
89      noInterrupts();
90      uint32_t pulsosAtuais = pulsos;
91      pulsos = 0;
92      interrupts();
93
94      // Se for a primeira leitura, segurar 3 segundos
95      if (tempoAnterior == 0) {
96        delay(4000);
97      }
98
99      dutyCycleCooler = 255;
100     dutyCycleResistor = 0;
101     ledcWrite(PWM_Resistor_Ch, dutyCycleResistor);
102     ledcWrite(PWM_Cooler_Ch, dutyCycleCooler);
103
104     // Log de dados
105     Serial.print (millis());
106     Serial.print (",");
107     Serial.println (pulsos);
108     // Reseta contador de pulsos e atualiza tempo
109     pulsos = 0;
110
111     // Reativa a interrupção (caso tenha sido desativada)
112     attachInterrupt(digitalPinToInterrupt(pinSensorVazao), contarPulsos, RISING);
113   }
114 }
```

Além disso, foi realizado o ensaio de temperatura com perturbação de vazão, que descreve o efeito do acionamento do cooler sobre a temperatura interna da câmara, funcionando como uma perturbação térmica. Para esse ensaio, iniciou-se com a resistência de aquecimento alimentada com PWM máximo até que a temperatura interna se estabilizasse. Após atingir o regime permanente, aplicou-se um sinal PWM de 100% no cooler, e o comportamento da temperatura foi registrado em função do tempo, seguindo um período de amostragem de 1,660 segundos, atingindo, ao final, uma temperatura de 39 ºC. Assim como no primeiro ensaio, a temperatura inicial foi subtraída dos dados para focar exclusivamente na variação provocada pela perturbação.

É importante ressaltar que, todos esses ensaios, visando viabilizar o projeto de controle e que, além disso, analisando os resultados obtidos, pôde-se verificar a temperatura e vazão máxima suportada pela caixa, definindo os limites de atuação da planta, além de outros quesitos importantes para construção dos controladores que serão citados no tópico que segue.

4.2. Modelagem do sistema e MATLAB

A modelagem do sistema teve como objetivo identificar funções de transferência que descrevessem de forma precisa a dinâmica da planta de controle de temperatura e vazão de ar. Para isso, foram realizadas diversas etapas experimentais, os ensaios relatados acima, seguidos de processamento de dados no ambiente MATLAB dos dados armazenados nos arquivos do tipo CSV dos ensaios.

Como citado, o processamento dos dados experimentais foi realizado inteiramente no MATLAB, utilizando ferramentas de manipulação de sinais e ajuste de modelos dinâmicos, neste caso, o comando *tfest*. Este comando permite estimar uma função de transferência contínua a partir de dados experimentais, ajustando automaticamente o modelo para melhor representar o comportamento dinâmico observado. A ordem dos sistemas foi definida como de primeira ordem, de modo que as funções de transferência resultantes assumiram, em geral, a forma clássica de sistemas de primeira ordem, expressa pela equação:

$$G(s) = \frac{K}{\tau s + 1}$$

Em que K representa o ganho estático do sistema e τ a constante de tempo associada à sua dinâmica.

4.2.1. Função de Transferência de Temperatura

Dado o arquivo CSV com as amostras, ele foi separado em dois vetores, um de tempo, outro de temperatura, e com isso, foi extraída a função de transferência em questão, através do código abaixo.

Figura 11: Printscreen código da extração da função de transferência de temperatura.

```
%% Determinacao da Funcao de Transferencia G_T(s) relaciona a temperatura a tensao aplicada na resistencia

filenameT = 'dados_temperatura.csv';              % Arquivo com dados medidos
data = readmatrix(filenameT);                     % Lê os dados do arquivo CSV

tempo = data(:,1)/1000;                           % Tempo em segundos
temperatura = data(:,2);                          % Temperatura em °C

length_ut = length(tempo);
kt = 3;                                           % Número de elementos iniciais iguais a zero

ut = [zeros(1, kt), ones(1, length_ut - kt)]';    % Vetor de entrada (pwm 0/1) transposta (em colunas)

dados = iddata(temperatura,ut,1.658);             % Vetores coluna
ft_ident = tfest(dados,1,0);

numt = ft_ident.Numerator;
dent = ft_ident.Denominator;

G_modelo_t = tf(numt,dent);                       % Função de Transferência (TEMPERATURA)
```

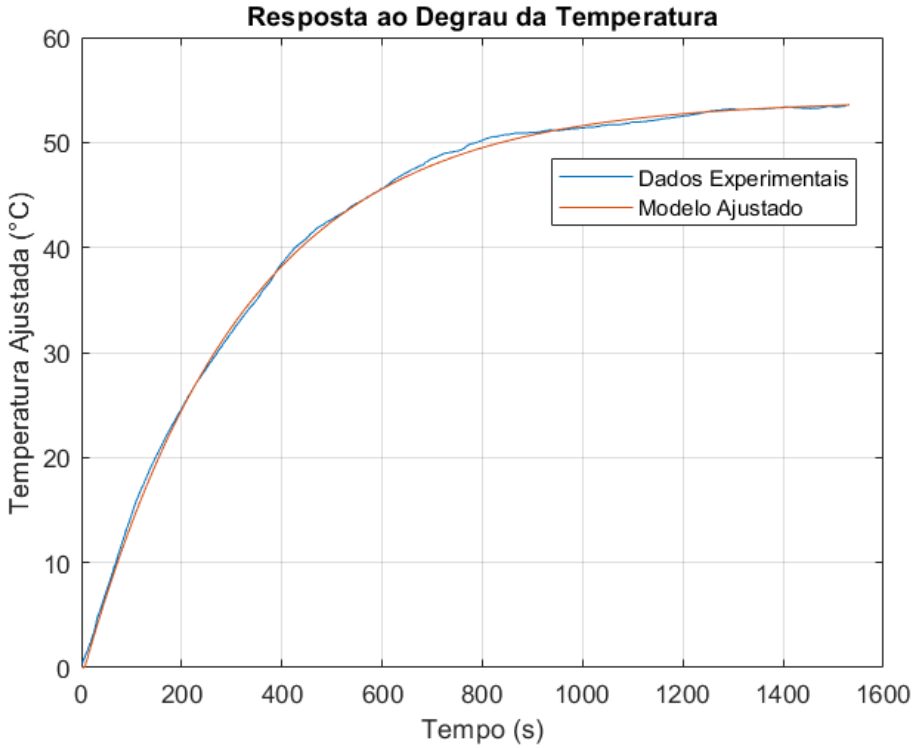
```
%% Gráficos Temperatura
y_mt = lsim(G_modelo_t,ut,tempo);
figure(1);
plot(tempo,temperatura);
hold on
plot(tempo,y_mt)
legend('Real','Modelo')
xlabel('Tempo (s)');
ylabel('Temperatura Ajustada (°C)');
title('Resposta ao Degrau da Temperatura');
legend('Dados Experimentais', 'Modelo Ajustado');
grid on;
```

Resultando na função:

$$G_{modelo_t}(s) = \frac{0{,}0006628}{s + 0{,}003129}$$

Para que a função obtida fosse validada, foram plotados os dados experimentais, juntamente com a função, chegando ao gráfico que segue.

Figura 12: Gráfico da resposta ao degrau da temperatura.



4.2.2. Função de Transferência de Vazão

O procedimento para obtenção da planta de vazão foi semelhante ao anterior realizado para a temperatura, seguindo o código abaixo.

Figura 13: Printscreen código da extração da função de transferência da vazão.

```
%% Determinacao da Funcao de Transferencia G_F(s) relaciona a vazao e a tensao aplicada no cooler

filenameF = 'vazao5.csv';                          % Arquivo com dados medidos
data = readmatrix(filenameF);                      % Lê os dados do arquivo CSV

tempo_f = data(:,1)/1000;                          % Tempo em segundos
vazao = data(:,2);                                 % Vazao em pulsos

length_uf = length(tempo_f);
kf = 1;                                            % Número de elementos iniciais iguais a zero

uf = [zeros(1, kf), ones(1, length_uf -kf)]';

dados = iddata(vazao,uf,0.500);                    % Vetores coluna
ft_ident = tfest(dados,1,0);

numf = ft_ident.Numerator;
denf = ft_ident.Denominator;

