PETEE UFMG

UFMG

Learning.m

Uma apostila de introdução
às ferramentas do Matlab

2020

Programa de Educação Tutorial - Engenharia Elétrica
Universidade Federal de Minas Gerais

Este material é uma reformulação da nossa apostila PETEE UFMG referente ao minicurso Matlab 1.0 de março de 2017. Ela foi reformulada para, sobretudo, fazer da apostila mais objetiva, clara e didática, por meio de melhores detalhamentos e explicações.

É um material de estudos criado com o intuito de auxiliar em um dos minicursos ministrados por nós do Programa de Educação Tutorial – Engenharia Elétrica da Universidade Federal de Minas Gerais: Matlab 1.0.

O nosso objetivo é ajudar a todos que tenham interesse em aprender sobre o software, fazendo da apostila um guia básico para entender melhor o funcionamento do software e suas ferramentas.

É com muito orgulho e esmero que realizamos esta atividade.

Arthur Henrique Dias Nunes

Programa de Educação Tutorial da Engenharia Elétrica - Universidade Federal de Minas Gerais HTTP://WWW.PETEE.CPDEE.UFMG.BR/

PETEE UFMG

/peteeUFMG petee.cpdee.ufmg.br @petee.ufmg

Grupo PETEE

O que é PET?

Os grupos PETs são organizados a partir de formações em nível de graduação nas Instituições de Ensino Superior do país, orientados pelo princípio da indissociabilidade entre **ensino**, **pesquisa** e **extensão** e da educação tutorial.

Por esses três pilares, entende-se por:

- Ensino: As atividades extra-curriculares que compõem o Programa têm como objetivo garantir a formação global do aluno, procurando atender plenamente as necessidades do próprio curso de graduação e/ou ampliar e aprofundar os objetivos e os conteúdos programáticos que integram sua grade curricular.
- Pesquisa: As atividades de pesquisa desenvolvidas pelos petianos têm como objetivo garantir a formação não só teórica, mas também prática, do aluno, de modo a oferecer a oportunidade de aprender novos conteúdos e já se familiarizar com o ambiente de pesquisa científica.
- Extensão: Vivenciar o processo ensino-aprendizagem além dos limites da sala de aula, com a possibilidade de articular a universidade às diversas organizações da sociedade, numa enriquecedora troca de conhecimentos e experiências.

PETEE UFMG

O Programa de Educação Tutorial da Engenharia Elétrica (PETEE) da Universidade Federal de Minas Gerais (UFMG) é um grupo composto por graduandos do curso de Engenharia Elétrica da UFMG e por um docente tutor.

Atualmente, o PETEE realiza atividades como oficinas de robôs seguidores de linha, minicursos de Matlab, minicursos de LaTeX, Competição de Robôs Autônomos (CoRA), escrita de artigos científicos, iniciações científicas, etc.

Assim como outras atividades, o grupo acredita que os minicursos representam a união dos três

pilares: O pilar de ensino, porque ampliam e desenvolvem os conhecimentos dos petianos; o pilar da pesquisa, pois os petianos aprendem novos conteúdos e têm de pesquisar para isso; o pilar da extensão, porque o produto final do minicurso é levar à comunidade os conhecimentos adquiridos em forma de educação tutorial.

Minicurso de Matlab

A programação está muito presente não apenas no cotidiano dos petianos, mas de todos os graduandos em engenharia e também de diversos outros cursos superiores. Visando a demanda dos estudantes em trabalhar com cálculo computacional, o grupo PETEE resolveu oferecer um minicurso de Matlab para os alunos da UFMG.

O Matlab trata-se de um software interativo de alta performance voltado para o cálculo numérico, que integra análise numérica, cálculo com matrizes, processamento de sinais e construção de gráficos. Além disso, com a utilização da ferramenta Simulink é possível modelar, simular e analisar sistemas dinâmicos através de diagramas de blocos.

As inscrições são abertas ao público, e a apostila é disponibilizada gratuitamente no site.

O Grupo

Tutora:
Luciana Pedrosa Salles
Discentes:
Álvaro Rodrigues Araújo
Amanda Andreatta Campolina Moraes
Arthur Henrique Dias Nunes
Diêgo Maradona Gonçalves Dos Santos
Gustavo Alves Dourado
Iago Conceição Gregorio
Isabela Braga da Silva
Israel Filipe Silva Amaral
Italo José Dias
José Vitor Costa Cruz
Lorran Pires Venetillo Dutra
Sarah Carine de Oliveira
Thais Ávila Morato
Tiago Menezes Bonfim
Vinícius Batista Fetter
Willian Braga da Silva

Contato

Site:
http://www.petee.cpdee.ufmg.br/

Facebook:
https://www.facebook.com/peteeUFMG/

Instagram:
https://www.instagram.com/petee.ufmg/

E-mail:
petee.ufmg@gmail.com

Localização:
Universidade Federal de Minas Gerais, Escola de Engenharia, Bloco 3, Sala 1050.

Agradecimentos

Agradecemos ao Ministério da Educação (MEC), através do Programa de Educação Tutorial (PET), Pró-Reitoria de Graduação da Universidade Federal de Minas Gerais (UFMG) e à Escola de Engenharia da UFMG pelo apoio financeiro e fomento desse projeto desenvolvido pelo grupo PET Engenharia Elétrica da UFMG (PETEE - UFMG).

Sumário

I Introdução

1 Apresentação 13
1.1 O Software 13
1.2 História 13

2 Ambiente de Trabalho 15
2.1 Command Window 16
2.2 Command History 17
2.3 Workspace 17
2.4 Variable Editor 19
2.5 Current Folder 20
2.6 Editor 21

II Conceitos Básicos

3 Estrutura de Dados 25
3.1 Variáveis 26
3.2 Arranjos e Matrizes 27
3.3 Tipos Primitivos 29
3.3.1 Inteiro 29
3.3.2 Ponto Flutuante 30
3.3.3 Lógico 30

3.3.4 Caractere . . . 30
3.4 Strings 31
3.5 Simbólicas 33
3.6 Complexos 33
3.7 Structs 34
3.8 Arranjos de Células 35
3.9 Constantes 35
Exercícios 37

4 Comandos e Funções . . . 39
4.1 Atalhos de Sintaxe 40
4.2 Comandos Básicos 40
4.3 Funções Básicas 41
4.3.1 Estrutura de dados . . . 41
4.3.2 Criação e manipulação de arranjos: . . . 42
4.3.3 Impressão e entrada . . . 42
4.4 Funções Simbólicas 43
4.5 Complexos 44
4.6 Randômicas 44
4.7 Trigonométricas 45
4.8 Aritméticas 46
4.9 Numéricas 46
4.10 Gráficos 47
4.11 Polinômios 47
Exercícios 48

5 Operadores . . . 51
5.1 Aritméticos 51
5.2 Lógicos 54
5.3 Relacionais 55
5.4 Precedência 55
Exercícios 56

6 Controle de Fluxo . . . 59
6.1 Estruturas Condicionais 59
6.1.1 if-else . . . 60
6.1.2 switch . . . 60
6.2 Estruturas de Repetição 61
6.2.1 while . . . 61
6.2.2 for . . . 62
6.3 Desvio Incondicional 62
6.3.1 break . . . 62
6.3.2 continue . . . 62

Exercícios 63

7 Scripts 65
7.1 Declaração de Funções 67
7.2 Documentação 69
Exercícios 69

III Conceitos Avançados

8 Gráficos 73
8.1 Bidimensionais 73
8.2 Tridimensionais 79
8.3 Múltiplos Gráficos 84
8.4 Comandos Auxiliares 86
8.5 Outras Plotagens 88
8.5.1 Funções Implícitas 88
8.5.2 Polígonos 91
8.5.3 Campos Vetoriais 91
8.6 Animações 93
Exercícios 94

9 Polinômios 97
9.1 Raízes 97
9.2 Produto e Divisão 98
9.3 Avaliação 99
9.4 Frações Parciais 99
9.5 Ajuste Polinomial 100
Exercícios 100

10 Simulink: Introdução 105
10.1 Ambiente 105
10.2 Modelagem 107
10.2.1 Exemplo 107
10.2.2 Intervalo de Amostragem 111
Exercícios 113

IV Índice

Resoluções 117
Estruturas de Dados 117
Comandos e Funções 120
Operadores 126

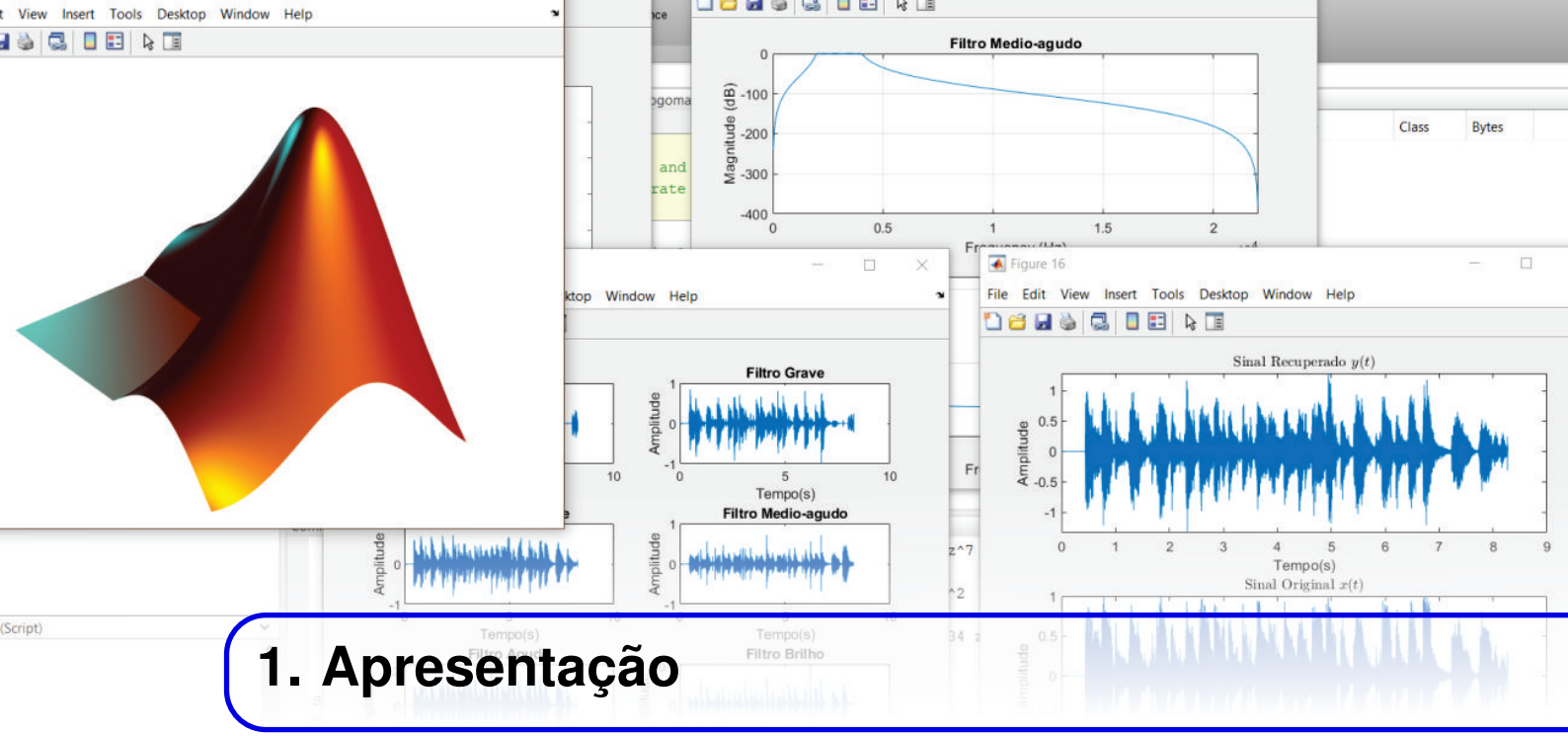
Controle de Fluxo **129**
Scripts **131**
Gráficos **134**
Polinômios **142**
Simulink: Introdução **147**

Bibliografia **155**

Introdução

1 Apresentação 13
1.1 O Software
1.2 História

2 Ambiente de Trabalho 15
2.1 Command Window
2.2 Command History
2.3 Workspace
2.4 Variable Editor
2.5 Current Folder
2.6 Editor



1. Apresentação

1.1 O Software

O nome vem de laboratório de matrizes (Matrix Laboratory), e não de matemática, como alguns pensam. Mesmo com esse nome, suas funcionalidades e utilidades vão muito além disso.

Matlab é um software interativo de alto nível, muito utilizado para desenvolvimento e implementação de algoritmos numéricos e simbólicos. Serve para construção e análise de imagens e gráficos, processamento de sinais e desenvolvimento e simulações de sistemas. Tem capacidade de atender vários tipos de usuários, do básico ao mais avançado.

É integrado ao Simulink, uma ferramenta poderosa de simulação e modelagem de sistemas, capaz de simular desde simples equações diferenciais até complexos sistemas mecânicos, elétricos e de controle.

OBS : Para a confecção desse material usamos a versão R2017b do Matlab.

1.2 História

A primeira versão foi desenvolvida no final dos anos 1970 na Universidade do Novo México com o objetivo de dar acesso a pacotes para Fortran sem requerer conhecimento dessa linguagem. Então, o programa era uma ferramenta bastante primitiva que servia apenas para fazer algumas operações numéricas com matrizes, daí o nome. Não possuía linguagem própria, toolboxes, arquivos ".m", tampouco parte gráfica. O objetivo era apenas auxiliar estudantes em aulas como Cálculo Numérico e Álgebra Linear.

CURIOSIDADE : Naquela época, não havia editor de texto, terminais, monitores ou discos de memória. Os programas eram escritos e lidos em cartões perfurados, nos quais a lógica binária era determinada por ter furo ou não. Após processado, a saída do programa perfurava outro cartão.

Em 1983, um engenheiro percebeu o potencial comercial da ferramenta e juntou-se ao desenvolvedor para reescrever o programa. Nessa época, já estavam aparecendo no mercado os primeiros computadores pessoais (PCs). Em 1984, a equipe reescreveu o programa em C utilizando um PC de 256kB de memória, sem disco rígido e a empresa Mathworks foi criada. A partir de então, o software foi gradativamente ganhando mercado e novas versões foram criadas.

No ano 2000, quando na versão 6, o Matlab foi novamente reescrito para basear-se no LAPACK, um conjunto de bibliotecas para manipulação de matrizes.

2. Ambiente de Trabalho

O ambiente de trabalho Matlab é composto, basicamente, de grandes barras de opções na parte superior e algumas janelas. Cada janela possui sua funcionalidade, que serão explicadas a seguir. O visual padrão é mostrado pela Fig. 2.0.1.

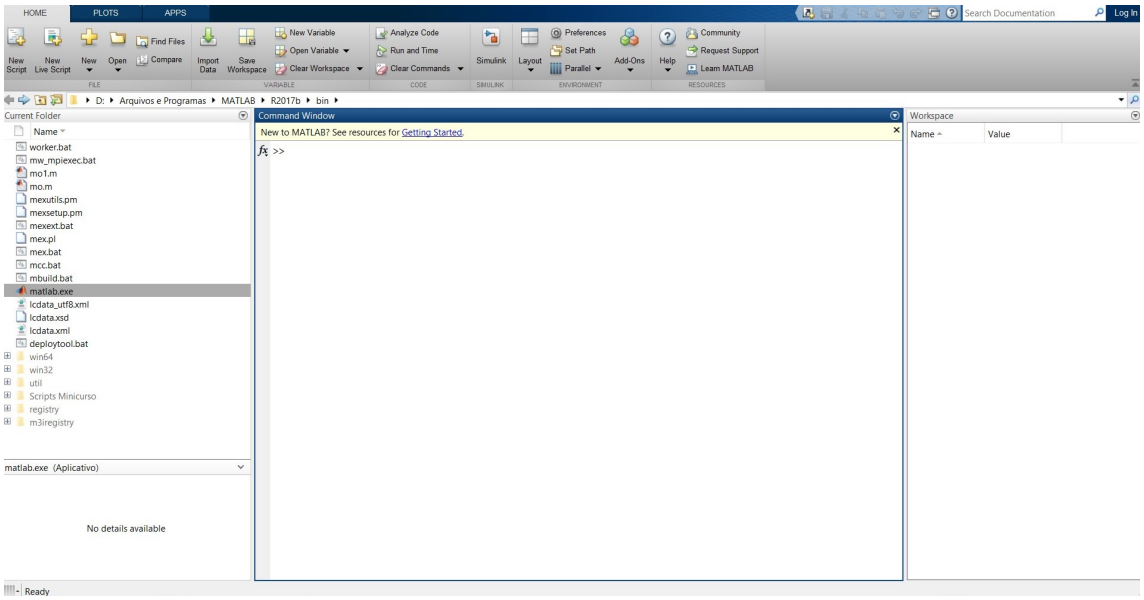


Figura 2.0.1: *Layout default*

Na primeira barra de opções *Home*, localizada na parte superior há a opção *layout* (Fig. 2.0.2), na qual é possível customizar o ambiente de trabalho da melhor forma que o usuário julgar. Também é possível customizar clicando nas janelas e as arrastando para a posição desejada.
Códigos podem ser escritos na Command Window e no Editor. Quando eles forem usados no primeiro, haverá um indicador de >> e as linhas não serão numeradas, enquanto que, quando os códigos forem escritos no Editor, não haverá aquele indicador e as linhas serão numeradas.

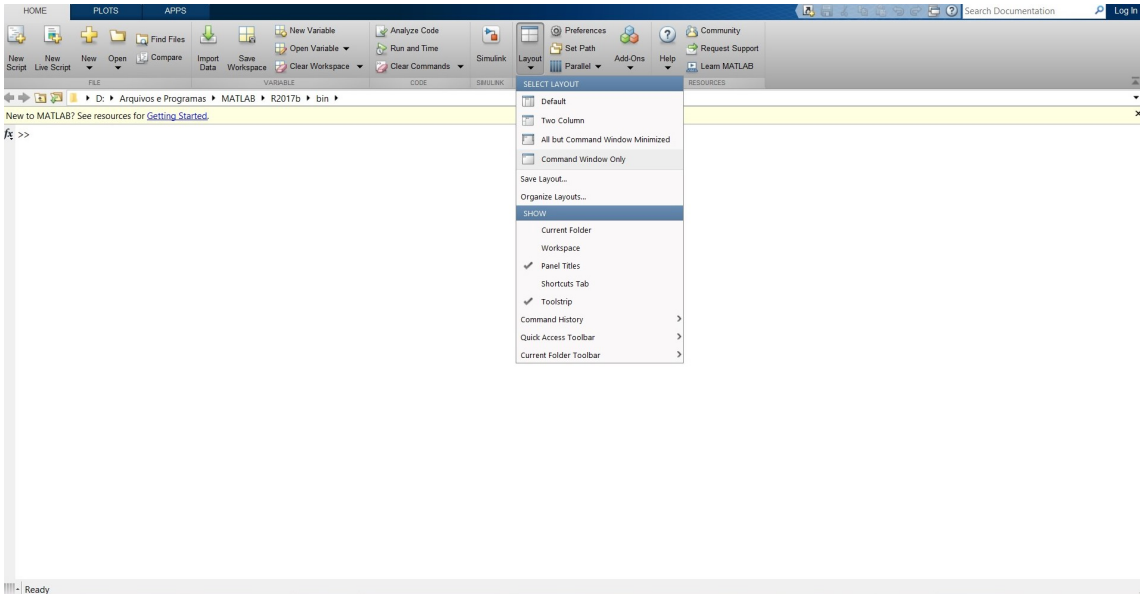


Figura 2.0.2: *Exemplo de customização de layout*

2.1 Command Window

Figura 2.1.1: *Command Window*

A Command Window, ou Janela de Comando, (Fig. 2.1.1) é análoga ao PowerShell ou CMD, e portanto serve para executar os comandos do Matlab. É mais recomendado para aqueles de rápida execução, uma vez que existe o editor de texto para programas e códigos maiores e mais complexos.

Alguns atalhos são úteis ao se usar a Command Window, como:

- ↑ Comando anterior;
- ↓ Comando posterior;
- ← Move para a esquerda;
- → Move para a direita;
- **Ctrl** + ← Move uma palavra para a esquerda;
- **Ctrl** + → Move uma palavra para a direita;
- **Home** Move para o começo da linha;
- **End** Move para o final da linha;
- **Backspace** Apaga um caractere à esquerda;
- **Del** Apaga um caractere à direita;
- **Shift** + **Enter** Pula para a próxima linha sem executar o comando das linhas anteriores;

2.2 Command History

O Command History, ou histórico de comandos, (Fig. 2.2.1) aparece quando se pressiona a seta para cima na Command Window, sendo salvo mesmo que se feche o programa ou desligue o computador.

Figura 2.2.1: *Command Window*

2.3 Workspace

O Workspace (Fig. 2.3.1) mostra as variáveis e as estruturas existentes na memória. Clicando duas vezes em cima de uma variável é possível editá-la, pois abrirá o **Variable Editor**, mostrado em sequência. Clicando com o botão direito em cima de sua barra, é possível configurá-lo (Fig 2.3.2).

Figura 2.3.1: *Workspace*

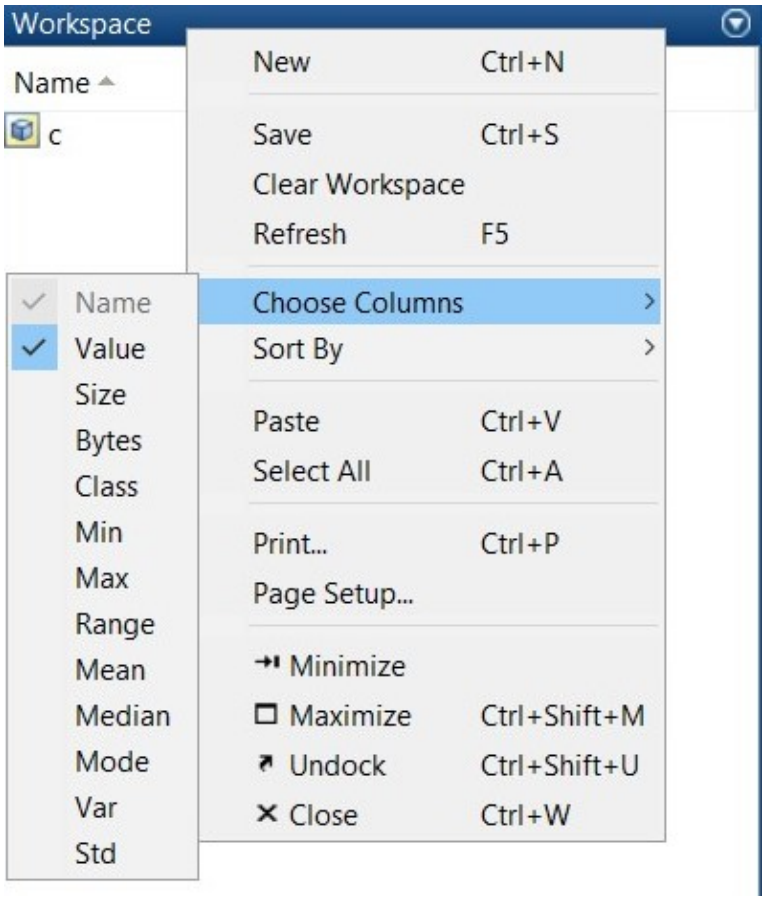


Figura 2.3.2: *Configurando o Workspace*

2.4 Variable Editor

A figura 2.4.1 é a janela aberta ao se clicar duas vezes em alguma variável do Workspace, conforme descrito na seção anterior. Aqui é possível editar e visualizar melhor a variável por meio de planilhas, ficando mais fácil compreender arranjos, vetores e até estruturas.

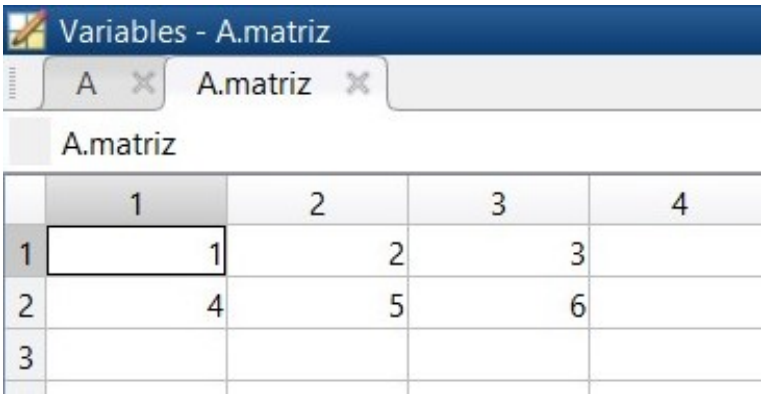


Figura 2.4.1: *Variable Editor*

2.5 Current Folder

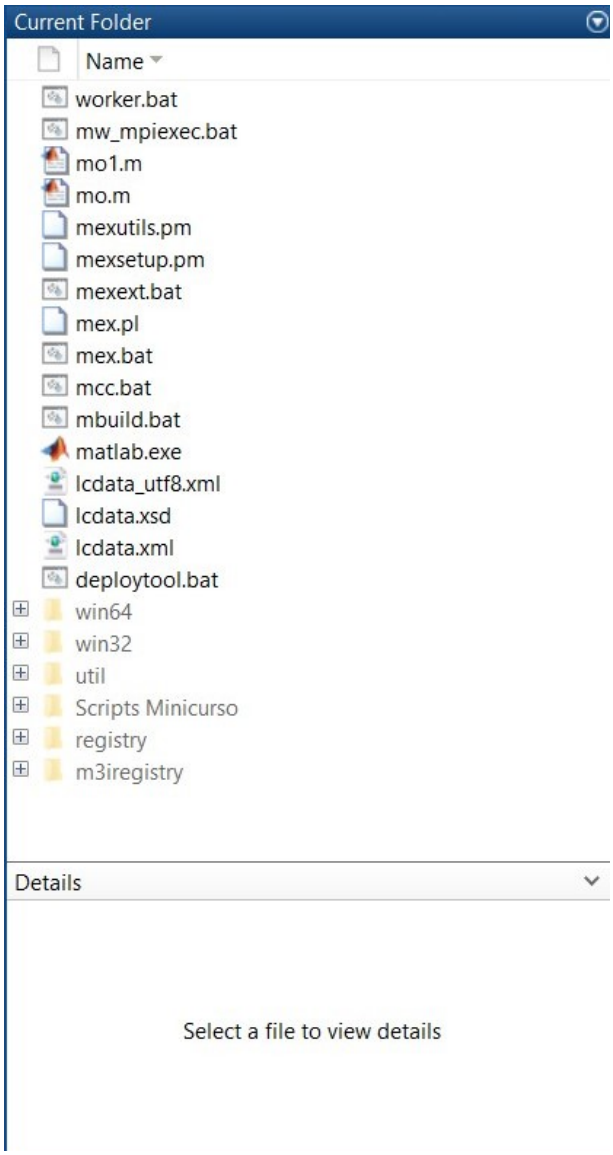


Figura 2.5.1: *Current Folder*

Current Folder (Fig. 2.5.1) é a janela que mostra o diretório corrente e os seus arquivos. É possível navegar nas pastas por esta janela, ou também usando os seguintes comandos na Command Window:

```
>> cd <diretório> % navega até o diretório
>> cd ..          % volta uma pasta
>> dir            % lista os arquivos
>> pwd            % mostra o caminho até o local corrente
```

O diretório corrente será importante na criação de scripts, arquivos *.m*, declaração de funções e na leitura e escrita de arquivos.

DICA: Toda pasta possui dois arquivos: O arquivo "."e o arquivo "..". O primeiro deles é responsável por conter o endereço do diretório corrente, enquanto que o segundo contém o endereço para o diretório que abriga o diretório atual. Por esta razão, o comando "cd .."volta um diretório.

2.6 Editor

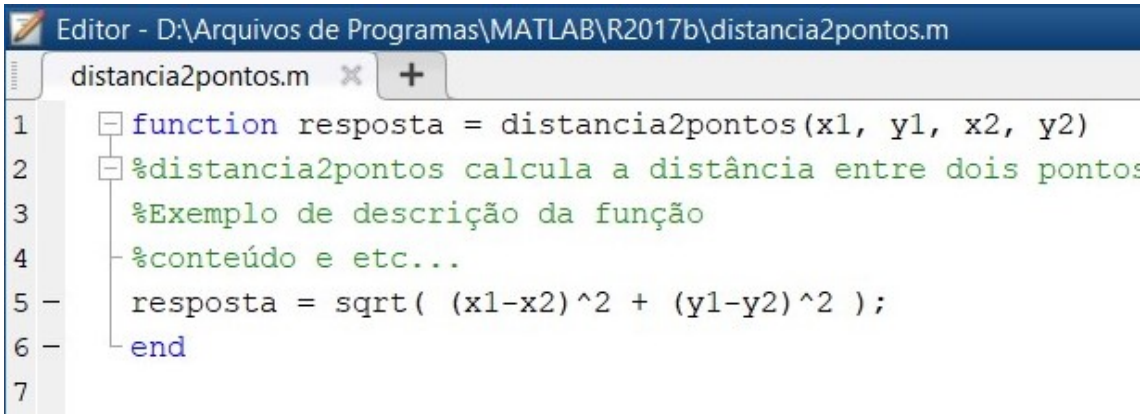


Figura 2.6.1: *Editor*

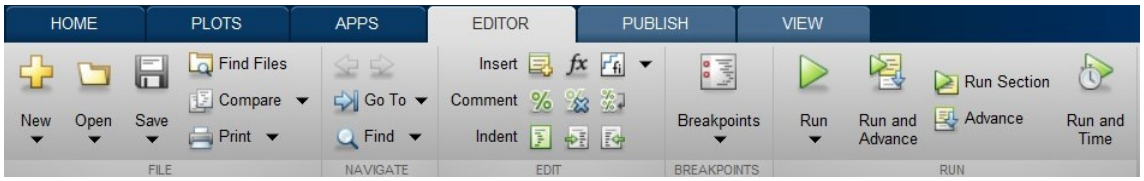


Figura 2.6.2: *Barra de tarefas do Editor*

Este é o editor de texto do Matlab (Fig. 2.6.1) que possui uma barra de tarefas (Fig. 2.6.2) com opções auxiliares. Pode ser aberto ao se clicar em “New Script” ou ao abrir arquivos .m. Nele é possível criar e executar algoritmos, criar arquivos de funções declaradas e edições de roteiros (scripts) em geral. Será abordado de forma mais detalhada no capítulo destinado a scripts e arquivos .m (Cáp. 7).

DICA: Outra forma de abrir o editor e um arquivo é digitando o comando: "edit arquivo.m"

II Conceitos Básicos

3 Estrutura de Dados 25
3.1 Variáveis
3.2 Arranjos e Matrizes
3.3 Tipos Primitivos
3.4 Strings
3.5 Simbólicas
3.6 Complexos
3.7 Structs
3.8 Arranjos de Células
3.9 Constantes
Exercícios

4 Comandos e Funções 39
4.1 Atalhos de Sintaxe
4.2 Comandos Básicos
4.3 Funções Básicas
4.4 Funções Simbólicas
4.5 Complexos
4.6 Randômicas
4.7 Trigonométricas
4.8 Aritméticas
4.9 Numéricas
4.10 Gráficos
4.11 Polinômios
Exercícios

5 Operadores 51
5.1 Aritméticos
5.2 Lógicos
5.3 Relacionais
5.4 Precedência
Exercícios

6 Controle de Fluxo 59
6.1 Estruturas Condicionais
6.2 Estruturas de Repetição
6.3 Desvio Incondicional
Exercícios

7 Scripts 65
7.1 Declaração de Funções
7.2 Documentação
Exercícios

3. Estrutura de Dados

Em algoritmos de computação é necessário armazenar informações diversas. Essas informações são traduzidas para dados de diferentes tipos e armazenadas na memória. Cada bloco de memória que armazena um dado, recebe uma etiqueta, como um nome, e essas estruturas são chamadas de **variáveis** do programa. Isto é, nomes de controle que existirão no código para armazenar dados.

A esse ramo da programação é dado o nome de estrutura de dados, que será introduzido neste capítulo.

Para o armazenamento, as informações devem ser comportadas em longas cadeias de dígitos binários, chamados de **bits**. Isto é, longas sequências de zeros e uns. Esses dados podem ser interpretados de diferentes maneiras, dependendo do seu uso e das operações a serem realizadas com eles.

Seu armazenamento se dá em variáveis, que são espaços reservados na memória para fácil acesso do usuário. No Matlab, todas as variáveis e constantes são arranjos, com uma ou mais dimensões. Até as mais simples, com apenas um elemento, são tratadas como arranjos 1x1. Com isso, pode-se fazer também strings, structs, células, containers e sets, conceitos familiares para quem conhece outras linguagens de programação.

Sua estrutura de dados foi construída de modo a obter alta eficiência em trabalhar com operações matriciais diretamente, sendo mais eficiente que trabalhar com laços iterativos.

CURIOSIDADE: O termo *bit* é uma sigla para se referir a dígito binário, do inglês, *B*inary dig*IT*.
DICA: Um conjunto de 8 bits é chamado de byte.

3.1 Variáveis

Diferente de outras linguagens como C e Java, no Matlab não é necessário declarar o tipo da variável, basta atribuir seu valor em tempo real. Essa característica é chamada de **tipagem dinâmica**, e também está presente em outras linguagens, como Python.

```
>> a = 'a';      % a é uma variável do tipo char
>> a = 2.2;      % a é uma variável do tipo double
```

A nomenclatura das variáveis podem ter números, letras e '_', sendo obrigatoriamente primeiro uma letra e no máximo 31 caracteres. Ainda, o software é **case sensitive**, isto é, o Matlab diferencia letras maiúsculas de minúsculas, sendo "A"uma variável diferente de "a".

Além disso, seus valores numéricos podem ser especificados com uma sequência numérica precedida ou não de "+" ou "-". O ponto "." indica decimais. Potências de 10 podem ser escritas usando e<expoente> ou d<expoente>, como por exemplo: 5.331e15, -2.2d-2.

Existem diferentes tipos de dados para representação e armazenamento de variáveis. Também podem ser convertidos em outros tipos de dados, usando o nome do tipo desejado e a variável como parâmetro. Exemplo:

```
>> x = 1.23 % por padrão, x será double
x =
    1.2300
>> a = int8(x)
a =
  int8
   1
```

O comando *class(x)* retorna o tipo de dado de x em formato de string. Alguns tipos de dados são mostrados na tabela 3.1.1

Tipo	Bytes	Descrição
double	8	Tipo padrão de armazenamento com precisão dupla
single	4	Armazenamento com metade da precisão double
int8	1	Inteiro com sinal, de 8 bits
int16	2	Inteiro com sinal, de 16 bits
int32	4	Inteiro com sinal, de 32 bits
int64	8	Inteiro com sinal, de 64 bits
uint8	1	Inteiro sem sinal, de 8 bits
uint16	2	Inteiro sem sinal, de 16 bits
unt32	4	Inteiro sem sinal, de 32 bits
uint64	8	Inteiro sem sinal, de 64 bits
logical	1	Tipo de dado booleano
sym	8	Tipo de dado simbólico
char	2	Caractere único

Tabela 3.1.1: *Palavras Reservadas*

O software também pode ser usado como uma calculadora científica, e portanto, possui uma variável de controle chamada **ans**. O nome vem de *answer*, e serve para armazenar a resposta de alguma operação, caso ela não seja atribuída a nenhuma outra variável. Mas diferente das calculadoras científicas, como a HP50G, ans armazena somente a última resposta. Exemplo:

```
>> 1+1
ans=
      2
>> ans + 5
ans=
      7
```

3.2 Arranjos e Matrizes

Um arranjo, como já discutido anteriormente, é como o Matlab interpreta todos os tipos de dados. Trata-se de um grupo de valores na memória. Um único valor constitui um arranjo de tamanho 1x1. Aquelas de tamanho mx1 ou 1xn podem ser chamados de **vetores** ou arranjos simples. Já os de tamanho mxn são chamados também de matrizes. Há ainda arranjos de múltiplas dimensões, por exemplo, uma imagem RGB de 100px x 100px pode ser interpretada como um arranjo 100x100x3.

