Parte I: MATLAB e il calcolo scientifico

Introduzione

Anche se la varietà dei problemi nei quali i calcolatori sono utilizzati è estremamente ampia e sempre crescente, esistono idee base che sono comuni a molte applicazioni e in diversi contesti.

Calcolo numerico e calcolo simbolico

Un calcolo numerico coinvolge direttamente dei numeri, non i simboli che rappresentano i numeri in un'espressione. Ad esempio quando scriviamo

$$(3.2 + 4.1)^2 = 53.29,$$

eseguiamo un calcolo numerico.

Al contrario un calcolo simbolico coinvolge la manipolazione di simbolici matematici senza alcun riferimento ai valori numerici che tali simboli possono assumere. Avremo quindi che

$$(a+b)^2 = a^2 + b^2 + 2a\,b,$$

rappresenta un calcolo simbolico.

Uno degli aspetti fondamentali nel calcolo numerico è l'utilizzo di approssimazioni al posto di risultati numerici esatti.

Ad esempio un numero razionale come $2/3 = 0.66666...$ o alcune costanti fondamentali quali $\pi = 3.14159...$ oppure $e = 2.71828...$, dove i punti "..." indicano la necessità di utilizzare un numero infinito di cifre.

Il calcolatore non utilizza i simboli $2/3$, π ed e ma una loro approssimazione numerica rappresentata dal troncamento del valore esatto ad un numero finito di cifre.

Esempio 1 (Calcolo simbolico e numerico con numeri razionali) Consideriamo il semplice calcolo

$$\frac{2}{3} + \frac{1}{3} + 1 = 2.$$

I simboli 2/3 e 1/3 hanno i valori numerici 0.66666... e 0.33333... e non possono essere rappresentati con un numero finito di cifre decimali. Se consideriamo solo le prime tre cifre decimali nell'eseguire il calcolo otteniamo

$$0.666 + 0.333 + 1.000 = 1.999.$$

Calcoli numerici possono essere effettuati tramite una semplice calcolatrice portatile oppure su calcolatori più potenti tramite linguaggi di programmazione quali *Fortran*, *Basic*, *Pascal*, *C*, *C++*, *Perl* e *Java*.

Molti di questi linguaggi dispongono di librerie numeriche opportune che consentono di eseguire in modo semplificato operazioni molto complesse.

Calcoli simbolici possono essere effettuati da programmi specifici quali *Maple*, *Mathematica*, *MATLAB*, *Scilab*, *Octave*, *Derive*. In particolare MATLAB può eseguire calcoli simbolici attraverso il *Symbolic Mathematics toolbox* Symbolic Mathematics che utilizza le routine di calcolo simbolico di Maple.

Come abbiamo visto il calcolo simbolico non comporta le approssimazioni introdotte dal calcolo numerico. Il prezzo da pagare per ottenere questo ricade sulla velocità d'esecuzione e sulla necessità di particolari strutture dati. In generale i risultati numerici possono essere ottenuti molto più velocemente tramite un calcolo numerico invece di una valutazione numerica di un calcolo simbolico.

Metodi numerici e algoritmi

La differenza principale tra un metodo numerico ed un algoritmo numerico è data dal fatto che un metodo numerico consiste in una descrizione matematica dei calcoli che devono essere eseguiti, mentre un algoritmo numerico è una sequenza precisa di azioni che consentono di ottenere il risultato desiderato.

Una conseguenza immediata di questa distinzione si ha nel fatto che lo stesso metodo numerico può essere implementato tramite differenti algoritmi.

Esempio 2 (Diversi algoritmi per prepararsi un caffè)
Consideriamo il semplice obiettivo di preparare una tazza di caffè. Innanzi tutto abbiamo metodi diversi per ottenerlo: una macchina espresso, una caffettiera tradizionale, un percolatore, il caffè solubile, ed altri ancora. Una volta che si è scelto un metodo, ad esempio la macchina espresso, esistono diversi algoritmi per fare il caffè.

Algoritmo 1

1. *Macinare del caffè fresco.*

2. *Misurare con cura la dose di caffè da mettere nel filtro.*

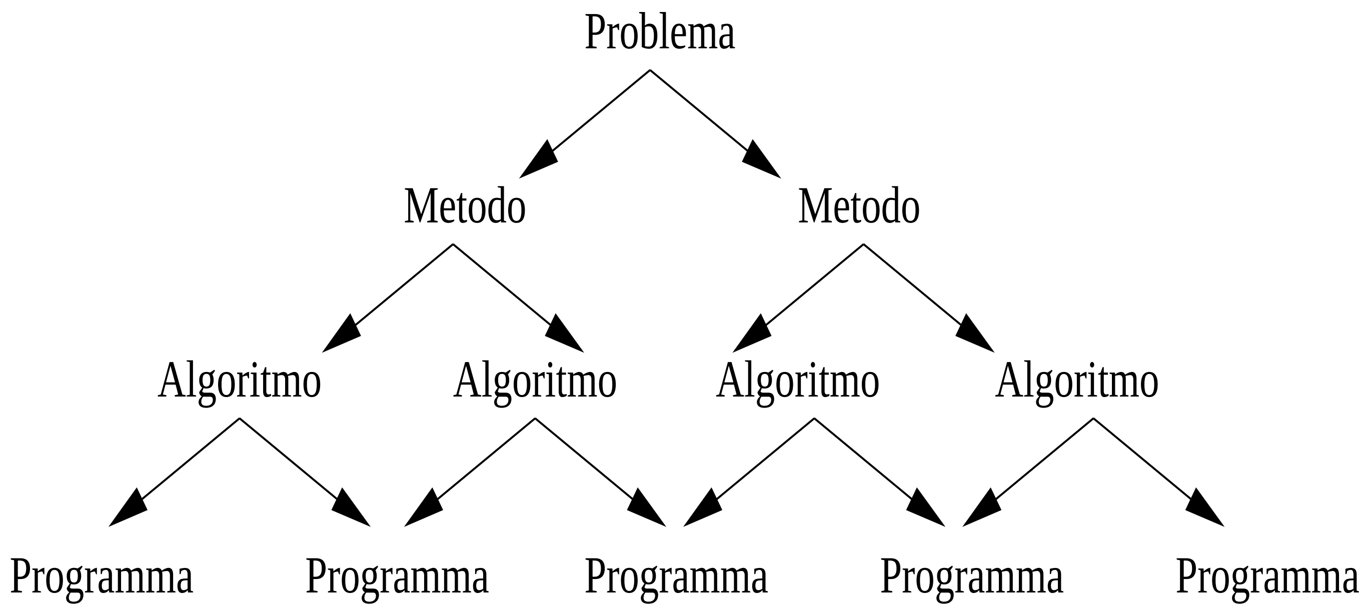
3. *Se necessario mettere acqua naturale in bottiglia nella macchina.*
4. *Prendere una tazzina pulita dalla dispensa.*
5. *Accendere la macchina e premere il pulsante per il caffè solo dopo averla fatta riscaldare diversi minuti.*
6. *Aggiungere un po' di latte dopo averlo riscaldato al vapore.*

