• Note bem: este módulo foi inicialmente concebido para uma cadeira do 3º semestre, pelo que, os alunos que nunca tiveram contacto com um ambiente de programação terão uma natural dificuldade em alguns dos temas focados.
.

Módulo

0

Introdução ao Matlab

TÓPICOS
Ambiente de Trabalho
Representação, Declaração e Afectação
Representação Gráfica
Complexos
Controlo de Fluxo
Alocação de Memória. Vectorização
Scripts e Funções
Cálculo Simbólico

Em paralelo com ITI, esta é a primeira de um grande conjunto de cadeiras em que, ao longo do seu curso, vai recorrer ao ambiente de trabalho e à linguagem *Matlab* para resolver problemas e ilustrar conceitos do âmbito das matérias específicas de cada uma das cadeiras. É essencial que desenvolva uma base sólida no domínio da linguagem *Matlab*, e este é o momento em que deve tomar, seriamente, a decisão de construir essa base.

Este 1º módulo apenas pretende fazer uma breve introdução ao ambiente de trabalho *Matlab*, de modo a que, em pouco tempo, consiga dominar as funcionalidades básicas do ambiente e da linguagem.

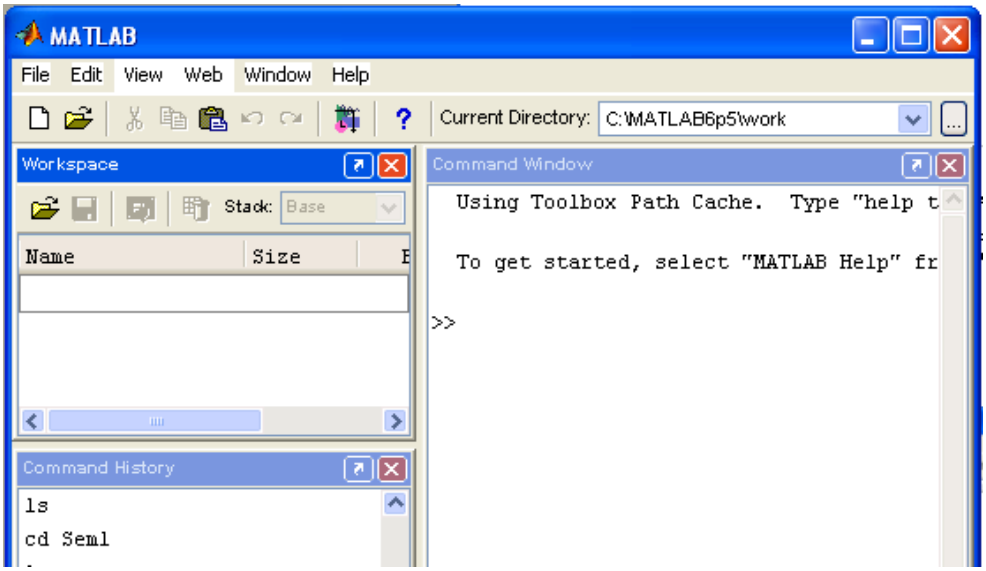
Optou-se pela não enumeração exaustiva de todas as funcionalidades do *Matlab*, que se considera enfadonha e inútil sem que se apresentem exemplos específicos de aplicação, justificativos do conhecimento dessas mesmas funcionalidades. Ao longo dos módulos seguintes, e a título exemplificativo dos conceitos teóricos que então se desejem esclarecer, serão propostos exercícios com recurso ao *Matlab*, sendo progressivamente apresentadas as potencialidades do ambiente de programação, as funções de base, e, especificamente, as pertencentes a *toolboxes*, que constituem um vasto conjunto de funções que permitem resolver rápida e eficientemente problemas de grande complexidade em diversas áreas de interesse, como sejam as do processamento de sinal, processamento de imagem, desenho de filtros, comunicações, controlo, etc.

A leitura deste módulo, como aliás a de todos os seguintes, deverá ser feita na presença de um PC com o *Matlab* instalado, de modo a que possa reproduzir os exemplos dados e fazer os exercícios propostos.

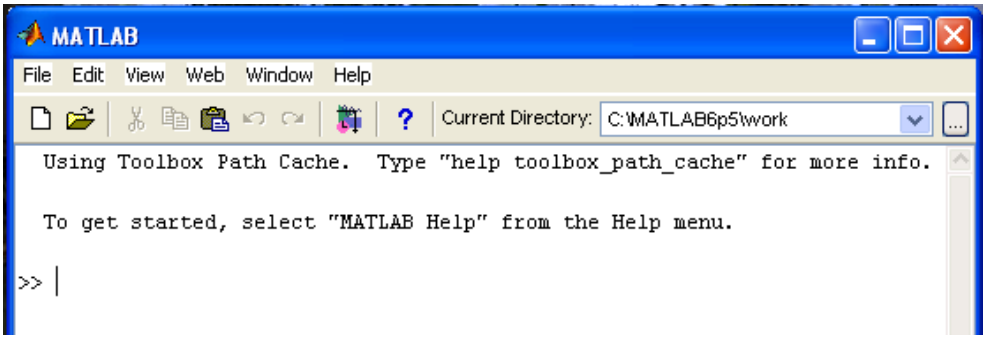
1. Ambiente de trabalho

Consola

Abra o *Matlab*. Dentro da janela *Matlab* é provável que existam diversas subjanelas (dependendo das pré-definições do seu ambiente de trabalho), como, por exemplo, as sub janelas *Workspace*, *Command History*, e *Command Window*, tendo o ambiente de trabalho o aspecto típico representado na figura seguinte.



Feche todas as janelas existentes para além da **consola** (*Command Window*). Por enquanto a consola é a única janela de que vai necessitar. O ambiente de trabalho deve ficar com o aspecto representado na figura seguinte.



Mude para a sua directoria de trabalho. Pode faze-lo quer através de uma operação de *browsing* no menu *Current Directory*, quer através de um comando na consola. Por exemplo

```
>> cd d:\work
```

Note que o *Matlab* aceita a maioria dos comandos básicos DOS e UNIX (por exemplo: `cd ; dir; ls; pwd;` etc.). Na maioria dos casos é mais simples recorrer aos comandos de consola do que a operações de rato sobre icons. Se quizer saber a directoria em que se encontra use o comando **pwd**. Se quizer listar os ficheiros existentes nessa directoria use o comando **ls**.

Operações aritméticas simples

Pode utilizar a consola para executar operações aritméticas simples, ou seja, como uma vulgar máquina de calcular. Por exemplo uma operação de adição

```
>> 2+3
ans =
     5
```

, ou de multiplicação

```
>>3*2
ans =
     6
```

As operações respeitam as regras de precedência comuns. Sempre que necessário ou sempre que tenha dúvidas utilize parênteses

```
>> 5*2+3*4/(4-5)
ans =
    -2
```

Reposição de comandos

Pressione o cursor ↑ e verifique que o *Matlab* repõe a última linha interpretada pela consola. Pressionar o cursor ↑ sucessivamente e verifique que todos os comandos interpretados anteriormente são repostos ordenadamente.

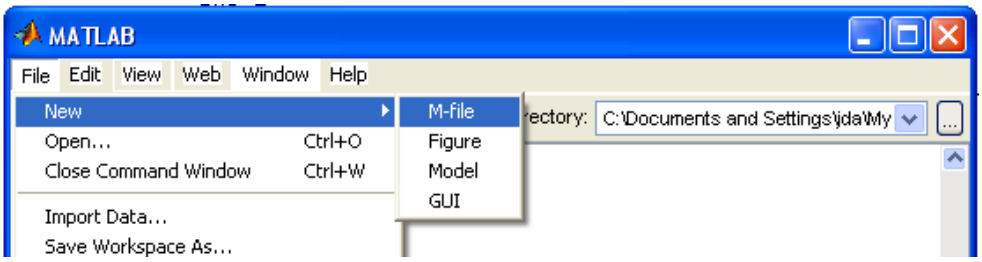
Se desejar repor um qualquer comando anteriormente interpretado, digite o(s) primeiro(s) carácter(es) desse comando e pressione seguidamente o cursor ↑. Neste caso, a pressão sucessiva do cursor ↑ apenas repõe os comandos anteriormente interpretados que comecem pelo(s) carácter(es) que digitou.

Ficheiros .m

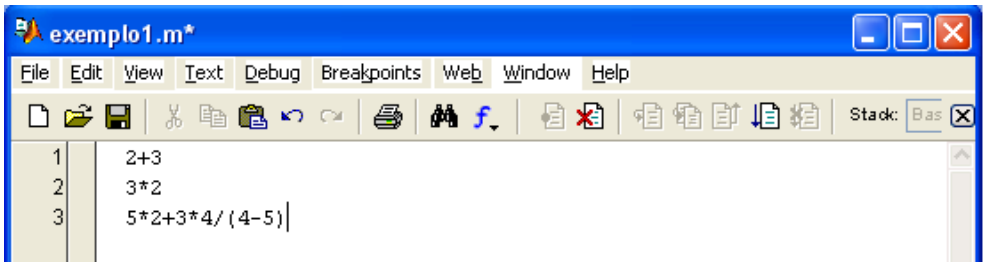
Sempre que seja necessário executar diversas linhas de código, deve habituar-se a criar um **ficheiro .m**. Execute o comando

```
>> edit exemplo1.m
```

, ou recorra ao menu **File→New→M-file**, conforme descrito na figura seguinte, para abrir a janela do editor associado ao *Matlab*.



Reescreva os comandos anteriormente escritos na janela da consola, salve o ficheiro com o nome `exemplo1.m` , e, na linha de comando, escreva o nome do ficheiro (sem a extensão)

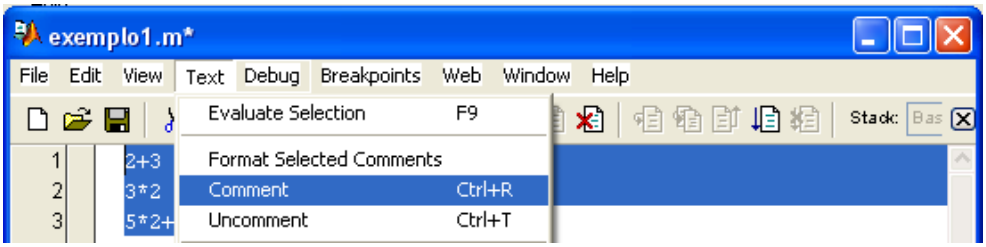



```
>> exemplo1
ans =
     5
ans =
     6
ans =
    -2
```

Como pode verificar, todas as linhas de código são **interpretadas** sucessivamente como se fossem apresentadas na linha de comando da consola.

Para além de assim poder guardar todos os comandos que executa sobre a consola, o editor permite-lhe, como é vulgar nos outros ambientes de programação, identar o código para uma mais fácil leitura, proceder ao *debuging*, etc..

