Capitolo 1

Il MATLAB

1.1 Introduzione al MATLAB

Il Matlab (acronimo delle parole inglesi *MATrix LABoratory*) è un software basato sulla manipolazione di matrici molto utilizzato nel campo della ricerca scientifica, non solo matematica, per la sua grande portabilità (infatti è disponibile sia per grandi workstation che per comuni personal computers), unita ad una notevole facilità d'uso e alle potenzialità di calcolo. Inoltre l'uso del Matlab è reso facile dalla presenza di un manuale dei comandi in linea, che può essere invocato tramite il comando `help`, e dalla presenza del comando `demo` che presenta numerosi e significativi esempi di applicazioni di tutte le funzioni Matlab. Nelle seguenti pagine faremo riferimento alle istruzioni presenti nelle ultime versioni del software.
Una volta lanciata l'esecuzione del programma compare il prompt del software

```
>>
```

il che significa che MatLab resta in attesa che venga effettuata una qualsiasi operazione (editare un programma, lanciare l'esecuzione di un file oppure eseguire un'istruzione digitata sulla riga di comando e così via).
Il comando `help`, come già detto, fornisce tutte le informazioni relative ad un particolare comando oppure una lista di tutti gli argomenti per i quali è presente un aiuto. La sintassi del comando è semplice:

```
>> help
```

oppure

```
>> help comando
```

Per esempio per sapere l'uso del comando `load`, che descriveremo in dettaglio nel seguito, è sufficiente scrivere

```
>> help load
```

Anche il comando `demo` ha una sintassi molto semplice:

```
>> demo
```

a questo punto compariranno sullo schermo alcuni menu e basterà scegliere, tramite il mouse, l'argomento del quale si vuole vedere una dimostrazione.
Il Matlab può essere considerato un interprete le cui istruzioni sono del tipo:

```
variabile = espressione
```

oppure

```
variabile
```

In quest'ultimo caso, quando cioè un'istruzione è costituita solo dal nome di una variabile viene interpretata come la visualizzazione del valore di tale variabile. Vediamo i seguenti esempi.

```
>> b=5;
>> b
ans =
   5
>>


>> b=5
b =
   5
>>
```

Nel primo caso il valore di output di `b` è stato attribuito alla variabile di comodo `ans` (abbreviazione per la parola inglese *answer*). Questo modo di procedere viene utilizzato anche quando si chiede di valutare un'espressione di tipo numerico senza l'ausilio di variabili.

```
>> 3+4
ans =
   7
>>
```

Ogni espressione introdotta viene interpretata e calcolata. Ogni istruzione può essere scritta anche su due righe purchè prima di andare a capo vengano scritti 3 punti $''\ldots''$. Più espressioni possono essere scritte sulla stessa riga purchè siano separate da una virgola o dal punto e virgola. Se una riga di un file Matlab inizia con % allora tale riga viene considerata come un commento. Il Matlab fa distinzione tra lettere minuscole e maiuscole, quindi se abbiamo definito una variabile `A` e facciamo riferimento a questa scrivendo `a` essa non viene riconosciuta.
Le frecce della tastiera consentono di richiamare e riutilizzare comandi scritti in precedenza; utilizzando infatti ripetutamente il tasto $\uparrow$ vengono visualizzate le linee di comando precedentemente scritte. Per tornare ad un'istruzione sorpassata basta premere il tasto $\downarrow$. Con i tasti $\longleftarrow$ e $\longrightarrow$ ci si sposta veso sinistra oppure verso destra sulla riga di comando su cui ci si trova.

1.2 Assegnazione di matrici

La prima cosa da imparare del Matlab è come manipolare le matrici che costituiscono la struttura fondamentale dei dati. Una matrice è una tabella di elementi caratterizzata da due dimensioni: il numero delle righe e quello delle colonne. I vettori sono matrici aventi una delle dimensioni uguali a 1. Infatti esistono due tipi di vettori: i *vettori riga* aventi dimensione $1 \times n$, e i *vettori colonna* aventi dimensione $n \times 1$. I dati scalari sono matrici di dimensione 1×1. Le matrici possono essere introdotte in diversi modi, per esempio possono essere assegnate esplicitamente, o caricate da file di dati esterni, o generate utilizzando funzioni predefinite.
Per esempio l'istruzione

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

assegna alla variabile `A` una matrice di tre righe e tre colonne. Gli elementi di una riga della matrice possono essere separate da virgole o dallo spazio, mentre le diverse righe sono separate da un punto e virgola. Se alla fine

dell'assegnazione viene messo il punto e virgola allora la matrice non viene visualizzata sullo schermo. In generale se vogliamo assegnare ad `A` una matrice ad m righe ed n colonne la sintassi è la seguente:

```
>> A = [riga 1; riga 2; ...; riga m];
```

Per assegnare ad una variabile `x` un vettore riga si ha

```
>> x = [3 -4 5];
```

gli elementi possono anche essere separati da una virgola

```
>> x = [3,-4,5];
```

Per assegnare invece ad una variabile un vettore colonna basta separare gli elementi con un punto e virgola:

```
>> y = [1;-3;6];
```

La stessa matrice `A` dell'esempio visto in precedenza può essere assegnata anche a blocchi:

```
>> A = [ 1 2 3; 4 5 6];
>> b = [ 7 8 9];
>> A = [ A; b];
```

mentre in modo analogo si può anche aggiungere una colonna:

```
>> A = [-1 2 3; 0 5 6; -5 4 3];
>> x = [-7; 0; 9];
>> A = [ A, x];
```

Descriviamo ora alcune funzioni predefinite che forniscono in output determinate matrici.

```
>> A=rand(m,n)
```

costruisce una matrice $m \times n$ di elementi casuali uniformemente distribuiti tra 0 e 1;

```
>> A=zeros(m,n)
```

costruisce una matrice $m \times n$ di elementi nulli;

```
>> A=ones(m,n)
```

costruisce una matrice $m \times n$ di elementi tutti uguali a 1;

```
>> A=eye(m,n)
```

costruisce una matrice $m \times n$ i cui elementi sono uguali a 1 sulla diagonale principale e 0 altrove. Per le funzioni appena viste se uno dei due parametri è omesso allora la matrice costruita viene considerata quadrata.
Il dimensionamento delle matrici è automatico. Per esempio se si pone

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

e successivamente

```
>> B = [1 0; 0 7];
```

il programma riconosce che la matrice `B` ha cambiato le dimensioni da 2×3 a 2×2.
Per identificare l'elemento della matrice che si trova al posto `j` nella riga `i` si usa la notazione `A(i,j)`.
Per esempio `A(4,2)` indica l'elemento che si trova nella quarta riga e in colonna 2. La numerazione delle righe e delle colonne parte sempre da 1 (quindi il primo elemento della matrice è sempre `A(1,1)`).
Per fare riferimento ad un singolo elemento di un vettore (sia riga che colonna) è sufficiente utilizzare un solo indice (quindi la notazione `x(i)` indica l'`i`-esima componente del vettore `x`).
Se si fa riferimento a un elemento di una matrice di dimensione $m \times n$ che non esiste allora il Matlab segnala l'errore con il seguente messaggio:
`Index exceeds matrix dimension`.
Se `C` è una matrice non ancora inizializzata allora l'istruzione

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

fornisce come risposta

```
C =
    0   0
    0   0
    0   1
```

cioè il programma assume come dimensioni per `C` dei numeri sufficientemente grandi affinchè l'assegnazione abbia senso. Se ora si pone

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

si ha:

```
C =
   0   0   2
   0   0   0
   0   1   0
```

In Matlab gli indici devono essere strettamente positivi mentre se si richiede un elemento di indice negativo oppure uguale a zero si ha sullo schermo il seguente messaggio di errore:
`Index into matrix is negative or zero`
L'uso del valore `end` come indice di una riga (o di una colonna) della matrice fa riferimento all'ultima riga (o colonna) della matrice senza indicarlo numericamente.

1.2.1 Sottomatrici e notazione :

Vediamo ora alcuni esempi che illustrano l'uso di : per vettori e matrici. Le istruzioni

```
>> x=[1:5];
```

e

```
>> x=1:5;
```

sono equivalenti all'assegnazione diretta del vettore `x`:

```
>> x=[1 2 3 4 5];
```

Ciò vale anche per vettori di elementi reali. Infatti l'istruzione

```
>> x=[0.2:0.2:1.2];
```

equivale a scrivere

```
>> x=[0.2 0.4 0.6 0.8 1.0 1.2];
```

Inoltre è possibile anche l'uso di incrementi negativi:

```
>> x=[5:-1:1];
```

è equivalente a

```
>> x=[5 4 3 2 1];
```

L'istruzione

```
>> x=x(n:-1:1);
```

inverte gli elementi del vettore `x` di dimensione n. La notazione : può essere anche applicata a matrici. Infatti se `A` è una matrice abbiamo:

```
>> y=A(1:4,3);
```

assegna al vettore colonna `y` i primi 4 elementi della terza colonna della matrice `A`;

```
>> y=A(4,2:5);
```

assegna al vettore riga `y` gli elementi della quarta riga di `A` compresi tra il secondo e il quinto;

```
>> y=A(:,3);
```

assegna al vettore colonna `y` la terza colonna di `A`;

```
>> y=A(2,:);
```

assegna al vettore riga `y` la seconda riga di `A`;

```
>> B=A(1:4,:);
```

assegna alla matrice `B` le prime 4 righe di `A`;

```
>> B=A(:,2:6);
```

assegna alla matrice `B` le colonne di `A` il cui indice è compreso tra 2 e 6;

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

assegna alla matrice `B` la seconda e la quarta colonna di `A`;

```
>> A(:,[2 4 5])=B(:,1:3);
```

sostituisce alle colonne 2, 4 e 5 della matrice `A` le prime 3 colonne della matrice `B`.

