Essential MATLAB commands

Enrico Bertolazzi

DIMS, University of Trento

Contents

1 Basic commands

MATLAB can be used as a calculator:

```
0001 >> 1+sin(3)*exp(-2)
0002
0003 ans =
0004
0005     1.0191
```

Standard operations like `+`, `-`, `*`, `/` works as expected. There are more or less all the standard functions `sin(x)`, `cos(x)`, `exp(x)` and so on.

The most important thing is that (nearly) all in MATLAB is a matrix. For example

```
0001 >> a = 1
0002 a =
0003      1
0004
0005 >> size(a)
0006 ans =
0007      1     1
```

thus, variable `a` is assigned yo the scalar 1 used as a 1×1 matrix. The command `size` shows this. In fact, the command `size` return a 1×2 matrix where the first element is the number of rows and the second is the number of columns.

Initialize a matrix in MATLAB is very easy, in fact:

```
0001 >> a = [ 1 2 3 ]
0002 a =
0003      1     2     3
0004
0005 >> size(a)
0006 ans =
0007      1     3
0008
0009 >> b = [ 1 2 3 ; 4 5 6 ]
0010 b =
0011      1     2     3
0012      4     5     6
0013
0014 >> size(b)
0015 ans =
0016      2     3
```

As you can see, a matrix is defined by listing the elements between square brakets. The elements of a matrix are listed by rows and separated by spaces (or optional commas). The rows are separated by `;`.

For big matrices it can be useful the continuation command `...` which tell MATLAB that the instruction continue in the next row. For example:

```
0001 >> a = [ 1 2 3 ; ...
0002           4 5 6 ; ...
0003           7 8 9 ]
0004 a =
0005      1     2     3
0006      4     5     6
0007      7     8     9
0008
0009 >> size(a)
0010 ans =
0011      3     3
```

1.1 Access to to the element of a matrix

Accessing to the elements of a matrix is quite easy, for example:

```
0001 >> a = [ 1 2 3 ; ...
0002           4 5 6 ; ...
0003           7 8 9 ]
0004 a =
0005      1     2     3
0006      4     5     6
0007      7     8     9
0008
0009 >> a(2,3)
0010 ans =
0011      6
0012
0013 >> a(2,1)
0014 ans =
0015      4
0016
0017 >> a(1)
0018 ans =
0019      1
0020
0021 >> a(2)
0022 ans =
0023      4
0024
0025 >> a(3)
0026 ans =
0027      7
0028
0029 >> a(4)
0030 ans =
0031      2
0032
0033 >> a(5)
0034 ans =
0035      5
```

notice that there are two ways to access the elements:

1. with two indices `a(i,j)` where `i` is the row index and `j` is the column index.
2. with one index `a(i)` where `i` is the index of the `i`th element of the matrix as *unrolled* by column.

one index access is useful for $1 \times n$ or $n \times 1$ matrices because they behave live vectors.

1.2 Matrix block accessing

Extraction and manipulation of portion of a matrix is as easy as accessing and manipulating a single element. Substituting scalars with vector in the index access command `a(i,j)` than a block of rows and columns extraction is performed (slicing). Similarly substituting the scalars with a vector in the index access command `a(i)` then a vector with the listed elements is extracted.

```
0001 >> a = [ 1 2 3 1 1 1 ; ...
0002           4 5 6 2 2 2 ; ...
0003           7 8 9 3 2 1 ]
0004 a =
0005      1     2     3     1     1     1
0006      4     5     6     2     2     2
0007      7     8     9     3     2     1
0008
0009 >> a( [1,3], [1,2,5] )
0010 ans =
0011      1     2     1
0012      7     8     2
0013
0014 >> a( [1,3,5,8,12] )
0015 ans =
0016      1     7     5     6     3
```

This way to work is useful in writing algorithm that operate on block of matrices or to vectorize conditional instruction.

1.3 Operation on matrices

For the matrix the operator `+`, `-`, `*` work as one expect:

```
0001 >> a = [ 1 2 3 ; ...
0002           4 5 6 ; ...
0003           7 8 9 ]
0004 a =
0005      1     2     3
0006      4     5     6
0007      7     8     9
0008
0009 >> b = [ 1 0 1 ; ...
0010           2 1 1 ; ...
0011           1 0 2 ]
0012 b =
0013      1     0     1
0014      2     1     1
0015      1     0     2
0016
0017 >> a+b
0018 ans =
```

```
0019      2     2     4
0020      6     6     7
0021      8     8    11
0022
0023 >> a-b
0024 ans =
0025      0     2     2
0026      2     4     5
0027      6     8     7
0028
0029 >> a*b
0030 ans =
0031      8     2     9
0032     20     5    21
0033     32     8    33
```

