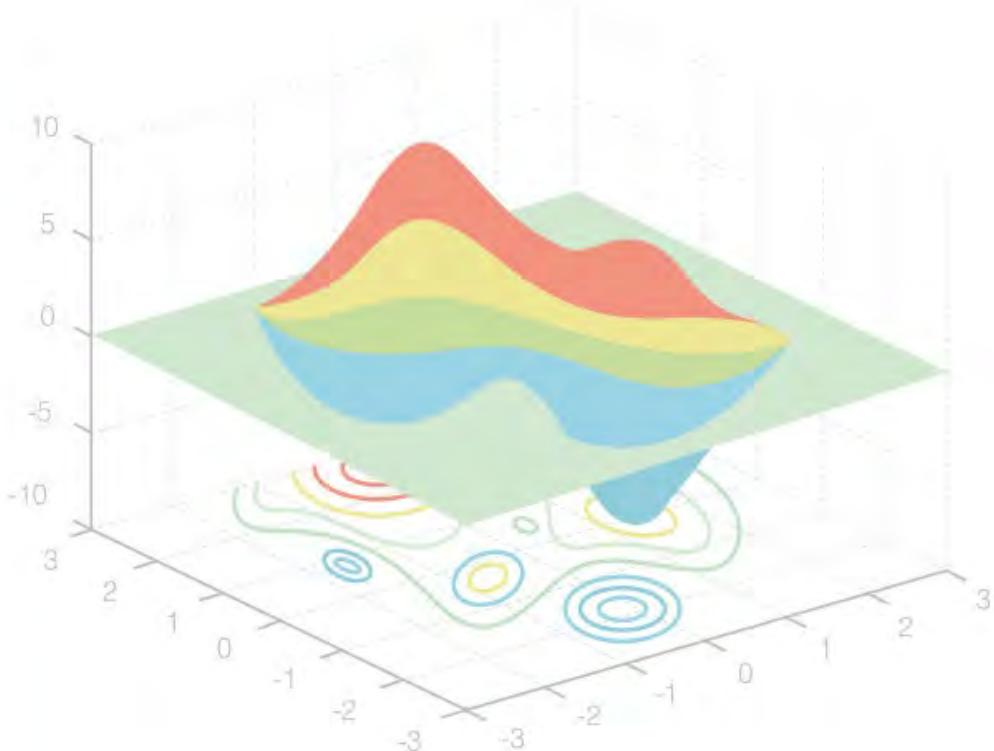


Octave

Seus primeiros passos na
programação científica



Casa do
Código

ALEXANDRE FIORAVANTE DE SIQUEIRA

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2016]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

AGRADECIMENTOS

Agradeço aos editores Paulo Silveira e Adriano Almeida, meus primeiros contatos com a Casa do Código. Também deixo um agradecimento especial à Vivian Matsui, a editora responsável por este livro, por todos os conselhos.

À minha esposa Dayane Machado, pelas leituras, revisões e sugestões.

Ao professor, padrinho e grande amigo Messias Meneguette, por ter me iniciado na linguagem Matlab.

Aos desenvolvedores de *software livre*, especialmente ao John Eaton e à equipe do GNU Octave, que dedicam seu tempo para que tenhamos programas de qualidade, livres em todos os sentidos.

Este livro foi escrito para você. Aproveite!

SOBRE O AUTOR

Alexandre é licenciado em Matemática, mestre e doutor em Ciência e Tecnologia de Materiais pela Unesp de Presidente Prudente, São Paulo. Trabalha há 10 anos com processamento de imagens digitais utilizando GNU Octave e a linguagem Matlab. Entusiasta do *software* livre, atualmente faz sua pesquisa de pós-doutorado na Unicamp, São Paulo. Escreve semanalmente sobre computação científica e *software* livre no blog Programando Ciência (<http://www.programandociencia.com>).

PREFÁCIO

A proposta deste livro é apresentar uma introdução à computação científica por meio do software GNU Octave, que possui estrutura similar à da linguagem Matlab. Matlab é uma linguagem de programação interpretada de alto nível, criada por Cleve Moler no fim da década de 1970.

Voltada ao cálculo matricial, à computação, visualização e programação numérica, suas ferramentas e funções matemáticas embutidas permitem implementar algoritmos mais rapidamente que utilizando ferramentas (como o Microsoft Excel) ou linguagens compiladas (como C, C++ ou Fortran).

O GNU Octave é um software livre que compartilha da filosofia GNU: qualquer usuário possui a liberdade de executá-lo, copiá-lo, distribuí-lo, estudá-lo, modificá-lo e melhorá-lo. Ele foi criado por John Eaton e vários colaboradores (veja a lista completa em <http://goo.gl/IzGafp>), e dispõe de ferramentas para a resolução de problemas relativos à álgebra linear, equações não lineares, equações diferenciais e algébrico-diferenciais, entre outros. Suas funcionalidades podem ser estendidas ao usarmos sua própria linguagem ou criando módulos em outras linguagens.

Este livro está organizado da seguinte maneira:

- Localizaremos os pacotes de instalação do Octave no capítulo *Instalando e iniciando o Octave*. Também veremos como instalá-lo em um sistema Linux, e como iniciá-lo pelo terminal e em sua interface gráfica.
- Depois, no capítulo *Primeiros passos*, aprenderemos a usar o Octave como uma calculadora científica poderosa, com suporte a funções matemáticas e

diferentes tipos de variáveis.

- Estudaremos vários tipos de gráficos de duas e três dimensões no capítulo *Produzindo gráficos no Octave*, além de opções para personalizá-los.
- O capítulo *Gravando e reaproveitando código*, por sua vez, abordará a reutilização de código por meio de scripts e funções. Além disso, aprenderemos a documentar nosso código usando comentários.
- Estruturas para controle de fluxo serão apresentadas no capítulo *Operadores e estruturas para controle de fluxo*. Trabalharemos com operadores relacionais e lógicos, além de estruturas condicionais e de repetição.
- O Octave-Forge, um repositório de extensões para o Octave, é visto com detalhes no capítulo *Octave-Forge: mais poder para seu Octave*. Listaremos os pacotes disponíveis, veremos como instalá-los e trabalhar com eles.
- Por fim, o capítulo *Para onde ir agora?* apresenta documentação e fontes online para continuar seu aprendizado e tirar possíveis dúvidas.

Caso tenha dúvidas ou sugestões, você pode recorrer ao fórum dedicado a este livro, no endereço <http://forum.casadocodigo.com.br>.

Além disso, os exemplos dados ao longo do livro podem ser baixados em <http://goo.gl/VnnAnF>.

Boa leitura e bons estudos!

Sumário

1 Instalando e iniciando o Octave	1
1.1 Obtendo os pacotes de instalação	1
1.2 Instalando o Octave em um ambiente Linux	2
1.3 Iniciando e encerrando o Octave	5
1.4 Resumindo	17
2 Primeiros passos	18
2.1 Meu primeiro programa em Octave	18
2.2 Conseguindo ajuda	19
2.3 Usando o Octave como uma calculadora básica	20
2.4 Estendendo o poder do Octave	22
2.5 Tipos de dados e variáveis	32
2.6 Resumindo	43
3 Operações com variáveis	45
3.1 Criando uma string em branco	45
3.2 Comparando strings	48
3.3 Procurando padrões em strings	49
3.4 Manipulando strings	51
3.5 Criando vetores padronizados	53
3.6 Matrizes especiais	58

3.7 Operações com matrizes e vetores	62
3.8 Resumindo	80
4 Produzindo gráficos no Octave	81
4.1 As funções essenciais para gráficos de duas e três dimensões	81
4.2 Plotando vários gráficos em uma janela	100
4.3 Plotando gráficos em áreas diferentes da janela gráfica	106
4.4 Tipos de gráficos bidimensionais	111
4.5 Tipos de gráficos tridimensionais	126
4.6 Resumindo	136
5 Gravando e reaproveitando código	137
5.1 Scripts	137
5.2 Comentários	139
5.3 Funções	141
5.4 Resumindo	154
6 Operadores e estruturas para controle de fluxo	155
6.1 Operadores	155
6.2 Estruturas de controle	160
6.3 Resumindo	182
7 Octave-Forge: mais poder para seu Octave	183
7.1 Instalando e removendo pacotes do Octave-Forge	183
7.2 Pacotes disponíveis no Octave-Forge	186
7.3 Resumindo	194
8 Para onde ir agora?	196
8.1 Aprendendo sobre o Octave no próprio Octave	196
8.2 Documentação e manuais disponíveis	197
8.3 Buscando ajuda com outros usuários	198
8.4 Como contribuir com o Octave	199

8.5 Resumindo	199
9 Referências bibliográficas	201

CAPÍTULO 1

INSTALANDO E INICIANDO O OCTAVE

Bem-vindo! No decorrer deste livro, aprenderemos como utilizar o Octave para computação científica, trabalhando com funções matemáticas, vetores, matrizes, representação gráfica, funções e ferramentas para problemas mais específicos. Antes de começar a nossa jornada, entretanto, vamos preparar o terreno.

Neste capítulo, deixaremos seu computador pronto para executar o Octave. Saberemos onde encontrar os pacotes de instalação para diferentes sistemas e os passos para instalá-lo em um sistema Linux. Além disso, veremos como iniciar o Octave no terminal Linux e estudaremos os detalhes de sua interface gráfica experimental.

1.1 OBTENDO OS PACOTES DE INSTALAÇÃO

Obter o Octave em um sistema Linux é simples, uma vez que seus pacotes estão presentes nos repositórios de grandes distribuições, como Debian (<http://www.debian.org>), Gentoo (<http://www.gentoo.org/>), Fedora (<http://www.fedoraproject.org/>), openSUSE (<http://www.opensuse.org/>), Arch (<http://www.archlinux.org>), entre outras. Para instalá-lo, utilize o gerenciador de pacotes da sua distribuição.

Sistemas BSD, como o FreeBSD (<https://www.freebsd.org>) e o

OpenBSD (<http://www.openbsd.org/>), também possuem o Octave em seus repositórios. Para obtê-lo em um BSD, consulte as instruções de instalação do seu sistema.

Em ambientes Windows, por sua vez, um pacote .exe pode ser obtido para instalação em <https://ftp.gnu.org/gnu/octave/windows/>. Após o *download*, a instalação do programa pode ser realizada normalmente.

Se você quiser instalar o Octave de uma forma diferente (por meio do *Cygwin*, por exemplo), encontrará informações na página http://wiki.octave.org/Octave_for_Microsoft_Windows.

Por outro lado, se o seu computador rodar o MacOS X, instale o Octave a partir das instruções presentes em http://wiki.octave.org/Octave_for_MacOS_X.

Existe ainda uma versão do Octave portada para o Android, que pode ser obtida no endereço <http://play.google.com/store/apps/details?id=com.octave>.

Por fim, caso prefira compilar seus próprios executáveis, o código-fonte do Octave está disponível em <ftp://ftp.gnu.org/gnu/octave>.

1.2 INSTALANDO O OCTAVE EM UM AMBIENTE LINUX

Há duas formas de instalar o Octave no Linux:

1. Utilizando um gerenciador gráfico de pacotes;
2. Instalando por meio de um terminal.

Instalar o Octave usando um gerenciador gráfico de aplicativos é fácil. Abra o seu gerenciador gráfico preferido. Neste exemplo,

usaremos o *mintInstall Software Manager*, padrão do Linux Mint 17.2, exibido na figura a seguir.



Figura 1.1: O gerenciador de pacotes gráfico padrão do Linux Mint 17.2. Note que a caixa de pesquisa (à direita, acima) está preenchida com o nome do programa que desejamos

A caixa de pesquisa indica que procuramos *octave* ; assim, o gerenciador retornará todos os pacotes que correspondam ao termo procurado. A tela com os resultados da pesquisa aparece na figura adiante. Veja que o ponteiro do mouse indica o pacote que queremos instalar.

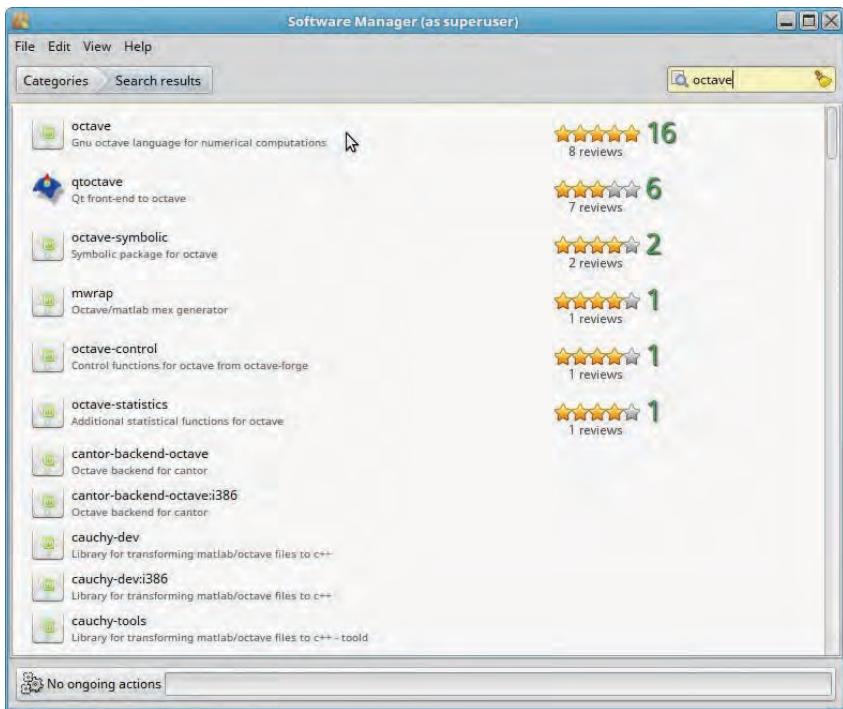


Figura 1.2: A tela referente aos resultados da pesquisa. O pacote Octave desejado é indicado pelo ponteiro do mouse.

Quando clicamos no pacote de interesse, recebemos uma tela com informações sobre o programa, capturas de tela, análises fornecidas por usuários, e um botão que instala o programa e suas dependências. Essa tela é dada na figura a seguir. Para instalar o Octave, pressione *Instalar/Install*.



Figura 1.3: A tela de instalação do Octave. Instale-o clicando em Instalar/Install

A instalação por meio do terminal também não apresenta dificuldades. Em distribuições baseadas em pacotes com extensão .deb (Debian, Ubuntu, Mint, entre outras), por exemplo, abra um terminal e digite o comando seguinte:

```
$ sudo apt-get -y install octave
```

O cifrão (\$) representa o prompt do terminal e não deve ser digitado. Após entrarmos com a senha do administrador, o gerenciador de pacotes apt-get baixa e instala o Octave dos repositórios de sua distribuição. Quando a instalação estiver completa, poderemos iniciar o Octave.

1.3 INICIANDO E ENCERRANDO O OCTAVE

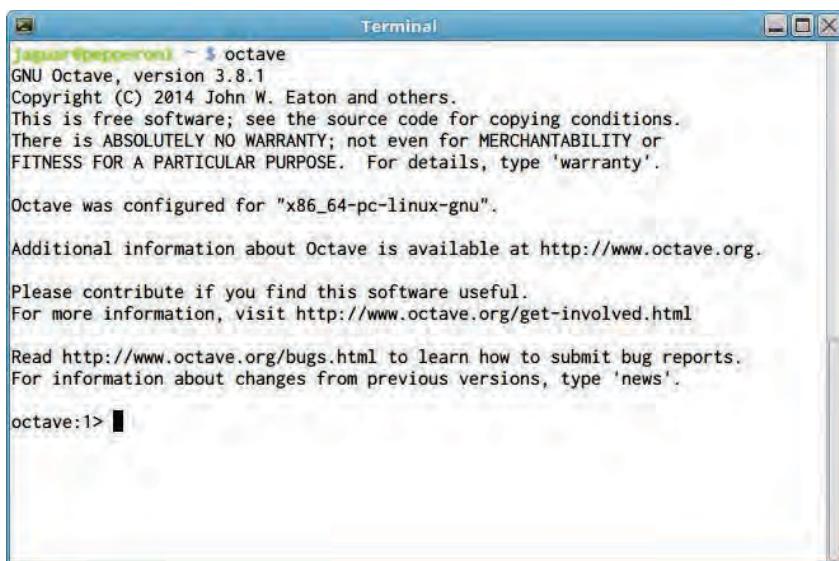
O Octave pode ser iniciado pelos ícones e atalhos criados durante sua instalação, além de podermos executá-lo por meio de um terminal ou em sua interface gráfica. Primeiro, vejamos como iniciá-lo em um terminal Linux.

Iniciando o Octave no terminal Linux

Em um terminal Linux, o Octave pode ser iniciado pelo comando a seguir:

```
$ octave
```

Tecle Enter para executar o comando. Sua inicialização é apresentada na figura a seguir:



The screenshot shows a terminal window titled "Terminal". The window title bar also displays the host name "Jaguar@pepperoni" and the command being run "\$ octave". The terminal output is as follows:

```
Jaguar@pepperoni ~ $ octave
GNU Octave, version 3.8.1
Copyright (C) 2014 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.
Octave was configured for "x86_64-pc-linux-gnu".
Additional information about Octave is available at http://www.octave.org.
Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html
Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.
octave:1> █
```

Figura 1.4: O GNU Octave sendo executado em um terminal Linux

Inicialmente, o Octave exibe informações sobre sua licença, configuração, endereços para a página oficial (<http://www.octave.org>) e informações sobre como contribuir com o projeto (<http://www.octave.org/get-involved>). Além disso, há

instruções para a submissão de relatórios de erros (<http://www.octave.org/bugs.html>). Por fim, o programa apresenta o comando `news`, que informa as mudanças da versão atual em relação às anteriores.

Após as primeiras informações, o Octave exibe o prompt de comando: `octave:1>`. Ele mostra que o programa está pronto para receber instruções, apresentando também o número da instrução atual. Se digitarmos uma instrução, o Octave vai executá-la, retornar a resposta e o prompt se tornará `octave:2>`.

Para simplificar, não vamos nos preocupar com o rótulo `octave` ou o número da instrução no prompt: no decorrer do texto, ele será indicado apenas com o símbolo "maior que" (`>`).

Iniciando o Octave em sua interface gráfica

A versão 3.8 do Octave trouxe uma interface gráfica experimental com editor de texto, depurador de programas, gerenciadores de arquivos e variáveis, entre outras funcionalidades. Essa será a interface padrão a partir da versão 4.0.

Para acessar essa interface em versões anteriores (a partir da 3.8), use o seguinte comando no terminal:

```
$ octave --force-gui
```

Antes de a interface gráfica iniciar pela primeira vez, o Octave fornece três telas de introdução:

- A primeira tela, apresentada na figura a seguir, indica a criação do arquivo de configuração responsável pela interface gráfica. Ele está presente na pasta `/home/usuario/.config/octave/qt-settings`, em que `usuario` é o nome do usuário atual (nesse exemplo, `jaguar`).



Figura 1.5: Primeira tela de introdução à interface gráfica do Octave: criação de um arquivo de configuração

- A segunda tela, mostrada na figura adiante, fornece a opção de receber notícias da comunidade Octave. Se você desejar, o Octave pode conectar-se à página oficial e mostrar notícias atuais da comunidade quando for iniciado. Para isso, marque a caixa de seleção.

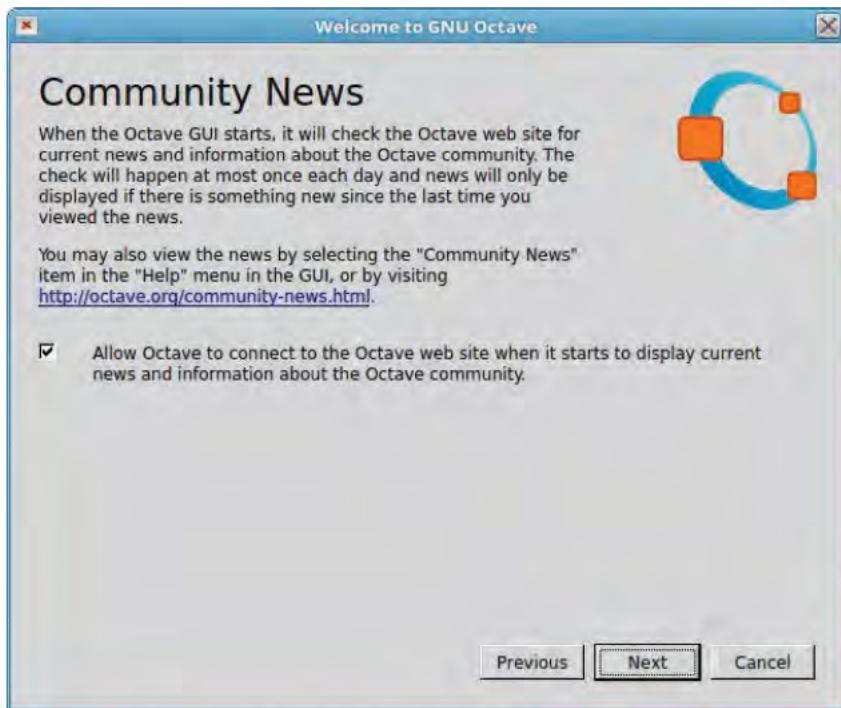


Figura 1.6: Segunda tela de introdução à interface gráfica do Octave: notícias da comunidade Octave

- A última tela instrui sobre o suporte e a documentação do programa, indicando onde encontrar ajuda caso você tenha problemas. Ela é vista na figura a seguir.

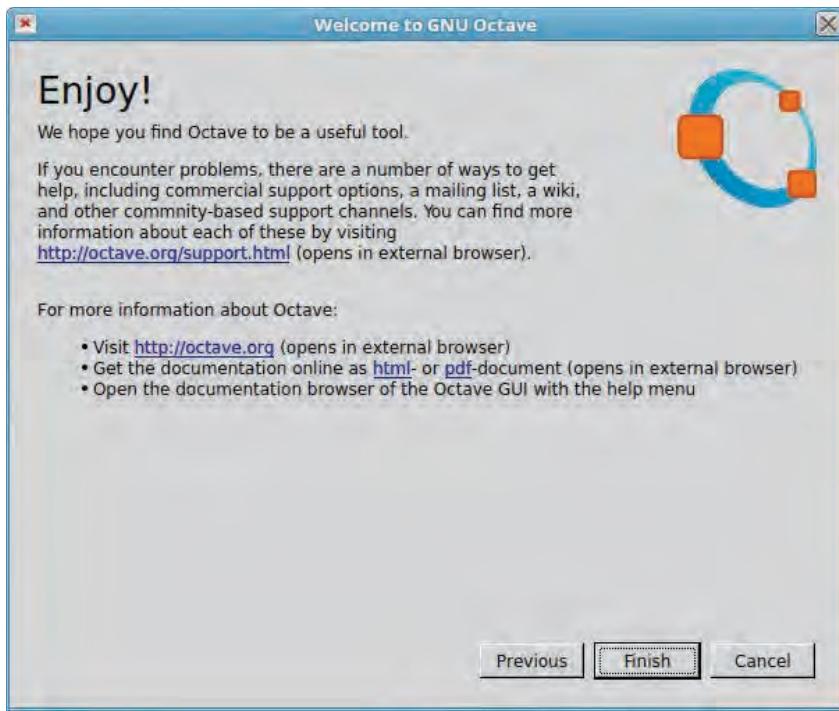


Figura 1.7: Última tela de introdução à interface gráfica do Octave: suporte e documentação

Após as telas iniciais, o Octave é executado em sua interface gráfica, apresentada na figura:

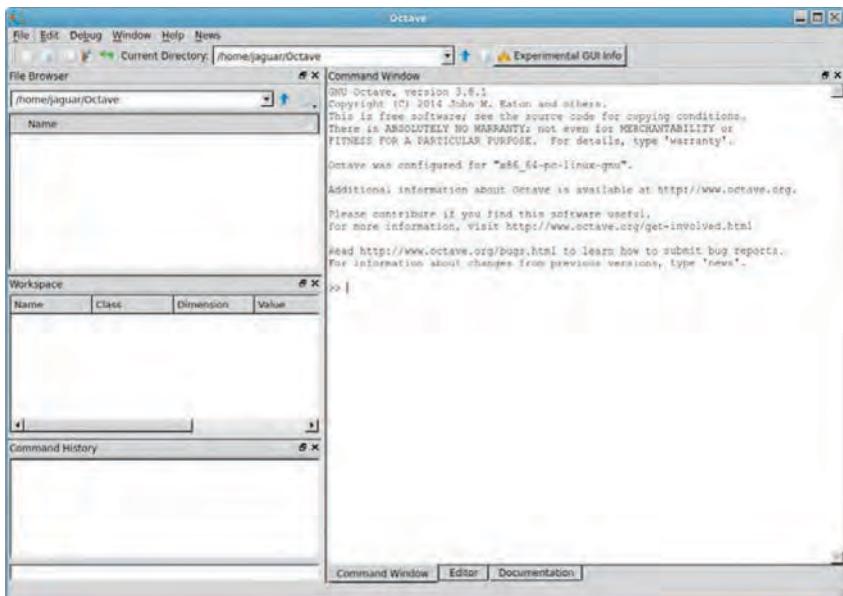


Figura 1.8: A interface gráfica do GNU Octave. Ela será a interface padrão a partir da versão 4.0, substituindo a execução no terminal

Veja que o prompt aparece na *Command Window*, área à direita da interface (figura adiante). O formato do prompt é diferente daquele apresentado pelo Octave rodando em um terminal; nessa janela, ele é dado por dois sinais de "maior que" (`>>`). Como dissemos anteriormente, o prompt será representado ao longo do livro pelo símbolo "maior que" (`>`).

Além da *Command Window*, a área da direita oferece outras duas abas: *Editor*, um editor de texto integrado, e *Documentation*, a área destinada à pesquisa da documentação do Octave.

The screenshot shows the Octave Command Window interface. At the top, there's a title bar with the text "Command Window". Below the title bar is a large text area containing the GNU Octave license and configuration information. At the bottom of this text area, there's a command prompt "=> |". At the very bottom of the window, there's a horizontal tab bar with three tabs: "Command Window" (which is selected), "Editor", and "Documentation".

```
Command Window
GNU Octave, version 3.8.1
Copyright (C) 2014 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

=> |
```

Command Window Editor Documentation

Figura 1.9: Área à direita da interface gráfica do Octave, que apresenta as abas Command Window, Editor e Documentation

À esquerda, existem três elementos, como visto na figura a seguir: *File Browser*, um gerenciador de arquivos; *Workspace*, um gerenciador de variáveis; e *Command History*, o histórico de comandos digitados.

Há uma caixa de texto abaixo de *Command History*. Ela pode ser utilizada para a pesquisa de comandos presentes no histórico.

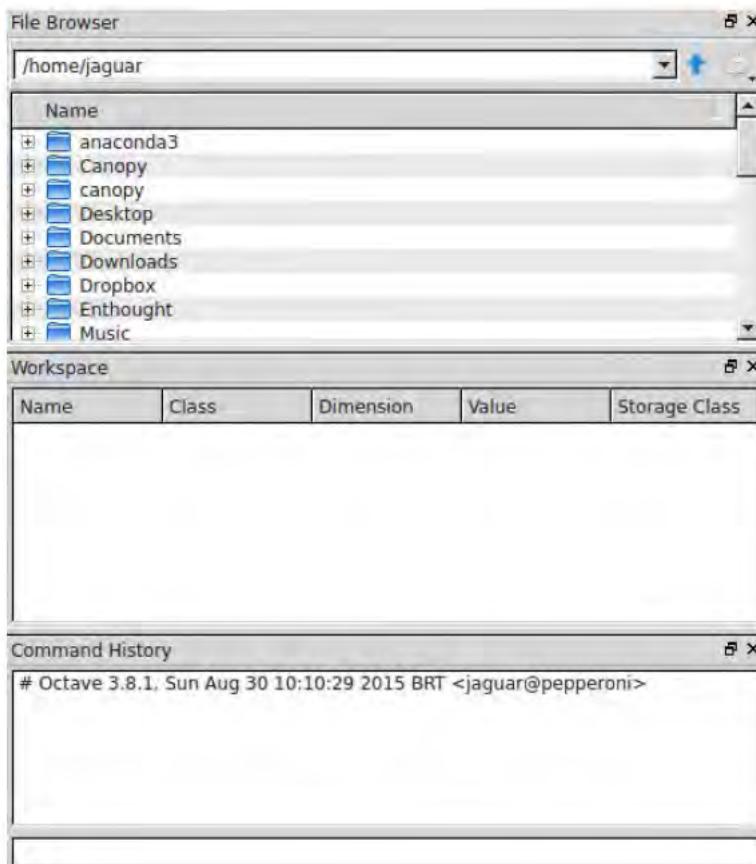


Figura 1.10: Área à esquerda da interface gráfica do Octave. Ela possui as áreas File Browser, Workspace e Command History com sua caixa de pesquisa

A interface gráfica também mostra uma barra de menus com as opções a seguir:

- **File:** comandos de arquivo. Opções para criar um novo arquivo ou abrir um já existente, reeditar arquivos recentes, carregar e gravar espaços de trabalho, alterar configurações e sair.

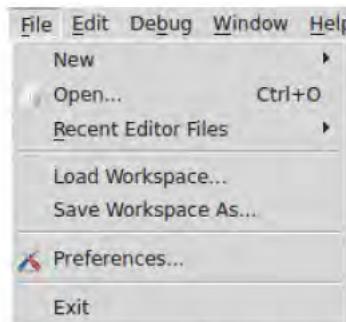


Figura 1.11: Menu File

- **Edit:** comandos de edição simples. Opções para voltar, copiar, colar, limpar área de transferência, localizar arquivos, limpar a janela de comandos, o histórico de comandos e o espaço de trabalho.

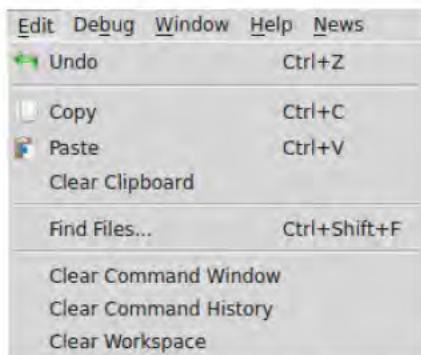


Figura 1.12: Menu Edit

- **Debug:** comandos para depuração. Opções para executar, incluir e excluir passos, continuar a execução e sair do modo de depuração.



Figura 1.13: Menu Debug

- **Window:** opções para alterar a estrutura da interface gráfica. Possibilita mostrar e esconder elementos, assim como redefinir os elementos da interface para sua disposição padrão.

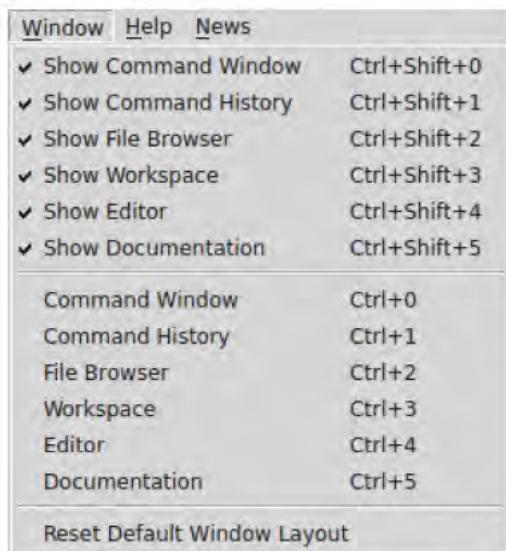


Figura 1.14: Menu Window

- **Help:** indica os caminhos para documentação (em disco e on-line), como reportar um erro, pacotes do Octave-Forge, como contribuir com o Octave e recursos para desenvolvedores, além do texto inicial

contendo informações sobre licença, configuração, endereços on-line, relatórios de erros e mudanças da versão atual.

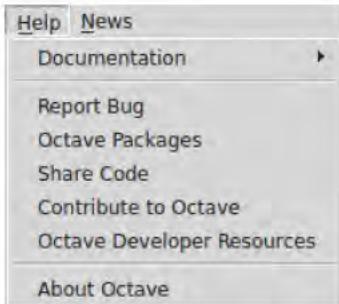


Figura 1.15: Menu Help

- **News:** apresenta opções para as notas de lançamento da versão e as notícias da comunidade Octave.

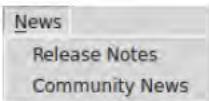


Figura 1.16: Menu News

Abaixo da barra de menus, existe uma barra de ferramentas com ícones para criar um novo *script*, abrir um arquivo, copiar, colar e voltar, verificar a pasta atual, ir para uma pasta acima e mudar de pasta. Essa barra pode ser vista na figura adiante.

Como na versão 3.8.1 a interface gráfica ainda é experimental, há um botão com informações sobre como contribuir com o seu desenvolvimento.

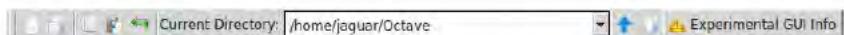


Figura 1.17: Barra de ferramentas que permite criar um novo script, abrir arquivos, copiar, colar e voltar, mudar de pasta e ainda possui informações sobre como auxiliar no desenvolvimento da interface

Encerrando o Octave

Para encerrar o programa, é suficiente digitar no prompt `exit` ou `quit` e teclar `Enter`. No terminal, o Octave também pode ser interrompido com a combinação de teclas `Ctrl + C`. A interface gráfica possui ainda a opção *Exit* no menu *File*.

1.4 RESUMINDO

Neste capítulo, vimos como obter os pacotes de instalação do Octave e como realizar sua instalação em um ambiente Linux. Também aprendemos a executá-lo em um terminal, além de estudarmos a sua interface gráfica. Com o Octave instalado e funcionando, estamos prontos para dar os primeiros passos!

No próximo capítulo, aprenderemos a utilizar o Octave como uma calculadora científica com várias funções disponíveis. Veremos também as variáveis básicas para trabalharmos vetores, *strings* e estruturas. Vamos começar?

CAPÍTULO 2

PRIMEIROS PASSOS

Daremos início à nossa jornada pelo Octave neste capítulo. Vamos passear pelo básico: veremos as funções matemáticas básicas e como lidar com elas. Também aprenderemos uma das coisas mais importantes para dominar uma nova linguagem: como pedir ajuda! De quebra, usaremos o Octave como uma calculadora científica completa e trabalharemos com variáveis básicas, como vetores, *strings* e estruturas, para armazenarmos nossos resultados durante a execução do programa.

2.1 MEU PRIMEIRO PROGRAMA EM OCTAVE

Começaremos criando no Octave a versão do programa mais querido da computação, o *Alô Mundo!*. No prompt do Octave, digite:

```
> disp('Alô Mundo!');
```

Ao final do comando, tecle Enter .

A função `disp` mostra qualquer variável que for colocada entre parênteses na próxima linha. Nesse caso, usamos uma *string*. Strings, representadas entre aspas no Octave, são conjuntos de caracteres. Entenda caractere como praticamente qualquer coisa que possa ser digitada: letras, números, símbolos etc.

Podemos armazenar o texto a ser exibido em uma *variável*:

```
> texto = 'Alô Mundo!';  
> disp(texto);
```

O símbolo `=` (igual) é o *operador de atribuição*. Com ele, atribuímos a expressão à direita do igual para o componente à esquerda. Nesse caso, a variável `texto` recebe a string "Alô Mundo!". Depois usamos a função `disp` para imprimir o conteúdo da variável na tela.

Vamos praticar! Modifique esse código para que ele apresente outra mensagem, como o seu nome por exemplo. Armazene sua mensagem na variável `texto`, depois use a função `disp` para mostrá-la.

2.2 CONSEGUINDO AJUDA

Quando não lembramos como usar corretamente uma função ou comando, podemos obter ajuda utilizando o comando `help`. Por exemplo, para mais informações sobre como usar o comando `disp`, digitamos:

```
> help disp
```

Geralmente, o sistema de ajuda também apresenta exemplos de uso da função ou comando de interesse. Para obter ajuda sobre o próprio sistema de ajuda, digite `help help`. Experimente! Para sair do sistema de ajuda, pressione `q`.

NOTA

A linguagem Matlab é *case sensitive*, ou seja, diferencia maiúsculas de minúsculas! Desse modo, se utilizarmos `HELP` em vez de `help`, o Octave retornará um erro:

```
HELP  
error: 'HELP' undefined near line 1 column 1
```

2.3 USANDO O OCTAVE COMO UMA CALCULADORA BÁSICA

Já pensou em ter sempre à mão uma calculadora com variáveis, *scripts* e gráficos? O Octave possui essas funções e muitas outras, podendo substituir sua calculadora gráfica sem custo algum!

