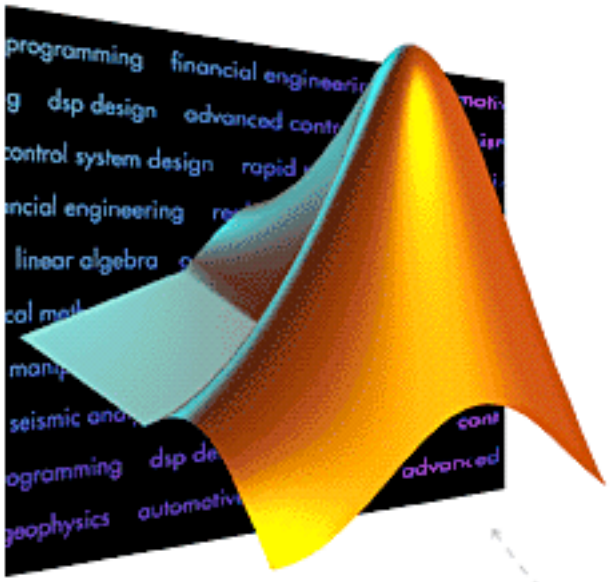
Introduzione all'uso di

MATLAB®

The Language of Technical Computing



The MATH WORKS Inc.

Francesco Odetti
DIPTEM
Università di Genova
A.a. 2009/10

MatLab è un programma studiato apposta per operare su matrici. Il nome è un'abbreviazione di **Matrix Laboratory**. **Le variabili sono infatti matrici** e le funzioni di base sono operazioni su matrici. Istruzioni più avanzate consentono tracciamento di grafici in 2D e in 3D, elaborazione di dati per il trattamento di segnali etc.

In queste pagine di introduzione elementare illustriamo alcune delle istruzioni più semplici.

INPUT DI MATRICI

Quando il programma attende un'istruzione, mostra un prompt del tipo

```
»
```

Per ottenere la matrice $a = \begin{pmatrix} 5 & 0 & 2 \\ 1 & 4 & -1 \\ -2 & 2 & 3 \end{pmatrix}$ si scrive:

```
» a=[5,0,2 ; 1,4,-1 ; -2,2,3]
```

e si preme [RETURN]. Le virgole servono per separare gli elementi di una riga, il simbolo `;` per separare le righe della matrice.
Si ottiene:

```
a =
     5     0     2
     1     4    -1
    -2     2     3
```

In questo modo si è assegnato alla matrice il nome `a` e la matrice può essere riutilizzata in seguito.
In luogo delle virgole si possono usare, come normalmente conviene, gli spazi, quindi per ottenere la matrice `a` si può anche scrivere

```
» a=[5 0 2 ; 1 4 -1 ; -2 2 3]
```

ALCUNE OPERAZIONI ELEMENTARI SULLE MATRICI

Trasposta di una matrice:

```
» a'
```

Somma di matrici (**errore** se `a` e `b` non hanno lo stesso formato):

```
» a+b
```

Prodotto tra matrice e scalare: (`k` è lo scalare, `a` è la matrice)

```
» k*a
```

Prodotto di matrici: (**errore** se numero colonne (`a`) $\neq$ numero righe (`b`))

```
» a*b
```

Potenza `n` -ma di una matrice (**errore** se non è quadrata) :

```
» a^n
```

Inversa di una matrice (**errore** se non è quadrata): è ottenuta con l'algoritmo gaussiano (anche se non si vede).

```
» inv(a)
```

Determinante di un matrice (**errore** se non è quadrata): (sempre mediante l'algoritmo gaussiano)

```
» d=det(a)
```

Risultato:

```
d =
    90
```

In questo modo si assegna allo scalare `d` il valore `det(a)` per uso successivo.
Caratteristica di una matrice: (ancora con l'algoritmo gaussiano)

```
» rank(a)
```

Risultato:

```
ans =
     3
```

Osservazione: dato che non si è assegnato un nome al numero `rank(a)` , il programma assegna d'ufficio il nome `ans` (answer), così il risultato non va perso fino al prossimo calcolo.

EDITING DI MATRICI

Per cambiare l'elemento (1,2) di `a` da 0 a 7

```
» a(1,2)=7
```

Si ottiene:

```
a =
     5     7     2
     1     4    -1
    -2     2     3
```

INPUT DI MATRICI SPARSE

Per ottenere la matrice diagonale 4×4 **s** che ha sulla diagonale 1 , 2 , 3 , 4 si può operare così:

```
» s1=[1 2 3 4];
» s=diag(s1)
```

Osservazione: dato che la matrice riga **s1** è solo un passaggio intermedio del conto, è possibile evitare che venga scritta su schermo apponendo il simbolo **;** dopo la prima istruzione.
Questa opzione verrà spesso usata anche in seguito.
Per ottenere la matrice a bocchi **b** si può operare così:

```
» b=[2 3 ; 7 8]
```

$$\mathbf{b} = \begin{pmatrix} 2 & 3 & 0 \\ 7 & 8 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

Si ottiene:

```
b =
     2     3
     7     8
```

Poi si pone:

```
» b(3,3)=9
```

Si ottiene:

```
b =
     2     3     0
     7     8     0
     0     0     9
```

Il formato di **b** è cambiato e gli elementi non definiti sono posti automaticamente uguali a zero.
La matrice identica **n** $\times$ **n** si ottiene con:

```
» eye(n)
```

Il curioso nome deriva dal fatto che in inglese la lettera I (che rappresenta la matrice identica) e la parola eye si pronunciano allo stesso modo, ma **eye(n)** è sicuramente più riconoscibile di **I(n)** .
La matrice nulla quadrata **n** $\times$ **n** si ottiene con:

```
» zeros(n)
```

La matrice nulla rettangolare **m** $\times$ **n** si ottiene con:

```
» zeros(m,n)
```

Per ottenere la matrice a blocchi **c** si può quindi per esempio operare così:

```
» c1=[2 3 ; 7 8];
» c2=[9 4 5 ; 6 1 3 ; 3 3 3];
» c=[c1 zeros(2,3) ; zeros(3,2) c2]
```

$$\mathbf{c} = \begin{pmatrix} 2 & 3 & 0 & 0 & 0 \\ 7 & 8 & 0 & 0 & 0 \\ 0 & 0 & 9 & 4 & 5 \\ 0 & 0 & 6 & 1 & 3 \\ 0 & 0 & 3 & 3 & 3 \end{pmatrix}$$

Analogamente alla matrice nulla è possibile ottenere una matrice tutta composta di numeri 1 coi comandi:

```
» ones(n)
» ones(m,n)
```

Per esempio

```
» a=5*ones(2,3)
a =
     5     5     5
     5     5     5
```

ESTRAZIONE DI RIGHE, COLONNE E DIAGONALI

La matrice riga **r** costituita dalla seconda riga di **a** si ottiene con

```
» r=a(2,:)
```

Risultato:

```
r =
     1     4    -1
```

La matrice colonna **m** costituita dalla terza colonna di **a** si ottiene con

```
» m=a(:,3)
```

La funzione **diag** applicata a una matrice e non a un vettore fornisce la matrice colonna contenente la diagonale della matrice data

```
» diag(a)
ans =
     5
     4
     3
```

Quindi la funzione `diag(diag(a))` applicata a una matrice fornisce la matrice diagonale con diagonale identica a quella di `a` .

```
» diag(diag(a))
ans =
     5     0     0
     0     4     0
     0     0     3
```

Analogamente le funzioni `tril(a)` `triu(a)` forniscono una matrice rispettivamente triangolare inferiore (lower triangular) e triangolare superiore (upper triangular) partendo da `a` . Per esempio

```
» tril(a)
ans =
     5     0     0
     1     4     0
    -2     2     3
```

La funzione `tril(triu(a))` è perfettamente equivalente a `diag(diag(a))` .

OPERAZIONI ELEMENTARI SULLE RIGHE

L'operazione $R_3 \to 5R_3$ su `a` si può eseguire con il comando seguente:

```
» a(3,:)=5*a(3,:)
```

L'operazione $R_3 \to R_3 + 5R_2$ su `a` si può eseguire con il comando seguente:

```
» a(3,:)=a(3,:)+5*a(2,:)
```

L'operazione $R_1 \leftrightarrow R_3$ su `a` è un po' più complicata e si può ottenere in uno dei seguenti due modi:

```
» y=a(3,:)  ; a(3,:)=a(1,:)  ; a(1,:)=y
```

dove si usa la variabile temporanea `y` . In questo caso sono state date tre istruzioni in una sola riga, separate con il simbolo `;` che impedisce che vengano scritti su schermo i risultati delle prime due istruzioni. Notare che, scrivendo semplicemente

```
» a(3,:)=a(1,:)  ; a(1,:)=a(3,:)
```

senza usare la variabile ausiliaria `y` , andrebbe persa la prima riga di `a` .
L'altro modo è quello di scrivere:

```
» a([1 3],:)=a([3 1],:)
```

(notare gli spazi tra `1` e `3` e tra `3` e `1` . La spiegazione dettagliata di questo tipo di comando viene data nel paragrafo seguente.

MANIPOLAZIONE DI MATRICI - IL COMANDO :

Come già visto, il simbolo `:` in **MatLab** è fondamentale per la manipolazione delle matrici.
Il manuale stesso recita: once you master `:` , you master **MatLab** (una volta appreso l'uso del simbolo `:` (*colon* in inglese) avrete appreso **MatLab**). Diamo alcuni esempi dei principali usi del simbolo `:` .
• Generazione di vettore che sia una progressione aritmetica di passo `1` :

```
» x=1:5
x =
     1     2     3     4     5
```

• Generazione di vettore che sia una progressione aritmetica di passo `0.3` :

```
» x=1:0.3:2
x =
    1.0000    1.3000    1.6000    1.9000
```

Per avere una progressione aritmetica di passo negativo occorre precisare il passo negativo, altrimenti si ottiene la progressione vuota:

```
» x=3:0
x =
   Empty matrix:  1-by-0
» x=3:-1:0
x =
     3     2     1     0
```

Come già visto, mediante il simbolo `:` si possono estrarre righe e colonne delle matrici e anche riordinarle.
Illustriamo alcuni tipici esempi dell'uso di `:` adoperando la curiosa matrice `m` di ordine 4 che si ottiene con la

funzione **magic** . La funzione **magic(n)** restituisce una matrice **n** $\times$ **n** che è un quadrato magico di ordine **n** (la somma di ogni riga, di ogni colonna e di ogni diagonale è sempre la stessa, in questo caso è 34).

```
» m=magic(4)
m =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Estrazione della sottomatrice costituita da R_1, R_3, C_3, C_4 .

```
» m([1 3], [3 4])
ans =
     3    13
     6    12
```

Estrazione della sottomatrice costituita da R_1, R_3, C_3, C_4 , ma scambiando le due colonne:

```
» m([1 3], [4 3])
ans =
    13     3
    12     6
```

Estrazione della sottomatrice costituita da tutte le righe e da C_3, C_4 , scambiate:

```
» m(:,[4,3])
ans =
    13     3
     8    10
    12     6
     1    15
```

Estrazione della sottomatrice costituita dalla quarta riga e da tutte le colonne da C_1 a C_3 :

```
» m(4,1:3)
ans =
     4    14    15
```

• Vediamo ancora un modo di costruire le matrici a blocchi.
La matrice **5** $\times$ **5** a blocchi **c** definita sopra si può ottenere anche così:

```
» c=[2 3 ; 7 8];
» c(3:5,3:5)=[9 4 5 ; 6 1 3 ; 3 3 3];
```

• Flattening della matrice
Consiste nel trasformare una matrice in un vettore colonna costituito da tutti gli elementi della matrice (nel nostro esempio 16 elementi) letti per colonne.

```
» m(:)
ans =
    16
     5
     9
     4
     2
    11
.........  etc ......
```

Aggiungiamo per inciso che il comando

```
» x(k)
```

applicato a un vettore, cioè a una matrice **1** $\times$ **n** o **n** $\times$ **1** , fornisce il **k** -mo elemento di **x** . Lo stesso comando applicato a una matrice qualunque fornisce il **k** -mo elemento del flattening della matrice. Per esempio

```
» m(6)
ans =
    11
```

fornisce il sesto elemento della matrice **m** letta per colonne.

Il comando **[ ]** fornisce la matrice vuota. Può essere usato per sopprimere parti di una matrice. Per esempio

```
» m(1:2,:)=[ ]
```

sopprime le prime due righe di **m** .

OPERAZIONI ALGEBRICHE SULLE MATRICI

Ricapitoliamo ora le operazioni algebriche tra matrici nelle loro varie forme.
• Somma tra matrici: è la somma usuale tra matrici dello stesso formato.

```
» a+b
```

• Somma tra una matrice e uno scalare: addiziona ad ogni elemento di `a` lo scalare. Per esempio:

```
» a+2
```

Attenzione ! Per matrici quadrate `a+k` non è equivalente a `a+k*I` .

• Prodotto tra matrici: è il prodotto usuale tra matrici: occorre che numero colonne (`a`) = numero righe (`b`).

