O meu co-autor e amigo Fausto Saleri faleceu
inesperadamente no dia 4 de Junho de 2007, com 41 anos.
Fausto foi um grande matemático e uma pessoa maravilhosa.
Se gostar deste livro, é ele que merece
o mais elevado reconhecimento (A.Q.)

Alfio Quarteroni
Fausto Saleri

CÁLCULO CIENTÍFICO

com MATLAB e Octave

Springer

ALFIO QUARTERONI
EPFL, Lausanne e
MOX, Politecnico di Milano

FAUSTO SALERI
MOX, Politecnico di Milano

As simulações numéricas reproduzidas na capa foram realizadas por
Davide Detomi e Nicola Parolini

Traduzido do italiano por:
Adélia Sequeira
Departamento de Matemática
Instituto Superior Técnico - Universidade Técnica de Lisboa

Tradução da obra italiana:
Introduzione al Calcolo Scientifico - Esercizi e problemi risolti con MATLAB
A. Quarteroni, F. Saleri

ISBN 978-88-740-0717-8 Springer Milan Berlin Heidelberg New York

Springer-Verlag Itália é membro da Springer Science+Business Media

springer.com

Versão final em camera-ready fornecida pelo tradutor
Projecto gráfico da capa: Simona Colombo (Milano)
Impresso em Itália: Signum Srl, Bollate (MI)

Springer-Verlag Italia Srl, Via Decembrio 28, 20137 Milano

Prefácio

Este livro é uma introdução ao Cálculo Científico. O seu objectivo consiste em apresentar vários métodos numéricos para resolver no computador certos problemas matemáticos que não podem ser tratados de forma mais simples. São abordadas questões clássicas como o cálculo de zeros ou de integrais de funções contínuas, a resolução de sistemas lineares, a aproximação de funções por polinómios e a resolução aproximada de equações diferenciais. No Capítulo 1, como preâmbulo, expõem-se as regras adoptadas pelos computadores quando armazenam e operam com números reais e complexos, vectores e matrizes.

A fim de tornar esta apresentação mais concreta e atraente, adoptam-se os ambientes de programação MATLAB ® [1] e Octave. Deve-se ter em conta que Octave é uma re-implementação de uma parte de MATLAB que inclui em particular muitos dos seus recursos numéricos e é distribuído gratuitamente por GNU *General Public License*. Neste livro, são introduzidos de forma gradual os principais comandos e instruções destas linguagens de programação, o que permitirá implementar todos os algoritmos considerados e verificar na prática propriedades teóricas como a estabilidade, a precisão e a complexidade. Inclui-se ainda a resolução de diversos problemas através de exercícios e exemplos, frequentemente ligados a aplicações concretas.

Em todo o livro utiliza-se muitas vezes a expressão "comando de MATLAB": neste contexto, MATLAB deve ser entendido como uma *linguagem* que é comum aos programas de MATLAB e Octave. Fez-se um esforço particular visando assegurar a compatibilidade dos códigos nas duas linguagens de programação. Nos poucos casos em que tal não foi possível, apresenta-se uma nota explicativa no final da secção correspondente.

[1] MATLAB é uma marca registada de TheMathWorks Inc., 24 Prime Park Way, Natick, MA 01760, Tel: 001+508-647-7000, Fax: 001+508-647-7001

Adopta-se diversos símbolos gráficos para tornar a leitura mais agradável. O comando de MATLAB (ou Octave) é colocado na margem, ao lado da linha onde se introduz pela primeira vez. O símbolo  serve para indicar a presença de exercícios, enquanto que o símbolo  é utilizado a fim de chamar a atenção do leitor para um comportamento crítico ou surpreendente de um algoritmo ou procedimento. As fórmulas matemáticas de especial relevância são postas numa caixa. Finalmente, o símbolo  indica um quadro resumindo os conceitos e conclusões que se acabam de expor.

No fim de cada capítulo encontra-se uma secção específica que apresenta assuntos não abordados e as referências bibliográficas que permitem ao leitor aprofundar os conhecimentos adquiridos.

Frequentemente faz-se referência ao livro [QSS07] que desenvolve a um nível mais avançado numerosas questões aqui abordadas e onde são demonstrados resultados teóricos. Para uma descrição mais completa de MATLAB referimos [HH05]. Todos os programas incluídos neste livro podem ser extraídos do seguinte endereço na web:

`mox.polimi.it/qs`

O leitor não necessita de quaisquer pré-requisitos, com excepção de um curso elementar de Cálculo. Contudo, no primeiro capítulo recordam-se os principais resultados de Cálculo e Geometria que são usados extensivamente no texto. Os assuntos menos elementares, dispensáveis numa primeira leitura, assinalam-se com o símbolo  .

Queremos deixar expresso o nosso agradecimento a Francesca Bonadei da Springer-Itália pela sua colaboração amigável e eficiente ao longo deste projecto, a Paola Causin por nos ter proposto vários problemas, a Christophe Prud'homme, John W. Eaton e David Bateman por nos terem ajudado na utilização de Octave, e ao apoio financeiro do projecto Poseidon da École Polytechnique Fédérale de Lausanne. Por fim, queremos exprimir o nosso reconhecimento a Adélia Sequeira pela tradução cuidadosa e crítica deste livro, assim como pelas suas numerosas e pertinentes sugestões.

Lausanne e Milano, Junho de 2007 Alfio Quarteroni, Fausto Saleri

Índice

Programas

1

O que não se pode ignorar

Neste livro, usaremos sistematicamente noções de matemática elementar que o leitor já deverá saber, mas que poderá não recordar de imediato.

Este capítulo tem como objectivo algumas revisões e a introdução de novos conceitos no domínio da Análise Numérica. Começaremos por explorar o seu significado e utilidade com a ajuda de MATLAB (MATrix LABoratory), um ambiente integrado de programação e visualização em cálculo científico. Usaremos também GNU Octave (abreviadamente, Octave) que é na sua quase totalidade compatível com o sistema MATLAB. Nas Secções 1.6 e 1.7 faremos uma introdução ao MATLAB e ao Octave, apenas suficiente para o que se pretende neste livro. Faremos igualmente referência às principais diferenças entre os dois programas e indicaremos aos leitores interessados os manuais [HH05] e [Eat02], para uma descrição completa das linguagens MATLAB e Octave, respectivamente.

Octave é uma re-implementação da parte de MATLAB que inclui a grande maioria dos seus recursos numéricos e que é distribuído gratuitamente sob a Licença Pública Geral GNU.

Ao longo do livro, usaremos frequentemente a expressão "comando de MATLAB": neste contexto, MATLAB deverá ser entendido como a *linguagem* comum a ambos os programas MATLAB e Octave.

Procurámos assegurar a utilização dos nossos códigos e programas pelo MATLAB e Octave. Nos poucos casos em que isso não foi possível, introduzimos um breve comentário no fim da secção correspondente.

Neste capítulo, apresentamos em resumo noções típicas dos cursos de Cálculo, Álgebra Linear e Geometria, reformulando-as com vista à sua aplicação no cálculo científico.

1.1 Números reais

Enquanto que o conjunto $\mathbb{R}$ dos números reais é bem conhecido, o modo como os computadores os manipulam é talvez menos conhecido. Dado

que as máquinas têm recursos limitados, só se pode representar um subconjunto $\mathbb{F}$ de dimensão finita de $\mathbb{R}$. Os números deste subconjunto designam-se por *números de vírgula flutuante*. Por outro lado, como veremos na Secção 1.1.2, as propriedades que caracterizam $\mathbb{F}$ são diferentes das de $\mathbb{R}$. Um número real x é geralmente truncado pela máquina, dando origem a um novo número (*número de vírgula flutuante*), que se designa por $fl(x)$, e que não coincide necessariamente com o número original x.

1.1.1 Como se representam

Para conhecer as diferenças entre $\mathbb{R}$ e $\mathbb{F}$, iremos fazer algumas experiências com o MATLAB que ilustram a forma como o computador (por exemplo, um PC) trata os números reais. Note-se que o uso de MATLAB ou Octave, em vez de outra linguagem, é apenas uma questão de conveniência. Os resultados dos nossos cálculos dependem principalmente do funcionamento interno do computador e, em menor grau, da linguagem de programação. Consideremos o número racional $x = 1/7$, cuja representação decimal é $0.\overline{142857}$. Trata-se de uma representação infinita, uma vez que tem uma infinidade de algarismos decimais. Para obter a sua representação no computador introduzimos o quociente `1/7` depois do *prompt* (representado pelo símbolo `»`) e obtemos

```
» 1/7
  ans =
       0.1429
```

que é um número apenas com quatro algarismos decimais, em que o último é diferente do quarto dígito do número original.

Se considerarmos agora 1/3 obtemos 0.3333 e, desta vez, o quarto algarismo decimal é exacto. Este comportamento deve-se ao facto de os números reais serem *arredondados* no computador. Isto significa, em primeiro lugar, que só se obtém um número fixo de casas decimais e que, além disso, a última decimal sofre um incremento de uma unidade sempre que a primeira decimal desprezada for superior ou igual a 5.

A primeira observação consiste em colocar a questão de se poder ou não representar os números reais apenas com quatro algarismos decimais. Com efeito, a representação interna do número utiliza 16 decimais, e o que vimos foi apenas um dos vários formatos de saída de MATLAB. O mesmo número pode representar-se por diferentes expressões dependendo da escolha do formato. Por exemplo, para o número 1/7, alguns *formatos* possíveis de saída são:

`format`

`format long`	dá	0.14285714285714,
`format short e`	”	$1.4286e-01$,
`format long e`	”	$1.428571428571428e-01$,
`format short g`	”	0.14286,
`format long g`	”	0.142857142857143.

Certos formatos são mais coerentes do que outros, com a representação interna dos números no computador. Na verdade, em geral, um computador guarda um número real da seguinte maneira

$$x = (-1)^s \cdot (0.a_1a_2 \ldots a_t) \cdot \beta^e = (-1)^s \cdot m \cdot \beta^{e-t}, \quad a_1 \neq 0 \tag{1.1}$$

em que s é 0 ou 1, β (um inteiro positivo maior ou igual a 2) é a *base* adoptada pelo computador específico em que estamos a trabalhar, m é um inteiro chamado *mantissa*, cujo comprimento t é o número máximo de algarismos armazenados a_i (com $0 \leq a_i \leq \beta - 1$), e e um número inteiro chamado *expoente*. O formato `long e` é o que mais se aproxima desta representação, e `e` designa o expoente; os seus dígitos, precedidos do sinal, declaram-se à direita do caracter `e`. Os números da forma (1.1) designam-se por números de vírgula flutuante, porque a posição da vírgula não é fixa. Os dígitos $a_1a_2 \ldots a_p$ (com $p \leq t$) são frequentemente chamados os p primeiros algarismos significativos de x.

A condição $a_1 \neq 0$ garante que um número não pode ter múltiplas representações. Por exemplo, sem esta restrição o número 1/10 poderia não só ser representado (no sistema decimal) por $0.1 \cdot 10^0$, mas também por $0.01 \cdot 10^1$, etc.

O conjunto $\mathbb{F}$ é assim completamente caracterizado pela base β, o número de algarismos significativos t e o intervalo (L, U) (com $L < 0$ e $U > 0$) de variação do índice e, e designa-se por $\mathbb{F}(\beta, t, L, U)$. Por exemplo, em MATLAB temos $\mathbb{F} = \mathbb{F}(2, 53, -1021, 1024)$ (com efeito, 53 algarismos significativos na base 2 correspondem aos 15 algarismos significativos de MATLAB na base 10 com o `format long`).

Felizmente, o *erro de arredondamento*que se gera inevitavelmente sempre que um número real $x \neq 0$ é substituído pelo seu representante $fl(x)$ em $\mathbb{F}$, é pequeno, uma vez que

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2}\epsilon_M \tag{1.2}$$

onde $\epsilon_M = \beta^{1-t}$ é a distância entre 1 e o número mais próximo em vírgula flutuante maior que 1. Note-se que ϵ_M depende de β e t. Por exemplo, em MATLAB ϵ_M pode obter-se com o comando `eps`, e tem-se `eps`
$\epsilon_M = 2^{-52} \simeq 2.22 \cdot 10^{-16}$. Assinale-se que em (1.2) é estimado o *erro relativo* sobre x, o que tem seguramente mais significado do que o *erro absoluto* $|x - fl(x)|$. Na verdade, este último, contrariamente ao erro relativo, não tem em conta a ordem de grandeza de x.

