Universidade Federal de Uberlândia
Faculdade de Engenharia Elétrica
Sistemas de Controle Realimentado
Docente: Gabriela Vieira Lima

Projeto Final

Controle de Temperatura e Fluxo

Autores:

- Beatriz Martins Gomes Silva - 12121EBI003
- Júlia Miranda Brito - 12121EBI020
- Luiz Felipe Spinola Silva - 12121EBI001

Uberlândia, 23 de outubro de 2024

Sumário

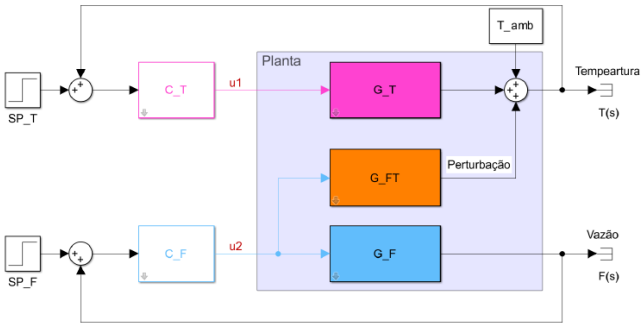
1 Objetivos

O objetivo principal deste projeto, realizado como trabalho final da disciplina de Experimental de Sistema de Controle Realimentando, é desenvolver um sistema de controle de temperatura e de vazão. Para alcançar esse objetivo, foi necessário, inicialmente, construir um ambiente físico para esse sistema e, em seguida, projetar, implementar e testar um controlador capaz de regular esses dois parâmetros. Dessa forma, este projeto visa consolidar os conhecimentos adquiridos nas aulas teóricas ao aplicá-los em um projeto prático.

2 Introdução

O projeto final da disciplina de Experimental de Sistemas de Controle Realimentado consiste em desenvolver um sistema de controle de temperatura e vazão, conforme observado na Figura 1. Nesse contexto, 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. Para tanto, eles utilizam sensores para captar informações sobre as condições operacionais da planta, ou seja, eles são responsáveis pelo monitoramento de um sistema físico, composto por elementos e componentes interconectados.

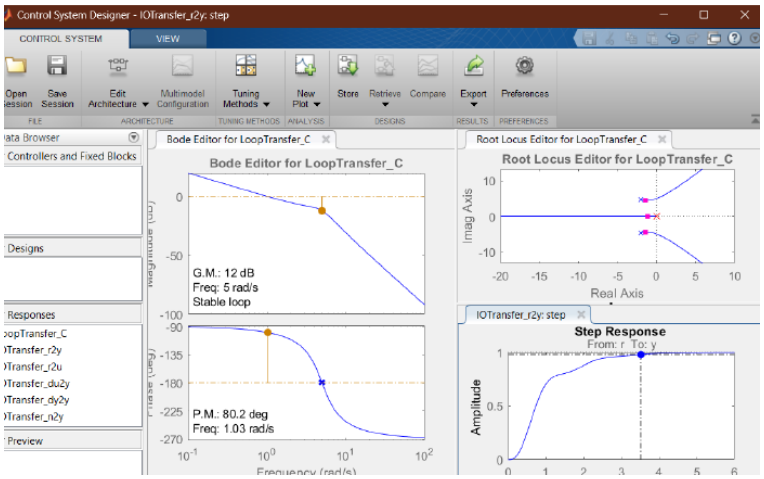
Figura 1 – Blocos gráficos do projeto final de Sistema de Controle Realimentado



Fonte: *Laboratório 7 - ESCR*, 2024

A ferramenta *Control System Designer*(*Sisotool*), disponível no Matlab, é utilizada frequentemente para projetos de controladores em sistemas SISO (*Single Input, Single Output*). Isso porque, ela oferece uma série de recursos interativos, observados na figura 2, que facilitam a análise e o *design* de controladores através de editores gráficos do diagrama de *Bode* e do lugar das raízes. Além disso, o *Sisotool* possibilita que haja interação com outros pacotes adicionais do *MATLAB*, como o *Simulink*, responsável pela simulação e modelagem de sistemas por meio de blocos gráficos.

Figura 2 – Interface *Sisotool*



Fonte: *Laboratório 5 - ESCR*, 2024

Os controladores PD (Proporcional Derivativo), PI (Proporcional Integral) e PID (Proporcional Integral Derivativo) são bastante utilizados em sistemas de controle e, muitas das vezes, são projetados por meio dessas ferramentas disponibilizadas pelo *MATLAB*. Enquanto os controladores PD são responsáveis por ajustar a saída do sistema por meio da proporção entre o valor desejado e o valor medido e por meio da taxa de variação desse erro, os controladores PID adicionam mais uma ação de controle, em que há o acúmulo do erro ao longo do tempo, corrigindo erros do regime estacionário e eliminando erros de regime permanente. Dessa forma, compensadores PI são utilizados onde o foco é a precisão no regime permanente e não há a necessidade de aumentar a velocidade da resposta daquele sistema.

Além disso, é importante notar que para a implementação digital desses compensadores é necessário realizar a discretização das equações de controle. Isso porque, microcontroladores, como *ESP32* e *Arduino*, operam de forma discreta, ou seja, eles processam e coletam informações em intervalos de tempo pré-definidos (tempo de amostragem). Dessa maneira, as equações projetadas no domínio contínuo de *Laplace* devem ser adaptadas para o domínio discreto, utilizando equações de diferenças.

Diante disso, este relatório possui como objetivo relatar, detalhadamente, o processo para o desenvolvimento desse sistema de controle de temperatura e vazão. Para isso, aborda desde o desenvolvimento do esquemático e *layout* até a montagem da Placa de Circuito Impresso. Além disso, inclui a identificação das plantas no domínio contínuo, o projeto dos controladores e sua implementação digital.

3 Materiais e Métodos

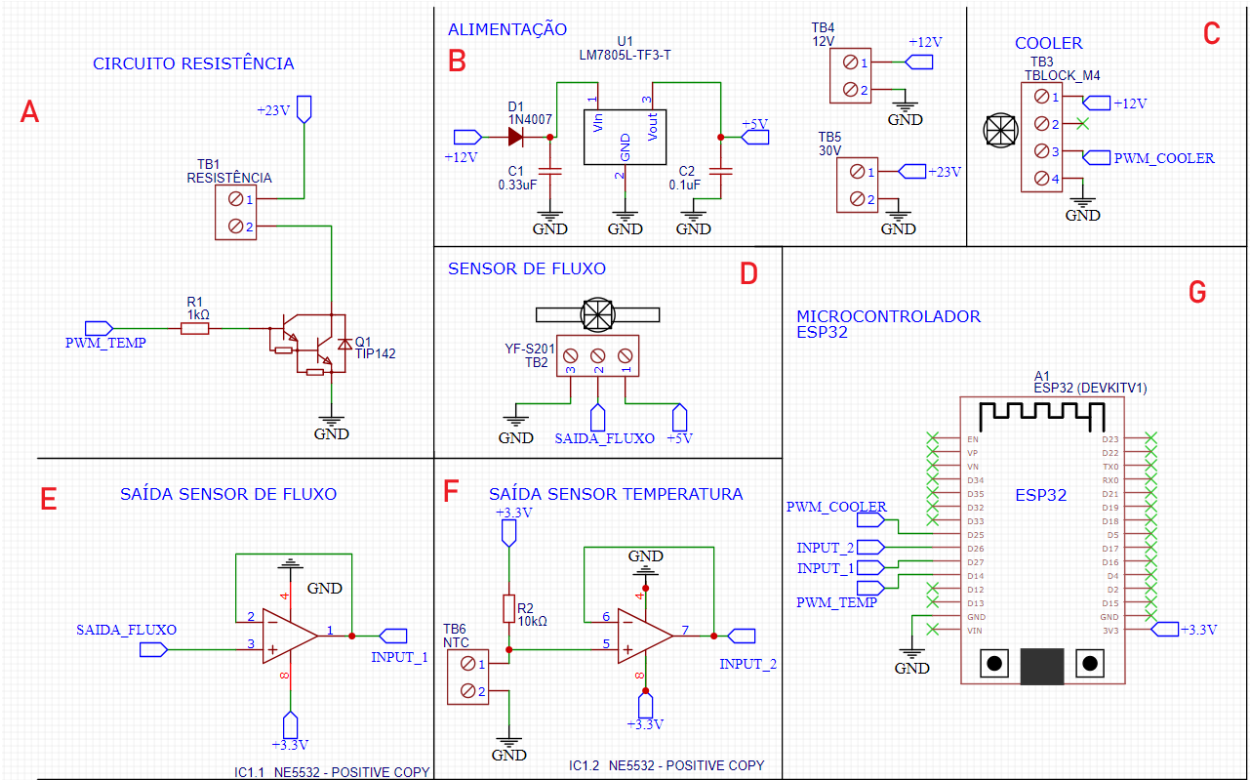
3.1 Esquemático

Para a construção do projeto, primeiramente, foi necessário o desenvolvimento de circuitos eletrônicos capazes de fornecerem as condições necessárias para o sistema desejado. O primeiro aspecto a ser determinado foram os componentes eletrônicos responsáveis para a criação das plantas de temperatura e fluxo, seus respectivos circuitos de alimentação, os sensores destas variáveis e a unidade de processamento desses dados (que posteriormente seria responsável pelo controle digital do sistema). A lista completa de componentes eletrônicos utilizados no projeto pode ser visto abaixo.

- 2 Amplificadores Operacionais NE5532
- 1 Capacitor de Poliéster de $330nF$
- 1 Capacitor de Poliéster de $100nF$
- 2 Conectores tipo *Borne* para *plug* "Banana"pretos
- 2 Conectores tipo *Borne* para *plug* "Banana"vermelhos
- 4 Conectores tipo *Borne* KRE de 2 entradas
- 1 Conector tipo *Borne* KRE de 3 entradas
- 1 *Cooler* $92x92x25mm$ $12V$ de 4 fios
- 1 Diodo 1N4007
- 1 Microcontrolador ESP32 USB-C de 30 fios
- 1 Regulador de tensão LM7805
- 1 Resistência de chuveiro WR 7500W
- 1 Resistor $1K\Omega$
- 1 Resistor $10K\Omega$
- 1 Sensor de fluxo Yf-s201
- 1 Sensor de temperatura NTC $10K\Omega$
- 1 Transistor NPN *Darlington* de Potência TIP142

A Figura 3 ilustra o esquemático desenvolvido no *software Easy EDA* para o sistema proposto utilizando os componentes listados.

Figura 3 – Esquemático do Circuito Eletônico do Sistema



Fonte: Autoria Própria

A planta de temperatura pode ser observada no Circuito A da Figura 3: A resistência de chuveiro (representada pelo conector TB1) é a responsável pela geração de calor e está conectada a uma alimentação de +23V externa e ao terminal coletor do transistor. Sobre o transistor, este está conectado na configuração emissor comum, ou seja, seu terminal emissor está conectado ao potencial zero (GND) do circuito. Além disso, a base do transistor está conectado a um resistor de $1K\Omega$ conectado em série ao pino 14 da ESP. Dessa forma, a passagem de corrente elétrica pelo resistor de chuveiro, responsável diretamente pela geração de calor do sistema, é controlada pelo chaveamento do transistor, definido pelo *duty cycle* do sinal de PWM (*Pulse Width Modulation*) do pino da ESP. (explicado na Seção 3.3).

O sensoriamento da temperatura do sistema é realizado por meio do sensor NTC, representado pelo conector TB6 do Circuito F da Figura 3. Este sensor é um termistor, ou seja, elemento resistivo que, ao ser submetido a quaisquer mudanças de temperatura, possui grande alteração em sua impedância. Dessa maneira, conhecendo a função que relaciona a impedância do sensor a sua respectiva temperatura, é possível determinar a temperatura da planta. Sua Leitura foi feita através da leitura de tensão sob o sensor, que está conectado em série a uma resistência de $10K\Omega$ (mesmo valor de impedância média do sensor em 25^{o}C), submetida a uma tensão de +3.3V providas pelo microcontrolador.

A planta do circuito de fluxo pode ser observado nos Circuitos C e D da Figura 3. Primeiramente, tem-se o *Cooler*, responsável pela ventilação e entrada de ar no sistema, este possui 4 conexões: duas para sua alimentação, de +12V e GND (ambas providas externamente por uma fonte de bancada), uma de PWM, para controlar a velocidade angular da ventoinha, isto é, a quantidade de ar ventilada para dentro do sistema,

além de uma conexão relacionada a um tacômetro interno do componente, que fornece um sinal referente a velocidade angular do *Cooler*. Somente as três primeiras conexões foram utilizadas no projeto.

