3. CAPÍTULO 3 – INTRODUÇÃO À PROGRAMAÇÃO DO MATLAB

TERMOS CHAVE			CONTEÚDO
programa de computador *scripts* algoritmo programa modular projeto de cima para baixo arquivo externo dispositivo de entrada padrão *prompting* dispositivo de saída padrão executar um *script* linguagens de alto nível linguagem de máquina executável compilador código fonte código objeto interpretador	documentação comentários comentário de bloco blocos de comentário entrada/saída (E/S) usuário *string* vazia mensagem de erro formatação *string* de formato marcador de espaço caracteres de conversão caractere nova linha largura de campo espaços em branco na frente zeros à direita símbolos de plotagem marcadores tipos de linha alternadores	modos escrevendo num arquivo acrescentando num arquivo lendo de um arquivo funções definidas pelo usuário chamada de função argumento controle valor de retorno cabeçalho da função argumento de saída argumentos de entrada corpo da função definição de função variáveis locais escopo de variáveis espaço de trabalho base	

Nós usamos até agora o produto MATLAB® interativamente na Janela de Comandos. Isso é suficiente quando tudo que se precisa é um cálculo simples. No entanto, em muitos casos, são necessários alguns passos antes que o resultado final possa ser obtido. Nesses casos, é mais conveniente agrupar instruções no que é chamado de ***programa de computador***.

Neste capítulo, apresentaremos os programas mais simples do MATLAB, chamados de ***scripts***. Exemplos de *scripts* que personalizam o traçado de gráficos simples ilustram o conceito. A entrada será introduzida, tanto a partir de arquivos como pelo usuário. Saída para arquivos e para a tela também serão apresentadas. Finalmente, serão descritas funções definidas pelo usuário que calculam e retornam um único valor. Estes tópicos servem como uma introdução à programação, que será expandida em Capítulo 6.

3.1 ALGORITMOS

Antes de escrever qualquer programa de computador, é útil descrever primeiro as etapas que serão necessárias. Um ***algoritmo*** é a sequência de passos necessárias para resolver um problema. Em uma abordagem ***modular*** para a programação, a solução do problema é dividida em passos separados e, em seguida, cada passo é refinado até que os passos resultantes sejam pequenos o suficiente para serem transformadas em instruções executáveis. Isso é chamado de abordagem de ***top-down design*** (***projeto de cima para baixo*** – da solução mais geral para a mais detalhada).

Como um exemplo simples, considere o problema de calcular a área de um círculo. Primeiro, é necessário determinar quais informações são necessárias para resolver o problema, que, nesse caso, é o raio do círculo. Em seguida, dado o raio do círculo, a área do círculo poderia ser calculada. Finalmente, uma vez que a área tenha sido calculada, ela deve ser exibida de alguma forma. O algoritmo básico é então composto por três passos:

- obter o dado de entrada – o raio
- calcular o resultado – a área
- exibir a saída.

Mesmo com um algoritmo tão simples, é possível refinar cada uma das etapas. Quando um programa é escrito para implementar esse algoritmo, os passos seriam os seguintes.

- De onde vem a entrada? Duas escolhas seriam possíveis: de um arquivo externo ou do usuário (a pessoa que está executando o programa) que entra com o número, digitando-o no teclado. Para cada sistema, um deles será o dispositivo padrão de entrada (o que significa que, se não especificado de outra forma, é de onde vem a entrada!). Se o usuário deve entrar com o raio, o usuário deve ser instruído a digitar o raio (e em qual unidade). Dizer ao usuário o que digitar é chamado de ***prompting*** (solicitação de digitação). Assim, a etapa de entrada realmente se torna dois passos: avisar o usuário para inserir um raio e depois lê-lo para o programa.
- Para calcular a área, a fórmula é necessária. Neste caso, a área do círculo é π multiplicado pelo quadrado do raio. Então, isso significa que o valor da constante para π é necessário ao programa.
- Para onde vai a saída? São duas possibilidades: (1) para um arquivo externo ou (2) para a tela. Dependendo do sistema, um deles será o dispositivo de saída padrão. Ao exibir a saída do programa, ela deve ser sempre informativa quanto possível. Em outras palavras, em vez de só imprimir a área (apenas o número), ela deve ser impressa em um formato de frase bonito. Além disso, para tornar a saída ainda mais clara, a entrada deve ser impressa. Por exemplo, a saída pode ser a frase “Para um círculo com um raio de 1 centímetro, a área é 3,1416 centímetros quadrados”.

Para a maioria dos programas, o algoritmo básico consiste nas três etapas descritas:

1. Obter a(s) entrada(s)
2. Calcular o(s) resultado(s)
3. Exibir o(s) resultado(s).

Como pode ser visto aqui, mesmo as soluções de problemas mais simples podem ser refinadas posteriormente. Este é o projeto *top-down*.

3.2 SCRIPTS DO MATLAB

Uma vez que um problema tenha sido analisado e o algoritmo para sua solução tenha sido escrito e refinado, a solução para o problema é então escrita em uma linguagem de programação específica. Um programa de computador é uma sequência de instruções, em uma determinada linguagem, que diz ao computador como realizar uma determinada tarefa. ***Executar*** um programa é fazer com que o computador siga realmente estas instruções sequencialmente.

Linguagens de alto nível têm comandos e funções semelhantes ao inglês, como “print this” (“imprima isso”) ou “if x < 5 do something” (“se x < 5 faça alguma coisa”). O computador, no

entanto, só pode interpretar comandos escritos em sua ***linguagem de máquina***. Programas que são escritos em linguagens de alto nível devem, portanto, ser traduzidos para linguagem de máquina antes que o computador possa realmente executar a sequência de instruções do programa. Um programa que faz essa conversão de uma linguagem de alto nível para um ***arquivo executável*** é chamado de ***compilador***. O programa original é chamado de ***código-fonte*** e o programa executável resultante é chamado de ***código-objeto***. Compiladores traduzem do código-fonte para o código-objeto; este é então executado como uma etapa separada.

Por outro lado, um ***interpretador*** percorre o código linha por linha, traduzindo e executando cada comando. O MATLAB usa o que chamamos de ***arquivos de script*** ou ***arquivos-M*** (a razão para isso é que a extensão no nome do arquivo é $.m$). Esses arquivos de *script* são interpretados, em vez de compilados. Portanto, a terminologia correta é que esses são *scripts* e não programas. No entanto, os termos são pouco usados por muitas pessoas, e a documentação no próprio MATLAB refere-se a *scripts* como programas. Neste livro, reservamos o uso da palavra "programa" para significar um conjunto de *scripts* e funções, conforme descrito brevemente na Seção 3.7 e, em seguida, mais detalhadamente no Capítulo 6.

Um *script* é uma sequência de instruções do MATLAB que é armazenada e salva em um arquivo-M. O conteúdo de um *script* pode ser exibido na Janela de Comandos usando o comando **type**. O *script* pode ser executado simplesmente inserindo o nome do arquivo (sem a extensão $.m$).

Antes de criar um *script*, verifique se o "Current Folder" (Pasta Atual), chamado também de "Current Directory" (Diretório Atual) em versões anteriores, está definido para a pasta na qual você deseja salvar seus arquivos.

As etapas envolvidas na criação de um *script* dependem da versão do MATLAB. Nas versões mais recentes, o método mais fácil é clicar em "New Script" na guia "HOME". Alternativamente, você pode clicar na seta para baixo sob "New" e depois escolher "Script" (veja a Figura 3.1)

Nas versões anteriores, alguém clicaria em "File", depois em "New", depois em "Script" , ou, mesmo em versões anteriores em "M-file" (arquivo-M). Aparecerá uma nova janela chamada de "Editor" (que pode ser encaixada). Nas versões mais recentes do MATLAB, esta janela tem três guias: "EDITOR", "PUBLISH" e "VIEW". Em seguida, basta digitar a sequência de instruções (observe que os números de linha aparecem à esquerda).

Quando terminar, salve o arquivo escolhendo "Save" na seta para baixo da guia EDITOR ou, em versões anteriores do MATLAB, escolhendo "File" e, em seguida, "Save". Certifique-se de que a extensão $.m$ esteja no nome do arquivo (esse deve ser o padrão). As regras para nomes de arquivos são as mesmas que para variáveis (elas devem começar com uma letra; depois disso, pode haver letras, dígitos ou sublinhado). Por exemplo, agora criaremos um script chamado $script1.m$ que calcula a área de um círculo. Ele atribui um valor para o raio e calcula a área com base nesse raio.

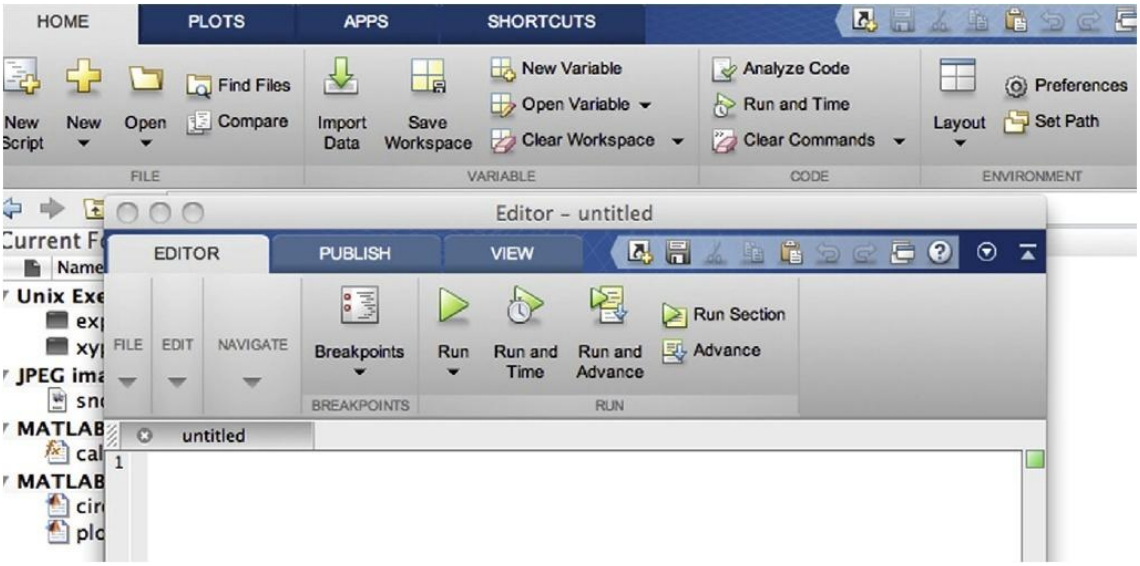

FIGURA 3.1 Barra de ferramentas e editor

Neste texto, os *scripts* serão exibidos em uma caixa com o nome do arquivo-M no topo.

`script1.m`

```
raio = 5
area = pi * (raio^2)
```

Há duas formas de visualizar um *script* depois de escrito: abra a Janela do Editor para visualizá-lo ou use o comando **type**, conforme mostrado aqui, para exibi-lo na Janela de Comandos. O comando **type** mostra o conteúdo do arquivo chamado $script1.m$; observe que o $.m$ não está incluído:

```
>> type script1
raio = 5
area = pi * (raio^2)
```

Para efetivamente executar ou executar o *script* a partir da Janela de Comandos, o nome do arquivo é inserido no ***prompt*** (novamente, sem o $.m$). Quando executado, os resultados das duas instruções de atribuição são exibidos, pois a saída não foi suprimida para nenhuma das instruções.

```
>> script1
raio =
    5
area =
    78.5398
```

Depois que o *script* for executado, você poderá descobrir que deseja fazer alterações nele (especialmente se houver erros!). Para editar um arquivo existente, existem vários métodos para abri-lo. Os mais fáceis são:

- dentro da janela da “Current Folder”, clique duas vezes no nome do arquivo na lista de arquivos

- escolher a seta para baixo “Open” (Abrir) mostrará uma lista “Recent Files” (Arquivos Recentes).

3.2.1 Documentação

É muito importante que todos os *scripts* sejam bem ***documentados***, para que as pessoas possam entender o que o *script* faz e como ele realiza sua tarefa. Uma maneira de documentar um *script* é colocar ***comentários*** nele. No MATLAB, um comentário é qualquer coisa, a partir do **%** até o final dessa linha em particular. Comentários são completamente ignorados quando o *script* é executado. Para colocar um comentário, basta digitar o símbolo % no início de uma linha, ou selecionar as linhas de comentário e depois clicar na seta para baixo “Edit” e clicar no símbolo %, e o editor colocará os símbolos % no início dessas linhas para os comentários.