Pode-se atribuir um arranjo de várias formas, utilizando colchetes e especificando elementos, utilizando dois pontos, usando a função **cat**, etc. Por exemplo, para criar um arranjo B que vai de 2 a 10 de 0.5 em 0.5(usando a notação de ":"), um arranjo C contendo 5 elementos igualmente espaçados entre 1 e 3(usando a função **linspace**) e um arranjo A contendo elementos especificados, pode-se usar três sintaxes distintas.

A primeira é a notação de dois pontos, onde representa intervalo. Se usado somente 2:10 para B, B seria um arranjo de 2 a 10, com seus itens variando de 1 em 1; como será usado 2:0.5:10, isso determina que B será ainda de 2 a 10, porém o passo, isto é, o intervalo entre um elemento e o próximo será de 0.5. A segunda, é utilizando a função **linspace**. Essa função cria um arranjo igualmente espaçado, recebendo três parâmetros, respectivamente, o primeiro valor, o último valor e a quantidade de elementos, muito útil na criação de arranjos para a construção de gráficos:

```
>> B= 2:0.5:10;
>> C=linspace(1, 3, 5);
>> A=[1, 2, 3; 4, 5, 6];
```

Neste último caso, por exemplo, o Matlab já computa o espaço de memória necessário para uma matriz 2x3 chamada A e já são atribuídos os valores. Se for executado o mesmo comando, porém sem o ponto e vírgula, será exposto o resultado da operação, que é a atribuição da variável:

```
>> A=[1, 2, 3; 4, 5, 6]
A=
     1 2 3
     4 5 6
```

DICA: Sendo assim, o operador ";"no final de uma linha de comando suprime a exibição do resultado.

Sendo assim, para se criar um vetor linha, além das notações ":"e linspace, pode-se declarar os elementos diretamente. Um vetor linha pode ser criado como:

```
>> V = [1, 2, 3];
```

Ao invés de vírgulas, pode-se usar também apenas espaços em branco. O Matlab interpretará que será o próximo elemento da linha

```
>> V = [1 2 3];
```

Para se criar um vetor coluna:

```
>> W = [1; 2; 3];
```

Ao invés de ponto e vírgula pode-se usar enter, no caso, shift+enter para poder saltar de linha sem executar o comando:

```
>> W [1
2
3];
```

Também é possível atualizar o tamanho, bem como os valores do arranjo.

Os exemplos a seguir utilizam a mesma matriz A do exemplo anterior.

- Atualizando o valor de um elemento:

```
>> A(1,1)=0
A =
     0     2     3
     4     5     6
```

- Expandindo o arranjo, nota-se que os outros elementos serão zero:

```
>> A(3,2)=1
A =
     0     2     3
     4     5     6
     0     1     0
```

- Recortando a matriz:

```
>> A(:,2)=[]
A =
      0      3
      4      6
      0      0
```

Como mostrado acima, os itens dos arranjos podem ser fatiados e selecionados. Em um vetor V, por exemplo, a notação V(2) indica o segundo elemento de V.

DICA: Diferente do padrão de linguagens de programação, no Matlab a indexação começa de 1 e não de 0.

Já em matrizes, A(1,1) indica o elemento da primeira linha e da primeira coluna de A. De maneira genérica, A(m,n) é o elemento da linha m e coluna n de A. No entanto, também pode-se fazer A(3), por exemplo, o que indicaria o terceiro elemento da matriz. Nesses casos, a matriz é percorrida coluna a coluna.

É possível ainda selecionar um subconjunto da matriz, como A(1:2, 2:4), estão sendo selecionados os elementos que estão nas linhas 1 até 2 e colunas 2 até 4. Esses recortes podem também ser feito saltando-se elementos, muito similar à notação de ":"na criação de vetores. Nestes casos, se usado A(:, 2), somente os ":"no lugar de se especificar a linha, indica que todas as linhas serão selecionadas.

DICA: A notação "[]"indica conjunto vazio. Sendo assim, ao se atribuir toda uma linha(ou coluna) de uma matriz como "[]"é o mesmo que remover aquela linha(ou coluna).

Ainda existem outras formas de se criar matrizes e arranjos, como o exemplo de criação de matrizes por concatenação direta na Fig. 3.2.1.

3.3 Tipos Primitivos

Para entender os tipos de dados em linguagens de programação, no geral, os primeiros tipos a serem entendidos são os tipos primitivos de dados, pois com eles se constroem e entendem os outros. Mas neste caso, faz mais sentido entender os arranjos e matrizes primeiro, já que são a base da interpretação de dados do Matlab.

Os tipos de dados dizem respeito a forma com que as cadeias de 0s e 1s serão interpretados. Existem quatro tipos básicos comuns para a maioria das linguagens, chamados tipos primitivos.

3.3.1 Inteiro

O primeiro tipo é o tipo inteiro. Nesse tipo os dados são interpretados a fim de determinar o sinal e o valor, assumindo valores negativos, positivos e o zero, exatamente como no conjuntos dos números inteiros. No Matlab, há os tipos int8, int16, int32 e int64 para representá-los.

Em algumas linguagens há o tipo de inteiro *unsigned*, isto é, sem sinal, no qual só é possível armazenar valores não negativos. Com isso, o maior valor possível de se representar é o dobro+1

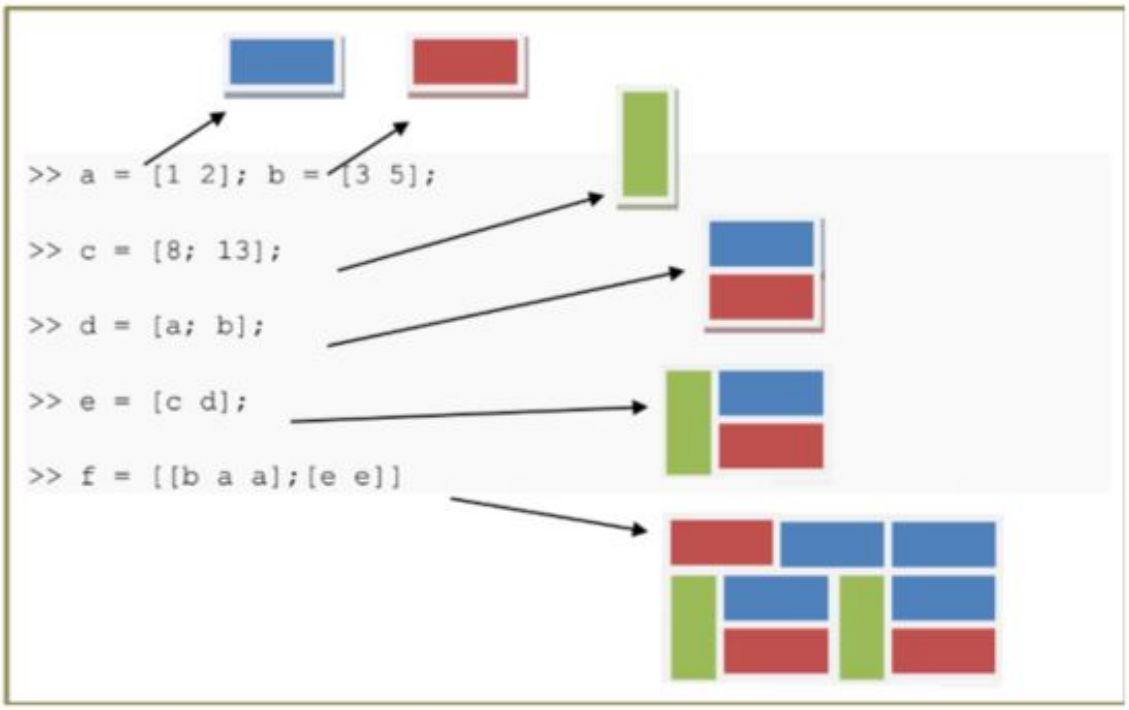


Figura 3.2.1: *Criação de matrizes por concatenação direta*

do tipo inteiro padrão, e no Matlab, eles são chamados de uint8, uint16, uint32 e uint64.

3.3.2 Ponto Flutuante

O tipo ponto flutuante é responsável por representar o conjunto dos números reais. Pode também representar inteiros, mas não tantos quanto o próprio tipo reservado para eles, pois a capacidade de armazenamento deste tipo está direcionada para poder suportar valores decimais.

Neste tipo, é possível representar valores como 2.5, -0.3333, etc.

No Matlab, eles são os tipos single e double, sendo o double o tipo padrão para valores numéricos.

3.3.3 Lógico

Já o tipo lógico armazena os valores **booleanos**, ou seja, verdadeiro ou falso, 0 ou 1, sendo representados pelo tipo de dado logical.

3.3.4 Caractere

O tipo caractere é responsável por armazenar valores de letras e outros caracteres, sendo chamados de char no Matlab. Geralmente, os caracteres são codificados em valores seguindo um padrão universal, conhecido como Tabela ASCII.

3.4 Strings

Strings são cadeias lineares de caracteres, e portanto, também são vetores. São criados utilizando aspas simples. Não são tratados como números, no entanto, ainda possuem várias funções e operações para as strings. Exemplos de manipulação de strings:

```
>> Nome = 'John'
Nome=
        'John'
>>Nome(1) = 'L'
Nome=
    'Lohn'
>> Nome(2) = 'a' +  4
Nome=
    'Lehn'
```

Podem também ser feitas matrizes e arranjos de n dimensões, normalmente:

```
>> Texto = ['SATOR';'AREPO';'TENET';'OPERA';'ROTAS']
Texto =
  5x5 char array
    'SATOR'
    'AREPO'
    'TENET'
    'OPERA'
    'ROTAS'
```

Cada caractere possui uma representação numérica, 'a', por exemplo, tem a representação 97. Logo, ao se fazer 'a' + 4, o resultado será o caractere que tem a representação 101.

```
>> double('a')

ans =

    97

>> 'a' + 4

ans =

   101

>> char(ans)

ans =

    'e'
```

A Fig. 3.4.1 mostra algumas representações numéricas de caracteres.

representação decimal dos caracteres															
33	!	45	-	57	9	69	E	81	Q	93	]	105	i	117	u
34	"	46	.	58	:	70	F	82	R	94	^	106	j	118	v
35	#	47	/	59	;	71	G	83	S	95	_	107	k	119	w
36	$	48	0	60	<	72	H	84	T	96	‘	108	l	120	x
37	%	49	1	61	=	73	I	85	U	97	a	109	m	121	y
38	&	50	2	62	>	74	J	86	V	98	b	110	n	122	z
39	’	51	3	63	?	75	K	87	W	99	c	111	o	123	{
40	(	52	4	64	@	76	L	88	X	100	d	112	p	124	|
41	)	53	5	65	A	77	M	89	Y	101	e	113	q	125	}
42	*	54	6	66	B	78	N	90	Z	102	f	114	r	126	~
43	+	55	7	67	C	79	O	91	[	103	g	115	s		
44	,	56	8	68	D	80	P	92	\	104	h	116	t		

representação decimal dos caracteres															
192	À	200	È	208	Ð	216	Ø	224	à	232	è	240	ð	248	ø
193	Á	201	É	209	Ñ	217	Ù	225	á	233	é	241	ñ	249	ù
194	Â	202	Ê	210	Ò	218	Ú	226	â	234	ê	242	ò	250	ú
195	Ã	203	Ë	211	Ó	219	Û	227	ã	235	ë	243	ó	251	û
196	Ä	204	Ì	212	Ô	220	Ü	228	ä	236	ì	244	ô	252	ü
197	Å	205	Í	213	Õ	221	Ý	229	å	237	í	245	õ	253	ý
198	Æ	206	Î	214	Ö	222		230	æ	238	î	246	ö	254	
199	Ç	207	Ï	215	×	223	ß	231	ç	239	ï	247	÷	255	ÿ

Figura 3.4.1: *Algumas representações numéricas de caracteres*

DICA: Aspas duplas também podem ser usadas para definir strings e caracteres no Matlab, mas geralmente são usadas aspas simples como boa prática de programação.

DICA: Note que é possível usar uma para escrever a outra como caractere: ' "' ou "' ".

DICA: Deve-se ter atenção para essa sintaxe, porque apesar de para o Matlab aspas simples ou duplas terem o mesmo efeito, em outras linguagens elas podem ser fundamentalmente diferentes. Em C, por exemplo, as duplas definem arranjos de caracteres, enquanto que as simples definem caracteres únicos. Em PHP, as duplas definem tipos dinâmicos e as simples tipos estáticos.

3.5 Simbólicas

Existem também as variáveis do tipo simbólicas. Com esse tipo de dado, é possível fazer funções e expressões matemáticas, avaliar a função em um ponto, derivar, dentre outros artifícios. Exemplo:

```
>> syms x
>> y = 2*x + 4;
>> subs(y, x, -2)
    ans =
        0
```

Há diversas funções para variáveis simbólicas, outro exemplo: simplificar a expressão $sen^2(x) + cos^2(x)$, que será igual a 1.

```
>> syms x
>> f = sin(x)^2 + cos(x)^2
f =
cos(x)^2 + sin(x)^2
>> simplify(f)
ans =
1
```

3.6 Complexos

Números complexos são aqueles que constituem os conjuntos dos números reais e imaginários, podem ser representados por $a + bi$, sendo $i = \sqrt{-1}$.

O Matlab realiza operações com os complexos normalmente. O tipo de dado utilizado é diferente do double comum de 8 bytes. É utilizado o **double(complex)** de 16 bytes, e é possível imaginá-lo como um campo double para a parte real e outro para a parte imaginária. O exemplo a seguir computa a raiz quadrada de -1, usando a função sqrt:

```
>> sqrt(-1)
ans =
```

```
   0.0000 + 1.0000i
>> x=ans^2
x =
    -1
```

Existem duas constantes pré-definidas, i e j, as quais possuem o mesmo valor de ans no exemplo anterior, 0 + 1*i. Sendo assim, é possível fazer todos os procedimentos normalmente com os complexos, criar matrizes, arranjos, operações e etc.

DICA: Ao se tratar de arranjos de complexos, deve-se lembrar que o operador aspas simples representa o complexo conjugado e também transposição de matrizes. Então, se há uma matriz de complexos e for usado o operador aspas simples, cada complexo é conjugado e a matriz é transposta ao mesmo tempo.

```
>> A = [i, 2*i, 3+i];
>> A'
ans =
   0.0000 - 1.0000i
   0.0000 - 2.0000i
   3.0000 - 1.0000i
```

3.7 Structs

Uma struct é uma estrutura que pode conter diferentes tipos de dados, organizando-os em um único bloco, em que cada campo deste bloco é identificado por um nome.

A construção pode ser feita por atribuição direta, usando a tipagem dinâmica:

```
>> patient.name = 'John Doe';
>> patient.billing = 127.00;
>> patient.test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];
```

Como também utilizando a sintaxe:

```
>> patient = struct('name',['John Doe'], 'billing', [127.00],
   'test', [79, 75, 73; 180, 178, 177.5; 220, 210, 205]);
```

Ao se digitar patient, todos os campos são expostos;

```
>> patient
patient =
  struct with fields:
       name: 'John Doe'
       billing: 127
       test: [3x3 double]
```

3.8 Arranjos de Células

Os arranjos de células são similares aos arranjos comuns. No entanto, diferente deles, cada elemento pode ser de qualquer tipo de dado, armazenar qualquer valor e ser de qualquer tamanho.
Esse tipo pode ser entendido como um arranjo de ponteiros, no qual cada elemento possui um endereço de memória. Nesse endereço de memória poderá estar uma variável de qualquer tipo.
São usadas chaves como sintaxe desse tipo de dado:

```
>> CA(1,1) = {ones(3)};
>> CA(1,2) = {'cell array'};
>> CA(2,1) = {[2]};
>> CA(2,2) = {[i]};
>> CA
CA =
  2x2 cell array
    {3x3 double}    {'cell array'       }
    {[        2]}    {[0.0000 + 1.0000i]}.
```

Para acessar o conteúdo apontado, basta identificar o conteúdo desejado dentro das chaves.

```
>> CA{1,1}
ans =
     1     1     1
     1     1     1
     1     1     1
```

Os arranjos de células podem ser úteis, por exemplo, para se armazenar vetores de diferentes tamanhos, já que isso não poderia ser acomodado em uma matriz.

3.9 Constantes

Também existem constantes predefinidas, usadas por funções e também pelo usuário. Podem ser alteradas, mas não é recomendado. Algumas são mostradas pela tabela 3.9.1

OBS: Alguns dos itens da tabela 3.9.1 são funções, ao invés de constantes.

Tabela 3.9.1: *Algumas Constantes*

Constante	Valor	Tamanho	Bytes	Tipo	Descrição
i	0+1i	1x1	16	double(complex)	$\sqrt{(-1)}$
j	0+1i	1x1	16	double(complex)	$\sqrt{(-1)}$
eps	2.22e-16	1x1	8	double	Menor valor ao somar a 1 para que se torne o próximo ponto flutuante maior que 1
realmin	2.23e-308	1x1	8	double	Menor valor de ponto flutuante
realmax	1.8e+308	1x1	8	double	Maior valor de ponto flutuante
NaN	NaN	1x1	8	double	Valor não numérico (Not a Number)
pi	3.1416	1x1	8	double	Valor de pi
inf	inf	1x1	8	double	Representação de infinito
true	1	1x1	1	logical	Booleano verdadeiro
false	0	1x1	1	logical	Booleano falso
eye(m)	mxm double	mxm	$8m^2$	double	Matriz identidade
ones(m,n)	mxn double	mxn	$8mn$	double	Matriz de 1s
zeros(m,n)	mxn double	mxn	$8mn$	double	Matriz de 0s

Exercícios

3.1 Utilizando o Matlab como uma simples calculadora, compute:
a) 5.17e-5 x 17;
b) 13.7112^2;
c) (resultado anterior) ÷ 5.17e-5;
d) -1 ÷ -1.6022e-19;
e) 50 ÷ 342,3;

3.2 Use novamente o Matlab como uma calculadora e peça para que ele calcule 0 ÷ 0 e depois 10 ÷ 0.

3.3 Utilizando a Command Window, atribua as seguintes variáveis:
a) Um escalar chamado NmrMatricula com valor de 201900;
b) Uma matriz linha (vetor) de tamanho 1x5, chamada Notas, com todos os valores iguais a 0;
c) Uma string chamada Ocorrencias preenchida com: O aluno não possui ocorrências;
d) Um arranjo 2x1 de struct, chamado Aluno, com cada struct com seus campos valendo:
i) Índice 1,1 campo nome = Jonatan
ii) Índice 1,1 campo matricula = NmrMatricula (letra a)
iii) Índice 1,1 campo notas = Notas (letra b)
iv) Índice 1,1 campo ocor = Ocorrencias (letra c)
v) Índice 2,1 campo nome = Patricia
vi) Índice 2,1 campo matricula = NmrMatricula (letra a)
vii) Índice 2,1 campo notas = Notas (letra b)
viii) Índice 2,1 campo ocor = Ocorrencias (letra c)

3.4 Utilizando o arranjo de struct da questão anterior, na aba Workspace, clique duas vezes no arranjo e observe a variável na janela aberta Variable Editor.

3.5 Ainda utilizando o arranjo de struct da questão 3, atualize os valores de seus campos para:
a) Campo matricula de Jonatan = NmrMatricula + 1.
b) Campo notas de Jonatan = 10 8 9.5 7 10.
c) Campo matricula de Patricia = NmrMatricula + 2.
d) Campo notas de Patricia = 10 10 3 9 7.8.
e) Campo ocor de Patricia = duas brigas em sala de aula.

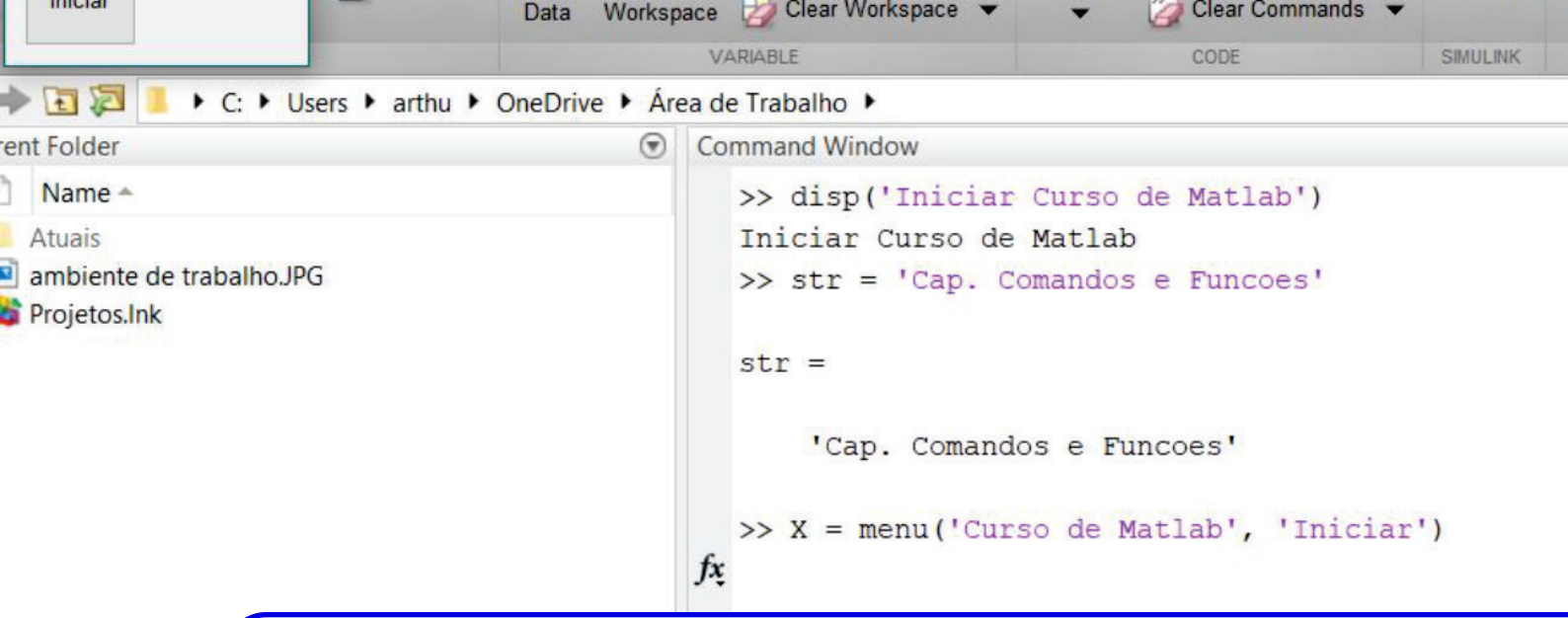
3.6 Repita o procedimento da questão 4) e observe o que mudou.

3.7 7) Crie uma matriz A e B de modo que:
a) A = 2+2i, 0, 7i, 13+2i.
b) B de modo que bij seja igual ao conjugado de aij, utilizando o operador aspas simples. Note que não é para que a matriz A seja transposta, mesmo utilizando o operador aspas simples.

3.8 Utilizando a função subs, **subs(Expressão, variável, valor)**, crie uma expressão simbólica y = 2x2 - 30x + 100 e substitua nessa expressão x por 10.

3.9 Utilizando a mesma função e a mesma expressão anterior, substitua agora x por 5.

3.10 Utilizando ainda a mesma função e a mesma expressão anterior, substitua agora x por 0.



4. Comandos e Funções

Entendido as estruturas de dados, é necessário entender os comandos e funções para melhor trabalhar com elas. Este capítulo apresentará os comandos e funções.

Basicamente, comandos são palavras chave que realizam ações predeterminadas. Exemplo: o comando **clear** remove as variáveis do Workspace. Já as funções são procedimentos que retornam dados, dependendo dos parâmetros passados.

$$[Variaveisretornadas] = Funcao([Parametros])$$

Essas funções são similares a funções matemáticas, isto é, dependendo do argumento, o resultado é diferente. O uso de funções em algum momento no script ou na Command Window recebe o nome de **chamada**. Quando uma função é chamada, ela pode receber **argumentos**, que são valores passados como **parâmetros** dentro de parênteses. As respostas da função, quando há, são chamadas de **retornos**.

Por vezes, comandos são tratados como funções e vice-versa; suas definições e utilização são similares e se misturam. O importante é saber encontrar e entender o procedimento adequado. Ambos podem ser executados tanto na Command Window quanto em um editor de texto. Um dos mais importantes é o comando **help**, com o qual o usuário pode se informar melhor sobre qualquer comando ou função que necessitar. Ele será explicado na sequência.

O comando é muito útil e completo, com explicações, referências, links e exemplos. Basta digitar na Command Window "help <expressão>". A expressão geralmente é uma palavra chave, ou comando/função que se deseja conhecer mais. Vale a pena usá-lo.

Há também o comando lookfor <palavra chave>, que procura por funções que possuem a palavra chave em sua descrição.

No capítulo destinado à criação de funções, será mostrado que também é possível colocar descrições nas funções criadas pelos usuários, que aparecerão nos comandos help e lookfor normalmente.

4.1 Atalhos de Sintaxe

- **;** no final da linha, suprime a exibição do resultado da operação;
- **%** indica comentários;
- **...** continua o comando na próxima linha;
- **,** separa comandos na mesma linha;

4.2 Comandos Básicos

- **help** exibe um texto de ajuda com links para mais aprofundamento;
- **help <função ou comando>** exibe um texto explicando a função ou comando com exemplos e outros tópicos relacionados;
- **help help** abre a documentação;
- **doc <função ou comando>** abre a documentação da função ou comando;
- **clear <variável1> <variável2> <...>** apaga as variáveis especificadas;
- **clear** remove todas as variáveis;
- **clearvariables** corresponde ao clear;
- **clear global** remove todas as variáveis globais;
- **clearfunctions** apaga as funções compiladas de M e MEX;
- **clearall** libera toda a memória apagando variáveis globais e locais e funções;
- **clc** limpa a Command Window;
- **close** fecha todas as figuras abertas;
- **who** lista todas as variáveis;
- **whos** lista todas as variáveis exibindo detalhes;
- **format** muda a formatação numérica das variáveis;
- **cd <diretório>** vai para o diretório;
- **cd ..** vai para o diretório anterior;

- **dir** lista os arquivos no diretório corrente;
- **pwd** exibe o caminho até o diretório corrente;
- **<Variável>** exibe o valor da variável selecionada;

4.3 Funções Básicas

4.3.1 Estrutura de dados

- **X = menu('Título', 'Opção1', 'Opção2', ...)** é aberta uma caixa de diálogo para o usuário escolher entre as opções. Se a opção n é escolhida, X recebe o valor n. Se nenhuma opção é escolhida, X será 0;
- **class(X)** retorna a classe de X em formato string;
- **double(X)** converte X para o tipo double;
- **single(X)** converte X para o tipo single;
- **int8(X)** converte X para o tipo int8;
- **int16(X)** converte X para o tipo int16;
- **int32(X)** converte X para o tipo int32;
- **int64(X)** converte X para o tipo int64;
- **uint8(X)** converte X para o tipo uint8;
- **uint16(X)** converte X para o tipo uint16;
- **uint32(X)** converte X para o tipo uint32;
- **uint64(X)** converte X para o tipo uint64;
- **logical(X)** converte X para o tipo logical;
- **char(X)** converte X para o tipo char;
- **save('Nome.mat', 'Variável1', 'Variável2', ...)** salva as variáveis atuais em um arquivo com o nome especificado, com a extensão .mat. Obs: todos os argumentos devem ser strings;
- **load('Nome.mat')** carrega o workspace salvo com o nome "nome.mat"). Caso salvo em formato padrão (.mat) pode ser atribuído à uma variável. Esta será uma struct contendo as variáveis do workspace salvas como campos. Exemplo:

```
>> a=2; b=3;
>> save('var1', 'a', 'b')
>> clear, clc
```

```
>> x = load('var1.mat')
x =
  struct with fields:
    a: 2
    b: 3
```

4.3.2 Criação e manipulação de arranjos:

- **linspace(Primeiro valor, Último valor, Quantidade de elementos)** cria um arranjo com os elementos igualmente espaçados do primeiro ao último valor. Útil para criar arranjos usados na plotagem de gráficos;
- **ones(Linhas, Colunas)** cria uma matriz do tamanho especificado onde todos os elementos são preenchidos com 1;
- **zeros(Linhas, Colunas)** cria uma matriz do tamanho especificado onde todos os elementos são preenchidos com 0;
- **eye(Ordem)** cria uma matriz identidade quadrada da ordem especificada;
- **cat(Dimensão, Arranjo1, Arranjo2, ...)** concatena arranjos, útil para criar arranjos de mais de duas dimensões;
- **size(Arranjo)** retorna um vetor com as dimensões do arranjo de dimensão n;
- **mean(Arranjo)** arranjo de n-1 dimensões com a média aritmética de cada vetor contido no arranjo de dimensão n;
- **std(Arranjo)** arranjo de n-1 dimensões com o desvio padrão de cada vetor de dimensão n;
- **prod(Arranjo)** arranjo de n-1 dimensões com o produto dos elementos de cada vetor de dimensão n;
- **max(Arranjo)** arranjo de n-1 dimensões com o maior elemento de cada vetor de dimensão n;
- **min(Arranjo)** arranjo de n-1 dimensões com o menor elemento de cada vetor de dimensão n;
- **sort(Arranjo)** arranjo de n-1 dimensões com cada vetor ordenado de dimensão n;
- **det(Matriz)** retorna o determinante da matriz;
- **trace(Matriz)** retorna o traço da matriz;
- **inv(Matriz)** retorna a inversa da matriz;

4.3.3 Impressão e entrada

- **input('mensagem')** exibe a mensagem e recebe um valor do teclado;
- **inputdlg('mensagem')** exibe a mensagem e recebe um valor do teclado usando uma caixa

de diálogo;

```
>> w = input('insira o valor de w ');
```

- **disp('mensagem única de saída')** imprime uma mensagem, recebe apenas um argumento, então para partir a mensagem para, por exemplo, imprimir um valor, é necessário fazer da entrada um arranjo;

- **fprintf('formato', variável)** imprime a variável no formato especificado.

```
>> w= input('insira o valor de w\n')
insira o valor de w
5
w=
5
>>disp(w)
5
>>disp(['w tem o valor de ', num2str(w)])
w tem o valor de 5
>>fprintf('w = %i\n', w)
w = 5
```

DICA: Há algumas máscaras para a impressão de variáveis usando a função printf:
%s - formato string;
%d - formato inteiro da variável;
%i - mesmo que **%d**;
%f - formato ponto flutuante;
%e - formato ponto flutuante em notação científica;
%s - formato mais compacto entre **%f** e **%e**;

DICA: Há também os caracteres:
n - quebra de linha **t** - tab

4.4 Funções Simbólicas

- **diff(Expressão)** retorna a derivada;

- **int(Expressão)** retorna a integral;

- **compose(F, G)** compõe $F(G(x))$;

- **expand(Expressão)** expande a expressão;

- **finverse(Expressão)** retorna a inversa da função;

- **pretty(Expressão)** expõe a função de forma mais bonita e organizada;

- **simplify(Expressão)** simplifica a expressão;
- **solve(Expressão)** encontra as raízes da expressão;
- **limit(Expressão, variável, valor)** calcula o limite da expressão quando a variável tende ao valor especificado. Pode ser incluído um quarto argumento ('right' ou 'left' impondo que o limite é pela esquerda ou direita;
- **subs(Expressão, variável, valor)** substitui na expressão a variável pelo valor.

```
>> syms x y
>> z = 2*x + 2*y
z=
2*x + 2*y
>> f = subs(z, x, 2)
f=
2*y + 4
>> subs(f, y, -2)
ans=
0
```

4.5 Complexos

- **real(Complexo)** retorna a parte real do número complexo;
- **imag(Complexo)** retorna a parte imaginária do número complexo;
- **abs(Complexo)** retorna o valor absoluto/módulo de um número complexo;
- **angle(Complexo)** calcula o ângulo de um número complexo;
- **conj(Complexo)** retorna o complexo conjugado;

4.6 Randômicas

- **rand** retorna números pseudo-aleatórios de distribuição uniforme entre 0 e 1;
- **randn** retorna números pseudo-aleatórios com distribuição normal padrão(média 0 e variância 1);
- **randi(A, m, ...)** O termo A pode ser um valor referente ao máximo valor retornável, ou um arranjo de dois valores, onde o primeiro será o mínimo e o segundo o máximo, limitando os valores possíveis da distribuição. m e os seguintes serão as dimensões do arranjo de saída. Caso seja somente p argumento m, será uma matriz mxm, caso contrário, os argumentos seguintes serão as dimensões do arranjo.

```
>> randi([0, 10], 2, 2, 2) %retornará um arranjo 2x2x2 com
   valores aleatórios de distribuição uniforme entre 0 e 10:
```

```
ans(:,:,1) =
     8     1
     9    10
ans(:,:,2) =
     6     3
     1     6
```

- **randperm(Quantidade)** realiza uma permutação aleatória de um vetor de valores inteiros de 1 até Quantidade. Ou seja, permuta-se os números naturais a partir do 1 até Quantidade, como se tivesse criado um vetor= linspace(1, Quantidade, Quantidade) e depois permutando os valores.

```
>>A=linspace(1, 10, 10)
A =
     1     2     3     4     5     6     7     8     9    10
>> randperm(10)
ans =
     8     3     9     6     7    10     5     1     2     4
```

4.7 Trigonométricas

- **deg2rad(X)** converte X de graus para radianos;
- **rad2deg(X)** converte X de radianos para graus;
- **acos(X)** retorna o arco cosseno de X;
- **acosh(X)** retorna o arco cosseno hiperbólico de X;
- **acot(X)** retorna o arco cotangente de X;
- **acoth(X)** retorna o arco cotangente hiperbólico de X;
- **acsc(X)** retorna o arco cossecante de X;
- **acsch(X)** retorna o arco cossecante hiperbólico de X;
- **asec(X)** retorna o arco secante de X;
- **asech(X)** retorna o arco secante hiperbólico de X;
- **asin(X)** retorna o arco seno de X;
- **asinh(X)** retorna o arco seno hiperbólico de X;
- **atan(X)** retorna o arco tangente de X;
- **atanh(Y/X)** retorna o arco tangente hiperbólico de Y/X;

- **atan2(Y,X)** retorna o arco tangente de quatro quadrantes de Y e X;
- **cos(X)** retorna o cosseno de X;
- **cosh(X)** retorna o cosseno hiperbólico de X;
- **cot(X)** retorna a cotangente de X;
- **coth(X)** retorna a cotangente hiperbólico de X;
- **csc(X)** retorna a cossecante de X;
- **csch(X)** retorna a cossecante hiperbólico de X;
- **sec(X)** retorna a secante de X;
- **sech(X)** retorna a secante hiperbólica de X;
- **sin(X)** retorna o seno de X;
- **sinh(X)** retorna o seno hiperbólico de X;
- **tan(X)** retorna a tangente de X;
- **tanh(X)** retorna a tangente hiperbólica de X;

4.8 Aritméticas

- **exp(X)** retorna a exponencial de X;
- **expm1(X)** mesmo que exp(X)-1;
- **log(X)** logaritmo natural de X;
- **log2(X)** logaritmo de X na base 2;
- **log10(X)** logaritmo de X na base 10;
- **sqrt(X)** raiz quadrada de X;

4.9 Numéricas

- **ceil(X)** retorna o arredondamento de X em direção ao infinito;
- **fix(X)** retorna o arredondamento X em direção ao zero;
- **floor(X)** retorna o arredondamento X em direção ao infinito negativo;
- **round(X)** retorna o arredondamento X para o inteiro mais próximo;

- **gcd(X, Y)** retorna o máximo divisor comum de X e Y;
- **lcm(X, Y)** retorna o mínimo múltiplo comum de X e Y;
- **rem(X, Y)** retorna o resto da divisão de X por Y, resto de X/Y;
- **sign(X)** retorna 0 caso X seja 0, retorna 1 caso X seja maior que 0 ou retorna -1 caso X seja menor que 0.