Para a medição de fluxo de ar, foi utilizado o sensor de fluxo Yf-s201, ilustrado no Circuito D da Figura 3. Este sensor possui uma conexão para alimentação de 5V, fornecida pela saída do regulador de tensão conectada a uma tensão de entrada externa de +12V (representada no Circuito B da Figura 3). Além dessa conexão, este sensor possui uma entrada para o GND do circuito e outra referente ao seu sinal de saída. Em resumo, este sensor possui um eixo com hélices as quais, ao serem sensibilizadas (ou seja, rotacionadas) pela passagem de um fluido no sentido correto, produzem uma onda quadrada de frequência diretamente proporcional a vazão do respectivo fluido (neste caso, do ar).

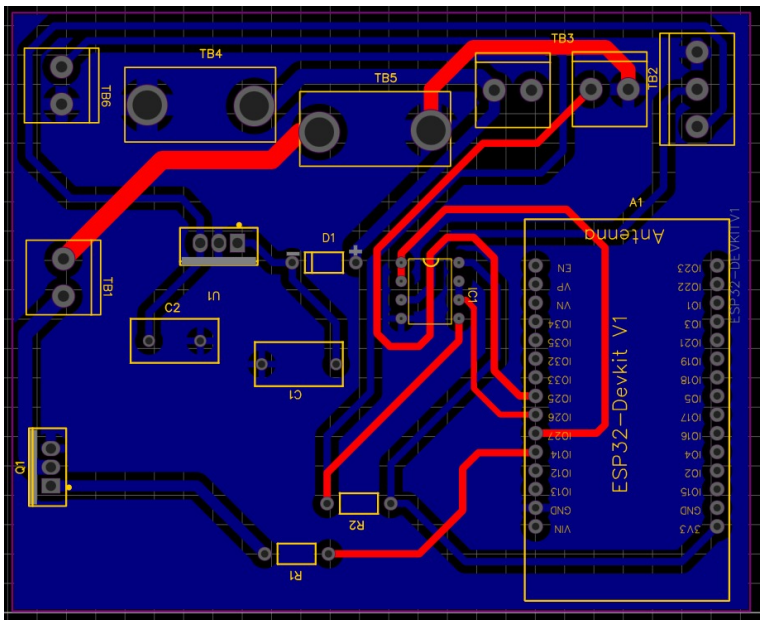
As saídas referentes a ambos sensores e suas conexões ao microcontrolador podem ser visualizadas nos Circuitos E, F e G da Figura 3. Para ambos sensores, foi utilizado um amplificador operacional entre o sinal e o microcontrolador. O AO (sigla para amplificador operacional), em ambos os casos, está projetado como um *Buffer* de tensão, ou seja, um amplificador não-inversor de ganho unitário, responsável pelo casamento de impedâncias entre o microcontrolador e o sinal externo, possibilitando uma leitura mais acurada do sinal. Além disso, o AO, por característica, limita seu sinal de saída a um intervalo referente a suas alimentações positivas e negativas e, como o AO é alimentado pelo próprio microcontrolador, isto impede a chegada de uma tensão superior a +3.3V a ESP, o que causaria comprometimento ao funcionamento do microcontrolador. Esta última utilidade do AO, em especial, é de suma importância neste projeto, já que o sinal de saída do sensor de fluxo possui amplitude de 5V que, caso fosse inserida diretamente em um pino de entrada da ESP, poderia causar seu estrago imediato.

Por fim, temos o microcontrolador, representado no Circuito G da Figura 3. Nele, utilizamos um total de 6 pinos, dois referentes a alimentação (+3.3V e GND) e 4 pinos para o envio do sinal de PWM e leitura do sinal provido pelo sensor para cada uma das duas plantas do projeto.

3.2 *Layout* e PCB

Após resultados favoráveis com o teste do esquemático na *protoboard*, foi possível montar o *layout* da placa de circuito impresso no *software* EasyEDA. Para isso, precisou-se definir o *footprint* de cada componente do esquemático, organizá-los da melhor forma e traçar as trilhas em duas superfícies diferentes. O resultado foi uma PCB de 10 cm x 8 cm, ilustrada na Figura 4.

Figura 4 – *Layout* da PCB

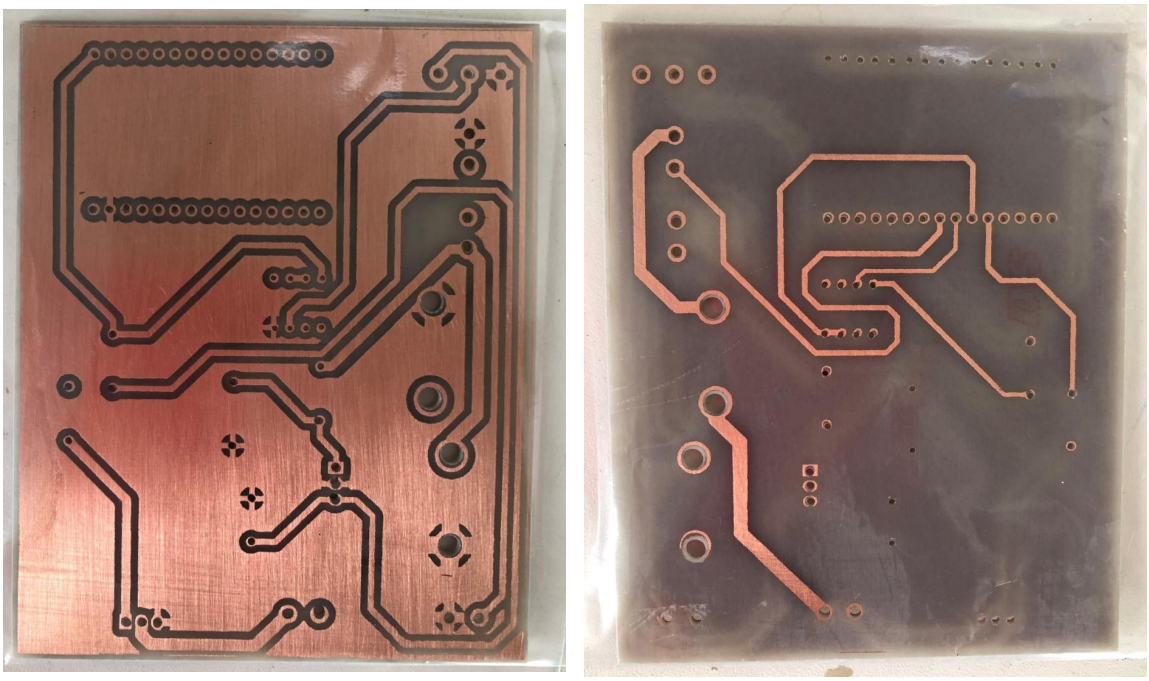


Fonte: Autoria Própria

Com o *layout* pronto e revisado, foi confeccionada a placa, como mostra a figura 5.

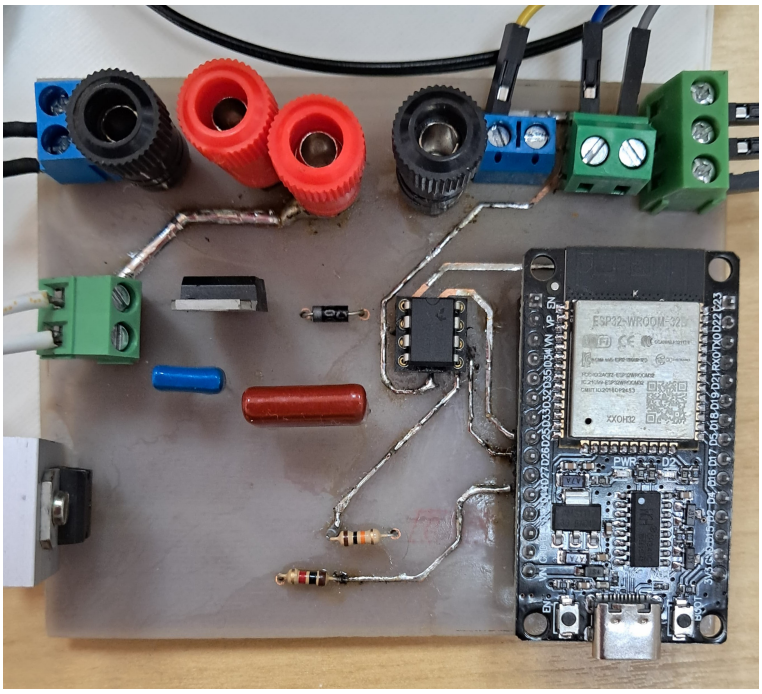
Por fim, os componentes foi inseridos e soldados na placa (Figura 6), checou-se a continuidade entre as trilhas e todo o projeto foi testado.

Figura 5 – Faces da PCB



Fonte: Autoria Própria

Figura 6 – PCB finalizada



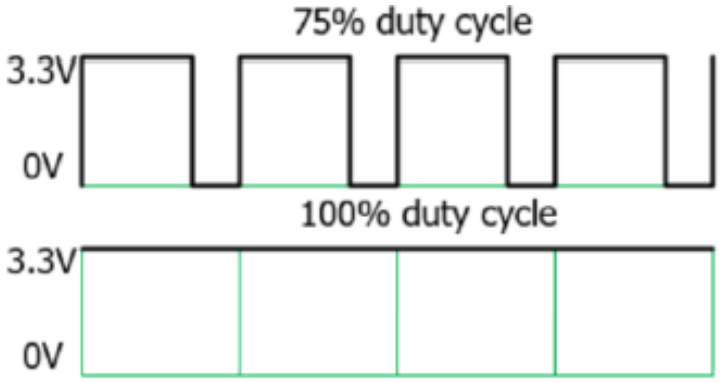
Fonte: Autoria Própria

3.3 Programação do Sistema em Malha Aberta no Microcontrolador

Com a finalização e montagem da Placa de Circuito Impresso (PCB), foi possível dar início à programação desse sistema em malha aberta. A principal função desse primeiro código desenvolvido é automatizar o controle dos sinais de onda *PWM* dos circuitos de aquecimento e resfriamento.

As portas *PWM* do microcontrolador ESP32 fornecem sinais de onda quadrada em que a proporção de tempo em que a onda está em estado "alto"e o tempo em que a onda está em "baixo"definem o ciclo de trabalho (*duty cycle*) e a potência média da onda. Dessa forma, se o *duty cycle* for definido como 100% a onda ficará todo o tempo em estado "alto". Já se definir um *duty cycle* de 75% a onda ficará 3/4 do tempo em estado "alto", conforme é possível observar na Figura 7. Dessa forma, haverá uma variação de temperatura e vazão, por meio da alteração no código, das variáveis de *PWM* do *cooler* (vazão) e do transistor do circuito da fonte de calor (temperatura).

Figura 7 – Sinais de onda quadrada de *PWM*



Fonte: *Laboratório 3 - SCR*, 2024

Com a ideia principal do código definida, foi possível implementá-la na IDE do Arduino, uma plataforma para programação de microcontroladores do tipo ESP32 ou Arduino. O primeiro passo, conforme pode ser visto na Figura 8, foi incluir as duas bibliotecas necessárias para esse projeto: <*Arduino.h*> e <*math.h*>. Em seguida, definiu-se os pinos 27 e 26 para recepção dos sinais correspondentes ao sensor de fluxo e ao sensor de temperatura, respectivamente. Já os pinos 14 e 25 foram configurados para realizar o envio dos sinais de *PWM* presentes no sistema, sendo o pino 14 responsável pelo *PWM* da temperatura e o pino 25 pelo *PWM* do *cooler* temperatura.

Para concluir esta primeira etapa, definiu-se as variáveis do projeto. A variável *int resolution* representa a resolução de 12 bits do conversor ADC do ESP32, permitindo até 4095 valores distintos. As variáveis de tempo, *long tempoAnt, long tempoAtual* e *dT*, são responsáveis por controlar o tempo de amostragem dos dados. Além disso, definiu-se as variáveis *float temperatura, float resistencia* e *vazao* para cálculos posteriores relacionados à medição da temperatura e vazão. Os parâmetros *int dutycycleTemp* e *int dutycycleCooler* são responsáveis por ajustar a potência enviada ao sistema de aquecimento e ao cooler, respectivamente. Por fim, as constantes Vs, R1, Beta, To, e Ro são específicas para o cáculo de temperatura baseado no sensor NTC, utilizando a equação de Steinhart-Hart.