1.3 Operazioni su matrici e vettori

In Matlab sono definite le seguenti operazioni su matrici e vettori:

+	addizione
−	sottrazione
*	moltiplicazione
^	elevazione a potenza
'	trasposto
/	divisione
( )	specificano l'ordine di valutazione delle espressioni

Ovviamente queste operazioni possono essere applicate anche a scalari. Se le dimensioni delle matrici coinvolte non sono compatibili allora viene segnalato un errore eccetto nel caso di operazione tra uno scalare e una matrice. Per esempio se A è una matrice di qualsiasi dimensione allora l'istruzione

```
>> C = A+2;
```

assegna alla matrice C gli elementi di A incrementati di 2.
Nel caso del prodotto tra matrici è necessario prestare molta attenzione alle dimensioni delle matrici. Infatti ricordiamo che se $A \in \mathbb{R}^{m\times p}$ e $B \in \mathbb{R}^{p\times n}$ allora la matrice

$$C = A \cdot B, \qquad C \in \mathbb{R}^{m\times n}$$

si definisce nel seguente modo:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}, \qquad i = 1, \ldots, m \; j = 1, \ldots, n$$

ed è la matrice che viene calcolata scrivendo l'istruzione

```
>> C = A*B
```

In caso contrario se scriviamo

```
>> C = B*A
```

allora il programma segnala errore a meno che non sia $m = n$.
È importante notare che le operazioni $*, \hat{}$ e / operano elemento per elemento se sono precedute da un punto:

`C=A.*B` $\Rightarrow$ $c_{ij} = a_{ij}b_{ij}$

`C=A./B` $\Rightarrow$ $c_{ij} = a_{ij}/b_{ij}$

`C=A.^B` $\Rightarrow$ $c_{ij} = a_{ij}^{b_{ij}}$

1.3.1 Costanti predefinite

Come la maggior parte dei linguaggi di programmazione il Matlab ha alcune costanti predefinite cioè delle variabili che hanno un proprio valore senza che esso venga esplicitamente assegnato:

`eps`	precisione di macchina ($\simeq 2.2 \cdot 10^{-16}$)
`pi`	π (cioe' 3.14159265358979)
`i`	unita' immaginaria ($\sqrt{-1}$)
`j`	unita' immaginaria ($\sqrt{-1}$)
`realmax`	il piu' grande numero floating point ($1.7976e+308$)
`realmin`	il piu' piccolo numero floating point ($2.2251e-308$)
`inf`	infinito (∞)
`NaN`	Not a Number.

La costante `inf` è ottenuta come risultato di una divisione per zero oppure il calcolo del logaritmo di zero o se il risultato è un overflow (per esempio `2*realmax`). La costante `NaN` invece è ottenuta come risultato di operazioni matematicamente non definite come $0/0$ oppure $\infty - \infty$.
Come accade per la maggior parte dei linguaggi di programmazione anche in Matlab è possibile definire variabili il cui nome è una costante predefinita, quindi per esempio è possibile usare la variabile `i` come indice intero.

1.3.2 Operatori relazionali e logici

I seguenti sono gli operatori relazionali

$<$	minore
$>$	maggiore
$<=$	minore o uguale
$>=$	maggiore o uguale
$==$	uguale
$\sim=$	diverso

Una relazione di tipo logico assume valore 0 o 1 a seconda del fatto se essa sia rispettivamente falsa o vera. Per esempio scrivendo

```
>> 3<5
```

otterremo

```
>> 3<5
ans =
   1
```

oppure scrivendo

```
>> 1>5
```

la risposta è

```
>> 1>5
ans =
   0
```

Quando un operatore relazionale è applicato a matrici di dimensioni uguali si ottiene come risultato una matrice i cui elementi sono 1 oppure 0. Vediamo il seguente esempio:

```
>> A=[2 1 ; 0 3];
>> B=[2 -1 ; -2 3];
>> A==B
ans =
   1 0
   0 1

>> A>B
ans =
   0 1
   1 0

>> A>=B
ans =
   1 1
   1 1
```

Gli operatori logici che il Matlab consente di utilizzare sono i seguenti:

&	AND logico per componente
&&	AND logico su valori scalari
|	OR logico per componente
||	OR logico su valori scalari
$\sim$	NOT
xor	OR logico esclusivo

Gli operatori || e && si applicano (in analogia a quanto previsto per la sintassi del linguaggio C) a dati di tipo scalari mentre gli operatori | e & si applicano nel confronto tra vettori e matrici ed operano componente per componente. In quest'ultimo caso il risultato sarà una matrice (o un vettore) delle stesse dimensioni dei dati ai quali sono stati applicati.
L'operatore `xor` è l'unico che ammette una sintassi differente in cui gli operandi devono essere specificati tra parentesi tonde

```
>> xor(A,B)
```

dove A e B sono due espressioni logiche ed il risultato è 1 solo se una di queste risulta essere vera mentre il risultato è 0 se A e B sono entrambe vere o sono entrambe false. per esempio

```
>> x=xor(3>5,1<6)
   x=
   1
>> y=xor(1<3,5==5)
   y=
   0
```

1.4 Funzioni predefinite

In MatLab è possibile utilizzare un gran numero di funzioni predefinite in grado di semplificare notevolmente la scrittura di programmi. In questo paragafo elenchiamo soltanto quelle più utilizzate nell'ambito di programmi di calcolo. Tali funzioni sono state suddivise in tre gruppi: le funzioni scalari, quelle vettoriali e quelle matriciali.

1.4.1 Funzioni scalari

Le funzioni Matlab riportate di seguito sono praticamente quelle di tipo matematico che accettano come argomento di input un numero reale (o complesso), che sintatticamente deve essere scritto tra parentesi tonde subito dopo il nome della funzione (esempio: `cos(2*pi)`, oppure `log(x+1)`).

`sin`	seno
`cos`	coseno
`tan`	tangente
`asin`	arcoseno
`acos`	arcocoseno
`atan`	arcotangente
`sinh`	seno iperbolico
`cosh`	coseno iperbolico
`tanh`	tangente iperbolica
`asinh`	arcoseno iperbolico
`acosh`	arcocoseno iperbolico
`atanh`	arcotangente iperbolica
`exp`	esponenziale
`log`	logaritmo naturale
`log10`	logaritmo in base 10
`sqrt`	radice quadrata
`abs`	valore assoluto
`rem`	resto della divisione
`mod`	resto della divisione
`sign`	segno
`round`	arrotondamento
`floor`	parte intera inferiore
`ceil`	parte intera superiore

Tra queste funzioni appena nominate le ultime tre meritano un piccolo approfondimento; nella seguente tabella sono riportati i valori di tali funzioni per differenti numeri reali:

`x`	`round(x)`	`floor(x)`	`ceil(x)`
3.7	4	3	4
3.1	3	3	4
-4.7	-5	-5	-4
-4.3	-4	-5	-4

Osserviamo che `floor(x)` è sempre minore di `x` mentre `ceil(x)` è maggiore di `x`.
Le funzioni riportate in precedenza possono essere applicate anche a variabili di tipo vettore (o matrice), in questo si comportano come se fossero applicate a ciascun elemento del vettore (o della matrice). Per esempio se `x` è un vettore riga allora `abs(x)` è anch'esso un vettore riga le cui componenti sono i valori assoluti delle componenti del vettore `x`.
Le funzioni`rem` e `mod` calcolano il resto della divisione tra numeri e pertanto richiedono due parametri (dividendo e divisore) in input. Si rimanda all'`help` in linea per verificare quali siano le differenze tra le due funzioni.