4.10 Gráficos

As funções e utilização dos gráficos estão melhor explicadas e descritas no capítulo destinado a elas, página 73.

- **plot(X, Y, TipoLinha, X2, Y2, TipoLinha2...)** plota em uma mesma imagem um gráfico de X versus Y com o tipo de linha especificado e mais quantos gráficos forem passados por parâmetro. Também pode ser usada como plot(Y), na qual será plotado os índices de Y versus seus valores;
- **fplot('funcao', [máximo, mínimo], TipoLinha...)** plota em uma mesma imagem um gráfico da função especificada pela string do valor máximo até o mínimo com o tipo de linha especificado e mais quantos gráficos forem passados por parâmetro;
- **title('Título')** insere ou modifica o título do gráfico;
- **xlabel('x')** insere ou modifica o rótulo no eixo x do gráfico;
- **ylabel('y')** insere ou modifica o rótulo no eixo y do gráfico;
- **zlabel('z')** insere ou modifica o rótulo no eixo z do gráfico;
- **grid** Mostra linhas de grade no gráfico caso não as tenha, ou as remove, caso contrário. Também pode ser usado como **grid on** ou **grid off** para especificar a ação de ligar ou desligar;
- **close(a)** fecha a janela(gráfico) de índice a;
- **close all** fecha todas as janelas(gráficos).

4.11 Polinômios

As funções e utilização dos polinômios estão melhor explicadas e descritas no tópico destinado a eles, página 97.

- **roots(Polinômio)** retorna as raízes do polinômio;
- **poly(Raízes)** retorna os coeficientes do polinômio em que as raízes são os números do arranjo de parâmetro;
- **conv(Polinômio1, Polinômio2)** retorna os coeficientes da multiplicação dos polinômios;
- **deconv(Polinômio1, Polinômio2)** retorna dois vetores, sendo o primeiro o quociente da divisão entre os polinômios, e o segundo o resto, normalmente usa-se [q, r] = deconv(y1, y2);

- **polyval(Polinômio, Valor(es))** retorna o resultado das avaliações do polinômio nos valores fornecidos;

- **polyfit(x, y, n)** retorna os coeficientes do polinômio ajustado de grau n nos valos de x e y;

- **[r, p, k] = residue(n, d)** sendo n e d, o numerador e denominador, respectivamente, a função retorna r=resíduo, p=polo, k=termo direto;

- **[n, d] = residue(r, p, k)** sendo r=resíduo, p=polo, k=termo direto, a função retorna n e d, o numerador e denominador, respectivamente.

Exercícios

4.1 1) Existe um procedimento chamado decomposição de Cholesky. Neste procedimento, a matriz, caso seja definida positiva, é decomposta em um produto entre uma matriz triangular inferior e sua adjunta, facilitando e otimizando os processos computacionais. O Matlab possui uma função para calcular a decomposição de Cholesky. Utilizando os comandos help e lookfor, descubra qual é essa função e calcule a decomposição de Cholesky para A=[9 6 -3 3; 6 20 2 22; -3 2 6 2; 3 22 2 28].

4.2 2) Tomando por base algumas das funções vistas anteriormente:
a) Calcule a raiz quadrada de 187.9915;
b) Arredonde o resultado em direção a:
i) O inteiro mais próximo;
ii) Zero;
c) Calcule a exponencial de 3;
d) Arredonde o resultado em direção a:
i) O inteiro mais próximo;
ii) Infinito;
e) Calcule o logaritmo natural de 3;
f) Arredonde o resultado em direção a:
i) O inteiro mais próximo;
ii) Infinito.

4.3 Utilizando funções de criação e manipulação de arranjos faça:
a) Um arranjo A 1x3 preenchido com 1, 2 ,3;
b) Uma matriz B 3x3 em que cada linha é igual a A;
c) Um arranjo P 1x2 preenchido com 2, 2;
d) Uma matriz Q 2x2 em que cada linha seja igual a P;
e) Um arranjo R 2x2x2 em que cada matriz 2x2 seja igual a Q.

4.4 Utilizando os arranjos criados na questão anterior e as funções de manipulação de arranjos, calcule:
a) As dimensões de R;
b) O traço de B;
c) O determinante de B;
d) O traço de Q;
e) O determinante de Q.

4.5 Usando as funções de impressão e entrada:
a) Imprima a mensagem Hello World;
b) Peça que o usuário insira um valor e o atribua a variável x;
c) Imprima a mensagem Olá, programador, x = <valor de x>.

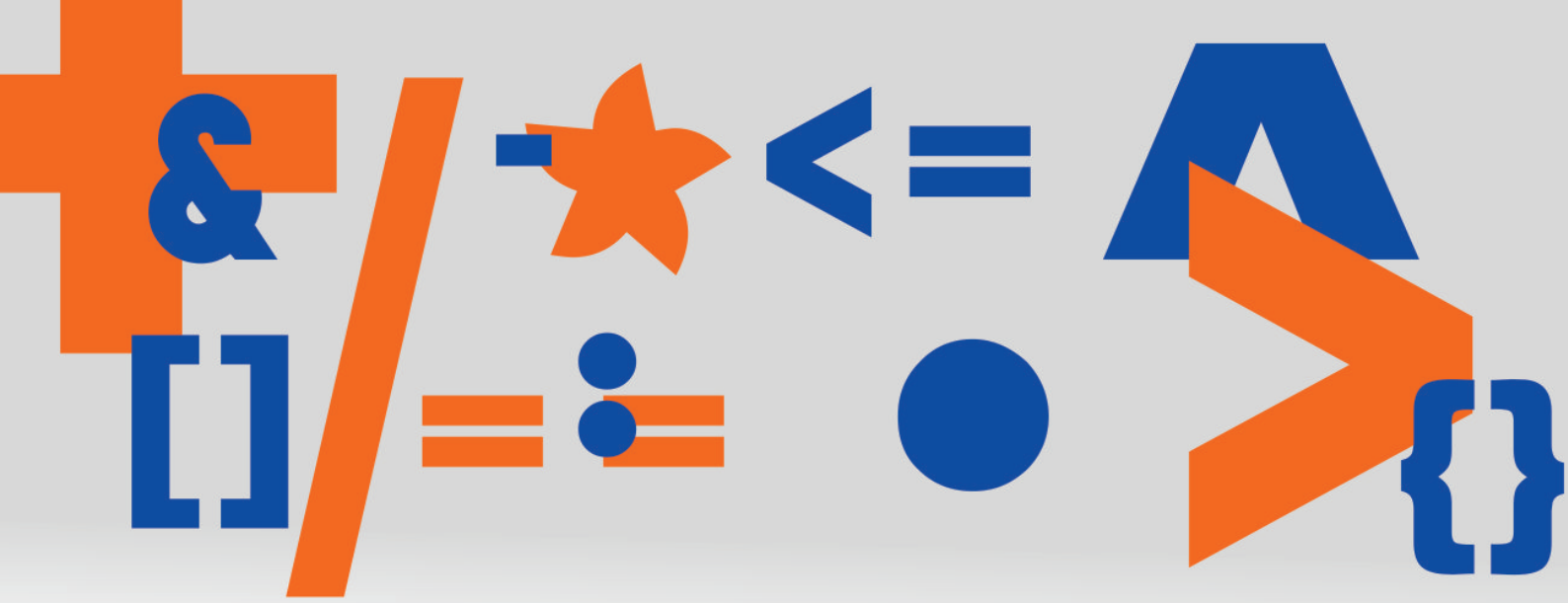
4.6 Crie:
a) Um arranjo a1, com 5 elementos igualmente espaçados entre 0 e 2π;
b) Um arranjo b1 = seno (a1);
c) Um arranjo a2, com 10 elementos igualmente espaçados entre 0 e 2π;
d) Um arranjo b2 = seno (a2);
e) Um arranjo a3, com 100 elementos igualmente espaçados entre 0 e 2π;
f) Um arranjo b3 = seno (a3);

4.7 Utilizando os arranjos anteriores, use a função plot para plotar, simultaneamente, três gráficos de ai x bi, i=1,2,3.

4.8 Compare os desenhos dos gráficos anteriores e diga qual é o 1, 2 e 3. Justifique sua resposta.

4.9 Calcule o ângulo e o módulo do complexo 13.5 + 13.5i, em seguida, converta o resultado do ângulo de radianos para graus.

4.10 Repita o procedimento anterior para o complexo 20i.



5. Operadores

Este capítulo abordará os operadores do Matlab, isto é, os elementos do código responsáveis por indicar as ações a serem feitas com dados. Alguns já foram abordados anteriormente, de forma implícita.

5.1 Aritméticos

Os operadores aritméticos são responsáveis por realizar as operações matemáticas básicas. Eles estão apresentados nas tabelas 5.1.1, 5.1.2, 5.1.3 e 5.1.4. Para facilitar a compreensão, as variáveis utilizadas serão:

- a,b vetores;
- A,B matrizes;
- c,d escalares;
- z escalar complexo;
- Z matriz de complexos.

O operador "." antes do operador aritmético indica que a operação é ponto a ponto. Por exemplo, uma multiplicação matricial padrão é expressa por A*B. No entanto, se for feito A.*B, significa que a multiplicação é ponto a ponto, ou seja, cada elemento A_{ij} será multiplicado pelo B_{ij}.

DICA: Se for utilizado o operador transposição/conjugação junto com o operador de ponto a ponto em uma matriz de complexos Z, isto é, **Z.'**, o resultado será a transposta de Z transposto, não conjugada.

Tabela 5.1.1: *Operadores Aritméticos 1*

Operação	Expressão	Descrição
Adição Escalar	c+d	Soma de $c+d$
Subtração Escalar	c-d	Soma de $c+d$
Multiplicação Escalar	c*d	Multiplicação de cd
Divisão Escalar	c/d	Divisão de $\frac{c}{d}$
Divisão Escalar(Barra invertida)	c\d	Divisão de $\frac{d}{c}$
Potenciação Escalar	c∧d	Potenciação de c^d
Adição Escalar	c.+d	Soma de $c+d$
Subtração Escalar	c.-d	Soma de $c+d$
Multiplicação Escalar	c.*d	Multiplicação de cd
Divisão Escalar	c./d	Divisão de $\frac{c}{d}$
Divisão Escalar(Barra invertida)	c.\d	Divisão de $\frac{d}{c}$
Potenciação Escalar	c.∧d	Potenciação de c^d

Tabela 5.1.2: *Operadores Aritméticos 2*

Operação	Expressão	Descrição
Adição Escalar	a+c	Soma de $[a_1+c, a_2+c, ...a_n+c]$
Subtração Escalar	a-c	Soma de $[a_1-c, a_2-c, ...a_n-c]$
Adição Vetorial	a+b	Soma de $[a_1+b_1, a_2+b_2, ...a_n+b_n]$
Subtração Vetorial	a-b	Soma de $[a_1-b_1, a_2-b_2, ...a_n-b_n]$
Multiplicação Escalar	a*c	Multiplicação de $[ca_1, ca_2, ..., ca_n]$
Multiplicação Vetorial Ponto a Ponto	a.*b	Multiplicação de $[a_1b_1, a_2b_2, ...a_nb_n]$
Divisão Vetorial Ponto a Ponto	a./b	Multiplicação de $[\frac{a_1}{b_1}, \frac{a_2}{b_2}, ..., \frac{a_n}{b_n}]$
Divisão Vetorial Ponto a Ponto(Barra Invertida)	a.\b	Multiplicação de $[\frac{b_1}{a_1}, \frac{b_2}{a_2}, ..., \frac{b_n}{a_n}]$
Potenciação Escalar Ponto a Ponto	a.∧c	Soma de $[a_1^c, a_2^c, ...a_n^c]$
Potenciação Escalar Ponto a Ponto	a.∧b	Soma de $[a_1^{b_1}, a_2^{b_2}, ...a_n^{b_2}]$

Tabela 5.1.3: *Operadores Aritméticos 3*

Operação	Expressão	Descrição
Adição Escalar	A+c	Soma de $A_{ij}+c, \forall i,j$
Subtração Escalar	A-c	Subtração de $A_{ij}-c, \forall i,j$
Adição Matricial	A+B	Soma de $A_{ij}+B_{ij}, \forall i,j$
Subtração Matricial	A-B	Subtração de $A_{ij}-B_{ij}, \forall i,j$
Multiplicação Escalar	A*c	Multiplicação de $cA_{ij}, \forall i,j$
Multiplicação Matricial Ponto a Ponto	A.*B	Multiplicação de $A_{ij}B_{ij}, \forall i,j$
Multiplicação Matricial	A*B	Multiplicação de AB
Divisão Matricial Ponto a Ponto	A./B	Divisão de $\frac{A_{ij}}{B_{ij}}, \forall i,j$
Divisão Matricial Ponto a Ponto(Barra Invertida)	A.\B	Divisão de $\frac{B_{ij}}{A_{ij}}, \forall i,j$
Divisão Matricial	A\B	Multiplicação de $A^{-1}B$
Potenciação	A∧c	Potenciação de A^c
Potenciação Ponto a Ponto	A.∧c	Potenciação de $A_{ij}^c, \forall i,j$
Potenciação Ponto a Ponto	A.∧B	Potenciação de $A_{ij}^{B_{ij}}, \forall i,j$
Potenciação Ponto a Ponto	c.∧A	Potenciação de $c^{A_{ij}}, \forall i,j$
Potenciação	c∧A	Potenciação de c^A

Tabela 5.1.4: *Operadores Aritméticos 4*

Operação	Expressão	Descrição
Conjugação Complexa	z’	Conjugado de z
Transposição	a’	a transposto
Transposição	A’	A transposto
Transposição e Conjugação	Z’	Z transposto e cada Z_{ij} conjugado

DICA: Para se calcular o produto vetorial entre dois vetores a e b, deve-se usar a função **cross(a, b)**, enquanto que o produto escalar pode ser feito transpondo um deles: **a'*b** ou **a*b'**, dependendo se a e b são vetores coluna ou vetores linha.

DICA: Para se calcular o resto de divisão entre dois valores c e d deve-se usar a função **rem(c,d)** ou **mod(c,d)**.

5.2 Lógicos

Os operadores lógicos são usados para avaliar mais de uma expressão, como em estruturas condicionais e de repetição, que serão abordadas à frente, página 59.

Para o Matlab, zero é falso e qualquer valor diferente de zero é verdadeiro, sendo 1(um) o valor padrão.

Exemplo:

```
>> 2 & 5
ans =
  logical
   1
```

Neste exemplo, o cálculo lógico foi feito diretamente na janela de comando (Command Window). A variável **ans** armazenou o resultado, sendo este verdadeiro(1) ou falso(0), do tipo de dado booleano (logical).

Os operadores lógicos estão na tabela 5.2.1.

Tabela 5.2.1: *Operadores Lógicos*

Operação	Expressão	Descrição
Conjugação	a & b	Conjunção que avalia a e b
Disjunção	a | b	Disjunção que avalia a ou b
Negação	~a	Negação de a
Conjugação Curto Circuito	a && b	Conjunção de curto circuito que avalia a e b
Disjunção Curto Circuito	a || b	Disjunção de curto circuito que avalia a ou b

Os operadores de curto circuito têm as mesmas repostas dos comuns. A diferença está na avaliação, isto é, no "e curto circuito", caso a primeira expressão seja falsa, o Matlab não avaliará a segunda, e retornará 0. Já no "e padrão", sempre ambas as expressões são avaliadas. Considerando o "ou curto circuito", a segunda expressão não é avaliada caso a primeira seja verdadeira, e 1 já é retornado. Exemplo:

```
>> x = [2 2]; y = [1 2 3];
>> a = 1; x = [2 2]; y = [1 2 3];
>> a == 0 && x+y == 1 % e curto circuito
ans =
  logical
   0
>> a == 0 & x+y == 1  % e
Matrix dimensions must agree.
```

Nota-se que x+y está incorreto porque os vetores não são do mesmo tamanho, entretanto, no caso curto circuito, o Matlab não avaliou a segunda expressão, uma vez que apenas com a primeira expressão é possível calcular a resposta, neste exemplo.

5.3 Relacionais

Eles são necessários para fazer avaliações comparativas, retornando verdadeiro(1) ou falso(0), assim como os lógicos. Esses operadores também são muito utilizados em estruturas condicionais e laços. São mostrados na tabela 5.3.1.

Tabela 5.3.1: *Operadores Relacionais*

Operação	Expressão	Descrição
Maior que	x > y	Avalia se x é maior que y
Maior ou igual a	x >= y	Avalia se x é maior ou igual a y
Menor que	x < y	Avalia se x é menor que y
Maior ou igual a	x <= y	Avalia se x é menor ou igual a y
Igual a	x == y	Avalia se x é igual a y
Diferente de	x ~= y	Avalia se x é diferente de y

5.4 Precedência

Expressões com mais de um operador são avaliadas segundo a precedência deles, similar a expressões matemáticas que também têm suas precedências. Na dúvida, deve-se usar os parênteses, pois são os primeiros a serem avaliados.

- ()
  Parênteses
- .' .^ ' ^
  Transposição e potenciação
- .^~ ^~ .^+ .^- ^+ ^-
  Pontenciações
- + - ~
  Unários

- **.* ./ .\ * / **
  Multiplicação e divisão
- **+ -**
  Soma e subtração
- **:**
  Dois pontos
- **< <= > >= == ~=**
  Relacionais
- **&**
  E
- **|**
  Ou
- **&&**
  E curto circuito
- **||**
  Ou curto circuito

Exercícios

5.1 Utilizando o Matlab, compute:
a) 13.711^2 ;
b) 13.711^3 ;
c) [4, 1, 2] transposto.

5.2 Veja a diferença entre os operadores \ e / ao se calcular 10/5 e 10\5.

5.3 É possível inserir algum valor de x para que a expressão lógica 10 > x > 2 retorne 1 (verdadeiro)? Justifique.

5.4 Para quais valores de x irá retornar 1 a expressão

```
>> ( 2 < 10 == 1 ) + 10 > x
```

irá retornar 1 (verdadeiro)? Justifique.

5.5 Determine pelo menos um valor para cada variável a, b, c e d para que a expressão seguinte retorne 1 (verdadeiro):

```
((a > 10) == 0) + b > c == ~d
```

5.6 Determine pelo menos um valor para cada variável a, b, c e d para que a expressão seguinte retorne 0 (falso):

```
((a > 10) == 0) + b > c == ~d
```

5.7 Declare:
a)

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

b)

$$B = \begin{pmatrix} 1 & 6 & 1 \\ 2 & 9 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

c)

$$C = \begin{pmatrix} 2 & 4 & 7 \end{pmatrix}$$

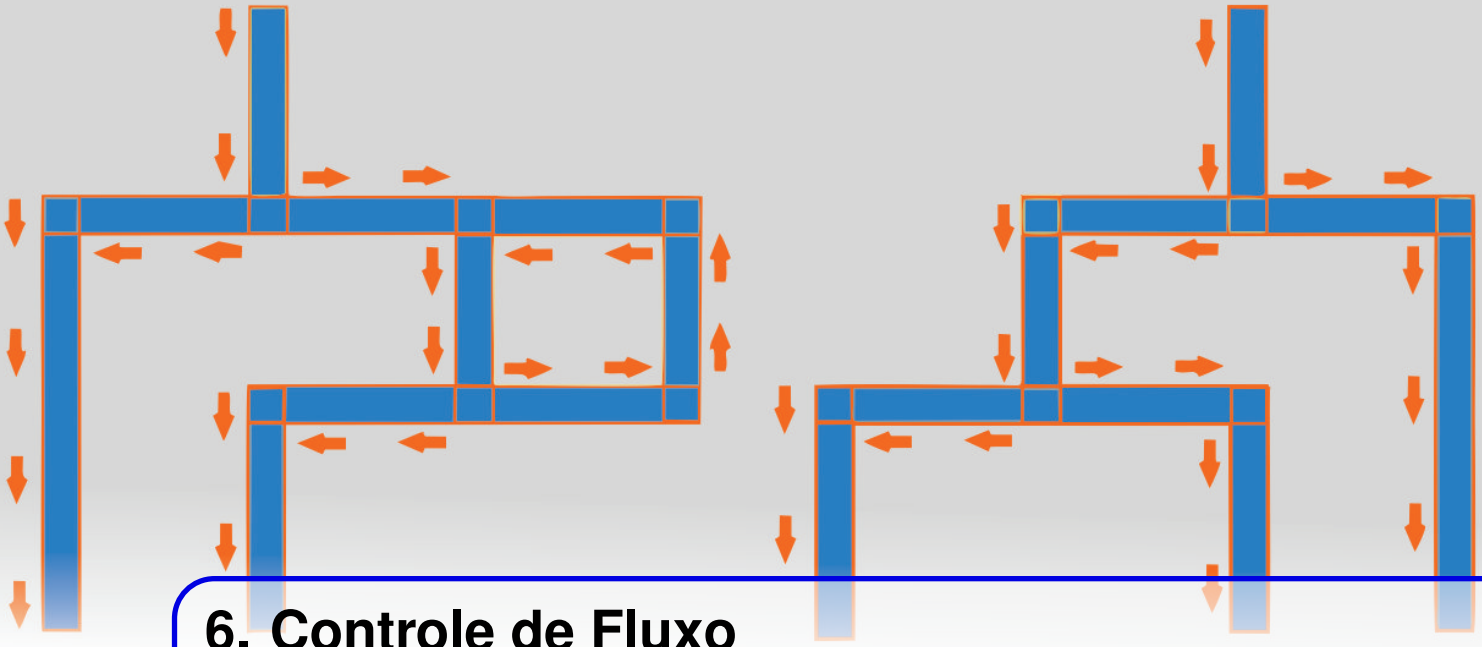
d)

$$D = 7$$

5.8 Utilizando os arranjos anteriores, calcule:
a) AB;
b) $A_{ij}B_{ij}, \forall i, j$;
c) DA;
d) DC_{ij};
e) DC.

5.9 Repita os cálculos de 8)a) e 8)b) e compare os resultados obtidos pelo uso do ".".

5.10 Crie um vetor A com seis elementos em que cada um vale $A_{ij} = rand > 0.5$. Sabendo que a função **rand** retorna números de distribuição uniforme entre 0 e 1, quantos 0s e quantos 1s você acha que A vai possuir? Repita o procedimento algumas vezes e observe.



6. Controle de Fluxo

O controle de fluxo é o controle da sequência de comandos executados pela rotina. Um código é sempre executado linha por linha, na ordem em que se encontram. Desta forma, se existem somente procedimentos simples, o funcionamento da rotina é linear. No entanto, com o controle de fluxo, é possível fazer com que o código tome decisões ou execute alguns procedimentos repetidas vezes.

Tais ações equivalem a dizer para o código instruções como "se isso, faça aquilo"ou "enquanto isso, execute aquilo".

No caso do Matlab, todas as estruturas de controle de fluxo são iniciadas por palavras chave e seu término é indicado pela palavra **end**. Desta forma, uma estrutura geral pode-se parecer com:

```
ESTRUTURA  <expressao>
    procedimento1
    procedimento2
end
```

Algumas estruturas podem ainda estar subordinadas a outras estruturas. Esse processo é chamado de **aninhamento de estruturas**.

6.1 Estruturas Condicionais

O primeiro conjunto de estruturas de controle de fluxo são as estruturas condicionais, as responsáveis por tomadas de decisão, que indicarão uns procedimentos ou outros dependendo da condição avaliada.

As mais populares das estruturas condicionais em programação são *if-else* e *switch*. Ambas podem ser usadas no Matlab.

6.1.1 if-else

A estrutura *if* avalia uma expressão e caso a mesma seja verdadeira, os procedimentos subordinados são executados. Não é obrigatório, mas em seguida, pode-se adicionar apenas o *else*, com outros procedimentos subordinados, que serão executados caso a condição do if seja falsa.

```
x = input('digite um valor: ')
if x<10
    disp('x<10')
else:
    disp('x>=10')
end
```

Neste exemplo, dependendo do valor que o usuário atribuir a x, uma mensagem diferente será impressa.

Como comentado anteriormente, pode haver também um tipo de estrutura subordinada chamada *else if*, no caso de haver mais de uma condição *senão*. Neste caso, deve-se usar a palavra reservada *elseif*.

```
if x==1
    disp('X vale 1')
elseif x==2
    disp('X vale 2')
elseif x==3
    disp('X vale 3')
else
    disp('X nao vale 1, 2 ou 3')
end
```

DICA : No Matlab não é obrigatório identação. Isto é, não é necessário usar o tab para recuar as instruções nas linhas. No entanto, esse procedimento é recomendado, trata-se de uma boa prática de programação. Um código bem identado é visualmente mais fácil de compreender, pois assim é possível ter uma noção rápida de quais estruturas estão subordinadas às outras.

6.1.2 switch

Há também outra estrutura condicional, o switch. Uma variável é avaliada e separada em casos, como no exemplo a seguir:

```
X=input('insira o valor de X')
switch X
    case 1
        disp('X vale 1')
    case 2
        disp('X vale 2')
    case 7
        disp('X vale 7')
    otherwise
        disp('X vale qualquer outra coisa')
end
```

Nesse exemplo, o valor contido na variável X é avaliado. Caso seja 1, será impressa a mensagem dentro de **case 1** e assim por diante. O caso indicado por **otherwise** é o procedimento que será executado caso todos os outros **cases** não tenham sido atendidos, porém não é obrigatório.

Usar switch ou aninhamentos if-else é, basicamente, a mesma coisa. A diferença será na praticidade e compreensão do código, dependendo da natureza das operações e procedimentos realizados.

6.2 Estruturas de Repetição

Já o segundo conjunto de estruturas, as estruturas de repetição, são responsáveis por criar laços que repetem os procedimentos a eles subordinados. A cada repetição é dada o nome de **iteração**. Elas também são conhecidas como laços ou loops.

6.2.1 while

O primeiro é o *while*. While avalia a condição a ele exposta a cada iteração, e enquanto ela for verdadeira, o laço é mantido. Quando a condição é falsa, o laço é interrompido.

```
>> valor=128, contador=0;
>> while valor>1
valor = valor/2
contador = contador + 1
A(contador) =  valor;
end
>> A
A=
 64 32 16 8 4 2 1
```

Neste exemplo, a variável **valor** começa com 128 e o contador com 0. A cada iteração, **valor** é dividido pela metade e o **contador** é incrementado de 1. Assim, **valor** é atribuído à um vetor **A**. Este procedimento acontece até que **valor** seja maior do que 1.

OBS: Estruturas de repetição que, a priori, nunca terminam são chamadas de **loops infinitos**. Um loop infinito pode ser facilmente demonstrado com:

```
>> while true
```

6.2.2 for

O segundo é o *for*. Ao invés de avaliar uma condição, como o while, o for, no Matlab, atribui valores a uma variável que percorrerá um vetor.

```
>> for i = 1:10
    A(i)=i^2;
end
>> A =
     1     4     9    16    25    36    49    64    81   100
```

6.3 Desvio Incondicional

Os desvios incondicionais são mudanças na execução do programa para outra linha, isto é, desviar o código para outra parte. Há dois comandos de desvio, **break** e **continue**.

6.3.1 break

O primeiro deles, o *break*, é, geralmente, utilizado dentro de uma estrutura condicional, que por sua vez, está dentro de uma estrutura de repetição. Serve para forçar a saída do loop onde está no momento em que é identificado no fluxo de execução do programa.

```
1 x=5;
2 while true
3     x=x+1;
4     if x>40
5     break
6     end
7 end
```

6.3.2 continue

O segundo desvio é o *continue*. Também é geralmente usado dentro de uma estrutura condicional, dentro de uma estrutura de repetição. Serve para forçar o laço a ir para a próxima iteração, ou sair do laço, caso seja a última iteração, ignorando os comandos ainda não executados na iteração presente.

```
>> for i = 1:5
if i == 4
continue
else
fprintf('%d ', i)
end
end
1 2 3 5
```

Neste exemplo, é realizada a impressão dos valores que a variável i assume ao passar pelo laço for, variando de 1 a 5. No entanto, caso i seja 4, não é impresso e o comando é pulado.

Exercícios

6.1 Crie a e b, dois escalares com valores à sua escolha. Receba do usuário um número para uma variável de controle e diga que, caso o número seja igual a 1, a e b serão somados. Faça uma estrutura condicional usando if-else que caso o número seja igual a 1, a e b serão somados.

6.2 Adicione à sequência de comandos anteriores, a opção de número 2, que indicará a subtração a – b.

6.3 Adicione à sequência de comandos anteriores, a opção de número 3, que indicará a multiplicação ab.

6.4 Adicione à sequência de comandos anteriores, a opção de número 4, que indicará a divisão $\frac{a}{b}$.

6.5 Refaça a sequência de comandos utilizando a estrutura condicional switch.

6.6 Reescreva a sequência de comandos a seguir utilizando a sintaxe if-else:

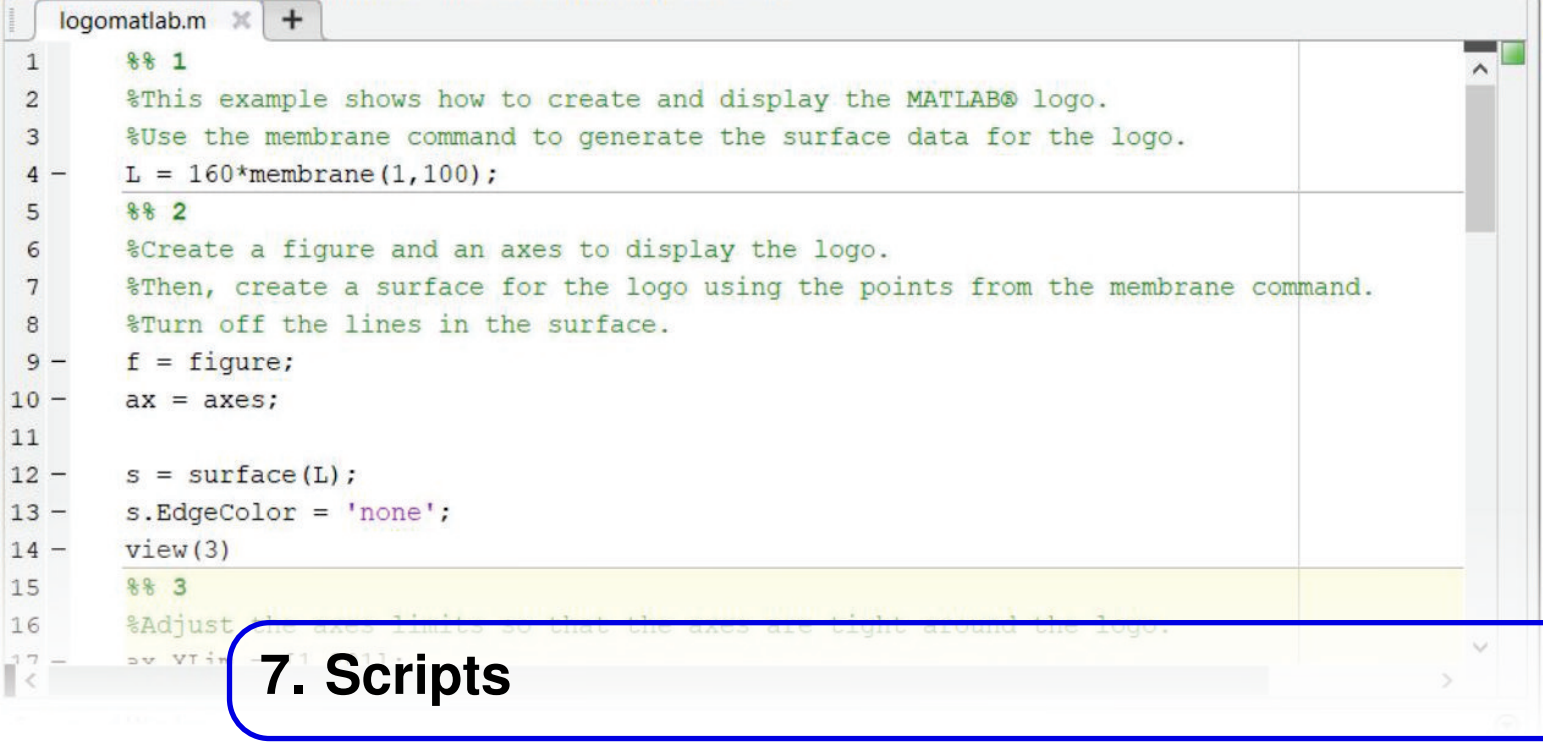
```
Nota=input('Digite a nota do aluno(0 a 5)')
Nota=round(nota)
switch Nota
    case 0
        disp('Procure o professor para ajuda')
    case 1
        disp('Precisa se dedicar mais')
    case 2
        disp('Ainda um pouco mais')
    case 3
        disp('Bom, 60 é 100')
    case 4
        disp('Parabens')
    case 5
        disp('Sensacional!!')
end
```

6.7 Utilizando uma estrutura de repetição, crie um vetor G de tamanho 1x10, com seus valores iguais a 102, 92, 82... 12. Lembre-se de conferir os valores do vetor.

6.8 Caso tenha usado for, repita a questão anterior usando while. Caso tenha usado while, repita usando for. Lembre-se de conferir os valores do vetor.

6.9 Utilizando a estrutura de repetição for, crie um vetor H de tamanho 1x5, com valores iguais a 33, 33, 13, 23, 33. Lembre-se de conferir os valores do vetor.

6.10 Faça o mesmo vetor H da questão anterior, agora utilizando while. Lembre-se de conferir os valores do vetor.



7. Scripts

Para tarefas mais longas e complexas, ao invés de digitar comando por comando na **Command Window** ou usar o atalho **shift + enter** várias vezes, é possível criar linhas de código para serem executadas em sequência e quando desejadas. São os chamados **scripts**, ou seja, roteiros. Pode-se criá-los usando algum editor de texto como o próprio bloco de notas do Windows ou Notepad++ e salvá-los com a extensão **.m**. Ou pode-se ainda usar o próprio editor do Matlab. Para isso, basta clicar em **New Script** no canto superior esquerdo (Fig. 7.0.1), usar o atalho **Ctrl + N** ou usar seguinte o comando na janela de comandos:

```
>> edit <arquivo>
```

Assim, uma nova janela será aberta juntamente com uma nova barra de tarefas, o editor.

Após escrever um código qualquer ele pode ser salvo em um diretório. Estando salvo, ele pode ser executado clicando no botão **run**, com o símbolo de um triângulo na barra de tarefas.

Para ser executado com este botão, o script precisa estar no diretório corrente do Matlab, isto é, aparecer no **Current Folder**. Desta forma, ele pode ser executado também apenas chamando o nome do arquivo na **Command Window**.

Caso o código não esteja no diretório corrente, uma janela de diálogo é aberta, como mostra a Fig. 7.0.2. Neste diálogo, há a primeira opção de *Change Folder*, em que o diretório será automaticamente setado para aquela pasta onde se encontra o arquivo e há a segunda opção de *Add to Path* cuja função está explicada a seguir.