Figura 8 – Definição das variáveis no código de Malha Aberta

```
#include <Arduino.h>
#include <math.h>    // Biblioteca para operações matemáticas (logaritmo, etc)

// Pinos
#define PWM_TEMP 14
#define PWM_COOLER 25
#define SENSOR_TEMP 26
#define SENSOR_FLUXO 27

// Variáveis
const int resolution = 4095;                                  // Resolução do conversor AD do ESP
volatile int pulsos = 0;                                      // Pulsos do sensor de fluxo
unsigned long tempoAnt = 0, tempoAtual = 0, dT = 0;           // Variáveis para manipulação do timer
float temperatura = 0, resistencia = 0, vazao = 0;            // Variáveis para cálculo da temperatura e vazão
int dutycycleTemp = 0, dutycycleCooler = 0;                   // Duty Cicle do PWM do transistor e do cooler

// Constantes do NTC e sistema
double Vs = 3.3;                                              // Tensão de referência do ESP32
double R1 = 10000;                                            // Resistor fixo de 10kΩ no divisor de tensão
double Beta = 3950;                                           // Coeficiente beta do NTC
double To = 298.15;                                           // Temperatura de referência (25°C em Kelvin)
double Ro = 10000;                                            // Resistência nominal do NTC a 25°C
```

Fonte: Autoria Própria, 2024

Após isso, foi dado início a configuração das funções deste projeto. A primeira função definida, observada na Figura 9, é a *void lerFluxo()*, executada sempre que o sensor de fluxo detecta um pulso, incrementando a variável *pulsos*.A função *void setup ()* é executada uma única vez ao inciar o programa, sendo responsável pela definição dos pinos dos sensores e dos pinos PWM como saída (*output*) ou entrada (*input*).Além disso, as funções *ledcAttach* e *ledcWrite* são usadas para configurar e gerar sinais PWM nos pinos de temperatura e *cooler*, com uma frequência de 1kHz e resolução de 8 bits. Ao final dessa função, a interrupção é configurada para o sensor de fluxo, chamando a função *void lerFluxo()* sempre que um pulso é detectado.

Figura 9 – Definição das funções *void lerFluxo()* e *void setup ()* no código de Malha Aberta

```
// Rotina da interrupção que realiza leitura do sensor de fluxo
void lerFluxo(){
  pulsos++;
}

void setup() {
  Serial.begin(230400);
  pinMode(SENSOR_TEMP, INPUT);
  pinMode(SENSOR_FLUXO, INPUT);
  pinMode(PWM_TEMP, OUTPUT);
  pinMode(PWM_COOLER, OUTPUT);

  dutycicleTemp = 50;                // Duty Cicle setado
  dutycicleCooler = 50;

  ledcAttach(PWM_TEMP, 1000, 8);     // Configurando PWMs
  ledcAttach(PWM_COOLER, 1000, 8);

  ledcWrite(PWM_TEMP, (dutycicleTemp*255)/100);  // Mandando sinais PWM
  ledcWrite(PWM_COOLER, (dutycicleCooler*255)/100);

  attachInterrupt(digitalPinToInterrupt(SENSOR_FLUXO), lerFluxo, RISING); // Interrupção para sensor de fluxo
}
```

Fonte: Autoria Própria, 2024

Por fim, a função *void loop* foi definida. Nessa função, conforme é visto na figura 10, o tempo é determinado por um *timer* em milissegundos. Em seguida, a variável dT é responsável por controlar o tempo de amostragem em 2 segundos, ou seja, toda a rotina desse código é executada de 2 em 2 segundos. Além disso, o valor do sensor de temperatura é lido analogicamente através do *analogRead* e, após isso, os cálculos de temperatura são feitos usando a equação de Steinhart-Hart. O valor de vazão é calculado diretamente a partir dos pulsos registrados pela interrupção. Ao final, os valores de temperatura e vazão são exibidos pelo monitor serial.

Figura 10 – Definição da função *void loop()* no código de Malha Aberta

```
48  void loop() {
49    tempoAtual = millis();
50    dT = tempoAtual - tempoAnt;
51
52    // Tempo de Amostragem 2000ms = 2s
53    if (dT >= 2000) {
54      tempoAnt = tempoAtual;
55
56      // Leitura do NTC e cálculo da temperatura diretamente no loop
57      int leituraADC = analogRead(SENSOR_TEMP);
58
59      // Cálculo da temperatura com a equação Steinhart-Hart
60      double Vout = leituraADC * Vs / resolution;  // Converte a leitura ADC para tensão
61      double Rt = R1 * Vout / (Vs - Vout);         // Calcula a resistência do NTC
62      temperatura = 1 / (1 / To + log(Rt / Ro) / Beta); // Equação de Steinhart-Hart
63      temperatura = temperatura - 273.15;  // Converte de Kelvin para Celsius
64
65      // Cálculo da vazão
66      vazao = pulsos;
67
68      pulsos = 0;
69
70      // Exibição da vazão e temperatura no Serial
71      Serial.println(">Vazão:");
72      Serial.print(vazao);
73      Serial.println("L/s");
74      Serial.println(">Temperatura:");
75      Serial.print(temperatura);
76      Serial.println("°C");
77    }
78  }
```

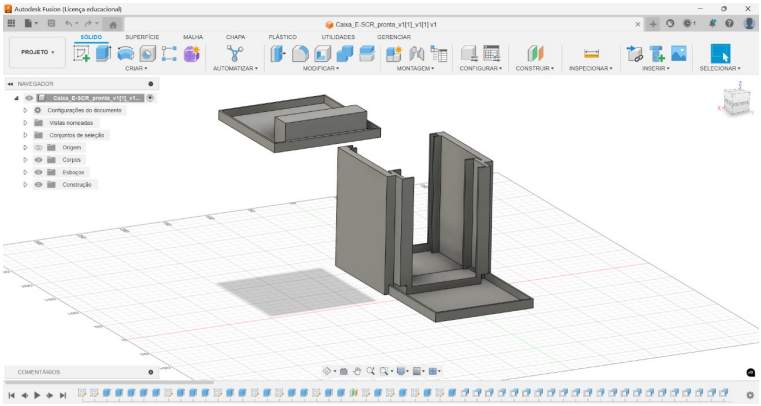
Fonte: Autoria Própria, 2024

3.4 Encerramento da 1ª Fase: Projeto Físico e Funcionamento

Com o código finalizado e a Placa de Circuito Impresso (PCB) pronta, foi possível integrar todos os componentes em um ambiente físico adequado. Para tanto, decidiu-se modelar esse sistema no *software Fusion 360*, ideal para modelagem em 3D. A ideia foi criar uma caixa retangular, vista na figura 11, para acomodar o sistema de aquecimento de forma eficiente. Em seguida, modelou-se uma tampa para essa caixa, juntamente com um compartimento específico para acoplar a PCB.

Além disso, um dos lados da caixa foi projetado com uma abertura específica para a instalação de um funil, que tem a função de acoplar o sensor de fluxo. Esse funil foi estrategicamente incluído para garantir que o ar gerado pelo *cooler* seja direcionado de maneira eficiente para o sensor, otimizando a medição da vazão de ar.

Figura 11 – Modelagem do sistema físico no *software Fusion 360*



Fonte: Autoria Própria, 2024

Após a modelagem 3D, todos os componentes foram impressos utilizando a impressora Ender 3 do laboratório BIOLAB. O processo de impressão levou, aproximadamente, mais de um dia, dividido entre a impressão da caixa juntamente com o suporte para PCB e a tampa. Durante a impressão, foram utilizados filamentos PLA na cor branca e o resultado pode ser visto na figura 12 e 13.

Figura 12 – Impressão da caixa e da tampa

Fonte: Autoria Própria, 2024

Figura 13 – Instalação do Funil

Fonte: Autoria Própria, 2024

Depois de todas a peças prontas e montadas, foi testado os códigos apresentados na seção 3.3 para verificar se a estrutura não atrapalhava a medição realizada pelos sensores. O resultado foi satisfatório visto que os valores de temperatura e vazão registrados estavam corretos.

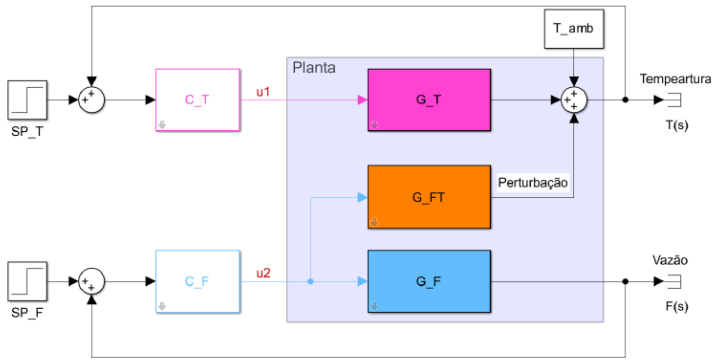
4 Resultados e Discussões

4.1 Identificação das Plantas G(s)

Inciou-se a segunda fase deste projeto, conforme pode ser observado na Figura 14, pela identificação das três funções de transferência G(s):

- Função de transferência da temperatura: G_T(s)
- Função de transferência da vazão (fluxo): G_F(s)
- Função de transferência da pertubação da vazão na temperatura: G_{FT}(s).

Figura 14 – Identificação das funções de transferência deste projeto



Fonte: *Laboratório 7 - ESCR*, 2024

Para identificar a função G_T(s), foi necessário fazer um ensaio em que o *cooler* estava desconectado do sistema e aplicou-se um sinal de *duty cycle* de 100% para a resistência de aquecimento. Dessa forma, coletou uma amostra da temperatura, exibidas no monitor serial, a cada período de amostragem de 0,4 segundos. Continuou-se o ensaio até que a temperatura atingisse o regime permanente.

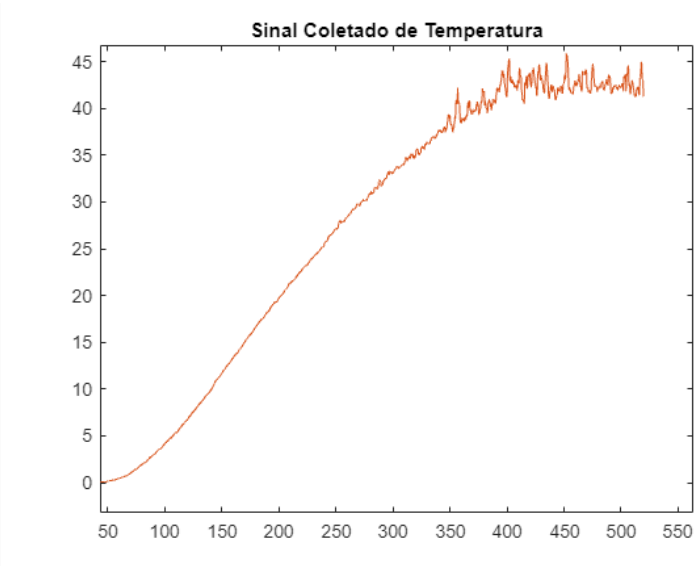
Em seguida, carregou-se os dados no *MATLAB*, observado na Figura 15, para dar início ao processo de identificação da função de temperatura. O próximo passo foi subtrair a temperatura ambiente para que o gráfico partisse da origem e definiu-se uma variável para o tempo de amostragem. Após isso, aplicou-se um filtro de média móvel para suavizar o sinal coletado e plotou-se o gráfico desse sinal, observado na Figura 16.

Figura 15 – Código para identificação da função de transferência da temperatura

```
temperatura = load("ENSAIO 3.2.txt").';                                  % Carregando sinal
temperatura = temperatura - 26.05;                                       % Delta de temperatura (comprimento do vetor: 2171)

tSample = 0.4;                                                           % Período de amostragem (s)

% Filtro de Média Móvel

kernelSize = 5;
kernel = (1/kernelSize)*ones(1,kernelSize);

tempMediamovel = filter(kernel,1,temperatura);

% Função de transferência da planta de temperatura
timeVector = (0:tSample:(1299)*tSample).';                               % Vetor de tempo
inputVector = cat(2, zeros(1,100),255.*(ones(1,1200))).';                % Sinal de entrada
outputVector = cat(2, zeros(1,100), tempMediamovel(1:1200)).';           % Sinal de saída
timeSample = 0.400;                                                      % Vetor de tempo

figure()
plot(timeVector,inputVector,timeVector,outputVector)
title('Sinal Coletado de Temperatura');
legend('Entrada','Saída');
xlim([-20,500])
ylim([0,50])
```

Fonte: *Autoria Própria*, 2024

Figura 16 – Gráfico do sinal coletado do ensaio de temperatura



Fonte: *Autoria Própria*, 2024

Para concluir a identificação de G_T, conforme pode ser observado na figura 17, criou-se um vetor *data2* que recebe um objeto *iddata*, responsável por identificar um sistema através das informações de entrada, saída e o tempo de amostragem. Em seguida, utilizou-se *tfest* para criar modelos matemáticos que possam prever o comportamento desse ensaio de temperatura. Para isso, foi necessário descrever a quantidade de polos e zeros, através de *np* e *nz*, que o sistema pode possuir. Dessa forma, o *iddata* organiza os dados experimentais de forma que possam ser utilizados por outro objeto, como *tfest*, para estimar a função de transferência desse sistema por meio de modelos matemáticos.

