UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ELETRÔNICA E COMPUTAÇÃO
PROGRAMAÇÃO DE SISTEMAS DE TEMPO REAL
PROF. ANDRÉ CAVALCANTE
SEMESTRE 2024/2

LABORATÓRIO 4 - Simulação de Sistemas 2 (Non-RT)

1. Objetivos

Este laboratório tem como objetivo continuar o sistema de controle baseado no movimento de um robô móvel com o uso de múltiplas threads. Porém, o sistema ficará mais complexo e será dividido em 7 tarefas diferentes. 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

As simulações dos laboratórios passados foram feitas a partir de um robô móvel, com acionamento diferencial. Nesse laboratório, o sistema pode ser descrito por esse modelo no espaço de estados:

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

$$y(t) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} x(t) + \begin{bmatrix} R\cos(x_3) \\ R\sin(x_3) \end{bmatrix}$$

onde $x(t) = [x1\ x2\ x3]^T = [x_c\ y_c\ \Theta]^T$, sendo (x_c, y_c) a posição do centro de massa do robô e Θ a sua orientação. A saída do sistema é $y(t)$, correspondendo a frente do robô cujo diâmetro $D = 2R = 0.6m$.

A entrada do sistema é dada por $u(t) = [v\ w]^T$, sendo v a velocidade linear e w a velocidade angular do robô, como podemos ver abaixo:

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

Para esse sistema, tem-se:

$$\begin{aligned}\dot{y}(t) &= \begin{bmatrix} \cos(x_3)u_1 \\ \sin(x_3)u_1 \end{bmatrix} + \begin{bmatrix} -R\sin(x_3)\dot{x_3} \\ R\cos(x_3)\dot{x_3} \end{bmatrix} \\ &= \begin{bmatrix} \cos(x_3) & -R\sin(x_3) \\ \sin(x_3) & R\cos(x_3) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \\ &= L(x)u(t)\end{aligned}$$

e portanto, como $L(x)$ é não singular, o sistema pode ser linearizado por realimentação fazendo-se:

$$u = L^{-1}(x)v(t)$$

sendo $v(t)$ a nova entrada do sistema linearizado e desacoplado dado por:

$$\dot{y}(t) = v(t)$$

Para controlar o sistema, será utilizado um controlador por modelo de referência, com $v(t)$ dada por:

$$v(t) = \begin{bmatrix} \dot{y}_{mx} + \alpha_1(y_{mx} - y1) \\ \dot{y}_{my} + \alpha_2(y_{my} - y2) \end{bmatrix}$$

sendo y_{mx} e y_{my} as saídas dos modelos de referência para a dinâmica do robô nas direções X e Y, respectivamente, dados por:

$$\begin{aligned}\dot{y}_{mx} &= \alpha_1(x_{ref} - y_{mx}) \\ \dot{y}_{my} &= \alpha_2(y_{ref} - y_{my})\end{aligned}$$

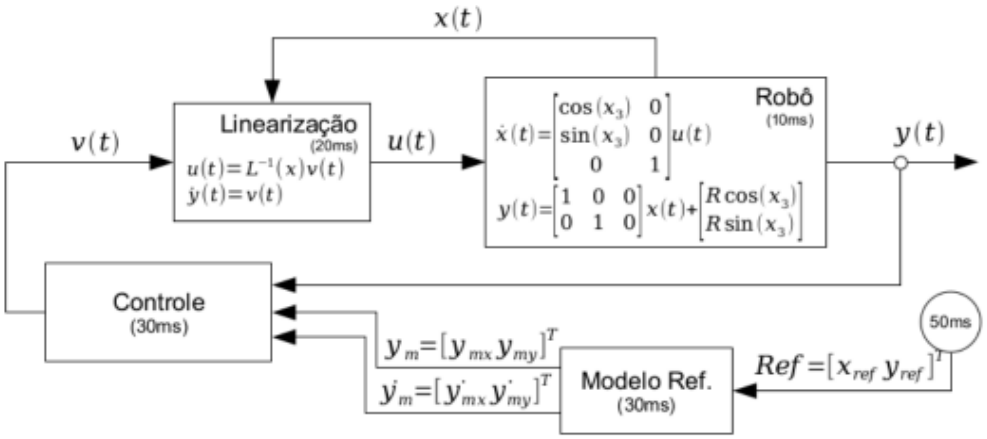
onde $a1$ e $a2$ são definidos empiricamente para melhor controle do robô e são dados como $a1 = a2 = 3$.