1.4.2 Funzioni vettoriali

Le seguenti funzioni Matlab operano principalmente su vettori (riga o colonna):

`max`	massimo elemento di un vettore
`min`	minimo elemento di un vettore
`sum`	somma degli elementi di un vettore
`prod`	prodotto degli elementi di un vettore
`sort`	ordinamento di un vettore
`length`	numero di elementi di un vettore
`fliplr`	inverte gli elementi di un vettore riga
`flipud`	inverte gli elementi di un vettore colonna

Le funzioni `max` e `min` possono fornire in uscita anche l'indice della componente massima (o minima) del vettore. La sintassi in questo caso è la seguente:

```
>> [massimo,k]=max(x);
>> [minimo,k]=min(x);
```

Le funzioni appena elencate possono essere applicate anche a variabili di tipo matrice, ma producono come risultato un vettore riga contenente i risultati della loro applicazione a ciascuna colonna. Per esempio scrivere

```
>> y=max(A);
```

significa assegnare al vettore (riga) `y` gli elementi massimi delle colonne della matrice `A`. Per ottenere il risultato della loro azione alle righe basta applicare la stessa funzione alla matrice trasposta `A'`. Volendo trovare il massimo

elemento della matrice A si dovrebbe scrivere `max(max(A))`. La funzione `max` può essere applicata per trovare anche la posizione dell'elemento massimo (ovvero l'indice della riga e della colonna in cui si trova). Infatti, se `A` è una matrice allora

```
>> [mass,r]=max(A);
```

calcola il vettore dei massimi (ovvero `mass`) e l'indice di riga in cui si trova. Per esempio il massimo della prima colonna è `mass(1)` che si trova sulla riga `r(1)`, il massimo della seconda colonna è `mass(2)` che si trova sulla riga `r(2)`, e così via. Applicando la funzione al vettore `mass`, si trova:

```
>> [massimo,c]=max(mass);
```

in cui `massimo` è il valore cercato che si trova nella colonna `c`, la riga sarà invece `r(c)`.
Se la funzione viene applicata a due vettori`x` e `y`, entrambi di lunghezza n,

```
>> z=max(x,y);
```

allora le componenti del vettore risultato `z` sono pari al massimo tra le componenti dei due vettori, ovvero

$$z_i = \max(x_i, y_i), \qquad i = 1, 2, \ldots, n.$$

Analogamente se viene applicata ad un vettore `x` e ad uno scalare α

```
>> z=max(x,α);
```

allora le componenti del vettore risultato `z` sono pari a

$$z_i = \max(x_i, \alpha), \qquad i = 1, 2, \ldots, n.$$

Nella seguente tabella sono riportate altre funzioni di tipo vettoriale di uso meno comune rispetto a quelle descritte in precedenza:

`any`	Funzione logica vera se esiste un elemento del vettore diverso da zero
`all`	Funzione logica vera se tutti gli elementi del vettore sono diversi da zero
`mean`	Calcola la media del vettore
`median`	Calcola il valore mediano del vettore
`cumsum`	Calcola la somma cumulativa degli elementi del vettore

Va specificato che la media di un vettore `x` è equivalente alla seguente istruzione:

```
sum(x)/length(x)
```

Il mediano di un vettore è uguale all'elemento del vettore che si trova al centro dello stesso. Per essere chiari, se un vettore ha un numero dispari `n` di componenti, allora, una volta ordinato, il mediano coincide con l'elemento in posizione `(n+1)/2`. Se invece il numero `n` è pari allora il mediano coincide con il valor medio degli elementi in posizione `n/2` ed `n/2+1`.

1.4.3 Funzioni di matrici

Le più utili funzioni di matrici sono le seguenti:

`eig`	autovalori e autovettori
`inv`	inversa
`det`	determinante
`size`	dimensioni
`norm`	norma
`cond`	numero di condizione in norma 2
`rank`	rango
`tril`	parte triangolare inferiore
`triu`	parte triangolare superiore
`diag`	fornisce in output un vettore colonna dove è memorizzata la parte diagonale di una matrice. Se la funzione è applicata invece ad un vettore allora in uscita avremo una matrice diagonale i cui elementi principali sono quelli del vettore di input.

Le funzioni Matlab possono avere uno o più argomenti di output. Per esempio `y = eig(A)`, o semplicemente `eig(A)`, produce un vettore colonna contenente gli autovalori di A mentre

```
>> [U,D] = eig(A);
```

produce una matrice U le cui colonne sono gli autovettori di A e una matrice diagonale D con gli autovalori di A sulla sua diagonale. Anche la funzione `size` ha due parametri di output:

```
>> [m,n] = size(A);
```

assegna a `m` ed `n` rispettivamente il numero di righe e di colonne della matrice `A`.
La funzione `norm` se viene applicata ad una matrice calcola la norma 2 della stessa matrice.

```
>> norm(A);
```

È tuttavia possibile calcolare anche altre norme specificando un secondo parametro. Per esempio

```
>> norm(A,'inf');
```

calcola la norma infinito di `A`, mentre

```
>> norm(A,1);
```

calcola la norma 1 di `A` e

```
>> norm(A,'fro');
```

calcola la norma di Frobenius di `A`. Ponendo il econdo parametro uguale a 2 viene calcolata analogamente la norma 2 della matrice.
Un'altra funzione matriciale molto utile è quella che calcola la decomposizione ai valori singolari di una matrice:

```
>> [U,S,V]=svd(A);
```

dove `U` e `V` sono matrici ortogonali mentre `S` è una matrice diagonale, tali che

$$A = USV^T.$$

Applicando la funzione nel modo seguente

```
>> [U,S,V]=svd(A,0);
```

si ottiene la decomposizione ai valori singolari economica (ovvero la SVD compatta ma solo per la matrice `U`), mentre

```
>> [U,S,V]=svd(A,'econ');
```

fa lo stesso ma solo per la matrice `V`. La funzione

```
>> x=svds(A,k);
```

calcola ed assegna al vettore `x` solo i primi `k` valori singolari della matrice (ovvero i più grandi). Riassumiano nella successiva tabella altre funzioni di matrici:

`repmat`	Duplica un vattore in una determinata dimensione

1.5 Le istruzioni for, while, if e switch

Il Matlab è dotato delle principali istruzioni che servono a renderlo un linguaggio strutturato. La più importante istruzione per la ripetizione in sequenza delle istruzioni è il `for`, che ha la seguente sintassi:

```
for var=val_0:step:val_1
    lista istruzioni
end
```

La variabile *var* assume come valore iniziale *val_0*, viene eseguita la lista di istruzioni che segue, poi è incrementata del valore *step*, vengono rieseguite le istruzioni che seguono e così via, finchè il suo valore non supera *val_1*. Il valore dello *step* può essere negativo, nel qual caso il valore di *val_0* deve essere logicamente superiore a *val_1*.
Nel caso in cui il valore di *step* sia uguale a 1 questo può essere omesso, in tutti gli altri casi va necessariamente specificato. Questo vuol dire che se $step = 1$ la sintassi dell'istruzione `for` è la seguente

```
for var=val_0:val_1
    lista istruzioni
end
```

La sintassi per l'istruzione `while` è la seguente.

```
while espressione logica
    istruzioni
end
```

Le *istruzioni* vengono eseguite fintantochè l'*espressione logica* rimane vera. La sintassi completa dell'istruzione `if` è la seguente:

```
if espressione logica
    istruzioni
elseif espressione logica
    istruzioni
else
    istruzioni
end
```

I rami `elseif` possono essere più di uno come anche essere assenti. Anche il ramo `else` può mancare. Vediamo ora alcuni esempi di come le istruzioni appena descritte possono essere applicate.
Se all'interno delle istruzioni che seguono il `for` o il `while` si verifica la necessità di interrompere il ciclo delle istruzioni allora ciò può essere fatto utilizzando l'istruzione `break`.
Ultima istruzione di questo tipo (e presente solo nell'ultima versione del programma) è l'istruzione `switch` che ha lo stesso ruolo e quasi la stessa sintassi dell'omonima istruzione in linguaggio C:

```
switch variabile
    case valore_0
        istruzioni
    case valore_1
        istruzioni
    case valore_2
        istruzioni
    otherwise
        istruzioni
end
```

che, in funzione del valore assunto dalla variabile, esegue o meno una serie di istruzioni. In particolare se nessuno dei valori previsti è assunto dalla variabile allora viene previsto un caso alternativo (`otherwise`) che li contempla tutti. Vediamo il seguente esempio:

```
switch rem(n,2)
    case 0
        disp('n e'' un numero pari')
    case 1
        disp('n e'' un numero dispari')
    otherwise
        disp('Caso impossibile')
end
```

1.6 Istruzioni per gestire il Workspace

Il comando

```
>> who
```

elenca le variabili presenti nell'area di lavoro, mentre il comando

```
>> whos
```

elenca, oltre al nome delle variabili, anche il tipo e l'occupazione di memoria. Una variabile può essere cancellata da tale area con il comando

```
>> clear nome variabile
```

mentre il comando

```
>> clear
```

cancella tutte le variabili presenti nell'area di lavoro.
Premendo contemporaneamente i tasti *Ctrl* e *c* si interrompe l'esecuzione di un file Matlab. L'istruzione

```
>> save
```

salva il contenuto dell'area di lavoro (cioè le variabili e il loro valore) nel file binario `matlab.mat`. Se invece si scrive

```
>> save nomefile
```

allora tutta l'area di lavoro viene salvata nel file `nomefile.mat`. Se invece si vogliono salvare solo alcune variabili e non tutta l'area di lavoro allora è possibile farlo specificando, oltre al nome del file, anche l'elenco di tali variabili. Per esempio

```
>> save nomefile A B x
```

salva nel file `nomefile.mat` solo il contenuto delle variabili `A, B` e `x`. Scrivendo

```
>> save nomefile A B x -ascii
```

allora il file `nomefile.mat` non ha il formato binario ma ascii, e questo è utile se si vuole verificare il contenuto del file.
Per ripristinare il contenuto dell'area di lavoro dal file `matlab.mat` il comando è

```
>> load
```

mentre è possibile anche in questo caso specificare il file da caricare. Facendo riferimento all'esempio del comando `save` allora scrivendo

```
>> load nomefile
```

ripristina le variabili e il loro valore che erano stati memorizzati nel file `nomefile.mat`.

1.7 M-files

Il Matlab può eseguire una sequenza di istruzioni memorizzate in un file. Questi file prendono il nome di *M-files* perchè la loro estensione è `.m`. Ci sono due tipi di M-files: gli *script files* e i *function files*.