O número 0 não pertence a $\mathbb{F}$, uma vez que nesse caso teríamos $a_1 = 0$ em (1.1): por esta razão é tratado separadamente. Por outro lado, L e U sendo finitos, não é possível representar números cujo valor absoluto seja

arbitrariamente grande ou arbitrariamente pequeno. Mais precisamente, o menor e o maior número real positivo de $\mathbb{F}$ são dados respectivamente por

$$x_{min} = \beta^{L-1},\ x_{max} = \beta^{U}(1-\beta^{-t}).$$

Em MATLAB estes valores podem-se obter através dos comandos `realmin` e `realmax`, que conduzem a

`realmin`
`realmax`

$$x_{min} = 2.225073858507201 \cdot 10^{-308},$$
$$x_{max} = 1.7976931348623158 \cdot 10^{+308}.$$

Um número positivo menor que x_{min} produz uma mensagem de erro chamada *underflow* e é tratado de um modo especial ou como se fosse nulo (ver, por exemplo, [QSS07], Capítulo 2). Um número positivo maior que x_{max} produz uma mensagem de erro chamada *overflow* e armazena-se na variável `Inf` (que é a representação de $+\infty$ no computador).

`Inf`

Os elementos de $\mathbb{F}$ são mais densos próximo de x_{min}, e menos densos quando se aproximam de x_{max}. Com efeito, o número de $\mathbb{F}$ mais próximo de x_{max} (à sua esquerda) e o mais próximo de x_{min} (à sua direita) são, respectivamente

$$x^{-}_{max} = 1.7976931348623157 \cdot 10^{+308},$$
$$x^{+}_{min} = 2.225073858507202 \cdot 10^{-308}.$$

Tem-se então $x^{+}_{min} - x_{min} \simeq 10^{-323}$, enquanto que $x_{max} - x^{-}_{max} \simeq 10^{292}$ (!). Apesar disso, a distância relativa é pequena em ambos os casos, como se deduz de (1.2).

1.1.2 Como calcular com os números de vírgula flutuante

Dado que $\mathbb{F}$ é um subconjunto próprio de $\mathbb{R}$, as operações algébricas elementares com números de vírgula flutuante não gozam de todas as propriedades das operações análogas em $\mathbb{R}$. Concretamente, a comutatividade verifica-se para a adição (isto é $fl(x+y) = fl(y+x)$) e para a multiplicação ($fl(xy) = fl(yx)$), mas outras propriedades tais como a associativa e a distributiva não se verificam. Além disso, o 0 já não é único. Com efeito, associemos à variável `a` o valor 1, e executemos as seguintes instruções:

```
» a = 1; b=1; while a+b ~= a; b=b/2; end
```

A variável `b` divide-se por dois em cada passo desde que a soma de `a` e `b` seja diferente (`~=`) de `a`. Se fizéssemos as mesmas operações com números reais, este programa nunca terminaria, enquanto que no nosso caso, termina após um número finito de iterações, obtendo-se o seguinte valor para `b`: `1.1102e-16`$= \epsilon_M/2$. Portanto, existe pelo menos um número `b`

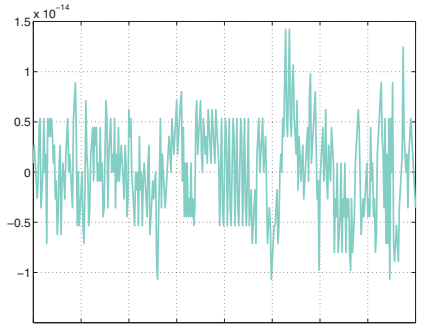

Figura 1.1. Comportamento oscilatório da função (1.3) devido a erros de cancelamento

diferente de 0 tal que `a+b=a`. Isto é possível porque $\mathbb{F}$ é constituído por números isolados; quando se somam dois números `a` e `b` com `b<a` e `b` inferior a ϵ_M, resulta sempre `a+b` igual a `a`. Em MATLAB, `a+eps(a)` é o menor número de $\mathbb{F}$ maior do que `a`. Assim a soma `a+b` dá origem a `a`, para todo `b < eps(a)`.

A associatividade perde-se sempre que ocorre uma situação de *overflow* ou de *underflow*. Consideremos, por exemplo, `a=1.0e+308`, `b=1.1e+308` e `c=-1.001e+308`, e efectuemos a soma por dois processos diferentes. Obtém-se

$$\texttt{a} + (\texttt{b} + \texttt{c}) = \texttt{1.0990e+308}, \quad (\texttt{a} + \texttt{b}) + \texttt{c} = \texttt{Inf}.$$

Trata-se de um exemplo particular do que ocorre quando se somam dois números com sinais opostos mas muito próximos em valor absoluto. Neste caso o resultado pode ser bastante impreciso e a situação designa-se por *perda*, ou *cancelamento, de algarismos significativos*. Por exemplo, calculemos $((1+x)-1)/x$ (o resultado óbvio é 1 para todo $x \neq 0$):

```
» x =  1.e-15; ((1+x)-1)/x
  ans = 1.1102
```

Este resultado é bastante impreciso, com um erro relativo superior a 11%!

Outro caso de cancelamento numérico acontece quando se calcula o valor da função

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \qquad (1.3)$$

em 401 pontos equidistantes de abcissas em $[1 - 2 \cdot 10^{-8}, 1 + 2 \cdot 10^{-8}]$. Obtemos o gráfico caótico representado na Figura 1.1 (o comportamento real é o de $(x-1)^7$, que é essencialmente constante e próximo da função nula numa pequena vizinhança de $x = 1$). Na Secção 1.4 introduziremos os comandos de MATLAB que permitiram construir este gráfico.

Finalmente, interessa observar que em $\mathbb{F}$ não existem formas indeterminadas tais como $0/0$ ou ∞/∞. Elas produzem o que se chama um *not a number* (`NaN` em MATLAB ou em Octave), ao qual não se aplicam as regras usuais de cálculo.

`NaN`

Observação 1.1 É verdade que os erros de arredondamento são geralmente pequenos mas, quando repetidos em algoritmos longos e complexos, podem ter efeitos catastróficos. Podemos citar dois exemplos notáveis: a explosão do míssil Arianne a 4 de Junho de 1996, gerada por um erro de *overflow* no computador de bordo; o fracasso da missão do míssil americano Patriot durante a Guerra do Golfo em 1991, devido a um erro de arredondamento no cálculo da sua trajectória.

Um exemplo com consequências menos catastróficas (mas, mesmo assim, desagradáveis) é dado pela sucessão

$$z_2 = 2,\ z_{n+1} = 2^{n-1/2}\sqrt{1-\sqrt{1-4^{1-n}z_n^2}},\ n = 2, 3, \ldots \tag{1.4}$$

que converge para π quando n tende para infinito. Quando se usa o MATLAB para calcular z_n, o erro relativo entre π e z_n diminui nas primeiras 16 iterações e em seguida aumenta, devido a erros de arredondamento (como mostra a Figura 1.2). •

Ver os Exercícios 1.1-1.2.

1.2 Números complexos

Os números complexos, cujo conjunto se designa por $\mathbb{C}$, são da forma $z = x + iy$, onde $i = \sqrt{-1}$ é a unidade imaginária (isto é $i^2 = -1$), e $x = \mathrm{Re}(z)$ e $y = \mathrm{Im}(z)$ são as partes real e imaginária de z, respectivamente. Em geral representam-se no computador como pares de números reais.

A menos que sejam redefinidas, as variáveis de MATLAB `i` e `j` designam a unidade imaginária. Para definir um número complexo com parte

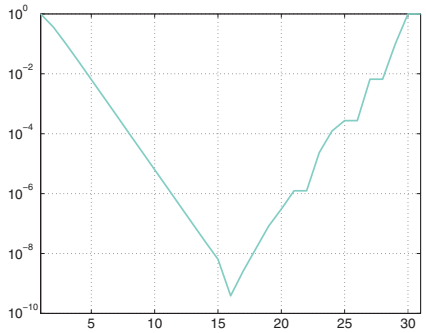

Figura 1.2. Logaritmo do erro relativo $|\pi - z_n|/\pi$ em função de n

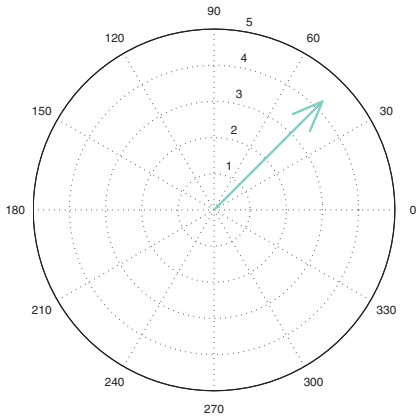

Figura 1.3. Resultado do comando `compass` de MATLAB

real `x` e parte imaginária `y`, pode-se escrever simplesmente `x+i*y`; em alternativa, pode usar-se o comando `complex(x,y)`. As representações exponencial (ou polar) e trigonométrica de um número complexo z, são equivalentes através da *fórmula de Euler*

`complex`

$$z = \rho e^{i\theta} = \rho(\cos\theta + i\sin\theta); \tag{1.5}$$

em que $\rho = \sqrt{x^2+y^2}$ é o módulo do número complexo (obtido com o comando `abs(z)`) e θ é o seu argumento, ou seja o ângulo que o vector de componentes (x, y) no plano complexo, faz com o eixo dos x. O argumento θ é obtido com o comando `angle(z)`. A representação (1.5) é dada por:

`abs`

`angle`

$$\texttt{abs(z)} * (\texttt{cos(angle(z))} + \texttt{i} * \texttt{sin(angle(z))}).$$

A representação polar gráfica de um ou mais números complexos pode-se obter com o comando `compass(z)`, em que `z` é um só número complexo ou um vector cujas componentes são números complexos. Por exemplo, ao digitar

`compass`

```
» z = 3+i*3; compass(z);
```

obtém-se o gráfico da Figura 1.3.

Para um dado número complexo `z`, pode-se extrair a sua parte real usando o comando `real(z)` e a sua parte imaginária com `imag(z)`. Finalmente, o complexo conjugado $\bar{z} = x - iy$ de$z$, é obtido escrevendo simplesmente `conj(z)`.

`real`

`imag`

`conj`

Em MATLAB efectuam-se todas as operações admitindo implicitamente que tanto os operandos como o resultado são complexos. Isto pode dar origem a alguns resultados aparentemente surpreendentes. Por

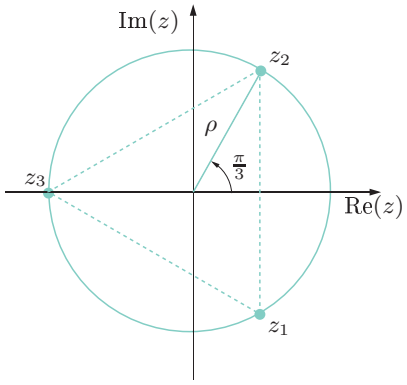

Figura 1.4. Representação no plano complexo das três raízes cúbicas do número real -5

exemplo, se calcularmos a raiz cúbica de -5 com o comando `(-5)^(1/3)` de MATLAB, em vez de $-1.7099\ldots$ obtemos o número complexo $0.8550+1.4809i$ (introduz-se aqui o símbolo `^` para o expoente da potência). Com efeito, os números da forma $\rho e^{i(\theta+2k\pi)}$, com k inteiro, não se podem distinguir de $z=\rho e^{i\theta}$. Calculando $\sqrt[3]{z}$ obtemos $\sqrt[3]{\rho}e^{i(\theta/3+2k\pi/3)}$, isto é, as três raízes distintas

$$z_1=\sqrt[3]{\rho}e^{i\theta/3},\ z_2=\sqrt[3]{\rho}e^{i(\theta/3+2\pi/3)},\ z_3=\sqrt[3]{\rho}e^{i(\theta/3+4\pi/3)}.$$

MATLAB seleccionará a primeira que se encontra ao percorrer o plano complexo no sentido contrário ao dos ponteiros do relógio, partindo do eixo dos reais. Dado que a representação polar de $z=-5$ é $\rho e^{i\theta}$ com $\rho=5$ e $\theta=-\pi$, as três raízes são (ver Figura 1.4 para a sua representação no plano de Gauss)

$$\begin{aligned}
z_1&=\sqrt[3]{5}(\cos(-\pi/3)+i\sin(-\pi/3))\simeq 0.8550-1.4809i,\\
z_2&=\sqrt[3]{5}(\cos(\pi/3)+i\sin(\pi/3))\simeq 0.8550+1.4809i,\\
z_3&=\sqrt[3]{5}(\cos(-\pi)+i\sin(-\pi))\simeq -1.7100.
\end{aligned}$$

A segunda raiz é a seleccionada pelo MATLAB.

