Introduzione a MATLAB

Alessandro Russo
Istituto di Analisi Numerica del CNR, Pavia
e-mail `russo@ian.pv.cnr.it`

Pavia, 22 febbraio 1999

MATLAB può essere definito come un ambiente di lavoro particolarmente adatto alle applicazioni numeriche. Una volta partito, si presenta in questo modo:

```
                    < M A T L A B (R) >
          (c) Copyright 1984-94 The MathWorks, Inc.
                    All Rights Reserved
                       Version 4.2c
                        Dec 31 1994

Commands to get started: intro, demo, help help
Commands for more information: help, whatsnew, info, subscribe

>>
```

La stringa `>>` è il prompt di MATLAB. MATLAB è un linguaggio *interpretato*: questo significa che legge un comando per volta e lo esegue immediatemante. I comandi possono essere dati da tastiera oppure letti da un file. Come esempio, definiamo una approssimazione di π e poi calcoliamo l'errore con il "vero" π.

```
>> pi_approx = 355/113;
>> diff = pi-pi_approx

diff =

  -2.6676e-07

>>
```

Da queste poche righe impariamo varie cose:

- le assegnazioni sono fatte come nei più comuni linguaggi di programmazione (C, Fortran, eccetera):

  ```
  <variabile> = <espressione>
  ```

- se non diciamo niente, le variabili vengono create come reali in doppia precisione (`double precision` in Fortran o `double` in C), che significa circa 17 cifre significative;
- un punto e virgola posto dopo l'assegnazione fa sì che il risultato NON venga stampato, mentre se il punto e virgola viene omesso MATLAB stampa immediatamente il risultato;
- MATLAB conosce già π (si chiama `pi`).

Le stesse istruzioni possono essere eseguite creando un file, diciamo `prova.m`, contenente le istruzioni

```
pi_approx = 355/113;
diff = pi-pi_approx
```

e poi eseguendo il file comandi `prova.m` semplicemente con il comando

```
>> prova

diff =

  -2.6676e-07

>>
```

I files contenenti comandi MATLAB si chiamano M-files.
Le cifre visualizzate sono 5, ma possiamo facilmente aumentarle con il comando `format`. Per vedere come funziona, chiediamolo a MATLAB stesso, usando il suo `help` di linea:

```
>> help format

 FORMAT Set output format.
        All computations in MATLAB are done in double precision.
        FORMAT may be used to switch between different output
        display formats as follows:
          FORMAT         Default. Same as SHORT.
          FORMAT SHORT   Scaled fixed point format with 5 digits.
          FORMAT LONG    Scaled fixed point format with 15 digits.
          FORMAT SHORT E Floating point format with 5 digits.
          FORMAT LONG E  Floating point format with 15 digits.
          FORMAT HEX     Hexadecimal format.
          FORMAT +       The symbols +, - and blank are printed
                         for positive, negative and zero elements.
                         Imaginary parts are ignored.
          FORMAT BANK    Fixed format for dollars and cents.
          FORMAT COMPACT Suppress extra line-feeds.
          FORMAT LOOSE   Puts the extra line-feeds back in.
```

```
          FORMAT RAT     Approximation by ratio of small integers.

>>
```

Il comando `help` è molto importante, perchè di solito è abbastanza dettagliato da permetterci di andare avanti senza leggere il Manuale. `help` senza argomenti produce la seguente lista:

```
>> help

HELP topics:

matlab/general       -  General purpose commands.
matlab/ops           -  Operators and special characters.
matlab/lang          -  Language constructs and debugging.
matlab/elmat         -  Elementary matrices and matrix manipulation.
matlab/specmat       -  Specialized matrices.
matlab/elfun         -  Elementary math functions.
matlab/specfun       -  Specialized math functions.
matlab/matfun        -  Matrix functions - numerical linear algebra.
matlab/datafun       -  Data analysis and Fourier transform functions.
matlab/polyfun       -  Polynomial and interpolation functions.
matlab/funfun        -  Function functions - nonlinear numerical methods.
matlab/sparfun       -  Sparse matrix functions.
matlab/plotxy        -  Two dimensional graphics.
matlab/plotxyz       -  Three dimensional graphics.
matlab/graphics      -  General purpose graphics functions.
matlab/color         -  Color control and lighting model functions.
matlab/sounds        -  Sound processing functions.
matlab/strfun        -  Character string functions.
matlab/iofun         -  Low-level file I/O functions.
matlab/demos         -  The MATLAB Expo and other demonstrations.
fuzzy/fuzzy          -  Fuzzy Logic Toolbox.
fuzzy/fuzdemos       -  Fuzzy Logic Toolbox Demos.
toolbox/ident        -  System Identification Toolbox.
toolbox/images       -  Image Processing Toolbox.
toolbox/local        -  Local function library.
nag/nag              -  NAG Foundation Toolbox - Numerical & Statistical Library
nag/examples         -  NAG Foundation Toolbox - Numerical & Statistical Library
toolbox/optim        -  Optimization Toolbox.
toolbox/signal       -  Signal Processing Toolbox.
toolbox/stats        -  Statistics Toolbox.
toolbox/symbolic     -  Symbolic Math Toolbox.
toolbox/uitools      -  User Interface Utilities.
russo/matlab         - (No table of contents file)

For more help on directory/topic, type "help topic".
```

```
>>
```