Sempre que desejar pode inserir comentários, ou passar a comentário as linhas de código que não deseja executar, utilizando para isso o carácter **%**. O modo mais fácil de comentar diversas linhas de código é através do menu **Text→Comment**, conforme a figura seguinte

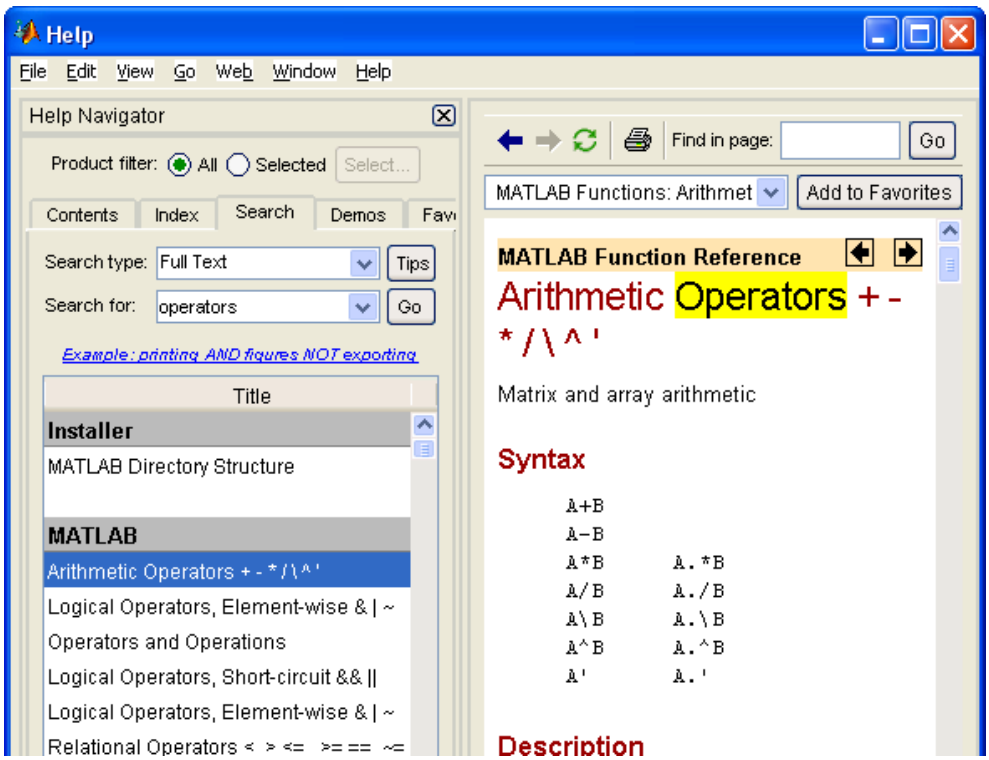


Help

Sempre que necessitar de ajuda sobre qualquer das funcionalidades do *Matlab* pode recorrer aos comandos **lookfor** e **help**. Por exemplo, para obter informação sobre os operadores existentes em *Matlab* pode fazer

```
>> lookfor operators
arith.m: %Arithmetic operators.
relop.m: %Relational operators.
>> [Ctrl+C]
>> help arith
 Arithmetic operators.
  +   Plus.
         X + Y adds matrices X and Y.  X and Y must have
         the same dimensions unless one is a scalar  ...
```

Se desejar informação mais detalhada pode recorrer ao menu **Help→MATLAB Help**, abrindo assim a janela *Help* do *Matlab*, e fazer uma busca *full text* sobre *operators*



2. Representação, declaração e afectação

Representação numérica

O *Matlab* armazena as quantidades numéricas e opera sobre estas utilizando **precisão dupla**, ou seja, são utilizados 64 bits para a representação interna, sendo 1 bit para o sinal, 11 bits para o expoente, e 52 bits para a mantissa. Contrariamente a outras linguagens de programação não existe a necessidade de declarar o tipo de representação interna com que se deseja trabalhar.

Não confunda a representação interna, o "verdadeiro" valor da quantidade numérica que está a operar, com a representação externa, isto é, com o valor da quantidade numérica que é mostrado na consola. O valor mostrado na consola, ou, mais exactamente, o formato com que a quantidade numérica é apresentada na consola, é controlado por si através do comando **format**. Execute

```
>> format short
>> 3/7
ans =
    0.4286
>> format long
>> 3/7
ans =
     0.42857142857143
>> format short e
>> 3/7
ans =
     4.2857e-001
```

Execute o comando `help format` para um completo esclarecimento sobre todos os formatos disponíveis.

Pode utilizar várias notações para enumerar as quantidades numéricas. Por exemplo, a quantidade 0.04 pode ser enunciada de diversos modos

```
>> 0.04;
>> 4*10^-2;
>> 4e-2
ans =
    0.0400
```

Variáveis. Declaração e afectação

Como se disse, em *Matlab*, não existe a necessidade de declarar o tipo de uma variável antes da sua utilização. A operação de declaração e afectação é feita simultaneamente através do operador **=**. Por exemplo, podemos declarar a variável `a` e atribuir-lhes o valor inteiro 2

```
>> a=2
a =
     2
```

Note que pode evitar que seja feito eco na consola do resultado das linhas de código terminando cada uma delas com o operador **;**

```
>> a=2;
```

Podemos de seguida declarar a variável `b` e atribuir-lhe o valor real 3.14

```
>> b=3.14;
```

No nome das variáveis, o *Matlab* distingue as letras maiúsculas das minúsculas, só atendendo aos primeiros 31 caracteres.

Podemos verificar que as variáveis `a` e `b` passaram a existir no espaço de trabalho através do comando **who**

```
>> who
Your variables are:
a  b
```

Se utilizarmos o comando **whos**, para além do nome das variáveis é-nos dada informação sobre a dimensão (*size*) das variáveis, espaço ocupado em memória, etc.

```
>> whos
  Name      Size                    Bytes  Class
  a         1x1                         8  double array
  b         1x1                         8  double array
```

Note que qualquer das variáveis é representada internamente como um valor real em formato *double* (8 bytes =8x8bits=64bits). Note ainda que qualquer delas é referida como tendo dimensão 1x1. O que isto significa é que qualquer delas é considerada como sendo uma matriz, no caso uma matriz com uma linha e uma coluna. No nome *Matlab*, "Mat" não é, como poderia pensar, a abreviatura de *Mathematic*, mas sim de *Matrix*. Na verdade as matrizes formam o núcleo do *Matlab* sendo todos os dados considerados como tal (por exemplo: um escalar é uma matriz 1x1; um vector é uma matriz linha ou uma matriz coluna).

Se quiser eliminar uma variável do espaço de trabalho utilize o comando **clear <variável>**, Se quiser eliminar todas as variáveis utilize o comando **clear all**.

Vectores

Já vimos como definir uma variável escalar. Vamos agora ver como definir um vector. O modo mais óbvio é por enumeração das suas componentes, por exemplo

```
>> v=[0 1 2 3]
v =

     0     1     2     3
```

define um vector linha com 4 elementos, na verdade, não esqueça, define uma matriz com 1 linha e 4 colunas. Podemos verificar tal facto com a função **size**

```
>> size(v)
ans =
     1     4
```

Como vê, a variável `v` é considerada como sendo uma matriz com 1 linha e 4 colunas. Podemos ter necessidade de definir um vector coluna. Nesse caso devemos enumerar os elementos do vector separados pelo operador **;**

```
>> v=[0; 1; 2; 3]
v =
     0
     1
     2
     3
```

Podemos verificar que temos agora uma matriz com 4 linhas e 1 coluna

```
>> size(v)
ans =
     4     1
```

Se o vector a definir tiver uma qualquer relação entre os seus elementos, é possível defini-lo de modo mais prático do que por enumeração, recorrendo ao operador **:**. Por exemplo, para definir um vector de elementos equiespaçados de 1 entre o **valor inicial** -4 e o **valor final** 3 basta fazer

```
>> v=-4:3
v =
    -4    -3    -2    -1     0     1     2     3
```

Se o equiespaçamento não for unitário basta referi-lo. Por exemplo, para definir um vector de elementos equiespaçados de 0.1 entre o valor inicial 4.3 e o valor final 4.6 basta fazer

```
>> v=4.3:0.1:4.6
v =
    4.3000    4.4000    4.5000    4.6000
```

Sempre que seja necessário transformar um vector linha num vector coluna, e vice-versa, o modo mais eficiente de o fazer é recorrer à **transposição**, utilizando o operador **'** , por exemplo

```
>> v=v'
v =
    4.3000
    4.4000
    4.5000
    4.6000
```

Note que no caso em que os elementos do vector sejam quantidades complexas a transformação de um vector linha num vector coluna não é equivalente à operação de transposição. Recorde da álgebra que existe neste caso uma operação de conjugação dos elementos. Assim

```
>> a=[1-2j 3-4j]
a =
   1.0000 - 2.0000i   3.0000 - 4.0000i
>> a'
ans =
   1.0000 + 2.0000i
   3.0000 + 4.0000i
>> conj(a')
ans =
   1.0000 - 2.0000i
   3.0000 - 4.0000i
```

Existem métodos mais sofisticados de definir vectores, e vectores especiais, de que falaremos mais tarde, quando for necessária a sua utilização.

Matrizes

Uma matriz é igualmente simples de definir por enumeração. Por exemplo

```
>> A=[1 2 3 4; 5 6 7 8; 9 10 11 12]
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
>> size(A)
ans =
     3     4
```

, define, como vê, uma matriz com 3 linhas e 4 colunas, composta pelos elementos enumerados, devendo cada uma das linhas ser separada pelo operador **;**. Poderíamos ter feito

```
>> A=[1:4;5:8;9:12]
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

O elemento da linha i e da coluna j de uma matriz A é designado por $A(i,j)$, sendo os índices **inteiros positivos**. Por exemplo, para seleccionar o elemento da linha 2 e coluna 3 da matriz definida anteriormente fazemos

```
>> A(2,3)
ans =
     7
```

Se pretendermos alterar este valor, por exemplo para 0, basta fazer

```
>> A(2,3)=0
A =
     1     2     3     4
     5     6     0     8
     9    10    11    12
```

Para **seleccionar uma linha ou uma coluna completa** recorremos ao operador **:**. Por exemplo

```
>> A(2,:)
ans =
     5     6     0     8
>> A(:,3)
ans =
     3
     0
    11
```

É igualmente fácil seleccionar uma **submatriz**. Por exemplo, se quisermos seleccionar os elementos das linhas 2 e 3, colunas 2 a 4 podemos fazer

```
>> A([2 3],[2 3 4])
ans =
     6     0     8
    10    11    12
```

, ou, simplesmente

```
>> A(2:3,2:4)
ans =
     6     0     8
    10    11    12
```

Os elementos de uma matriz podem ainda ser seleccionados sequencialmente, sendo a enumeração feita ao longo das colunas, da direita para a esquerda. Por exemplo

```
>> A(2:5)
ans =
     5     9     2     6
>>
```

Remoção de linhas e colunas

Para remover qualquer conjunto de linhas ou colunas de uma dada matriz, basta atribuir uma matriz vazia às linhas e colunas que se pretende remover. Por exemplo, recordando a matriz *A*

```
>> A=[1:4;5:8;9:12]
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
>>
```

podemos remover facilmente a 2ª coluna da matriz

```
>> A(:,2)=[]
A =
     1     3     4
     5     7     8
     9    11    12
>>
```

, ou a 3ª linha

```
>> A(3,:)=[]
A =
     1     2     3     4
     5     6     7     8
>>
```

Concatenação

A operação de juntar várias matrizes, ou vectores, numa só matriz é designada por **concatenação**. Por exemplo

```
>> A=[1 2;3 4]
A =
     1     2
     3     4
>> B=[5 6;7 8]
B =
     5     6
     7     8
>> C=[A B]
C =
     1     2     5     6
     3     4     7     8
>> C=[A; B]
C =
     1     2
     3     4
     5     6
     7     8
```

Matrizes e vectores especiais

Uma matriz de zeros é fácil de criar em Matlab, recorrendo à **função zeros**

```
>> B=zeros(2,3)
B =

     0     0     0
     0     0     0
>>
```

Tem especial interesse, a curto prazo, a criação de um vector linha, ou coluna, contendo apenas zeros

```
>> V1=zeros(1,3)
V1 =

     0     0     0
>> V2=zeros(3,1)
V2 =
     0
     0
     0
>>
```

Note que quando se pretende criar um vector é necessário especificar duas dimensões, uma delas, evidentemente, igual a 1. Um erro comum é a especificação de apenas uma dimensão. Note que nesse caso é criada uma matriz quadrada, da dimensão especificada, e não um vector

```
>> zeros(4)
ans =
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
>>
```

De igual interesse é a criação de uma matriz de uns. Recorremos para isso à **função ones**

```
>> C=ones(3,2)
C =
     1     1
     1     1
     1     1
>>
```

Para um vector linha, e coluna, temos, evidentemente

```
>> V1=ones(1,3)
V1 =

     1     1     1
>> V2=ones(3,1)
V2 =
```

```
     1
     1
     1
>>
```

Vimos que é fácil criar um **vector com um valor inicial, um incremento, e um valor final**