Finalmente, com (1.5) obtemos

$$\cos(\theta)=\frac{1}{2}\left(e^{i\theta}+e^{-i\theta}\right),\ \sin(\theta)=\frac{1}{2i}\left(e^{i\theta}-e^{-i\theta}\right). \tag{1.6}$$

Octave 1.1 O comando `compass` não está disponível em Octave, mas pode ser obtido através da seguinte função:

```
function compass(z)
xx = [0 1 .8 1 .8].';
yy = [0 0 .08 0 -.08].';
arrow = xx + yy.*sqrt(-1);
z = arrow * z;
[th,r] = cart2pol(real(z),imag(z));
polar(th,r);
return
```

■

1.3 Matrizes

Sejam n e m inteiros positivos. Uma matriz com m linhas e n colunas é um conjunto de $m \times n$ elementos a_{ij}, com $i = 1, \ldots, m$, $j = 1, \ldots, n$, representado pela seguinte tabela:

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}. \tag{1.7}$$

De forma compacta escreve-se $\mathrm{A} = (a_{ij})$. Se os elementos de A forem números reais, escreve-se $\mathrm{A} \in \mathbb{R}^{m \times n}$, e $\mathrm{A} \in \mathbb{C}^{m \times n}$ se forem complexos.

As matrizes quadradas de dimensão n são aquelas em que $m = n$. Uma matriz com uma só coluna é um *vector coluna*, enquanto que uma matriz com uma só linha é um *vector linha*.

Para definir uma matriz em MATLAB tem de se escrever os seus elementos da primeira à última linha, utilizando o caracter `;` para separar as diferentes linhas. Por exemplo, o comando

```
» A = [ 1 2 3; 4 5 6]
```

conduz a

```
A =
     1     2     3
     4     5     6
```

isto é, uma matriz 2×3 cujos elementos são os indicados acima. A matriz nula $m \times n$ é aquela em que todos os elementos (a_{ij}) são nulos, para $i = 1, \ldots, m, j = 1, \ldots, n$; pode-se construir em MATLAB com o comando `zeros(m,n)`. O comando `eye(m,n)` dá origem a uma matriz com todos os elementos nulos excepto os da diagonal principal a_{ii}, $i = 1, \ldots, \min(m, n)$, que são iguais a 1. A matriz identidade $n \times n$ obtém-se com o comando `eye(n)`: os seus elementos são $\delta_{ij} = 1$ se $i = j$, e 0 no caso contrário, para $i, j = 1, \ldots, n$. Finalmente, com o comando `A=[ ]` define-se uma matriz vazia.

`zeros`
`eye`
`[ ]`

Recordamos as seguintes operações matriciais:

1. se $\mathrm{A} = (a_{ij})$ e $\mathrm{B} = (b_{ij})$ são matrizes $m \times n$, a *soma* de A e B é a matriz $\mathrm{A} + \mathrm{B} = (a_{ij} + b_{ij})$;
2. o *produto* de uma matriz A por um número real ou complexo λ é a matriz $\lambda\mathrm{A} = (\lambda a_{ij})$;
3. o *produto* de duas matrizes só é possível se as suas dimensões forem compatíveis, isto é, se o número de colunas da primeira for igual ao número de linhas da segunda; mais precisamente, se A é $m \times p$ e B é $p \times n$, para algum inteiro positivo p, então $\mathrm{C} = \mathrm{AB}$ é uma matriz $m \times n$ de elementos

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \text{ para } i = 1, \ldots, m,\ j = 1, \ldots, n.$$

Segue-se um exemplo de soma e de produto de duas matrizes:

```
» A=[1 2 3; 4 5 6];
» B=[7 8 9; 10 11 12];
» C=[13 14; 15 16; 17 18];
» A+B

  ans =
       8     10     12
      14     16     18

» A*C

  ans =
      94    100
     229    244
```

Note-se que o MATLAB dá uma mensagem de erro quando se tenta executar operações com matrizes de dimensões incompatíveis. Por exemplo:

```
» A=[1 2 3; 4 5 6];
» B=[7 8 9; 10 11 12];
» C=[13 14; 15 16; 17 18];
» A+C

  ??? Error using ==> +
  Matrix dimensions must agree.

» A*B

  ??? Error using ==> *
  Inner matrix dimensions must agree.
```

Se A for uma matriz quadrada de dimensão n, a sua *inversa* (se existir) é uma matriz quadrada de dimensão n, que se designa por A^{-1}, e que satisfaz a relação matricial $\mathrm{AA}^{-1} = \mathrm{A}^{-1}\mathrm{A} = \mathrm{I}$. A matriz A^{-1} pode-se
inv obter com o comando `inv(A)`. A inversa de A existe se e só se o *determinante* de A, um número que se designa por det(A), for não nulo. Esta última condição verifica-se sse os vectores colunas de A forem linearmente independentes (ver Secção 1.3.1). O determinante de uma matriz

quadrada define-se pela seguinte fórmula recursiva (*regra de Laplace*):

$$\det(\mathrm{A}) = \begin{cases} a_{11} & \text{if } n = 1, \\ \displaystyle\sum_{j=1}^{n} \Delta_{ij} a_{ij}, \text{ for } n > 1, \ \forall i = 1, \ldots, n, \end{cases} \tag{1.8}$$

em que $\Delta_{ij} = (-1)^{i+j} \det(\mathrm{A}_{ij})$ e A_{ij} é a matriz que se obtém por eliminação da i-ésima linha e j-ésima coluna da matriz A (o resultado é independente do índice de linha ou de coluna).

Em particular, se $\mathrm{A} \in \mathbb{R}^{2\times 2}$ tem-se

$$\det(\mathrm{A}) = a_{11}a_{22} - a_{12}a_{21},$$

e se $\mathrm{A} \in \mathbb{R}^{3\times 3}$

$$\det(\mathrm{A}) = a_{11}a_{22}a_{33} + a_{31}a_{12}a_{23} + a_{21}a_{13}a_{32}$$
$$-a_{11}a_{23}a_{32} - a_{21}a_{12}a_{33} - a_{31}a_{13}a_{22}.$$

Para um produto de matrizes verifica-se a seguinte propriedade: se $\mathrm{A} = \mathrm{BC}$, então $\det(\mathrm{A}) = \det(\mathrm{B})\det(\mathrm{C})$.

Para inverter uma matriz 2×2 e calcular o seu determinante procede-se do seguinte modo:

```
» A=[1 2; 3 4];
» inv(A)

  ans =
     -2.0000     1.0000
      1.5000    -0.5000

» det(A)

  ans =
     -2
```

Se uma matriz for singular, o sistema MATLAB dará uma mensagem de erro, seguida de uma matriz cujos elementos são todos iguais a `Inf`, como se ilustra no exemplo seguinte:

```
» A=[1 2; 0 0];
» inv(A)

  Warning: Matrix is singular to working precision.
  ans =
     Inf    Inf
     Inf    Inf
```

Para certos tipos de matrizes quadradas, o cálculo das inversas e dos determinantes é bastante simples. Se A for uma *matriz diagonal*, isto é, uma matriz em que apenas os elementos da diagonal a_{kk}, $k = 1, \ldots, n$, são não nulos, o seu determinante é dado por $\det(\mathrm{A}) = a_{11}a_{22}\cdots a_{nn}$.

Em particular, A é não singular sse $a_{kk} \neq 0$ para todo o k. Nesse caso, a inversa de A é ainda uma matriz diagonal com elementos a_{kk}^{-1}.

diag Seja `v` um vector de dimensão `n`. O comando `diag(v)` de MATLAB produz uma matriz diagonal cujos elementos são as componentes do vector `v`. O comando mais geral `diag(v,m)` dá uma matriz quadrada `n+abs(m)` cuja m-ésima diagonal superior (isto é, a diagonal constituída pelos elementos de índices $i, i+m$) contém as componentes de `v`, e com os restantes elementos nulos. Note-se que esta extensão também é válida para valores negativos de `m`: neste caso só são afectados os elementos das diagonais inferiores.
Por exemplo, se `v = [1 2 3]` então:

```
» A=diag(v,-1)

  A =
       0       0       0       0
       1       0       0       0
       0       2       0       0
       0       0       3       0
```

Outras matrizes particulares importantes são as *triangulares superiores* e as *triangulares inferiores*. Uma matriz quadrada de dimensão n é *triangular inferior* (respectivamente, *superior*) se todos os elementos situados acima (respectivamente, abaixo) da diagonal principal forem nulos. O seu determinante será então simplesmente o produto dos termos diagonais.

tril triu Os comandos `tril(A)` e `triu(A)`, permitem extrair as partes triangulares inferior e superior de uma matriz `A` de dimensão `n`. As suas extensões `tril(A,m)` ou `triu(A,m)`, com `m` compreendido entre `-n` e `n`, permitem extrair as partes triangulares aumentadas ou diminuídas das m diagonais secundárias.
Por exemplo, dada a matriz `A =[3 1 2; -1 3 4; -2 -1 3]`, com o comando `L1=tril(A)` obtém-se

```
L1 =
     3     0     0
    -1     3     0
    -2    -1     3
```

enquanto que com `L2=tril(A,1)`, se obtém

```
L2 =
     3     1     0
    -1     3     4
    -2    -1     3
```

Recordemos que se $A \in \mathbb{R}^{m\times n}$ a sua transposta $A^T \in \mathbb{R}^{n\times m}$ é a matriz que se obtém trocando as linhas com as colunas de A. Quando A' $n = m$ e $A = A^T$ a matriz A diz-se *simétrica*. Finalmente, `A'` é o comando de MATLAB que designa a transposta de `A` se `A` for real, ou a sua transposta conjugada (isto é, A^H) se `A` for complexa. Uma matriz

quadrada complexa que coincide com a sua transposta conjugada A^H diz-se *hermitiana*.

A mesma notação, `v'`, é utilizada para o transposto conjugado $\mathbf{v}^H$ do vector `v`. Designando por v_i as componentes de $\mathbf{v}$, o vector adjunto $\mathbf{v}^H$ é um vector linha cujas componentes são os complexos conjugados $\bar{v}_i$ de v_i. `v'`

Octave 1.2 Octave dá também uma mensagem de erro quando se tenta efectuar operações com matrizes de dimensões incompatíveis. Retomando os exemplos anteriores de MATLAB obtém-se:

```
octave:1> A=[1 2 3; 4 5 6];
octave:2> B=[7 8 9; 10 11 12];
octave:3> C=[13 14; 15 16; 17 18];
octave:4> A+C

  error: operator +: nonconformant arguments (op1 is
  2x3, op2 is 3x2)
  error: evaluating binary operator ‘+’ near line 2,
  column 2

octave:5> A*B

  error: operator *: nonconformant arguments (op1 is
  2x3, op2 is 2x3)
  error: evaluating binary operator ‘*’ near line 2,
  column 2
```

Se A for singular e se pretendermos invertê-la, Octave dá uma mensagem de erro seguida da matriz a inverter, como se ilustra no exemplo seguinte:

```
octave:1> A=[1 2; 0 0];
octave:2> inv(A)

  warning: inverse: matrix singular to machine
  precision, rcond = 0
  ans =
    1  2
    0  0
```

■

1.3.1 Vectores

Neste livro, os vectores serão indicados a negrito; mais precisamente, $\mathbf{v}$ designará um vector coluna cuja i-ésima componente é v_i. Quando todas as componentes forem reais escrever-se-á $\mathbf{v} \in \mathbb{R}^n$.