Script files

Uno script file consiste in una sequenza di normali istruzioni Matlab. Se il file ha come nome `prova.m` allora basterà eseguire il comando

```
>> prova
```

per far sì che le istruzioni vengano eseguite. Le variabili di uno script file sono di tipo globale, per questo sono spesso utilizzati anche per assegnare dati a matrici di grosse dimensioni, in modo tale da evitare errori di input. Per esempio se in file `assegna.m` vi è la seguente assegnazione:

```
A=[0 -2 13 4; -5 3 10 -8; 10 -12 14 17; -1 4 5 6];
```

allora l'istruzione `assegna` servirà per definire la matrice `A`.

Function files

Permettono all'utente di definire funzioni che non sono standard. Le variabili definite nelle funzioni sono locali, anche se esistono delle istruzioni che permettono di operare su variabili globali. Vediamo il seguente esempio.

```
function a = randint(m,n)
% randint(m,n) Fornisce in output una matrice
% di dimensioni m×n di numeri casuali
% interi compresi tra 0 e 9.
a = floor(10*rand(m,n));
end
```

Osserviamo preliminarmente che l'istruzione `end` alla fine del file è in realtà opzionale, tuttavia se, utilizzando l'editor MatLab per scrivere un *function file*, essa viene inserita di default.
Tale funzione va scritta in un file chiamato `randint.m` (corrispondente al nome della funzione). La prima linea definisce il nome della funzione e gli argomenti di input e di output. Questa linea serve a distinguere i function files dagli script files. Quindi l'istruzione Matlab

```
>> c=randint(5,4);
```

assegna a c una matrice di elementi interi casuali di 5 righe e 4 colonne. Le variabili m, n e a sono interne alla funzione quindi il loro valore non modifica il valore di eventuali variabili globali aventi lo stesso nome.
Vediamo ora il seguente esempio di funzione che ammette un numero di argomenti di output superiore a 1,

```
function [somma, prodotto]= stat(x)
% stat(x) Fornisce in output due numeri
% contenenti la somma ed il prodotto degli
% elementi di un vettore di input x.
somma = sum(x);
prodotto = prod(x);
end
```

La funzione `stat` deve essere richiamata mediante l'istruzione:

```
>> [p q]=stat(y);
```

così alle variabili `p` e `q` vengono assegnate rispettivamente la somma e il prodotto degli elementi di `y`.
In definitiva se la funzione ammette più di un parametro di output allora la prima riga del function file deve essere modificata nel seguente modo:

```
function [var_0,var_1,var_2]= nomefunzione(inp_0,inp_1)
```

Come ulteriore esempio scriviamo una funzione Matlab per calcolare il valore assunto da un polinomio per un certo valore x. Ricordiamo che assegnato un polinomio di grado n

$$p(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_n x^{n-1} + a_{n+1} x^n$$

la seguente *Regola di Horner* permette di valutare il polinomio minimizzando il numero di operazioni necessarie. Tale regola consiste nel riscrivere lo stesso polinomio in questo modo:

$$p(x) = a_1 + x(a_2 + x(a_3 + \cdots + x(a_n + a_{n+1}x)\ldots))).$$

In questo modo il numero di moltiplicazioni necessarie passa all'incirca da $O(n^2/2)$ a $O(n)$. Vediamo ora la funzione Matlab che implementa tale regola.

```
function y= horner(a,x,n)
% horner(a,x,n) Fornisce in output il valore
% di un polinomio di grado n nel punto x
% a vettore di n+1 elementi contenente i
%  coefficienti del polinomio
% x punto dove si vuol calcolare il polinomio
% n grado del polinomio
p=0;
for i=n+1:-1:1
   p=p*x+a(i);
end
y=p;
```

Per interrompere l'esecuzione di una funzione e tornare al programma chiamante si usa l'istruzione

```
return
```

Tale istruzione può essere anche quella che chiude una funzione (anche se la sua presenza non è obbligatoria).
Un altro modo di definire le funzioni è quello di utilizzare la funzione `inline`, che tuttavia pur essendo attualmente disponibile nelle librerie MatLab sarà eliminata in una delle prossime versioni. Un esempio di applicazione di tale funzione è il seguente:

```
g = inline('t^2')
```

Alla variabile `g` viene assegnata la funzione t^2. Questa assegnazione consente di poter utilizzare la funzione come variabile che può essere passata come parametro di input ad altre funzioni. Quando una funzione viene passata come parametro può essere calcolata usando la funzione `feval` nel seguente modo

```
y = feval(g,0.1)
```

In questo caso il valore di `y` sarà quello della funzione `g` calcolata nell'argomento 0.1.
Un modo migliore per definire funzioni dalla riga di comando è quello di usare il simbolo @ nel seguente modo

```
f = @(x)sin(x)+cos(x)*exp(x);
```

Dopo il simbolo @ si indicano gli argomenti della funzione tra parentesi tonde e poi l'espressione della funzione. In questo modo la funzione può essere passata come parametro ad un'altra funzione e per valutarla non è necessario utilizzare il comando `feval`.
All'interno di una funzione possono essere utilizzate due variabili, `nargin`, che è uguale al numero di parametri di input che sono passati alla funzione, e `nargout`, uguale al numero di parametri che si vogliono in output. Infatti una caratteristica sintattica è che una funzione può accettare un numero variabile di parametri di input o fornire un numero variabile di parametri di output.

1.8 Messaggi di errore, Istruzioni di Input

Stringhe di testo possono essere visualizzate sullo schermo mediante l'istruzione `disp`. Per esempio

```
disp('Messaggio sul video')
```

Se la stringa tra parentesi contiene un apice allora deve essere raddoppiato. La stessa istruzione può essere utilizzata anche per visualizzare il valore di una variabile: è sufficiente scrivere, al posto della stringa e senza apici, il nome della variabile.
I messaggi di errore possono essere visualizzati con l'istruzione `error`. Consideriamo il seguente esempio:

```
if a==0
   error('Divisione per zero')
else
   b=b/a;
end
```

l'istruzione `error` causa l'interruzione nell'esecuzione del file.
In un M-file è possibile introdurre dati di input in maniera interattiva mediante l'istruzione `input`. Per esempio quando l'istruzione

```
iter = input('Inserire il numero di iterate ')
```

viene eseguita allora il messaggio è visualizzato sullo schermo e il programma rimane in attesa che l'utente digiti un valore da tastiera e tale valore, di qualsiasi tipo esso sia, viene assegnato alla variabile `iter`.
Vediamo ora un modo per poter memorizzare in un file di ascii l'output di un M-file oppure di una sequenza di istruzioni Matlab. Infatti

```
>> diary nomefile
>> istruzioni
>> diary off
```

serve a memorizzare nel file *nomefile* tutte le istruzioni e l'output che è stato prodotto dopo la prima chiamata della funzione e prima della seconda.
Un'altra istruzione di output è `fprintf` che consente di stampare messaggi e contenuto delle variabili su video (di default) o su file (specificando un identificatore di file aperto con il comando `fopen`). Il primo parametro è una stringa che descrive il formato di visualizzazione dell'output. Giusto per fare un esempio

```
>> fprintf('%6.2f %12.8f\n',x,y);
```

stampa la variabile reale `x` utilizzando il formato per numeri floating point (lettera `f`) con 6 cifre delle quali 2 per la parte decimale, la variabile reale `y` viene rappresentata con 12 cifre delle quali 8 per la parte decimale, e \n fa andare a capo. Si rimanda al manuale MatLab (o all'help in linea per una descrizione più dettagliata di tale funzione).

1.8.1 Formato di output

In Matlab tutte le elaborazioni vengono effettuate in doppia precisione. Il formato con cui l'output compare sul video può però essere controllato mediante i seguenti comandi.

```
format short
```

È il formato utilizzato per default dal programma ed è di tipo fixed point con 4 cifre decimali;

```
format long
```

Tale formato è di tipo fixed point con 14 cifre decimali;

```
format short e
```

Tale formato è la notazione scientifica (esponenziale) con 4 cifre decimali;

```
format long e
```

Tale formato è la notazione scientifica (esponenziale) con 15 cifre decimali. Vediamo per esempio come i numeri $4/3$ e $1.2345e-6$ sono rappresentati nei formati che abbiamo appena descritto e negli altri disponibili:

```
format short            1.3333
format short e          1.3333e+000
format short g          1.3333
format long             1.33333333333333
format long e           1.333333333333333e+000
format long g           1.33333333333333
format rat              4/3


format short            0.0000
format short e          1.2345e-006
format short g          1.2345e-006
format long             0.00000123450000
format long e           1.234500000000000e-006
format long g           1.2345e-006
format rat              1/810045
```

Oltre ai formati appena visti il comando

```
format compact
```

serve a sopprimere le righe vuote e gli spazi dell'output scrivendo sullo schermo il maggior numero di informazioni possibile, in modo appunto compatto.