Algoritmo 2

1. *Prendere del caffè già macinato.*
2. *Mettere un po' di caffè nel filtro.*
3. *Se necessario mettere un po' di acqua del rubinetto nella macchina.*
4. *Risciacquare la vecchia tazzina che si trovava nel lavandino.*
5. *Accendere la macchina e premere il pulsante per il caffè dopo pochi secondi.*
6. *Aggiungere un po' di latte preso dal frigorifero.*

La scelta di un algoritmo invece dell'altro può dipendere da diversi fattori, ad esempio dalla precisione e dall'accuratezza dei gusti personali, dal tempo a disposizione e dal tipo di applicazione.

Tipicamente le caratteristiche principali che caratterizzano un algoritmo sono la non ambiguità, ossia le istruzioni devono essere univocamente interpretabili, la generalità ossia la possibilità di essere applicato ad una classe di problemi più ampi e la finitezza, ossia il risultato deve potere essere ottenuto con un numero finito di passi.
Una volta che si è scelto un dato algoritmo, la fase successiva consiste nell'implementarlo tramite un programma per il calcolatore. Un programma è la traduzione di un algoritmo in una sequenza di istruzioni in un determinato linguaggio per calcolatore. Quindi come un dato metodo può essere realizzato con diversi algoritmi, un dato algoritmo può essere implementato in diversi linguaggi.



Le strade per realizzare un programma che risolve un problema.

Analisi numerica e calcolo scientifico

Molti dei metodi numerici che vedremo sono stati sviluppati molto prima dell'attuale "era informatica". Infatti per molti problemi pratici in ambito scientifico per i quali non si poteva trovare una soluzione analitica esatta o esplicita si sono sviluppate procedure di calcolo approssimato di soluzioni.

Ovviamente l'importanza di minimizzare il numero di operazioni massimizzando l'accuratezza in assenza di calcolatori elettronici era essenziale ed ha condotto allo sviluppo dell'analisi numerica come settore della matematica.

Nonostante il termine calcolo scientifico sia spesso considerato sinonimo di analisi numerica è opportuno fare una distinzione.

Lo scopo principale dell'analisi numerica è lo studio del comportamento dei metodi numerici da un punto di vista matematico e in maniera minore l'applicazione di questi metodi ad un problema specifico. Di questo secondo aspetto si occupa invece principalmente il calcolo scientifico.

Cominciamo a usare MATLAB

L'ambiente MATLAB è di uso estremamente semplice ed è possibile avere una panoramica introduttiva utilizzando i comandi `intro` e `demo`. Per ulteriori informazioni e approfondimenti rimandiamo alla guida all'uso di MATLAB ed al sito Web `http://www.mathworks.com`.

MATLAB come semplice calcolatrice elettronica

L'aspetto principale di MATLAB è quello di forzare l'utente ad avere un approccio di tipo vettoriale a molti problemi. Fondamentale è quindi familiarizzare fin dal principio con le operazioni vettoriali così come naturalmente si ha dimestichezza con le operazioni scalari.

MATLAB può essere utilizzato in modo diretto per calcolare semplici espressioni matematiche

```
>> 5 - 2 + 3
ans =
   6
```

La risposta è del tipo `ans=...`, dove `ans` è una variabile generata automaticamente da MATLAB quando un'espressione non è assegnata ad una variabile definita dall'utente. In questo secondo caso possiamo ad esempio scrivere

```
>> a = 5 - 2
a =
     3
>> b = 3
b =
     3
>> c = a+b
c =
     6
```

In generale MATLAB visualizza il risultato di un operazione a meno che questa non termini con un punto e virgola. Quindi

```
>> a = 5 - 2;
>> b = 3;
>> c = a+b;
```

esegue le stesse operazioni assegnando $a = 3$, $b = 3$ e $c = 6$ ma senza visualizzazione dei risultati. Per visualizzare il contenuto di una variabile è sufficiente scriverne il nome. Nel caso di più variabili i nomi vanno separati con una virgola

```
>> a,b,c
a =
     3
b =
     3
c =
     6
```

Risulta evidente dai precedenti esempi che le variabili sono create automaticamente da MATLAB al momento del loro uso. Se una variabile non esiste è creata non appena compare nel termine di sinistra di una uguaglianza. I nomi di variabili possono essere lunghi un massimo di 31 caratteri con la distinzione tra lettere maiuscole e minuscole (ad esempio le variabili `a1` e `A1` sono distinte). 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* "_".

Variabile	Significato
`ans`	valore ultima operazione eseguita non assegnata ad una variabile
`i,j`	unità immaginaria
`pi`	π, 3.14159265...
`eps`	precisione di macchina
`realmax`	massimo numero macchina positivo
`realmin`	minimo numero macchina positivo
`Inf`	∞, ossia un numero maggiore di `realmax`
`NaN`	Not a Number, tipicamente il risultato di un'espressione `0/0`

Alcune variabili predefinite in MATLAB

Alcune delle variabili predefinite in MATLAB sono riportate in Tabella . Nonostante sia ammesso assegnare valori diversi a queste variabili, in generale è buona regola evitare di farlo, fatta eccezione per le variabili `i` e `j` spesso usate come indici interi. Il significato delle variabili `eps`, `realmax`, `realmin`, `Inf` e `NaN` verrà discusso in dettaglio in seguito.

Operazione	Significato
+	addizione
-	sottrazione
*	moltiplicazione
/	divisione
^	elevamento a potenza
.*	moltiplicazione termine a termine per vettori
./	divisione termine a termine per vettori
.^	elevamento a potenza termine a termine per vettori

Principali operazioni in MATLAB

Le principali operazioni possibili sono indicate in Tabella , vedremo in seguito il significato delle operazioni precedute da un punto, o puntuali. L'utilizzo di operazioni su numeri complessi è ammesso, potremo quindi scrivere espressioni del tipo

```
>> a=3+2i;
>> b=3.6+2.4*i;
>> a+b
ans =
   6.6000 + 4.4000i
>> a*b
ans =
   6.0000 +14.4000i
```

L'unità immaginaria è rappresentata dalle variabili `i` e `j` ed è tale che $i^2 = -1$, $j^2 = -1$. Le forme `a=3+2i`, `a=3+2*i`, `a=3+2j`, `a=3+2*j` sono accettate e sono equivalenti.