Começaremos a utilizar o Octave como uma calculadora simples, apresentando exemplos das quatro operações básicas. Algumas operações de adição (+), subtração (-), multiplicação (*) e divisão (/) são dadas a seguir:

```
> 1+1  
ans = 2  
> 50-120  
ans = -70  
> 20*3  
ans = 60  
> 170/4  
ans = 42.500
```

O resultado da operação é gravado na variável auxiliar `ans` (do inglês *answer*), que sempre armazena o resultado da última operação realizada. Podemos usá-lo na continuação das operações:

```
> 130-50  
ans = 80  
> ans*2
```

```
ans = 160
> ans/5
ans = 32
```

Além de fazer operações mais longas utilizando `ans`, é possível realizar operações maiores em conjunto, utilizando apenas uma expressão.

```
> 130-50*2+230-140/3
ans = 213.33
```

NOTA

Cuidado com a *precedência* dos operadores! Obedecendo às regras matemáticas, o programa efetua primeiro multiplicações e divisões; depois, adições e subtrações. Quando a precedência é a mesma, o programa faz as operações da esquerda para a direita.

Quando for necessário efetuar uma operação antes, poderemos alterar a precedência dos operadores usando parênteses:

```
4*5+3
ans = 23
4*(5+3)
ans = 32
2/3*2
ans = 1.3333
2/(3*2)
ans = 0.33333
```

Vamos reforçar o nosso aprendizado? No prompt do Octave, calcule as operações a seguir:

```
> 2*(1+1/3+(1*2)/(3*5)+(1*2*3)/(3*5*7)+(1*2*3*4)/(3*5*7*9))
> 3+(8/60)+(29/60^2)+(44/60^3)
> 6*(1/2+1/(2*3*2^3)+(1*3)/(2*4*5*2^5)+(1*3*5)/(2*4*6*7*2^7))
```

Caso o Octave retorne algum erro, confira se não falta algum

elemento nas expressões. Por exemplo, se o prompt retornar apenas `>` e não mostrar o resultado da operação quando os comandos forem digitados, tecle `CTRL + C` e verifique se um dos parênteses está faltando.

Reparou como as operações que acabamos de fazer resultam em números próximos? Elas são *aproximações do número pi*, um número irracional dado por $3,14159265\dots$. Não é necessário decorar esse valor; podemos digitar `pi` para utilizá-lo em uma operação. Um exemplo:

```
> 3*pi/4
ans = 2.3562
```

Vamos praticar! Faça as operações seguintes no Octave:

```
> pi-1/pi
> (pi-1)/pi
> 3*pi-4/10
> 3*(pi-4)/10
> 5+3*2+pi/10
> 5+3*(2+pi/10)
> 5+(3*2+pi)/10
```

2.4 ESTENDENDO O PODER DO OCTAVE

Agora que sabemos usar o Octave como uma calculadora simples, que tal acrescentarmos mais funções? Assim poderemos utilizá-lo como uma calculadora científica completa. Para esse fim, adotaremos algumas das funções matemáticas que estão integradas ao Octave, como veremos a seguir.

NOTA

Se surgir a necessidade de recordar algumas funções matemáticas que serão citadas aqui, existem ótimas referências na internet. Um exemplo é a *Academia Khan* (<http://pt.khanacademy.org/>), que possui vídeos sobre Matemática, entre outros assuntos.

Também há a *Scholarpedia* (<http://www.scholarpedia.org/>), uma encyclopédia aberta cujos artigos são revisados por especialistas (incluindo pesquisadores laureados com os prêmios Nobel, Fields e Dirac).

Fatorial

O fatorial de um número inteiro pode ser calculado pela função `factorial()`.

```
> factorial(10)
ans = 3628800
> factorial(5)
ans = 120
```

NOTA

Obedecendo à definição matemática, o Octave retornará um erro se informarmos um número negativo:

```
factorial(-2)
error: factorial: N must all be non-negative integers
```

Potenciação e radiciação

A potenciação (ou exponenciação) de dois números X e Y , com X elevado a Y , é dada pela função `power(X,Y)` .

```
> power(2,4)
ans = 16
```

Também é possível usar sua forma simplificada, o circunflexo: X^Y .

```
> 2^(-1)
ans = 0.50000
```

Por sua vez, a função raiz quadrada é dada por `sqrt()` . Seu nome é a abreviação do termo em inglês, *square root*. Raízes com índices maiores que 2 podem ser obtidas pela função `nthroot(X,N)` , com X e N sendo o radicando e o índice, respectivamente. Em inglês, *nth root* é a "raiz enésima" de um número.

Veja alguns exemplos:

```
> sqrt(100)
ans = 10
> nthroot(8,3)
ans = 2
> power(2,3)
ans = 8
> nthroot(250,5)
ans = 3.0171
```

Funções complexas

Os números complexos e funções relacionadas a eles também estão disponíveis no Octave. O número imaginário, equivalente à raiz quadrada de -1, é representado por `i` ou `j` . Então, definimos números complexos no Octave por meio de suas partes *real* e *imaginária*. Por exemplo:

```
> 3i
> 2 + 3j
> 4 + (1/2)i
```

A parte imaginária do número complexo é o número acompanhado de `i`. As duas partes podem ser separadas por meio das funções `real()` e `imag()`.

```
> real(2+3i)
ans = 2
> imag(2+3j)
ans = 3
> real(3i)
ans = 0
```

O valor absoluto (ou módulo) de um número complexo é definido como a raiz quadrada de suas partes real e imaginária ao quadrado. Essa operação é dada no Octave pela função `abs()`. Veja no exemplo a seguir como podemos usar tanto a função `abs()` quanto a definição por meio da raiz quadrada para encontrar o valor absoluto de um número:

```
> abs(4+i)
ans = 4.1231
> sqrt(real(4+i)^2 + imag(4+i)^2)
ans = 4.1231
```

No caso de a parte real ou a parte imaginária serem iguais a zero, o valor absoluto se torna o valor do número quando o sinal não é considerado. Veja alguns exemplos:

```
> abs(2)
ans = 2
> abs(-10)
ans = 10
> abs(-3i)
ans = 3
```

Podemos calcular também o ângulo entre o vetor que representa um número complexo e o semieixo positivo das abscissas (`angle()`), ou seu conjugado complexo (`conj()`):

```
> angle(1)
ans = 0
> conj(1)
ans = 1
```

```
> angle(2+3i)
ans = 0.98279
> conj(2+3i)
ans = 2 - 3i
```

Exponencial e logaritmo

Contaremos uma história para introduzir as funções exponencial e logarítmica. Lembra da potenciação, definida no Octave pela função `power()` ou pelo seu atalho, o circunflexo (`^`)? Ela vai ser muito útil.

Um número muito importante na Matemática é o chamado *número de Euler*, também conhecido como número exponencial, entre outros nomes. Ele é dado no Octave por meio do caractere `e`:

```
> e
ans = 2.7183
```

Como visto no exemplo, o Octave representa `e` como 2,7183. Ele é um número irracional, assim como *pi*, e seu valor pode ser aproximado por meio da função fatorial:

```
> 1+1+(1/factorial(2))+(1/factorial(3))+(1/factorial(4))
ans = 2.7083
```

Repare como o termo `1/factorial()` aparece várias vezes na soma. Cada vez que um novo termo é somado, o resultado da expressão se aproxima do valor de *e*. Os próximos termos são `1/factorial(5)` , `1/factorial(6)` , e assim por diante. Experimente somar mais termos à expressão anterior, e verifique como o número obtido se aproxima do valor de *e*.

A função exponencial é obtida pela potenciação da *base*, que chamaremos de *a*, pelo *expoente*, que nomearemos como *n*: a^n . Conforme variamos o valor de *n* , temos vários pontos que determinam uma relação entre *a* e *n*; de forma mais exata, uma função.

Nesse momento, estudaremos apenas um valor de a^n , mantendo n constante. Trataremos mais detalhadamente das funções no capítulo *Gravando e reaproveitando código*.

Imagine que o e é tão importante que existe uma função exponencial só dele! Essa função é a `exp(n)` , chamada de *exponencial natural*. A exponencial natural calcula e elevado a n , e^n , como pode ser visto nos exemplos a seguir:

```
> exp(0)
ans = 1
> e^0
ans = 1
> exp(1/2)
ans = 1.6487
> e^(1/2)
ans = 1.6487
> exp(1)
ans = 2.7183
> e^1
ans = 2.7183
```

A função inversa da exponencial é a função *logaritmo natural* (`log()`), que representa o expoente ao qual a base a é elevada para obter um número. No caso da exponencial e do logaritmo naturais (na base e), $\log(x) = n$ quando $e^n = x$, e vice-versa.

Complicado? Veja um exemplo que mostra essa relação:

```
> log(10)
ans = 2.3026
> e^2.3026
ans = 10.000
```

Precisa de mais bases para trabalhar com logaritmos, fora a base e ? O Octave oferece as funções `log2()` (base 2: $\log_2(x) = n$ quando $2^n = x$) e `log10()` (base 10: $\log_{10}(x) = n$ quando $10^n = x$).

```
> log2(100)
ans = 6.6438562
> log10(100)
```

```
ans = 2
> log2(-1/2)
ans = -1.0000 + 4.5324i
> log10(-1/2)
ans = -0.30103 + 1.36438i
```

Perceba que o Octave arredonda os valores, conduzindo a resultados ligeiramente diferentes:

```
> log10(40)
ans = 1.6021
> 10^1.6021
ans = 40.004
> 10^1.60206
ans = 40.000
```

Tivemos de utilizar `1.60206`, um número mais preciso que `1.6021` para que as expressões retornassem resultados iguais nesse exemplo. Mas não se preocupe! Se quisermos mais precisão nos resultados, podemos usar a função `output_precision()`, que indica o número de caracteres depois do ponto. Como padrão, a resposta ao comando é 5:

```
> output_precision
ans = 5
```

Para alterar o valor para 8, por exemplo, fazemos:

```
> output_precision(8)
> output_precision
ans = 8
```

Calculando novamente o exemplo anterior com `output_precision()` igual a 8, veja que a igualdade é mantida:

```
> log10(40)
ans = 1.6020600
> 10^ans
ans = 40.000000
```

Repare como a precisão aumentou! Lembre-se de que `ans` representa o resultado da última operação.

A função `output_precision()` também pode ser usada para aumentar a precisão dos valores de números irracionais, como `pi`, `e` ou `sqrt(2)`. Experimente! Utilize os exemplos a seguir e veja a diferença entre os resultados com diferentes valores para `output_precision()`:

```
> output_precision(5)
> e
> pi
> log(exp(pi))
> sqrt(2)
> output_precision(25)
> e
> pi
> log(exp(pi))
> sqrt(2)
```

Seno, cosseno e seus amigos

Várias funções trigonométricas estão implementadas no Octave: seno (`sin()`), cosseno (`cos()`), tangente (`tan()`), secante (`sec()`), cossecante (`csc()`) e cotangente (`cot()`).

Para utilizar qualquer uma dessas funções, fornecemos um número e o Octave retorna o resultado:

```
> sin(pi/3)
ans = 0.86603
> cos(pi/4)
ans = 0.70711
> sec(pi/3)
ans = 2.0000
```

Suas funções inversas, arco-seno (o arco cujo seno tem o valor desejado, `asin()`), arco-cosseno (`acos()`), arco-tangente (`atan()`), arco-secante (`asec()`), arco-cossecante (`acsc()`) e arco-cotangente (`acot()`) também estão disponíveis. Veja alguns exemplos:

```
> asin(pi/2)
ans = 1.5708 + 1.0232i
```

```
> acos(pi/3)
ans = 1.0472
> acot(pi)
ans = 0.30817
> acsc(pi/4)
ans = 1.57080 + 0.72337i
```

NOTA

Os argumentos das funções trigonométricas são dados em radianos. Para utilizar as medidas em graus, usamos as funções acrescidas de um d : `sind()`, `cosd()`, `tand()`, e assim por diante. Também vale para as funções inversas: `asind()`, `acosd()`, `atand()` etc.

```
sind(90)
ans = 1
atand(180)
ans = 89.682
cscd(90)
ans = 1
asecd(180)
ans = 89.682
```

Caso seja necessário converter seu ângulo de radianos para graus, use a expressão `rad*180/pi`, em que `rad` é o valor do ângulo em radianos. O resultado é a medida do ângulo em graus:

```
> pi*180/pi
ans = 180
> (pi/2)*180/pi
ans = 90
> (pi/4)*180/pi
ans = 45
> (3*pi/4)*180/pi
ans = 135
> (3*pi/2)*180/pi
ans = 270
```

Funções hiperbólicas

Para obter funções hiperbólicas, basta acrescentar um `h` na frente do nome de sua correspondente trigonométrica: seno hiperbólico é `sinh()`, cosseno hiperbólico é `cosh()`, e assim por diante.

```
> sinh(pi/2)
ans = 2.3013
> cosh(pi/2)
ans = 2.5092
> coth(1/4)
ans = 4.0830
> sech(pi)
ans = 0.086267
```

Assim como no caso das funções trigonométricas, para obter as funções inversas, acrescente um `a` no início do nome da função: arco-seno hiperbólico é `asinh()`, arco-cosseno hiperbólico é `acosh()` etc.

```
> atanh(1/2)
ans = 0.54931
> acsch(pi/3)
ans = 0.84914
```

Vamos reforçar nosso aprendizado? Efetue as operações a seguir:

1. Calcule os três primeiros termos da *série de Maclaurin* que aproximam o valor de $\log(1+0.5)$:

```
> 0.5 - (1/2)*0.5^2 + (1/3)*0.5^3
```

À medida que somamos mais termos, o resultado fica mais próximo do valor real de $\log(1+0.5)$. Experimente! Os próximos termos são $-(1/4)*\text{power}(0.5, 4)$ e $(1/5)*\text{power}(0.5, 5)$.

2. Baseado na aproximação anterior, calcule o valor aproximado de $\log(1+0.5)$ por meio dos dez primeiros termos da série de Maclaurin. Repare que os termos pares possuem sinal

- negativo.
3. Da mesma forma, calcule os três primeiros termos da série de Maclaurin para $\cos(\pi)$:

```
> 1 - (1/factorial(2))*pi^2 + (1/factorial(4))*pi^4
```
 4. Os próximos termos dessa série são -
 $(1/\text{factorial}(6)) * \text{power}(\pi, 6)$ e
 $(1/\text{factorial}(8)) * \text{power}(\pi, 8)$. Verifique que o resultado se aproxima do valor real de $\cos(\pi)$ quando os somamos à série.
 5. Calcule o valor aproximado de $\cos(\pi)$ utilizando os dez primeiros termos da série de Maclaurin.

NOTA

A página sobre a *série de Taylor* da Wikipédia contém uma lista de séries de Maclaurin para as funções mais comuns. Confira em http://en.wikipedia.org/wiki/Taylor_series.

2.5 TIPOS DE DADOS E VARIÁVEIS

Até agora vimos como utilizar o Octave como uma calculadora de resposta imediata: você digita o comando, ele retorna o resultado. Mas para criarmos ferramentas mais complexas, precisamos de uma mãozinha das *variáveis*. Elas nos permitem separar informações em espaços reservados da memória, podendo usá-las sempre que desejarmos.

Vimos um exemplo do uso de variáveis no programa *Alô Mundo!*, no qual a variável `texto` continha uma string. Da mesma

forma, quando necessário, podemos inserir um valor numérico em uma variável:

```
> valor = 2*36  
valor = 72
```

Dizemos que o valor à esquerda ($2*36$) é *atribuído* à variável `valor`. O Octave retorna o valor da variável na próxima linha, como no exemplo anterior. Se você não quiser que essa confirmação seja exibida, adicione um ponto e vírgula (;) na frente da atribuição:

```
> valor = 2*36;
```

No exemplo anterior, a variável `valor` armazenou o resultado da multiplicação, que está reservado para quando precisarmos dele. Para conferir o conteúdo da variável, digitamos seu nome no prompt:

```
> valor  
valor = 72
```

NOTA

O nome de uma variável pode ter em torno de 30 caracteres entre letras, números e o sublinhado: `_`. Contudo, não é permitido iniciar com um número:

```
var1 = 3
var1 = 3
1var = 3
parse error:
  syntax error
>> 1var = 3
```

Como podemos escolher o nome que desejarmos, uma forma mais significativa de identificar as variáveis é relacionar o nome ao objeto armazenado. Por exemplo, a variável `hip` representa melhor a hipotenusa de um triângulo retângulo do que as variáveis `h` ou `a270B535`.

O Octave suporta três tipos de dados básicos:

- **Vetores:** eles são objetos que podem ter várias dimensões e acomodar elementos ordenados em seu interior.
- **Strings:** uma string é um conjunto de caracteres. Um exemplo já citado é o *Alô Mundo!*.
- **Estruturas:** permitem a organização de dados por campos, podendo armazenar vetores e strings.

Também há a possibilidade de trabalhar com classes e com Orientação a Objeto (OO). Entretanto, neste livro, vamos abordar apenas os três tipos mencionados anteriormente. Se você conhece um pouco mais sobre programação e tem interesse em programação orientada a objeto no Octave, a documentação oficial do programa

(<https://www.gnu.org/software/octave/doc/interpreter/Object-Oriented-Programming.html>) pode ser um bom começo.

Vetores

Em Octave, vale a máxima *tudo é um vetor!* Mesmo os números simples são tratados como vetores. Por exemplo, atribuiremos um número à variável `A`:

```
> A = 3  
A = 3
```

Vejamos quais são as dimensões de `A` de acordo com o Octave. Para isso, utilizaremos a função `size()`:

```
> size(A)  
ans =  
    1   1
```

Nesse exemplo, o Octave diz que nossa variável `A` é um vetor de uma linha e uma coluna.

Podemos definir um vetor com mais elementos utilizando colchetes. Por exemplo, atribuiremos números de 1 a 5 ao vetor `V`:

```
> V = [1 2 3 4 5]  
V =  
    1   2   3   4   5
```

Para acessar cada elemento de `V`, os parênteses são necessários: `V(ind)`, em que `ind` é o índice do vetor. O último elemento do vetor também pode ser acessado usando `end` como argumento:

```
> V(1)  
ans = 1  
> V(3)  
ans = 3  
> V(end)  
ans = 5
```

Qual seria o tamanho de `V`? Novamente, é possível calcular

com a função `size`:

```
> size(V)
ans =
 1   5
```

Sabemos que `V` possui uma linha e cinco colunas. Esse tipo de vetor é conhecido como *vetor-linha*, por conter apenas uma linha.

Iniciamos novas linhas em um vetor usando ponto e vírgula (`;`). Dessa forma, podemos definir um vetor com apenas uma coluna, chamado de *vetor-coluna*:

```
> Vcol = [1; 2; 3; 4; 5]
Vcol =
 1
 2
 3
 4
 5
```

Podemos confirmar com a função `size()` que `Vcol` possui cinco linhas e uma coluna:

```
> size(Vcol)
ans =
 5   1
```

NOTA

Lembre-se de que o Octave é *case sensitive*:

```
v = [1 2 3 4 5]
V =
 1   2   3   4   5
v
error: 'v' undefined near line 1 column 1
```

Como `v` é diferente de `V`, o Octave diz que a variável `v` não está definida.

Podemos criar vetores com números de linhas e colunas diferentes de 1. Basta listarmos os elementos e indicarmos com ponto e vírgula onde começa uma nova linha:

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

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

NOTA

As linhas de um vetor devem ter a mesma quantidade de elementos:

```
A = [1 2; 3 4 5]
error: vertical dimensions mismatch (1x2 vs 1x3)
```

Nesse exemplo, o Octave retorna um erro por tentarmos criar um vetor em que a primeira e a segunda linha tenham 2 e 3 elementos, respectivamente.

Verificaremos o tamanho e o comprimento dos vetores `A` e `B` definidos anteriormente. Para determinar o comprimento de uma variável, usaremos a função `length()`:

```
> size(A)
ans =
 3   3
> size(B)
ans =
 3   2
> length(A)
ans = 3
```

```
> length(B)
ans = 3
```

Veja que os tamanhos de `A` e `B` são 3 por 3 (três linhas e três colunas) e 3 por 2 (três linhas e duas colunas), respectivamente. Ao longo do livro, vamos nos referir a um vetor de duas ou mais dimensões como uma *matriz*, um conceito muito importante na Matemática.

Para acessar os elementos de uma matriz, indicamos os índices de linha e coluna entre parênteses. Adotando as matrizes definidas no exemplo anterior, `A(2, 3)` retorna o elemento da segunda linha e da terceira coluna, 1.

Podemos obter o último elemento da matriz usando `A(end, end)`:

```
> A(end, end)
ans = 1
```

Repare que `length()` retorna o comprimento de `A` e `B` como sendo igual a 3. Essa função escolhe o maior número entre linhas e colunas: como `A` possui o mesmo número de linhas e colunas, 3, `length(A)` retorna 3; da mesma forma, como `B` tem três linhas e duas colunas, `length(B)` retorna 3.

Existe ainda a função `numel()`, que retorna o número de elementos de uma variável. Por exemplo, `A` e `B` possuem, respectivamente, nove e seis elementos:

```
> numel(A)
ans = 9
> numel(B)
ans = 6
```

Veremos mais sobre vetores e matrizes no capítulo *Operações com variáveis*.

Strings

Como observamos em exemplos anteriores, definimos uma string colocando seu conteúdo entre aspas:

```
> str1 = 'Este é um exemplo de string';
> str2 = 'Esteéoutroexemplo';
> str3 = 'Este, ainda outro';
> str4 = 'E_mais_outro';
```

Até as strings são vetores no Octave! Utilizando a função `size()`, veja que cada string é considerada um vetor com uma linha e o número de colunas igual ao de elementos:

```
> size(str1)
ans =
    1   28
> size(str2)
ans =
    1   18
> size(str3)
ans =
    1   17
> size(str4)
ans =
    1   12
```

Como já mencionamos, Alô Mundo! também é um exemplo de string. Vamos criar uma variável que guarde a string Alô Mundo! novamente:

```
> texto = 'Alô Mundo!'
```

NOTA

O Octave aceita tanto as aspas simples como as duplas na definição de strings:

```
texto = 'Alô Mundo!'
texto = Alô Mundo!
texto = "Alô Mundo!"
texto = Alô Mundo!
```

Contudo, o ambiente MATLAB da Mathworks não aceita as aspas duplas.

Caso você precise de compatibilidade de código, é recomendável usar apenas aspas simples.

Vejamos as dimensões, o comprimento e a quantidade de caracteres nessa string por meio das funções `size()`, `length()` e `numel()`:

```
> size(texto)
ans =
    1   11
> length(texto)
ans = 11
> numel(texto)
ans = 11
```

Os comandos indicam que `texto` possui 1 linha e 11 colunas, seu comprimento é 11 e o número de elementos também é 11.

Vamos nos aprofundar no estudo das strings no capítulo seguinte.

Estruturas

Estruturas são variáveis mais complexas, que guardam várias informações diferentes. Você pode pensar nelas como caixinhas de

variáveis.

Criaremos uma agenda de endereços e telefones para mostrar como as estruturas funcionam. Digite no prompt:

```
> agenda(1).nome = 'Fulano';
> agenda(1).endr = 'Abbey Road';
> agenda(1).email = 'fulano@abbeyroad.com';
> agenda(1).fone = '18990001234';
```

A variável agenda guarda as informações do nosso primeiro contato. Ao digitar o nome da variável no prompt, o Octave retornará as informações gravadas:

```
> agenda
agenda =
scalar structure containing the fields:
  nome = Fulano
  endr = Abbey Road
  email = fulano@abbeyroad.com
  fone = 18990001234
```

Podemos adicionar outros registros à estrutura gravando mais índices na variável agenda . Nesse exemplo, gravaremos os índices 2 e 3:

```
> agenda(2).nome = 'Sicrano';
> agenda(2).endr = 'Bourbon Street';
> agenda(2).email = 'sicrano@bourbonst.com';
> agenda(2).fone = '219943215300';

> agenda(3).nome = 'Beltrano';
> agenda(3).endr = 'Rua Augusta';
> agenda(3).email = 'beltrano@ruaaugusta.com';
> agenda(3).fone = '11912320091';
```

Agora, ao chamar a variável agenda , o programa retornará o nome dos campos e a quantidade de registros. Nesse exemplo, temos três registros contendo nome , endr , email e fone :

```
> agenda
agenda =
1x3 struct array containing the fields:
  nome
```

```
endr  
email  
fone
```

Para acessar índices na nossa agenda, digitamos `agenda` e o número do índice desejado:

```
> agenda(1)  
> agenda(2)  
> agenda(3)
```

Alterando tipos de variáveis

Podemos alterar o tipo de uma variável, atribuindo a ela um conteúdo diferente do original:

```
> k = [1 2 3; 2 3 1; 3 2 1]  
k =  
 1   2   3  
 2   3   1  
 3   2   1  
  
> k = 3  
k = 3  
  
> k = 'isso é uma string'  
k = isso é uma string
```

Nesse exemplo, definimos a variável `k` como uma matriz, depois como um inteiro, e depois ainda como uma string, e o Octave aceitou a mudança normalmente.

Evitando se perder com uma lista de variáveis

Depois de trabalhar bastante com várias funções, senos e cossenos, somas e tudo o mais, provavelmente teremos várias variáveis definidas. Para vê-las, podemos usar o comando `who`:

```
> who  
Variables in the current scope:  
agenda    estrut    matriz    nome      numero    numero2
```

O comando `whos` é uma variação que mostra uma saída mais detalhada: ele apresenta o nome e as dimensões das variáveis (colunas `Name` e `Size`), a quantidade de *bytes* usados (`Bytes`) e sua classe (`Class`).

Com as variáveis definidas anteriormente, o resultado do comando `whos` é:

```
> whos  
Variables in the current scope:
```

Attr	Name	Size	Bytes	Class
=====	=====	=====	=====	=====
	agenda	1x3	154	struct
	estrut	1x1	13	struct
	matriz	3x3	72	double
	nome	1x9	9	char
	numero	1x1	8	double
	numero2	1x1	8	double

```
Total is 24 elements using 264 bytes
```

Por fim, para deletar uma variável, podemos usar a função `clear` seguida do nome da variável. Por exemplo:

```
> clear agenda
```

Assim, a variável `agenda` é apagada. Confirme com o comando `who`:

```
> who  
Variables in the current scope:
```

```
estrut    matriz    nome    numero    numero2
```

Se o interesse é apagar todas as variáveis, podemos usar o comando `clear all`. Faça um teste! Confira depois com o comando `who`.

2.6 RESUMINDO

Neste capítulo, focamo-nos na utilização básica do Octave. Digitamos um programa simples (o *Alô Mundo!*), aprendemos a obter ajuda sobre comandos e funções, usamos o Octave como uma calculadora científica e assimilamos várias funções matemáticas para nossa caixa de ferramentas. Por fim, vimos como trabalhar com variáveis básicas (vetores, strings e estruturas).

No próximo capítulo, continuaremos trabalhando com variáveis, aprendendo a fazer operações entre elas e adicionando mais funções à nossa biblioteca pessoal. Também faremos operações com vetores e matrizes, obteremos matrizes inversas, transpostas, autovalores e autovetores de uma matriz, entre outros.

As próximas funções facilitarão a construção de programas científicos com menos esforço. Até lá!

CAPÍTULO 3

OPERAÇÕES COM VARIÁVEIS

Neste capítulo, colocaremos a mão na massa! Ou melhor, nas variáveis! Criaremos variáveis especiais e faremos operações entre elas. Começaremos criando, modificando e comparando strings.

3.1 CRIANDO UMA STRING EM BRANCO

No capítulo anterior, vimos que uma string é criada quando digitamos seu conteúdo entre aspas simples:

```
> 'E assim criamos strings'  
ans = E assim criamos strings
```

Em um programa interativo, entretanto, pode ser útil criar uma string vazia para armazenar o texto do usuário. Nesse caso, podemos usar a função `blanks(n)`, em que `n` é a quantidade de caracteres que serão reservados.

Por exemplo, atribuiremos dez espaços na variável `vazio` para uso posterior:

```
> vazio = blanks(10)  
vazio =
```

Repare que o Octave retornou a variável `vazio` como um espaço em branco! Como saber se o espaço que pedimos está realmente lá? Para isso, usaremos a função `whos`, que mostra todas

as variáveis definidas:

```
> whos
Variables in the current scope:

Attr Name      Size            Bytes  Class
==== ==       =====           ===== 
vazio        1x10             10    char

Total is 10 elements using 10 bytes
```

Observe que o tamanho da variável `vazio` é de 10 elementos e sua classe é `char` (caractere), indicando-nos que ela é uma string de 10 caracteres.

No próximo exemplo, combinaremos as funções `disp()` e `blanks()` para criar um programa que receba uma opção do usuário.

Imagine que estamos na *Octave Burgers*, uma famosa rede de restaurantes. O sistema está sendo automatizado com terminais que executam um programa para receber o pedido dos clientes. Como o sistema está em testes, ele fornece opções apenas para um sanduíche e um refrigerante:

```
> opcao = blanks(1);
> disp('Bem-vindo ao Octave Burgers! Qual é o seu pedido?');
> disp('Linus Burger: escolha com a letra L.');
> disp('Open Soda: escolha com a letra O.');
> opcao(1) = input('Qual será seu pedido? Escolha L ou O. ', 's');
> disp('Seu pedido foi:');
> disp(opcao);
```

Esse programa se apresenta, exibe algumas opções e espera a entrada do usuário. Note os novos elementos na linha de código a seguir:

```
opcao(1) = input('Qual será seu pedido? Escolha L ou O. ', 's');
```

Primeiro, atribuímos um valor à variável `opcao` de uma forma diferente. Aqui, definimos a *posição* que o valor deve ocupar. Como

`opcao` foi criada apenas com um espaço (`blanks(1)`), indicamos que o valor escolhido pelo usuário será alocado nesse espaço (`opcao(1)`). Se a variável tivesse 10 espaços e quiséssemos gravar a informação no sétimo espaço, usariámos este código:

```
> opcao = blanks(10);
> opcao(7) = input('Qual será seu pedido? Escolha L ou O. ', 's');
```

Nesse programa, também introduzimos a função `input()`, que exibe um texto (como a função `disp`) e espera uma resposta do usuário, que pode ser atribuída a uma variável. Essa função é muito útil na criação de programas que recebem valores ou caracteres durante sua execução.

No código anterior, utilizamos `input()` com a seguinte estrutura: `variavel = input('texto a ser exibido', 's')`. O argumento '`s`' indica que a variável recebida é tratada como uma string.

Se quisermos que a variável obtida seja uma string e '`s`' não estiver presente na definição de `input()`, devemos utilizar aspas. Caso contrário, o Octave interpretará a entrada como uma variável:

```
> opcao = blanks(1);
> opcao(1) = input('Qual sera seu pedido? Escolha L ou O. ');
Qual sera seu pedido? Escolha L ou O. L
error: 'L' undefined near line 1 column 1
> opcao
opcao =
```

Note que o erro foi ocasionado por não inserirmos aspas para definir nossa escolha. Se usarmos aspas, conseguiremos preencher corretamente a variável `opcao`:

```
> opcao = blanks(1);
> opcao(1) = input('Qual sera seu pedido? Escolha L ou O. ');
Qual sera seu pedido? Escolha L ou O. 'L'
> opcao
opcao = L
```

Portanto, para que a função `input()` receba strings sem problemas, use o argumento '`s`' em sua definição.

3.2 COMPARANDO STRINGS

Imagine a seguinte situação: estamos trabalhando com um arquivo de texto e precisamos saber se duas frases em diferentes pontos do texto são idênticas. Podemos comparar strings pequenas rapidamente, certo? E se tivermos strings de vários caracteres, ou se desejarmos automatizar o processo?

Para comparar duas strings no Octave, usaremos a função `strcmp(string1,string2)`. Se `string1` e `string2` forem iguais, o Octave retornará 1; caso contrário, o resultado será zero. Veja no exemplo a seguir:

```
> string1 = 'A1b2C3d4E5f6G7h8I9';
> string2 = 'A1b2C3d4E5f6G7h8I9';
> strcmp(string1,string2)
ans = 1
> string3 = 'A1b2C3d4e5f6G7h8I9';
> strcmp(string1,string3)
ans = 0
```

Repare que `strcmp` é *case sensitive!* No exemplo anterior, `string1` e `string3` possuem os mesmos caracteres, exceto na posição 9, em que `string1(9)` e `string3(9)` são E e e, respectivamente. Quando a diferença entre maiúsculas e minúsculas não é importante, usamos a função `strcmpi(string1,string2)`:

```
> string1 = 'A!bCdEfGhI%';
> string3 = 'A!bCdEfGhI%';
> strcmpi(string1,string3)
ans = 1
```

Caso o interesse seja saber apenas se um certo número de caracteres é igual, a função utilizada é a `strncmp(string1,string2,n)`, em que `n` é o valor do último

elemento a ser comparado:

```
> string1 = 'AbCdEfGhIjKlMnOpQrStUvWxYz';
> string2 = 'AbCdEfGhIjKlMnOpQrStUvWxYz';
> strcmp(string1,string2,9)
ans = 1
> strcmp(string1,string2,10)
ans = 0
```

Nesse exemplo, as strings são idênticas até o nono elemento; `string1(10)` é igual a `j`, enquanto que `string2(10)` é igual a `J`. Observamos então que `strcmp` também é *case sensitive*. Para essa situação há a função `strncMPI(string1, string2, n)`, que não é sensível:

```
> string1 = 'AbCdEfGhIjKlMnOpQrStUvWxYz';
> string2 = 'AbCdEfGhIjKlMnOpQrStUvWxYz';
> strncMPI(string1, string2, 10)
ans = 1
```

Perceba que as funções `strcmp` e `strncMP` são semelhantes, assim como `strcmPI` e `strncMPI`.

Quando o argumento `n` nas funções `strncMP` e `strncMPI` é igual ao tamanho da string:

- `strcmp` é equivalente a `strncMP`;
- `strcmPI` é equivalente a `strncMPI`.

3.3 PROCURANDO PADRÕES EM STRINGS

E se o desafio fosse procurar algum elemento em uma string? É fácil localizar uma letra ou número em particular em strings pequenas, mas e quando a string tem muitos caracteres? Nesse caso, é melhor ter a ajuda de funções como as apresentadas a seguir.

Uma das funções para procurar por padrões em uma string é a `index(string, padr, dir)`, em que fornecemos o padrão `padr` a ser pesquisado em `string` na direção `dir` ('first' ou

'last' para começar a pesquisar do começo ou do final, respectivamente).

Vamos testar seu funcionamento em uma string razoavelmente grande:

```
> str1 = 'abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz';
> index(str1, 'a', 'first')
ans = 1
> index(str1, 'b', 'last')
ans = 28
> index(str1, 'abc')
ans = 1
```

Repare no último comando: omitimos o último argumento, e ele entendeu como 'first'! Para "omitir" o 'last', pode-se usar outra função, a `rindex()`, que começa a pesquisa do final. Os argumentos são a string e o padrão de interesse, os mesmos da função `index()`:

```
> rindex(str1, 'abc')
ans = 27
> rindex(str1, 'bcd')
ans = 28
```

Nesse exemplo, utilizamos strings como padrão de pesquisa, e a função retornou o índice do primeiro caractere da sequência desejada. Quando procuramos por um argumento que não existe na string, as funções `index()` e `rindex()` retornam zero:

```
> index(str1, 'bd')
ans = 0
> rindex(str1, 'kmp')
ans = 0
```

Repare que `str1` possui dois caracteres `a` e `b`, mas a função `index()` retorna apenas o índice do primeiro caractere correspondente. Para encontrar mais elementos correspondentes contidos em uma string, podemos usar a função `strfind(string,padr)`. Da mesma forma que as funções `index()` e `rindex()`, os argumentos de `strfind()` são a

string desejada e o padrão padrão a ser pesquisado na string. Essa função retorna um vetor contendo os índices dos elementos correspondentes:

```
> str2 = 'abcdefghijklmnopqrstuvwxyzwxyzvwutsrqponmlkjihgfedcba';
> strfind(str2, 'a')
ans =
 1 51
> strfind(str2, 'c')
ans =
 3 49
> strfind(str2, 'cb')
ans = 49
> strfind(str2, 'z')
ans = 26
```

3.4 MANIPULANDO STRINGS

Vejamos algumas funções para conectar strings, pesquisar padrões e substituí-los por diferentes elementos. Uma das funções que podemos usar para unir strings é a `strcat({string1, string2, ...})`. Ela cria uma string que contém todas as strings passadas como argumento. Podemos usá-la para unir quantas strings quisermos:

```
> strcat('Octave', 'é', 'demais')
ans = Octaveédemais
> strcat('O', 'Octave', 'facilita', 'a', 'sua', 'vida')
ans = 0Octavefacilitaasuavida
```

Strings presentes em variáveis também podem ser unidas pela função `strcat()`. É possível encadear várias strings, formando frases:

```
> str_1 = 'Octave';
> str_2 = 'é';
> str_3 = 'demais';
> str_4 = 'facilita';
> str_5 = 'sua';
> str_6 = 'vida';
> strcat(str_1, str_2, str_3)
ans = Octaveédemais
```

```
> strcat(str_1,str_4,str_5,str_6)
ans = Octavefacilitasuavida
> strcat(str_1,str_2,str_3,str_4,str_5,str_6)
ans = Octaveédemaisfacilitasuavida
> strcat(str_1,str_2,str_3,'e',str_4,str_5,str_6)
ans = Octaveédemaisefacilitasuavida
```

Como visto nesses exemplos, as strings resultantes são um bloco de caracteres. Podemos usar a função `strjoin({string1,string2,...},sep)` para unir strings, inserindo o separador `sep` entre cada string dada como argumento. Quando omitido, o separador padrão é o espaço:

```
> strjoin({'Octave','é','demais'})
ans = Octave é demais
> strjoin({'Octave','é','demais'},'*')
ans = Octave*é*demais
> strjoin({'O','Octave','facilita','a','sua','vida'})
ans = O Octave facilita a sua vida
```

Com as funções `input()` e `strcat()`, escreveremos um programa que obtenha o nome do usuário e retorne uma saudação a ele:

```
> usuario = input('Qual o nome do usuário? ','s')
Qual o nome do usuário? Linus
usuario = Linus
> strcat('Graaaande',usuario,'! Bem-vindo!')
ans = GraaaandeLinus! Bem-vindo!
```

O nome do usuário voltou colado na string anterior! Para melhorar o resultado, podemos usar as funções `strcat()` e `strjoin()` em conjunto:

```
> usuario = input('Qual o nome do usuário? ','s')
Qual o nome do usuário? Linus
usuario = Linus
> string1 = strjoin({'Graaaande',usuario});
> strcat(string1,'! Bem-vindo!')
ans = Graaaande Linus! Bem-vindo!
```

Observe que atribuímos o nome do usuário e parte da mensagem a uma string, `string1`, e o espaço foi incluído.