A comment should be done for the operators `/` and `\`. In fact `a/b` is formally equivalent to `a*inv(b)` where `inv(b)` is the function that build teh inverse of `b`. In practice `a/b` do not build the inverse of `b` but solve the problem `a=x*b`. The command `a\b` is formally equivalent to `inv(a)*b`. Practically `a\b` do not build the inverse of `b` but solve the problem `a*x=b`.

```
0001 >> a = [ 4 1 1 ; ...
0002           1 4 1 ; ...
0003           1 2 4 ]
0004 a =
0005      4     1     1
0006      1     4     1
0007      1     2     4
0008
0009 >> b = [ 3 0 1 ; ...
0010           2 3 1 ; ...
0011           1 1 3 ]
0012 b =
0013      3     0     1
0014      2     3     1
0015      1     1     3
0016
0017 >> norm( a/b - a*inv(b) )
0018 ans =
0019    1.2776e-16
0020
0021 norm( a\b - inv(a)*b )
0022 ans =
0023    1.5348e-16
```

The function `norm(a)` is the L^2 norm of the matrix `a`, i.e.

$$\|\boldsymbol{A}\| = \sqrt{\sigma(\boldsymbol{A}^T\boldsymbol{A})}$$

where $\sigma(\boldsymbol{B})$ is the spectral radius of $\boldsymbol{B}$, i.e the maximum module of the eigenvalues of $\boldsymbol{B}$.

Another important operator on matrice is transposition done by `'`:

```
0001 >> a = [ 4 2 2 ; ...
0002          1 4 1 ; ...
0003          1 2 4 ]
0004 a =
0005      4     2     2
0006      1     4     1
0007      1     2     4
0008
0009 >> a'
0010 ans =
0011      4     1     1
0012      2     4     2
0013      2     1     4
```

1.4 Element by element operations

A useful characteristic of MATLAB is the possibility to operate easily on a matrix element by element. The operators working in this way are prefixed by the *dot* ".". For example:

```
0001 a = [ 4 2 2 ; ...
0002       1 2 4 ] ;
0003
0004 b = [ 1 2 3 ; ...
0005       2 3 3 ] ;
0006
0007 a .* b % prodotto elemento per elemento
0008 ans =
0009      4     4     6
0010      2     6    12
0011
0012 a ./ b % divisione elemento per elemento
0013 ans =
0014     4.0000    1.0000    0.6667
0015     0.5000    0.6667    1.3333
0016
0017 a .^ b  % elevamento a potenza elemento per elemento
0018 ans =
0019      4     4     8
0020      1     8    64
```

In the example `^` is the power operator, further notice that by writing `;` at the end of the instruction there is the *output suppression*. In fact normally the result of every instruction is echoed, putting `;` at the end of the instruction this echoes is suppressed. Finally notice that the operators behave differently if prefixed by `.`, in fact:

```
0001 a = [ 2 2 ; ...
0002       1 2 ] ;
0003
0004 a.^3
```

```
0005 ans =
0006      8     8
0007      1     8
0008
0009 a^3
0010 ans =
0011     20    28
0012     14    20
0013
0014 a*a*a
0015 ans =
0016     20    28
0017     14    20
```

(quite) all the function of MATLAB work element by element, for example

```
0001 a = [ 2.1 4.2    2.1 ; ...
0002       1   2.001  -0.1] ;
0003
0004 sin(a)
0005 ans =
0006     0.8632   -0.8716    0.8632
0007     0.8415    0.9089   -0.0998
0008
0009 cos(a)
0010 ans =
0011    -0.5048   -0.4903   -0.5048
0012     0.5403   -0.4171    0.9950
0013
0014 exp(a)
0015 ans =
0016     8.1662   66.6863    8.1662
0017     2.7183    7.3964    0.9048
```

Attention that if you want the matrix exponential, i.e.

$$\exp(\boldsymbol{A}) = \boldsymbol{I} + \boldsymbol{A} + \frac{1}{2}\boldsymbol{A}^2 + \cdots$$

you must use the function `expm`:

```
0001 >> a = [ 2.1 4.2   ; ...
0002          1   2.001 ] ;
0003
0004 >> exp(a)  % exponential element by element
0005 ans =
0006     8.1662   66.6863
0007     2.7183    7.3964
0008
0009 >> expm(a) % matrix xponential
```

```
0010 ans =
0011    31.4019   60.8176
0012    14.4804   29.9683
```

