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 Sistemas 2 (Non-RT)

Nome: Matheus Carvalho Reges – 22152027

1. Objetivos

O trabalho consiste em desenvolver um programa em linguagem C para simular o comportamento e o controle de um robô móvel com acionamento diferencial. O programa utiliza múltiplas threads para dividir as funcionalidades do sistema, como simulação, controle e armazenamento de dados. A implementação deve considerar as seguintes tarefas:

1. **Simulação do robô**: Calcular a evolução do estado do robô (x, y, θ) ao longo do tempo, integrando as equações diferenciais do sistema.
2. **Linearização por realimentação**: Ajustar as entradas do sistema com base em um modelo matemático para desacoplar as dinâmicas nas direções X e Y.
3. **Controle por modelo de referência**: Determinar os comandos do sistema com base nos erros entre a saída atual e as referências desejadas.
4. **Simulação dos modelos de referência**: Reproduzir o comportamento esperado nas direções X e Y, fornecendo as variáveis necessárias para o controle.
5. **Geração de referências**: Produzir as trajetórias desejadas $x_{ref}(t)$ *e* $y_{ref}(t)$ de acordo com o tempo e condições específicas.
6. **Armazenamento dos dados**: Registrar a posição, orientação e tempo em um arquivo ASCII no formato especificado.
7. **Interação com o usuário**: Permitir ajustes de parâmetros de controle (α_1 e α_2) e visualização dos dados da simulação.

Além disso, o programa deve avaliar o desempenho do sistema, medindo períodos e jitter para as tarefas com e sem carga, e gerar gráficos comparando a trajetória do robô com as referências. O relatório final deve documentar as etapas de desenvolvimento, estrutura do código e análise crítica dos resultados.

2. Introdução

A simulação de sistemas de robôs móveis é uma prática fundamental para avaliar o desempenho e a eficiência de modelos de controle antes da implementação física. Este trabalho aborda um robô móvel com acionamento diferencial, que é um modelo amplamente utilizado devido à sua simplicidade mecânica e flexibilidade de controle.

O robô é descrito por um modelo matemático no espaço de estados, com as seguintes equações:

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

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

onde:

- $x(t) = [x_1, x_2, x_3]^T = [x_c, y_c, \theta]^T$ representa a posição e a orientação do robô.
- $u(t) = [v, w]^T$ são as entradas de controle (velocidade linear e angular).
- $y(t)$ é a posição da frente do robô, considerando $D = 2\mathrm{R} = 0.6\mathrm{m}$.

Para controlar o sistema, a linearização por realimentação transforma as entradas $v(t)$ em $u(t)$, permitindo que o modelo original seja desacoplado em dinâmicas independentes nas direções X e Y. Um controlador por modelo de referência ajusta as entradas para minimizar os erros de seguimento em relação às referências $x_{ref}(t)$ e $y_{ref}(t)$.

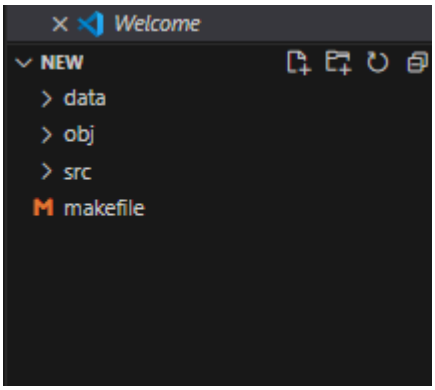
O programa é estruturado em múltiplas threads, com as seguintes funcionalidades principais:

1. **Simulação do robô**: Atualiza a posição (x, y) e orientação (θ) com base nas entradas $u(t)$, sendo executada a cada 30 ms.
2. **Linearização por realimentação**: Calcula $u(t)$, a partir das variáveis de controle e estado, com período de 40 ms.
3. **Controle**: Gera os sinais de controle $v(t)$ a partir dos modelos de referência (y_{mx}, y_{my}) e dos erros, com período de 50 ms.
4. **Geração de referências**: Produz $x_{ref}(t)$ e $y_{ref}(t)$. com mudanças específicas no tempo, executando a cada 120 ms.
5. **Simulação dos modelos de referência**: Atualiza os valores y_{mx} e y_{my} para representar o comportamento desejado do robô, com período de 50 ms.
6. **Armazenamento dos dados**: Registra a posição, orientação e tempo em um arquivo de saída no formato especificado.
7. **Interação com o usuário**: Ajusta os parâmetros de controle (α_1, α_2) durante a execução.

O programa é sincronizado por meio de mutexes e variáveis de condição para evitar condições de corrida no acesso às variáveis compartilhadas. Essa abordagem modular garante organização, eficiência e escalabilidade, além de facilitar a análise do desempenho do sistema em condições reais e simuladas.