G_modelo_f = tf(numf,denf);                        % Função de Transferência (VAZÃO)
```

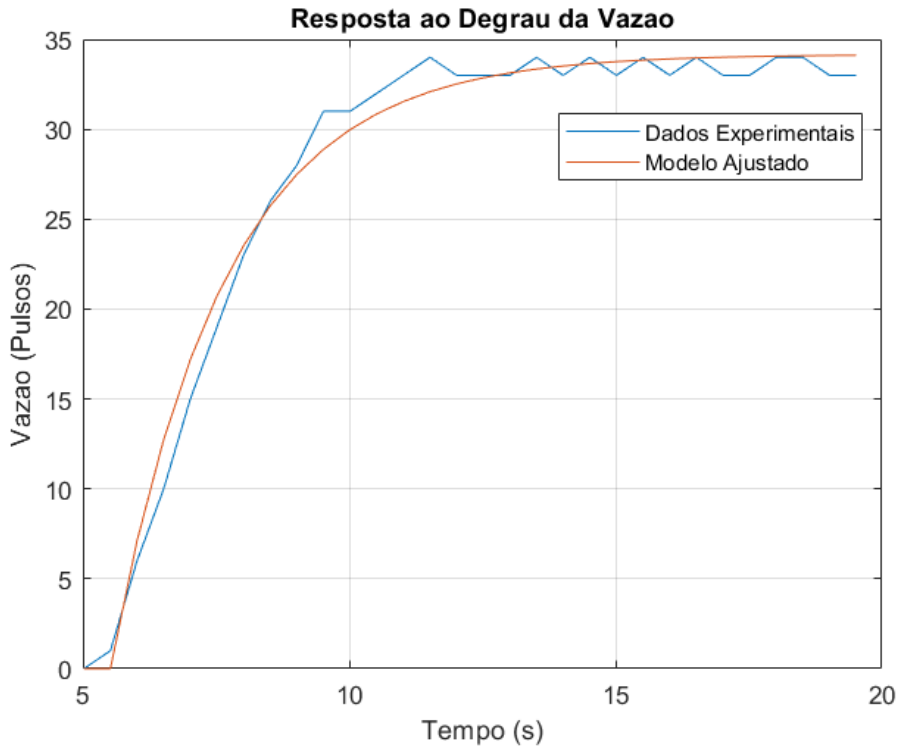
```
%% Gráficos Vazão

y_mf = lsim(G_modelo_f,uf,tempo_f);
figure(2);
plot(tempo_f,vazao);
hold on
plot(tempo_f,y_mf)
legend('Real','Modelo')
xlabel('Tempo (s)');
ylabel('Vazao (Pulsos)');
title('Resposta ao Degrau da Vazao');
legend('Dados Experimentais', 'Modelo Ajustado');
grid on;
```

Obtendo a função de transferência:

$$G_{modelo_f}(s) = \frac{0{,}06253}{s + 0{,}4667}$$

Que, comparada com os dados experimentais, têm-se o gráfico a seguir.

Figura 14: Gráfico da resposta ao degrau da vazão.



4.2.3. Função de Transferência de Temperatura com perturbação

Dessa vez, para a função de transferência que relaciona a perturbação na temperatura de saída com a tensão aplicada ao cooler, o procedimento, ainda que semelhante, foi um pouco diferente, uma vez que a planta em questão é negativa, assim, o código mostrado pela Figura 11 foi ajustado para este caso.

Figura 15: Printscreen código da extração da função de transferência de temperatura com perturbação.

```
%% Determinacao da Funcao de Transferencia G_TF(s) relaciona a Temperatura a Perturbacao gerada pelo cooler

filenameTF = 'dados_perturbacao.csv';            % Arquivo com dados medidos
data = readmatrix(filenameTF);                   % Lê os dados do arquivo CSV

t = data(:,1);                                   % Tempo em milisegundos
tempo_p = (t(274:end) - 456500)/1000;            % Vetor correto de tempo em segundos
temp = data(:,2);                                % Temperatura em °C
temperatura_p = temp(274:end) - 80;              % Vetor correto de temperatura desconsiderando o valor inicial

length_u = length(tempo_p);
k = 314-274;                                     % Número de elementos iniciais iguais a zero

u = [zeros(1, k), ones(1, length_u  -k)];
u = u';                                          % Matriz transposta para formacao do vetor correto
dados = iddata(temperatura_p,u,1.660);           % Vetores coluna
ft_ident = tfest(dados,1,0);

num = ft_ident.Numerator;
den = ft_ident.Denominator;

G_modelo = tf(num,den);                          % Funcao de Transferência (TEMPERATURA com PERTURBAÇÃO)
```

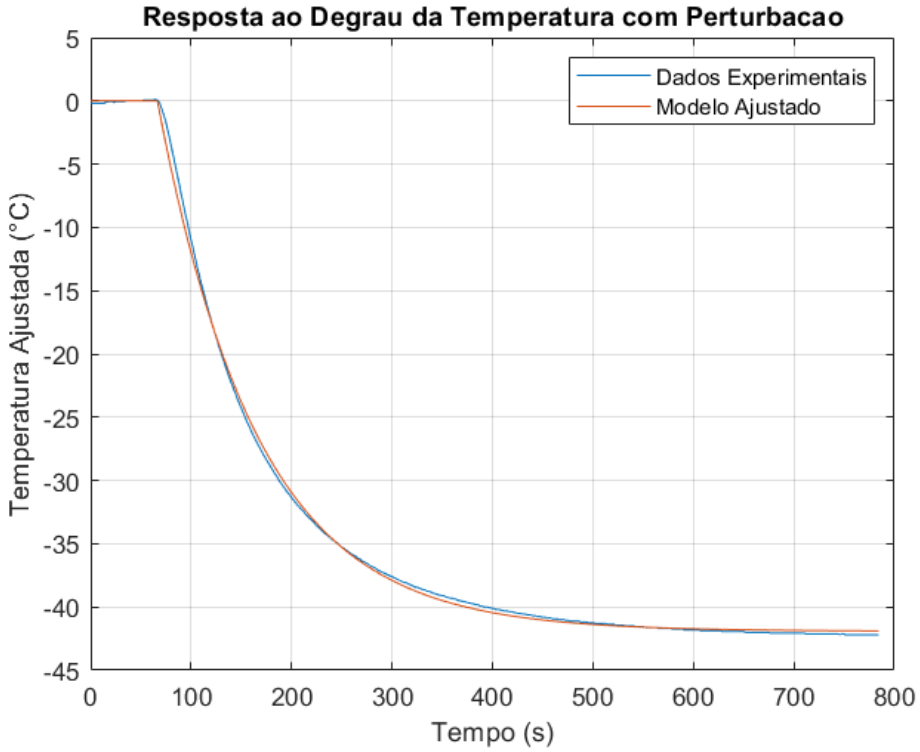
```
%% Gráficos da Temperatura com Perturbação

y_m = lsim(G_modelo,u,tempo_p);
figure(3);
plot(tempo_p,temperatura_p);
hold on
plot(tempo_p,y_m);
legend('Real','Modelo')
xlabel('Tempo (s)');
ylabel('Temperatura Ajustada (°C)');
title('Resposta ao Degrau da Temperatura com Perturbacao');
legend('Dados Experimentais', 'Modelo Ajustado');
grid on;
```

Gerando a função de transferência:

$$G_{modelo}(s) = \frac{-0{,}001646}{s + 0{,}01003}$$

Que, quando comparado com os dados experimentais, gerou o gráfico abaixo.

Figura 16: Gráfico da resposta ao degrau da temperatura com perturbação.



4.3. Discretização das Plantas

Após a modelagem no domínio contínuo, foi necessária a discretização das plantas para implementação prática no microcontrolador ESP32, que opera de forma digital. Utilizou-se o comando `c2d` do MATLAB para realizar a conversão dos modelos contínuos para modelos discretos, adotando o método de retenção de ordem zero (Zero-Order Hold - `'zoh'`). Este método é apropriado para sistemas em que a entrada permanece constante entre os instantes de amostragem, o que reflete bem o comportamento de sinais PWM utilizados no projeto. A Figura 17 mostra o código com os comandos utilizados para a discretização das plantas.

Figura 17: Printscreen código para a discretização das plantas.

