Comandi essenziali di MATLAB

Enrico Bertolazzi

Lezione del 4 marzo 2008

Indice

1 Comandi base

MATLAB può essere usato come una calcolatrice:

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

Le operazioni standard `+`, `-`, `*`, `/` funzionano come uno si aspetta. Ci sono più o meno tutte le funzioni standard `sin(x)`, `cos(x)`, `exp(x)` e cosi via.

La cosa più importante è che ogni cosa in MATLAB è una matrice. Ad esempio

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

cioè la variabile `a` è assegnata allo scalare 1 ma è trattata come una matrice 1×1 come il comando `size` fa vedere. Il comando `size` infatti restituisce una matrice 1×2 dove il primo elemento è il numero di righe e il secondo numero è il numero di colonne.

Inizializzare una matrice in MATLAB è molto facile infatti

```
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
```

come si vede dall'esempio, una matrice è definita elencando gli elementi tra parentesi quadre. Gli elementi sono elencati per righe e separati da spazi (o virgole opzionali). Le righe sono separate da `;`.

Nel caso di matrici con molte righe può essere utile spezzare il comando su più righe con il comando di continuazione `...`, ad esempio

```
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 Accedere agli elementi di una matrice

Accedere agli elementi di una matrice è piuttosto facile e intuitivo, ad esempio

```
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
```

notate che ci sono due modi di accedere ad un elemento di una matrice,

1. con due indici `a(i,j)` dove `i` è l'indice di riga e `j` è l'indice di colonna dell'elemento.
2. con un indice `a(i)` dove `i` è l'indice dell'elemento `i` esimo della matrice *srotolata* per colonne.

accedere con un solo indice è comodo nel caso di matrici $1 \times n$ o $n \times 1$ perché vengono trattate di fatto come dei vettori (di fatto lo sono).

1.2 Accedere a sottoblocchi di una matrice

Accedere a sottoblocchi di matrici è altrettanto facile che accedere a degli elementi della stessa. Se al posto degli indici si mettono dei vettori riga allora è possibile estrarre blocchi di righe e colonne. Se al posto degli indici si mette un vettore allora si estraggono gli elementi elencati.

```
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
```

Questo modo di operare è molto utile nella scrittura di algoritmi che operano su matrici a blocchi o su istruzioni condizionali che vedremi più avanti.

1.3 operazioni sulle matrici

Per le matrici gli operatori `+`, `-`, `*` funzionano come uno si aspetta:

```
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
```

Un commento a parte va fatto per gli operatori `/` e `\`. Infatti `a/b` è formalmente equivalente a `a*inv(b)` dove `inv(b)` è l'operatore che costruisce la matrice inversa di `b`. In pratica `a/b` non costruisce l'inversa di `b` ma risolve il problema `a=x*b`. Invecie `a\b` è formalmente equivalente a `inv(a)*b`. In pratica `a\b` non costruisce l'inversa di `b` ma risolve il problema `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
```

Il comando `norm(a)` è la norma L^2 della matrice `a` che vale

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

dove $\sigma(\boldsymbol{B})$ è il raggio spettrale di $\boldsymbol{B}$ cioè il massimo del modulo degli autovalori di $\boldsymbol{B}$.

Un'altra operazione importante importante sulle matrici è la trasposizione fatta con l'apice:

```
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 operazioni elemento per elemento

Un'altra caratteristica molto comoda di MATLAB è la possibilità di operare elemento per elemento su una matrice in modo elementare. GLi operatori elemento per elemento hanno il prefisso *punto* ".". Ad esempio

```
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
```

Nell'esempio `^` è l'operatore elevamento a potenza, inoltre si nota che ponendo alla fine di una istruzione `;` si ha la soppressione dell'*output*. Cioè normalmente il risultato di ogni istruzione viene stampato subito dopo, il `;` a fine istruzione sopprime la stampa. Attenzione che gli operatori funzionano in modo diverso se prefissati da `.`, infatti:

```
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
```

tutte le funzioni scalari di MATLAB funzionano elemento per elemento, ad esempio