Podemos também embutir a função `strjoin()` em `strcat()` , em vez de criar outra string. O resultado é o mesmo:

```
> usuario = input('Qual o nome do usuário? ','s')
Qual o nome do usuário? Linus
usuario = Linus
> strcat(strjoin({'Graaaande',usuario}),'! Bem-vindo!')
ans = Graaaande Linus! Bem-vindo!
```

Para substituir caracteres e padrões em uma string, empregaremos a função `strrep(string, padvelho, padnovo)` . Precisamos fornecer uma string, um padrão antigo (`padvelho`) e o novo padrão (`padnovo`), que substituirá o antigo na variável `string` .

Por exemplo, vamos mudar o nome do usuário de Linus para Stallman na resposta do programa anterior:

```
> strlinus = 'Graaaande Linus! Bem-vindo!';
> strrep(strlinus,'Linus','Stallman')
ans = Graaaande Stallman! Bem-vindo!
```

Podemos também substituir todos os `in` na string principal por uma sequência qualquer, digamos `12345@?` :

```
> strlinus = 'Graaaande Linus! Bem-vindo!';
> strrep(strlinus,'in','12345@?')
ans = Graaaande L12345@?us! Bem-v12345@?do!
```

NOTA

Alguns dos exemplos apresentados nas seções anteriores contiveram strings formadas apenas por letras. No entanto, lembre-se de que uma string também pode conter números e símbolos.

3.5 CRIANDO VETORES PADRONIZADOS

Nesta seção, aprenderemos as funções necessárias para criar vetores que obejam a alguns padrões. Com elas, poderemos obter variáveis igualmente espaçadas com vários elementos, sem ter de pensar muito sobre isso.

Esses vetores serão particularmente úteis na construção de gráficos de funções, como veremos no capítulo *Produzindo gráficos no Octave*.

Criando vetores por incremento

Imagine que precisamos de um programa que apresente distâncias de 1 a 5 centímetros. Para isso, criaremos no Octave uma variável que contenha os números de 1 a 5. Uma solução possível é a vista no capítulo anterior, na qual definimos um vetor e seus elementos, um de cada vez:

```
> vetor = [1 2 3 4 5]
vetor =
    1   2   3   4   5
```

Esse método funciona para vetores pequenos, mas quando tivermos de criar vetores de 50, 100 ou mais elementos, ele não será viável. Nesses casos, a *notação de dois pontos* será muito útil. Assim, o mesmo vetor definido anteriormente poderia ser obtido pela expressão a seguir:

```
> vetor = 1:5
vetor =
    1   2   3   4   5
```

Simples, não? É só lembrar da fórmula **número inicial : número final**. Dessa forma, fica fácil definir um vetor com mil números:

```
> vetor = 1:1000;
```

NOTA

Nesse caso, a variável `vetor` tem mil elementos! Usamos ponto e vírgula (;) para que o Octave não mostre o conteúdo atribuído à variável.

Se houver a necessidade de construir um vetor que tenha um *incremento* (ou passo) menor ou maior que 1 entre seus elementos, definiremos o incremento entre o número inicial e o número final. Se desejarmos um passo de 50 entre 0 e 1000, a fórmula será **número inicial : 50 : número final**, como no exemplo a seguir.

Quando o número de colunas do resultado é muito grande para ser mostrado em uma linha, o Octave divide a variável em quantas linhas for preciso e informa as colunas exibidas em cada intervalo.

```
> vetor_inc = 0:50:1000
vetor_inc =
  Columns 1 through 11:
    0      50     100     150     200     250     300     350     400
 450      500
  Columns 12 through 21:
    550     600     650     700     750     800     850     900     950
 1000
```

Criando vetores espaçados

Outra forma de definir um vetor no Octave é utilizar funções que criam vetores igualmente espaçados. A função `linspace()` , por exemplo, define um vetor espaçado linearmente. Ela é dada por `linspace(base, limite, pontos)` , com `base` sendo o ponto inicial do vetor e `limite` , seu ponto final. O número de pontos que vai separar os extremos inicial e final é informado em `pontos` .

Alguns exemplos de uso da função `linspace()` são dados a

seguir:

```
> vetor_esp = linspace(1,10,6)
vetor_esp =
    1.0000    2.8000    4.6000    6.4000    8.2000   10.0000
> vetor_esp2 = linspace(2,10,7)
vetor_esp2 =
    2.0000    3.3333    4.6667    6.0000    7.3333    8.6667
10.0000
```

NOTA

É possível utilizar `linspace()` informando apenas `base` e `limite`. Desse modo, o Octave assume que o número de pontos é igual a 100.

Repare que, quando tomamos um valor para `base` maior que o valor de `limite`, o vetor é definido de forma *decrescente*:

```
> vetor_inv = linspace(10,1,6)
vetor_inv =
    10.0000    8.2000    6.4000    4.6000    2.8000    1.0000
> vetor_inv2 = linspace(10,2,7)
vetor_inv2 =
    10.0000    8.6667    7.3333    6.0000    4.6667    3.3333
2.0000
```

A função `linspace()` pode ser usada para criar vetores iguais aos obtidos com a notação de incremento. Nesse caso, é suficiente saber quantos elementos possui o vetor criado por incremento e usar esse valor como o número de pontos na função `linspace()`:

```
> vetor_inc = 0:50:1000;
> length(vetor_inc)
ans = 21
> vetor_incesp = linspace(0,1000,21)
vetor_incesp =
  Columns 1 through 11:
    0      50     100     150     200     250     300     350     400
450      500
```

```
Columns 12 through 21:  
550    600    650    700    750    800    850    900    950  
1000
```

Nesse exemplo, utilizamos a função `length()` para calcular a quantidade de elementos de `vetor_inc`.

Outra função disponível é a `logspace()`, que define os valores espaçados logaritmicamente em vez de linearmente. Sua forma é idêntica à da função `linspace()`, mas existe um detalhe importante! Os números `base` e `limite` são considerados potências de dez.

O Octave entende como se digitássemos `logspace(10^base, 10^limite, pontos)`.

Vamos exemplificar essa situação. No exemplo a seguir, tomamos `base = 2` e `limite = 3`. Como dissemos, o primeiro elemento do vetor é igual a 100 (10 ao quadrado) e o último é igual a 1000 (10 ao cubo):

```
> logspace(2,3,7)  
ans =  
100.00    146.78    215.44    316.23    464.16    681.29  
1000.00
```

NOTA

Quando o valor de `pontos` é omitido, a função `logspace()` o considera igual a 50 (e não 100, como na função `linspace()`).

Veja mais alguns exemplos:

```
> vetor_log = logspace(0,2,3)  
vetor_log =  
1    10    100
```

```
> vetor_log2 = logspace(1,3,5)
vetor_log2 =
    10.000    31.623   100.000   316.228   1000.000
```

Da mesma forma, na função `linspace()`, quando tomamos um valor para `base` maior que o valor de `limite`, o vetor é definido de forma decrescente:

```
> vetor_loginv = logspace(0,2,3)
vetor_loginv =
    1    10    100
> vetor_loginv2 = logspace(1,3,5)
vetor_loginv2 =
    10.000    31.623   100.000   316.228   1000.000
```

Veremos agora a exceção à regra. Quando tomamos `limite = pi`, o Octave não interpreta `pi` como 10^{π} ! Observe:

```
> logspace(0,pi,10)
ans =
Columns 1 through 8:
    1.0000    1.1356    1.2897    1.4646    1.6632    1.8888    2.1450
2.4360
Columns 9 and 10:
    2.7664    3.1416
```

NOTA

O caso especial `pi` na função `logspace()` existe no Octave para compatibilidade com o programa MATLAB, da MathWorks.

Verifique que essa condição vale para `pi`, mas deixa de existir para `2*pi`, `3*pi`, e assim por diante.

3.6 MATRIZES ESPECIAIS

Antes de seguirmos para as operações com vetores e matrizes,

veremos algumas funções para gerar matrizes especiais no Octave. Elas evitarão que digitemos muitas linhas para criar matrizes básicas, auxiliando na compreensão do código.

Começaremos com uma das matrizes especiais mais básicas, dada pela função `eye()`. Ela retorna uma matriz cujos elementos são zeros, exceto pelos elementos da diagonal, que recebem o valor 1. Essa matriz é conhecida como *matriz identidade*. O valor dado como argumento na função determina o número de "uns" na diagonal, sendo uma forma de indicar o tamanho do vetor criado:

```
> eye(1)
ans = 1
> eye(2)
ans =
Diagonal Matrix
 1  0
 0  1
> eye(3)
ans =
Diagonal Matrix
 1  0  0
 0  1  0
 0  0  1
> eye(4)
ans =
Diagonal Matrix
 1  0  0  0
 0  1  0  0
 0  0  1  0
 0  0  0  1
```

Veja que nesses exemplos a função `eye()` gera matrizes *quadradas*, ou seja, os números de linhas e colunas da matriz obtida são iguais. O parâmetro recebido pela função corresponde ao número de linhas e colunas.

Para obtermos uma matriz identidade com números de linhas e colunas diferentes, fornecemos dois valores como parâmetros na função `eye()`. O primeiro parâmetro corresponde ao número de linhas, enquanto o segundo diz respeito ao número de colunas:

```

> eye(1,2)
ans =
Diagonal Matrix
 1  0
> eye(2,1)
ans =
Diagonal Matrix
 1
 0
> eye(2,3)
ans =
Diagonal Matrix
 1  0  0
 0  1  0
> eye(4,2)
ans =
Diagonal Matrix
 1  0
 0  1
 0  0
 0  0
> eye(2,6)
ans =
Diagonal Matrix
 1  0  0  0  0  0
 0  1  0  0  0  0

```

A função `diag()` , por sua vez, preenche a diagonal de uma matriz com os números que escolhermos. O argumento dessa função é um vetor, que vai ocupar a diagonal da matriz:

```

> diag([1 2])
ans =
Diagonal Matrix
 1  0
 0  2
> diag([1 2 3])
ans =
Diagonal Matrix
 1  0  0
 0  2  0
 0  0  3
> diag([1 2 3 4])
ans =
Diagonal Matrix
 1  0  0  0
 0  2  0  0

```

```
0   0   3   0  
0   0   0   4
```

NOTA

O argumento da matriz diagonal deve vir entre colchetes, como se estivéssemos definindo uma variável:

```
diag(1 2 3 4)  
parse error:  
  syntax error  
>> diag([1 2 3 4])
```

O Octave retornou um erro por não fornecermos corretamente um vetor como argumento na criação de uma matriz diagonal.

Outros exemplos de matrizes especiais já definidas no Octave são as funções `zeros()` e `ones()`. Como você já deve ter imaginado, elas criam matrizes apenas com zeros e uns:

```
> zeros(1)  
ans = 0  
> zeros(2)  
ans =  
  0  0  
  0  0  
> ones(1)  
ans = 1  
> ones(2)  
ans =  
  1  1  
  1  1
```

Há também a função `rand()`, que gera uma matriz aleatória cujos elementos estão entre zero e um:

```
> rand(1)  
ans = 0.32122  
> rand(2)  
ans =  
  0.71022  0.10996
```

```

0.83830 0.47751
> rand(3)
ans =
0.285251 0.496105 0.067233
0.952798 0.435623 0.233161
0.668566 0.909135 0.743966
> rand(4)
ans =
0.44568 0.14085 0.69094 0.76619
0.48349 0.80860 0.55819 0.27226
0.42233 0.76229 0.51461 0.45531
0.37441 0.69650 0.26674 0.55073

```

NOTA

Para obtermos diferentes números de linhas e colunas em matrizes de zeros, uns e aleatórias, fornecemos os dois valores como parâmetros nas funções `zeros()`, `ones()` e `rand()`, da mesma forma que para a função `eye()`.

Agora é a sua vez! Podemos criar matrizes que contenham elementos das funções matemáticas que aprendemos no capítulo anterior. Que tal testarmos essas funções? Alguns exemplos para aguçar sua curiosidade:

```

> M = [sind(90) 0; 0 cosd(90)]
> N = [exp(cosh(1)); exp(sqrt(2)); expacos(3))]

```

3.7 OPERAÇÕES COM MATRIZES E VETORES

Aprenderemos agora algumas operações com matrizes e vetores que vão ajudá-lo muito nas aulas de Álgebra Linear. Estudaremos o cálculo de determinantes, autovalores e autovetores, matrizes inversas etc. Mas antes, vejamos como ficam as operações básicas, vistas no capítulo anterior, para matrizes e vetores.

Soma e subtração

A soma/subtração entre dois vetores ou matrizes é feita diretamente. Cada elemento da primeira variável é somado ou subtraído do elemento correspondente na segunda variável.

```
> vetA = [1 2 3 4];
> vetB = [5 6 7 8];
> vetA+vetB
ans =
    6     8    10    12
> vetA-vetB
ans =
    -4    -4    -4    -4
> matA = [1 2 3; 4 5 6; 7 8 9];
> matB = [9 8 7; 6 5 4; 3 2 1];
> matA+matB
ans =
    10    10    10
    10    10    10
    10    10    10
> matA-matB
ans =
    -8    -6    -4
    -2     0     2
     4     6     8
```

NOTA

Fique atento apenas ao número de linhas e colunas das variáveis! Para que possamos somá-las ou subtraí-las, eles devem ser os mesmos:

```
A = [0.5 3 1];
B = [2 9 pi 6];
C = [5 3; 4 0];
A+B
error: operator +: nonconformant arguments
        (op1 is 1x3, op2 is 1x4)
A-C
error: operator -: nonconformant arguments
        (op1 is 1x3, op2 is 2x2)
```

Multiplicação entre variáveis e por um elemento

De acordo com a multiplicação matemática entre vetores ou matrizes, o número de colunas da primeira variável deve ser igual ao número de linhas da segunda variável. Nesse caso, podemos multiplicar as variáveis utilizando o operador de multiplicação (*).

A igualdade imposta para a multiplicação pode ser confirmada com a função `size()` :

```
> A = [2 3 4; 1 5 2];
> B = [1; 3; 5];
> size(A)(2) == size(B)(1)
ans = 1
```

Note que `size(A)(2)` nos dá o número de colunas de `A` , enquanto `size(B)(1)` retorna o número de linhas de `B` . Para verificar se elas são iguais, basta usar o *operador de igualdade*, dado por "dois iguais" (==). Se as variáveis têm o mesmo valor, o Octave retorna 1; caso contrário, a resposta do Octave é zero.

Veremos os operadores de relação com mais detalhes no capítulo *Operadores e estruturas para controle de fluxo*.

NOTA

Cuidado quando utilizar o operador de igualdade! Ele não deve ser confundido com o operador de atribuição (`=`), que atribuirá o valor da segunda variável à primeira:

```
vetA = [0.5 10 6];
vetB = [3 e 7 4];
vetA(2)
ans = 10
vetA(2) == vetB(2)
ans = 0
vetA(2) = vetB(2)
vetA =
    0.50000   2.71828   6.00000
vetA(2)
ans = 2.7183
```

Como visto no exemplo, `vetA(2)` foi definida como 10, mas quando o operador `=` foi usado, `vetA(2)` tornou-se igual a e !

A matriz resultante da multiplicação terá o número de linhas da primeira variável e o número de colunas da segunda variável, como podemos confirmar pela função `size()`:

```
> A = [2 3 4; 1 5 2];
> B = [1; 3; 5];
> A*B
ans =
    31
    26
> size(A)
ans =
    2    3
> size(B)
ans =
    3    1
> size(A*B)
ans =
```

2 1

Também podemos multiplicar um elemento por uma matriz utilizando o operador "ponto e asterisco" (`.*`). Com ele, é possível multiplicar um número por todos os elementos de uma matriz, ou obter o produto de duas variáveis de mesmo tamanho. Nesse caso, cada elemento da primeira variável é multiplicado pelo seu correspondente da segunda variável:

```
> a = 5;
> b = 3i
> A = [6 2; 3 1];
> B = [sin(45); cos(45); tan(45)];
> C = [1 factorial(3); 0 exp(2)];
> a.*A
ans =
    30    10
    15     5
> b.*A
ans =
    0 + 18i    0 +  6i
    0 +  9i    0 +  3i
> b.*B
ans =
    0.0000 + 2.5527i
    0.0000 + 1.5760i
    0.0000 + 4.8593i
> a.*B
ans =
    4.2545
    2.6266
    8.0989
> b.*C
ans =
    0.00000 + 3.00000i    0.00000 + 18.00000i
    0.00000 + 0.00000i    0.00000 + 22.16717i
```

NOTA

A ordem é importante na multiplicação entre matrizes: `matA*matB` geralmente é diferente de `matB*matA`. Entretanto, não há diferença na multiplicação entre elementos: `matA.*matB` é sempre igual a `matB.*matA`.

Podemos conferir a igualdade com a função `isequal()`, que compara duas variáveis:

```
A = [6 2; 3 1];
C = [1 factorial(3); 0 exp(2)];
isequal(A*C,C*A)
ans = 0
isequal(A.*C,C.*A)
ans = 1
```

Veja que `isequal()` retorna 1 quando as variáveis são iguais, e zero caso contrário.

Divisão por um elemento

De forma semelhante à multiplicação por um elemento, podemos dividir matrizes ou vetores por um elemento utilizando o operador "ponto e barra" (`./`). Com ele, dividimos todos os elementos da matriz por um número:

```
> a = sin(45)*i;
> b = cos(45);
> A = [5 0 2; 1 3 0];
> B = [3 1 0; 9 3 1];
> A./a
ans =
    0.00000 - 5.87611i    0.00000 + 0.00000i
    0.00000 - 2.35044i
    0.00000 - 1.17522i    0.00000 - 3.52566i
    0.00000 + 0.00000i
> A./b
```

```

ans =
  9.51797   0.00000   3.80719
  1.90359   5.71078   0.00000
> B./a
ans =
  0.00000 - 3.52566i   0.00000 - 1.17522i
  0.00000 + 0.00000i
  0.00000 - 10.57699i   0.00000 - 3.52566i
  0.00000 - 1.17522i
> B./b
ans =
  5.71078   1.90359   0.00000
  17.13235   5.71078   1.90359

> B./(a+b)
ans =
  1.57597 - 2.55271i   0.52532 - 0.85090i
  0.00000 + 0.00000i
  4.72790 - 7.65813i   1.57597 - 2.55271i
  0.52532 - 0.85090i
> A./(a+b)
ans =
  2.62661 - 4.25452i   0.00000 + 0.00000i
  1.05064 - 1.70181i
  0.52532 - 0.85090i   1.57597 - 2.55271i
  0.00000 + 0.00000i

```

Também podemos dividir duas matrizes de mesmo tamanho. Nesse caso, cada elemento da primeira matriz é dividido pelo seu correspondente da segunda matriz:

```

> A = [5 0 2; 1 3 0];
> B = [3 1 0; 9 3 1];
> A./B
ans =
  1.66667   0.00000      Inf
  0.11111   1.00000   0.00000
> B./A
ans =
  0.60000      Inf   0.00000
  9.00000   1.00000      Inf

```

NOTA

Da mesma forma que na multiplicação entre matrizes, `matA./matB` geralmente é diferente de `matB./matA`, como visto no último exemplo.

Observe que alguns elementos das matrizes resultantes são iguais a `Inf`; eles representam o *infinito*, que foi alcançado, nesses exemplos, pela divisão por zero.

Outra forma de gerar um número `Inf` é atribuir um número maior do que aqueles com que o Octave pode lidar. Esse fenômeno é chamado de *overflow*, ou estouro de memória. Reproduziremos o estouro de memória a seguir, gerando números muito grandes:

```
> 10^10
ans = 1.0000e+10
> 10^10^10
ans = 1.0000e+100
> 10^10^10^10
ans = Inf
```

Utilizando `./`, podemos dividir um número por todos os elementos de uma matriz, gerando outra matriz:

```
> a = sin(45)*i;
> b = cos(45);
> A = [5 0 2; 1 3 0];
> B = [3 1 0; 9 3 1];
> a./A
ans =
    0.00000 + 0.17018i      NaN +     Infi
    0.00000 + 0.42545i
    0.00000 + 0.85090i    0.00000 + 0.28363i
NaN +     Infi
> b./A
ans =
    0.10506      Inf    0.26266
    0.52532    0.17511      Inf
```

```

> (a+b)./A
ans =
    0.10506 + 0.17018i      Inf +     Infi
0.26266 + 0.42545i
    0.52532 + 0.85090i   0.17511 + 0.28363i
Inf +     Infi
> (a+b)./B
ans =
    0.175107 + 0.283635i   0.525322 + 0.850904i
Inf +     Infi
    0.058369 + 0.094545i   0.175107 + 0.283635i
0.525322 + 0.850904i

```

Note que, além de `Inf`, tivemos o elemento `NaN` nas matrizes resultantes. Este diz que o resultado não é um número (*not a number*).

Um `NaN` pode ser obtido pelas chamadas *formas indeterminadas* da matemática: divisão de zero por zero, multiplicação de zero por `Inf`, subtração de `Inf` e `Inf`, ou divisão de `Inf` por `Inf`.

```

> 0/0
warning: division by zero
ans = NaN
> 0*Inf
ans = NaN
> Inf-Inf
ans = NaN
> Inf/Inf
ans = NaN

```

Potenciação entre variáveis e por um elemento

Podemos efetuar a potenciação de uma matriz por um número de forma parecida com a da multiplicação e da divisão. O operador a ser utilizado para a potenciação é o "ponto e circunflexo" (`.^`). Nesse caso, todos os elementos da matriz são elevados ao número à direita do operador.

```

> a = cos(30)*i;
> b = cos(60);
> A = [1 2; 5 3; 2 1];

```

```

> B = [5 4; 3 1; 1 0];
> A.^2
ans =
    1     4
   25     9
    4     1
> A.^b
ans =
    1.00000  0.51677
    0.21592  0.35122
    0.51677  1.00000
> B.^4
ans =
   625    256
    81      1
    1      0
> B.^a
ans =
    0.96934 + 0.24572i  0.97722 + 0.21221i
    0.98568 + 0.16865i  1.00000 + 0.00000i
    1.00000 + 0.00000i      NaN +      NaNi

```

Com o operador `.^`, também podemos fazer a potenciação de um número por uma matriz, assim como a potenciação entre matrizes:

```

> a = cos(30)*i;
> b = cos(60);
> A = [1 2; 5 3; 2 1];
> B = [5 4; 3 1; 1 0];
> a.^B
ans =
    0.00000 + 0.00009i  0.00057 + 0.00000i
   -0.00000 - 0.00367i  0.00000 + 0.15425i
    0.00000 + 0.15425i  1.00000 + 0.00000i
> b.^A
ans =
   -0.95241   0.90709
   -0.78366  -0.86392
    0.90709  -0.95241
> A.^A
ans =
    1     4
   3125    27
    4     1
> A.^B
ans =

```

```

      1   16
    125   3
      2   1
> B.^A
ans =
      5   16
    243   1
      1   0

```

NOTA

Da mesma forma que na multiplicação entre matrizes e na divisão por um escalar, `matA.^matB` geralmente é diferente de `matB.^matA`.

Transposição de vetores e matrizes

Agora, veremos como transpor matrizes e vetores no Octave. O elemento transposto é obtido pela troca das linhas e colunas no elemento original. A transposição é obtida pela função `transpose()`:

```

> Areal = [1 2 3 4];
> transpose(Areal)
ans =
      1
      2
      3
      4

```

Nesse exemplo, o vetor `Areal` tem uma linha e quatro colunas; sua transposta passou a ter quatro linhas e uma coluna. Os elementos que pertenciam às colunas se tornaram linhas.

Podemos usar também o operador "aspas simples" ('):

```

> Breal = [1 2 3; 4 5 6; 7 8 9];
> Breal'
ans =

```

```

1   4   7
2   5   8
3   6   9

```

Contudo, no caso de matrizes ou vetores complexos, o operador `'` representa a *transposta do conjugado complexo* do vetor. Observe o exemplo a seguir, no qual tomamos `Acomplexo` como uma matriz que contém números complexos:

```

> Acomplexo = [i 2i 3i 4i];
> Acomplexo'
ans =
0 - 1i
0 - 2i
0 - 3i
0 - 4i

```

Repare que o sinal dos números foi invertido! É como se usássemos a função `conj()` e tomássemos a transposta do resultado. O conjugado complexo de um número real é igual a ele mesmo, por isso as operações transposta e transposta do conjugado complexo são equivalentes em números reais:

```

> Areal = [1 2 3 4];
> Aconjugado = conj(Areal)
ans =
1   2   3   4
> Aconjugado'
ans =
1
2
3
4

```

Note que o resultado é o mesmo. Logo, podemos usar "aspas simples" na transposição de matrizes e vetores com valores reais.

Para transpor números complexos, além da função `transpose()`, podemos utilizar o operador "ponto e aspa simples" (`.'`). Vamos transpor, então, a variável `Acomplexo`:

```

> Acomplexo = [i 2i 3i 4i];
> Acomplexo.'

```

```
ans =
 0 + 1i
 0 + 2i
 0 + 3i
 0 + 4i
```

Veja que as colunas se tornaram linhas, e o sinal dos vetores é o mesmo, alcançando o resultado esperado. Para confirmar as dimensões das variáveis, podemos utilizar a função `size()` :

```
> Areal = [1 2 3 4];
> size(Areal)
ans =
 1   4
> size(Areal')
ans =
 4   1
> Acomplexo = [i 2i 3i 4i];
> size(Acomplexo)
ans =
 1   4
> size(Acomplexo.')
ans =
 4   1
```

Álgebra linear no Octave

Para concluirmos este capítulo, veremos algumas funções úteis da Álgebra Linear, uma área da matemática que estuda objetos em uma, duas ou mais dimensões. Começaremos com o *traço* de uma matriz, que é a soma dos elementos de sua diagonal. Ele é dado pela função `trace()` .

```
> A = [1 2 3; 4 5 6; 7 8 9];
> trace(A)
ans = 15
> B = [1 2i 3; 4 5i 6; 7 8i 9];
> trace(B)
ans = 10 + 5i
```

O *determinante* associa uma matriz a um número. Ele é dado no Octave pela função `det()` .

```
> A = [sin(30) sin(45) sin(90); cos(30) cos(45) cos(90);
```

```

tan(30) tan(45) tan(90)]
A =
-0.98803   0.85090   0.89400
 0.15425   0.52532  -0.44807
-6.40533   1.61978  -1.99520
> det(A)
ans = 6.2540
> B = [sec(30) sec(45) sec(90); csc(30) csc(45) csc(90);
cot(30) cot(45) cot(90)]
B =
 6.48292   1.90359  -2.23178
-1.01211   1.17522   1.11857
-0.15612   0.61737  -0.50120
> det(B)
ans = -8.6086

```

NOTA

A matriz precisa ser quadrada para que possamos calcular seu determinante:

```

A = [1 2 3; 4 5 6]
A =
 1   2   3
 4   5   6
det(A)
error: det: argument must be a square matrix

```

Como o número de linhas da matriz que definimos não é igual ao número de colunas, o Octave retorna um erro.

Por sua vez, a função `inv()` calcula a inversa de uma matriz. O resultado da multiplicação de uma matriz `A` pela sua inversa, `inv(A)`, é a matriz identidade (`eye()`) de mesma ordem:

```

> A = [1 0 3; -4 -3 0; 7 0 2];
> inv(A)
ans =
-0.10526   0.00000   0.15789
 0.14035  -0.33333  -0.21053
 0.36842   0.00000  -0.05263

```

```
> A*inv(A)
ans =
1.00000  0.00000  0.00000
0.00000  1.00000  0.00000
0.00000  0.00000  1.00000
```

NOTA

A operação `A*inv(A)` retorna a matriz identidade, mas alguns erros de arredondamento podem levar a um resultado próximo da `eye()`.

A operação de inversão só é possível se o determinante da matriz desejada for diferente de zero; caso contrário, a matriz será chamada de *singular*, ou não inversível. O Octave descreve essa condição com um aviso:

```
> B = [3 1 0; -4 -3 0; 7 2 0];
> det(B)
ans = 0
> inv(B)
warning: inverse: matrix singular to machine precision, rcond = 0
ans =
Inf   Inf   Inf
Inf   Inf   Inf
Inf   Inf   Inf
```

Os autovalores de uma matriz são um conjunto de números que resumem as suas propriedades. Eles podem ser obtidos pela função `lambda = eig()`, e a variável `lambda` receberá seus valores:

```
> B = [3 1 0; -4 -3 0; 7 2 0];
> lambda1 = eig(B)
lambda1 =
0.00000
-2.23607
2.23607
> C = [1 0 3; -4 -3 0; 7 0 2];
> lambda2 = eig(C)
lambda2 =
```

```
-3.0000
-3.1098
6.1098
```

Veja que um dos autovalores da matriz B é igual a zero, o que indica que B é singular. Seu determinante também é igual a zero, como visto no exemplo anterior.

Da mesma forma, a função `eig()` retorna os autovetores da matriz quando usada na forma `[V, lambda] = eig(A)`. Nesse caso, a variável `V` contém os autovetores, e `lambda` é uma matriz cuja diagonal contém os autovalores.

```
> A = [3 0 -1; 10 -2 0; 3 1 3];
> [V1, lambda1] = eig(A)
V1 =
 0.03148 + 0.00000i  -0.02471 + 0.33439i  -0.02471 - 0.33439i
 -0.98537 + 0.00000i  0.19025 + 0.56835i  0.19025 - 0.56835i
  0.16748 + 0.00000i  0.72688 + 0.00000i  0.72688 - 0.00000i
lambda1 =
Diagonal Matrix
 -2.3195 + 0.0000i          0          0
          0  3.1598 + 2.1620i          0
          0          0  3.1598 - 2.1620i
> B = [3 -4 6; 7 0 9; 1 1 -3];
> [V2, lambda2] = eig(B)
V2 =
 0.06604 + 0.53965i  0.06604 - 0.53965i  -0.68893 + 0.00000i
  0.82511 + 0.00000i  0.82511 - 0.00000i  -0.27419 + 0.00000i
  0.15196 - 0.02270i  0.15196 + 0.02270i  0.67096 + 0.00000i
lambda2 =
Diagonal Matrix
 2.2177 + 4.3307i          0          0
          0  2.2177 - 4.3307i          0
          0          0  -4.4354 + 0.0000i
```

Quando a função `eig()` retorna os autovalores e autovetores, a relação `matA = autoV*lambda*inv(autoV)` é válida (salvo os erros de arredondamento):

```
> A = [3 0 -1; 10 -2 0; 3 1 3];
> [V1, lambda1] = eig(A);
> B = [3 -4 6; 7 0 9; 1 1 -3];
> [V2, lambda2] = eig(B);
> V1*lambda1*inv(V1)
```

```

ans =
 3.00000 + 0.00000i -0.00000 + 0.00000i -1.00000 + 0.00000i
 10.00000 + 0.00000i -2.00000 + 0.00000i -0.00000 + 0.00000i
 3.00000 + 0.00000i 1.00000 - 0.00000i 3.00000 + 0.00000i
> V2*lambda2*inv(V2)
ans =
Columns 1 and 2:
 3.0000e+00 - 2.3183e-19i -4.0000e+00 - 2.1203e-17i
 7.0000e+00 - 9.2265e-20i 2.9976e-15 - 8.4386e-18i
 1.0000e+00 + 2.2578e-19i 1.0000e+00 + 2.0650e-17i
Column 3:
 6.0000e+00 - 5.5124e-18i
 9.0000e+00 - 2.1938e-18i
-3.0000e+00 + 5.3686e-18i

```

NOTA

Repare que os elementos resultantes em `V2*lambda2*inv(V2)` vieram acompanhados de e mais (ou menos) um número. Essa é a *notação científica*, que o Octave utiliza quando os números são muito grandes (ou muito pequenos).

Por exemplo, `e-18i` indica um número complexo muito próximo de zero. Verifique que quando desconsideramos esses números extremamente pequenos, resultantes dos erros de arredondamento nos cálculos do Octave, `B` é igual a `V2*lambda2*inv(V2)`.

O FAQ do Octave diz que "assim como a morte e os impostos, erros de arredondamento são um fato da vida". Para uma explicação mais completa sobre erros computacionais e alternativas para contornar o problema, visite http://wiki.octave.org/FAQ#Why_is_this_floating_point_computation_wrong.3F.

Uma vez que trabalhamos apenas com a divisão de uma matriz

por um escalar, encerraremos o capítulo discutindo divisões entre matrizes. Essa divisão pode ser feita de duas formas. A primeira é a *divisão à direita*, dada por A/B . Ela calcula uma matriz X , tal que $X*B$ é igual à matriz A .

```
> A = [3 0 -1; 5 -2 0; 3 1 3];
> B = [3 4 2; 1 0 5; 1 1 3];
> X = A/B
X =
    3.2000    6.2000   -12.8000
    5.8000   12.8000   -25.2000
    2.0000    4.0000   -7.0000
> X*B
ans =
    3.00000    0.00000   -1.00000
    5.00000   -2.00000    0.00000
    3.00000    1.00000    3.00000
```

A outra divisão é a *divisão à esquerda*, $A\Bslash B$. Por sua vez, essa forma de dividir calcula X tal que $A*X$ é igual a B .

```
> A = [3 0 -1; -4 -2 0; 3 1 -3];
> B = [3 4 2; -1 0 5; 1 1 -3];
> X = A\B
X =
    1.06250    1.37500    0.81250
   -1.62500   -2.75000   -4.12500
    0.18750    0.12500    0.43750
> A*X
ans =
    3    4    2
   -1    0    5
    1    1   -3
```

Repare que a divisão à esquerda nada mais é do que a resolução de um *sistema linear*. Temos uma matriz A e um vetor b , e queremos encontrar o X tal que $A*X = b$:

```
> A = [1 0 5 3; -4 0 1 -3; 3 1 2 4; 5 0 0 1];
> b = [1; 3; -1; 0];
> X = A\b
X =
    0.20290
    0.91304
    0.76812
```

```
-1.01449
> A*X
ans =
1.00000
3.00000
-1.00000
0.00000
```

3.8 RESUMINDO

Neste capítulo, demos maior atenção às strings, aos vetores e às matrizes. Aprendemos a criar, comparar e manipular strings, além de localizar padrões em seu conteúdo. Vimos também como gerar alguns vetores e matrizes especiais, e como realizar operações entre eles. Finalmente, ampliamos nossa caixa de ferramentas com funções da Álgebra Linear.

No próximo capítulo, usaremos todas as funções e técnicas que aprendemos até aqui para desenhar gráficos de duas e três dimensões, que vão nos permitir visualizar fenômenos e analisar funções matemáticas de uma maneira muito mais interessante. Nos vemos lá!

CAPÍTULO 4

PRODUZINDO GRÁFICOS NO OCTAVE

Em vez de apresentar apenas listas intermináveis de números, podemos utilizar gráficos para enriquecer nosso trabalho. Eles expressam ideias de forma eficaz, mostrando particularidades dos elementos que não seriam tão aparentes em uma tabela.