```
%% Discretização das Plantas

% Planta Temperatura
Ta_temp = 1.658;
G_t_disc = c2d(G_modelo_t, Ta_temp, 'zoh');

% Planta Vazão
Ta_vazao = 0.5;
G_f_disc = c2d(G_modelo_f, Ta_vazao, 'zoh');

% Planta Temperatura com perturbacao de Vazao
Ta_perturbacao = 1.660;
G_tf_disc = c2d(G_modelo, Ta_perturbacao, 'zoh');
```

As funções discretizadas obtidas para as plantas de temperatura, vazão e perturbação foram, respectivamente:

$$G_{t_{disc}}(z) = \frac{0{,}001096}{z - 0{,}9948}$$

$$G_{f_{disc}}(z) = \frac{0.02789}{z - 0{,}7919}$$

$$G_{tf_{disc}}(z) = \frac{-0{,}002715}{z - 0{,}9835}$$

Novamente, para validar o processo realizado, foram gerados os gráficos comparando a função em tempo contínuo com a função discretizada, dispostas nas Figuras 18, 19 e 20, a seguir.

Figura 18: Gráfico comparação tempo contínuo e discreto Temperatura

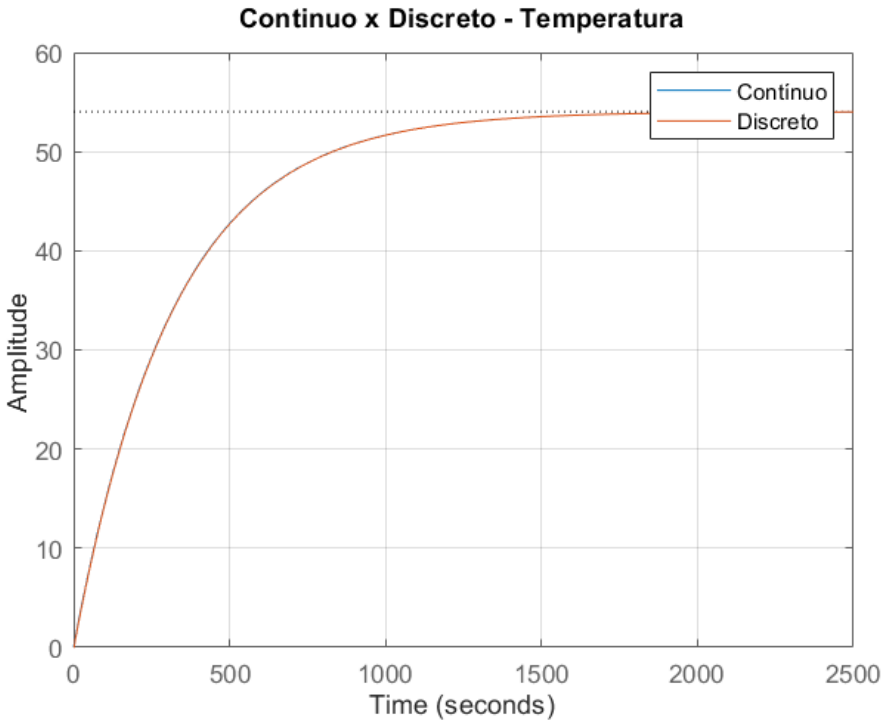


Figura 19: Gráfico comparação tempo contínuo e discreto Vazão

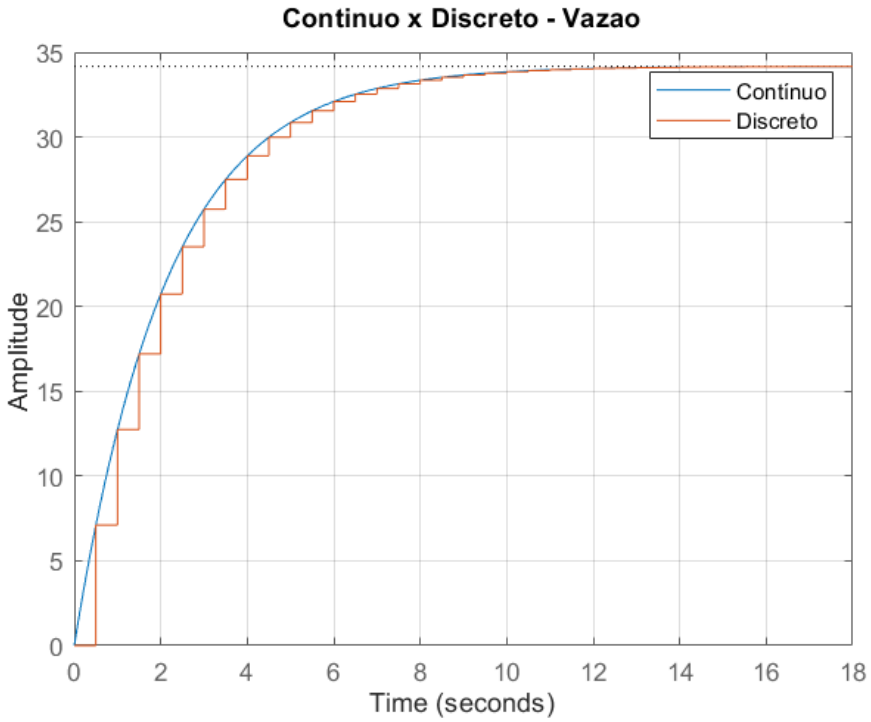
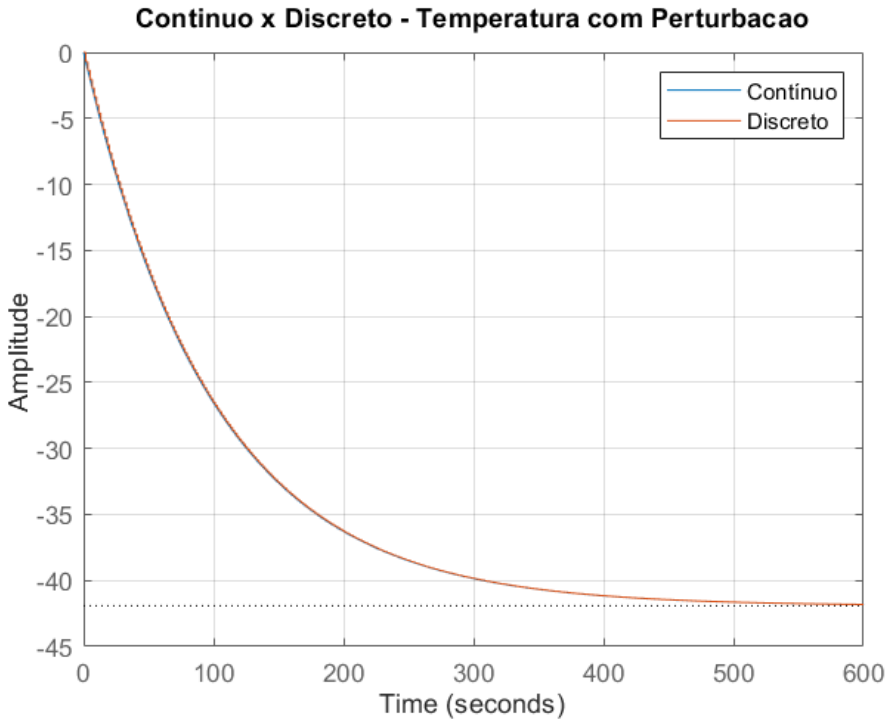


Figura 20: Gráfico comparação tempo contínuo e discreto Temperatura com Perturbação



5. Projeto dos Controladores

Com as plantas discretizadas previamente identificadas, foi possível avançar para o projeto dos controladores responsáveis pela regulação da temperatura e da vazão do sistema. O projeto foi realizado no domínio discreto utilizando a ferramenta Sisotool do MATLAB, que permite a análise e o ajuste de controladores de forma interativa e gráfica. O objetivo principal do projeto foi atender aos seguintes requisitos de desempenho: garantir erro em regime permanente nulo e manter o tempo de acomodação da malha fechada próximo ao tempo de acomodação da planta em malha aberta. Esses requisitos foram definidos para assegurar um comportamento estável, preciso e seguro do sistema em operação.