Oltre alle operazioni di base, molte delle funzioni comunemente presenti su una calcolatrice scientifica sono presenti in MATLAB. Una funzione necessita di alcuni parametri in ingresso, elencati tra parentesi tonde, e solitamente restituisce un risultato che può essere assegnato ad una variabile. Per esempio l'espressione

```
>> y=cos(pi/4)
y =
    0.7071
```

utilizza la funzione coseno con argomento $\pi/4$ e ne assegna il risultato alla variabile y. Al contrario un comando MATLAB è utilizzato con uno spazio tra il nome del comando stesso e il suo argomento, ad esempio

```
>> help cos
COS    Cosine.
        COS(X) is the cosine of the elements of X
```

visualizza una descrizione rapida della funzione coseno in MATLAB.

Il comando `help` consente di avere una descrizione immediata di una funzione, un comando oppure un operazione MATLAB, semplicemente passando il nome della funzione, del comando oppure dell'operazione come argomento. La descrizione è in lingua inglese, in quanto non esistono versioni MATLAB in lingua italiana.

Si noti che il risultato dell'operazione cos($\pi/4$) è visualizzato utilizzando quattro cifre decimali. Questa è l'impostazione standard di MATLAB. È possibile modificarla tramite il comando `format`. Ad esempio la sequenza di istruzioni

```
>> format long
>> cos(pi/4)
ans =
   0.70710678118655
>> format short
```

abilita prima il formato a 14 cifre decimali, calcola il risultato, poi riattiva il formato standard a 4 cifre decimali. È importante evidenziare che la modifica della visualizzazione di un risultato tramite `format` non ha nulla a che vedere con l'effettiva precisione con cui MATLAB effettua il calcolo.

Funzione	**Significato**
`sin`	seno
`cos`	coseno
`asin`	arcoseno
`acos`	arcocoseno
`tan`	tangente
`atan`	arcotangente
`exp`	esponenziale
`log`	logaritmo naturale
`sqrt`	radice quadrata
`abs`	valore assoluto
`sign`	la funzione segno

Alcune funzioni predefinite in MATLAB

Alcune delle principali funzioni elementari predefinite in MATLAB sono riportate nella precedente Tabella, per una lista più ampia si provi il comando `help elfun`.

Un altro comando di particolare utilità è il comando `lookfor` che consente di identificare le funzioni relative ad un particolare argomento. Essenzialmente il comando identifica tutte le funzioni all'interno della cui descrizione compare l'argomento passato al comando `lookfor`. Ad esempio

```
>> lookfor logarithm
LOGSPACE Logarithmically spaced vector.
LOG      Natural logarithm.
LOG10    Common (base 10) logarithm.
LOG2     Base 2 logarithm and dissect floating point number.
BETALN   Logarithm of beta function.
GAMMALN  Logarithm of gamma function.
LOGM     Matrix logarithm.
```

restituisce una lista di funzioni (in maiuscolo) con una breve descrizione delle stesse.

Si provino i seguenti comandi

```
>> help ans
>> help help
>> help format
>> help lookfor
```

per avere un sommario di alcune delle caratteristiche di MATLAB viste fino ad ora.

Vettori e Matrici

Negli esempi precedenti le variabili utilizzate apparentemente erano quantità scalari, ossia semplici valori numerici. In realtà ogni variabile è per MATLAB una struttura di tipo vettoriale o *array*.

Un array è un insieme di valori ordinati, secondo uno o più indici, cui ci si riferisce con un singolo nome di variabile. Dalla versione 5 di MATLAB, e successive, è possibile utilizzare array con più di due indici. Tipicamente un array ad un indice è detto vettore, ed un array a due indici è chiamato matrice.

Esempio 3 (array a uno e due indici) La sequenza di numeri interi $\{2, -3, 7, 2, 11, -8\}$ può essere rappresentata in forma di array ad un solo indice x nelle forme

$$\overbrace{\begin{array}{|c|c|c|c|c|c|}\hline 2 & -3 & 7 & 2 & 11 & -8 \\ \hline\end{array}}^{x} \qquad x\left\{\begin{array}{|c|}\hline 2 \\ \hline -3 \\ \hline 7 \\ \hline 2 \\ \hline 11 \\ \hline -8 \\ \hline\end{array}\right.$$

L'array di sinistra è detto vettore riga, quello di destra vettore colonna. Entrambi hanno lunghezza 6. Tramite l'uso degli indici, la cui numerazione

va da 1 a 6, da sinistra a destra nei vettori riga e dall'alto verso il basso nei vettori colonna, possiamo accedere ad un dato valore all'interno dell'array. Ad esempio a $x(2)$ corrisponderà il valore -3, così come a $x(5)$ il valore 11.
La stessa sequenza di numeri può anche essere memorizzata come array a due indici A nelle forme

$$\overbrace{\begin{array}{|c|c|c|}\hline 2 & -3 & 7 \\ \hline 2 & 11 & -8 \\ \hline \end{array}}^{A} \qquad \overbrace{\begin{array}{|c|c|}\hline 2 & 2 \\ \hline -3 & 11 \\ \hline 7 & -8 \\ \hline \end{array}}^{A}$$

La matrice di sinistra è detta di tipo 2×3, dove 2 indica il numero di righe e 3 il numero di colonne. Conseguentemente quella di destra è detta di tipo 3×2. Il numero di righe per il numero di colonne fornisce il numero di elementi della matrice, ossia 6.

In questo caso avremo due indici, il primo riferito alle righe dell'array, il secondo riferito alle colonne. Entrambi iniziano da uno, la numerazione del primo indice va dall'alto al basso (righe) e quella del secondo da sinistra a destra (colonne). Per la matrice di sinistra $A(1,2)$ corrisponde a -3 ed $A(2,2)$ a 11, mentre per quella di destra $A(2,1)$ corrisponde a -3 e $A(2,2)$ a 11.

In particolare si noti che il vettore x del precedente esempio nella forma riga o colonna è rispettivamente una matrice di tipo 1×6 o 6×1. Questo è esattamente il modo in cui MATLAB definisce i vettori. Non solo, per MATLAB anche le quantità scalari sono semplici matrici di tipo 1×1.

