



LABORATÓRIO 3 - Simulação de Sistemas com Threads

1. Objetivos

Este laboratório tem como objetivo continuar o sistema de controle baseado no movimento de um robô móvel, porém implementando o conceito de múltiplas threads. Além disso, também será necessário importar as bibliotecas criadas no Laboratório 1 para realizar os cálculos necessários e gerar gráficos das funções resultantes por meio do programa de plotagem Gnuplot.

2. Problema

A simulação do laboratório passado foi feita a partir de um robô móvel, que tem seu acionamento diferencial descrito pelo seguinte modelo de espaço de estados:

$$\dot{x}(t) = \begin{bmatrix} \sin(x_3) & 0 \\ \cos(x_3) & 0 \\ 0 & 1 \end{bmatrix} u(t) \quad y(t) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} x(t)$$

Também foi feito um programa na linguagem C que simulou a resposta desse sistema para a entrada $u(t)$ descrita abaixo:

$$u(t) = \begin{cases} 0 & , \text{ para } t < 0 \\ \begin{bmatrix} 1 \\ 0.2\pi \end{bmatrix} & , \text{ para } 0 \leq t < 10 \\ \begin{bmatrix} 1 \\ -0.2\pi \end{bmatrix} & , \text{ para } t \geq 10 \end{cases}$$

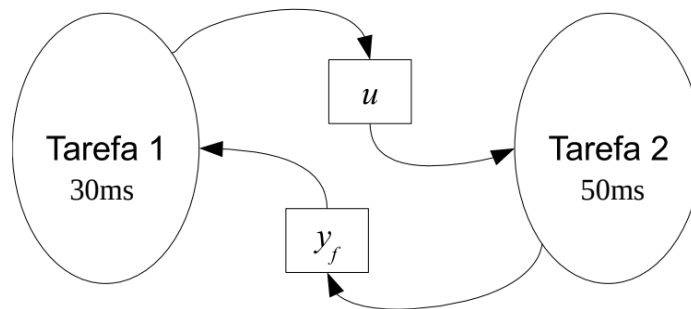
Agora é preciso dividir a simulação em duas tarefas:

- A primeira fará a simulação em si;
- E a segunda fará a geração de $u(t)$ e a amostragem de $y_f(t)$, dada por:

$$y_f(t) = x(t) + \begin{bmatrix} 0.5 \times D \times \cos(x_3) & 0 & 0 \\ 0 & 0.5 \times D \times \sin(x_3) & 0 \\ 0 & 0 & 1 \end{bmatrix} x(t)$$

A partir disso, será feito um programa na linguagem C que simula a resposta desse sistema utilizando múltiplas tarefas (threads). Na imagem a seguir temos um diagrama que representa o programa com a implementação de threads:

Figura 1. Diagrama que representa o fluxo do programa.



Fonte: CAVALCANTI, André.

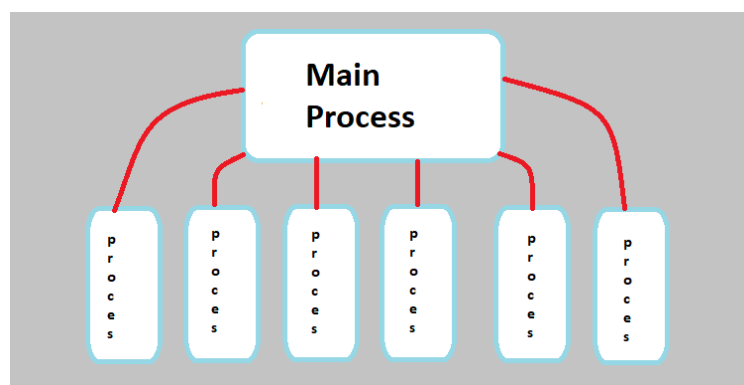
3. Introdução teórica

Antes de começar a explicar a implementação do programa, primeiro é necessário comentar sobre como funciona um sistema com múltiplas threads.

- **Threads**

Uma thread, ou tarefa, é um fluxo de sequência única dentro de um processo. Elas são uma forma de melhorar uma aplicação por meio do paralelismo. Por exemplo, em um navegador web, cada guia aberta pode ser uma thread diferente.

Figura 2. Representação de um sistema multi-threaded.



Fonte: TUFAN, Burak Hamdi.

Apesar de possuir algumas características semelhantes a um processo, uma tarefa acaba operando de forma mais rápida pelos seguintes fatores:

- A criação de threads é muito mais rápida;
- A alternância de contexto entre threads é mais rápida;
- As threads podem ser encerradas facilmente;
- E a comunicação entre elas é mais rápida.

Para escrever um programa multithreading na linguagem C, é preciso utilizar a biblioteca POSIX Threads (ou Pthreads) e, para a implementação do pthread, foi utilizado o compilador gcc.

4. Estrutura de diretórios utilizada

Antes de ter uma visão sobre os arquivos fontes gerados, é possível discutir sobre a estrutura dos diretórios. Na imagem abaixo pode-se observar todos os 12 arquivos fontes e 3 pastas:

Figura 3. Estrutura dos diretórios utilizada.