```
» a*b
```

• Prodotto tra una matrice e uno scalare: è il prodotto usuale tra matrice e scalare. Per esempio:

```
» a*2
```

• Prodotto puntuale `.*` : è il prodotto elemento per elemento tra matrici dello stesso formato.

```
» a.*b
```

• Divisione a sinistra: è logicamente equivalente a `inv(a)*b` , ma è ottenuta con l'algoritmo gaussiano. Per i dettagli di questa operazione, vedi più avanti l'algoritmo di Gauss.

```
» a\b
```

• Divisione a destra: equivalente logicamente a `b*inv(a)` , quindi eseguibile se i formati lo consentono, ma ottenuta anch'essa con l'algoritmo gaussiano. In effetti è ottenuta calcolando `(a'\b')'`

```
» b/a
```

Prestare attenzione al fatto che i due simboli `\` e `/` determinano la soluzione ai minimi quadrati nei casi non standard.

• Divisione puntuale a destra `./` : è la divisione elemento per elemento tra matrici dello stesso formato.
Attenzione: `a` fornisce i numeratori, `b` i denominatori.

```
» a./b
```

• Divisione puntuale a sinistra `.\` : è la divisione elemento per elemento di matrici dello stesso formato.
È perfettamente equivalente alla precedente, ma **attenzione**: anche qui `a` fornisce i numeratori, `b` i denominatori.

```
» b.\a
```

• La potenza `n`-esima (`n` numero intero) di una matrice **quadrata** è il prodotto della matrice `a` per sé stessa `n` volte.

```
» a^n
```

Questa funzione può essere calcolata anche se `n` non è un numero intero, in tal caso il significato è assai più complesso e l'algoritmo più lungo (cfr. più avanti la funzione `expm(a)`).

• La potenza puntuale `.^` consiste nell'elevare alla `n` (numero reale) ogni elemento di `a` (matrice qualunque)

```
» a.^n
```

• Formato di una matrice: è spesso necessario conoscere il formato di una variabile definita (ricordiamo che tutte le variabili in **MatLab** sono matrici). La funzione è

```
» size(a)
```

che restituisce il formato di `a` , cioè una matrice 1×2 con il numero di righe e il numero di colonne di `a` .
Si può per esempio scrivere

```
» [r,c]=size(a)
```

per avere in `r` e in `c` rispettivamente il numero di righe e il numero di colonne di `a` .
Se si vuole solo il numero di righe di o il numero di colonne della matrice `a` , le funzioni rispettive sono

```
» size(a,1)
» size(a,2)
```

Un'altra funzione spesso usata è la seguente

```
» length(a)
```

che restituisce la **massima** dimensione di `a` ed è quindi usata di solito per conoscere la lunghezza di un vettore.

HELP ON LINE

Per avere informazioni su un dato comando esiste il comando `help` che fornisce rapidamente (in inglese) uno *help-on-line*, cioè una sintetica descrizione del comando e del suo uso. Per esempio

```
» help length

 LENGTH Length of vector.
        LENGTH(X) returns the length of vector X. It is equivalent
    to MAX(SIZE(X)) for non-empty arrays and 0 for empty ones.
```

Attenzione: i comandi e le funzioni in **MatLab** vanno introdotti con le lettere minuscole, mentre lo help-on-line

le riporta in carattere maiuscolo per metterle in evidenza.

Comunque le versioni più recenti di **MatLab** hanno uno help sotto forma di ipertesto assai completo (anche se meno immediato).

COSTANTI, VARIABILI E FORMATO

Le variabili in **MatLab** possono avere un nome lungo un massimo di 31 caratteri. La prima lettera di una variabile deve essere un carattere alfabetico (`a-z` , `A-Z`) mentre dalla seconda lettera in avanti possiamo utilizzare un qualsiasi carattere alfanumerico incluso il simbolo underscore _ .
È bene tener presente che anche le variabili in **MatLab** sono "case-sensitive" (cioè sensibili al maiuscolo minuscolo); per esempio le variabili `a` e `A` sono differenti e la funzione `Det` (con la d maiuscola) non esiste.

Le funzioni predefinite in **MatLab** per ottenere le costanti più comuni sono: `pi` (pi greco) `i` , `j` (entrambe unità immaginaria), `eps` (precisione macchina).
Inoltre possono essere ottenute le variabili speciali

`Inf` (infinito), quando si divide un numero per 0 o si calcola il logaritmo di 0 o si va in overflow.
`NaN` (Not a Number) che compare quando si esegue l'operazione $0/0$ o l'operazione `Inf/Inf` .

Prestare inoltre attenzione al fatto che `pi` , `i` , `j` , `Inf` , `eps` , `NaN` possono essere definite come variabili e perdere in alcuni comandi il loro significato di funzioni o di variabili predefinite. Ma i **comandi** `pi` , `i` , `eps` , `j` rimangono comunque validi.

L'insieme delle variabili definite dall'utente viene detto **workspace** ed è possibile visualizzarne l'elenco mediante il comando

```
» who
```

Per avere maggiori informazioni per ogni variabile si può usare

```
» whos
```

che fornisce, per ogni variabile anche il formato, il numero di byte usati e la classe.
Esistono varie classi di variabili.
Le matrici di default sono formate da numeri in doppia precisione che occupano 8 byte per elemento (classe `double`). Oltre a queste ci sono altri tipi di variabili quali le stringhe alfanumeriche (classe `char`), le variabili logiche (classe `logical`) che vedremo più avanti, e altre.

Per sapere a che classe appartiene una data variabile (per esempio `a`) esiste il comando

```
» class(a)
```

Osserviamo comunque che le ultime versioni di **MatLab** possono visualizzare in un'apposita finestra il **Workspace** ed è disponibile un editore per le variabili (che sono matrici) simile a un foglio elettronico (tipo Excel).

Per eliminare una variabile (per esempio `a`) dal *workspace* esiste il comando.

```
» clear a
```

Lo stesso comando senza argomento elimina dal *workspace* tutte le variabili

```
» clear
```

Normalmente ogni variabile è memorizzata internamente in doppia precisione e quindi è esatta fino a circa la 15-ma cifra decimale, ma in genere vengono visualizzate solo le prime quattro cifre decimali. Per esempio

```
» x=pi/2
x =
    1.5708
```

Per far sì che **MatLab** visualizzi tutte le cifre decimali disponibili di ogni numero occorre il comando

```
» format long
x =
    1.57079632679490
```

Per ripristinare la visualizzazione breve si usa

```
» format short
```

Il comando `format` ha inoltre le due opzioni

```
» format compact
» format loose
```

che fanno passare rispettivamente alla visualizzazione compatta senza righe vuote (che permette di far comparire più righe sullo schermo) e a quella larga con una riga vuota tra un comando e la sua risposta.
Le schermate riportate in questo manuale sono state ottenute con l'opzione `format compact` per risparmiare spazio.

FUNZIONI ELEMENTARI SULLE MATRICI

Sono definite le principali funzioni elementari reali (e complesse) che consistono nell'eseguire la funzione elementare **su ogni elemento** della matrice.
Le principali funzioni applicabili a una matrice `x` di qualunque formato sono:

• Valore assoluto (modulo se l'elemento è complesso).

```
» abs(x)
```

• Ci sono quattro funzioni per la parte intera:

  • Il numero intero più prossimo a `x` :

```
» round(x)
```

  • Il numero intero più prossimo a `x` in direzione dello 0, in pratica togliendo i decimali:

```
» fix(x)
```

  • La parte intera tradizionale (il più grande numero intero minore o uguale a `x`):

```
» floor(x)
```

  • Il più piccolo numero intero maggiore o uguale a `x` :

```
» ceil(x)
```

• Radice quadrata (se l'elemento è negativo o complesso fornisce una delle radici quadrate complesse, come vedremo in seguito). L'espressione `sqrt` è l'abbrevazione di "square root" (radice quadrata)

```
» sqrt(x)
```

• Esponenziale (funziona anche se l'elemento è complesso mediante l'uso della formula di Eulero, come vedremo in seguito).

```
» exp(x)
```

• Logaritmo naturale in base `e` (funziona anche se l'elemento `x` è negativo o complesso come vedremo in seguito).

```
» log(x)
```

• Le tre funzioni trigonometriche dirette e inverse (funzionano anche se l'elemento è complesso).

```
» sin(x)
» cos(x)
» tan(x)
```

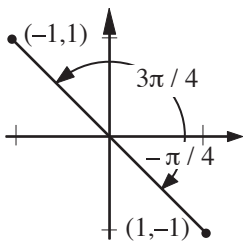
```
» asin(x)
» acos(x)
» atan(x)
```

È bene ricordare che la funzione arcotangente `atan(x)` restituisce l'unico numero θ con $-\pi/2 < \theta < \pi/2$ tale che tang(θ) sia `x` (θ in radianti).

Esiste quindi un'altra funzione denotata `atan2` (arcotangente a due variabili) che tiene conto del quadrante nel piano cartesiano.
Precisamente, `atan2(y,x)` è l'unico numero θ con $-\pi < \theta \leq \pi$ tale che $\begin{cases} \cos(\theta) = x \Big/ \sqrt{x^2+y^2} \\ \sin(\theta) = y \Big/ \sqrt{x^2+y^2} \end{cases}$

Geometricamente `atan2(y,x)` è l'angolo orientato θ in radianti tra l'asse positivo delle x e il segmento orientato $(0,0) - (x,y)$.
Nel primo e nel quarto quadrante del piano cartesiano si ha `atan2(y,x)=atan(y/x)` ; mentre nel secondo e nel terzo quadrante la funzione `atan2(y,x)` aggiunge π a `atan(y/x)` .
Osservare che negli argomenti di `atan2` l'ordinata `y` viene prima dell'ascissa `x` .
Esempio

```
» atan(-1)
ans =
   -0.7854
» atan2(-1,1)
ans =
   -0.7854
» atan2(1,-1)
ans =
    2.3562
```



che va interpretato così:
l'arcotangente di -1 è -0.7854 (circa $-\pi/4$);
anche l'arcotangente2 del punto $(1,-1)$ (nel quarto quadrante) è $-\pi/4$.
ma l'arcotangente2 del punto $(-1,1)$ (nel secondo quadrante) è 2.3562 (circa $3\pi/4$).

• Accenniamo anche alla funzione

```
» expm(a)
```

che permette di calcolare l'**esponenziale matriciale** di una matrice quadrata `a` .

Se **a** è una matrice quadrata, l'esponenziale matriciale di **a** si può definire grosso modo come il limite della sommatoria **I+a+a^2/2!+a^3/3!+...** (**I** matrice identica)
Se **a** è diagonalizzabile e quindi **a = p d p^(-1)** , allora si può definire **expm(a)=p exp(d) p^(-1)** dove l'esponenziale della matrice diagonale è quello ovvio. In realtà l'esponenziale è ottenuto attraverso un altro algoritmo, più complesso a descriversi, ma più efficiente.
È possibile, in modo analogo, calcolare altre funzioni matriciali, quali **logm(a)** , **sqrtm(a)** , **a^n** (con **n** qualunque) e diverse altre.

MATRICI CASUALI

Le funzioni per ottenere una matrice casuale sono:

```
» rand(m,n)
» rand(n)
```

con **m,n** numeri interi.
Si ottengono rispettivamente una matrice casuale **m** $\times$ **n** e una matrice casuale quadrata di ordine **n** (possono essere utili per collaudare comandi o funzioni). Ognuno degli elementi è scelto casualmente con distribuzione uniforme ed è un numero compreso tra 0 e 1.
Da notare che in realtà i numeri sono *pseudo*-casuali e l'algoritmo di generazione dei numeri pseudocasuali viene resettato ad ogni sessione di **MatLab** per cui il primo numero ottenuto con **rand** è sempre 0.9501 e la successione di numeri casuali è sempre la stessa.
Per avere un numero veramente imprevedibile, viene consigliato di resettare la successione di numeri casuali con un comando del tipo

```
» rand('state',sum(100*clock))
```

dopodiché il primo numero pseudocasuale è generato usando la data e l'ora corrente.

La curiosa funzione già vista sopra

```
» magic(n)
```

con **n** numero intero fornisce una matrice **n** $\times$ **n** che è un quadrato magico (stessa somma in tutte le righe, colonne e diagonali) (anche questa può essere utile per collaudare dei comandi).

ALGORITMO DI GAUSS

Se **a** è una matrice quadrata invertibile e **b** una matrice colonna con con lo stesso numero di righe di **a** , il sistema lineare **ax = b** ha un'unica soluzione. **MatLab** è in grado di determinare la soluzione mediante l'algoritmo gaussiano facendo uso della pivotizzazione parziale, cioè scegliendo per ogni colonna il pivot con valore assoluto più alto. Il comando è il seguente

```
» x=a\b
```

Osservare che il simbolo dell'operazione è **** (backslash) e non **/** (slash).
Formalmente il comando equivale al seguente

```
» x=inv(a)*b
```

Ma, come è noto, il tempo di calcolo richiesto dall'algoritmo gaussiano è circa un terzo di quello richiesto dal calcolo dell'inversa. L'inversa di **a** quindi non viene calcolata. In effetti lo stesso manuale di **MatLab** fa osservare che la funzione **inv(a)** è in pratica di uso assai raro.

> Osserviamo per inciso che il comando **a\b** ha un significato assai più ampio e lo si può usare anche se **a** e **b** sono matrici qualunque (di qualunque formato e di qualunque rango, purché **a** e **b** abbiano lo stesso numero di righe). In questo caso viene trovata la cosiddetta **soluzione ai minimi quadrati**.
> In poche parole, dato che non esiste o non è unica una **x** tale che **ax - b = 0** , viene trovata la **x** (di minimo modulo se ce n'è più di una) tale che la norma di **ax - b** sia minima.

Se è necessario studiare ed eventualmente risolvere un sistema qualunque, occorre ridurre totalmente la matrice **completa** del sistema.
La funzione **rref** (row reduced echelon form) calcola una matrice **r** **totalmente ridotta** ed equivalente per righe alla matrice **a** .