1.5 Building succession

Programming in MATLAB, often is convenient to have some operators which build a succession of integer or real with constant step. This operator is : (double dot):

```
0001 >> 1:4   % list form 1 to 4
0002 ans =
0003      1     2     3     4
0004
0005 >> 13:34 % list form 13 to 34
0006 ans =
0007   Columns 1 through 12
0008     13    14    15    16    17    18    19    20    21    22    23    24
0009   Columns 13 through 22
0010     25    26    27    28    29    30    31    32    33    34
0011
0012 >> 1:4:13 % list form 1 to 13 with step 4
0013 ans =
0014      1     5     9    13
0015
0016 >> 0.24:0.1:3
0017 ans =
0018   Columns 1 through 7
0019     0.2400    0.3400    0.4400    0.5400    0.6400    0.7400    0.8400
0020   Columns 8 through 14
0021     0.9400    1.0400    1.1400    1.2400    1.3400    1.4400    1.5400
0022   Columns 15 through 21
0023     1.6400    1.7400    1.8400    1.9400    2.0400    2.1400    2.2400
0024   Columns 22 through 28
0025     2.3400    2.4400    2.5400    2.6400    2.7400    2.8400    2.9400
```

these succession can be used to extract matrix subblock, for example

```
0001 >> a = [ 1  2  3  4  5 ;
0002          6  7  8  9  10 ;
0003          11 12 13 14 15 ] ;
0004
0005 >> a(1:2,3)
0006 ans =
0007      3
0008      8
0009
0010 >> a(2:end,1:end-1)
0011 ans =
0012      6     7     8     9
0013     11    12    13    14
```

The keyword end contains the row or column dimension depending on position. It is very useful on operating on matrices. For example if you want to *rotate* the elements of a vector you can use the following instructions

```
0001 >> v = [ 1  2  3  4  5 ]
0002 v =
0003      1     2     3     4     5
0004
0005 >> w = [ v(end) v(1:end-1) ]
0006 w =
0007      5     1     2     3     4
```

notice that if an element of a matrix is a block matrix this is expanded in the matrix

```
0001 >> v = [ 1 [1 2 3 4] ]
0002 v =
0003      1     1     2     3     4
```

thus, in MATLAB it is not possible to build recursive structure (not with the matrix), i.e. matrix with elements that are matrix. This restriction is quite useful because simplify the building of complex matrices. for example

```
0001 >> I3 = eye(3) % 3x3 identity matrix
0002 I3 =
0003      1     0     0
0004      0     1     0
0005      0     0     1
0006
0007 >> I4 = eye(4) % 4x4 identity matrix
0008 I4 =
0009      1     0     0     0
0010      0     1     0     0
0011      0     0     1     0
0012      0     0     0     1
0013
0014 >> M = [ I3 zeros( 3, 4) ; ones(4,3) I4 ]
0015 M =
0016      1     0     0     0     0     0     0
0017      0     1     0     0     0     0     0
0018      0     0     1     0     0     0     0
0019      1     1     1     1     0     0     0
0020      1     1     1     0     1     0     0
0021      1     1     1     0     0     1     0
0022      1     1     1     0     0     0     1
```

The function `eye(n)` build an $n \times n$ identity matrix, while `zeros(n,m)` and `ones(n,m)` builds $n \times m$ matrices filled by 0 or 1 respectively.

A shortcut for `1:end` is to use `:` as follows

```
0001 >> a = [ 1 2 3 4 ;
0002          5 6 7 8 ] ;
```

```
0003 >> a(:,1)
0004 ans =
0005      1
0006      5
0007
0008 >> a(1:end,1)
0009 ans =
0010      1
0011      5
0012
0013 >> a(2,:)
0014 ans =
0015      5     6     7     8
0016
0017 >> a(2,1:end)
0018 ans =
0019      5     6     7     8
```