Também temos a referência, que é dada por:

$$x_{ref}(t) = \frac{5}{\pi}\cos(0.2\pi t)$$

$$y_{ref}(t) = \begin{cases} \frac{5}{\pi}\sin(0.2\pi t) & \text{, para } 0 <= t < 10 \\ -\frac{5}{\pi}\sin(0.2\pi t) & \text{, para } t >= 10 \end{cases}$$

Juntando todas essas equações e modificações, foi obtido o seguinte diagrama:



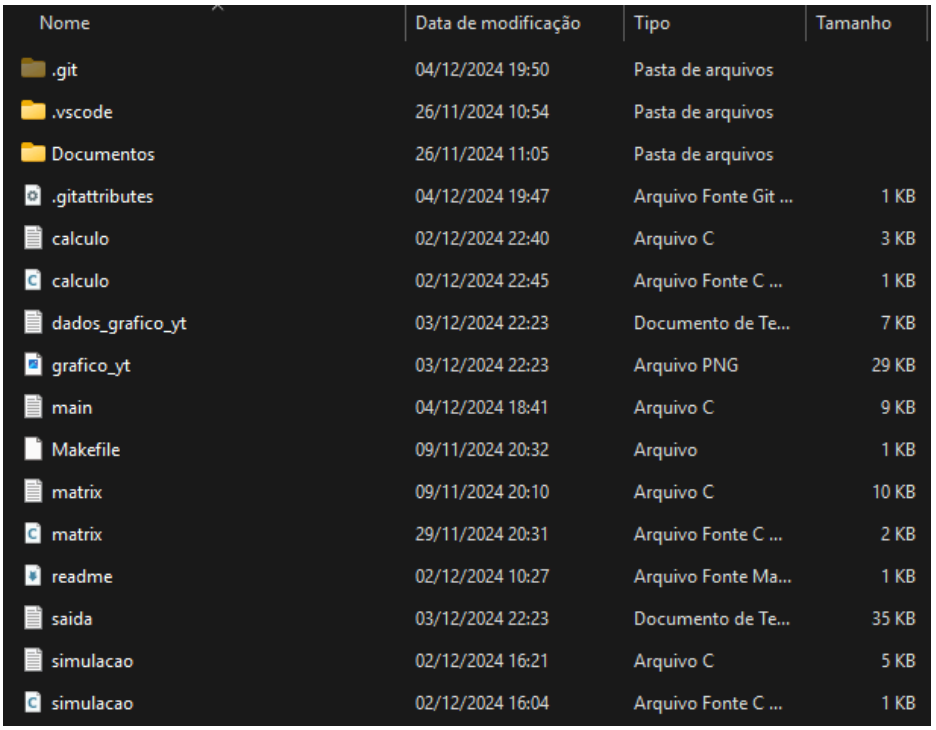
A partir disso, será feito um programa de simulação desse robô móvel em movimento, que será dividido em diversas *threads*:

a. Simulação do robô em si, recebendo u e gerando x e y, com período de 30 ms;
b. Malha de linearização por realimentação, gerando u a partir de x e v com período de 40 ms;
c. Malha de controle, gerando v a partir de y e das saídas dos modelos de referência y_{mx} e y_{my}, com período de 50 ms;
d. Simulação do modelo de referência na direção X, gerando y_{mx} a partir de x_{ref}, com período de 50 ms;
e. Simulação do modelo de referência na direção Y, gerando y_{my} a partir de y_{ref}, com período de 50 ms;
f. Geração da referência, x_{ref} e y_{ref}, com período de 120 ms;
g. Interface com o usuário e a gravação dos dados no disco.

3. 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 13 arquivos fontes e 3 pastas:

Figura 1. Estrutura dos diretórios utilizada.