3. Estrutura de Diretórios

A organização de diretórios segue boas práticas de engenharia de software, facilitando a manutenção, reutilização e expansão do código. A seguir, detalhamos a hierarquia e funções de cada diretório.



3.1 Pasta Principal

A pasta raiz do programa contém:

- O arquivo **Makefile**, utilizado para gerenciar a compilação e execução.
- Diretórios src, obj e data.
- Arquivos de saída gerados após a compilação.

Essa organização separa os artefatos gerados pelo programa do código-fonte, permitindo uma abordagem mais limpa e estruturada.

3.2 Arquivos .c e .h

- Armazenados na pasta src.
- Diretório que armazena os arquivos-fonte (.c) e cabeçalhos (.h) do projeto.
- É a principal área de desenvolvimento, onde reside o código reutilizável e modular.

3.3 Arquivos .o

- Armazenados na pasta obj.
- Armazena os arquivos objetos (.o) gerados na etapa de compilação.
- Essa separação ajuda a manter os arquivos intermediários organizados e fora da raiz do projeto.

3.4 Arquivos .txt

- Armazenados na pasta data.
- Armazena saídas dos códigos (arquivos de resultados gerados pela simulação ou controle) como:
    - **pos_or_log.txt**: arquivo de saída log que registra os estados ou a posição ao longo do experimento.
    - **registro_ref.txt, registro_ymx.txt, registro_ymy.txt, tempo_e_jitter**.txt: Arquivos de registro de diferentes variáveis e saídas relevantes usadas no experimento.
    - **posYxposX.m, gráficos_e_tabela, tempo_e_jitter**: Script MATLAB para análise ou visualização de dados experimentais.
    -
- Contém também os códigos em Octave utilizados para visualização de gráficos e tabelas.
- Serve como uma ponte para a análise de resultados e validação do funcionamento do sistema.

3.5 Arquivos raiz

- **main**: o executável gerado após a compilação.
- **makefile**: Script para automação de compilação, facilitando o gerenciamento do processo.

3.5 Makefile

O arquivo Makefile, localizado na pasta raiz, automatiza o processo de:

- Compilação (é necessário executar `make`).
- Execução (é necessário executar `make run` após a compilação).

Essa automação reduz erros manuais e garante um fluxo de trabalho consistente.

3.5 Benefícios Adicionais da Organização

- **Clareza na análise dos resultados**: Com as saídas do programa organizadas em `data`, é mais fácil acessar os arquivos para análise posterior. Além disso, centralizar os scripts do Octave no mesmo local garante que os gráficos e tabelas sejam gerados diretamente a partir dos resultados, promovendo uma integração prática.
- **Reutilização e automação**: Os scripts de visualização em Octave podem ser reutilizados em diferentes experimentos, o que é útil ao realizar alterações no sistema e testar novamente.
- **Melhor organização de dados experimentais**: Separar resultados experimentais em `data` ajuda a evitar que arquivos temporários ou irrelevantes fiquem misturados com o código-fonte, preservando a clareza.

3.6 Análise Crítica

- **Facilidade de Visualização**: A inclusão dos scripts em Octave junto às saídas permite que os resultados sejam analisados rapidamente por meio de gráficos e tabelas. Isso é crucial para validar o desempenho do sistema e identificar melhorias necessárias.

- **Modularidade e Reuso**: Com os scripts de visualização organizados em uma única pasta, eles podem ser facilmente reutilizados para novos experimentos, economizando tempo e esforço na geração de gráficos e tabelas.

3.7 Compilação

O processo de compilação envolve os seguintes passos:

1. Navegar até o diretório raiz do programa.
2. Executar make para gerar os arquivos binários.
3. Utilizar make run para executar o programa.

4. Arquivos Fonte

A seguir, detalhamos as funcionalidades de cada arquivo .c e .h presentes no diretório src:

4.1 main.c

- **Objetivo**: Função principal que gerencia a execução multithread do sistema. Coordena inicialização, execução e término das threads.
- **Estrutura**:
    - **finalizarExecucao**: Captura o sinal SIGINT (Ctrl+C) para finalizar o programa.
    - **Execução principal**:
        1. Configura parâmetros e inicializa estruturas compartilhadas (DadosCompartilhados e BufferArmazenamento).
        2. Cria threads principais e auxiliares responsáveis pela simulação, coleta de dados e controle.
        3. Aguarda a conclusão de todas as threads e libera os recursos alocados.
    - **Responsabilidades das threads**:
        - Simulação do movimento do robô.
        - Controle baseado em dados de sensores e armazenamento de resultados.

4.2 estate.c

- **Objetivo**: Implementa funções relacionadas ao estado e controle do robô, como cálculo de movimento, criação de matrizes de transformação e controle de velocidade.