2 Main matrix functions

MATLAB has a lot of functions for matrix manipulation, for example:

```
0001 >> M = [ 1 2 3 4 ;
0002          5 6 7 8 ;
0003          1 1 1 1 ;
0004          2 3 1 0 ] ;
0005
0006 >> e = eig(M) % eigenvalues of M
0007 e =
0008    11.5977
0009    -3.0282
0010    -0.5695
0011    -0.0000
0012
0013 >> [V,e] = eig(M)% eigenvalues and eigenvectors of M
0014 V =
0015     0.3200    0.5808    0.8121    0.1667
0016     0.8883    0.3638   -0.4641    0.1667
0017     0.1421   -0.0542   -0.3190   -0.8333
0018     0.2972   -0.7262    0.1527    0.5000
0019 e =
0020    11.5977         0         0         0
0021          0   -3.0282         0         0
0022          0         0   -0.5695         0
0023          0         0         0   -0.0000
0024
0025
0026 >> M * V(:,1) - e(1) * V(:,1) % check the first couple eigenvalue-eigenvector
```

```
ans =
   1.0e-13 *

         0
    0.1066
    0.0155
    0.0178


>> [L,U,P] = lu(M) % lu decomposition
L =
    1.0000         0         0         0
    0.2000    1.0000         0         0
    0.4000    0.7500    1.0000         0
    0.2000   -0.2500   -0.0000    1.0000

U =
    5.0000    6.0000    7.0000    8.0000
         0    0.8000    1.6000    2.4000
         0         0   -3.0000   -5.0000
         0         0         0    0.0000

P =
     0     1     0     0
     1     0     0     0
     0     0     0     1
     0     0     1     0

>> norm(P*L*U - M)  % check lu decomposition
ans =
     0

>> [Q,R] = qr(M) % QR factorization
Q =
   -0.1796    0.7671    0.5690   -0.2357
   -0.8980   -0.3002    0.2188    0.2357
   -0.1796   -0.2668   -0.0875   -0.9428
   -0.3592    0.5003   -0.7878   -0.0000

R =
   -5.5678   -7.0046   -7.3638   -8.0822
         0    0.9672    0.4336    0.4002
         0         0    2.3635    3.9392
         0         0         0   -0.0000

>> norm(Q*R - M)  % check QR factorization
ans =
   1.7681e-15
```

```
0075
0076 >> [U,S,V] = svd(M) % singular values decomposition
0077 U =
0078    -0.3636    0.4329    0.7904    0.2357
0079    -0.9034    0.0224   -0.3575   -0.2357
0080    -0.1349   -0.1026   -0.2870    0.9428
0081    -0.1829   -0.8953    0.4062    0.0000
0082
0083 S =
0084    14.5996         0         0         0
0085          0    2.9146         0         0
0086          0         0    0.5982         0
0087          0         0         0    0.0000
0088
0089 V =
0090    -0.3686   -0.4627   -0.7888    0.1667
0091    -0.4679   -0.6137    0.6138    0.1667
0092    -0.5296    0.1569   -0.0206   -0.8333
0093    -0.6039    0.6203    0.0240    0.5000
0094
0095 >> norm(U*S*V' - M) % check singular values decomposition
0096 ans =
0097    5.0559e-15
0098
0099 >> norm(M,1)       % 1-norm of matrix
0100 ans =
0101     13
0102
0103 >> norm(M,'inf')   % infinity-norm of matrix
0104 ans =
0105     26
0106
0107 norm(M,'fro')   % Frobenius-norm of matrix
0108 ans =
0109    14.8997
```

2.1 Some useful matrix functions

There are some matrix functions useful in the algorithm building. The functions `max(a)` and `min(a)` returns a row vector where each elements is the maximum (or the minimum) of the corresponding column. If the matrix is a column or row vector `max(a)` and `min(a)` return the maximum and minimum value respectively. Analogously the function `sum(a)` return a row vector with the cum of the values of the corresponding column of `a`. If the matrix is a column or row vector `sum(a)` return the sum of each components. Analogously work the function `prod(a)`.

```
0001 >> a = [ 1 2 3 -4 ;
0002          0 1 0 -2 ;
0003          2 3 0 -1 ] ;
```

```
0004
0005 >> max(a)
0006 ans =
0007      2     3     3    -1
0008
0009 >> min(a)
0010 ans =
0011      0     1     0    -4
0012
0013 >> sum(a)
0014 ans =
0015      3     6     3    -7
0016
0017 >> max(max(a))
0018 ans =
0019      3
0020
0021 >> min(min(a))
0022 ans =
0023     -4
0024
0025 >> sum(sum(a))
0026 ans =
0027      5
0028
0029 >> max(sum(abs(a))) % 1-norm
0030 ans =
0031      7
0032 >> norm(a,1)
0033 ans =
0034      7
0035
0036 >> max(sum(abs(a'))) % infinity-norm
0037 ans =
0038     10
0039 >> norm(a,'inf')
0040 ans =
0041     10
```

Assigning the results of function `max(a)` or `min(a)` to a couple of variables

```
0001 [res,idx] = max(a)
0002 [res,idx] = min(a)
```

then `res` contains a row vector with the maximum (or minimum) elements of each columns, while `idx` contains the row index of each maximum (minimum). For example

```
0001 >> a = [ 1 2 3 -4 ;
0002          0 1 0 -2 ;
0003          2 3 0 -1 ] ;
```

```
0004
0005 >> [res,idx] = max(a)
0006 res =
0007      2     3     3    -1
0008 idx =
0009      3     3     1     3
0010
0011 >> [res,idx] = min(a)
0012 res =
0013      0     1     0    -4
0014 idx =
0015      2     2     2     1
```