Por exemplo, o *script* anterior para calcular a área de um círculo pode ser modificado para ter comentários:

`scriptcirculo.m`

```
% Este script calcula a área de um círculo

% Primeiro o raio é atribuído
raio = 5
% A área é calculada com base no raio
area = pi * (raio^2)
```

O primeiro comentário no início do *script* descreve ser o que o *script* faz; isso às vezes é chamado de ***comentário de bloco***. Em seguida, durante todo o *script*, os comentários descrevem as diferentes partes do *script* (não habitualmente um comentário para cada linha, no entanto!). Os comentários não afetam o que um *script* faz, portanto, a saída desse *script* seria a mesma da versão anterior.

O comando **help** no MATLAB trabalha com *scripts*, bem como com funções internas. Os primeiros comentários de bloco (definidos como linhas contíguas no início) serão exibidos. Por exemplo, para o $scriptcirculo$:

```
>> help scriptcirculo
Este script calcula a área de um círculo
```

A razão pela qual uma linha em branco foi inserida no *script* entre os dois primeiros comentários é que, de outra forma, ambos teriam sido interpretados como comentários contíguos e ambas as linhas teriam sido exibidas com **help**. A primeira linha de comentário é chamada de "linha H1"; esta linha é onde a função **lookfor** procura.

PRÁTICA 3.1

Escreva um *script* para calcular a circunferência de um círculo ($C = 2\,\pi\,r$). Comente o *script*.

Comentários mais longos, chamados de ***blocos de comentários***, consistem em tudo entre %{ e %}, que deve estar sozinho em linhas separadas. Por exemplo:

```
%{
   esse é realmente
   realmente
   REALMENTE
   um comentário longo
%}
```

3.3 ENTRADA E SAÍDA

O *script* anterior seria muito mais útil se fosse mais geral; por exemplo, se o valor do raio pudesse ser lido de uma fonte externa, em vez de ser atribuído no *script*. Além disso, seria melhor que o *script* imprimisse a saída de uma maneira interessante e informativa. As instruções que realizam essas tarefas são chamadas de instruções de ***entrada/saída*** ou ***E/S*** para abreviar. Embora, por simplicidade, exemplos de instruções de entrada e saída sejam mostrados aqui na Janela de Comandos, essas instruções farão mais sentido em *scripts*.

3.3.1 Função de Entrada

Instruções de entrada lêm valores do dispositivo de entrada padrão. Na maioria dos sistemas, o dispositivo de entrada padrão é o teclado, portanto, a instrução de entrada lê valores que foram inseridos pelo ***usuário***, ou seja, pela pessoa que está executando o *script*. Para permitir que o usuário saiba o que deve inserir, o *script* deve primeiro solicitar ao usuário os valores especificados.

A função de entrada mais simples no MATLAB é chamada de **input**. A função **input** é usada em uma instrução de atribuição. Para chamá-la, uma ***string*** (cadeia de caracteres) é passada, que é o ***prompt*** que aparecerá na tela, e quaisquer que seja o tipo do dado digitado pelo usuário, o dado será armazenado na variável nomeada à esquerda da instrução de atribuição. Para facilitar a leitura do *prompt*, é útil colocar dois-pontos e, em seguida, um espaço após o *prompt*. Por exemplo,

```
>> raio = input('Digite o raio: ')
Digite o raio: 5
raio =
   5
```

Se a entrada de caractere ou string for desejada, um 's' deve ser adicionado como um segundo argumento para a função **input**:

```
>> letra = input('Digite um caractere: ', 's')
Digite um char: g
letra =
g
```

Se o usuário inserir apenas espaços ou tabulações antes de pressionar a tecla Enter, eles serão ignorados e uma sequência vazia será armazenada na variável:

```
>> meucarac = input('Digite um caractere: ', 's')
Digite um caractere:
meucarac =
   ''
```

Nota

Embora normalmente os apóstrofos não sejam mostrados em torno de um caractere ou string, neste caso elas são mostradas para demonstrar que não há nada dentro da string.

No entanto, se espaços em branco forem inseridos antes de outros caracteres, eles serão incluídos na sequência. No próximo exemplo, o usuário pressionou a barra de espaço quatro vezes antes de entrar com “ir”. A função **length** retorna o número de caracteres da string.

```
>> minhacadeia = input('Digite uma string: ', 's')
Digite uma string: ir
minhacadeia =
    ir
>> length(minhacadeia)
ans =
   6
```

PERGUNTA RÁPIDA!

Qual seria o resultado se o usuário inserir espaços em branco depois de outros caracteres? Por exemplo, o usuário aqui inseriu "xyz" seguido de quatro espaços em branco:

```
>> meuscarac = input('Inserir caracteres: ', 's')
Inserir caracteres: xyz
meuscarac =
xyz
```

Resposta

Os caracteres de espaço seriam armazenados na variável string. É difícil ver acima, mas fica claro a partir do comprimento da string.

```
>> length(meuscarac)
ans =
   7
```

O comprimento pode ser visto na Janela de Comandos usando o mouse para destacar o valor da variável; o xyz e quatro espaços serão destacados.

Também é possível que o usuário digite apóstrofos delimitando a string, em vez de incluir o segundo argumento 's' na chamada para a função de entrada.

```
>> nome = input ('Digite seu nome: ')
Digite seu nome: 'Maria'
```

```
nome =
Maria
```

No entanto, isso pressupõe que o usuário saberia fazer isso, portanto, é melhor indicar que é desejada a entrada de caracteres, na própria função **input**. Além disso, se o 's' for especificado e o usuário inserir apóstrofos, eles se tornarão parte da sequência.

```
>> nome = input('Digite seu nome: ', 's')
Digite seu nome: 'Maria'
nome =
'Maria'
>> length(nome)
ans =
   7
```

Observe o que acontece se a entrada da cadeia de caracteres (string) não tiver sido especificada, mas o usuário inserir uma letra em vez de um número.

```
>> num = input('Digite um número: ')
Digite um numero: t
Error using input
Undefined function ou variabble 't'.

Digite um numero: 3
num =
   3
```

MATLAB deu uma ***mensagem de erro*** e repetiu o *prompt*. No entanto, se *t* for o nome de uma variável, o MATLAB assumirá seu valor como entrada.

```
>> t = 11;
>> num = input('Digite um numero: ')
Digite um numero: t
num =
   11
```

Instruções **input** separadas são necessárias se mais de uma entrada for desejada. Por exemplo,

```
>> x = input('Digite a coordenada x: ');
>> y = input('Digite a coordenada y: ');
```

Normalmente, em um *script*, os resultados das instruções **input** são suprimidos se um ponto e vírgula é digitado no final das instruções de atribuição.

PRÁTICA 3.2

Crie um *script* que solicite do usuário um comprimento e, em seguida, use "p" para pés ou "m" para metros e armazene as duas entradas em variáveis. Por exemplo, quando executado, ficaria assim (assumindo que o usuário digita 12.3 e depois m):

```
Digite o comprimento: 12.3
Isto é p (pes) ou m (metros)?: m
```

3.3.2 Instruções de Saída: disp e fprintf

As instruções de saída exibem cadeias e/ou os resultados de expressões e podem permitir a formatação ou a personalização de como são exibidos. A função de saída mais simples no MATLAB é **disp**, que é usada para exibir o resultado de uma expressão ou uma string sem atribuir qualquer valor à variável padrão ans. No entanto, **disp** não permite a formatação. Por exemplo,

```
>> disp('Oi')
Oi
>> disp(4^3)
   64
```

A saída formatada pode ser impressa na tela usando a função **fprintf**. Por exemplo,

```
>> fprintf('O valor é %d, com certeza!\n', 4^3)
O valor é 64, com certeza!
>>
```

Para a função **fprintf**, primeiro é passada uma string (denominada ***string de formatação*** ou ***cadeia de formatação***) que contém qualquer texto a ser impresso, bem como ***especificações de formato*** das expressões a serem impressas. Neste exemplo, o %d é um exemplo de especificações de formato.

Às vezes, o %d é chamado de ***marcador de lugar*** ou ***espaço reservado*** porque especifica onde o valor da expressão que está após a sequência deve ser impresso. O caractere no marcador de lugar é chamado de ***caractere de conversão*** e especifica o tipo de valor que está sendo impresso. Existem outros, mas o que segue é uma lista dos marcadores de lugar simples:

```
%d integer (ele significa inteiro decimal)
%f float (número real)
%c caractere único
%s string (cadeia de caracteres)
```

Nota

Não confunda o % no marcador de lugar com o símbolo usado para designar um comentário.

O caractere "\n" no final da string é um caractere especial chamado de ***caractere nova linha***; o que acontece quando é impresso é que a saída que segue é impressa na próxima linha abaixo.

PERGUNTA RÁPIDA!

O que você acha que aconteceria se o caractere nova linha fosse omitido do final de uma instrução **fprintf**?

Resposta

Sem ele, o próximo *prompt* terminaria na mesma linha da saída. Ainda é um *prompt*, e assim uma expressão pode ser inserida, mas parece confusa como mostrado aqui.

```
>> fprintf('O valor é %d, com certeza!',...
4^3)
O valor é 64, com certeza!>> 5 + 3
ans =
    8
```

Note que com a função **disp**, no entanto, o *prompt* sempre aparecerá na próxima linha:

```
>> disp('Oi')
Oi
>>
```

Além disso, observe que as reticências podem ser usadas após uma string, mas não no meio.

PERGUNTA RÁPIDA!

Como você pode obter uma linha em branco na saída?

Resposta

Tendo dois caracteres de nova linha em uma linha.

```
>> fprintf('O valor é %d,\n\nOK!\n', 4^3)
O valor é 64,

OK!
```

Isso também indica que o caractere de nova linha pode estar em qualquer lugar da string; quando é impresso, a saída desce para a próxima linha.

Observe que o caractere de nova linha também pode ser usado no *prompt* da instrução de entrada; por exemplo:

```
>> x = input('Digite a coordenada \nx:');
Digite a coordenada
x: 4
```

No entanto, esse é o único caractere de formatação permitido no *prompt* da função ***input***.

Para imprimir dois valores, deveria haver dois marcadores de posição na cadeia de formatação e duas expressões depois da cadeia de formatação. As expressões preenchem os marcadores de lugar, em sequência.

```
>> fprintf ('O int é %d e o char é %c\n',...
33-2, 'x')
O int é 31 e o char é x
```

Uma ***largura de campo*** também pode ser incluída no espaço reservado em **fprintf**, que especifica quantos caracteres serão usados na impressão. Por exemplo, %5d indicaria uma largura de campo de 5 para imprimir um inteiro e %10s indicaria uma largura de campo de 10 para uma cadeia. Para um $float$, o número de casas decimais também pode ser especificado; por exemplo, %6.2f significa uma largura de campo de 6 (incluindo o ponto decimal e as $duas$ casas decimais) com duas casas decimais. Para um $float$, apenas o número de casas decimais também pode ser especificado; por exemplo, %.3f indica três casas decimais, independentemente da largura do campo.

```
>> fprintf('O int é %3d e o float é %6.2f\n', 5, 4.9)
O int é 5 e o float é 4.90
```

Nota

Se a largura do campo for maior que o necessário, os ***espaços em branco iniciais*** serão impressos e, se casas decimais são especificadas mais do que o necessário, ***zeros à direita*** são impressos.

PERGUNTA RÁPIDA!

O que você acha que aconteceria se você tentasse imprimir 1234.5678 em uma largura de campo de 3 com 2 casas decimais?

```
>> fprintf('%3.2f\n', 1234.5678)
```

Resposta

Ele imprimiria toda a parte inteira 1234, mas arredondaria as casas decimais para duas casas, ou seja,

```
1234.57
```

Se a largura do campo não for grande o suficiente para imprimir o número, a largura do campo será aumentada. Basicamente, cortar o número daria um resultado enganoso, mas arredondar as casas decimais não altera muito o número.

PERGUNTA RÁPIDA!

O que aconteceria se você usasse o caractere de conversão $\%d$, mas estivesse tentando imprimir um número real?

Resposta

O MATLAB mostrará o resultado usando a notação exponencial

```
>> fprintf('%d\n', 1234567.89)
1.234568e+006
```

Observe que, se você quiser uma notação exponencial, essa não é a maneira correta de obtê-la; Em vez disso, existem caracteres de conversão que podem ser usados. Use o navegador **help** para ver esta opção, assim como muitas outras!