- **Funções principais**:
    1. **computeRobotEstate**: Atualiza a posição e orientação do robô baseado em velocidade angular (angVel) e intervalo de tempo (deltaTime).
    2. **criaMatrizTransformacao**: Gera uma matriz de transformação linear baseada na orientação do robô.
    3. **criaMatrizTransformacaoInv**: Calcula a matriz inversa da transformação gerada, usada para conversão de coordenadas.
    4. **calcCtrl**: Calcula os sinais de controle (ex.: rotações das rodas) a partir das velocidades desejadas usando a matriz inversa de transformação.

4.3 estate.h

- **Objetivo**: Declara as funções implementadas em `estate.c` e expõe sua interface para outros módulos do sistema.
- **Conteúdo**:
    - Declaração das funções (`computeRobotEstate`, `criaMatrizTransformacao`, `calcCtrl`, etc.) e inclusão de dependências relevantes (`matrix.h`, `math.h`, etc.).

4.4 matrix.c

- **Objetivo**: Fornece operações de álgebra linear para manipulação de matrizes, usadas em cálculos de controle e simulação.
- **Funções principais**:

    1. **initialize_matrix / destroy_matrix**: Alocam e liberam memória para matrizes.
    2. **Operações matemáticas**:
    3. Soma, subtração, multiplicação de matrizes.
    4. Escalonamento e transposição.
    5. **Cálculo do determinante**: Necessário para verificar se uma matriz é invertível.
    6. **invert_matrix**: Calcula a matriz inversa para transformações de coordenadas.
    7. **Exibição e cópia de matrizes**: Para debug e manipulação.

4.5 matrix.h

8. **Objetivo**: Define a estrutura de dados para matrizes (Matrix) e declara as funções implementadas em matrix.c.
- **Conteúdo**:

    - Estrutura Matrix: Contém número de linhas, colunas e ponteiro para os dados.
    - Funções de inicialização, destruição e operações matemáticas.

4.6 simulation.C

- **Objetivo**: Implementa a lógica de simulação do sistema robótico, incluindo controle, coleta de dados e armazenamento.
- **Funções principais**:
    1. **calcularSaida**: Calcula uma saída (ex.: posição X ajustada) baseada no estado atual do robô.
    2. **executarSimulacao**: Simula o movimento do robô, atualizando seu estado periodicamente.

3. **controleEColeta**: Gerencia o controle do robô e coleta dados para armazenamento.
4. **storageThread**: Armazena os dados coletados em um arquivo de saída (data/pos_or_log.txt).
5. **linearizacaoThread**: Realiza cálculos de linearização (sinais de controle).
6. **controlThread**: Implementa a aplicação de sinal de controle baseado em modelos.
7. **Inicialização e destruição de recursos**:
    - inicializarBuffer, inicializarDadosCompartilhados, destruirRecursos.

4.7 simulation.h

- **Objetivo**: Define as funções utilizadas para a simulação, além de estruturas auxiliares (DadosCompartilhados, BufferArmazenamento).
- **Estruturas principais**:

1. **DadosCompartilhados**: Contém o estado atual do robô, entrada de controle e mutex para sincronização.
2. **BufferArmazenamento**: Armazena amostras coletadas durante a simulação.

4.8 simulation_model.C

- **Objetivo**: Fornecer as ferramentas matemáticas e de lógica necessárias para modelar o comportamento do robô no ambiente de simulação, incluindo cálculos de movimento, detecção de colisão, e restrições.
- **Funções principais**:
    1. **imporLimitesVelocidade**:
        - Garante que a velocidade do robô esteja dentro de limites pré-definidos. Caso ultrapasse, ajusta ao limite permitido.
    2. **atualizarEstado**:
        - Atualiza as coordenadas (posX, posY) e a orientação (orientacao) do robô, com base em velocidade, ângulo de rotação e tempo decorrido.
    3. **calcularProximaPosicao**:
        - Estima a próxima posição e orientação do robô sem alterar o estado atual. Útil para prever comportamento antes de aplicá-lo.
    4. **calcularErro**:
        - Calcula o erro entre o valor desejado (setpoint) e o valor real, fundamental para algoritmos de controle como PID.
    5. **ajustarOrientacao**:
        - Normaliza o ângulo de orientação do robô, garantindo que esteja em um intervalo lógico (ex.: −π-\pi−π a +π+\pi+π).
    6. **detectarColisao**:
        - Avalia se o robô colidiu com obstáculos ou ultrapassou os limites do ambiente simulado, retornando o status da colisão.
    7. **limitarAngulo**:
        - Ajusta ângulos para que permaneçam em um intervalo fixo (similar a ajustarOrientacao).
    8. **calcularDistancia**:
        - Calcula a distância euclidiana entre dois pontos (x1, y1) e (x2, y2), utilizada para medir proximidade de alvos ou obstáculos.
    9. **gerarRuido**:

- Adiciona ruído aleatório a valores como medições de sensores ou movimentação, simulando as imperfeições do mundo real.