Em sistemas operacionais, **PATH** é uma variável de ambiente que indica um conjunto de diretórios nos quais programas executáveis podem estar localizados. No Matlab, ela desempenha a mesma função. Sendo assim, ao adicionar um diretório ao Path, o conteúdo desse diretório pode ser

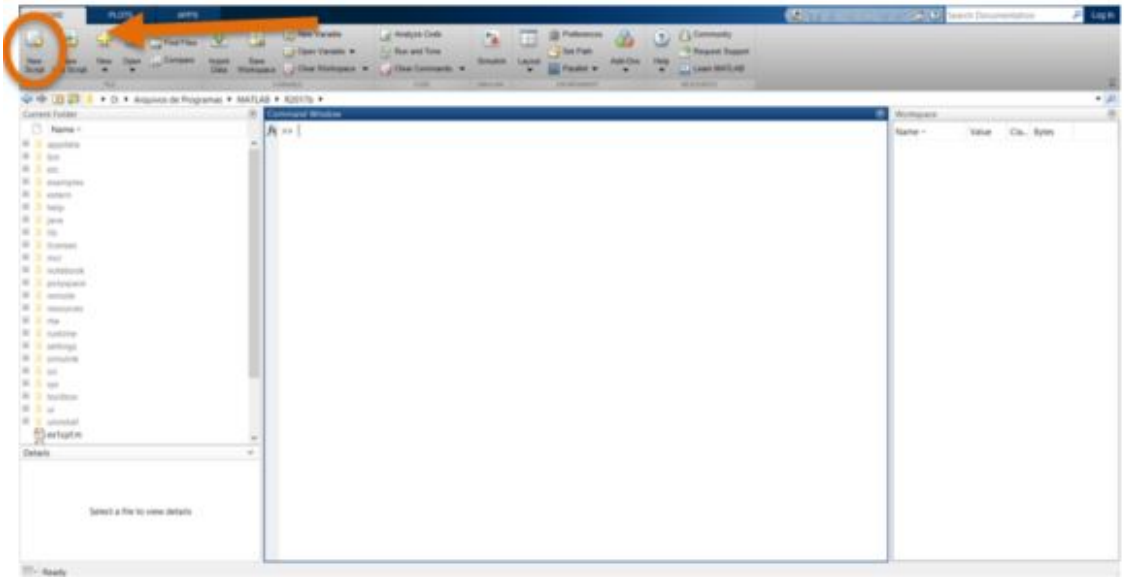


Figura 7.0.1: *Botão de novo script*

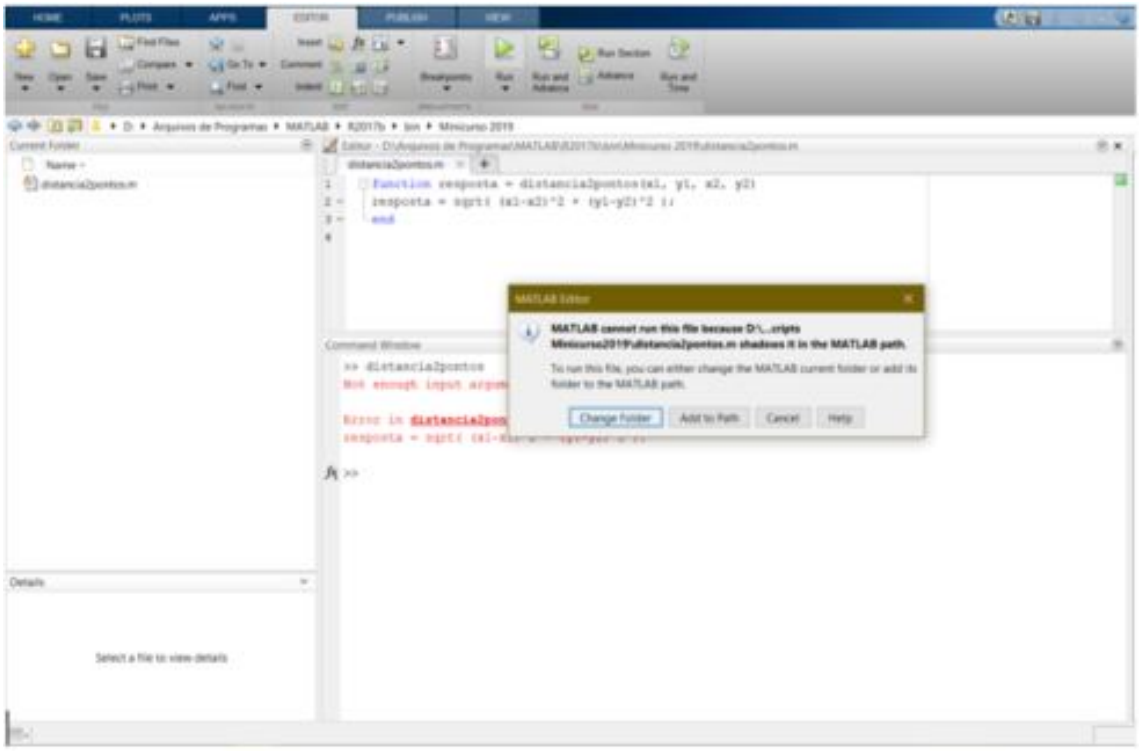


Figura 7.0.2: *Caixa de diálogo ao se clicar em* ***run***

acessado independente do **Current Folder**.

O Path pode ser visualizado digitando o comando:

```
>> path
```

Uma pasta pode ser adicionada ao Path clicando com o botão direito sobre ela no Current Folder. As alterações feitas no Path durante uma execução do Matlab não são salvas, para isso, deve-se usar o comando:

```
>> savepath
```

Por fim, para se configurar o Path manualmente, pode-se clicar na opção **Set Path** na barra de tarefas **Home** ou digitar o seguinte comando que abrirá um diálogo:

```
>> pathtool
```

7.1 Declaração de Funções

É possível usar os scripts para se declarar funções, nesse caso, o nome do arquivo .m deve ser obrigatoriamente o mesmo nome da função. A sintaxe é a seguinte:

```
1 function [<variaveis retornadas>] = <nome da funcao avaliada nos
     parametros>
2     <procedimentos>
3 end
```

A exemplo, uma função que recebe quatro parâmetros, coordenadas x e y de dois pontos, e retorna a distância entre esses pontos é explicada a seguir e ilustrada pela Fig. 7.1.1. (O arquivo deve estar devidamente salvo com o mesmo nome da função e estar contido no diretório corrente ou no Path).
distancia2pontos.m

```
1 function resposta = distancia2pontos(x1, y1, x2, y2)
2     resposta = sqrt( (x1-x2)^2 + (y1-y2)^2 );
3 end
```

Na **Command Window**:

```
>> distancia2pontos(0,0,3,4)
ans =
     5
```

Ainda nesse exemplo, a função pode ser alterada para retornar duas variáveis, o módulo da distância e o ângulo:
distancia2pontos.m

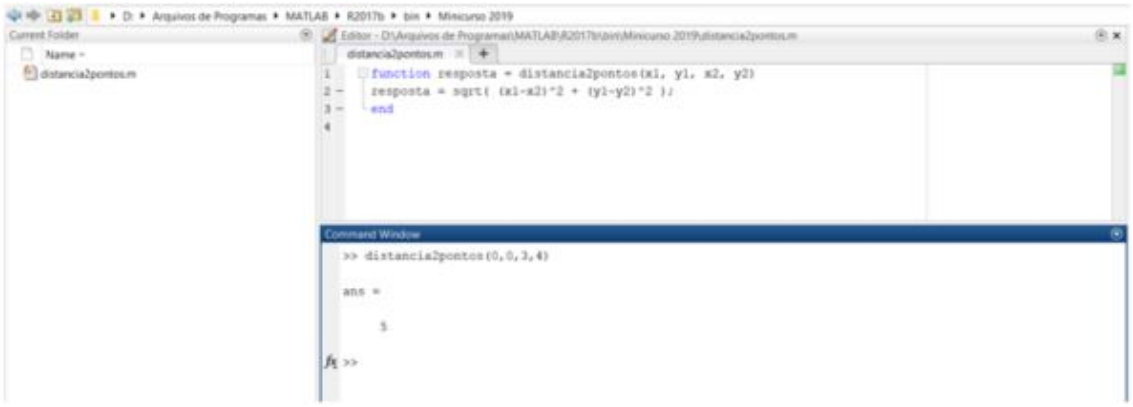


Figura 7.1.1: *Exemplo de declaração de função*

```
function [r, theta] = distancia2pontos(x1, y1, x2, y2)
    r = sqrt( (x1-x2)^2 + (y1-y2)^2 );
    theta = atan2((y2-y1),(x2-x1));
end
```

Na **Command Window**:

```
>> [a b] = distancia2pontos(0,0,3,4)
a =
     5
b =
    0.9273
```

As funções possuem descrições que podem ser usadas para auxílio quando executados os comandos **help** e **lookfor**. Essas descrições podem ser também adicionadas às funções criadas pelos usuários. Basta adicionar comentários em suas descrições. Ainda na função exemplo, explicada a seguir e ilustrada pela Fig. 7.1.2:

```
function resposta = distancia2pontos(x1, y1, x2, y2)
%distancia2pontos calcula a distancia entre dois pontos
%Exemplo de descricao da funcao
%conteudo e etc...
    resposta = sqrt( (x1-x2)^2 + (y1-y2)^2 );
end
```

Na **Command Window**:

```
>> lookfor distancia
distancia2pontos                - calcula a distancia entre dois
   pontos
>> help distancia2pontos
 distancia2pontos calcula a distancia entre dois pontos
 Exemplo de descricao da função
```

```
conteudo e etc...
```

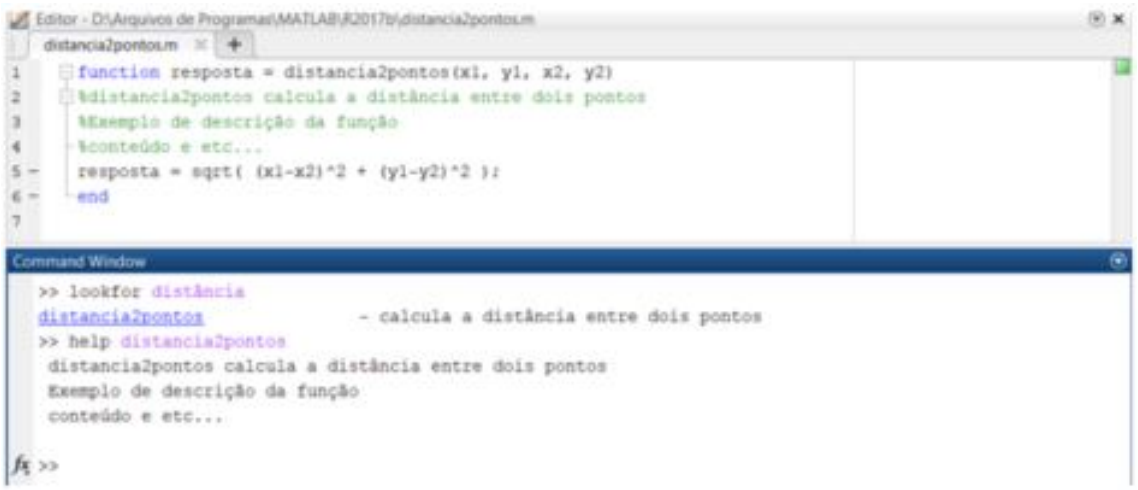


Figura 7.1.2: *Descrição da função distancia2pontos*

DICA: Em programação é comum o uso de funções do tipo **void**, isto é, funções que não têm como objetivo retornar valores, apenas realizar outros procedimentos como impressões de dados. Para se fazer uma função que não retorna valores no Matlab, basta omitir a parte das variáveis retornadas.

7.2 Documentação

O Matlab possui uma função que pode ser extremamente útil na hora de documentar um trabalho ou atividade. Se o código estiver bem organizado, comentado e separado em seções por %%, então basta usar o comando:

```
>> publish('codigo.m', 'pdf')
```

Assim, um PDF será gerado com os resultados e as descrições do código. É possível documentar também em HTML, Doc e LaTeX.

Exercícios

7.1 Refaça o exercício 6) do capítulo Controle de Fluxo, em um script. Inicie o script limpando as variáveis e a Command Window.

7.2 Refaça o exercício 7) do capítulo Controle de Fluxo, em um script. Inicie o script limpando as variáveis e a Command Window.

7.3 Refaça o exercício 9) do capítulo Controle de Fluxo, em um script. Inicie o script limpando as variáveis e a Command Window.

7.4 Crie um arquivo .m chamado ex1log.m que define a função ex1log. A função receberá dois escalares x e y e retornará o $log_y x$. Lembre-se que: $log_b a = \frac{log_c a}{log_c b}$.

7.5 Modifique a função anterior para que x e y possam ser arranjos de mesmo tamanho, fazendo com que a função retorne outro arranjo com cada elemento com valor $log_{y_i} x_i$.

7.6 Crie um arquivo .m chamado ex1ord que define a função ex1ord. A função deverá receber um arranjo linear e retornar outro arranjo, com os mesmos valores do primeiro, ordenados em ordem crescente. Use o método de ordenação que lhe deixar mais confortável.

7.7 Os dois últimos dígitos de um cpf são calculados da seguinte forma:
O número é composto por nove dígitos e mais dois verificadores, chamemos de ABCDEFGHI-JK. Para se calcular J é feito: $\frac{(10A+9B+8C+7D+6E+5F+4G+3H+2I)}{11}$ E desta conta analisamos o resto. Se o resto for 0 ou 1 J será 0. Senão, $J = 11 - resto$.
Para K, fazemos: $\frac{(11A+10B+9C+8D+7E+6F+5G+4H+3I+2J)}{11}$ Analisamos o resto. Se o resto for 0 ou 1 K será 0. Se não, K = 11 - resto.
Crie um arquivo .m chamado de ex1cpf que defina a função ex1cpf. A função deverá receber um número de cpf completo em forma de escalar e conferir se os dois últimos dígitos estão corretos, retornando 1 caso estejam e 0 caso estejam incorretos.

7.8 Faça um script no Matlab que pede ao usuário um número de cpf. Logo após, execute o mesmo procedimento da questão anterior e imprima a mensagem "CPF correto", caso esteja correto ou "CPF incorreto", caso esteja incorreto.

7.9 Crie um script no Matlab para calcular a média de notas de uma determinada quantidade de alunos. O procedimento deverá ser da seguinte forma: O programa deve pedir ao usuário um número N referente à quantidade de alunos. Em seguida, deverá coletar as N notas, também do usuário. No final, deverá imprimir o valor da nota média entre os alunos.

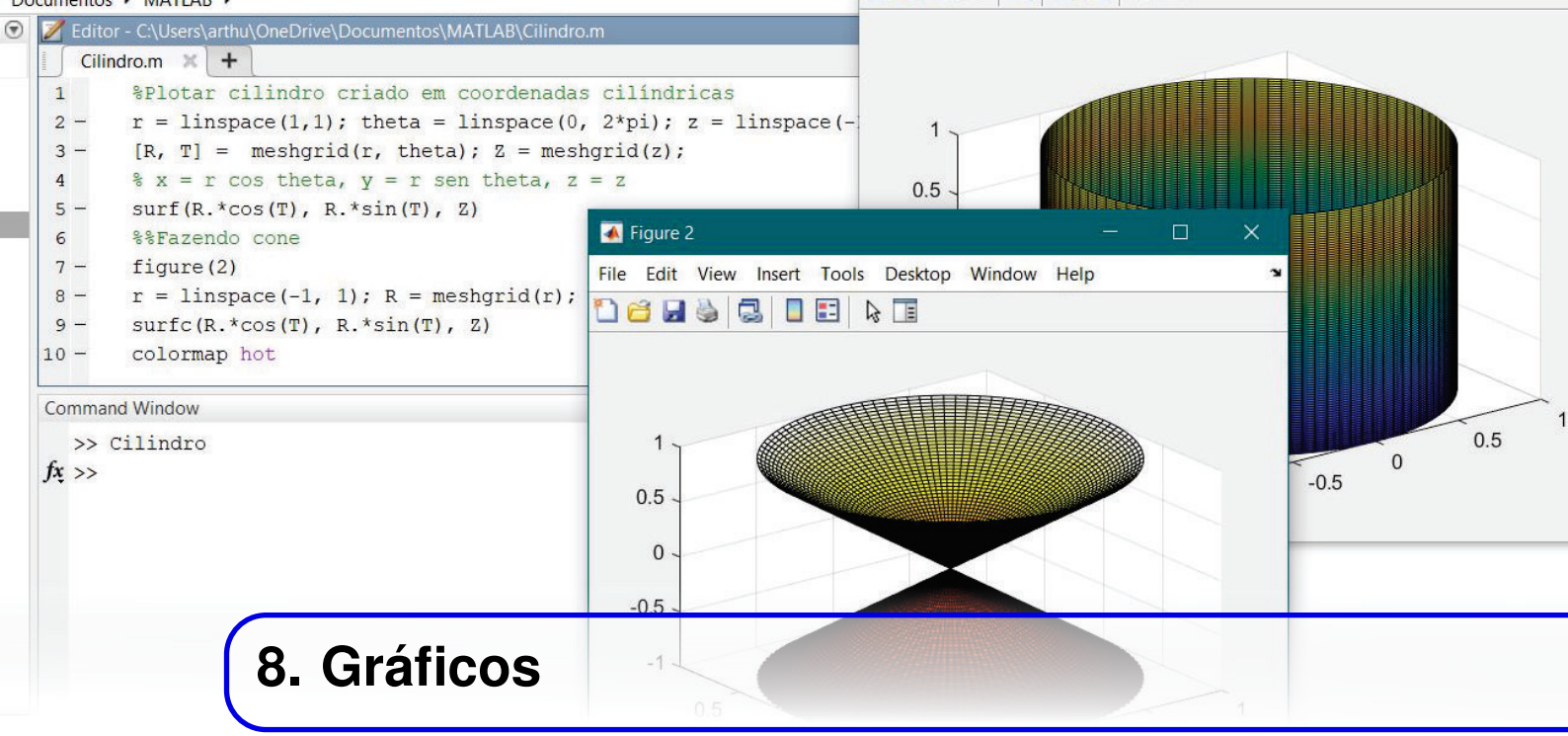
7.10 Crie um script no Matlab que peça ao usuário dois vetores lineares. Em seguida, peça que digite 1 ou 2. Caso o usuário digite 1, imprima o escalar resultante do produto interno entre os vetores. Caso o usuário digite 2, imprima a matriz quadrada resultante do produto externo desses vetores.
Note que, a diferença entre o cálculo no caso 1 ou no caso 2, pode ser uma simples transposição.

III Conceitos Avançados

8 Gráficos 73
8.1 Bidimensionais
8.2 Tridimensionais
8.3 Múltiplos Gráficos
8.4 Comandos Auxiliares
8.5 Outras Plotagens
8.6 Animações
Exercícios

9 Polinômios 97
9.1 Raízes
9.2 Produto e Divisão
9.3 Avaliação
9.4 Frações Parciais
9.5 Ajuste Polinomial
Exercícios

10 Simulink: Introdução 105
10.1 Ambiente
10.2 Modelagem
Exercícios



8. Gráficos

Os gráficos representam uma grande e famosa parte entre as utilidades do Matlab. As opções com eles, sejam em duas ou três dimensões são vastas, podendo ser realizados desde estudos de curvas até simples animações. Isso faz com que o software seja uma importante ferramenta de análise e processamento de dados.

8.1 Bidimensionais

A função principal para se plotar um gráfico bidimensional é a função plot:
plot(X, Y) plota um gráfico de X versus Y. Também pode ser usada apenas como **plot(Y)**, dessa forma serão plotados os índices de Y versus seus valores.

O principal a se compreender deste tipo de função é que ela recebe dois vetores X e Y como argumentos. Eles contêm um conjunto de pares (x, y) de pontos, os quais definem uma curva na ordem em que os pares são apresentados.

Exemplo:

```
>> %X será um vetor igualmente espaçado de zero ate 2pi, com 100
   valores, e Y o seno de X.
>> X = linspace(0, 2*pi, 100); Y=sin(X);
>> plot(X,Y)
```

Será aberta uma nova figura com o gráfico mostrado na Fig. 8.1.1.

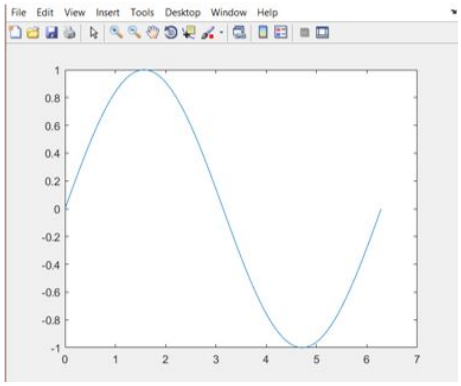


Figura 8.1.1: *Gráfico de* $y = sen(x)$

DICA: A função plot aceita um terceiro argumento que dirá o estilo do tracejado do gráfico. Os parâmetros de estilo de tracejado serão abordados a frente. Eles podem ser encontrados na tabela 8.1.1.

DICA: Com a função plot também é possível plotar várias curvas em um mesmo gráfico. Para isso, deve-se usá-la como:

```
>> plot(x1, y1, estilo1, x2, y2, estilo2, x3, y3,
   estilo3) % e assim por diante
```

A seguir serão listadas algumas funções com usos similares aos do plot, no entanto, o gráfico final apresentará características diferentes.

- **Escalas Logarítmicas**
  semilogy(X, Y) Plota um gráfico com a escala do eixo y logarítmica.

```
>> X=linspace(0, 100, 100); Y=exp(X);
>> semilogy(X, Y)
```

O resultado é mostrado pela Fig. 8.1.2

DICA: Analogamente, há a função semilogx com escala logarítmica apenas no eixo x e também há a função loglog, com escalas logarítmicas em ambos os eixos.

- **Barras e Histogramas**
  bar plota em gráfico de barras; **stairs** em degraus; **errorbar** em barras de erro; **hist** em histograma (a sequência precisa ser monótona e não decrescente) e **rose** em histograma em ângulo.

```
>> X = linspace(2, 22, 20); Y = randperm(20);  F= X.^2; %F
   servirá para o histograma
```

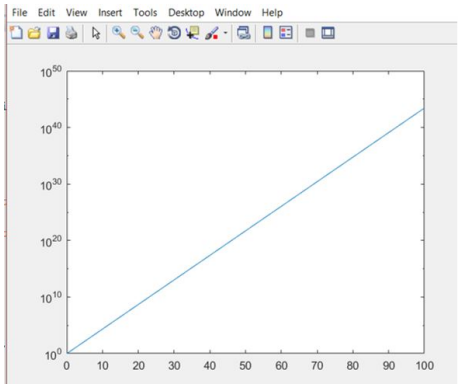


Figura 8.1.2: *Exemplo semilogy*

```
>> bar(X, Y)
>> stairs(X, Y)
>> errorbar(X, Y)
>> hist(X, Y)
>> rose(X, Y)
```

Os gráficos estão representados na Fig. 8.1.3

> **OBS:** Ao se plotar um gráfico logo após outro, observa-se que o gráfico anterior é sobrescrito pelo novo na mesma figura. Opções de como plotar múltiplos gráficos, curvas e figuras serão mostradas a frente, página 84.

- **Setas**
  compass plota o gráfico em forma de bússola e **feather** em forma de pena.

```
>> X = linspace(2, 22, 20); Y = randperm(20);
>> bar(X,Y)
>> compass(X, Y)
>> feather(X, Y)
```

Os gráficos estão representados na Fig. 8.1.4

- **Sequência Discreta**
  Para tal, há a função **stem**.

```
>> X = linspace(2, 22, 20); Y = randperm(20);
>> stem(X, Y)
```

O gráfico está representado na Fig. 8.1.5

- **Coordenadas Polares**
  Para plotar gráficos em coordenadas polares usa-se a função **polar** ou a função **polarplot**, que fazem o mesmo procedimento.

```
>> theta = linspace(0, 2*pi, 100); R=cos(2*theta);
```

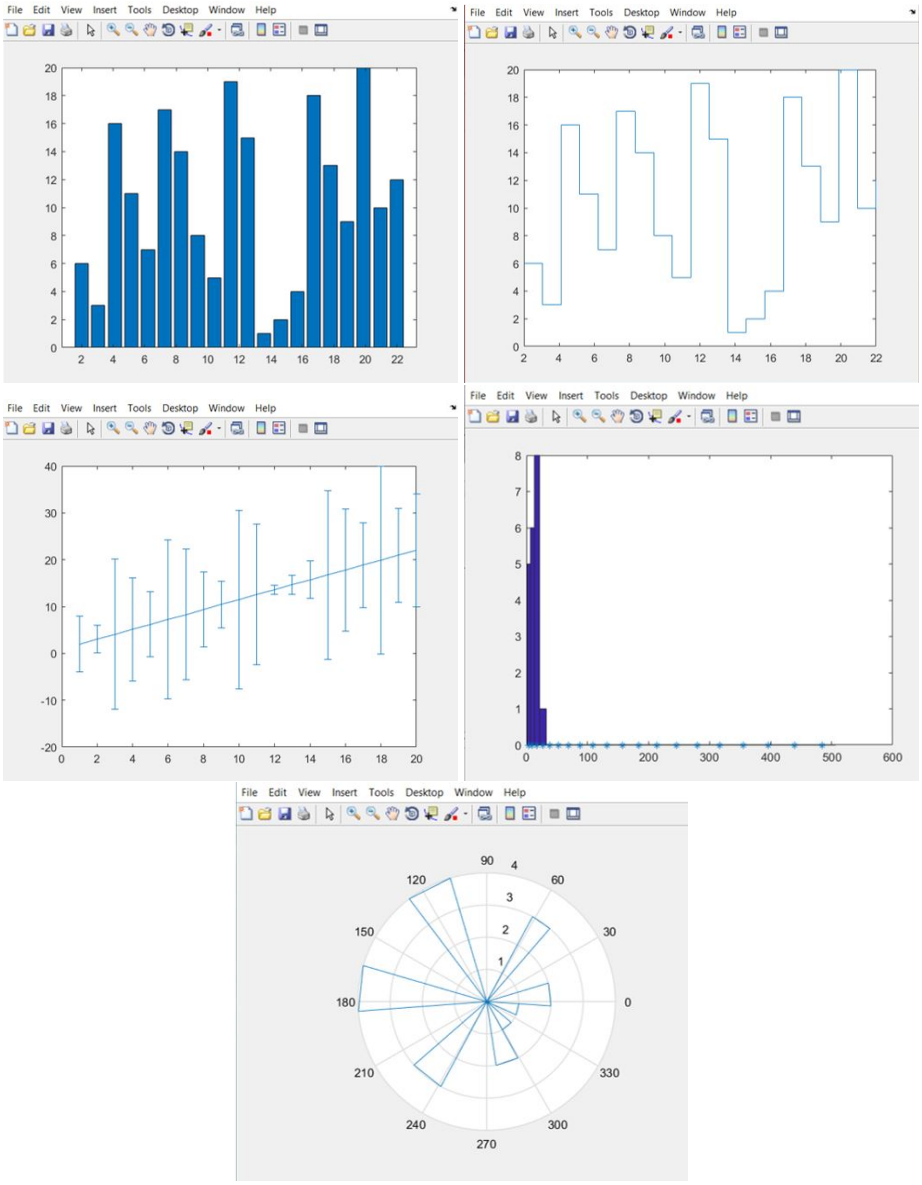


Figura 8.1.3: *Exemplos Barras e Histogramas*

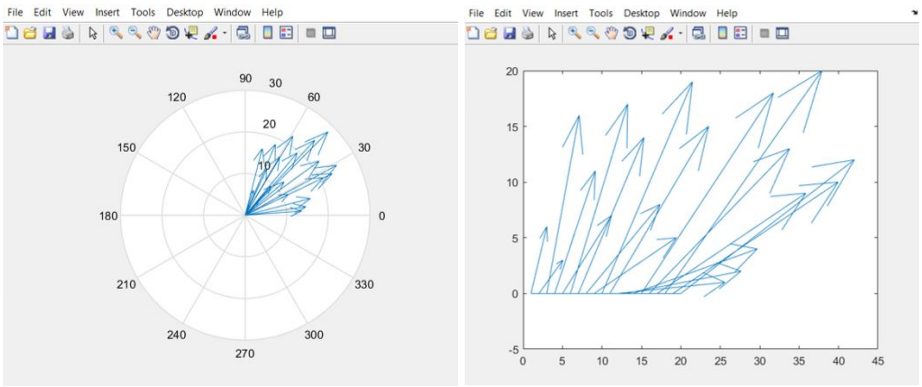


Figura 8.1.4: *Exemplos Setas*

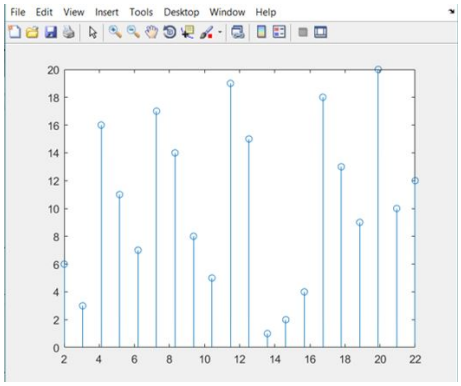


Figura 8.1.5: *Exemplo Sequência Discreta*

```
>> polar(theta, R)
```

O gráfico está representado na Fig. 8.1.6

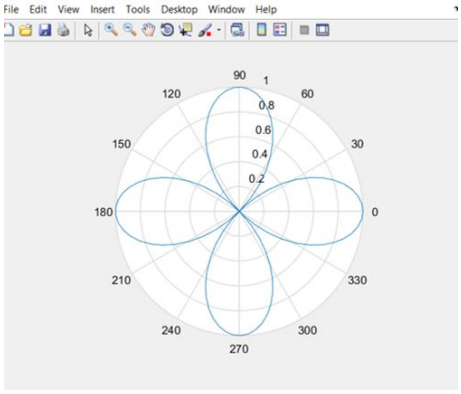


Figura 8.1.6: *Exemplo Coordenadas Polares*

- **Pizza**
  Para gráficos em pizza, usa-se a função **pie(Y, explode)**, a qual plota Y nesse estilo. O vetor explode é para destacar algumas fatias. É um arranjo do mesmo tamanho de Y, preenchido com 0 caso não se deseje destacar e 1 caso contrário. Esse último argumento pode ser omitido, nesse caso será tomando como padrão sempre 0.

```
>> x = randperm(10)
x =
     6     3     7     8     5     1     2     4     9    10
>> explode = (x == 10)
explode =
  1x10 logical array
  0   0   0   0   0   0   0   0   0   1

>> pie(x, explode)
```

O gráfico está na Fig. 8.1.7

O terceiro argumento que pode ser usado na função plot e afins é o argumento de estilo. Esse é sempre uma string com uma combinação de caracteres. Cada caracter descreve um aspecto do

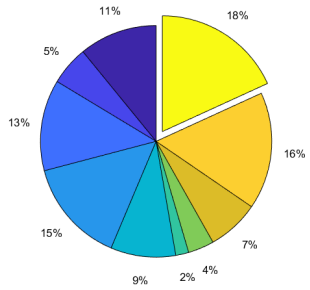


Figura 8.1.7: *Exemplo Pizza*

estilo, seja a cor ou o traçado. Os caracteres e seus efeitos estão apresentados na tabela 8.1.1.

Tabela 8.1.1: *Modificadores de Estilo*

Caractere	Tipo	Efeito
y	Cor	Amarelo
m	Cor	Lilás
c	Cor	Ciano
r	Cor	Vermelho
g	Cor	Verde
b	Cor	Azul
w	Cor	Branco
k	Cor	Preto
.	Traço	. . . . .
o	Traço	o o o o o o
*	Traço	* * * * * *
+	Traço	+ + + + + +
x	Traço	x x x x x x
s	Traço	(Quadrados)
v	Traço	(Triângulos)
P	Traço	(Estrelas)
-	Traço	(Contínua)
–	Traço	– – – – – – –
-.	Traço	-.-.-.-.-.-
:	Traço	.....

```
>> fplot('sin', [pi, 3*pi], 'k--')
```

O gráfico com o estilo $k--$ está representado pela Fig. 8.1.8

```
>> syms x
```

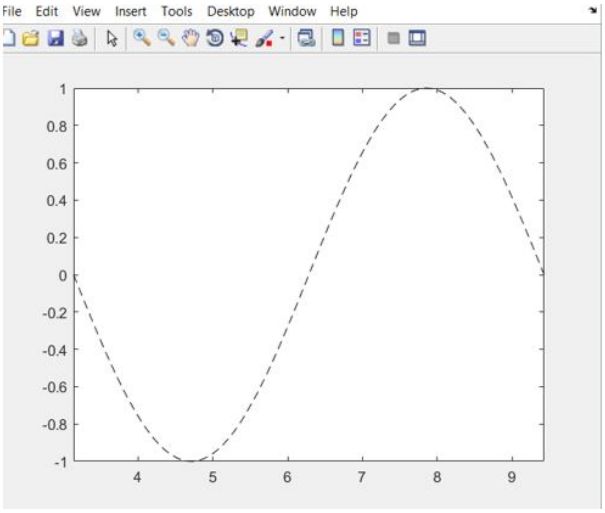


Figura 8.1.8: *Exemplo de Estilo de Curva*

```
>> f=2*x^2 - x + 2;
>> fplot(f, [-10, 5], 'g-.+')
```

O gráfico com o estilo **g-.x** está representado pela Fig. 8.1.9

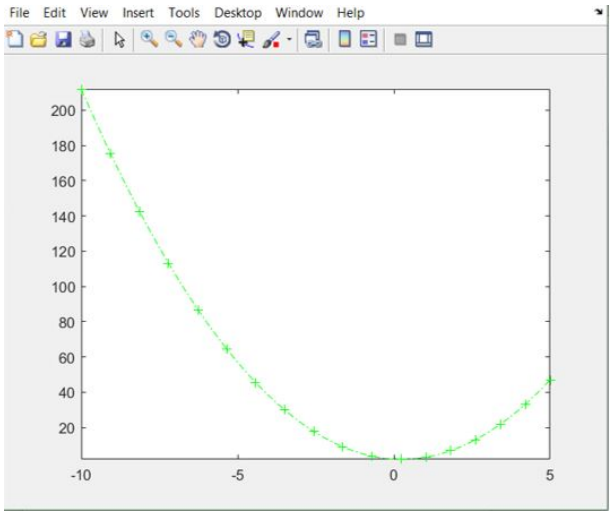


Figura 8.1.9: *Exemplo de Estilo de Curva*

DICA: Nos dois últimos exemplos foi usada a função **fplot** para plotar os gráficos. Ela é um pouco diferente da função plot. Nela, o primeiro argumento já é a função a ser plotada, em string (primeiro exemplo) ou função simbólica (segundo exemplo). O segundo argumento é o intervalo no qual se deseja plotar a função. Por fim, o último argumento é o estilo.

8.2 Tridimensionais

A progressão natural para compreender os gráficos tridimensionais, tendo entendido os bidimensionais, é plotar uma curva em três dimensões. A maneira mais simples de fazê-lo é plotar uma

curva qualquer usando os comandos já conhecidos (em duas dimensões) e, posteriormente, usar o comando:

```
>> view(3)
```

Desta forma o gráfico se transformará em um de três dimensões e a curva de duas dimensões será interpretada como outra curva cujo z sempre vale 0.

DICA: A figura gerada pelo software possui uma barra de tarefas com opções de utilizar o mouse para rodar a visualização, dar zoom, ver a escala de cores, dentre outras, o que pode variar um pouco dependendo da versão do Matlab.

Como já observado, o comando **view** altera a perspectiva da visualização da figura. Pode ser usado de algumas formas:

- **view(2)** ou **view(3)** para setar uma visualização bidimensional ou tridimensional, respectivamente.
- **view([x y z])** para setar uma visualização 3D a partir do ângulo indicado pelas coordenadas cartesianas (xyz)
- **view([Az El])** para setar o ângulo horizontal Az e a elevação El.

Para se propriamente plotar uma curva em três dimensões, na qual x, y e z variam, deve-se usar a função **plot3(x, y, z)**. Analogamente à função plot, como ela define uma curva no espaço 3D, então x, y e z devem ser três vetores os quais compõem uma sequência tríplice (x,y,z), a qual indicará uma curva.

Por exemplo, para se plotar uma hélice:

```
>>t = linspace(-2*pi, 2*pi);
>>x = 2*cos(t); y = 2*sin(t); z = t;
>>plot3(x, y, z);
>>grid on
```

O resultado é mostrado na Fig. 8.2.1.