Neste capítulo, conheceremos a arte de *plotar* (ou "imprimir na tela"). Estudaremos gráficos de diferentes tipos, que podem ser obtidos facilmente no Octave. As representações gráficas disponíveis nos proporcionarão várias formas de esboçar dados. Começaremos pelas funções multiuso para gráficos de duas e três dimensões.

4.1 AS FUNÇÕES ESSENCIAIS PARA GRÁFICOS DE DUAS E TRÊS DIMENSÕES

A função elementar para gráficos bidimensionais, `plot()`, é versátil e adaptável para diferentes situações. Sua forma mais simples é `plot(f)`, em que `f` é a função desejada.

Pensando no plano cartesiano, o eixo `X` mostra o intervalo que começa em zero e termina com o número de elementos de `f`, ou seja, `numel(f)`. O eixo `Y` representa os valores da função `f` correspondentes a cada valor presente no eixo `X`.

NOTA

Para lembrar o que é e como se usa o plano cartesiano, confira o *Matemática Didática* (<http://www.matematicadidatica.com.br/PlanoCartesiano.aspx>).

Para se aprofundar no assunto, há conteúdo sobre no site *Só Matemática* (<http://www.somatematica.com.br/fundam/paresord.php>) e também na página da *Wikipédia* dedicada ao tema (http://pt.wikipedia.org/wiki/Sistema_de_coordenadas_cartesiano).

No capítulo *Primeiros passos*, utilizamos valores únicos nas funções. Por exemplo:

```
> x = pi/2;  
> sin(x)  
ans = 1
```

Se usarmos o `x` que acabamos de definir, o gráfico de `sin(x)` apresentará um único ponto. Verifique: o comando a ser utilizado é `plot(sin(x))`.

Para gerar um gráfico razoável, precisamos de um número maior de elementos. Podemos selecionar uma variável `x` em um intervalo e calcular o seno de todos os valores de `x`. Para isso, criaremos um vetor por incremento (visto no capítulo anterior) e aplicaremos a função seno em todos os seus pontos:

```
> x = 0:1:10;  
> y = sin(x);
```

Podemos, então, usar o comando `plot(y)` para verificar o

gráfico do seno nos pontos x . Entretanto, como temos os valores de x e y , utilizaremos a função na forma `plot(x,y)`. Assim, o Octave entende que o eixo X será ocupado pela variável x , e o eixo Y vai receber os valores de y .

Podemos também poupar a variável y e plotar o gráfico diretamente, informando `sin(x)` no lugar de y :

```
> x = 0:1:10;  
> plot(x,sin(x));
```

O Octave apresentará a figura a seguir. O gráfico não parece muito com um gráfico usual da função seno; ele é formado por retas e possui pontas, tendo um aspecto serrilhado. Verifique com o comando `numel(x)` que a variável x possui apenas 11 elementos. O gráfico é obtido da função seno calculada nesses pontos, por meio de `sin(x)`.

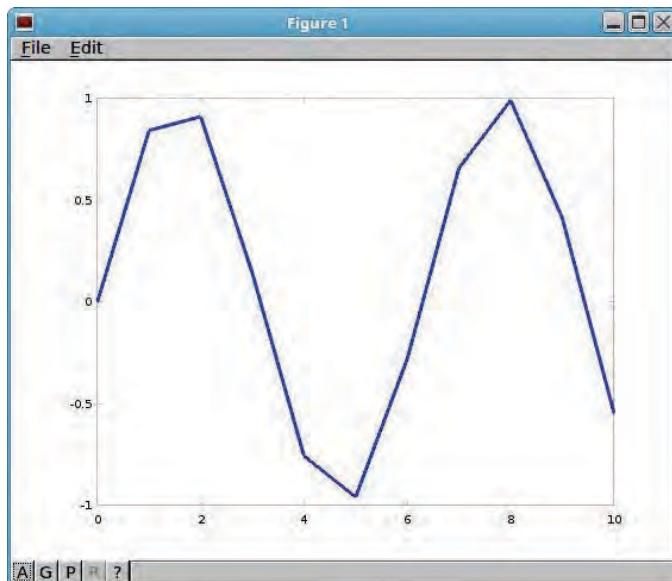


Figura 4.1: Gráfico da função `sin()` em duas dimensões. Perceba que a amostragem de $\sin(x)$ é pequena (11 amostras); logo, o gráfico aparece serrilhado

A função `plot()` liga os pontos do gráfico com retas. Como a quantidade de elementos na variável do gráfico é pequena, dizemos que nosso gráfico está *subamostrado*.

Precisamos de mais amostras para produzir um gráfico com melhor aparência. As amostras, nesse exemplo, são os valores de $\sin(x)$. Para obter mais deles, fornecemos mais valores de x :

```
> x = 0:0.1:10;
> numel(x)
ans = 101
> plot(x,sin(x));
```

O gráfico resultante é dado na figura:

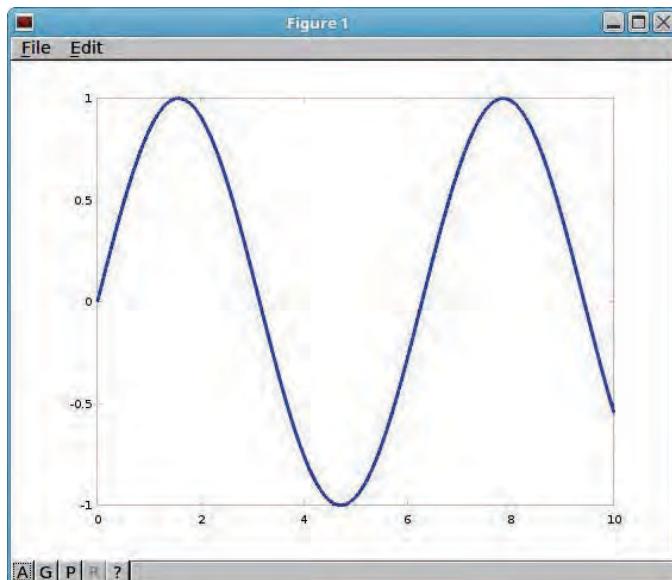


Figura 4.2: Gráfico da função $\sin()$. Como aumentamos a amostragem de $\sin(x)$ (101 amostras), o gráfico resultante possui melhor aparência quando comparado ao anterior

Por sua vez, `plot3()` é a função elementar para gráficos tridimensionais. Sua forma mais simples é `plot3(x, y, z)`.

Novamente recorrendo ao plano cartesiano, as variáveis x , y

e z representam os eixos X (comprimento), Y (largura) e Z (altura), respectivamente.

Em `plot3()`, os elementos de um eixo qualquer podem depender dos valores dos outros eixos, da mesma forma que o eixo Y depende da variável do eixo X em `plot()`. Para exemplificar, criaremos uma variável t em um intervalo, e calcularemos o seno e o cosseno para todos os valores de t . Como em `plot()`, a variável t é um vetor por incremento. Nesse exemplo, t vai de -3π a 3π com um espaço de 0.1 entre cada elemento:

```
> t = -3*pi:0.1:3*pi;
> sint = sin(t);
> cost = cos(t);
> plot3(t,sint,cost);
```

Como resultado, temos a figura a seguir

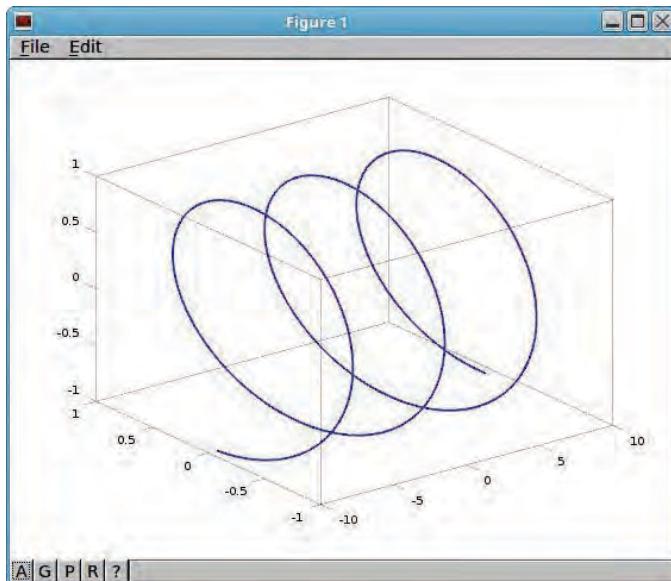


Figura 4.3: Gráfico tridimensional obtido por meio da função `plot3()`. Ele é baseado nas funções `sin(t)` e `cos(t)`, em que t é um vetor por incremento que começa em -3π e termina em 3π , de 0.1 em 0.1

Repare que o gráfico gerado é uma curva no espaço tridimensional; `plot3()` gera curvas em três dimensões, sendo correspondente a `plot()`, que gera curvas em duas dimensões.

Vamos praticar! Experimente trocar as variáveis de eixo: `plot3(sint, cost, t)`, `plot3(cost, t, sint)` etc. Verifique as diferenças entre o gráfico original e as novas curvas.

Vimos até aqui a forma padrão dos gráficos gerados pelas funções `plot()` e `plot3()`. Contudo, isso não significa que seus gráficos serão sempre uma reta azul. Essas funções possuem vários argumentos para gerar gráficos de diferentes formas, cores e tamanhos. Vamos a eles.

Modificando a aparência dos seus gráficos

Seus gráficos podem ser personalizados de acordo com as suas preferências. Podemos, por exemplo, alterar o traçado da linha, as cores, adicionar título e legenda.

NOTA

A maioria dos exemplos dessa seção apresenta apenas gráficos de duas dimensões. Contudo, as funções e os argumentos que veremos podem ser utilizados em todos os tipos de gráficos dados neste capítulo.

Iniciaremos pelas opções referentes ao traçado das linhas. Elas podem ser:

- **Sólidas:** o padrão, dado pelo argumento "traço" (`'-'`);
- **Tracejadas:** argumento "dois traços" (`'--'`);
- **Pontilhadas:** "dois pontos" (`'::'`);

- **Traço-pontilhadas:** "traço-ponto" (`'-.'`).

A forma da linha apresentada no gráfico é modificada colocando o argumento desejado na frente da função a ser desenhada em `plot()`. Por exemplo, plotaremos um gráfico traço-pontilhado da função logaritmo, com a variável `x` contendo 101 elementos entre 0 e 10:

```
> x = 0:0.1:10;  
> plot(x,log(x),'-.');
```

O resultado é a figura adiante. Veja como o traço é diferente das figuras anteriores.

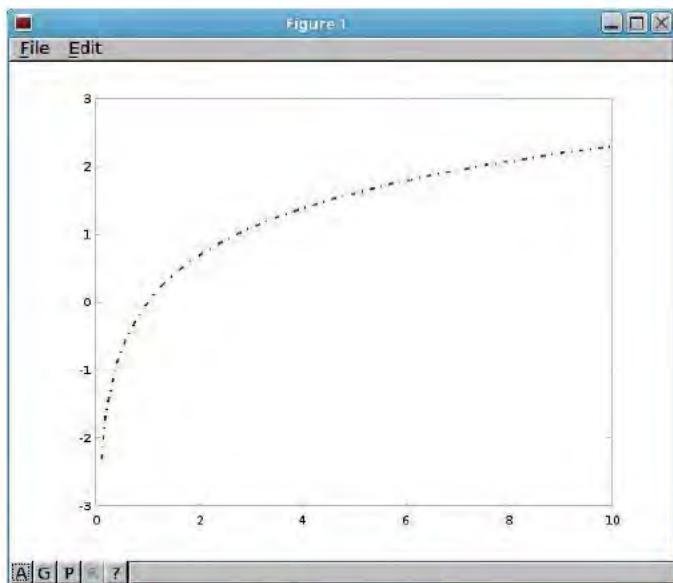


Figura 4.4: Gráfico traço-pontilhado da função `log(x)`, com `x` contendo 101 elementos

Que tal modificar nossos gráficos mais radicalmente? Além de plotar linhas, podemos usar símbolos! Para usar um deles, proceda da mesma forma que para alterar o traçado da linha: digite o argumento correspondente no comando `plot()`, na frente da função a ser desenhada.

As opções disponíveis para os símbolos são:

- **Mira** ('+');
- **Círculo** ('o');
- **Estrela** ('*');
- **Ponto** ('.');
- **Cruz** ('x');
- **Quadrado** ('s');
- **Diamante** ('d');
- **Triângulos voltados para cima, baixo, direita e esquerda** (respectivamente, '^', 'v', '>' e '<');
- **Pentagramas** ('p');
- **Hexagramas** ('h').

Ufa!

O uso dos símbolos é exemplificado com um gráfico da função cosseno, em que a variável x contém 200 elementos entre -2π e 2π . O símbolo utilizado será o círculo:

```
> x = linspace(-2*pi,2*pi,200);
> plot(x,cos(x), 'o');
```

O resultado é a figura seguinte. Veja o destaque que os círculos dão ao gráfico!

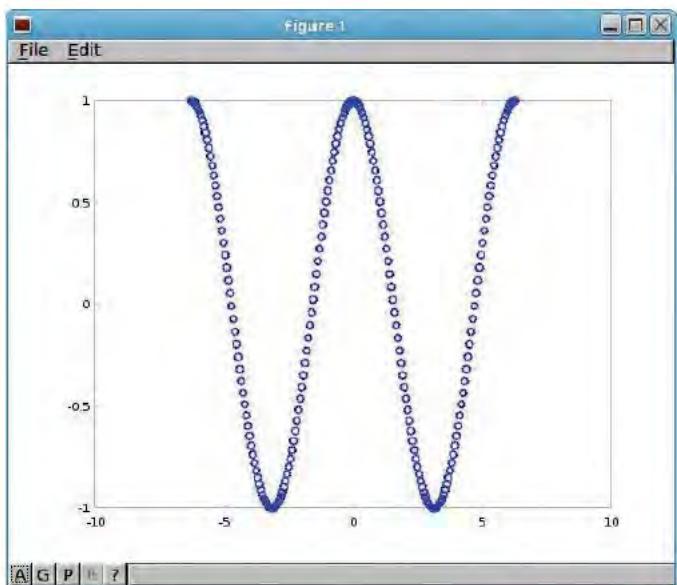


Figura 4.5: Gráfico da função $\cos()$, cujos pontos são representados por círculos, obtidos a partir do argumento 'o' na função `plot()`. Os símbolos dão maior ênfase às áreas do gráfico

Vejamos agora como alterar a cor da linha (ou símbolo) do gráfico. As opções são:

- **azul** (é o padrão, e seu argumento é 'b');
- **preto** ('k');
- **vermelho** ('r');
- **verde** ('g');
- **amarelo** ('y');
- **magenta** ('m');
- **ciano** ('c');
- **branco** ('w').

Assim como para alterar o traço/símbolo do gráfico, o argumento é informado depois da função a ser desenhada.

Por exemplo, vamos plotar um gráfico vermelho da função exponencial `exp()` com `x` contendo 100 elementos entre zero e

```
2*pi :  
  
> x = linspace(0,2*pi,100);  
> plot(x,exp(x), 'r');
```

O resultado é o da figura a seguir. Foi fácil alterar a cor, não acha?

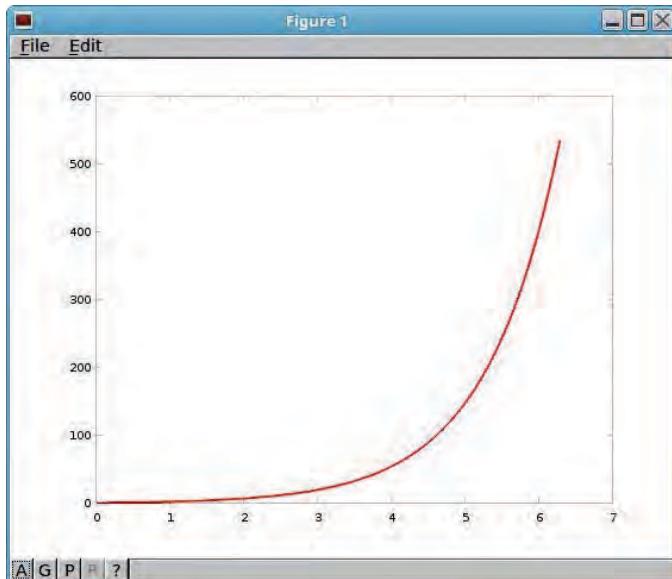


Figura 4.6: Gráfico vermelho da função $\exp()$, de zero a 2π . O argumento utilizado na função $\text{plot}()$ é 'r'

Quando desejamos alterar o traço e a cor em conjunto, podemos colocar os dois argumentos dentro das mesmas aspas. Para ilustrar, criaremos um gráfico verde com o símbolo "diamante" no lugar do traço. Usaremos a função $\sin(\cos(x))$ com a variável x contendo 100 elementos entre $-\pi$ e π :

```
> x = linspace(-pi,pi,100);  
> plot(x,sin(cos(x)), 'dg');
```

Veja que agrupamos nas mesmas aspas os argumentos 'd' e 'g', do símbolo "diamante" e da cor verde, e o Octave entendeu o

que queríamos. O resultado é dado na figura:

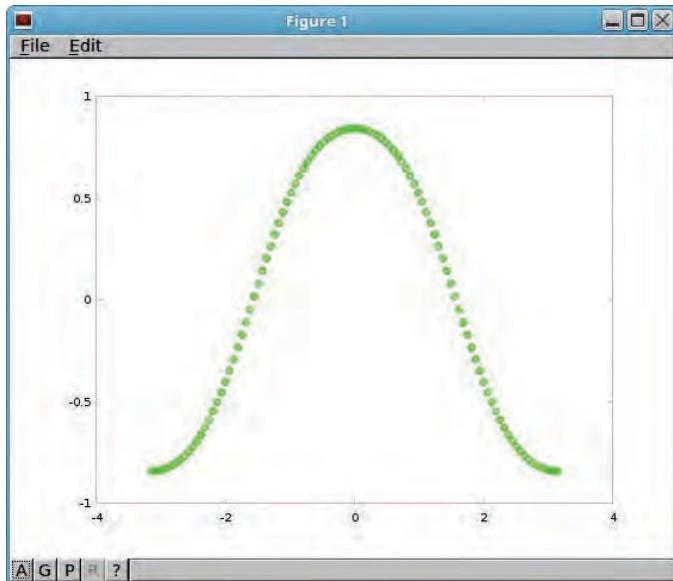


Figura 4.7: Gráfico verde com diamantes da função $\sin(\cos())$. O argumento usado é 'dg'

Se a intenção fosse gerar um gráfico traço-pontilhado magenta, por exemplo, o argumento utilizado seria '`- .m`'. Experimente!

NOTA

Para colorir os símbolos, use o argumento '`markerfacecolor`' seguido da cor que deseja. Definimos a cor da mesma forma que para as linhas; por exemplo, o argumento '`markerfacecolor`', '`r`' assume que os símbolos contidos no gráfico deverão ser preenchidos com a cor vermelha.

Podemos também aumentar ou diminuir a espessura da linha

utilizada no gráfico. Para isso, usamos o argumento 'linewidth' seguido de um número.

NOTA

Os gráficos obtidos nesse livro utilizaram os valores 2 ou 3 para a espessura das linhas, para facilitar a visualização das figuras.

Vejamos um exemplo com o argumento 'linewidth' igual a 10. A função utilizada é a tangente, e a variável `x` contém 100 elementos entre 0 e 2π :

```
> x = linspace(0,2*pi,100);
> plot(x,tan(x), 'linewidth',10);
```

O resultado é a figura adiante. O traço é bem maior que o dos outros gráficos obtidos até aqui.

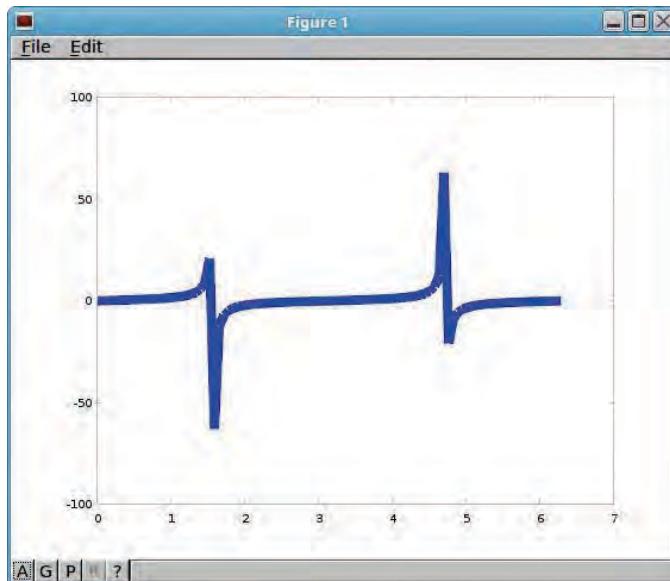


Figura 4.8: Gráfico exagerado da função `tan()`, criado por meio do argumento 'linewidth', 10.
Veja como o traço é mais espesso do que o dos gráficos anteriores

NOTA

Enquanto o argumento 'linewidth' diz respeito à espessura do traço, podemos usar o argumento 'markersize' para aumentar ou diminuir o tamanho do símbolo empregado no gráfico.

Veja como o gráfico anterior continha uma grande área em branco. Podemos ajustar a área apresentada por meio da função `axis(vet)`.

O vetor `vet`, argumento da função `axis()`, deve conter quatro números: o primeiro e o último pontos do eixo X e o primeiro e o último pontos do eixo Y.

Como exemplo, usaremos novamente o código anterior, com 'linewidth' igual a 2. O eixo X começa em zero e termina em 7, enquanto o eixo Y começa em -100 e termina em 100. Definiremos o eixo Y de -10 a 10, e o eixo X de zero a 2π . Logo, o vetor argumento da função `axis()` será `[0 2*pi -10 10]`:

```
> x = linspace(0,2*pi,100);
> plot(x,tan(x),'LineWidth',2);
> axis([0 2*pi -10 10]);
```

A figura seguinte é o resultado desses comandos. Repare como o gráfico está melhor distribuído.

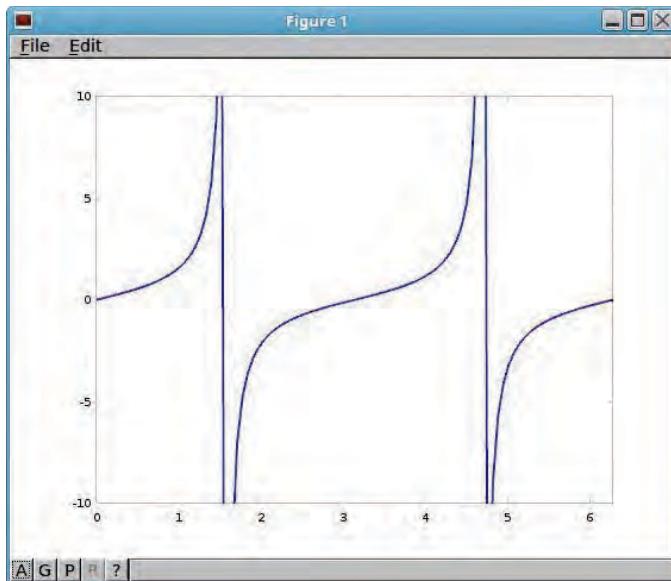


Figura 4.9: Gráfico da função $\tan()$ com os eixos ajustados. Compare com o gráfico anterior

NOTA

A ampliação do gráfico também pode ser ajustada (embora com menos precisão do que utilizando a função `axis()`) por meio da roda do *mouse*; rode para frente e para trás e o gráfico será ampliado e reduzido, respectivamente. Para ampliar uma região específica, selecione-a com o botão direito do mouse.

O gráfico pode ter ainda uma malha de fundo. Após plotar o gráfico, os comandos `grid` ou `grid on` adicionam uma malha de maior espessura. Utilizando o último exemplo:

```
> x = linspace(0,2*pi,100);
> plot(x,tan(x),'linewidth',2);
> axis([0 2*pi -10 10]);
> grid on
```

Verifique o resultado plotando o gráfico. Para retirar a malha, use `grid off`.

Podemos adicionar também uma malha de menor espessura usando os comandos `grid minor` ou `grid minor on`. Para retirá-la, o comando `grid minor off` é suficiente. Teste as malhas nos exemplos anteriores.

NOTA

A malha também pode ser inserida ou retirada do gráfico pressionando `G` no teclado, ou por meio do botão `G`, localizado no canto inferior esquerdo da janela do gráfico.

Agora, adicionaremos títulos aos nossos gráficos. Não poderia ser mais simples: a função necessária é a `title(strnome)`. O argumento `strnome` é dado por uma string, que será o título do gráfico:

```
> x = -10:0.01:10;
> plot(x,exp(cos(x.^2)),'^c','markerfacecolor','b');
> title('Gráfico de exp(cos(x^2)).')
```

O resultado é exibido na figura adiante. Note que o gráfico gerado apresenta triângulos com o contorno na cor ciano (argumento `'^c'`), preenchidos pela cor azul (`'markerfacecolor','b'`).

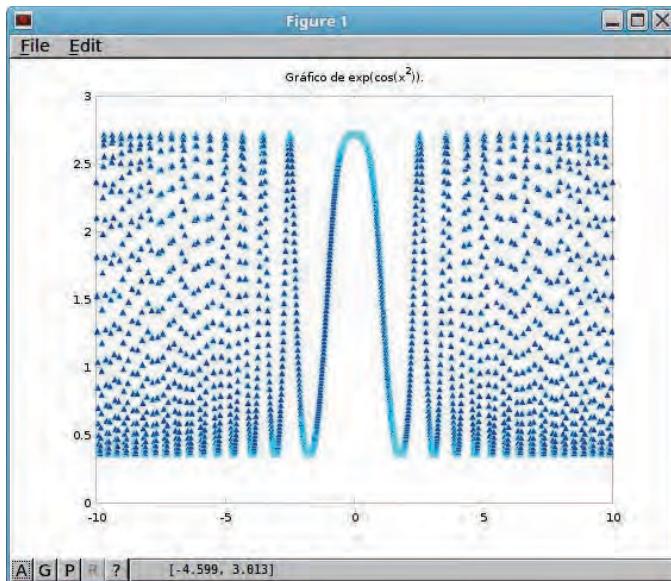


Figura 4.10: Gráfico de uma função com título obtido pela função title()

Também é possível adicionar rótulos nos eixos X (com a função xlabel(strnome)) e Y (função ylabel(strnome)). A string strnome contém o texto que aparecerá nos eixos, como no argumento da função title()

```
> x = 0:0.01:10;
> plot(x,cos(log(x./16)), 'pk', 'markersize', 10, ...
    'markerfacecolor', 'y');
> title('Gráfico de cos(log(x/16)).')
> xlabel('Valores de X')
> ylabel('Valores de cos(log(x/16))')
```

As reticências no código são o *marcador de continuação*. Elas indicam que a instrução continua na próxima linha, e são bastante úteis quando precisamos digitar grandes linhas de código.

A figura a seguir mostra o gráfico gerado.

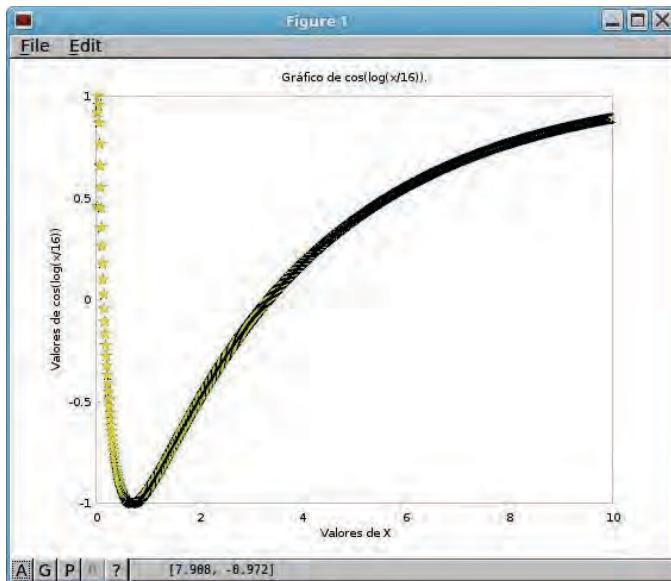


Figura 4.11: Gráfico de uma função bidimensional com rótulos em seus eixos, obtidos pelas funções xlabel() e ylabel()

Nesse gráfico, usamos pentagramas com o contorno em preto (estrelas, se preferir; o argumento é 'pk'), preenchidos pela cor amarela ('markerfacecolor', 'y'). O tamanho de cada símbolo é 10 ('markersize', 10). Teste com outros tamanhos!

Para gráficos tridimensionais, também existe a função zlabel(strnome), que adiciona um rótulo no eixo Z. Veja no exemplo a seguir:

```
> k = -2*pi:0.1:2*pi;
> plot3(k,exp(sin(k)),exp(cos(k)), 'dr', 'markerfacecolor', 'r')
> xlabel('Valores de k')
> ylabel('Valores de exp(sin(k))')
> zlabel('Valores de exp(cos(k))')
> grid on
```

O gráfico resultante, que é criado com diamantes preenchidos com a cor vermelha e exibe uma malha, aparece na figura:

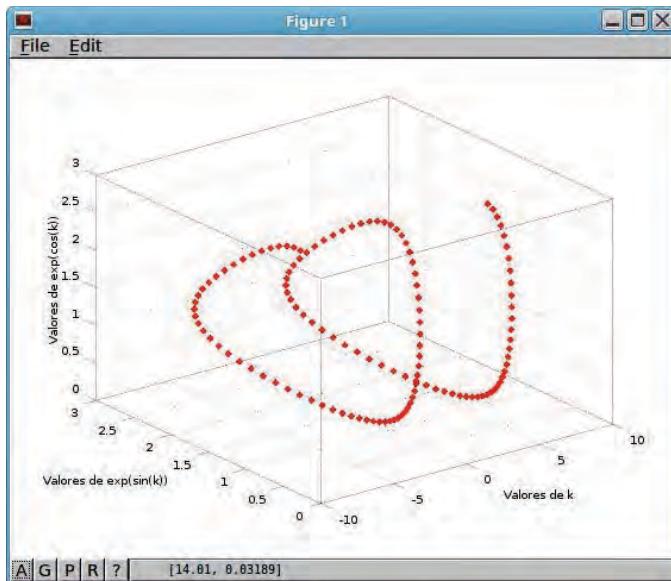


Figura 4.12: Gráfico de uma função tridimensional com rótulos em seus eixos, obtidos pelas funções xlabel(), ylabel() e zlabel()

Por fim, podemos adicionar legendas aos nossos gráficos por meio da função `legend(strnome)`. Uma legenda relacionará o traço (ou símbolo) com o nome de cada função, que pode ser escolhido por meio da string `strnome`. Isso torna o gráfico mais legível:

```
> x = 0:0.05:10;
> plot(x,sin(log(x)), 'hm', 'markersize', 8, 'markerfacecolor', 'm')
> legend('sin(log(x))')
```

O gráfico resultante é exibido na figura seguinte. Ele apresenta uma string com o nome da função que plotamos (`'sin(log(x))'`).

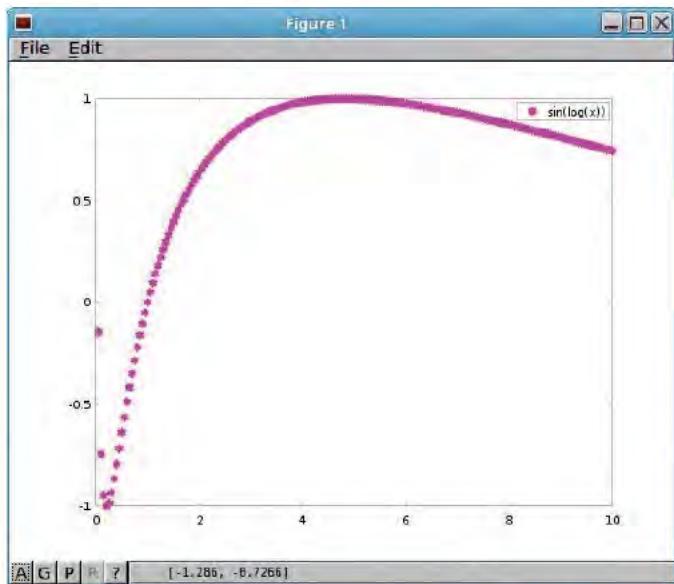


Figura 4.13: Gráfico com legenda adicionada pela função `legend()`

A legenda dessa figura ficou muito próxima do gráfico? Não tem problema. Podemos escolher o lugar em que ela aparecerá por meio do argumento '`location`'. As opções são:

- **À direita, em cima** (é o padrão; seu argumento é '`northeast`');
- **No centro, em cima** (`'north'`);
- **À esquerda, em cima** (`'northwest'`);
- **À direita, no meio** (`'east'`);
- **À esquerda, no meio** (`'west'`);
- **À direita, embaixo** (`'southeast'`);
- **No centro, embaixo** (`'south'`);
- **À esquerda, abaixo** (`'southwest'`).

Experimente! Utilize o exemplo anterior.

A função `legend()` permite ainda mais personalização. Para retirar o contorno da legenda, por exemplo, digite

`legend('boxoff')` após gerá-la. Da mesma forma, para posicionar o texto à esquerda do símbolo na legenda, digite `legend('left')`.

Utilizando o exemplo anterior, podemos então gerar um gráfico com legenda sem contorno e com texto à esquerda do símbolo, desta forma:

```
> x = 0:0.05:10;
> plot(x,sin(log(x)), 'hm', 'markersize', 8, 'markerfacecolor', 'm')
> legend('sin(log(x))')
> legend('boxoff')
> legend('left')
```

Plote esse gráfico no Octave e perceba a diferença em relação à legenda anterior. Podemos voltar aos valores-padrão dessas opções, como você deve ter imaginado, utilizando `legend('boxon')` e `legend('right')`.

NOTA

Os comandos apresentados para personalizar o gráfico (`axis()`, `grid`, `title()` etc.) são executados sempre depois da função `plot()`. Para que os resultados sejam os esperados, não podemos fechar a janela do gráfico enquanto trabalhamos; caso contrário, o Octave desconsidera o último comando `plot()` e teremos como resultado uma janela em branco!

4.2 PLOTANDO VÁRIOS GRÁFICOS EM UMA JANELA

Agora que já sabemos como configurar nossos gráficos, veremos como gerar vários na mesma janela gráfica. Esse recurso é

interessante para comparação visual de duas funções ou mais.

Basicamente, existem duas formas para plotar vários gráficos na mesma janela: podemos informar as funções desejadas como argumento em `plot()` ou `plot3()`, ou utilizar o comando `hold`, que informa à janela gráfica que ela receberá mais de um gráfico.

Informando vários gráficos como argumento na mesma função

Para plotar mais de um gráfico utilizando apenas `plot()` ou `plot3()`, devemos informar as funções na seguinte ordem:

1. **Primeiro gráfico:** variável do eixo X, variável do eixo Y, opções (cor, tamanho, símbolo etc.);
2. **Segundo gráfico:** variável do eixo X, variável do eixo Y, opções;
3. **Terceiro gráfico:** variável do eixo X etc.
4. Assim por diante, para o quarto gráfico e os seguintes.

Vejamos um exemplo com as funções seno e cosseno compartilhando a mesma figura, por meio da função `plot()`:

```
> k = -2*pi:0.1:2*pi;
> plot(k,sin(k), 'linewidth',8,k,cos(k), 'linewidth',4)
> legend('sin(k)', 'cos(k)')
> legend('boxoff')
```

Os gráficos resultantes são dados na figura:

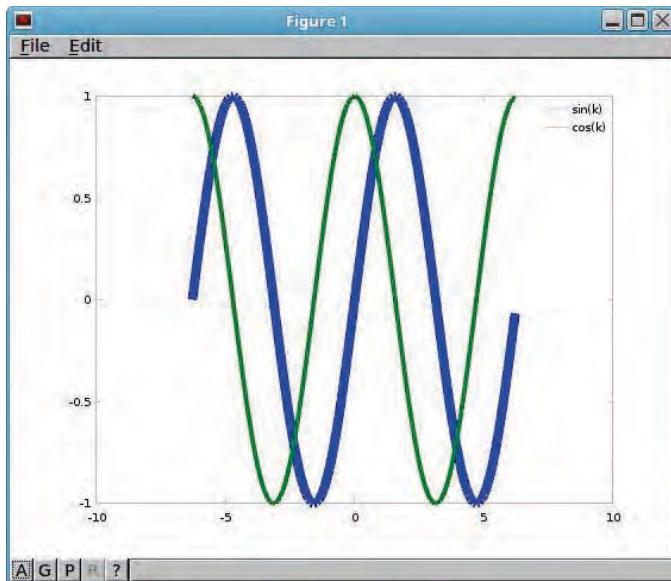


Figura 4.14: Gráficos das funções seno e cosseno compartilhando a mesma figura, obtidos pela função `plot()`, que recebeu as duas funções com argumentos diferentes

Note a ordem dos gráficos. A opção '`linewidth', 8`' se aplica apenas a `sin(k)`; a espessura de `cos(k)` é 4, como informamos em suas opções. O próprio Octave fez a diferenciação dos dois gráficos, utilizando diferentes cores sem precisarmos atribuí-las.