```
» r=rref(a)
```

La matrice ridotta è determinata mediante la pivotizzazione parziale e l'algoritmo di Gauss-Jordan che è una variante di quello di Gauss.
Nell'algoritmo di Gauss-Jordan, in ogni colonna viene cercato il pivot col valore assoluto più alto, la riga viene portata al primo posto utile e viene divisa immediatamente per il pivot. Quindi tutta la colonna del pivot viene annullata con operazioni elementari (anche nelle righe sopra il pivot, diversamente dall'algoritmo di Gauss).
Tra le demo c'è un'interessante funzione a scopo didattico, purtroppo mancante nelle ultime versioni di **MatLab**

```
» r=rrefmovie (a)
```

che consente di visualizzare passo-passo la riduzione totale di `a` .

FATTORIZZAZIONE LU

Un altro modo di realizzare l'algoritmo di Gauss per un sistema lineare `ax = b` con `a` matrice invertibile è quello di passare attraverso la fattorizzazione LU .
La fattorizzazione LU di una matrice quadrata invertibile `a` , ha come risultato due matrici. La sintassi per ottenere due output deve specificare il nome dei due risultati:

```
» [l,u]=lu(a)
```

Con questa funzione si ottengono due matrici: `l` invertibile ed "essenzialmente triangolare inferiore" e `u` triangolare superiore tali che `a=l*u` . In pratica `u` è la matrice triangolare superiore ottenuta da `a` mediante l'algoritmo di Gauss con pivotizzazione parziale, mentre `l` è, **a meno di una permutazione delle righe**, triangolare inferiore con tutti numeri 1 sulla diagonale.
Per esempio

```
» a =[1 -3 -1 ; 1 0 2 ; -4 2 1];
» [l,u]=lu(a)
l =
   -0.2500    1.0000         0
   -0.2500   -0.2000    1.0000
    1.0000         0         0
u =
   -4.0000    2.0000    1.0000
         0   -2.5000   -0.7500
         0         0    2.1000
```

La matrice `u` è triangolare superiore e ha sulla diagonale i pivot di `a` , mentre la matrice `l` risulta triangolare inferiore con diagonale di 1 purché vengano opportunamente permutate le righe.
Per risolvere un sistema lineare `ax = b` si risolve il sistema ridotto equivalente `ux = c` dove la matrice `c` dei termini noti del sistema ridotto è la soluzione del sistema lineare essenzialmente ridotto `ly = b` . Ciò si ottiene eseguendo i due comandi

```
» c=l\b;
» x=u\c
```

Il tempo complessivo richiesto dai tre comandi è essenzialmente quello richiesto dal comando

```
» x=a\b
```

Il vantaggio della fattorizzazione LU è il seguente:
Se occorre risolvere due o più sistemi lineari aventi tutti la stessa `a` come matrice dei coefficienti, conviene calcolare una sola volta la sua fattorizzazione LU e quindi risolvere i sistemi lineari con i due comandi precedenti. In questo modo la riduzione gaussiana di `a` viene eseguita una sola volta con evidente risparmio di tempo, specialmente per matrici molto grandi.

È anche possibile ottenere la tradizionale fattorizzazione $PA = LU$ in cui la matrice `l` è proprio triangolare inferiore con diagonale di 1 e la matrice `p` è di permutazione con la seguente funzione a tre output

```
» [l,u,p]=lu(a)
```

NUMERI COMPLESSI

MatLab è in grado di lavorare con i numeri complessi e quindi anche con matrici a elementi complessi.
Occorre distinguere tra il comando `i` e la variabile `i` . Se alla variabile `i` non è stato assegnato alcun valore, allora, mediante il comando

```
» i
```

si ottiene:

```
i =
          0 + 1.0000i
```

In questo caso è possibile assegnare alle variabili di **MatLab** valori complessi con un comando del tipo:

```
» z=1+2*i
```

che assegna alla variabile `z` il numero $1 + 2i$. Si ottiene:

```
z =
   1.0000 + 2.0000i
```

Se invece la variabile `i` è stata definita ed è stato assegnato un valore diverso da i , per esempio se si è posto

```
» i=3
```

si può avere il risultato imprevisto

```
» z=1+i
z =
   4
```

Si tenga comunque presente che per assegnare alla variabile `z` il numero $1+2i$ si possono usare le sintassi seguenti, senza il simbolo `*`, che prescindono dalla eventuale definizione della variabile `i`.

```
» z=1+2i
z =
   1.0000 + 2.0000i
» z=1+1i
z =
   1.0000 + 1.0000i
```

Comunque per riassegnare alla variabile `i` (o se si preferisce a `j`) il valore di unità immaginaria si può scrivere:

```
» i=sqrt(-1)
i =
        0 + 1.0000i
```

Il perché `i` debba essere $\sqrt{-1}$ (simbologia che, di norma, in matematica è bene evitare), verrà chiarito in seguito. Oppure si può dare il comando

```
» clear i
```

che cancella qualunque valore assegnato alla variabile `i`. In ogni caso il **comando** `i` (e anche, con le stesse cautele, il **comando** `j`) fornisce comunque l'unità immaginaria.

MODULO, ARGOMENTO E CONIUGATO

• Parte reale del numero complesso `z`.

```
» real(z)
```

• Parte immaginaria del numero complesso `z`.

```
» imag(z)
```

• Modulo del numero complesso `z`.

```
» abs(z)
```

• Argomento. Per calcolare l'argomento del numero complesso `z` si può usare la funzione `atan2`. La funzione `atan2(y,x)` fornisce, tra tutti gli argomenti di `z`, l'**unico argomento** θ con $-\pi < \theta \leq \pi$ del numero complesso $x+iy$. Quindi si può scrivere:

```
» atan2(imag(z),real(z))
```

Comunque c'è anche la funzione predefinita `angle(z)` che svolge la stessa identica funzione.

```
» angle(z)
```

MatLab conosce la formula di Eulero. Per esempio con

```
» exp(pi*i)
```

che corrisponde all'usuale $e^{\pi i}$ si ottiene:

```
ans =
     -1.0000 + 0.0000i
```

È anche possibile calcolare il logaritmo `log(x)` di un numero complesso `x`. In questo caso la funzione fornisce **una** soluzione `z` dell'equazione `exp(z)=x`. Per questo calcolo il numero `x` viene scritto come `x=r exp(i`θ`)` con $-\pi < \theta \leq \pi$, quindi l'equazione diventa `exp(z)=r exp(i`θ`)`, ovvero `exp(z)=exp(log(r)+i`θ`)`.
Tra le infinite soluzioni dell'equazione la funzione fornisce `z=log(r)+i`θ.
Per esempio è possibile ottenere il numero π in questo modo:

```
» log(-1)
ans =
        0 + 3.1416i
```

• Il coniugato del numero complesso `z` è

```
» conj(z)
```

• Lavorando con matrici a elementi complessi occorre prestare attenzione al fatto che la funzione di trasposizione `a'` applicata a matrici complesse fornisce la matrice **trasposta coniugata** complessa, mentre la semplice trasposta si ottiene con la funzione `a.'`. Per esemplificare:

```
» a=[i 1+i; 2+i 3-i]
a =
        0 + 1.0000i   1.0000 + 1.0000i
   2.0000 + 1.0000i   3.0000 - 1.0000i
» a'
ans =
        0 - 1.0000i   2.0000 - 1.0000i
   1.0000 - 1.0000i   3.0000 + 1.0000i
» a.'
ans =
        0 + 1.0000i   2.0000 + 1.0000i
   1.0000 + 1.0000i   3.0000 - 1.0000i
```

RADICI DI UN NUMERO COMPLESSO

Come appena detto, **MatLab** assegna ad ogni numero complesso, tra i tanti argomenti, un argomento privilegiato θ con $-\pi < \theta \leq \pi$, quindi per calcolare la radice n-esima di un numero complesso divide per n questo argomento e trova la radice aritmetica n-esima del modulo.
Per esempio, per calcolare la radice terza di $2+2i$ e assegnarla alla variabile **z** si scrive

```
» z=(2+2*i)^(1/3)
```

e si ottiene:

```
z =
     1.3660 + 0.3660i
```

che tra le radici terze di $2+2i$ è quella ottenuta dividendo per 3 l'argomento privilegiato $\pi/4$ di $2+2i$.
Ciò spiega anche perché con **sqrt(-1)** si ottenga i , dato che -1 ha argomento π (e i ha argomento $\pi/2$).
Per ottenere le altre due radici di $2+2i$ occorre aumentare l'argomento di $2\pi/3$, cosa che si può fare moltiplicando questo numero successivamente per $e^{2\pi i/3}$ e per $e^{4\pi i/3}$. Per esempio col comando

```
» z*exp(2*pi*i/3)
```

si ottiene la successiva radice terza di $2+2i$.

```
ans =
    -1.0000 + 1.0000i
```

POLINOMI E LORO RADICI

MatLab è in grado di calcolare (in modo approssimato !) radici di polinomi mediante un complesso algoritmo. Prima però occorre trasformare un polinomio in una matrice. Per esempio il polinomio $p = x^4 - 3x^3 - 5x + 2$ che, essendo di quarto grado, ha cinque coefficienti (uno è zero), va introdotto come quintupla di numeri, con i suoi coefficienti dalla potenza più alta alla più bassa:

```
» p=[1 -3 0 -5 2]
```

Dato che i polinomi possono essere considerati come funzioni, esiste un comando per calcolare la funzione polinomiale $p(x) = x^4 - 3x^3 - 5x + 2$ in qualsiasi punto x . Per esempio per calcolare $p(1)$ si scrive

```
» polyval(p,1)
ans =
    -5
```

E in effetti e $p(1) = -5$. Per moltiplicare due polinomi occorre usare la funzione **conv** (convoluzione). Per esempio, se $d = 3x^2 - 5x + 2$, il prodotto $p \cdot d$ si trova così:

```
» d=[3 -5 2];
» conv(p,d)
ans =
     3   -14    17   -21    31   -20     4
```

e in effetti $p \cdot d = 3x^6 - 14x^5 + 17x^4 - 21x^3 + 31x^2 - 20x + 4$.

La somma di polinomi è un po' più complicata: si sommano i vettori che li rappresentano, ma occorre che abbiano lo stesso grado. Per esempio, se $d = 3x^2 - 5x + 2$, per sommare p e d è necessario usare il polinomio ausiliario $d1$ con gli stessi coefficienti di d , ma grado uguale a a quello di p, cosa che si ottiene aggiungendo degli zeri.

```
» d1=[0 0 3 -5 2];
» p+d1
ans =
     1    -3     3   -10     4
```

Si può anche eseguire la divisione con resto di due polinomi. La funzione relativa è **deconv** (deconvoluzione). Dato che però il risultato è dato da due oggetti, il quoto **q** e il resto **r** , occorre una sintassi leggermente diversa:

```
» [q,r]=deconv(p,d)
q =
    0.3333   -0.4444   -0.9630
r =
    0.0000   -0.0000   -0.0000   -8.9259    3.9259
```

Si noti che `r` , nonostante abbia grado minore di quello di `d` , ha virtualmente grado uguale a quello di `p` in modo che si possa facilmente verificare che $q \cdot d + r = p$. Per eseguire questo calcolo occorre il comando

```
» conv(q,d)+r
```

con cui si ottiene p (forse non esattamente, perché possono esserci arrotondamenti).
Le radici di un polinomio si trovano con la funzione `roots` . Per calcolare le radici di `p` si scrive

```
» roots(p)
```

e si ottiene una matrice colonna i cui elementi sono le quattro radici di p.

```
ans =
   3.3848
  -0.3788 + 1.2006i
  -0.3788 - 1.2006i
   0.3728
```

Per esempio, per trovare le radici terze di $2 + 2i$ si può anche usare la funzione:

```
» roots([1 0 0 -2-2*i])
```

cioè trovare le radici del polinomio $x^3 - 2 - 2i$. Ma non è detto che l'ordine delle radici sia quello per argomento.

NORME E PRODOTTI TRA VETTORI

La funzione `norm` calcola la norma euclidea di un vettore.