Dados os requisitos de desempenho selecionados, optou-se pela utilização de controladores do tipo Proporcional-Integral (PI) para ambas as variáveis, considerando também as características observadas nas plantas. A escolha pela utilização de controladores PI, e não PID, foi motivada pelas próprias características das plantas modeladas. Observou-se que tanto a planta de temperatura quanto a planta de vazão apresentaram comportamentos bem representados por sistemas de primeira ordem, com respostas lentas e sem grandes oscilações naturais. Nessas condições, a presença de uma ação derivativa não se mostrou necessária, pois

o objetivo principal do controle era garantir uma correção adequada do erro ao longo do tempo e não acelerar a resposta ou amortecer oscilações excessivas. Dessa forma, o uso do controlador PI mostrou-se suficiente para atender aos requisitos estabelecidos, proporcionando simplicidade na implementação e robustez frente a pequenas incertezas e ruídos do sistema.

Portanto, dados os requisitos, o erro nulo em regime permanente pode ser facilmente obtido ao adicionarmos um integrador (polo em $z = 1$), já previsto pelo controlador PI, e, para o requisito do tempo de acomodação, basta ajustarmos o controlador para garantir um tempo de acomodação equivalente a quatro vezes a constante de tempo obtida na planta. Com isso, ambos os controladores foram projetados, e têm-se abaixo as Figuras 21 e 22, que apresentam o resultado obtido na ferramenta do Sisotool.

Figura 21: Projeto de controle via Sisotool para a Temperatura

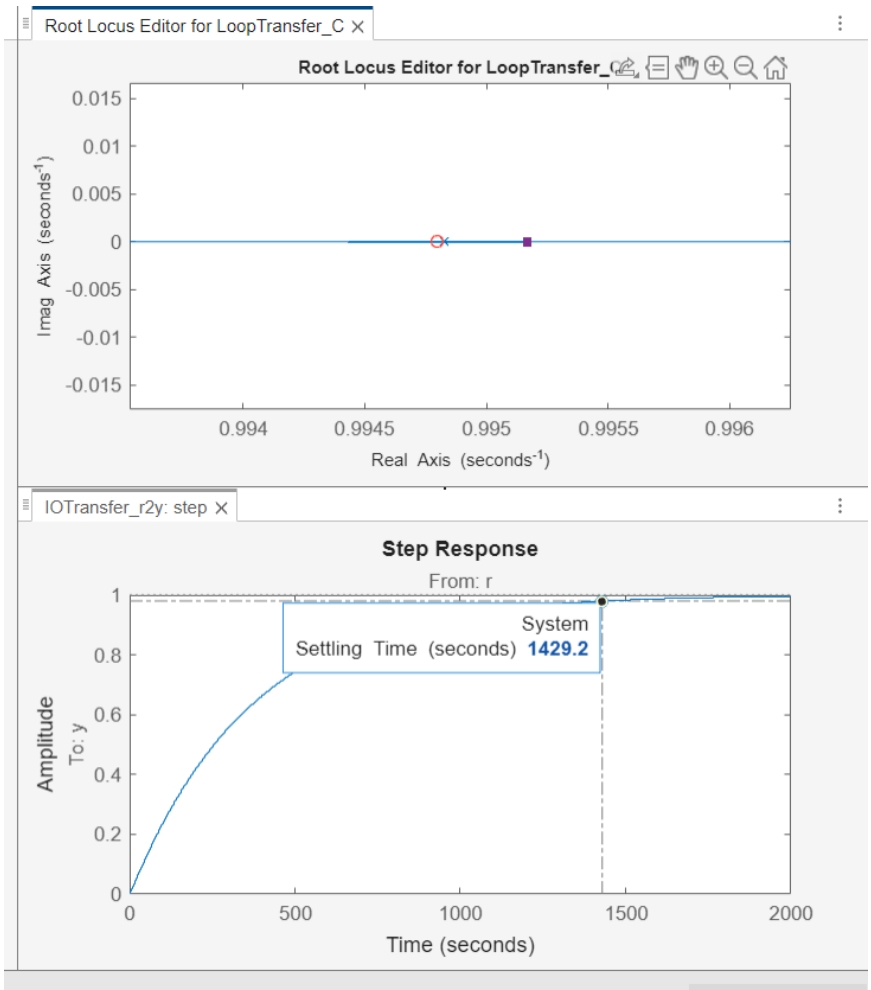
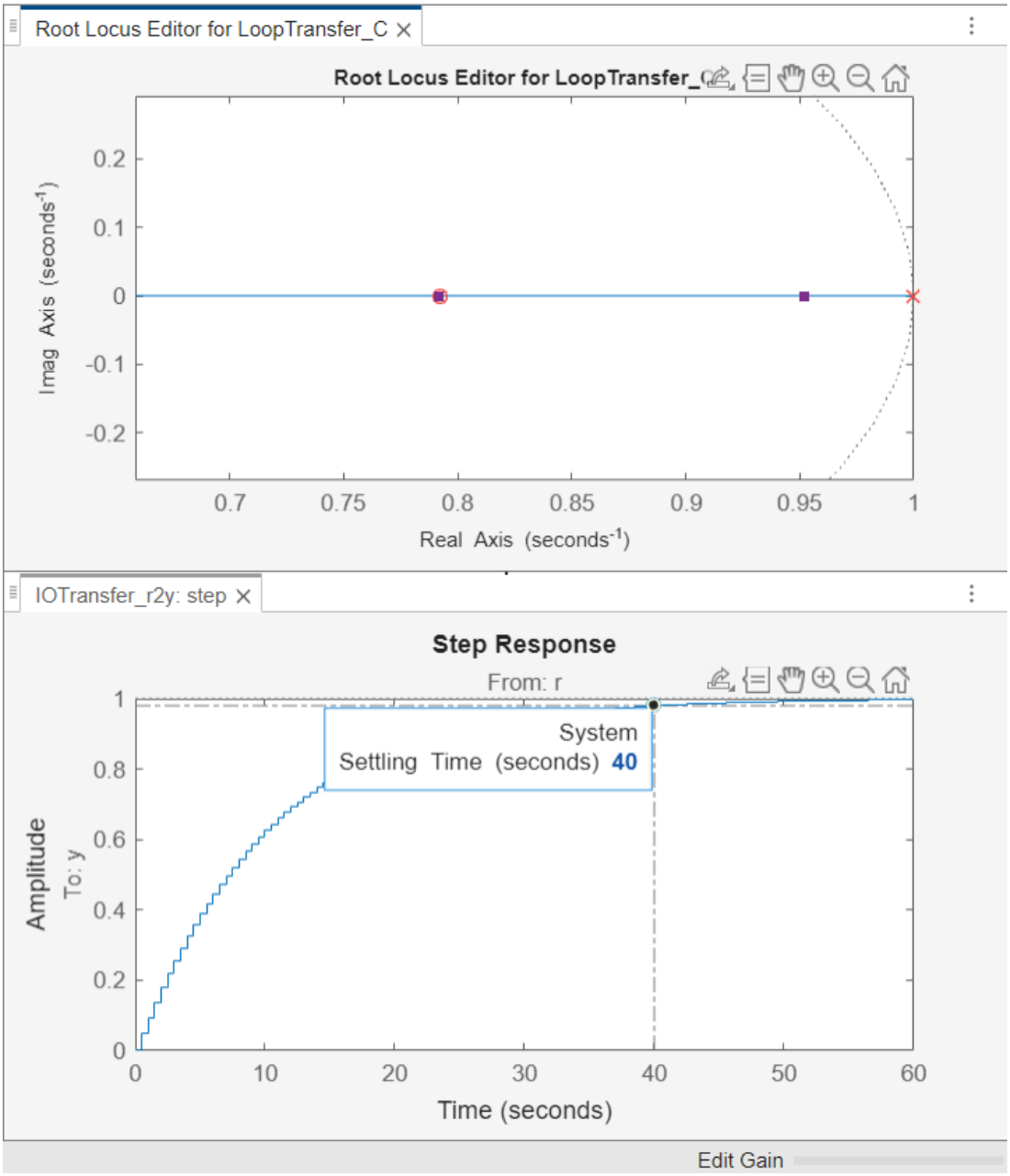


Figura 22: Projeto de controle via Sisotool para a Vazão



Os controladores obtidos são os mostrados a seguir.

$$C_{t_{disc}}(z) = \frac{4{,}097(z - 0{,}995)}{z - 1}$$

$$C_{f_{disc}}(z) = \frac{1{,}72(z - 0{,}7920)}{z - 1}$$

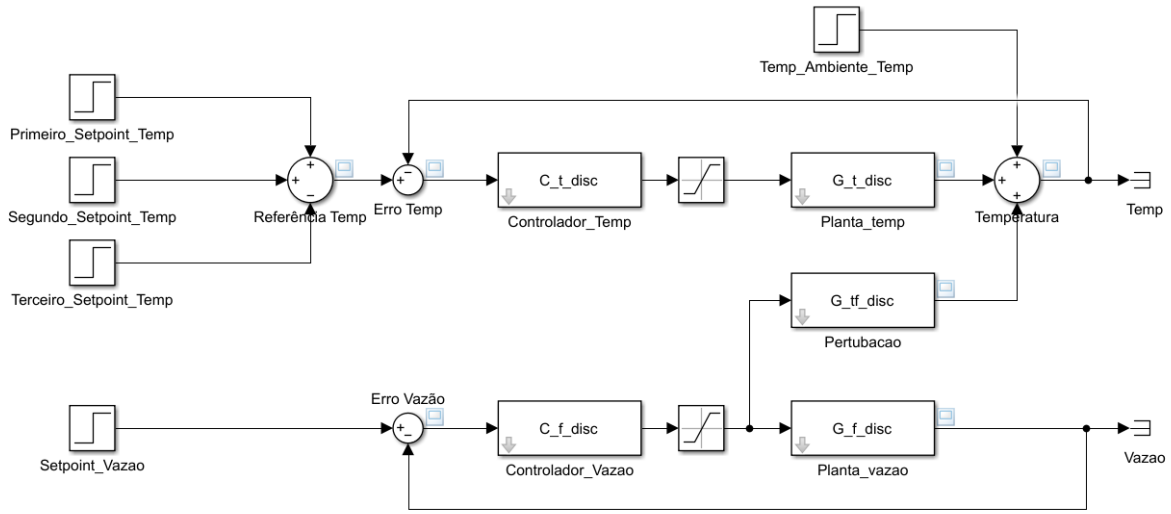
Por fim, ao término do ajuste no Sisotool, as funções de transferência discretas dos controladores PI foram definidos no código do MATLAB e exportadas em forma de equações de diferenças, prontas para implementação prática na plataforma ESP32. Dessa maneira, assegurou-se que os controladores projetados atenderem aos requisitos de desempenho, mantendo o sistema operando de forma estável, precisa e eficiente.