```
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
```

Attenzione che se invece volessimo fare l'esponenziale di matrice cioè

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

bisogna usare la funzione `expm`:

```
0001 >> a = [ 2.1 4.2   ; ...
0002          1   2.001 ] ;
0003
0004 >> exp(a)  % esponenziale componente per componente
0005 ans =
0006     8.1662   66.6863
0007     2.7183    7.3964
0008
```

```
0009 >> expm(a) % esponenziale matriciale
0010 ans =
0011    31.4019   60.8176
0012    14.4804   29.9683
```

1.5 Costruzione di successioni

Spesso programmando con MATLAB è conveniente avere degli operatori per costruire delle successioni di interi o reali a passo costante. In MATLAB l'operatore : (due punti) serve allo scopo

```
0001 >> 1:4   % succesione da 1 a 4
0002 ans =
0003      1     2     3     4
0004
0005 >> 13:34 % succesione da 13 a 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 % succesione da 1 a 13 con passo 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
```

queste successioni possono essere utilmente utilizzate per estrarre blocchi di matrici, ad esempio

```
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
```

la parola chiave end contiene la dimensione massima di riga o di colonna della matrice su cui opera. E' molto comoda nella operazione sulle matrici. Ad esempio se volessimo *ruotare* gli elementi di un vettore basta fare le seguenti operazioni

```
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
```

notate che se in un elemento di una matrice si mette un blocco matrice questo viene *spiattellato*

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

cioè in MATLAB non si possono fare strutture ricorsive (non con le matrici perlomeno) cioè matrici i cui elementi a loro volta sono matrici. Questa caratteristica in realtà è molto comoda perché diventa molto facile costruire matrici complesse a partire da blocchi semplici, ad esempio

```
0001 >> I3 = eye(3) % matrice identità 3x3
0002 I3 =
0003      1     0     0
0004      0     1     0
0005      0     0     1
0006
0007 >> I4 = eye(4) % matrice identità 4x4
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
```

La funzione `eye(n)` costruisce una matrice identità $n \times n$, mentre `zeros(n,m)` e `ones(n,m)` costruiscono matrici di 0 o di 1 $n \times m$.

Una scorciatoia per `1:end` è usare semplicemente due punti com segue

```
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 Principali funzioni di matrice

Esistono un gran numero di funzioni per manipolare le matrici, ad esempio:

```
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) % calcola solo gli autovalori
0007 e =
0008    11.5977
0009    -3.0282
0010    -0.5695
0011    -0.0000
0012
0013 >> [V,e] = eig(M)% calcola gli autovalori e gli autovetori di 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) % controlla la prima coppia autovalore autovettore
0027 ans =
0028    1.0e-13 *
0029
0030          0
0031     0.1066
0032     0.0155
0033     0.0178
0034
0035
0036 >> [L,U,P] = lu(M) % decompositione lu
0037 L =
0038     1.0000         0         0         0
0039     0.2000    1.0000         0         0
0040     0.4000    0.7500    1.0000         0
0041     0.2000   -0.2500   -0.0000    1.0000
0042
0043 U =
0044     5.0000    6.0000    7.0000    8.0000
0045          0    0.8000    1.6000    2.4000
0046          0         0   -3.0000   -5.0000
0047          0         0         0    0.0000
0048
0049 P =
0050      0     1     0     0
0051      1     0     0     0
0052      0     0     0     1
0053      0     0     1     0
0054
0055 >> norm(P*L*U - M)  % controllo la decompositione
0056 ans =
0057      0
0058
0059 >> [Q,R] = qr(M) % fattorizzazioen QR
0060 Q =
0061    -0.1796    0.7671    0.5690   -0.2357
0062    -0.8980   -0.3002    0.2188    0.2357
0063    -0.1796   -0.2668   -0.0875   -0.9428
0064    -0.3592    0.5003   -0.7878   -0.0000
0065
0066 R =
0067    -5.5678   -7.0046   -7.3638   -8.0822
0068          0    0.9672    0.4336    0.4002
0069          0         0    2.3635    3.9392
0070          0         0         0   -0.0000
0071
0072 >> norm(Q*R - M)  % controllo la fattorizzazione
```

```
0073 ans =
0074    1.7681e-15
0075
0076 >> [U,S,V] = svd(M) % decomposizione ai valori singolari
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) % controllo la decompositione
0096 ans =
0097    5.0559e-15
0098
0099 >> norm(M,1)       % norma 1 di matrice
0100 ans =
0101     13
0102
0103 >> norm(M,'inf')   % norma inifinito di matrice
0104 ans =
0105     26
0106
0107 norm(M,'fro')   % norma di Frobenius
0108 ans =
0109    14.8997
```