Per memorizzare il precedente vettore x nella forma riga in MATLAB possiamo utilizzare la seguente espressione

```
>> x=[2 -3 7 2 11 -8]
x =
     2    -3     7     2    11    -8
```

Le parentesi quadre delimitano gli elementi del vettore, mentre gli spazi le singole componenti del vettore riga. Se invece considerassimo

```
>> x=[2; -3; 7; 2; 11; -8]
x =
     2
    -3
     7
     2
    11
    -8
```

si otterrebbe un vettore colonna. Le componenti in questo caso sono delimitate da un punto e virgola. Per visualizzare il valore di una componente del vettore basta scrivere

```
>> x(2)
ans =
    -3
```

Se cambiamo il valore di una componente del vettore, dobbiamo ricordarci di usare il punto e virgola altrimenti MATLAB visualizzerà l'intero vettore e la cosa può risultare noiosa nel caso di vettori molto lunghi

```
>> x(2)=-7
x =
     2
    -7
     7
     2
    11
    -8
```

Per passare da vettori riga a vettori colonna si utilizza il simbolo di apostrofo. Quindi

```
>> x=[2 -3 7 2 11 -8]'
x =
     2
    -3
     7
     2
    11
```

```
   -8
```

Dal punto di vista dell'algebra lineare l'operazione che si effettua è esattamente la trasposizione.

MATLAB fornisce la funzione **length** che consente di determinare la lunghezza di un vettore

```
>> length(x)
ans =
     6
```

Se ora vogliamo inserire la matrice 2×3 vista nell'esempio abbiamo

```
>> A=[2 -3 7; 2 11 -8]
A =
     2    -3     7
     2    11    -8
```

dove anche in questo caso gli spazi separano gli elementi per colonna e il punto e virgola separa le righe. La stessa matrice può anche essere inserita utilizzando il tasto Invio per separare le righe

```
>> A=[2 -3 7
      2 11 -8]
A =
     2    -3     7
     2    11    -8
```

Potremo accedere agli elementi della matrice in maniera naturale utilizzando i corrispondenti indici

```
>> A=(1,2)
ans=
    -3
```

Se vogliamo quindi cambiare l'elemento `A(1,2)` basta scrivere

```
>> A=(1,2)=-7
A =
     2    -7     7
     2    11    -8
```

prestando attenzione che senza punto e virgola finale viene visualizzata l'intera matrice. La funzione MATLAB che ci consente di determinare le dimensioni di una matrice è `size`

```
>> size(A)
ans =
     2     3
```

che restituisce un vettore riga di due elementi interi, il primo indica il numero di righe ed il secondo il numero di colonne. Possiamo utilizzare la funzione `size` per avere conferma del modo in cui MATLAB considera quantità scalari e vettori

```
>> a=3;
```

```
>> size(a)
ans =
     1     1
>> b=3+3i;
>> size(b)
ans =
     1     1
>>  x=[2 -3 7 2 11 -8];
>> size(x)
ans =
     1     6
>> size(x')
ans =
     6     1
```

Se vogliamo controllare le variabili che attualmente MATLAB ha memorizzato possiamo usare i comandi `who` e `whos`

```
>> who
```

```
Your variables are:

A          a          b          x

>> whos
  Name         Size            Bytes  Class

  A            2x3                48  double array
  a            1x1                 8  double array
  b            1x1                16  double array (complex)
  x            1x6                48  double array

Grand total is 14 elements using 120 bytes
```

Oltre alle informazioni sulla dimensione e la lunghezza delle variabili MATLAB vengono fornite alcune indicazioni utili sullo spazio occupato in memoria dalle stesse variabili. Come si può notare una variabile non complessa come a occupa 8 bytes in memoria, mentre una variabile complessa come

b occupa 16 bytes. Conseguentemente x e A, entrambi costituiti da 6 elementi non complessi, occupano $6\times 8 = 48$ bytes in memoria. Per rimuovere le variabili dalla memoria lavoro di MATLAB si utilizza l'istruzione `clear`

```
>> clear
```

Provare **`help clear`** per avere una descrizione approfondita dell'uso del comando per rimuovere singole variabili e funzioni.

La notazione due punti

Per la costruzione di vettori e matrici MATLAB ha diverse funzioni predefinite. Un operatore di fondamentale importanza per costruire vettori equispaziati e per operare con indici è la notazione due punti. La sintassi di base dell'operatore è

Vettore=Inizio:Passo:Fine,

dove *Vettore* è un vettore riga, *Inizio* e *Fine* indicando il valore iniziale e finale del vettore e *Passo* è un parametro opzionale che indica l'incremento relativo o la spaziatura tra gli elementi (se omesso *Passo=1*).

Si considerino i seguenti esempi

```
>> x=1:10
x =
     1      2      3      4      5      6      7      8      9     10
>> x=10:-1:1
x=
     10      9      8      7      6      5      4      3      2      1
```

Per costruire vettori colonna basterà racchiudere l'espressione tra parentesi tonde ed utilizzare l'operatore apostrofo

```
>> x=(2:5)'
x =
```

```
     2
     3
     4
     5
```

Se il valore iniziale è maggiore di quello finale ed il passo è positivo viene creato un vettore nullo

```
>> x=10:1
x =
     []
```

lo stesso risultato si ha chiaramente anche se il valore iniziale è minore di quello finale e l'incremento è negativo. Il vettore nullo ha lunghezza zero, dimensione 0×0 ed è denotato da due parentesi quadre senza nulla all'interno.

La notazione due punti, funziona anche con valori non interi, ad esempio

```
>> x=0:0.1:0.5
x =
     0    0.1000    0.2000    0.3000    0.4000    0.5000
```

Nel caso in cui il passo non sia intero può risultare difficile l'uso dell'operatore due punti. Ad esempio se vogliamo creare un vettore con un numero prefissato di punti equispaziati all'interno di un dato intervallo. In questo caso è preferibile utilizzare il comando `linspace` la cui sintassi è

`linspace(`*Inizio*, *Fine*, *Numero di Punti*`)`

dove il parametro *Numero di Punti* è opzionale, e se omesso viene preso uguale a 100.