Nome	Data de modificação	Tipo	Tamanho
.git	22/11/2024 14:31	Pasta de arquivos	
.vscode	15/01/2024 22:47	Pasta de arquivos	
Documentos	16/02/2024 22:20	Pasta de arquivos	
calculo	11/11/2024 15:08	Arquivo C	3 KB
calculo	10/11/2024 11:48	Arquivo Fonte C ...	1 KB
dados_grafico_yft	23/11/2024 16:54	Documento de Te...	1 KB
grafico_yft	23/11/2024 16:54	Arquivo PNG	40 KB
main	23/11/2024 16:48	Arquivo C	4 KB
Makefile	09/11/2024 20:32	Arquivo	1 KB
matrix	09/11/2024 20:10	Arquivo C	10 KB
matrix	09/11/2024 20:05	Arquivo Fonte C ...	2 KB
readme	23/11/2024 16:38	Arquivo Fonte Ma...	2 KB
saida	23/11/2024 16:54	Documento de Te...	2 KB
simulacao	23/11/2024 16:48	Arquivo C	6 KB
simulacao	23/11/2024 16:48	Arquivo Fonte C ...	1 KB

Fonte: Explorador de Arquivos do Windows 11.

- *.git*: pasta criada para armazenar todas as versões do código, que podem ser acessadas quando necessárias;
- *.vscode*: pasta criada pelo Visual Studio Code e que contém algumas configurações do projeto em questão, podendo ser definições de depuração, tarefas e/ou configurações específicas do ambiente;
- *Documentos*: pasta que contém este relatório, o arquivo oferecido pelo professor e o arquivo .ggb, no qual os gráficos dos resultados estão.

Além disso, todos esses arquivos estão na pasta *Laboratório 3*, e o mesmo será entregue para avaliação na forma de arquivo “.zip”.

5. Arquivos fontes

Foi utilizada a biblioteca `Matrix`, desenvolvida no Laboratório 1, e a partir dela foram realizados os cálculos necessários para este trabalho. Esses cálculos foram feitos em um arquivo separado chamado `calculo.c`, no qual os algoritmos dos cálculos de $u(t)$, $x(t)$, $y(t)$ e $yf(t)$ foram desenvolvidos. Além disso, foi feito um arquivo chamado `simulacao.c` que realiza algumas das operações ao simular o programa. Também foi feito um `Makefile` para executar todos os arquivos em uma única operação. A seguir, será mostrado os arquivos presentes no programa e também as modificações feitas em relação aos códigos feitos em trabalhos anteriores:

5.1. `matrix.c`

Esse código, já realizado nos laboratórios passados, define uma Abstract Data Type (ADT) para representar matrizes em linguagem C. Essa ADT, denominada `Matrix`, encapsula os dados e as operações como criar, acessar, modificar e realizar operações matemáticas com matrizes, como soma, subtração, multiplicação, transposição e cálculo de determinantes e inversa. Em comparação aos outros trabalhos, não houveram mudanças em relação a estrutura do código.

5.2. `matrix.h`

O arquivo `matrix.h` continua servindo como interface para essa ADT, declarando as funções que podem ser utilizadas para manipular matrizes. Também não houveram mudanças significativas na estrutura deste código.

5.3. `calculo.c`

Esse código implementa numericamente um sistema dinâmico, calculando as matrizes $u(t)$, $x(t)$, $y(t)$ e $yf(t)$ em função do tempo t . O sistema parece ser modelado por equações diferenciais, onde a matriz $u(t)$ atua como uma entrada e as matrizes $x(t)$ e $y(t)$ representam as saídas do sistema. Entre as mudanças feitas neste código estão:

- **integrar(valor, a, b, n)**: Essa função calcula a integral de uma matriz (antes feita pelo arquivo *integral.c*);
- **calcula_ut(t)**: Essa função irá calcular o valor de $u(t)$, porém irá acessar os elementos da matriz utilizando a notação `ut->dados[i][j]`, e isso torna o código um pouco mais eficiente, pois evita chamadas para a função `set_matriz`;
- **calcula_xt(t)**: Essa função irá calcular o valor de $x(t)$ e, assim como a função `calcula_u(t)`, irá acessar os elementos da matriz utilizando a notação `xt->dados[i][j]`, evitando chamadas desnecessárias;
- **calcula_yt(t)**: Essa função irá calcular o valor de $y(t)$ e, ao invés de usar como parâmetros t e $u(t)$, a função irá utilizar somente $*x(t)$, facilitando assim a compreensão do código;
- **calcula_yft(t)**: Essa função irá calcular o valor de $yf(t)$ dada as matrizes $x(t)$ e $u(t)$ e o diâmetro do robô:
 1. Primeiramente é criada uma matriz identidade auxiliar `m_aux` de dimensões 3x3 e a mesma modifica seus elementos com base no valor de $u(t)$;
 2. Depois é realizado o produto entre as matrizes `m_aux` e $x(t)$;
 3. É somado o resultado da multiplicação com a matriz $x(t)$;
 4. E por fim a memória alocada para `m_aux` é liberada e é retornado o valor de $yf(t)$.

5.4. calculo.h

Esse código continua definindo a interface `calculo.h`, que serve como um contrato para as funções que implementam as operações de cálculo do sistema de controle. Essa interface estabelece a comunicação entre o código principal e as funções responsáveis por calcular as matrizes $u(t)$, $x(t)$, $y(t)$ e $yf(t)$.