2.1 Alcune utili funzioni di matrice

Ci sono alcune funzioni di matrice che sono molto utili nella costruzione di algoritmi. Le funzioni `max(a)` e `min(a)` restituiscono un vettore riga nel quale ogni elemento è il massimo (o il minimo) dei valori nella colonna. Se la matrice è un vettore riga o colonna `max(a)` e `min(a)` restituisce il massimo o minimo del vettore. In modo analogo la funzione `sum(a)` restituisce il vettore riga con la somma dei valori sulle colonne di `a`. Se la matrice è un vettore riga o colonna `sum(a)` restituisce la somma delle componenti del vettore. In modo analogo funziona la funzione `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))) % norma 1
0030 ans =
0031      7
0032 >> norm(a,1)
0033 ans =
0034      7
0035
0036 >> max(sum(abs(a'))) % norma inifinito
0037 ans =
0038     10
0039 >> norm(a,'inf')
0040 ans =
0041     10
```

Se assegnamo i risultati della funzione `max(a)` o `min(a)` ed una coppia di variabili

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

allora `res` contiene il vettore riga con gli elementi massimi (o minimi) della colonna e `idx` contiene gli indici di riga dell'elemento massimo (o minimo) ad esempio

```
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
```

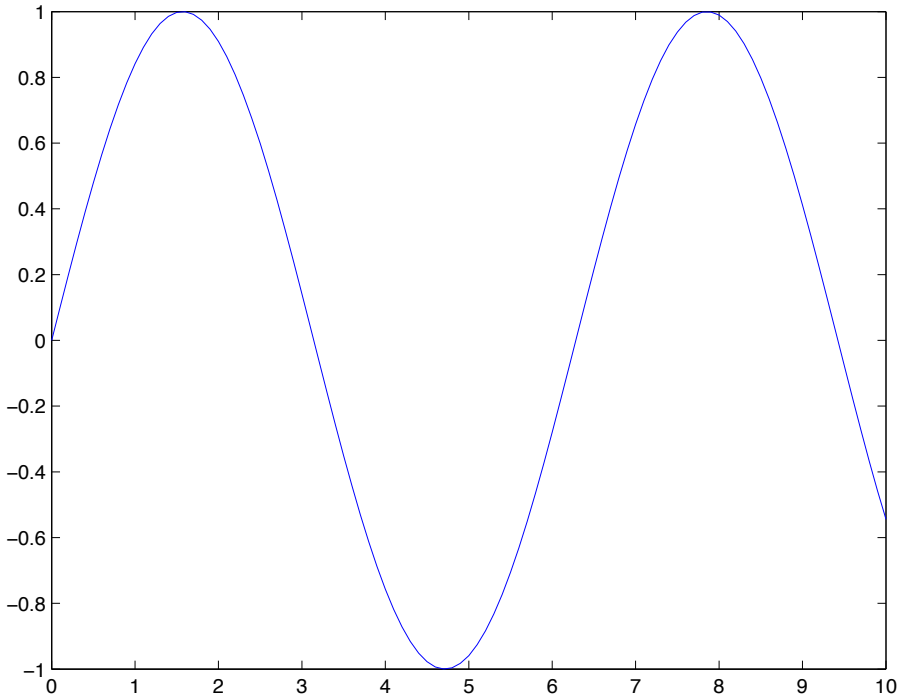
La funzione `length(a)` è equivalente a `max(size(a))`.

3 Parte grafica

In MATLAB è molto facile fare il grafico di una funzione con il comando plot, ad esempio il comando

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

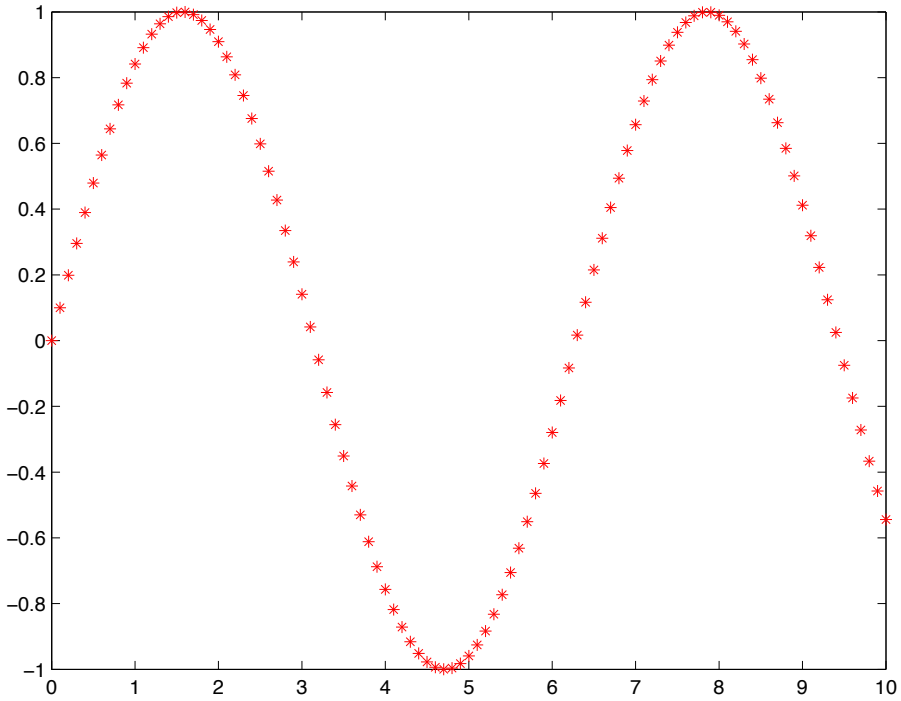
produce la figura



mentre il seguente

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

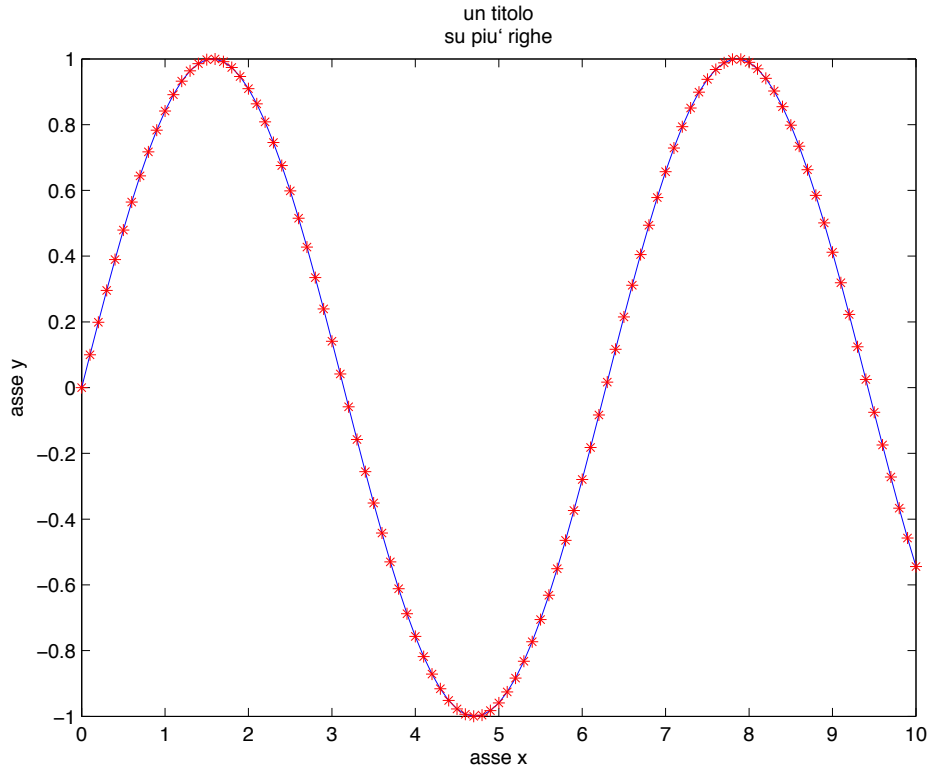
produce la figura



con il comando `hold` si possono combinare vari plot per ottenere grafici complessi. Ad esempio i comandi