1.9 La grafica con il Matlab

Il Matlab dispone di numerose istruzioni per grafici bidimensionali e tridimensionali e anche di alcune funzioni per la creazione di animazioni. Il comando `plot` serve a disegnare curve nel piano xy. Infatti se `x` e `y` sono due vettori di uguale lunghezza allora il comando

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

traccia una curva spezzata che congiunge i punti `(x(i),y(i))`. Per esempio

```
>> x=-4:.01:4;
>> y=sin(x);
>> plot(x,y)
```

traccia il grafico della funzione seno nell'intervallo $[-4, 4]$.
L'istruzione `plot` ammette un parametro opzionale di tipo stringa (racchiuso tra apici) per definire il tipo e il colore del grafico. Infatti è possibile scegliere tra 4 tipi di linee, 5 di punti e 8 colori base. In particolare

`'-'`	linea continua
`'--'`	linea tratteggiata
`'-.'`	linea tratteggiata e a punti
`':'`	linea a punti

`'+'`	piu'
`'o'`	cerchio
`'x'`	croce
`'.'`	punto
`'*'`	asterisco

`'y'`	colore giallo
`'r'`	colore rosso
`'c'`	colore ciano
`'m'`	colore magenta
`'g'`	colore verde
`'w'`	colore bianco
`'b'`	colore blu
`'k'`	colore nero

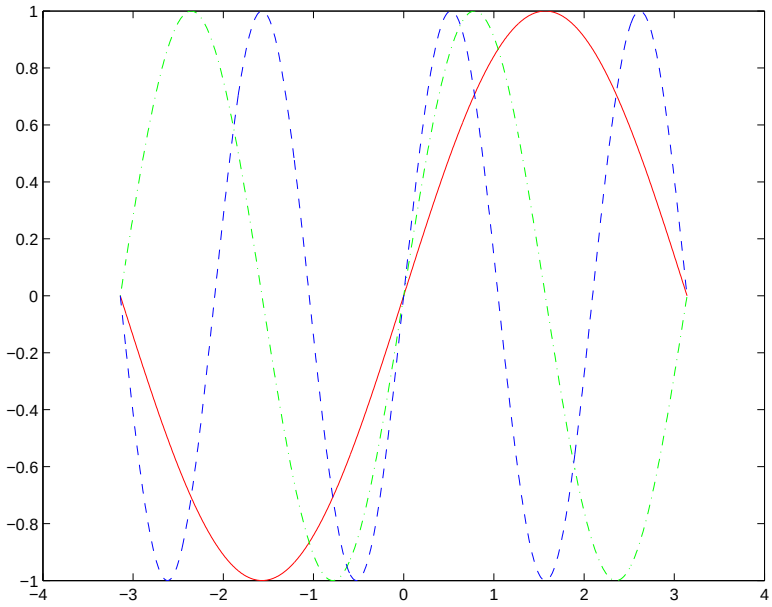

Figura 1.1:

Volendo tracciare per esempio un grafico con linea a puntini e di colore verde l'istruzione è:

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

L'istruzione `plot` consente di tracciare più grafici contemporaneamente. Per esempio

```
>> x=-pi:pi/500:pi;
>> y=sin(x);
>> y1=sin(2*x);
>> y2=sin(3*x);
>> plot(x,y,'r',x,y1,'-.g',x,y2,'--b')
```

traccia tre grafici nella stessa figura, il primo a tratto continuo (tratto di default) rosso, il secondo verde tratteggiato e a punti, il terzo tratteggiato e di colore blu. Nella Figura 1.1 è riportato il risultato di tale istruzione.
È possibile anche decidere la larghezza del tratto, impostando il parametro `LineWidth` dell'istruzione `plot`:

```
>> plot(x,y,'LineWidth',2)
```

Il valore intero specificato indica lo spessore del tratto della curva tracciata
Vediamo ora le altre più importanti istruzioni grafiche:

```
>> title(stringa)
```

serve a dare un titolo al grafico che viene visualizzato al centro nella parte superiore della figura;

```
>> xlabel(stringa)
```

stampa una stringa al di sotto dell'asse delle ascisse;

```
>> ylabel(stringa)
```

stampa una stringa a destra dell'asse delle ordinate (orientata verso l'alto).
Per inserire un testo in una qualsiasi parte del grafico esiste il comando

```
>> text(x,y,'testo')
```

che posiziona la stringa di caratteri *testo* nel punto di coordinate (x, y) (`x` e `y` non devono essere vettori). Di tale comando ne esiste anche una versione che utilizza il mouse:

```
>> gtext('testo')
```

posiziona il testo nel punto selezionato all'interno del grafico schiacciando il pulsante sinistro del mouse.
La finestra grafica può essere partizionata in un certo numero di finestre più piccole, per esempio il comando

```
>> subplot(m,n,p)
```

divide la finestra grafica in una matrice `m` $\times$ `n` di finestre più piccole e seleziona la $p-$esima. Le finestre sono numerate iniziando dalla prima in alto a sinistra e poi si procede verso destra per poi passare alla riga successiva, verso il basso.
Il grafico tracciato con il comando `plot` è scalato automaticamente, questo vuol dire che le coordinate della finestra grafica sono calcolate dal programma, tuttavia l'istruzione

```
>> axis([x_min x_max y_min y_max])
```

consente di ridefinire gli assi, e quindi le dimensioni della finestra del grafico corrente. A volte può essere utile, una volta tracciato un grafico, ingrandire alcune parti dello stesso. Questo può essere fatto utilizzando il comando `zoom`. Per attivare tale caratteristica è sufficiente il comando

```
>> zoom on
```

mentre per disattivarlo bisogna scrivere:

```
>> zoom off
```

Il funzionamento di tale istruzione è molto semplice. Una volta attivato lo zoom per ingrandire un'area del grafico è sufficiente portare il puntatore del mouse in tale area e cliccare con il tasto sinistro dello stesso. Tale operazione può essere ripetuta alcune volte (non si può ottenere l'ingrandimento un numero molto grande di volte). Per effettuare uno zoom a ritroso bisogna cliccare con il tasto destro del mouse.
La stessa funzione può essere attivata utilizzando una delle icone che si trovano nel menu della finestra grafica.
Le istruzioni grafiche del Matlab permettono di tracciare curve in tre dimensioni, superfici, di creare delle animazioni e così via. Per approfondire le istruzioni che consentono queste operazioni si può richiedere l'`help` per le istruzioni `plot3`, `mesh` e `movie`.
Nel caso di una superficie, per tracciare il grafico si devono definire due vettori, uno per le ascisse, cioè `x`, uno per le ordinate, `y`, e una matrice `A` per memorizzare le quote, cioè tale che

```
A(j,i)=f(x(j),y(i))
```

Nella Figura 1.2 è tracciato come esempio il grafico della funzione

$$f(x,y) = x(10-x) + y(10-y), \qquad 0 \le x, y \le 10.$$

Le istruzioni per tracciare tale grafico sono le seguenti:

```
x=[0:0.1:10];
y=[0:0.1:10];
n=length(x);
for i=1:n
    for j=1:n
```

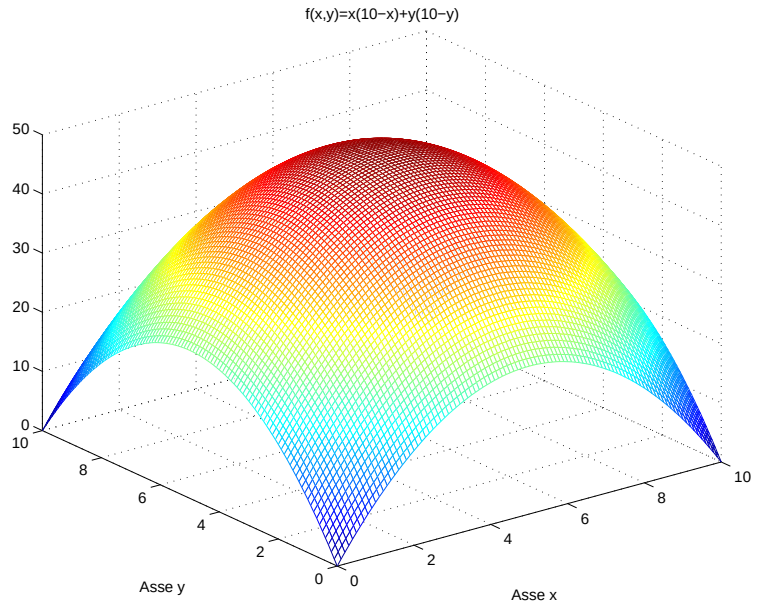

Figura 1.2:

```
        A(j,i)=x(j)*(10-x(j))+y(i)*(10-y(i));
    end
end
mesh(x,y,A);
xlabel('Asse x');
ylabel('Asse y');
title('f(x,y)=x(10-x)+y(10-y)');
```

Capitolo 2

Laboratorio MatLab

2.2 Introduzione a MATLAB

Scopo: Eseguire alcune semplici istruzioni MatLab e imparare l'uso della grafica.