Existem muitas outras opções para a especificação de formato. Por exemplo, o valor que está sendo impresso pode ser justificado à esquerda na largura do campo usando um sinal de menos. O exemplo a seguir mostra a diferença entre imprimir o inteiro 3 usando $\%5d$ e usando $\% - 5d$. Os x's abaixo são usados para mostrar o espaçamento.

```
>> fprintf('O inteiro é xx%5dxx e xx%-5dxx\n', 3, 3)
O inteiro é xx    3xx e xx3    xx
```

Além disso, strings podem ser truncadas especificando as “casas decimais”:

```
>> fprintf('A cadeia é %s ou %.2s\n', 'rua', 'rua')
A cadeia é rua ou ru
```

Existem vários caracteres especiais que podem ser impressos na cadeia de formatação, além do caractere de nova linha. Para imprimir uma barra, são usadas duas barras em uma linha, e também para imprimir um apóstrofo, são usados dois apóstrofos em uma linha. Além disso, $'\backslash t'$ é o caractere de tabulação.

```
>> fprintf('Tente isso: tab\t apóstrofo '' contra-barra \\ \n')
Tente isso: tab  apóstrofo ' contra-barra \
```

3.3.2.1 Imprimir vetores e matrizes

Para um vetor, se um caractere de conversão e o caractere nova linha estiverem na cadeia de formatação, ele será impresso em uma coluna, independentemente de o vetor em si ser um vetor de linha ou um vetor de coluna.

```
>> vet = 2:5;
>> fprintf('%d\n', vet)
2
3
4
5
```

Sem o caractere nova linha, ele seria impresso em uma linha, mas o próximo *prompt* apareceria na mesma linha:

```
>> fprintf('%d', vet)
2345 >>
```

No entanto, em um *script*, um caractere de nova linha separado pode ser impresso para evitar esse problema. Também é muito melhor separar os números com espaços.

imprimevet.m

```
% Isso demonstra a impressão de um vetor

vet = 2:5;
fprintf('%d ', vet)
fprintf('\n')
```

```
>> imprimevet
2 3 4 5
>>
```

Se o número de elementos no vetor for conhecido, muitos caracteres de conversão poderão ser especificados e, em seguida, a nova linha:

```
>> fprintf ('%d %d %d %d\n', vet)
2 3 4 5
```

Isto não é muito geral e, portanto, não é preferível.

Para matrizes, o MATLAB "desenrola" a matriz coluna por coluna. Por exemplo, considere a seguinte matriz $2\ x\ 3$:

```
>> mat = [5 9 8; 4 1 10]
mat =
    5    9    8
    4    1   10
```

Especificar um caractere de conversão e, em seguida, o caractere nova linha imprimirá os elementos da matriz em uma coluna. Os primeiros valores impressos são da primeira coluna, depois da segunda coluna e assim por diante.

```
>> fprintf('%d\n', mat)
5
4
9
1
8
10
```

Se três dos caracteres de conversão $\%d$ forem especificados, a **fprintf** imprimirá três números em cada linha de saída, mas novamente a matriz é desenrolada coluna por coluna. Novamente, ele imprime primeiro os dois números da primeira coluna (na primeira linha da saída), depois o primeiro valor da segunda coluna e assim por diante.

```
>> fprintf('%d %d %d\n', mat)
5 4 9
1 8 10
```

Se a transposição da matriz for impressa, no entanto, usando os três caracteres de conversão $\%d$, a matriz é impressa como aparece quando criada.

```
>> fprintf('%d %d %d\n', mat') % Observe a transposição
5 9 8
4 1 10
```

Para vetores e matrizes, embora a formatação não possa ser especificada, a função **disp** pode ser mais fácil de usar em geral do que a **fprintf**, pois exibe o resultado de maneira direta. Por exemplo,

```
>> mat = [15 11 14; 7 10 13]
mat =
   15   11   14
    7   10   13
>> disp(mat)
   15   11   14
    7   10   13
>> vet = 2:5
vet =
   2   3   4   5
>> disp(vet)
   2   3   4   5
```

Observe que quando *loops* (laços) forem abordados no Capítulo 5, a formatação da saída de matrizes será mais fácil. Por enquanto, no entanto, **disp** funciona bem.

3.4 SCRIPTS COM ENTRADA E SAÍDA

Juntando tudo isso agora, podemos implementar o algoritmo usado desde o começo deste capítulo. O *script* a seguir calcula e imprime a área de um círculo. Ele primeiro solicita ao usuário um raio, lê o raio e depois calcula e imprime a área do círculo com base nesse raio.

`circuloES.m`

```
% Este script calcula a área de um círculo
% Ele solicita ao usuário o raio

% Solicita ao usuário o raio e calcula
% a área com base nesse raio
fprintf('Nota: as unidades serão em centímetros.\n')
raio = input('Por favor digite o raio: ');
area = pi * (raio^2);
% Imprime todas as variáveis em um formato de sentença
fprintf('Para um círculo com um raio de %.2f centímetros,\n', raio)
fprintf ('a área é de %.2f centímetros quadrados\n', area)
```

A execução do *script* produz a seguinte saída:

```
>> circuloES
Nota: as unidades serão em centímetros.
Por favor insira o raio: 3.9
Para um círculo com um raio de 3.90 centímetros,
a área é de 47.78 centímetros quadrados
```

Observe que a saída das duas primeiras instruções de atribuição (incluindo a **input**) é suprimida colocando ponto e vírgula no final. Isso geralmente é feito em *scripts*, para que o formato exato do que é exibido pelo programa seja controlado pelas funções **fprintf**.

PRÁTICA 3.3

Escreva um script para solicitar ao usuário separadamente um caractere e um número e, imprima o caractere em uma largura de campo de 3 e o número à esquerda justificado em uma largura de campo de 8 com 3 casas decimais. Teste isso digitando números com larguras variadas.

3.5 SCRIPTS PARA PRODUZIR E PERSONALIZAR PLOTAGENS SIMPLES

O MATLAB possui muitos recursos gráficos. A personalização de gráficos geralmente é desejada, e isso é mais fácil de realizar criando um script em vez de digitar um comando de cada vez na Janela de Comandos. Por esse motivo, gráficos simples e como personalizá-los serão apresentados neste capítulo sobre programação do MATLAB.

Os tópicos de ajuda que contêm funções gráficas incluem **graph2d** e **graph3d**. Digitando **help graph2d** exibiria algumas das funções gráficas bidimensionais, além de funções para manipular os eixos e colocar rótulos e títulos nos gráficos. A Documentação de Pesquisa (Search Documentation) no MATLAB Graphics (Gráficos do MATLAB) também possui uma seção “2-D and 3-D Plots” (Plotagens 2D e 3D).

3.5.1 A Função Plot

Por enquanto, começaremos com um gráfico muito simples de um ponto usando a função **plot**.

O *script* a seguir, $plotaumponto$, plota um ponto. Para fazer isso, primeiro, valores são dados para as coordenadas x e y do ponto em variáveis separadas. O ponto é traçado usando um asterisco vermelho (‘*’). O gráfico é personalizado especificando os valores mínimo e máximo – primeiro no eixo x, e depois no y. Os rótulos são então colocados no eixo x, no eixo y e no próprio gráfico usando as funções **xlabel**, **ylabel** e **title**. (Nota: não há rótulos padrão para os eixos.)

Tudo isso pode ser feito a partir da Janela de Comandos, mas é muito mais fácil usar um *script*. A seguir é mostrado o conteúdo do script $plotaumponto$ que faz isso. A coordenada x representa a hora do dia (por exemplo, 11 a.m.) e a coordenada y representa a temperatura (por exemplo, em graus Fahrenheit) nesse momento.

plotaumponto.m

```
% Esta é uma plotagem muito simples de apenas um ponto!

% Cria variáveis das coordenadas e traça um '*' vermelho
x = 11;
y = 48;
plot(x, y, 'r*')
% Altera e rotula os eixos
axis([9 12 35 55])
xlabel('Hora')
ylabel('Temperatura')
% Coloca um título na plotagem
title('Hora e Temperatura')
```

Na chamada para a função **axis**, um vetor é passado. Os dois primeiros valores são o mínimo e o máximo para o eixo x, e os dois últimos são o mínimo e o máximo para o eixo y. Executar este *script* traz uma Janela de Figura com o gráfico (veja a Figura 3.2).

Para ser mais geral, o *script* poderia solicitar ao usuário o tempo e a temperatura, em vez de apenas atribuir valores. Em seguida, a função de **axis** pode ser usada com base em quaisquer valores de x e y, como no exemplo a seguir:

```
axis([x-2 x+2 y-10 y+10])
```

Além disso, embora sejam as coordenadas x e y de um ponto, as variáveis nomeadas tempo e temperatura podem ser mais simbólicascas do que x e y.

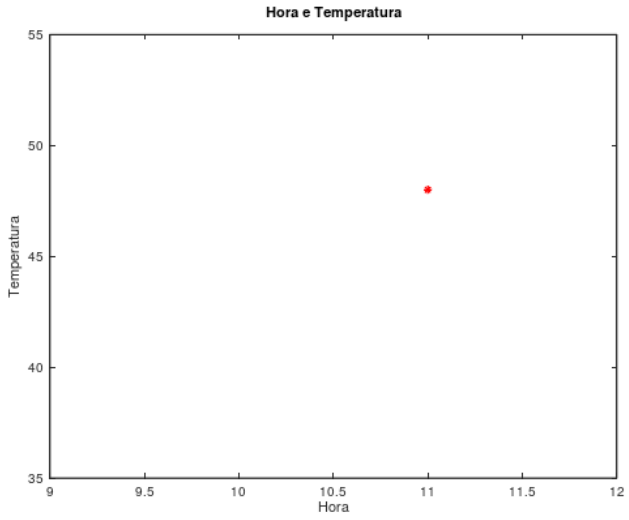

FIGURA 3.2 Plotagem de um ponto de dado

PRÁTICA 3.4

Modifique o ponto de plotagem do *script* para solicitar ao usuário a hora e a temperatura e defina os eixos com base nesses valores.

Para plotar mais de um ponto, os vetores x e y são criados para armazenar os valores dos pontos (x, y). Por exemplo, para traçar os pontos

```
(1, 1)
(2, 5)
(3, 3)
(4, 9)
(5, 11)
(6, 8)
```

primeiro, é criado um vetor x que possui os valores de x (como eles variam de 1 a 6 em passos de 1, o operador de dois pontos pode ser usado) e então um vetor y é criado com os valores de y. Os itens a seguir criarão (na Janela de Comandos) os vetores x e y e, em seguida, plotarão os pontos (veja a Figura 3.3).

```
>> x = 1: 6;
>> y = [1 5 3 9 11 8];
>> plot(x, y)
```

Note que os pontos são traçados com linhas retas desenhadas no meio. Além disso, os eixos são configurados de acordo com os dados; por exemplo, os valores de x variam de 1 a 6 e os valores de y de 1 a 11, assim é como os eixos são configurados.

Além disso, observe que, nesse caso, os valores x são os índices do vetor y (o vetor y tem seis valores, portanto os índices são repetidos de 1 a 6). Quando este é o caso, não é necessário criar o vetor x. Por exemplo,

```
>> plot(y)
```

irá traçar exatamente a mesma figura sem usar um vetor x.

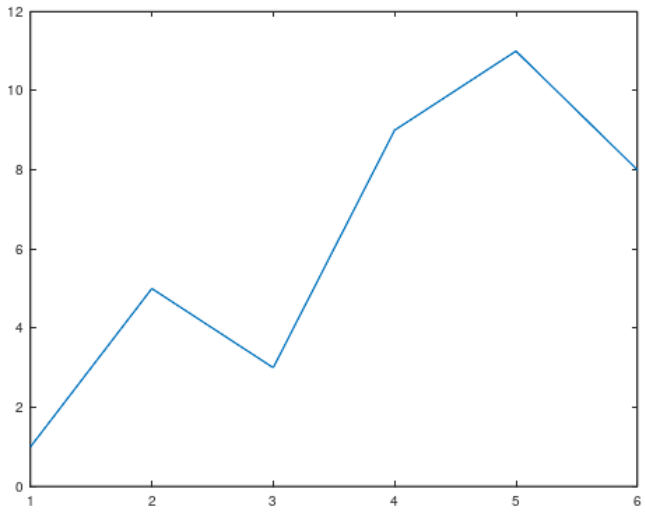

FIGURA 3.3 Plotagem dos pontos dos dados de vetores

3.5.1.1 Personalizando um gráfico: cor, tipos de linha, tipos de marcador

Os gráficos podem ser feitos na Janela de Comandos, como mostrado aqui, se forem realmente simples. No entanto, muitas vezes é desejável personalizar o gráfico com rótulos, títulos e assim por diante, portanto, faz mais sentido fazer isso em um *script*. Usando **help** para a função **plot** mostrará as várias opções, como os tipos de linhas e cores. No *script* anterior $plotaumponto$, a string "r*" especificou um asterisco vermelho para o tipo do ponto. O LineSpec, ou especificação de linha, pode especificar até três propriedades diferentes em uma *string*, incluindo a cor, o tipo de linha e o símbolo ou marcador usado para os pontos de dados.