Nome	Data de modificação	Tipo	Tamanho
.git	04/12/2024 19:50	Pasta de arquivos	
.vscode	26/11/2024 10:54	Pasta de arquivos	
Documentos	26/11/2024 11:05	Pasta de arquivos	
.gitattributes	04/12/2024 19:47	Arquivo Fonte Git ...	1 KB
calculo	02/12/2024 22:40	Arquivo C	3 KB
calculo	02/12/2024 22:45	Arquivo Fonte C ...	1 KB
dados_grafico_yt	03/12/2024 22:23	Documento de Te...	7 KB
grafico_yt	03/12/2024 22:23	Arquivo PNG	29 KB
main	04/12/2024 18:41	Arquivo C	9 KB
Makefile	09/11/2024 20:32	Arquivo	1 KB
matrix	09/11/2024 20:10	Arquivo C	10 KB
matrix	29/11/2024 20:31	Arquivo Fonte C ...	2 KB
readme	02/12/2024 10:27	Arquivo Fonte Ma...	1 KB
saida	03/12/2024 22:23	Documento de Te...	35 KB
simulacao	02/12/2024 16:21	Arquivo C	5 KB
simulacao	02/12/2024 16:04	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 4*, e o mesmo será entregue para avaliação na forma de arquivo “.zip”.

4. Arquivos fontes

Foi utilizada novamente 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)` e `y(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)`, `b(t)`, que é a derivada de `x(t)`, e `y(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:

- **`calcula_entrada(t)`:** Essa função calcula o valor da entrada `u(t)` a partir do valor do tempo `t`, que pode ser entre `t < 0`, `0 ⩽ t < 10` ou `t ⩾ 10`.

- **`calcula_prox_estado(*b, *u)`:** Essa função é responsável por calcular o próximo estado do robô, retornando o produto entre as matrizes `b(t)` e `x(t)`.

- **`integra_x(*x)`:** Essa função irá calcular a integral da matriz `x(t)` ao atualizar os valores de cada elemento de uma forma mais direta.

- **`atualiza_b(*b, *x)`:** Essa função irá atualizar os valores presentes na matriz `b(t)` também de forma direta, como visto nas fórmulas mostradas anteriormente.

- **`calcula_saida(*b, *x, r)`:** Essa função irá calcular o valor de `y(t)` dada as matrizes `x(t)` e `b(t)` e o raio do robô:
  1. Primeiramente serão criadas as matrizes auxiliares `aux_y1` e `aux_y2`, sendo a primeira de dimensões 2x1 e a segunda 2x3;
  2. Atualiza-se os valores das matrizes auxiliares com base no problema proposto;
  3. Depois é realizado o produto entre as matrizes `aux_y2` e `x(t)`;
  4. É somado o resultado da multiplicação com a matriz `aux_y1`;
  5. 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)`, `b(t)` e `y(t)`. Entre as mudanças feitas neste código estão:

- **`Struct Array`:** Essa estrutura foi criada e utilizada para calcular o valor da entrada `u(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:

- **`salva_resultados(t, *ut, *yt, *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 de `t` e das matrizes `u(t)` e `y(t)` em uma linha do arquivo, separados por tabulações;
3. Por fim, a função fecha o arquivo, garantindo que as informações sejam salvas corretamente.
4. O código também tentará abrir o arquivo `'dados_grafico_yt.txt'` para salvar os dados da saída `y(t)` e assim obter o gráfico a partir da função `gera_grafico`;

- **`gera_grafico()`:** Essa função irá criar um gráfico a partir dos dados contidos no arquivo `'dados_grafico_yt.txt'` e exibi-lo na tela quando o programa for executado. Para isso, ele utiliza o programa Gnuplot:
    1. A função tentará abrir o arquivo no modo de leitura (“r”). Caso o arquivo não for aberto, será exibido uma mensagem de erro;
    2. Depois de ler os dados do arquivo, o mesmo será fechado;
    3. Utilizando a função `snprintf`, será construído um comando gnuplot completo. Esse comando contém todas as informações necessárias para gerar o gráfico, incluindo:
        - o formato da saída do terminal (PNG);
        - o nome do arquivo de saída;
        - o título do gráfico;
        - os rótulos dos eixos (X e Y);
        - o grid (grade) do gráfico;
        - o estilo da linha;
        - e o comando plot, que lê os dados do arquivo e plota um gráfico de linhas.
    4. Depois será feita a execução do comando Gnuplot, com o uso da função `system`;
    5. Por fim, a função `feh` será utilizada para abrir o arquivo PNG gerado e exibir o gráfico na tela.

- **`leitura_dados(*nome_arquivo, *u_00, *u_01, *y_00, *y_01, 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 `y(t)` e armazená-los nas variáveis passadas por referência;
3. Por fim, o arquivo será fechado após a leitura.