mentre `help general` dà

```
>> help matlab/general

 General purpose commands.
 MATLAB Toolbox  Version 4.2a 25-Jul-94

 Managing commands and functions.
   help        - On-line documentation.
   doc         - Load hypertext documentation.
   what        - Directory listing of M-, MAT- and MEX-files.
   type        - List M-file.
   lookfor     - Keyword search through the HELP entries.
   which       - Locate functions and files.
   demo        - Run demos.
   path        - Control MATLAB's search path.

 Managing variables and the workspace.
   who         - List current variables.
   whos        - List current variables, long form.
   load        - Retrieve variables from disk.
   save        - Save workspace variables to disk.
   clear       - Clear variables and functions from memory.
   pack        - Consolidate workspace memory.
   size        - Size of matrix.
   length      - Length of vector.
   disp        - Display matrix or text.

 Working with files and the operating system.
   cd          - Change current working directory.
   dir         - Directory listing.
   delete      - Delete file.
   getenv      - Get environment value.
   !           - Execute operating system command.
   unix        - Execute operating system command & return result.
   diary       - Save text of MATLAB session.

 Controlling the command window.
   cedit       - Set command line edit/recall facility parameters.
   clc         - Clear command window.
   home        - Send cursor home.
   format      - Set output format.
   echo        - Echo commands inside script files.
```

```
   more        - Control paged output in command window.

 Starting and quitting from MATLAB.
   quit        - Terminate MATLAB.
   startup     - M-file executed when MATLAB is invoked.
   matlabrc    - Master startup M-file.

 General information.
   info        - Information about MATLAB and The MathWorks, Inc.
   subscribe   - Become subscribing user of MATLAB.
   hostid      - MATLAB server host identification number.
   whatsnew    - Information about new features not yet documented.
   ver         - MATLAB, SIMULINK, and TOOLBOX version information.

>>
```

Torniamo a `format`. Per vedere più cifre, possiamo quindi usare il comando

```
>> format long
```

Se eseguiamo di nuovo il programma `prova.m`, avremo il seguente output:

```
>> prova

diff =

    -2.667641894049666e-07

>>
```

Il comando costituito soltanto dal nome di una variabile visualizza il contenuto di quella variabile:

```
>> diff

diff =

  -2.667641894049666e-07

>>
```

Ovviamente, come visto prima,

```
>> diff;
```

non produce alcun risultato.
Se scriviamo una operazione senza assegnarla ad una variabile, MATLAB crea per noi una variabile di nome `ans` (answer) e la stampa, come al solito, a seconda che l'istruzione sia stata terminata con il punto e virgola o no:

```
>> 2^16

ans =

       65536

>> 2^16;
>> ans

ans =

       65536

>>
```

Le variabili sono per ora globali, cioè visibili tutte contemporaneamente, dal prompt o da un M-file. Vedremo poi in seguito come definire le `function`, dove invece le variabili saranno locali.

In MATLAB tutte[1] le variabili sono reali in doppia precisione; non ci sono variabili intere e le stringhe sono memorizzate come un vettore contenente una sequenza di byte.

Per eliminare tutte le variabili dal workspace bisogna usare il comando `clear` senza argomenti. `clear <nomevar>` elimina invece solo la variabile `<nomevar>`:

```
>> clear diff
>> diff
??? Undefined function or variable diff.

>>
```

In MATLAB sono definite tutte le funzioni elementari standard (e anche di più). Per esempio:

```
>> x = sin(34)*cos(1)-log(9)^3.5-asin(0.1)-sqrt(89)

x =

  -24.97219466249224

>>
```

eccetera. MATLAB conosce inoltre varie costanti, tra cui π e i (l'unità immaginaria):

```
>> pi

ans =
```

[1] Questo non è più vero con la versione 5 di Matlab

```
   3.14159265358979

>> i

ans =

                0 + 1.00000000000000i

>>
```

Il numero e si può ottenere come `exp(1)`:

```
>> e = exp(1)

e =

   2.71828182845905

>>
```

Una variabile importante è `flops`, che contiene il numero di operazioni floating-point finora eseguite:

```
>> flops

ans =

    23

>>
```

Non molte per ora!

La caratteristica fondamentale di MATLAB che lo differenzia da (quasi) tutti gli altri linguaggi di programmazione è la sua capacità di lavorare con matrici e vettori in modo simbolico; non per nulla, MATLAB è un acronimo per MATrix LABoratory. MATLAB non distingue tra matrici e vettori: per lui i vettori sono semplicemente delle matrici con una dimensione uguale a 1. Pertanto è importante distinguere attentamente tra *vettori riga* e *vettori colonna*. Cominciamo col definire un vettore riga:

```
>> x = [1 2 3]

x =

     1     2     3

>>
```

Prima di andare avanti, vediamo quali variabili abbiamo definito nel workspace. Per fare questo, usiamo il comando `whos`:

```
>> whos
             Name        Size       Elements     Bytes    Density   Complex

              ans       1 by 1              1         8       Full      No
                e       1 by 1              1         8       Full      No
         pi_approx       1 by 1              1         8       Full      No
                x       1 by 3              3        24       Full      No

Grand total is 6 elements using 48 bytes

>>
```

Abbiamo 3 variabili reali e un vettore di 3 elementi che danno quindi 6 variabili reali in totale. Siccome le variabili reali usano 8 bytes l'una, abbiamo un totale di 48 bytes. I conti tornano! La dimensione del vettore `x` è `1 by 3`, ovvero 1 riga e 3 colonne. In altre parole, è un vettore riga lungo 3. Trasformiamolo in un vettore colonna (cioè calcoliamo il suo trasposto). In MATLAB si fa con un apice:

```
>> y = x'

y =

     1
     2
     3

>>
```

Moltiplichiamo `y` per lo scalare 3:

```
>> y = 3*y

y =

     3
     6
     9

>>
```

`y` è un vettore colonna, cioè una matrice `3 x 1`. Ha senso quindi fare la moltiplicazione (nel senso delle matrici) di `x` e `y` e il risultato dovrebbe essere una matrice `1 x 1`, cioè uno scalare. Vediamo se funziona:

```
>> x*y
```

```
ans =

    42

>>
```

Il risultato è il prodotto scalare tra `x` e `y`. Se moltiplichiamo `y` per `x`, il risultato è una matrice `3 x 3`:

```
>> y*x

ans =

     3     6     9
     6    12    18
     9    18    27

>>
```

La norma euclidea di un vettore si ottiene prendendo la radice quadrata del prodotto scalare del vettore dato con sè stesso; siccome `x` è un vettore riga, avremo

```
>> normax = sqrt(x*x')

normax =

   3.74165738677394

>>
```

mentre per il vettore colonna `y`

```
>> normay = sqrt(y'*y)

normay =

  11.22497216032182

>>
```