Ad esempio

```
>> a = 0; b=1; n=5;
>> x = linspace(a,b,n)
```

```
x =
     0    0.2500    0.5000    0.7500    1.0000
```

restituisce un vettore riga x di lunghezza n con la proprietà che l'elemento di indice i vale

$$x(i) = a + (i-1) * (b-a)/(n-1).$$

In particolare le due istruzioni `x=linspace(a,b)` ed `x=linspace(a,b,100)` sono equivalenti. Anche in questo caso vettori colonna possono essere costruiti tramite l'operatore apostrofo di trasposizione

```
>> a = 0; b=1; n=5;
>> x = linspace(a,b,n)';
>> x =

         0
    0.2500
    0.5000
    0.7500
```

```
    1.0000
```

Un uso particolarmente efficace della notazione due punti si ha nella gestione di indici di vettori e matrici. In particolare tale notazione consente di identificare facilmente un intera riga o colonna di una matrice

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> A(:,1)
ans =
     1
     4
     7
>> A(2,:)
ans =
     4     5     6
```

Alternativamente è possibile specificare un intervallo di indici ed estrarre così parti di vettori (o matrici)

```
>> x=0:0.1:0.5;
>> x(2:4)
ans =
      0.1000    0.2000    0.3000
```

o parti di righe e colonne in matrici e sottomatrici

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> A(2,2:3)
ans =
      5     6
>> A(1:2,2:3)
ans =

     2     3
     5     6
```

La notazione due punti può essere usata anche per assegnare in modo rapido nuovi valori a righe e colonne di matrici

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

Un operazione che può risultare utile è quella di cancellare alcuni elementi in un vettore cambiandone allo stesso tempo la dimensione

```
>> x = 1:10;
>> x(1:3)=[]
x =
     4     5     6     7     8     9    10
```

la stessa tecnica può essere usata nel caso di matrici per rimuovere intere righe o colonne

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

```
>> A(:,1)=[]
A =
     2     3
     5     6
     8     9
```

Funzione	Significato
linspace	vettore riga di elementi equispaziati
logspace	vettore riga di elementi equispaziati in scala logaritmica
zeros	matrice contenente solo elementi uguali a zero
ones	matrice contenente solo elementi uguali a uno
rand	matrice contenente numeri casuali
eye	matrice identità
diag	matrice diagonale
magic	matrice a valori interi con somme uguali per righe e colonne

Alcune funzioni predefinite per la costruzione di vettori e matrici in MATLAB

Vettorizzazione e operazioni puntuali

Molte funzioni predefinite in MATLAB accettano come argomenti array a più indici. Questa caratteristica di MATLAB è molto importante in quanto consente di scrivere in forma molto chiara e compatta sequenze di istruzioni eliminando in molti casi l'uso di strutture e cicli che agiscono a livello scalare.

Esempio **4 (Tabella di valori di seno e coseno)** Per costruire una semplice tabella di valori delle funzioni seno e coseno nell'intervallo $[0, \pi]$ possiamo procedere nel seguente modo

```
>> n = 5;
>> x = linspace(0,pi,n);
>> c = cos(x);
>> s = sin(x);
>> [x' c' s']
ans =
```

```
         0    1.0000         0
    0.7854    0.7071    0.7071
    1.5708    0.0000    1.0000
    2.3562   -0.7071    0.7071
    3.1416   -1.0000    0.0000
```

L'istruzione `c=cos(x)` applicata ad un vettore x restituisce un vettore c di uguali dimensioni e tipo con la proprietà che l'elemento di indice i è $c(i) = \cos(x(i))$. Risulta quindi equivalente all'istruzione `c = [cos(x(1)) cos(x(2)) cos(x(3)) cos(x(4)) cos(x(5))]`. Analogo discorso per l'istruzione vettoriale `s=sin(x)`. Infine l'istruzione `[x' c' s']` crea una matrice le cui colonne sono i vettori trasposti `x'`, `c'` e `s'`.

Consideriamo alcune semplici operazioni che possiamo utilizzare su array a più indici e che agiscono simultaneamente su tutte le componenti dell'array

```
>> x = 1:5;
>> y = [50 10 30 40 20];
>> 2*x
```

```
ans =
     2     4     6     8    10
>> x+y
ans =
    51    12    33    44    25
>> y-x
ans =
    49     8    27    36    15
>> x.*y
ans =
    50    20    90   160   100
>> y./x
ans =
    50     5    10    10     4
>> y.^x
ans =
          50         100       27000     2560000     3200000
```

Le prime tre operazioni seguono le regole dell'algebra lineare numerica e sono la moltiplicazione di un vettore per uno scalare, la somma e la differenza tra vettori.

Il comando `2*x` moltiplica ogni componente di x per la quantità scalare 2. Il vettore risultato ha esattamente la stessa lunghezza del vettore x.

Il comando `x+y` somma le rispettive componenti dei vettori x e y e può essere utilizzato solo su vettori che hanno la stessa dimensione.

Il comando `y-x` sottrae dal vettore y le corrispondenti componenti di x. Entrambi i comandi restituiscono vettori aventi la stessa dimensione dei vettori argomento.

Al contrario le ultime tre operazioni, ossia la moltiplicazione puntuale, la divisione puntuale e l'elevamento a potenza puntuale sono tipiche dell'ambiente MATLAB.

Non hanno un corrispondente dal punto di vista dell'algebra lineare in quanto agiscono su vettori e matrici intesi come strutture di dati più che entità matematiche.

L'istruzione `x.*y` utilizza la moltiplicazione puntuale tra vettori e fornisce un vettore con la proprietà che ogni sua componente è uguale al prodotto delle corrispondenti componenti dei vettori x e y.

La divisione puntuale `y./x` restituendo un vettore le cui componenti sono il risultato della divisione delle corrispondenti componenti di y per quelle di x. Infine il comando `y.^x` eleva ogni componente di y alla corrispondente componente di x.

Le stesse operazioni possono essere applicate nel caso di vettori colonna o più in generale nel caso di matrici. La cosa essenziale è che gli operandi siano dello stesso tipo ed abbiano le stesse dimensioni.

Uniche eccezioni a questa regola sono date dal caso in cui le precedenti operazioni vengano applicate tra un vettore ed una costante. In tal caso MATLAB considererà la costante come un vettore di pari dimensioni avente tutte componenti costanti. Ad esempio

```
>> x = 1:5;
>> x+1
ans =
     2     3     4     5     6
>> 1-x
ans =
      0    -1    -2    -3    -4
>> 2./x
ans =
   2.0000    1.0000    0.6667    0.5000    0.4000
>> x.^2
      1     4     9    16    25
```

Esempio **5 (Vettorizzazione di una funzione)** Consideriamo ora il problema di visualizzare una tabella di valori di una funzione non predefinita in

MATLAB, ad esempio

$$f(x) = \frac{15120 - 6900x^2 + 313x^4}{15120 + 660x^2 + 13x^4}$$

sull'intervallo $[0, \pi]$. Tale funzione è una particolare approssimazione della funzione $\cos(x)$. Per costruire una tabella di 5 valori possiamo utilizzare la sequenza di istruzioni

```
>> x = linspace(0,pi,5);
>> y = (15120-6900*x.^2+313*x.^4)./(15120+660.*x.^2+13.*x.^4);
>> [x' y']
ans =
         0    1.0000
    0.7854    0.7071
    1.5708    0.0000
    2.3562   -0.7057
    3.1416   -0.9821
```

Utilizzando le operazioni vettoriali i valori della funzione in corrispondenza della successione di punti definita dal vettore x sono creati con una singola istruzione vettoriale ed assegnati al vettore y.