```
>> v=0:0.5:3
v =
         0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
>>
```

E se a intenção for criar um **vector com um valor inicial, um valor final, e um dado número de elementos**? Por exemplo, valor inicial 0, valor final 3 e 6 elementos. Bom, podemos pensar que se trata de um problema de dividir o espaço num número de intervalos, ou seja de ter um incremento igual à dimensão do intervalo, a dividir pelo número de elementos menos 1, e por isso

```
>> v=0:(3-0)/5:3
v =
         0    0.6000    1.2000    1.8000    2.4000    3.0000
>>
```

Alternativamente, podemos recorrer à **função linspace**

```
>> v=linspace(0,3,6)
v =
         0    0.6000    1.2000    1.8000    2.4000    3.0000
>>
```

Operadores aritméticos

Uma consequência directa das entidades numéricas serem sempre consideradas matrizes é o facto de os operadores serem por defeito considerados operadores matriciais. Assim, os **operadores +**, **-**, *****, **/** e **^**, são, não esqueça, **operadores matriciais**.

Operação	Operador
Adição	**+**
Subtracção	**-**
Multiplicação	*****
Divisão	**/**
Potenciação	**^**

Estes operadores podem ser usados com quantidades escalares dado que não existe qualquer problema em manipular matrizes 1×1, no entanto, quando aplicados a vectores ou matrizes é evidentemente necessário respeitar as regras do cálculo matricial. Por exemplo, é muito comum que, quando se pretende elevar os diversos elementos de um vector `v` ao quadrado se escreva

```
>> v^2
??? Error using ==> ^
Matrix must be square.
>>
```

,ou que, quando se pretende multiplicar dois vectores elemento a elemento se escreva

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

Elevar os diversos elementos de um vector a uma potência e multiplicar dois vectores elemento a elemento, são dois exemplos de **operações elemento a elemento (ponto a ponto)**, devendo ser utilizados os **operadores .*, ./ e .^** . No caso da adição e da subtracção não existe qualquer diferença entre os operadores.

Operação	Operador
Multiplicação ponto a ponto	.*
Divisão ponto a ponto	./
Potência ponto a ponto	.^

Por exemplo, podemos definir dois vectores

```
>> v=[2 3 4];
>> w=[5 6 7];
>>
```

E proceder a uma operação matricial de multiplicação, ou divisão

```
>> v*w'
ans =
    56
>> w/v
ans =
    1.9310
>>
```

que será executada sem erro se estiverem verificadas as condições para que a operação matricial seja efectuada (note que no caso da multiplicação foi necessário transpor um dos vectores). Caso as operações desejadas fossem, por exemplo, a multiplicação ou divisão dos dois vectores ponto a ponto, deveríamos fazer

```
>> v.*w
ans =
    10    18    28
>> w./v
ans =
    2.5000    2.0000    1.7500
>>
```

A confusão inadvertida entre os dois tipos de operação é muito comum. Tome sempre isto em muita atenção na escrita do código. Note que por vezes a confusão entre as operações matricial e ponto a ponto pode não ser detectada, por estar imersa numa sequência de cálculos. Considere a matriz

```
>> A=[ 2 3; 4 5]
A =
     2     3
     4     5
>>
```

Note que

```
>> A^2
ans =
    16    21
    28    37
>>
```

é uma operação matricial possível, devido ao facto da matriz ser quadrada, que nada tem a ver com operação ponto a ponto

```
>> A.^2
ans =
     4     9
    16    25

>>
```

Operadores relacionais

Para os operadores relacionais em *Matlab* são utilizados os símbolos

Igual	Diferente	Menor	Maior	Menor ou Igual	Maior ou Igual
==	~=	<	>	<=	>=

A operação de comparação devolve 1 se a relação se verificar e 0 se não se verificar. Por exemplo

```
>> a=2;
>> b=2;
>> c=(a==b)
c =
     1
>> c=(a~=b)
c =
     0
>>
```

Como foi referido, o *Matlab* armazena as quantidades numéricas utilizando precisão dupla, ou seja, utilizando uma mantissa com 52 bits. A precisão de representação é portanto finita. Sempre que (e trata-se da maior parte dos casos) a quantidade numérica não tenha representação exacta, em binário, por utilização de "apenas" 52 bits, é cometido um erro de representação. A menor quantidade representável em precisão dupla (2^{-52}) é designada em *Matlab* por **eps**

```
>> eps
ans =
  2.2204e-016
```

Chama-se aqui à atenção para esse facto, não pelas implicações que tem na precisão dos cálculos, nesta cadeira, mas porque aquele dá origem a um erro muito comum na utilização dos operadores relacionais. Considere por exemplo as variáveis

```
>> a=5/7.2
a =
    0.6944
>> b=5/(3.1+4.1)
b =
    0.6944
>>
```

Teoricamente a duas quantidades são iguais, no entanto, pode verificar que, em resultado do erro de representação, resulta

```
>> a-b
ans =
-1.1102e-016
>>
```

Um valor desprezável (*eps*/2), evidentemente, se não estiverem em causa decisões tomadas sobre a igualdade das quantidades ... E se, baseado no facto de as quantidades serem teoricamente iguais, e ignorando a existência de erros de representação, você tomasse uma decisão com base na igualdade

```
>> if (a==b)
        c=1;
   else
        c=2;
   end
c =
     2
>>
```

Como vê o resultado era catastrófico. Conclusão:

Nunca baseie decisões com base em critérios de igualdade entre quantidades numéricas reais

O que fazer então, bem ..., já que a quantidade *eps* está definida, podemos, por exemplo, fazer

```
>> if (a-b)<eps
     c=1;
   else
     c=2;
   end
c =
     1
>>
```

Uma das utilidades dos operadores relacionais é o da selecção eficiente de elementos de uma matriz, ou vector, e construção de novos vectores. Por exemplo, construindo o vector v com os valores absolutos dos inteiros entre -5 e 5, recorrendo à **função abs**

```
>> v=abs(-5:5)
v =
     5     4     3     2     1     0     1     2     3     4     5
>>
```

é fácil construir um vector v_2 com o valor 0 sempre que $v_i > 2$ e o valor 1 sempre que $v_i \leq 2$

```
>> v2=v<=2
v2 =
     0     0     0     1     1     1     1     1     0     0     0
>>
```

ou saber quais são os índices de v para os quais $v_i \geq 4$, recorrendo à **função find**

```
>> find(v>=4)
ans =
     1     2    10    11
>>
```

ou substituir os valores de v que verificam $v_i \geq 4$ por 0

```
>> v(find(v>=4))=0
v =
     0     0     3     2     1     0     1     2     3     0     0
>>
```

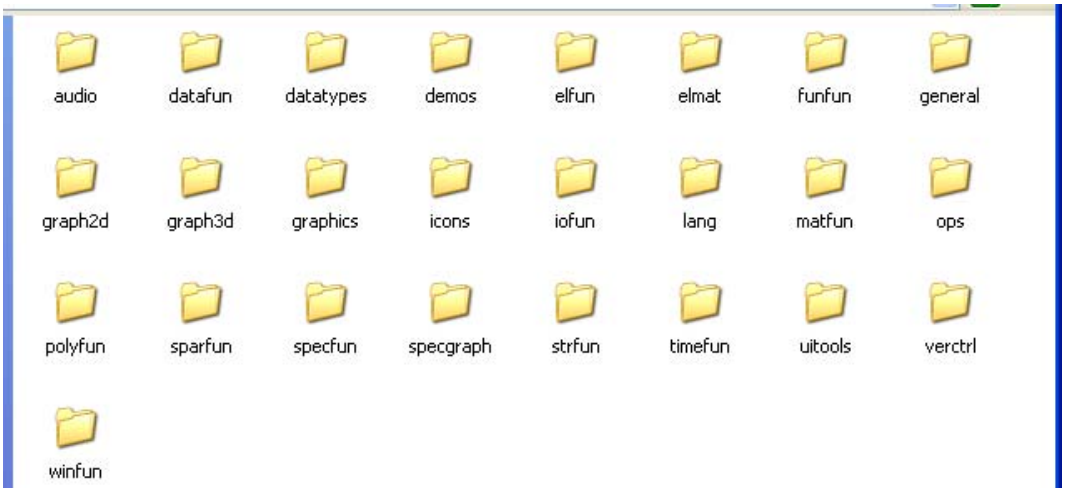
Operadores lógicos

Para os operadores lógicos em Matlab são utilizados os símbolos

AND	OR	NOT	XOR
&	**|**	**~**	**xor**

Funções base

Existe um número imenso de funções pré-definidas em *Matlab*, pelo que seria fastidioso estar aqui a enumerar e descrever a funcionalidade de cada uma delas. A estrutura de directórios em disco em que o *Matlab* está instalado, reflecte os diversos grupos de funções existentes. Para conhecer os diversos grupos de funções faça *browsing* do directório `...\toolbox\matlab\`.



Por exemplo, as funções elementares estão no directório `[C:\MATLAB]\toolbox\matlab\elfun` etc. Para conhecer o conjunto de **funções elementares**, ou, em qualquer momento, recordar a sintaxe de alguma delas, faça `help elfun`

```
>> help elfun
  Elementary math functions.
  Trigonometric.
    sin         - Sine.
    sinh        - Hyperbolic sine.
    asin        - Inverse sine.
    asinh       - Inverse hyperbolic sine.
    cos         - Cosine.
    cosh        - Hyperbolic cosine.
    acos        - Inverse cosine.
    acosh       - Inverse hyperbolic cosine.
    tan         - Tangent.
      ...
```

Para conhecer o conjunto de **funções de análise de dados**, ou, em qualquer momento, recordar a sintaxe de alguma delas, faça `help datafun`

```
>>help datafun
  Data analysis and Fourier transforms.
  Basic operations.
    max         - Largest component.
    min         - Smallest component.
    mean        - Average or mean value.
    median      - Median value.
    std         - Standard deviation.
    var         - Variance.
      ...
```

Para conhecer o conjunto de **funções de manipulação matricial**, ou, em qualquer momento, recordar a sintaxe de alguma delas, faça `help elmat`

```
>> >> help elmat
  Elementary matrices and matrix manipulation.
  Elementary matrices.
    zeros       - Zeros array.
    ones        - Ones array.
    eye         - Identity matrix.
    repmat      - Replicate and tile array.
    rand        - Uniformly distributed random numbers.
    randn       - Normally distributed random numbers.
    linspace    - Linearly spaced vector.
    logspace    - Logarithmically spaced vector.
      ...
```

Para conhecer o conjunto de **funções matriciais**, ou, em qualquer momento, recordar a sintaxe de alguma delas, faça `help matfun`

```
>> help matfun
  Matrix functions - numerical linear algebra.
  Matrix analysis.
    norm        - Matrix or vector norm.
    normest     - Estimate the matrix 2-norm.
    rank        - Matrix rank.
    det         - Determinant.
      ...
```

Para obter informação sobre qualquer das funções, já sabe, faça `help <func>`. Por exemplo

```
>> help linspace
LINSPACE Linearly spaced vector.
    LINSPACE(X1, X2) generates a row vector of 100 linearly
    equally spaced points between X1 and X2.
    LINSPACE(X1, X2, N) generates N points between X1 and X2.
    For N < 2, LINSPACE returns X2.
    See also LOGSPACE, :.
>>
```

.

3. Representação gráfica

Uma das grandes vantagens do ambiente *Matlab* é a existência, e fácil utilização, de funções de criação e manipulação de gráficos 2D e 3D.

Plot

Admita que pretende representar graficamente a função $f(x) = \cos(x)$ no intervalo $[0, 2\pi]$. A primeira coisa a fazer é criar um vector com valores da variável independente no intervalo pretendido. Vamos, por exemplo, criar 100 valores de x, equiespaçados, no intervalo $[0, 2\pi]$.

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

Seguidamente criamos um vector com o correspondente valor da função em cada um dos pontos

```
>> y=cos(x);
>>
```

Estamos agora na posse de tudo o que é necessário para representar a função: **um vector de valores da abcissa e um vector dos correspondentes valores da ordenada**. Para fazer a representação gráfica recorremos à **função plot**

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

Obtemos assim o gráfico que se mostra na figura M1.1.

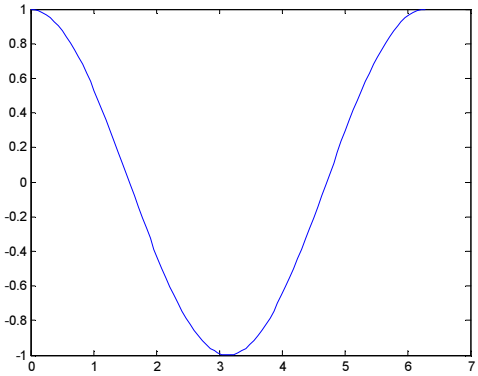


Figura M1.1

E se estivermos apenas na posse do valor da função a representar, desconhecendo os valores da variável independente que lhe deu origem? Nesse caso fazemos

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

, sendo a **representação feita em função dos índices do vector** (no caso presente de 1 a 100), como pode ver na figura M1.2.

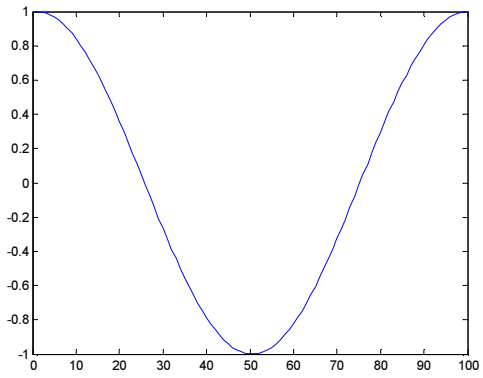


Figura M1.2

Admita agora que o valor da função é calculado em apenas 10 pontos do intervalo $[0, 2\pi]$.