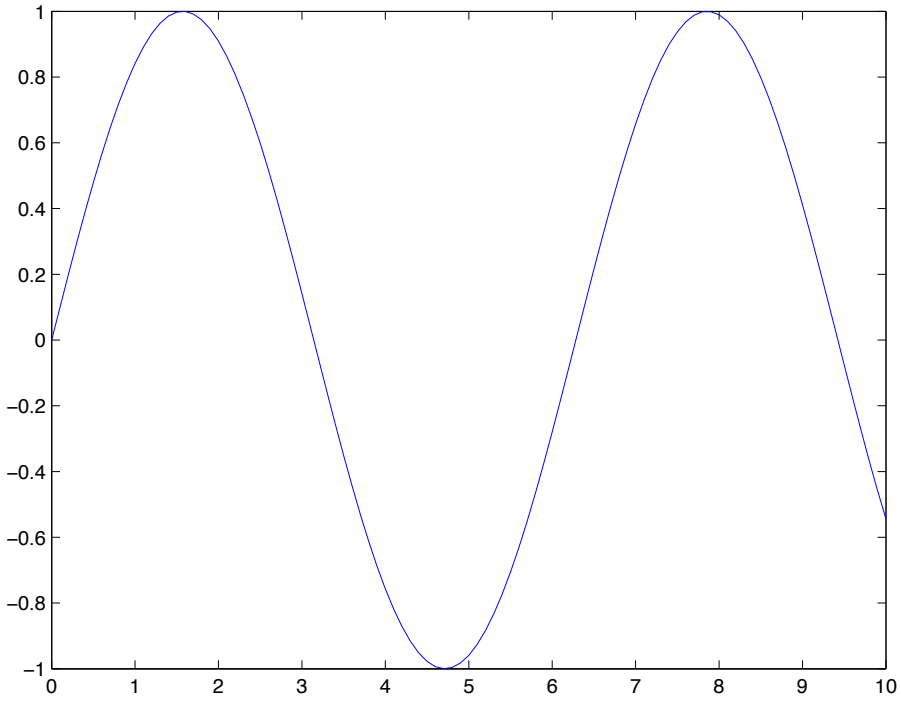
The function `length(a)` is equivalent to `max(size(a))`.

3 Graphical commands

In MATLAB is easy to plot a function or more with command `plot`, for example the command

```
0001 >> x = [0:0.1:10];
0002 plot(x, sin(x))
```