- **`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.

- **`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. Entre outras modificações, temos:

- **`Struct Dados`:** Essa estrutura armazena o conjunto de estatísticas que serão utilizadas na função `processa_dados`, sendo elas: média, variância, desvio padrão, valores máximos e mínimos e os 3 quadrantes `q1`, `q2` e `q3`.

5.7. main.c

Esse é o arquivo que será executado na simulação, implementando as funções de todos os outros arquivos já citados e utilizando múltiplas threads para otimizar o desempenho. Ele divide o processo em três tarefas principais: produtor, consumidor e encerramento, cada uma sendo executada em uma thread separada. A sincronização entre as threads foi realizada através de semáforos e mutexes. Entre outras funcionalidades e modificações feitas estão:

- **`Defines (constantes)`:** Essas constantes possuem valores que não se alteram durante a execução do programa e serão utilizadas ao longo de toda a simulação. Dentre elas temos:
  1. *INTERVALO (0.03)*: representa o passo de tempo da simulação, ou seja, a quantidade de tempo que avança a cada iteração;
  2. *TEMPO_INICIAL (0.0)*: define o instante inicial da simulação, que começa no tempo de 0 segundos;
  3. *TEMPO_FINAL (20.0)*: define o instante final da simulação. A simulação continuará até atingir 20 segundos;
  4. *R (0.3)*: representa o raio do robô móvel, que foi definido como 0,3 centímetros;
  5. *D (0.6)*: representa o diâmetro do robô móvel, que foi definido como 0,6 centímetros;
  6. *A (3)*: representa a constante de controle do sistema, cujo valor definido foi de 3 unidades;

- **`Mutex`:** O mutex `lock` é usado para proteger o acesso às matrizes `b(t)`, `x(t)`, `u(t)` e `y(t)`, garantindo que apenas uma thread faça modificações nessas matrizes por vez.

- **`Semáforos`:** Os semáforos `semaforo_produtor`, `semaforo_consumidor` e `semaforo_termino` são utilizados para coordenar a produção e o consumo de dados, garantindo que o produtor não produza dados mais rápido do que o consumidor possa consumir.

- **`thread_produtor()`:** Essa função tem como objetivo calcular a próxima entrada do sistema, `u(t)`, e o próximo estado do sistema, `x(t)`, em cada iteração da simulação. Essa função é executada em uma thread separada, permitindo a execução paralela com outras tarefas:
  1. A função entra em um loop que se repete até que o tempo da simulação alcance o valor de `TEMPO_FINAL`;
  2. A thread espera até que haja espaço para novos dados a partir da função `sem_wait(&semaforo_produtor)`, garantindo que o consumidor não esteja sobrecarregado;
  3. Depois adquire o bloqueio do mutex, por meio da função `pthread_mutex_lock(&lock)`, garantindo acesso exclusivo às matrizes compartilhadas;
  4. A partir da função `calcula_entrada(t)`, calcula a entrada do sistema no tempo `t`, retornando um vetor com os valores de entrada;
  5. Depois, na função `calcula_prox_estado(b,u)`, calcula o próximo estado do sistema com base na matriz de sistema `b` e na entrada `u`;
  6. Depois que os valores calculados forem atribuídos às matrizes `u` e `x`, o bloqueio do mutex será liberado pela função `pthread_mutex_unlock(&lock)`, permitindo que outras threads acessem as matrizes;
  7. A partir da função `sem_post(&semaforo_consumidor)`, o consumidor será sinalizado, informando que novos dados estarão disponíveis para processamento;
  8. Por fim, a thread aguarda por 40 milissegundos antes de iniciar a próxima iteração, retornando NULL para indicar o fim da execução da thread.