```
» norm([1 2])
ans =
    2.2361
```

Vale la pena di osservare che la funzione `norm` applicata, invece che a un vettore, a una matrice quadrata `a` ne calcola la **norma-2**, il più grande **valore singolare** della matrice.
I valori singolari di una matrice sono le radici quadrate degli autovalori della matrice simmetrica `a'*a` (che ha tutti autovalori non negativi).
La norma-2 è utile nello studio della stabilità, rispetto alle variazioni dei suoi elementi, delle soluzioni di un sistema lineare associato alla matrice.

• Prodotto scalare: Il calcolo del prodotto scalare di due vettori **colonna** `x` e `y` si può ricondurre a un prodotto di matrici mediante il comando

```
» x'*y
```

Comunque esiste la funzione

```
» dot(x,y)
```

• Per il prodotto vettoriale $\mathtt{x} \wedge \mathtt{y}$ tra vettori **a tre componenti** esiste la funzione

```
» cross(x,y)
```

AUTOVALORI E AUTOVETTORI

Mediante complessi algoritmi, **MatLab** è in grado di determinare gli autovalori e gli autovettori di una matrice. Se `a` è una matrice quadrata $\mathtt{n} \times \mathtt{n}$, la funzione

```
» eig(a)
```

produce una matrice colonna con tutti gli `n` autovalori di `a` (`eig` sta per eigenvalues: autovalori in inglese). Naturalmente è assai facile che, anche in una matrice reale, alcuni degli autovalori non siano reali. In questo caso alcuni degli elementi della matrice degli autovalori saranno complessi. Per esempio

```
» a =[1 2 3 ; 4 0 1 ; 1 2 -1];
» eig(a)
 ans =
   4.5810
  -2.2905 + 1.3187i
  -2.2905 - 1.3187i
```

Gli autovalori sono **grosso modo** in ordine decrescente di modulo, ma non sempre, dato che l'ordine in cui sono calcolati dipende dal complesso algoritmo effettuato da **MatLab** .
Se si desiderano anche gli autovettori, oltre gli autovalori, la sintassi della funzione va modificata, in modo da consentire l'output di due risultati, nel modo seguente:

```
» [p,d]=eig(a)
p =
   0.6645    -0.3953 - 0.2599i    -0.3953 + 0.2599i
   0.6577     0.1037 + 0.6559i     0.1037 - 0.6559i
   0.3548     0.4787 - 0.3259i     0.4787 + 0.3259i
d =
   4.5810          0                    0
        0  -2.2905 + 1.3187i             0
        0          0            -2.2905 - 1.3187i
```

In questo modo si ottengono perciò due matrici `p` e `d` tali che `a p = p d`. La matrice `d` è diagonale e ha sulla diagonale gli autovalori. Ogni colonna della matrice `p` contiene le coordinate di un autovettore relativo all'autovalore della colonna corrispondente di `d`. Se la matrice `a` è diagonalizzabile, allora `p` è invertibile. Se invece, come può capitare, la matrice `a` non è diagonalizzabile, allora la matrice `p` non è invertibile e le colonne corrispondenti agli autovalori con molteplicità maggiore di 1 possono essere uguali o proporzionali, se gli autospazi sono deficitari. È possibile anche ottenere il polinomio caratteristico di una matrice quadrata `a` di ordine `n` mediante la funzione

```
» poly(a)
```

che produce un vettore di lunghezza `n+1` avente come elementi i coefficienti del polinomio caratteristico della matrice. I coefficienti sono elencati dalla potenza più alta alla più bassa. Il primo coefficiente è sempre 1.

È interessante sapere che il comando `roots` per determinare le radici di un polinomio `p` equivale in realtà al comando `eig(compan(p))`.
La matrice **compagna** di un polinomio `p`, che si ottiene col comando `compan` è una semplice matrice quadrata che ha come polinomio caratteristico proprio `p`.
Quindi, per determinare le radici di un polinomio, **MatLab** calcola gli autovalori di una matrice e non viceversa, come poteva essere naturale pensare.

E' utile notare che **MatLab** sceglie sempre autovettori (le colonne della matrice `p`) di modulo 1. Nel nostro esempio la cosa può essere verificata per esempio coi comandi

```
» v=p(:,2); norm(v)
 ans =
   1.0000
```

ll primo comando estrae la seconda colonna dalla matrice `p` (nel nostro esempio un autovettore dell'autovalore $-2.2905 + 1.3187i$), il secondo ne calcola la norma euclidea.
Nel caso di autovalori di molteplicità superiore a 1, il programma non calcola basi ortonormali per gli autospazi relativi, anche perché difficilmente viene rilevato che un autovalore abbia molteplicità maggiore di 1, essendoci errori di arrotondamento. Le basi però sono ortonormali nel caso di matrice simmetrica. In questo caso la matrice `p` ottenuta mediante la funzione `eig` a due output è una matrice **ortogonale**, cioè una matrice in cui tutte le colonne hanno modulo 1 e sono a due a due ortogonali (anche non perfettamente, a causa di errori da approssimazione).

FATTORIZZAZIONE QR E ORTONORMALIZZAZIONE

La fattorizzazione QR di una matrice quadrata invertibile `a` è ottenibile con la seguente funzione a due output

```
» [q,r]=qr(a)
```

Con questa funzione si ottengono due matrici `q` e `r` tali che `q*r=a`.
La prima matrice `q` è ortogonale e le sue colonne sono un'**ortonormalizzazione** dei vettori colonna di `a` ottenuta però, non con l'algoritmo di Gram-Schmidt, ma con il più efficiente algoritmo di Householder e quindi leggermente differente da quella ottenibile con Gram-Schmidt.
La seconda matrice `r` è triangolare superiore e rappresenta la matrice di passaggio tra le due.

È possibile calcolare con la stessa funzione la fattorizzazione QR di una matrice `a` di qualunque formato `m` $\times$ `n` e di qualunque rango (anche se di solito si ha `m>n` e la matrice ha caratteristica pari al numero di colonne).
La matrice `q` ottenuta con questa funzione è quadrata e ortogonale, la matrice `r` ha lo stesso formato di `a` ed è triangolare superiore.

La fattorizzazione economica QR di una matrice qualunque si ottiene in questo modo:

```
» [q1,r1]=qr(a,0)
```

La matrice `q1` ottenuta in questo caso ha lo stesso formato di `a` e ha le colonne a due a due ortogonali e di modulo 1, e quindi è una **ortonormalizzazione** delle colonne di `a`, mentre `r1` è quadrata `n` $\times$ `n` ed è triangolare superiore.
La `q1` ottenuta nel modo economico è costituita dalle prime `n` colonne della `q` ottenuta col primo comando, mentre `r1` è ottenuta eliminando le ultime righe di `r` (che, se `m>n`, sono sempre nulle).

Osserviamo comunque che le colonne delle matrici `q` ottenute colla funzione `qr`, a causa degli errori di approssimazione, sono di solito non perfettamente ortonormali, ma hanno comunque un prodotto scalare molto piccolo (dell'ordine di 10^{-15}) e un modulo molto prossimo a 1 (sempre a meno di 10^{-15} circa).

GLI OPERATORI RELAZIONALI

Gli operatori relazionali che consentono di confrontare gli elementi di due matrici sono i seguenti:

- `==` uguale (da non confondersi con il segno `=` che serve per assegnare valori alle variabili)
- `~=` diverso (il contrario del precedente)
- `<` minore (strettamente)
- `>=` maggiore o uguale (il contrario del precedente)
- `>` maggiore (strettamente)
- `<=` minore o uguale (il contrario del precedente)

Dei sei operatori, quindi tre sono esattamente contrari degli altri tre.
Si tenga presente che gli operatori relazionali sono operazioni binarie esattamente come la somma e il prodotto, e danno quindi luogo a un risultato.
La differenza è che il risultato può assumere solo due valori: `0` (se la relazione è falsa) e `1` (se la relazione è vera) e che il risultato è una matrice di tipo particolare, la matrice di classe **logical**.
Le matrici di classe logical sono utili in alcuni comandi di selezione di elementi di una matrice e in questo si distinguono da altre matrici formate solo di zeri e di uni, ma non ottenute da operatori relazionali.
L'operatore relazionale viene applicato ad ogni elemento delle matrici che si confrontano. Le matrici devono perciò avere lo stesso formato, oppure una di esse deve essere uno scalare.
Per esempio:

```
» x=[0 3 8 5 7]; y=[5 0 -1 6 7];
» x<y
ans =
     1     0     0     1     0
» x<=y
ans =
     1     0     0     1     1
» x==y
ans =
     0     0     0     0     1
» x<=5
ans =
     1     1     0     1     0
```

Gli operatori relazionali vengono usati soprattutto nei test condizionali che seguono di solito il comando `if`.
Comunque le matrici di classe logical ottenute in questo caso possono anche servire per scegliere gli elementi che interessano di una matrice. Per esempio mediante il comando

```
» x(x<=5)
ans =
     0     3     5
```

si ottiene la lista degli elementi di `x` che sono minori o uguali a 5. Il comando precedente può essere applicato anche a una matrice `a` che non sia un vettore, ma in questo caso si ottiene una sottomatrice del flattening di `a`.

RICERCA DI ELEMENTI IN UNA MATRICE

I comandi che fanno uso delle matrici logical permettono di ottenere la lista degli elementi di una matrice che soddisfano particolari criteri, ma non la loro posizione nella matrice.
A questo scopo c'è però la funzione `find` che **elenca gli indici degli elementi non nulli** di una matrice. Per esempio.

```
» find(x)
ans =
     2     3     4     5
```

elenca gli indici degli elementi non nulli di `x`. Combinando la funzione `find` con gli operatori relazionali si può ottenere la lista degli indici degli elementi di una matrice che soddisfano particolari criteri. Per esempio

```
» find(x<=5)
ans =
     1     2     4
```

elenca gli indici degli elementi di `x` che sono minori o uguali a 5. Sono pertanto equivalenti le due funzioni seguenti (la prima è quella vista sopra).

```
» x(x<=5)
» x(find(x<=5))
```

Osserviamo che se si lavora con una matrice `a` che non sia un vettore, la funzione `find` elenca gli indici degli elementi non nulli del flattening della matrice `a(:)`.

Se si vogliono i veri indici degli elementi non nulli di una matrice **a** , la sintassi è la seguente:

```
» [id,jd]=find(a)
```

La matrice colonna **id** fornisce gli indici di riga e la matrice **jd** gli indici di colonna degli elementi cercati. Per esempio

```
» a=[0 5 2; 6 1 7; 0 1 -1];
» [id,jd]=find(a>5)
id =
     2
     2
jd =
     1
     3
```

Gli elementi maggiori di 5 della matrice **a** sono quelli di indici $(2,1)$ e $(2,3)$.

• Il massimo e il minimo elemento di un vettore **x** si ottengono con le funzioni

```
» max(x)
» min(x)
```

Se si vogliono conoscere anche le posizioni del massimo e del minimo, la sintassi è la seguente

```
» [m,id]=max(x)
```

e si ottengono **m** , il valore del massimo e **id** l'indice del massimo (il primo indice se più di uno degli elementi di **x** è massimo). Analogamente per il minimo.

• Per riordinare gli elementi di un vettore dal minimo al massimo si usa la funzione **sort** . Per esempio

```
» x=[0 8 3 7 5];
» sort(x)
ans =
     0     3     5     7     8
```

Se si vogliono conoscere anche le posizioni degli elementi in ordine di grandezza, la sintassi è

```
» [x1,id]=sort(x)
x1 =
     0     3     5     7     8
id =
     1     3     5     4     2
```

La matrice **id** fornisce le posizioni in cui si trovavano in **x** gli elementi della matrice riordinata.
Questo è utile per esempio, se si ha anche una matrice **y** i cui elementi sono in corrispondenza con quelli di **x** .
È possibile riordinare **y** mantenendo la corrispondenza con **x** , mediante la matrice **id** .