Veja que atribuímos a legenda aos gráficos usando duas strings separadas por vírgula: `legend('sin(k)', 'cos(k)')`. O processo se repete para três gráficos ou mais: `legend('Gráfico 1', 'Gráfico 2', 'Gráfico 3', ...)`.

No caso de gráficos em três dimensões, a ordem é a seguinte:

1. **Primeiro gráfico:** variável do eixo X, variável do eixo Y, variável do eixo Z, opções (cor, tamanho, símbolo etc.);
2. **Segundo gráfico:** variável do eixo X, variável do eixo Y, variável do eixo Z, opções;
3. **Terceiro gráfico:** variável do eixo X etc.

4. Assim por diante, para os gráficos seguintes.

O exemplo a seguir mostra duas curvas tridimensionais compartilhando a mesma figura por meio da função `plot3()`:

```
> k = -2*pi:0.1:2*pi;
> plot3(k,sin(k),cos(k),'linewidth',6, ...
k,cos(k),sin(k),'dr','linewidth',4)
```

Nesse exemplo, o primeiro gráfico é dado por k no eixo X, $\sin(k)$ no eixo Y e $\cos(k)$ no eixo Z, e a espessura de sua linha é 6. Por sua vez, o segundo gráfico é dado também por k no eixo X, mas por $\cos(k)$ no eixo Y e $\sin(k)$ no eixo Z. Esse gráfico é formado por diamantes vermelhos, e a espessura é 4. O resultado pode ser visto na figura:

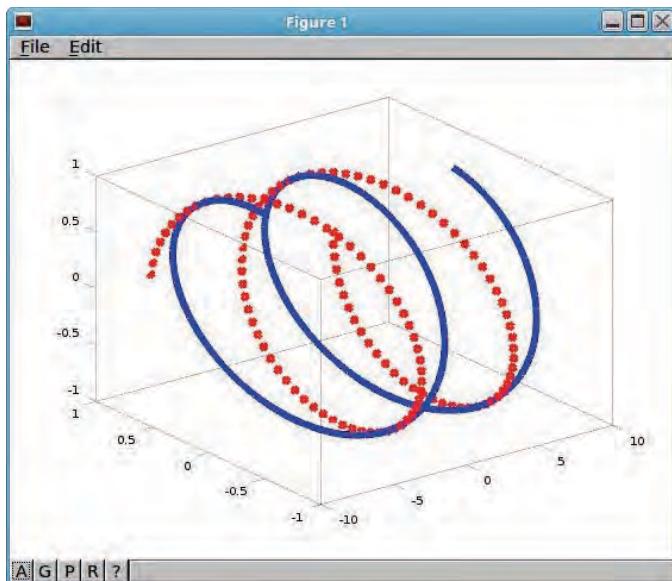


Figura 4.15: Gráficos de duas curvas tridimensionais compartilhando a mesma figura, obtidos pela função `plot3()`, que recebeu as duas funções com argumentos diferentes

Mantendo os gráficos na mesma janela gráfica

Digitar todas as curvas na mesma função `plot()` pode ser

trabalhoso (e não tão comprehensível) quando temos um número maior de gráficos a fazer. Nesses casos, podemos usar o comando `hold`, que conserva os gráficos já inseridos e possibilita a inserção de outros na mesma janela, até que o comando `hold off` seja digitado.

A figura anterior também pode ser obtida por meio de `hold`:

```
> k = -2*pi:0.1:2*pi;
> plot(k,sin(k), 'linewidth',8)
> hold all
> plot(k,cos(k), 'linewidth',4)
> legend('sin(k)', 'cos(k)')
> legend('boxoff')
> hold off
```

Se usarmos `hold` ou `hold on`, e não alterarmos as cores dos gráficos utilizando argumentos, a cor dos próximos será igual a do primeiro. Em vez disso, utilizamos o comando `hold all`. Ele retém as informações do gráfico anterior, assegurando que os próximos gráficos possuam cores diferentes.

O exemplo a seguir cria cinco diferentes gráficos utilizando `hold all`:

```
> x = 0:0.1:4*pi;
> plot(x,sin(x), 'linewidth',3)
> hold all
> plot(x,cos(x), '--', 'linewidth',3)
> plot(x,tan(x), '-.', 'linewidth',3)
> plot(x,exp(x), ':', 'linewidth',3)
> plot(x,log(x), '*', 'linewidth',3)
> axis([0,10,-2,4])
> legend('Seno', 'Cosseno', 'Tangente', 'Exponencial', 'Logaritmo')
> legend('boxoff')
> hold off
```

Fizemos várias personalizações, como utilizar diferentes tipos de linhas e símbolos. Também inserimos uma legenda e ajustamos os eixos. O resultado é a figura:

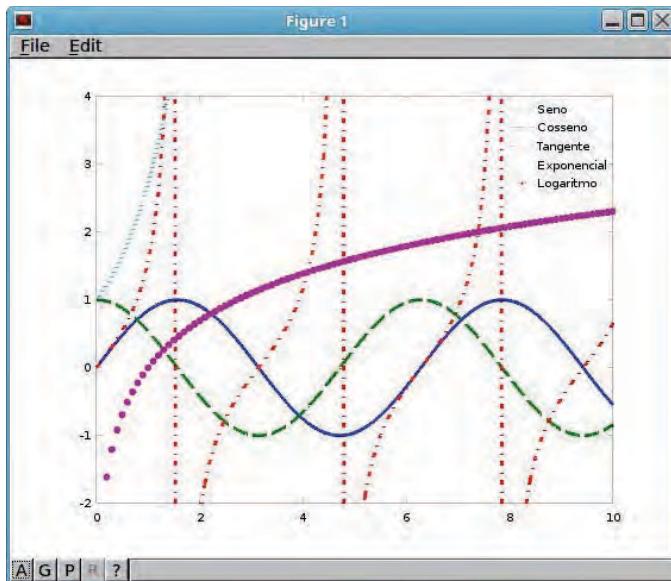


Figura 4.16: Vários gráficos customizados compartilhando a mesma figura por meio do comando hold

O próximo exemplo aborda o uso do comando `hold` para gráficos em três dimensões, brincando com senos e cossenos:

```
> x = 0:0.1:10*pi;
> y = z = x;
> plot3(x,y,z, '*', 'linewidth', 3)
> hold all
> plot3(x,sin(y),z, 'linewidth', 3)
> plot3(cos(x),y,z, '--', 'linewidth', 3)
> plot3(cos(x),y,sin(z), '-.', 'linewidth', 3)
> plot3(x,sin(y),cos(z), ':', 'linewidth', 3)
> hold off
```

Veja a segunda linha de código desse exemplo, `y = z = x`. Essa instrução cria duas novas variáveis, `y` e `z`, que recebem o mesmo valor de `x`. Poderíamos definir todas as variáveis de uma vez:

```
> x = y = z = 0:0.1:10*pi;
```

O gráfico gerado pelo exemplo aparece na figura a seguir:

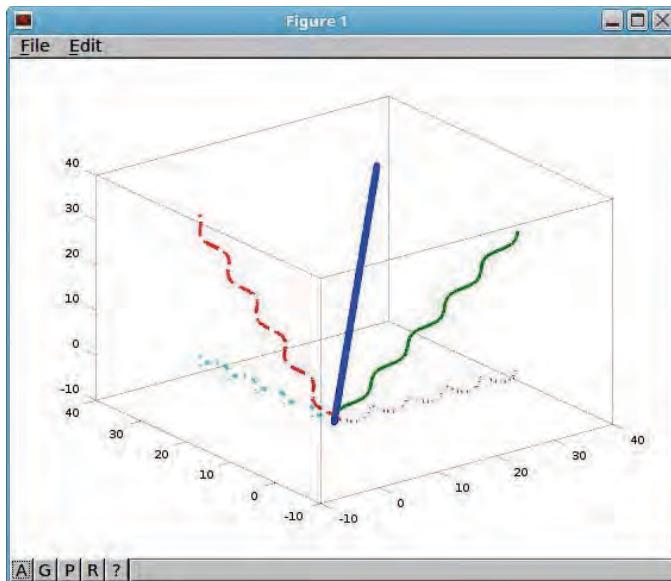


Figura 4.17: Várias curvas tridimensionais customizadas, compartilhando a mesma figura por meio do comando hold

4.3 PLOTANDO GRÁFICOS EM ÁREAS DIFERENTES DA JANELA GRÁFICA

Na seção anterior, aprendemos a plotar vários gráficos em uma mesma janela. Isso pode não ser interessante quando temos vários gráficos ou queremos focar diferentes características em cada um deles.

Em vez de trabalhar com todos os gráficos em apenas uma janela gráfica, podemos utilizar *subjanelas*. A janela de gráficos pode ser dividida em quantas subjanelas desejarmos. Para isso, usamos o comando `subplot(lin,col,ind)`, que divide a janela gráfica de acordo com o número de linhas `lin`, o número de colunas `col` e o índice que será usado para o gráfico, `ind`:

```
> x = -2*pi:0.1:2*pi;
> subplot(2,1,1)
> plot(x,sin(x),'.r','linewidth',3)
```

```

> axis([-3*pi,4*pi,-3,1.5])
> title('Gráfico de sin(x).')
> xlabel('x')
> ylabel('Seno de x')
> subplot(2,1,2)
> plot(x,cos(x), '-.m', 'linewidth',3)
> axis([-2.5*pi,2.1*pi,-1.5,2.5])
> title('Gráfico de cos(x).')
> xlabel('x')
> ylabel('Cosseno de x')

```

Dividimos a janela em duas linhas e uma coluna. Temos, então, dois espaços para os gráficos, que podem ser acessados por meio dos seus índices. Então, o primeiro e o segundo gráficos podem ser acessados por meio de `subplot(2,1,1)` e `subplot(2,1,2)` , respectivamente. Esse exemplo resulta na figura:

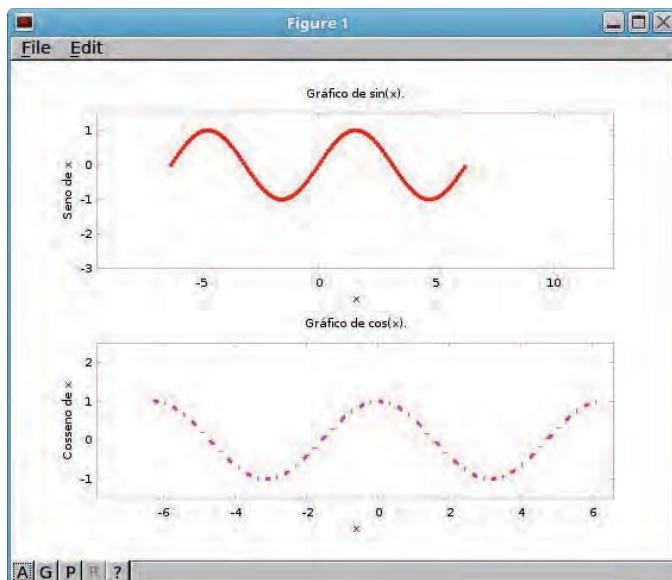


Figura 4.18: Dois gráficos separados na mesma figura, divididos em linhas por meio do comando `subplot()`

Verifique que cada gráfico dessa figura possui título, posicionamento e rótulos dos eixos independentes. Como o comando `subplot()` divide a janela original em várias janelas

gráficas, cada gráfico possui suas próprias configurações.

Trocando os comandos `subplot(2,1,1)` e `subplot(2,1,2)` do exemplo anterior por `subplot(1,2,1)` e `subplot(1,2,2)`, respectivamente, teremos uma janela gráfica com uma linha e duas colunas. O gráfico resultante dessa operação é dado na figura a seguir; compare-o com a figura anterior.

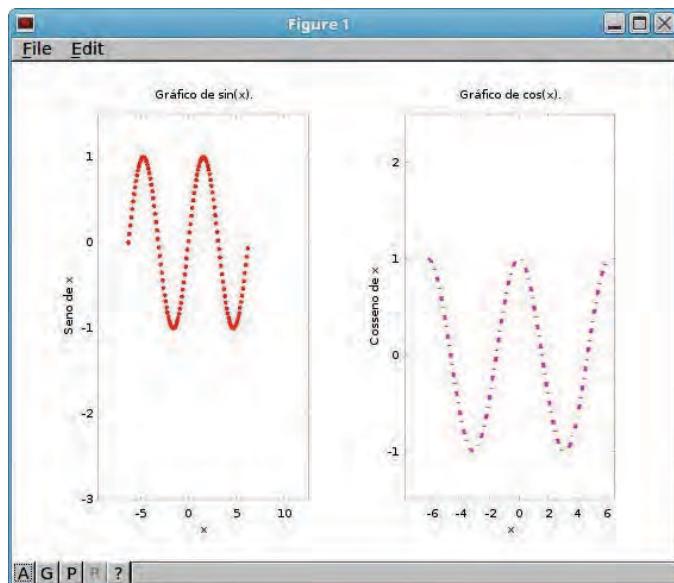


Figura 4.19: Dois gráficos separados na mesma figura, divididos em colunas por meio do comando `subplot()`

NOTA

As vírgulas entre os argumentos do comando `subplot()` podem ser omitidas. Caso quiséssemos acessar o espaço 5 em um gráfico com duas linhas e três colunas, por exemplo, `subplot(235)` seria suficiente.

O índice em `subplot()` também pode ser definido como um vetor por incremento, como no código a seguir. Ele define o índice como o intervalo de 1 a 3 em um gráfico com 2 linhas e 3 colunas, pelo comando `subplot(2,3,1:3)`.

```
> x = -2*pi:0.2:2*pi;
> subplot(2,3,1:3)
> plot(x,sin(x).*exp(-abs(x)), 'linewidth',3)
> axis([-2*pi,2*pi,-0.5,0.5])
> subplot(2,3,4)
> plot(x,sin(x).*exp(-abs(x)), 'r', 'linewidth',3)
> grid on
> axis([-2*pi,-1,-0.5,0.5])
> subplot(2,3,5)
> plot(x,sin(x).*exp(-abs(x)), 'sk', 'linewidth',3)
> axis([-1,1,-0.5,0.5])
> grid minor on
> subplot(2,3,6)
> plot(x,sin(x).*exp(-abs(x)), '--g', 'linewidth',3)
> axis([1,2*pi,-0.5,0.5])
> grid on
```

Relembre as funções que utilizamos para criar os gráficos de seno, exponencial e valor absoluto no capítulo *Primeiros passos*. Você pode recriá-los modificando as funções como preferir.

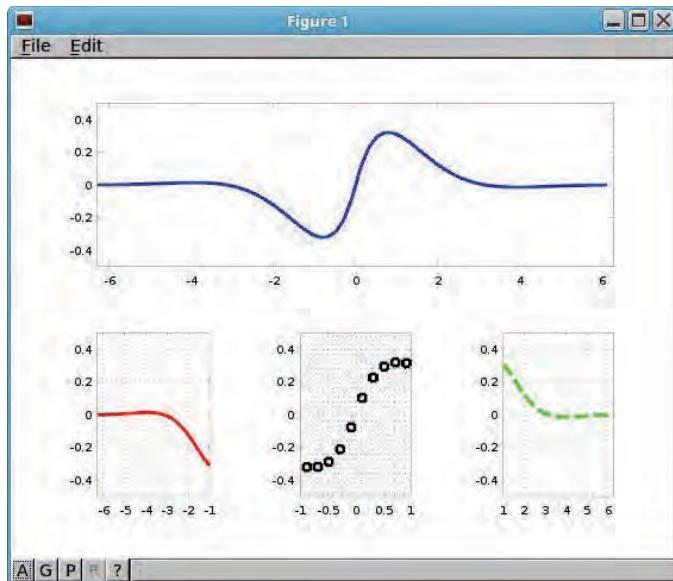


Figura 4.20: Gráfico ocupando os índices de 1 a 3 na janela por meio do comando subplot(2,3,1:3)

O primeiro gráfico traçado ocupa as posições de 1 a 3 da janela, como visto na figura anterior. Lembre-se de que cada gráfico possui suas opções distintas, como cor, grade e posicionamento dos eixos.

Para vários gráficos tridimensionais, o processo é o mesmo. Modificando o código anterior:

```
> x = -2*pi:0.2:2*pi;
> subplot(2,3,1:3)
> plot3(x,sin(x).*exp(-abs(x)),cos(x).*exp(-abs(x)),
  'linewidth',3)
> axis([-2*pi,2*pi,-0.5,0.5])
> subplot(2,3,4)
> plot3(x,sin(x).*exp(-abs(x)),cos(x).*exp(-abs(x)),
  'r','linewidth',3)
> grid on
> axis([-2*pi,-1,-0.5,0.5])
> subplot(2,3,5)
> plot3(x,sin(x).*exp(-abs(x)),cos(x).*exp(-abs(x)),
  'sk','linewidth',3)
> axis([-1,1,-0.5,0.5])
> grid minor on
```

```

> subplot(2,3,6)
> plot3(x,sin(x).*exp(-abs(x)),cos(x).*exp(-abs(x)),
    '--g','linewidth',3)
> axis([1,2*pi,-0.5,0.5])
> grid on

```

O resultado é a figura:

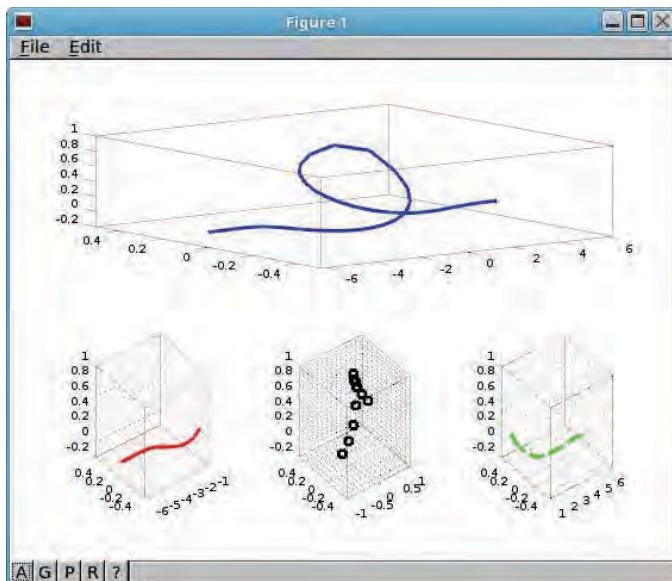


Figura 4.21: Gráfico tridimensional ocupando os índices de 1 a 3 na janela por meio do comando `subplot(2,3,1:3)`

NOTA

No caso do índice ser definido como um vetor, as vírgulas entre os argumentos são obrigatórias!

4.4 TIPOS DE GRÁFICOS BIDIMENSIONAIS

Além dos gráficos produzidos pela função `plot()`, há outras

opções de gráficos de duas dimensões. Vamos a elas.

A função `hist()`

O primeiro tipo de gráfico especial é o histograma, ou a *distribuição de frequências* de uma variável, dado pela função `hist(vetor,interv)`. O histograma é criado com a organização da variável `vetor` em ordem crescente e a divisão dos valores em intervalos (`interv`):

```
> vet_ale = 100*rand(1,100);
> hist(vet_ale,25)
> title('Histograma')
> xlabel('rand(1,100)')
> ylabel('Frequência')
```

Nesse exemplo, utilizamos a função `rand(lin,col)`, que gera uma matriz com números aleatórios distribuídos uniformemente entre zero e 1. Os argumentos são o número de linhas (`lin`) e colunas (`col`) da matriz resultante. Dessa forma, o vetor do exemplo terá 100 elementos, divididos em 1 linha e 100 colunas. Dividimos o vetor em 25 áreas iguais por meio do argumento `interv` igual a 25. O resultado é a figura:

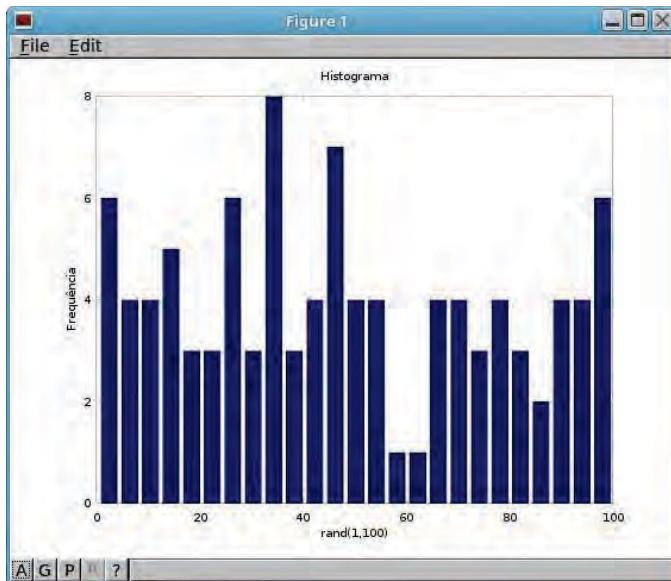


Figura 4.22: Histograma gerado por meio da função hist()

Como a função `rand()` gera números aleatórios, é provável que você tenha um gráfico diferente do que criamos aqui. Para termos sempre o mesmo conjunto de números aleatórios, podemos *configurar a semente* da função por meio de `rand('seed', num)` :

```
> rand('seed', 0);
> vet_ale = 100*rand(1,100);
```

Isso garante que sempre que `num` for zero, `rand()` retornará o mesmo conjunto de números. Experimente! Inclua `rand('seed', 0)` antes da função `rand()` no código anterior e você terá o mesmo gráfico gerado aqui. Para verificar a semente atual, use `rand('seed')` .

É possível personalizar a cor do histograma por meio dos argumentos '`facecolor`' (cor da frente) e '`edgecolor`' (cor da borda) seguidos da cor desejada. Os argumentos para as cores são os mesmos usados nos comandos `plot()` e `plot3()` . Por exemplo, o comando para gerarmos o histograma anterior verde com as

bordas vermelhas é dado a seguir. Experimente também com outras cores!

```
> hist(vet_ale,25,'facecolor','g','edgecolor','r')
```

A função bar()

Para obter gráficos de barra simples no Octave, podemos usar a função `bar(var)`. O argumento `var` pode ser um vetor ou uma matriz. O número de linhas divide os grupos; por sua vez, a quantidade de barras em cada grupo é baseada no número de colunas. A altura da barra é definida pelo valor de cada elemento:

```
> vetor = [2 1; 3 4; 3 1; 5 1; 4 2];
> bar(vetor)
```

O resultado é a figura adiante. Note que `vetor` tem cinco linhas (grupos) e duas colunas (barras em cada grupo).

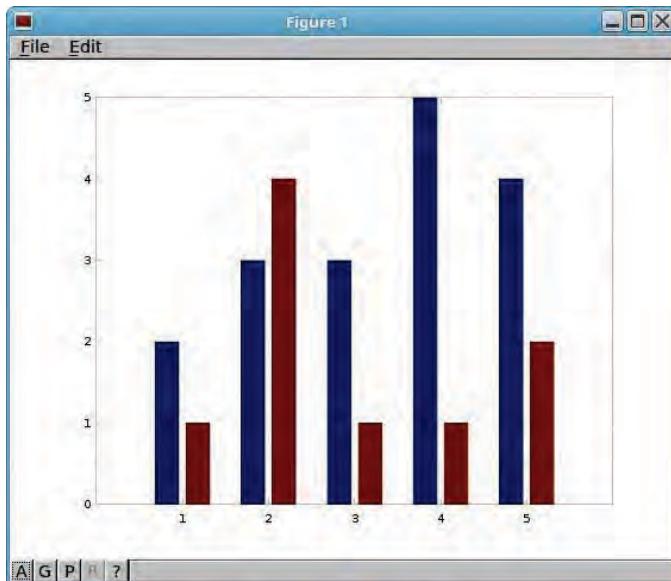


Figura 4.23: Gráfico de barras obtido por meio da função `bar()`

Da mesma forma que na função `hist()`, podemos mudar a cor

das barras utilizando os argumentos `'facecolor'` e `'edgecolor'`. Por exemplo, para gerar o gráfico anterior com barras amarelas e contorno magenta, usamos a função `bar()`:

```
> bar(x, 'facecolor', 'y', 'edgecolor', 'm')
```

A função `bar()` também pode receber um parâmetro para modificar o formato das barras. As opções disponíveis são:

- **Agrupado:** o padrão. Nele, as barras são centradas no eixo X com um espaço entre elas. O argumento é `'grouped'`.
- **Empilhado:** as barras são empilhadas; cada valor de X possui apenas uma barra. O argumento é `'stacked'`.
- **Histograma:** parecido com o formato agrupado; a diferença é que as barras não possuem espaço entre elas, como em um histograma (função `hist()`). O argumento é `'hist'`.
- **Histograma alinhado:** semelhante ao histograma, mas alinhado à esquerda em relação ao eixo X . O argumento é `'histc'`.

Por exemplo, um gráfico de barras empilhado, formado por três linhas e dez colunas de números aleatórios é dado a seguir:

```
> rand('seed', 0)
> dados_ale = rand(3, 10)
> bar(dados_ale, 'stacked')
```

Configuramos a semente (`rand('seed', 0)`) para que você consiga desenhar exatamente o mesmo gráfico, que é dado na figura:

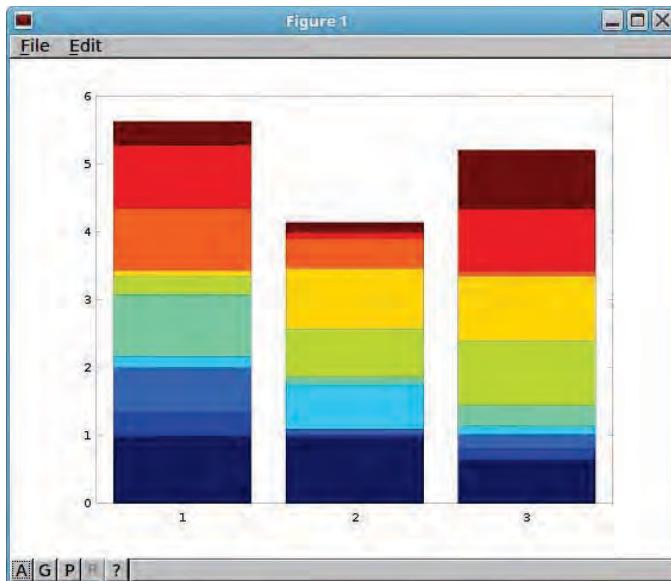


Figura 4.24: Gráfico de barras empilhado, dado pela função `bar()` com o argumento 'stacked'

As cores das barras também podem ser modificadas por *mapas de cor*, matrizes definidas no Octave com conjuntos de cores. Para isso utilizamos a função `colormap()`. Vamos modificar o código anterior, de modo que ele apresente as barras do gráfico seguindo o mapa de cor *cobre* ('copper'):

```
> rand('seed',0)
> dados_ale = rand(3,10)
> bar(dados_ale, 'stacked')
> colormap('copper')
```

O gráfico resultante está na figura a seguir. Para ver todos os mapas de cores disponíveis, digite `colormap('list')`.

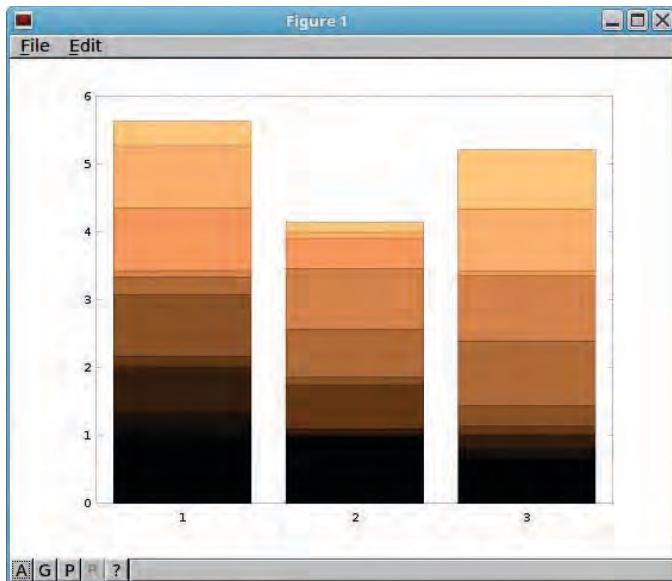


Figura 4.25: Gráfico de barras empilhado, dado pela função `bar()` com o argumento 'stacked'

NOTA

A função `bar()` plota apenas barras verticais. Para obter barras horizontais, utilize a função `barh()`, que recebe os mesmos argumentos e opções de `bar()`.

A função `scatter()`

O Octave pode plotar gráficos de dispersão (*scatter plots*) por meio da função `scatter(vet_x,vet_y)`. Esse tipo de gráfico representa a relação entre os vetores `vet_x` e `vet_y`:

```
> vetor1 = rand(1,100);
> vetor2 = rand(1,100);
> scatter(vetor1,vetor2)
```

Utilizamos dois vetores com 100 números aleatórios. A figura

seguinte corresponde ao resultado.

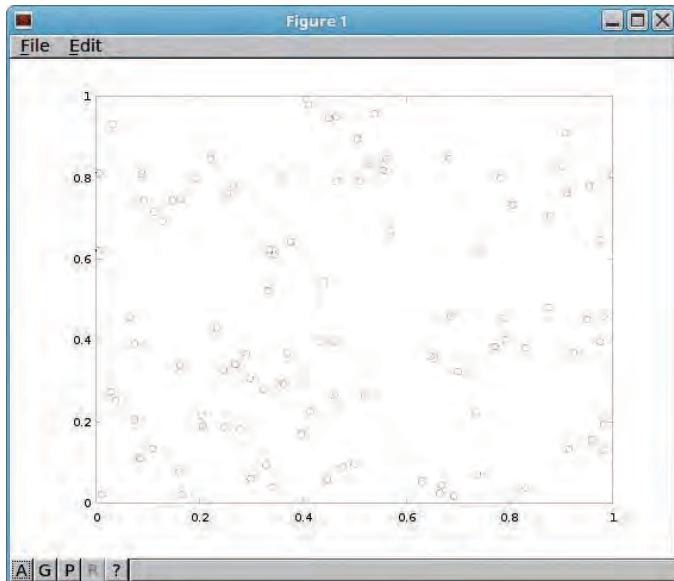


Figura 4.26: Gráfico de dispersão dado pela função scatter()

A função `scatter()` também aceita modificações. Por exemplo, podemos variar o tamanho dos elementos (vetor `escala_tam`) e também preenchê-los (argumento '`filled`'). Utilizaremos um gráfico de $\cos(\exp(1/4 \cdot x))$ com x dividido entre 200 elementos de zero a 4π para exemplificar esses argumentos:

```
> x = linspace(0,4*pi,200);
> y = cos(exp((1/4)*x));
> escala_tam = linspace(1,10,length(x));
> scatter(x,y,escala_tam,'filled')
```

Confira o gráfico resultante na figura:

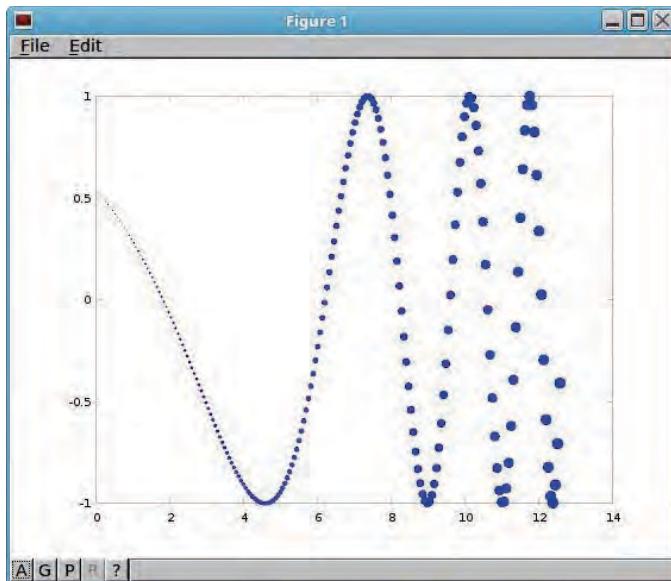


Figura 4.27: Exemplo da função scatter() com elementos preenchidos e de tamanho variante

Podemos variar também as cores dos elementos preenchidos por meio do vetor `escala_cor` . Veja que informamos um vetor vazio (`[]`) como argumento para poder alterar a escala de cores sem modificar os tamanhos:

```
> x = linspace(0,4*pi,200);
> y = cos(exp((1/4)*x));
> escala_cor = linspace(1,10,length(x));
> scatter(x,y,[],escala_cor,'filled')
```

O resultado é dado pela figura:

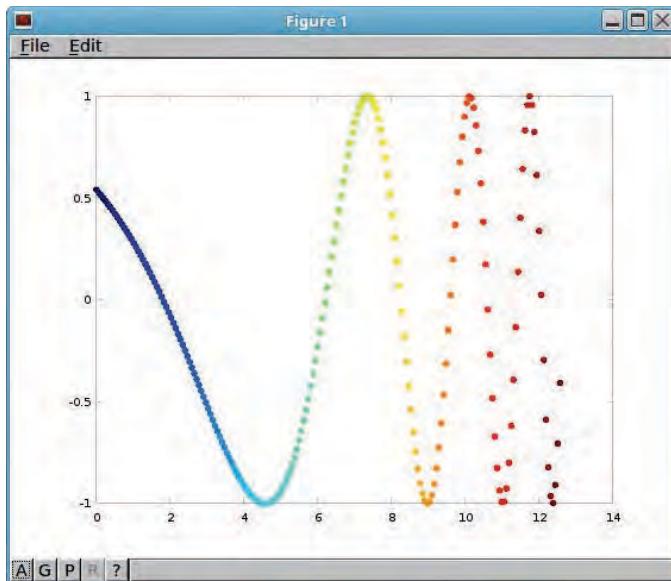


Figura 4.28: Exemplo da função scatter() com elementos preenchidos, de cores e tamanhos diferentes

Por fim, podemos preencher os elementos, variar tamanho e cor no mesmo comando, e ainda exibir um símbolo diferente. Exemplificaremos por meio de uma *função paramétrica*, em que t é o parâmetro começando em zero e terminando em 1, dividido em 250 elementos:

```
> t = linspace(0,1,250);
> x = exp(t).*sin(100*t);
> y = exp(t).*cos(100*t);
> escala_tam = linspace(1,10,length(t));
> escala_cor = linspace(1,10,length(t));
> scatter(x,y,escala_tam,escala_cor,'^','filled')
```

O resultado é a figura a seguir. Com poucos comandos, pudemos gerar um gráfico excelente!

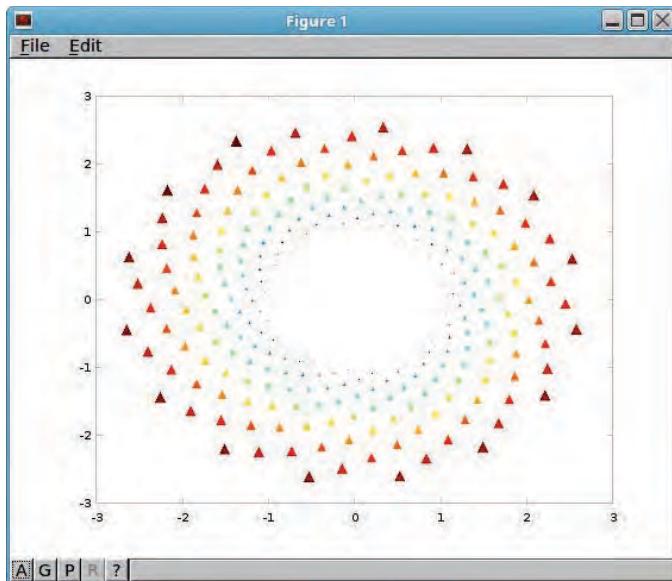


Figura 4.29: Gráfico de uma função paramétrica gerada por meio da função scatter(), com símbolo, elementos preenchidos e de tamanho e cor variantes

Note que o vetor argumento para a variação do tamanho vem na terceira posição, enquanto o argumento para a variação da cor vem na quarta (após o vetor para variação do tamanho).