Em MATLAB, os vectores são considerados como matrizes particulares. Para definir um vector coluna é preciso indicar as suas componentes entre parêntesis rectos separadas por um ponto e vírgula, enquanto que para um vector linha bastará escrever as suas componentes, separadas por espaços ou por vírgulas. Por exemplo, as instruções `v = [1;2;3]` e `w = [1 2 3]` definem o vector coluna $\mathbf{v}$ e o vector linha $\mathbf{w}$, ambos de dimensão 3. O comando `zeros(n,1)` (respectivamente, `zeros(1,n)`) define um `zeros`

vector coluna (respectivamente, linha) de dimensão `n` com elementos nulos, que se designará por $\mathbf{0}$. Analogamente, o comando `ones(n,1)` gera o vector coluna, que se designa por $\mathbf{1}$, cujas componentes são todas iguais a 1.

ones

Um sistema de vectores $\{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$ é *linearmente independente* se a relação

$$\alpha_1 \mathbf{y}_1 + \ldots + \alpha_m \mathbf{y}_m = \mathbf{0}$$

implicar que todos os coeficientes $\alpha_1, \ldots, \alpha_m$ sejam nulos. Um sistema $\mathcal{B} = \{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ de n vectores linearmente independentes de $\mathbb{R}^n$ (ou $\mathbb{C}^n$) diz-se uma *base* de $\mathbb{R}^n$ (ou $\mathbb{C}^n$), isto é, qualquer vector $\mathbf{w}$ de $\mathbb{R}^n$ pode-se escrever como combinação linear dos elementos de $\mathcal{B}$,

$$\mathbf{w} = \sum_{k=1}^{n} w_k \mathbf{y}_k,$$

e os coeficientes $\{w_k\}$ são únicos. Estes últimos dizem-se as *componentes* de $\mathbf{w}$ na base $\mathcal{B}$. Por exemplo, a base canónica de $\mathbb{R}^n$ é o conjunto de vectores $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$, em que $\mathbf{e}_i$ tem a sua i-ésima componente igual a 1, e todas as outras nulas. Esta é a base de $\mathbb{R}^n$ que normalmente se usa.

O *produto escalar* de dois vectores $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ define-se por

$$(\mathbf{v}, \mathbf{w}) = \mathbf{w}^T \mathbf{v} = \sum_{k=1}^{n} v_k w_k,$$

onde $\{v_k\}$ e $\{w_k\}$ são as componentes de $\mathbf{v}$ and $\mathbf{w}$, respectivamente. O comando de MATLAB correspondente é `w'*v`, onde o apóstrofe designa a transposição de um vector, ou ainda `dot(v,w)`. O comprimento (ou módulo) de um vector $\mathbf{v}$ é dado pela sua norma euclidiana

dot

$$\|\mathbf{v}\| = \sqrt{(\mathbf{v}, \mathbf{v})} = \sqrt{\sum_{k=1}^{n} v_k^2}$$

e pode-se calcular com o comando `norm(v)`.

norm

O produto vectorial de dois vectores $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, $n \geq 3$, que se designa por $\mathbf{v} \times \mathbf{w}$ ou $\mathbf{v} \wedge \mathbf{w}$, é o vector $\mathbf{u} \in \mathbb{R}^n$ ortogonal a $\mathbf{v}$ e $\mathbf{w}$ cujo módulo é $|\mathbf{u}| = |\mathbf{v}|\,|\mathbf{w}| \sin(\alpha)$, onde α é o ângulo formado por $\mathbf{v}$ e $\mathbf{w}$. Pode-se obter com o comando `cross(v,w)`.

cross

Em MATLAB, pode-se visualizar um vector usando o comando `quiver` em $\mathbb{R}^2$ e `quiver3` em $\mathbb{R}^3$.

quiver
quiver3

Os comandos de MATLAB `x.*y` ou `x.^2` indicam que as operações são efectuadas componente a componente. Por exemplo, definindo os vectores

.*
.^

```
» v = [1; 2; 3]; w = [4; 5; 6];
```

a instrução

```
» w'*v

  ans =
       32
```

dá o seu produto escalar, enquanto que

```
» w.*v

  ans =
        4
       10
       18
```

dá um vector cuja i-ésima componente é igual a $x_i y_i$.

Por fim, recordemos que um vector $\mathbf{v} \in \mathbb{C}^n$, $\mathbf{v} \neq \mathbf{0}$, é um *vector próprio* de uma matriz $\mathrm{A} \in \mathbb{C}^{n\times n}$ associado ao número complexo λ se

$$\mathrm{A}\mathbf{v} = \lambda\mathbf{v}.$$

O número complexo λ diz-se *valor próprio* de A. Em geral, o cálculo dos valores próprios é bastante difícil. Exceptuam-se os casos das matrizes diagonais e triangulares, cujos valores próprios são simplesmente os seus termos diagonais.

Ver os Exercícios 1.3-1.6.

1.4 Funções reais

Esta secção ocupa-se de funções reais definidas no intervalo (a, b). O comando `fplot(fun,lims)` traça o gráfico da função `fun` (definida por uma cadeia de caracteres) no intervalo (`lims(1)`,`lims(2)`). Por exemplo, para representar $f(x) = 1/(1+x^2)$ no intervalo $(-5, 5)$, podemos escrever

fplot

```
» fun ='1/(1+x.^2)'; lims=[-5,5]; fplot(fun,lims);
```

ou, mais directamente,

```
» fplot('1/(1+x.^2)',[-5 5]);
```

Em MATLAB o gráfico obtém-se a partir de uma amostra da função num conjunto de pontos não equidistantes e reproduz o gráfico real de f com uma tolerância de 0.2%. Para melhorar a precisão poderíamos usar o comando

```
» fplot(fun,lims,tol,n,'LineSpec',P1,P2,...)
```

onde `tol` indica a tolerância desejada e o parâmetro `n`(≥ 1) assegura que a função será traçada com um mínimo de `n` + 1 pontos. `LineSpec` é uma cadeia de caracteres que especifica o tipo de linha ou a cor da linha usada para traçar o gráfico. Por exemplo, `LineSpec='-'` indica

uma linha a tracejado, `LineSpec='r-.'` uma linha vermelha de traços e pontos, etc. Para utilizar valores por defeito para `tol`, `n` ou `LineSpec` podem introduzir-se matrizes vazias (`[ ]`).

O valor de uma função `fun` num ponto `x` determina-se escrevendo
eval `y=eval(fun)`, depois de iniciar `x`. O valor correspondente é guardado em `y`. Notar que `x`, e portanto `y`, podem ser vectores. Ao usar este comando, a restrição é que o argumento da função `fun` deverá ser `x`. Se o argumento de `fun` tiver um nome diferente (o que acontece frequentemente quando este argumento é gerado no interior de um programa) o comando `eval` deverá substituir-se por `feval` (ver Observação 1.3).

grid Assinalemos por fim que se escrevermos `grid on` depois do comando `fplot`, podemos obter uma grelha de fundo como a da Figura 1.1.

Octave 1.3 Quando se utiliza o comando `fplot(fun,lims,n)` em Octave, o gráfico obtém-se a partir de uma amostra da função definida por `fun` (esse é o nome de uma *função* ou de uma expressão que contenha `x`) sobre um conjunto de pontos não equidistantes. O parâmetro opcional `n` (≥ 1) garante que a função será traçada com um mínimo de `n`+1 pontos. Por exemplo, os comandos seguintes permitem traçar o gráfico de $f(x) = 1/(1+x^2)$:

```
» fun ='1./(1+x.^2)'; lims=[-5,5];
» fplot(fun,lims)
```
■

1.4.1 Os zeros

Recorde-se que se $f(\alpha) = 0$, α diz-se *zero* de f ou *raiz* da equação $f(x) = 0$. Um zero é *simples* se $f'(\alpha) \neq 0$, e *múltiplo* no caso contrário.

Podemos determinar os zeros reais de uma função a partir do seu gráfico (com uma certa tolerância). O cálculo directo de todos os zeros de uma dada função nem sempre é possível. Para as funções polinomiais de grau n com coeficientes reais, isto é, da forma

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = \sum_{k=0}^{n} a_k x^k, \quad a_k \in \mathbb{R},\ a_n \neq 0,$$

podemos obter o único zero $\alpha = -a_0/a_1$, quando $n = 1$ (o gráfico de p_1 é uma linha recta) ou se existirem, os dois zeros α_+ e α_-, quando $n = 2$ (o gráfico de p_2 é uma parábola)

$$\alpha_\pm = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0a_2}}{2a_2}.$$

Contudo, não existem fórmulas explícitas para determinar os zeros de um polinómio arbitrário p_n quando $n \geq 5$.

No que se segue iremos designar por $\mathbb{P}_n$ o espaço dos polinómios de grau menor ou igual a n,

$$p_n(x) = \sum_{k=0}^{n} a_k x^k \tag{1.9}$$

onde a_k são coeficientes dados, reais ou complexos.

Em geral, o número de zeros de uma função não pode ser determinado *a priori*. Exceptua-se o caso particular das funções polinomiais, para as quais o número de zeros (reais ou complexos) coincide com o grau do polinómio. Além disso, se o complexo $\alpha = x + iy$ com $y \neq 0$ for um zero de um polinómio com coeficientes reais, o seu conjugado $\bar{\alpha} = x - iy$ também é um zero.

Em MATLAB podemos utilizar o comando `fzero(fun,x0)` para calcular um zero de uma função `fun`, na vizinhança de um dado valor `x0`, real ou complexo. O resultado é um valor aproximado desse zero, e também o intervalo onde se efectuou a sua pesquisa. Com o comando `fzero(fun,[x0 x1])`, procura-se um zero de `fun` no intervalo cujos extremos são `x0,x1`, desde que f mude de sinal entre `x0` e `x1`. `fzero`

Consideremos, por exemplo, a função $f(x) = x^2 - 1 + e^x$. Observando o seu gráfico, verifica-se que tem dois zeros em $(-1, 1)$. Para os calcular, precisamos de executar os seguintes comandos:

```
» fun=inline('x^2 - 1 + exp(x)','x');
» fzero(fun,1)

 ans =
   5.4422e-18

» fzero(fun,-1)

 ans =
   -0.7146
```

Em alternativa, depois de constatar pela função `plot` que existe um zero no intervalo $[-1, -0.2]$ e outro em $[-0.2, 1]$, poderia ter-se escrito:

```
» fzero(fun,[-0.2 1])

 ans =
  -5.2609e-17

» fzero(fun,[-1 -0.2])

 ans =
   -0.7146
```

O resultado obtido para o primeiro zero é ligeiramente diferente do precedente, devido à diferente iniciação do algoritmo implementado em `fzero`.

No Capítulo 2, serão introduzidos e estudados vários métodos para o cálculo aproximado dos zeros de uma função arbitrária.