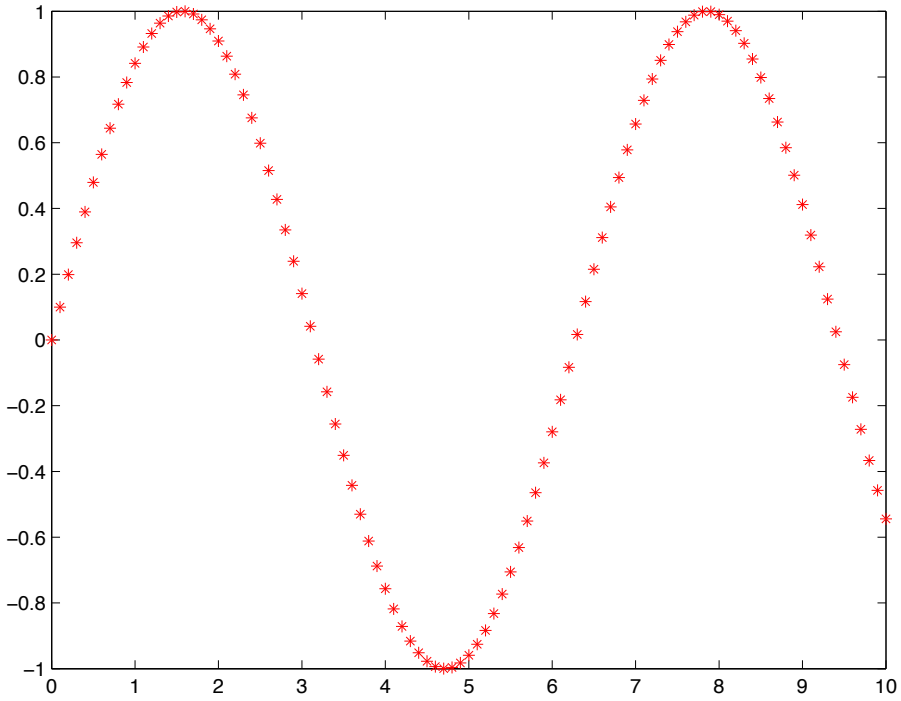
produce the following figure



while the command

```
0001 >> x = [0:0.1:10];
0002 plot(x,sin(x),'r*')
```

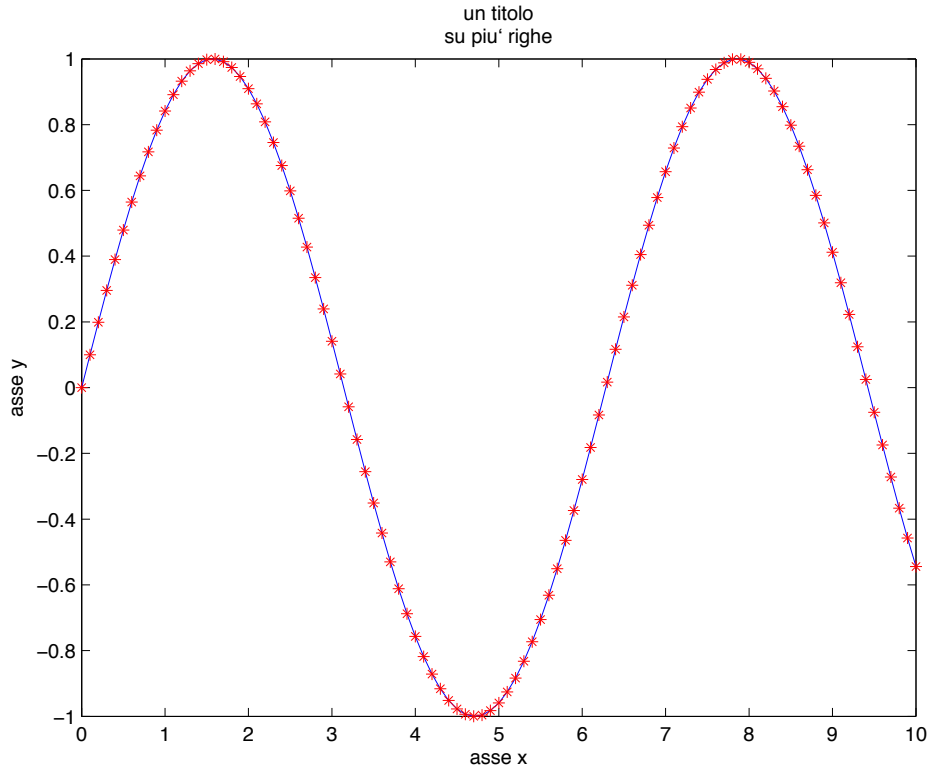
produce the figure



With the command `hold` we can merge many plots to build complex graphs. For example the commands

```
0001 >> plot(x, sin(x))
0002 >> hold on
0003 >> plot(x, sin(x),'r*')
0004 >> xlabel('asse x')
0005 >> ylabel('asse y')
0006 >> title({'un titolo','su piu‘ righe'})
0007 >> hold off
```

produce the figure



Graphics in MATLAB is complex and wide, to have some details of the many possibility type `doc plot` in teh command window.

4 More than matrix

In the construction of complex algorithms some time vector and matrix are not enough but some more complex data structures are needed. MATLAB has two interesting possibility, *cell array* and *struct*. A cell array can be thought as an heterogeneous vector which can contains any thing. The syntax is similar to the matrix syntax while round bracket are substituted by `{` and `}`. For example

```
0001 >> a = { [1 2 3], 'a string', 2, [ 1 2 ; 3 4] } % cell array example
0002 a =
0003     [1x3 double]    'a string'    [2]    [2x2 double]
0004
0005 >> disp('access to a{1}') ; a{1}
0006 access to a{1}
0007 ans =
0008      1     2     3
0009
0010 >> disp('access to a{2}') ; a{2}
0011 access to a{2}
0012 ans =
0013 a string
0014
0015 >> disp('access to a{3}') ; a{3}
```

```
0016 access to a{3}
0017 ans =
0018      2
0019
0020 >> disp('access to a{4}') ; a{4}
0021 access to a{4}
0022 ans =
0023      1     2
0024      3     4
```

The following example shows as to build complex data structures.

```
0001 % cell array example
0002 >> a = { [1 2 3], { 1 2 3 { 2 [ 1 2 ; 3 4] } } }
0003 a =
0004     [1x3 double]    {1x4 cell}
0005
0006 >> disp('access to a{1}') ; a{1}
0007 access to a{1}
0008 ans =
0009      1     2     3
0010
0011 >> disp('access to a{2}') ; a{2}
0012 access to a{2}
0013 ans =
0014     [1]    [2]    [3]    {1x2 cell}
0015
0016 >> disp('access to a{2}{1}') ; a{2}{1}
0017 access to a{2}{1}
0018 ans =
0019      1
0020
0021 >> disp('access to a{2}{2}') ; a{2}{2}
0022 access to a{2}{2}
0023 ans =
0024      2
0025
0026 >> disp('access to a{2}{3}') ; a{2}{3}
0027 access to a{2}{3}
0028 ans =
0029      3
0030
0031 >> disp('access to a{2}{4}') ; a{2}{4}
0032 access to a{2}{4}
0033 ans =
0034     [2]    [2x2 double]
0035
0036 >> disp('access to a{2}{4}{1}') ; a{2}{4}{1}
0037 access to a{2}{4}{1}
```

```
0038 ans =
0039      2
0040
0041 >> disp('access to a{2}{4}{2}') ; a{2}{4}{2}
0042 access to a{2}{4}{2}
0043 ans =
0044      1     2
0045      3     4
```

In the previous example notice the use of teh function `disp` which print a string to the terminal.