NOTA

O `scatter()` possui uma função correspondente para gráficos de dispersão tridimensionais, a `scatter3(x,y,z)`. Todos os argumentos referentes à `scatter()` funcionam para `scatter3()`:

```
rand('seed',0)
vet_x = rand(1,100);
vet_y = rand(1,100);
vet_z = rand(1,100);
escala_tam = linspace(1,10,length(vet_x));
escala_cor = linspace(1,10,length(vet_x));
scatter3(vet_x,vet_y,vet_z,escala_tam,escala_cor,'filled')
```

Depois de plotar o gráfico original, experimente modificar as funções e os parâmetros. Por exemplo:

```
escala_tam = escala_cor = linspace(0,pi,length(vet_x));
```

A função `pie()`

Podemos criar um "gráfico de torta" no Octave por meio da função `pie(vet)`. Nesse caso, fornecemos um vetor `vet` em que cada elemento é uma "fatia", e `pie()` apresenta um disco com as porcentagens de cada valor relacionadas ao todo:

```
> vet_entr = [1 1 3 2 4];
> pie(vet_entr)
> title('Gráfico de torta')
```

A figura a seguir é o resultado. Repare que as porcentagens foram distribuídas considerando a soma dos valores como 100%.

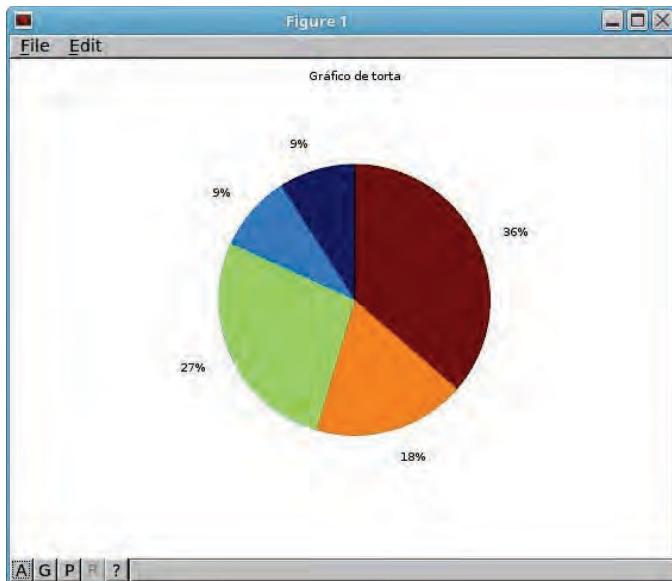


Figura 4.30: Gráfico de torta gerado pela função pie()

A função `pie()` admite também como argumentos os vetores `rot`, que contém strings com os rótulos correspondentes a cada fatia, e `explode`, cujos elementos indicam qual fatia deve ser representada separadamente:

```
> vet_entr = [1 1 3 2 4];
> rotulos = {'Casa', 'Carro', 'Celular', 'Computador', 'Outros'};
> expl = [0 0 0 0 1];
> pie(vet_entr, rotulos, expl)
> title('Gráfico de torta')
```

O resultado das instruções é a figura:

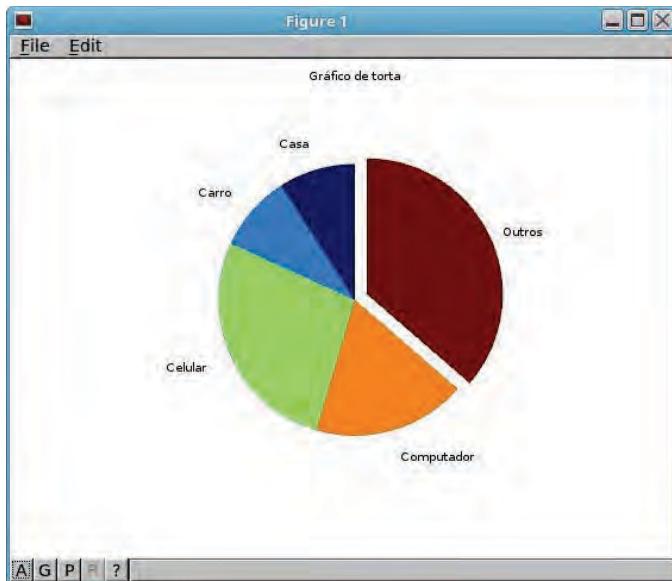


Figura 4.31: Gráfico de torta de um vetor com cinco elementos, com rótulo e com uma das fatias separada

O elemento de `vet_entr` cujo correspondente em `expl` é diferente de zero será a fatia a ser separada do gráfico. No nosso exemplo, como o elemento 5 em `expl` é igual a 1, a fatia do elemento 5 em `vet_entr` será separada.

NOTA

Observe que os argumentos `rot` e `explode` devem ter o mesmo número de elementos que o vetor de entrada; caso contrário, a função `pie()` retornará um erro!

A função `stairs()`

Um gráfico de "escadas", que conecta dois pontos em um gráfico por linhas horizontais, é dado no Octave pela função `stairs()`.

Esse tipo de gráfico é útil para visualizar funções subamostradas, como a função `sin()` do primeiro exemplo:

```
> x = 0:1:10;
> stairs(x,sin(x), 'r', 'linewidth', 4)
> xlabel('x')
> ylabel('sin(x)')
> title('Gráfico de escadas (seno)')
```

Como vimos nesse código, a função `stairs()` aceita os argumentos para diferentes cores e comprimentos de linha. Também colocamos um título no gráfico, assim como rótulos nos eixos X e Y . Além disso, podemos usar traços ou símbolos diferentes, como no comando `plot()`.

O gráfico resultante é dado na figura a seguir. Compare com a figura da seção *As funções essenciais para gráficos de duas e três dimensões*; veja a diferença entre a conexão dos pontos nas duas funções.

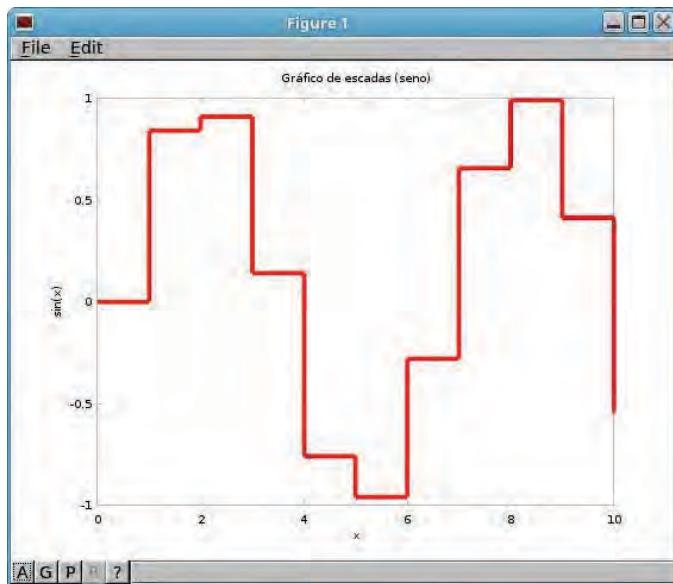


Figura 4.32: Gráfico de escadas dado pela função `stairs()`

A função stem()

Encerrando o estudo das funções para gráficos bidimensionais, temos a função `stem()`, uma representação para dados discretos (ou espaçados). Essa função liga o ponto ao eixo X por uma haste, facilitando a comparação entre os pontos. Plotando a função `sin()` do primeiro exemplo com `stem()`, temos:

```
> x = 0:1:10;
> stem(x,sin(x), 'm', 'filled', 'linewidth', 3)
> xlabel('x')
> ylabel('sin(x)')
> title('Gráfico discreto, dado por stem (seno)')
```

A função `stem()` também admite várias personalizações, como cores e espessura das linhas. O argumento '`filled`' preenche os círculos das hastes. O resultado é visto na figura:

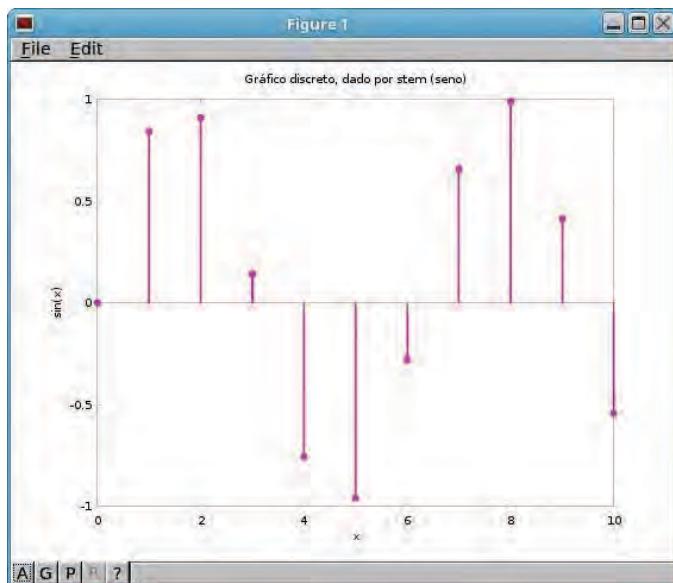


Figura 4.33: Gráfico discreto obtido pela função `stem()`

4.5 TIPOS DE GRÁFICOS TRIDIMENSIONAIS

Além de `plot3()` e `scatter3()`, há outras funções para gráficos de três dimensões. Elas são basicamente as funções `mesh()`, `surf()` e `contour()`. Entretanto, em vez de gerar retas (como `plot3()`) ou elementos dispersos (como `scatter3()`), essas funções plotam *superfícies*.

A função `mesh()`

O primeiro gráfico de superfícies que abordaremos é o gráfico de *malha*, dado pela função `mesh(mat_x, mat_y, mat_z)`. As variáveis `mat_x`, `mat_y` e `mat_z` são matrizes que representam os valores dos pontos em cada eixo (X , Y e Z , respectivamente).

Focaremos nos exemplos para funções de duas variáveis. Nesse caso, geralmente criamos um vetor-base com os pontos que representarão os eixos X e Y , e definimos seus valores por meio da função `meshgrid(vet_x)`. Essa função cria uma *grade* com várias cópias de `vet_x`, de forma que seu resultado seja uma matriz quadrada:

```
> comp = 1:5
comp =
  1  2  3  4  5
> area = meshgrid(comp)
area =
  1  2  3  4  5
  1  2  3  4  5
  1  2  3  4  5
  1  2  3  4  5
  1  2  3  4  5
```

Note que como `comp` tem 5 elementos, a variável `area` recebe uma matriz de ordem 5 (5 linhas e 5 colunas). Para gerar duas variáveis correspondendo aos eixos X e Y com os valores dos pontos baseados no mesmo vetor, podemos pedir à `meshgrid()` que ela retorne os dois valores:

```
> vetor = [1 2 3 4 5];
> [area_x,area_y] = meshgrid(vetor)
```

```

area_x =
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
area_y =
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5

```

Se quisermos basear os valores do eixo Y em um vetor diferente daquele informado para o eixo X, informaremos os dois vetores como argumentos em `meshgrid()` :

```

> vet1 = [1 2 3 4 5];
> vet2 = [-1 -2 -3 -4 -5];
> [area_x,area_y] = meshgrid(vet1,vet2)
area_x =
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
area_y =
-1 -1 -1 -1 -1
-2 -2 -2 -2 -2
-3 -3 -3 -3 -3
-4 -4 -4 -4 -4
-5 -5 -5 -5 -5

```

Usando `meshgrid()` , podemos obter vários pontos para plotar uma função de duas variáveis. O código a seguir, por exemplo, utiliza um vetor grade de -8 a 8 com 41 pontos. Então, as variáveis `xx` e `yy` recebem a grade gerada, e a função `zz` é calculada com base em seus valores. Por fim, usamos a função `mesh()` para obter a superfície:

```

> grade = linspace(-8,8,41);
> [xx,yy] = meshgrid(grade);
> zz = sin(sqrt(xx.^2 + yy.^2)) ./ sqrt(xx.^2 + yy.^2);
> mesh(xx,yy,zz, 'linewidth',2)

```

Esse código plota a famosa função *sombrero*, apresentada na figura:

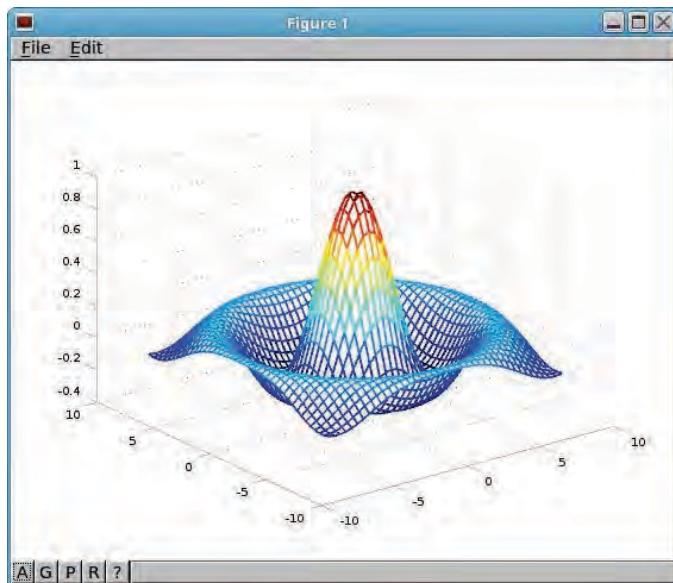


Figura 4.34: Gráfico da função sombrero, obtido por `mesh()` e `meshgrid`

NOTA

A superfície pode ser rotacionada pressionando `R` no teclado, ou por meio do botão `R` no canto inferior esquerdo da janela.

A função `mesh()` também aceita diferentes mapas de cor. Experimente! Utilize os comandos `colormap('autumn')` e `colormap('winter')` no exemplo anterior.

NOTA

O gráfico anterior pode ser reproduzido usando a função `sombrero(num_grade)` , com `num_grade` sendo o número de pontos da grade para cada variável:

```
[xx,yy,zz] = sombrero(41);
mesh(xx,yy,zz, 'linewidth', 2)
```

Existe também a função `meshc()` , que desenha um contorno da projeção da superfície no eixo X. Seus argumentos são os mesmos que os da função `mesh()` :

```
> grade = linspace(-3,3,49);
> [xx,yy] = meshgrid(grade);
> zz = 3*(1-xx).^2.*exp(-xx.^2 - (yy+1).^2) ...
- 10*(xx./5 - xx.^3 - yy.^5).*exp(-xx.^2-yy.^2) ...
- 1/3.*exp(-(xx+1).^2 - yy.^2);
> meshc(xx,yy,zz, 'linewidth', 2)
```

Esse código desenha a função *peaks*, que possui vários picos e depressões (conhecidos na matemática como *pontos de máximo e mínimo*), além dos contornos de sua projeção no eixo X. Veja o gráfico na figura:

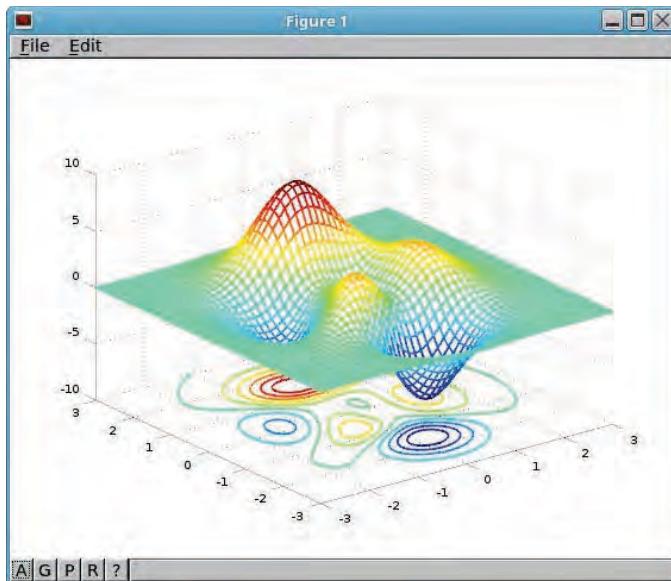


Figura 4.35: Gráfico da função peaks e contornos de sua projeção no eixo X obtidos por meshc() e meshgrid

NOTA

O gráfico anterior pode ser reproduzido com a função `peaks(num_grade)`, com `num_grade` sendo o número de pontos da grade, da mesma forma que para `sombrero()`:

```
[xx,yy,zz] = peaks(49);
meshc(xx,yy,zz,'linewidth',2)
```

A função surf()

A função `surf()` é utilizada da mesma forma que `mesh()`. A diferença é que as superfícies geradas por `surf()` são preenchidas:

```
> grade = linspace(-5,5,41);
> [xx,yy] = meshgrid(grade);
> zz = sin(sqrt(xx.^2 + yy.^2)) .* sqrt(xx.^2 + yy.^2);
```

```
> surf(xx,yy,zz)
> colormap('hsv')
```

Utilizamos a função `meshgrid()` para gerar os pontos `xx` e `yy`, a partir do vetor `grade`. Também modificamos as cores da superfície por meio da função `colormap()`; o mapa de cores escolhido foi o '`hsv`'. O gráfico resultante é dado na figura a seguir.

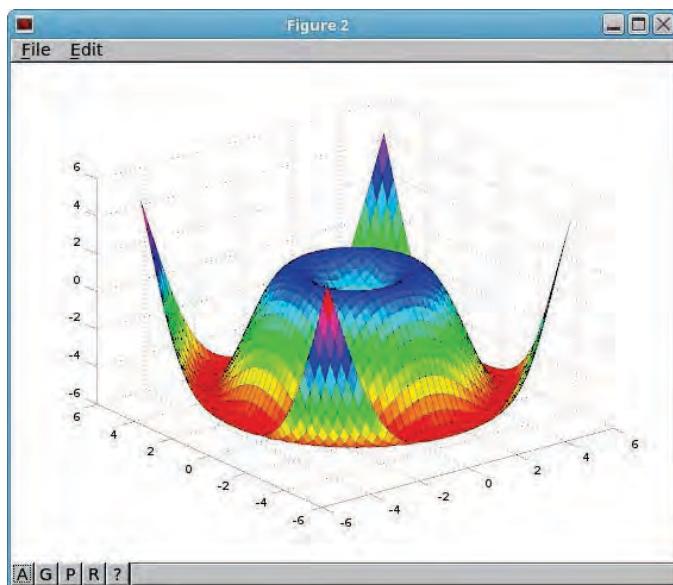


Figura 4.36: Superfície obtida pela função `surf()`

A função `surf()` também possui uma correspondente que plota os contornos da projeção da superfície no eixo X, a `surf()`. Os argumentos são os mesmos que para `surf()`:

```
> grade = linspace(-2,2,40);
> [xx,yy] = meshgrid(grade);
> zz = exp(cos(xx.^2 + yy.^2)); % .* sqrt(xx.^2 + yy.^2);
> surf(xx,yy,zz)
> shading interp
```

O comando `shading interp` após `surf()` retira as linhas da grade. Veja o gráfico resultante na figura:

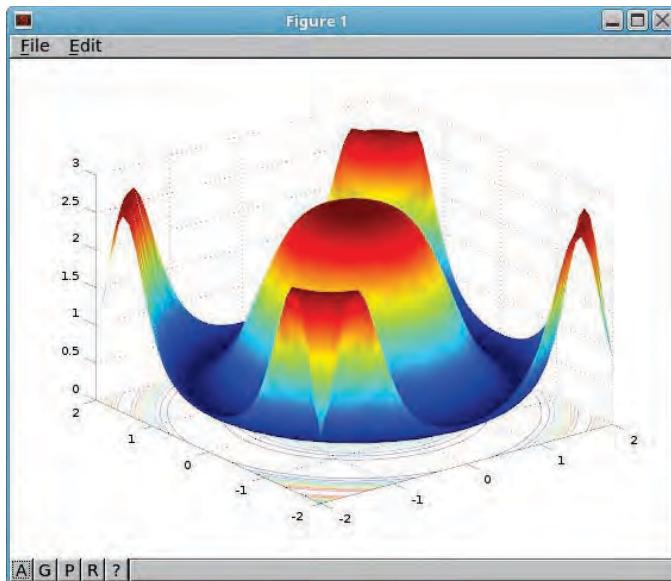


Figura 4.37: Superfície de uma função e contornos de sua projeção no eixo X obtidos pela função `surf()`

NOTA

Existe também a função `surf1(xx,yy,zz)` , que baseia as cores da superfície em modelos de luz. Ela pode ser usada da mesma forma que `surf()` e `surf()` . Experimente usando os exemplos anteriores!

A função `contour()`

A última função que veremos para gráficos tridimensionais, `contour()` , não traz como resultado um gráfico de superfície: ela gera o contorno da projeção da superfície no eixo X. Seus argumentos são os mesmos que os de `mesh()` ou `surf()` . Usaremos as funções *sombrero* e *peaks* como exemplo:

```

> subplot(211)
> [xx,yy,zz] = sombrero(41);
> contour(xx,yy,zz, 'linewidth',2)
> axis off
> subplot(212)
> [xx,yy,zz] = peaks(49);
> contour(xx,yy,zz, 'linewidth',2)

```

Nesse código, dividimos a janela gráfica em duas linhas com a função `subplot()`. O primeiro gráfico é o contorno de *sombrero*, enquanto que o segundo é o contorno de *peaks*. Veja o resultado na figura:

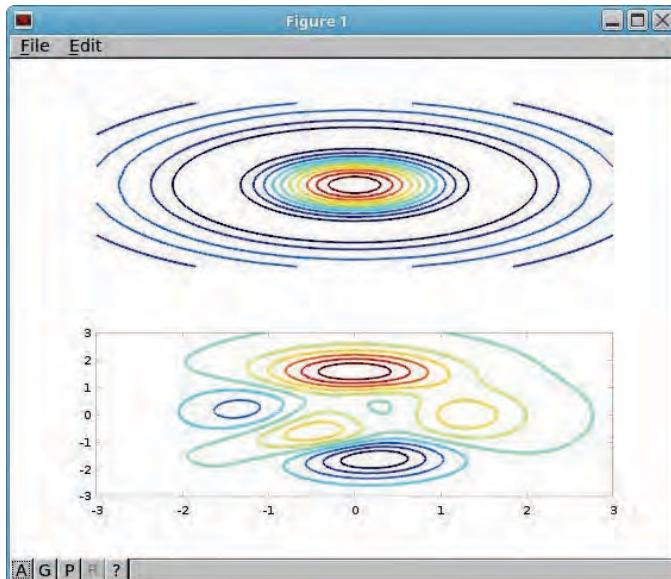


Figura 4.38: Contornos da projeção de sombrero e peaks no eixo X, obtidos pela função `contour()`

NOTA

Observe que utilizamos o comando `axis off`, que retira os valores dos eixos de *sombrero*. Esse comando pode ser usado em qualquer gráfico.

A função `contour()` também possui uma variante, a `contourf()`. Como você já deve ter previsto, essa função plota os contornos preenchidos:

```
> grade = linspace(-10,10,60);
> [xx,yy] = meshgrid(grade);
> zz = cos(sqrt(xx.^2 + yy.^2)) .* sqrt(xx.^2 + yy.^2);
> contourf(xx,yy,zz)
```

O resultado é a figura adiante. O gráfico é mais representativo que `contour()`, não acha?

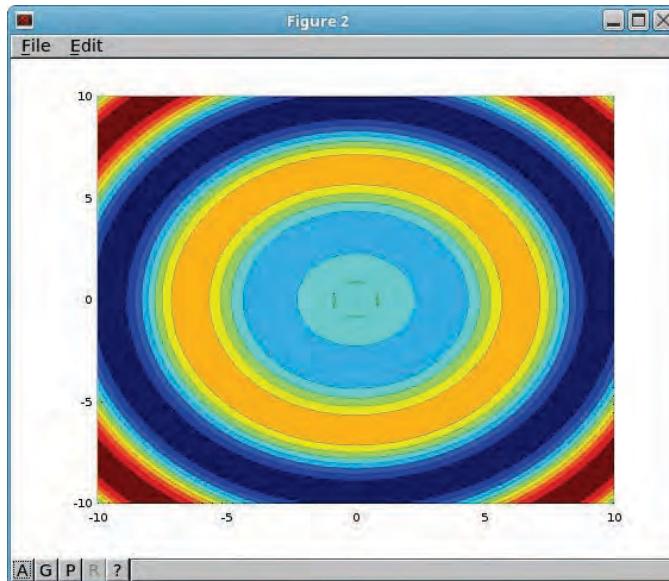


Figura 4.39: Gráfico de contornos preenchidos, obtido com o uso da função `contourf()`

Vamos reforçar o nosso aprendizado? Um exercício interessante é comparar o gráfico gerado por `meshc()` ou `surf()`, com aquele obtido por `contour()` ou `contourf()`:

```
> subplot(211)
> [xx,yy,zz] = sombrero(41);
> meshc(xx,yy,zz, 'linewidth',2)
> subplot(212)
> contour(xx,yy,zz, 'linewidth',2)
> figure;
> [xx,yy,zz] = peaks(49);
> subplot(211)
> surf(xx,yy,zz, 'linewidth',2)
> subplot(212)
> contourf(xx,yy,zz, 'linewidth',2)
```

Repare no comando `figure`, que cria uma nova janela gráfica. Assim, os resultados de `meshc()` e `contour()` são apresentados em uma janela, enquanto que os resultados de `surf()` e `contourf()` aparecem em outra.

Repita esses exemplos utilizando outras superfícies. Verifique a relação entre `contour()` / `contourf()` e a superfície.

4.6 RESUMINDO

Estudamos vários tipos de gráficos de duas e três dimensões neste capítulo. Vimos como personalizar suas cores, formas, tamanhos e símbolos, aprendemos como alterar os intervalos dos eixos (e até retirá-los), além de plotar vários gráficos na mesma janela, ou dividir a janela para acomodar gráficos em diferentes espaços.

No próximo capítulo, veremos como reutilizar nossos programas criando *scripts* e funções no Octave. Também aprenderemos a comentar nosso código com informações relevantes para que seja mais fácil mantê-lo e aprimorá-lo no futuro. Até mais!

CAPÍTULO 5

GRAVANDO E REAPROVEITANDO CÓDIGO

Até agora, usamos o prompt de comando diretamente para interagir com o Octave, o que é satisfatório para resolver pequenos problemas ou plotar alguns gráficos. Contudo, quando a complexidade dos programas em que estivermos trabalhando aumentar, digitar todos os comandos no prompt poderá não ser tão prático.

Para reutilizar alguns dos programas nos quais viemos trabalhando, podemos gravar os códigos em um arquivo, que pode ser digitado diretamente em um editor de textos, e dizer ao Octave para executar as instruções a partir dele. Um arquivo contendo código do Octave é chamado de *arquivo M*, por possuir a extensão `.m`. Os arquivos M podem ser apenas *scripts* ou conter funções.

5.1 SCRIPTS

Um script é um arquivo M que contém uma série de instruções a serem seguidas, como se digitássemos o código diretamente no prompt do Octave. Esses arquivos podem ser utilizados para executar uma sequência de código de uma só vez, e são úteis para evitar que o mesmo programa seja digitado várias vezes.

Gerar um script é simples: abra o seu editor de texto preferido, escreva o código como se estivesse trabalhando no prompt e grave o resultado em disco. A escolha do nome do arquivo é livre, mas a extensão deve ser `.m`.

Como exemplo, utilizaremos o último código do capítulo anterior:

```
grade = linspace(-10,10,60);
[xx,yy] = meshgrid(grade);
zz = cos(sqrt(xx.^2 + yy.^2)) .* sqrt(xx.^2 + yy.^2);
contourf(xx,yy,zz)
```

Grave esse código em um arquivo com o nome `contorno_cheio.m`. Depois, no prompt do Octave, utilize o comando `cd` para ir ao diretório em que `contorno_cheio.m` está gravado; por exemplo, se o arquivo estiver na pasta `~/Documentos/Octave`, o comando digitado no prompt será:

```
> cd ~/Documentos/Octave
```

Para verificar em que pasta o Octave está sendo executado, usamos o comando `pwd`. O comando `ls`, por sua vez, lista os arquivos presentes na pasta atual. A instrução `dir` é semelhante a `ls`, mas exibe também os arquivos ocultos.

```
> pwd
ans = ~/Documentos/Octave
> ls
contorno_cheio.m
> dir
.
..
contorno_cheio.m
```

Quando o Octave estiver executando na pasta em que o arquivo `contorno_cheio.m` foi gravado, poderemos digitar seu nome no prompt:

```
> contorno_cheio
```

Então o Octave executará todo o código gravado no arquivo, linha por linha, sem que precisemos digitar seu conteúdo no prompt. Simples, não é mesmo?

Quando o script não está na mesma pasta em que o Octave está sendo executado, usamos a função `run('caminho_script')`. Nesse caso, `caminho_script` é o caminho completo do script, formado pela pasta na qual ele está juntamente com seu nome. Para o arquivo `contorno_cheio`, a função é a seguinte:

```
> run('~/Documentos/Octave/contorno_cheio')
```

As funções, que veremos adiante, também podem ser gravadas junto com scripts em um mesmo arquivo M. Mas, antes de estudá-las, vejamos como documentar seu código por meio de comentários.

5.2 COMENTÁRIOS

Escrevendo o seu trabalho de conclusão de curso, você se lembra de que aquele código que gera os gráficos da sua apresentação já foi escrito há alguns anos, com a ajuda de um colega. Ele escreveu parte do código e usou algumas variáveis de nome diferente das suas, e você acaba tentando adivinhar o que vocês pensaram na época em que o código foi escrito.

Para que serve cada variável? E aquela instrução mais elaborada, o que seria? É interessante lembrar como resolvemos algum problema, e comentar nossos programas pode ser uma solução eficaz para isso.

Comentários são indicados pelos caracteres cerquilha (`#`) ou porcento (`%`), e podem ser utilizados para documentar o código contido nos arquivos M. O texto que digitarmos em frente de `'#'` ou `%` não é executado:

```
raio = input('Digite o valor do raio da circunferência:');
```

```
area = pi*raio^2; % Calcula a área para o raio fornecido
disp('O valor da área dessa circunferência é: ');
disp(area);
```

Grave esse código em um arquivo de nome `area_circ.m`. Nesse programa, o trecho `% Calcula a área para o raio fornecido` não será interpretado pelo Octave. O comentário também pode ser feito em uma linha somente dele, como no código a seguir:

```
raio = input('Digite o valor do raio da esfera: ');
% O volume da esfera é calculado a partir do raio fornecido
volume = (4*pi*raio^3)/3;
disp('O valor do volume dessa esfera é: ');
disp(volume);
```

Esse exemplo será gravado em um arquivo nomeado `vol_esfera.m`. Da mesma forma que no código anterior, a linha `% O volume da esfera é calculado a partir do raio fornecido` não será executada pelo Octave.

É possível construir um bloco de comentários com várias linhas inserindo os comentários entre os marcadores de início (`#{` ou `%{`) e fim (`#}` ou `%}`). No código a seguir, os comentários em bloco descrevem o que o programa faz. Grave-o em um arquivo com o nome `cels_fahr.m`.

```
%{
  O código a seguir recebe uma temperatura do usuário em graus
  Celsius e a converte para graus Fahrenheit.
}

celsius = input('Digite o valor da temperatura
    em graus Celsius: ');
fahrenheit = celsius*1.8 + 32;
disp('O valor da temperatura em graus Fahrenheit é: ');
disp(fahrenheit);
```

Comentar seus scripts com expressões significativas será útil quando precisar resolver problemas neles e ampliar suas funcionalidades, e também ajudará outras pessoas que usem e

ampliem esse código.

Nas funções que veremos a seguir, os comentários são ainda mais importantes: além de mostrar ao usuário o que seu código faz, os blocos de comentários podem ser utilizados como um cabeçalho, contendo informações relevantes como um guia para utilizar a função, seu autor, a licença de uso, ou até um agradecimento ao colega que o ajudou a escrevê-la.

5.3 FUNÇÕES

Funções são partes independentes do programa que aceitam a entrada de dados e podem retornar resultados. Elas são excelentes para o reaproveitamento de código, auxiliando na criação de programas maiores.

Uma função é declarada no Octave pela palavra `function` seguida de seu nome. Terminamos a função com a expressão `end`.

A função mais simples possível começa com `function nome_funcao` e termina logo depois:

```
> function nao_faco_nada  
> end
```

Após digitarmos essa função no Octave, ele passa a reconhecer a expressão `nao_faco_nada` como a função que declaramos. Desse modo, ela é executada quando digitamos seu nome no prompt:

```
> nao_faco_nada  
>
```

Repare que a função não possui entrada ou saída, nem apresenta texto na tela; o prompt aparece em seguida, pronto para voltar ao trabalho. Podemos definir uma função um pouco mais útil, a `texto_simples`, que mostre uma frase na tela:

```
> function texto_simples
```

```
>     disp('Só imprimo este texto.')
> end
```

Da mesma forma, digitamos seu nome no prompt para que ela seja executada:

```
> texto_simples
Só imprimo este texto.
```

Veja que a função `texto_simples` retorna um texto, mas não recebe entrada ou manipula dados. Para definir uma função que receba e retorne valores, indicamos as variáveis que a função receberá como seus argumentos, à direita do nome, enquanto os valores obtidos serão declarados antes, separados do nome por um sinal de igual:

```
function [res_1, res_2, res_3] = funcao_teste(val_1, val_2)
    comandos;
end
```

Nessa representação, a função `funcao_teste` recebe os argumentos `val_1` e `val_2`, e retorna `res_1`, `res_2` e `res_3`. Usaremos esses conceitos na função `quad()` a seguir. Ela utiliza a fórmula de Bhaskara para encontrar as raízes de uma equação do segundo grau. Os comentários em bloco são utilizados como um cabeçalho, apresentando a função.

```
function [raiz_1, raiz_2] = quad(val_a, val_b, val_c)
%
%[
[raiz_1, raiz_2] = quad(val_a, val_b, val_c)

QUAD encontra as raízes (ou zeros) X1 (raiz_1) e X2 (raiz_2)
da equação quadrática A*X^2 + B*X + C = 0 pela fórmula de
Bhaskara.
%}

delta = val_b^2 - 4*val_a*val_c;
raiz_1 = (-val_b + sqrt(delta))./(2*val_a);
raiz_2 = (-val_b - sqrt(delta))./(2*val_a);
end
```

NOTA

Não se recorda da fórmula de Bhaskara? A Academia Khan possui um tutorial sobre equações do segundo grau, sua representação gráfica e suas soluções. Veja em <https://pt.khanacademy.org/math/algebra/quadratics>.

Após definir a função `quad()`, podemos usá-la no Octave digitando seu nome e os argumentos em seguida. Suponhamos que `val_a` , `val_b` e `val_c` sejam iguais a 2, -4 e zero, respectivamente, e que `x1` e `x2` receberão os valores das raízes:

```
> [x1, x2] = quad(2, -4, 0)
x1 = 8
x2 = 0
```

Vamos reforçar nosso aprendizado? Crie funções que solucionem as situações a seguir:

1. Reescreva o código de `contorno_cheio.m` , de forma que os argumentos da função `linspace()` sejam passados a ela por uma função criada por você.
2. Com base no arquivo `area_circ.m` , crie uma função de nome `area_circunferencia()` que receba o valor de `raio` como argumento, e retorne a variável `area` .
3. Como no exemplo anterior, com base no arquivo `vol_esfera.m` , crie uma função de nome `volume_esfera()` que receba o valor de `raio` como argumento e retorne a variável `volume` .
4. Crie uma função nomeada `celsius_fahr()` , com base no arquivo `cel_fahr.m` , que receba graus Celsius como argumento e retorne esse valor em graus Fahrenheit.
5. Escreva também uma função similar, de nome

`fahr_celsius()` , que faça a conversão contrária; ela deve receber uma temperatura em graus Fahrenheit e convertê-la para graus Celsius.

Há a opção de gravar uma função em um arquivo M, criando assim um *arquivo de função*. Nesse caso, os nomes do arquivo e da função devem coincidir. O arquivo pode conter várias funções, desde que o nome da primeira função seja o nome do arquivo.

Chamando uma função no código de outra função

Podemos pedir que uma função definida execute outras funções em seu código, como se a chamássemos no prompt; devemos indicar o nome da função e, se for necessário, seus argumentos de entrada e saída.

Um exemplo de função que chama outras funções é `grafico_quad()` , definida a seguir. Essa função esboça o gráfico de uma função do segundo grau, marcando também suas raízes como pontos vermelhos no gráfico.

```
function grafico_quad()
{
    grafico_quad()

    GRAFICO_QUAD plota o gráfico da função do segundo grau
    Y = A*X^2+B*X+C, com A (coef_a), B (coef_b) e C (coef_c)
    fornecidos pelo usuário, além das raízes X1 (x0_1) e X2
    (x0_2), obtidas com o auxílio das funções DISCR e RAIZES.
}

disp('*** Gráfico de Y = A*X^2+B*X+C,
      com X = [-50, 50]. ***');
coef_a = input('Digite o valor do coeficiente A: ');
coef_b = input('Digite o valor do coeficiente B: ');
coef_c = input('Digite o valor do coeficiente C: ');

% Chamando a função discr() com os argumentos obtidos.
val_delta = discr(coef_a, coef_b, coef_c);

% Obtendo X1 e X2 por meio da função raizes().
```

```

[raiz_1, raiz_2] = raizes(val_delta, coef_a, coef_b);

% Calculando os pontos usando a função quadratica().
[eixo_x, eixo_y] = quadratica(coef_a, coef_b, coef_c);

% Plotando os gráficos com a função parabola().
parabola(eixo_x, eixo_y, raiz_1, raiz_2);
end

```

Veja que `grafico_quad()` não requer argumentos. Os coeficientes A (`coef_a`), B (`coef_b`) e C (`coef_c`), necessários para o cálculo das raízes da equação quadrática, são informados por meio da função `input()` durante a execução do programa.