```
C_t_disc =

  4.097 (z-0.995)
  ---------------
       (z-1)

C_f_disc =

  4.805 (z-0.792)
  ---------------
       (z-1)
```

6. Simulação

Com os controladores definidos, foi montado o Simulink apresentado pela figura abaixo, nele, foram definidos setpoints de temperatura intermediários, a fim de mostrar o funcionamento do sistema, respeitando os requisitos estabelecidos.

Figura 23: Simulink para o funcionamento do sistema.

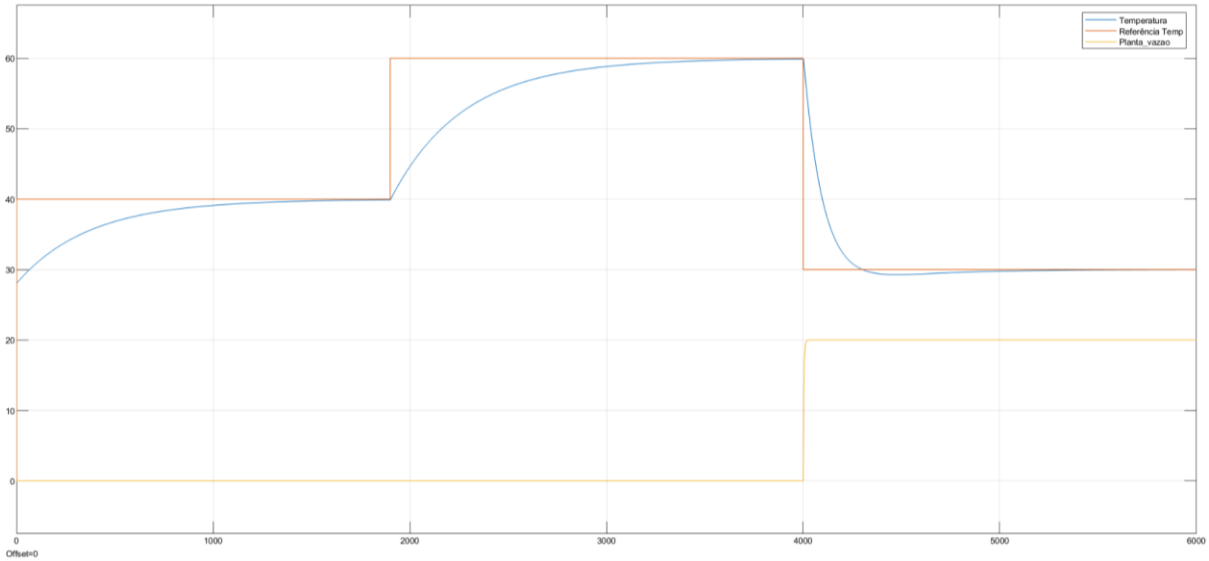


Portanto, para a simulação, os setpoints definidos foram:

- Primeiro Setpoint: 40 ºC.
- Segundo Setpoint: 60 ºC.
- Terceiro Setpoint: 30 ºC.

Desse modo, verificou-se a seguinte resposta do sistema aos setpoints definidos.

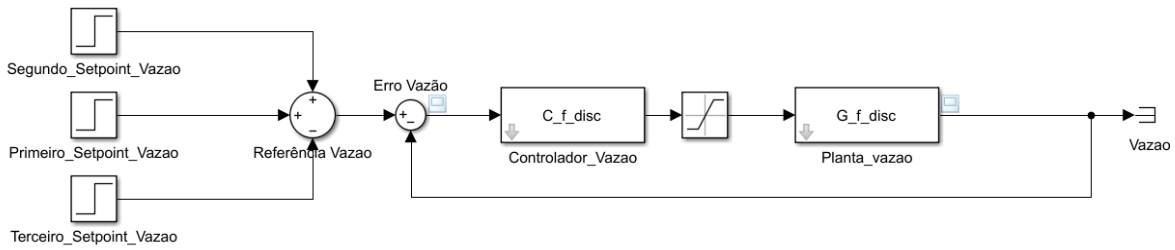
Figura 24: Resposta do Sistema.



Note que, ao ligar a vazão, representado pela linha amarela (setpoint de vazão) e laranja (sistema em si), a resposta da temperatura é afetada, de modo que como a planta de perturbação tem efeito negativo e o novo setpoint de temperatura é menor que o anterior, faz com que a resposta obtida seja mais rápida.

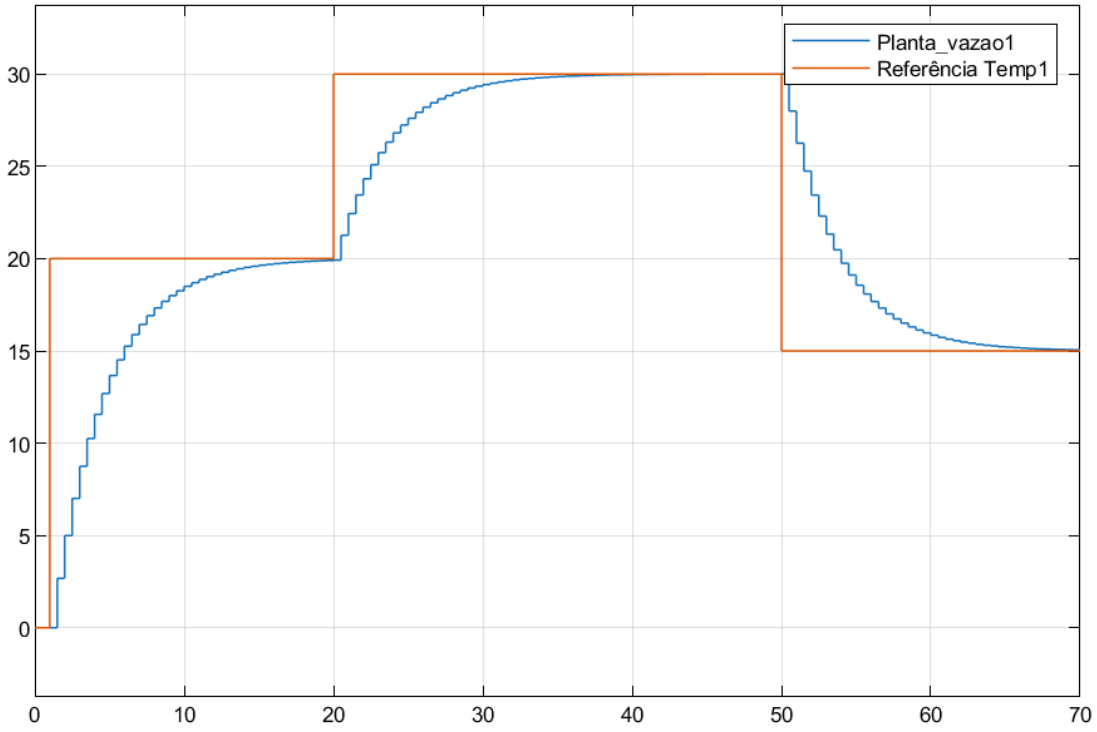
Assim como realizado para a temperatura, o mesmo teste foi realizado para a vazão, definindo diferentes setpoints para validar o sistema montado. Conforme mostrado pelo Simulink abaixo.

Figura 25: Simulink para o funcionamento do sistema de Vazão.



Com setpoints de 20, 30 e 15 pulsos, respectivamente, gerando o gráfico abaixo.

Figura 26: Resposta pulsos.



7. Implementação Digital

Após a validação dos controladores de temperatura e vazão por meio de simulações no Simulink, estes foram implementados em código para aplicação no sistema real. A implementação foi realizada com base nas equações de diferenças dos compensadores projetados que proporcionam os efeitos de *anti-windup* e *bumpless transfer*, contribuindo para uma operação mais estável e segura do sistema de controle.

Dessa forma, como o controlador foi originalmente projetado no domínio contínuo, sua implementação em um microcontrolador exigiu a conversão para o domínio discreto. Para isso, foi utilizado o método de Tustin no MATLAB, considerando um tempo de amostragem (TS) igual a 1 segundo.

A lógica foi implementada com uso de interrupções por hardware para contar os pulsos de vazão e para gerar uma flag periódica de controle baseada em um temporizador (`hw_timer_t`). A cada 500 ms, a flag ativa o ciclo de controle de vazão, e a cada 1,5 segundos (a cada três ciclos), a leitura de temperatura é atualizada e o controle térmico é executado.

Além disso, o sistema realiza mudanças automáticas de setpoint com base na estabilização da temperatura em torno da referência por pelo menos 1 minuto, caracterizando um controle baseado em estados. O código também implementa proteções contra saturação, limitando o sinal de controle entre 0 e 255, faixa correspondente ao duty cycle do PWM de 8 bits.

A estrutura geral do firmware foi organizada para oferecer boa responsividade, leitura precisa dos sensores, controle eficiente e segurança operacional durante a execução contínua do sistema

Figura 27: Arquivo .ini

platformio.ini

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:esp32dev]
platform = espressif32
board = esp32dev
framework = arduino
monitor_speed = 230400
upload_port = COM7
lib_deps =
    paulstoffregen/OneWire @ ^2.3.7
    milesburton/DallasTemperature @ ^3.9.0
```