- **`thread_consumidor()`:** Essa função é responsável por consumir os dados produzidos pela thread `produtor`, realizar cálculos e atualizar o estado do sistema, ou seja, ela processa os dados gerados pela thread produtora:
    1. A thread espera até que o produtor sinalize que novos dados estão disponíveis a partir da função `sem_wait(&semaforo_consumidor)`, evitando que o consumidor tente acessar dados que ainda não foram produzidos;
    2. Depois adquire o bloqueio do mutex, por meio da função `pthread_mutex_lock(&lock)`, garantindo acesso exclusivo às matrizes compartilhadas;
    3. A partir da função `integra_x(x)`, realiza cálculos de integração na matriz `x`;
    4. Depois, na função `atualiza_b(b,x)`, atualiza a matriz `b` com base no estado atual `x`;
    5. Será calculada a saída do sistema `y` com base na matriz `b`, no estado atual `x` e no raio do robô `R`;
    6. Ao atualizar o tempo de término da simulação, os resultados serão salvos dentro de um arquivo especificado na função `salva_resultados(tempo_termino,u,y,arquivo_resultados)`;
    7. Depois que os valores forem salvos, o produtor será sinalizado pela função `sem_post(&semaforo_produtor)`, permitindo que ele possa continuar produzindo novos dados;
    8. O bloqueio do mutex será liberado pela função `pthread_mutex_unlock(&lock)`, permitindo que outras threads acessem as matrizes compartilhadas;
    9. Por fim, a thread aguarda por 50 milissegundos antes de retornar NULL, indicando o fim da execução da thread.

- **`thread_encerramento()`:** Essa função coordena o encerramento da simulação, agindo como um mecanismo de controle para finalizar as outras threads quando a simulação atingir seu ponto final:
    1. A função entra em um loop `while` que continua enquanto a variável global `flag` estiver com o valor 1. Esse loop mantém a thread em execução até que a condição de término seja atingida;

2. A thread espera no semáforo `sem_wait(&semaforo_termino)` até que o consumidor sinalize o término da simulação. Essa sincronização garante que a thread de encerramento não finalize a simulação antes que todas as tarefas tenham sido concluídas;
3. Por fim, ocorre a aquisição e liberação do mutex e eles servem principalmente para garantir que a thread de encerramento tenha acesso exclusivo ao recurso protegido pelo mutex.

- **`inicializar()`:** Essa função prepara o ambiente para a execução da simulação, configurando os elementos essenciais para o funcionamento das threads e inicializando as estruturas de dados utilizadas:
    1. O `semaforo_produtor` é inicializado com o valor 5, permitindo que a thread produtora execute 5 vezes antes de ser bloqueada, caso o consumidor ainda não esteja pronto. Isso dá uma certa margem para o produtor começar a gerar dados antes que o consumidor esteja totalmente sincronizado;
    2. Já o `semaforo_consumidor` é inicializado com o valor 0, indicando que a thread consumidora deve esperar inicialmente, pois não há dados disponíveis para serem processados;
    3. O `semaforo_termino` é inicializado com o valor 0, sendo utilizado para sinalizar o término da simulação;
    4. Logo depois são criadas as matrizes `x`, `b`, `u` e `y`, que provavelmente representam o estado do sistema, a matriz de sistema, a entrada e a saída, respectivamente;
    5. Por fim, as matrizes são inicializadas com os valores especificados na parte teórica.

- **`main()`:** Essa é a função principal do programa, onde serão executados todos os códigos listados anteriormente:
    1. A inicialização do programa ocorre a partir da medição dos tempos inicial e final;
    2. Também ocorre a criação das threads produtora, consumidora e de encerramento, responsáveis, respectivamente, por gerar os dados de entrada, por processar os dados e gerar a saída e por controlar a finalização da simulação;
    3. Logo depois ocorre a criação do menu inicial, com uma mensagem sinalizando o início da simulação do robô móvel;

4. Depois temos a configuração de escalonamento `SCHED_RR` (Round Robin), garantindo que a thread de maior prioridade receba tempo de CPU de forma equitativa;
5. Depois disso, as matrizes `b(t)`, `u(t)`, `x(t)` e `y(t)` serão inicializadas com os valores previstos na parte teórica;
6. Antes da execução das threads em si, os arquivos de saída são excluídos para garantir que não haja dados antigos e assim não interferir os dados do gráfico de `y(t)`;
7. As threads serão iniciadas pela função `pthread_create`;
8. A thread principal espera pela finalização das threads produtora e consumidora por causa da função `pthread_join`;
9. Logo depois, a thread principal sinaliza o término da simulação para a thread de encerramento, esperando pela finalização da última;
10. O tempo de execução então é calculado e impresso no terminal;
11. Os semáforos são destruídos;
12. O gráfico da saída `y(t)` (`'grafico_yt.png'`) é gerado a partir do arquivo `'dados_grafico_yt.txt'`;
13. Por fim, as matrizes são liberadas da memória.