```
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
```

producono la figura



La parte grafica di MATLAB è molto complessa e estesa, per vedere in dettaglio le varie possibilità basta digitare `doc plot` nella finestra di comando.

4 Oltre alle matrici

Nella costruzione di algoritmi complessi a volte non bastano solo vettori e matrici ma occorrono strutture dati più complesse. MATLAB fornisce due possibilità interessanti, i cell array e le strutture. Un cell array può essere visto come un vettore eterogeneo che può contenere qualunque cosa. La sintassi è simile a quelle delle matrici dove vengono sostituite le parentesi tonde e quadre con le parentesi grafe. Ad esempio

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

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

Il seguente esempio fa vedere come si possono costruire strutture molto complesse:

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

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

nell'esempio precedente si nota l'uso della funzione `disp` che stampa una stringa a terminale.

Per accedere agli elementi di un cell array si usano gli interi. Un struttura è simile al cell array nel senso che permette di costruite strutture dati complesse ma l'accesso agli elementi è tramite l'operatore punto "." e i nomi dei *campi* corrispondenti. Ad esempio

```
0001 % esempio di sruttura
0002 >> a . pippo    = [ 1 2 3 ] ; % il campo pippo è un vettore riga
0003 >> a . pluto    = [ 1 ; 3 ] ; % il campo pluto è un vettore colonna
0004 >> a . paperino = { 'a' 'b' 3 } ; % il campo paperino è un cell array
0005 >> a . subs . a = 1 ; % il campo subs è una struttura a sua volta
0006 >> a . subs . b = 2 ; % il campo subs è una struttura a sua volta
0007 >> a . subs . c = 3 ; % il campo subs è una struttura a sua volta
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
```

Strutture e matrici si possono combinare. Ad esempio si possono fare vettori di strutture e strutture di vettori.

5 L'istruzione find

MATLAB supporta le istruzioni condizionali `if`–`else`–`end` cicli `while`–`end` e `for`–`end`. Con queste istruzioni si può programmare in maniera procedurale, ma non è la maniera *giusta* di programmare in MATLAB. Se si riesce è meglio cercare di costruire gli algoritmi in maniera vettoriale cercando di fare blocchi di istruzioni che operano su matrice e vettori e non sulle loro componenti. A questo proposito la funzione `find` permette di vettorizzare cicli condizionali. L'istruzione `find` si può usare essenzialmente in 2 modi:

Modo 1

Si usa `find` per farsi restituire gli indici di riga e colonna che soddisfano una condizione logica, ad esempio

```
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
```

Modo 2

Si usa `find` per farsi restituire un indice corrispondente che soddisfano una condizione logica, ad esempio

```
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
```

Possiamo usare la funzione `find` per fare cose che normalmente faremmo con dei cicli. Ad esempio se vogliamo cambiare tutte le componenti negative di una matrice con i loro quadrati:

```
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 Instruzioni condizionali

L'istruzione `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
```

Il ciclo `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
```