```
» x=[0 8 3 7 5]; y=[0 18 33 27 15];
» [x1,id]=sort(x);
» y(id)
ans =
     0    33    15    27    18
```

• Le funzioni **any** e **all** : la prima ritorna 1 se almeno un elemento della matrice è non nullo e 0 in caso contrario, la seconda ritorna 1 se tutti gli elementi della matrice sono non nulli e 0 in caso contrario. Le due funzioni sono solitamente usate in congiunzione con gli operatori relazionali.
Per comprenderne l'uso conviene osservare attentamente il seguente esempio:

```
» x=[1 2 3] ; y=[1 5 7] ; z=[10 11 12];
» any(x==y)
ans =
     1
```

```
» all(x==y)
ans =
     0
```

```
» any(x==z)
ans =
     0
```

```
» any(x~=y)
ans =
     1
```

```
» all(x~=y)
ans =
     0
```

```
» all(x~=z)
ans =
     1
```

```
» all(x<y)
ans =
     0
```

```
» all(x<z)
ans =
     1
```

INTRODUZIONE ALLA GRAFICA IN 2D

MatLab ha grosse capacità grafiche. Descriviamo il comando elementare **plot(x,y)** che è alla base di comandi più avanzati. Il comando **plot** ha in pratica la funzione di disegnare una spezzata nel piano. Per esempio per disegnare la spezzata che congiunge i punti di coordinate $(1,1)$ $(2,2)$ $(-1,1)$ $(3,0)$ $(2,1)$ occorre formare due matrici riga **x** e **y** , la prima con le ascisse dei cinque punti, la seconda con le ordinate.

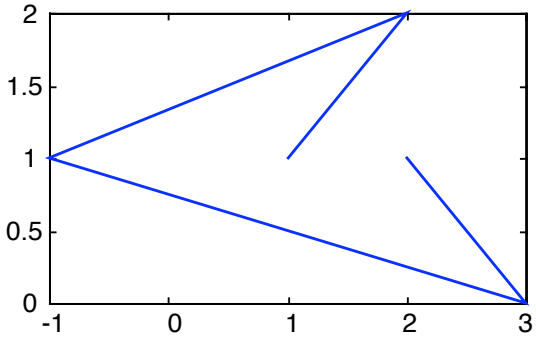
```
x=[1 2 -1 3 2];
y=[1 2 1 0 1];
```

Dopodiché il comando

```
» plot(x,y)
```

disegna la spezzata che appare nella finestra grafica. Se la finestra grafica non appare, la si può richiamare col comando **shg** (showgraph)

```
» shg
```



Il grafico appare non monometrico, senza assi e solo circondato da una cornice graduata avente come limiti i minimi e i massimi delle ascisse e delle ordinate dei punti. Nell'esempio le scale del grafico hanno passo 0.5, ma dipendono solo dalla grandezza della finestra grafica.
Altri comandi grafici consentono di impostare opzioni differenti.
Il comando

```
» axis=equal
```

consente per esempio di avere un grafico con assi monometrici.

Per far sì che la finestra grafica, anziché essere compresa tra i minimi e i massimi, sia delimitata da un qualsiasi rettangolo $[x_0, x_1] \times [y_0, y_1]$ occorre il comando

```
» axis([x0 x1 y0 y1])
```

GRAFICO UNA CURVA NEL PIANO

Cominciamo col disegnare il grafico di una funzione **y = f(x)** in un intervallo **[a,b]** .
Osserviamo che ciò equivale a disegnare la spezzata che ha come vertici vari punti del grafico.
Di solito conviene suddividere l'intervallo in parti uguali scegliendo un passo più o meno ampio a seconda del dettaglio che si vuole ottenere.
Per esempio, se l'intervallo è $[-3, 2]$ e il passo scelto è 0.1 allora si può porre

```
» x=-3:0.1:2;
```

e si ottiene una matrice riga con 51 elementi.
Se, a titolo di esempio, vogliamo disegnare il grafico della funzione **y = log (x^3-3x+2)** nell'intervallo suddetto, occorre tabulare la funzione, cioè calcolarla in tutti i punti scelti dell'intervallo in questione.
Costruiamo quindi una matrice 1×51 **y** con tutti i valori corrispondenti ai valori di **x** .

```
» y=log(x.^3-3*x+2);
```

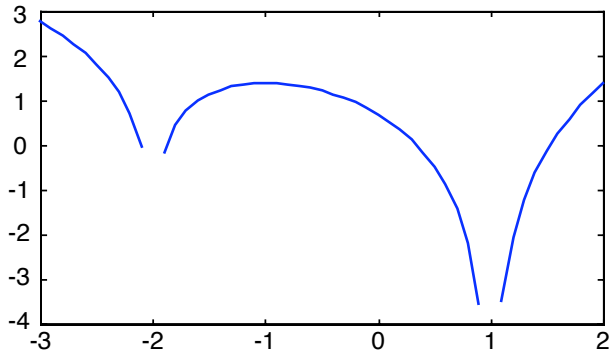
Notiamo il segno **.^** (potenza puntuale) per la potenza. Invece per moltiplicare **x** per lo scalare **3** non occorre il segno **.*** e anche la somma per uno scalare si ottiene semplicemente col segno **+** . Avendo posto il segno **;** la matrice **y** non è riportata sul display, comunque compare la scritta

```
Warning:  Log of zero.
```

che significa: Attenzione: log di zero. Questo perché tra i punti in cui è calcolata la funzione ci sono -2 e 1 dove la funzione non è definita.
In realtà la funzione non sarebbe definita anche tra -3 e -2 dato che si tratta del logaritmo di un numero negativo, ma in questo caso l'effetto è solo quello di ottenere valori non reali.
Ora possiamo procedere a disegnare il grafico con

```
» plot(x,y)
```

e si ottiene nella finestra grafica il disegno seguente:

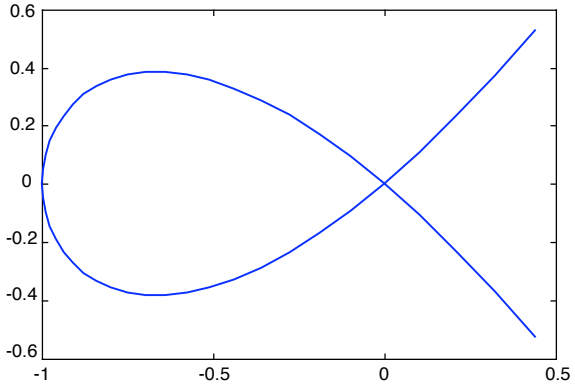


Ovviamente la funzione non viene disegnata tra -2.9 e -1.9 e tra 0.9 e 1.1, dato che la matrice **y** non è definita in -2 e in 1. Per quanto riguarda il pezzo di grafico tra -3 e -2 dove la funzione non è definita, esso è in realtà il grafico della funzione $y = \mathrm{Re}\,(\log(x^3 - 3x + 2))$ (parte reale). Se si lavora con funzioni di variabile reale il pezzo va trascurato.

È altrettanto semplice disegnare una curva data mediante la sua rappresentazione parametrica, creando l'array **x** delle ascisse e quello **y** delle ordinate.
Per esempio per raffigurare la curva avente la rappresentazione parametrica a lato, va innanzitutto scelta la porzione più significativa, quella che si ottiene per $t \in [-1.2, 1.2]$. Il grafico viene generato dai seguenti comandi

$$\begin{cases} x = t^2 - 1 \\ y = t(t^2 - 1) \end{cases}$$

```
» t=-1.2:.05:1.2;
» x=t.^2-1
» y=t.*(t.^2-1)
» plot(x,y)
```



Osserviamo che, mentre è semplice disegnare una curva assegnata mediante rappresentazione parametrica, assai più complesso è il disegno di una curva piana nota attraverso la sua rappresentazione cartesiana, se una delle due coordinate non è facilmente esplicitabile. Qualche indicazione in proposito verrà data nel paragrafo sulla grafica tridimensionale.

GRAFICA IN 2D: OPZIONI E GRAFICI MULTIPLI

Il comando **plot(x,y)** ha molte opzioni. Esiste la capacità di cambiare l'aspetto del grafico (che, ricordiamo, è sempre una spezzata). Mediante il comando

```
» help plot
```

si ottiene l'elenco delle opzioni principali.

Le opzioni vanno assegnate mediante una stringa di caratteri racchiusa da due apici. Per esempio

```
» plot(x,y,'r:')
```

impone che il grafico sia rosso (`'r'`) e punteggiato (`':'`). Le due opzioni vanno combinate in un'unica stringa.

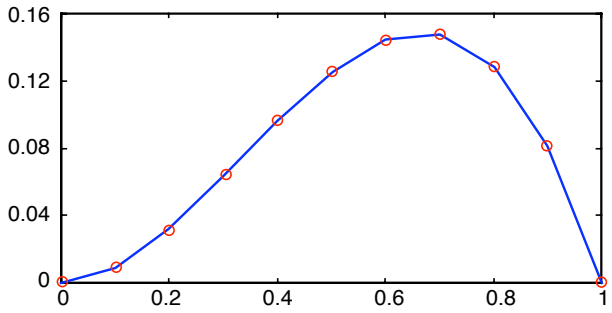
Esiste la possibilità di sovrapporre due grafici distinti con un solo comando. Per esempio

```
» plot(x,y,'r:',x1,y1,'b--')
```

disegnerà due spezzate, la prima rossa e punteggiata relativa alle array **x** e **y** e la seconda blu e tratteggiata relativa alle array **x1** e **y1** . I grafici possono essere più di due e dotati di opzioni o no.
Un altro esempio è il seguente:

```
x=0:.1:1;
y=x.^2-x.^3;
» plot(x,y,x,y,'or')
```

Viene disegnata la stessa funzione, la prima volta col colore e il tratto di default (blu, tratto continuo), dato che non sono state specificate opzioni, la seconda volta senza tratto, ma coi vertici segnati mediante pallini (`'o'`) di colore rosso (`'r'`).



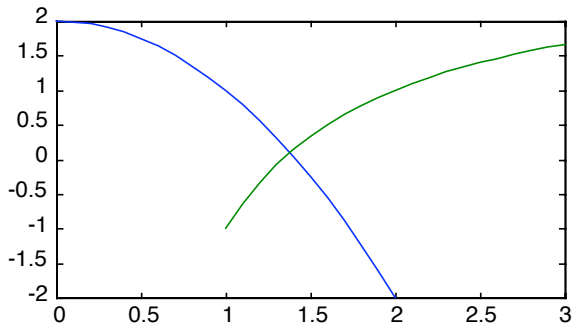
Un altro modo di sovrapporre due grafici differenti è quello di usare il comando **hold** .
Infatti, normalmente ogni comando **plot** cancella il precedente grafico e disegna il nuovo. Ma scrivendo

```
» hold on
```

si disabilita questa opzione e i grafici successivi verranno sovrapposti a quelli già esistenti. Per esempio possiamo disegnare due curve differenti anche come dominio e sovrapporre i grafici

```
» x=0:.1:2; y=2-x.^2;
» plot(x,y)
» x1=1:.1:3; y1=(3*x1-4)./(x1);
» hold on
» plot(x1,y1)
```

Come risultato abbiamo due grafici sovrapposti



Per ripristinare l'opzione che il grafico venga cancellato ad ogni **plot** si scrive

```
» hold off
```

Comunque per cancellare la finestra grafica, indipendentemente dallo status di **hold** basta il comando

```
» clf
```

C'è anche la posibilità di disegnare più grafici, non sovrapposti, ma affiancati nella stessa finestra, usando il comando **subplot** .
In pratica è possibile dividere la finestra in una matrice $n \times m$ in cui ogni elemento della matrice è un grafico.

Come esempio disegniamo 6 grafici disposti in due righe e tre colonne, usando le due funzioni sopra definite e una

terza definita da **x2** e **y2** , dove **x2=0:.1:pi;** e **y2=sin(x2);** .
Il comando **subplot** per creare il primo grafico stabilisce innanzitutto che la matrice dei grafici è 2×3 e che il grafico ottenuto col seguente **plot** è il primo

```
» subplot(2,3,1) , plot(x,y)
```

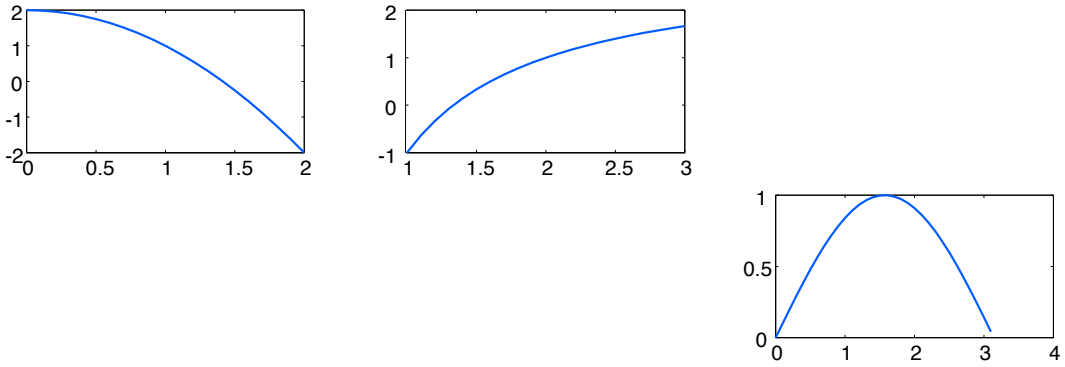
Il secondo grafico si ottiene con

```
» subplot(2,3,2) , plot(x1,y1)
```

Per l'ultimo grafico (il sesto), il comando è