As cores possíveis são:

```
b  azul
c  ciano
g  verde
k  preto
m  magenta
r  vermelho
w  branco
y  amarelo
```

O caractere único listado acima ou o nome completo da cor podem ser usados na *string* para especificar a cor. Os símbolos de plotagem, ou marcadores, que podem ser usados são:

```
o  círculo
d  losango
h  hexagrama
p  pentagrama
+  mais
.  ponto
s  quadrado
*  asterisco
v  triângulo para baixo
<  triângulo para a esquerda
>  triângulo retângulo
^  triângulo para cima
x  letra x
```

Os tipos de linha também podem ser especificados pelo seguinte:

```
--  tracejadas
-.  traços e pontos
:   pontilhado
-   linha sólida
```

Se nenhum tipo de linha for especificado, uma linha sólida será traçada entre os pontos, conforme visto no último exemplo.

3.5.2 Funções de plotagem relacionadas simples

Outras funções que são úteis na personalização de gráficos incluem **clf**, **figure**, **hold**, **legend** e **grid**. Descrições breves dessas funções são dadas aqui; use **help** para descobrir mais sobre elas.

clf limpa a Janela de Figura removendo tudo dela.

figure cria uma nova Janela de Figura vazia quando chamada sem quaisquer argumentos. Chamando-a como **figura(n)**, onde n é um número inteiro, é uma maneira de criar e manter várias Janelas de Figura e de referenciar cada individualmente.

hold é uma função alternadora que congela o gráfico atual na Janela de Figura, para que novos gráficos sejam sobrepostos no atual. Apenas **hold** por si só já é uma alternadora, por isso, chamar esta função uma vez, a ativa, e então, da próxima vez, a desliga. Alternativamente, os comandos **hold on** e **hold off** podem ser usados.

legend exibe numa caixa de legenda na Janela de Figura, *strings* passadas para ela na ordem das plotagens na Janela de Figura.

grid exibe linhas de grade em um gráfico. Chamada por si só, é uma função alternadora que liga (exibe) e desliga (apaga) as linhas de grade. Alternativamente, os comandos **grid on** e **grid off** podem ser usados.

Além disso, existem muitos tipos de plotagem. Veremos mais no Capítulo 11, mas outro tipo de plotagem simples é um ***gráfico de barras***.

Por exemplo, o *script* a seguir cria duas janelas separadas. Primeiro, limpa a Janela de Figura. Em seguida, cria um vetor x e dois vetores y diferentes ($y1$ e $y2$). Na primeira Janela de Figura, os valores $y1$ são plotados usando um gráfico de barras. Na segunda Janela de Figura, ele plota os valores de $y1$ como linhas pretas, coloca **hold on** para que o próximo gráfico seja sobreposto e plota os valores de $y2$ com círculos pretos. Ele também coloca uma legenda nesse gráfico e usa uma grade. Rótulos e títulos são omitidos neste caso, pois são dados genéricos.

A execução do *script* seguinte produzirá duas janelas separadas. Se não houver nenhuma outra Janela de Figura ativa, a primeira, que é o gráfico de barras, estará na que está intitulada “Figure 1” no MATLAB. O segundo será na “Figure 2”. Veja a Figura 3.4 para ambos os gráficos.

plota2figuras.m

```
% Esse cria 2 plotagens diferentes, em 2 diferentes Janelas
% de Figura, para demonstrar alguns recursos de plotagem

clf
x = 1:5; % Não é necessário
y1 = [2 11 6 9 3];
y2 = [4 5 8 6 2];
% Coloca um gráfico de barras na Figura 1
figure(1)
bar(x, y1)
% Coloca plotagens usando diferentes valores y em um gráfico
% com uma legenda
figure(2)
plot(x, y1, 'k')
hold on
plot(x, y2, 'ko')
grid on
legend('y1', 'y2')
```

(a) **(b)**

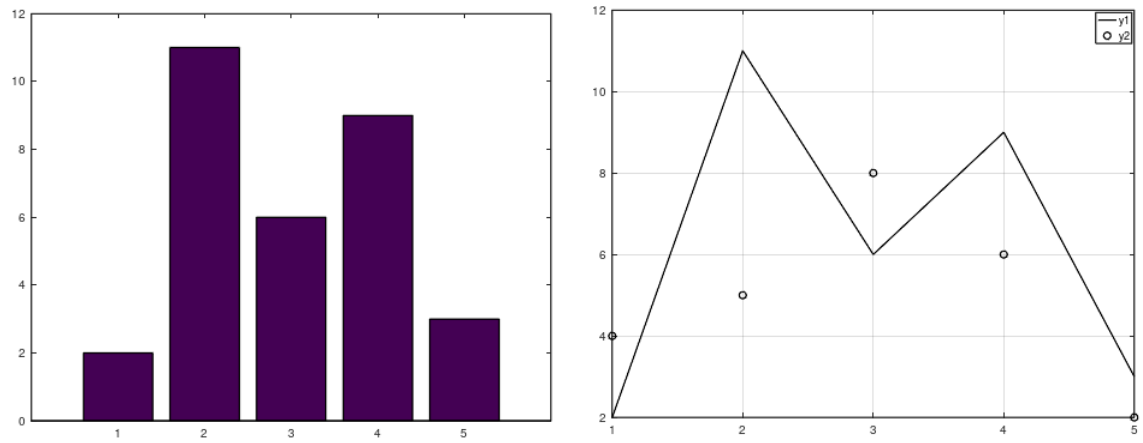

FIGURA 3.4 (a) Gráfico de barras produzido pelo *script*. (b) Plotagem produzida pelo *script*, com grade e legenda.

Observe que o primeiro e o último pontos estão nos eixos, o que dificulta a visão. É por isso que a função **axis** é usada com freqüência, pois cria espaço ao redor dos pontos para que todos sejam visíveis.

PRÁTICA 3.5

Modifique o *script* $plota2figuras$ usando a função **axis** para que todos os pontos sejam facilmente vistos.

A capacidade de passar um vetor para uma função e fazer com que a função avalie todos os elementos do vetor pode ser muito útil na criação de gráficos. Por exemplo, o *script* a seguir exibe graficamente a diferença entre as funções **sin** e **cos**:

`senoecosseno.m`

```
% Este script plota sin(x) e cos(x) na mesma Janela de Figura
% para valores de x variando de 0 a 2*pi

clf
x = 0:2*pi/40:2*pi;
y = sin(x);
plot(x, y, 'ro')
hold on
y = cos(x);
plot(x, y, 'b+')
legend('seno', 'cosseno')
xlabel('x')
ylabel('sin(x) ou cos(x)')
title('seno e cosseno em um gráfico')
```

O *script* cria um vetor x; iterando todos os valores de 0 a $2 * \pi$ em passos de $2 * \pi/40$, dá pontos suficientes para obter um bom gráfico. Em seguida, ele localiza o seno de cada valor x e plota esses pontos usando círculos vermelhos. O comando **hold on** congela o gráfico na Janela de Figura, de modo que a próxima plotagem o sobreporá. Em seguida, ele encontra o cosseno de cada valor x e plota esses pontos usando símbolos azuis mais (+). A função de **legend** cria uma legenda; a primeira string é emparelhada com a primeira plotagem e a segunda sequência com a segunda plotagem. A execução deste *script* produz o gráfico visto na Figura 3.5.

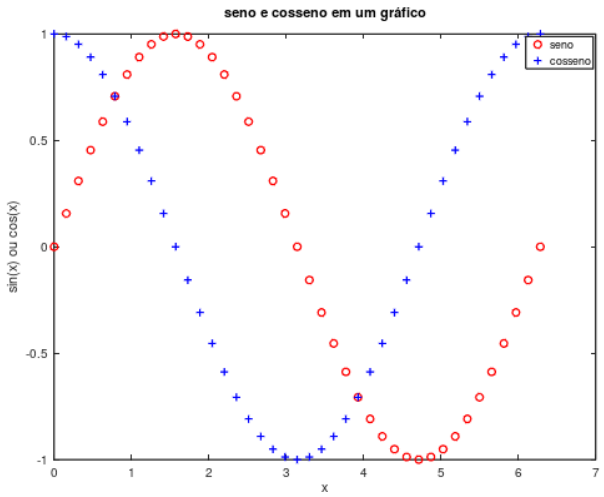

FIGURA 3.5 plotagem de **sin** e **cos** na Janela de Figura com uma legenda

Observe que, em vez de usar o **hold on**, ambas as funções poderiam ter sido plotadas usando uma única chamada para a função **plot**:

```
plot(x, sin(x), 'ro', x, cos(x), 'b+')
```

PRÁTICA 3.6

Escreva um *script* que plote **exp(x)** e **log(x)** para valores de x variando de 0 a 3.5.

3.6 INTRODUÇÃO A ENTRADA/SAÍDA DE ARQUIVOS (CARREGAR E SALVAR)

Em muitos casos, a entrada para um *script* virá de um arquivo de dados que foi criado por outra fonte. Além disso, é útil poder armazenar a saída em um arquivo externo que possa ser manipulado e/ou impresso posteriormente. Nesta seção, os métodos mais simples usados para ler de um arquivo de dados externo e também para gravar em um arquivo de dados externo serão mostrados.

Existem basicamente três operações diferentes ou ***modos*** nos arquivos. Arquivos podem ser:

- ler de
- escrever em
- acrescentar a.

Escrever em um arquivo significa gravar em um arquivo desde o início. ***Acrescentar a um arquivo*** também estará gravando, mas começando no final do arquivo, em vez do início. Em outras palavras, acrescentar a um arquivo significa adicionar ao que já estava lá.

Existem muitos tipos diferentes de arquivos, que usam diferentes extensões de nome de arquivo. Por enquanto, vamos mantê-lo simples e apenas trabalhar com arquivos $.dat$ ou $.txt$ ao trabalhar com arquivos de dados ou texto. Existem vários métodos para ler arquivos e gravar em arquivos; Por enquanto, usaremos a função **load** para ler e a função **save** para gravar em arquivos. Mais tipos de arquivos e funções para manipulá-los serão discutidos no Capítulo 9.

3.6.1 Escrevendo dados em um arquivo

O comando **save** pode ser usado para gravar dados de uma matriz em um arquivo de dados ou para anexar a um arquivo de dados. O formato é:

```
save nome_do_arquivo nome_da_variável_matriz -ascii
```

O qualificador "-ascii" é usado ao criar um arquivo de texto ou dados. Por exemplo, o comando seguinte cria uma matriz e, em seguida, salva os valores da variável matriz em um arquivo de dados chamado "$arquivodeteste.dat$":

```
>> minhamat = rand(2, 3)
minhamat =
   0.57482   0.91626   0.14491
   0.11360   0.86674   0.95365
>> save arquivodeteste.dat minhamat -ascii
```

Isso cria um arquivo chamado "$arquivodeteste.dat$" que armazena os números:

```
   0.57482   0.91626   0.14491
   0.11360   0.86674   0.95365
```

O comando **type** pode ser usado para exibir o conteúdo do arquivo; note que a notação científica é usada:

```
>> type arquivodeteste.dat
 5.74816537e-001 9.16257536e-001 1.44909486e-001
 1.13602425e-001 8.66736658e-001 9.53648367e-001
```

Observe que, se o arquivo já existir, o comando **save** sobrescreverá o arquivo; **save** sempre escreve desde o início de um arquivo.

3.6.2 Acrescentando Dados a um Arquivo de Dados

Uma vez que um arquivo de texto existe, dados podem ser anexados a ele. O formato é o mesmo que o anterior, com a adição do qualificador "-append". Por exemplo, o seguinte comando cria uma nova matriz aleatória e a anexa ao arquivo recém-criado:

```
>> mat2 = rand(3, 3)
mat2 =
   0.95089   0.34439   0.87456
   0.46497   0.50845   0.37631
   0.80524   0.69820   0.50147
>> save o arquivodeteste.dat mat2 -ascii -append
```

Isso resulta no arquivo "$arquivodeteste.dat$" contendo o seguinte:

```
   0.57482   0.91626   0.14491
   0.11360   0.86674   0.95365
   0.95089   0.34439   0.87456
   0.46497   0.50845   0.37631
   0.80524   0.69820   0.50147
```

PRÁTICA 3.7

Solicite ao usuário o número de linhas e colunas de uma matriz, crie uma matriz com muitas linhas e colunas de inteiros aleatórios e grave-a em um arquivo.