Il ciclo `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
```

Ogni ciclo `for` o `while` può essere interrotto da una istruzione `break` in ogni punto del ciclo.

7 Confronti

Nelle istruzioni condizionali abbiamo già visto alcune istruzioni condizionali MATLAB supporta le seguenti

- `eq`, `==` uguaglianza
- `ne`, `~=` diseguaglianza
- `lt`, `<` minore
- `gt`, `>` maggiore
- `le`, `<=` minore o uguale

- `ge`, `>=` maggiore o uguale

e gli operatori logici

- `and`, `&` and
- `or`, `|` or
- `not`, `~` not
- `xor` or esclusivo
- `any` vera se qualche elemento del vettore è non zero
- `all` vera se ogni elemento del vettore è non zero

8 Gli script

In MATLAB si possono definire funzioni e scriverle su file esterni che poi possono essere richiamati da riga di comando. Il file deve chiamarsi `comando.m` dove `comando` è il nome del comando che si vuole usare. All'interno di `comando.m` si troverà la definizione della funzione `comando` tipo

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

consideriamo ad esempio il file `distanza.m`

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

la funzione `distanza(x,y)` restituisce la distanza tra due vettori vettori, ad esempio:

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

Alcune considerazioni

- A differenza di altri linguaggi di programmazione il blocco `function` non necessariamente è terminato da `end`.
- Il nome della funzione è **necessariamente** lo stesso del file `.m`.
- È possibile definire altre funzioni all'interno dello stesso script. Queste funzioni sono pero visibili *solo* all'interno dello script.
- Gli argomenti sono passati per *valore* (non è esattamente così ma rende l'idea) quindi se modificati all'interno della funzione le modifiche non sono viste dal programma chiamante.

8.1 Restituzione di più valori

Se servono più valori da restituire questi si mettono in una lista. Ad esempio dato lo script `trenorme.m`

```
0001 % calcola varie norme di un vettore
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)) ;
```

calcola le norme 1, 2 e ∞ di un vettore. Il seguente comando produce:

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

8.2 Numero variabile di argomenti

Una altra caratteristica molto utile di MATLAB è la possibilità di usare un numero variabile di argomenti in ingresso e uscita. Per avere più informazioni su questa possibilità usare

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

8.3 Esempio complesso

```
0001 %
0002 % Risolve il sistema lineare A*x = b
0003 %
0004 function x = linearsistemsolve( A, b )
0005
0006   [nr,nc] = size(A) ;
0007
0008   % controllo dimensioni
0009   if nr ~= nc
```

```
0010      error( sprintf( 'la matrice A (%d x %d) non e' quadrata ', nr,nc) ) ;
0011    end
0012
0013    neq = length(b) ;
0014
0015    if nr ~= neq
0016      error( sprintf( 'la matrice A (%d x %d) non e' compatibile\ncon la lunghezza d
0017    end
0018
0019    M = [ A b ] ; % costruisco la matrice aumentata
0020    for i=1:neq-1
0021      % cerco l'elemento di massimo modulo colonna i-esima
0022      [V,k] = max(abs(M(i:end,i))) ;
0023
0024      % rimetto l'indice k alla giusta posizione
0025      k = k + i-1 ;
0026
0027      % scambio la riga i con la riga k
0028      BF     = M(i,:) ;
0029      M(i,:) = M(k,:) ;
0030      M(k,:) = BF ;
0031
0032      % controllo singolarità
0033      if  V == 0
0034        error( 'Matrice singolare' ) ;
0035      end
0036
0037      % costruisco la matrice di Frobenius
0038      L             = eye(neq) ;
0039      L(i+1:end,i)  = -M(i+1:end,i)./M(i,i) ;
0040
0041      % azzero gli elementi della colonna i
0042      M             = L * M ;
0043
0044    end
0045
0046    % alloco il vettore soluzione
0047    x = zeros(neq,1) ;
0048
0049    % a questo punto la matrice è in forma triangolare
0050    % applichiamo le operazioni di ritorno
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
```

Esempio di uso

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

```
0002 >> b = [ 1 2 3 ]
0003 >> x = linearsistemsolve(A,b)
0004 x =
0005      4
0006     -3
0007      1
```

Riferimenti bibliografici

[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.