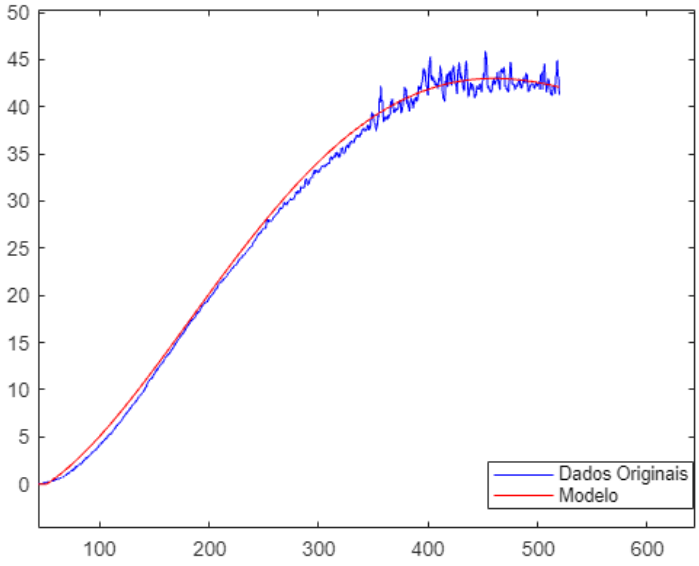
Figura 17 – Código para identificação da função de transferência da temperatura

```
% Função de transferência da planta de temperatura
timeVector = (0:tSample:(1299)*tSample).';                          % Vetor de tempo
inputVector = cat(2, zeros(1,100),255.*(ones(1,1200))).';           % Sinal de entrada
outputVector = cat(2, zeros(1,100), tempMediamovel(1:1200)).';      % Sinal de saída
timeSample = 0.400;                                                 % Vetor de tempo

figure()
plot(timeVector,inputVector,timeVector,outputVector)
title('Sinal Coletado de Temperatura');
legend('Entrada','Saída');
xlim([-20,500])
ylim([0,50])

%Identificação do sistema
data2 = iddata(outputVector,inputVector,timeSample);                % Ordem: Saída, entrada, tempo de amostragem
np = 2;                                                             % Número de polos
nz = 1;                                                             % Número de Zeros
ft_ident2 = tfest(data2,np,nz, NaN);                                % Último parâmetro: atraso
y_sim2 = lsim(ft_ident2,inputVector,timeVector);

% Resultados
figure()
plot(timeVector,outputVector,'b',timeVector,y_sim2,'r',timeVector,inputVector, 'g');
legend('Dados Originais','Modelo')
ylim([-5 50])
```

Fonte: *Autoria Própria*, 2024

Finalmente, plotou-se o gráfico, visto na figura 18, do comportamento dado pelo *tfest* e descobriu-se a função de transferência da temperatura dada pela equação (1).

Figura 18 – Gráfico do sinal de temperatura tratado



Fonte: *Autoria Própria*, 2024

$$G_T(s) = e^{-12s}\frac{0.00030071(s+0.02221)}{s^2+0.004922s+5.324*10^{-5}} \tag{1}$$

Para identificar a próxima função de transferência G_F(s), foi necessário fazer um ensaio em que o *cooler*

inicia desligado mas, após alguns segundos, é aplicado um sinal PWM de 100% nele. Dessa forma, coletou uma amostra da vazão, exibidas no monitor serial, a cada período de amostragem de 0.2 segundos. Continuou-se o ensaio até que a vazão atingisse o regime permanente.

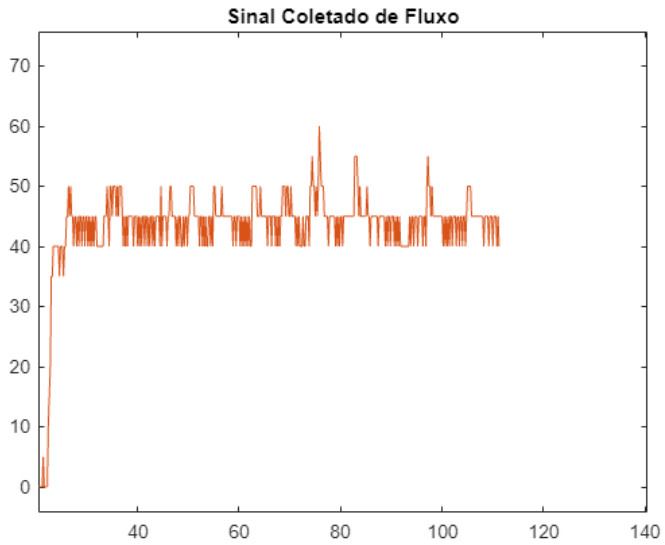
Em seguida, repetiu-se o mesmo próximo de coleta de dados no *Matlab* feita anteriormente, conforme pode ser na figura 19. Após isso, plotou-se o gráfico do sinal coletado de fluxo, visto na figura 20.

Figura 19 – Código para identificação da função de transferência de fluxo

```
% Função de transferência da planta de fluxo
fluxo = load("FLUXO SCR.txt").';
outputFluxo = fluxo./0.2;                                   % Pulsos por segundo
passoFluxo = 0.2000;                                        % Tempo de amostragem
tempoFluxo = (0:passoFluxo:(length(fluxo)+99)*passoFluxo);  % Vetor de tempo
inputFluxo = cat(2, zeros(1,100),255.*(ones(1,457))).';     % Sinal de entrada
outputFluxo = cat(2, zeros(1,100), outputFluxo).';          % (Sinal de saída)

figure()
plot(tempoFluxo,inputFluxo,tempoFluxo,outputFluxo)
title('Sinal Coletado de Fluxo');
legend('Entrada','Saída');
xlim([0,120])
ylim([0,80])
```

Fonte: *Autoria Própria*, 2024

Figura 20 – Gráfico do sinal coletado do ensaio de fluxo



Fonte: *Autoria Própria*, 2024

Utilizou-se, novamente, *iddata* e *tfest* para criar o modelo matemático que possa prever o comportamento desse ensaio de fluxo, conforme visto na figura 21.

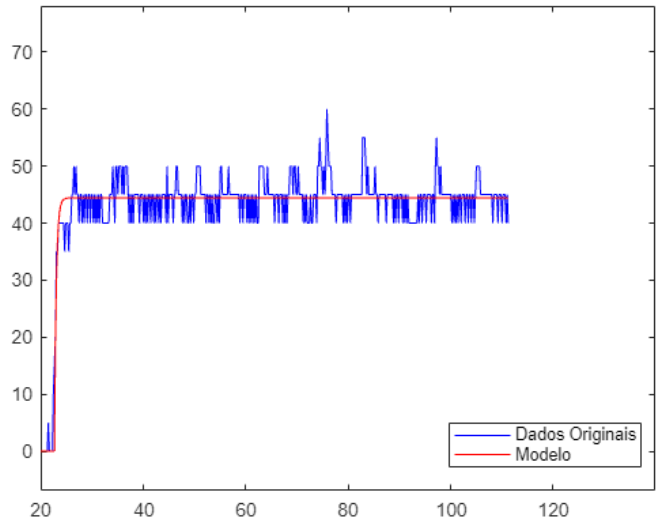
Figura 21 – Código para identificação da função de transferência de fluxo

```
% Identificação do sistema
dataFluxo = iddata(outputFluxo,inputFluxo,passoFluxo);                    % Ordem: Saída, entrada, tempo de amostragem
np = 1;                                                                   % Número de polos
nz = 0;                                                                   % Número de zeros
ft_identFluxo = tfest(dataFluxo,np,nz, NaN);                              % Último parâmetro: atraso
y_simFluxo = lsim(ft_identFluxo,inputFluxo,tempoFluxo);

% Resultados
figure()
plot(tempoFluxo,outputFluxo,'b',tempoFluxo,y_simFluxo,'r',tempoFluxo,inputFluxo, 'g');
legend('Dados Originais','Modelo')
ylim([-5,80])
```

Fonte: *Autoria Própria*, 2024

Finalmente, plotou-se o gráfico, visto na figura 22, do comportamento dado pelo *tfest* e descobriu-se a função de transferência do fluxo dada pela equação (2).

Figura 22 – Gráfico do sinal de fluxo tratado



Fonte: *Autoria Própria*, 2024

$$G_F(s) = e^{-2.6s}\frac{0.49794}{(s+2.859)} \tag{2}$$

Por fim, para finalizar a Seção de Identificação das funções de transferência, encontrou-se G_{FT}(s). Para isso, iniciou o ensaio com a resistência ligada com PWM máximo e aguardou a temperatura estabilizar. Em seguida, aplicou um sinal *PWM* de 100% no *cooler*. Dessa forma, coletou-se uma amostra da temperatura a cada período de amostragem. Continuou-se o ensaio até que a temperatura atinja um novo regime permanente.

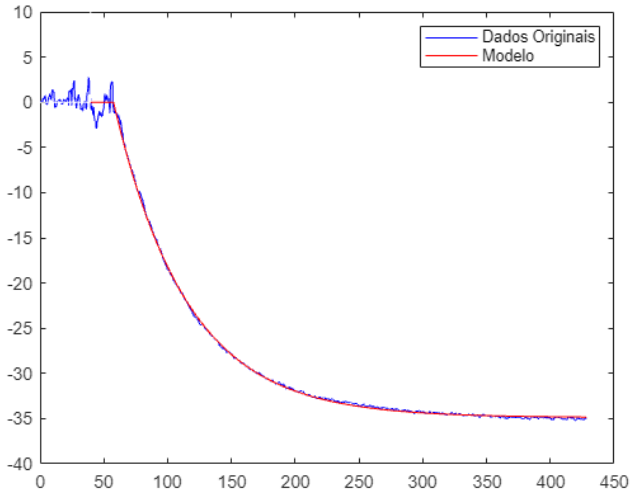
Posteriormente, repetiu-se o mesmo próximo de coleta de dados no *Matlab* feita nas etapas anteriores, conforme pode ser na figura 23. Após isso, plotou-se o gráfico do sinal coletado tratado, visto na figura 24.

Figura 23 – Código para identificação da função de perturbação do sistema

```
% Função de transferência da pertubação

t = (0:tSample:(1071)*tSample).';
u = cat(2, zeros(1,100),255.*(ones(1,972))).';
y = tempMediamovel(1100:2171).';
y = y -y(1);
dT = 0.4000;

%Identificação do sistema

data = iddata(y,u,dT);
np = 1;
nz = 0;

ft_ident = tfest(data,np,nz, 17.6);

y_sim = lsim(ft_ident,u,t);

% Resultados
figure()
plot(t,y,'b',t,y_sim,'r',t,u, 'g');
legend('Dados Originais','Modelo')
ylim([-40,10])
```

Fonte: *Autoria Própria*, 2024

Figura 24 – Gráfico do sinal de perturbação tratado



Fonte: *Autoria Própria*, 2024

Por fim, descobriu-se a função de transferência da perturbação dada pela equação (3).

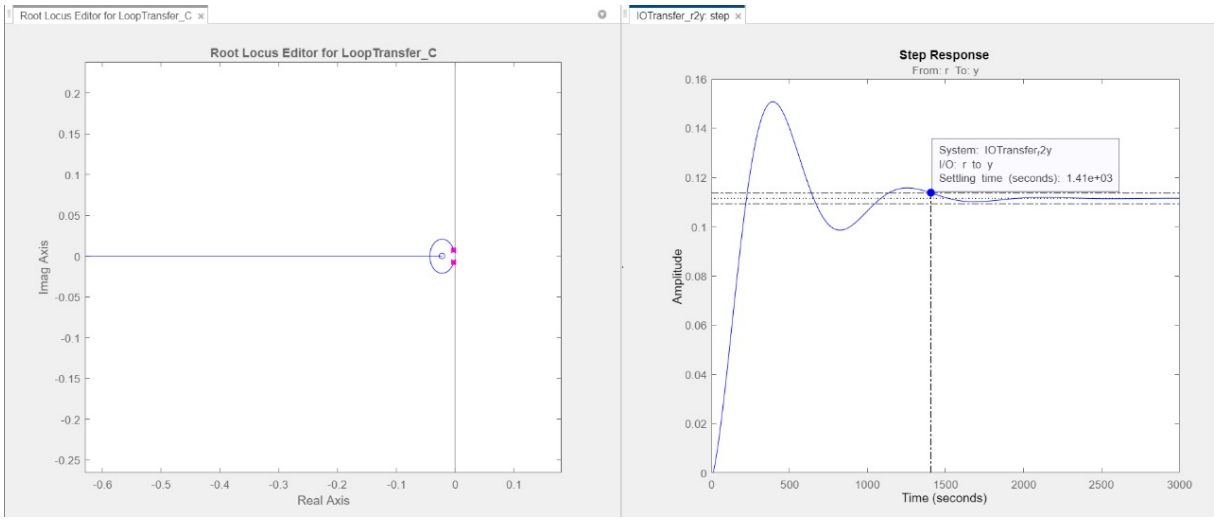
$$G_{TF}(s) = e^{-18s}\frac{-0.0023813}{(s+0.01741)} \tag{3}$$

4.2 Projeto do Controlador e Implementação Digital

Com as funções de transferência encontradas, o próximo passo foi projetar o controlador buscando remover o *overshoot* e fazer a malha aberta e a malha fechada terem o mesmo tempo de acomodação.