```
>>> x=linspace(0,2*pi,10);
>> y=cos(x);
>> plot(x,y)
>>
```

Pode observar o gráfico resultante na figura M1.3. Não lhe parece um $\cos()$? O que poderá ter corrido mal? Bem, não esqueça que está a representar um conjunto de pares ordenados (x, y) que constituem, neste caso, 10 pontos em $\Re^2$. A função *plot* permite que se especifique um símbolo para os pontos a representar, assim como o tipo de traço que os une. Por exemplo, repitamos a representação anterior, mas especificando agora que cada um

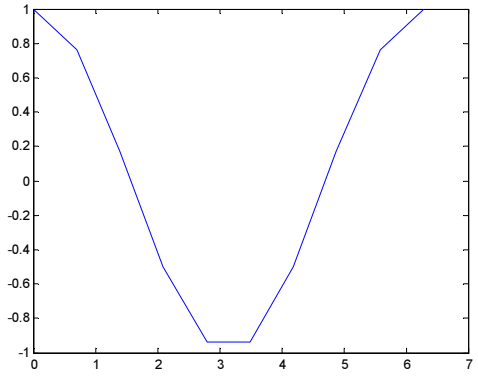


Figura M1.3

dos pontos no plano deve ser representado pelo símbolo o, e os pontos devem ser unidos por uma linha a tracejado.

```
>> plot(x,y,'o:')
>>
```

Obtemos assim o gráfico da figura M1.4. É agora fácil interpretar a figura M1.3.

Podemos optar por representar os pontos no plano sem qualquer linha de união. Optemos por representar 20 pontos da função, usando agora o símbolo *

```
>> x=linspace(0,2*pi,20);
>> y=cos(x);
>> plot(x,y,'*')
>>
```

Caso pretendêssemos unir os pontos por uma linha contínua faríamos

```
>> plot(x,y,'-*')
>>
```

Obtemos assim os gráficos das figuras M1.5 e M1.6.

Por defeito, a função plot não usa qualquer símbolo para representar os pontos no plano e une as sucessivas posições dos pares ordenados por segmentos em linha contínua. Nunca esqueça este facto.

A ilusão do contínuo

Se o número de pontos em que calcula a função a representar não for suficiente, pode resultar que a interpretação do gráfico resultante seja completamente errada. Não esqueça que, na maioria dos casos, você pode não saber qual é o comportamento da função a representar, e o que a representação gráfica pretende é exactamente ilustrar esse comportamento. Por exemplo, considere que o $\cos()$ a representar tem uma frequência muito mais elevada, digamos $20\times$, $g(x)=\cos(20x)$, façamos a sua representação no intervalo $[0,2\pi]$, mas usemos apenas 20 pontos, criando para isso um novo vector

```
>> x=linspace(0,2*pi,20);
>> y2=cos(20*x);
>> plot(x,y1,'-*')
>>
```

O gráfico resultante é o que se mostra na figura M1.7. Compare-o com figura M1.6. São iguais!? A função $y=\cos(x)$ e $y_2=\cos(20x)$ têm o mesmo comportamento no intervalo $[0,2\pi]$!?

Na verdade, o número de pontos escolhido para representar $g(x)$ é excessivamente

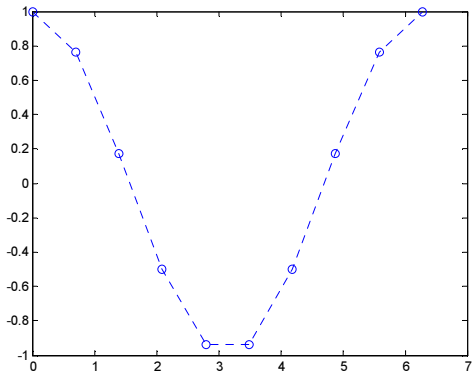


Figura M1.4

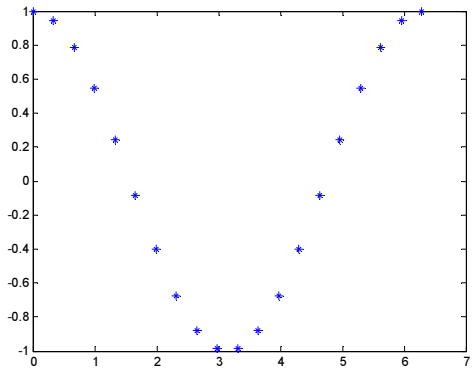


Figura M1.5

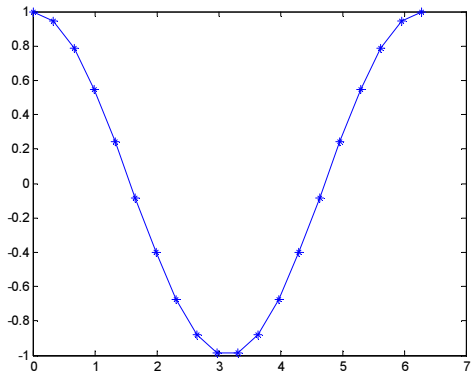


Figura M1.6

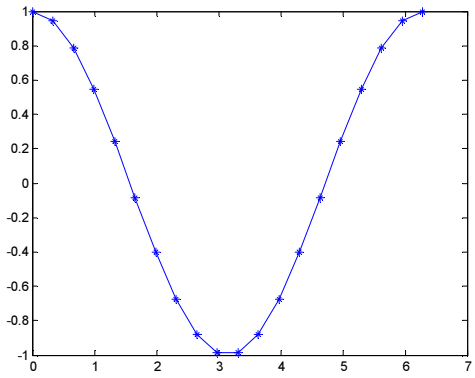


Figura M1.7

pequeno. Vamos agora representar $g(x)$ usando um número de pontos muito mais elevado, por exemplo 800, e vamos sobrepor o gráfico assim obtido ao gráfico da figura M1.7. Para sobrepor os gráficos usamos o comando **hold on**, este permite segurar o espaço gráfico em que estamos a trabalhar, de modo que a partir daí todas os gráficos serão executados sobre esse mesmo espaço. Se quisermos libertar de novo o espaço gráfico usamos o comando **hold off**. Para além disso, para melhor distinguir os dois gráficos, vamos executar o segundo a vermelho (**r***ed*).

```
>> x=linspace(0,2*pi,20);
>> y2=cos(20*x);
>> plot(x,y2,'-*')
>> hold on
>>
>> x=linspace(0,2*pi,800);
>> y3=cos(20*x);
>> plot(x,y3,'r')
>> hold off
>>
```

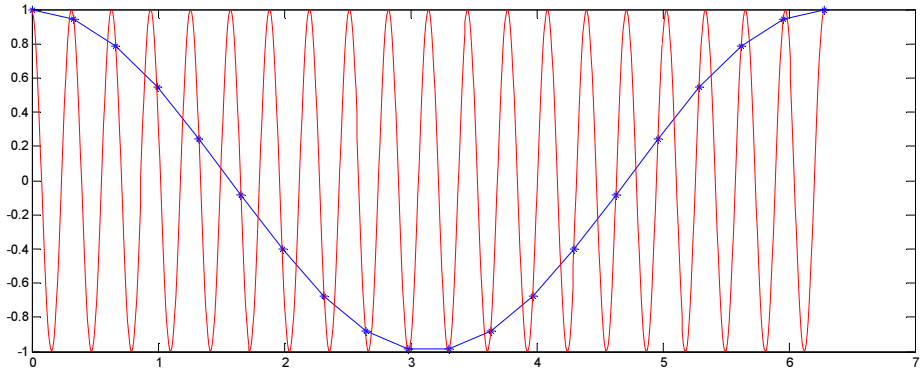


Figura M1.8

Pode ver o gráfico resultante na figura M1.8. Podemos ver agora o verdadeiro comportamento da função $g(x) = \cos(20x)$. Pode compreender agora a figura M1.7: como, por defeito, o *Matlab* une os sucessivos pontos do vector a representar, o gráfico resultante provoca a ilusão de se tratar de uma função $\cos()$ com uma frequência muito mais baixa, não tendo na verdade nada a ver com o verdadeiro comportamento da função que se deseja representar.

Argumentos de *plot* e funções associadas

Execute o comando `help plot` para um completo esclarecimento sobre todos os argumentos que a função *plot* pode receber, e siga as diversas funções *see also* para conhecer as funcionalidades das diversas funções que lhe permitem controlar o aspecto do espaço gráfico. Os exemplos seguintes procuram ser suficientemente esclarecedores sobre as principais opções. Procure interpretar a linhas de código observando as figuras e recorrendo ao *help*.

Exemplo 1

- Representação simultânea de várias funções.
- Controle da cor, do símbolo e do tipo de linha de representação.
- Inserção de grelha.
- Inserção de etiqueta em cada um dos eixos.
- Inserção de nome de figura..
- Inserção de legenda

```
x=linspace(0,2*pi,30);
y1=cos(x);
y2=sin(x);

plot(x,y1,':or',x,y2,'-*b');
```

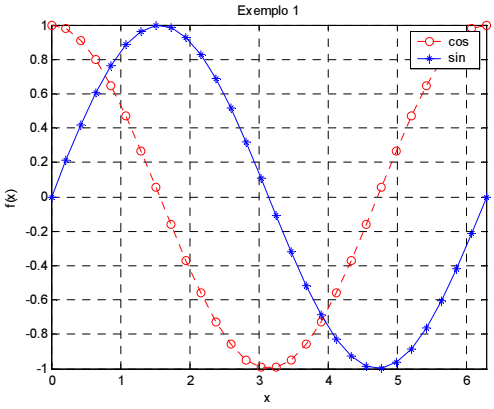


Figura M1.9

```
grid on;
xlabel('x');
ylabel('f(x)');
axis([0 2*pi -1 1]);
title('Exemplo 1');
legend('cos','sin');
```

Exemplo 2

- Representação simultânea de várias funções recorrendo a uma matriz.
- Eliminação da caixa de legendas.

```
x=[2:0.1:5];
Y=[x; x.^2; x.^3; exp(x)];
plot(x,Y,'-o');
legend('x','x^2','x^3','e^x',2);
legend boxoff
grid on
```

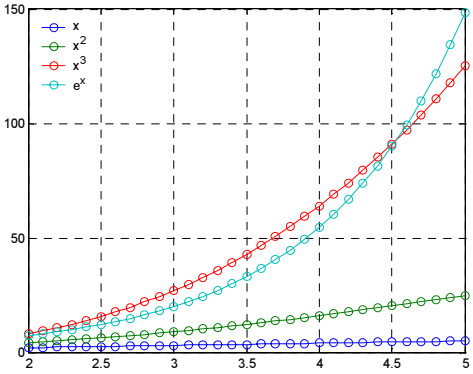


Figura M1.10

Exemplo 3

- Representação simultânea de várias funções recorrendo ao comando *hold on*.
- Inserção de texto no espaço gráfico.