Una matrice `3 x 2` può essere introdotta nel seguente modo:

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

A =

     1     2
     3     4
     5     6

>>
```

Trasponiamo `A` e avremo una matrice `2 x 3`:

```
>> A = A'

A =

     1     3     5
     2     4     6

>>
```

Possiamo adesso eseguire la moltiplicazione di `A`, matrice `2 x 3`, con `y`, vettore colonna lungo 3:

```
>> A * y

ans =

    66
    84

>>
```

Otteniamo un vettore colonna lungo 2. Nella definizione della matrice `A` abbiamo usato il punto e virgola per cambiare riga. Lo stesso effetto può essere ottenuto anche nel seguente modo, più pittoresco:

```
>> A = [ 1 2
         3 4
         5 6 ]

A =

     1     2
     3     4
     5     6

>>
```

Questo modo di rappresentazione è estremamente utile quando si costruiscono le matrici a blocchi. Costruiamo come esempio una matrice `2 x 3` (2 righe e 3 colonne) avente come riga 1 il vettore `x` e come riga 2 il vettore `5*x`:

```
>> B = [  x
         5*x ]

B =

     1     2     3
```

```
     5    10    15

>>
```

Basta quindi giustapporre i blocchi come si farebbe con carta e penna.
MATLAB ha diversi meccanismi per generare matrici e vettori. Uno è il seguente:

```
>> a = 1:10

a =

     1     2     3     4     5     6     7     8     9    10

>>
```

MATLAB crea un vettore con i dentro i numeri consecutivi da 1 a 10. L'incremento di default è 1, ma possiamo facilmente cambiarlo:

```
>> z=1:2:10

z =

     1     3     5     7     9

>>
```

L'incremento (e ovviamente gli estremi) possono essere numeri reali qualunque.
Le funzioni elementari applicate a vettori (o matrici) generalmente restituiscono un vettore (o una matrice) di dimensione uguale a quella di partenza con la funzione applicata ai singoli elementi:

```
>> sin(a)

ans =

  Columns 1 through 4

   0.84147098480790   0.90929742682568   0.14112000805987  -0.75680249530793

  Columns 5 through 8

  -0.95892427466314  -0.27941549819893   0.65698659871879   0.98935824662338

  Columns 9 through 10

   0.41211848524176  -0.54402111088937

>>
```

Per inciso, questo è quello che succede quando ci sono troppe colonne e non stanno nel display. Vediamo ora il modo più semplice di rappresentare dei dati graficamente con MATLAB. Cerchiamo di disegnare `sin(x)` nell'intervallo $[0, 10\pi]$. La funzione più semplice per disegnare è `plot`. Cominciamo con

```
>> help plot

 PLOT   Plot vectors or matrices.
        PLOT(X,Y) plots vector X versus vector Y. If X or Y is a matrix,
        then the vector is plotted versus the rows or columns of the matrix,
        whichever line up.

        PLOT(Y) plots the columns of Y versus their index.
        If Y is complex, PLOT(Y) is equivalent to PLOT(real(Y),imag(Y)).
        In all other uses of PLOT, the imaginary part is ignored.

        Various line types, plot symbols and colors may be obtained with
        PLOT(X,Y,S) where S is a 1, 2 or 3 character string made from
        the following characters:

              y     yellow         .     point
              m     magenta        o     circle
              c     cyan           x     x-mark
              r     red            +     plus
              g     green          -     solid
              b     blue           *     star
              w     white          :     dotted
              k     black          -.    dashdot
                                   --    dashed

        For example, PLOT(X,Y,'c+') plots a cyan plus at each data point.

        PLOT(X1,Y1,S1,X2,Y2,S2,X3,Y3,S3,...) combines the plots defined by
        the (X,Y,S) triples, where the X's and Y's are vectors or matrices
        and the S's are strings.

        For example, PLOT(X,Y,'y-',X,Y,'go') plots the data twice, with a
        solid yellow line interpolating green circles at the data points.

        The PLOT command, if no color is specified, makes automatic use of
        the colors specified by the axes ColorOrder property.  The default
        ColorOrder is listed in the table above for color systems where the
        default is yellow for one line, and for multiple lines, to cycle
        through the first six colors in the table.  For monochrome systems,
        PLOT cycles over the axes LineStyleOrder property.

        PLOT returns a column vector of handles to LINE objects, one
```

```
        handle per line.

        The X,Y pairs, or X,Y,S triples, can be followed by
        parameter/value pairs to specify additional properties
        of the lines.

        See also SEMILOGX, SEMILOGY, LOGLOG, GRID, CLF, CLC, TITLE,
        XLABEL, YLABEL, AXIS, AXES, HOLD, and SUBPLOT.

>>
```

Vediamo quindi che dobbiamo procurarci 2 vettori `x` e `y`, uno con le ascisse e l'altro con le ordinate (non importa che siamo vettori riga o vettori colonna), e poi usare `plot(x,y)`. Decidiamo di suddividere l'intervallo in 100 parti uguali.

```
>> x = 0:10*pi/100:10*pi;
>> y = sin(x);
>> plot(x,y)
```

Il risultato è in Figura 1. Avremmo ottenuto lo stesso risultato con il comando

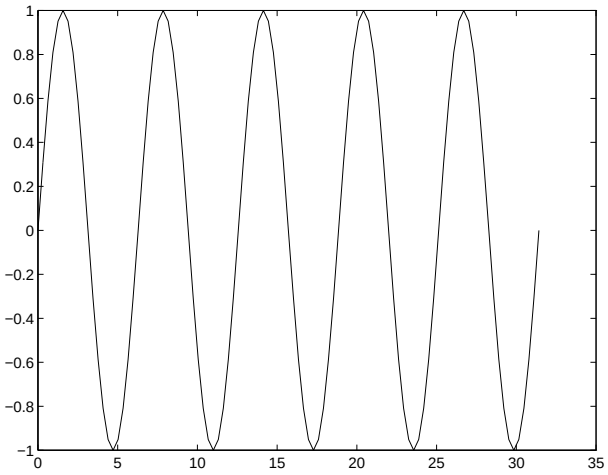

Figura 1: `sin(x)` nell'intervallo $[0, 10\pi]$

```
>> plot(x,sin(x))
```

senza definire `y`.

Prima di proseguire con le matrici, vediamo come funzionano i cicli e le istruzioni condizionali. Cominciamo con l'istruzione `for ... end`.