3.6.3 Lendo de um Arquivo

A leitura de um arquivo é realizada usando-se **load**. Uma vez que um arquivo é criado (como no exemplo precedente), ele pode ser lido em uma variável matriz. Se o arquivo for um arquivo de dados, o comando **load** lerá o arquivo "nomedoarquivo.ext" (por exemplo, a extensão pode ser .dat) e criará uma matriz com o mesmo nome do arquivo. Por exemplo, se o arquivo de dados "$arquivodeteste.dat$" tivesse sido criado como mostrado na seção anterior, isso seria lido e armazenado o resultado em uma variável de matriz chamada $arquivodeteste$:

```
>> clear
>> load arquivodeteste.dat
>> who
Variables in the current scope:
arquivodeteste
>> arquivodeteste
arquivodeteste =
   0.57482   0.91626   0.14491
   0.11360   0.86674   0.95365
   0.95089   0.34439   0.87456
   0.46497   0.50845   0.37631
   0.80524   0.69820   0.50147
```

O comando **load** só funciona se houver o mesmo número de valores em cada linha, de forma que os dados possam ser armazenados em uma matriz, e o comando **save** só grava de uma matriz para um arquivo. Se esse não for o caso, as funções de Entrada/Saída de arquivo de baixo nível devem ser usadas; estas serão discutidas no Capítulo 9.

Nota

Embora tecnicamente uma matriz de qualquer tamanho possa ser anexada a esse arquivo de dados, para poder lê-lo novamente em uma matriz, teria que haver o mesmo número de valores em cada linha (ou, em outras palavras, o mesmo número de colunas).

3.6.3.1 Exemplo: Carregar de um Arquivo e Plotar os Dados

Por exemplo, um arquivo chamado "$horatemp.dat$" armazena duas linhas de dados. A primeira linha é a hora do dia e a segunda linha é a temperatura registrada em cada um desses momentos. O primeiro valor 0 para a hora, representa meia-noite. Por exemplo, o conteúdo do arquivo pode ser:

```
0     3     6     9     12    15    18    21
55.5  52.4  52.6  55.7  75.6  77.7  70.3  66.6
```

O *script* a seguir carrega os dados do arquivo em uma matriz chamada $horatemp$. Em seguida, ele separa a matriz em vetores para as horas e as temperaturas e, em seguida, plota os dados usando asteriscos pretos ($*$).

horaetemp.m

```
% Este lê os dados de hora e temperatura de um dia,
% de um arquivo e plota os dados

load horatemp.dat
% As horas estão na primeira linha, temperaturas na segunda
% linha
hora = horatemp(1, :);
temp = horatemp(2, :);
% Plota os dados e rotula o gráfico
plot(hora, temp, 'k*')
xlabel('Hora')
ylabel('Temperatura')
title('Temperaturas do Dia')
```

A execução do *script* produz o gráfico visto na Figura 3.6.

Note que é difícil ver o ponto na hora 0 quando ele cai no eixo y. A função **axis** pode ser usada para alterar os eixos dos padrões mostrados aqui.

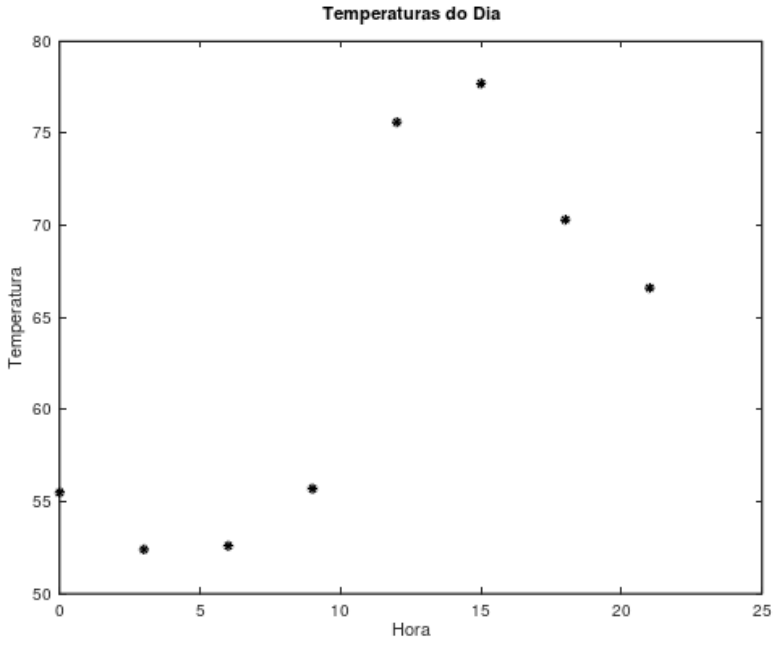

FIGURA 3.6 Plotagem de dados de temperatura a partir de um arquivo

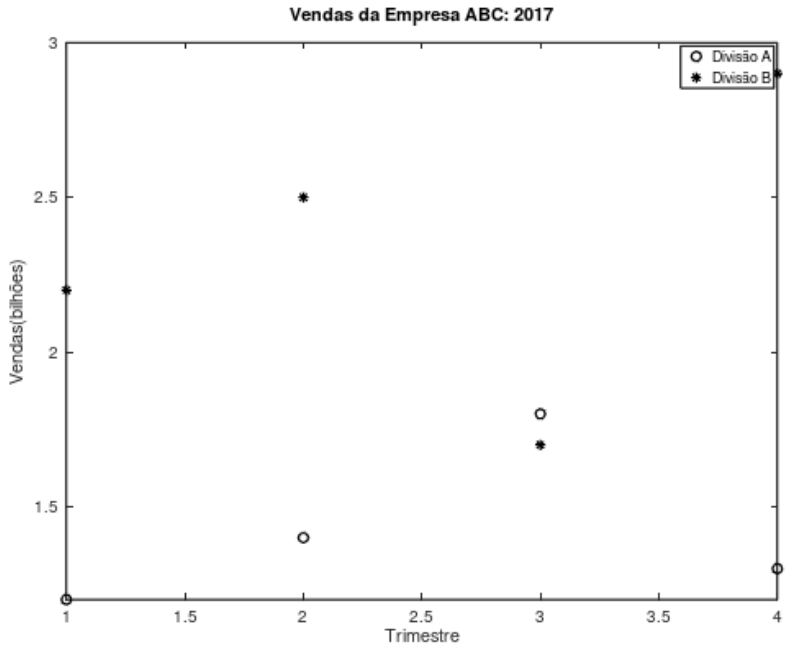

FIGURA 3.7 Plotagem dos dados de vendas a partir de um arquivo

Para criar o arquivo de dados, o "Editor" do MATLAB pode ser usado; não é necessário criar uma matriz e salvá-la em um arquivo. Em vez disso, basta digitar os números em um novo arquivo de *script* e Salvar como "$horatemp.dat$", certificando-se de que a pasta atual está definida.

PRÁTICA 3.8

As vendas (em bilhões) para duas divisões separadas da Empresa ABC para cada um dos quatro trimestres de 2013 são armazenadas em um arquivo chamado "$vendasabc.dat$":

```
1.2 1.4 1.8 1.3
2.2 2.5 1.7 2.9
```

- Primeiro, crie este arquivo (basta digitar os números no Editor e Salvar como "$vendasabc.dat$").
- Então, escreva um *script* que
  - carrega os dados do arquivo em uma matriz
  - separa esta matriz em 2 vetores
  - cria a plotagem vista na Figura 3.7 (que usa círculos e asteeriscos pretos como símbolos de plotagem).

PERGUNTA RÁPIDA!

Às vezes, os arquivos não estão no formato desejado. Por exemplo, um arquivo "$resultadosexp.dat$" foi criado com alguns resultados experimentais, mas a ordem dos valores é invertida no arquivo:

```
  4   53.4
  3   44.3
  2   50.0
  1   55.5
```

Como poderíamos criar um novo arquivo que inverte a ordem?

Resposta

Podemos carregar este arquivo em uma matriz, usar a função **flipud** para "inverter" a matriz de cima para baixo e depois salvar essa matriz em um novo arquivo:

```
>> load resultadosexp.dat
>> resultadosexp
resultadosexp =
    4.0000   53.4000
    3.0000   44.3000
    2.0000   50.0000
    1.0000   55.5000
>> ordemcorreta = flipud(resultadosexp)
ordemcorreta =
    1.0000   55.5000
    2.0000   50.0000
    3.0000   44.3000
    4.0000   53.4000
>> save novaordem.dat ordemcorreta -ascii
```

3.7 FUNÇÕES DEFINIDAS PELO USUÁRIO QUE RETORNAM UM ÚNICO VALOR

Já vimos o uso de muitas funções no MATLAB. Usamos muitas funções internas, como **sin**, **fix**, **abs** e **double**. Nesta seção, as funções definidas pelo usuário serão introduzidas. Essas são funções que o programador define e usa na Janela de Comandos ou em um *script*.

Existem vários tipos diferentes de funções. Por enquanto, vamos nos concentrar no tipo de função que calcula e retorna um único resultado. Outros tipos de funções serão introduzidos no Capítulo 6.

Primeiro, vamos revisar algumas das coisas que já sabemos sobre funções, incluindo o uso de funções internas. Embora, a essa altura, o uso dessas funções seja simples, explicações serão dadas com algum detalhe aqui para comparar e contrastar com o uso de funções definidas pelo usuário.

A função **length** é um exemplo de uma função interna que calcula um único valor; ela retorna o comprimento de um vetor. Como um exemplo,

```
length(vet)
```

é uma expressão que representa o número de elementos no vetor vet. Essa expressão pode ser usada na Janela de Comandos ou em um *script*. Normalmente, o valor retornado dessa expressão pode ser atribuído a uma variável:

```
>> vet = 1:3:10;
>> cv = length(vet)
cv =
    4
```

Alternativamente, o comprimento do vetor pode ser impresso:

```
>> fprintf('O comprimento do vetor é %d\n', length(vet))
O comprimento do vetor é 4
```

A ***chamada de função*** para a função **length** consiste no nome da função, seguido pelo ***argumento*** entre parênteses. A função recebe como entrada o argumento e ***retorna*** um resultado. O que acontece quando a chamada para a função é encontrada é que o controle é passado para a própria função (em outras palavras, a função começa a ser executada). O(s) argumento(s) também é(são) passado(s) para a função.

A função executa suas instruções e faz o que tem que fazer (o conteúdo real das funções internas não é geralmente conhecido ou visto pelo usuário) para determinar o número de elementos do vetor. Como a função está calculando um único valor, esse resultado é retornado e se torna o valor da expressão. O controle também é passado de volta para a expressão que a chamou em primeiro lugar, que então continua (por exemplo, no primeiro exemplo, o valor seria então atribuído à variável cv e, no segundo exemplo, o valor foi impresso).

3.7.1 Definições das Funções

Existem diferentes maneiras de organizar *scripts* e funções, mas, por enquanto, todas as funções que escrevemos serão armazenadas em um arquivo-M separado, e é por isso que elas são comumente chamadas de "funções de arquivo-M". Embora, para digitar funções no "Editor", é possível escolher a seta para baixo "New" e, em seguida, "Function", será mais fácil digitar a função, escolhendo "New Script" (isso ignora os padrões que são fornecidos quando você escolhe "Function").

Uma função no MATLAB que retorna um único resultado consiste no seguinte.

- O cabeçalho da função (a primeira linha), composto por:
  - a palavra reservada **function**
  - o nome do argumento de saída seguido pelo operador de atribuição (=), conforme a função **retorna** um resultado
  - o nome da função (importante – este deve ser o mesmo que o nome do arquivo-M no qual esta função está armazenada, para evitar confusão)
  - os argumentos de entrada entre parênteses, que correspondem aos argumentos que são passados para a função, na chamada de função.
- Um comentário que descreve o que a função faz (isso é exibido quando **help** é usado).
- O corpo da função, que inclui todas as instruções e que, eventualmente, deve colocar um valor no argumento de saída.
- **end** no final da função (note que isso não é necessário em muitos casos nas versões atuais do MATLAB, mas é considerado bom estilo de qualquer maneira).