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 `y(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 `y(t)` para a plotagem do gráfico 2D. Dentro do arquivo, como fizemos a simulação de 0 a 20 segundos, e cada iteração tem o intervalo de 0,03 segundos, então temos 667 linhas de valores da matriz `y(t)`, com 2 dados cada.

5.12. grafico_yft.png

Essa imagem descreve o gráfico bidimensional da matriz `y(t)`, que possui três eixos (Eixo X e Eixo Y contendo os dados resultantes da simulação do robô.

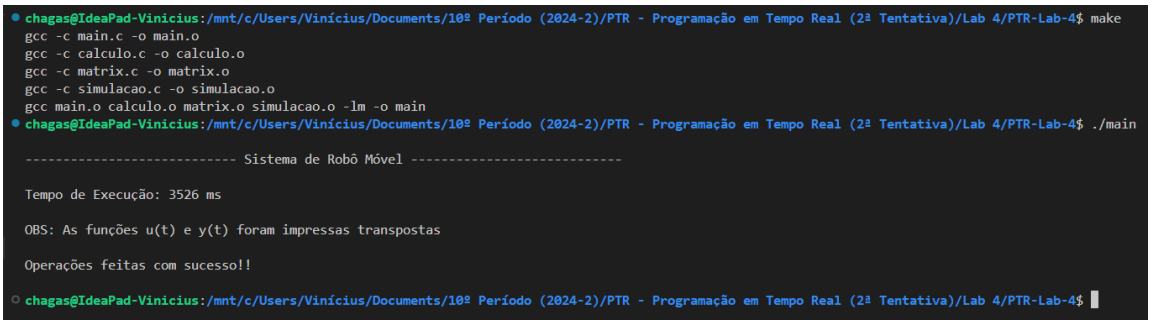
5. Resultados

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, o programa será executado e produzirá a saída (`'saida.txt'`) com base na simulação especificada.
6. O programa também produzirá um gráfico de `y(t)` (`'grafico_yt.png'`) com base nos dados presentes no arquivo `'dados_grafico_yt.txt'`.

Na figura abaixo, podemos ver o resultado da simulação do robô móvel com um diâmetro de 0,6 cm:

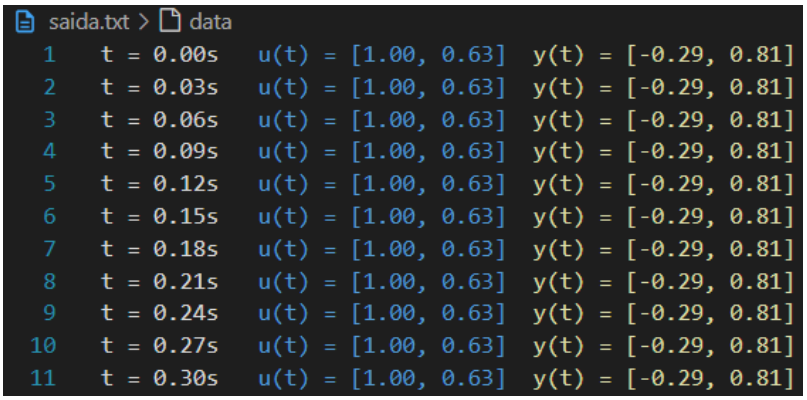
Figura 2. Compilação do código *main.c*, utilizando o VS Code.