Concludiamo questa sezione osservando come occasionalmente si possa avere la necessità di convertire dati memorizzati in un certo tipo di array in un array di tipo differente ma con lo stesso numero di elementi. A questo scopo si può utilizzare la funzione `reshape`

```
>> A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
A =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
>> B = reshape(A,2,6)
B =
     1     7     2     8     3     9
     4    10     5    11     6    12
```

La sintassi è quindi

reshape(*Matrice*, *Righe*, *Colonne*)

dove il parametri *Righe* e *Colonne* si riferiscono alla nuova matrice prodotta come risultato.

Esiste una forma molto compatta di convertire matrici in vettori colonna

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

```
     3
     6
     9
```

Chiaramente qualora si abbia la necessità di un vettore riga basterà considerare il trasposto della precedente espressione. Si noti che se applicata ad un vettore x l'espressione `x(:)` restituisce sempre un vettore colonna, indipendentemente dal tipo originale del vettore, a differenza dell'operatore apostrofo `x'` che invece cambia vettori colonna in vettori riga e viceversa.

Grafica in MATLAB

MATLAB possiede numerose capacità grafiche che consentono di rappresentare dati memorizzati in vettori e matrici. Grafici bidimensionali vengono utilizzati per visualizzare la variazione di una variabile rispetto ad un'altra, ad esempio il grafico di una funzione $y = f(x)$.

Esempio **6 (Un semplice grafico di funzione)** Supponiamo di volere visualizzare in forma grafica una successione di valori della funzione $\cos(x)$

sull'intervallo $[0, 2\pi]$. Per prima cosa dovremo creare due vettori x e y contenenti rispettivamente la successione di valori nell'intervallo ed i corrispondenti valori della funzione. Possiamo quindi scrivere

```
>> n = 21;
>> x = linspace(0,2*pi,n);
>> y = cos(x);
```

dove in questo caso abbiamo scelto 21 punti equispaziati. Per realizzare il grafico basta aggiungere il comando

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

In questo modo otteniamo una rappresentazione grafica della tabella di valori ottenuta semplicemente raccordando con segmenti di retta nel piano xy i vertici $(x(i), y(i))$ in modo ordinato al variare di i da 1 a n. La scelta della scala di visualizzazione è automatica. Se vogliamo evidenziare i vertici della poligonale così costruita possiamo utilizzare l'istruzione

```
>> plot(x,y,'-o')
```

che disegna una linea con un circoletto in corrispondenza dei vertici.

La sintassi di base del comando `plot` è quindi

`plot(`*Vettore1*`,` *Vettore2*`,` *Opzioni*`)`,

dove *Vettore1* e *Vettore2* sono i vettori di dati (ascisse e ordinate dei punti) e *Opzioni* è una stringa opzionale che definisce il tipo di colore, di simbolo e di linea usato nel grafico.

Si noti che se il comando viene usato nella semplice forma `plot(Vettore)`, MATLAB realizza il grafico del vettore rispetto ai propri indici. Possiamo inoltre commentare il grafico aggiungendo le istruzioni

```
>> title('Grafico della funzione cos(x)')
>> xlabel('x')
```