Figura 28: Parte 1 do código final do sistema de controle de temperatura e vazão.

src > main.cpp > controlador_temperatura(float, float)

```
#include <Arduino.h>
#include <math.h>
#include <OneWire.h>
#include <DallasTemperature.h>

// Definindo pinos utilizados
#define pinSensorVazao 2   // Pino de leitura para Vazão
#define pinSensorTemp 14   // Pino de leitura da temperatura
#define PWM_Cooler 15      // Pino de saída do sinal PWM do Cooler
#define PWM_Resistor 19    // Pino de saída do sinal PWM da Resistência

// PWM setup
#define PWM_Cooler_Ch 0
#define PWM_Resistor_Ch 1
#define PWM_Freq 5000
#define PWM_Res 8

volatile uint32_t pulsos = 0;
volatile bool flagTimer = false;
hw_timer_t *timer = NULL;

// Variáveis para o sensor de temperatura DS18B20
OneWire oneWire(pinSensorTemp);
DallasTemperature sensors(&oneWire);
float temperatura = 0.0;

// Variáveis Controlador Temperatura
float d_t = 0.0;
int duty_cycle_t = 0;
float u_t = 0.0, ek_t = 0.0, uk_1t = 0.0, ek_1t = 0.0;

```

Figura 29: Parte 2 do código final do sistema de controle de temperatura e vazão.

```
// Variáveis Controlador Vazão
float d_f = 0.0;
int duty_cycle_f = 0;
float u_f = 0.0, ek_f = 0.0, uk_1f = 0.0, ek_1f = 0.0;

// Vetores de setpoints ao longo do tempo
float setpoints_T[] = {40.0, 45.0, 35.0};  // Temperatura
float setpoints_V[] = {0.0, 0.0, 35.0};  // Vazão
int i = 0;
float referencia_T = setpoints_T[i];
float referencia_V = setpoints_V[i];

// Constantes do Controlador PI
float b0_t = 4.097;
float b1_t = -0.995*b0_t;
float b0_f = 1.72;
float b1_f = -0.792*b0_f;

// Tolerância e temporização
const float tolerancia_T = 0.5;
const unsigned long tempoEsperaT = 60000;  // 1 minutos em ms
// Controle de estados de espera
static bool esperandoTemp = false;
static unsigned long tempoInicioTemp = 0;

static bool esperandoVazao = false;
static unsigned long tempoInicioVazao = 0;

void IRAM_ATTR contarPulsos() {
  pulsos++;
}
```

Figura 30: Parte 3 do código final do sistema de controle de temperatura e vazão.

```
void IRAM_ATTR timerInterrupt() {
  flagTimer = true;
}

// Controlador PI para Temperatura
float controlador_temperatura(float r, float y) {
  ek_t = r - y;
  u_t = uk_1t + b0_t * ek_t + b1_t * ek_1t;

  if (u_t > 255) u_t = 255;
  else if (u_t < 0) u_t = 0;

  ek_1t = ek_t;
  uk_1t = u_t;

  return u_t;
}

// Controlador PI para Vazão
float controlador_vazao(float r, float y) {
  ek_f = r - y;
  u_f = uk_1f + b0_f * ek_f + b1_f * ek_1f;

  if (u_f > 255) u_f = 255;
  else if (u_f < 0) u_f = 0;

  ek_1f = ek_f;
  uk_1f = u_f;

  return u_f;
}
```

Figura 31: Parte 4 do código final do sistema de controle de temperatura e vazão.

```
void setup() {
  Serial.begin(230400);

  pinMode(pinSensorVazao, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(pinSensorVazao), contarPulsos, FALLING);

  pinMode(pinSensorTemp, INPUT);

  ledcSetup(PWM_Cooler_Ch, PWM_Freq, PWM_Res);
  ledcSetup(PWM_Resistor_Ch, PWM_Freq, PWM_Res);
  ledcAttachPin(PWM_Cooler, PWM_Cooler_Ch);
  ledcAttachPin(PWM_Resistor, PWM_Resistor_Ch);

  timer = timerBegin(0, 80, true); // 80 MHz / 80 = 1 us
  timerAttachInterrupt(timer, &timerInterrupt, true);
  timerAlarmWrite(timer, 500000, true); // 500 ms
  timerAlarmEnable(timer);

  ledcWrite(PWM_Cooler_Ch, 0);
  ledcWrite(PWM_Resistor_Ch, 0);

  sensors.begin();
}

void loop() {
  static uint8_t contadorTemperatura = 0;

  if (flagTimer) {
    flagTimer = false;

    // --- Controle de Vazão ---
    noInterrupts();
```

Figura 32: Parte 5 do código final do sistema de controle de temperatura e vazão.

```
    noInterrupts();
    uint32_t pulsosAtuais = pulsos;
    pulsos = 0;
    interrupts();

    float leituraVazao = pulsosAtuais; // Pode aplicar fator de conversão se necessário

    d_f = controlador_vazao(referencia_V, leituraVazao);
    duty_cycle_f = int(d_f);
    ledcWrite(PWM_Cooler_Ch, duty_cycle_f);

    Serial.printf("Vazao | Tempo: %lu ms, Pulsos: %lu, PWM Cooler: %d\n", millis(), pulsosAtuais, duty_cycle_f);

    // --- Controle de Temperatura ---
    contadorTemperatura++;
    if (contadorTemperatura >= 3) {
      contadorTemperatura = 0;

      sensors.requestTemperatures();
      temperatura = sensors.getTempCByIndex(0);

      d_t = controlador_temperatura(referencia_T, temperatura);
      duty_cycle_t = int(d_t);
      ledcWrite(PWM_Resistor_Ch, duty_cycle_t);

      Serial.printf("Temperatura | Tempo: %lu ms, Temp: %.2f °C, PWM Resistor: %d\n", millis(), temperatura, duty_cycle_t);

      if (!esperandoTemp && fabs(temperatura - referencia_T) <= tolerancia_T) {
        esperandoTemp = true;
        tempoInicioTemp = millis();
```

Figura 33: Parte 6 do código final do sistema de controle de temperatura e vazão.

```
        } else if (esperandoTemp && (millis() - tempoInicioTemp >= tempoEsperaT)) {
          esperandoTemp = false;
          if (i < 2) {
            i++;
            referencia_T = setpoints_T[i];
            referencia_V = setpoints_V[i];
          }
        }
      }
    }
  }
```

O código desenvolvido tem como objetivo o controle da vazão e da temperatura em um sistema embarcado utilizando a plataforma ESP32. Inicialmente, são incluídas bibliotecas fundamentais para o funcionamento do sistema: a biblioteca <*math.h*> para operações matemáticas e as bibliotecas <*OneWire.h*> e <*DallasTemperature.h*> para comunicação e leitura do sensor de temperatura DS18B20, que opera via protocolo *OneWire*.

Na sequência, são definidos os pinos utilizados pela ESP32, tanto para leitura dos sensores quanto para saída dos sinais PWM que controlam os atuadores (cooler e resistência). Também são declaradas as variáveis globais utilizadas ao longo de todo o código, incluindo variáveis de controle (como erros atuais e anteriores), setpoints e duty cycles, além das constantes dos controladores PI. Vale destacar os coeficientes dos controladores (`b0` e `b1`), obtidos previamente a partir de um projeto de controle realizado no MATLAB.

Os controladores de temperatura e vazão, implementados com lógica PI discreta, são chamados continuamente dentro da função *loop()*. A função *setup()*, por sua vez, é responsável por toda a configuração inicial do sistema, incluindo a inicialização da comunicação serial, definição dos pinos de entrada e saída, configuração dos canais PWM e do timer de hardware para interrupções periódicas, além da inicialização do sensor de temperatura.

A lógica de controle é executada na função *loop()*, que roda continuamente. Um temporizador por interrupção é utilizado para garantir que o controle de vazão seja executado a cada 500 ms. Já o controle de temperatura é executado a cada 1500 ms, por meio de um contador interno que acumula as interrupções do timer. Em cada ciclo de controle, são exibidas informações relevantes na serial, como o tempo decorrido, os valores medidos de temperatura e vazão, e os respectivos sinais PWM aplicados.

Adicionalmente, o código implementa uma lógica de temporização para garantir que, ao atingir o valor de referência com uma tolerância definida, o sistema mantenha esse valor por um minuto antes de alterar para o próximo setpoint, conforme definido nos vetores de referência. Isso garante estabilidade e evita oscilações indesejadas antes de avançar para a próxima etapa de controle.