5.5. simulacao.c

Esse código simula o comportamento do sistema de controle do robô móvel ao longo do tempo. Ele utiliza as operações matriciais encontradas em `matrix.c` para modelar a dinâmica do sistema e gerar dados que podem ser utilizados para análise e visualização. Entre as funções presentes no código, temos:

- **`printa_transposta(*m)`**: Essa função recebe como parâmetros um ponteiro para uma matriz `m` e tem o objetivo de exibir na tela a transposta de `m`:
 1. Primeiramente, inicia-se a impressão da matriz `m` com um colchete "[";
 2. Depois dois laços são utilizados para percorrer todos os elementos da matriz transposta;
 3. Então cada elemento será imprimido com três casas decimais ("%3f");
 4. E, ao final, será finalizada a impressão com um colchete "]"
- **`qsort_double(*a, *b)`**: Essa função é utilizada para ordenar um array de números de ponto flutuante (tipo `double`) em ordem crescente:
 1. Ela converte os ponteiros `void*` para ponteiros `double*` para acessar os valores;
 2. Calcula a diferença entre os dois valores;
 3. E retorna um valor que indica a ordem relativa com base no sinal de diferença.
- **`simula(*ut, *xt, *yt, *yft, diametro, t)`**: Essa será uma das principais funções do programa, pois as funções do `calcula.c` serão chamadas e os cálculos das matrizes `u(t)`, `x(t)`, `y(t)` e `yf(t)` serão realizadas:
 1. A função irá receber como parâmetros as matrizes `u(t)`, `x(t)`, `y(t)` e `yf(t)`, o diâmetro do robô e o tempo total da simulação;
 2. Depois será criado um loop e, a cada iteração, as funções de cálculo de cada matriz serão chamadas para atualizar as respectivas matrizes;

3. Os resultados da simulação serão impressos na tela em formato tabular, mostrando o valor de $u(t)$ e de $yf(t)$ em cada instante de t ;
4. Por fim, a função `usleep` introduz uma pausa entre cada iteração, permitindo a visualização gradual dos resultados.

- **`salva_resultados(*ut, *xt, *yt, *yft, t, *nome_arquivo)`**: Essa função irá gravar os resultados da simulação em um arquivo de texto, permitindo uma melhor análise dos dados gerados:

1. A função tentará abrir um arquivo no modo de append ("`a`"), ou seja, os novos dados serão adicionados ao final do arquivo, evitando conflito com resultados de simulações anteriores;
2. Se o arquivo for aberto com sucesso, a função escreve os valores das matrizes $u(t)$, $x(t)$, $y(t)$ e $yf(t)$ em uma linha do arquivo, separados por tabulações;
3. Por fim, a função fecha o arquivo, garantindo que as informações sejam salvas corretamente.

- **`gera_grafico(*nome_arquivo)`**: Essa função irá gravar os resultados da simulação em um arquivo de texto, permitindo uma melhor análise dos dados gerados:

1. A função tentará abrir um arquivo no modo de append ("`a`"), ou seja, os novos dados serão adicionados ao final do arquivo, evitando conflito com resultados de simulações anteriores;
2. Se o arquivo for aberto com sucesso, a função escreve os valores das matrizes $u(t)$, $x(t)$, $y(t)$ e $yf(t)$ em uma linha do arquivo, separados por tabulações;
3. Por fim, a função fecha o arquivo, garantindo que as informações sejam salvas corretamente.

- **`leitura_dados(*nome_arquivo, *uT_00, *uT_01, *yfT_00, *yfT_01, *yfT_02, tam)`**: Essa função lê os dados da simulação a partir de um arquivo de texto e armazena-os em um conjunto de variáveis:

1. Primeiramente o arquivo será aberto no modo de leitura ("`r`");

2. Cada linha do arquivo será lida utilizando a função `sscanf` para extrair os valores de `t`, `u(t)` e `yf(t)` e armazená-los nas variáveis passadas por referência;
 3. Por fim, o arquivo será fechado após a leitura.
- **`processa_dados(*dados, tam)`**: Essa função calcula e retorna um conjunto de estatísticas (média, variância, desvio padrão e valores máximos e mínimos), dada uma matriz de dados numéricos e seu tamanho:
 1. A função recebe como entrada um ponteiro para um array de double e seu tamanho;
 2. Depois disso, será utilizada a função `qsort` para ordenar os dados em ordem crescente, facilitando assim o cálculo dos quartis e dos valores mínimo e máximo;
 3. Soma todos os valores e divide pelo número total de elementos, obtendo assim a média;
 4. Depois, é calculada a soma dos quadrados das diferenças entre cada valor e a média, dividindo pelo número total de elementos e obtendo assim a variância;
 5. Calcula-se a raiz quadrada da variância, obtendo o desvio padrão;
 6. Identifica os valores que dividem os dados ordenados em quatro partes iguais, obtendo assim os quartis;
 7. E calcula-se os valores mínimo e máximo, obtidos diretamente dos primeiros e últimos elementos do array ordenado;
 8. Por último, a função retorna uma estrutura `Dados` contendo todas as estatísticas calculadas.
 - **`salva_tabela_dados(tam, *nome_arquivo)`**: Essa função tem o objetivo de coletar o conjunto de estatísticas calculados anteriormente e montar uma tabela de $T(k)$ e $J(k)$, para o sistema sem carga e com carga, onde $T(k)$ é o período e $J(k)$ é o jitter do período. Infelizmente, não foi possível completar essa função e a tabela não foi montada, pois a forma como obteve-se os resultados da simulação não foi adequada para fazer esse tipo de análise.

5.6. simulacao.h

Esse código define a interface `simulacao.h`, que tem a funcionalidade de fornecer funções para a realização da simulação do robô móvel, visualizar os resultados e salvá-los em um arquivo de texto.