Octave 1.4 Em Octave, `fzero` só aceita funções definidas com a palavra chave `function`. A sintaxe correspondente é a seguinte:

```
function y = fun(x)
  y = x.^2 - 1 + exp(x);
end

» fzero("fun", 1)

  ans =  2.3762e-17

» fzero("fun",-1)

  ans =  -0.71456
```

■

1.4.2 Polinómios

polyval

Os polinómios são funções muito particulares tratadas em MATLAB de forma especial por uma *toolbox*[1] `polyfun`. O comando `polyval` permite determinar o valor de um polinómio em um ou vários pontos. Os seus argumentos de entrada são um vector `p` e um vector `x`, em que as componentes de `p` são os coeficientes do polinómio guardados por ordem decrescente, de a_n a a_0, e as componentes de `x` são os pontos onde o valor do polinómio será determinado. O resultado pode ser guardado num vector `y`, escrevendo

```
» y = polyval(p,x)
```

Por exemplo, os valores de $p(x) = x^7 + 3x^2 - 1$, nos pontos equidistantes $x_k = -1 + k/4$ para $k = 0, \ldots, 8$, podem-se obter procedendo da seguinte maneira:

```
» p = [1 0 0 0 0 3 0 -1]; x = [-1:0.25:1];
» y = polyval(p,x)

  y =
   Columns 1 through 5:

     1.00000    0.55402   -0.25781   -0.81256   -1.00000
   Columns 6 through 9:

    -0.81244   -0.24219    0.82098    3.00000
```

Em alternativa, poderá usar-se o comando `feval`. Contudo, neste caso seria necessário fornecer a expressão analítica completa do polinómio numa cadeia de caracteres, e não apenas os seus coeficientes.

roots

O programa `roots` dá uma aproximação dos zeros de um polinómio e requer apenas a entrada do vector `p`.

Por exemplo, para calcular os zeros de $p(x) = x^3 - 6x^2 + 11x - 6$, escreve-se:

[1] Uma *toolbox* é uma colecção de funções de MATLAB associadas a uma aplicação particular

```
» p = [1 -6 11 -6]; format long;
» roots(p)

  ans =
       3.00000000000000
       2.00000000000000
       1.00000000000000
```

Infelizmente o resultado nem sempre é preciso. Por exemplo, no caso do polinómio $p(x) = (x+1)^7$, cujo único zero é $\alpha = -1$ com multiplicidade 7, obtemos (surpreendentemente):

```
» p = [1 7  21 35  35  21  7  1];
» roots(p)

  ans =
       -1.0101
       -1.0063 + 0.0079i
       -1.0063 - 0.0079i
       -0.9977 + 0.0099i
       -0.9977 - 0.0099i
       -0.9909 + 0.0044i
       -0.9909 - 0.0044i
```

Com efeito, os métodos numéricos para o cálculo das raízes de um polinómio são particularmente sensíveis a erros de arredondamento, se as raízes forem de multiplicidade maior que 1 (ver Secção 2.5.2).

Acrescente-se que com o comando `p=conv(p1,p2)` se obtém os coeficientes do polinómio resultante do produto de dois polinómios cujos coeficientes estão contidos nos vectores `p1` e `p2`. Do mesmo modo, o comando `[q,r]=deconv(p1,p2)` dá os coeficientes do quociente `q` e do resto `r` da divisão de `p1` por `p2`, isto é, `p1 = conv(p2,q) + r`.

conv

deconv

Consideremos por exemplo o produto e o quociente de dois polinómios $p_1(x) = x^4 - 1$ e $p_2(x) = x^3 - 1$:

```
» p1 = [1 0 0 0 -1];
» p2 = [1 0 0 -1];
» p=conv(p1,p2)

p =
     1     0     0    -1    -1     0     0     1

» [q,r]=deconv(p1,p2)

q =
     1     0
r =
     0     0     0     1    -1
```

Encontramos assim os polinómios $p(x) = p_1(x)p_2(x) = x^7 - x^4 - x^3 + 1$, $q(x) = x$ e $r(x) = x - 1$ tais que $p_1(x) = q(x)p_2(x) + r(x)$.

Os comandos `polyint(p)` e `polyder(p)` fornecem, respectivamente, os coeficientes da primitiva (que se anula em $x = 0$) e da derivada do polinómio cujos coeficientes são dados pelas componentes do vector `p`.

polyint

polyder

Se `x` for um vector de abcissas e se `p` (respectivamente, $\texttt{p}_1$ e $\texttt{p}_2$) for um vector que contém os coeficientes de um polinómio p (respectivamente, p_1 e p_2), os comandos precedentes resumem-se na Tabela 1.1.

comandos	resultados
`y=polyval(p,x)`	`y` = valores de $p(x)$
`z=roots(p)`	`z` = raízes de p tais que $p(z)=0$
`p=conv(`$\texttt{p}_1$`,`$\texttt{p}_2$`)`	`p` = coeficientes do polinómio $p_1 p_2$
`[q,r]=deconv(`$\texttt{p}_1$`,`$\texttt{p}_2$`)`	`q` = coeficientes de q, `r` = coeficientes de r tais que $p_1 = qp_2 + r$
`y=polyder(p)`	`y` = coeficientes de $p'(x)$
`y=polyint(p)`	`y` = coeficientes de $\int_0^x p(t)\,dt$

Tabela 1.1. Comandos de MATLAB para operações com polinómios

`polyfit` Um outro comando, `polyfit`, permite calcular os $n+1$ coeficientes do polinómio p de grau n, desde que se disponha dos valores de p em $n+1$ pontos distintos (ver Secção 3.1.1).

`polyderiv` `polyinteg` **Octave 1.5** Os comandos `polyderiv` e `polyinteg` têm as mesmas funcionalidades que `polyder` e `polyint`, respectivamente. Notar que `polyder` está igualmente disponível em Octave, ver Secção 1.6. ■

1.4.3 Integração e derivação

Neste livro serão frequentemente invocados os seguintes resultados:

1. *teorema fundamental da integração*: se f for uma função contínua em $[a,b)$, então

$$F(x) = \int_a^x f(t)\,dt \qquad \forall x \in [a,b),$$

é uma função derivável, chamada a *primitiva* de f, que satisfaz,

$$F'(x) = f(x) \qquad \forall x \in [a,b);$$

2. *primeiro teorema do valor médio para integrais*: se f for uma função contínua em $[a,b)$ e x_1, $x_2 \in [a,b)$ com $x_1 < x_2$, então $\exists \xi \in (x_1, x_2)$ tal que

$$f(\xi) = \frac{1}{x_2 - x_1} \int_{x_1}^{x_2} f(t)\,dt.$$

Mesmo quando existe, uma primitiva pode ser difícil ou mesmo impossível de determinar. Por exemplo, é inútil saber que $\ln|x|$ é uma primitiva de $1/x$ se não se souber calcular os logaritmos de modo eficiente. No Capítulo 4 iremos introduzir vários métodos para calcular, com uma dada precisão, o integral de uma função contínua arbitrária, sem o conhecimento da sua primitiva.

Recordemos que uma função f definida num intervalo $[a, b]$ é derivável num ponto $\bar{x} \in (a, b)$ se existir e for finito o seguinte limite

$$f'(\bar{x}) = \lim_{h \to 0} \frac{1}{h}(f(\bar{x} + h) - f(\bar{x})). \tag{1.10}$$

O valor de $f'(\bar{x})$ dá o declive da recta tangente ao gráfico de f no ponto $\bar{x}$.

Dizemos que uma função contínua e com derivada contínua em todos os pontos de $[a, b]$ pertence ao espaço $C^1([a, b])$. Mais geralmente, uma função com derivadas contínuas até à ordem p (um inteiro positivo) diz-se que pertence a $C^p([a, b])$. Em particular, $C^0([a, b])$ designa o espaço das funções contínuas em $[a, b]$.

Um resultado que será muitas vezes usado é o *teorema do valor médio*: se $f \in C^1([a, b])$, existe $\xi \in (a, b)$ tal que

$$f'(\xi) = (f(b) - f(a))/(b - a).$$

Recordemos, finalmente, que uma função que, numa vizinhança de x_0, é contínua e admite derivadas contínuas até à ordem n, pode ser aproximada nessa vizinhaça pelo chamado *polinómio de Taylor de grau* n no ponto x_0:

$$\begin{aligned} T_n(x) &= f(x_0) + (x - x_0)f'(x_0) + \ldots + \frac{1}{n!}(x - x_0)^n f^{(n)}(x_0) \\ &= \sum_{k=0}^{n} \frac{(x - x_0)^k}{k!} f^{(k)}(x_0). \end{aligned}$$

A *toolbox* `symbolic` de MATLAB contém os comandos `diff`, `int` e `taylor` que fornecem, respectivamente, a expressão analítica da derivada, o integral indefinido (isto é, uma primitiva) e o polinómio de Taylor, de uma dada função. Em particular, tendo definido uma função com a cadeia de caracteres `f`, a função `diff(f,n)` dá a sua derivada de ordem `n`, `int(f)` o seu integral indefinido, e `taylor(f,x,n+1)` o seu polinómio de Taylor de grau `n` numa vizinhança de $x_0 = 0$. A variável `x` deve ser declarada como *simbólica* usando o comando `syms x`. Isto permitirá a sua manipulação algébrica sem especificar o seu valor.

`diff` `int` `taylor` `syms`

Para aplicar à função $f(x) = (x^2 + 2x + 2)/(x^2 - 1)$, procede-se do seguinte modo:

Figura 1.5. Interface gráfica do comando `funtool`

```
» f = '(x^2+2*x+2)/(x^2-1)';
» syms x
» diff(f)

 (2*x+2)/(x^2-1)-2*(x^2+2*x+2)/(x^2-1)^2*x

» int(f)

 x+5/2*log(x-1)-1/2*log(1+x)

» taylor(f,x,6)

 -2-2*x-3*x^2-2*x^3-3*x^4-2*x^5
```

`simple` Observemos que o comando `simple` permite reduzir as expressões geradas por `diff`, `int` e `taylor` de modo a torná-las tão simples quanto pos-
`funtool` sível. O comando `funtool` permite uma simples manipulação simbólica de funções arbitrárias, usando a interface gráfica ilustrada na Figura 1.5.

Octave 1.6 O cálculo simbólico ainda não está disponível em Octave, mas o desenvolvimento desta funcionalidade é objecto de trabalhos em curso.[2] ■

Ver os Exercícios 1.7-1.8.

1.5 Errar não é só humano

Reformulando a frase em latim *errare humanum est*, poderíamos mesmo dizer que em cálculo numérico errar é mesmo inevitável.

[2] `http://www.octave.org`

Tal como vimos, o simples facto de usar um computador para representar números reais introduz erros. Por conseguinte, mais do que esforçar-se por eliminar erros, o que importa é procurar controlar os seus efeitos.

Em geral, é possível identificar vários níveis de erros que ocorrem na aproximação e resolução de um problema físico (ver Figura 1.6).

No nível superior, encontra-se o erro e_m que resulta de forçar a realidade física (PF designa o problema físico e x_f a sua solução) a satisfazer algum modelo matemático (MM, cuja solução é x). Tais erros limitarão a aplicabilidade do modelo matemático a certas situações e saem do domínio do Cálculo Científico.

Um modelo matemático (expresso por um integral, como no exemplo da Figura 1.6, por uma equação algébrica ou diferencial, ou por um sistema linear ou não linear) geralmente não se pode resolver de forma explícita. A sua resolução por algoritmos numéricos envolverá necessariamente pelo menos o aparecimento e propagação de erros de arredondamento. Chamemos e_a a estes erros.

Por outro lado, é muitas vezes necessário introduzir erros adicionais, uma vez que o cálculo no computador de soluções de modelos matemáticos, que envolvam um número infinito de operações aritméticas, só se efectua de modo aproximado. Por exemplo, a soma de uma série será necessariamente calculada de modo aproximado, procedendo a uma truncatura conveniente.

Torna-se por isso necessário definir um problema numérico, PN, cuja solução x_n difere de x por um erro e_t que se chama *erro de truncatura*. Estes erros não ocorrem apenas nos modelos matemáticos já definidos em dimensão finita (por exemplo, quando se resolve um sistema linear).

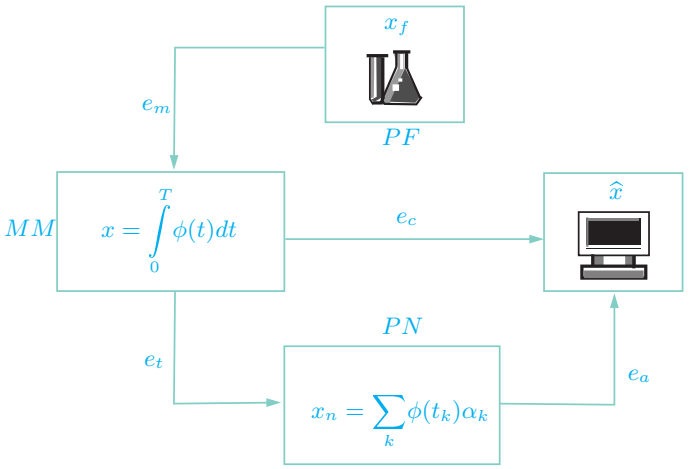

Figura 1.6. Diversos tipos de erro num processo de cálculo

A soma dos erros e_a e e_t constitui o *erro computacional* e_c, que é a quantidade que nos interessa.