```
» subplot(2,3,6) , plot(x2,y2)
```

Il risultato è il seguente. Avendo disegnato solo tre grafici, rimane posto per altri tre grafici

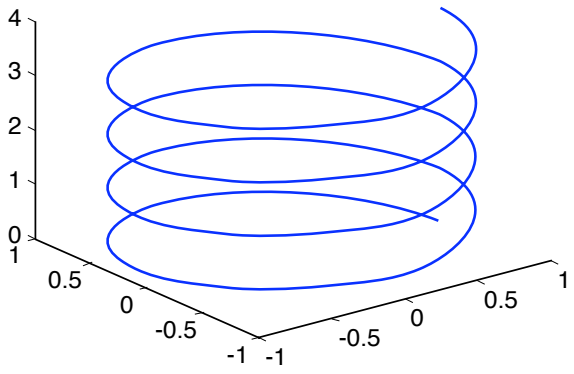


GRAFICA IN 3D: LINEE

Il comando **plot3** è perfettamente analogo al comando **plot** , ma consente di usare tre coordinate e ottenere il disegno di una spezzata elementare in 3 dimensioni.

Per esempio in questo modo si ottiene una porzione di elica cilindrica a passo costante

```
» t=0:.05:8*pi;
» x=cos(t) ;y=sin(t); z=t/(2*pi);
» plot3(x,y,z)
```



Ovviamente si tratta della raffigurazione assonometrica di un oggetto tridimensionale.
Se non si forniscono altre indicazioni, **MatLab** sceglie come angolo di visione (azimuth) rispetto agli assi x, y l'angolo -37.5^0 (la misura dell'angolo di azimuth parte dalla parte negativa dell'asse y), mentre l'altezza (elevazione) è di 30^0.
Nella figura, la retta graduata a destra ha la direzione dell'asse x, quella a sinistra ha la direzione dell'asse y e quella verticale dell'asse z. Non sono esattamente gli assi coordinati, perché l'origine delle coordinate è al centro dell'elica.

È possibile cambiare il punto di vista mediante il seguente comando (con **az** e **el** in gradi)

```
» view(az,el)
```

Scegliendo come angoli rispettivamente 0^0 e 90^0 si ottiene la vista dall'alto con gli assi x e y disposti nel modo solito, ovvero un grafico bidimensionale.

Comunque vari pulsanti nella finestra grafica consentono di cambiare interattivamente il punto di vista del disegno.

GRAFICA IN 3D: SUPERFICI

La rappresentazione di superfici è uno degli aspetti più spettacolari di **MatLab** . Le opzioni disponibili sono

parecchie. Illustriamo qui solo gli aspetti base, lasciando alla documentazione di **MatLab** il compito di descrivere tutte le innumerevoli possibilità di rappresentazione.

Vediamo quindi come è possibile disegnare la superficie grafico di una funzione di due variabili $z = f(x, y)$ in un dominio rettangolare $[x_0, x_m] \times [y_0, y_n]$.

È necessario innanzitutto calcolare la funzione f in vari punti del dominio.
Quindi occorre costruire una griglia dividendo gli intervalli $[x_0, x_m]$ e $[y_0, y_n]$ in un certo numero di punti.
Si avranno le successioni $x_0, x_1, x_2, ..., x_m$ e $y_0, y_1, y_2, ..., y_m$.

Di solito queste divisioni sono uniformi e ottenute scegliendo passi **h** e **k** e quindi generate con comandi tipo

```
» x=x0:h:xm;
» y=y0:k:yn;
```

La funzione dovrà essere calcolata nei punti aventi queste ascisse e queste ordinate. Le coordinate di questi punti possono formare una matrice.

$$\begin{array}{llll} (x_0,y_0) & (x_0,y_1) & (x_0,y_2) & \cdots \quad (x_0,y_n) \\ (x_1,y_0) & (x_1,y_1) & (x_1,y_2) & \cdots \quad (x_1,y_n) \\ \cdots & \cdots & & \ddots \\ (x_m,y_0) & (x_m,y_1) & (x_m,y_2) & \cdots \quad (x_m,y_n) \end{array}$$

In realtà le matrici sono due: una per le ascisse, una per le ordinate dei punti della griglia. Esiste una funzione che genera facilmente queste due matrici partendo dai vettori **x** e **y** .
Dato che la funzione ha due output, la sintassi sarà

```
» [xx,yy]=meshgrid(x,y)
```

e si ottengono due matrici **xx** e **yy** di formato $(m + 1) \times (n + 1)$.
Da notare che nella matrice **xx** tutte le colonne sono uguali, mentre nella matrice **yy** tutte le righe sono uguali.
Ora è possibile calcolare la funzione $f(x, y)$ in tutti i punti della griglia e generare una terza matrice $(m+1)\times(n+1)$ che chiameremo per esempio **zz**

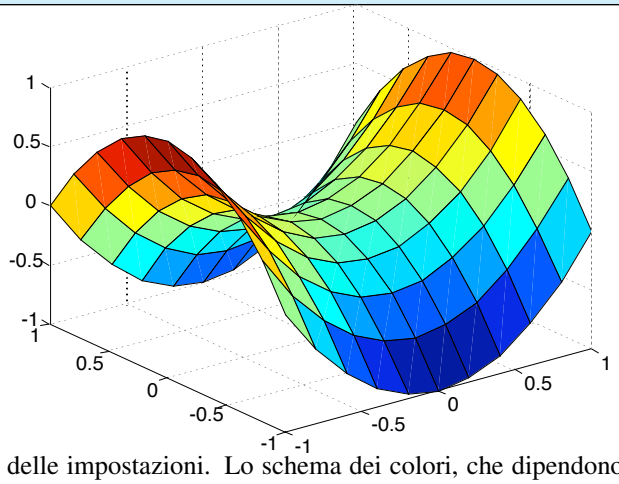
```
» zz=...% funzione di xx,yy
```

Per ottenere il grafico sono disponibili due comandi

```
» mesh(xx,yy,zz)
» surf(xx,yy,zz)
```

Il comando **mesh** rappresenta la funzione mediante una maglia costituita da quadrilateri.
Il comando **surf** rappresenta la funzione sempre mediante un maglia costituita da quadrilateri però riempiti di colore. I colori, senza ulteriori indicazioni, dipendono dalla quota dei singoli quadrilateri.
Il classico esempio è il paraboloide iperbolico nel dominio $[-1, 1] \times [-1, 1]$ generato coi seguenti comandi

```
» x=-1:.2:1;
» y=-1:.2:1;
» [xx,yy]=meshgrid(x,y)
» zz=xx.^2-yy.^2;
» surf(xx,yy,zz)
```



È possibile cambiare molte delle impostazioni. Lo schema dei colori, che dipendono dalla quota, inizialmente è quello predefinito col nome " **jet** ", ma per esempio col comando

```
» colormap copper
```

i quadrilateri vengono riempiti con una tonalità "rame".
Gli schemi di colore predefiniti sono **gray** , **hot** , **cool** , **bone** , **copper** , **pink** , **flag** , **prism** ,

jet , **hsv** . Mediante **help colormap** si ottengono le istruzioni per generare qualunque schema di colori.
Anche l'aspetto della superficie può essere variato col comando **shading** . Le tre opzioni sono

```
» shading flat
» shading interp
» shading faceted
```

Il primo comando toglie le maglie e mostra solo i colori dei quadrilateri.
Il secondo comando toglie le maglie e interpola i colori dando così un aspetto calibrato piacevole alla superficie.
Il terzo ripristina l'opzione iniziale e quindi mostra sia le maglie che le colorazioni dei quadrilateri.

Come nel comando **plot3** , il punto di vista iniziale ha azimuth -37.5^0 e elevazione 30^0, ma può essere cambiato con **view** o mediante i pulsanti della finestra grafica.

GRAFICA IN 2D E 3D: LINEE DI LIVELLO E CURVE IMPLICITE

Una volta definita una funzione di due variabili mediante tre matrici **xx** , **yy** , **zz** come sopra, è possibile tracciare il grafico bidimensionale delle linee di livello di **zz** mediante il comando

```
» contour(xx,yy,zz)
```

che disegna nel piano (x, y) le proiezioni di alcune delle linee di livello.

Se si vogliono alcune ben precise linee di livello, il comando è

```
» contour(xx,yy,zz,[q0 q1 ... qn])
```

che disegnerà le linee di livello alle quote **q0,q1,...,qn**

Sono disponibili alcune delle opzioni grafiche dei grafici bidimensionali, tipo **'r'** e **':'**

In particolare, volendo una sola linea di livello, alla quota **q** , in colore blu, il comando dovrà essere

```
» contour(xx,yy,zz,[q q],'b')
```

La quota va indicata precisamente con **[q q]** e non semplicemente con **q** .

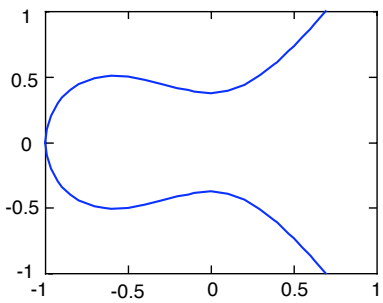
Il comando **contour** fornisce un grafico bidimensionale, ovvero le proiezioni sul piano $[xy]$ delle linee di livello. È però possibile disegnare le linee di livello in 3D, posizionandole alla loro vera quota. Questo fornisce un altro modo di rappresentare una superficie in 3D diverso da quello delle maglie. Il comando è

```
» contour3(xx,yy,zz)
```

Le opzioni per avere le linee desiderate sono le stesse di **contour** .
Normalmente per avere un buon disegno in questo modo è bene specificare un numero abbastanza alto di linee di livello.

Usando il comando **contour** è anche possibile disegnare, con una certa approssimazione, una curva piana assegnata mediante equazione implicita $f(x, y) = 0$.
Occorrerà infatti creare la funzione **zz** , come per disegnare il grafico della funzione $z = f(x, y)$, e disegnare la curva di livello a quota 0. Per esempio, per disegnare la porzione della curva di equazione $8x^3 - 7y^2 + 7x^2 + 1 = 0$ nel quadrato $[-1, 1] \times [-1, 1]$, si possono usare i comandi

```
» x=-1:.1:1;
» y=-1:.1:1;
» [xx,yy]=meshgrid(x,y);
» zz=8*xx.^3-7*yy.^2+7*xx.^2+1;
» contour(xx,yy,zz,[0 0],'b')
```



Da tenere presente che non si può pretendere la precisione assoluta da un grafico di questo tipo e spesso nei punti critici il disegno della curva può essere non del tutto affidabile.
Inoltre il problema spesso consiste nell'individuare un rettangolo contenente la porzione più interessante della curva.

INTERFACCIA UTENTE E STRINGHE

Per visualizzare su schermo il valore di una variabile **a** senza farla precedere dalla scritta **a=** , si usa il comando **disp** (abbreviazione di *display*). Per esempio

```
» a=pi/2;
» disp(a)
    1.5708
```

Il comando **disp** funziona per variabili di qualunque classe e di qualunque formato, ma è in grado di scrivere solo una variabile alla volta.
Per visualizzare con un solo comando due variabili già definite **a** e **b** , occorre quindi inserirle in un'unica variabile. Per esempio se **a** e **b** fossero scalari, si potrebbe scrivere

```
» disp([a b])
» disp([a;b])
```

per visualizzarle in riga o in colonna.

Benché **MatLab** lavori principalmente con numeri, esiste la possibilità di trattare stringhe alfanumeriche, con le quali è anche possibile gestire una buona interfaccia con l'utente.
Quindi oltre alle variabili di classe **double** (numeri in doppia precisione) e alle variabili di classe **logical** (generate dagli operatori relazionali), esistono le variabili di classe **char**, cioè stringhe di caratteri. Per esempio

```
» s='ciao'
```

definisce una variabile **s** di classe **char**.
Le stringhe vanno sempre racchiuse tra due apici singoli (e non tra virgolette doppie).

Dato che non è possibile combinare assieme variabili di classe **char** con variabili di altra classe, per scrivere su una stessa riga una stringa di caratteri e un numero occorre qualche accorgimento.
Per esempio per scrivere la frase "La variabile a vale 6.2832" (dove $a = 2\pi$) occorre convertire **a** in stringa mediante la funzione **num2str** (che si legge "number to string") che trasforma un numero in una stringa di caratteri.
Quindi si può formare una matrice riga **s** costituita da stringhe (quindi di classe **char**), che affianchi la frase e il numero **a** convertito in stringa.