Primeiramente, utilizou-se a ferramenta *sisotool* do MATLAB para analisar o lugar das raízes (LGR) da planta da temperatura (sem aproximação de Padé) e acompanhar a resposta à entrada degrau simultaneamente, como mostra a figura 25.

Figura 25 – Interface *sisotool* da planta da temperatura

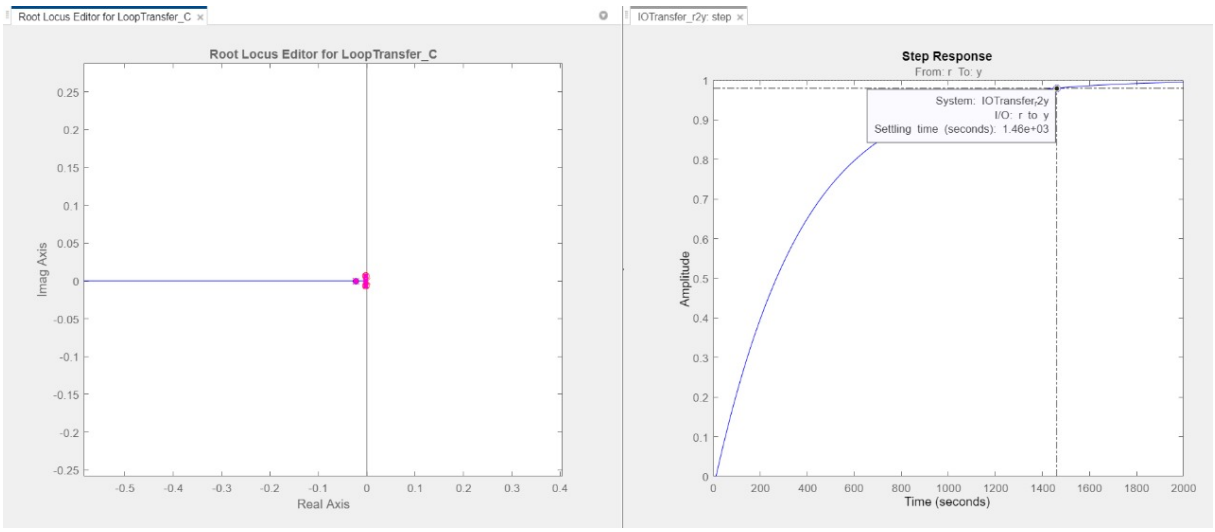


Fonte: *Autoria Própria*, 2024

Ao analisar a figura 25, considerou projetar um controlador proporcional integral derivativo (PID) para a temperatura, o qual possui um integrador, dois zeros e um filtro derivativo (passa-baixa). Para isso, foi colocado um polo na origem, um par de zeros conjugados em -0.0025 ± 0.0069i, que se anulam com o denominador da planta e, por fim um polo em - 0.0222. Assim, é possível visualizar na equação 4 o controlador encontrado, já com o ganho ajustado e na figura 26 o LGR da planta compensada.

$$C_T(s) = 8.6\frac{s^2 + 0.00492 * s + 5.32e - 05}{s * (s + 0.0222)} \tag{4}$$

Figura 26 – Interface *sisotool* da planta da temperatura controlada



Fonte: *Autoria Própria*, 2024

Entretanto esse controlador foi encontrado em tempo contínuo, mas sua implementação no microcontrolador deve ser em tempo discreto. Logo, precisou discretizá-lo pelo método de *Tustin* através do comando *c2d(Ct, TS, 'tustin')* no MATLAB com o tempo de amostragem (TS) de 1 segundo, como é observado na figura 27.

Figura 27 – Comando utilizado para discretização do C_t

```
>> Cd_t = c2d(C_t, 1, 'tustin');
>> Cd_t

Cd_t =

  8.527 z^2 - 17.01 z + 8.485
  ---------------------------
    z^2 - 1.978 z + 0.978
```

Fonte: *Autoria Própria*, 2024

Nota-se que se comparar o *Cd_t* obtido com a equação 5 é possível encontrar todas as variáveis necessárias para escrever a equação discretizada do controlador da temperatura.

$$PID(z) = \frac{u(z)}{e(z)} = \frac{b_0 z^2 + b_1 z + b_2}{z^2 + a_1 z + a_2} \tag{5}$$

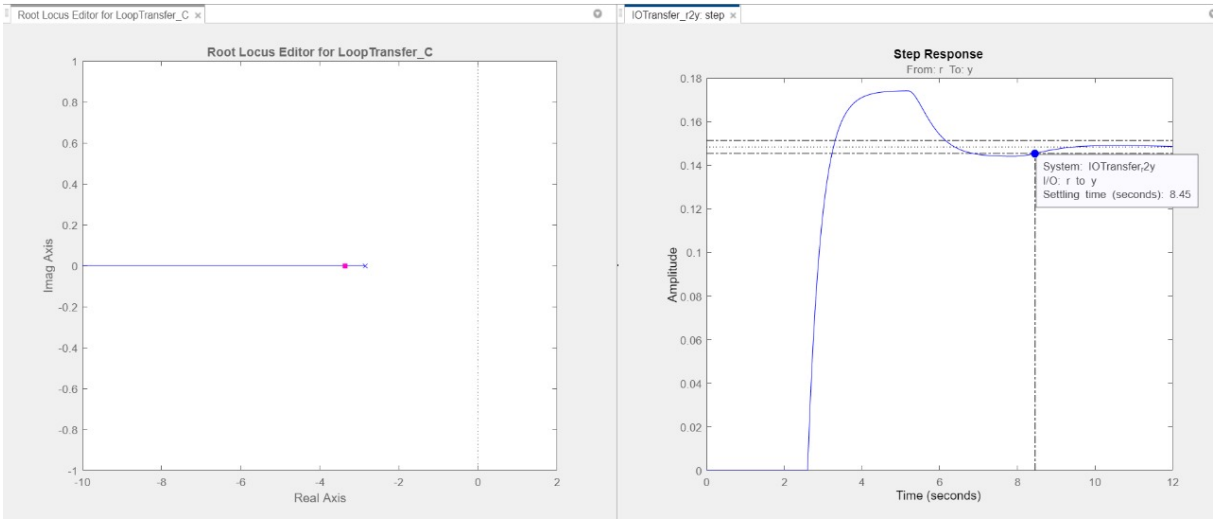
Como é visível, b_0 é 8.527, b_1 é -17.01, b_2 é 8.485, a_1 é -1.978 e a_2 é 0.978. Esses valores serão utilizados na implementação do controlador no código, além disso com eles é possível escrever a equação de diferenças (equação 7)

$$u(k) = -a_1 * u(k-1) - a_2 * u(k-2) + b_0 * e(k) + b_1 * e(k-1) + b_2 * e(k-2) \tag{6}$$

$$u(k) = 1.978 * u(k-1) - 0.978 * u(k-2) + 8.527 * e(k) - 17.01 * e(k-1) + 8.485 * e(k-2) \tag{7}$$

Todo o processo de projetar o controlador e fazer a implementação digital foi repetido para o sistema do fluxo. Então, primeiro plotou o LGR da planta do fluxo (sem aproximação de Padé) juntamente com a resposta ao degrau utilizando o *sisotool* e o resultado está ilustrado na figura 28.

Figura 28 – Interface *sisotool* da planta do fluxo

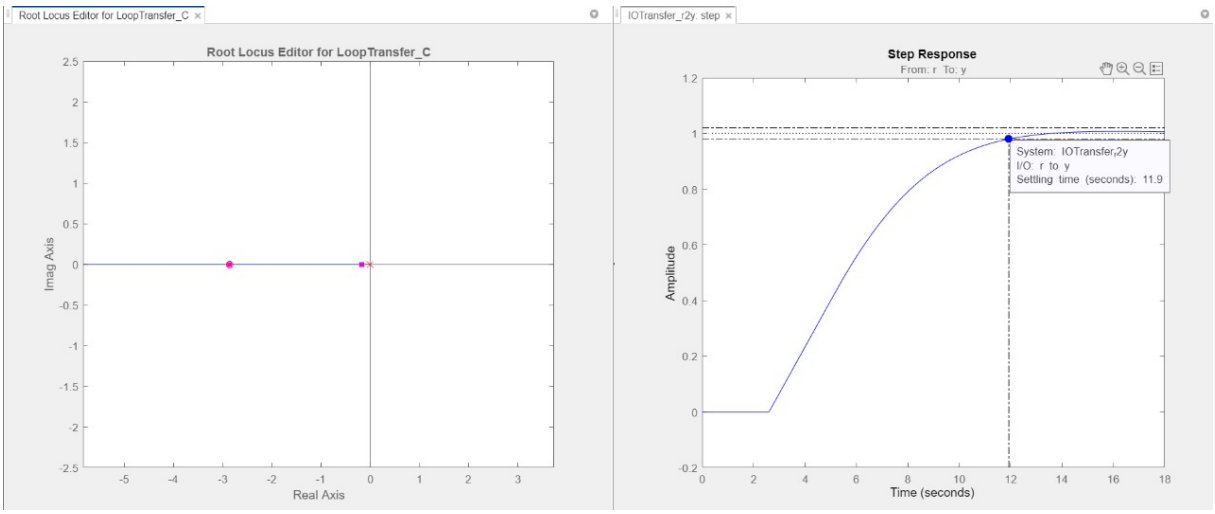


Fonte: *Autoria Própria*, 2024

Diferente do projeto anterior, nesse foi projetado um compensador proporcional integral (PI), cuja estrutura é um integrador e um zero. Assim, foi inserido um polo na origem, um zero em -2.86 e o ganho foi ajustado. A Equação 8 descreve o controlador encontrado e a figura 29 mostra como ficou o LGR da planta compensada.

$$C_F(s) = 0.33461 \frac{s + 2.86}{s} \tag{8}$$

Figura 29 – Interface *sisotool* da planta do fluxo controlada



Fonte: *Autoria Própria*, 2024

Novamente, o controlador estava tempo contínuo e precisou passar pela discretização de *Tustin*, como mostra a figura 30.

Figura 30 – Comando utilizado para discretização do C_f

```
>> Cd_f = c2d(C_f, 1, 'tustin');
>> Cd_f

Cd_f =

  0.8131 z + 0.1439
  -----------------
        z - 1
```

Fonte: *Autoria Própria*, 2024

A fim de encontrar as variáveis e a equação de diferenças, comparou-se *Cd_f* com a equação 9.

$$PI(z) = \frac{u(z)}{e(z)} = \frac{b_0 z + b_1}{z - 1} \tag{9}$$

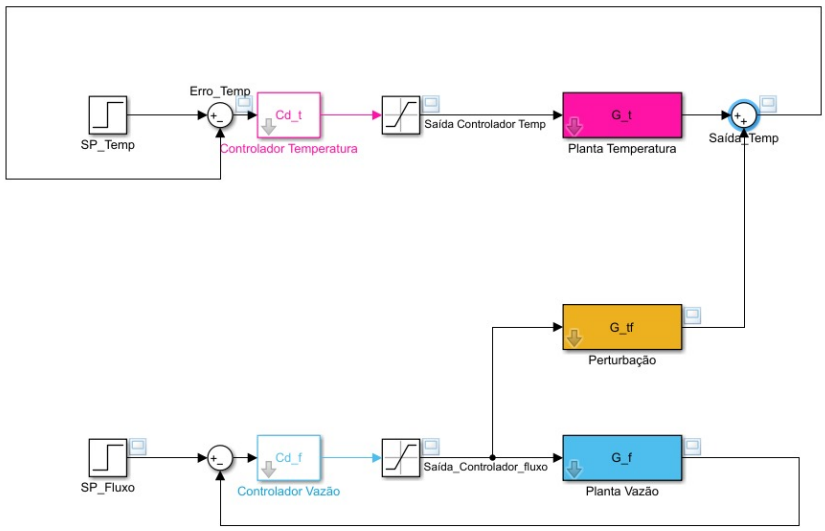
Nota-se que b_0 é 0.8131 e b_1 é 0.143, sendo assim a equação de diferenças pode ser escrita pela equação 11 e essas constantes serão utilizadas na implementação do PI no código.

$$u(k) = u(k-1) + b_0 * e(k) + b_1 * e(k-1) \tag{10}$$

$$u(k) = u(k-1) + 0.8131 * e(k) + 0.143 * e(k-1) \tag{11}$$

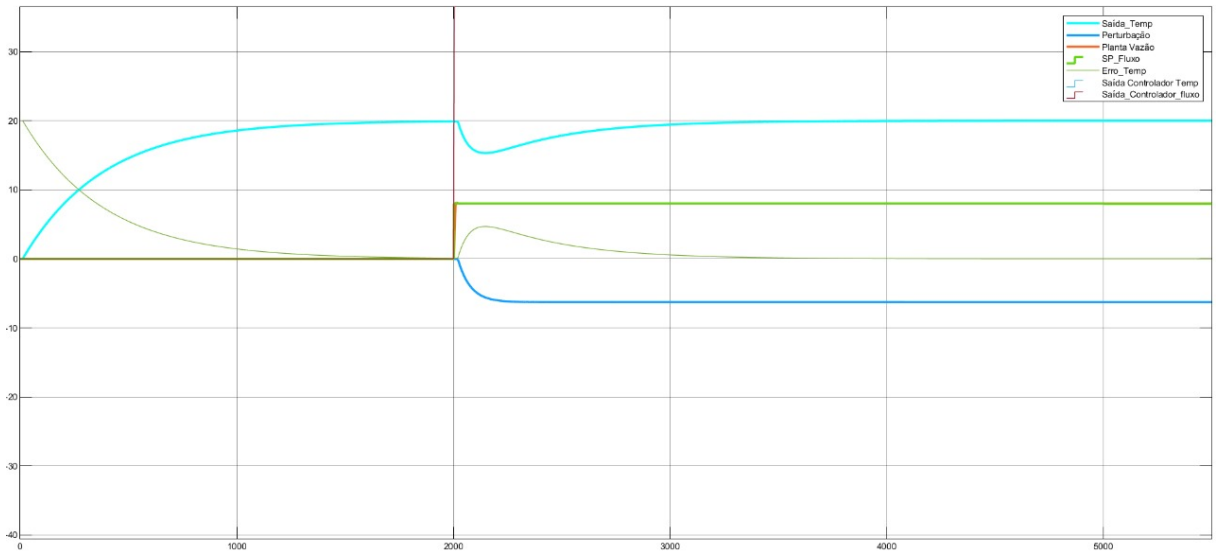
Antes de aplicar os compensadores no projeto real, montou-se um diagrama de blocos baseado na figura 1 pela ferramenta *simulink* do MATLAB para verificar a efetividade dos controladores, as figuras 31 e 32 ilustram esse processo.