Scopo di questa prima esercitazione è quello di iniziare a conoscere l'ambiente MatLab ed in particolare le istruzioni per la manipolazione di matrici e vettori, le funzioni predefinite, le istruzioni per la grafica e quelle di iterazione e selezione.
Una volta lanciato il programma iniziare la sessione di lavoro assegnando alcuni vettori o matrici:

```
>> x = [1 4 5 3 -4 5]
>> y = [-1; 0; -5; 13; 4; -5]
>> length(x)
>> length(y)
>> z=x+y
>> z=x'+y
>> a=x*y
>> a=y*x
>> x=[x 10]
>> n=length(x)
>> x=x(n:-1:1)
```

Dal risultato delle operazioni precedenti si è potuto osservare che la somma dei due vettori non è consentita a meno che questi non abbiano esattamente

le stesse dimensioni (cioè siano due vettori riga o colonna della stessa lunghezza). Anche per il prodotto le dimensioni devono essere compatibili. Un vettore riga (di dimensione $1 \times n$) può essere moltiplicato per un vettore colonna (di dimensione $n \times 1$) e dà come risultato un valore scalare. Un vettore colonna (di dimensione $n \times 1$) può essere moltiplicato per un vettore riga (di dimensione $1 \times n$) e produce come risultato una matrice quadrata di dimensione n. In ultimo osserviamo che l'ultima istruzione di questo blocco inverte gli elementi del vettore.

```
>> a=max(x)
>> [a i]=max(x)
>> a1=min(x)
>> [a k]=min(x)
>> sort(x)
```

In questo caso possiamo osservare come le funzioni predefinite `max` e `min` possano dare due tipi di output diversi, cioè possono fornire solo il valore del massimo (o del minimo) del vettore ma anche l'indice della componente massima (o minima).
Vediamo ora alcune istruzioni che riguardano le matrici.

```
>> A = [1 4 5 3; 0 1 -4 5; 3 4 5 6; -1 0 1 9 ...
; 0 7 6 -9]
>> A(1:3,4)
>> A(2,2:4)
>> A(:,4)
>> A(5,:)
>> A(1:3,2:4)
>> A([1 5],:)=A([5 1],:)
>> [m,n]=size(A)
>> x=max(A)
>> x=max(max(A))
>> B=cos(A)
```

La prima istruzione di questo blocco consiste nell'assegnazione di una matrice 5×4 alla variabile A. Si osservi la funzione dei tre punti che servono a spezzare su più righe istruzioni troppo lunghe. Nelle altre possiamo osservare come la cosiddetta notazione due punti permetta di visualizzare in modo

compatto porzioni di righe o di colonne, o intere sottomatrici. La sesta istruzione permette di poter scambiare simultaneamente due righe di una stessa matrice (istruzione analoga vale anche per le colonne) senza l'ausilio di vettori ausiliari. Va infine osservato cosa succede se si applica una funzione di tipo vettoriale (in questo caso `max`) ad una matrice: il risultato è un vettore, che (in questo caso) contiene i massimi delle colonne di A. Applicandolo due volte si ottiene come risultato il massimo elemento della matrice.
Vediamo ora di scrivere la seguente funzione che calcoli le radici del polinomio di secondo grado

$$ax^2 + bx + c$$

che indichiamo con `x1` e `x2`. Come noto, posto

$$\Delta = b^2 - 4ac$$

allora

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a}, \qquad x_2 = \frac{-b - \sqrt{\Delta}}{2a}.$$

```
function [x1,x2]=radici(a,b,c)
%
% Sintassi [x1,x2]=radici(a,b,c)
%
% Calcola le radici di un polinomio di secondo grado
%
Delta = b^2-4*a*c;
x1 = (-b+sqrt(Delta))/(2*a);
x2 = (-b-sqrt(Delta))/(2*a);
end
```

Applichiamo ora la funzione al polinomio che ammette come radici i due numeri $x_1 = 10^7$ e $x_2 = 10^{-7}$. In questo caso i valori dei coefficienti a, b e c sono:

$$a = 1, \qquad b = -(10^7 + 10^{-7}), \qquad c = 1.$$

Scriviamo pertanto le seguenti istruzioni:

```
>> a = 1;
>> b = -(10^7+10^(-7));
>> c = 1;
>> [x1,x2] = radici(a,b,c);
```

Adesso provvediamo a modificare la funzione nel seguente modo:

```
function [x1,x2]=radici1(a,b,c)
%
% Sintassi [x1,x2]=radici1(a,b,c)
%
% Calcola le radici di un polinomio di secondo grado
%
Delta = b^2-4*a*c;
x1 = (-b-sign(b)*sqrt(Delta))/(2*a);
x2 = c/x1;
end
```

Scriviamo pertanto le seguenti istruzioni:

```
>> a = 1;
>> b = -(10^7+10^(-7));
>> c = 1;
>> [r1,r2] = radici1(a,b,c);
```

Osserviamo la differenza tra i valori calcolati.
Proviamo ora a tracciare il grafico di una funzione. In MatLab ciò può essere fatto in molti modi diversi, vediamone solo i più semplici. Innanzitutto scegliamo una funzione, per esempio:

$$f(x) = \sin^2(x)\cos(x) + (\sin(e^x))^2 + 1$$

e decidiamo di tracciarne il grafico nell'intervallo $[0, 2\pi]$. Come è noto un grafico in MatLab non è nient'altro se non una spezzata che congiunge un insieme discreto di punti del piano. Per prima cosa dobbiamo scegliere nell'intervallo un certo numero di punti equidistanti, per esempio 100 punti, utilizzando la seguente istruzione:

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

Adesso dobbiamo calcolare il valore della funzione $f(x)$ nel vettore delle ascisse appena assegnato. Il modo più semplice è quello di utilizzare una variabile di tipo stringa per memorizzare la funzione attraverso la funzione `inline`:

```
>> funz=inline('(sin(x).^2).*cos(x)+(sin(exp(x))).^2+1')
```

Osserviamo che quando alla variabile `funz` viene assegnata una funzione le operazioni che compaiono nella stringa devono essere considerate come se fossero applicate a vettori.
A questo punto per calcolare il valore della funzione nel vettore `x` si può utilizzare la funzione `feval`:

```
>> y=feval(funz,x);
```

A questo punto si può procedere a tracciare il grafico della funzione:

```
>> plot(x,y,'b-');
```

Il grafico è stato tracciato in blu a tratto continuo, ma possiamo anche variare il colore e il tipo di tratto, proviamo le seguenti istruzioni:

```
>> plot(x,y,'y--');
>> plot(x,y,'r:');
>> plot(x,y,'go');
```

Un secondo modo per tracciare il grafico è quello di utilizzare la funzione predefinita `fplot`. In questo caso il modo di procedere è lo stesso tranne per la definizione del vettore delle ascisse che non va assegnato:

```
>> fplot(funz,[0 2*pi]);
```

Infatti i parametri di tale funzione sono solo la stringa contenente la funzione e l'intervallo di variabilità delle ascisse.
Tracciando i diversi grafici si è potuto osservare che ogni volta che viene aperta una nuova figura la precedente viene cancellata. Per poter tracciare più grafici su una stessa figura va utilizzata l'opzione `hold on` nel seguente modo:

```
>> fplot(funz,[0 2*pi]);
>> hold on
>> g=inline('2+sin(x).*cos(x)');
>> y1=feval(g,x);
>> plot(x,y1);
```

Una volta che tale opzione è eseguita essa rimane attiva per tutta la sessione di lavoro. Questo vuol dire che tutti i grafici che saranno tracciati successivamente si andranno a sovrapporre sulla stessa figura. Per disattivare tale opzione è sufficiente l'istruzione

```
>> hold off
```

2.3 Equazioni non lineari

Argomento: **Il metodo delle successive bisezioni**
Riferimenti teorici: Paragrafo 2.3
Scopo: Implementare il metodo delle successive bisezioni per la soluzione di equazioni non lineari.

```
function [alfa,k]=bisezione(f,a,b,tol)
%BISEZIONE Calcola la soluzione approssimata di un'equazione
% non lineare applicando il metodo di bisezione
%
% Parametri di input
% f = funzione della quale calcolare la radice
% a = estremo sinistro dell'intervallo
% b = estremo destro dell'intervallo
% tol = precisione fissata
%
% Parametri di output
% alfa = approssimazione della radice
% k = numero di iterazioni
%
if nargin==3
    tol = 1e-8; % Tolleranza di default
end
fa = f(a);
fb = f(b);
if fa*fb>0
    error('Il metodo non e'' applicabile')
end
c = (a+b)/2;
fc = f(c);
k = 0;
while (b-a)>tol | abs(fc)>tol
    if fa*fc<0
        b = c;
        fb = fc;
    else
```

```
        a = c;
        fa = fc;
    end
    c = (a+b)/2;
    fc = f(c);
    if nargout==2
        k = k+1;
    end
end
alfa = c;
end
```

Esempio di applicazione: La funzione richiede in ingresso la variabile stringa dove è memorizzata la funzione, gli estremi dell'intervallo e la precisione voluta. Come esempio si può considerare la funzione

$$f(x) = x - e^{-x}$$

e prendere come intervallo iniziale $[0, 1]$ e fissare come precisione $\varepsilon = 10^{-8}$.

```
>> format long e
>> f=@(x)x-exp(-x)
>> a=0;
>> b=1;
>> epsilon=1e-8;
>> [alfa, iter]=bisez(f,a,b,epsilon)
```

Possibili modifiche:
La funzione appena descritta prevede come parametri di output un'approssimazione della radice e il numero di iterate, tuttavia quest'ultimo può essere calcolato a priori tenendo conto che, una volta nota la precisione richiesta il numero di iterate necessario per calcolare l'approssimazione è

$$k > \log_2 \left(\frac{b-a}{\varepsilon} \right).$$

Si potrebbe calcolare tale valore di k e trasformare il ciclo `while` in un ciclo `for` e vedere se i risultati del metodo sono gli stessi nei due casi.