A forma geral de uma ***definição de função*** para uma função que calcula e retorna um valor se parece com isto:

nomedafuncao.m

```
function argumentodesaida = nomedafuncao(argumentos de entrada)
% Comentário descrevendo a função

Comandos aqui; % estes devem incluir colocar um valor no
               % argumento de saída

end % fim da função
```

Por exemplo, o seguinte é uma função chamada $calcarea$ que calcula e retorna a área de um círculo; ele é armazenado em um arquivo chamado $calcarea.m$.

calcarea.m

```
function area = calcarea(raio)
% calcarea calcula a área de um círculo
% Formato da chamada: calcarea(raio)
% Retorna a área

area = pi*raio*raio;
end
```

O raio de um círculo é passado para a função para o argumento de entrada $raio$; a função calcula a área desse círculo e a armazena na área do argumento de saída.

No cabeçalho da função, temos a palavra reservada **function**, seguido do argumento de saída $area$, seguido pelo operador de atribuição =, depois, o nome da função (igual ao nome do arquivo-M) e, em seguida, o argumento de entrada $raio$, que é o raio do círculo. Como há um argumento de saída no cabeçalho da função, em algum lugar no corpo da função, devemos colocar um valor nesse argumento de saída. Assim, é como um valor é retornado da função. Neste caso, a função é simples e tudo o que temos a fazer é atribuir ao argumento de saída $area$ o valor da constante interna pi multiplicada pelo quadrado do argumento de entrada $raio$.

A função pode ser exibida na Janela de Comandos usando o comando **type**.

```
>> type calcarea
function area = calcarea(raio)
% calcarea calcula a área de um círculo
% Formato da chamada: calcarea(raio)
% Retorna a área

area = pi*raio*raio;
end
```

Nota

Muitas das funções no MATLAB são implementadas como funções de arquivo-M; estes também podem ser exibidos usando o **type**.

3.7.2 Chamando uma Função

O seguinte é um exemplo de uma chamada para esta função na qual o valor retornado é armazenado na variável padrão ans:

```
>> calcarea(4)
ans =
   50.2655
```

Tecnicamente, chamar a função é feito com o nome do arquivo no qual a função reside. Para evitar confusão, é mais fácil dar à função o mesmo nome que o nome do arquivo, deste modo é que será apresentado neste livro. Neste exemplo, o nome da função é $calcarea$ e o nome do arquivo é $calcarea.m$. O resultado retornado dessa função também pode ser armazenado em uma variável com uma instrução de atribuição; o nome pode ser o mesmo que o nome do argumento de saída da própria função, mas isso não é necessário. Então, por exemplo, qualquer uma dessas atribuições estaria bem:

```
>> area = calcarea(5)
area =
   78.5398
>> miinhaarea = calcarea(6)
myarea =
   113.0973
```

A saída também pode ser suprimida ao chamar a função:

```
>> minhaa = calcarea(5.2);
```

O valor retornado da função $calcarea$ também pode ser impresso usando **disp** ou **fprintf**:

```
>> disp(calcarea(4))
   50.2655
>> fprintf ('A área é %.1f\n', calcarea(4))
A área é 50.3
```

Nota

A impressão não é feita na própria função; em vez disso, a função retorna a área e, em seguida, uma instrução de saída pode imprimi-la ou exibi-la.

PERGUNTA RÁPIDA!

Poderíamos passar um vetor de raios para a função $calcarea$?

Resposta

Esta função foi escrita assumindo que o argumento era escalar, então chamá-la com um vetor, em vez disso, produziria uma mensagem de erro:

```
>> calcarea(1:3)
Error using *
Inner matrix dimentions must agree.

Error in calcarea (line 6)
area = pi*raio*raio;
```

Isso ocorre porque o $*$ foi usado para multiplicação na função, mas $.*$ deve ser usado ao multiplicar vetores elemento a elemento. Mudar isto na função permitiria escalares ou vetores serem passados para esta função:

calcareas.m

```
calcareas.m
area function = calcareas(raio)
% calcareas retorna a área de um círculo
% O argumento de entrada pode ser um vetor
% de raios
% Format: calcareas(vetorDeRaios)

area = pi * raio .* raio;
end
```

```
>> calcareas(1:3)
ans =
    3.1416   12.5664   28.2743
>> calcareas(4)
ans =
   50.2655
```

Observe que o operador $.*$ só é necessário para multiplicar o vetor de raios por ele mesmo. Multiplicar por pi é multiplicação escalar, então o operador $.*$ não é necessário ali. Nós poderíamos ter usado também:

```
area = pi * raio .^ 2;
```

Usando **help** com qualquer uma dessas funções exibe o bloco contíguo de comentários sob o cabeçalho da função (o comentário de bloco). É útil colocar o formato da chamada para a função neste comentário de bloco:

```
>> help calcarea
```

```
calcarea calcula a área de um círculo
 Formato da chamada: calcarea(raio)
 Retorna a área
```

Muitas organizações têm padrões sobre quais informações devem ser incluídas no comentário de bloco em uma função. Estes podem incluir:

- nome da função
- descrição do que a função faz
- formato da chamada de função
- descrição dos argumentos de entrada
- descrição do argumento de saída
- descrição das variáveis usadas na função
- nome e data do programador
- informações sobre revisões.

Embora este seja um excelente estilo de programação, na maior parte deste livro será omitida simplesmente para economizar espaço. Além disso, a documentação no MATLAB sugere que o nome da função deve estar em todas as letras maiúsculas no início do comentário do bloco. No entanto, isso pode ser um pouco enganoso, pois o MATLAB é sensível a maiúsculas e minúsculas e geralmente são usadas letras minúsculas para o nome real da função.

3.7.3 Chamando uma Função Definida pelo Usuário a Partir de um Script

Agora, vamos modificar nosso *script* que solicita ao usuário o raio e calcula a área de um círculo para chamar nossa função $calcarea$ para calcular a área do círculo em vez de fazer isso no $script$.

funcaoChamaCirculo.m

```
% Este script calcula a área de um círculo
% Ele solicita ao usuário o raio

raio = input('Por favor, digite o raio: ');
% Em seguida, ela chama nossa função para calcular a
% área e, em seguida, imprime o resultado
area = calcarea(raio);
fprintf('Para um círculo de raio %.2f, ', raio)
fprintf('a área é %.2f\n', area)
```

Rodando este, produzirá o seguinte:

```
>> funcaoChamaCirculo
Por favor, digite o raio: 5
Para um círculo de raio 5.00, a área é 78.54
```

3.7.3.1 Programas Simples

Neste livro, um *script* que chama função(ões) é o que nós chamaremos de programa MATLAB. No exemplo anterior, o programa consistia no *script* $funcaoChamaCirculo$ e na função que ele chama, $calcarea$. A forma geral de um programa simples, consistindo de um *script* que chama uma função para calcular e retornar um valor, se parece com o diagrama mostrado na Figura 3.8.

Também é possível que uma função chame outra (seja interna ou definida pelo usuário).

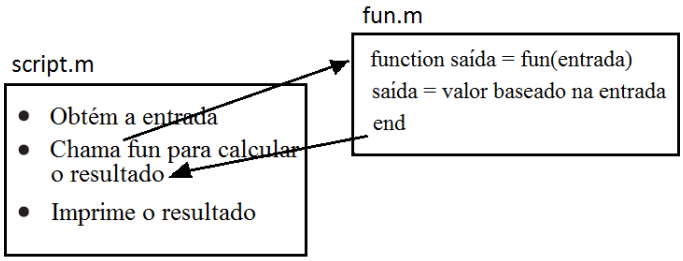

FIGURA 3.8 Forma geral de um programa simples

3.7.4 Passando Múltiplos Argumentos

Em muitos casos, é necessário passar mais de um argumento para uma função. Por exemplo, o volume de um cone é dado por

$$V = \frac{1}{3}\pi r^2 h$$

onde r é o raio da base circular e h é a altura do cone. Portanto, uma função que calcula o volume de um cone precisa do raio e da altura:

volcone.m

```
function argsaida = volcone(raio, altura)
% volcone calcula o volume de um cone
% Formato da chamada: volcone(raio, altura)
% Retorna o volume

argsaida = (pi/3)*raio .^ 2 .* altura;
fim
```

Como a função tem dois argumentos de entrada no cabeçalho da função, dois valores devem ser passados para a função quando ela é chamada. A ordem faz diferença. O primeiro valor que é passado para a função é armazenado no primeiro argumento de entrada (neste caso, $raio$) e o segundo argumento na chamada de função é passado para o segundo argumento de entrada no cabeçalho da função.

Isso é muito importante: os argumentos na chamada de função devem corresponder um a um com os argumentos de entrada no cabeçalho da função.

Aqui está um exemplo de chamar essa função. O resultado retornado da função é simplesmente armazenado na variável padrão ans.

```
>> volcone(4,6,1)
ans =
   102.2065
```

No próximo exemplo, o resultado é impresso com um formato de duas casas decimais.

```
>> fprintf('O volume do cone é %.2f\n', volcone(3, 5.5))
O volume do cone é 1.84
```

Observe que, usando os operadores de exponenciação e multiplicação de matriz, seria possível transmitir matrizes para os argumentos de entrada, desde que as dimensões sejam as mesmas.

PERGUNTA RÁPIDA!

Nada está tecnicamente errado com a seguinte função, mas o que nela não faz sentido?

`fun.m`

```
function saida = fun(a, b, c)
saida = a * b;
fim
```

Resposta

Por que passar o terceiro argumento se não for usado?

PRÁTICA 3.9

Escreva um *script* que solicite ao usuário o raio e a altura, chame a função $volcone$ para calcular o volume do cone e imprima o resultado num formato de frase legal. Assim, o programa consistirá em um *script* e na função $volcone$ que ele chama.

PRÁTICA 3.10

Para um projeto, precisamos de algum tecido para formar um retângulo. Escreva uma função $calcarearet$ que receberá o comprimento e a largura de um retângulo em centímetros como argumentos de entrada e retornará a área do retângulo. Por exemplo, a função poderia ser chamada como mostrado, em que o resultado é armazenado em uma variável e, em seguida, a quantidade de tecido necessário é impresso, arredondado para os centímetros quadrados mais próximos.

```
>> ar = calcarearet(3.1, 4.4)
ar =
   13.600
>> fprintf('Precisamos de %d centímetros quadrados.\n',...
ceil(ar))
Precisamos de 14 centímetros quadrados.
```

3.7.5 Funções com Variáveis Locais

As funções discutidas até agora foram muito simples. No entanto, em muitos casos, os cálculos em uma função são mais complicados e podem exigir o uso de variáveis extras dentro da função; estas são chamadas ***variáveis locais***.

Por exemplo, um cilindro fechado está sendo construído de um material que custa uma certa quantia em reais por metro quadrado. Escreveremos uma função que calculará e retornará o

custo do material, arredondado até metro quadrado mais próximo, para um cilindro com um determinado raio e uma determinada altura. A área total da superfície do cilindro fechado é

$$SA = 2 \pi r h + 2 \pi r^2$$

Para um cilindro com um raio de 320 centímetros, altura de 730 centímetros e custo por metro quadrado do material de R$ 4,50, o cálculo seria dado pelo seguinte algoritmo.

- Calcule a área de superfície SA = 2 * π * 320 * 730 + 2 * π * 320 * 320 centímetros quadrados.
- Converta o SA de centímetros quadradas em metros quadrados = SA / 100.
- Calcule o custo total = SA em metros quadrados * custo por metro quadrado.

A função inclui variáveis locais para armazenar os resultados intermediários.

custocil.m

```
function custototal = custocil(raio, altura, custo)
% custocil calcula o custo de construção de um
% cilindro
% Formato da chamada: custocil(raio, altura, custo)
% Retorna o custo total

% O raio e a altura estão em centímetros
% O custo é por metro quadrado

% Calcular área de superfície em centímetros quadradas
area_sup = 2 * pi * raio .* altura + 2 * pi * raio .^ 2;

% Converte a área de superfície em metro quadrado e arredonda
% para cima
area_sup_mq = ceil(area_sup/100);

% Calcular custo
custototal = area_sup_mq .* custo;
end
```

A seguir, exemplos de como chamar a função:

```
>> custocil(320, 730, 4.50)
>> custocil(320, 730, 4.50)
ans =
   95004
>> fprintf('O custo seria R$ %.2f\n', custocil(320, 730, 4.50))
O custo seria R$ 95004.00
```

3.7.6 Introdução ao Escopo

É importante entender o escopo das variáveis, que é onde elas são válidas. Mais, será descrito no Capítulo 6, mas, basicamente, as variáveis usadas em um *script* também são conhecidas na Janela de Comandos e vice-versa. Todas as variáveis usadas em uma função, no entanto, são locais para essa função. Tanto a Janela de Comandos quanto os *scripts* usam um espaço de trabalho comum, o espaço de trabalho base. Funções, no entanto, têm seus próprios espaços de trabalho. Isso significa que, quando um *script* é executado, as variáveis podem ser vistas

posteriormente na Janela da Área de Trabalho e podem ser usadas a partir da Janela de Comandos. No entanto, este não é o caso das funções.