Figura 31 – Diagrama de blocos do projeto final



Fonte: *Autoria Própria*, 2024

Figura 32 – Simulação do projeto final



Fonte: *Autoria Própria*, 2024

Essa simulação permite observar a atuação dos controladores em tempo real. Nota-se que no instante que ocorre a perturbação, o erro da temperatura aumenta ligeiramente e depois volta a ser nulo e a temperatura

reduz levemente e retorna ao *setpoint* logo em seguida, o que sugere que os controladores estão funcionando de acordo com o esperado, tentando estabilizar o sistema.

4.3 Programação do Sistema em Malha Fechada no Microcontrolador

Após projetar os controladores para o nosso sistema, foi necessário modificar o código desenvolvido e descrito na Seção 3.3 para um que realize o controle das duas variáveis (temperatura e fluxo). Primeiramente, foi necessário adicionar novas variáveis para tornar possível o controle. A Figura 33 ilustra as definições das variáveis básicas do novo código.

Figura 33 – Definição das variáveis básicas do código final

```
10  // Variáveis
11  const int resolution = 4095;                                    // Resolução do conversor AD do ESP
12  volatile int pulsos = 0;                                        // Pulsos do sensor de fluxo
13  unsigned long tempoAnt_temp = 0, tempoAtual_temp = 0, dT_temp = 0;          // Variáveis para manipulação do timer
14  unsigned long tempoAnt_fluxo = 0, tempoAtual_fluxo = 0, dT_fluxo = 0;       // Variáveis para manipulação do timer
15  float temperatura = 0, resistencia = 0, vazao = 0;              // Variáveis para cálculo da temperatura e vazão
16  int dutycycleTemp = 0, dutycycleCooler = 0;                     // Duty Cicle do PWM do transistor e do cooler
```

Fonte: *Autoria Própria*, 2024

Em comparação a Figura 5, a única adição se encontra nas linhas 13 e 14, com a modificação das variáveis referentes ao *timer* que, no código de malha aberta era apenas um *timer*, agora são dois, sendo um voltado para a planta de temperatura e outro voltado para a planta de fluxo. Outra adição a esta parte inicial do código são as variáveis de controle, que podem ser vistas na Figura 34.

Figura 34 – Definição das variáveis de controle do código final

```
25  //Variáveis Controlador
26
27  int cont_temp;
28  int cont_fluxo;
29
30  int setpoint_temp[1] = {40};
31  int setpoint_fluxo[1]= {0};
32
33  int ref_temp = 0;
34  int ref_fluxo = 0;
35
36  int i = 0;
37  int j = 0;
38
39  float d_temp = 0.0;
40  float d_fluxo = 0.0;
41
42  int pwm_max = 255;
43  int pwm_min = 0;
44
45  float u_temp = 0.0, ek_temp = 0.0, uk_1_temp = 0.0, ek_1_temp = 0.0, uk_2_temp = 0.0, ek_2_temp = 0.0;
46  float u_fluxo = 0.0, ek_fluxo = 0.0, uk_1_fluxo = 0.0, ek_1_fluxo = 0.0;
47
48  int sampleTime_temp = 1000;
49  int sampleTime_fluxo = 400;
50
51  float temp1 = 25.0, temp2 = 25.0, temp3 = 25.0, mediatemp = 25.0;
52  float fluxo1 = 0.0, fluxo2 = 0.0, fluxo3 = 0.0, mediafluxo = 0.0;
53
54  float controle_temp(float r_temp, float y_temp);
55  float controle_fluxo(float r_fluxo, float y_fluxo);
```

Fonte: *Autoria Própria*, 2024

As variáveis *cont_temp* e *cont_fluxo* são contadores utilizados para a mudança de *setpoint* de temperatura e fluxo, respectivamente. Por sua vez, as variáveis *setpoint_temp* e *setpoint_fluxo* são vetores do tipo inteiro, que armazena, em ordem, os *setpoints* de temperatura e fluxo, respectivamente, desejados ao longo do tempo. O tamanho destes vetores pode variar dependendo do ensaio que se deseja fazer, no caso descrito na Figura 34, estas variáveis estão definidas conforme a descrição do Ensaio I, descrito na Seção 4.4.

As variáveis *ref_temp ref_fluxo* correspondem as referências, ou seja, os *setpoints* das minhas plantas, definidos pela posição i e j (definidas nas linhas 36 e 37 na Figura 34) dos vetores *setpoint_temp* e *setpoint_fluxo*. Já as variáveis *d_temp* e *d_fluxo* são os valores de *duty cycle* definidos pela saída do controlador. As variáveis *pwm_max* e *pwm_min* são variáveis limitadoras para limitar o valor de saída do controlador para a escala de *pwm*.

As variáveis descritas nas linhas 45 e 46 são referentes a equação de diferenças da implementação digital dos controladores projetados, o prefixo e e u significam, respectivamente, erro e a saída do controlador. Na Figura 34, por fim, temos as declarações das funções *controle_temp* e *controle_fluxo*, responsáveis pela ação do controlador de temperatura e de fluxo, respectivamente. Ambas as funções recebem como parâmetros os valores de saída das plantas e de referência, declarados como *y_temp, r_temp* e *y_fluxo, r_fluxo*.

As Figuras 35 e 36 ilustram as declarações das constantes de controle, encontradas e contextualizadas na Seção 4.2 e as funções de implementação dos dois controladores, respectivamente.

Figura 35 – Definição das constantes dos controladores

```
57  //------------------------------------------------------------
58  // ---------- Constantes Controlador -------------------------
59  float a1_temp = -1.978;
60  float a2_temp = 0.978;
61  float b0_temp = 8.527;
62  float b1_temp = -17.01;
63  float b2_temp = 8.485;
64
65  float b0_fluxo = 0.526;
66  float b1_fluxo = -0.1432;
67  // -----------------------------------------------------------
68  //------------------------------------------------------------
```

Fonte: *Autoria Própria*, 2024

Figura 36 – Função de Implementação do dos Controladores

```
70  //Implementação do Controlador PID Digital
71  float controle_temp(float r_temp, float y_temp){
72
73    ek_temp = r_temp-y_temp;
74    u_temp = ((-1)*a1_temp*uk_1_temp) - a2_temp*uk_2_temp + b0_temp*ek_temp + b1_temp*ek_1_temp + b2_temp*ek_2_temp;
75
76    if (u_temp>255) u_temp=255;     // limitador superior
77    else if (u_temp<0) u_temp=0;    // limitador inferior
78
79    ek_2_temp = ek_1_temp;
80    uk_2_temp = uk_1_temp;
81    ek_1_temp = ek_temp;
82    uk_1_temp = u_temp;
83
84
85    return float(u_temp);
86  }
87
88  //Implementação do Controlador PI Digital
89  float controle_fluxo(float r_fluxo, float y_fluxo){
90
91    ek_fluxo = r_fluxo-y_fluxo;
92    u_fluxo = uk_1_fluxo + b0_fluxo*ek_fluxo + b1_fluxo*ek_1_fluxo;
93
94    if (u_fluxo>255) u_fluxo=255;     // limitador superior
95    else if (u_fluxo<0) u_fluxo=0;    // limitador inferior
96
97    ek_1_fluxo = ek_fluxo;
98    uk_1_fluxo = u_fluxo;
99
100   return float(u_fluxo);
101 }
```

Fonte: *Autoria Própria*, 2024

Como pode ser observado na Figura 36, ambas funções de controladores possuem estruturas similares, com suas diferenças localizadas nas equações de diferenças (linhas 74 e 92 da Figura 36), por se tratarem de controladores diferentes, sendo um PID com filtro derivativo no primeiro e um PI no segundo. Por causa dessa diferença, por característica, duas variáveis a mais precisam ser atualizadas no controlador PID (linhas 79 a 82) em comparação ao controlador PI (linhas 97 e 98). Como saída, ambas retornar uma variável u que se refere a saída do controlador, na escala de *duty cycle* de 0 a 255.

A Figura 37 representa a função de leitura do sensor de fluxo, a função *void setup()* e o início da função *void loop()*.

Figura 37 – Função de leitura do sensor de fluxo, a função *void setup()* e o início da função *void loop()*

```
103  // Rotina da interrupção que realiza leitura do sensor de fluxo
104  void lerFluxo(){
105    pulsos++;
106  }
107
108  void setup() {
109    Serial.begin(115200);
110    pinMode(SENSOR_TEMP, INPUT);
111    pinMode(SENSOR_FLUXO, INPUT);
112    pinMode(PWM_TEMP, OUTPUT);
113    pinMode(PWM_COOLER, OUTPUT);
114
115    ledcAttach(PWM_TEMP, 1000, 8);      // Configurando PWMs
116    ledcAttach(PWM_COOLER, 1000, 8);
117
118    attachInterrupt(digitalPinToInterrupt(SENSOR_FLUXO), lerFluxo, RISING); // Interrupção para sensor de fluxo
119
120
121  }
122
123  void loop() {
124    tempoAtual_temp = millis();
125    dT_temp = tempoAtual_temp - tempoAnt_temp;
126    tempoAtual_fluxo = millis();
127    dT_fluxo = tempoAtual_fluxo - tempoAnt_fluxo;
```

Fonte: *Autoria Própria*, 2024

Em comparação as Figuras 6 e 7, as únicas diferenças com a Figura 37 se dão na questão do envio do sinal de PWM, agora realizados na função *void loop()* e a utilização de um timer para o tratamento de cada planta do sistema (linhas 124 a 127 da Figura 37). A continuação da função *void loop()* exibe-se na Figura 38, onde é possível observar o *timer* da planta de temperatura.

Figura 38 – *Timer* da planta de temperatura

```
129  // TIMER TEMPERATURA -- 1 segundo
130  if (dT_temp >= sampleTime_temp) {
131    tempoAnt_temp = tempoAtual_temp;
132
133    // Cálculo da vazão em L/s
134    // vazao =  double((pulsos) / (120));
135
136    // Leitura do NTC e cálculo da temperatura diretamente no loop
137    int leituraADC = analogRead(SENSOR_TEMP);
138
139    // Cálculo da temperatura com a equação Steinhart-Hart
140    double Vout = leituraADC * Vs / resolution;                // Converte a leitura ADC para tensão
141    double Rt = R1 * Vout / (Vs - Vout);                       // Calcula a resistência do NTC
142    temperatura = 1 / (1 / To + log(Rt / Ro) / Beta);          // Equação de Steinhart-Hart
143    temperatura = temperatura - 273.15;                        // Converte de Kelvin para Celsius
144
145    temp1 = temp2 ;
146    temp2 = temp3 ;
147    temp3 = temperatura ;
148    mediatemp = ( temp1 + temp2 + temp3 ) /3.0;
149
150    // Sistema só começa após 2 s
151    if(tempoAtual_temp>=2000){
152      ref_temp = setpoint_temp[i];
153      cont_temp++;
154    }
155    else{
156      ref_temp= 0;
157    }
158
159    d_temp = controle_temp(ref_temp, mediatemp);
160    dutycycleTemp = int(d_temp);
161
162    ledcWrite(PWM_TEMP, dutycycleTemp);  // Mandando sinal PWM
163
164    Serial.print (tempoAtual_temp);
165    Serial.print (",");
166    Serial.print (mediatemp);
167    Serial.print (",");
168    Serial.print (mediafluxo);
169    Serial.print (",");
170    Serial.print (dutycycleTemp);
171    Serial.print (",");
172    Serial.print (dutycycleCooler);
173    Serial.print (",");
174    Serial.println (cont_temp);
```

Fonte: *Autoria Própria*, 2024

De adições, temos, na Figura 38, a realização da média dos 3 últimos valores lidos de temperatura, realizado para filtrar ruído de altas frequências na leitura do sensor (linhas 145 a 148), além disso, temos uma variável condicional *if* que permite a atribuição do *setpoint* a variável de referência apenas após 2 segundos da inicialização do sistema. Após atender esta condição, é enviado um sinal de PWM de valor determinado a partir da saída da função do controlador de temperatura. Além disso, os valores das variáveis de tempo, temperatura, fluxo, *duty cycle* de temperatura e fluxo, além do contador para mudança do *setpoint* de temperatura. Esta ação é realizada dentro desta iteração para os valores serem imprimidos a cada 1 segundo sem a necessidade de criar uma iteração extra. Quanto ao *timer* de fluxo, este pode ser

observado na Figura 39.