10. **atualizarVelocidade**:
    - Atualiza a velocidade do robô com base na aceleração e no tempo decorrido, assegurando que respeite os limites físicos definidos

4.9 simulation_model.h

- **Objetivo**: Declarar todas as funções e estruturas utilizadas no módulo simulation_model.c.
- **Funções declaradas**:

1. imporLimitesVelocidade(double velocidade);
2. atualizarEstado(EstadoRobo *estado, double velocidade, double angulo, double deltaTempo);
3. calcularProximaPosicao(EstadoRobo *estadoAtual, EstadoRobo *estadoFuturo, double velocidade, double angulo, double deltaTempo);
4. calcularErro(double valorDesejado, double valorAtual);
5. ajustarOrientacao(double orientacao);
6. detectarColisao(double posX, double posY);
7. limitarAngulo(double angulo);
8. calcularDistancia(double x1, double y1, double x2, double y2);
9. gerarRuido(double intensidade);
10. atualizarVelocidade(double velocidadeAtual, double aceleracao, double deltaTempo);

- **Estruturas principais**:
    - typedef struct EstadoRobo { ... } EstadoRobo;
        - Representa o estado atual do robô (posição, orientação, velocidade, etc.).

5. Threads no Programa

O sistema implementado utiliza diversas threads, cada uma responsável por executar uma tarefa específica para simular e controlar o robô. A seguir, apresentamos uma descrição detalhada do papel de cada thread:

5.1. Thread de Simulação (executarSimulacao)

- **Objetivo**: Simular o movimento do robô em função de parâmetros como a entrada de controle angular e o passo de tempo.
- **Funcionamento**:
    - Atualiza a posição e a orientação do robô chamando a função computeRobotEstate.
    - O cálculo é realizado a cada incremento de tempo (PASSO_TEMPO).
    - Utiliza o mutex de DadosCompartilhados para garantir que a atualização do estado do robô seja feita de forma segura em relação às outras threads.
- **Interação**: Colabora com outras threads que utilizam o estado do robô.

5.2. Thread de Controle e Coleta (controleEColeta)

- **Objetivo**:
  - Coletar dados do robô, como posição e orientação, para fins de monitoramento e controle.
  - Armazenar esses dados no buffer compartilhado.
- **Funcionamento**:
  - Calcula as entradas do sistema e os valores de saída (e.g., posição atual e orientação).
  - Cria uma nova amostra de dados e a adiciona ao buffer compartilhado.
  - Utiliza mutexes para sincronizar o acesso ao buffer e ao estado compartilhado.
- **Interação**: Coordena-se com a thread de armazenamento para registrar os dados coletados.

5.3. Thread de Armazenamento (storageThread)

- **Objetivo**: Registrar as informações do estado do robô em um arquivo de saída.
- **Funcionamento**:
  - Monitora o buffer de dados e escreve as amostras coletadas no arquivo data/pos_or_log.txt.
  - Quando o buffer está vazio, a thread entra em espera condicional até que novos dados estejam disponíveis ou a execução seja finalizada.
  - Sincroniza o acesso ao buffer para evitar inconsistências nos dados armazenados.
- **Interação**: Trabalha em conjunto com a thread de coleta para processar os dados registrados no buffer.

5.4. Thread de Linearização (linearizacaoThread)

- **Objetivo**: Realizar cálculos de linearização e determinar os sinais de controle baseados no estado atual do robô.
- **Funcionamento**:
  - Utiliza a função calcCtrl para calcular os sinais de controle com base em velocidades desejadas e na orientação atual.
  - Atualiza os valores de controle no estado compartilhado, protegendo os dados com mutexes.
- **Interação**: Atua como suporte para a thread de controle, auxiliando na aplicação de comandos corretos.