```
w=0:0.01:6;
f1=cos(w).*exp(w/4)-1;
plot(w,f1,'LineWidth',2);
grid on;
hold on;
f2=cos(w);
plot(w,f2);
grid on;

text(1.3,0.5,'\leftarrow \fontname{Garamond}\fontsize{14}cos(\omega_i)');
text(0.6,-2.5,'\fontname{Garamond}\fontsize{14}cos(\omega_i) e^{ j
  \omega_i/4}  \rightarrow');
hold off
```

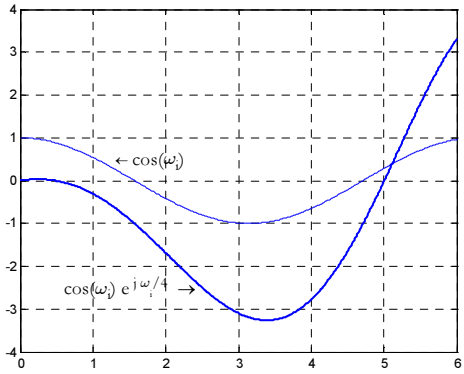


Figura M1.11

Subplot

Por vezes há conveniência em fazer a representação simultânea de várias funções na mesma janela gráfica sem fazer a sua sobreposição. Em *Matlab* isto pode ser feito através da **função subplot**. A função *subplot(m, n, i)* cria uma janela gráfica, define sobre essa janela uma matriz de $m \times n$ espaços gráficos e activa o elemento i da matriz. Seguidamente, sobre esse elemento i pode fazer o gráfico desejado através, por exemplo, da função *plot*. Dão-se seguidamente vários exemplos que se consideram auto-explicativos. Interprete as linhas de código por observação das figuras e recurso ao comando *help subplot*.

Exemplo 1

- Representação de funções em coluna.

```
x=0:0.01:6;
f1=exp(x/4)-1;
f2=cos(x);

subplot(2,1,1);
plot(x,f1);
grid on;

subplot(2,1,2);
plot(x,f2);
grid on;
```

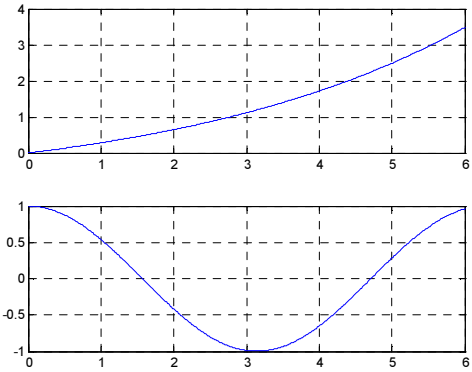


Figura M1.12

Exemplo 2

- Representação de funções em linha.

```
x=0:0.01:6;
f1=exp(x/4)-1;
f2=cos(x);

subplot(1,2,1);
plot(x,f1);
grid on;

subplot(1,2,2);
plot(x,f2);
grid on;
```

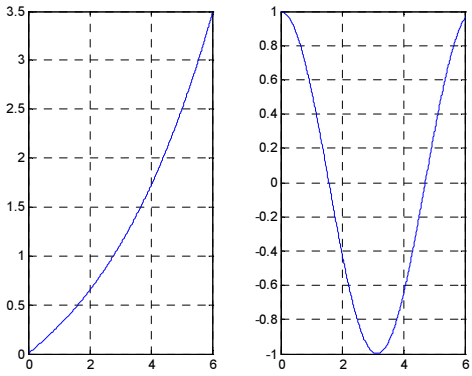


Figura M1.13

Exemplo 3

- Representação sobre uma matriz 2×2.

```
x=0:0.01:6;
f1=exp(x/4)-1;
f2=cos(x);
f3=sqrt(x);
f4=sin(x);
subplot(2,2,1);
plot(x,f1);
grid on;
subplot(2,2,2);
plot(x,f2);
grid on;
subplot(2,2,3);
plot(x,f3);
grid on;
subplot(2,2,4);
plot(x,f4);
grid on;
```

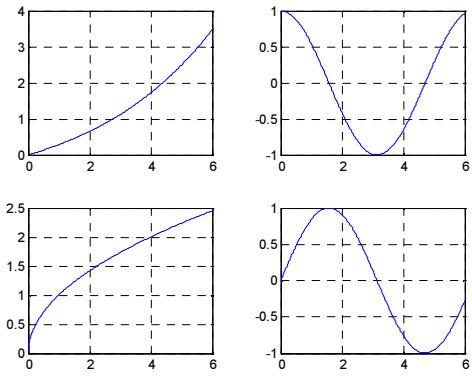


Figura M1.14

Exemplo 4

- Variantes

```
x=0:0.01:6;
f1=exp(x/4)-1;
f2=cos(x);
f3=sin(x);
subplot(2,2,1);
plot(x,f1);
grid on;
subplot(2,2,2);
plot(x,f2);
grid on;
subplot(2,1,2);
plot(x,f3);
grid on;
```

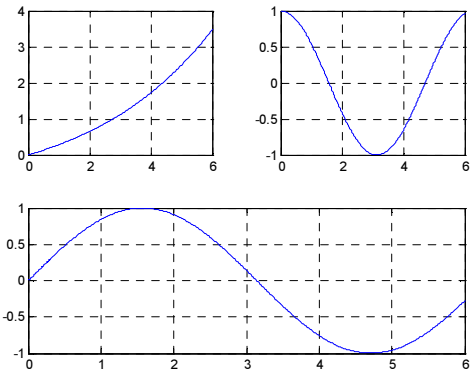


Figura M1.15

Exemplo 5

- Variantes

```
x=0:0.01:6;
f1=exp(x/4)-1;
f2=cos(x);
f3=sin(x);
subplot(2,2,1);
plot(x,f1);
grid on;
subplot(2,2,3);
plot(x,f2);
grid on;
subplot(1,2,2);
plot(x,f3);
grid on;
```

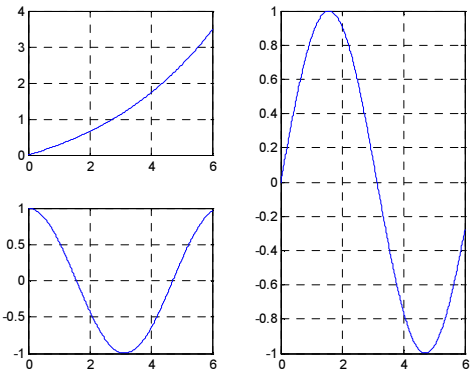


Figura M1.16

Stem

Nem sempre estaremos interessados em dar a ilusão de que a quantidade representada, que é sempre a sequência de elementos de um vector e portanto é inerentemente discreta, que a quantidade representada, dizia-mos, é uma sequência contínua de valores.

Utilizando a função *plot* podemos realçar que a quantidade a representar é um conjunto discreto de pontos. Por exemplo, especificando o símbolo * e sem linha de ligação

```
n=0:0.25:6;
f=cos(n);
plot(n,f,'*')
grid on;
```

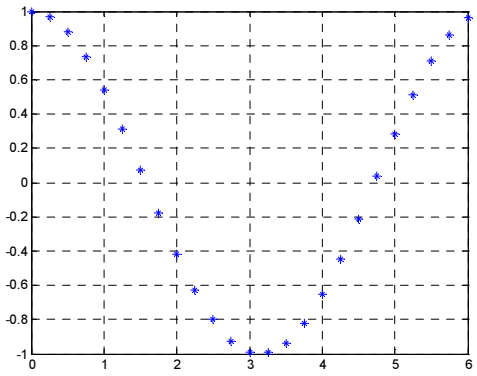


Figura M1.17

Podemos dar um realce ainda maior à característica discreta da quantidade a representar recorrendo à **função stem**. Por exemplo

```
n=0:0.25:6;
f=cos(n);
stem(n,f)
grid on;
```

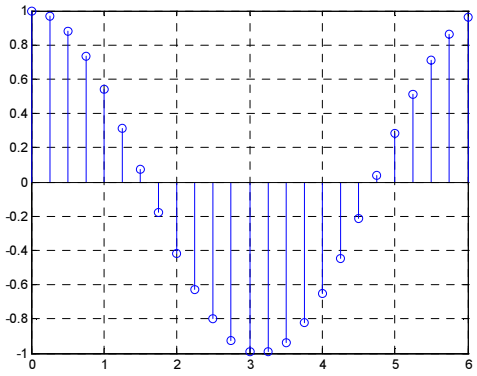


Figura M1.18

Recorra ao comando *help stem* para conhecer os parâmetros que a função *stem* aceita. Podemos por exemplo representar os pontos a cheio

```
n=0:0.25:6;
f=cos(n);
stem(n,f,'filled')
grid on
```

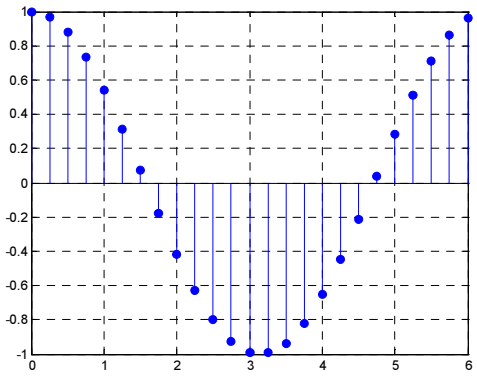


Figura M1.19

Por vezes é importante realçar que se está na presença de um fenómeno que embora representado por um conjunto discreto de pontos, respeita a evolução de uma determinada função contínua. Podemos nestes casos utilizar, no mesmo espaço gráfico, a função *plot* e a função *stem*

```
x=0:0.1:12
f1=cos(x).*exp(-x/4);
plot(x,f1,':k');
hold on
n=0:12;
f2=cos(n).*exp(-n/4);
stem(n,f2,'filled')
grid on
hold off
```

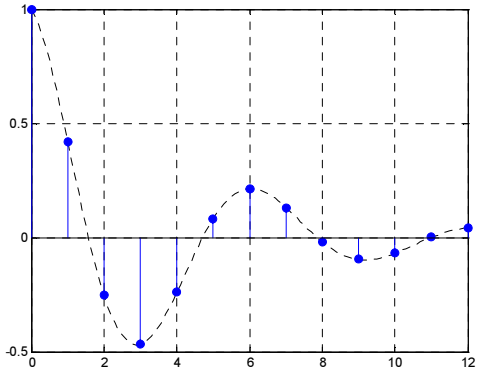


Figura M1.20

4. Complexos

A unidade imaginária, $j = \sqrt{-1}$, é representada em *Matlab* quer pela letra i, quer pela letra j. Podemos facilmente criar uma variável, ou um vector complexo

```
>> z=2-3j
z =
   2.0000 - 3.0000i
>> z1=[2.1-3.4j 1+2.0j 2-j 1.5+j]
z1 =
   2.1000 - 3.4000i   1.0000 + 2.0000i   2.0000 - 1.0000i   1.5000 + 1.0000i
>>
```

A representação gráfica de um complexo não deve ser feita recorrendo à função *plot*. O *Matlab* colocará sequencialmente os valores complexos no plano, representando a parte imaginária em função da parte real, e uni-los-á com uma linha contínua, donde resultará um gráfico provavelmente irreconhecível.