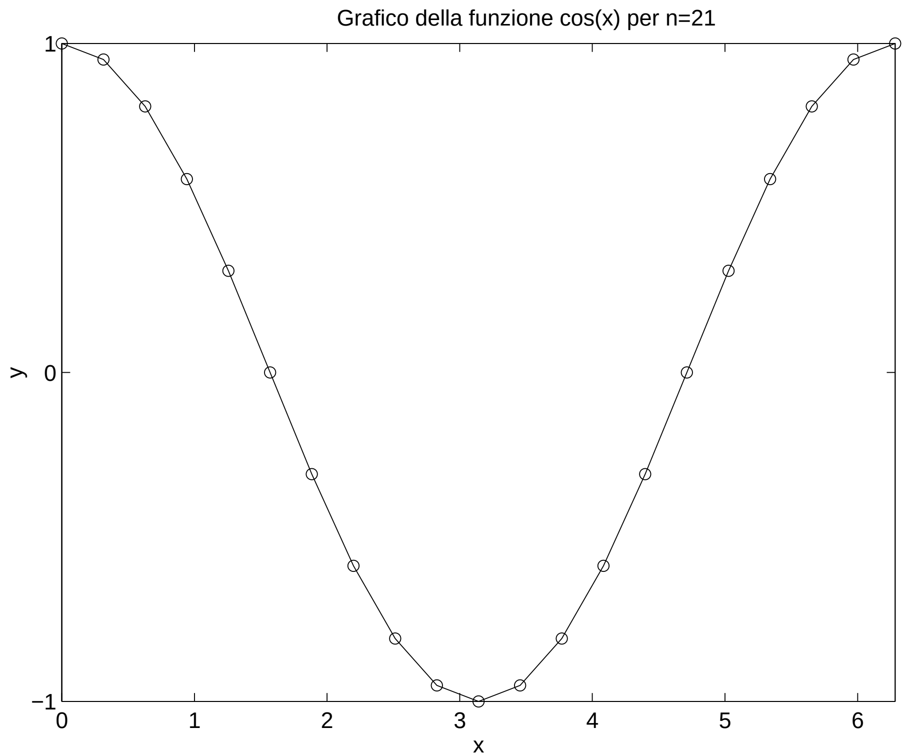
```
>> ylabel('y')
```

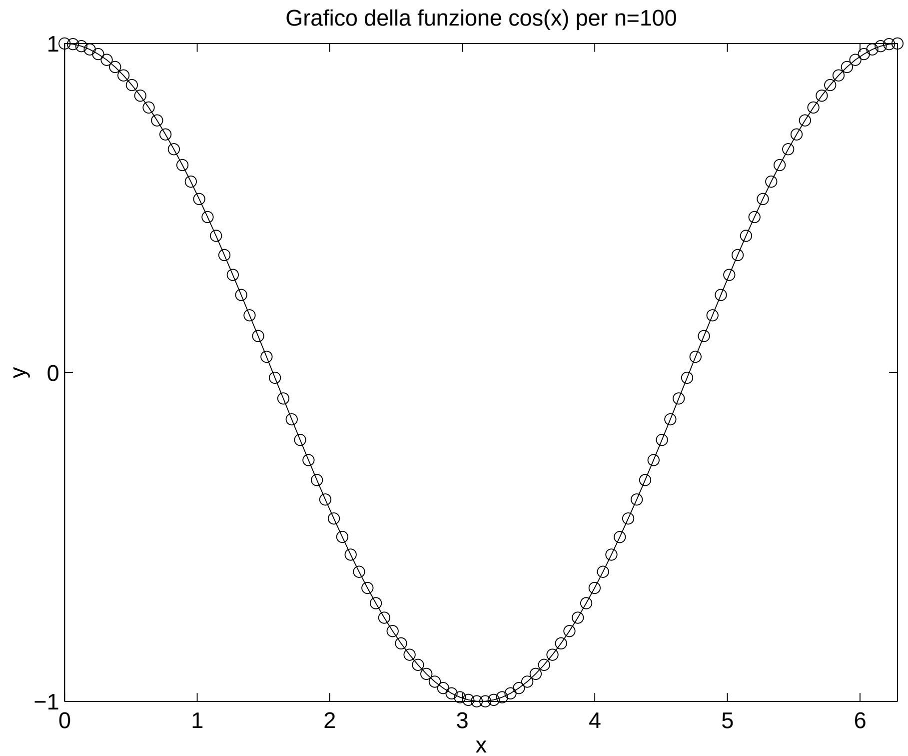
che consentono di inserire informazioni sul titolo del grafico e sugli assi x e y.

Colore		Simbolo		Linea	
y	giallo	.	punto	-	linea continua
m	magenta	o	circoletto	:	linea punteggiata
c	ciano	x	per	-.	linea punto
r	rosso	+	più	--	linea tratteggiata
g	verde	*	asterisco		
b	blu	s	quadratino		
w	bianco	d	diamante		
k	nero	v	triangolo		

Alcune opzioni del comando `plot` in MATLAB

Se vogliamo produrre un risultato visivamente privo di “spigoli” sarà sufficiente aumentare il numero di punti n, in modo tale che i segmenti di raccordo siano così piccoli da rendere l’impressione di una curva continua.





Rappresentazione grafica dei valori della funzione $\cos(x)$ in $[0, 2\pi]$.

Una caratteristica importante è quella di potere riscalare i grafici tramite la funzione `axis`. Ad esempio possiamo scrivere

```
>> x = linspace(0,2*pi);
>> y = cos(x);
>> plot(x,y);
>> a = axis
a =
     0     7    -1     1
>> axis([a(1) pi a(3) a(4)])
```

per ottenere il grafico soltanto sull'intervallo $[0, \pi]$. La funzione `axis([xmin xmax ymin ymax])` impone che l'intervallo di valori di x sia compreso tra $xmin$ e $xmax$ e quello di y tra $ymin$ e $ymax$. Il comando `axis` usato da solo riporta alla scala automatica e ritorna come risultato un vettore contenete i valori attuali della scala.

Esempio **7 (Grafici sovrapposti di seno e coseno)** Ci poniamo il problema di realizzare i grafici delle funzioni seno e coseno sull'intervallo $[0, 2\pi]$ nella stessa finestra grafica. Per fare questo abbiamo due possibilità. Innanzitutto dobbiamo costruire i vettori di valori per le due funzioni

```
>> x = linspace(0,2*pi);
>> y1 = cos(x);
>> y2 = sin(x);
```

Per disegnarne i grafici sovrapposti possiamo scrivere

```
>> plot(x,y1,'-')
>> hold on
>> plot(x,y2,'--')
>> hold off
```

Il comando `hold on` fa sovrapporre tutti i grafici successivi nella finestra grafica. Il comando `hold off` ritorna all'impostazione originale.

Lo stesso risultato può essere ottenuto tramite la singola istruzione

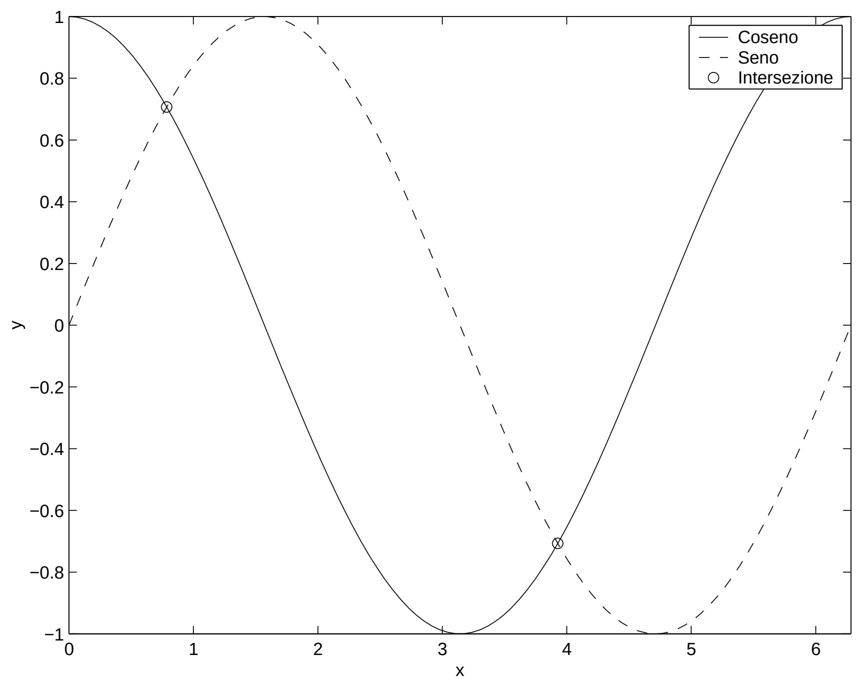
```
>> plot(x,y1,x,y2)
```

che utilizza in modo automatico linee di tipo differente per i diversi grafici.

Se vogliamo realizzare i due grafici evidenziando ad esempio i punti di intersezione delle due curve possiamo scrivere

```
>> plot(x,y1,'-',x,y2,'--',[1 5]*pi/4,[1 -1]/sqrt(2),'o')
>> legend('Coseno', 'Seno, 'Intersezione')
```

dove abbiamo inserito la funzione `legend` che inserisce una legenda nel grafico che identifica le curve disegnate con linee e simboli differenti.



In Tabella riportiamo le principali funzioni che consentono di aggiungere commenti ad un grafico. Ulteriori funzioni possono essere sperimentate utilizzando il comando `help graph2d`.