```
>> for i=1:3, i, end

i =
```

```
     1


i =

     2


i =

     3

>>
```

Qui le virgole sono usate per separare più istruzioni sulla stessa riga. Avremmo potuto mettere ";", ma in questo modo non avremmo visto nulla:

```
>> for i=1:3; i; end
>>
```

Notare che il range della variabile di ciclo non è nient'altro che un vettore: possiamo infatti avere per esempio

```
>> for i=0:pi/5:1, i , end

i =

     0


i =

   0.62831853071796

>>
```

e così via. In generale:

```
>> help for

 FOR    Repeat statements a specific number of times.
        The general form of a FOR statement is:

           FOR variable = expr, statement, ..., statement END

        The columns of the expression are stored one at a time in
        the variable and then the following statements, up to the
        END, are executed. The expression is often of the form X:Y,
```

```
        in which case its columns are simply scalars. Some examples
        (assume N has already been assigned a value).

             FOR I = 1:N,
                 FOR J = 1:N,
                     A(I,J) = 1/(I+J-1);
                 END
             END

        FOR S = 1.0: -0.1: 0.0, END steps S with increments of -0.1
        FOR E = EYE(N), ... END  sets E to the unit N-vectors.

>>
```

Notare: `The columns of the expression are stored one at a time in the variable and then the following statements, up to the END, are executed.` Questo vuol dire che se vogliamo un ciclo usuale, dobbiamo definire il range del contatore in un vettore *riga*. Se usiamo il vettore colonna `(1:10)'`, il ciclo viene ripetuto *una sola volta* e l'indice `i` prende solo il valore del vettore `(1:10)'`:

```
>> for i=(1:10)', i, end

i =

     1
     2
     3
     4
     5
     6
     7
     8
     9
    10

>>
```

Per avere esempi più interessanti dobbiamo scrivere dei file comandi. Per esempio scriviamo un M-file `expo.m` che calcola i primi 5 termini della serie di Taylor centrata in 0 per e^x, e confronta il polinomio ottenuto con la funzione vera. Campioniamo i due grafici in 101 punti sull'intervallo [-5,5]. Il programma è il seguente:

```
% cancelliamo tutte le variabili
clear

% estremi dell'intervallo
a = -5;
b = 5;
```

```
% numero di suddivisioni (100 suddivisioni = 101 campionamenti)
ns = 100;

% termini della serie di Taylor
nt = 5;

% campionamenti;
% ascisse:
xc = a:(b-a)/ns:b;
% ordinate
yc = zeros(1,ns+1);

% La funzione zeros(n,m) crea una matrice di zeri con n righe e m colonne;
% i percento % servono per i commenti;
% si possono lasciare righe bianche.

j = 0;

for x = a:(b-a)/ns:b

  j = j+1;
  expval = 1;
  xi = 1;
  ifatt = 1;    % i commenti si possono anche mettere a fianco delle istruzioni!

  for i=1:nt

    xi = xi*x;
    ifatt = ifatt*i;

    expval = expval + xi/ifatt;

  end

  yc(j) = expval;

end

plot(xc,yc,'-',xc,exp(xc),':')

% la stringa '-' nel comando plot disegna la curva a tratto pieno,
% mentre ':' la disegna a puntini. Ci sono altre opzioni, utili
% quando ci sono molte curve.
```

Il grafico che si ottiene è contenuto in Figura 2; la curva "vera" di e^x è quella a puntini. Vediamo ora l'istruzione `if ... elseif ... else ... end`. È meglio chiedere a

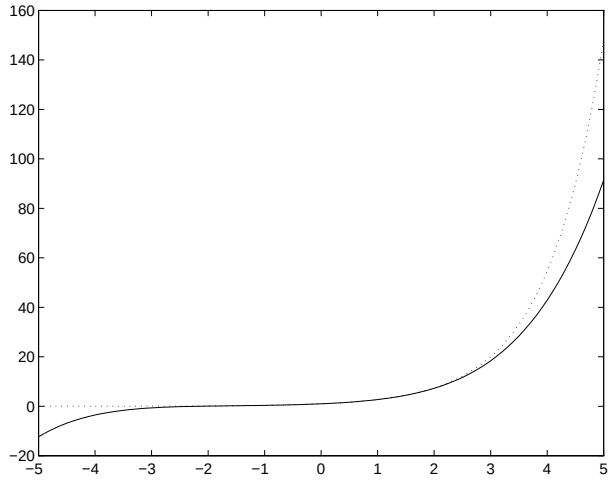

Figura 2: Confronto tra `exp(x)` (puntini) e il suo polinomio di Taylor nello zero di grado 5 (tratto pieno).

MATLAB:

```
>> help if

 IF     Conditionally execute statements.
        The general form of an IF statement is:
                IF variable, statements, END
        The statements are executed if the real part of the variable
        has all non-zero elements. The variable is usually the result of
        expr rop expr where rop is ==, <, >, <=, >=, or ~=.
        For example:

               IF I == J
                  A(I,J) = 2;
               ELSEIF ABS(I-J) == 1
                  A(I,J) = -1;
               ELSE
                  A(I,J) = 0;
               END

>>
```

Credo che questa spiegazione sia più che sufficiente. L'unica cosa da notare è che gli operatori condizionali sono

```
==      uguale
<       minore
<=      minore uguale
~=      diverso
```

In MATLAB non ci sono variabili logiche. Falso è rappresentato del numero 0, e vero dal numero 1 (o meglio, da qualunque numero diverso da 0). Quindi `52==78` è il numero 0:

```
>> 52==78

ans =

     0

>>
```

Un'altra istruzione di tipo condizionale è `while`:

```
>> help while

 WHILE  Repeat statements an indefinite number of times.
        The general form of a WHILE statement is:

           WHILE variable, statement, ..., statement, END

        The statements are executed while the variable has all
        non-zero elements. The variable is usually the result of
        expr rop expr where rop is ==, <, >, <=, >=, or ~=.
        For example (assuming A already defined):

                E = 0*A; F = E + EYE(E); N = 1;
                WHILE NORM(E+F-E,1) > 0,
                   E = E + F;
                   F = A*F/N;
                   N = N + 1;
                END

>>
```

Per esempio, il programma

```
i = 113;

while i>1
  i = 1/3;
end

i
```

produce

```
>> i
```

```
i =

   0.33333333333333

>>
```

Torniamo ora alle matrici. Questa è una lista dei possibili comandi relativi alle matrici (output di `help elmat` e `help matfun`):

```
 Elementary matrices and matrix manipulation.

 Elementary matrices.
   zeros       - Zeros matrix.
   ones        - Ones matrix.
   eye         - Identity matrix.
   rand        - Uniformly distributed random numbers.
   randn       - Normally distributed random numbers.
   linspace    - Linearly spaced vector.
   logspace    - Logarithmically spaced vector.
   meshgrid    - X and Y arrays for 3-D plots.
   :           - Regularly spaced vector.

 Matrix manipulation.
   diag        - Create or extract diagonals.
   fliplr      - Flip matrix in the left/right direction.
   flipud      - Flip matrix in the up/down direction.
   reshape     - Change size.
   rot90       - Rotate matrix 90 degrees.
   tril        - Extract lower triangular part.
   triu        - Extract upper triangular part.
   :           - Index into matrix, rearrange matrix.

 Matrix functions - numerical linear algebra.

 Matrix analysis.
   cond        - Matrix condition number.
   norm        - Matrix or vector norm.
   rcond       - LINPACK reciprocal condition estimator.
   rank        - Number of linearly independent rows or columns.
   det         - Determinant.
   trace       - Sum of diagonal elements.
   null        - Null space.
   orth        - Orthogonalization.
   rref        - Reduced row echelon form.

 Linear equations.
   \ and /     - Linear equation solution; use "help slash".
```

```
   chol        - Cholesky factorization.
   lu          - Factors from Gaussian elimination.
   inv         - Matrix inverse.
   qr          - Orthogonal-triangular decomposition.
   qrdelete    - Delete a column from the QR factorization.
   qrinsert    - Insert a column in the QR factorization.
   nnls        - Non-negative least-squares.
   pinv        - Pseudoinverse.
   lscov       - Least squares in the presence of known covariance.

 Eigenvalues and singular values.
   eig         - Eigenvalues and eigenvectors.
   poly        - Characteristic polynomial.
   polyeig     - Polynomial eigenvalue problem.
   hess        - Hessenberg form.
   qz          - Generalized eigenvalues.
   rsf2csf     - Real block diagonal form to complex diagonal form.
   cdf2rdf     - Complex diagonal form to real block diagonal form.
   schur       - Schur decomposition.
   balance     - Diagonal scaling to improve eigenvalue accuracy.
   svd         - Singular value decomposition.

 Matrix functions.
   expm        - Matrix exponential.
   expm1       - M-file implementation of expm.
   expm2       - Matrix exponential via Taylor series.
   expm3       - Matrix exponential via eigenvalues and eigenvectors.
   logm        - Matrix logarithm.
   sqrtm       - Matrix square root.
   funm        - Evaluate general matrix function.
```

Per esempio, definiamo una matrice quadrata e estraiamone la sua triangolare inferiore con il comando `tril`. Usiamo il comando `rand(n)` che genera una matrice di numeri casuali.

```
>> format short
>> A = rand(4)

A =

    0.5717    0.4985    0.8907    0.2128
    0.8024    0.9554    0.6248    0.7147
    0.0331    0.7483    0.8420    0.1304
    0.5344    0.5546    0.1598    0.0910

>> tril(A)
```

```
ans =

    0.5717         0         0         0
    0.8024    0.9554         0         0
    0.0331    0.7483    0.8420         0
    0.5344    0.5546    0.1598    0.0910

>>
```

Possiamo anche calcolare l'inversa di `A`:

```
>> inv(A)

ans =

    1.1718   -0.2578   -1.2536    1.0824
   -1.4097    0.0202    1.2101    1.4026
    1.2939   -0.3068    0.2244   -0.9372
   -0.5624    1.9298   -0.4063   -2.2707

>>
```

e gli autovalori:

```
>> eig(A)

ans =

   2.2248
   0.2644 + 0.4260i
   0.2644 - 0.4260i
  -0.2935

>>
```

In questo caso gli autovalori sono numeri complessi, ma MATLAB è capace di lavorare anche in $\mathbb{C}$. Se `b` è un vettore colonna, possiamo facilmente calcolare la soluzione del sistema `Ax = b`. Definiamo `b` come il vettore colonna di componenti 1, 2, 3, 4:

```
>> b = [1 2 3 4]';
```

Un modo di risolvere `Ax = b` è ovviamente usare l'inversa di `A` in modo esplicito:

```
>> x = inv(A)*b

x =

    1.2249
```

```
    7.8716
   -2.3955
   -7.0045

>>
```

Ma un modo migliore è usare l'eliminazione di Gauss (con pivot parziale) che si può ottenere con una "moltiplicazione a sinistra":

```
>> x = A\b

x =

    1.2249
    7.8716
   -2.3955
   -7.0045

>>
```

Quest'ultimo metodo non calcola l'inversa della matrice `A`, ma usa la decomposizione `LU` di `A`, come spiegato in `help slash`:

```
>> help slash

  \   Backslash or left division.
      A\B is the matrix division of A into B, which is roughly the
      same as INV(A)*B , except it is computed in a different way.
      If A is an N-by-N matrix and B is a column vector with N
      components, or a matrix with several such columns, then
      X = A\B is the solution to the equation A*X = B computed by
      Gaussian elimination. A warning message is printed if A is
      badly scaled or nearly singular.  A\EYE(SIZE(A)) produces the
      inverse of A.
      If A is an M-by-N matrix with M < or > N and B is a column
      vector with M components, or a matrix with several such columns,
      then X = A\B is the solution in the least squares sense to the
      under- or overdetermined system of equations A*X = B. The
      effective rank, K, of A is determined from the QR decomposition
      with pivoting. A solution X is computed which has at most K
      nonzero components per column. If K < N this will usually not
      be the same solution as PINV(A)*B.  A\EYE(SIZE(A)) produces a
      generalized inverse of A.