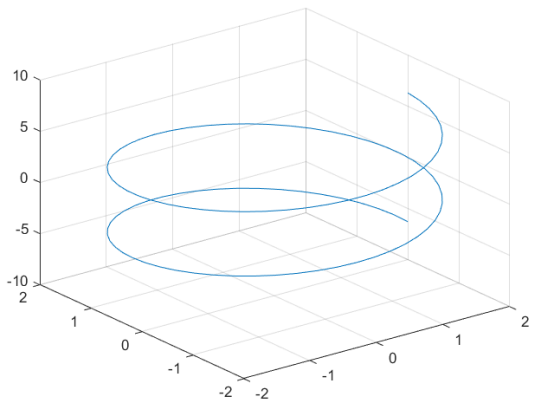


Figura 8.2.1: *Exemplo de Curva em 3D: Hélice Paramétrica*

Agora, para se plotar superfícies ao invés de curvas, são necessárias três matrizes ao invés de três vetores. É como se cada linha ou coluna das matrizes, isto é, cada vetor, formasse uma curva e esse

conjunto de curvas formasse a superfície.

Por exemplo, se forem criadas as matrizes necessárias para a curva e ainda assim usada a função plot3, serão observados várias curvas próximas que quase formam uma superfície:

```
>> a = linspace(-5,5,100);
>> [X, Y] = meshgrid(a);
>> Z = 1./((X.^2-1)+(Y.^2-1)+10);
>> plot3(X, Y, Z)
```

O resultado é mostrado na Fig. 8.2.2.

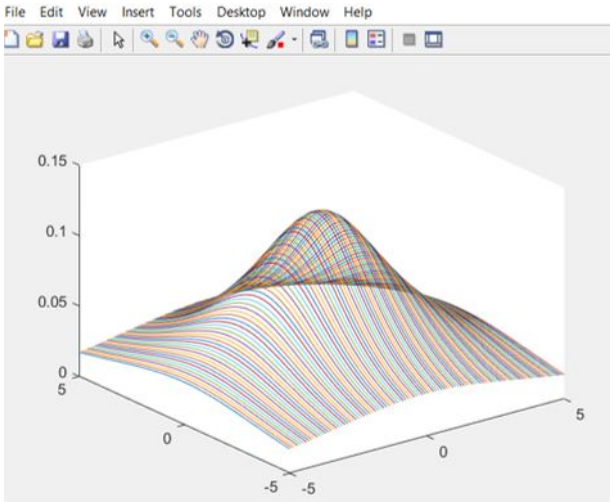


Figura 8.2.2: *Exemplo de Superfície Usando plot3*

Mas para que uma superfície seja desenhada de fato deve-se usar o função **surf(X, Y, Z)**.

```
>> X = [1, 2, 3, 4
     1, 2, 3, 4
     1, 2, 3, 4
     1, 2, 3, 4];
>> Y = [1, 1, 1, 1
     2, 2, 2, 2
     3, 3, 3, 3
     4, 4, 4, 4];
>> Z=[ 1, 2, 6, 5
    1, 3, 5, 5
    1, 2, 5, 6
    2, 4, 6, 10];
>> surf(X, Y, Z) %ou simplesmente surf(Z)
```

O resultado é mostrado na Fig. 8.2.3.
Para definir as matrizes ideais para a plotagem de modo mais fácil há a função **meshgrid**. Seu uso se dá da seguinte forma:

Se deseja criar duas matrizes ideais para plotagem, X e Y, que serão usadas para a criação de uma terceira matriz $Z = f(X,Y)$. Z está definida para $-1 \leq X \leq 2, 0 \leq Y \leq 3$. Com isso observa-se que

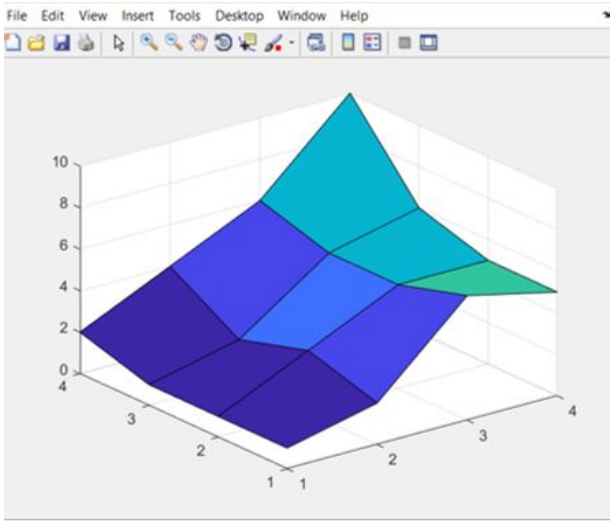


Figura 8.2.3: *Exemplo de Superfície Usando Surf*

o domínio da função que Z representa deve estar limitado de -1 a 2 no eixo x e de 0 a 3 no eixo y. Portanto, para se criar as matrizes desejadas respeitando as definições da tarefa deve-se fazer:

```
>> a = linspace(-1, 2); b = linspace(0, 3);
>> [X Y] = meshgrid(a, b)
```

No exemplo imediatamente anterior poderia ter sido usado, então:

```
>> a=[1, 2, 3, 4];
>> [X, Y]= meshgrid(a);
```

Há outras funções de plotagem para superfícies: A função **contour** plota o gráfico bidimensional de X, Y em curvas de nível; **contour3** plota as curvas de nível em três dimensões; **mesh** plota a superfície em malha; **meshc** é a junção de mesh e contour; **surfc** é a junção de surf e contour; **surfl** é a surf com iluminação.

```
>> a = linspace(-5,5,100); [X, Y] = meshgrid(a);
>> Z = 1./((X.^2-1)+(Y.^2-1)+10);
>> surf(X, Y, Z)
>> contour(X, Y, Z)
>> contour3(X, Y, Z)
>> mesh(X, Y, Z)
>> meshc(X, Y, Z)
>> surfc(X, Y, Z)
>> surfl(X, Y, Z)
```

Os gráficos citados estão representados na Fig. 8.2.4
Também é possível configurar o estilo da superfície, isto é, o sombreamento e o mapa de cores. Para se configurar o sombreamento deve-se usar o comando **shading** e o sombreamento desejado

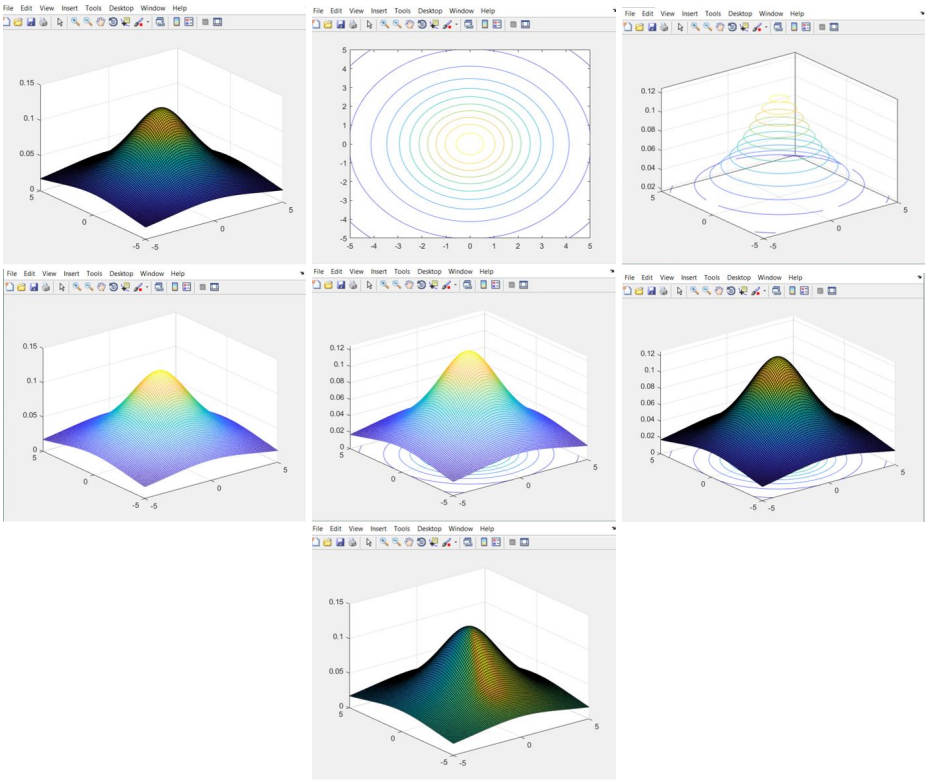


Figura 8.2.4: *Variações de Plotagem de Superfícies*

faceted, **flat** ou **interp**. Por padrão, é usado o *faceted*.

```
>> a = linspace(-5, 5); [X, Y]=meshgrid(a);
>> Z = sinc(sqrt(X.^2+Y.^2));
>> surf(X, Y, Z) %shading faceted
>> shading interp
>> shading flat
```

As variações de sombreamento estão representadas na Fig. 8.2.5.

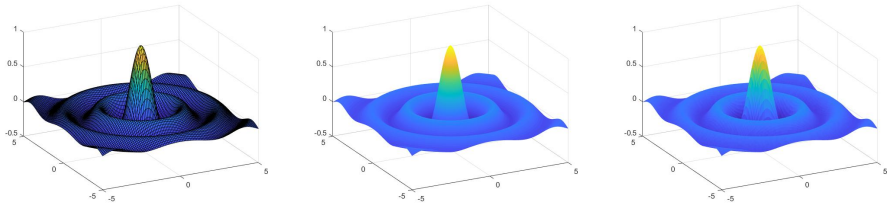


Figura 8.2.5: *Variações de Sombreamento de Superfícies*

O mapa de cores representa qual deve ser a cor da superfície à medida em que a altura (valor no eixo z) variar. O padrão é uma variação de azul até amarelo, esse mapa de cores é chamado de *default*. Para se alterar o mapa de cores pode se fazer de uma das duas formas seguintes ao usar um dos mapas predefinidos pelo Matlab:

```
>> colormap default
>> colormap('default')
```

Existem 19 variações de colormap pré-definidas pelo Matlab:
default, parula, jet, hsv, hot, cool, spring, summer, autumn, winter, gray, bone, copper, pink, lines, colorcube, prism, flag, white.

É possível também definir um colormap. Ele é definido por uma matriz com 3 colunas. Desta forma, cada linha representa uma tríplice RGB, em que cada elemento pode assumir valores de 0 até 1. A primeira linha da matriz representará a cor da parte mais baixa da superfície, enquanto que a última linha representará a cor da parte mais alta. As linhas intermediárias, portanto, serão referentes aos níveis intermediários e da variação.

Sendo assim, para se fazer um colormap que varie gradualmente de preto(ou um tom muito escuro) até vermelho, deve-se fazer uma matriz parecida com:

$$M = \begin{pmatrix} 0 & 0 & 0 \\ . & . & . \\ 1 & 0 & 0 \end{pmatrix}$$

E as linhas intermediárias devem incrementar gradualmente a primeira coluna. Uma boa forma de criar este colormap, é, portanto:

```
M = [linspace(0.2, 1, 64);linspace(0.2, 0.2, 64);linspace(0.2,
    0.2, 64)]';
```

E este colormap pode ser aplicado a uma superfície qualquer:

```
>> r = (0:0.025:1)'; theta = pi*(-1:0.05:1);
>> z = r*exp(i*theta); w = z.^3;
>> surf( real(z), imag(z), real(w))
>> colormap(M)
>> colorbar
```

O gráfico com a cor resultante é mostrado na Fig. 8.2.6.

OBS: O comando **colorbar** mostra o mapa de cores à esquerda da figura.

8.3 Múltiplos Gráficos

Foi observado anteriormente que ao se usar em sequência um comando para a plotagem de um novo gráfico, a figura anterior era sobrescrita pela nova. Para poder se plotar mais de um gráfico, existem, basicamente três formas de se fazer.

1. Várias curvas(ou superfícies) em um mesmo gráfico
   Para tal, pode-se usar a função plot com vários pares de x e y, como explicado anteriormente ou pode-se usar o comando **hold on**. Ao abrir o gráfico em uma figura e usar o comando uma única vez, a figura entrará em outro modo. Neste novo modo de *hold on*, ao se plotar um novo gráfico utilizando outra função de plotagem, as novas curvas(ou superfícies), coexistirão no

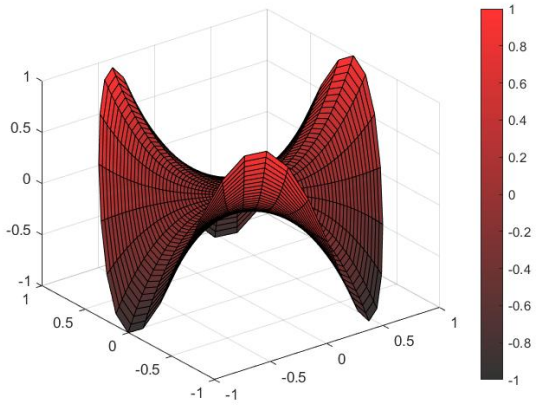


Figura 8.2.6: *Exemplo de Alteração do Mapa de Cores*

mesmo gráfico. Exemplo: x1 e y1 definem uma curva, x2 e y2 outra curva e x3 e y3 uma terceira curva diferente das anteriores:

```
>> plot(x1, y1)
>> hold on
>> plot(x2, y2)
>> plot(x3, y3)
>> hold off
```

OBS: Usar o comando **hold off** não é obrigatório, ele apenas desativa o modo hold.

2. Vários gráficos em uma mesma figura
   A segunda maneira é subdividir uma figura em vários gráficos, para isso usa-se o comando **subplot(m,n, I)**. O comando divide a figura em uma matriz m x n, onde cada posição será um gráfico, e já seleciona a posição I. Exemplo:

```
>> a = linspace(-5,5); [X, Y] = meshgrid(a); Z =
   1./((X.^2-1)+(Y.^2-1)+10);
>> subplot(2, 1, 1)
>> surf(X, Y, Z)
>> shading interp
>> subplot(2, 1, 2)
>> contour(X, Y, Z)
```

   A figura gerada ao final é mostrada na Fig. 8.3.1.
3. Várias figuras
   A terceira forma é criando várias figuras. Para isso, antes de cada gráfico deve-se usar o comando **figure(I)**, em que I indicará a figura a ser trabalhada. Em outras palavras, cada figura é uma janela a ser aberta. Para se plotar o primeiro gráfico, pode-se fazer **figure(1)**, o segundo **figure(2)** e assim por diante. Dessa maneira, haverão várias figuras abertas. Para se voltar a uma figura anterior e modificá-la, deve-se chamá-la novamente pelo comando figure e utilizar o seu número. Se o comando for usado sem argumento, isto é, apenas **figure()**, uma nova figura é aberta, independente de quantas já houverem.

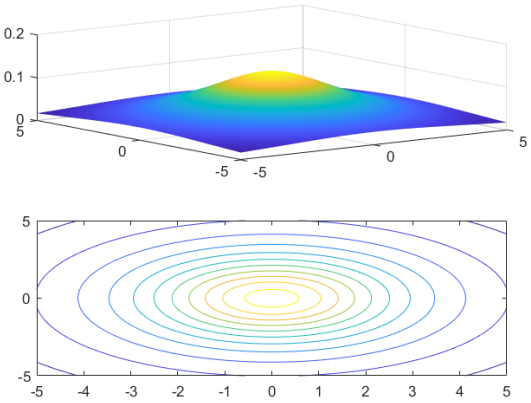


Figura 8.3.1: *Dois gráficos em uma mesma figura: subplot*

8.4 Comandos Auxiliares

Há alguns comandos auxiliares dos gráficos, como **hold** e **view** já apresentados anteriormente.

- grid
  O comando grid pode ser usado como **grid on** ou **grid off** que liga ou desliga as linhas de grade do gráfico.
- axis
  O comando axis altera as dimensões dos eixos. Pode ser usado como: **axis([XMIN XMAX YMIN YMAX])** define os limites do eixo x e y;
  axis([XMIN XMAX YMIN YMAX ZMIN ZMAX]) define os limites do eixo x, y e z (para gráficos tridimensionais);
  axis([XMIN XMAX YMIN YMAX ZMIN ZMAX CMIN CMAX]) define os limites do eixo x, y e z e os limites da escala de cores; **axis on** ou **axis off** liga ou desliga os eixos da figura;
  axis square iguala os tamanhos (não os intervalos) dos eixos;
  axis equal iguala os intervalos dos eixos;
  axis auto retorna os eixos para a configuração default;
  axis normal retira as edições feitas.
- title
  O comando insere ou modifica o título da figura, recebendo o novo título em formato de string:

```
>> title('NovoTitutlo')
```

- xlabel, ylabel, zlabel
  Estes comandos inserem ou modificam os escritos nos eixos x, y ou z:

```
>> xlabel('eixo x')
>> ylabel('eixo y')
>> zlabel('eixo z')
```

- gtext O comando insere o texto informado em uma posição indicada pelo mouse.
- text O comando insere o texto nas coordenadas cartesianas indicadas. Pode ser usado da seguinte forma:

```
>> text(x, y, z, 'O texto sera inserdo nas coordenadas
   indicadas')
```

Para os comandos de modificação de texto, isto é, title, xlabel, ylabel, zlabel, gtext e text, o Matlab suporta um subconjunto de marcadores TeX. Portanto, é possível adicionar aos gráficos caracteres especiais e letras gregas, bem como sobrescritos, subscritos e modificar o tipo de texto utilizando a marcação TeX.

Para isso, ao usar tais funções deve-se adicionar mais dois parâmetros no final: 'Interpreter' e 'latex'. Exemplo:

```
>> a = linspace(-1.5, 1.5);
>> [X Y] = meshgrid(a);
>> Z = X.^2;
>> mesh(X, Y, Z);
>> xlabel('$ x (m) $', 'Interpreter', 'latex')
>> ylabel('$ y (m) $', 'Interpreter', 'latex')
>> zlabel('$ z (m) $', 'Interpreter', 'latex')
>> title('$z=x^{2}$', 'Interpreter', 'latex')
```

O gráfico resultante é mostrado na Fig. 8.4.1.

Alguns modificadores TeX suportados pelo Matlab estão mostrados na Fig. 8.4.2 e alguns caracteres especiais na Fig. 8.4.3.

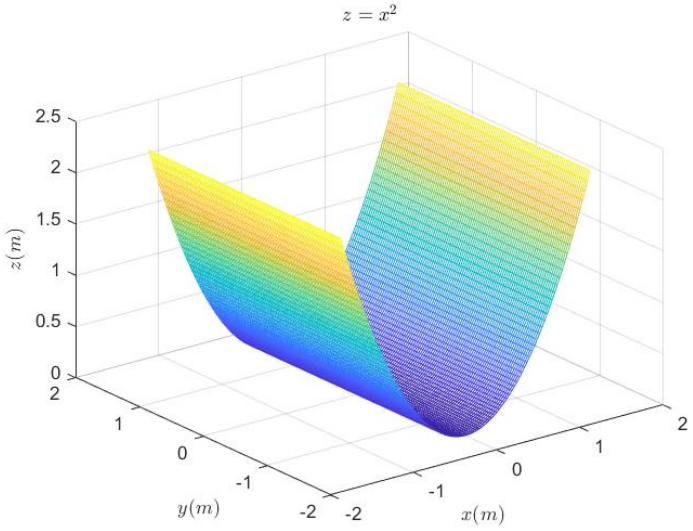


Figura 8.4.1: *Usando marcadores TeX em gráficos*

^{}	Sobrescrito	'texto^{sobrescrito}'
_{}	Subscrito	'texto_{subscrito}'
\bf	Negrito	'\bf texto'
\it	Itálico	'\it texto'
\sl	Oblíqua(geralmente itálico)	'\sl texto'
\rm	Fonte padrão	'\rm texto'
\fontname{fonte}	Alterar a fonte	'\fontname{Calibri} texto'
\fontsize{tamanho}	Alterar o tamanho	'\fontsize{11} texto'
\color{nome}	Alterar a cor	'\color{red} texto'
\color[rgb]{valor}	Alterar a cor	'\color[rgb]{0.2, 0, 0.5} texto'

Figura 8.4.2: *Marcadores TeX Suportados*

8.5 Outras Plotagens

8.5.1 Funções Implícitas

Até agora foram plotados gráficos nos quais a função que descrevia a curva era da forma $y = f(x)$ ou a superfície era da forma $z = f(x,y)$. No entanto, é desejável também plotar funções implícitas, isto é, funções nas quais não se é possível isolar alguma variável para que ela seja função das outras. A equação de um círculo de raio unitário é um exemplo simples: $x^2 + y^2 = 1$.

DICA: Neste caso do círculo de raio unitário, pode-se isolar y (ou x) na forma $y = \pm\sqrt{1-x^2}$, plotar a parte positiva e a negativa de y, dividindo a curva em dois gráficos. Mas esta forma não se aplica a todas as funções implícitas.

```
>> x = linspace(-1, 1);
>> y = sqrt(1-x.^2);
>> plot(x, y, 'k-', x, -y, 'k-')
```

Para curvas e gráficos bidimensionais, pode-se criar uma outra variável g de forma que $g = f(x,y)$ e plotar a curva de nível k desejada $g = k$. No caso de círculos, $f(x,y) = x^2 + y^2$, para o círculo de raio unitário, deve-se pegar a curva de nível k=1 deste função. O primeiro passo é avaliar a variável g, em seguida deve-se usar a função **contour** para se plotar a curva, especificando que se deseja plotar a curva de nível 1.

```
>> [x y] = meshgrid(linspace(-1,1));
>> g = x.^2 + y.^2;
>> contour(x, y, g, [1 1])
```

O resultado está mostrado na Fig. 8.5.1.
Outra forma de fazer este mesmo procedimento é usar a função **ezplot**. Seu argumento será a própria função $f(x,y)$, em formato string. Ela plotará a curva de nível 0 da função. Desta forma, para replicar a curva anterior é necessário mudar ligeiramente $f(x,y)$ para que sua curva de nível 0 seja o círculo desejado, ao invés do nível 1.

\alpha	α	\upsilon	υ
\angle	∠	\phi	ϕ
\ast	*	\chi	χ
\beta	β	\psi	ψ
\gamma	γ	\omega	ω
\delta	δ	\Gamma	Γ
\epsilon	ϵ	\Delta	Δ
\zeta	ζ	\Theta	Θ
\eta	η	\Lambda	Λ
\theta	θ	\Xi	Ξ
\vartheta	ϑ	\Pi	Π
\iota	ι	\Sigma	Σ
\kappa	κ	\Upsilon	ϒ
\lambda	λ	\Phi	Φ
\mu	μ	\Psi	Ψ
\nu	ν	\Omega	Ω
\xi	ξ	\forall	∀
\pi	π	\exists	∃
\rho	ρ	\ni	∋
\sigma	σ	\cong	≅
\varsigma	ς	\approx	≈
\tau	τ	\Re	ℜ
\equiv	≡	\oplus	⊕
\Im	ℑ	\cup	∪
\otimes	⊗	\subseteq	⊆
\cap	∩	\in	∈
\supset	⊃	\lceil	⌈
\int	∫	\cdot	·
\rfloor	⌋	\neg	¬
\lfloor	⌊	\times	x
\perp	⊥	\surd	√
\wedge	∧	\varpi	ϖ
\rceil	⌉	\rangle	〉
\vee	∨	\langle	〈
\sim	~	\propto	∝
\leq	≤	\partial	∂
\infty	∞	\bullet	•
\clubsuit	♣	\div	÷
\diamondsuit	♦	\neq	≠
\heartsuit	♥	\aleph	ℵ
\spadesuit	♠	\wp	℘
\leftrightarrow	↔	\oslash	∅
\leftarrow	←	\supseteq	⊇
\Leftarrow	⇐	\subset	⊂
\uparrow	↑	\o	o
\rightarrow	→	\nabla	∇
\Rightarrow	⇒	\ldots	...
\downarrow	↓	\prime	′
\circ	º	\0	∅
\pm	±	\mid	|
\geq	≥	\copyright	©

Figura 8.4.3: *Caracteres Especiais TeX Suportados*

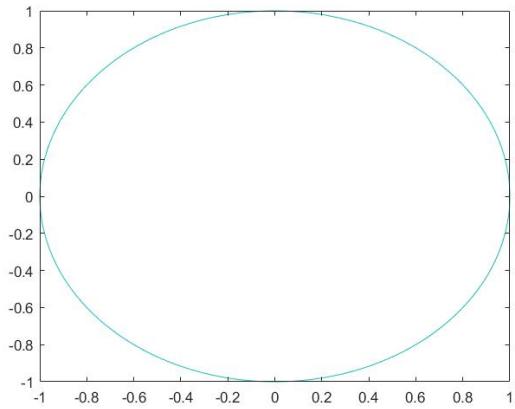


Figura 8.5.1: *Curva da função implícita* $x^2 + y^2 = 1$*, usando contour*

```
>> ezplot('x.^2 + y.^2 - 1')
```

O resultado está mostrado na Fig. 8.5.2.

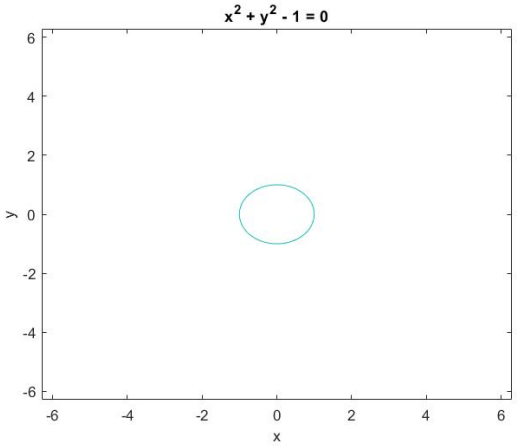


Figura 8.5.2: *Curva da função implícita* $x^2 + y^2 - 1 = 0$*, usando ezplot*

Por fim, para plotar uma superfície que está descrita em uma função implícita, usa-se a função **isosurface**. O procedimento é um pouco mais complexo, mas analogamente ao exemplo anterior, será plotado uma casca de esfera de raio unitário, para facilitar a compreensão.

Primeiro cria-se os arranjos de 3 dimensões X, Y e Z usando a função meshgrid. Em seguida, uma variável W é calculada, de forma que $W = f(X,Y,Z) = X^2 + Y^2 + Z^2$. Por fim, é plotada a superfície de nível 1 do conjunto X, Y, Z, W e depois são feitas apenas alterações para melhorar a visualização.

```
>> [X Y Z] = meshgrid(linspace(-1.5,1.5));
>> W = X.^2 + Y.^2 + Z.^2;
>> sup = isosurface(X, Y, Z, W, 1);
>> graf = patch(sup);
>> view(3)
```

```
>> camlight; axis square; grid on
>> set(graf, 'FaceColor', 'blue'); set(graf, 'EdgeColor', 'none')
```

O resultado está mostrado na Fig. 8.5.3.

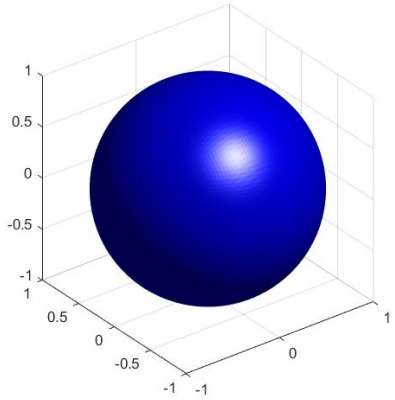


Figura 8.5.3: *Superfície da função implícita* $x^2 + y^2 + z^2 = 1$*, usando isosurface*

8.5.2 Polígonos

É possível plotar figuras preenchidas com as funções **fill** e **fill3**. A diferença entre elas é que a primeira recebe apenas os parâmetros x e y e plota em duas dimensões, enquanto que a segunda plota em três dimensões, recebendo x, y e z. Ambas as funções devem receber um último argumento especificando a cor.

Os parâmetros x e y (ou x, y e z) devem representar um conjunto de pontos. Ao percorrer esse conjunto de pontos, em ordem, deve-se ligá-los, ligando também o último ao primeiro. A figura geométrica formada será preenchida.

Por exemplo, para se desenhar um quadrado em duas dimensões, pode-se usar os pontos:
(-1, 1), (1, 1), (1, -1), (-1, -1).
Sendo assim, então:

```
>> x = [-1 1 1 -1];
>> y = [1 1 -1 -1];
>> fill(x, y, 'r')
>> axis([-2 2 -2 2])
>> axis square
```

O resultado está mostrado na Fig. 8.5.4.
Desde que os pontos estejam na ordem e definam a figura geométrica desejada, ela será plotada.

8.5.3 Campos Vetoriais

Para se plotar campos vetoriais em duas dimensões é necessário dois pares para cada vetor: (x, y) e (u, v). O primeiro par indicará a posição que o vetor estará plotado no gráfio, enquanto que o

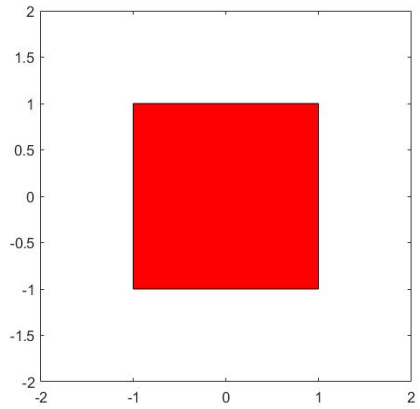


Figura 8.5.4: *Uso da função fill para plotar um quadrado*

segundo indicará o seu módulo e sua direção. Desta forma, sendo os arranjos x, y, u e v de vários vetores, pode-se plotá-los usando **quiver(x, y, u, v)**.

```
>> t = linspace(0, 2*pi, 20);
>> x = cos(t); y = sin(t);
>> u = -sin(t); v = cos(t);
>> quiver(x, y, u, v, 'm', 'LineWidth', 1, 'AutoScaleFactor', 0.4)
>> grid on
```

O resultado está mostrado na Fig. 8.5.5.

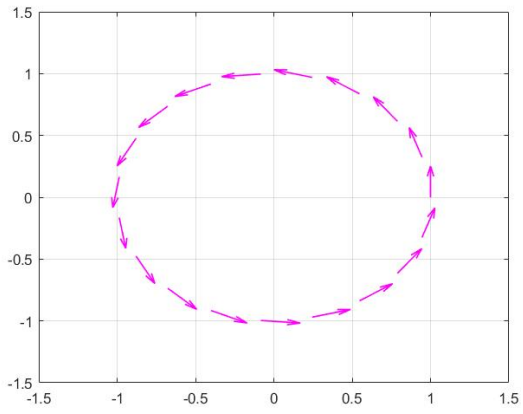


Figura 8.5.5: *Exemplo de Campo Vetorial*

Para um campo vetorial em três dimensões há a função **quiver3(x, y, z, u, v, w)**, que recebe a tríplice (x, y, z) indicando a posição de cada vetor e a tríplice (u, v, w) indicando seus valores.

8.6 Animações

Ao plotar vários gráficos em sequência em uma figura, em que cada novo gráfico sobrescreve o anterior, tem-se uma animação. Isto pode ser feito, por exemplo, para representar a trajetória 3D de um corpo ao longo do tempo, sendo possível ter uma melhor visualização do movimento ao invés de três gráficos lineares em x, y e z.

No entanto, ao invés de plotar um novo gráfico toda vez, há uma maneira melhor de criar a animação: alterando os valores do gráfico já plotado. Para isso, deve-se cumprir uma boa prática de programação em Matlab: atribuir o resultado de uma plotagem a uma variável, por exemplo:

```
grafico = plot(x, y);
```

Desta forma, a variável **grafico** será usada para modificar os valores. Enquanto a janela do gráfico ainda não foi fechada é possível observar vários componentes da variável. Clicando em Show all properties é possível visualizar ainda mais.

```
>> x = 0:.01:1; y = x.^2;
>> grafico = plot(x, y);
>> grafico

grafico =

  Line with properties:

              Color: [0 0.4470 0.7410]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [1x101 double]
              YData: [1x101 double]
              ZData: [1x0 double]

  Show all properties
```

Dependendo da função utilizada para desenhar o gráfico, plot, mesh, fill, stem, etc. as propriedades da variável serão diferentes.

Sendo o gráfico atribuido a uma variável, é possível alterar sua prioridade como valor em x, y, z, cor, transparência, etc usando a função set. Usando a função set dentro de um loop é possível visualizar a animação:
No exemplo a seguir, é plotado um quadrado vermelho idêntico à Fig. 8.5.4. Daí em diante, a função set é usada em um for para alterar a *FaceColor*, isto é, a cor do quadrado para que ele se torne mais azul. **Animacao1.m**

```
1 x = [-1 1 1 -1];
2 y = [1 1 -1 -1];
```

```
3
4  R = 1:-0.01:0;
5  G = 0*R;
6  B = 0:0.01:1;
7
8  cores = [R', G', B'];
9
10 graf = fill(x, y, cores(1, :));
11 axis([-2 2 -2 2])
12 axis square
13
14 for i = 1:101
15    set(graf, 'FaceColor', cores(i, :))
16    pause(.05)
17 end
```

Já no segundo exemplo, a função set é usada para alterar o *XData*, *YData* e *ZData* para que o ponto plotado se mova em uma trajetória de hélice. **Animacao2.m**

```
1  t = 1:1:1000;
2  x = cos(t/10); y = t/1000; z = sin(t/10);
3  graf = plot3(x(1), y(1), z(1), 'ro','MarkerSize', 5);
4  grid on
5  axis([-2 2 -2 2 -2 2])
6
7  for i = t
8      set(graf, 'XData', x(i));
9      set(graf, 'YData', y(i));
10     set(graf, 'ZData', z(i));
11     pause(.1)
12 end
```

Exercícios

8.1 O Matlab possui funções para se plotar algumas figuras tridimensionais. Utilize o comando help, caso necessário, e use as seguintes funções para plotar alguns sólidos:
a) sphere;
b) cylinder;
c) ellipsoid.

8.2 Utilizando a função plot, plote em um mesmo gráfico a função seno, na cor azul, e a função cosseno, na cor vermelho. Ambas, no intervalo de $(0, 2\pi)$.

8.3 Coloque na imagem da questão anterior:
a) Título "Seno e Cosseno";
b) Legenda no eixo x "0 a 2π";
c) Legenda no eixo y "-1 a 1".

8.4 Realize a sequência de comandos e observe o gráfico:
a) k = 1:20;
b) k_str = num2str(k);

c) f = ['sinc([', k_str, ']*x)'];
d) fplot(f, [-1 1]).

8.5 Plote três gráficos em três figuras diferentes com as seguintes especificações:
a) $y = x^2 - 2x + 5$, de -10 a 10, de azul;
b) $y = \frac{2^x}{x}$, de -5 a 5, de vermelho;
c) $y = sinc(x)$, de -50 a 50, de verde.

8.6 A superfície cone é definida como: $\frac{x^2}{a^2} + \frac{y^2}{b^2} = \frac{z^2}{c^2}$. Plote o gráfico dessa função utilizando a função mesh, sendo a=2, b=3 e c=5. Adicione ainda, nesse mesmo gráfico, o contorno 2D. Dica: isole z, plote o gráfico de z positivo, congele a imagem, plote a parte negativa.

8.7 Plote o gráfico de uma superfície conhecida como paraboloide hiperbólico definido por: $f(x,y) = (x-3)^2 - (y-2)^2, 2 \leq x \leq 4, 1 \leq y \leq 3$ Note que sua variável z será a própria f(x,y).