8. Resultados

Após a conclusão das etapas descritas nos tópicos anteriores, que abrangeram desde a confecção e montagem da placa até a obtenção das plantas e análise dos resultados via simulação, a fase final do desenvolvimento do projeto consistiu na avaliação do sistema em condições reais de operação. Os controladores foram implementados no sistema físico, e então foram analisadas as respostas obtidas a partir da realização de diferentes casos de teste, com o objetivo de verificar o desempenho do controle implementado.

Através da implementação digital implementada, novamente utilizando o código *"Leitura_Serial"* foram extraídos os dados obtidos e exportados para o arquivo *"resultados.csv"* que foi tratado via Matlab para obtenção dos resultados gráficos que serão desenvolvidos. Assim, primeiramente, segue o código relativo à leitura dos dados do arquivo csv.

Figura 34: Código utilizado para leitura da temperatura e vazão do mesmo arquivo *"resultados.csv"*.

```
%% Leitura do arqivo

% Abrir o arquivo original
filename = 'resultados.csv';
fid = fopen(filename, 'r');

% Inicializar vetores que receberao os valores de temperatura e vazao
linhas_temperatura = {};
linhas_vazao = {};
i = 1;
while ~feof(fid)
    linha = fgetl(fid);
    if mod(i-1, 4) == 0  % Se a linha for múltipla de 4
        linhas_temperatura{end+1} = linha;

    else
        linhas_vazao{end+1} = linha;                    % Pega todas as linhas exceto 1, 5, 9...
    end
    i = i + 1;
end
fclose(fid);


```

Com as linhas de temperatura e vazão identificadas, a seguir tem-se o código do tratamento dos dados e o gráfico dos resultados obtidos para a temperatura e vazão, respectivamente.

Figura 35: Código para tratamento dos dados de temperatura e plotagem de temperatura.

```
%% Dados de Temperatura

% Salvar as linhas filtradas em um novo arquivo
fid_out = fopen('linhas_multiplos_de_4.csv', 'w');
for k = 1:length(linhas_temperatura)
    fprintf(fid_out, '%s\n', linhas_temperatura{k});
end
fclose(fid_out);

% Inicializar células para armazenar colunas
coluna1 = {}; coluna2 = {}; coluna3 = {};

% Iterar sobre cada linha e separar por vírgula
for i = 1:length(linhas_temperatura)
    campos = strsplit(linhas_temperatura{i}, ',');

    % Armazenar em vetores coluna
    if length(campos) >= 3                                  % garantir que há ao menos 3 campos
        coluna1{end+1, 1} = strtrim(campos{1});             % Dados do Tempo
        coluna2{end+1, 1} = strtrim(campos{2});             % Dados do Temperatura
        coluna3{end+1, 1} = strtrim(campos{3});             % Dados do PWM
    end
end

% Inicializar vetores numéricos
tempo_valores = zeros(length(coluna1), 1);                  % tempo em ms
temp_valores = zeros(length(coluna2), 1);                   % temperatura em °C
pwm_valores = zeros(length(coluna3), 1);                    % PWM de 0 a 255

% Extrair números do campo Tempo
for i = 1:length(coluna1)
    match = regexp(coluna1{i}, '\d+', 'match');             % pega números inteiros
    if ~isempty(match)
        tempo_valores(i) = str2double(match{1});
    end
end
tempo_valores = tempo_valores/1000;                         % tempo em segundos

% Extrair números do campo Temperatura
for i = 1:length(coluna2)
    % Pegar o número dentro da string
    match = regexp(coluna2{i}, '[\d.]+', 'match');
    if ~isempty(match)
        temp_valores(i) = str2double(match{1});             % Converte o valor de texto para double
    end
end

% Extrair números do campo PWM
for i = 1:length(coluna3)
    match = regexp(coluna3{i}, '\d+', 'match');  % pega PWM
    if ~isempty(match)
        pwm_valores(i) = str2double(match{1});              % Converte o valor de texto para double
    end
end

% Exemplo: plotar temperatura vs. tempo
figure(1);
plot(tempo_valores, temp_valores);
xlabel('Tempo (s)');
ylabel('Temperatura (°C)');
title('Temperatura ao longo do tempo');
grid on;

```

Figura 36: Gráfico do resultado da temperatura ao longo do tempo.

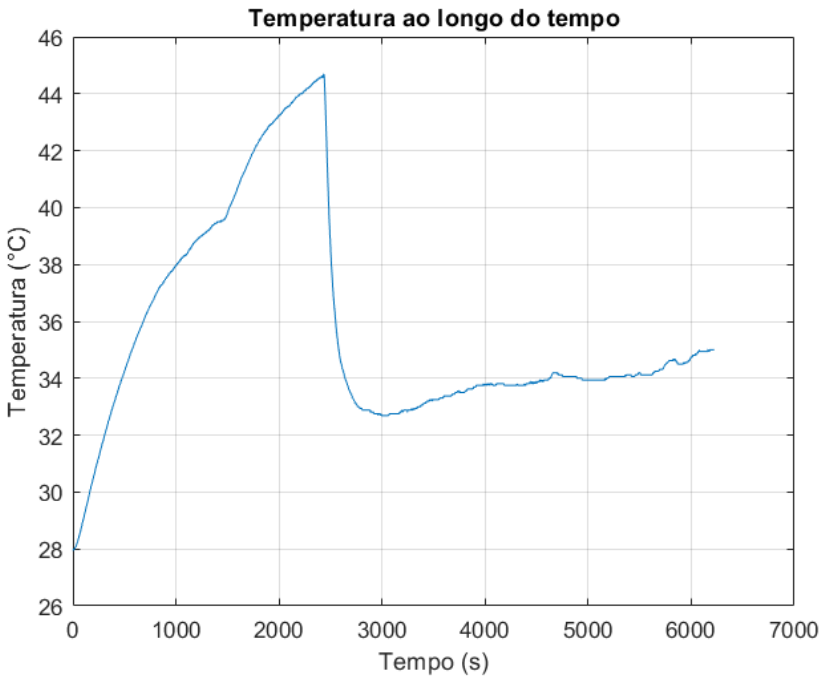
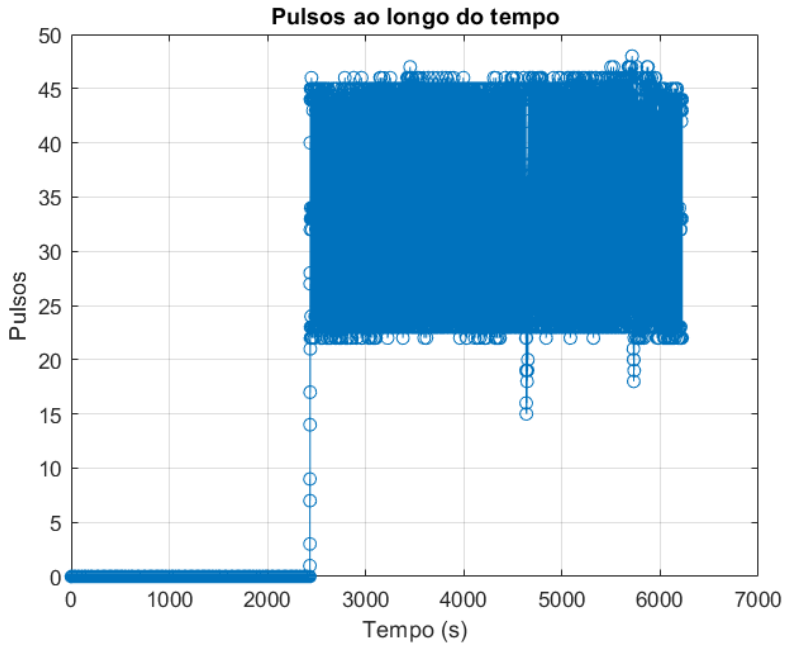


Figura 37: Código do tratamento dos dados de vazão e plotagem dos resultados de vazão e PWM.