  /   Slash or right division.
      B/A is the matrix division of A into B, which is roughly the
      same as B*INV(A) , except it is computed in a different way.
      More precisely, B/A = (A'\B')'. See \.
```

```
  ./  Array division.
      B./A denotes element-by-element division.  A and B
      must have the same dimensions unless one is a scalar.
      A scalar can be divided with anything.

>>
```

La decomposizione `LU` di `A` si può ottenere esplicitamente con

```
>> help lu

 LU     Factors from Gaussian elimination.
        [L,U] = LU(X) stores a upper triangular matrix in U and a
        "psychologically lower triangular matrix", i.e. a product
        of lower triangular and permutation matrices, in L , so
        that X = L*U.

        [L,U,P] = LU(X) returns lower triangular matrix L, upper
        triangular matrix U, and permutation matrix P so that
        P*X = L*U.

        By itself, LU(X) returns the output from LINPACK'S ZGEFA routine.

>>
```

Cioè:

```
>> [L,U,P] = lu(A)

L =

    1.0000         0         0         0
    0.0412    1.0000         0         0
    0.7124   -0.2569    1.0000         0
    0.6661   -0.1153   -0.2477    1.0000


U =

    0.8024    0.9554    0.6248    0.7147
         0    0.7089    0.8163    0.1010
         0         0    0.6553   -0.2705
         0         0         0   -0.4404


P =
```

```
     0     1     0     0
     0     0     1     0
     1     0     0     0
     0     0     0     1

>> P*A-L*U

ans =

   1.0e-15 *

         0         0         0         0
         0         0         0         0
    0.1110         0         0         0
         0         0         0   -0.0416

>>
```

Il risultato finale non è esattamente zero perchè c'è un piccolo errore di arrotondamento. Anche la matrice inversa viene calcolata da MATLAB usando la decomposizione `LU`.

Con il comando `chol` è possibile calcolare la decomposizione di Cholesky di una matrice simmetrica e definita positiva `A`. Se `R = chol(A)`, allora `A = R'*R`. Possiamo procurarci una matrice simmetrica e definita positiva costruendo una matrice a caso `B` e poi formando il prodotto `B'*B`:

```
>> B=rand(4);
>> A=B'*B;
>> R=chol(A)

R =

    1.5493    0.8885    1.2580    0.8435
         0    0.3121   -0.1823   -0.5733
         0         0    0.5953   -0.0774
         0         0         0    0.0766

>> R'*R - A

ans =

   1.0e-15 *

   -0.4441         0         0         0
         0         0         0         0
         0         0         0         0
         0         0         0         0
```

```
>>
```

Anche in questo caso c'è un piccolo errore di arrotondamento.
In generale è utile avere un modo di misurare il tempo impiegato per l'esecuzione di un dato algoritmo. In MATLAB questo si può fare con i comandi `tic` e `toc`.

```
>> help tic

 TIC    Start a stopwatch timer.
        The sequence of commands
            TIC
            any stuff
            TOC
        prints the time required for the stuff.

        See also TOC, CLOCK, ETIME, CPUTIME.

>>
```

Quindi

```
>> A = rand(100);
>> tic;inv(A);toc

elapsed_time =

    0.1462

>>
```

è il tempo impiegato (in secondi) a invertire una matrice `100 x 100`. Il tempo impiegato per una matrice `200 x 200` è

```
>> A = rand(200);
>> tic;inv(A);toc

elapsed_time =

    1.0958
```

Il rapporto tra i due tempi è

```
>> 1.0958/0.1462 = 7.4952
```

Infatti, siccome l'algoritmo usato è la decomposizione `LU`, ci dovevamo aspettare circa un fattore `8 = 2^3`. Per il calcolo del numero di operazioni impiegate, possiamo fare nel seguente modo:

```
>> flops0 = flops;               % flops iniziali
>> A = rand(100);                % matrice 100 x 100 random
>> inv(A);                       % calcola l'inversa
>> flops1 = flops-flops0;        % flops impiegati
>> A = rand(200);                % matrice 200 x 200 random
>> inv(A);                       % calcola l'inversa
>> flops2 = flops-flops1-flops0; % flops impiegati
>> flops2

flops2 =

    16157493

>> flops1

flops1 =

     2039808

>> flops2/flops1                 % rapporto tra i flops

ans =

    7.9211

>>
```

Anche qui abbiamo un fattore pari circa a 8.

Vediamo qualcosa sulla precisione di MATLAB. MATLAB implementa le specifiche del sistema floating point IEEE 754 (solo in doppia precisione). Abbiamo quindi base 2, 53 bits di mantissa e 11 di esponente. I bit di mantissa realmente rappresentati sono 52, in quanto si usa la proprietà che il primo bit significativo è sicuramente 1. Avanza quindi un bit che serve per il segno. Il range degli esponenti varia tra -1021 e 1024, corrispondenti ai numeri $10^{\pm 308}$. La *roundoff unit* u è quindi $u = \frac{1}{2}2^{-52} = 2^{-53}$, corrispondente alla metà della distanza tra 1 e il numero successivo. MATLAB ha predefinita una variabile `eps` che è *il doppio* della roundoff unit u.