3.8 COMANDOS E FUNÇÕES

Alguns dos comandos que usamos (por exemplo, **format**, **type**, **save** e **load**) são apenas atalhos para as chamadas de função. Se todos os argumentos a serem passados para uma função forem strings e a função não retornar nenhum valor, ela poderá ser usada como um comando. Por exemplo, os itens a seguir produzem os mesmos resultados:

```
>> type script1

raio = 5
area = pi * (raio^2)

>> type('script1')

raio = 5
area = pi * (raio^2)
```

Usar **load** como um comando cria uma variável com o mesmo nome do arquivo. Se um nome de variável diferente é desejado, é mais fácil usar a forma funcional de **load**. Por exemplo,

```
>> type coordsdoponto.dat

3.3   1.2
4     5.3

>> pontos = load('coordsdoponto.dat')
pontos =
   3.3000   1.2000
   4.0000   5.3000
```

Explorar Outros Recursos Interessantes

Observe que este capítulo serve como uma introdução a vários tópicos, a maioria dos quais será abordada com mais detalhes em capítulos futuros. Antes de chegar a esses capítulos, a seguir estão algumas coisas que você pode querer explorar.

- O comando **help** pode ser usado para ver explicações curtas de funções internas. No final, um link de página do documento também é listado. Essas páginas de documentação frequentemente têm muito mais informações e exemplos úteis. Elas também podem ser acessadas digitando “doc nomefun”, onde nomefun é o nome da função.
- Observe o formatSpec na página de documentos na função **fprintf** para obter mais formas de formatação das expressões (por exemplo, preenchendo números com zeros e imprimindo o sinal de um número).
- Use a Documentação de Pesquisa para localizar os caracteres de conversão usados para imprimir outros tipos, como inteiros sem sinal e notação exponencial.

Resumo

Armadilhas Comuns

- Escrever um nome de variável de maneiras diferentes em lugares diferentes em um *script* ou função.
- Esquecer de adicionar o segundo argumento 's' à função **input** quando a entrada de caracteres é desejada.
- Não usar o caractere de conversão correto ao imprimir.
- Confundir **fprintf** e **disp**. Lembre-se que somente o **fprintf** pode formatar.

Diretrizes de Estilo de Programação

- Especialmente para *scripts* e funções mais longs, comece escrevendo um algoritmo.
- Use comentários para documentar *scripts* e funções, da seguinte maneira:
  - um bloco de comentários contíguos no topo para descrever um *script*
  - um bloco de comentários contíguos sob o cabeçalho da função para funções
  - comentários em qualquer arquivo-M (*script* ou função) para descrever cada seção.
- Certifique-se de que a linha de comentários "H1" tenha informações úteis.
- Use as diretrizes de estilo padrão da sua organização para comentários de bloco.
- Use nomes de identificadores mnemônicos (nomes que fazem sentido, por exemplo, raio em vez de xyz) para nomes de variáveis e nomes de arquivos.
- Torne toda a saída informativa e fácil de ler.
- Coloque um caractere nova linha no final de cada sequência impressa por **fprintf** para que a próxima saída ou o *prompt* apareça na linha abaixo.
- Coloque rótulos informativos nos eixos x e y, e um título em todos os gráficos.
- Mantenha funções curtas e normalmente não mais do que uma página de comprimento.
- Suprima a saída de todas as instruções de atribuição em funções e *scripts*.
- Funções que retornam um valor normalmente não imprimem o valor; deve simplesmente ser retornado pela função.
- Use os operadores de matriz .*, ./, .\ e .^ em funções para que os argumentos de entrada possam ser matrizes e não apenas escalares.

Palavras Reservadas do MATLAB	
function	end

Funções e Comandos do MATLAB			
type	xlabel	clf	grid
input	ylabel	figure	bar
disp	title	hold	load
fprintf	axis	legend	save
plot			

Operadores MATLAB	
comentário %	bloco de comentários %{, %}

Exercícios

1. Escreva um *script* simples que calcule o volume de uma esfera oca, onde r_i é o raio interno e r_e é o raio externo. Atribua um valor a uma variável para o raio interno e também atribua um valor a outra variável para o raio externo. Em seguida, usando essas variáveis, atribua o volume a uma terceira variável. Inclua comentários no *script*.

$$\frac{4\pi}{3}\left(r_e^3 - r_i^3\right)$$

2. O peso atômico é o peso de um mol de átomos de um elemento químico. Por exemplo, o peso atômico do oxigênio é 15.9994 e o peso atômico do hidrogênio é 1.0079. Escreva um *script* que calcule o peso molecular do peróxido de hidrogênio, que consiste de dois átomos de hidrogênio e dois átomos de oxigênio. Inclua comentários no *script*. Use **help** para visualizar o comentário no seu *script*.

3. Escreva um comando **input** que indique ao usuário o nome de um elemento químico como uma *string*. Então, encontre o comprimento da *string*.

4. A função **input** pode ser usada para inserir um vetor, como:

```
>> vet = input('Digite um vetor: ')
Digite um vetor: 4:7
vet =
   4   5   6   7
```

   Experimente e descubra como o usuário pode inserir uma matriz.

5. Escreva um comando **input** que irá solicitar ao usuário um número real e armazená-lo em uma variável. Em seguida, use a função **fprintf** para imprimir o valor dessa variável usando duas casas decimais.

6. Experimente, na Janela de Comandos, com o uso da função **fprintf** para números reais. Anote o que acontece para cada um. Use **fprintf** para imprimir o número real 12345.6789

   - sem especificar qualquer largura de campo
   - com uma largura de campo de 10 com 4 casas decimais
   - com uma largura de campo de 10 com 2 casas decimais
   - com uma largura de campo de 6 com 4 casas decimais
   - com uma largura de campo de 2 com 4 casas decimais.

7. Experimente, na Janela de Comandos, o uso da função **fprintf** para números inteiros. Anote o que acontece para cada um. Use o **fprintf** para imprimir o inteiro 12345

   - sem especificar qualquer largura de campo
   - com uma largura de campo de 5
   - com uma largura de campo de 8
   - com uma largura de campo de 3.

8. No sistema métrico, a vazão de fluido é medido em metros cúbicos por segundo (m^3/s). Um pé cúbico por segundo ($pé^3/s$) é equivalente a 0.028 m^3/s. Escreva um *script* intitulado $vazao$ que irá solicitar ao usuário o fluxo em metros cúbicos por segundo e imprimirá a vazão equivalente em pés cúbicos por segundo. Aqui está um exemplo de execução do *script*. Seu *script* deve produzir a saída exatamente no mesmo formato que este:

```
>> vazao
```

```
Digite o fluxo em m^3/s: 15.2
Uma vazão de 15.200 metros por segundo
é equivalente a 542.857 pés por segundo
```

9. Escreva um *script* chamado *ecoastring* que solicitará ao usuário uma *string* e fará o eco da string entre aspas:

```
>> ecoastring
Digite sua string: Olá
Sua string foi: 'Olá'
```

10. Se os comprimentos de dois lados de um triângulo e o ângulo entre eles forem conhecidos, o comprimento do terceiro lado pode ser calculado. Dados os comprimentos dos dois lados (b e c) de um triângulo e o ângulo α entre eles em graus, o terceiro lado a é calculado da seguinte forma:

$$a^2 = b^2 + c^2 - 2\ b\ c\ \cos(\alpha)$$

Escreva um *script terceirolado* que avise o usuário e leia os valores para b, c e α (em graus) e, em seguida, calcule e imprima o valor de a com três casas decimais. O formato da saída do *script* deve ser exatamente assim:

```
>> terceirolado
Digite o primeiro lado: 2.2
Digite o segundo lado: 4.4
Digite o ângulo entre eles: 50

O terceiro lado é 3.429
```

Para ser mais prático, escreva uma função para calcular o terceiro lado, para que o *script* chame essa função.

11. Escreva um script que solicite ao usuário um caractere e imprima-o duas vezes; uma vez justificado à esquerda em uma largura de campo de 5 e novamente justificado à direita em uma largura de campo de 3.

12. Escreva um *script lumin* de que calcule e imprima a luminosidade L de uma estrela em Watts. A luminosidade L é dada por $L = 4\ \pi\ d^2\ b$, onde d é a distância do sol em metros e b é o brilho em Watts/metros^2. Aqui está um exemplo de execução do *script*:

```
>> lumin
Este script irá calcular a luminosidade de uma estrela.
Quando solicitado, digite a distância da estrela ao sol
em metros, e seu brilho em W/metros quadrados.

Digite a distância: 1.26e12
Digite o brilho: 2e-17
A luminosidade desta estrela é 399007399,75 watts
```

13. Na engenharia mecânica, um vetor é um conjunto de números que indicam magnitude e direção. Unidades como velocidade e força são grandezas vetoriais. Um exemplo de um vetor pode ser $< 2.34, 4.244, 5.323 >$ metros/segundo. Este vetor descreve a velocidade de uma partícula em um certo ponto no espaço tridimensional, <x, y, z>. Na resolução de problemas relacionados a vetores, é útil conhecer o vetor unitário de uma determinada medida. Um vetor unitário é um vetor que tem uma certa direção, mas uma magnitude de 1. A equação para um vetor unitário no espaço tridimensional é:

$$\vec{u} = \frac{\langle x, y, z \rangle}{\sqrt{x^2 + y^2 + z^2}}$$

Escreva um *script* que solicite ao usuário os valores x, y e z e calcule o vetor unitário.

14. Escreva um *script* que atribua valores para a coordenada x e para a coordenada y de um ponto e plote isso usando um + verde.

15. Plote **sin(x)** para valores x variando de 0 a π (em separado na Janela de Figura):

    - usando 10 pontos neste intervalo
    - usando 100 pontos neste intervalo.

16. Propriedades atmosféricas, como temperatura, densidade do ar e pressão do ar, são importantes na aviação. Crie um arquivo que armazene temperaturas em graus Kelvin em várias altitudes. As altitudes estão na primeira coluna e as temperaturas na segunda. Por exemplo, pode parecer com isso:

    ```
    1000    288
    2000    281
    3000    269
    ```

    Escreva um *script* que carregue esses dados em uma matriz, separe-os em vetores e, em seguida, plote os dados com rótulos de eixo apropriados e um título.

17. Gere um inteiro randômico n, crie um vetor dos inteiros de 1 a n em passos de 2, calcule os quadrados deles e plote os quadrados.

18. Crie uma matriz 3 x 6 de inteiros randômicos, cada um no intervalo de 50 a 100. Escreva isso em um arquivo chamado $arquivorand.dat$. Em seguida, crie uma nova matriz de inteiros randômicos, mas desta vez uma matriz 2 x 6 de inteiros randômicos, cada um no intervalo de 50 a 100. Anexe essa matriz ao arquivo original. Então, leia o arquivo em (que será para uma variável chamada $randfile$), apenas para ter certeza de que funcionou!

19. Na hidrologia, os hetógrafos são usados para exibir a intensidade da chuva durante uma tempestade. A intensidade pode ser a quantidade de chuva por hora, registrada a cada hora por um período de 24 horas. Crie seu próprio arquivo de dados para armazenar a intensidade em milímetros por hora a cada hora por 24 horas. Use um ***gráfico de barras*** para exibir as intensidades.

20. Uma peça está sendo torneada em um torno. O diâmetro da peça deve ser de 20.000 mm. O diâmetro é medido a cada 10 minutos e os resultados são armazenados em um arquivo chamado $diampeca.dat$. Crie um arquivo de dados para simular isso. O arquivo armazenará o tempo em minutos e o diâmetro de cada momento. Plote os dados.