```
>> plot(z1)
>> grid on
>>
```

Veja o gráfico resultante na figura M1.21. Podemos verificar o procedimento calculando a parte real dos complexos com a **função real** e a parte imaginária com a **função imag**, e representando graficamente a parte real em função da parte imaginária.

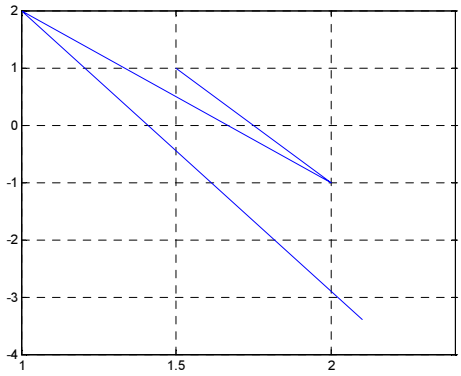


Figura M1.21

```
>> rz1=real(z1)
rz1 =
    2.1000    1.0000    2.0000
1.5000
>> iz1=imag(z1)
iz1 =
   -3.4000    2.0000   -1.0000
1.0000
>> plot(rz1,iz1,'ok:');grid on
>> xlabel('Rel (z1)');
>> ylabel('Imag (z1)');
>>
```

Este não é o modo conveniente de representar um complexo. A representação deverá ser feita, por exemplo, através de um diagrama polar, recorrendo à **função compass**

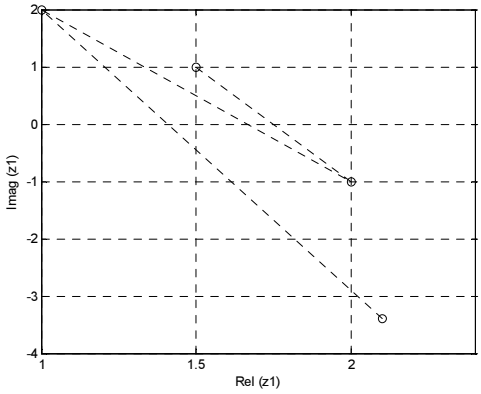


Figura M1.22

```
>> compass(z1)
>>
```

Observe e interprete a figura M1.23. Podemos verificar o módulo e o argumento de cada vector complexo recorrendo, respectivamente, à **função abs** e à **função angle**

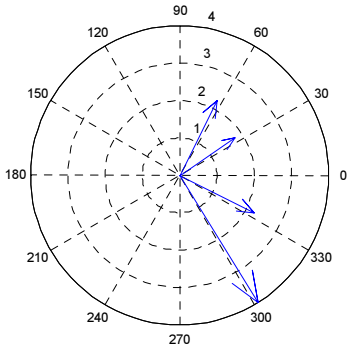


Figura M1.23

```
>> mz1=abs(z1)
mz1 =
    3.9962    2.2361    2.2361
1.8028
>> az1=angle(z1)
az1 =
   -1.0175    1.1071   -0.4636
0.5880
>>
```

Note que o ângulo está expresso em radianos. Podemos facilmente converte-lo em graus, para uma melhor interpretação da figura

```
>> gz1=az1*180/pi
```

```
gz1 =
  -58.2986   63.4349  -26.5651   33.6901
>>
```

Pode fazer a conversão recorrendo às **funções rad2deg** e **deg2rad**

```
>> rad2deg(az1)
ans =
  -58.2986   63.4349  -26.5651   33.6901
>> deg2rad(gz1)
ans =
   -1.0175    1.1071   -0.4636    0.5880
>>
```

Outro modo de representar convenientemente um vector complexo é o traçado de dois gráficos em separado, um com a representação das componentes reais e outro com a representação das componentes imaginárias, ou, ainda, um gráfico com a representação dos módulos e outro com a representação dos argumentos. Estas representações são incontornáveis sobretudo para vectores complexo de grandes dimensões. Considere por exemplo a função

$$f(x) = 0.9^x e^{j0.5x}$$

com $x \in [0, 6\pi]$. O recurso à função *compass* resulta num gráfico incompreensível. Quanto muito poderíamos recorrer à **função polar** para representar a evolução do afixo do complexo no plano de Argand. Veja a figura M1.24.

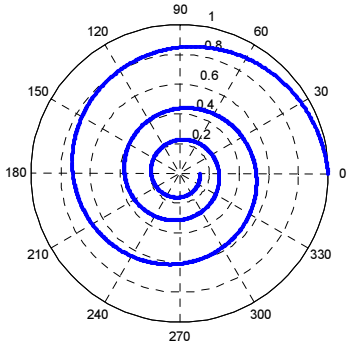


Figura M1.24

```
x=0:0.01:6*pi;
f=0.9.^x.*exp(j*x);
figure(1);
polar(angle(f),abs(f),'.');
```

Mas o mais comum seria representar o módulo e o argumento. Veja a figura M1.25.

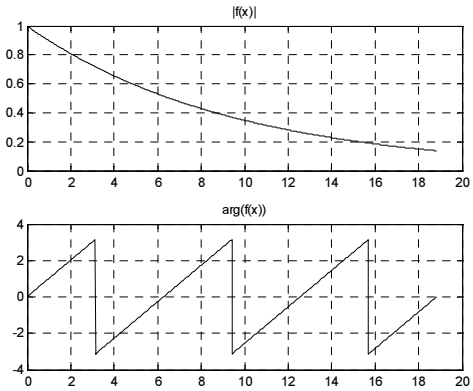


Figura M1.25

```
figure(2);
subplot(2,1,1);
plot(x,abs(f),'k');
grid on;
title('|f(x)|');
subplot(2,1,2);
plot(x,angle(f),'k');
grid on;
title('arg(f(x))');
```

Ou a parte real e a parte imaginária. Veja a figura M1.26.

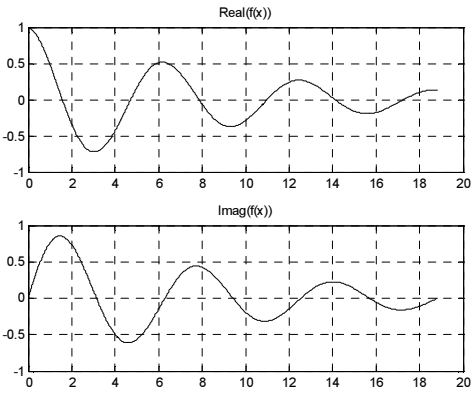


Figura M1.26

```
figure(3);
subplot(2,1,1);
plot(x,real(f),'k');
grid on;
title('Real(f(x))');
subplot(2,1,2);
plot(x,imag(f),'k');
grid on;
title('Imag(f(x))');
```

5. Controlo de fluxo.

If-then-else

Tal como nas linguagens de programação que já conhece, existe em *Matlab* a instrução **if**, que lhe permite executar condicionalmente um dado conjunto de instruções, atendendo à veracidade ou não de uma dada condição. Execute o comando `help if` para conhecer a sintaxe da instrução de controlo de fluxo **if**. Dão-se seguidamente alguns exemplos (Crie um ficheiro .m com as linhas de código, execute-o e interprete os resultados)

```
a=1;
b=2;
c=3;

if b>a
    d=4
end
%%%%%%%%%%%%%%%%%
if b<a
    d=2
else
    d=4
end
%%%%%%%%%%%%%%%%%
if b<a
    d=2
elseif c<a
    d=1
else
    d=4
end
%%%%%%%%%%%%%%%%%
```

For

Como sabe, a instrução for permite-lhe a repetição de um conjunto de instruções. Execute o comando `help for` para conhecer a sintaxe da instrução **for**. Por exemplo

```
n=0;
for i=1:10
    n=n+1;
end
```

Dado que o *Matlab* é uma linguagem interpretada, e dadas as suas capacidades de vectorização, o recurso a ciclos *for* deve ser reduzido ao mínimo estritamente necessário.

6. Alocação de memória. Vectorização

Como se disse, os operadores aritméticos básicos são, em *Matlab*, por defeito, operadores matriciais. Embora não o tivéssemos explicitado, reparou com certeza que as funções pré-definidas podem receber como argumento vectores ou matrizes, não sendo necessário escrever o código necessário para a aplicação da função a cada um dos elementos dessas estruturas. Por exemplo, considerando um vector, v, com valores no intervalo $[0, 2\pi]$, para criar um vector , v_2, cujos elementos sejam o valor do seno de cada elemento do vector v, basta fazer

```
v=0:0.1:2*pi;
v2=sin(v);
```

Se o argumento da função seno fosse um escalar, seria obrigado a construir um ciclo *for* que percorresse todos os elementos do vector v

```
v=0:0.1:2*pi;
for i=1:length(v)
    v2(i)=sin(v(i));
end
```

O conjunto de instruções acima infringe duas regras fundamentais que deve ter sempre em atenção na escrita do seu código:

- **Nunca alocar memória dinamicamente**
- **Nunca executar ciclos *for* desnecessariamente**

Alocação de memória

No conjunto de instruções imediatamente acima, o vector v_2 não tem dimensão definida antes de entrar no ciclo *for*. Como em cada ciclo a sua dimensão é acrescida de uma unidade de memória, isso implica a existência de um pedido de reserva de memória por ciclo, o que torna o algoritmo altamente ineficiente. Sempre que seja necessário executar um ciclo de cálculo, deve ser alocada memória para as varáveis nele intervenientes, especificando a sua dimensão final e preenchendo as variáveis a ser utilizadas com zeros. No caso do exemplo acima, deveríamos ter escrito

```
v=0:0.1:2*pi;
lv=length(v);
v2=zeros(1,lv);
for i=1:lv
    v2(i)=sin(v(i));
end
```

Se o número de ciclos for muito grande, a diferença entre os tempos de processamento é muito significativa. Podemos verificar isso facilmente

```
clear all
v=linspace(0,2*pi,10000);
lv=length(v);
v2=zeros(1,lv);
t0=cputime;
for i=1:lv
    v2(i)=sin(v(i));
end
tca=cputime-t0;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear v2
v=linspace(0,2*pi,10000);
t0=cputime;
for i=1:lv
    v2(i)=sin(v(i));
end
tsa=cputime-t0;
disp(sprintf('Tempo de processamento com alocação de memória: %2.1f
                ms',tca*1e3));
disp(sprintf('Tempo de processamento sem alocação de memória: %4.1f  s
                \n',tsa));
disp(sprintf('O tempo sem alocação é %4.1f x superior ao tempo com
```

```
alocação',tsa/tca));
```

Num Pentium IV 1.7 GHz a resposta deverá ser, tipicamente

```
>>
Tempo de processamento com alocação de memória: 15.0 ms
Tempo de processamento sem alocação de memória:  0.8  s

O tempo sem alocação é 50.0 x superior ao tempo com alocação
>>
```

Vectorização

Uma vez que o *Matlab* é uma linguagem interpretada, a existência de ciclos de cálculo (ciclos *for* ou outros) torna os algoritmos escritos em *Matlab* extremamente ineficientes, uma vez que antes da sua execução cada linha de código tem de ser lida e interpretada, sendo na maior parte dos casos completamente evitável. Vimos acima que, dado que a função seno admite um vector como argumento, o cálculo da função elemento a elemento é completamente desnecessária. Vejamos outro exemplo. Admita que, dados dois vectores, v_1 e v_2, pretendemos calcular a soma do produto ordenado das componentes de cada um dos vectores, ou seja, o seu produto interno

$$< v_1, v_2 >= \sum_{i=1}^{N} v_1(i)\, v_2(i)$$

Podemos facilmente calcular a diferença entra os tempos de processamento entre o modo correcto (vectorizado) e incorrecto (recorrendo a um ciclo for) de escrita deste algoritmo

```
N=1e6;
v1=rand(1,N);
v2=rand(1,N);
t0=cputime;
s=v1*v2';
tcv=cputime-t0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s=0;
t0=cputime;
for i=1:N
        s=s+v1(i)*v2(i);
    end
end
tsv=cputime-t0

disp(sprintf('Tempo de processamento com vectorização: %4.1f ms',tcv*1e3));
disp(sprintf('Tempo de processamento sem vectorização: %4.1f ms
\n',tsv*1e3));
disp(sprintf('O tempo sem vectorização é %4.1f x superior ao tempo com
vectorização',tsv/tcv));
```

Num Pentium IV 1.7 GHz a resposta deverá ser, tipicamente

```
Tempo de processamento com vectorização: 16.0 ms
Tempo de processamento sem vectorização: 47.0 ms