Figura 39 – *Timer* da planta de fluxo

```
178   //Timer Fluxo: 400ms
179   if (dT_fluxo >= sampleTime_fluxo) {
180     tempoAnt_fluxo = tempoAtual_fluxo;
181
182     vazao = pulsos;
183     pulsos = 0;
184
185     fluxo1 = fluxo2 ;
186     fluxo2 = fluxo3 ;
187     fluxo3 = vazao ;
188     mediafluxo = ( fluxo1 + fluxo2 + fluxo3 ) /3.0;
189
190     // Sistema só começa após 2 s
191     if(tempoAtual_fluxo>=2000){
192       ref_fluxo = setpoint_fluxo[j];
193       cont_fluxo++;
194     }
195     else{
196       ref_fluxo= 0;
197     }
198
199     d_fluxo = controle_fluxo(ref_fluxo, mediafluxo);
200     dutycycleCooler = int(d_fluxo);
201
202     ledcWrite(PWM_COOLER, dutycycleCooler);
203   }
204
205   //if ((cont_fluxo>=3750)&&(j<1))
206   //{
207   //  j++;
208   //  cont_fluxo = 0;
209  // }
210
211   //if ((cont_temp>=2000)&&(i<2))
212   //{
213   //  i++;
214   //  cont_temp = 0;
215   //}
216 }
```

Fonte: *Autoria Própria*, 2024

Este *timer* possui mesma estrutura básica do *timer* de temperatura, modificando apenas as versão das variáveis e funções para esta planta. Ao final desta iteração há, comentados, duas condicionais para a mudança de *setpoint* para cada uma das variáveis, esta condição é verdadeira caso o contador de mudança de *setpoint cont_temp cont_fluxo* atinga um valor específico, calculado por uma proporção entre o tempo desejado para cada *setpoint* e o tempo de amostragem para cada planta. Ao acessar esta iteração, a variável de posição do vetor de *setpoints* é incrementada e o contador é zerado.

4.4 Encerramento da 2ª Fase: Análise do Controle da Temperatura e Fluxo

Após a a implementação digital dos controladores projetados no *firmware* desenvolvido, foi-se necessário a execução de distintos ensaios para análise e observação do funcionamento dos sistemas de controle. No total, foram realizados 3 ensaios com as seguintes características:

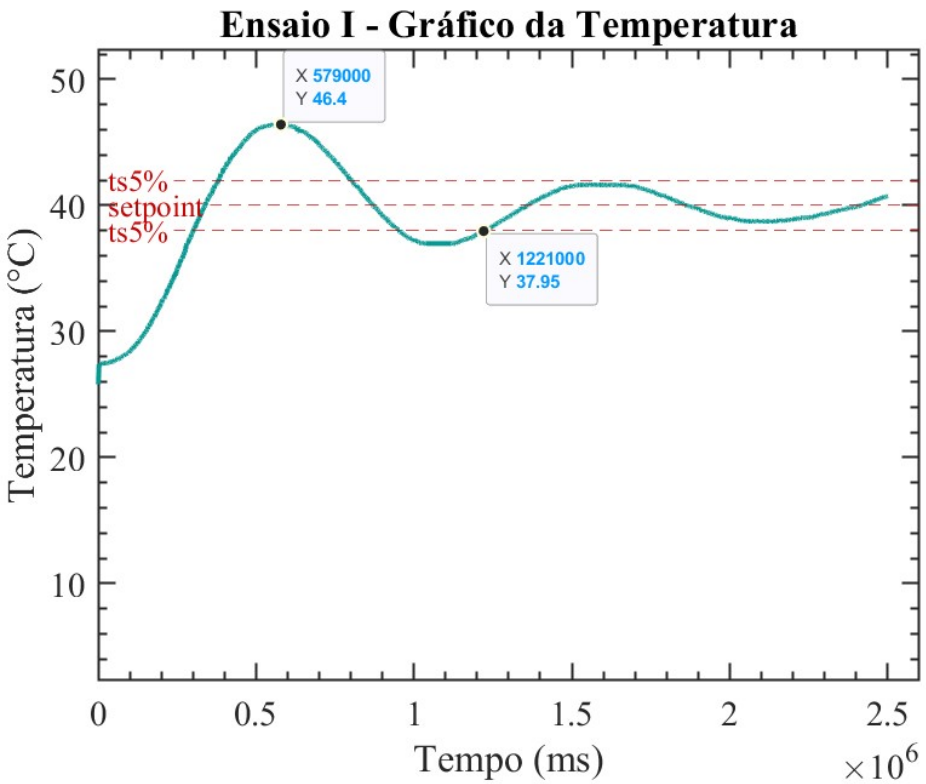
- **Ensaio I**- 1 *setpoint* de temperatura em 40ºC e 1 *setpoint* de vazão em 0 pulsos.
- **Ensaio II**- 3 *setpoints* de temperatura na seguinte ordem: 30ºC que, após 2000 segundos, muda-se para 50ºC que, após mais 2000 segundos, muda-se para 40ºC. Além de 1 *setpoint* de vazão em 0 pulsos.
- **Ensaio III** - 1 *setpoint* de temperatura em 35ºC e 2 *setpoints* de vazão em 0 pulsos que, após 1800 segundos, muda-se para 9 pulsos.

Para a análise destes dados, foi utilizado um código em *Python* já utilizado nas atividades experimentais da disciplina para leitura serial e armazenamento automático dos dados em um arquivo .csv. Além disso, foi utilizado o *software MATLAB* para o processamento dos sinais e construção dos gráficos.

4.4.1 Ensaio I

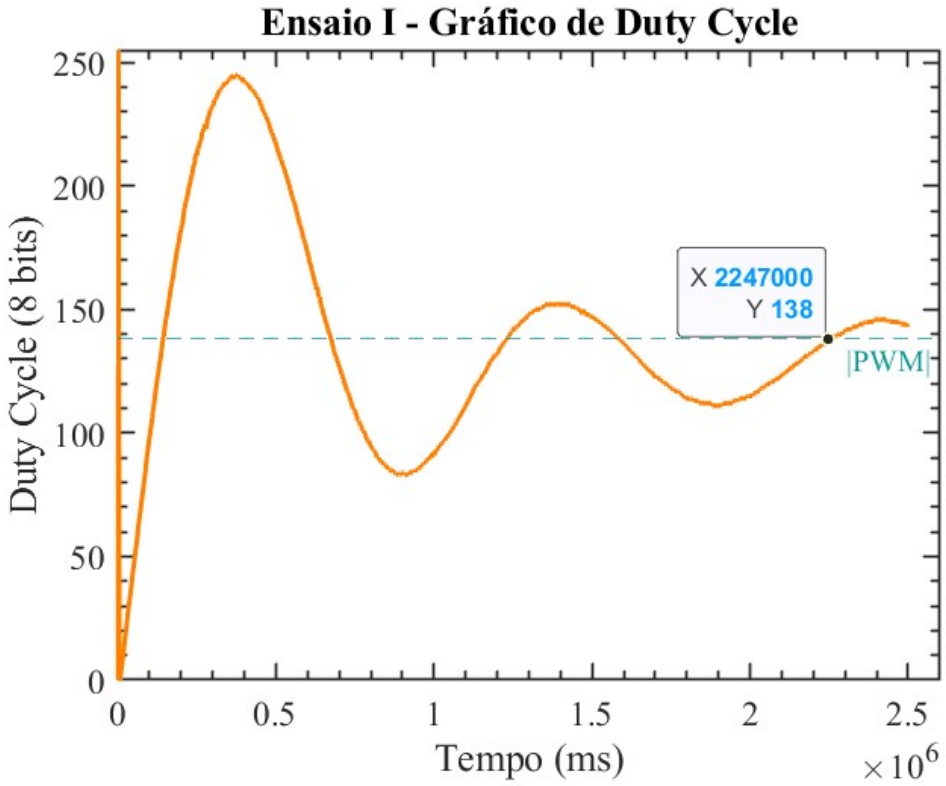
As Figuras 40 e 41 se referem a, respectivamente, o gráfico de temperatura e de *duty cyle* no tempo do Ensaio I

Figura 40 – Gráfico de Temperatura do Ensaio I



Fonte: *Autoria Própria*, 2024

Figura 41 – Gráfico de *Duty Cycle* do Ensaio I



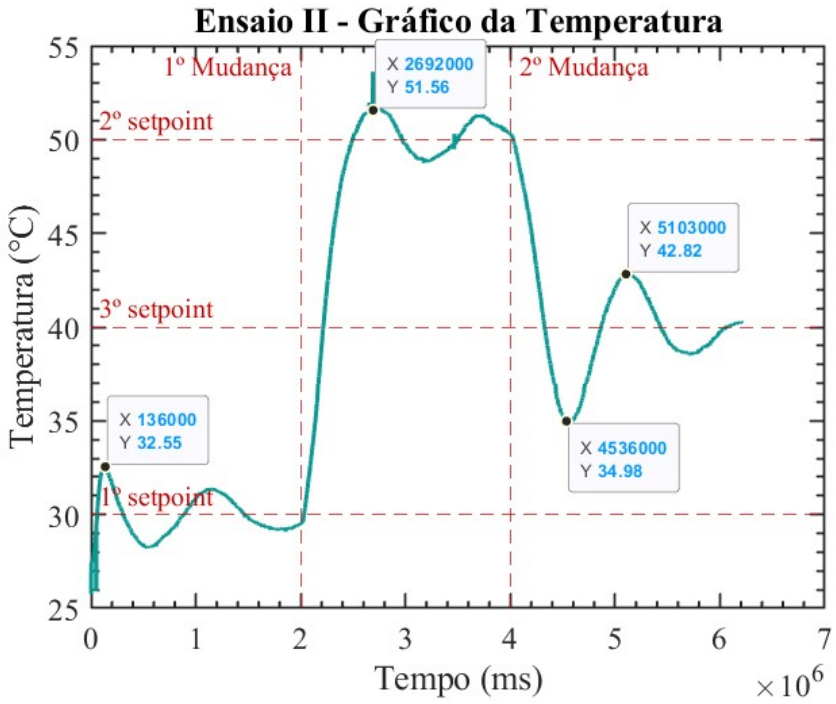
Fonte: *Autoria Própria*, 2024

Na figura 40, é possível observar que a resposta possui resposta subamortecida, com *overshoot* de cera de 16%, além disso, seu tempo de acomodação é de aproximadamente 1221 segundos (aproximadamente, 20,35 minutos), estabilizando sobre o valor de *setpoint* estabelecido. Ao analisar a Figura 41, podemos ver o valor de PWM ser estabilizada por volta de um valor médio de 138.