21. Um arquivo "$numsflut.dat$" foi criado para uso em uma experiência. No entanto, ele contém números flutuantes (reais) e o que é desejado são números inteiros. Além disso, o arquivo não está exatamente no formato correto; os valores são armazenados em coluna em vez de em linha. Por exemplo, se o arquivo contiver o seguinte:

    ```
    90.5792   27.8498   97.0593
    12.6987   54.6882   95.7167
    91.3376   95.7507   48.5376
    63.2359   96.4889   80.0280
     9.7540   15.7613   14.1886
    ```

o que é realmente desejado é:

```
91   13   91   63   10
28   55   96   96   16
97   96   49   80   14
```

Crie o arquivo de dados no formato especificado. Escreva um *script* que leia o arquivo $numsflut.dat$ em uma matriz, arredonde os números e escreva a matriz no formato desejado para um novo arquivo chamado $numsint.dat$.

22. Crie um arquivo chamado "$testtan.dat$" composto de duas linhas com três números reais em cada linha (alguns negativos, alguns positivos, no intervalo de 1 a 3). O arquivo pode ser criado a partir do "Editor" ou salvo de uma matriz. Em seguida, carregue o arquivo em uma matriz e calcule a tangente de cada elemento na matriz resultante.

23. Um arquivo chamado "$tempalta.dat$" foi criado há algum tempo, que armazena, em cada linha, um ano seguido pela temperatura mais alta em um local específico para cada mês daquele ano. Por exemplo, o arquivo pode ter esta aparência:

```
89   42   49   55   72   63   68   77   82   76   67
90   45   50   56   59   62   68   75   77   75   66
91   44   43   60   60   60   65   69   74   70   70
etc.
```

Como pode ser visto, apenas dois dígitos foram usados para o ano (o que era comum no último século). Escreva um *script* que leia esse arquivo em uma matriz, crie uma nova matriz que armazene os anos corretamente como $19xx$ e, em seguida, grave isso em um novo arquivo chamado "$tempporano.dat$". (Dica: adicione 1900 a toda a primeira coluna da matriz). Tal arquivo, por exemplo, ficaria assim:

```
1989   42   49   55   72   63   68   77   82   76   67
1990   45   50   56   59   62   68   75   77   75   66
1991   44   43   60   60   60   65   69   74   70   70
etc.
```

24. Escreva uma função $calcarearet$ que irá calcular e retornar a área de um retângulo. Passe para a função, como argumentos de entrada, o comprimento e a largura.

25. Escreva uma função $perim$ que receba o raio r de um círculo e calcule e retorne o perímetro P do círculo ($\mathrm{P} = 2\ \pi\ \mathrm{r}$). Aqui estão alguns exemplos de uso da função:

```
>> perimetro = perim(5.3)
perímetro =
   33.3009
>> fprintf ('O perímetro é %.1f\n', perim(4))
O perímetro é 25.1
>> help perim
Calcula o perímetro de um círculo
```

26. Fontes renováveis de energia, como a biomassa, estão ganhando cada vez mais atenção. Unidades de energia de biomassa incluem megawatt horas (MWh) e gigajoules (GJ). Um MWh é equivalente a 3.6 GJ. Por exemplo, 1 metro cúbico de aparas de madeira produz 1 MWh.

Escreva uma função mwh_para_gj que irá converter de MWh para GJ.

27. A velocidade de uma aeronave é tipicamente dada em milhas/hora ou em metros/segundo. Escreva uma função que receberá um argumento de entrada, a velocidade de um avião em milhas por hora e retornará a velocidade em metros por segundo. Os fatores de conversão relevantes são: 1 hora = 3600 segundos, 1 milha = 5280 pés e 1 pé = 0.3048 metros.

28. Liste algumas diferenças entre um *script* e uma função.

29. A velocidade de um fluido em movimento pode ser encontrada a partir da diferença entre as pressões total e estática P_t e P_e. Para a água, isso é dado por

$$V = 1.016\sqrt{P_t - P_e}$$

    Escreva uma função que receberá como argumentos de entrada as pressões total e estática, e retornará a velocidade da água.

30. Escreva uma função $cincos$ que receberá dois argumentos, o número de linhas e de colunas, e retornará uma matriz com esse tamanho e todos os elementos iguais a cinco.

31. Escreva uma função $ehdivpor4$ que receberá um argumento de entrada inteiro e retornará lógico 1 para ***verdadeiro*** se o argumento de entrada for divisível por 4 ou ***falso lógico*** se não for.

32. Escreva uma função $ehint$ que receberá um argumento numérico de entrada $nument$ e retornará 1 para ***verdadeiro*** se esse número for um inteiro ou 0 para ***falso*** se não for. Use o fato de que $nument$ deve ser do tipo **int32(nument)** se for um inteiro. Infelizmente, devido a erros de arredondamento, deve-se notar que é possível obter 1 **lógical** para ***verdadeiro*** se o argumento de entrada estiver próximo de um inteiro. Portanto, a saída pode não ser o que você poderia esperar, como mostrado aqui.

```
>> ehint(4)
ans =
   1
>> ehint(4.9999)
ans =
   0
>> ehint(4.99999999999999999999999999)
ans =
   1
```

33. Um ***Triplo Pitagórico*** é um conjunto de inteiros positivos (a, b, c) tal que $a^2 + b^2 = c^2$. Escreva uma função $ehpitag$ que receberá três inteiros positivos (a, b, e c, nessa ordem) e retornará 1 **logical** para ***verdadeiro***, se formar um ***Triplo Pitagórico*** ou 0 para ***falso***, se não for.

34. Uma função pode retornar um vetor como resultado. Escreva uma função $saivet$ que receberá um argumento inteiro e retornará um vetor que incrementa do valor do argumento de entrada para seu valor mais 5, usando o operador de dois pontos. Por exemplo,

```
>> saivet(4)
ans =
   4   5   6   7   8   9
```

35. Escreva uma função $replicavet$ que receba um vetor e o número de vezes que cada elemento deve ser duplicado. A função deve retornar o vetor resultante. Faça este problema usando somente funções internas. Aqui estão alguns exemplos de chamar a função:

```
>> replicavet(5:-1:1, 2)
ans =
   5   5   4   4   3   3   2   2   1   1
>> replicavet([0 1 0], 3)
ans =
   0   0   0   1   1   1   0   0   0
```

36. Escreva uma função que é chamada de $pegaum$, que receberá um argumento de entrada x, que é um vetor, e retornará um elemento aleatório do vetor. Para exemplo,

```
>> disp(pegaum(-2:0))
  -1
>> help pegaum
pegaum(x) retorna um elemento aleatório do vetor x
```

37. O custo de fabricação de n unidades (onde n é um número inteiro) de um produto específico em uma fábrica é dado pela equação:

$C(n) = 5n^2 - 44n + 11$

Escreva um *script* $custofab$ que irá:

- solicitar ao usuário o número de unidades n
- chamar uma função $custon$ que irá calcular e retornar o custo de fabricação n unidades
- imprima o resultado (o formato deve ser exatamente como mostrado abaixo).
- Em seguida, escreva a função $costn$, que simplesmente recebe o valor de n como argumento de entrada, calcula e retorna o custo de fabricação de n unidades.

Aqui está um exemplo de execução do *script*:

```
>> custofab
Digite o número de unidades: 100
O custo para 100 unidades será de R$ 45611.00
```

38. A conversão depende da temperatura e de outros fatores, mas uma aproximação é que 1 centímetro de chuva equivale a 13 centímetros de neve. Escreva um *script* que solicite ao usuário o número de polegadas de chuva, chama uma função para retorne a quantidade equivalente de neve e imprima esse resultado. Escreva a função também!

39. O volume V de um tetraedro regular é dado por

$$V = \frac{1}{12}\sqrt{2}\,s^3$$

onde s é o comprimento dos lados dos triângulos equiláteros que formam as faces do tetraedro. Escreva um programa para calcular esse volume. O programa consistirá em um *script* e uma função. A função receberá um argumento de entrada, que é o comprimento dos lados, e retornará o volume do tetraedro. O *script* perguntará ao usuário o comprimento dos lados, chamará a função para calcular o volume e imprimirá o resultado em com um bom formato de frase. Por simplicidade, vamos ignorar as unidades.

40. Muitos modelos matemáticos em engenharia usam a função exponencial. A forma geral da função exponencial de decaimento é:

$$y(t) = Ae^{-\tau t}$$

onde, A é o valor inicial em t = 0 e τ é a constante de tempo para a função. Escreva um *script* para estudar o efeito da constante de tempo. Para simplificar a equação, defina A igual a 1. Solicite ao usuário dois valores diferentes para a constante de tempo e para valores iniciais e finais para o intervalo de um vetor t. Em seguida, calcule dois vetores y diferentes usando a equação acima e as duas constantes de tempo e plote os gráficos das duas funções exponenciais, no mesmo gráfico, dentro do intervalo especificado pelo usuário. Use uma função para calcular y. Plote um gráfico em vermelho. Certifique-se de rotular o gráfico e os dois eixos. O que acontece com a taxa de decaimento à medida que a constante de tempo aumenta?

41. Uma senóide exponencial decrescente tem propriedades muito interessantes. Na dinâmica de fluidos, por exemplo, a seguinte equação modela os padrões de onda de um determinado líquido quando o líquido é perturbado por uma força externa:

$$\mathrm{y(x)} = Fe^{\mathrm{ax}} \sin(\mathrm{bx})$$

onde F é a magnitude da força de impulso externa, e a e b são constantes associadas à viscosidade e densidade, respectivamente. Os seguintes dados foram coletados para os seguintes tipos de fluidos:

Fluido	*valor a*	*valor b*
Álcool Etílico	0.246	0.806
Água	0.250	1.000
Óleo	0.643	1.213

Escreva um *script* que solicite ao usuário um valor para F. Em seguida, crie um vetor-x (você decide os valores) e, em seguida, um vetor-y usando a equação acima (escreva uma função para isso). Plote isso, que modela o padrão de onda de um fluido quando perturbado. Faça isso para os três fluidos diferentes; traçe usando os valores acima e compare-os.

42. Um arquivo chamado $custosevendas.dat$ armazena para uma empresa alguns valores de custo e vendas para os últimos n trimestres (n não é definido para um tempo futuro). Os custos estão na primeira coluna e as vendas estão na segunda coluna. Por exemplo, se cinco trimestres fossem representados, haveria cinco linhas no arquivo e poderia ser assim:

```
1100   800
1233   650
1111   1001
1222   1300
 999   1221
```

Escreva um *script* chamado $vendasecustos$ que lerá os dados desse arquivo em uma matriz. Quando o *script* é executado, ele fará três coisas. Primeiro, será impresso quantos trimestres foram representados no arquivo, como:

```
>> vendasecustos
Havia 5 trimestres no arquivo
```

A seguir, os custos serão plotados usando círculos pretos e vendas usando asteriscos pretos ($*$) em uma Janela de Figura com uma legenda (usando eixos padrão), como mostra a Figura 3.9. Finalmente, o *script* gravará os dados em um novo arquivo chamado $arquivonovo.dat$ em uma ordem diferente. As vendas serão a primeira linha e os custos serão a segunda linha. Por exemplo, se o arquivo for como mostrado acima, o arquivo resultante armazenará o seguinte:

```
800     650     1001    1300    1221
1100    1233    1111    1222    999
```

Não deve ser assumido que o número de linhas no arquivo é conhecido.

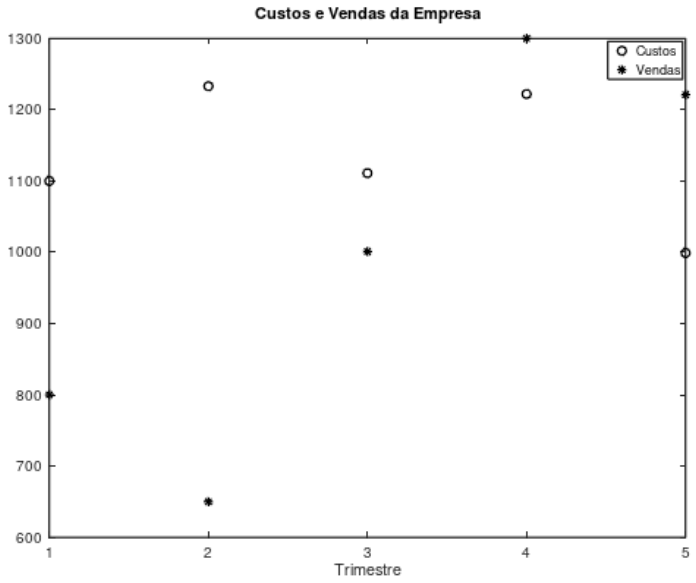

FIGURA 3.9 Plotagem de dados de custo e vendas

Referência

ATAWAY, S. MATLAB A Pratical Introduction to Programming and Program Solving. Butterworth-Heinemann/Elsevier, Waltham, MA, USA, Third Edition, 2013.