8.8 A função exponencial e^x pode ser interpretada como um somatório de termos infinitos $\sum_{n=0}^{\infty} \frac{x^n}{n!}$. Esse processo é chamado de expansão em série de Taylor. Utilizando linhas tracejadas e o intervalo de -2 a 5 (para facilitar a visualização) plote, em uma mesma figura, este somatório com n variando de 0 a:
a)2;
b)3;
c)4;
d)5;
Em seguida, plote a própria função exponencial e^x , em linha contínua e no intervalo de -2 a 5, e observe como o somatório converge para a função a medida em que somamos o próximo termo. Insira legenda. Dica: expandindo o somatório teremos: $e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$

8.9 Repita o processo da questão anterior com a função cos(x), usando o intervalo de 0 a π. Sua expansão em série é descrita por: $\sum_{n=0}^{\infty} \frac{x^{2n}}{2n!}(-1)^n$, ou seja $cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$

8.10 Utilize a função randn para introduzir ruídos a uma função de primeiro grau: Uma função de primeiro grau é gerada pela forma geral $f(x) = ax + b$. Tome a=2 e b=1. Utilize um intervalo de -2, 10. Plote em uma mesma figura dois gráficos: o gráfico da função original e o gráfico da função com ruídos ($f_2(x) = ax + b + ruidos$).

$$det(B - \lambda I_3) = \begin{vmatrix} 2-\lambda & 2 & 2 \\ 0 & 0-\lambda & 0 \\ 1 & 2 & 3-\lambda \end{vmatrix} = \lambda^3 - 5\lambda^2 + 4\lambda$$

```
>> B = [2, 2, 2; 0, 0, 0; 1, 2, 3];
>> pp = poly(B)
pp =
     1     -5      4      0
>> roots(pp)
ans =
     0
     4
     1
```

9. Polinômios

O Matlab também possui ferramentas para se trabalhar com polinômios facilmente, além das que possui para se trabalhar com matrizes. Eles são representados por um vetor contendo os coeficientes do polinômio em ordem decrescente, por exemplo, $x^3 + 2x - 5$ será representado por p = [1, 0, 2, -5]. Algumas das funções para polinômios já foram expostas anteriormente, e serão revisadas e melhor detalhadas.

9.1 Raízes

Pode-se calcular as raízes de um polinômio, e também um polinômio através de raízes com os comandos **roots(P)** e **poly(R)**. As raízes que serão retornadas ou utilizadas também formam um arranjo. Exemplo:

```
>> P=[ 1, 1, 4, 0];
>> roots(P)
ans =
   0.0000 + 0.0000i
  -0.5000 + 1.9365i
  -0.5000 - 1.9365i
>> P2=poly(ans)
P2 =
    1.0000    1.0000    4.0000         0
```

Se o argumento de **poly()** for uma matriz quadrada, então ela irá retornar os coeficientes do polinômio característico dessa matriz. Isto é, seja uma matriz B definida por:

$$B = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 0 & 0 \\ 1 & 2 & 3 \end{pmatrix}$$

Então seu polinômino característico é representado por:

$$det([B]-\lambda[I_3]) = \begin{vmatrix} 2-\lambda & 2 & 2 \\ 0 & 0-\lambda & 0 \\ 1 & 2 & 3-\lambda \end{vmatrix} = \lambda^3 - 5\lambda^2 + 4\lambda$$

Desta forma, pode-se encontrar os autovalores da Matriz B calculando as raízes de seu polinômio característico:

```
>> B = [2, 2, 2; 0, 0, 0; 1, 2, 3];
>> pp = poly(B)
pp =
     1     -5      4      0
>> roots(pp)
ans =
     0
     4
     1
```

9.2 Produto e Divisão

A seção parte direto para produto e divisão porque soma e subtração podem ser feitas normalmente com dois polinômios desde que eles sejam do mesmo tamanho, ou seja, desde que o menor seja completado com zeros à esquerda. Exemplo:

$$p_1 = x^3 + 10x - 5$$

$$p_2 = x$$

$$P = p_1 + p_2$$

```
>> p1 = [1 0 10 -5];
>> p2 = [1 0]
>> P = p1 + [0 0 p2];
```

Agora, para produto e divisão deve-se usar as funções **conv(Polinômio1, Polinômio2)**, que retorna os coeficientes da multiplicação dos polinômios e **deconv(Polinômio1, Polinômio2)** que retorna dois vetores, o primeiro é o quociente da divisão entre os polinômios, e o segundo o resto, normalmente usa-se **[q, r] = deconv(y1, y2)**.

No caso da divisão, também é retornado o resto:

```
>>p1 = [1 -5 4 0]; p2 = [1 -4];
>>conv(p1, p2)
ans =
     1     -9     24    -16      0
>> [q, r] = deconv(p1, p2)
q =
     1     -1      0
r =
     0      0      0      0
```

Observa-se que o p2 é exatamente uma raiz de p1, portanto ao dividir p1 por p2, esta operação representa uma fatoração, ou seja, a retirada de uma raiz, no caso com valor 4 (e também por isso, o resto é 0).

```
>> roots(p1), roots(q)
ans =
     0
     4
     1
ans =
     0
     1
```

9.3 Avaliação

Pode-se avaliar valores em polinômios usando a função **polyval(Polinômio, Valores)**, que retornará o resultado das avaliações do polinômio nos valores fornecidos. Isto é útil se conhece uma função polinomial $P(x)$ e deseja plotar um gráfico de x versus y para determinados valores de x. Exemplo:

```
>> P=[4 4 0 2 -5]; x=0:0.5:2;
>> y = polyval(P, x)
y =
   -5.0000   -3.2500    5.0000   31.7500   95.0000
```

9.4 Frações Parciais

Transformação em frações parciais pode ser utilizada em situações como transformadas de Laplace e Fourier, ou qualquer outro problema que envolva divisão de polinômios. A ideia é transformar uma razão entre dois polinômios $\frac{B}{A}$, em uma soma de outras frações, para facilitar os cálculos.

$$\frac{B(s)}{A(s)} = \frac{10s^3 + 80s^2 + 177s + 95}{s^3 + 8s^2 + 19s + 12}$$

$$\frac{B(s)}{A(s)} = \frac{9}{s+4} + \frac{-7}{s+3} + \frac{-2}{s+1} + 10$$

Para isso há a função *residue* que pode ser usada de duas formas:

- **[r, p, k] = residue(n, d)** sendo n e d, o numerador e denominador, respectivamente, a função retorna r=resíduo, p=polo, k=termo direto;
- **[n, d] = residue(r, p, k)** sendo r=resíduo, p=polo, k=termo direto, a função retorna n e d, o numerador e denominador, respectivamente;

```
>>B=[10 80 177 95]; A=[1 8 19 12];
>> [res, pol, k]= residue(B, A)
res =
    9.0000
```

```
    -7.0000
    -2.0000
pol =
    -4.0000
    -3.0000
    -1.0000
k =
     10
```

9.5 Ajuste Polinomial

O ajuste ou regressão polinomial pode ser feito utilizando a função **polyfit(x, y, n)** que retorna os coeficientes do polinômio ajustado de grau n nos valores de x e y.

Nesta função é utilizado o método dos quadrados mínimos, no qual buscam-se os valores dos coeficientes do polinômio que fazem com que, o quadrado da distância entre os pontos fornecidos e a curva gerada, seja o mínimo possível.

CURIOSIDADE : O método dos quadrados mínimos é uma técnica de otimização que busca minimizar a soma dos quadrados das diferenças entre o valor estimado e o valor medido. Para esse procedimento, são usadas as derivadas da expressão.

```
>> x=0:0.1:1; y = [-0.447 1.978 3.28 6.16 7.08 7.34 7.66 9.56 9.48
   9.30 11.2];
>>pp2 = polyfit(x,y,2); pp3 = polyfit(x,y,3); pp10 =
   polyfit(x,y,10);
>>yp2 = polyval(pp2, x); yp3 = polyval(pp3, x); yp10 =
   polyval(pp10, x);
>>plot(x, y, 'o', x, yp2, 'r', x, yp3, 'g', x, yp10, 'b')
>> legend('sem ajuste', 'grau 2', 'grau 3', 'grau 10')
```

O gráfico resultante é mostrado na Fig. 9.5.1. No exemplo acima há um conjunto de pares x e y que podem definir uma medição com erro. Com esse conjunto x e y é feito o ajuste polinomial de graus 2, 3 e 10 usando a função polyfit para se ter os polinômios. Em cada vez que se fez o ajuste polinomial, se desejpu plotar o gráfico, mas para plotar o gráfico é necessário que haja outro conjunto x e y2 com os respectivos pontos. Para isso, usou-se a função polyval nos polinômios ajustados.

CURIOSIDADE: Na Fig. 9.5.1 nota-se que são 11 pontos, e pp10, o de azul, é o polinômio de grau 10. Pelo ajuste, este polinômio será o próprio polinômio interpolador, pois por n+1 pontos, passa somente um único polinômio de grau n.

Exercícios

9.1 Em análise numérica, a partir de um polinômio $P(x)$ o qual se deseja estimar as raízes, três polinômios auxiliares podem ser calculados $P_1(x)$, $P_2(x)$ e $P_3(x)$. Eles ajudarão a limitar as

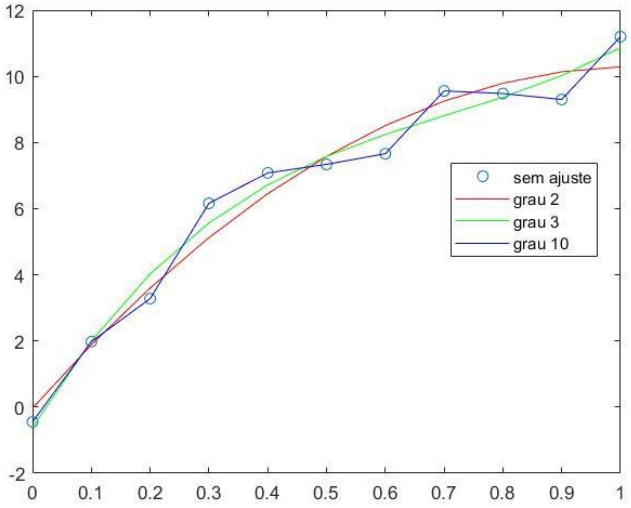


Figura 9.5.1: *Ajuste Polinomial*

raízes de $P(x)$, pois suas raízes tem certas propriedades em relação às raízes originais. Os auxiliares são definidos como:
$P_1(x) = x^n P(\frac{1}{x})$;
$P_2(x) = P(-x)$;
$P_3(x) = x^n P(\frac{-1}{x})$;
Desta forma, utilizando a função roots, encontre as raízes dos seguintes polinômios e observe a sua relação:
$P(x) = x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120$;
$P_1(x) = -120x^5 + 274x^4 - 225x^3 + 85x^2 - 15x + 1$;
$P2(x) = -x^5 - 15x^4 - 85x^3 - 225x^2 - 274x - 120$;
$P3(x) = 120x^5 + 274x^4 + 225x^3 + 85x^2 + 15x + 1$.

9.2 Há a função eig(Matriz) que retorna diretamente os autovalores de uma Matriz. Utilizando as funções poly e roots, ou seja, sem usar a função eig, encontre os autovalores das seguintes matrizes:
a)

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

; b)

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

; c)

$$C = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

; d)

$$D = \begin{pmatrix} 98 & 88 & 2.04 \\ 0.13 & -12 & 9.08 \\ 11.7 & 11.7 & 2.22 \end{pmatrix}$$

.

9.3 O ajuste polinomial coincide com a interpolação polinomial quando o grau do polinômio ajustado é suficientemente grande, a saber, por n+1 pontos passa somente um polinômio interpolador de grau n.
Tomando os pares de pontos (x, y) como:
x = 0.0 0.1 0.2 0.3 0.4 0.5
y = 15.0317 15.0234 15.0710 15.1970 15.3796 15.5509
Encontre:
a) O polinômio interpolador, usando ajuste polinomial;
b) O ajuste polinomial de grau 1;
c) O ajuste polinomial de grau 2.

9.4 Tendo os polinômios da questão anterior, crie um vetor H de 100 elementos igualmente espaçados entre eles e o avalie em cada um dos três polinômios encontrados. Em seguida, plote, na mesma imagem os seguintes gráficos:
a) O polinômio interpolador;
b) O polinômio de grau 1;
c) O polinômio de grau 2;
d) Os pontos x e y da questão anterior.

9.5 Realize as seguintes operações com os polinômios a, b e c:
a) $a = x^4 + 5x^3 - 5x^2 + 10$;
b) $b = x^5$;
c) $c = (x-1)(x-3)$;
d) $\frac{b}{a}$;
e) ac;
f) $10b$;
g) raízes de $5a - c$.

9.6 Decomposição em frações parciais é um processo muito importante na resolução de algumas integrais e transformações de Laplace. Use a função residue para encontrar a decomposição em frações parciais da seguinte função racional:

$$f(x) = \frac{(x+1)}{(x(x+2)(x+4))}$$

.

9.7 Faça o processo inverso da questão anterior: use a função residue e as frações parciais para chegar à função racional:

$$f(x) = \frac{-3}{8(x+4)} + \frac{1}{4(x+2)} + \frac{1}{8x}$$

.

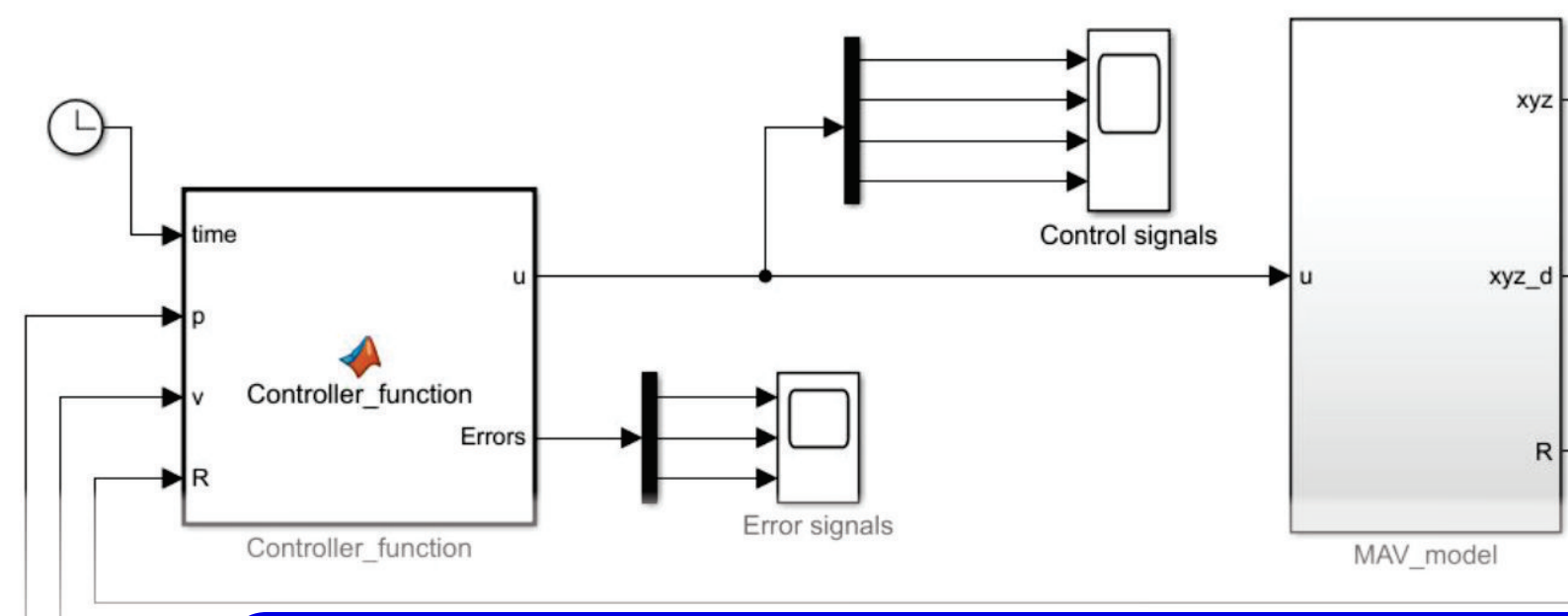
9.8 Em um determinado experimento de física, deseja-se medir a constante elástica k de uma certa mola de 10cm. Para isso, estão à disposição seis pesos de 0.5kg e uma haste com uma régua. O tamanho da mola a cada quantidade de massa colocada foi medida, e foram montados os pares (x, y) nos vetores:
x = 0 0.5000 1.0000 1.5000 2.0000 2.5000 3.0000 referente à massa em kg
y = 0 0.0550 0.1010 0.1630 0.2000 0.2520 0.29203900 referente à deformação da mola em m.

Sabendo que a força elástica de uma mola é dada segundo a equação $F = kx$, e que a força, neste experimento, é o peso $P = mg$, encontre a constante k da mola. Use $g = 9.8m/s^2$. Dicas: compare a equação da força elástica com a função $y = ax + b$. Use o ajuste polinomial. Preste atenção em quais valores do ajuste considerar e o porquê.

9.9 Plote em uma mesma imagem os pontos do experimento e a função linear devidamente ajustada. Observe o resultado.

9.10 O procedimento a seguir calcula as raízes de um polinômio f utilizando funções e variáveis simbólicas. Calcule as raízes para o mesmo polinômio, porém usando variáveis e funções polinomiais.

```
>>syms x
>> f = x^4 -3*x^3 + 4*x;
>> solve(f)
ans =
 -1
  0
  2
  2
```



10. Simulink: Introdução

O Matlab possui uma ferramenta poderosa e cheia de utilidades: o Simulink. Pode ser usado para modelar, analisar e simular sistemas dinâmicos, comportando também sistemas lineares e não-lineares. É possível simular desde uma equação diferencial até grandes sistemas mecânicos e elétricos.

É integrado ao Matlab, porque além de compartilharem funções e procedimentos, os dados de um podem ser facilmente exportados e importados para o outro.

10.1 Ambiente

Para acessar o Simulink basta digitar **simulink** na Command Window ou clicar no ícone na barra de utilidades, na parte superior. Será aberta uma janela como mostra a Fig. 10.1.1.

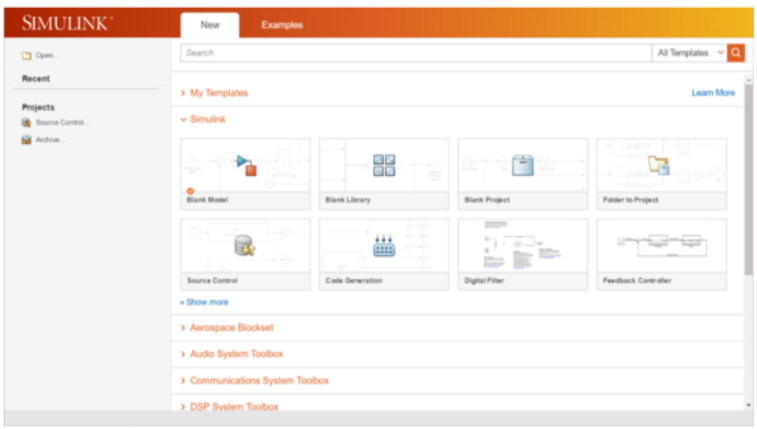


Figura 10.1.1: *Ambiente Simulink*

O Simulink possui interface gráfica do usuário, chamado de GUI (Graphical User Interface). Com isso, o usuário pode modelar e programar visualmente como se estivesse usando lápis e papel.

Na página inicial há vários modelos, templates e exemplos de sistemas. Clicando em "Blank Model" é aberto um novo modelo em branco para o usuário trabalhar. Este modelo está ilustrado na Fig. 10.1.2.

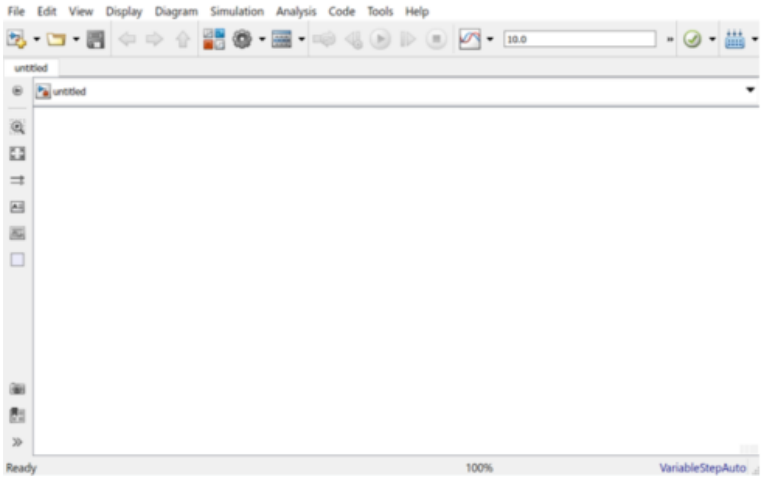


Figura 10.1.2: *Blank Model*

Modelos podem ser salvos e abertos pela aba "File". Clicando duas vezes sobre qualquer espaço branco do modelo em branco, pode-se adicionar uma anotação qualquer.

A barra superior apresenta opções úteis, como **run** – rodar a simulação, novo modelo, abrir, etc. Também há a opção **Library Browser**. Clicando nela, como mostra a Fig. 10.1.3, é aberta a biblioteca (com outras bibliotecas dentro) onde estão blocos pré-definidos:

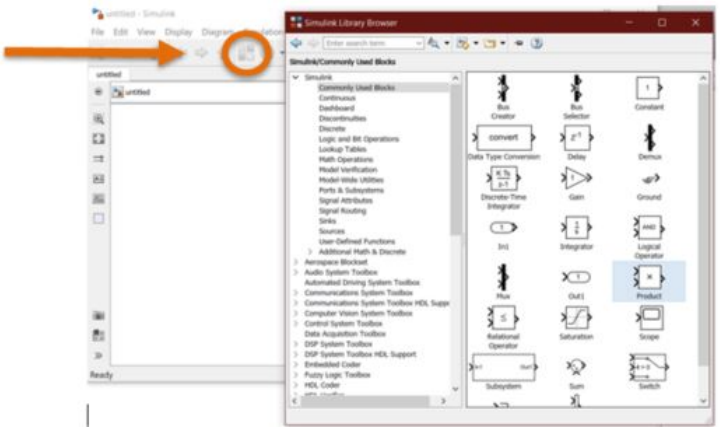


Figura 10.1.3: *Library Browser*

Aqui observa-se uma série de blocos: operações matemáticas; contínuos; discretos; fontes de sinais; atribuição de sinais; processamento de sinais; dentre vários outros.

Clicando com o botão direito sobre um determinado bloco e depois na opção **Block parameters** ou clicando diretamente duas vezes com o botão esquerdo sobre o bloco é aberto seu guia de parâmetros. Caso o bloco estivesse em uso, isto é, dentro do modelo, alguns parâmetros seriam

editáveis. Nessa aba, aparece também a descrição do bloco, muito útil para se saber o que o bloco faz, uma vez que existem diversos deles. Um exemplo de visualização de parâmetros é mostrado pela Fig. 10.1.4.

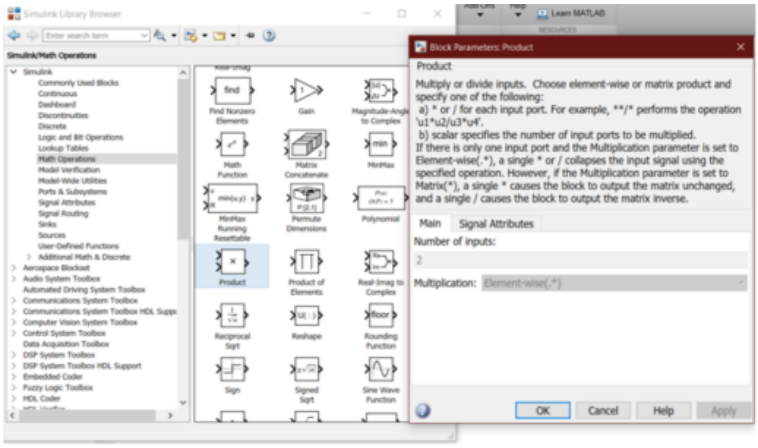


Figura 10.1.4: *Exemplo de parâmetros do bloco Product, dentro de Math Operations da biblioteca Simulink*

10.2 Modelagem

Um bloco pode ser adicionado a um modelo clicando com o botão direito sobre ele e em **Add block to model** ou simplesmente clicando e arrastando-o até o modelo. Os blocos possuem inputs e outputs, conectando-os é montado um sistema. Deve-se lembrar também que alguns parâmetros dos blocos em uso podem ser alterados clicando em "Block parameters" ou duas vezes sobre o bloco.

Para melhor compreensão da modelagem, os conceitos serão apresentados seguindo um exemplo.

10.2.1 Exemplo

Objetivo 1: Coletar dados de uma fonte senoidal.
Para a fonte senoidal será usado o bloco **Sine Wave**, encontrado em Sources.

Adicionando-o ao modelo, pode-se arrastá-lo com o mouse para qualquer posição, renomeá-lo clicando duas vezes sobre seu nome e seus parâmetros podem ser editados como mostra a Fig. 10.2.1.
Os parâmetros serão alterados: amplitude 2 e frequência para π. Note que pode usar a mesma sintaxe que no MATLAB, logo π será a constante "pi".

Agora, é preciso de um mecanismo para visualizar a onda.Será usado o bloco **Scope** de Sinks.

> **OBS:** O bloco **Display** mostra o valor recebido em tempo real, logo, seria visualizado, no fim do processo, apenas o último valor, e não toda a onda. Por isso que será usado **Scope**.

Clicando na saída do **Sine Wave** e arrastando a seta até **Scope** é transmitido o sinal do primeiro para o segundo. O tempo da simulação será alterada de 10 para 2 segundos. Isto pode ser feito na pequena janela de tempo, como indica a Fig. 10.2.2.

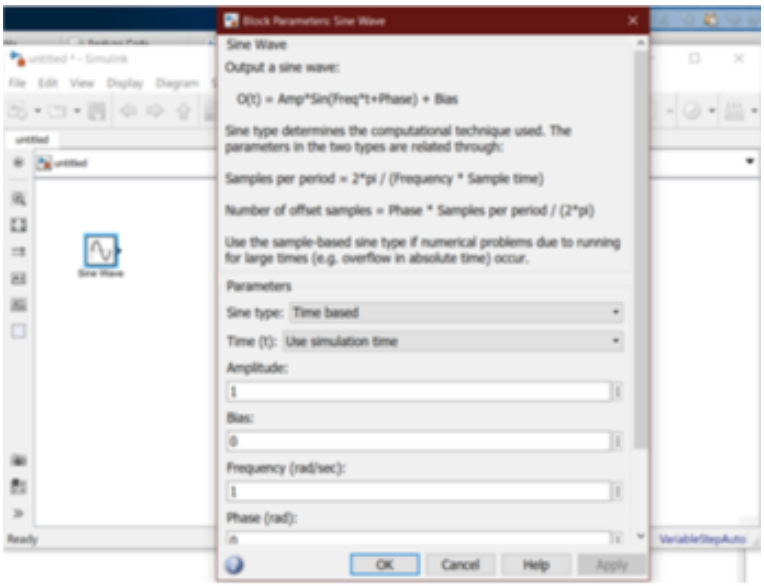


Figura 10.2.1: *Parâmetros Sine Wave*

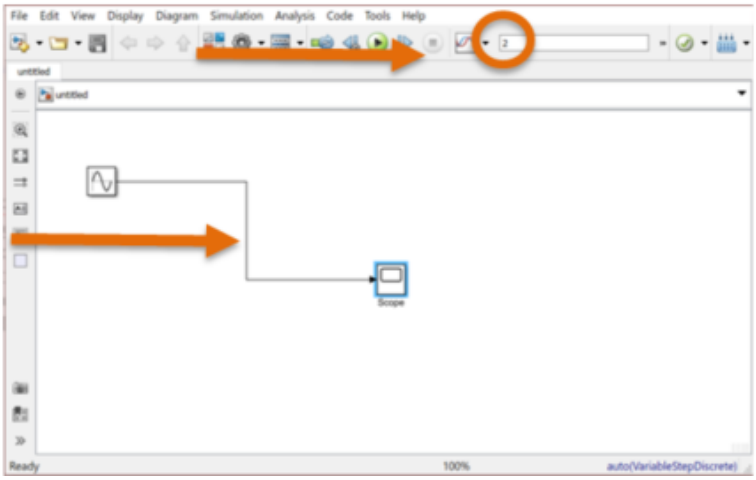


Figura 10.2.2: *Alterações no tempo de simulação*

Clicando em **Run** (círculo verde com símbolo de play) o sistema é simulado.

Para poder abrir o **Scope** e vizualizar a onda, basta clicar duas vezes sobre ele. A janela aberta irá se parecer com a Fig. 10.2.3.

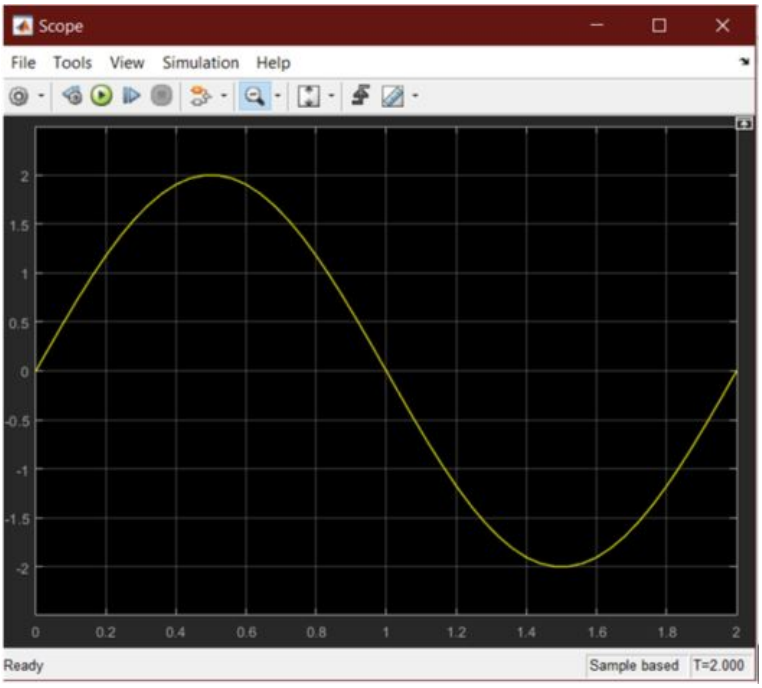


Figura 10.2.3: *Scope*

O resultado é como esperado: uma curva seno com 2 de amplitude definida entre 0 e 2π (frequência = π rad/s e 2 segundos de execução).

O primeiro objetivo está cumprido, agora será executado o segundo.

Objetivo 2: Somar duas fontes senoidas diferentes.

Primeiro, o tempo de execução será voltado para 10 segundos.

Em seguida, será somada a onda que já está configurada uma perturbação devido a outra onda, porém apenas após 2 segundos.

Para isso, vamos serão adicionados outro **Sine Wave** e os blocos **Step** de Sources, **Product** e **Add** de Math Operations.

O novo Sine Wave terá seus parâmetros configurados para: frequência de 3*pi.

DICA: O **Step** é um bloco que salta de um valor inicial para um valor final após um determinado tempo, isto é, ele representa um degrau ideal.

Será alterado somente o tempo de salto (Step time) para 2 segundos do bloco Step, mantendo o valor inicial como 0 e o final como 1.

Feito isso, o **Product** será usado para multiplicar os dois blocos alterados anteriormente, o novo **Sine Wave** e o **Step**. Assim, o valor resultante dessa multiplicação será 0 até 2 segundos e depois disso uma senoide de frequência 3π.

Por fim, o bloco **Add** irá somar a saída da multiplicação anterior e o sinal senoidal antigo. Daí, o resultado será transmitido até o osciloscópio **Scope**. O sistema ficará como mostra a Fig. 10.2.4.

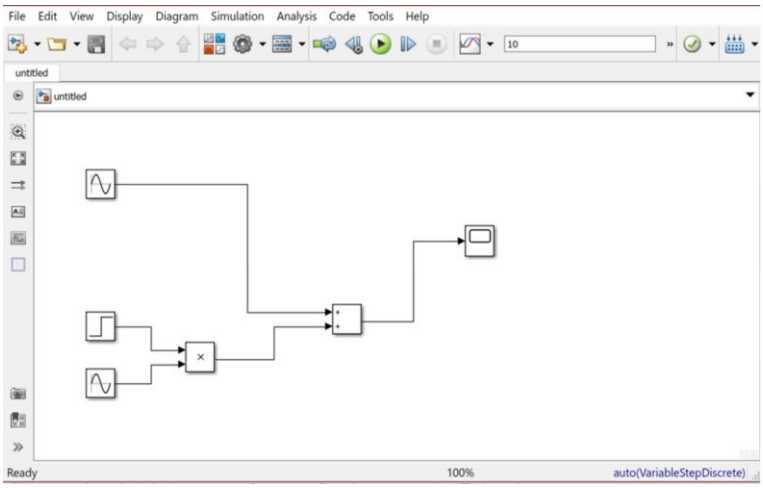


Figura 10.2.4: *Exemplo de duas fontes senoides somadas*

Ao executar o sistema clicando novamente em run, o resultado pode ser observado abrindo o Scope, como mostra a Fig.10.2.5.

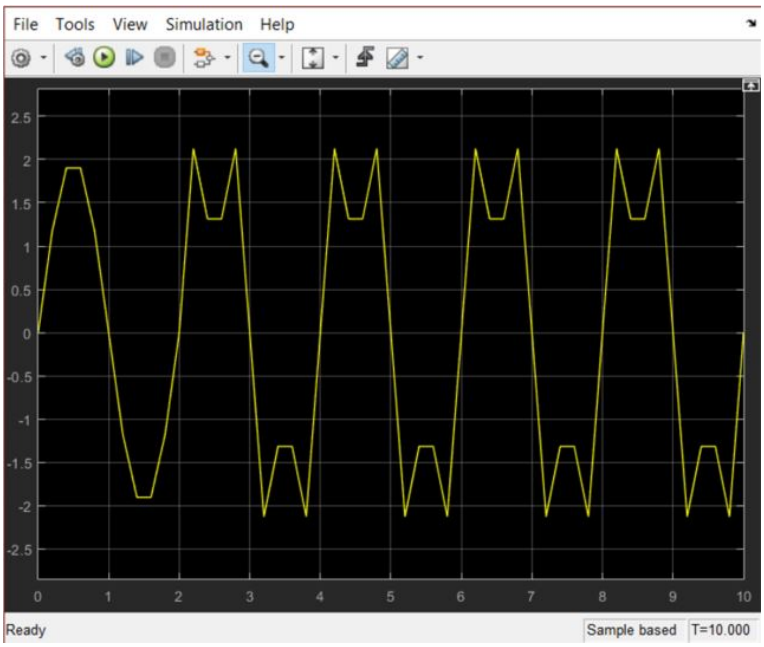


Figura 10.2.5: *Resultado mostrado pelo Scope da soma de duas fontes senoidais*

O segundo objetivo foi cumprido.

10.2.2 Intervalo de Amostragem

Ainda no exemplo anterior, na Fig. 10.2.5, pode-se observar que há algumas partes da curva resultante que se parecem com retas quebradas, e não como a curva esperada. Isso acontece porque os dados da simulação não são curvas matemáticas perfeitamente contínuas. Por se tratar de uma ferramenta computacional, a curva é um conjunto de pontos, similar aos conjuntos usados na função plot. Com isso, o espaçamento entre os pontos, isto é, o intervalo entre as amostras é tal que é possível observar essas imperfeições.

É possível alterá-lo seguindo alguns passos. Primeiro, clicando em **Model configuration parameters** e alterando o tipo de passo do modelo de **Variable-step** para **Fixed-step**, como mostra a Fig. 10.2.6.