O erro computacional *absoluto* é a diferença entre x, a solução exacta do modelo matemático, e $\widehat{x}$, a solução obtida no final do processo numérico,

$$e_c^{abs} = |x - \widehat{x}|,$$

enquanto que (se $x \neq 0$) o erro computacional *relativo* é

$$e_c^{rel} = |x - \widehat{x}|/|x|,$$

onde $|\cdot|$ designa o módulo, ou qualquer outra medida (valor absoluto, norma) dependendo do significado de x.

O cálculo numérico consiste geralmente em aproximar o modelo matemático fazendo intervir um parâmetro de discretização, que designaremos por h e que se supõe positivo. Se, quando h tender para 0, a solução aproximada do modelo matemático tender para a sua solução exacta, diremos que o método numérico é *convergente*. Além disso, se o erro (absoluto ou relativo) for majorado por uma função de h da seguinte maneira

$$\boxed{e_c \leq Ch^p} \tag{1.11}$$

em que C é um número positivo independente de h e p , diremos que o método é *convergente de ordem* p. Quando, para além de um majorante (1.11), se tem também um minorante $C'h^p \leq e_c$ (C' é uma outra constante independente de h e p) pode-se substituir o símbolo $\leq$ por $\simeq$.

Exemplo 1.1 Suponhamos que se aproxima a derivada de uma função f no ponto $\bar{x}$ pela razão incremental que aparece em (1.10). Obviamente, se f for derivável em $\bar{x}$, o erro cometido quando se substitui f' pela razão incremental tende para 0 quando $h \to 0$. Contudo, como veremos na Secção 4.1, o erro só se comporta como Ch se $f \in C^2$ numa vizinhança de $\bar{x}$. ■

Ao estudar as propriedades de convergência de um método numérico recorre-se frequentemente a gráficos que representam o erro em função de h numa escala logarítmica isto é, com $\log(h)$ no eixo das abcissas e $\log(e_c)$ no eixo das ordenadas. O objectivo desta representação é claro: se $e_c = Ch^p$ então $\log e_c = \log C + p \log h$. Portanto, em escala logarítmica p representa o declive da linha recta $\log e_c$ e, deste modo, se pretendermos comparar dois métodos, a recta de maior declive corresponde ao método de ordem mais elevada. Os gráficos na escala logarítmica obtêm-se facilmente em MATLAB, bastando para isso digitar `loglog(x,y)`, sendo `x` e `y` os vectores que contêm as abcissas e as ordenadas dos dados a representar.

loglog

Por exemplo, na Figura 1.7 as linhas rectas representam o comportamento do erro em dois métodos diferentes. A linha contínua corresponde

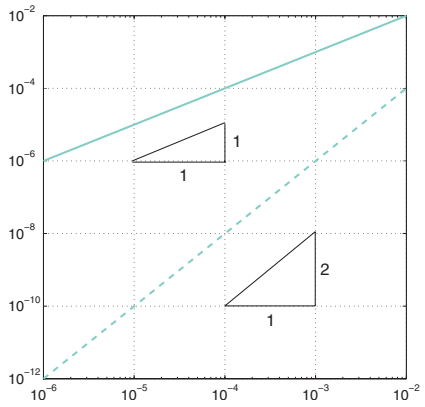

Figura 1.7. Gráfico em escala logarítmica

a uma aproximação de primeira ordem, enquanto que a linha a tracejado corresponde a uma aproximação de segunda ordem.

Há uma alternativa ao processo gráfico para determinar a ordem de um método quando se conhece os erros e_i relativos para alguns valores h_i do parâmetro de discretização, $i = 1, \ldots, N$: consiste em supor que e_i é igual a Ch_i^p, onde C não depende de i. Podemos então aproximar p pelos valores

$$p_i = \log(e_i/e_{i-1})/\log(h_i/h_{i-1}), \qquad i = 2, \ldots, N. \tag{1.12}$$

Com efeito, o erro não se pode calcular directamente uma vez que depende da solução desconhecida. Torna-se por isso necessário introduzir valores calculáveis, as chamadas *estimativas de erro*, que permitam estimar o próprio erro. Veremos alguns exemplos nas Secções 2.2.1, 2.3 e 4.4.

1.5.1 Falando de custos

Em geral, resolve-se um problema no computador usando um algoritmo, que é uma instrução precisa em forma de texto, especificando a execução de uma sequência finita de operações elementares.

O *custo computacional* de um algoritmo é o número de operações com vírgula flutuante requeridas para a sua execução. Muitas vezes, a velocidade de um computador mede-se pelo número máximo de operações com vírgula flutuante que pode executar num segundo (*flops*). Em particular, é comum usar as seguintes notações abreviadas: Mega-flops igual a 10^6 *flops*, Giga-flops igual a 10^9 *flops*, Tera-flops igual a 10^{12} *flops*. Hoje em dia, os computadores mais rápidos atingem cerca de 40 Tera-flops.

De um modo geral, o conhecimento do número exacto de operações efectuadas por um certo algoritmo não é essencial. Bastará, em vez disso, determinar a sua ordem de grandeza em função de um parâmetro d associado à dimensão do problema. Assim, dizemos que um algoritmo tem complexidade *constante* se usar um número de operações independente de d, isto é $\mathcal{O}(d)$ operações; tem complexidade *linear* se requerer $\mathcal{O}(d)$ operações, ou, mais geralmente, complexidade *polinomial* se usar $\mathcal{O}(d^m)$ operações, em que m é um inteiro positivo. Outros algoritmos podem ter complexidade *exponencial* ($\mathcal{O}(c^d)$ operações) ou mesmo *factorial* ($\mathcal{O}(d!)$ operações). Recordemos que o símbolo $\mathcal{O}(d^m)$ significa "comporta-se, para grandes valores de d, como uma constante vezes d^m".

Exemplo 1.2 (produto matriz-vector) Seja A uma matriz quadrada de ordem n e seja $\mathbf{v}$ um vector de $\mathbb{R}^n$. A j-ésima componente do produto A$\mathbf{v}$ é dada por

$$a_{j1}v_1 + a_{j2}v_2 + \ldots + a_{jn}v_n,$$

o que requer n produtos e $n-1$ adições. Assim, é necessário efectuar $n(2n-1)$ operações para calcular todas as componentes. Deste modo, o algoritmo envolve $\mathcal{O}(n^2)$ operações, e tem por isso uma complexidade quadrática relativamente ao parâmetro n. O mesmo algoritmo necessitaria de $\mathcal{O}(n^3)$ operações para calcular o produto de duas matrizes de ordem n. Contudo, existe um algoritmo devido a Strassen, que requer "apenas" $\mathcal{O}(n^{\log_2 7})$ operações e um outro, devido a Winograd e Coppersmith, que requer a execução de $\mathcal{O}(n^{2.376})$ operações. ■

Exemplo 1.3 (cálculo do determinante de uma matriz) Tal como foi anteriormente mencionado, o determinante de uma matriz quadrada de ordem n pode ser calculado usando a fórmula recursiva (1.8). O algoritmo correspondente tem uma complexidade factorial em n e só seria aplicável a matrizes de pequena dimensão. Por exemplo, se $n = 24$, um computador capaz de realizar 1 Peta-flops (isto é, 10^{15} operações com vírgula flutuante por segundo) necessitaria de 20 anos para executar este cálculo. Torna-se por isso necessário recorrer a algoritmos mais eficazes. Na verdade, existe um algoritmo que permite o cálculo de determinantes à custa de produtos matriz-matriz com uma complexidade de $\mathcal{O}(n^{\log_2 7})$ operações aplicando o algoritmo de Strassen mencionado (ver [BB96]). ■

O número de operações não é o único parâmetro a ter em conta na análise de um algoritmo. Um outro factor importante é o tempo de acesso à memória do computador (que depende da maneira como o algoritmo foi programado). Um indicador do desempenho de um algoritmo é por isso o tempo CPU (CPU significa *central processing unit*), quer dizer o tempo de cálculo. Em MATLAB pode ser obtido com o comando `cputime`. O tempo total que decorre entre as fases de *entrada* e *saída* pode-se obter com o comando `etime`.

`cputime`

`etime`

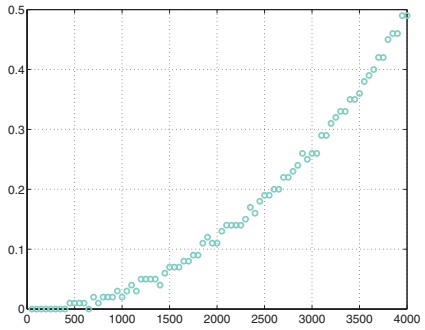

Figura 1.8. Produto matriz-vector: o tempo CPU (em segundos) em função da dimensão n da matriz (num PC a 2.53 GHz)

Exemplo 1.4 Para calcular o tempo necessário para executar um produto matriz-vector, considera-se o seguinte programa:

```
» n = 4000; step = 50; A = rand(n,n); v = rand(n); T=[];
» sizeA = [ ]; count = 1;
» for k = 50:step:n
    AA = A(1:k,1:k); vv = v(1:k)';
    t = cputime;  b = AA*vv; tt = cputime - t;
    T = [T, tt]; sizeA = [sizeA,k];
end
```

A instrução `a:step:b` que aparece no ciclo `for` gera todos os números da forma `a+step*k`, onde `k` é um inteiro que varia de `0` a `kmax`, sendo `kmax` o maior inteiro tal que `a+step*kmax` é menor que `b` (no caso considerado, `a=50`, `b=4000` e `step=50`). O comando `rand(n,m)` define uma matriz `n`×`m` cujos elementos são aleatórios. Por fim, `T` é o vector cujas componentes contêm o tempo CPU necessário para executar cada produto matriz-vector, enquanto que `cputime` dá o tempo CPU (em segundos) utilizado pelo processo de MATLAB desde o seu início. O tempo necessário à execução de um único programa é, portanto, a diferença entre o tempo CPU efectivo e o calculado antes da execução do programa em curso, guardado na variável `t`. O gráfico da Figura 1.8, obtido com o comando `plot(sizeA,T,'o')`, mostra que o tempo CPU aumenta como o quadrado da ordem da matriz `n`. ■

`a:step:b`

`rand`

1.6 Os ambientes MATLAB e Octave

Os programas de MATLAB e Octave são ambientes integrados de cálculo científico e visualização. Estão escritos nas linguagens C e C++.

MATLAB é distribuído por The MathWorks (ver o endereço na web `www.mathworks.com`). O seu nome vem de *MATrix LABoratory* porque foi inicialmente desenvolvido para o cálculo matricial.

Octave, também conhecido por GNU Octave (ver o endereço na web `www.octave.org`), é um *software* que se distribui livremente. É possível

redistribuí-lo e/ou modificá-lo nos termos da licença GNU *General Public License* (GPL) publicada pela *Free Software Foundation*.

Tal como mencionámos no início deste capítulo, existem diferenças entre os ambientes, as linguagens e as *toolboxes* de MATLAB e Octave. Contudo, o seu nível de compatibilidade permite executar a maior parte dos programas deste livro, tanto em MATLAB como em Octave. Quando tal não for possível, porque os comandos não têm a mesma sintaxe, ou porque funcionam de modo diferente, ou apenas porque não existem, escrever-se-á uma nota no fim de cada secção que fornece uma explicação e indica o que deverá ser feito.