5.5. Thread de Controle (controlThread)

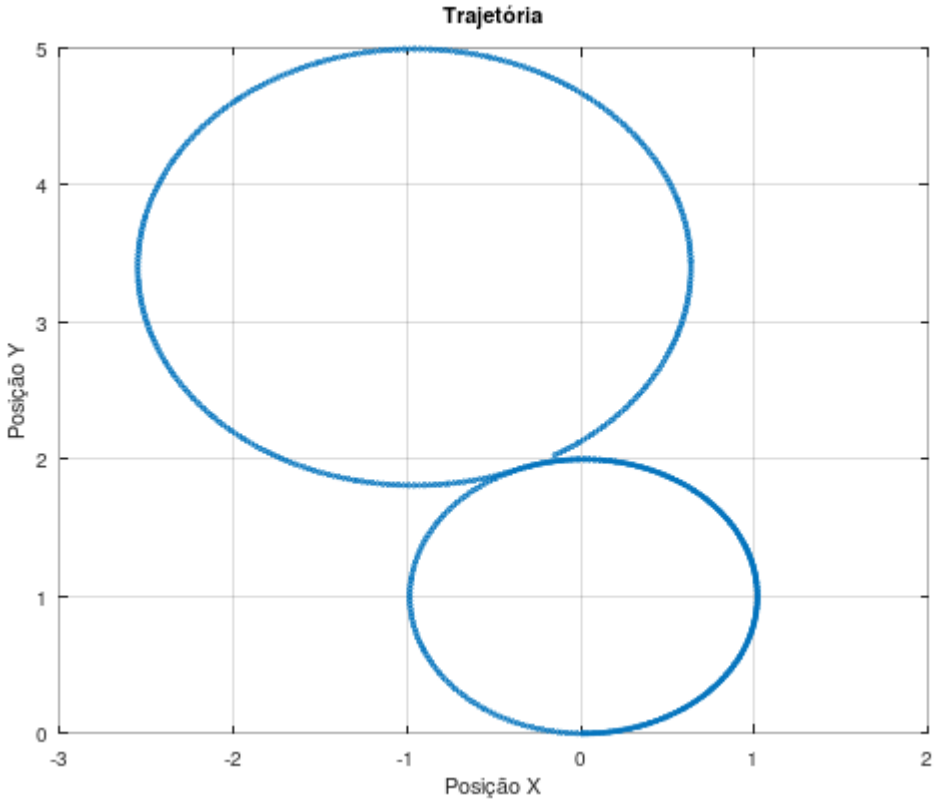
- **Objetivo**: Calcular e aplicar os sinais de controle no robô.
- **Funcionamento**:
  - Calcula os comandos baseados nos valores de entrada desejados (ymx, ymy) e na posição atual do robô.
  - Atualiza o estado compartilhado para refletir os ajustes de controle aplicados.
- **Interação**: Trabalha em conjunto com a thread de simulação para monitorar e corrigir o movimento do robô.

5.6. Estrutura Geral

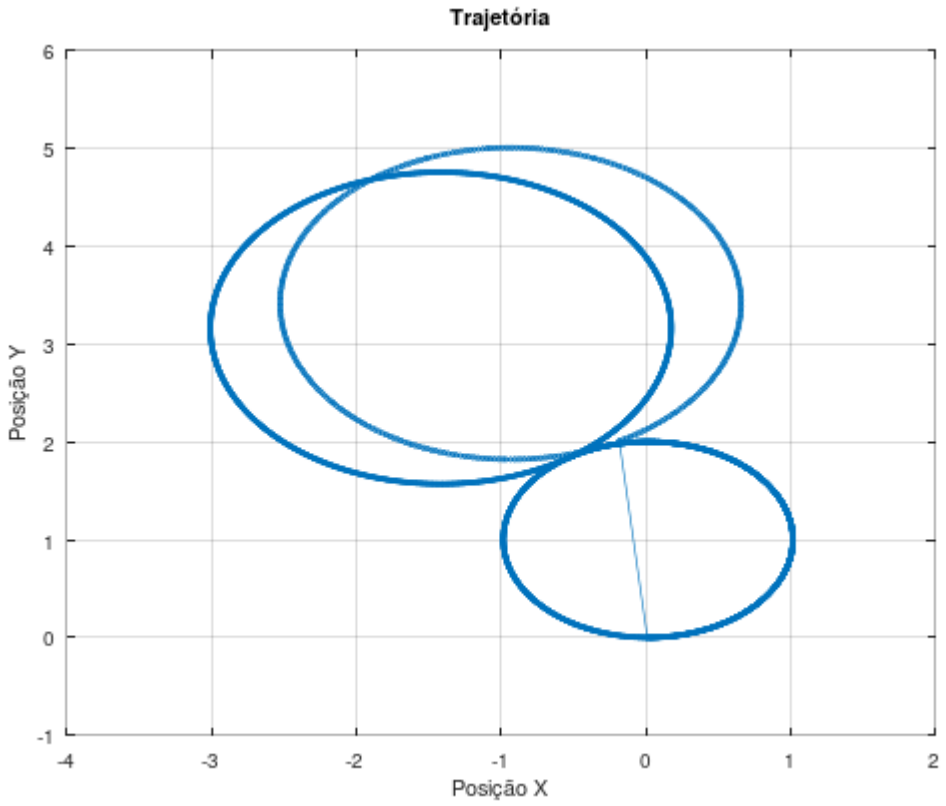
O sistema é projetado para executar simultaneamente as threads principais, utilizando mecanismos de sincronização como mutexes e variáveis condicionais para garantir a integridade dos dados compartilhados. Essa abordagem multithreaded permite dividir o trabalho de forma eficiente e lidar com a complexidade das tarefas do sistema.