5.7. main.c

Esse é o código que será executado na simulação, implementando as funções de todos os arquivos já citados e utilizando múltiplas threads para otimizar o desempenho. Ele divide o processo em duas tarefas principais: simulação e cálculo, cada uma sendo executada em uma thread separada. Entre as principais estruturas, funcionalidades e modificações feitas estão:

- **ThreadArgs:** Essa estrutura encapsula todos os argumentos necessários para as funções das threads, incluindo as matrizes $u(t)$, $x(t)$, $y(t)$ e $yf(t)$, parâmetros como `diametro` e `t`, e o ponteiro `*nome_arquivo` para funções de salvamento;
- **thread_simula(arg):** Função utilizada para simular o sistema em uma thread separada. Isso permite a execução paralela de múltiplas simulações, potencialmente aumentando a performance do sistema:
 1. A função irá receber um ponteiro `arg` que é convertido para um ponteiro `ThreadArgs`, essa estrutura contém os parâmetros necessários para a simulação, incluindo as matrizes $u(t)$, $x(t)$, $y(t)$ e $yf(t)$, o tempo `t` e o diâmetro do robô;
 2. Depois os parâmetros serão extraídos da estrutura `ThreadArgs` para serem utilizados na simulação;
 3. A função `simula` é chamada para realizar a simulação;
 4. E por fim, a função `pthread_exit` é chamada para terminar a execução da thread.
- **thread_calcula(arg):** Essa função é bem parecida com a anterior, porém será utilizada para calcular as matrizes do sistema em uma outra thread, ajudando na performance:
 1. A função irá receber um ponteiro `arg` que é convertido para um ponteiro `ThreadArgs`, essa estrutura contém os parâmetros

- necessários para a simulação, incluindo as matrizes $u(t)$, $x(t)$, $y(t)$ e $yf(t)$, o tempo t e o diâmetro do robô;
2. Depois os parâmetros serão extraídos da estrutura `ThreadArgs` para serem utilizados na simulação;
 3. Dentro de um loop, os valores das matrizes serão calculados;
 4. Ao final de cada loop, os resultados das matrizes serão salvos dentro de um arquivo, que será passado pelo usuário;
 5. Após todos os cálculos, a função `pthread_exit` será chamada para terminar a execução da thread.
- **main():** Essa é a função principal do programa, onde serão executados todos os códigos listados anteriormente:
 1. Após inicializar o programa, será apresentado ao usuário um menu onde ele poderá informar o diâmetro do robô e o nome do arquivo onde os resultados da simulação serão salvos;
 2. Depois disso, serão criadas as matrizes $u(t)$, $x(t)$, $y(t)$ e $yf(t)$ para armazenar os resultados dos cálculos;
 3. Nisso, duas threads serão criadas. A primeira delas, chamada `thread1`, ficará responsável pela simulação em si, enquanto a outra, chamada `thread2`, ficará responsável somente por realizar os cálculos das matrizes;
 4. As threads serão sincronizadas utilizando `pthread_join` para garantir que todas as tarefas sejam concluídas antes da finalização do programa;
 5. Depois de finalizar a simulação, a função `gera_grafico` será chamada para plotar o gráfico de $yf(t)$, que pode ser visto no arquivo `grafico_yft.png`;
 6. Ao final, algumas mensagens de confirmação serão exibidas para o usuário e as matrizes utilizadas serão libertas da memória pela função `libera_matriz`.

5.8. Makefile

O Makefile define as regras para compilar e criar um executável a partir dos arquivos de código fonte (.c) e dos arquivos de cabeçalho (.h) do projeto em questão. Como visto anteriormente, o projeto envolve cálculos numéricos, com base nas dependências dos arquivos como `matrix.h`, `simulacao.h` e `calculo.h`.

5.9. **readme.md**

O README fornece instruções concisas sobre como compilar e executar o projeto. Ele guia o usuário através do processo de construção do programa, desde a abertura do terminal até a execução do programa e a verificação dos resultados.

5.10. **saida.txt**

É o arquivo de texto responsável por armazenar a saída gerada pelo código. Dentro do arquivo temos os valores de t , de $u(t)$ transposta e de $yf(t)$ também transposta, todos separados pelo caractere *tab*.

5.11. **dados_grafico_yft.txt**

É o arquivo de texto responsável por armazenar os dados de $yf(t)$ para a plotagem do gráfico 3D. Dentro do arquivo, como fizemos a simulação de 0 a 20 segundos, temos 21 linhas de valores da matriz $yf(t)$, com 3 dados cada.

5.12. **grafico_yft.png**

Essa imagem descreve o gráfico tridimensional da matriz $yf(t)$, que possui três eixos (Eixo X, Eixo Y e Eixo Z) contendo os dados resultantes da simulação do robô.

6. Resultados

Após ter o código pronto, é preciso executar o programa em um prompt de comando. Para isso, basta seguir os passos listados no [readme.md](#), também mostrados abaixo:

1. Abra um terminal ou prompt de comando;
2. Navegue até o diretório onde os arquivos do código estão localizados usando o comando `'cd'`;
3. Compile o código:
 - a. Usando o comando `'make'`;

- b. Ou caso o código já esteja compilado, use primeiro o comando `'make clean'` para depois chamar o comando `'make'`;
4. Depois que o código for compilado com sucesso, execute o programa digitando o comando `'./main'`;
5. Ao compilar o código, será exibido um menu pedindo um valor para o diâmetro do robô móvel. O usuário poderá então digitar um número que desejar;
6. O programa será executado e produzirá a saída (`'saida.txt'`) com base na simulação especificada.
7. O programa também produzirá um gráfico de $y_f(t)$ (`'grafico_yft.png'`) com base nos dados presentes no arquivo `'dados_grafico_yft.txt'`.