Argomento: **Il metodo di Newton-Raphson**
Riferimenti teorici: Paragrafo 2.4.2
Scopo: Implementare il metodo di Newton-Raphson per la soluzione di equazioni non lineari.

```
function [alfa,k]=newton(f,f1,x0,tol,Nmax)
%NEWTON Calcola la soluzione approssimata di un'equazione
% non lineare applicando il metodo di Newton-Raphson
%
% Parametri di input
% f = funzione della quale calcolare la radice
% f1 = derivata prima della funzione f
% x0 = approssimazione iniziale della radice
% tol = precisione fissata
% Nmax = numero massimo di iterazioni fissate
%
% Parametri di output
% alfa = approssimazione della radice
% k = numero di iterazioni
%
if nargin==3
    tol=1e-8;
    Nmax=1000;
end
k=0;
x1=x0-f(x0)/f1(x0);
fx1 = f(x1);
while abs(x1-x0)>tol || abs(fx1)>tol
    x0 = x1;
    x1 = x0-f(x0)/f1(x0);
    fx1 = f(x1);
    k=k+1;
    if k>Nmax
        error('Il metodo non converge');
        break
    end
end
```

```
alfa=x1;
end
```

Esempio di applicazione: La funzione richiede in ingresso le variabili di tipo stringa dove sono memorizzate la funzione e la sua derivata prima, l'approssimazione iniziale x_0, la precisione voluta e il numero massimo di iterate. Per esempio si può considerare la funzione

$$f(x) = x^3 - 3x + 2 \qquad\qquad f'(x) = 3x^2 - 3$$

prendendo come approssimazione iniziale prima $x_0 = -2.5$ e poi $x_0 = 1.4$, e fissando come precisione $\varepsilon = 10^{-8}$.

```
>> format long e
>> f=@(x)x^3-3*x+2;
>> f1=@(x)3*x^2-3;
>> x0=-2.5;
>> epsilon=1e-8;
>> maxiter=100;
>> [alfa0,iter0]=newtraph(f,f1,x0,epsilon,maxiter)
>> x0=1.4;
>> [alfa1,iter1]=newtraph(f,f1,x0,epsilon,maxiter)
```

Dai risultati emerge il diverso comportamento del metodo per le due diverse radici: infatti la radice -2 è semplice e il metodo di Newton-Raphson converge con ordine 2 (quindi più rapidamente), mentre per la radice doppia 1 la convergenza è più lenta poichè l'ordine è 1.

2.4 Risoluzione di sistemi lineari triangolari

Riferimenti teorici: Capitolo 3, Paragrafo 3.2
Scopo: Implementare i metodi di sostituzione in avanti e all'indietro per sistemi triangolari inferiori e superiori.

```
function x=indietro(A,b)
%INDIETRO Risolve un sistema triangolare superiore
% applicando il metodo di sostituzione all'indietro
```

```
%
%
% Parametri di input:
% A = Matrice triangolare superiore
% b = Vettore colonna
%
% Parametri di output:
% x = Vettore soluzione
%
n=length(b);
x=zeros(n,1);
if abs(A(n,n))<eps
  error('La matrice A e'' singolare ');
end
x(n)=b(n)/A(n,n);
for k=n-1:-1:1
  x(k)=b(k);
  for i=k+1:n
      x(k)=x(k)-A(k,i)*x(i);
  end
  if abs(A(k,k))<eps
     error('La matrice A e'' singolare ');
  else
     x(k)=x(k)/A(k,k);
  end
end
end
```

Possibili modifiche:
La routine appena descritta risolve un sistema triangolare superiore. Osserviamo innanzitutto che se viene incontrato un elemento diagonale più piccolo, in modulo, della precisione di macchina allora l'algoritmo segnala un errore. Si può inoltre osservare che la routine potrebbe essere scritta in modo più compatto utilizzando la notazione : del MatLab. Infatti il ciclo descritto dalla variabile `i` si potrebbe sostituire con un'unica istruzione:

```
x(k)=b(k)-A(k,k+1:n)*x(k+1:n);
```

Per completezza vediamo anche l'implementazione del metodo di sostituzione in avanti per matrici triangolari inferiori.

```
function x=avanti(A,b)
%AVANTI Risolve un sistema triangolare inferiore
% applicando il metodo di sosttituzione in avanti
%
% Parametri di input:
% A = Matrice triangolare inferiore
% b = Vettore colonna
%
% Parametri di output:
% x = Vettore soluzione
%
n=length(b);
x=zeros(n,1);
if abs(A(1,1))<eps
   error('La matrice A e'' singolare ');
end
x(1)=b(1)/A(1,1);
for k=2:n
    x(k)=b(k)-A(k,1:k-1)*x(1:k-1);
    if abs(A(k,k))<eps
       error('La matrice A e'' singolare ');
    else
       x(k)=x(k)/A(k,k);
    end
end
end
```

2.5 Risoluzione di sistemi lineari

Argomento: **Il metodo di eliminazione di Gauss**
Riferimenti teorici: Paragrafo 3.3
Scopo: Implementare il metodo di eliminazione per la soluzione di sistemi lineari e le tecniche di pivoting parziale e totale.

```
function x=Gauss(A,b)
%GAUSS Risolve un sistema lineare applicando
% il metodo di eliminazione di Gauss
%
% Parametri di input:
% A = Matrice dei coefficienti del sistema
% b = Vettore dei termini noti del sistema
%
% Parametri di output:
% x = Vettore soluzione del sistema lineare
%
n = length(b);
x = zeros(n,1);
for k=1:n-1
  if abs(A(k,k))<eps
    error('Elemento pivotale nullo ')
  end
  for i=k+1:n
    A(i,k) = A(i,k)/A(k,k);
    b(i) = b(i)-A(i,k)*b(k);
    for j=k+1:n
      A(i,j) = A(i,j)-A(i,k)*A(k,j);
    end
 end
end
%
% Risoluzione del sistema triangolare superiore
%
x(n) = b(n)/A(n,n);
for i=n-1:-1:1
  x(i) = (b(i)-A(i,i+1:n)*x(i+1:n))/A(i,i);
end
end

function x=Gauss_pp(A,b)
%GAUSS_PP Risolve un sistema lineare applicando il metodo di
% eliminazione di Gauss e la strategie di pivoting parziale
%
```

```
% Parametri di input:
% A = Matrice dei coefficienti del sistema
% b = Vettore dei termini noti del sistema
%
% Parametri di output:
% x = Vettore soluzione del sistema lineare
%
n = length(b);
x = zeros(n,1);
for k=1:n-1
  [a,i] = max(abs(A(k:n,k)));
  i = i+k-1;
  if i~=k
    A([i k],:) = A([k i],:);
    b([i k]) = b([k i]);
  end
  for i=k+1:n
    A(i,k) = A(i,k)/A(k,k);
    b(i) = b(i)-A(i,k)*b(k);
    for j=k+1:n
      A(i,j) = A(i,j)-A(i,k)*A(k,j);
    end
 end
end
%
% Risoluzione del sistema triangolare superiore
%
x(n) = b(n)/A(n,n);
for i=n-1:-1:1
  x(i) = (b(i)-A(i,i+1:n)*x(i+1:n))/A(i,i);
end
end

function x=Gauss_pt(A,b)
%GAUSS_PT Risolve un sistema lineare applicando il metodo di
% eliminazione di Gauss e la strategie di pivoting totale
%
% Parametri di input:
```

```
% A = Matrice dei coefficienti del sistema
% b = Vettore dei termini noti del sistema
%
% Parametri di input:
% x = Vettore soluzione del sistema lineare
%
n = length(b);
x = zeros(n,1);
x1 = x;
indice = [1:n];
for k=1:n-1
  [a,riga] = max(abs(A(k:n,k:n)));
  [mass,col] = max(a);
  j = col+k-1;
  i = riga(col)+k-1;
  if i~=k
  A([i k],:) = A([k i],:);
  b([i k]) = b([k i]);
  end
  if j~=k
  A(:,[j k]) = A(:,[k j]);
    indice([j k]) = indice([k j]);
  end
  for i=k+1:n
    A(i,k) = A(i,k)/A(k,k);
    b(i) = b(i)-A(i,k)*b(k);
    for j=k+1:n
      A(i,j) = A(i,j)-A(i,k)*A(k,j);
    end
  end
end
%
% Risoluzione del sistema triangolare superiore
%
x1(n) = b(n)/A(n,n);
for i=n-1:-1:1
  x1(i) = (b(i)-A(i,i+1:n)*x1(i+1:n))/A(i,i);
end
```

```
%
% Ripermutazione del vettore
%
for i=1:n
  x(indice(i))=x1(i);
end
end
```

Esempi di applicazione: Per verificare il funzionamento dell'algoritmo si può applicare ad un sistema lineare avente una matrice dei coefficienti a predominanza diagonale per colonne.

```
>> A=[6 4 1 0;-1 8 1 1;3 0 6 -3;1 -2 1 7]
>> b=[1;2;3;4]
>> x=gauss(A,b)
```