4.4.2 Ensaio II

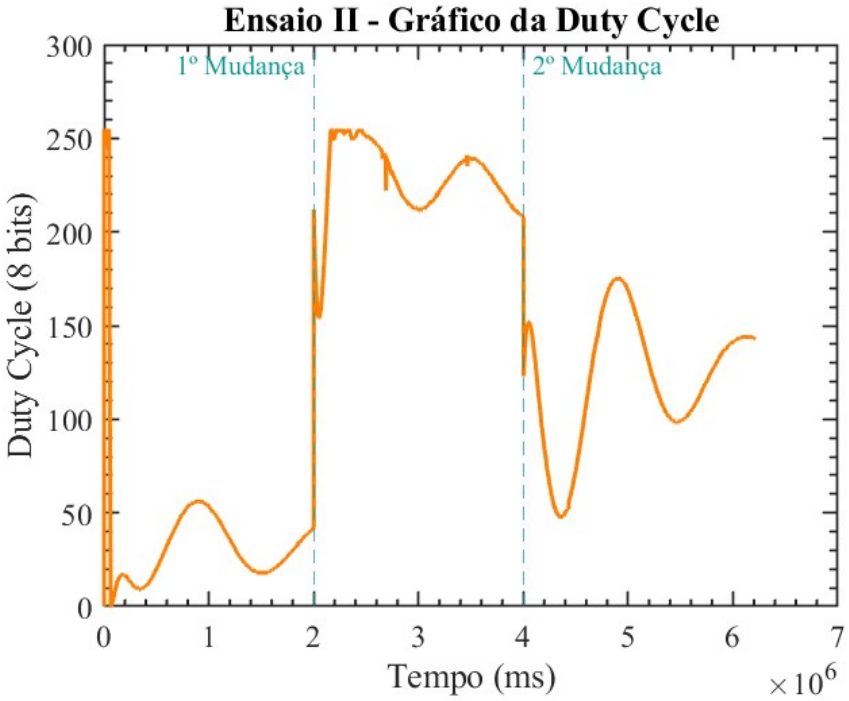
As Figuras 42 e 43 se referem a, respectivamente, o gráfico de temperatura e de *duty cyle* no tempo do Ensaio II

Figura 42 – Gráfico de Temperatura do Ensaio II



Fonte: *Autoria Própria*, 2024

Figura 43 – Gráfico de *Duty Cycle* do Ensaio II



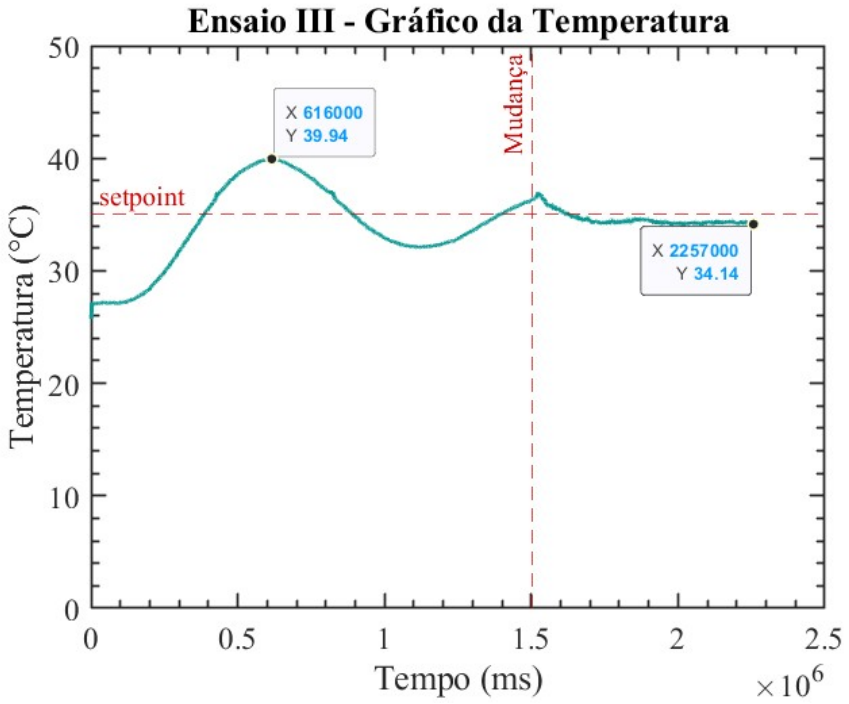
Fonte: *Autoria Própria*, 2024

Ao analisar a Figura 42, é possível observar o amortecimento e estabilização da temperatura nos setpoints descritos e sua porção transitória após a mudança do *setpoint* nos instantes de tempo com as linhas tracejadas com legenda de mudança. Para o primeiro *setpoint*, houve *overshoot* de cerca de 8,5%, para o segundo *setpoint*, houve *overshoot* de cerca de 3,12% e para o terceiro *setpoint*, houve *undershoot* de cerca de 12,55%. Já sobre a Figura 43, é possível observar a variação de *duty cycle* ao longo do tempo - Abrupta, nos instantes transitórios de mudança de referência e amortecida no regime permanente.

4.4.3 Ensaio III

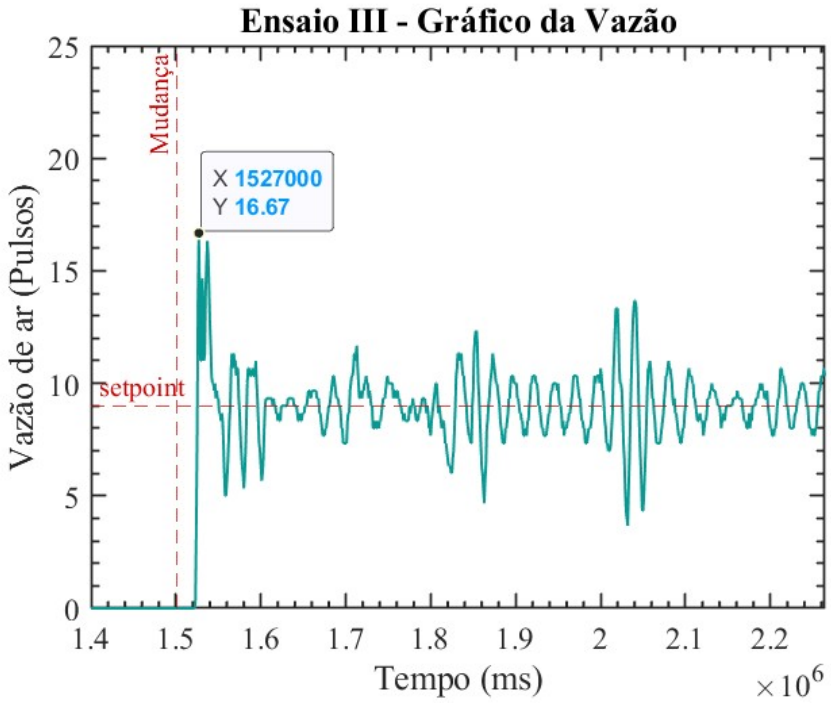
As Figuras 44, 45, 46 e 47 se referem a, respectivamente, o gráfico de temperatura, do fluxo, de *duty cyle* do sistema de temperatura e *duty cycle* do sistema de vazão tempo do Ensaio III

Figura 44 – Gráfico de Temperatura do Ensaio III



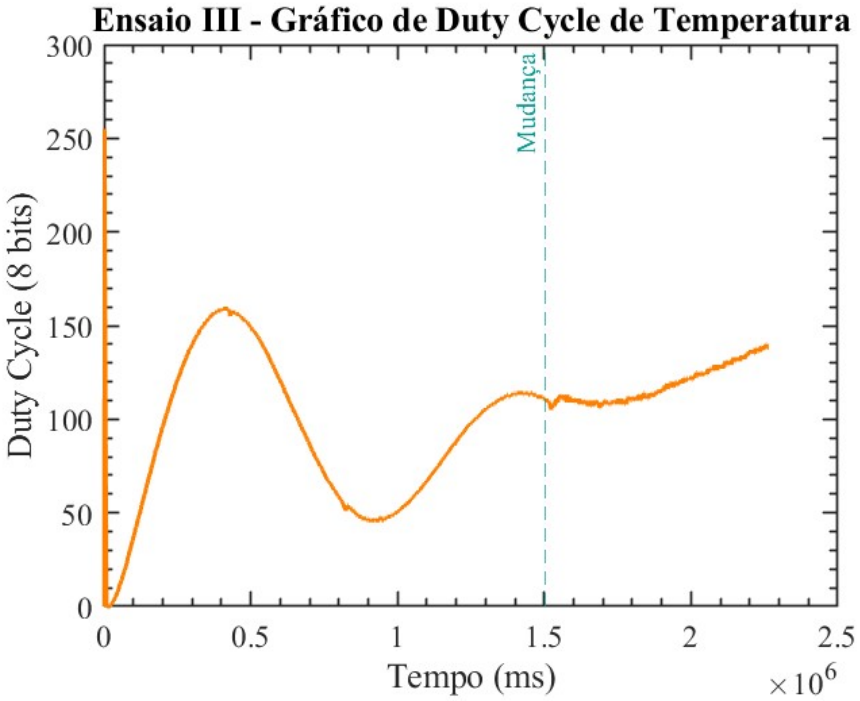
Fonte: *Autoria Própria*, 2024

Figura 45 – Gráfico de Vazão do Ensaio II



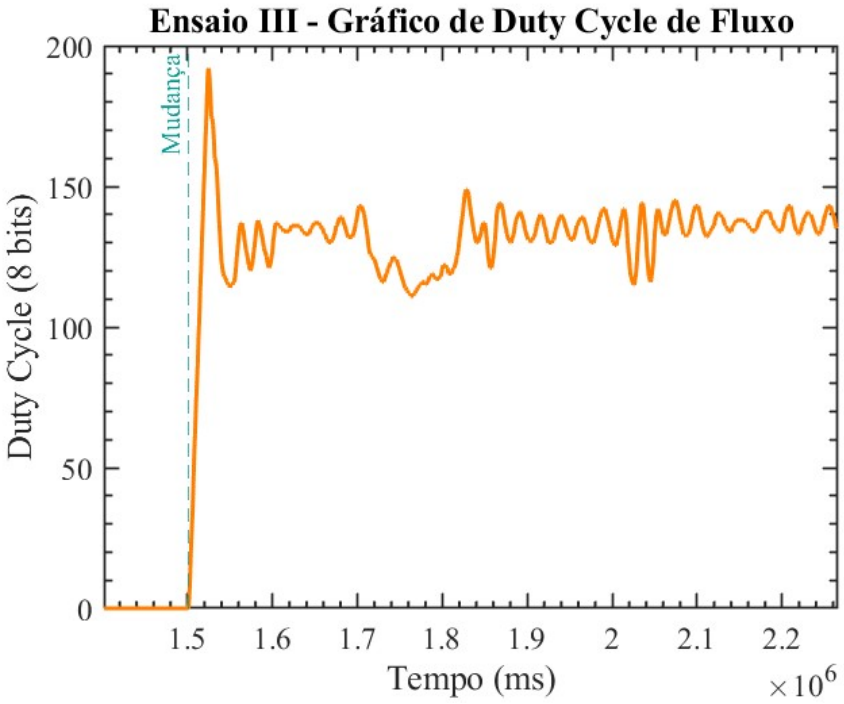
Fonte: *Autoria Própria*, 2024

Figura 46 – Gráfico de *Duty Cycle* do sistema de Temperatura do Ensaio III



Fonte: *Autoria Própria*, 2024

Figura 47 – Gráfico de *Duty Cycle* do sistema de Vazão do Ensaio III



Fonte: *Autoria Própria*, 2024

Ao analisar a Figura 44, podemos ver o mesmo padrão subamortecido do sistema previamente observado nos ensaios anteriores, apresentando *overshoot* de aproximadamente 14%. Após o *setpoint* de fluxo após 25 minutos de ensaio, a perturbação agravou o amortecimento do sinal sob o *setpoint*, estabilizando a temperatura sob a referência estabelecida. Por sua vez, quando a Figura 46 é analisada, podemos observar o valor de *duty cycle* que, no intervalo do ensaio prévio a 25 minutos, começava a estabilizar-se em um valor, começar a aumentar para compensar a perturbação no sistema e manter a temperatura desejada.

Ao analisar a Figura 45, podemos observar o valor de fluxo como algo altamente oscilatório sob o regime permanente, apesar de estável sob um valor médio de pulsos estabelecido (nesse caso, do *setpoint* solicitado de 9 pulsos de vazão de ar). Este comportamento já era esperado, conhecendo a resposta de sua planta realizada na identificação e ilustrada na Figura 20. Por fim, ao analisar a Figura 47, podemos ver um comportamento similar em seu formato, todavia, com variação em regime permanente menor de que o valor de pulsos.

5 Conclusão

Para o desenvolvimento desse sistema de controle de temperatura e fluxo foi necessário separá-lo em diversas etapas. Primeiramente, foi esquematizado circuitos eletrônicos com alimentação, sensores e processadores, responsáveis por gerarem as plantas posteriormente estudadas. Além disso, foi necessário construir uma caixa bem vedada e com formato que favorecia a realização do experimento. Após testar os circuitos na *protoboard*, montou-se a PCB e toda a estrutura física do projeto.

A segunda fase consistia em identificar as funções de transferência da temperatura, do fluxo e da pertubação e a partir dessas projetar compensadores para as plantas da temperatura e do fluxo. Porém, esses controladores encontrados estavam em tempo contínuo, então passaram pelo método de discretização de *Tustin* para serem implementados no código da ESP32. Por fim, foram realizados ensaios para verificar se o controle estava sendo efetivo na realidade.

Em resumo, esse projeto foi importante para revisar e aplicar o conteúdo teórico e experimentos realizados nas aulas práticas da disciplina de Sistema de Controle Realimento. Além disso, o desenvolvimento deste desafiou os alunos no processo de modelagem de um novo sistema, o qual apresenta duas variáveis que se influenciavam através da pertubação.

6 Referências

1. NISE, N. S. Engenharia de Sistemas de Controle. [s.l: s.n.].
2. DORF, R. C.; BISHOP, R. H. Sistemas de Controle Modernos. [s.l: s.n.].