6. Resultados

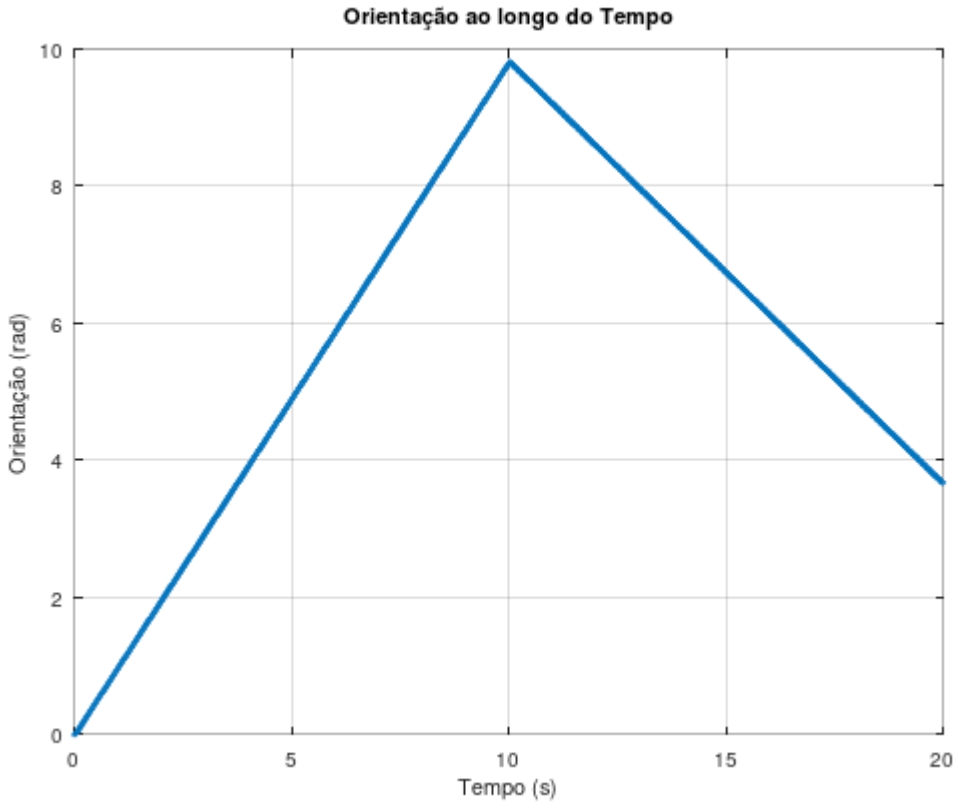
6.1 gráfico y(k) com carga



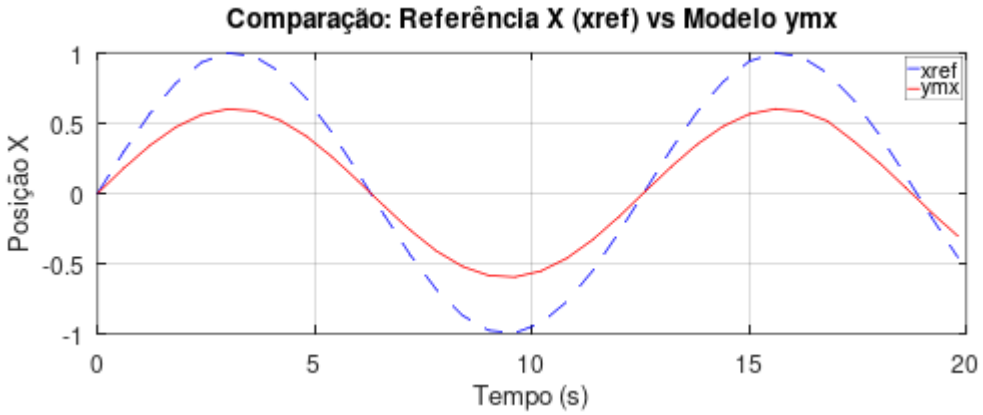
6.2 gráfico y(k) sem carga



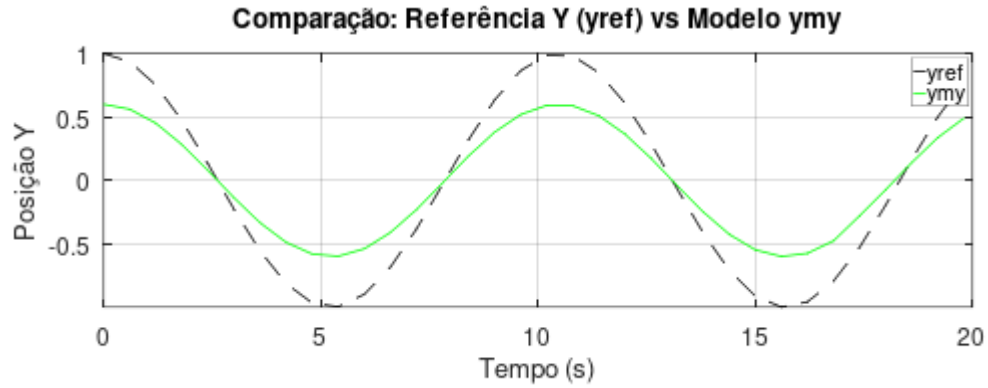
6.3 orientação ao longo do tempo com carga