O tempo sem vectorização é  2.9 x superior ao tempo com vectorização
```

7. Scripts e Funções

Scripts

Neste momento já está familiarizado com a utilização de um **ficheiro .m**, contendo um conjunto de comandos que pode executar em sequência, escrevendo na consola o nome do ficheiro. Em *Matlab* este tipo de ficheiros é designado por **script**. Todas as variáveis definidas dentro de ficheiros deste tipo passa, após a execução do ficheiro, a fazer parte do espaço de trabalho, isto é, são **variáveis globais**, podendo ser evocadas na consola ou dentro de outros ficheiros *script*.

Para além das vantagens já apontadas, de poder guardar todos os comandos que executa, identação do código, e *debuging*, os ficheiros podem, evidentemente, ser usados para realizar repetidamente um conjunto de linhas de código em que, por edição do ficheiro, se alteram algumas características. Para este tipo de procedimento, no entanto, é mais aconselhável a utilização de **funções definidas pelo utilizador**.

Funções

Tal como para os *scripts*, para definirmos uma função criamos um ficheiro .m. Quando evocamos o nome do ficheiro, na consola, dentro de um ficheiro *script*, ou dentro do corpo de outra função, o conjunto de linhas de código existente no ficheiro é executado. Para que o ficheiro .m corresponda a uma função e não a um *script* a 1ª linha de código deve ter a sintaxe

```
function [variáveis de saída]=nome_da_função(variáveis de entrada)
```

Faz todo o sentido que o nome da função corresponda ao nome do ficheiro .m. Note que, se assim não for (por engano) o *Matlab* reconhece a função pelo nome do ficheiro e não pelo nome que conste no cabeçalho. Não use nomes em inglês pois podem entrar em conflito com funções pré-definidas em *Matlab*.

Todas as variáveis definidas dentro do corpo da função só têm existência enquanto a função estiver a ser executada, isto é, são **variáveis locais**, não podendo por isso ser evocadas fora do corpo da função. Dão-se seguidamente alguns exemplos que se consideram auto-explicativos. Para qualquer dúvida, já sabe, recorra ao comando *help function*.

Exemplo 1

Função com apenas um parâmetro de entrada e um parâmetro de saída definida num ficheiro `minha_media.m`

```
function y = minha_media(x)
% Calcula a media dos valores de x
%
y=sum(x)/length(x);
```

Uma vez salvo o ficheiro, a função pode agora ser evocada a partir da consola, sendo o resultado atribuído ou não a uma variável. Já sabe, se não quiser eco da execução da função use o operador ;

```
>> media= minha_media ([1 2 3 4])
media =
    2.5000
>>
>> minha_media([1 2 3 4])
ans =
    2.5000
>>
>> media= minha_media ([1 2 3 4]);
>> media
media =
    2.5000
>>
```

Deve habituar-se a escrever comentários logo abaixo do cabeçalho, explicativos da utilização da função. Ao fazer *help nome_da_função* estes comentários são apresentados na consola, evitando assim que seja necessário editar a função para conhecer a sua funcionalidade.

```
>> help minha_media

  Calcula a media dos valores de x
>>
```

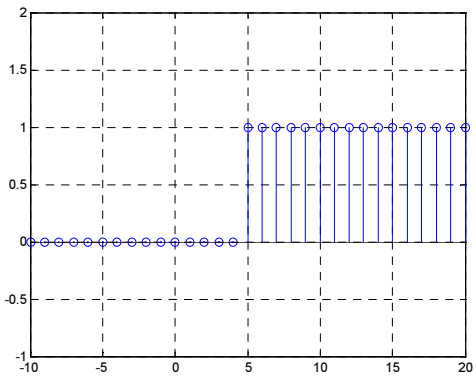
Exemplo 2

Função com três parâmetros de entrada e dois parâmetros de saída definida num ficheiro `meu_heaviside.m`

```
function [n, x] = meu_heaviside(n0,n1,n2)
%
% [n, x] = meu_heaviside(n0,n1,n2)
%
% Define a função de Heaviside para
% valores inteiros entre n1 e n2
% com o degrau situado em n0.
%    x[n]=u[n-n0] ; n1<= n <=n0

n=[n1:n2];
x=[(n-n0)>=0];
```

Uma vez salvo o ficheiro, a função pode agora ser evocada a partir da consola

```
>> help meu_heaviside

  [n, x] = meu_heaviside(n0,n1,n2)

 Define a função de Heaviside para
  valores inteiros entre n1 e n2
  com o degrau situado em n0.
     x[n]=u[n-n0] ; n1<= n <=n0
>>
>> [m,z]= meu_heaviside(5,-10,20);
>> stem(m,z)
>> grid on
>> axis([-10 20 -1 2])
>>
```



Note que ao definir uma função esta só pode ser evocada se se encontrar na corrente directoria de trabalho. Se a sua função tiver um nome coincidente com uma outra pré-definida, o *Matlab* passa a executar a sua função, e não a existente algures no sistema de directorias *Matlab* e acessível porque tal directoria está definida no **path**.

Se definir um grupo de funções que lhe interessa evocar de qualquer directoria de trabalho, coloque-as numa só directoria e junte essa directoria ao seu *path*. Recorra ao comando `help path` para esclarecer o procedimento a ter. Por exemplo, se quiser juntar a directoria `as_minhas_funcoes` ao seu *path* faça

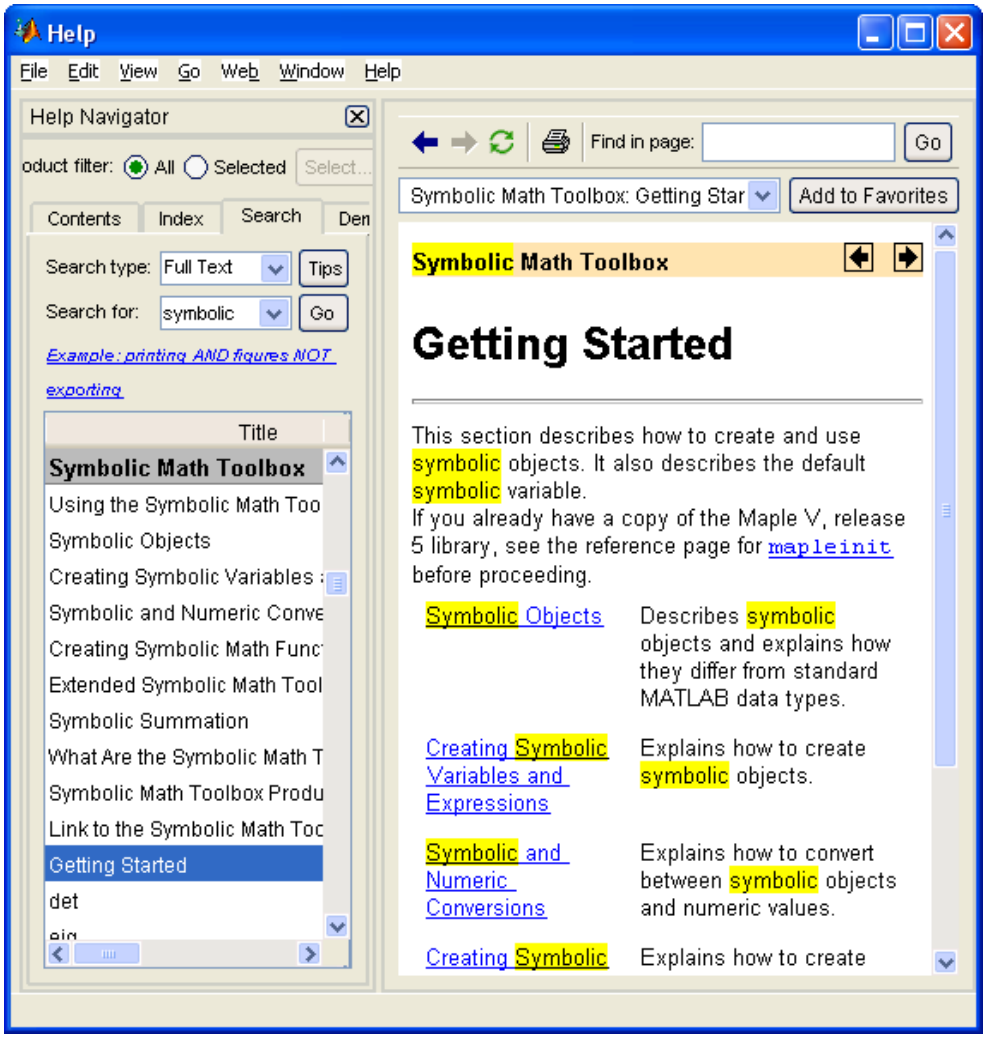
```
>>path(path,'c:\a_minha_arvore\as_minhas_funcoes')
>>
```

Pode assim construir uma **biblioteca** com as suas próprias funções.

8. Cálculo simbólico

Symbolic Math Toolbox

Para uma referência rápida ao conjunto de funções existentes em *MatLab* no domínio do cálculo simbólico faça *help symbolic*. Para uma completa descrição do modo como pode criar e manipular variáveis e expressões simbólicas abra a janela *help* do *Matlab*, faça uma busca *full text* sobre a palavra *symbolic* e, na subsecção *Symbolic Math Toolbox* active a opção *Getting Started*.



Conceitos básicos

Se a *Symbolic Math Toolbox* estiver instalada, pode manipular expressões que, além de valores e variáveis numéricas, contenham variáveis simbólicas. Por exemplo, para definir x como uma variável simbólica fazemos

```
>> syms x
>>
```

Uma vez definido que x é uma variável simbólica, podemos, por exemplo, definir funções que envolvam esta variável,

```
>> f=2*x^2+3*x-5;
>> g=x^2-x+7;
>>
```

e fazer uma série de operações que envolvam estas funções. Por exemplo

```
>> f+g
ans =
3*x^2+2*x+2
>>
```

```
>> f-g
ans =
x^2+4*x-12
>>
>> f*g
ans =
(2*x^2+3*x-5)*(x^2-x+7)
>>
>> expand(ans)
ans =
2*x^4+x^3+6*x^2+26*x-35
>>
>> f/g
ans =
(2*x^2+3*x-5)/(x^2-x+7)
>>
>> expand(ans)
ans =
2/(x^2-x+7)*x^2+3/(x^2-x+7)*x-5/(x^2-x+7)
>>
>> pretty(ans)
```

$$2\,\frac{x^2}{x^2-x+7}+3\,\frac{x}{x^2-x+7}-\frac{5}{x^2-x+7}$$

```
>>
>> f^2
ans =
(2*x^2+3*x-5)^2
>>
>> expand(ans)
ans =
4*x^4+12*x^3-11*x^2-30*x+25
>>
```

Note que o *Matlab* não faz as simplificações ou expansões automaticamente. Recorremos para isso à **função simplify** e à **função expand**. No caso de expressões demasiado complexas, podemos recorrer à **função pretty** para uma mais fácil leitura. Podemos ainda recorrer à **função simple**, que tenta encontrar a forma mais simples de escrever uma expressão.

```
>> f^2
ans =
(2*x^2+3*x-5)^2
>> simple(ans)
simplify:
      (2*x^2+3*x-5)^2
radsimp:
      (2*x^2+3*x-5)^2

combine(trig):
      4*x^4+12*x^3-11*x^2-30*x+25
factor:
      (2*x+5)^2*(x-1)^2
expand:
      4*x^4+12*x^3-11*x^2-30*x+25
combine:
      (2*x^2+3*x-5)^2
convert(exp):
      (2*x^2+3*x-5)^2
convert(sincos):
      (2*x^2+3*x-5)^2
convert(tan):
      (2*x^2+3*x-5)^2
collect(x):
      4*x^4+12*x^3-11*x^2-30*x+25
ans =
(2*x^2+3*x-5)^2
>> pretty(ans)
```

$$(2x^2+3x-5)^2$$

```
>>
```

Outras funções de interesse no domínio do cálculo simbólico são: a **função compose** que determina a expressão da função composta

```
>> f=1/(1-x^2);
>> g=sin(x);
>> compose(f,g)
ans =
     1/(1-sin(x)^2)
>>
```

; a **função finverse** que determina a função inversa

```
>> finverse(g)
ans =
     asin(x)
>>
```

; a **função subs** que lhe permite substituir uma variável simbólica por uma quantidade numérica.