To access the elements of a cell array we use integer. A *struct* is similar to cell array in the sense that it permits to build complex data structures, but the access of the element is by the `,` operator and field names.. For example

```
0001 % struct example
0002 >> a . pippo    = [ 1 2 3 ] ; % the field pippo is a row vector
0003 >> a . pluto    = [ 1 ; 3 ] ; % the field pluto is a column vector
0004 >> a . paperino = { 'a' 'b' 3 } ; % the field paperino is a cell array
0005 >> a . subs . a = 1 ; % the field subs is a struct
0006 >> a . subs . b = 2 ; % the field subs is a struct
0007 >> a . subs . c = 3 ; % the field subs is a struct
0008
0009 >> disp(a)
0010        pippo: [1 2 3]
0011        pluto: [2x1 double]
0012     paperino: {'a'  'b'  [3]}
0013         subs: [1x1 struct]
0014
0015 >> disp(a.subs)
0016     a: 1
0017     b: 2
0018     c: 3
```

Matrix and struct can be combined. For example you can build matrix of structs or struct of matrices.

5 The find command

MATLAB support conditional instruction `if`–`else`–`end` and cycling instruction `while`–`end` e `for`–`end`. With these command procedural programming is possible. This is not the *rith* way to use MATLAB. If possible is better to build algorithm in vectorial way trying to split the algorithm as a sum of matrix block operations. The instruction `find` help a lot in this direction because vectorize conditional loops.

The instruction `find` can be used in two ways:

Mode 1

`find` return the row and column indices which satisfy the logical condition, for example

```
0001 >> A = [ 1 2 3 -4 ;
0002          0 1 0 -2 ;
0003          2 3 0 -1 ] ;
0004
0005 >> [I,J] = find( A > 0 )
0006 I =
0007      1
0008      3
0009      1
0010      2
0011      3
0012      1
0013 J =
0014      1
0015      1
0016      2
0017      2
0018      2
0019      3
0020
0021 >> [I,J] = find( A > 0 & A < 2 )
0022 I =
0023      1
0024      2
0025 J =
0026      1
0027      2
```

Mode 2

`find` return the vector of indices which satisfy the logical condition, for example

```
0001 >> A = [ 1 2 3 -4 ;
0002          0 1 0 -2 ;
0003          2 3 0 -1 ] ;
0004
0005 >> [IDX] = find( A > 0 )
0006 IDX =
0007      1
0008      3
0009      4
0010      5
0011      6
0012      7
0013
0014 >> [IDX] = find( A > 0 & A < 2 )
0015 IDX =
0016      1
0017      5
```

We can use `find` to perform task normally build with cycle. For example if we want to change all the negative components of a matrix with its squares:

```
0001 >> A = [ 1 2 3 -4 ;
0002          0 1 0 -2 ;
0003          2 3 0 -1 ]
0004
0005 >> IDX = find( A < 0 )
0006 IDX =
0007     10
0008     11
0009     12
0010
0011 >> A(IDX) = A(IDX) .^ 2
0012 A =
0013      1     2     3    16
0014      0     1     0     4
0015      2     3     0     1
```

6 Conditional instructions

The instruction `if–elseif–else–end`:

```
0001 >> a = 1 ;
0002 >> if a > 0
0003      disp('a>0')  ;
0004    else
0005      disp('a<=0')  ;
0006    end
0007 a>0
0008
0009 >> b = 0 ;
0010 >> if b > 0
0011      disp('b>0')  ;
0012    elseif b < 0
0013      disp('b<0')  ;
0014    else
0015      disp('b==0')  ;
0016    end
0017 b==0
```

The cycle `for–end`:

```
0001 >> x = [] ;
0002 >> for i=1:10
0003       x = [ x i ] ;
0004    end
0005 >> disp(x) ;
0006      1     2     3     4     5     6     7     8     9    10
```

```
0007
0008 >> x = [] ;
0009 >> for i=10:-1:1
0010      x = [ x i ] ;
0011    end
0012 >> disp(x) ;
0013     10     9     8     7     6     5     4     3     2     1
0014
0015 >> A = zeros(5,5) ;
0016 >> for i=1:5
0017      for j=1:5
0018        A(i,j) = (i-j)^2 + i*j ;
0019      end
0020    end
0021 >> disp(A) ;
0022      1     3     7    13    21
0023      3     4     7    12    19
0024      7     7     9    13    19
0025     13    12    13    16    21
0026     21    19    19    21    25
```

The cycle `while`–`end`:

```
0001 >> a = 129 ;
0002 >> b = 0 ;
0003 >> while a > 1
0004      a = floor(a/2) ;
0005      b = b + 1 ;
0006    end
0007 >> disp( b ) ;
0008      7
```

Every cycle `for` or `while` can be interrupted with the instruction `break` everywhere inside the cycle.