Assim como MATLAB tem as suas *toolboxes*, Octave possui um vasto conjunto de funções disponíveis através do projecto chamado Octave-forge (ver o sítio na web `octave.sourceforge.net`). Este repositório de funções não cessa de se enriquecer em muitas áreas diferentes, tais como álgebra linear, matrizes esparsas, optimização, para mencionar apenas algumas. Para executar correctamente todos os programas e exemplos deste livro em Octave, é indispensável instalar Octave-forge.

Depois de instalar MATLAB ou Octave, podemos aceder a um ambiente de trabalho caracterizado pelo *prompt* » em MATLAB e `octave:1>` em Octave, respectivamente. Por exemplo, quando executamos o MATLAB no nosso computador pessoal vemos

```
                    < M A T L A B >
          Copyright 1984-2004 The MathWorks, Inc.
             Version 7.0.0.19901 (R14)
                    May 06, 2004

     To get started, select MATLAB Help or Demos from the Help menu.

»
```

Por outro lado, quando executamos Octave vemos

```
GNU Octave, version 2.1.72 (x86_64-pc-linux-gnu).
Copyright (C) 2005 John W. Eaton.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTIBILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

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/help-wanted.html

Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful
report).

octave:1>
```

1.7 A linguagem MATLAB

Depois das observações introdutórias da secção anterior, estamos em condições de trabalhar nos ambientes MATLAB ou Octave. A partir de agora MATLAB designará um subconjunto de comandos comuns a MATLAB e Octave.

Depois de digitar *entrada* (ou *saída*), tudo o que se escrever depois do *prompt* será interpretado. [3] Mais precisamente, MATLAB verifica primeiro se o que foi escrito corresponde a variáveis já definidas ou ao nome de um dos programas ou comandos definidos em MATLAB. Se estas verificações falharem, MATLAB dá uma mensagem de erro. Caso contrário, o comando é executado e irá possivelmente visualizar-se uma *saída*. Em todos os casos, o sistema volta de novo ao *prompt* para manifestar que está pronto a receber novos comandos. Para terminar uma sessão de MATLAB devemos escrever o comando `quit` (ou `exit`) e digitar *enter*. A partir de agora deixaremos de indicar que para a execução de qualquer programa ou comando se torna necessário digitar *enter*. Além disso, utilizaremos indiferentemente os termos programa, função ou comando. Quando o nosso comando se limitar a uma das estruturas elementares de MATLAB (por exemplo, um número ou uma cadeia de caracteres entre aspas) essa estrutura obtém-se imediatamente em *saída* na variável por *defeito* `ans` (abreviatura do inglês *answer*). Por exemplo:

quit
exit

ans

```
» 'casa'

ans =
    casa
```

Se escrevermos em seguida uma nova cadeia (ou número) de caracteres, `ans` tomará este novo valor.

Podemos desactivar a apresentação automática da *saída* escrevendo um ponto e vírgula depois da cadeia de caracteres. Assim, se escrevermos `'casa';` MATLAB irá simplesmente devolver o *prompt* (associando contudo o valor `'casa'` à variável `ans`).

Mais geralmente, o comando `=` permite atribuir um valor (ou uma cadeia de caracteres) a uma dada variável. Por exemplo, para afectar a cadeia `'Benvindo a Lisboa'` à variável `a`, podemos escrever

=

```
» a='Benvindo a Lisboa';
```

Por conseguinte, não é necessário declarar o *tipo* de uma variável, já que MATLAB o fará automática e dinamicamente. Por exemplo, se escrevermos `a=5`, a variável `a` irá agora conter um número e não uma cadeia de caracteres. Esta flexibilidade não é gratuita. Por exemplo, se fixarmos uma variável chamada `quit` e lhe atribuirmos o número `5`, estamos a inibir o uso do comando `quit` de MATLAB. Evitaremos por isso

[3] Assim, um programa de MATLAB não tem necessidade de ser compilado, contrariamente a outras linguagens, como o Fortran ou o C.

utilizar variáveis com o nome de comandos do MATLAB. No entanto, o
clear comando `clear` seguido do nome da variável (por exemplo, `quit`), permite cancelar a definição e recuperar o significado original do comando `quit`.

save Através do comando `save` todas as variáveis da sessão (guardadas no chamado *espaço de trabalho*) estão no ficheiro binário `matlab.mat`. Estes
load dados podem ser recuperados com o comando `load`. Depois de `save` ou `load` pode-se especificar o nome de um ficheiro. Para salvar variáveis seleccionadas tais como `v1`, `v2` e `v3`, num dado ficheiro chamado por exemplo `area.mat`, a sintaxe é `save area v1 v2 v3`.

help O comando `help` permite visualizar todos os comandos e variáveis pré-definidos, incluindo as chamadas *toolboxes* que são conjuntos de comandos especializados. Entre eles recordemos os que definem as funções
sin cos elementares como o seno (`sin(a)`), o coseno (`cos(a)`), a raiz quadrada
sqrt exp (`sqrt(a)`), a exponencial (`exp(a)`).

Certos caracteres especiais não podem fazer parte do nome de uma variável ou de um comando. É, por exemplo, o caso dos operadores al-
+ - * / gébricos (`+`, `-`, `*` e `/`), dos operadores lógicos *and* (`&`), *or* (`|`), *not* (`~`),
& | ~ e dos operadores de comparação *maior que* (`>`), *maior ou igual a* (`>=`),
> >= < *menor que* (`<`), *menor ou igual a* (`<=`), *igual a* (`==`). Finalmente, assi-
<= == nalemos ainda que um nome nunca pode começar por um dígito, uma chaveta ou qualquer sinal de pontuação.

1.7.1 Instruções de MATLAB

Uma linguagem especial de programação, a linguagem MATLAB, permite igualmente ao utilizador escrever novos programas. Apesar de não ser necessário dominar o MATLAB para poder utilizar os diversos programas propostos neste livro, o seu conhecimento poderá permitir ao leitor a capacidade de modificar e escrever novos programas.

A linguagem MATLAB inclui instruções usuais tais como testes e ciclos.

O teste *if-elseif-else* tem a seguinte forma geral:

```
if condição(1)
   instrução(1)
elseif condição(2)
   instrução(2)
  .
  .
  .
else
   instrução(n)
end
```

onde `condição(1)`, `condição(2)`, ... representam conjuntos de instruções lógicas de MATLAB, cujos valores são 0 ou 1 (falso ou verdadeiro). A primeira condição com o valor 1 implica a execução da ins-

trução correspondente. Se todas as condições forem falsas, a `instrução-(n)` será executada. Se o valor da `condição(k)` for 0, passa-se à condição seguinte.

Por exemplo, para calcular as raízes de um polinómio quadrático $ax^2 + bx + c$ podemos usar as seguintes instruções (o comando `disp(.)` afixa simplesmente o que está escrito entre parêntesis):

```
» if  a  ~= 0
    sq = sqrt(b*b - 4*a*c);
    x(1) = 0.5*(-b + sq)/a;
    x(2) = 0.5*(-b - sq)/a;
  elseif  b  ~= 0
    x(1) = -c/b;
  elseif  c  ~= 0
    disp(' Equação impossível');
  else
    disp(' A equação dada é uma identidade');
  end
```
(1.13)

Note-se que MATLAB não executa a construção completa antes de se ter digitado a instrução `end`.

MATLAB permite dois tipos de ciclos, um ciclo *for* (comparável ao ciclo *do* de Fortran ou ao ciclo *for* de C) e um ciclo *while*. Um ciclo *for* repete as instruções enquanto o seu índice percorrer os valores de um dado vector linha. Por exemplo, para calcular os primeiros seis termos de uma sucessão de Fibonacci $f_i = f_{i-1} + f_{i-2}$, para $i \geq 3$, com $f_1 = 0$ e $f_2 = 1$, podemos usar as seguintes instruções:

```
» f(1) = 0; f(2) = 1;
» for i = [3 4 5 6]
    f(i) = f(i-1) + f(i-2);
 end
```

Note-se que se pode usar um ponto e vírgula para separar várias instruções de MATLAB que aparecem na mesma linha. Observe-se ainda que se pode substituir a segunda instrução por `» for i = 3:6`, que é equivalente. O ciclo *while* repete um bloco de instruções enquanto a `condição` dada for verdadeira. Por exemplo, o seguinte conjunto de instruções pode ser usado como alternativa ao conjunto anterior:

```
» f(1) = 0; f(2) = 1; k = 3;
» while k <= 6
    f(k) = f(k-1) + f(k-2); k = k + 1;
 end
```

Existem outras instruções cujo uso é talvez menos frequente, como *switch*, *case*, *otherwise*. O leitor interessado poderá aceder à sua descrição através do comando `help`.

1.7.2 Programação em MATLAB

Expliquemos brevemente como escrever programas em MATLAB. Um novo programa deve ser colocado num ficheiro cujo nome inclua a exten-

são `.m`, chamado *m-file*. Estes ficheiros devem ser colocados num dos directórios onde o MATLAB procura automaticamente os *m-files*; a sua lista pode-se obter com o comando `path` (ver `help path` para aprender a adicionar um directório a esta lista). O primeiro directório examinado por MATLAB é o directório do trabalho em curso.

A este nível é importante distinguir entre *scripts* e *funções*. Um *script* é simplesmente uma colecção de comandos de MATLAB num *m-file* que pode ser usada interactivamente. Por exemplo, o conjunto de instruções (1.13) pode dar origem a um *script* (a que poderemos chamar `equation`) copiando-o no ficheiro `equation.m`. Para o lançar, escreve-se simplesmente a instrução `equation` depois do prompt » de MATLAB. Mostramos dois exemplos:

```
» a = 1; b = 1; c = 1;
» equation

  ans =
       -0.5000 + 0.8660i   -0.5000 - 0.8660i

» a = 0; b = 1; c = 1;
» equation

  ans =
       -1
```

Como não há interface de entrada/saída, todas as variáveis utilizadas num *script* são também as variáveis da sessão de trabalho. Estas variáveis só serão apagadas depois de digitar explicitamente o comando (`clear`). Isto não é de todo satisfatório, quando se pretende escrever programas complexos. Com efeito, estes envolvem muitas variáveis temporárias e, comparativamente, poucas variáveis de entrada/saída, que são as únicas que podem ser efectivamente guardadas quando terminr a execução do programa. Neste sentido, as *funções* são muito mais flexíveis do que os *scripts*.

Uma *função* também se define num *m-file*, por exemplo `nome.m`, mas tem uma interface de entrada/saída bem definida, que se introduz com o comando `function`

```
function [out1,...,outn]=name(in1,...,inm)
```

onde `out1,...,outn` são as variáveis de saída e `in1,...,inm` são as variáveis de entrada.

O ficheiro seguinte, chamado `det23.m`, define uma nova função, `det23` que, tendo em vista a fórmula dada na Secção 1.3, calcula o determinante de uma matriz de ordem 2 ou 3:

```
function det=det23(A)
%DET23 calcula o determinante de uma matriz quadrada
% de ordem 2 ou 3
[n,m]=size(A);
if n==m
  if n==2
    det = A(1,1)*A(2,2)-A(2,1)*A(1,2);
```

```
  elseif n == 3
    det = A(1,1)*det23(A([2,3],[2,3]))-...
          A(1,2)*det23(A([2,3],[1,3]))+...
          A(1,3)*det23(A([2,3],[1,2]));
  else
    disp(' Apenas matrizes 2x2 ou 3x3');
  end
else
  disp(' Apenas matrizes quadradas');
end
return
```

Note-se o uso de reticências `...` para indicar que a instrução continua na linha seguinte e do caracter % para iniciar comentários. A instrução `A([i,j],[k,l])` permite a construção de uma matriz 2×2 cujos elementos são os da matriz original A situados nas intersecções das `i`-ésima e `j`-ésima linhas com as `k`-ésima e `l`-ésima colunas.

Quando se chama uma *função*, MATLAB cria um espaço de trabalho local (o *espaço de trabalho da função*). Os comandos no interior da *função* não se podem referir às variáveis do espaço de trabalho global (interactivo) a não ser que estas passem como parâmetros de entrada. Em particular, as variáveis usadas numa função são apagadas quando a execução termina, a menos que sejam devolvidas como parâmetros de saída.