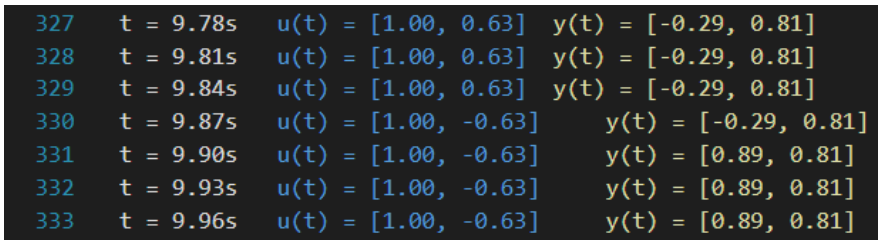


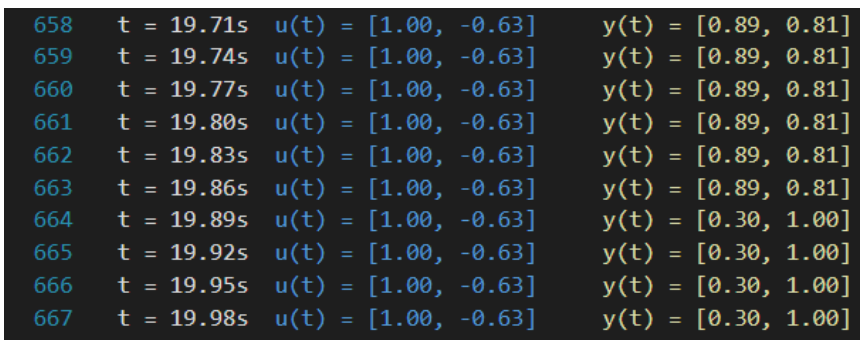
Fonte: Visual Studio Code.

Podemos notar 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 é iniciado e a simulação começa. Depois que a operação acaba, é exibido na tela o tempo de execução, uma mensagem de observação indicando que as matrizes no arquivo de saída estão transpostas e uma última mensagem indicando que as operações foram realizadas com sucesso.

Figura 3, 4 e 5. Arquivo gerado com os dados obtidos na simulação.

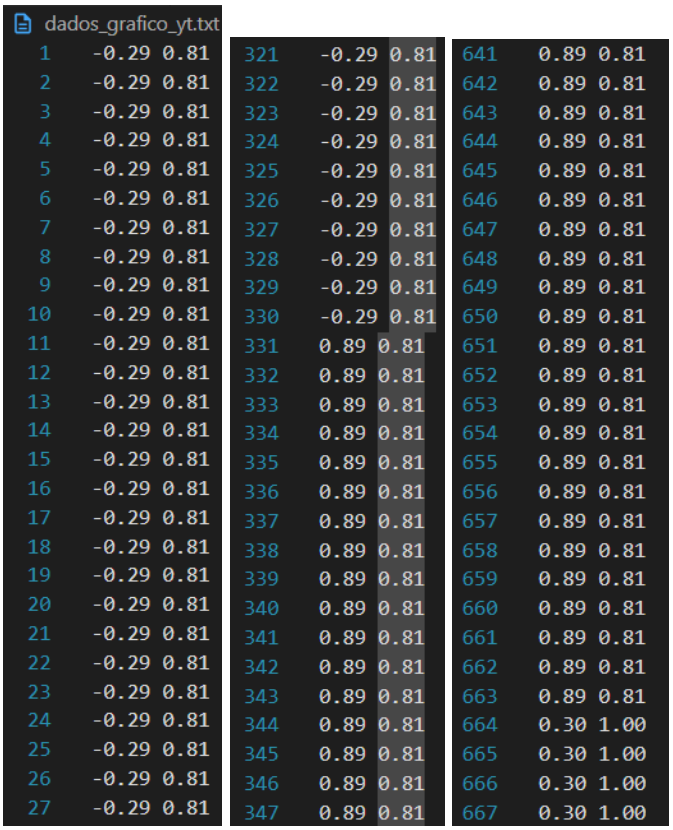






Fonte: Visual Studio Code.

Figura 6, 7 e 8. Arquivo gerado com os dados de y(t) para a plotagem do gráfico.



Fonte: Visual Studio Code.

Nas Figuras acima, podemos ver os arquivos onde os resultados da simulação foram armazenados.

Ao analisar o arquivo `'saida.txt'`, podemos notar que os resultados obtidos na entrada `u(t)` na Figura 4 não seguem exatamente o comportamento adequado, pois antes de `t = 10.00s`, os valores já estavam exibindo os resultados esperados para `t ⩾ 10`.

Esse comportamento segue para os valores de `y(t)`. Nesse caso tiveram dois momentos em que a variável mostrou dados fora do padrão:

- `t = 9.87s` (Figura 4): enquanto `u(t)` apresenta a mudança no seu comportamento, `y(t)` continua com o comportamento anterior. Isso se deve provavelmente pela forma como as threads foram executadas, e também pelo intervalo de cada iteração, que mostrou esses erros que não existiam nos trabalhos anteriores;

- `t = 19.89s` (Figura 5): aqui o comportamento de `u(t)` continua como o previsto, porém `y(t)` muda seu comportamento. Isso também se deve à mesma justificativa dada anteriormente.