```
>> eps

eps =

     2.220446049250313e-16

>> 2^(-53)

ans =
```

```
     1.110223024625157e-16

>>
```

`eps` di MATLAB è quindi pari alla distanza tra 1 e il numero successivo, cioè lo *spacing* tra 1 e 2. Definiamo come `u` la roundoff unit:

```
>> u = 2^(-53)

u =

     1.110223024625157e-16

>>
```

Studiamo il modo di arrotondare. Lo standard IEEE 754 è "rounding to even", cioè in caso di parità scegliere il numero che ha l'ultima cifra pari (cioè 0 dal momento che siamo in base 2). Quindi `1 + u` deve dare ancora 1:

```
>> (1+u) - 1

ans =

     0

>>
```

mentre `(1+eps) + u` deve dare `1 + 2*eps`:

```
>> ((1+eps) + u) - 1

ans =

     4.440892098500626e-16

>>
```

Dalla teoria sappiamo che oltre 2^{52} ci dovrebbero essere solo i numeri interi. Verifichiamolo.

```
>> x = 2^52;
>> sprintf('%16.4f',x)

ans =

4503599627370496.0000

>> sprintf('%16.4f',x+0.4)
```

```
ans =

4503599627370496.0000

>> sprintf('%16.4f',x+0.8)

ans =

4503599627370497.0000

>>
```

Il comando `sprintf('%16.4f',x)` stampa il numero `x` con 16 cifre intere e 4 decimali.

Il seguente programma confronta e^x con il valore ottenuto sommando la serie di Taylor nello 0. Il numero di termini considerati nella serie di Taylor è determinato nel modo seguente: il primo termine trascurato è quello che *non cambia più* la somma.

```
clear

% estremi dell'intervallo

a = 0;
b = +100;

% passo di discretizzazione

step = 1;

x = a:step:b;

n = size(x,2);
y = zeros(1,n);

for i=1:n

  % serie di taylor

  oldy = 0;
  newy = 1;
  add = 1;
  j = 0;

  while oldy ~= newy    % proseguo sommando termini
                        % finche' la somma non cambia
    j = j+1;

    oldy = newy;
```

```
    add = add/j * x(i); % in add c'e' il termine da sommare
    newy = oldy + add;

  end

  y(i) = newy;

end

yex = exp(x);

er = abs(y-yex)./yex;   % calcolo l'errore relativo assumendo che
                        % matlab calcoli exp(x) correttamente

semilogy(x,er,'+');     % grafico in scala semilogaritmica dell'errore rel.
```

Per $x > 0$ abbiamo una somma di termini tutti positivi, quindi ci aspettiamo che l'errore di tipo floating point introdotto sia piccolo. Viceversa, se $x < 0$ i termini sono di segno alterno, e per di più il risultato è piccolo mentre gli addendi in valore assoluto possono diventare molto grandi. Ci aspettiamo dunque un grande errore di tipo cancellazione. Vediamo i risultati. In particolare, si vede che intorno a $x = -20$ l'errore relativo è di circa il 100%.

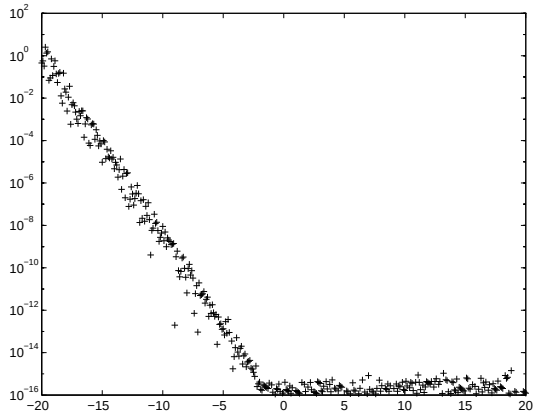

Figura 3: Intervallo $[-20, +20]$, passo 0.1

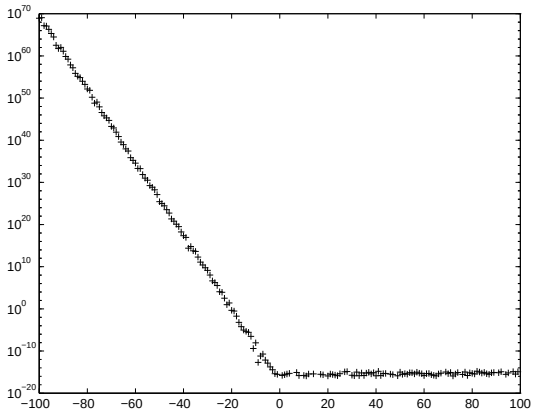

Figura 4: Intervallo $[-100, +100]$, passo 0.1

Vediamo ora come utilizzare le functions in MATLAB. Scriviamo una function che rappresenti $1/(1 + x^2)$ e che sia utilizzabile come le funzioni intrinseche di MATLAB. Creiamo un file `hill.m` contenente le righe

```
function y=hill(x)

y=1/(1+x^2);

return;
```

Possiamo adesso usare `hill` come se fosse una funzione intrinseca tipo `sin` o `cos`:

```
>> hill(5)

ans =

    0.0385

>> hill(0)

ans =

     1

>> hill(i)

Warning: Divide by zero

ans =

   Inf

>>
```

Le variabili `x` e `y` utilizzate all'interno della function `hill` sono *locali* alla function: in altre parole, se esistono fuori dalla function variabili con lo stesso nome, non verranno modificate. Infatti:

```
>> x=2;
>> hill(3)

ans =

    0.1000

>> x

x =

     2

>>
```

Il tipo degli argomenti di una function non è specificato nella function stessa, ma viene determinato al momento dell'esecuzione. Con una piccola modifica, possiamo fare in modo che la nostra function `hill` accetti anche argomenti vettoriali:

```
function y=hill(x)
```

```
y=1./(1+x.^2);

return;
```

In questo modo, la variabile in ingresso può essere un array e l'output sarà definito di conseguenza:

```
>> hill([1 2 3])

ans =

    0.5000    0.2000    0.1000

>>
```

Possiamo facilmente tracciare un grafico della nostra function con i comandi:

```
>> x=-5:0.01:5;
>> plot(x,hill(x))
```

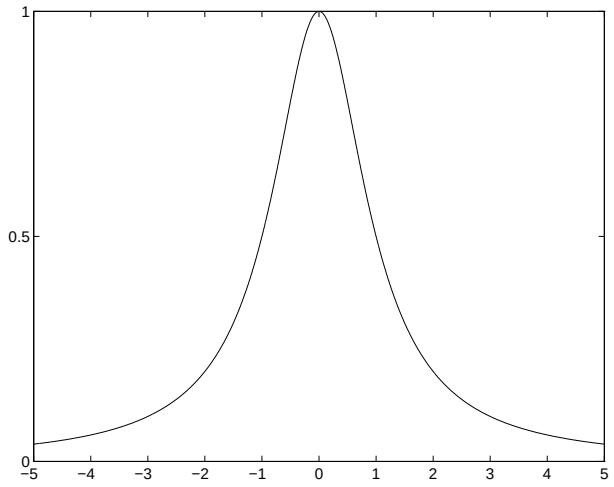

Figura 5: `hill(x)` nell'intervallo $[-5, 5]$

Se vogliamo parametrizzare la nostra function, ma senza modificare le variabili di input, dobbiamo fare in modo che il valore di un parametro `P` sia visibile contemporaneamente dal prompt di MATLAB e dalla function HILL. Per fare questo, bisogna che entrambi gli ambienti dichiarino `P global`:

```
>> help global

 GLOBAL Define global variables.
        GLOBAL X Y Z defines X, Y, and Z as global in scope.
```

```
        Ordinarily, each MATLAB function, defined by an M-file, has its
        own local variables, which are separate from those of other functions,
        and from those of the base workspace and non-function scripts.
        However, if several functions, and possibly the base workspace,
        all declare a particular name as GLOBAL, then they all share
        a single copy of that variable.  Any assignment to that variable,
        in any function, is available to all the other functions declaring
        it GLOBAL.

        Stylistically, global variables often have long names with all
        capital letters, but this is not required.

        For an example, see the functions TIC and TOC.

        See also ISGLOBAL, CLEAR, WHO.

>>
```

Modificheremo quindi `hill.m` nel modo seguente:

```
function y=hill(x)

global P

y=1./(1+(x-P).^2);

return;
```

La nuova funzione è la vecchia *traslata* di `P`.

```
>> global P
>> P=2;
>> plot(x,hill(x))

>> P=-1;
>> plot(x,hill(x))
```

Se inseriamo una o più righe di commento subito dopo l'inizio della function queste vengono stampate sul video con la funzione `help`:

```
function y=hill(x)

%La funzione hill traslata di P (da dichiarare global)

global P

y=1./(1+(x-P).^2);

return;
```

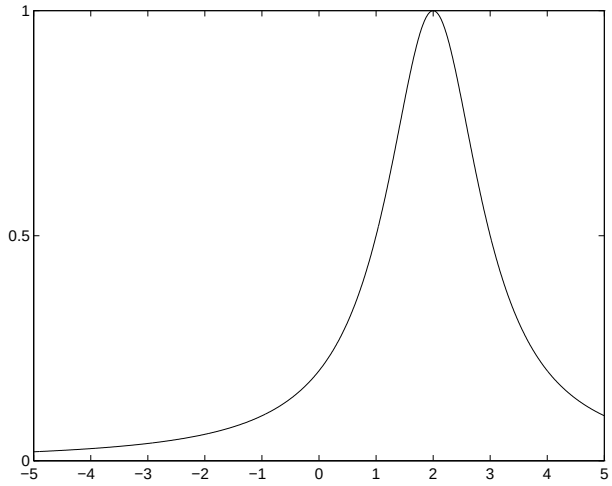

Figura 6: `hill(x)` traslata di 2 nell'intervallo $[-5, 5]$

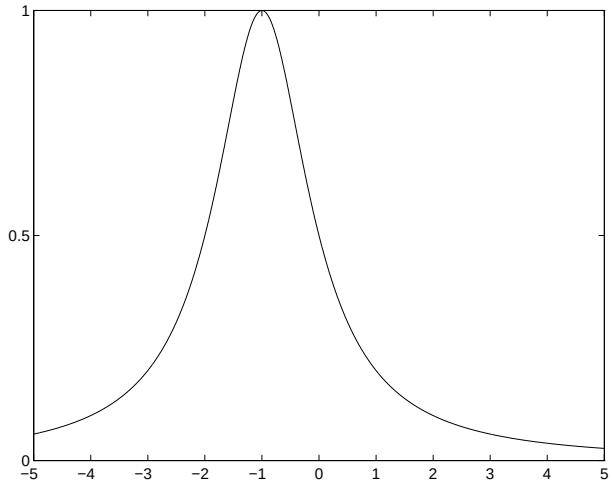

Figura 7: `hill(x)` traslata di -1 nell'intervallo $[-5, 5]$

ed ecco l'output di `help hill`:

```
>> help hill

 La funzione hill traslata di P (da dichiarare global)

>>
```

Ecco alcune note generali sull'uso delle functions:

- l'istruzione `return` alla fine non è necesssaria;
- nella prima riga della function `hill`, non è obbligatorio che compaia il nome "hill"; il nome della function è determinato solo dal nome dell'M-file;

- gli argomenti sono passati alla function per valore, ossia non è possibile modificarli da dentro la function stessa. Tuttavia, per risparmiare memoria, ne viene fatta una copia *solo se* la function tenta di modificarli. Questo è essenziale per le function cha hanno come argomento grossi array: a patto di non modificarli, la function userà la locazione di memoria originale.

- una function si può usare con un numero di argomenti minore di quello che compare nella sua sintassi. È possibile sapere il numero di argomenti effettivamente passati con la variable `nargin`, attiva dentro la function.