```
>> subs(f,x,4)
ans =
     -0.0667
>>
```

; a **função diff** que determina a derivada

```
>> diff(g)
ans =
     cos(x)
>>
```

; a **função int** que determina a primitiva

```
>> int(f)
ans =
     atanh(x)
>>
```

; a **função limit** que lhe permite calcular o limite da função quando a variável simbólica tende para um valor especificado

```
>> limit(f,x,0)
ans =
     1
>> limit(f,x,inf)
ans =
     0
>>
```

9. Exercícios M1

Exemplo 1

- Crie a matriz

$$A=\begin{bmatrix}0&0&0&1&1&1\\0&0&0&1&1&1\\0&0&0&1&1&1\\2&2&2&3&3&3\\2&2&2&3&3&3\\2&2&2&3&3&3\end{bmatrix}$$

```
>> B=ones(3);
>> A=[0*B B;2*B 3*B];
A =
     0     0     0     1     1     1
     0     0     0     1     1     1
     0     0     0     1     1     1
     2     2     2     3     3     3
     2     2     2     3     3     3
     2     2     2     3     3     3
>>
```

- A partir da matriz A crie a matriz

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

```
>> B=A;
>> B(:,3:4)=[]
B =
     0     0     1     1
     0     0     1     1
     0     0     1     1
     2     2     3     3
     2     2     3     3
     2     2     3     3

>>
```

- A partir da matriz B crie a matriz

$$C=\begin{bmatrix}0&1\\0&1\\2&3\\2&3\end{bmatrix}$$

```
>> C=B(2:5,2:3)
C =
     0     1
     0     1
     2     3
     2     3
>>
```

- Altere os elementos $1,13$ e 25 da matriz A para 7.

```
>> A(1:13:25)=7
A =
     7     0     7     1     7     1
```

```
     0     0     0     1     1     1
     0     0     0     1     1     1
     2     2     2     3     3     3
     2     2     2     3     3     3
     2     2     2     3     3     3
>>
```

- Seleccione as colunas ímpares da matriz A

```
>> A(:,1:2:length(A))
ans =
     7     7     7
     0     0     1
     0     0     1
     2     2     3
     2     2     3
     2     2     3
>>
```

Exemplo2

- Com base nos conceitos de álgebra linear resolva o sistema de equações

$$\begin{cases} x_1 + x_2 + x_3 = 4 \\ x_1 - 2x_2 - x_3 = -3 \\ x_1 + x_2 = 1 \end{cases}$$

Do sistema escrito na forma matricial

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & -1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ 1 \end{bmatrix}$$

Resulta de imediato

```
>> A=[1 1 1;1 -2 -1;1 1 0]
A =
     1     1     1
     1    -2    -1
     1     1     0
>> a=[4 -3 1]'
a =
     4
    -3
     1
>> x=A^-1*a
x =
    0.6667
    0.3333
    3.0000
>>
```

Exemplo 3

- Gere um vector com os múltiplos de 5 compreendidos entre 100 e 200 por ordem decrescente.

```
>> v=200:-5:100;
>>
```

- Gere um vector com elementos entre 0 e 2π com um passo $\pi/8$.

```
>> v=0:pi/8:2*pi;
>>
```

- Gere um vector v com 40 elementos equiespaçados do valor inicial 0 ao valor final 77. Gere um vector v_2 contendo os elementos de índice par do vector v. Gere um vector v_3 contendo os elementos de índice impar do vector v.

```
>> v =linspace(0,77,40);
>> v2=v(2:2:length(v));
>> v3=v(1:2:length(v));
>>
```

Exemplo 4

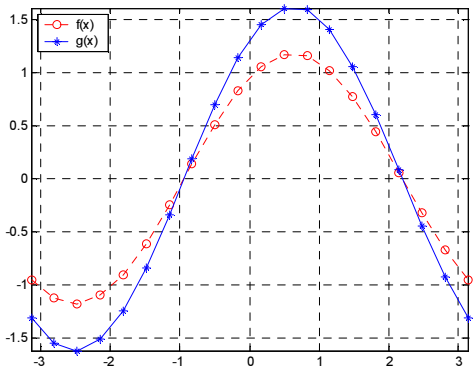
- Gere um vector, x, com 100 elementos equiespaçados pertencentes ao intervalo $[-\pi, \pi]$.

```
>> x=linspace(-pi,pi,100);
>>
```

- Calcule o valor das funções $f(x) = \sin(x) + \cos(x + \pi/10)$ e $g(x) = \cos(x) + \text{sen}(x + \pi/10)$ em 100 pontos equiespaçados do intervalo $[-\pi, \pi]$.

```
>>f=sin(x)+cos(x+pi/10);
>>g=cos(x)+sin(x+pi/10);
>>
```

- Trace os gráficos das duas funções. Represente $f(x)$ a vermelho com círculos e linha a tracejado. Represente $g(x)$ a azul com asteriscos e linha a cheio. Limite a escala horizontal e vertical aos valores limite das grandezas representadas. Coloque uma legenda e uma grelha no gráfico.




```
>> plot(x,f,':ro',x,g,'-*b')
>> grid on>> axis([-pi pi
     min(min(g,f)) max(max(g,f))])
>> legend('f(x)','g(x)',2)
>>
```

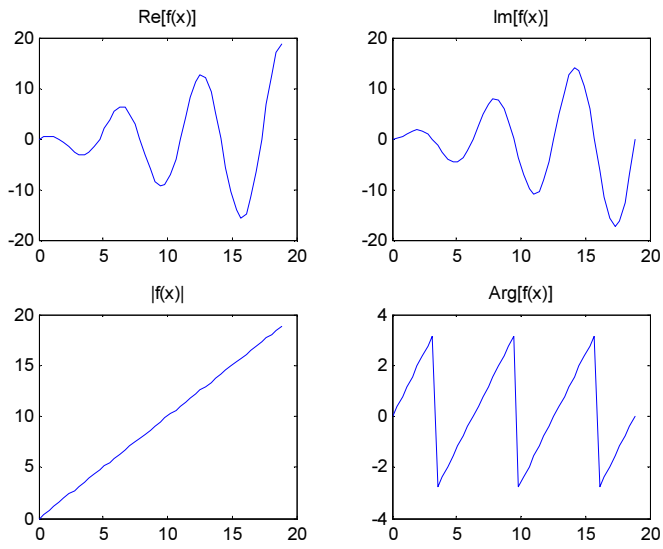
Exemplo 6

Trace o gráfico do módulo, do argumento, da parte real e da parte imaginária da função

$$f(x) = xe^{jx}$$

com $x \in [0, 6\pi]$ e incrementos de $\pi/8$.

```
>> x=0:pi/8:6*pi;
>> f=x.*exp(j*x);
>> subplot(2,2,1)
>> plot(x,real(f))
>> title('Re[f(x)]')
>> subplot(2,2,2)
>> plot(x,imag(f))
>> title('Im[f(x)]')
>> subplot(2,2,3)
>> plot(x,abs(f))
>> title('|f(x)|')
>> subplot(2,2,4)
>> plot(x,angle(f))
>> title('Arg[f(x)]')
>>
```



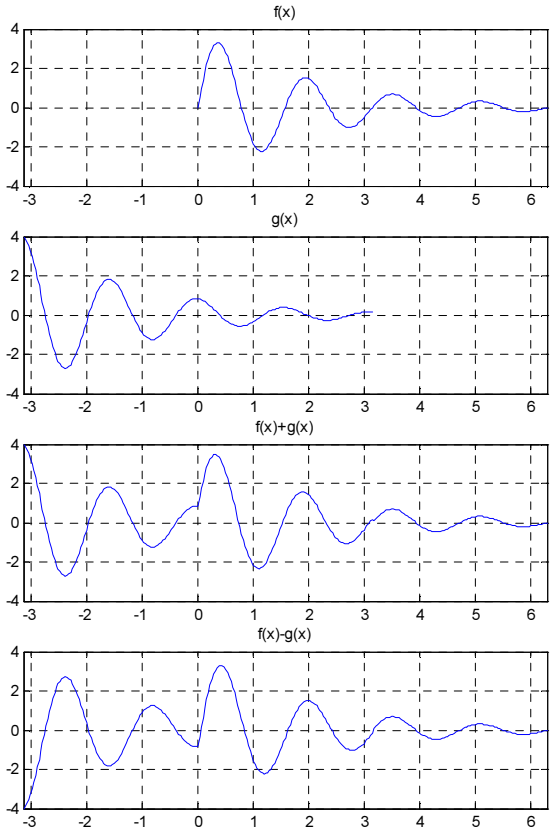
Exemplo 7

Defina dois vectores com as funções $f(x) = 4e^{-x/2}\,\text{sen}(4x)$ com $x \in [0, 2\pi]$ e $f(x) = 4e^{-x/2}\cos(4x)$ com $x \in [-\pi, \pi]$, e ambas as abcissas com incrementos $i = \pi/100$.

Defina uma função *Matlab* que calcule a soma e a diferença das duas funções. Defina uma outra função *Matlab* que trace o gráfico de cada uma das funções, da sua soma, e da sua diferença.

```
%%%%%%%%%%%%%%%%%%%%%%%%%
function [sfg,dfg,x] = funcao1(x1,f,x2,g,d)
  x=min(min(x1),min(x2)):d:max(max(x1),max(x2));
  y1=zeros(1,length(x));
  y2=y1;
  y1(find((x>=min(x1))&(x<=max(x1))==1))=f;
  y2(find((x>=min(x2))&(x<=max(x2))==1))=g;
  sfg=y1+y2;
  dfg=y1-y2;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%
function
funcao2(x1,f,x2,g,s,d,x)
  mx=min(x);
  Mx=max(x);
  my=min([f,g,s,d]);
  My=max([f,g,s,d]);

  subplot(4,1,1);
  plot(x1,f);
  grid on
  axis([mx Mx my My])
  title('f(x)')
  subplot(4,1,2);
  plot(x2,g);
  grid on
  axis([mx Mx my My])
  title('g(x)')
  subplot(4,1,3);
  plot(x,s);
  grid on
  axis([mx Mx my My])
  title('f(x)+g(x)')
  subplot(4,1,4);
  plot(x,d);
  grid on
  axis([mx Mx my My])
  title('f(x)-g(x)')
```



10. Auto-avaliação

Verifique os conhecimentos adquiridos através da resolução dos exercícios seguidamente propostos.

Exercício 1

- Crie o vector

$$V = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

- A partir do vector V crie a matriz

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

- A partir da matriz A crie a matriz

$$B = \begin{bmatrix} 4 & 8 \\ 6 & 12 \end{bmatrix}$$

- Altere os elementos 1, 4 e 7 da matriz A para 5.

- Seleccione as linhas pares da matriz A.

Exercício 2

- Gere um vector, x, com 41 elementos equiespaçados pertencentes ao intervalo $[0, 2\pi]$.

- Calcule o valor das funções $f(x) = 3\sin(x)$ e $g(x) = 2\cos(2x)$ em 41 pontos do intervalo $[0,2\pi]$.

- Trace os gráficos das duas funções. Represente $f(x)$ a azul com círculos e linha a tracejado. Represente $g(x)$ a preto com asteriscos e linha a cheio. Limite a escala horizontal e vertical aos valores limite das grandezas representadas. Coloque uma legenda e uma grelha no gráfico.

Exercício 3

- Trace o gráfico do módulo, do argumento, da parte real e da parte imaginária da função $f(x) = e^{jx}$ com $x \in [0, 2\pi]$ e incrementos $\pi/16$. Trace os gráficos sobre um espaço gráfico com 1 coluna e 4 linhas. Coloque títulos em todos os gráficos e etiquetas em todos os eixos.

Exercício 4

- Recorrendo à biblioteca de cálculo simbólico do *Matlab*, calcule

$$\lim_{x \to 0} \frac{\text{sen}(x)}{x}$$

$$\lim_{x \to \infty} \frac{\text{sen}(x)}{x}$$