7 Comparison

In the conditional instruction we have already seen some conditional operators. MATLAB has the following conditional operators

- `eq`, `==` equality
- `ne`, `~=` inequality
- `lt`, `<` less than
- `gt`, `>` greater than
- `le`, `<=` less or equal than
- `ge`, `>=` greater or equal than

and logical operator

- `and`, `&` and
- `or`, `|` or
- `not`, `~` not
- `xor` exclusive or
- `any` true if some elements is not zero
- `all` true if all elements are not zero

8 Scripting

In MATLAB some functions can be written on external files that can be used in following as MATLAB commands. The file must be called `command.m` where `command` is the name of the function implemented. In the file `command.m` the definition of the function `command` is as follows

```
0001 function res = comando( x, y, x )
0002   ...
```

for example the file `dist.m` contains

```
0001 function res = dist( x, y )
0002   res = sqrt( sum((x-y).^2) ) ;
```

where the function `dist(x,y)` return the euclidean distance of two vectors, for example:

```
0001 >>  d = dist( [1 2 3], [ 0 1 -2]) ;
0002 >>  disp(d)
0003     5.1962
```

Some considerations

- Differently to other languages the block `function` not necessarily is terminated by`end`.
- The name of the function is **necessarily** the same as the file `.m`.
- It is possibile to define other functions inside the same script. This functions are *local* and visible only by the function inside the script.
- The arguments are passed by *values* (is not exact by values but it is close) hence is the arguments are modified inside the script they are not on the caller.

8.1 Returning more values

If you need more values returned by a script ypu can use `[ ]`. For example `trenorme.m`

```
0001 % evaluate varoious norm or a vector
0002 %
0003 %                                          ------
0004 %                  n                      /  n
0005 %                =====                   / =====
0006 %                \                      /  \
0007 %  max |x | ,     >    |x | ,         /    >    x
0008 %      | i|      /     | i|         /    /
0009 %                =====             /     =====
0010 %                i = 1           \/      i = 1
0011 %
0012 function [norma2,norma1,normainf] = trenorme( x )
0013   norma2   = sqrt(sum(x.^2)) ;
0014   norma1   = sum(abs(x)) ;
0015   normainf = max(abs(x)) ;
```

evaluate the 1, 2 and ∞ norm of a vector. The following command produce:

```
0001 >> [a,b,c] = trenorme([1 2 3])
0002 a =
0003     3.7417
0004 b =
0005      6
0006 c =
0007      3
```

8.2 Variable number of arguments

Another useful possibility of MATLAB is the variable number of arguments for input and output. For more information digit

```
0001 >> doc varargin
0002 >> doc varargout
```

8.3 A complex eample

```
0001 %
0002 % Solve the linear system A*x = b
0003 %
0004 function x = solvelinearsystem( A, b )
0005
0006   [nr,nc] = size(A) ;
0007
0008   % check dimension
0009   if nr ~= nc
0010     error( sprintf( 'the matrix A (%d x %d) is not square', nr,nc) ) ;
```

```
0011   end
0012
0013   neq = length(b) ;
0014
0015   if nr ~= neq
0016     error( sprintf( 'the matrix A (%d x %d) is not compatible with b, length(b) =
0017   end
0018
0019   M = [ A b ] ; % build augmented matrix
0020   for i=1:neq-1
0021     % find element with maximum module on i-th column
0022     [V,k] = max(abs(M(i:end,i))) ;
0023
0024     % set k to the right position
0025     k = k + i-1 ;
0026
0027     % swap i and k rows
0028     BF     = M(i,:) ;
0029     M(i,:) = M(k,:) ;
0030     M(k,:) = BF ;
0031
0032     % check singularity
0033     if  V == 0
0034       error( 'Singular Matrix' ) ;
0035     end
0036
0037     % build Frobenius matrix
0038     L             = eye(neq) ;
0039     L(i+1:end,i)  = -M(i+1:end,i)./M(i,i) ;
0040
0041     % set zero for some element of i-th column
0042     M             = L * M ;
0043
0044   end
0045
0046   % allocate for solution
0047   x = zeros(neq,1) ;
0048
0049   % matrix is now in triangular form
0050   % final operations
0051   x(neq) = M(neq,neq) / M(neq,neq+1) ;
0052   for i=neq-1:-1:1
0053     x(i) = ( M(i,neq+1) - M(i,i+1:end-1) * x(i+1:end) ) / M(i,i) ;
0054   end
```

Example of use

```
0001 >> A = [ 1 2 3 ; 1 2 4 ; 2 3 4 ] ;
0002 >> b = [ 1 2 3 ]
```

```
0003 >> x = solvelinearsystem(A,b)
0004 x =
0005      4
0006     -3
0007      1
```

References

[1] Timothy A. Davis and Kermit Sigmon. *MATLAB Primer, Seventh Edition*. Crc Press, 2005.

[2] Di Stephen Robert Otto and James P. Denier. *An Introduction to Programming and Numerical Methods in MATLAB*. Springer, 2005.