O comportamento visto no arquivo `'dados_grafico_yt.txt'` se assemelha ao que pode ser visto com os valores de `y(t)` no arquivo de saída.

6. Gráficos

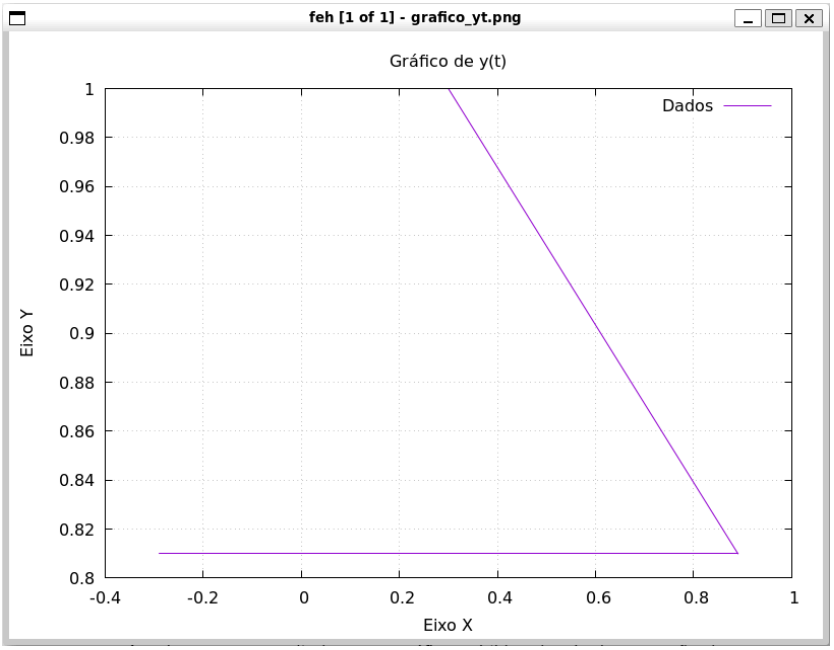
Novamente foi utilizado o programa de plotagem de gráficos chamado Gnuplot. Ele permitiu criar o gráfico da saída do sistema a partir dos dados obtidos na simulação e armazenados no arquivo `'dados_grafico_yt.txt'`.

a. Gráfico de y(t)

Na imagem abaixo, podemos visualizar o gráfico de `y(t)` no intervalo de tempo de 0 a 20s com um valor de diâmetro igual a 0,6 cm. A partir disso, podemos chegar a algumas conclusões:

- Os Eixos X e Y representam, respectivamente, a Coluna 1 e 2 do arquivo `'dados_grafico_yft.txt'`;
- O gráfico retrata duas linhas, interligando os três pontos vistos no arquivo base.
- Primeiramente ele mantém um comportamento crescente em relação o Eixo X;
- Depois, quase no final da simulação, o valor da saída muda para `[0.30 1.00]`, formando uma linha que segue um comportamento crescente em relação ao Eixo Y, porém um comportamento decrescente no Eixo X. O que leva a crer que esse valor de saída foi causado pelas threads e pelo intervalo das iterações.

Gráfico 1. Representação gráfica de y(t).



Fonte: Gnuplot.

7. Considerações finais

Neste trabalho, deu-se continuação ao programa que simula um sistema de controle de um robô móvel utilizando o conceito de múltiplas threads.

Além disso, foi utilizado novamente o *Virtual Studio Code* para a implementação dos códigos e também foram gerados dois arquivos com a saída gerada do sistema `y(t)`. Também há gráficos que mostram o comportamento de `y(t)`, utilizando o programa Gnuplot. Em relação ao código, desta vez foram utilizados mutexes e semáforos para um melhor controle das threads.

Ao observar os resultados e os gráficos obtidos depois da execução do código, pudemos notar algumas melhorias em relação ao código e à forma de obtenção dos dados. Porém, pudemos perceber alguns comportamentos irregulares do sistema em relação aos valores de entrada e saída do sistema, `u(t)` e `y(t)` respectivamente. Isso ocorreu por causa do momento em que as threads foram chamadas no programa e também pelo intervalo de tempo de `0.03s` entre cada iteração, erros esses que não existiam em trabalhos anteriores onde o intervalo era maior.

8. 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.

9. Apêndices

Link de Acesso ao Git:
https://github.com/vinichagas/PTR-Lab-4.git