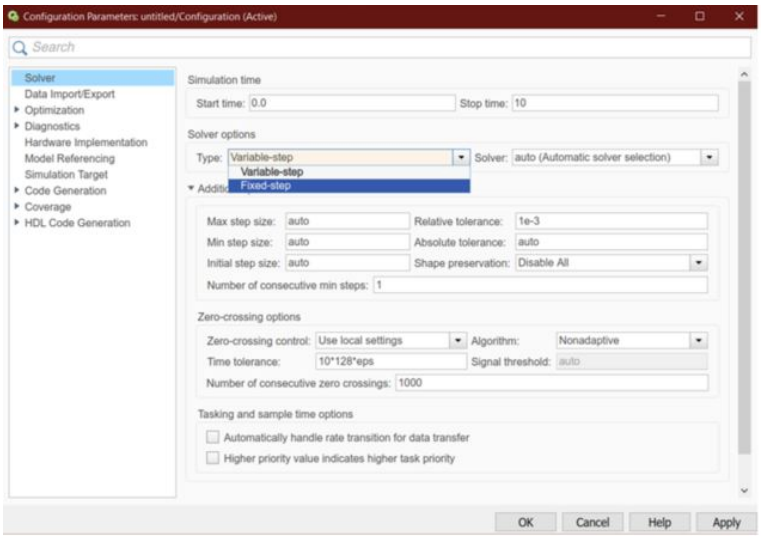


Figura 10.2.6: *Configurando para Fixed-step*

Em seguida, na aba **Fixed-step size (fundamental sample size)** pode-se diminuir o intervalo, como por exemplo, usando 1e-3 (1×10^{-3}), como mostra a Fig. 10.2.7.

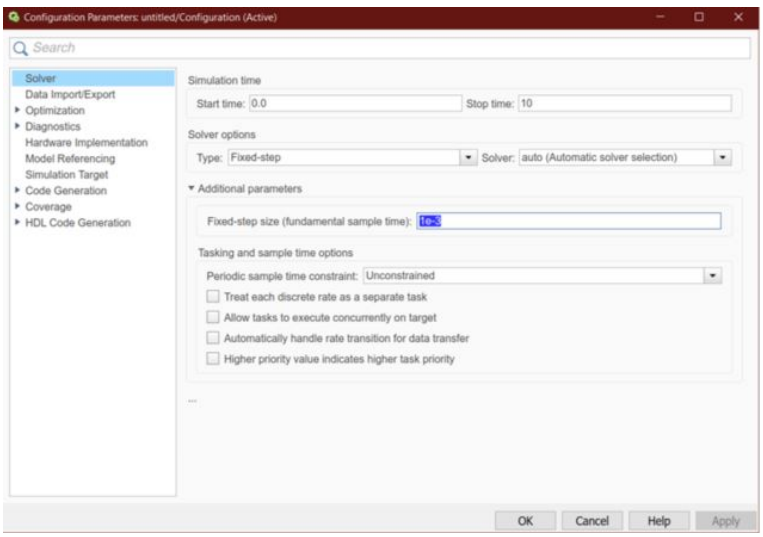


Figura 10.2.7: *Colocando o intervalo para 1e-3*

Por fim, clica-se em **Apply** e em **Ok**. Após executar novamente a simulação e verificar os resultados,

observa-se uma curva como a Fig. 10.2.8.

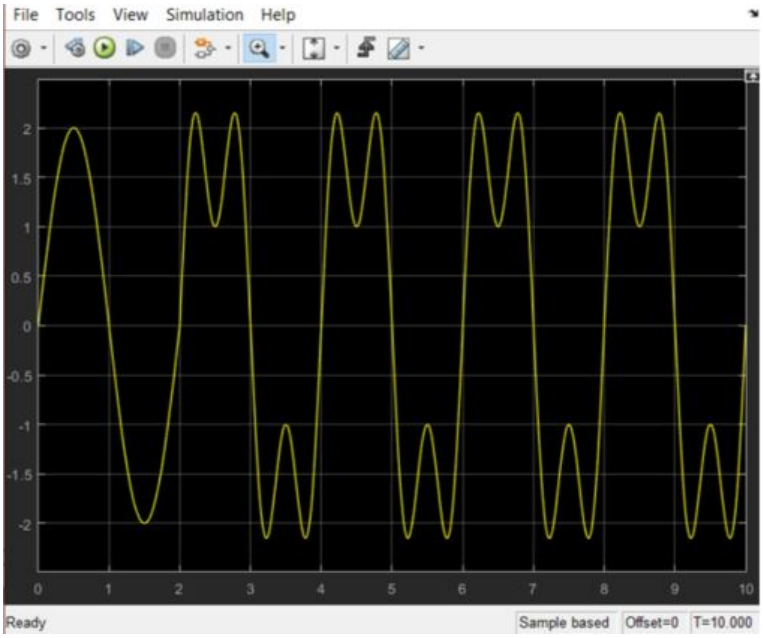


Figura 10.2.8: *Novo resultado após alterar o tamanho do passo do modelo*

Ao comparar a nova curva, Fig. 10.2.8, com a anterior, Fig. 10.2.5, observa-se que as imperfeições foram corrigidas, deixando o gráfico melhor definido. Mas ainda assim, se for dado zoom o suficiente, ainda será observado as imperfeições por aproximações lineares, como mostra a Fig. 10.2.9.

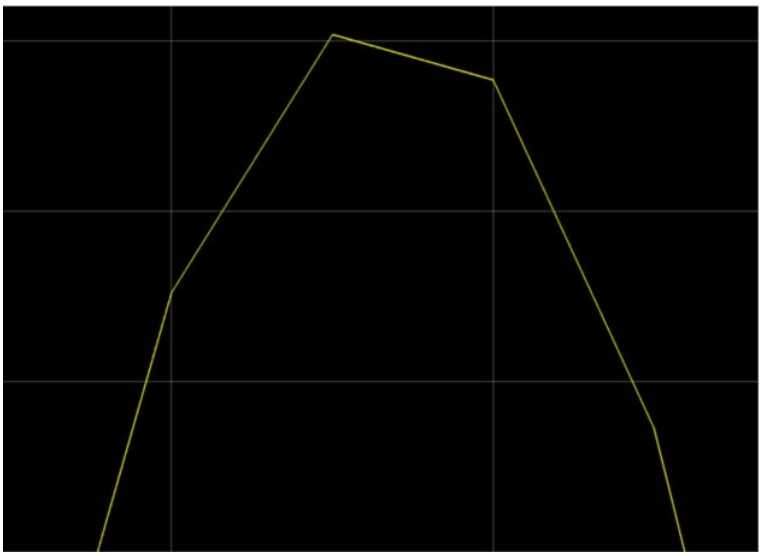


Figura 10.2.9: *Imperfeições na curva ao se dar muito zoom*

DICA: O bloco **Mux**, de Singal Routing, pode ser usado em conjunto com o **Scope** para visualizar múltiplas curvas simultaneamente, como mostra a Fig. 10.2.10.

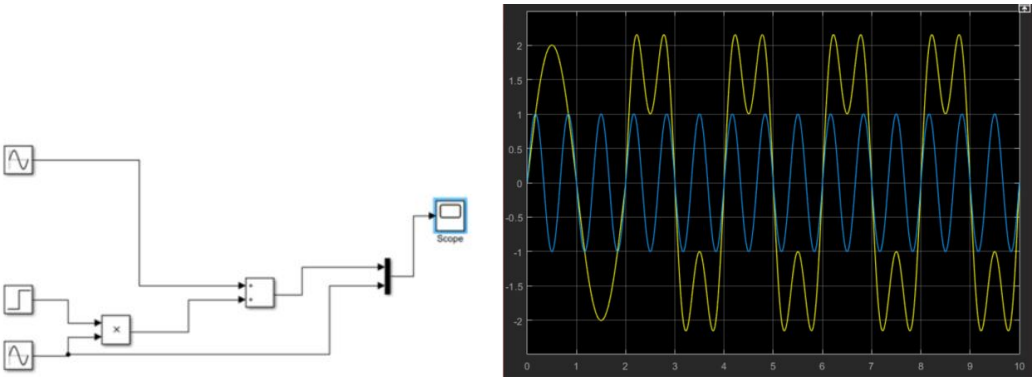


Figura 10.2.10: *Uso do Mux*

Exercícios

10.1 Crie um simples sistema para realizar uma multiplicação entre dois valores arbitrários. Exiba o resultado em um **Display**.

10.2 Crie um modelo que recebe um sinal senoidal, multiplica por dois, arredonda para o inteiro mais próximo, e divide por dois. Exiba o resultado em um **Scope**.

10.3 Crie um novo modelo. Utilizando os blocos **Sine Wave**, **Mux** e **Scope**, mostre em um mesmo osciloscópio as funções, de 0 até 2π:
a) $f(x) = sin(x)$
b) $f(x) = sin(x + \frac{\pi}{2})$
c) $f(x) = sin(\frac{x}{2})$
d) $f(x) = \frac{sin(2x)}{x}$

10.4 No modelo da questão anterior, utilize o bloco **To Workspace** como saída para poder exportar os dados coletados para a área de trabalho do Matlab. Use como parâmetros, exportação como arranjo.

10.5 Utilizando os dados da questão anterior, no Matlab, plote os quatro gráficos.

10.6 Em lançamentos oblíquos sem atrito, a posição em y em função do tempo é descrita como

$$y(t) = y_0 + V_{0y}t + \frac{(at^2)}{2}$$

Crie um sistema para simular a variação de y em função do tempo de um determinado lançamento oblíquo. Use $a = g = -9.8m/s^2$, $V_{0y} = 50$ e $y_0 = 0$.

10.7 Nos mesmos lançamos, a posição x em função do tempo é descrita como:

$$x(t) = x_0 + V_{0x}t$$

No mesmo sistema anterior, altere $V_{0y} = 40$, e adicione o modelo de x em função do tempo. Transfira os dados coletamos em forma de vetor para a área de trabalho do Matlab. Considere $V_{0x} = 30$ e $x_0 = 0$. Por fim, plote um gráfico de x(t) por y(t), ajuste o eixo y para não mostrar valores negativos do seu lançamento.

10.8 Modele a seguinte equação diferencial e observe o gráfico:

$$y'' + 2y' + 5y = 1$$

$$y'(0) = y(0) = 0$$

10.9 Modele a seguinte equação diferencial e observe o gráfico:

$$y'' + 4y' + 5y = 3$$

$$y'(0) = y(0) = 0$$

10.10 Produza pulsos quadrados utilizando os blocos **Sine Wave** e **Sign**.

IV

Índice

Resoluções **117**
Estruturas de Dados
Comandos e Funções
Operadores
Controle de Fluxo
Scripts
Gráficos
Polinômios
Simulink: Introdução

Bibliografia **155**

Resoluções

Estruturas de Dados

3.1 a)

```
>>5.17e-5 * 17
ans =
    8.7890e-04
```

b)

```
>>13.711 * 13.711
ans =
    187.9915
```

c)

```
>>ans / 5.17e-5
ans =
    3.6362e+06
```

d)

```
>>-1 / -1.6022e-19
ans =
    6.2414e+18
```

e)

```
>>50 / 342.3
ans =
    0.1461
```

3.2

```
>>0 / 0
ans =
    NaN

>>10 / 0
ans =
    Inf
```

NaN e Inf são constantes do MATLAB que não representam valores numéricos. NaN é o valor que representa "não é um número", e Inf representa infinito.

3.3 a)

```
>>NmrMatricula = 201900
```

b)

```
>>Notas = [0 0 0 0 0]
```

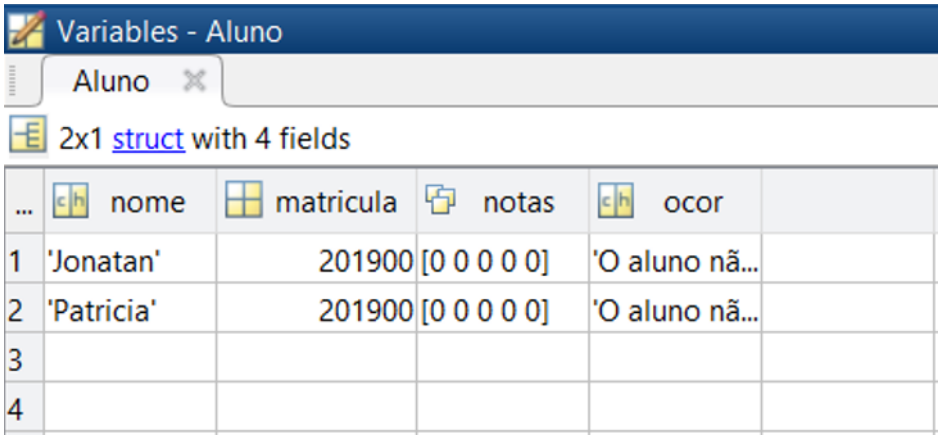
c)

```
>>Ocorrencias = 'O aluno não possui ocorrencias'
```

d)

```
>>Aluno(1,1).nome = 'Jonatan'; Aluno(1,1).matricula =
   NmrMatricula; Aluno(1,1).notas = Notas; Aluno(1,1).ocor =
   Ocorrencias

>>Aluno(2,1).nome = 'Patricia'; Aluno(2,1).matricula =
   NmrMatricula; Aluno(2,1).notas = Notas; Aluno(2,1).ocor =
   Ocorrencias
```

3.4 Ver Fig. 10.2.11



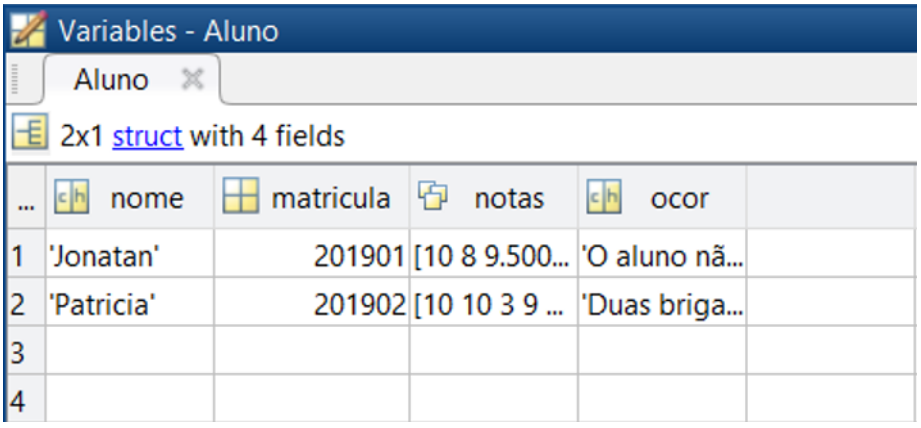
...	nome	matricula	notas	ocor
1	'Jonatan'	201900	[0 0 0 0 0]	'O aluno nã...
2	'Patricia'	201900	[0 0 0 0 0]	'O aluno nã...
3				
4				

Figura 10.2.11: *Variáveis na janela Variable Editor.*

3.5 a)

```
>>Aluno(1,1).matricula = NmrMatricula + 1

>>Aluno(1,1).notas = [10 8 9.5 7 10]

>>Aluno(2,1).matricula = NmrMatricula + 2

>>Aluno(2,1).notas = [10 10 3 9 7.8]

>>Aluno(2,1).ocor = 'Duas brigas em sala de aula'
```

3.6 Ver Fig. 10.2.12



...	nome	matricula	notas	ocor
1	'Jonatan'	201901	[10 8 9.500...	'O aluno nã...
2	'Patricia'	201902	[10 10 3 9 ...	'Duas briga...
3				
4				

Figura 10.2.12: *Variáveis na janela Variable Editor.*

3.7 a)

```
>>A = [2+2i, 0, 7i, 13+2i]
```

b)

```
>> B = [A(1)', A(2)', A(3)', A(4)']
B =
    2.0000 - 2.0000i   0.0000 + 0.0000i   0.0000 - 7.0000i
       13.0000 - 2.0000i
```

3.8

```
>>syms x

>>y = 2*x^2 - 30*x + 100

>>subs(y, x, 10)
ans =
    0
```

3.9

```
>>subs(y, x, 5)
ans =
    0
```

3.10

```
>>subs(y, x, 0)
ans =
    100
```

Comandos e Funções

4.1

```
>>help cholesky
cholesky not found.
Use the Help browser search field to search the documentation,
   or
type "help help" for help command options, such as help for
   methods.

>> lookfor cholesky
chol            - Cholesky factorization.
cholupdate      - Rank 1 update to Cholesky factorization.
ichol           - Sparse Incomplete Cholesky factorization
finchol         - Cholesky factor for positive semi-definite
   correlation matrices.
covlamb         - Covariance matrix of Cholesky factor of
cholcov         - Cholesky-like decomposition for covariance
   matrix.
```

```
>>help chol
 chol   Cholesky factorization.
chol(A) uses only the diagonal and upper triangle of A.
The lower triangle is assumed to be the (complex conjugate)
   transpose of the upper triangle. If A is positive definite,
   then R = chol(A) produces an upper triangular R so that
   R'*R = A. If A is not positive definite, an error message
   is printed
L = chol(A,'lower') uses only the diagonal and the lower
   triangle of A to produce a lower triangular L so that L*L'
   = A.  If A is not positive definite, an error message is
   printed. When A is sparse, this syntax of chol is typically
   faster.
[R,p] = chol(A), with two output arguments, never produces an
   error message. If A is positive definite, then p is 0 and R
   is the same as above. But if A is not positive definite,
   then p is a positive integer. When A is full, R is an upper
   triangular matrix of order q = p-1 so that R'*R =
   A(1:q,1:q). When A is sparse, R is an upper triangular
   matrix of size q-by-n so that the L-shaped region of the
   first q rows and first q columns of R'*R agree with those
   of A.
[L,p] = chol(A,'lower'), functions as described above, only a
   lower triangular matrix L is produced.  That is, when A is
   full, L is a lower triangular matrix of order q = p-1 so
   that L*L' = A(1:q,1:q). When A is sparse, L is a lower
   triangular matrix of size n-by-q so that the L-shaped
   region of the first q rows and first q columns of L*L'
   agree with those of A.
[R,p,S] = chol(A), when A is sparse, returns a permutation
   matrix S which is a preordering of A obtained by AMD. When
   p = 0, R is an upper triangular matrix such that R'*R =
   S'*A*S.  When p is not zero, R is an upper triangular
   matrix of size q-by-n so that the L-shaped region of the
   first q rows and first q columns of R'*R agree with those
   of S'*A*S.  The factor of S'*A*S tends to be sparser than
   the factor of A.
[R,p,s] = chol(A,'vector') returns the permutation information
   as a vector s such that A(s,s) = R'*R, when p = 0.  The
   flag 'matrix' may be used in place of 'vector' to obtain
   the default behavior.
[L,p,s] = chol(A,'lower','vector') uses only the diagonal and
   the lower triangle of A and returns a lower triangular
   matrix L and a permutation vector s such that A(s,s) =
   L*L', when p = 0. As above, 'matrix' may be used in place
   of 'vector' to obtain a permutation matrix.
    For sparse A, CHOLMOD is used to compute the Cholesky
       factor.
    See also cholupdate, ichol, ldl, lu.
Reference page for chol
Other functions named chol

>>A=[9 6 -3 3; 6 20 2 22; -3 2 6 2; 3 22 2 28];

>>chol(A)
```

```
ans =
     3     2    -1     1
     0     4     1     5
     0     0     2    -1
     0     0     0     1
```

4.2 a)

```
>>sqrt(187.9915)
ans =
    13.7110
```

b)

```
>>round(sqrt(187.9915))
ans =
    14

>>fix(sqrt(187.9915))
ans =
    13
```

c)

```
>> exp(3)
ans =
    20.0855
```

d)

```
>> round(exp(3))
ans =
    20

>> ceil(exp(3))
ans =
    21
```

e)

```
>> log(3)
ans =
    1.0986
```

f)

```
>> round(log(3))
ans =
```

```
     1

>> ceil(log(3))
ans =
     2
```

4.3 a)

```
>>A = [1 2 3]
```

b)

```
>>B = [A; A; A]
```

c)

```
>>P = [2 2]
```

d)

```
>>Q = [P; P]
```

e)

```
>> R=cat(3, Q, Q)
```

4.4 a)

```
>>size(R)
ans =
     2     2     2
```

b)

```
>>trace(B)
ans =
     6
```

c)

```
>>det(B)
ans =
    0
```

d)

```
>>trace(Q)
ans =
     4
```

e)

```
>>det(Q)
ans =
     0
```

4.5 a)

```
>>display('Hello World')
Hello World
```

b)

```
>> x=input('insira o valor de x\n')
insira o valor de x
4
x =
    4
```

c)

```
>> fprintf('Olá, programador, x = %i\n', x)
Olá, programador, x = 4
```

4.6 a)

```
>>a1 = linspace(0, 2*pi, 5)
```

b)

```
>>b1 = sin(a1)
```

c)

```
>>a2 = linspace(0, 2*pi, 10)
```

d)

```
>>b2 = sin(a2)
```

e)

```
>>a3 = linspace(0, 2*pi, 100)
```

f)

```
>>b3 = sin(a3)
```

4.7

```
>>plot(a1, b1, a2, b2, a3, b3)
```

O gráfico plotado pode ser visto na Fig. 10.2.13

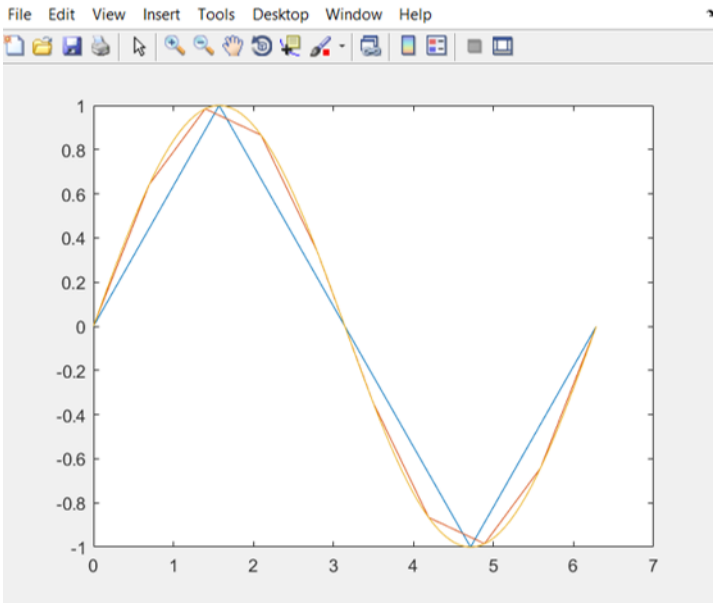


Figura 10.2.13: *Gráficos de ai x bi.*

4.8 O gráfico 1 é o de azul, o 2 o de amarelo e o 3 o de laranja. Podemos tirar essa conclusão pois os três espaços ai vão de 0 até 2π, o que difere um do outro é a quantidade de termos. Quanto menos termos, menor a precisão do gráfico seno, logo, o menos preciso é o 1, e o mais preciso o 3.

4.9

```
>> abs(13 + 13i)
ans =
   18.3848

>> angle(13 + 13i)
ans =
    0.7854
```

```
>> rad2deg(ans)
ans =
    45
```

4.10

```
>> abs(20i)
ans =
    20

>> angle(20i)
ans =
    1.5708

>> rad2deg(ans)
ans =
    90
```

Operadores

5.1 a)

```
>>13.711^2
ans =
    187.9915
```

b)

```
>> 13.711^3
ans =
    2.5776e+03
```

c)

```
>>[4, 1, 2]'
ans =
     4
     1
     2
```

5.2

```
>>10/5, 10\5
ans =
    2
ans =
    0.5000
```

A inclinação da barra determina quem ficará em cima e em baixo na fração, sendo / a forma mais usual.

5.3 Não. Pois, 10 > x retornará 1 (verdadeiro) ou 0 (falso) de acordo com o valor de x, mas 1 > 2 ou 0 > 2 sempre retornará 0 (falso).

5.4 Irá retornar 1 para valores de x menores do que 11, pois (2 < 10 == 1) + 10 é igual a 10.

5.5 ((a > 10) == 0) + b > c == d
d = 0, a = 11
(1 == 0) + b > c == 1
b > c == 1
b = 1; c = 0
1 == 1
1

5.6 ((a > 10) == 0) + b > c == d
d = 0, a = 11
(1 == 0) + b > c == 1
b > c == 1
b = 0; c = 1
0 == 1
0

5.7 a)

```
>>A = [1 1 1; 0 0 0; 1 1 1]
```

b)

```
>>B = [1 6 1; 2 9 0; 1 1 1]
```

c)

```
>>C = [2 4 5]
```

d)

```
>>D = 7
```

5.8 a)

```
>>A * B
ans =
     4    16     2
     0     0     0
     4    16     2
```

b)

```
>>A  .* B
ans =
      1      6      1
      0      0      0
      1      1      1
```

c)

```
>>D  * A
ans =
      7      7      7
      0      0      0
      7      7      7
```

d)

```
 >>C  .^ D
ans =
          128        16384        78125
```

e)

```
>> D  .^ C
ans =
           49         2401        16807
```

5.9

```
>>A  * B
ans =
      4     16      2
      0      0      0
      4     16      2

]>>A  .* B
ans =
      1      6      1
      0      0      0
      1      1      1
```

O operador "." indica que a operação será ponto a ponto, ou seja, Aij x Bij, e não uma multiplicação matricial comum.

5.10

```
A = [ rand>0.5 rand>0.5 rand>0.5 rand>0.5 rand>0.5 rand>0.5 ]
A =
  1x6 logical array
   1    0    1    0    0    1
```

Controle de Fluxo

6.1

```
>>a = 3; b = 2;

>>operador=input('Operação: 1(soma)\n')
Operação: 1(soma)
1

>>if operador == 1
a + b
end
ans =
     5
```

6.2

```
>>if operador == 2
a - b
end
```

6.3

```
>>if operador == 3
a * b
end
```

6.4

```
>>if operador == 4
a / b
end
```

6.5

```
>>operador=input('Operação: 1(soma)\n');

>>switch operador
case 1
a + b
case 2
a - b
case 3
a * b
case 4
a / b
end
```

6.6

```
>>Nota=input('Digite a nota do aluno(0 a 5)')
if Nota == 0
display('Procure o professor para ajuda')
end
if Nota == 1
display('Precisa se dedicar mais')
end
if Nota == 2
display('Ainda um pouco mais')
end
if Nota == 3
display('Bom, 60 é 100')
end
if Nota == 4
display('Parabéns')
end
if Nota == 5
display('Sensacional!!')
end
```

6.7

```
>>contador = 10

>>for i = 1:10
G(contador) = i^2
contador = contador - 1
end

>>G
G =
   100     81     64     49     36     25     16      9      4      1
```

6.8

```
>>contador = 1; valor = 10

>>while contador < 10
G(contador) = valor^2;
contador = contador + 1; valor = valor - 1;
end

>>G
G =
   100     81     64     49     36     25     16      9      4      1
```

6.9

```
>>contador = 1

>>for i = [3, 3, 1, 2, 3]
H(contador) = i^3;
contador = contador + 1;
end

>>H
H =
    27    27     1     8    27
```

6.10

```
>>contador = 1; valor = [3 3 1 2 3]
>>while contador < 5
H(contador) = valor(contador)^3;
contador = contador + 1;
end

>>H
H =
    27    27     1     8    27
```

Scripts

7.1 Clique em *New Script* no canto superior esquerdo para iniciar um novo script. Código:

```
clear, clc %Limpar a Command Window e o Workspace
Nota=input('Digite a nota do aluno(0 a 5)')
if Nota == 0
display('Procure o professor para ajuda')
end
if Nota == 1
display('Precisa se dedicar mais')
end
if Nota == 2
display('Ainda um pouco mais')
end
if Nota == 3
display('Bom, 60 é 100')
end
if Nota == 4
display('Parabéns')
end
if Nota == 5
display('Sensacional!!')
end
```

Clique em *Save*, nomeie e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

7.2 Clique em *New Script* no canto superior esquerdo para iniciar um novo script. Código:

```
clear, clc %Limpar a Command Window e o Workspace
contador = 10;
for i = 1:10
G(contador) = i^2;
contado r =contador - 1;
end
G

G =
   100    81    64    49    36    25    16     9     4     1
```

Clique em *Save*, nomeie e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

7.3 Clique em *New Script* no canto superior esquerdo para iniciar um novo script. Código:

```
clear, clc %Limpar a Command Window e o Workspace
contador = 1;
for i = [3, 3, 1, 2, 3]
H(contador) = i^3;
contador = contador + 1;
end
H

H =
    27    27     1     8    27
```

Clique em *Save*, nomeie e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

7.4 Clique em *New Script* no canto superior esquerdo para iniciar um novo script. Código:

```
function c = ex1log(x,y)
c = log(x) / log(y);
end
```

Clique em *Save*, nomeie como ex1log.m e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

7.5 A função precisa ser modificada apenas em um detalhe, a divisão necessita ser ponto a ponto:

```
function c = ex1log(x,y)
c = log(x) ./ log(y);
end
```

7.6 Clique em *New Script* no canto superior esquerdo para iniciar um novo script. Código:

```
function ordenado = ex1ord(x)
for i = 1:size(x,2)
    for j = (i+1):size(x,2)
        if x(i) > x(j)
```

```
            aux = x(i);
            x(i) = x(j);
            x(j) = aux;
        end
    end
end
ordenado = x;
end
```

Clique em *Save*, nomeie como ex1ord.m e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

7.7 Clique em *New Script* no canto superior esquerdo para iniciar um novo script. Código:

```
function confere = ex1cpf(x)
aux=1e10;
for i = 1:9 % Extrair os 9 dígitos em forma de arranjo
    dig(i)= floor( x/aux );
    x = x - dig(i)*aux;
    aux = aux/10;
end % Aqui, x vale exatamente os dígitos verificadores
% Início do cálculo dos dígitos verificadores
verifica = 0; s
resto = 0;
aux=10;
for i = 1:9
    resto = resto + aux*dig(i);
    aux =  aux - 1;
end
resto = rem(resto, 11);
if resto == 0 | resto == 1
    verifica = 0;
else
    verifica = (11 - resto);
end
%Já calculado o primeiro verificador, inciando o cálculo do
   segundo
resto = 0;
aux=11;
for i = 1:9
    resto = resto + aux*dig(i);
    aux =  aux - 1;
end
resto = resto + aux*verifica;
resto = rem(resto, 11);
verifica = verifica*10;
if resto == 0 | resto == 1
else
    verifica = verifica + (11 - resto);
end
if verifica == x % verificando e atribuindo 1 ou 0
    confere = 1;
else
    confere = 0;
end
```

```
end
```

Clique em *Save*, nomeie como ex1cpf.m e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

7.8 Podemos fazer o script utilizando a função criada na questão anterior, ou até mesmo no mesmo script. Código:

```
x = input ('Insira o cpf');
if ex1cpf(x)
display('CPF correto')
else
display('CPF incorreto')
end
```

Clique em *Save*, nomeie e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

7.9 Clique em *New Script* no canto superior esquerdo para iniciar um novo script. Código:

```
N = input('Quantos alunos são?\n');
media = 0;
for i = 1:N
  media = media + input('nota:\n');
end
media = media/N;
display(['A nota média é ', num2str(media)])
```

Clique em *Save*, nomeie e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

7.10 Clique em *New Script* no canto superior esquerdo para iniciar um novo script. Código:

```
x = input('Insira o vetor x\n');
y = input('Insira o vetor y\n');
opera = input('O que deseja?(1 ou 2)\n');
if opera == 1
x * y'
end
if opera == 2
x' * y
end
```

Clique em *Save*, nomeie e tente executar o script. Ao tentar executá-lo pela primeira vez, clique em *Change Folder* ou *Add to Path*.