Dividir o código em várias funções menores auxilia quando for necessário acrescentar novas funcionalidades, assim como para resolver eventuais problemas. Quatro funções são chamadas por `grafico_quad()` : `discr()` , `parabola()` , `quadratica()` e `raizes()` . Vamos defini-las a seguir, na ordem em que são executadas.

A função `discr()` calcula o discriminante da equação quadrática definida pelos argumentos A (`vala`), B (`valb`) e C (`valc`), retornando a variável `delta` .

```

function delta = discr(vala, valb, valc)
%{
delta = discr(vala, valb, valc)

DISCR calcula o discriminante (delta) da equação
quadrática  $A \cdot X^2 + B \cdot X + C = 0$ , a partir dos valores
de A (vala), B (valb) e C (valc).
%}

delta = valb^2 - 4*vala*valc;
end

```

Em seguida, a função `raizes()` calcula as raízes da equação quadrática por meio da fórmula de Bhaskara. Seus argumentos são o discriminante `delta` e os coeficientes `vala` e `valb` . As variáveis resultantes são as raízes `X1` (`raiz1`) e `X2` (`raiz2`).

```

function [raiz1, raiz2] = raizes(delta, vala, valb)
{
    [raiz1, raiz2] = raizes(delta, vala, valb)

    RAIZES encontra as raízes X1 (raiz1) e X2 (raiz2) da
    equação quadrática A*X^2 + B*X + C = 0 pela fórmula de
    Bhaskara utilizando o discriminante (delta) e os
    coeficientes A (vala) e B (valb).
}

raiz1 = (-valb + sqrt(delta))/(2*vala);
raiz2 = (-valb - sqrt(delta))/(2*vala);
end

```

A função `quadratica()`, por sua vez, define X entre -50 e 50 (`eixox`), e calcula os valores de Y por meio da função quadrática (`eixoy`). Os argumentos de entrada são os coeficientes `vala`, `valb` e `valc`.

```

function [eixox, eixoy] = quadratica(vala, valb, valc)
{
    [eixox, eixoy] = quadratica(vala, valb, valc)

    QUADRATICA calcula os valores da função quadrática
     $Y = A \cdot X^2 + B \cdot X + C$  para todos os valores de  $X$  (eixox),
    definido entre -50 e 50.
}

% O valor da equação quadrática depende dos valores de x.
eixox = -50:0.01:50;
% Como eixox é um vetor, usamos .* e .^ para multiplicação
% e exponencial por todos os elementos
eixoy = vala.*eixox.^2 + valb.*eixox + valc;
end

```

Finalmente, a função `parabola()` é responsável por plotar os gráficos da equação quadrática e de suas raízes. Seus argumentos são os valores dos eixos X e Y (`eixox` e `eixoy`) e as raízes da equação (`raiz1` e `raiz2`).

```

function parabola(eixox, eixoy, raiz1, raiz2)
{
    parabola(eixox, eixoy, raiz1, raiz2)

    PARABOLA é responsável por plotar o gráfico da função

```

```

quadrática e também as raízes X1 (raiz1) e X2 (raiz2),
obtidas por meio da função RAIZES.

}

disp('As raízes são:');
disp(raiz1);
disp(raiz2);

% Gráfico da função e das raízes.
plot(eixox, eixoy, 'linewidth', 2)
hold on;
plot(raiz1, 0, 'r*', 'linewidth', 4)
plot(raiz2, 0, 'r*', 'linewidth', 4)
hold off;
end

```

Note que `grafico_quad()` só é responsável por receber os parâmetros da equação; as outras tarefas são delegadas às outras funções. Podemos gravar essas funções em um mesmo arquivo. Se a função `grafico_quad()` vier em primeiro lugar no arquivo, devemos tomar o cuidado de nomeá-lo como `grafico_quad.m`: como visto anteriormente, o nome da primeira função deve ser o nome do arquivo.

Vamos reforçar nosso aprendizado? Resolva os problemas a seguir:

1. Crie uma função que, a partir do valor de um raio dado pelo usuário, calcule a área da circunferência e o volume da esfera definidos por esse raio. Para isso, essa função deve utilizar as funções `area_circunferencia()` e `volume_esfera()` criadas anteriormente.
2. Modifique a função `quadratic()` para que o valor dos pontos inicial e final de `eixox` seja recebido como argumento da função.
3. Modifique a função `grafico_quad()` de forma que ela utilize a função `quad()`, em vez de `discr()` e `raizes()`, no cálculo das raízes da equação quadrática.

Variáveis locais de uma função

Voltando ao exemplo anterior, repare no comando usado em `grafico_quad()` para executar `discr()`. Os argumentos de entrada são `coef_a`, `coef_b` e `coef_c`, enquanto a variável resultante é `delta`.

```
val_delta = discr(coef_a, coef_b, coef_c);
```

Agora, compare com a forma que `discr()` foi definida:

```
delta = discr(vala, valb, valc)
```

Os argumentos de entrada são `vala`, `valb` e `valc`; a variável resultante é `delta`. Eles são diferentes dos chamados por `grafico_quad()`, mas `discr()` não se importa com isso.

Dentro dessa função, os valores passados a ela assumem as variáveis `vala`, `valb` e `valc`. Elas são o que chamamos de *variáveis locais*.

Variáveis locais são definidas dentro de uma função, sendo conhecidas apenas por ela. Por exemplo, vejamos a função `cubica()`, que fornece o gráfico de uma função do terceiro grau a partir de seus coeficientes:

```
function cubica(coef_a, coef_b, coef_c, coef_d)
{
    cubica(coef_a, coef_b, coef_c, coef_d)

    CUBICA calcula os pontos da função cúbica
    Y = A*X^3 + B*X^2 + C*X + D e gera seu gráfico.
}

% Definindo valx e calculando valy
valx = -3:0.01:3;
valy = coef_a.*valx.^3 + coef_b.*valx.^2 + ...
       coef_c.*valx + coef_d;

% Gráfico de valy em função de valx
plot(valx, valy, 'linewidth', 2);
axis([-3, 3, -10, 10]);
```

end

Suponha que já trabalhamos no prompt do Octave, e as variáveis `valx` e `valy` foram definidas fora da função, como a seguir.

```
> valx = 10;  
> valy = valx^2 + 3;  
> cubica(3, -3, 2, 2);
```

A declaração de `valx` e `valy` anterior à execução de `cubica()` não interfere em seus valores dentro da função. A função não toma conhecimento das variáveis exteriores a ela, assim como as variáveis fora da função não afetam seu funcionamento.

Dividindo funções em diferentes arquivos

Criamos algumas funções menores para dividir as tarefas de `grafico_quad()`, que poderia se tornar ilegível caso seu código fosse escrito em apenas uma função. Essas funções podem ser gravadas todas em um único arquivo, como dissemos anteriormente.

Contudo, conforme os programas ficam maiores, gravar várias funções em um mesmo arquivo pode torná-lo difícil de corrigir ou ampliar. A solução é separar as funções em vários arquivos.

No exemplo anterior, `grafico_quad()` pode ser dividido em cinco arquivos, cada um contendo uma função. Cada arquivo recebe o nome de sua função:

- O arquivo `grafico_quad.m` contém a função `grafico_quad()`.
- O arquivo `discr.m` contém a função `discr()`.
- `raizes.m` contém a função `raizes()`.
- `quadratic.a.m` contém a função `quadratic()`.
- Por fim, `parabola.m` contém a função `parabola()`.

Quando dividirmos funções em diferentes arquivos, devemos gravá-los na mesma pasta, ou adicionar sua pasta ao caminho de pesquisa (o chamado *path*). O caminho de pesquisa fará com que o Octave reconheça as funções que estiverem na pasta, independente da pasta em que ele estiver trabalhando.

Para incluir uma pasta no caminho de pesquisa do Octave, utilizamos a função `addpath()`. Considerando que a pasta que contém os arquivos está contida em `~/Documentos/Octave/` e possui o nome `quadratica_octave`, o comando para adicioná-la é:

```
> addpath('~/Documentos/Octave/quadratica_octave')
```

Se usarmos `addpath()`, a pasta é esquecida quando o Octave é encerrado, e devemos executar o comando novamente para incluir a pasta a cada vez que iniciarmos o Octave. Para que o Octave mantenha a pasta em seu caminho de pesquisa mesmo depois de encerrado, usamos a função `savepath()`:

```
> savepath('~/Documentos/Octave/quadratica_octave')
```

Caso quisermos remover uma pasta da lista de caminhos disponíveis, a função a ser utilizada é `rmpath()`:

```
> rmpath('~/Documentos/Octave/quadratica_octave')
```

Para obter uma lista dos caminhos de pesquisa inclusos no Octave, use o comando `path`. Para sair, pressione `q`.

Vamos reforçar nosso aprendizado? Resolva as tarefas a seguir:

1. Divida a função `cubica()` em duas funções menores: `calc_cubica()`, que calculará `valx` e `valy`, e `plot_cubica()`, que fará o gráfico da função cúbica correspondente. Grave-as em arquivos de função separados.
2. Crie uma função, de nome `grafico_cub()`, que coordene as funções `calc_cubica()` e `plot_cubica()`. Para isso,

- inspire-se em `grafico_quad()`. Assegure-se de que todos os arquivos de função estão na mesma pasta, ou coloque as pastas no caminho de pesquisa com o comando `addpath()`.
3. Modifique `grafico_cub()`, de forma que ela receba os pontos inicial e final de `valx` como argumento.

Unindo scripts e funções em um mesmo arquivo M

Como dissemos anteriormente, podemos unir scripts e funções em um mesmo arquivo M, o que nos possibilita testar o código sem depender do prompt do Octave.

Para exemplificar esse processo, usaremos a função `cilindro()` dada a seguir, que recebe o valor do raio e da altura de um cilindro, e calcula suas áreas da base e lateral, além do volume.

```
function [area_base, area_lateral, volume] =
    cilindro(raio, altura)
{
    [area_base, area_lateral, volume] = cilindro(raio, altura)

    CILINDRO calcula os valores das áreas da base (area_base),
    lateral (area_lateral) e do volume (volume) por meio do
    raio (raio) e da altura (altura) fornecidos como argumento.
}

% Cálculo de area_base, area_lateral e volume.
area_base = pi*raio^2;
area_lateral = 2*pi*raio*altura;
volume = area_base*altura;

% O programa exibe uma mensagem de retorno das variáveis,
% dividida em duas strings para melhor visualização.
disp('Os valores da área da base, da área lateral e');
disp('do volume desse cilindro são, respectivamente: ');

% Exibição dos valores obtidos.
disp(area_base);
disp(area_lateral);
disp(volume);
end
```

Se a função `cilindro()` for gravada em um arquivo diretamente, o Octave o interpretará como um arquivo de função. Para evitar que isso aconteça, a primeira declaração do script não pode ser a palavra `function` (lembre-se de que o Octave não considera os comentários ou espaços em branco).

Considerando a possibilidade de fornecer valores de teste para as variáveis `raio` e `altura`, uma opção é declarar esses valores e executar a função com seus argumentos antes de declará-la:

```
%{  
Script: cilindro_variaveis.m  
  
Contém a função CILINDRO, para cálculo das áreas da base,  
lateral e do volume de um cilindro. Começa com o valor das  
variáveis de teste.  
%}  
  
raio = 5;  
altura = 12;  
  
[base, lat, vol] = cilindro(raio, altura);  
  
function [area_base, area_lateral, volume] =  
    cilindro(raio, altura)  
    % Aqui vai a declaração da função CILINDRO.  
end
```

Assim, quando esse script for executado, o Octave retornará o valor da função para as variáveis definidas.

```
> cilindro_variaveis  
Os valores da área da base, da área lateral e  
do volume desse cilindro são, respectivamente:  
78.540  
376.99  
942.48
```

Quando não precisamos declarar variáveis ou chamar uma função, utilizamos uma declaração sem efeito, como a seguir:

```
%{  
Script: cilindro_declsemef.m
```

Contém a função CILINDRO, para cálculo das áreas da base, lateral e do volume de um cilindro. Começa com uma declaração sem efeito.

```
%}
```

```
% O zero abaixo não está perdido; ele previne que o Octave  
% identifique o arquivo como uma função.
```

```
0;
```

```
function [area_base, area_lateral, volume] =  
    cilindro(raio, altura)  
    % Aqui vai a declaração da função CILINDRO.  
end
```

O zero nesse código, por exemplo, não altera o funcionamento do programa e informa ao Octave que esse é um script, não um arquivo de função.

Se você considerar que o código fica poluído usando declarações sem efeito, uma alternativa é usar a função `disp()` informando o nome do script:

```
%{  
Script: cilindro_apres.m
```

```
Contém a função CILINDRO, para cálculo das áreas da base,  
lateral e do volume de um cilindro. Começa com uma apresentação.  
%}
```

```
disp('Aqui começa o script CILINDRO.');
```

```
function [area_base, area_lateral, volume] =  
    cilindro(raio, altura)  
    % Aqui vai a declaração da função CILINDRO.  
end
```

Vamos reforçar nosso aprendizado? Pratique com as atividades a seguir:

1. Modifique a função `grafico_quad()` para que ela receba as variáveis `coef_a`, `coef_b` e `coef_c` como argumentos. Então, realize testes com essa função, executando-a após atribuir diferentes valores para as variáveis. Para isso, inspire-

- se em `cilindro_variaveis.m`.
2. Da mesma forma, teste a função `grafico_cub()`: execute-a após atribuir diferentes valores às variáveis `coef_a`, `coef_b`, `coef_c` e `coef_d`.

5.4 RESUMINDO

Neste capítulo, vimos como gravar código, testá-lo e reutilizá-lo por meio de scripts e funções. Verificamos também que comentar nossos programas com instruções sobre como os desenvolvemos é uma boa forma de auxiliar na compreensão do seu código, ou de lembrar como uma ideia foi convertida em algoritmo. Nossos programas estão ainda mais elaborados, não é mesmo?

Aprendemos muito nestes capítulos, e agora é hora de ensinar nossos programas a tomar algumas decisões. Para isso, no próximo, empregaremos operadores relacionais e lógicos, estruturas condicionais e de repetição, para o controle do fluxo dos nossos códigos. Até breve!

CAPÍTULO 6

OPERADORES E ESTRUTURAS PARA CONTROLE DE FLUXO

Mesmo que nossos programas estejam se tornando mais complexos a cada capítulo, eles ainda não conseguem comparar variáveis entre si, ou tomar decisões com base em seus valores.

Neste capítulo, preencheremos essas lacunas. Tornaremos nosso código mais flexível, deixando o Octave controlar o fluxo dos programas de acordo com condições que estabeleceremos. Essas condições são dadas por operadores, que estudaremos a seguir.

6.1 OPERADORES

Operadores são conectivos que indicam uma operação entre elementos. Trabalhamos com operadores aritméticos (`+`, `-`, `*` e `/`) no capítulo *Primeiros passos*. Veremos outros dois tipos, os comparativos e os lógicos, nesta seção.

Operadores de comparação

Operadores comparativos, também chamados de *relacionais*, estabelecem comparações entre variáveis:

- **Igual** (`==`): o resultado da comparação é verdade se as

duas variáveis são iguais.

```
> igu = 4;  
> igu == 3  
ans = 0  
> igu == 2*2  
ans = 1
```

- **Diferente (!= ou ~=):** o resultado da comparação é verdade se as duas variáveis são diferentes.

```
> dif = 6;  
> dif != 3*2  
ans = 0  
> dif ~= igu  
ans = 1
```

- **Menor que (<):** o resultado da comparação é verdade se a variável à esquerda do sinal possui menor valor que o da variável à direita.

```
> meq = 3;  
> meq < dif  
ans = 1  
> igu < meq  
ans = 0
```

- **Menor que ou igual a (<=):** o resultado da comparação é verdade se a variável à esquerda do sinal possui valor menor que ou igual ao da variável à direita.

```
> mei = 5;  
> mei <= dif  
ans = 1  
> dif - meq  
ans = 3  
> mei <= (dif - meq)  
ans = 0
```

- **Maior que (>):** o resultado da comparação é verdade se a variável à esquerda do sinal possui maior valor que o da variável à direita.

```
> maq = 2;
```

```
> maq > igu
ans = 0
> 3*maq > igu
ans = 1
> (maq + dif) > 2*mei
ans = 0
> (maq + dif) > (igu + meq)
ans = 1
```

- **Maior que ou igual a (\geq):** o resultado da comparação é verdade se a variável à esquerda do sinal possui valor maior que ou igual ao da variável à direita.

```
> mai = 7;
> mai >= (meq + igu)
ans = 1
> (mai - igu) >= dif
ans = 0
> sqrt(mei) >= (mai - mei)
ans = 1
```

NOTA

`mai` , `mei` , `maq` , `meq` mostram a importância de se escolher um nome significativo para as variáveis: nomes como esses podem nos confundir!

Veja que podemos comparar variáveis, inserir elementos e fazer operações diretamente. Quando a comparação é verdadeira, o Octave retorna 1; nos casos em que a comparação é falsa, a resposta é zero.

NOTA

Lembre-se de que o operador de atribuição é representado apenas por um igual (=). Não o confunda com o operador de igualdade:

```
varsin = 2*sin(2*pi/3)
varsin = 1.7321
varcos = 2*cos(2*pi/3)
varcos = -1.00000
varsin == varcos
ans = 0
varsin = varcos
varsin = -1.00000
```

Nesse exemplo, `varsin` e `varcos` possuem valores diferentes. O operador de igualdade confirma a diferença, mas o operador de atribuição declara `varsin` sendo igual a `varcos` !

Operadores lógicos

Operadores lógicos, também conhecidos como *booleanos*, efetuam operações entre valores lógicos (verdadeiro e falso):

- **Operador E (&& ou and())**: o resultado da comparação é verdade se a expressão à esquerda e a expressão à direita são verdadeiras.

```
> (3 >= 2) && (4 <= 6)
ans = 1
> var_fac = factorial(3);
> var_exp = 3^2;
> (var_exp != var_fac) && (var_exp == 3 + var_fac)
ans = 1
> (3 > var_fac) && (var_exp <= var_fac)
ans = 0
```

- **Operador OU (|| ou or())**: o resultado da comparação é verdade se a expressão à esquerda *ou* a expressão à direita são verdadeiras.

```
> (1 == 2) || (9 > 7)
ans = 1
> (10 < 3) || (2 == 2)
ans = 1
> (3 > var_fac) || (var_exp <= 5)
ans = 0
```

- **Operador OU EXCLUSIVO (xor())**: o resultado da comparação é verdade se apenas um dos argumentos é verdadeiro: *ou* o primeiro, *ou* o segundo.

```
> xor(1 == 2, 9 > 7)
ans = 1
> xor(10 > 2, 5 > 3)
ans = 0
> var_sin = sin(0);
> var_cos = cos(0);
> xor(var_sin == 0, var_cos > 1)
ans = 1
```

- **Operador NÃO (~ , ! ou not())**: é a negação da expressão. Se a expressão é verdadeira, o resultado é falso; se a expressão é falsa, o resultado é verdadeiro.

```
> !(1 == 2)
ans = 1
> ~(3*exp(2) < 10)
ans = 1
> not((var_exp != var_fac) && (var_exp == 3 + var_fa
c))
ans = 0
```

NOTA

Quando os operadores são usados para comparar vetores ou matrizes, eles agem elemento a elemento:

```
A = [1 4 7];
B = [2 4 8];
A == B
ans =
    0   1   0
A != B
ans =
    1   0   1
not(A != B)
ans =
    0   1   0
A < B
ans =
    1   0   1
```

Sendo assim, não é possível comparar elementos de diferentes tamanhos:

```
C = [1 2 2 1];
A == C
error: mx_el_eq: nonconformant arguments
        (op1 is 1x3, op2 is 1x4)
```

Com base nos operadores, podemos utilizar estruturas de controle para executar diferentes partes do código de acordo com a resposta que recebermos de expressões definidas pelos operadores, controlando o fluxo dos nossos programas. Essas estruturas são apresentadas na seção seguinte.

6.2 ESTRUTURAS DE CONTROLE

As estruturas de controle coordenam o andamento do programa, repetindo e executando partes específicas do código, de

acordo com o resultado de expressões criadas com o auxílio dos operadores. Vamos a elas.

A estrutura se

Uma das estruturas que podemos utilizar no Octave é a *se*, dada pela instrução `if` seguida de uma expressão:

```
> if expressao  
>   bloco_de_comandos;  
> end
```

Neste código, se `expressao` for verdadeira, os comandos de `bloco_de_comandos` serão executados. Caso `expressao` for falsa, o Octave não executa as instruções até `end` e continua o programa. Observe que `bloco_de_comandos` pode conter várias instruções e ocupar quantas linhas precisarmos.

Suponha que precisamos desenvolver um sistema para obter e analisar estatisticamente as idades de um grupo de pessoas. O código a seguir usa a função `input()` para verificar a idade de um usuário, e dizer se a pessoa está no grupo de pesquisa:

```
> maior_idade = 18;  
> idade_usuario = input('Digite sua idade: ');\n> if idade_usuario >= maior_idade  
>   disp('Usuário entra no grupo!')  
> end
```

Repare que o programa retorna informação ao usuário apenas se ele entrar com um número maior ou igual que 18. Para o sistema retornar uma resposta ao usuário que entrar com uma idade menor que 18 anos, utilizaremos a estrutura *se não*, por meio da instrução `else`. Essa instrução deve ser incluída no corpo de `if`:

```
> if expressao  
>   bloco_de_comandos;  
> else  
>   outro_bloco;  
> end
```

Se `expressao` for verdadeira, o Octave executará os comandos em `bloco_de_comandos`. Se não, `outro_bloco` será executado.

Adaptando o código anterior, conseguimos estabelecer quais são os usuários menores de idade:

```
> maior_idade = 18;
> idade_usuario = input('Digite sua idade: ');
> if idade_usuario >= maior_idade
>   disp('Usuário entra no grupo!')
> else
>   disp('Usuário fora do grupo. Desculpe!')
> end
```

Se nosso sistema precisar obter apenas pessoas entre 18 e 60 anos, serão necessárias mais condições para separar os usuários de interesse. Além de `if` e `else`, usaremos a estrutura *se não, se* para dividir o grupo. Ela é dada por meio da instrução `elseif`. Essa instrução também deve ser incluída no corpo de `if`:

```
> if expressao
>   bloco_de_comandos;
> elseif outra_expressao
>   outro_bloco;
> else
>   outro_bloco_ainda;
> end
```

Nesse código, se `expressao` for verdadeira, o Octave executará os comandos em `bloco_de_comandos`; se não, se `outra_expressao` for verdadeira, `outro_bloco` será executado. Se nem `expressao` nem `outra_expressao` forem verdadeiras, as instruções executadas serão as contidas em `outro_bloco_ainda`.

Adaptando novamente o código, conseguimos filtrar o grupo de interesse:

```
> limite_baixo = 18;
> limite_alto = 60;
> idade_usuario = input('Digite sua idade: ');
> if idade_usuario < limite_baixo
>   disp('Usuário fora do grupo (limite menor). Desculpe!');
```

```

> elseif idade_usuario > limite_alto
>   disp('Usuário fora do grupo (limite maior). Desculpe!');
> else
>   disp('Usuário entra no grupo!')
> end

```

Agora um desafio pra você! Crie uma função baseada nesse código, de forma que a idade do usuário seja passada a ela como um argumento.

Trabalhamos com a função `quad()`, que encontra as raízes para uma equação do segundo grau, no capítulo anterior. Seu código é dado novamente a seguir, sem o cabeçalho dessa vez.

```

function [raiz_1, raiz_2] = quad(val_a, val_b, val_c)
    delta = val_b^2 - 4*val_a*val_c;
    raiz_1 = (-val_b + sqrt(delta))./(2*val_a);
    raiz_2 = (-val_b - sqrt(delta))./(2*val_a);
end

```

Essa função não verifica a entrada do usuário, o que pode conduzir a erros quando não informamos o número correto de argumentos. Por exemplo, imagine que esquecemos de digitar a variável `val_c`:

```

> [x1, x2] = quad(2, -4)
error: 'val_c' undefined near line 2 column 31
error: called from:
error: quad at line 2, column 11

```

Para resolver esse problema, usaremos `if` em conjunto com duas funções: uma é a instrução `nargin`, que retorna o número de argumentos recebidos por uma função; a outra se chama `usage('como_usa')`, e apresenta a string `como_usa` ao usuário.

A função `quad()` corrigida, `quadnargin()`, fica assim:

```

function [raiz_1, raiz_2] = quadnargin(val_a, val_b, val_c)
    %
    [raiz_1, raiz_2] = quadnargin(val_a, val_b, val_c)

QUADNARGIN encontra as raízes (ou zeros) X1 (raiz_1) e X2
(raiz_2) da equação quadrática A*X^2 + B*X + C = 0, pela

```

```

fórmula de Bhaskara. Confere a entrada do usuário por meio
de nargin.

}

if (nargin != 3)
    usage('quadnargin (val_a, val_b, val_c)');
end

delta = val_b^2 - 4*val_a*val_c;
raiz_1 = (-val_b + sqrt(delta))./2*val_a;
raiz_2 = (-val_b - sqrt(delta))./2*val_a;
end

```

Utilizamos 'quadnargin (val_a, val_b, val_c)' como argumento de `usage()`. Caso a função `quadnargin()` seja chamada com um número de argumentos diferente do necessário, o erro apresentado pelo Octave mostra o argumento que definimos:

```

> [x1, x2] = quadnargin(2, -4)
usage: quadnargin (val_a, val_b, val_c)
error: quadnargin at line 3, column 9

```

Vamos reforçar nosso aprendizado? Utilize `if` nas situações a seguir:

1. No capítulo *Primeiros passos*, falamos que o fatorial de um número não pode ser negativo. Crie uma função de nome `fator()` que receba um número como argumento do usuário. Caso esse número for negativo, obtenha o fatorial de seu valor absoluto (função `abs()`).
2. Corrija a função `cubica()`, do capítulo *Gravando e reaproveitando código*, para que ela não aceite menos que quatro argumentos. Inspire-se em `quadnargin()` para resolver esse problema.
3. Crie uma função de nome `grafico_poli()`. Caso essa função receba três argumentos, ela fará um gráfico de uma função do segundo grau utilizando `grafico_quad()`. Se ela receber quatro argumentos, o gráfico gerado será o de uma função do terceiro grau, por meio de `grafico_cub()`. Por

sua vez, se o número de argumentos for diferente de três ou quatro, a função retornará uma mensagem sobre seu uso (`usage()`). Consulte o capítulo *Gravando e reaproveitando código* para mais detalhes sobre `grafico_quad()` e `grafico_cub()`.

A estrutura switch

Nosso sistema de classificação de grupos funciona bem para poucas restrições, mas imagine que precisamos classificar os elementos em cinco grupos entre 18 e 60 anos:

- **Grupo 1:** entre 18 e 21 anos;
- **Grupo 2:** entre 22 e 30 anos;
- **Grupo 3:** entre 31 e 40 anos;
- **Grupo 4:** entre 41 e 50 anos;
- **Grupo 5:** entre 51 e 60 anos.

Iríamos precisar de vários `elseif` para dividir esses grupos! Entretanto, nesses casos é melhor usar a estrutura `switch`, representada pela instrução `switch`:

```
> switch expressao
>   case cond_1
>     bloco_comandos;
>   case cond_2
>     outro_bloco;
>   case cond_3
>     mais_um_bloco;
>   ...
>   otherwise
>     outro_blocoainda;
> end
```

Nessa forma, `switch` recebe uma expressão ou variável em `expressao`. Cada caso (`case`) possui suas condições para ser executado, como um `elseif`. Os blocos de comandos abaixo de cada `case` serão executados se `expressao` for igual à condição do

caso. Se nenhuma das condições for satisfeita, o bloco abaixo de `otherwise` é executado.

É mais fácil manter o código com `switch`, pois as repetições são mais explícitas e legíveis. A diferença é que `switch` recebe as condições (`cond_1`, `cond_2`, `cond_3` etc.) como um tipo de dados nomeado *vetor celular*, com contêineres de dados indexados (as células).

Por exemplo, a função a seguir foi modificada de um programa disponível na wiki "Aprender GNU Octave", de Jorge Rocha, da Universidade do Minho, Portugal (<http://goo.gl/uQ2h1M>).

Os vetores celulares desse código são `{'a' 'A'}` , `{'c' 'C'}` , `{'g' 'G'}` e `{'t' 'T'}` . O usuário entra com uma letra e `switch` compara a entrada com os vetores de cada `case` .

```
function bases(letra_base)
%
bases(letra_base)

BASES recebe uma letra do usuário e retorna o nome da base
correspondente.

Modificado de:
http://octave.di.uminho.pt/index.php/as\_condições\_switch
%}

letra_base = input('Indique a letra da base [ACGT]: ', 's');
switch letra
    case {'a' 'A'}
        disp('Adenina');
    case {'c' 'C'}
        disp('Citosina');
    case {'g' 'G'}
        disp('Guanina');
    case {'t' 'T'}
        disp('Timina');
    otherwise
        disp('Não existe nenhuma base correspondente!');
end
```

A fim de usar `switch` em nosso sistema de classificação de grupos, precisamos transformar vetores comuns em celulares. Para isso, usaremos a função `num2cell()`. Esse código classifica a idade do usuário nos cinco grupos e ainda desconsidera a entrada se ela não estiver entre 18 e 60. Veja a diferença desse código para a solução anterior:

```
> idade_usuario = input('Digite sua idade: ');
> switch (idade_usuario)
>   case num2cell([18:21])
>     disp('Usuário no Grupo 1');
>   case num2cell([22:30])
>     disp('Usuário no Grupo 2');
>   case num2cell([31:40])
>     disp('Usuário no Grupo 3');
>   case num2cell([41:50])
>     disp('Usuário no Grupo 4');
>   case num2cell([51:60])
>     disp('Usuário no Grupo 5');
>   otherwise
>     disp('Usuário não cumpre requerimentos. Desculpe!');
> end
```

Ainda não acabou! Para terminarmos essa seção, falta a sua ajuda. Com base nessa solução, crie uma função nomeada `classifica_grupos()`. Faça com que ela receba a idade do usuário como argumento. Você pode consultar uma das soluções possíveis em <http://goo.gl/O3o7Xz>.

A estrutura para

Como as estruturas `if` e `switch` executam porções de código específicas de acordo com as condições, costumamos chamá-las de *estruturas condicionais*. Vejamos outro tipo de estruturas, as *estruturas de repetição*.

Uma das estruturas de repetição é a declaração *para*, dada pela instrução `for`:

```
> for iter = intervalo
```

```
> bloco_comandos;  
> end
```

Nessa forma, `for` indica que os comandos seguintes serão repetidos até que `iter` percorra a variável `intervalo`. Por exemplo, para imprimir o seno dos números inteiros de 1 até 10, fazemos:

```
> for iter = 1:10  
> disp(sin(iter))  
> end
```

A variável `iter`, nesse caso, é conhecida como *variável de iteração*. Na primeira passagem pelo `for`, seu valor é 1. Depois, a instrução seguinte é executada: `disp(sin(1))`. Após o comando, o programa volta ao `for`, e `iter` assume o valor 2. Então, a instrução `disp(sin(2))` é executada. O programa volta ao `for` novamente, até que a condição seja satisfeita: `iter = 10`.

Conhece a sequência de Fibonacci? O primeiro e o segundo termo são 1, e cada termo a partir do terceiro corresponde à soma dos dois anteriores: 1, 1, 2, 3, 5, e assim por diante.

A revista Mundo Estranho produziu uma matéria sobre essa sequência, mostrando também exemplos em que ela aparece na natureza. Veja em <http://www.mundoestranho.com.br/materia/o-que-e-a-sequencia-de-fibonacci>.

Com a ajuda da estrutura `for`, construiremos uma função para gerar quantos termos da sequência de Fibonacci desejarmos:

```
function seq = fibonacci(num_termos)  
%  
seq = fibonacci(num_termos)  
  
FIBONACCI gera a sequência de Fibonacci para o número de  
termos num_termos.  
%}  
  
if (nargin != 1)  
    usage('fibonacci (num_termos)');
```

```

    end
    seq = ones(1, num_termos);
    for termo = 3:num_termos
        seq(termo) = seq(termo-1) + seq(termo-2);
    end
end

```

Primeiro usamos a função `ones()` para criar um vetor com várias cópias do número 1. A quantidade de termos desse vetor é dada por `num_termos`. Note que `nargin` está presente, verificando se o usuário forneceu o argumento necessário. Depois, utilizamos `for` para gerar os termos da sequência a partir do terceiro, uma vez que o primeiro e o segundo termos não precisam ser modificados. Veja que `for` usa a variável `termos` para as iterações, que começa com o valor 3 e vai até `num_termos`.

Para executar essa função, digitamos `fibonacci(num_termos)`, em que `num_termos` é o número de termos desejado:

```

> fibonacci(10)
ans =
    1     1     2     3     5     8    13    21    34    55

```

Agora, incrementaremos nosso sistema de classificação de grupos. Em vez de receber a idade de uma pessoa apenas, podemos usar `for` para obter a entrada do usuário quantas vezes quisermos. Supondo que o grupo estudado possui oito pessoas, podemos criar um vetor para armazenar suas idades usando a função `zeros()`:

```

> tam_grupo = 8;
> idade_grupo = zeros(1, tam_grupo);
> for elem = 1:tam_grupo
>     idade_grupo(elem) = input('Digite sua idade: ');
> end

```

Veja que utilizamos `idade_grupo = zeros(1, tam_grupo)` para criar uma matriz de uma linha. A quantidade de colunas é dada pela variável `tam_grupo`; assim, caso o grupo mude, alteraremos apenas o valor dessa variável.

A cada iteração, `elem` muda seu valor e `idade_grupo` avança para o próximo índice, gravando cada idade digitada pelo usuário em um índice diferente.

Integrando esse código com `switch`, criamos a função `classgruposid()`, que recebe a idade desse grupo de pessoas e as classifica nos grupos que definimos anteriormente:

```
function classgruposid()
{
    classgruposid()

    CLASSGRUPOSID() recebe as idades de um grupo de pessoas com
    tamanho predefinido (tam_grupo) e as classifica em cinco
    grupos distintos utilizando a estrutura switch.

}

tam_grupo = 8;
idade_grupo = zeros(1, tam_grupo);
for elem = 1:tam_grupo
    idade_grupo(elem) = input('Digite sua idade: ');
    switch (idade_grupo(elem))
        case num2cell([18:21])
            disp('Usuário no Grupo 1');
        case num2cell([22:30])
            disp('Usuário no Grupo 2');
        case num2cell([31:40])
            disp('Usuário no Grupo 3');
        case num2cell([41:50])
            disp('Usuário no Grupo 4');
        case num2cell([51:60])
            disp('Usuário no Grupo 5');
        otherwise
            disp('Usuário não cumpre requerimentos.
                  Desculpe!');
    end
end
end
```

Um exemplo de execução desse programa é dado a seguir. Note que as idades fornecidas durante a execução foram 4, 18, 32, 7, 35, 42, 31 e 48.

```
> classgruposid()
Digite sua idade: 4
```

```
Usuário não cumpre requerimentos. Desculpe!
Digite sua idade: 18
Usuário no Grupo 1
Digite sua idade: 32
Usuário no Grupo 3
Digite sua idade: 7
Usuário não cumpre requerimentos. Desculpe!
Digite sua idade: 35
Usuário no Grupo 3
Digite sua idade: 42
Usuário no Grupo 4
Digite sua idade: 31
Usuário no Grupo 3
Digite sua idade: 48
Usuário no Grupo 4
```

Agora é a sua vez! Quando `classgruposid()` pedir uma idade, experimente inserir uma string, como '`'dezoito'`'. O que acontece? Como poderíamos contornar essa situação, usando `if`? Uma dica é usar a função `isfloat(var)`, que retorna verdadeiro se `var` é um número real (número em ponto flutuante).

NOTA

Além de `isfloat()`, o Octave possui diversas funções que verificam o tipo de uma variável em questão. Uma lista completa dessas funções está em <https://www.gnu.org/software/octave/doc/interpreter/Predicates-for-Numeric-Objects.html>.