Abaixo temos o resultado da simulação do robô com um diâmetro de 2 cm:

Figura 4. Compilação do código *main.c*, utilizando o VS Code.

```

chagas@IdeaPad-Vinicius:/mnt/c/Users/Vinicius/Documents/10º Período (2024-2)/PTR - Programação em Tempo Real (2ª Tentativa)/Lab 3/Laboratório 3$ make
gcc -c main.c -o main.o
gcc -c calculo.c -o calculo.o
gcc -c matrix.c -o matrix.o
gcc -c simulacao.c -o simulacao.o
gcc main.o calculo.o matrix.o simulacao.o -lm -o main
chagas@IdeaPad-Vinicius:/mnt/c/Users/Vinicius/Documents/10º Período (2024-2)/PTR - Programação em Tempo Real (2ª Tentativa)/Lab 3/Laboratório 3$ ./main
----- Sistema de Robô Móvel -----
Informe o diâmetro do robô (em cm): 2
----- SIMULAÇÃO para Diametro = 2 -----
formato -> func(t): [matriz]
matrix.o      0.000017 0.000000 0.000000
matrix.o      0.000017 0.000000 0.000000
matrix.o      0.000000 0.587785 0.000000
matrix.o      0.000000 0.000000 1.000000
simulacao.o   0.000000 0.587785 0.000000
simulacao.o   0.000000 0.000000 1.000000
  
```

Fonte: Visual Studio Code.

Na Figura 4, podemos ver que o usuário abriu o terminal WSL e, já dentro do diretório do programa, executou o comando `'$ make'`. Logo depois, o usuário iniciou o programa digitando o comando `'$./main'`. Em seguida o programa pergunta ao usuário qual valor do diâmetro do robô ele quer realizar a simulação e, depois que esse número é informado, a simulação começa.

Figura 5. Arquivo gerado com os dados da simulação.

```
saida.txt
1  t = 0  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
2  t = 1  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
3  t = 2  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
4  t = 3  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
5  t = 4  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
6  t = 5  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
7  t = 6  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
8  t = 7  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
9  t = 8  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
10 t = 9  uT = [1.00, 0.63]  yfT = [-1.46, 0.93, 1.65]
11 t = 10 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
12 t = 11 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
13 t = 12 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
14 t = 13 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
15 t = 14 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
16 t = 15 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
17 t = 16 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
18 t = 17 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
19 t = 18 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
20 t = 19 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
21 t = 20 uT = [-0.63, 0.00] yfT = [-2.00, 0.00, -1.26]
```

Fonte: Visual Studio Code.

Figura 6. Arquivo gerado com os dados de $y_f(t)$ para a plotagem do gráfico.

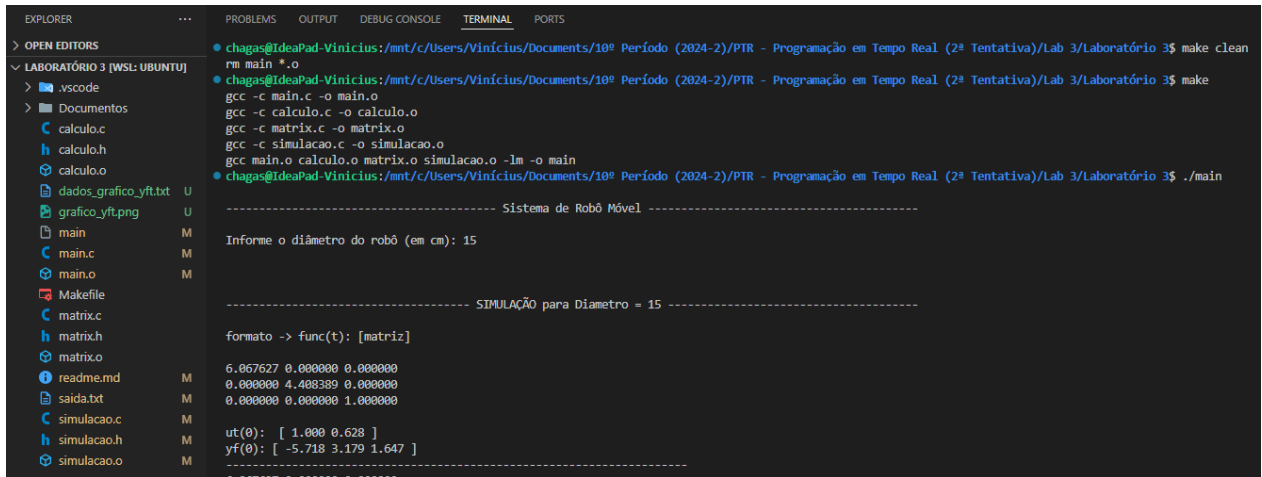
```
dados_grafico_yft.txt
1  -1.46 0.93 1.65
2  -1.46 0.93 1.65
3  -1.46 0.93 1.65
4  -1.46 0.93 1.65
5  -1.46 0.93 1.65
6  -1.46 0.93 1.65
7  -1.46 0.93 1.65
8  -1.46 0.93 1.65
9  -1.46 0.93 1.65
10 -1.46 0.93 1.65
11 -2.00 0.00 -1.26
12 -2.00 0.00 -1.26
13 -2.00 0.00 -1.26
14 -2.00 0.00 -1.26
15 -2.00 0.00 -1.26
16 -2.00 0.00 -1.26
17 -2.00 0.00 -1.26
18 -2.00 0.00 -1.26
19 -2.00 0.00 -1.26
20 -2.00 0.00 -1.26
21 -2.00 0.00 -1.26
```

Fonte: Visual Studio Code.

Nas Figuras 5 e 6, podemos ver os arquivos onde os resultados da simulação foram armazenados. Esses resultados mostram que a simulação seguiu o comportamento esperado nos intervalos de tempo adotados na entrada do sistema $u(t)$.

Para compararmos os resultados anteriores, temos agora o resultado da simulação do robô com um diâmetro de 15 cm:

Figura 7. Compilação do código *main.c*, utilizando o VS Code.