Gráficos

8.1 a)

```
>>figure; sphere;
```

Ver Fig. 10.2.14

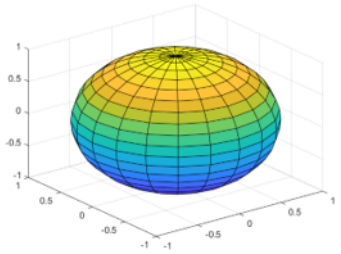


Figura 10.2.14: *Esfera.*

b)

```
>>figure; cylinder;
```

Ver Fig. 10.2.15

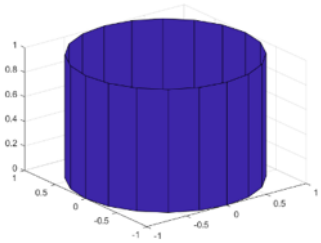


Figura 10.2.15: *Cilindro.*

c)

```
>>figure; ellipsoid(1,1,1,5,10,1);
```

Ver Fig. 10.2.16

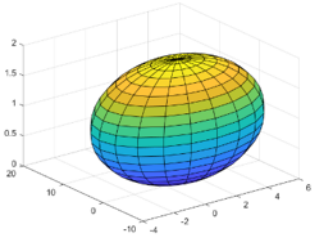


Figura 10.2.16: *Elipsóide.*

8.2

```
>>fplot('sin', [0 2*pi],'b'); hold on; fplot('cos', [0 2*pi],
    'r')
```

Ver Fig. 10.2.17

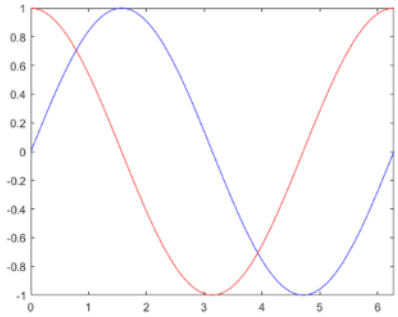


Figura 10.2.17: *Seno e cosseno plotados.*

8.3 a)

```
>>title('Seno e Cosseno')
```

b)

```
>>xlabel('0 a 2\pi')
```

c)

```
>>ylabel('0 a 1')
```

Ver Fig. 10.2.18

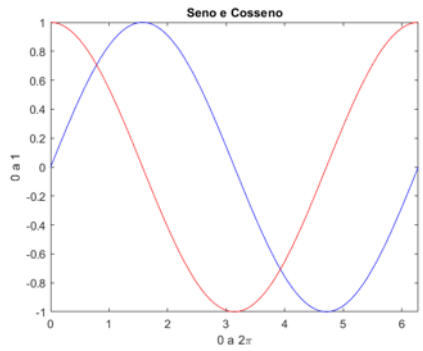


Figura 10.2.18: *Título e legendas no gráfico.*

8.4 a)

```
>>k = 1:20;
```

b)

```
>>k_str = num2str(k);
```

c)

```
>>f = ['sinc([', k_str, ']*x)'];
```

d)

```
>>fplot(f, [-1 1])
```

Ver Fig. 10.2.19

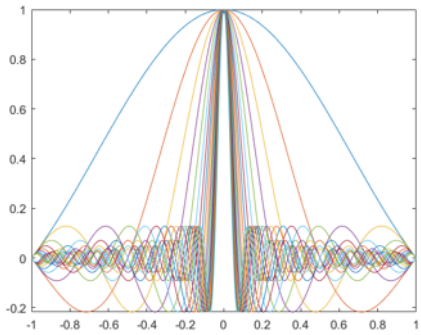


Figura 10.2.19: *Gráfico a ser obtido.*

8.5 a)

```
>>syms x; y1 = x^2 -2*x + 5;

>>figure; fplot(y1, [-10 10], 'b');
```

Ver Fig. 10.2.20

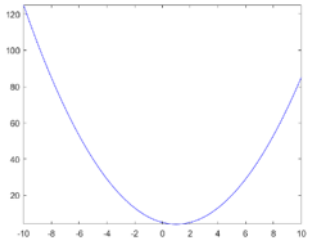


Figura 10.2.20: *Gráfico azul.*

b)

```
>>y2 = 2^x/x;

>>figure; fplot(y2, [-5 5], 'r');
```

Ver Fig. 10.2.21

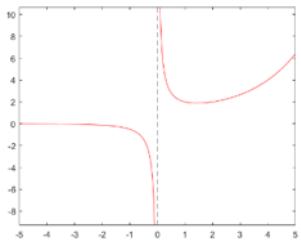


Figura 10.2.21: *Gráfico vermelho.*

c)

```
>>y3 = sinc(x);

>>figure; fplot(y3, [-50 50], 'g');
```

Ver Fig. 10.2.22

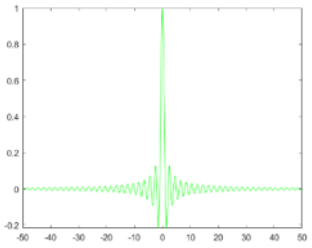


Figura 10.2.22: *Gráfico verde.*

8.6

```
>>x = linspace(-10, 10, 100); [x, y] = meshgrid(x);

>>z = 5*sqrt(x.^2/4 + y.^2/9);

>>mesh(x, y, z); hold on; mesh(x, y, -z)

>>contour(x, y, z)
```

Ver Fig. 10.2.23

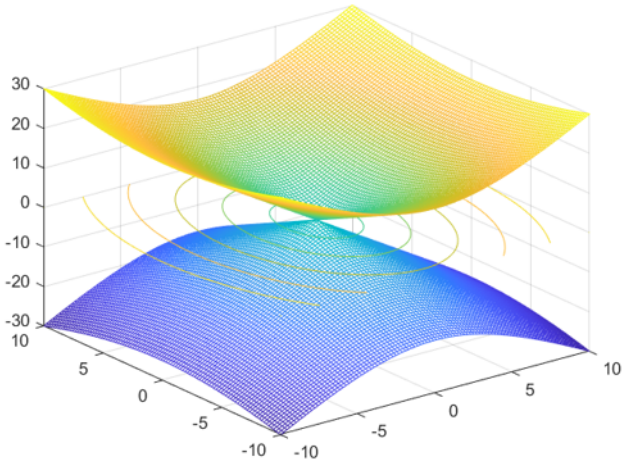


Figura 10.2.23: *Gráfico do cone com contorno 2D.*

8.7

```
>>x = linspace(2, 4); y = linspace(1, 3);

>>[X, Y] = meshgrid(x, y);

>>Z = (X-3).^2 - (Y-2).^2;

>>surf(X, Y, Z)
```

Ver Fig. 10.2.24

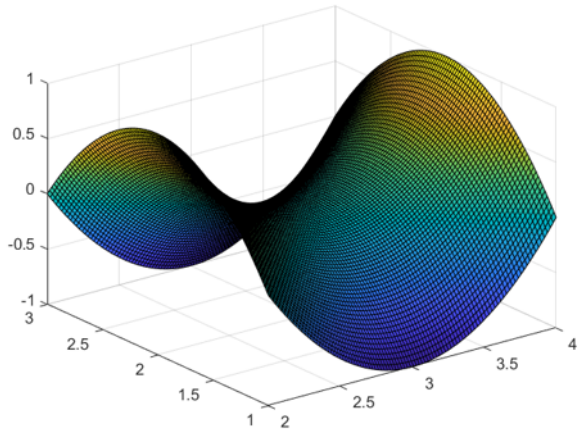


Figura 10.2.24: *Gráfico do paraboloide hiperbólico.*

8.8 a)

```
>>syms x;
```

```
>>s2 = 1 + x + x^2/factorial(2);

>>fplot(s2, [-2, 5], '--');

>>hold on;
```

b)

```
>>s3 = s2 + x^3/factorial(3);

>>fplot(s3, [-2, 5], '--');
```

c)

```
>>s4 = s3 + x^4/factorial(4);

>>fplot(s4, [-2, 5], '--');
```

d)

```
>>s5 = s4 + x^5/factorial(5);

>>fplot(s5, [-2, 5], '--');
```

```
>>fplot(exp(x), [-2, 5]);

>>legend('grau 2', 'grau 3', 'grau 4', 'grau 5', 'exp(x)');
```

Ver Fig. 10.2.25

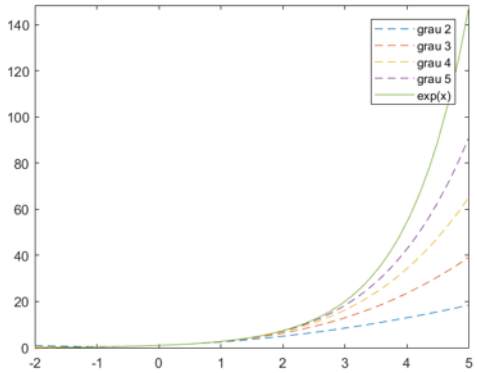


Figura 10.2.25: *Comparação entre a função exponencial e sua representação através do somatório, observe a convergência.*

```
>>syms x;

>>s2 = 1 - x^2/factorial(2) + x^4/factorial(4);

>>s3 = s2 - x^6/ factorial(6);

>>s4 = s3 + x^8/factorial(8);

>>s5 = s4 - x^10/factorial(10);

>>fplot(s2, [0, pi], '--'); hold on;

>>fplot(s3, [0, pi], '--'); fplot(s4, [0, pi], '--');
   fplot(s5, [0, pi], '--');

>>fplot(cos(x), [0, pi]);

>>legend('grau 4', 'grau 6', 'grau 8', 'grau 10', 'cos(x)');
```

Ver Fig. 10.2.26

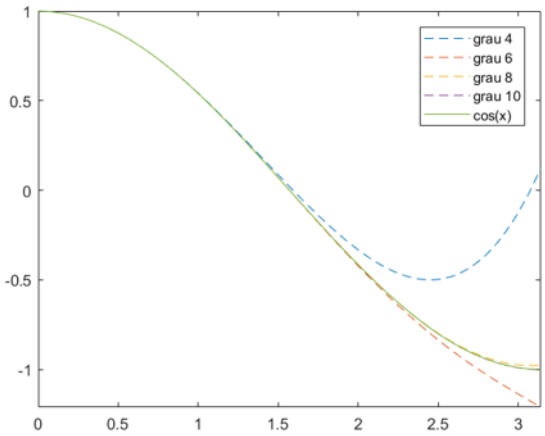


Figura 10.2.26: *Comparação entre a função cosseno e sua representação através do somatório, observe a convergência.*

8.10 Sabendo que a função randn retorna um arranjo com valores em distribuição normal padrão, iremos somá-la à função y para adquirir os ruídos.

```
>>x=linspace(-2, 10);

>>y=2*x + 1;

>>y2 = y+rand(1, 100); %ruídos

>>plot(x, y, x, y2);
```

Ver Fig. 10.2.27

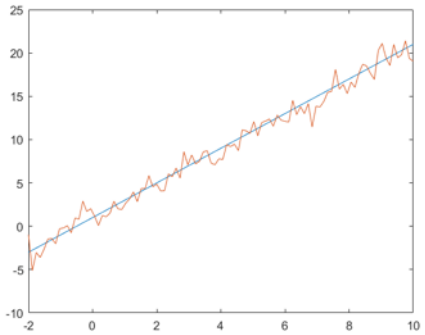


Figura 10.2.27: *Função original e função com adição de ruídos.*

Polinômios

9.1

```
>>P = [1 -15 85 -225 274 120]; P = [1 -15 85 -225 274 -120];
   P1 = [-120 274 -225 85 -15 1]; P2 = [-1 -15 -85 -225 -274
   -120]; P3 = [120 274 225 85 15 1];

>>roots(P)
ans =
    5.0000
    4.0000
    3.0000
    2.0000
    1.0000

>>roots(P1)
ans =
    1.0000
    0.5000
    0.3333
    0.2500
    0.2000

>>roots(P2)
ans =
   -5.0000
   -4.0000
   -3.0000
   -2.0000
   -1.0000

>>roots(P3)
ans =
   -1.0000
   -0.5000
   -0.3333
   -0.2500
   -0.2000
```

Podemos observar que as raízes de P1 são o inverso das raízes de P, enquanto que as de P2 são o oposto, e as de P3 o inverso e oposto.

9.2 a)

```
>>poly([1 2; 2 1])
ans =
     1     -2     -3

>>roots(ans)
ans =
     3.0000
    -1.0000
```

b)

```
>>poly([1 2 3; 4 5 6; 7 8 9])
ans =
     1.0000   -15.0000   -18.0000    -0.0000

>>roots(ans)
ans =
     16.1168
    -1.1168
    -0.0000
```

c)

```
>>poly([0 0; 1 1])
ans =
      1     -1      0

>>roots(ans)
ans =
      0
      1
```

d)

```
>>poly([98 88 2.04; 0.13  -12 9.08; 11.7  11.7 2.22])
ans =
     1.0e+03 *
      0.0010    -0.0882    -1.1266     3.4090

>>roots(ans)
ans =
     99.2277
    -13.5442
     2.5365
```

9.3 a) Como x e y formam um conjunto de seis pontos, o polinômio interpolador via ajuste polinomial será de grau 5.

```
>>p5 = polyfit(x, y, 5)
p5 =
    -1.5000   -16.9583    14.3000    -0.2854    -0.1803    15.0317
```

b)

```
>>p1 = polyfit(x, y, 1)
p1 =
    1.0830    14.9382
```

c)

```
>>p2 = polyfit(x, y, 2)
p2 =
    2.5679    -0.2009    15.0238
```

9.4

```
>>H = linspace(0, 0.5);

>>y5 = polyval(p5, H); y1 = polyval(p1, H); y2 = polyval(p2,
   H);

>>plot(H, y5, H, y1, H, y2, x, y, '+')

>>legend('grau 5', 'grau 1', 'grau 2', 'pontos')
```

Ver Fig. 10.2.28

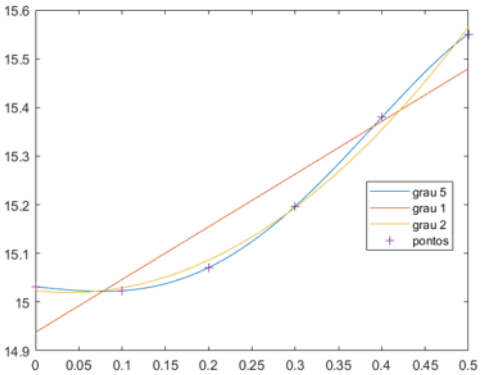


Figura 10.2.28: .

9.5 a)

```
>>a = [1 5 -5 0 10];
```

b)

```
>>b = [1 0 0 0 0 0];
```

c)

```
>>c = conv([1 -1],[1 -3]);
```

d)

```
>>deconv(b,a)
ans =
    1    -5
```

e)

```
>>conv(a, c)
ans =
    1     1   -22    35    -5   -40    30
```

f)

```
>>10*b
ans =
    10     0     0     0     0     0
```

g)

```
>>c1 = [0 0 c]; %ajustando a dimensão de c para a subtração

>>roots(5*a-c1)
ans =
   -5.8635 + 0.0000i
    0.9180 + 0.8977i
    0.9180 - 0.8977i
   -0.9725 + 0.0000i
```

9.6 Primeiro precisamos escrever os polinômios P e Q, para Q, vamos usar a função conv:

```
>>P = [1 1];

>>Q = conv([1 0],[1 2]); Q = conv(Q, [1 4]);
```

```
>>[r p k] = residue(P, Q)
r =
   -0.3750
    0.2500
    0.1250
p =
    -4
    -2
     0
k =
     []
```

Assim, obtemos as frações parciais:

$$f(x) = \frac{-3}{8(x+4)} + \frac{1}{4(x+2)} + \frac{1}{8x}$$

9.7 Agora basta usar a função utilizando os parâmetros r, p e k. No lugar de k, podemos também usar 0, uma vez que não há termo direto:

```
>>[A B] = residue(r, p, k)
A =
    0     1     1

B =
    1    6     8     0
```

9.8 Basta usarmos os vetores x e y para um ajuste polinomial de grau 1. Deve ser usado grau 1, pois a força elástica é linear em x. Feito isso, teremos a aproximação linear do gráfico de deformação (m) x massa (kg). O segundo termo é referente à deformação da mola quando a massa sob ela é 0, e portanto, apenas o primeiro termo nos interessa. Por fim, basta fazermos gravidade/(primeiro termo).
(I) $P = aX + b$: expressão polinomial de grau 1
(II) $kx = mg$: x = deformação, m = massa
$x = \frac{g}{k}m$
$\frac{g}{k} = a$
$k = \frac{g}{a}$

```
>>x = 0:0.5:3; y =  [0    0.0550    0.1010    0.1630    0.2000
      0.2520    0.2920];

>>a = polyfit(x, y, 1);

>>k = 9.8/a(1)
k =
    100.2191
```

9.9

```
>>y2 = polyval(a, x);

>>plot(x, y, 'r+', x, y2, 'b')
```

Ver Fig. 10.2.29

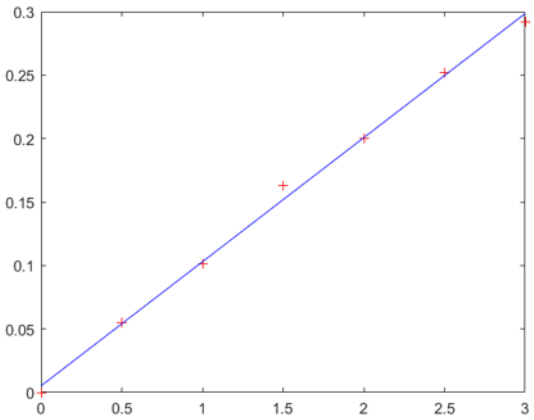


Figura 10.2.29: .

9.10

```
>>F = [1 -3 0 4 0];

>>roots(F)
ans =
          0
     2.0000
     2.0000
    -1.0000
```

Simulink: Introdução

10.1 Basta adicionarmos duas constantes, um multiplicador e um display: Ver Fig. 10.2.30

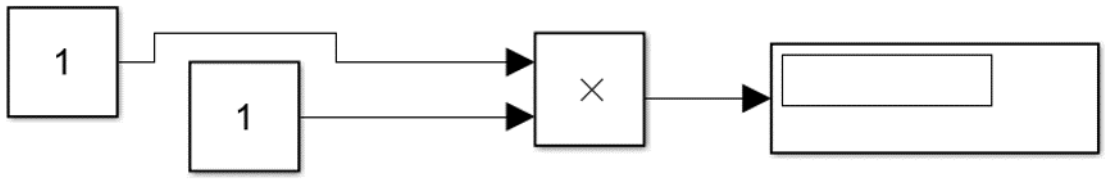


Figura 10.2.30: .

10.2 Para o sinal senoidal iremos usar "*Sine Wave*". Para a multiplicação e divisão, iremos usar os blocos "*Gain*". Para arredondar iremos usar o bloco que realiza a função com o mesmo nome no MATLAB: "*Round*". Montaremos os blocos assim: Ver Fig. 10.2.32 e **??**

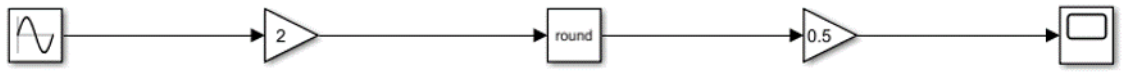


Figura 10.2.31: .

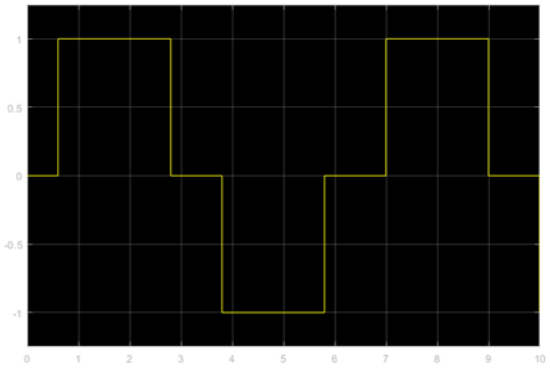


Figura 10.2.32: .

10.3 Para esse sistema, utilizaremos quarto blocos "*Sine Wave*", um bloco "*Mux*" para juntarmos os valores e um bloco "*Scope*", para coletarmos os dados e vermos as curvas devidamente. Ver Fig. 10.2.33

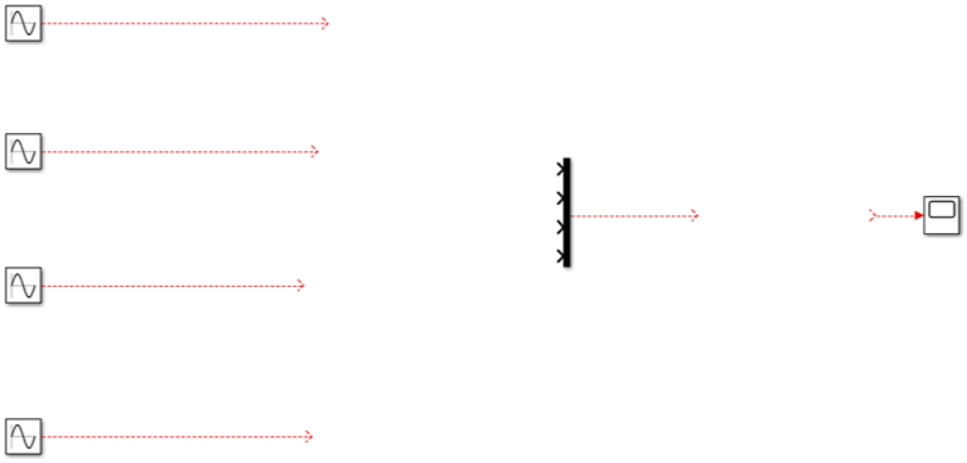


Figura 10.2.33: .

Devemos modificar os parâmetros das fontes senoidais para podermos obtermos os resultados desejados. O primeiro bloco será a função padrão e a referência para as outras três. Vamos utilizar, arbitrariamente, o tempo de execução de 10 segundos. O bloco senoidal emite valores da seguinte forma:

$$O(t) = Amp * Sin(Freq * t + Phase) + Bias$$

Com isso, para obtermos, do primeiro bloco, a função seno padrão de 0 a 2π, devemos ajustar a frequência para "0.2π", mantendo os outros parâmetros. Para o segundo, devemos colocar

o mesmo valor de frequência que o primeiro, e mudar a "*Phase*" para "$\pi/2$". Para o terceiro, a frequência deve ser metade da do primeiro, logo "0.1π". Para o último, a frequência deve ser o dobro da do primeiro (0.4π), e a amplitude a metade (0.5). Após ligarmos nosso sistema e coletarmos os dados: Ver Fig. 10.2.34 e 10.2.35

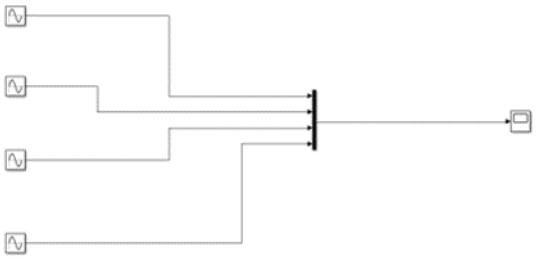


Figura 10.2.34: .

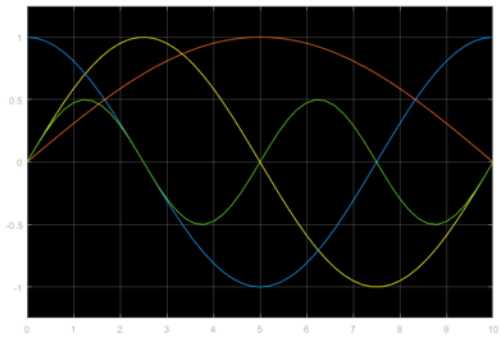


Figura 10.2.35: .

10.4 Vamos adicionar como saída o bloco "*To Workspace*". Ver Fig. 10.2.36

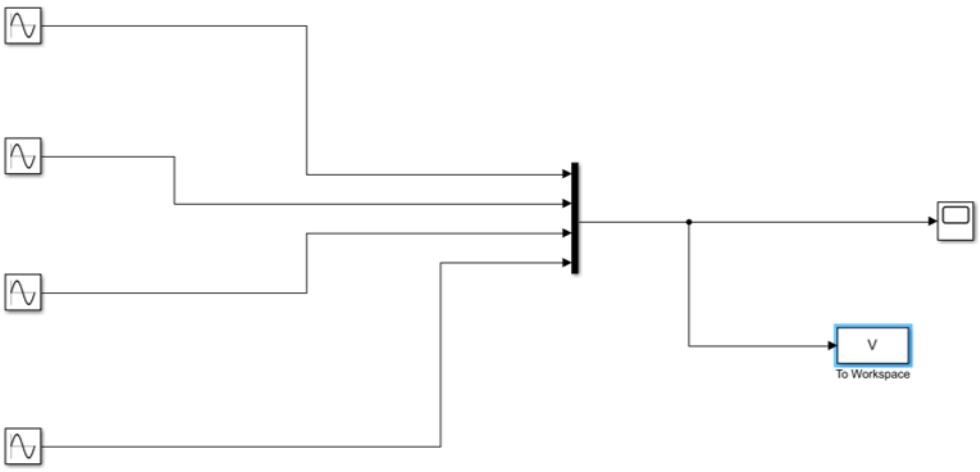


Figura 10.2.36: .

Mudamos o nome da variável para V e o tipo de saída para arranjo: Ver Fig. 10.2.37

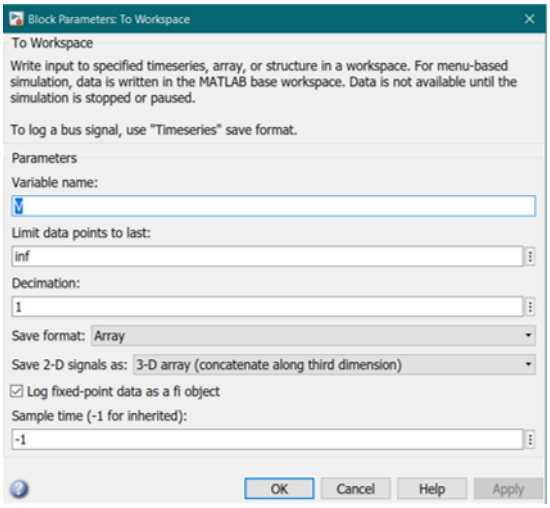


Figura 10.2.37: .

10.5 Observe que também foi exportado a variável tout, referente à sequência dos tempos de saída. Agora, para plotar, basta fazermos:

```
>>plot(tout, V(:,1), tout, V(:,2), tout, V(:,3), tout, V(:,4))
```

Ver Fig. 10.2.38

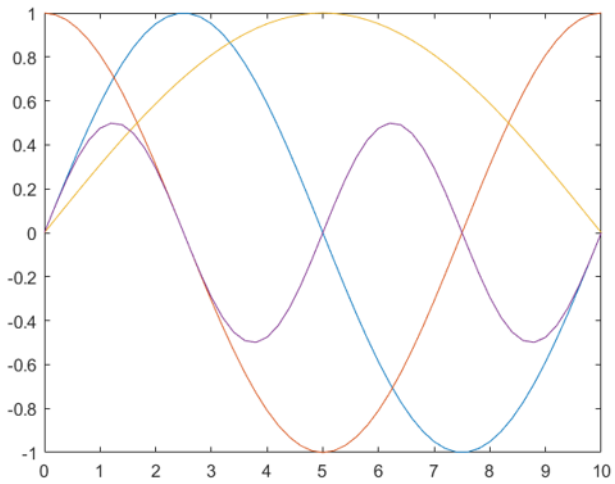


Figura 10.2.38: .

10.6 Utilizaremos dois blocos "*Constant*", três blocos "*Integrator*", um bloco "*Add*" e um bloco "*Scope*". Ver Fig. 10.2.39

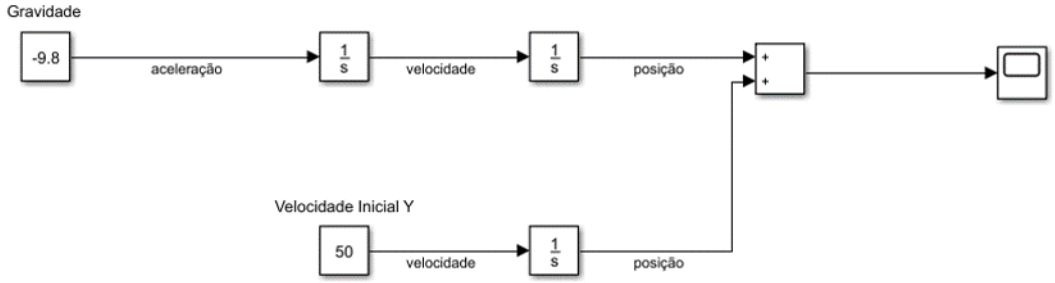


Figura 10.2.39: .

Como a aceleração é a derivada da velocidade, e a velocidade a derivada da posição, podemos integrar a gravidade duas vezes, e a velocidade inicial uma. Somando, teremos a variação de y com o tempo. Ver Fig. 10.2.40

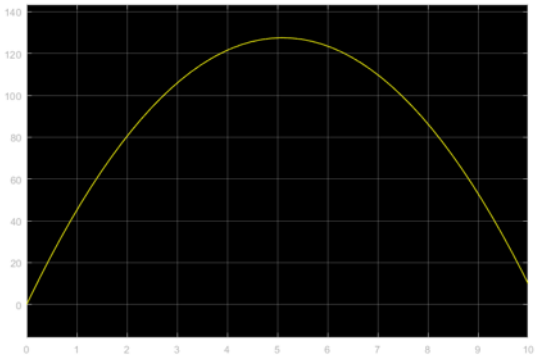


Figura 10.2.40: .

10.7 Vamos adicionar um "*Mux*" e usar para a parte do x(t), uma parte igual a da velocidade inicial de Y. Os dados serão transmitidos para um bloco "*To Workspace*" Após ajustar o valor da velocidade inicial de X e Y, e configurar os parâmetros do bloco que enviará os dados para a área de trabalho para arranjo, basta plotarmos e configurarmos o eixo: Ver Fig. 10.2.41

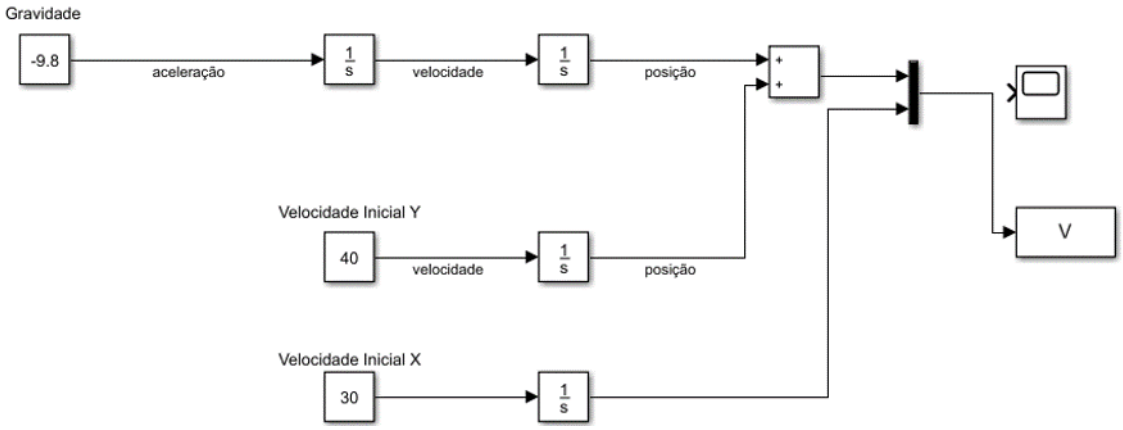


Figura 10.2.41: .

```
>>plot(V(:,2), V(:,1))

>>axis([0 300 0 100])
```

Ver Fig. 10.2.42

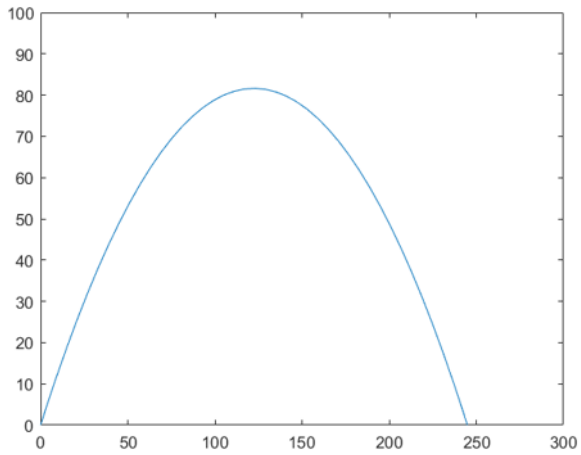


Figura 10.2.42: .

10.8 Primeiro isolamos a segunda derivada e modelamos o sistema da seguinte forma: Ver Fig. 10.2.43 e 10.2.44

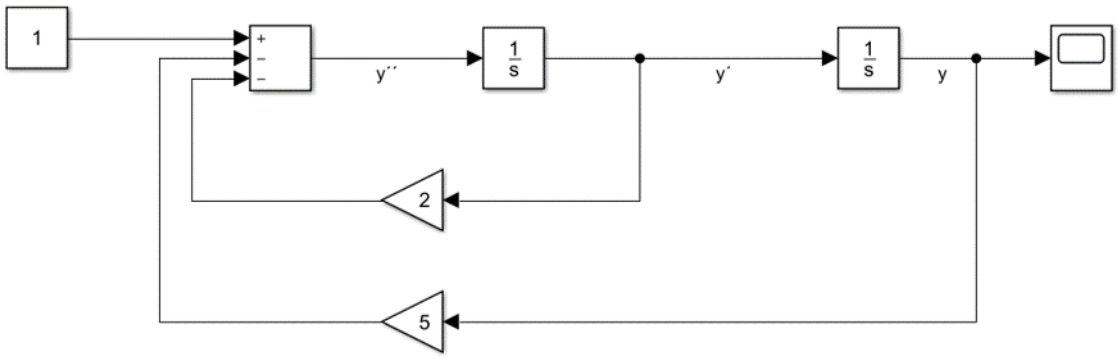


Figura 10.2.43: .

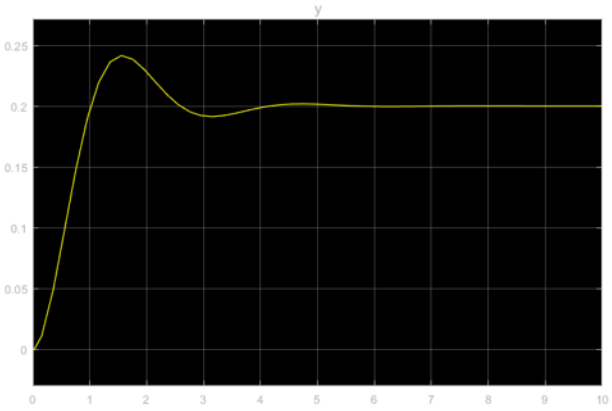


Figura 10.2.44: .

10.9 Primeiro isolamos a segunda derivada e modelamos o sistema da seguinte forma: Ver Fig. 10.2.45 e 10.2.46

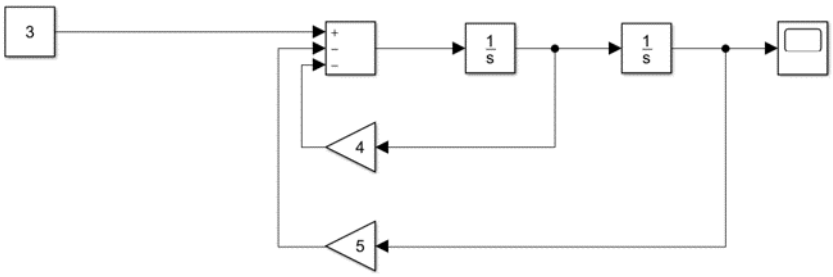


Figura 10.2.45: .

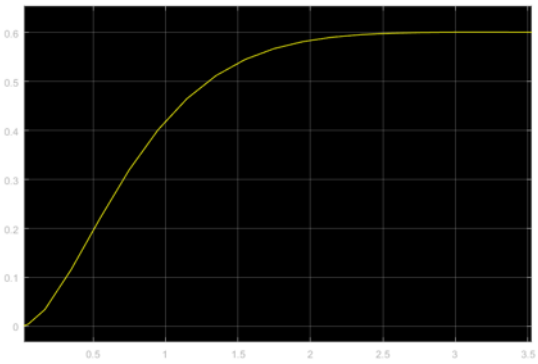


Figura 10.2.46: .

10.10 Basta montarmos o seguinte sistema: Ver Fig. 10.2.47 e 10.2.48

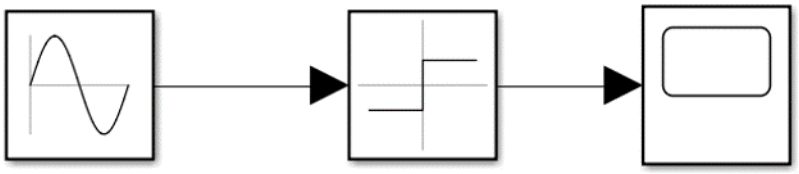


Figura 10.2.47: .

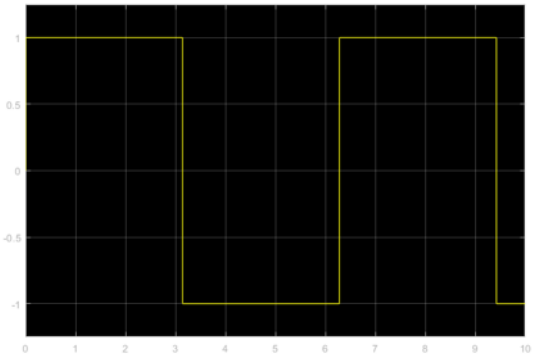
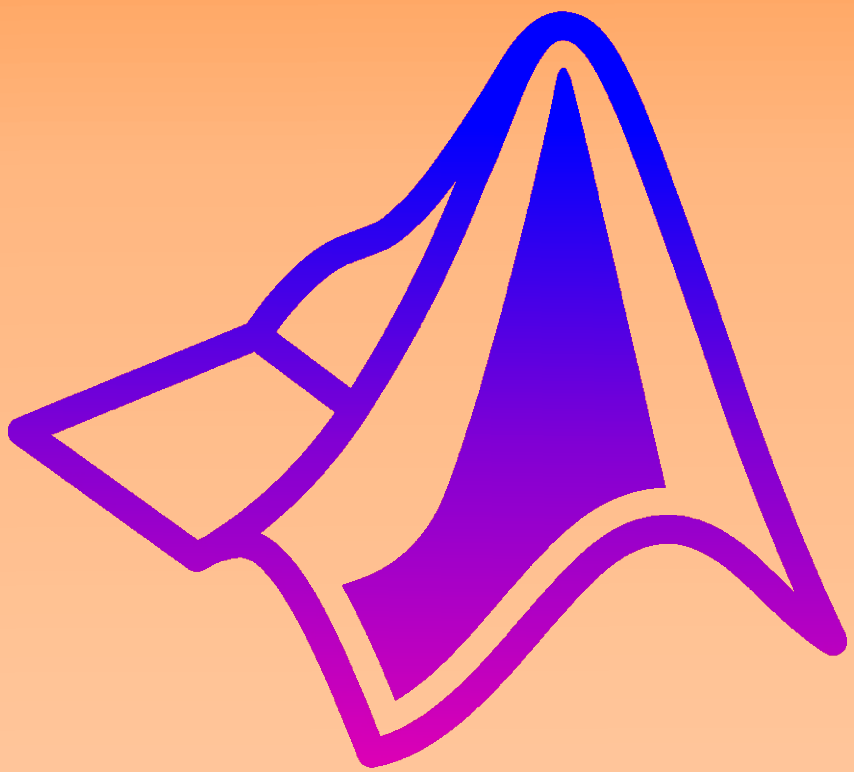


Figura 10.2.48: .

Bibliografia

- https://www.mathworks.com
- Reginaldo de Jesus Santos, "Introdução ao Matlab,"Departamento de Matemática, ICEX, UFMG `http://www.mat.ufmg.br/~regi`
- Frederico Ferreira Campos Filho, "Apostila de Matlab,"Departamento de Ciência da Computação, ICEX, UFMG
- Frederico F. Campos, Introdução ao MATLAB
- Grupo PET, "Curso de MATLAB,"Engenharia Elétrica - UFMS `http://www.del.ufms.br/tutoriais/matlab/apresentacao.htm`
- B.P. Lathi, Sinais e Sistemas lineares
- Pedro Henrique Galdino Silva, Minicurso de MATLAB - Consultoria e Projetos Elétricos Jr.
- Introdução ao Simulink, Apostila adaptada de MANUAL DE INTRODUÇÂO AO MATLAB/SIMULINK, prof. Luis Filipe Baptista



PETEE UFMG

UFMG