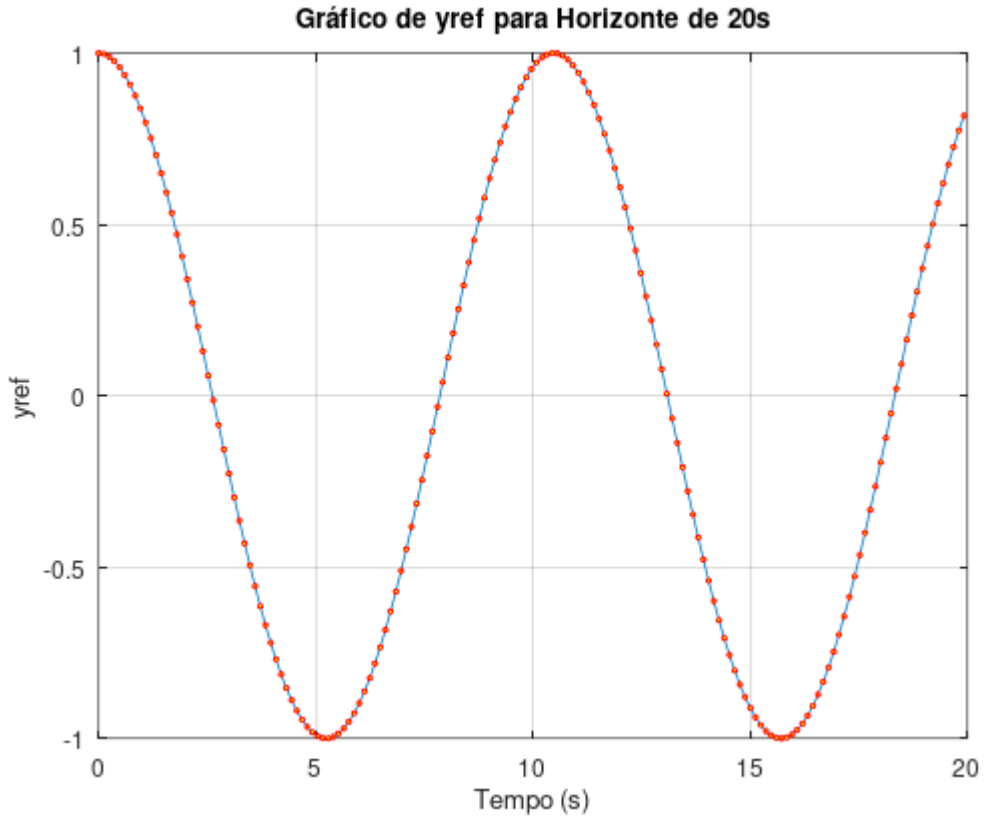
6.4 xref x ymx (com carga)



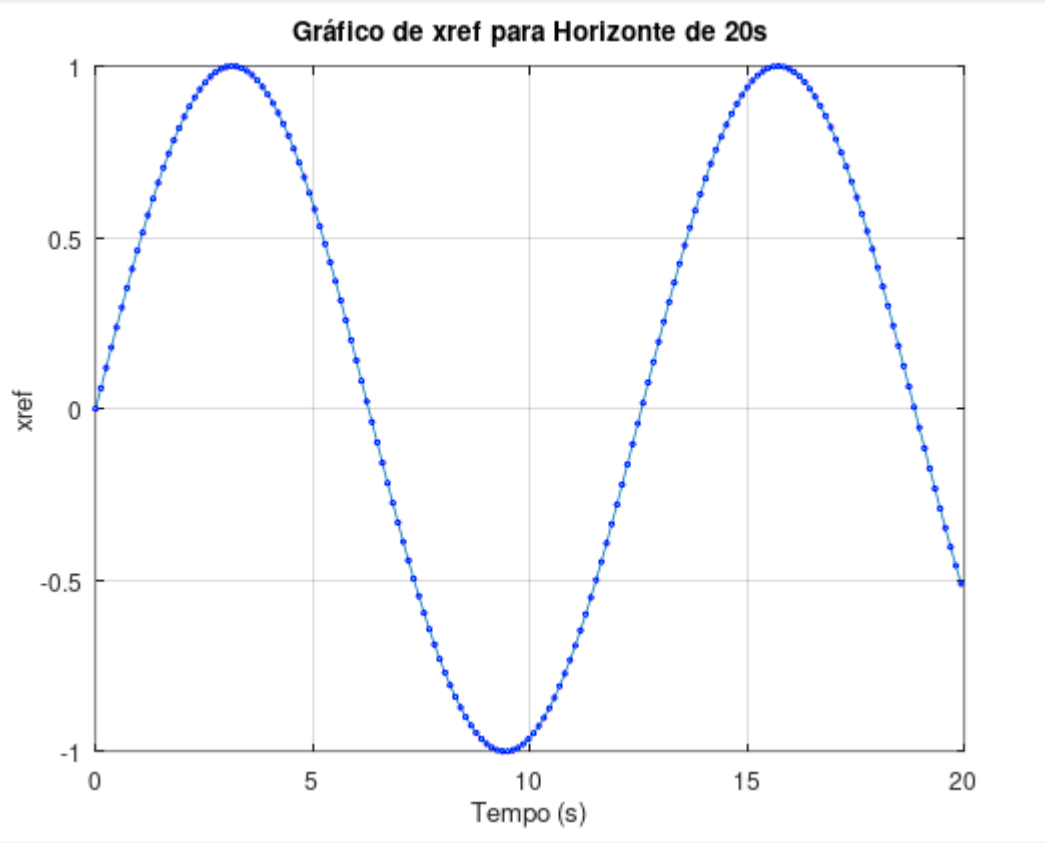
6.5 yref x ymy (Com carga)