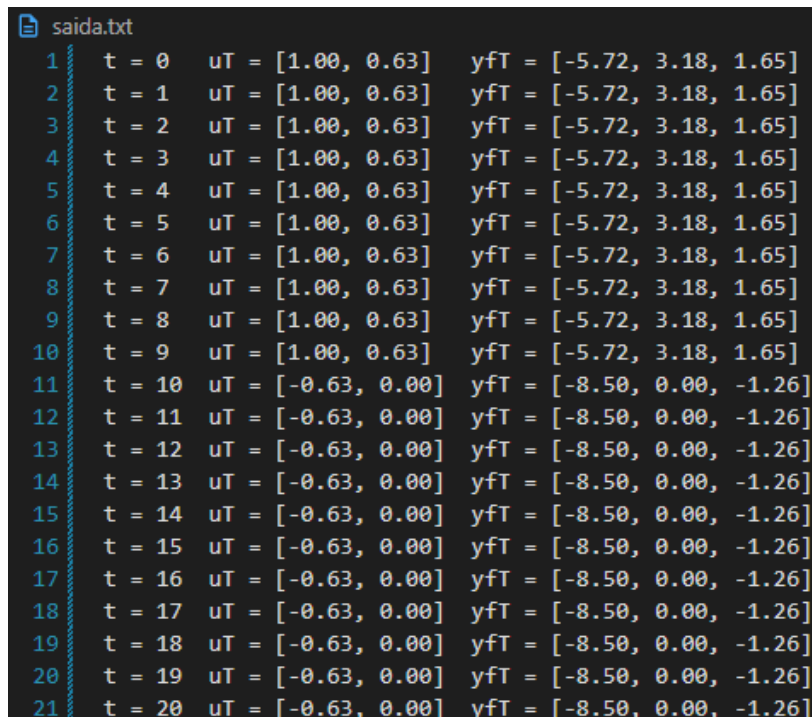


```
chagas@Ideapad-Vinicius:/mnt/c/Users/Vinicius/Documents/10º Período (2024-2)/PTR - Programação em Tempo Real (2ª Tentativa)/Lab 3/Laboratório 3$ make clean
rm main *.o
chagas@Ideapad-Vinicius:/mnt/c/Users/Vinicius/Documents/10º Período (2024-2)/PTR - Programação em Tempo Real (2ª Tentativa)/Lab 3/Laboratório 3$ make
gcc -c main.c -o main.o
gcc -c calculo.c -o calculo.o
gcc -c matrix.c -o matrix.o
gcc -c simulacao.c -o simulacao.o
gcc main.o calculo.o matrix.o simulacao.o -lm -o main
chagas@Ideapad-Vinicius:/mnt/c/Users/Vinicius/Documents/10º Período (2024-2)/PTR - Programação em Tempo Real (2ª Tentativa)/Lab 3/Laboratório 3$ ./main
----- Sistema de Robô Móvel -----
Informe o diâmetro do robô (em cm): 15
----- SIMULAÇÃO para Diâmetro = 15 -----
formato -> func(t): [matriz]
6.067627 0.000000 0.000000
0.000000 4.408389 0.000000
0.000000 0.000000 1.000000
ut(0): [ 1.000 0.628 ]
yf(0): [ -5.718 3.179 1.647 ]
-----
6.067627 0.000000 0.000000
```

Fonte: Visual Studio Code.

Na Figura 7, podemos ver que o usuário seguiu os mesmos passos listados no exemplo anterior (Figura 4), porém executou antes o comando '\$ *make clean*' para remover os arquivos objetos e demonstrar a utilidade do comando.

Figura 8. Arquivo gerado com os dados da simulação.



```
saida.txt
1 t = 0 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
2 t = 1 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
3 t = 2 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
4 t = 3 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
5 t = 4 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
6 t = 5 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
7 t = 6 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
8 t = 7 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
9 t = 8 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
10 t = 9 uT = [1.00, 0.63] yfT = [-5.72, 3.18, 1.65]
11 t = 10 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
12 t = 11 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
13 t = 12 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
14 t = 13 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
15 t = 14 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
16 t = 15 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
17 t = 16 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
18 t = 17 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
19 t = 18 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
20 t = 19 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
21 t = 20 uT = [-0.63, 0.00] yfT = [-8.50, 0.00, -1.26]
```

Fonte: Visual Studio Code.

Figura 9. Arquivo gerado com os dados de $y_f(t)$ para a plotagem do gráfico.

```
dados_grafico_yft.txt
1 -5.72 3.18 1.65
2 -5.72 3.18 1.65
3 -5.72 3.18 1.65
4 -5.72 3.18 1.65
5 -5.72 3.18 1.65
6 -5.72 3.18 1.65
7 -5.72 3.18 1.65
8 -5.72 3.18 1.65
9 -5.72 3.18 1.65
10 -5.72 3.18 1.65
11 -8.50 0.00 -1.26
12 -8.50 0.00 -1.26
13 -8.50 0.00 -1.26
14 -8.50 0.00 -1.26
15 -8.50 0.00 -1.26
16 -8.50 0.00 -1.26
17 -8.50 0.00 -1.26
18 -8.50 0.00 -1.26
19 -8.50 0.00 -1.26
20 -8.50 0.00 -1.26
21 -8.50 0.00 -1.26
```

Fonte: Visual Studio Code.