A estrutura enquanto

Outra estrutura de repetição é *enquanto*, representada pela instrução `while`:

```
> while condicao
>   bloco_de_comandos;
> end
```

Nessa forma, `while` continua repetindo os comandos em `bloco_de_comandos` enquanto `condição` seja verdadeira.

Para exemplificar a aplicação de `while`, criaremos a função `seriegeomwhile()`, que calcula o valor da soma dos primeiros termos da famosa série geométrica $1/2 + 1/4 + 1/8 + 1/16 + \dots$. Uma série geométrica é uma soma cujos termos sucessivos possuem razão constante entre si.

```
function soma = seriegeomwhile(termos)
{
    soma = seriegeomwhile(termos)

    SERIEGEOMWHILE calcula a soma da série geométrica
    1/2 + 1/4 + 1/8 + 1/16 + ... utilizando do
    e recebe como argumento a quantidade de termos
    a serem somados, retornando o valor da soma.
}

soma = 0;
cont = 1;
while (cont != termos)
    soma = soma + (1/2)^cont;
    cont++;
end
end
```

As variáveis `soma` e `cont` armazenam o valor da soma e o número do termo atual, respectivamente. Enquanto `cont` não for igual a `termos`, o número de termos que desejamos, `soma` recebe o resultado da soma até então. Depois passamos ao termo seguinte, somando 1 à variável `cont`. Isso é feito com o comando `cont++`.

Utilizaremos `while` para criar a função `classgruposid2()` e melhorar nosso sistema de classificação, tomando as idades de usuários sem estipular um tamanho fixo para o grupo:

```
function idade_grupo = classgruposid2()
{
    idade_grupo = classgruposid2()

    CLASSGRUPOSID2() retorna as idades de um grupo
```

```

de pessoas (idade_grupo) até que receba zero e
as classifica em cinco grupos distintos,
utilizando a estrutura switch.
%}

idade_grupo = -1;
while (idade_grupo(end) != 0)
    idade_grupo(end+1) = input('Digite sua idade: ');
    switch (idade_grupo(end))
        case num2cell([18:21])
            disp('Usuário no Grupo 1');
        case num2cell([22:30])
            disp('Usuário no Grupo 2');
        case num2cell([31:40])
            disp('Usuário no Grupo 3');
        case num2cell([41:50])
            disp('Usuário no Grupo 4');
        case num2cell([51:60])
            disp('Usuário no Grupo 5');
        otherwise
            disp('Usuário não cumpre requerimentos.
                  Desculpe!');
    end
end
disp('Fim do processamento!')
end

```

Agora o programa retorna as idades recebidas por meio da variável `idade_grupo`. Tivemos de iniciá-la para que `while` fosse executado pela primeira vez.

Para que a variável não interfira nas idades registradas ou não encerre o programa sem receber idades, ela recebe o valor -1. Atribua a ela o valor inicial zero e veja o que acontece quando você executar novamente a função.

Novos elementos foram adicionados utilizando o índice `idade_grupo(end+1)`, para que não precisássemos definir o tamanho total do vetor. Lembre-se de que `end` é o índice do último elemento do vetor, como visto no capítulo *Primeiros passos*. Quando o programa receber zero como `idade`, ele exibe uma mensagem dizendo que terminou o processamento.

Executaremos o novo programa usando as idades fornecidas a `classgruposid()` durante sua execução de exemplo: 4, 18, 32, 7, 35, 42, 31 e 48.

```
> idades = classgruposid2();
Digite sua idade: 4
Usuário não cumpre requerimentos. Desculpe!
Digite sua idade: 18
Usuário no Grupo 1
Digite sua idade: 32
Usuário no Grupo 3
Digite sua idade: 7
Usuário não cumpre requerimentos. Desculpe!
Digite sua idade: 35
Usuário no Grupo 3
Digite sua idade: 42
Usuário no Grupo 4
Digite sua idade: 31
Usuário no Grupo 3
Digite sua idade: 48
Usuário no Grupo 4
Digite sua idade: 0
Usuário não cumpre requerimentos. Desculpe!
Fim do processamento!
```

Note que quando informamos zero como idade para encerrar o programa, ele classificou como uma possível entrada nos grupos. Vejamos a variável `idades` :

```
> idades
idades =
-1    4    18    32     7    35    42    31    48     0
```

Os valores -1 e zero, além dos que não se encaixam nos cinco grupos de interesse, poderiam prejudicar nossos cálculos. Vamos melhorar ainda mais o programa, com base na estrutura que apresentaremos a seguir.

A estrutura *faça até que*

Por fim, há a estrutura *faça até que*, dada pelas instruções `do` e `until` :

```
> do  
>   bloco_de_comandos  
> until (condicao)
```

As estruturas `do-until` e `while` são bastante similares, mas há diferenças importantes entre as duas:

- As condições de `while` e `do-until` ficam no começo e no fim da estrutura, respectivamente.
- `while` continua enquanto a condição for verdadeira; por sua vez, `do-until` continua enquanto a condição for *falsa*.
- O bloco `do-until` é executado ao menos uma vez, pois sua condição está no fim da estrutura; entretanto, se a condição de `while` é falsa, ele não é executado.

Para mostrar essas diferenças, recriaremos a função `seriegeomwhile()` usando `do-until` no lugar de `while`.

```
function soma = seriegeomdo(termos)  
%  
soma = seriegeomdo(termos)  
  
SERIEGEOMDO calcula a soma da série geométrica  
1/2 + 1/4 + 1/8 + 1/16 + ... utilizando do  
e recebe como argumento a quantidade de termos  
a serem somados, retornando o valor da soma.  
%}  
  
soma = 0;  
cont = 1;  
do  
    soma = soma + (1/2)^cont;  
    cont++;  
until (cont == termos)  
end
```

Observe que a condição de parada fica ao fim do bloco, e o bloco continua sendo executado até que `cont` seja igual a `termos`. Na função `seriegeomwhile()`, a condição é contrária: `cont != termos`.

Imagine que, por descuido, trocamos a condição. Trabalharemos apenas com uma parte do código da função para exemplificar, e faremos `termos = 20`.

```
> soma = 0;
> cont = 1;
> termos = 20;
> do
>   soma = soma + (1/2)^cont;
>   cont++;
> until (cont != termos) % condição trocada por descuido!
> disp('Soma: '); disp(soma);
Soma:
0.50000
```

Pelas funções anteriores, vimos que a soma dessa série é 1. Como `do-until` será executado *até que* `cont = 1` seja diferente de `termos = 20`, ele para na primeira execução, somando apenas o primeiro termo!

Trabalharemos na última versão do nosso sistema de classificação usando `do-until`. Além disso, implementaremos mais algumas modificações:

1. Como `do-until` só confere a condição ao final, podemos iniciar `idade_grupo` como um vetor em branco:
`idade_grupo = []`.
2. Separaremos os elementos de `idade_grupo` nos cinco grupos de interesse utilizando a estrutura `grupos.idade`: cada índice de `grupos` representará um grupo.
3. Colocaremos as instruções `for` e `if` em conjunto para apagar as entradas que não queremos em `idade_grupo`: números menores que 18 e maiores que 60.

Essas mudanças estão implementadas na função `classgruposidfinal()`:

```
function [idade_grupo, grupos] = classgruposidfinal()
{
```

```

[idade_grupo, grupos] = classgruposidfinal()

CLASSGRUPOSIDFINAL() retorna as idades de um grupo
de pessoas (idade_grupo) até que receba zero e
as classifica em cinco grupos distintos,
utilizando a estrutura switch.
%}

idade_grupo = [];
for i = 1:5
    grupos(i).idade = [];
end

do
    idade_grupo(end+1) = input('Digite sua idade: ');
    switch (idade_grupo(end))
        case num2cell([18:21])
            disp('Usuário no Grupo 1');
            grupos(1).idade(end+1) = idade_grupo(end);
        case num2cell([22:30])
            disp('Usuário no Grupo 2');
            grupos(2).idade(end+1) = idade_grupo(end);
        case num2cell([31:40])
            disp('Usuário no Grupo 3');
            grupos(3).idade(end+1) = idade_grupo(end);
        case num2cell([41:50])
            disp('Usuário no Grupo 4');
            grupos(4).idade(end+1) = idade_grupo(end);
        case num2cell([51:60])
            disp('Usuário no Grupo 5');
            grupos(5).idade(end+1) = idade_grupo(end);
        otherwise
            disp('Usuário não cumpre requisitos.
                  Desculpe!');
    end
until (idade_grupo(end) == 0)

% Mostrando o número total de entradas.
disp('Número total de usuários: '); disp(numel(idade_grupo));

% Desconsiderando idades menores que 18 e maiores que 60.
for iter = numel(idade_grupo):-1:1
    if (idade_grupo(iter) < 18 || idade_grupo(iter) > 60)
        idade_grupo(iter) = [];
    end
end
disp('Fim do processamento!')
end

```

Veja que `idade_grupo` e `grupos` foram criadas com elementos vazios, e novos elementos foram adicionados utilizando o índice `idade_grupo(end+1)`. Repare também que `for` foi usado *ao contrário*; começamos pelo último índice de interesse e fomos descendo um a um. Veja o efeito no exemplo, que mostra os números a partir de 5 e vai até 1:

```
> for iter = 5:-1:1  
> disp(iter)  
> end  
5  
4  
3  
2  
1
```

Faça uma execução de exemplo dessa função, utilizando a linha de código a seguir:

```
> [idade, grupo] = classgruposidfinal();
```

As idades para você testar são 4, 18, 32, 14, 42, 60, 80, 51, 35, 26, 40, 45, 30, 60, 15, 18, 21, 43, 42 e zero, para encerrar o programa. A partir dessa execução, teremos o vetor `idade` sem os elementos que não se encaixam nos grupos definidos, e a estrutura `grupo : grupo` :

```
> idade  
idade =  
    18    32    42    60    51    35    26    40    45  
30    60    18    21    43    42  
> grupo.idade  
ans =  
    18    18    21  
ans =  
    26    30  
ans =  
    32    35    40  
ans =  
    42    45    43    42  
ans =  
    60    51    60
```

Note que `grupo.idade` retorna cinco `ans`, referentes aos cinco grupos separados. Para acessar cada grupo, devemos usar seu índice: `grupo(1).idade`, `grupo(2).idade`, e assim por diante.

Para verificar a quantidade de pessoas em cada grupo, usamos `numel()`. Por exemplo, para os grupos 1 e 4:

```
> numel(grupo(1).idade)
ans = 3
> numel(grupo(4).idade)
ans = 4
```

Vamos reforçar nosso aprendizado? Estude com as tarefas a seguir:

1. Experimente usar `for` de forma crescente em `classgruposidfinal()` e perceba os erros que isso pode causar. Faça um teste com essa função depois de substituir sua linha `for` por essa:

```
for iter = 1:numel(idade_grupo)
```

2. Utilizando as idades dadas no exemplo, crie um histograma a partir da quantidade de pessoas em cada grupo. Antes, lembre-se da função `hist()` no capítulo *Produzindo gráficos no Octave*.

As estruturas `quebre` e `continue`

As instruções `for`, `while` e `do-until` possuem ainda as estruturas auxiliares *quebre* e *continue*, dadas pelas instruções `break` e `continue`, respectivamente. Essas instruções são usadas apenas dentro dos blocos `for`, `while` ou `do-until`.

A instrução `break` determina que o bloco `for`, `while` ou `do-until` em que ela está contida seja interrompido, e o programa `continue`.

Para exemplificar `break`, criamos a função `menordivisor(num)`. Ela responde se `num` é um número primo (divisível apenas por 1 e por ele mesmo); caso contrário, ela retorna seu menor divisor. Essa função foi adaptada de um exemplo do manual do Octave, que pode ser consultado em <http://www.gnu.org/doc/octave/The-break-Statement.html>.

```
function div = menordivisor(num)
{
    div = menordivisor(num)

    MENORDIVISOR() retorna o menor divisor (div) de
    um número, se ele não for primo.
}

div = 2;
while (power(div,2) <= num)
    if (rem(num, div) == 0)
        break;
    end
    div++;
end
if (rem (num, div) == 0)
    disp('Menor divisor: '); disp(div)
else
    disp('O número é primo!');
    div = 1;
end
end
```

Repare que para verificar se o número informado é primo, `menordivisor()` usa a função `rem()`, que retorna o resto da divisão entre dois números. Enquanto o resto da divisão não é zero, a condição em `if` não é satisfeita e o bloco `while` continua sendo executado.

Por outro lado, quando o resto da divisão entre `num` e `div` é igual a zero, sabemos que `div` é divisor de `num`. Então `num` não é primo, e o comando `break` "quebra" o bloco `while` para informar seu menor divisor.

Diferentemente de `break`, a instrução `continue` não

abandona o bloco em que estiver contida. Ela apenas salta o restante dos comandos do bloco, forçando uma nova execução imediata dos comandos.

A seguir, há o código da função `multitres()`, exemplificando a instrução `continue`. Ela constrói um vetor de números inteiros aleatórios usando `rand()` em conjunto com `round()`, que retorna o número inteiro mais próximo.

Por fim, `multitres()` exibe os números nesse vetor que são múltiplos de 3, utilizando a função `rem()`. Ela foi adaptada de um exemplo da documentação online do Octave, disponível em <http://www.gnu.org/software/octave/doc/interpreter/The-continue-Statement.html>.

```
function vetor = multitres()
{
    vetor = multitres()

    MULTITRES() gera um vetor de 30 números aleatórios
    e exibe quais deles são múltiplos de 3.
}

vetor_rand = round(rand(1, 30) * 100);
disp('Múltiplos de 3: ');

for mult = vetor_rand
    if (rem(mult, 3) != 0)
        continue;
    end
    disp(mult);
end
end
```

Repare que quando o resto da divisão do número por 3 é diferente de zero (indicando que o número não é múltiplo de 3), `continue` diz ao bloco para recomeçar, desprezando a linha `disp(mult)`.

Agora é a sua vez! Resolva os exercícios:

1. Adapte a função `classgruposidfinal()` com um `break`, que deixa o laço `do-until` quando recebe uma idade igual a zero.
2. Crie uma função de nome `eounaoeprimo()` , que receba números do usuário e imprima a mensagem `Esse aqui é primo!` quando o número for primo. Para isso, utilize `continue`. A função deve encerrar quando receber o valor zero.

6.3 RESUMINDO

Neste capítulo, aprendemos a lidar com operadores e estruturas, ferramentas muito úteis para criarmos programas que tomem decisões de acordo com condições estabelecidas por nós. Vimos como criar expressões com operadores lógicos e de comparação, além de como utilizar essas expressões para alterar a progressão dos nossos códigos por meio das estruturas condicionais e de repetição.

No próximo capítulo, conheceremos o Octave-Forge, uma coleção de pacotes para funções específicas no Octave. Aprenderemos a instalar pacotes, veremos uma lista de ferramentas disponíveis e prepararemos o Octave para trabalhar com elas. Nos vemos lá!

CAPÍTULO 7

OCTAVE-FORGE: MAIS PODER PARA SEU OCTAVE

A biblioteca de funções do Octave é enorme, com muitas funções disponíveis. Entretanto, pode ser que precisemos de programas que tratem de questões mais específicas, como resolver uma equação diferencial ou retirar ruído de uma imagem, por exemplo.

Pensando em resolver tais problemas, por vezes recriamos funções que já estão implementadas. Para não desperdiçarmos tempo e esforço quando precisarmos de algumas dessas funções, podemos recorrer ao Octave-Forge.

O Octave-Forge, ou apenas Forge, disponibiliza uma coleção de pacotes que fornece mais funcionalidade para o Octave em diversas áreas, como estatística, processamento de sinais e imagens, lógica fuzzy, programação paralela, entre outras.

7.1 INSTALANDO E REMOVENDO PACOTES DO OCTAVE-FORGE

Para usar um pacote do Octave-Forge, é necessário instalá-lo por meio do comando `pkg install` :

```
> pkg install -forge nomedopacote
```

Nesse caso, `nomedopacote` é o nome do pacote que desejamos

instalar; os pacotes disponíveis no Forge estão presentes na seção *Pacotes disponíveis no Octave-Forge*.

Durante a instalação, o Octave se conecta ao banco de dados do Forge e baixa o pacote correspondente. A instalação do pacote `signal`, por exemplo, ocorre da seguinte maneira:

```
> pkg install -forge signal  
For information about changes from previous versions of the  
signal package, run 'news signal'.
```

Ao fim desse comando, o pacote `signal` estará instalado no Octave. Como visto nesse exemplo, o comando `news` nomeado do pacote informa as mudanças entre a versão disponível e as versões anteriores. Para listar todos os pacotes instalados, usamos o comando `pkg list`.

NOTA

Para instalar alguns pacotes do Octave-Forge no Linux, é necessária a instalação da biblioteca de desenvolvimento do Octave.

Em distribuições derivadas do Debian, essa biblioteca recebe o nome de `liboctave-dev`, e pode ser instalada pelo seu gerenciador de pacotes favorito ou pelo terminal. Nesse último caso, usamos o comando a seguir:

```
$ sudo apt-get install liboctave-dev
```

Lembre-se de que o cífrão representa o prompt do terminal Linux e não deve ser digitado.

Alguns pacotes do Forge dependem de outros pacotes. Por exemplo, o pacote `bim` possui os pacotes `fpl` e `msh` como

dependências:

```
> pkg install -forge bim  
error: the following dependencies were unsatisfied:  
  bim needs fpl >= 0.0.0  
  bim needs msh >= 0.0.0
```

Nesse caso, devemos instalar as dependências `fpl` e `msh` junto com o pacote desejado:

```
> pkg install -forge bim fpl msh  
error: the following dependencies were unsatisfied:  
  msh needs splines >= 0.0.0
```

Além das dependências de `bim`, essa instrução mostra que o pacote `msh` depende de `splines`. Vamos incluí-lo na instalação:

```
> pkg install -forge bim fpl msh splines  
configure: WARNING: dolfin headers could not be found, some  
functionalities will be disabled, don't worry your package  
will still be working, though.  
For information about changes from previous versions of the  
splines package, run 'news splines'.
```

Ao fim da instalação, o pacote `bim` e suas dependências estarão prontas para o uso. As mensagens de aviso são referentes aos pacotes e geralmente não indicam problemas.

Após instalar um pacote do Forge, precisamos carregá-lo no Octave antes de poder utilizá-lo. Para isso, empregamos o comando `pkg load`:

```
> pkg load nomedopacote
```

Para carregar o pacote `bim`, por exemplo, digitamos:

```
> pkg load bim
```

Todas as dependências do pacote são carregadas automaticamente.

Para carregar todos os pacotes do Forge já instalados, usamos o comando a seguir:

```
> pkg load all
```

NOTA

Para que um pacote seja carregado quando o Octave iniciar, de forma que não precisemos usar `pkg load nomedopacote`, podemos instalá-lo com a opção `-auto`:

```
pkg install -forge -auto nomedopacote
```

Para liberar um pacote que foi carregado no Octave, utilizamos o comando `pkg unload`:

```
> pkg unload nomedopacote
```

Por sua vez, o comando a seguir libera todos os pacotes do Forge carregados pelo comando `pkg load`:

```
> pkg unload all
```

Por fim, para desinstalar um pacote, usamos o comando `pkg uninstall`:

```
> pkg uninstall nomedopacote
```

Para desinstalar o pacote `signal`, por exemplo, o comando é:

```
> pkg uninstall signal
```

Agora que sabemos como instalar e desinstalar pacotes do Forge, vejamos quais são os pacotes disponíveis e suas aplicações.

7.2 PACOTES DISPONÍVEIS NO OCTAVE-FORGE

Nesta seção, listaremos os pacotes disponíveis no Octave-Forge, indicando também em que situação cada um deles pode ser usado.

A lista de funções de cada pacote está disponível no endereço <http://octave.sourceforge.net/pacote/overview.html>. No endereço, substitua pacote pelo nome do pacote desejado.

B

- **bim:** esse pacote contém funções para resolver equações diferenciais parciais de advecção-difusão-reação, que representam a concentração de espécies químicas em um processo de advecção-difusão-reação. Um estudo mais aprofundado sobre essas equações e sua solução numérica é dado nas notas de aula de Willen Hundsdorfer (<http://goo.gl/7zbkUY>), professor do *Centrum Wiskunde & Informatica*, Amsterdã, Holanda.

C

- **cgi:** funções para programação CGI (*Common Gateway Interface*) no Octave, que permite passar parâmetros para um programa em um servidor web.
- **communications:** contém funções relacionadas a comunicação digital, sinais aleatórios, campos de Galois, entre outros.
- **control:** ferramentas para projeto de sistemas de controle assistidos por computador (*computer-aided control system design*, ou CACSD). Esse pacote é baseado na biblioteca SLICOT (<http://www.slicot.org/>), desenvolvida para o Fortran.

D

- **data-smoothing:** algoritmos para suavizar dados com

ruído.

- **database**: provê uma interface para bancos de dados SQL utilizando PostgreSQL e a biblioteca `libpq`.
- **dataframe**: ferramentas para manipulação de dados semelhantes à função `data.frame`, disponível na linguagem R.
- **dicom**: possibilita entrada e saída de dados no formato DICOM (*Digital Communications In Medicine*). Depende da biblioteca *Grassroots DiCoM* (GDCM), disponível em <http://sourceforge.net/projects/gdcm/>.

NOTA

Algumas distribuições Linux contém a biblioteca GDCM em seus repositórios. Consulte os pacotes disponíveis em sua distribuição utilizando seu gerenciador de pacotes preferido.

- **divand**: funções para interpolação em n dimensões, com observações localizadas arbitrariamente.
- **doctest**: pacote útil para testes em projetos maiores. Ele encontra blocos de código de exemplo formatados dentro dos arquivos de documentação. Então, executa esse código e confirma se a saída está correta.

E

- **econometrics**: funções econométricas baseadas na estimação de máxima verossimilhança (*Maximum Likelihood Estimation* – MLE) e no método dos

momentos generalizado (*Generalized Method of Moments*, GMM).

F

- **fem-fenics:** resolução de equações diferenciais parciais com base no projeto FEniCS (<http://fenicsproject.org/about/>).
- **financial:** ferramentas para cálculos financeiros no Octave.
- **fits:** entrada e saída de dados no formato FITS (*Flexible Image Transport System*), usando a biblioteca `libcfitsio`.
- **fl-core:** funções básicas para lógica difusa (*fuzzy*).
- **fpl:** rotinas para exportar dados produzidos pelo método dos elementos finitos, ou simulações de volume finitas nos formatos VTK ou OpenDX.
- **fuzzy-logic-toolkit:** biblioteca para lógica difusa, compatível em sua maioria com o MATLAB.

G

- **ga:** códigos para otimização em genética.
- **general:** ferramentas de uso geral para o Octave.
- **generate_html:** gera código HTML com texto de ajuda para um conjunto de funções.
- **geometry:** funções para manipular e exibir primitivas geométricas.

I

- **image:** biblioteca de processamento de imagens, contendo funções para extração de características, operações morfológicas, transformações espaciais e geométricas, entre outras.
- **image-acquisition:** captura de imagens para dispositivos conectados, com suporte à API Video4Linux.
- **instrument-control:** funções de entrada e saída para interfaces seriais, paralelas, I2C, TCP, GPIB, VXI-11 e USBTMC.
- **interval:** permite calcular funções em subconjuntos de seu domínio.
- **io:** entrada e saída em formatos externos, como CSV, DBF, XML, XLS, ODS, entre outros.

J

- **java:** habilita manipulação de objetos similar à Orientação a Objetos em Java.

L

- **level-set:** cálculo da evolução temporal e extração de informação geométrica de superfícies de nível.
- **linear-algebra:** funções adicionais de álgebra linear.
- **lssa:** ferramentas para cálculo de decomposições espetrais de séries temporais irregularmente espaçadas.

- **ltfat**: várias funções para análise em tempo-frequência, processamento de sinais e transformadas *wavelet*.

M

- **mapping**: funções para mapeamento simples e sistema de informação geográfica (*Geographic Information System – GIS*).
- **mechanics**: ferramentas para cálculo numérico em mecânica clássica e análise estrutural.
- **miscellaneous**: funções que não pertencem a um pacote específico.
- **mpi**: funções básicas para MPI e processamento paralelo.
- **msh**: cria malhas triangulares e tetraedrais para resolução de equações diferenciais parciais, por meio dos métodos de elementos finitos ou de volume finito.
- **mvn**: funções para *clustering* da distribuição normal multivariada.

N

- **nan**: biblioteca para estatística e aprendizagem de máquina para manipulação de dados, contendo valores faltantes ou não.
- **ncarray**: acesso a arquivos NetCDF como um vetor de n dimensões.
- **netcdf**: interface NetCDF compatível com o MATLAB.
- **nurbs**: coleção de funções para a criação e manipulação

de *B-splines* racionais não-uniformes (*Non-Uniform Rational B-Spline – NURBS*).

O

- **ocs**: pacote para resolução de equações de circuitos elétricos transitórios e de corrente contínua.
- **octcdf**: interface NetCDF (obsoleta).
- **octclip**: operações booleanas com polígonos.
- **octproj**: executa funções da biblioteca PROJ.4, utilizada na conversão de projeções cartográficas.
- **odepkg**: biblioteca para resolução de equações diferenciais ordinárias.
- **optics**: funções voltadas à óptica.
- **optim**: otimização não linear.
- **optiminterp**: funções para interpolações ótimas em n dimensões, com pontos distribuídos arbitrariamente.

P

- **parallel**: pacote para execução de programas em paralelo.

Q

- **quaternion**: funções para trabalhar com quatérnios.
- **queueing**: biblioteca para enfileiramento de redes e análise de cadeias de Markov discretas e contínuas, com funções para convolução, análise do valor médio etc.

- **secs1d, secs2d, secs3d:** simulador de conveção-difusão para dispositivos semicondutores de uma, duas e três dimensões, respectivamente.
- **signal:** funções para processamento, filtragem, janelamento e visualização de sinais.
- **sockets:** funções de soquete para comunicação em rede a partir do Octave.
- **sparsersb:** implementação para o formato de matriz esparsa RSB por meio de uma interface para o pacote *librsb*.
- **specfun:** funções especiais – funções elípticas, integrais de seno, cosseno, exponencial, funções de Dirac, Heaviside, polinômio de Laguerre etc.
- **splines:** funções adicionais para manipular *splines*.
- **statistics:** pacote com funções adicionais para estatística.
- **stk:** ferramentas para a técnica de interpolação/regressão conhecida como *krigagem*, além de funções úteis em geoestatística, aprendizagem de máquina, regressão não paramétrica etc.
- **strings:** funções adicionais para manipular strings.
- **struct:** funções adicionais para manipulação de estruturas.
- **symbolic:** possibilita a computação simbólica no Octave, com ferramentas como operações algébricas,

resolução de equações, transformadas de Fourier e Laplace, entre outras.

T

- **tsa:** análise de séries temporais com base em conceitos estocásticos e métodos de máxima entropia.

V

- **vrml:** gráficos 3D utilizando linguagem para modelagem de realidade virtual (*Virtual Reality Modeling Language – VRML*).

W

- **windows:** disponibiliza uma interface COM e funcionalidade adicional para o sistema operacional Windows.

7.3 RESUMINDO

Neste capítulo, desvendamos o Octave-Forge, um repositório de pacotes destinado a fornecer funções específicas para o Octave. Aprendemos a instalar e desinstalar os pacotes do Forge, como carregar um pacote para execução no Octave e também uma lista de pacotes disponíveis. Com o que aprendemos sobre o Octave na nossa jornada, já temos conhecimento suficiente para desenvolver programas complexos e específicos, além de podermos resolver problemas de várias áreas científicas.

No próximo e último capítulo deste livro, veremos onde encontrar respostas para suas dúvidas com o Octave, indo além do `help`. Trataremos de documentos e fontes para consulta, assim

como fóruns para pedir ajuda quando a documentação não for suficiente. Até lá!

CAPÍTULO 8

PARA ONDE IR AGORA?

Ao longo deste livro, estudamos novos conceitos e resolvemos diferentes tipos de problemas. Mas há muitas fontes para ir além destas páginas e aprender mais. Neste capítulo, listaremos algumas delas.

8.1 APRENENDENDO SOBRE O OCTAVE NO PRÓPRIO OCTAVE

Como vimos no capítulo *Primeiros passos*, um dos canais de ajuda sobre o Octave é o próprio Octave: o comando `help` retorna um pequeno manual sobre o comando ou função informado como argumento. Por exemplo, a ajuda para o operador `for` mostra sua definição, um exemplo de uso e algumas funções similares:

```
> help for
-- Keyword: for I = RANGE
Begin a for loop.

    for i = 1:10
        i
    endfor

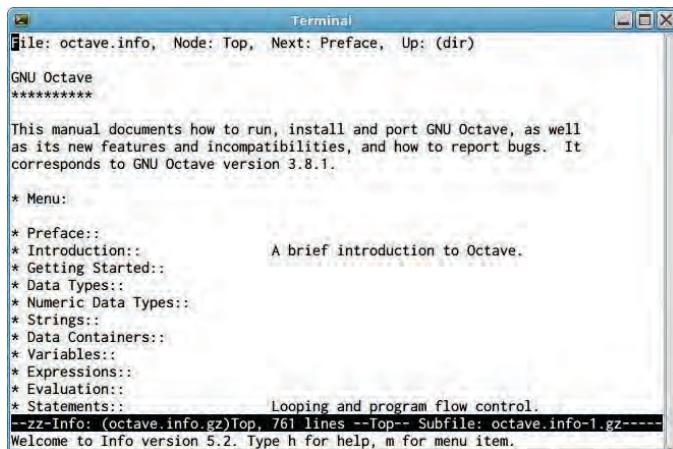
See also: do, parfor, while.
```

Additional help for built-in functions and operators is available in the online version of the manual. Use the command '`doc <topic>`' to search the manual index.

Help and information about Octave is also available on the WWW at <http://www.octave.org> and via the `help@octave.org`

mailing list.

Além do comando `help`, o Octave fornece um manual de referência completo a partir do comando `doc`. Esse sistema é exibido na figura:



A screenshot of a terminal window titled "Terminal". The window displays the contents of the file "octave.info". The text includes the title "GNU Octave", a copyright notice, and a menu section with items like "Preface", "Introduction", "Getting Started", etc. At the bottom, it shows "Welcome to Info version 5.2. Type h for help, m for menu item."

```
File: octave.info, Node: Top, Next: Preface, Up: (dir)
GNU Octave
*****
This manual documents how to run, install and port GNU Octave, as well
as its new features and incompatibilities, and how to report bugs. It
corresponds to GNU Octave version 3.8.1.

* Menu:

* Preface::                                     A brief introduction to Octave.
* Introduction::                                ...
* Getting Started::                             ...
* Data Types::                                  ...
* Numeric Data Types::                         ...
* Strings::                                     ...
* Data Containers::                            ...
* Variables::                                   ...
* Expressions::                                 ...
* Evaluation::                                  ...
* Statements::                                  Looping and program flow control.
--zz-Info: (octave.info.gz)Top, 761 lines --Top-- Subfile: octave.info-1.gz---
Welcome to Info version 5.2. Type h for help, m for menu item.
```

Figura 8.1: A primeira tela do sistema de documentação do Octave, executado em um terminal

8.2 DOCUMENTAÇÃO E MANUAIS DISPONÍVEIS

Além da ajuda disponível internamente, o Octave possui outras referências para consulta. O manual de referência do Octave (disponível por meio do comando `doc`) possui versões online em dois formatos: HTML (<http://www.octave.org/doc/interpreter/>) e PDF (<http://www.octave.org/octave.pdf>).

O Octave também possui sua própria *wiki*, com uma lista de perguntas frequentes (*frequently asked questions* – FAQ), informações sobre a instalação e a criação de pacotes, Octave-Forge, editores de texto compatíveis com o realce de sintaxe da linguagem, exemplos e tutoriais, além de relatórios sobre o desenvolvimento das novas versões. Confira em <http://wiki.octave.org/>.

Além disso, a wiki mantém um *livro de receitas*, com soluções para problemas frequentemente expostos. Veja em <http://wiki.octave.org/Cookbook>.

Como o Octave é semelhante ao MATLAB, da MathWorks, você provavelmente encontrará alguma informação compatível na página de suporte do MATLAB (<http://www.mathworks.com/help/matlab/>).

A página *File Exchange*, comunidade na qual usuários de MATLAB disponibilizam código, funções e aplicativos, também pode lhe fornecer algum material útil. Verifique: <http://www.mathworks.com/matlabcentral/fileexchange/>.

8.3 BUSCANDO AJUDA COM OUTROS USUÁRIOS

Caso nenhuma das opções dessas opções resolva um problema que você possa vir a ter, a alternativa é tirar suas dúvidas com outros usuários.

Um dos exemplos é a lista oficial de e-mails do Octave. Para que os desenvolvedores da lista recebam sua dúvida, envie-a ao e-mail `help@octave.org`. Os arquivos anteriores dessa lista estão no endereço <http://lists.gnu.org/archive/html/help-octave/>. Procure por lá; talvez sua dúvida já tenha sido respondida em outra situação.

Prefere usar um programa de IRC (*internet relay chat*) para ter contato com outros usuários? O canal de IRC oficial do Octave é o `#octave`, no servidor `irc.freenode.net`.

O Nabble (<http://www.nabble.com/>), um site que hospeda fóruns, também possui um fórum sobre o Octave: <http://octave.1599824.n4.nabble.com/>.

Sua pergunta também pode ser postada no Stack Overflow (<http://stackoverflow.com/>). Mas antes de postar, verifique a seção de ajuda (<http://stackoverflow.com/help>), que explica as "regras de convivência" do site, e procure pela resposta em seus arquivos, que já têm mais de 10 milhões de questões.

Além de todas essas opções, este livro possui seu próprio fórum do Google, em <http://forum.casadocodigo.com.br>. Quando nenhuma das dicas anteriores for suficiente, poste sua dúvida lá. Eu e outros usuários teremos o prazer de ajudá-lo no que for possível.

8.4 COMO CONTRIBUIR COM O OCTAVE

Quer se envolver com o desenvolvimento do Octave? Todo tipo de ajuda é necessária: você pode criar novas ferramentas, resolver bugs, auxiliar no aprimoramento das páginas web e a documentação, e até responder a questões na lista de e-mail. Tudo o que você precisa saber está neste endereço: <http://www.gnu.org/software/octave/get-involved.html>.

8.5 RESUMINDO

Chegamos ao fim da nossa jornada com o Octave! Relembrando, neste livro:

- Vimos como instalar e configurar o Octave em seu computador.
- Aprendemos a utilizar o Octave como uma calculadora científica completa.
- Fizemos operações com os tipos básicos de variáveis: strings, vetores e estruturas.
- Estudamos como plotar gráficos de duas e três dimensões, com formas e cores variadas.
- Entendemos como reaproveitar código utilizando

scripts e funções.

- Exploramos expressões e operadores, flexibilizando nossos programas.
- Investigamos o Octave-Forge, compreendendo como instalar e utilizar seus pacotes.
- Por fim, conhecemos várias formas de aprimorar nosso aprendizado, recorrer à ajuda online e contribuir para o desenvolvimento do Octave.

Esperamos que este livro tenha sido uma boa referência no seu processo de aprendizado! Para tanto, queremos saber a sua opinião: o que achou do livro, quais os pontos que mais gostou, o que poderíamos melhorar... Deixe seus comentários no fórum destinado a esse livro: <http://groups.google.com/d/forum/compcieoctave>.

Lá, você encontrará uma área para se apresentar, e conhecer outros leitores e usuários.

Não se esqueça de baixar e analisar os códigos que estudamos até aqui. Eles estão disponíveis em <https://goo.gl/VnnAnF>. Se houver quaisquer dúvidas sobre esses arquivos, não hesite: recorra ao fórum!

Ficaremos felizes com o seu retorno! Um grande abraço e até breve!

CAPÍTULO 9

REFERÊNCIAS BIBLIOGRÁFICAS

AMUASI, Henri; SCHEFFLER, Carl; PICKLES, Mike. *Octave programming tutorial*. Wikibooks, 2014. Disponível em: http://en.wikibooks.org/wiki/Octave_Programming_Tutorial.

EATON, John W.; et al. *Software GNU Octave, versão 3.8.1*. Disponível em: <http://www.gnu.org/software/octave/>.

EATON, John W.; BATEMAN, David; HAUBERG, Soren; WEHBRING, Rik. *GNU Octave: a high-level interactive language for numerical computations (version 4.0.0 manual)*. 2015. Disponível em: <http://www.gnu.org/software/octave/doc/interpreter/>.

HUNT, Brian; LIPSMAN, Ronald; ROSENBERG, Jonathan; COOMBES, Kevin; OSBORN, John; STUCK, Garrett. *A guide to MATLAB for beginners and experienced users*. Cambridge: Cambridge University Press, 2001.

KNIGHT, Andrew. *Basics of MATLAB and beyond*. Chapman and Hall/CRC, 1999.