6.6 yref (Com carga)



6.7 xref(Com carga)



6.8 Tabela comparando os valores de média, variância, desvio padrão, valores máximos e mínimos dos tempos de computação e do respectivo jitter para o sistema com carga e analise

Tabela Comparativa: Média, Variância, Desvio Padrão, Máximo, Mínimo

Métrica	T(k)	Jitter(k)
Média	0.010341	0.000341
Variância	0.000001	0.000001
Desvio Padrão	0.000919	0.000919
Máximo	0.051058	0.041058
Mínimo	0.010051	0.000051

1. **Média**:
    - **Tempo (T(k))**: 0.010341 segundos
    - **Jitter (Jitter(k))**: 0.000341 segundos A média de tempo é um valor bastante pequeno, indicando que o sistema realiza as simulações com uma latência muito baixa. O valor de jitter também é pequeno, sugerindo que o sistema tem um bom controle sobre as variações de tempo entre as simulações.

2. **Variância**:
    - **Tempo (T(k))**: 0.000000001
    - **Jitter (Jitter(k))**: 0.000000001 A variância baixa tanto para o tempo quanto para o jitter indica que o sistema apresenta um comportamento bastante estável, sem grandes flutuações nos tempos de execução entre as diferentes simulações.
3. **Desvio Padrão**:
    - **Tempo (T(k))**: 0.000919 segundos
    - **Jitter (Jitter(k))**: 0.000919 segundos O desvio padrão também é pequeno, indicando que a maioria das simulações tem tempos de execução próximos à média, e as variações (jitter) são igualmente controladas.
4. **Máximo**:
    - **Tempo (T(k))**: 0.051058 segundos
    - **Jitter (Jitter(k))**: 0.041058 segundos O valor máximo para o tempo de execução e jitter ainda é consideravelmente pequeno, o que sugere que o sistema, mesmo em piores condições, não apresenta picos extremos de latência.
5. **Mínimo**:
    - **Tempo (T(k))**: 0.010051 segundos
    - **Jitter (Jitter(k))**: 0.000051 segundos O tempo mínimo de execução é muito próximo da média, mostrando que o sistema é eficiente na maioria dos casos. O jitter mínimo é muito pequeno, indicando um controle eficaz sobre as variações.

Análise da escalonabilidade:

- O sistema mostra uma excelente estabilidade, pois as métricas de variância e desvio padrão são extremamente pequenas. Isso indica que, mesmo com o aumento do número de simulações (ou threads), a variabilidade nas medições de tempo e jitter não aumenta significativamente, o que é um bom sinal de que o sistema é escalável.
- A estabilidade no tempo de execução e jitter sugere que o sistema pode ser escalado com um aumento no número de threads ou simulações sem prejudicar o desempenho ou introduzir grande variação no comportamento das simulações.

7. Conclusão

O sistema demonstrou um bom desempenho e uma escalabilidade positiva, já que as variações de tempo e jitter permanecem baixas mesmo quando aumentadas as simulações. A capacidade de manter um desempenho constante, com um desvio padrão e variância baixos, é crucial para sistemas que precisam escalar bem, especialmente em contextos onde múltiplas simulações ou threads são executadas em paralelo. Portanto, o sistema parece ser bem projetado para operar de forma eficiente e escalável, mantendo sua estabilidade mesmo sob carga crescente.