Ao comparar as Figuras 5 e 8, podemos notar que os resultados obtidos da entrada $u(t)$ não sofreram alterações com o decorrer da simulação, mas os valores de $y_f(t)$ sim. Para isso, precisamos olhar novamente para o cálculo da matriz e assim poder fazer uma análise mais profunda:

Figura 10. Cálculo da matriz $y_f(t)$.

$$y_f(t) = x(t) + \begin{bmatrix} 0.5 \times D \times \cos(x_3) & 0 & 0 \\ 0 & 0.5 \times D \times \sin(x_3) & 0 \\ 0 & 0 & 1 \end{bmatrix} x(t)$$

Fonte: CAVALCANTI, André.

- **para $0 \leq t < 10$:**

Diâmetro	Coluna 1	Coluna 2	Coluna 3
2 cm	-1.46	0.93	1.65
15 cm	-5.72	3.18	1.65

- Os valores obtidos nas Colunas 1 e 2 foram diretamente proporcionais aos valores do diâmetro escolhidos, o que, de acordo com a multiplicação feita no cálculo da matriz, faz sentido;

- Já os valores obtidos na Coluna 3 foram iguais, isso porque a multiplicação é feita por 1 e o cálculo não depende do valor do diâmetro inserido.

- **para $t \geq 10$:**

Diâmetro	Coluna 1	Coluna 2	Coluna 3
2 cm	-2.00	0.00	-1.26
15 cm	-8.50	0.00	-1.26

- Na Coluna 1, para valores de $t \geq 10$, podemos ver que os valores continuam diretamente proporcionais;
- Agora para os valores da Coluna 2 a situação mudou, isso porque o valor de $\sin\left(\frac{-\pi}{5}\right) = 0$;
- Por fim, os valores obtidos na Coluna 3 também mostraram o mesmo comportamento visto anteriormente, ou seja, a multiplicação ainda é feita por 1 e o cálculo não depende do valor do diâmetro.

7. Gráficos

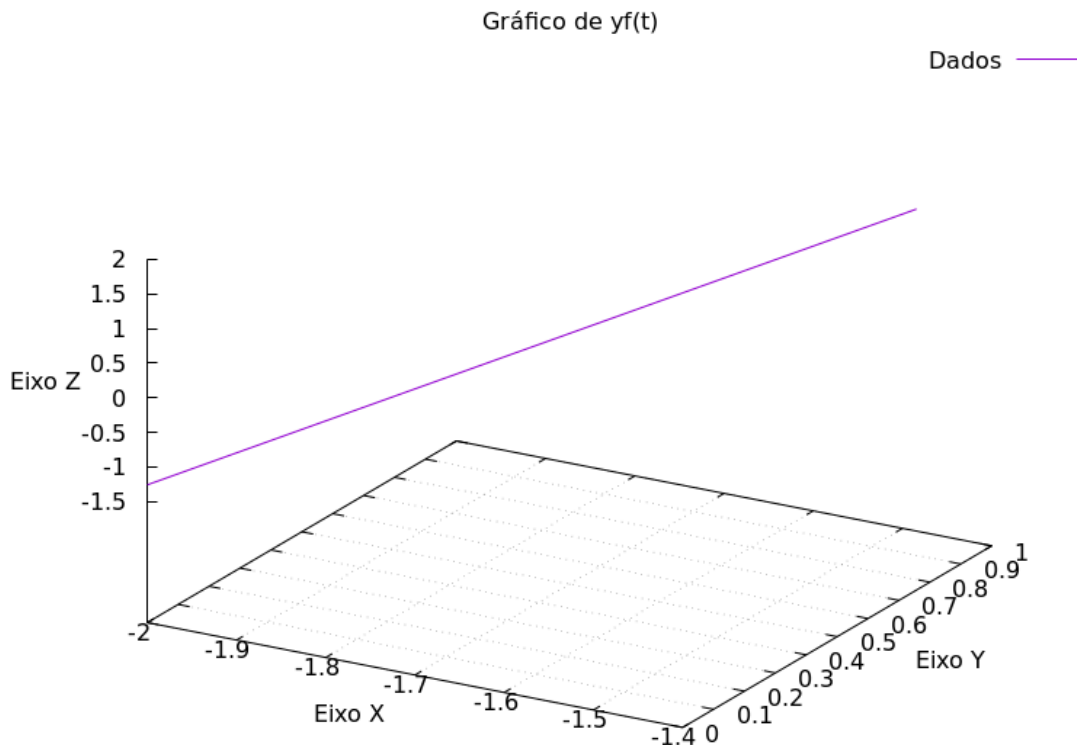
O Gnuplot é um programa de plotagem de gráficos de código aberto. Ele permite criar gráficos a partir de dados em arquivos ou de funções definidas pelo usuário. Nesse projeto, foi possível montar o gráfico de $yf(t)$ graças a esse programa. Logo abaixo é possível visualizar os gráficos obtidos após as simulações com diferentes diâmetros:

a. Gráfico de $yf(t)$ com $D = 2$ cm

Na imagem abaixo, podemos visualizar o gráfico de $yf(t)$ no intervalo de tempo de 0 a 20s com um valor de diâmetro igual a 2 cm. A partir disso, podemos chegar a algumas conclusões:

- Os Eixos X, Y e Z representam, respectivamente, a Coluna 1, 2 e 3 do arquivo '`dados_grafico_yft.txt`';
- O gráfico é representado por uma linha, que apresenta uma tendência ascendente, indicando que o valor de $yf(t)$ aumenta à medida que os valores de X e/ou de Y aumentam;
- A linha ascendente também indica que o robô está mantendo um comportamento linear quando se desloca.

Gráfico 1. Representação gráfica de $y_f(t)$ para um diâmetro de 2 cm.