Observação 1.2 (variáveis globais) Como foi dito acima, cada função de MATLAB possui as suas próprias variáveis locais, que são disjuntas das variáveis das outras funções e das variáveis do espaço de trabalho. Contudo, se várias funções (e eventualmente o espaço de trabalho) declararem uma mesma variável como `global`, então todas elas partilham uma cópia desta variável. Qualquer modificação da variável em alguma das funções repercute-se em todas as funções que declarem essa variável como global. •

A execução de uma função termina em geral quando se atinge o fim do código correspondente. No entanto, a instrução `return` pode ser usada para forçar uma interrupção prematura (quando se verificar uma certa condição).

Por exemplo, construa-se uma função para aproximar o *número de ouro* $\alpha = 1.6180339887\ldots$, que é o limite quando $k \to \infty$ do quociente de dois termos consecutivos da sucessão de Fibonnacci f_k/f_{k-1}. Itera-se até que a diferença entre dois quocientes consecutivos seja inferior a 10^{-4}, e pode-se construir a seguinte função:

```
function [golden,k]=fibonacci0
f(1) = 0; f(2) = 1; goldenold = 0;
kmax = 100; tol = 1.e-04;
for k = 3:kmax
   f(k) = f(k-1) + f(k-2);
   golden = f(k)/f(k-1);
   if abs(golden - goldenold) <= tol
     return
   end
```

```
   goldenold = golden;
end
return
```

A sua execução interrompe-se ao fim de `kmax=100` iterações ou quando o valor absoluto da diferença entre duas iteradas consecutivas for menor do que `tol=1.e-04`. Podemos então escrever

```
» [alpha,niter]=fibonacci0

 alpha =
    1.61805555555556
 niter =
    14
```

Ao fim de 14 iterações a função deu origem a um valor aproximado em que os primeiros 5 algarismos significativos coincidem com os de α.

O número de parâmetros de entrada e de saída de uma função de MATLAB pode variar. Por exemplo, poderíamos modificar a *função* Fibonacci da seguinte maneira:

```
function [golden,k]=fibonacci1(tol,kmax)
if nargin == 0
  kmax = 100; tol = 1.e-04; % valores por defeito
elseif nargin == 1
  kmax = 100; % valor por defeito apenas para kmax
end
f(1) = 0; f(2) = 1; goldenold = 0;
for k = 3:kmax
   f(k) = f(k-1) + f(k-2);
   golden = f(k)/f(k-1);
   if abs(golden - goldenold) <= tol
     return
   end
   goldenold = golden;
end
return
```

nargin

A *função* `nargin` dá o número de parâmetros de entrada. Na nova versão da *função* `fibonacci` pode-se fixar o número máximo de iterações internas permitidas (`kmax`) e especificar uma tolerância `tol`. Se esta informação faltar, a função tomará valores por defeito (no nosso caso, `kmax = 100` e `tol = 1.e-04`). A título de exemplo, consideremos:

```
» [alpha,niter]=fibonacci1(1.e-6,200)

  alpha =
        1.61803381340013
  niter =
        19
```

Note-se que usando uma tolerância mais restritiva, obtivémos um novo valor aproximado em que os 8 primeiros algarismos significativos coincidem com os de α.

Pode-se usar `nargin` no exterior da função para obter o número dos seus parâmetros de entrada. Por exemplo:

```
» nargin('fibonacci1')

    ans =
         2
```

Observação 1.3 (funções *inline*) O comando `inline`, cuja sintaxe mais simples é `g=inline(expr,arg1,arg2,...,argn)`, declara a função `g` que depende das cadeias `arg1,arg2,...,argn`. A cadeia `expr` contém a expressão de `g`. Por exemplo, `g=inline('sin(r)','r')` declara a função $g(r) = \sin(r)$. O comando abreviado `g=inline(expr)` supõe implicitamente que `expr` é uma função da variável por defeito `x`. Logo que uma função *inline* tenha sido declarada, poderemos calculá-la em qualquer conjunto de variáveis através do comando `feval`. Por exemplo, para calcular `g` nos pontos `z=[0 1]` podemos escrever

inline

```
» feval('g',z);
```

Note-se que, contrariamente a `eval`, o comando `feval` não exige que o nome da variável (`z`) coincida com o nome simbólico (`r`) que figura no comando `inline`. •

No seguimento desta rápida introdução, sugerimos que o leitor explore o MATLAB usando o comando *help*, e se familiarize com a implementação de vários algoritmos usando os programas descritos neste livro. Por exemplo, digitando `help for` obtemos não só uma descrição completa do comando `for` mas também uma indicação sobre instruções semelhantes a `for`, tais como `if`, `while`, `switch`, `break` e `end`. Efectuando de novo um *help* para cada uma destas funções, melhora-se progressivamente o conhecimento da linguagem MATLAB.

Octave 1.7 De um modo geral, é nas aplicações gráficas que MATLAB e Octave têm mais diferenças. Verificámos que a maior parte dos comandos gráficos que aparecem no livro podem ser usados nos dois programas, mas possuem na realidade diferenças fundamentais. Por defeito, Octave utiliza GNUPlot para os gráficos; os comandos de desenho são diferentes e não funcionam como em MATLAB. No momento em que escrevemos esta secção existem outras bibliotecas gráficas em Octave, tais como `octaviz` (ver, o sítio web `http://octaviz.sourceforge.net/`), `epstk` (`http://www.epstk.de/`) e `octplot` (`http://octplot.sourceforge.net`). Esta última tenta reproduzir os comandos gráficos de MATLAB em Octave. ■

Ver os Exercícios 1.9-1.14.

1.7.3 Exemplos de diferenças entre as linguagens MATLAB e Octave

Tal como já referimos, o que se disse sobre a linguagem MATLAB na secção precedente aplica-se tanto nos ambientes MATLAB como Octave. Existem no entanto algumas diferenças entre estas duas linguagens. Programas escritos em Octave podem não ser executáveis em MATLAB e vice versa. Por exemplo, Octave suporta cadeias de caracteres com aspas simples e duplas

```
octave:1> a="Benvindo a Lisboa"

  a = Benvindo a Lisboa

octave:2> a='Benvindo a Lisboa'

  a = Benvindo a Lisboa
```

enquanto que MATLAB suporta apenas aspas simples (as aspas duplas dão um erro de sintaxe).

Incluímos aqui uma lista de algumas incompatibilidades entre as duas linguagens:

- MATLAB não permite um espaço antes do operador de transposição. Por exemplo, `[0 1]'` está correcto em MATLAB, mas `[0 1] '` não está. Octave trata correctamente os dois casos;
- MATLAB precisa sempre de `...`, para as linhas muito longas,

  ```
  rand (1, ...
        2)
  ```

  enquanto que se podem usar as notações

  ```
  rand (1,
        2)
  ```

  e

  ```
  rand (1, \
        2)
  ```

  em Octave para além de `...`;
- para a potência, Octave pode usar `^` ou `**`; MATLAB requer `^`;
- para terminar um bloco, Octave pode usar `end` e também `endif`, `endfor`, `...`; MATLAB requer apenas `end`.

1.8 O que não vos foi dito

Uma apresentação sistemática dos números de vírgula flutuante pode encontrar-se em [Übe97], [Hig02] e em [QSS07].

No que se refere a problemas de complexidade, referimos por exemplo [Pan92].

Para uma introdução mais sistemática ao MATLAB o leitor interessado pode consultar o manual de MATLAB [HH05] bem como livros mais específicos, incluindo [HLR01], [Pra02], [EKM05], [Pal04] ou [MH03].

Para Octave recomendamos o manual indicado no início deste capítulo.

1.9 Exercícios

Exercício 1.1 Quantos números pertencem ao conjunto $\mathbb{F}(2, 2, -2, 2)$? Qual é o valor de ϵ_M para este conjunto?

Exercício 1.2 Mostrar que o conjunto $\mathbb{F}(\beta, t, L, U)$ contém precisamente $2(\beta - 1)\beta^{t-1}(U - L + 1)$ elementos.

Exercício 1.3 Provar que i^i é um número real, e verificar em seguida este resultado usando MATLAB ou Octave.

Exercício 1.4 Escrever as instruções de MATLAB para construir uma matriz triangular superior (respectivamente, inferior) de dimensão 10, tendo 2 na diagonal principal e -3 na diagonal superior (respectivamente, inferior).

Exercício 1.5 Escrever em MATLAB as instruções que permitem efectuar a troca entre a terceira e a sétima linhas das matrizes construídas no Exercício 1.3, e em seguida as instruções que permitem a troca entre a quarta e a oitava colunas.

Exercício 1.6 Verificar se os seguintes vectores de $\mathbb{R}^4$ são linearmente independentes:

$$\mathbf{v}_1 = [0\ 1\ 0\ 1],\ \mathbf{v}_2 = [1\ 2\ 3\ 4],\ \mathbf{v}_3 = [1\ 0\ 1\ 0],\ \mathbf{v}_4 = [0\ 0\ 1\ 1].$$

Exercício 1.7 Escrever as funções seguintes e calcular as suas primeiras e segundas derivadas, bem como as suas primitivas, usando a *toolbox* `symbolic` de MATLAB:

$$f(x) = \sqrt{x^2 + 1}, \qquad g(x) = \sin(x^3) + \cosh(x).$$

Exercício 1.8 Para um dado vector `v` de dimensão n, construir com o comando `c= poly(v)` os $n+1$ coeficientes do polinómio $p(x) = \sum_{k=1}^{n+1} \mathtt{c}(k)x^{n+1-k}$ que é igual a $\Pi_{k=1}^{n}(x - \mathtt{v}(k))$. Em aritmética exacta, deveria obter-se `v = roots(poly(c))`. No entanto, isto não acontece devido a erros de arredondamento. Verificar este resultado, usando o comando `roots(poly([1:n]))`, onde `n` varia entre 2 e 25.

poly

Exercício 1.9 Escrever um programa para calcular a seguinte sucessão:

$$I_0 = \frac{1}{e}(e-1),$$
$$I_{n+1} = 1 - (n+1)I_n, \text{ for } n = 0, 1, \ldots.$$

Comparar o resultado numérico com o limite exacto $I_n \to 0$ quando $n \to \infty$.

Exercício 1.10 Explicar o comportamento da sucessão (1.4) quando calculada com MATLAB.

Exercício 1.11 Considerar o seguinte algoritmo para calcular π. Gerar n pares $\{(x_k, y_k)\}$ de números aleatórios no intervalo $[0, 1]$ e calcular em seguida o número m dos que se encontram no primeiro quadrante do círculo unitário. Naturalmente, π é o limite da sucessão $\pi_n = 4m/n$. Escrever um programa em MATLAB para calcular esta sucessão e observar a evolução do erro para valores crescentes de n.

Exercício 1.12 Sendo π a soma da série

$$\pi = \sum_{m=0}^{\infty} 16^{-m} \left(\frac{4}{8m+1} - \frac{2}{8m+4} + \frac{1}{8m+5} + \frac{1}{8m+6} \right),$$

podemos calcular uma aproximação de π somando os n primeiros termos, para n suficientemente grande. Escrever uma *função* de MATLAB para calcular as somas parciais desta série. Para que valores de n é que se obtém uma aproximação de π com a mesma precisão da variável π?

Exercício 1.13 Escrever um programa para calcular os coeficientes binomiais $\binom{n}{k} = n!/(k!(n-k)!)$, onde n e k são dois números naturais com $k \leq n$.

Exercício 1.14 Escrever em MATLAB uma *função* recursiva que calcule o n-ésimo elemento f_n da sucessão de Fibonacci. Sabendo que

$$\begin{bmatrix} f_i \\ f_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{i-1} \\ f_{i-2} \end{bmatrix} \tag{1.14}$$

escrever uma outra *função* que calcule f_n com base nesta forma recursiva. Finalmente, calcular o tempo de CPU correspondente.

http://www.springer.com/978-88-470-0717-8

CÁLCULO CIENTÍFICO com MATLAB e Octave
Quarteroni, A.; Saleri, F.
2007, XII, 320 p., Softcover
ISBN: 978-88-470-0717-8