```
» a=2*pi;
» s=['La variabile a vale ',num2str(a)];
» disp(s)
La variabile a vale 6.2832
```

La funzione **num2str** di norma visualizza al massimo 4 cifre dopo il punto decimale, ma ha anche un argomento opzionale per il massimo numero di decimali. Usando opportunamente questa funzione è possibile ottenere una buona interfaccia utente.
Illustriamo la procedura con questo esempio

```
» a=[pi/2 , 2.3];
» s=['La matrice a e'' formata da ', num2str(a(1,1),5) , ...
' e ',num2str(a(1,2))])
» disp(s)
La matrice a e' formata da 1.57080 e 2.3
```

Commento:
- Viene definita una matrice 1×2 **a** .
- Si definisce la matrice 1×4 **s** i cui elementi sono quattro stringhe.
- Le quattro stringhe di **s** vanno racchiuse tra parentesi quadre **[ ]** e separate da virgole (o spazi).
- La prima stringa è "La matrice a e' formata da ".

  La stringa contiene un apice. Per inserirlo nella stringa, senza che venga considerato come il delimitatore della stringa, l'apice va raddoppiato.
- La seconda stringa è il numero $\pi/2$ (cioè $a(1,1)$) trasformato in stringa con 5 cifre decimali con **num2str** .

  Dato che la definizione di **s** non sta per intero nella riga, è possibile continuarla nella riga seguente andando a capo mediante la serie di tre punti **...**
- La terza stringa è " e ", cioè la congiunzione e racchiusa tra due spazi.
- La quarta stringa è il numero 2.3 (cioè $a(1,2)$), trasformato in stringa mediante **num2str** .
- Viene dato il comando **disp** il cui argomento è racchiuso tra parentesi tonde.

La funzione **num2str** risolve in modo relativamente elementare alcuni problemi di visualizzazione, ma la soluzione migliore è quella di usare la funzione **sprintf** che è mutuata direttamente dal linguaggio C.
Le opzioni della funzione sono tantissime e in molti casi piuttosto complicate, per cui rimandiamo ai manuali di C per la descrizione di tutte le possibilità, limitandoci a un esempio con le opzioni più semplici.

La sintassi è

```
» sprintf('xxxx', elenco variabili)
```

dove **xxxx** è una stringa, detta stringa di formato. Questa stringa fornisce, con apposito codice, le istruzioni per visualizzare le variabili che seguono. Dopo la virgola va dato un elenco di una o più variabili, separate da virgole.
Esempio

```
» x=2; y=[3.56890988 , 3]; z='questi sono numeri reali'
» s=sprintf('Si ha x = %d e y= %f %f\n%s',x,y,z);
» disp(s)
Si ha x = 2 e y= 3.568910 3.000000
questi sono numeri reali
```

La stringa di formato di **sprintf** comprende
• due stringhe di testo: "Si ha x= " , " e y= "
• la stringa di formato **%d** che indica un numero intero e si applicherà alla prima variabile in elenco, cioè **x** .
• le due stringhe di formato **%f** che indicano un numero decimale e si applicheranno ai due elementi della seconda variabile, cioè **y** .
• la stringa speciale **\n** che indica ritorno a capo.
• la stringa di formato **%s** che indica una stringa e si applicherà alla terza variabile, cioè **z** .
Osservazioni:
• **sprintf** è una funzione che restituisce una stringa, in questo caso **s** , che può essere visualizzata mediante il comando **disp** .
• In mancanza di altre opzioni, un numero decimale viene scritto con sei cifre decimali.
• Il numero 2 viene visualizzato senza parte decimale come conseguenza dell'uso di **%d** , mentre il numero 3 viene scritto con sei cifre decimali perché si è usato **%f** .

• Se una delle variabili è una matrice, la formattazione viene applicata a tutti i suoi elementi, letti per colonne, senza interruzioni.

Oltre a quelli visti, i principali codici sono:
%e : formato in notazione scientifica
la stringa speciale **\t** che indica il carattere di tabulazione

Inoltre è possibile specificare il numero di decimali. Per esempio **%0.3f** specifica notazione decimale con 3 cifre dopo il punto decimale.

INTERFACCIA CON L'ESTERNO

È possibile salvare l'intero workspace (cioè tutte le variabili definite fino a quel momento) mediante il comando

```
» save mievariabili.mat
```

dove "mievariabili" è un nome di fantasia. Il suffisso consigliato è ".mat". Le variabili salvate nel file "mievariabili" potranno essere recuperate successivamente con il comando

```
» load mievariabili.mat
```

che recupera tutte le variabili salvate. Se qualche variabile è stata ridefinita, il valore letto nel file viene sostituito a quello corrente. È possibile anche salvare solo parte del workspace nel seguente modo

```
» save mievariabili.mat x y z ...
```

dove **x y z** etc. è l'elenco delle variabili da salvare separate da spazi (e non da virgole).
Nel file "mievariabili.mat" le variabili sono salvate con codifica binaria, per cui il file non è leggibile in modo elementare da un altro programma applicativo.
Per salvare variabili in un file leggibile da un programma esterno (per esempio un editore di testo o un foglio elettronico tipo Excel), si deve aggiungere l'opzione **-ascii** . Il comando

```
» save alcunevariabili.txt x y z -ascii
```

salva i valori delle variabili **x,y,z** nel file "alcunevariabili.txt", leggibile da qualunque editore di testo.
Osserviamo che, usando questa opzione, si salvano solo i valori delle variabili, ma se ne perdono i nomi.
Il comando **save** ha altre opzioni. Per vederle tutte usare **help save**

È inoltre possibile recuperare una serie di dati da un file di testo creato da un programma esterno.
Come esempio, creiamo con qualunque editore di testo (anche con Word, pur di salvare il file come puro testo con suffisso .txt) un file così fatto

1	1.98	56
2.6	−.17	0

in cui gli elementi di ogni riga sono separati mediante spazi o mediante il carattere TAB e le righe sono separate con il carattere RETURN. Salviamo quindi i dati in un file di puro testo con un nome di fantasia, per esempio "matr.txt"

in una folder conosciuta da **MatLab** .
Eseguendo il comando

```
» load matr.txt
```

viene creata nel workspace la variabile **matr** come matrice 2×3 con i valori forniti.
Per concludere accenniamo alla funzione **xlsread** che consente di recuperare i dati da uno spreadsheet creato con Excel. La funzione

```
» a=xlsread('dati.xls')
```

crea una nuova variabile **a** che contiene i dati del file "dati.xls". Ovviamente occorre che il foglio elettronico sia strutturato in un modo da avere dati gestibili da **MatLab** .

GLI M-FILE

Dovendo eseguire più volte una serie di comandi complessi, non occorre ricopiarli ogni volta. Si può invece scrivere la lista dei comandi in uno "script" che va registrato in un apposito file detto M-file. Lo script può poi essere richiamato quando necessario.

Per esempio se volessimo moltiplicare tutti gli elementi della diagonale di una matrice quadrata **a** per **5** potremmo eseguire i seguenti comandi

```
» d=diag(a)
```

che crea la matrice colonna contenente gli elementi della diagonale di **a** . Poi

```
» d1=diag(d)
```

che crea una matrice diagonale che ha la diagonale identica a quella di **a** . Poi

```
» e=a-d1
```

che crea una matrice **e** con diagonale nulla e per il resto identica ad **a** . Poi

```
» d2=d1*5
```

che moltiplica la diagonale di **d1** per **5** , ottenendo **d2** . Infine

```
» a1=d2+e
```

costruisce la matrice voluta. Il risultato finale è la matrice **a1** .

Se vogliamo eseguire molte volte questa procedura, possiamo creare un M-File (c'è il comando apposito nel Menu "File") e inserire la lista delle istruzioni in questo modo

```
1 |   d=diag(a);
2 |   d1=diag(d);
3 |   e=a-d1;
4 |   d2=d1*5;
5 |   a1=d2+e
```

(i numeri compaiono automaticamente e servono per il "debugging").
Conviene porre dei segni **;** alla fine di ogni riga del file tranne l'ultima, affinché eseguendo il comando non compaiano sul display i risultati intermedi delle matrici **d, d1, d2, e** , ma compaia solo **a1** . Indi si salva il file per esempio come "**diagocinque.m**" (suffisso **.m** obbligatorio) in una directory (folder) di quelle che il programma **MatLab** riconosce come valide nel suo elenco di "Path" o nella sua folder di default che appare nella parte alta dello schermo.
Ogni volta che si scriverà

```
» diagocinque
```

MatLab eseguirà la serie di istruzioni dell'M-file **sulla matrice** **a** proprio come se li avessimo scritti ogni volta. Le matrici **d, d1, d2, e, a1** rimangono allocate nel *workspace* corrente, ovvero vengono definite o vengono ridefinite, se già esistevano, e possono essere richiamate.

DEFINIZIONE DI NUOVE FUNZIONI IN MATLAB

Naturalmente si vorrebbe poter eseguire la serie di comandi su una qualsiasi matrice senza che questa abbia necessariamente nome **a** . Per questo occorre definire un nuova funzione **MatLab** .
Questo si può fare mediante un M-File di tipo diverso che definisce una nuova funzione ed è così congegnato:

```
1 |   function y = diagoper5(a)
2 |   % diagoper(a) moltiplica la diagonale di a per cinque
3 |   d=diag(a);
4 |   d1=diag(d);
5 |   e=a-d1;
6 |   d2=d1*5;
7 |   y=d2+e;
```

Si noti il "cappello" **function y = diagoper5(a)** .

L'istruzione **function** serve a fornire tutti i dati relativi alla nuova funzione definita:
Questa istruzione, che deve essere la prima del file, svolge i seguenti compiti:

• dichiara che il file contiene una funzione e non uno script e che quindi le variabili usate rimagono locali.
• dichiara il nome della nuova funzione definita, in questo caso **diagoper5** . La funzione viene aggiunta all'elenco di funzioni che **MatLab** conosce.

L'M-File va salvato con il nome " **diagoper5.m** " **che deve essere identico** al nome dato alla funzione nella prima riga del listato.

• dichiara il numero e il nome delle variabili di input. In questo caso è una sola e ha nome **a** .
• dichiara il numero e il nome delle variabili di output. In questo caso è una sola e ha nome **y** .
Ogni volta che si calcola la funzione

```
» diagoper5(x)
```

su una qualunque matrice quadrata **x** , si ottiene come **ans** la matrice ottenuta quintuplicando la diagonale di **x** .

Altre osservazioni:
• Le righe del listato precedute dal simbolo **%** sono commenti ad uso di chi scrive e non vengono eseguite. Ma i commenti opzionali posti subito dopo la parola **function** sono ad uso dell'utilizzatore della funzione. Infatti appaiono nella finestra di comando quando si richiede aiuto per la funzione col comando **help**

```
» help diagoper5

  diagoper(a) moltiplica la diagonale di a per cinque
```

• La differenza fondamentale tra un M-file di tipo "script" e uno **di tipo funzione** è il fatto che in quest'ultimo le variabili sono "locali". Per esempio, nella funzione ora definita, le variabili usate **a, d, d1, d2, e, y** (comprese quindi quelle di input e output) vengono cancellate appena terminata l'esecuzione della funzione e non interferiscono con le variabili definite dall'utente nel *workspace* che possono anche avere gli stessi nomi.

• Si noti che al risultato finale occorre dare il nome **y** perché **y** è il nome scelto per la variabile di output. Se **y** non viene definita all'interno dell'M-file, il risultato della funzione va perduto.

• Dato che la variabile **y** è locale, conviene porre il simbolo **;** anche alla fine dell'ultima istruzione per evitare che il risultato compaia due volte sul display, una volta come variabile locale, la seconda come risultato della funzione.

• Alla funzione va dato un nome di fantasia, in questo caso si è scelto **diagoper5** .

• Occorre badare che il nome scelto non sia già il nome di un comando **MatLab** o di un M-File già esistente.
Nel primo caso l'esistente comando **MatLab** non sarebbe più riconosciuto, a favore della nuova funzione definita dall'utente.
Nel secondo caso l' M-File già esistente andrebbe perso, se salvato nella stessa folder, oppure potrebbe non essere riconosciuto a favore del nuovo comando, se salvato in altra folder.

Le funzioni possono avere anche molti argomenti. Per esempio, la funzione precedente può essere generalizzata, in modo che abbia due argomenti in entrata:
- la matrice su cui operare
- il numero per cui moltiplicare la diagonale che può essere qualunque e non solo 5.
Per ottenere questo occorre modificare le seguenti istruzioni

```
1 |   function y = diagoper(a,num)
2 |   % diagoper(a) moltiplica la diagonale di a per num
    ----------------------------
6 |   d1=d1*num;
```

Le funzioni possono inoltre avere diverse variabili in uscita. Per esempio, la funzione precedente può essere modificata, in modo che fornisca, oltre alla matrice con la diagonale cambiata, anche la matrice diagonale con la diagonale originale (che, nell'esecuzione, è la variabile temporanea **d1**).
Andrebbe solo modificata la prima riga del file.