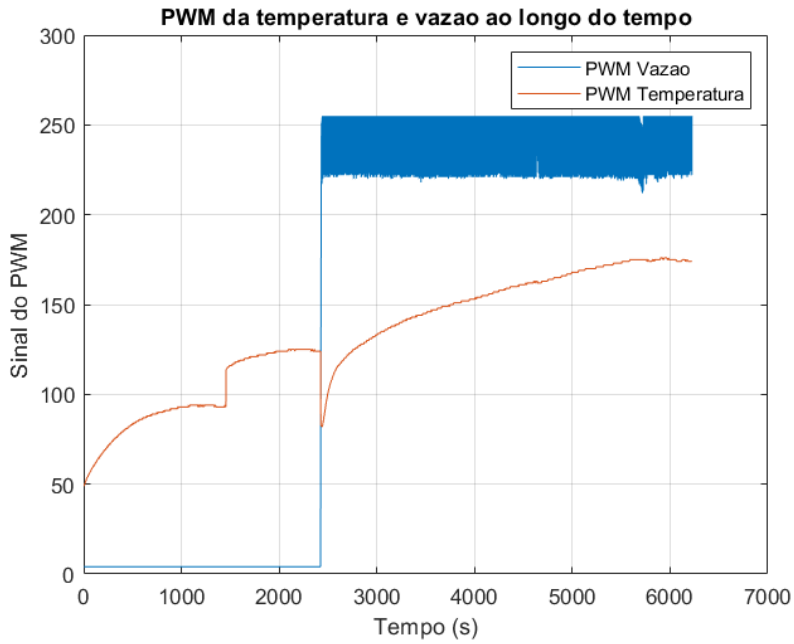
```
%% Dados de Vazao
% Abrir o arquivo original
filename = 'resultados.csv';
fid = fopen(filename, 'r');


fclose(fid);

% Inicializar colunas para cada campo
col1 = {};                                                  % Tempo
col2 = {};                                                  % Pulsos
col3 = {};                                                  % PWM Cooler

% Separar os campos por virgula
for j = 1:length(linhas_vazao)
    campos = strsplit(linhas_vazao{j}, ',');
    if length(campos) >= 3
        col1{end+1, 1} = strtrim(campos{1});
        col2{end+1, 1} = strtrim(campos{2});
        col3{end+1, 1} = strtrim(campos{3});
    end
end

% Extrair valores numéricos
tempo_vazao = zeros(length(col1), 1);
pulsos_vazao = zeros(length(col2), 1);
pwm_vazao = zeros(length(col3), 1);

for k = 1:length(col1)
    m1 = regexp(col1{k}, '\d+', 'match');
    if ~isempty(m1), tempo_vazao(k) = str2double(m1{1}); end

    m2 = regexp(col2{k}, '\d+', 'match');
    if ~isempty(m2), pulsos_vazao(k) = str2double(m2{1}); end

    m3 = regexp(col3{k}, '\d+', 'match');
    if ~isempty(m3), pwm_vazao(k) = str2double(m3{1}); end
end
tempo_vazao = tempo_vazao/1000;

% Exemplo de gráfico: Pulsos ao longo do tempo
figure(2);
plot(tempo_vazao, pulsos_vazao, '-o');
xlabel('Tempo (s)');
ylabel('Pulsos');
title('Pulsos ao longo do tempo');
grid on;

figure(3);
plot(tempo_vazao, pwm_vazao);
hold on
plot(tempo_valores, pwm_valores)
xlabel('Tempo (s)');
ylabel('Sinal do PWM');
title('PWM da temperatura e vazao ao longo do tempo');
legend('PWM Vazao', 'PWM Temperatura')
grid on;

```

Figura 38: Gráfico do resultado da vazão ao longo do tempo.



Outra análise importante de ser realizada é quanto ao sinal de PWM enviado, assim, abaixo tem-se o gráfico do sinal enviado relativos à vazão e temperatura.

Figura 39: Gráfico do sinal enviado ao PWM da resistência (temperatura) e cooler (vazão).

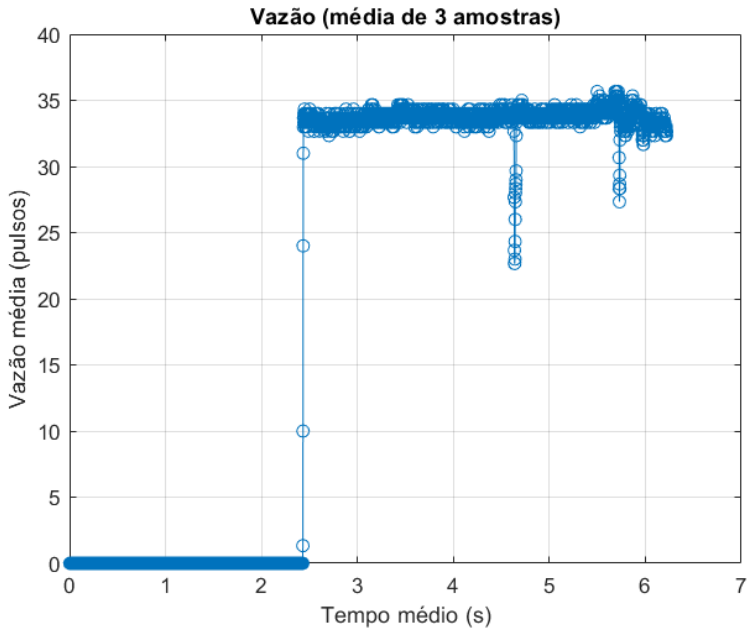


Pela Figura 38 do gráfico da vazão percebe-se que os pulsos se mantém, em torno do setpoint definido de 35 pulsos, porém, para melhor visualização, foi realizada a média dos pulsos a cada 3 amostragens, e, com isso, foi montado o gráfico abaixo, com a vazão média, a partir do código também disponibilizado a seguir..

Figura 40: Código para tratamento dos dados de vazão e aproximação pela média.

```
%% Vazao Media
% Supondo que você já tem o vetor pulsos_vazao

% Número total de grupos de 3
n_grupos = floor(length(pulsos_vazao) / 3);

% Inicializar vetor de médias
vazao_media = zeros(n_grupos, 1);

for i = 1:n_grupos
    % início do grupo
    idx = (i-1)*3 + 1;
    grupo = pulsos_vazao(idx:idx+2);
    vazao_media(i) = mean(grupo);
end

% Para tempo médio dos mesmos grupos (opcional, para graficar)
tempo_medio = zeros(n_grupos, 1);
for i = 1:n_grupos
    idx = (i-1)*3 + 1;
    grupo = tempo_vazao(idx:idx+2);
    tempo_medio(i) = mean(grupo);
end
tempo_medio = tempo_medio/1000;

% Gráfico: Vazão média por tempo médio
figure(4);
plot(tempo_medio, vazao_media, '-o');
xlabel('Tempo médio (s)');
ylabel('Vazão média (pulsos)');
title('Vazão (média de 3 amostras)');
grid on;
```

Figura 41: Gráfico do resultado de vazão através da média de 3 amostras.



9. Conclusão

Para o desenvolvimento do sistema de controle de temperatura e vazão, foi necessário várias etapas. Inicialmente, foram elaborados os circuitos eletrônicos responsáveis pela alimentação, aquisição de dados por meio de sensores e processamento, os quais deram origem às plantas que seriam posteriormente analisadas e controladas. Paralelamente, foi construída uma estrutura física adequada. Após a validação inicial dos circuitos em protoboard, procedeu-se à montagem da Placa de Circuito Impresso (PCB) e à integração de toda a estrutura física do sistema.

Após, o projeto consistiu na identificação das funções de transferência associadas às variáveis de temperatura, vazão e perturbação. A partir dessas funções, foram projetados compensadores específicos para as plantas de temperatura e de vazão, com o objetivo de garantir um desempenho satisfatório do sistema. Inicialmente, esses controladores foram obtidos no domínio contínuo e, para viabilizar sua implementação na ESP32, foi necessário discretizá-los utilizando o método de Tustin. Por fim, foram realizados ensaios práticos para avaliar a efetividade do controle no sistema real, permitindo validar o desempenho dos compensadores projetados.

Portanto, os resultados obtidos nos ensaios em ambiente real demonstraram um desempenho satisfatório do sistema de controle de temperatura e vazão desenvolvido. Isso evidencia a eficácia dos controladores projetados e valida as etapas de desenvolvimento adotadas ao longo do projeto, desde a modelagem e simulação até a implementação prática. Dessa forma, pode-se afirmar que os objetivos propostos foram alcançados com êxito.

Em síntese, o projeto alcançou seus objetivos, integrando com sucesso conhecimentos teóricos e práticos para desenvolver um sistema de controle eficiente e estável. Os desafios enfrentados ao longo do processo foram superados por meio de abordagens sistemáticas e colaborativas, resultando em um sistema que atende às especificações desejadas e valida as metodologias empregadas. Este trabalho reforça a importância do controle realimentado em aplicações práticas e destaca a relevância de ferramentas computacionais, como MATLAB e Simulink, no projeto e análise de sistemas dinâmicos.

Referências Bibliográficas

[1] DEPARTAMENTO, D.; DE COMPUTAÇÃO, E.; AUTOMAÇÃO. UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE CENTRO DE TECNOLOGIA SISTEMAS DE CONTROLE I Professor: Fábio Meneghetti Ugulino de Araújo. [s.l: s.n.]. Disponível em: <https://www.dca.ufrn.br/~meneghet/FTP/Controle1/Controle%20I%20-%20Apostila.pdf>.

[2] NISE, N. S. Engenharia de Sistemas de Controle. [s.l: s.n.]