Per verificare invece che il metodo di Gauss non funziona se la matrice dei coefficienti ammette un minore principale uguale a zero si può applicarlo in questa circostanza e verificare che la routine appena scritta segnala tale circostanza.

```
>> A=[1 1 2 1 0;2 1 3 1 -4;-1 -1 -2 3 0;4 2 -1 1 0;5 2 -2 1 7]
>> b=[1;2;3;4;5]
>> x=gauss(A,b)
```

Ci sono casi in cui il metodo di eliminazione di Gauss può fornire una soluzione del sistema molto diversa da quella teorica. Vediamo il seguente esempio: scegliamo come matrice dei coefficienti una cosiddetta matrice di Hilbert, definita nel seguente modo:

$$h_{ij} = \frac{1}{i+j-1} \qquad i,j = 1,\ldots,n.$$

Per esempio se $n = 4$ la matrice sarebbe

$$H = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}.$$

Proviamo ora ad applicare il metodo di Gauss ad un sistema di dimensione 15 avente come matrice dei coefficienti quella di Hilbert e come soluzione il vettore avente tutte le componenti uguali a 1 e confrontiamo la soluzione che ci fornisce il metodo di Gauss con quella teorica.

```
>> clear
>> format long e
>> n=15;
>> A=hilb(n);
>> x=ones(n,1);
>> b=A*x;
>> y=gauss(A,b)
>> norm(x-y,'inf')
```

Le prime due istruzioni servono rispettivamente a cancellare tutte le variabili presenti nell'area di lavoro del MatLab e a scrivere i valori delle variabili in formato esponenziale lungo, cioè con 15 cifre decimali. La funzione `hilb(n)` assegna ad una variabile la matrice di Hilbert della dimensione indicata. Il vettore `b` viene assegnato in modo tale che la soluzione del sistema, cioè il vettore colonna `x`, sia nota. Nella variabile `y` viene memorizzata la soluzione del sistema calcolata utilizzando il metodo di Gauss. L'ultima istruzione serve a dare una misura della differenza tra la soluzione teorica del sistema e quella calcolata utilizzando la funzione `norm` che, in questo caso, misura la norma infinito della differenza tra i due vettori, cioè il massimo valore assoluto del vettore differenza `x-y`.

2.5.1 Fattorizzazione LU

Argomento: **Calcolo diretto della fattorizzazione con le tecniche di Crout e Doolittle**
Riferimenti teorici: Paragrafo 3.3.3

```
function [L,U]=crout(A);
%CROUT Calcola la fattorizzazione LU di una matrice quadrata
% applicando la tecnica di Crout
%
% Parametri di input
```

```
% A = matrice da fattorizzare
% Parametri di output
% L = matrice triang. inferiore con elementi diagonali
%     uguali a 1
% U = matrice triangolare superiore
%
[m n] = size(A);
U = zeros(n);
L = eye(n);
U(1,:) = A(1,:);
for i=2:n
    for j=1:i-1
        L(i,j) = (A(i,j) - L(i,1:j-1)*U(1:j-1,j))/U(j,j);
    end
    for j=i:n
        U(i,j) = A(i,j) - L(i,1:i-1)*U(1:i-1,j);
    end
end
end

function [L,U]=doolittle(A);
%DOOLITTLE Calcola la fattorizzazione LU di una matrice quadrata
% applicando la tecnica di Doolittle
%
% Parametri di input
% A = matrice da fattorizzare
% Parametri di output
% L = matrice triang. inferiore con elementi diagonali
%     uguali a 1
% U = matrice triangolare superiore
%
[m n] = size(A);
L = eye(n);
U = zeros(n);
U(1,:) = A(1,:);
for i=1:n-1
    for riga=i+1:n
        L(riga,i)=(A(riga,i)-L(riga,1:i-1)*U(1:i-1,i))/U(i,i);
```

```
    end
    for col=i+1:n
        U(i+1,col) = A(i+1,col)-L(i+1,1:i)*U(1:i,col);
    end
end
end
```

2.6 Interpolazione e approssimazione

Argomento: **Il polinomio interpolante di Lagrange**

Argomento: **Calcolo diretto della fattorizzazione con le tecniche di Crout e Doolittle**
Riferimenti teorici: Paragrafo 4.2
Scopo: Calcolare il polinomio interpolante di Lagrange un insieme discreto di dati.

```
function  yy=lagrange(x,y,xx);
%LAGRANGE calcola il polinomio interpolante di Lagrange
% in un vettore assegnato di ascisse
%
% Parametri di input
% x = vettore dei nodi
% y = vettore delle ordinate nei nodi
% xx = vettore delle ascisse in cui calcolare il polinomio
% Parametri di output
% yy = vettore delle ordinate del polinomio
%
n = length(x);
m = length(xx);
yy = zeros(size(xx));
for i=1:m
    yy(i)=0;
    for k=1:n
        yy(i)=yy(i)+prod((xx(i)-x([1:k-1,k+1:n]))./...
            (x(k)-x([1:k-1,k+1:n])))*y(k);
    end
```

```
end
end
```

Argomento: **Visualizzazione del fenomeno di Runge**
Riferimenti teorici: Paragrafo 4.2
Scopo: Sono rappresentati i polinomi di Lagrange che interpolano la funzione di Runge

$$f(x) = \frac{1}{1+x^2}.$$

su nodi equidistanti e di Chebyshev.

```
clear
format long e
a = input('Inserire estremo sinistro ');
b = input('Inserire estremo destro ');
n = input('Inserire il numero di nodi ');
%
% Calcolo del vettore dei nodi di Chebyshev
%
x = (a+b)/2+(b-a)/2*cos((2*[0:n-1]+1)*pi./(2*n));
xx = linspace(a,b,200);
y = 1./(x.^2+1);
yy = 1./(xx.^2+1);
%
% Calcolo del polinomio interpolante
%
zz = lagrange(x,y,xx);
figure(1)
plot(xx,yy)
hold on
pause
plot(x,y,'ok')
pause
plot(xx,zz,'r')
title('Grafico della funzione e del polinomio interpolante ')
hold off
figure(2)
plot(xx,abs(yy-zz))
title('Grafico dell'errore nell''interpolazione')
```

2.7 Metodi numerici per equazioni differenziali

```
function [tn,yn] = Eulero_esp(f,t0,T,y0,N)
%EULERO_ESP Calcola l'approssimazione di un problema
% ai valori iniziali con il metodo di Eulero esplicito
%   Parametri di input
% f = funzione che definisce il problema di Cauchy
% t0 = istante di tempo iniziale
% T = istante di tempo finale
% y0 = condizione iniziale
% N = numero di sottointervalli
%
h = (T-t0)/N;
tn = linspace(t0,T,N+1);
yn = zeros(size(tn));
yn(1) = y0;
for n=1:N
    yn(n+1)=yn(n)+h*f(tn(n),yn(n));
end
end
```

```
function [tn,yn] = Eulero_imp(f,t0,T,y0,N,tol)
%EULERO_IMP Calcola l'approssimazione di un problema
% ai valori iniziali con il metodo di Eulero implicito
%   Parametri di input
% f = funzione che definisce il problema di Cauchy
% t0 = istante di tempo iniziale
% T = istante di tempo finale
% y0 = condizione iniziale
% N = numero di sottointervalli
% tol = precisione fissata per il metodo di iterazione funzionale
%
h = (T-t0)/N;
tn = linspace(t0,T,N+1);
yn = zeros(size(tn));
yn(1) = y0;
for n=1:N
```

```
    x0 = yn(n);
    errore = 2*tol;
%
% Risoluzione dell'equazione non lineare con il
% metodo di iterazione funzionale
%
    while errore>tol
        x1 = yn(n)+h*f(tn(n+1),x0);
        errore = abs(x1-x0);
        x0 = x1;
    end
    yn(n+1)=x1;
end
end

function [tn,yn] = Midpoint(f,t0,T,y0,N)
%MIDPOINT Calcola l'approssimazione di un problema
% ai valori iniziali con il metodo del Midpoint
%   Parametri di input
% f = funzione che definisce il problema di Cauchy
% t0 = istante di tempo iniziale
% T = istante di tempo finale
% y0 = condizione iniziale
% N = numero di sottointervalli
%
h = (T-t0)/N;
tn = linspace(t0,T,N+1);
yn = zeros(size(tn));
yn(1) = y0;
yn(2) = yn(1)+h*f(tn(1),yn(1));
for n=1:N-1
    yn(n+2)=yn(n)+2*h*f(tn(n+1),yn(n+1));
end
end

function [tn,yn] = Trapezi(f,t0,T,y0,N,tol)
%TRAPEZI Calcola la soluzione approssimata di un problema
% ai valori iniziali con il metodo dei Trapezi
```

```
%   Parametri di input
% f = funzione che definisce il problema di Cauchy
% t0 = istante di tempo iniziale
% T = istante di tempo finale
% y0 = condizione iniziale
% N = numero di sottointervalli
% tol = precisione fissata per il metodo di iterazione funzionale
%
h = (T-t0)/N;
tn = linspace(t0,T,N+1);
yn = zeros(size(tn));
yn(1) = y0;
h2 = h/2;
for n=1:N
    x0 = yn(n);
    errore = 2*tol;
    gn = yn(n)+h2*f(tn(n),yn(n));
    while errore>tol
        x1 = gn+h2*f(tn(n+1),x0);
        errore = abs(x1-x0);
        x0 = x1;
    end
    yn(n+1)=x1;
end
end
```