```
1 |   function [y,d1] = diagoper(a,num)
```

Eseguendo però la funzione si otterrebbe solo la matrice modificata, cioè solo la prima variabile di output. Per ottenerle entrambe, al momento di calcolare la funzione vanno specificate entrambe le variabili a cui assegnare i due risultati, per esempio così

```
» [m,n]=diagoper(b,t)
```

La matrice modificata sarà **m** , mentre **n** sarà la matrice con la diagonale originale.

IL CICLO for...end

Esistono molti comandi per creare M-File assai complessi e versatili e strutture logiche simili a quelle usate nei

comuni linguaggi di programmazione, tipo **for...end , while...wend , if...then...else** con ampia gamma di opzioni.
Uno dei cicli più usati è il loop **for...end** .
Dovendo per esempio eseguire il comando $R_j \to R_j - (a(j,1))/a(1,1))R_1$ (primo passo dell'algoritmo gaussiano) sulle righe di una matrice **a** di 10 righe si dovrebbero eseguire le nove istruzioni

```
» a(2,:)=a(2,:)-(a(2,1)/a(1,1))*a(1,:);
» a(3,:)=a(3,:)-(a(3,1)/a(1,1))*a(1,:);
.......
» a(10,:)=a(10,:)-(a(10,1)/a(1,1))*a(1,:);
```

Conviene quindi creare un M-File con loop del tipo **for** :

```
1 |   for index=2:10
2 |        a(index,:)=a(index,:)-(a(index,1)/a(1,1))*a(1,:);
3 |   end
```

Il comando **for** fa eseguire tutte le istruzioni successive fino al comando **end** facendo assumere a **index** successivamente i valori $2, 3, ..., 10$.
L'editore degli M-Files provvede in generale ad indentare automaticamente tutti i comandi che appaiono tra **for** e **end** e che fanno parte del loop, cioè a spostarli a destra in modo che il file sia più leggibile e il loop rimanga evidenziato. Questo accade però solo se le istruzioni vengono inserite nell'ordine. Quando un M-file viene modificato, spesso le indentazioni automatiche non vengono create. È buona norma tenere comunque indentati i loop.
Se si vuole che **index** assuma i valori $2, 4, 6, ..., 20$ la sintassi sarà

```
1 |   for index=2:2:20
```

Se si vuole che **index** assuma i valori 1 , 1.1 , 1.2, ..., 3 , la sintassi sarà

```
1 |   for index=1:0.1:3
```

Se si vuole che **index** assuma successivamente i valori $10, 9, 8, ..., 0$ la sintassi sarà

```
1 |   for index=10:-1:0
```

I loop **for...end** possono essere "nidificati", cioè inseriti uno dentro l'altro. Per esempio, se si vuole una funzione che crei una matrice quadrata **n x n** in cui l'elemento di posto **(i,j)** sia **i*j+2** , si può creare una funzione del tipo seguente che dipende solo da **n** .

```
1 |   function a = pippo(n)
2 |
3 |   for jndex=1:n
4 |        for index=1:n
5 |             a(index,jndex)=index*jndex+2;
6 |        end
7 |   end
```

Notare i due loop dipendenti da due variabili diverse **index** e **jndex** , il loop in **index** nidificato dentro al loop in **jndex** e la doppia indentazione che rende più visibili i due loop.

Come nome per le variabili di loop abbiamo usato **index** e **jndex** , invece che **i** e **j** , per non confonderle con l'unità immaginaria.

IL COMANDO if...end

Il test condizionale con **if...end** viene usato quando si vuole che alcune istruzioni vengano eseguito solo se è verificata una certa condizione. Il test con **if** viene usato quasi esclusivamente negli M-files, anche se sarebbe possibile usarlo nella riga di comando. La sintassi è

```
if [test]
   [istruzioni da eseguire se il risultato non è 0]
else
   [istruzioni da eseguire se il risultato è 0]
end
```

Di solito il test che segue il comando **if** è un operatore relazionale che restituisce il valore **1** se il test è vero e il valore **0** se è falso. Quindi la prima serie di istruzioni è eseguita solo se la relazione è vera.
Le istruzioni che seguono **else** sono eseguite se la relazione è falsa. Il comando **else** è opzionale e può essere omesso. In tal caso, se la relazione è falsa, non viene fatto niente.
Esempio 1: Se si vuole eseguire direttamente l'algoritmo di Gauss su una matrice, si potrebbe voler evitare di eseguire l'operazione $R_3 \to R_3 - cR_1$ su una matrice **a** quando il numero **c** è zero.
Il comando per l'operazione elementare è:

```
» a(3,:)=a(3,:)-c*a(1,:)
```

Quindi si possono dare le seguenti istruzioni:

```
if c~=0
    a(3,:)=a(3,:)-c*a(1,:)
end
```

Esempio 2: Si vogliono cambiare in 0 tutti gli elementi non reali di una matrice riga `x`. È possibile creare mediante un M-File una funzione che esegua questa operazione per ogni matrice riga. La funzione seguente agisce su una matrice riga e ne crea un'altra seguendo appunto questa regola.

```
1 |   function y=zeroc(x)
2 |
3 |   s=size(x,2);
4 |   for index=1:s
5 |        if imag(x(index))~=0
6 |             y(index)=0;
7 |        else
8 |             y(index)=x(index);
9 |        end
10|   end
```

Osservazioni sulle righe del listato:

1 La funzione ha un nome di fantasia `zeroc`. Il nome della variabile di input è `x`, quello della variabile di output è `y`. Notiamo che, dato che l'M-file è una funzione, `x` e `y` sono variabili locali, cioè non interferiscono con le variabili definite nel *workspace*.

3 La funzione `size(x,2)` calcola il numero di colonne della matrice `x` per sapere quante volte va eseguito il loop.

4-10 Questo è il loop `for...end` che esamina tutti gli elementi della matrice riga `x`. La variabile di controllo del loop è stata chiamata `index`.

5-9 Questo è il comando `if...else...end`. L'istruzione 6 viene eseguita se la parte immaginaria dell'elemento di `x` non è zero (cioè se l'elemento non è reale). In caso contrario viene eseguita l'istruzione 8 che ricopia l'elemento di `x` nel corrispondente di `y`.

Esiste un'altra opzione per la struttura `if...else...end` ed è l'uso di `elseif` al posto di `else` o prima di `else`.
Benché possibile, l'uso di `elseif` è sconsigliato a favore del comando `switch` che viene descritto in seguito.

ERRORI E AVVERTIMENTI

Quando **MatLab** incontra un errore nell'eseguire un comando o una funzione, interrompe l'esecuzione e scrive un messaggio di errore.
Per esempio, se si tenta di calcolare l'inversa di una matrice non quadrata `x`, si ottiene

```
» inv(x)
???  Error using ==> inv
Matrix must be square.
```

Ogni altro calcolo viene interrotto. Se l'errore avviene all'interno di un M-file, la sua esecuzione viene interrotta.
In casi meno gravi, se l'esecuzione non pregiudica il proseguimento del calcolo, **MatLab** lancia solo un avvertimento (warning) e non interrompe i calcoli.
Per esempio se si tenta di calcolare l'inversa di una matrice quadrata, ma non invertibile `x`, si ottiene

```
» inv(x)
Warning:  Matrix is singular to working precision.
```

I calcoli proseguono, ma **MatLab** avverte che probabilmente i risultati sono inaffidabili.

È possibile simulare errori e avvertimenti all'interno di un M-File coi comandi seguenti

```
error('questa operazione non e'' lecita')
warning('questa operazione puo'' dare risultati errati)
```

Il comando `error` scrive il messaggio di errore e interrompe l'esecuzione dell'M-file.
Il comando `warning` scrive il messaggio di avvertimento, ma non interrompe i calcoli. In pratica è come un comando `disp` con qualche opzione in più (vedi `help warning`).
Per esempio nell'M-file `diagoper` creato sopra come esempio, sarebbe bene evitare che l'utenti usi la funzione su una matrice non quadrata.
Questo si può ottenere inserendo questa istruzione subito dopo il "cappello" `function` + stringa di aiuto

```
[r,c]=size(a);
if r~=c
     error('matrice non quadrata')
end
```

Se si tenta di usare la funzione su una matrice non quadrata, l'esecuzione si arresta e compare la scritta "matrice non quadrata".

GLI OPERATORI LOGICI

Spesso gli operatori relazionali `==` , `<=` etc. sono usati congiuntamente agli operatori logici booleani AND e OR che in **MatLab** sono generati mediante i simboli seguenti: `&` per AND `|` per OR

Esempi:

```
if a>0 & a<=5
   [istruzioni]
end
```

le istruzioni verranno eseguite solo se $0 < a \leq 5$

```
if a<=0 | a>5
   [istruzioni]
end
```

le istruzioni verranno eseguite solo se a è esterno all'intervallo $(0, 5]$

```
if (a>0 & a<1)| a>2
   [istruzioni]
end
```

le istruzioni verranno eseguite solo se a è nell'intervallo $(0, 1)$ o nell'intervallo $(2, +\infty)$.

OPERATORI RELAZIONALI E MATRICI

Occorre qualche cautela nell'usare il comando `if...end` con gli operatori relazionali, quando ci si riferisce a matrici e non a scalari, dato che gli operatori relazionali confrontano tutti gli elementi delle matrici. Per esempio se si vogliono eseguire certe istruzioni solo sulle matrici simmetriche, la sintassi corretta è

```
if a==a.'
   [istruzioni da eseguire se a è simmetrica]
end
```

Se invece si vogliono eseguire certe istruzioni solo su matrici non simmetriche, la sintassi seguente non funziona

```
if a~=a.'
   [istruzioni da eseguire se a non è simmetrica]
end
```

Le istruzioni non verranno mai eseguite, perché l'operatore relazionale `a~=a.'` restituisce sempre una matrice non completamente fatta di 1. Conviene quindi una delle seguenti due sintassi:

```
if a==a.'
else
   [istruzioni da eseguire se a non è simmetrica]
end
```

Oppure, sfruttando la funzione `any` :

```
if any(a~=a.')
   [istruzioni da eseguire se a non è simmetrica]
end
```

IL COMANDO switch...case...end

Benché il test `if...end` sia a prima vista il test condizionale più naturale, è molto più conveniente in molti casi (anzi è consigliato da molti programmatori e da molte norme) il ciclo `switch...end`
Come esempio creiamo un M-file che interpreta un numero intero `a` da 0 a 10 come voto e fornisce il giudizio.

```
if a==10
     disp('ottimo')
elseif a==8 | a==9
     disp('buono')
elseif a<6
     disp('scarso')
else
     disp('sufficiente')
end
```

Confrontare la leggibilità del listato sopra col seguente che fa le stesse cose

```
switch a
case 10
     disp('ottimo')
case {8,9}
     disp('buono')
case {0,1,2,3,4,5}
     disp('scarso')
otherwise
     disp('sufficiente')
end
```

Purtroppo il ciclo **switch...end** non prevede la sintassi **case <6** che è invece usata in altri linguaggi di programmazione, costringendo, in certi casi, a strutture più complesse.
Esiste comunque un trucco, non riportato in generale dai manuali, per aggirare l'ostacolo ed è quello di usare per **switch** una variabile che valga 1

```
vero=1;
switch vero
case a==10
     disp('ottimo')
case (a==8 | a==9)
     disp('buono')
case a<6
     disp('scarso')
otherwise
     disp('sufficiente')
end
```

Osserviamo comunque che nel ciclo **switch...end** vengono eseguite solo le istruzioni che seguono **il primo** test **case** che soddisfa il criterio richiesto e non vengono considerati eventuali **case** successivi che soddisfano il criterio, diversamente dal linguaggio C. Per esempio

```
switch a
case 3
   [istruzione 1]
case {3,4}
   [istruzione 2]
end
```

Se **a** vale 3, verrà eseguita l'istruzione 1 e non l'istruzione 2. Se **a** non è né 3 né 4, non verrà eseguito niente.

IL CICLO while...end

Questo ciclo viene usato quando non è possibile prevedere a priori la lunghezza di un loop, ma si vuole che questo venga eseguito fintantoché è verificata (o non è verificata) una certa condizione.
La sintassi è la seguente

```
while [condizione]
   [istruzioni da eseguire fintantoché la condizione è valida]
end
```

Consideriamo a titolo di esempio la successione ricorsiva così definita: $a_1 = 1 \quad ; \quad a_{n+1} = \sqrt{a_n^2/(a_n+2)}$
Notoriamente il limite della successione è 0. Si vuole determinare il primo n per cui $a_n < 10^{-5}$.
Si può usare uno script così fatto

```
n=1;
a=1;
while a>=1e-05
     a=sqrt(a^2/(a+2));
     n=n+1;
end
disp(n)
```

• Vengono inizializzati l'indice **n=1** e il primo elemento della successione **a=1** .
• L'iterazione con **while** viene eseguita fintantoché **a** è maggiore o uguale a 10^{-5}.
Notare il numero 10^{-5} inserito usando la notazione esponenziale.
• Ad ogni passo viene calcolato il nuovo **a** e viene incrementato l'indice **n** .
• Quando **a** è minore di 10^{-5} il ciclo **while** non viene più eseguito e viene visualizzato l'indice **n** in corrispondenza del quale è terminato il ciclo.

È possibile usare lo stesso script con un'altra successione di cui non si conosce il limite. Se la successione non converge a 0, il ciclo **while...end** può durare indefinitamente.
Conviene allora fissare un tetto massimo per le iterazioni, per esempio $n = 1000$, e stabilire di uscire comunque dal ciclo **while...end** quando **n** arriva a **1000** .
Questo si realizza col comando **break** . Il ciclo va così modificato.

```
while a>1e-05
     a= ------- % altra successione
     n=n+1;
     if n==1000
          break
     end
end
```

Il comando **break** consente di uscire dal ciclo **while...end** .
Il comando **break** può essere usato anche all'interno di un ciclo **for...end** o di uno **switch...end** e, in caso di loop nidificati, fa uscire solo da quello al cui interno è stato inserito.