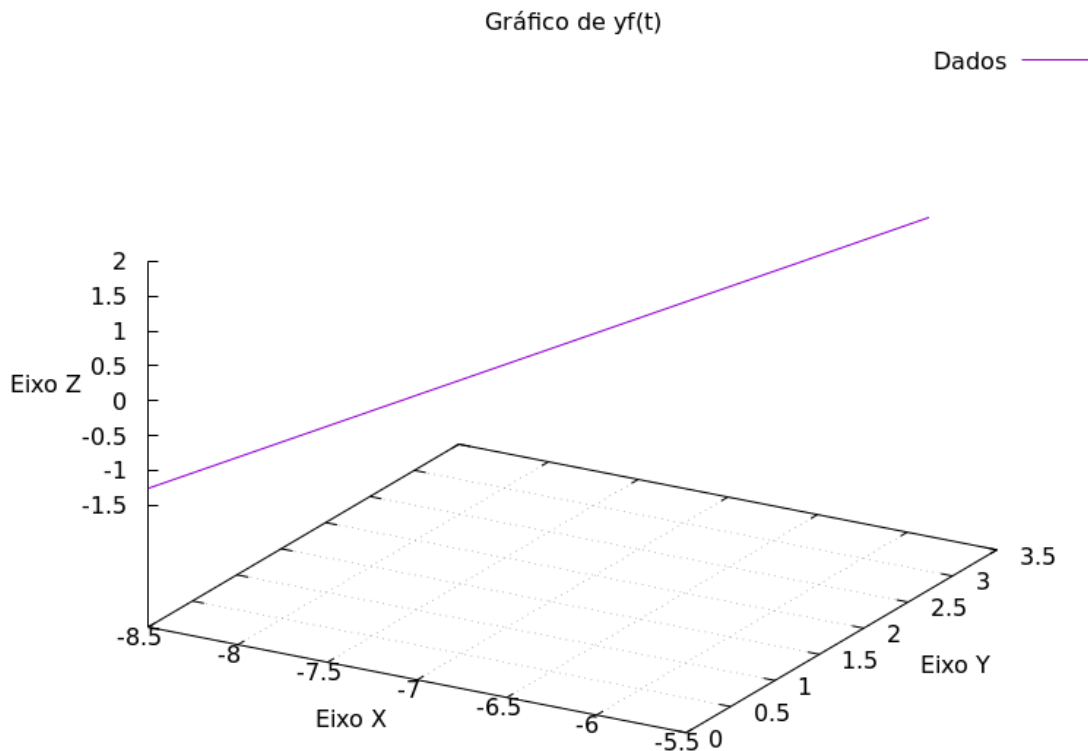
Fonte: Gnuplot.

b. Gráfico de $y_f(t)$ com $D = 15$ cm

Neste próximo gráfico, podemos visualizar o comportamento de $y_f(t)$ no intervalo de tempo de 0 a 20s com um valor de diâmetro igual a 15 cm. Com isso, podemos chegar às seguintes conclusões:

- Os Eixos X, Y e Z também representam, respectivamente, a Coluna 1, 2 e 3 do arquivo '`dados_grafico_yft.txt`';
- O gráfico também é representado por uma linha, que apresenta uma tendência ascendente, indicando que o valor de $y_f(t)$ aumenta à medida que os valores de X e/ou de Y aumentam;
- A linha ascendente também indica que o robô está mantendo um comportamento linear quando se desloca.
- A única diferença que se observa ao comparar os dois gráficos está nos valores dos Eixos X e Y, que são maiores devido ao diâmetro. Isso significa que, quanto maior for o diâmetro do robô, maior será o deslocamento dele pelo tempo.

Gráfico 2. Representação gráfica de $y_f(t)$ para um diâmetro de 15 cm.



Fonte: Gnuplot.

8. Considerações finais

Neste trabalho, foi realizado um algoritmo na linguagem C, utilizando o *Visual Studio Code*, e com o intuito de simular um sistema de controle de um robô móvel utilizando o conceito de threads. Além disso, foram gerados dois arquivos com a saída gerada pelo código e a saída da função $y_f(t)$. Também há gráficos que mostram o comportamento da saída $y_f(t)$ do sistema em duas medidas de diâmetro diferentes, utilizando o programa Gnuplot.

Ao observar os resultados e os gráficos obtidos depois da execução do código, pudemos perceber os diferentes comportamentos do sistema em relação ao valor do diâmetro do robô e concluir que, quanto maior for esse valor, maior também será o deslocamento do robô em relação ao tempo.

9. Referências

ARORA, Himanshu. **Introduction To Linux Threads – Part I, II e III**. The Geek Stuff. 30/03/2012. Disponível em: <<http://www.thegeekstuff.com/2012/03/linux-threads-intro/>>. Acesso em 23 de novembro de 2024.

AZEVEDO, Caio. **Introdução, notação e revisão sobre matrizes**. IMECC-Unicamp. Acesso em 15 de fevereiro de 2024. JOMAR. Matrizes. Universidade Federal do Paraná. Acesso em 23 de novembro de 2024.

OGATA, K. (1995). **Introduction to Discrete-Time Control Systems**. Englewood Cliffs, New Jersey, EUA: Prentice Hall.

SANTANA, Adrielle C. **Sistemas discretos em espaço de estados**. Disponível em: http://professor.ufop.br/sites/default/files/adrielle/files/aula_8_3.pdf. Acesso em 23 de novembro de 2024.

SILVA, Guilherme S. **Integrais**. Todo Estudo. Disponível em: <https://www.todoestudo.com.br/matematica/integrais>. Acesso em 23 de novembro de 2024.

WILLIAMS, Thomas; KELLEY, Colin. **Gnuplot Git Repository**. SourceForge, 2024. Disponível em: <https://sourceforge.net/projects/gnuplot/>. Acesso em 24 de novembro de 2024.