Funzione	**Significato**
`axis`	prescrive i valori minimi e massimi sugli assi x e y
`title`	inserisce un titolo nel grafico
`xlabel`	inserisce un nome per l'asse x
`ylabel`	inserisce un nome per l'asse y
`zlabel`	inserisce un nome per l'asse z
`grid`	inserisce una griglia sugli assi x e y
`legend`	inserisce una legenda per identificare i diversi grafici
`text`	inserisce una stringa di testo in una posizione specificata

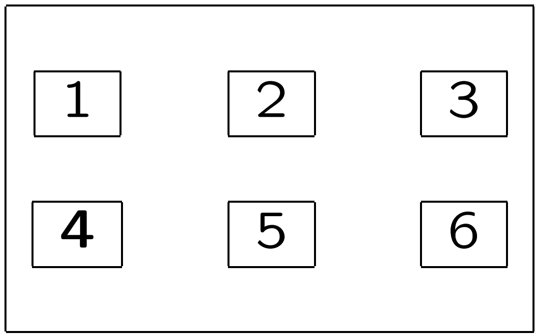
Alcune funzioni per commentare grafici in MATLAB

Sottografici e grafici multidimensionali

Spesso ci si pone il problema di disegnare diversi grafici separati in una stessa finestra. L'obiettivo può essere raggiunto facilmente utilizzando la funzione `subplot` la cui sintassi è

`subplot(`*Righe*, *Colonne*, *Sottofinestra*`)`

dove *Righe* e *Colonne* definiscono la struttura della matrice di sottofinestre grafiche all'interno della finestra grafica principale e *Sottofinestra* indica il numero della sottofinestra grafica attiva. Consideriamo a titolo di esempio l'istruzione `subplot(2,3,4)`. Tale istruzione suddivide la finestra in una matrice 2×3 di sottofinestre ed attiva la quarta sottofinestra grafica. Le sottofinestre sono numerate disposte come segue

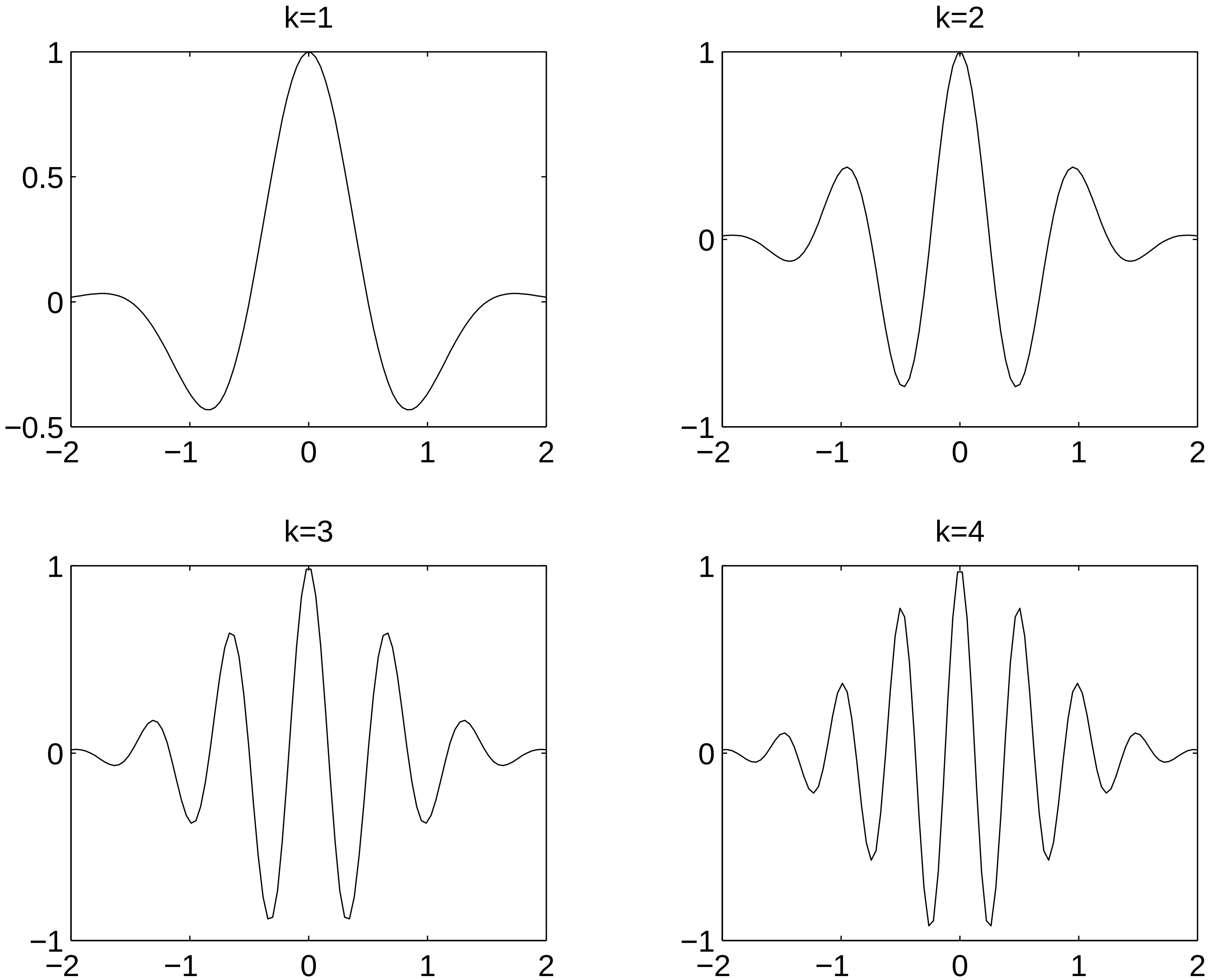


Esempio **8 (Grafici in sottofinestre)** Se vogliamo disegnare in diverse sottofinestre grafici di diverse funzioni, ad esempio della funzione

$$f(x) = e^{-x^2} \cos(kx),$$

sull'intervallo $[-2, 2]$ al variare di k intero potremo scrivere

```
>> x=linspace(-2,2);
>> y=exp(-x.^2).*cos(pi*x);
>> subplot(2,2,1);
>> plot(x,y);  title('k=1');
>> y=exp(-x.^2).*cos(2*pi*x);
>> subplot(2,2,2);
>> plot(x,y); title('k=2');
>> y=exp(-x.^2).*cos(3*pi*x);
>> subplot(2,2,3);
>> plot(x,y); title('k=3');
>> y=exp(-x.^2).*cos(4*pi*x);
>> subplot(2,2,4);
>> plot(x,y); title('k=4');
```



k=1
1
0.5
0
−0.5
−2
−1
0
1
2
k=2
1
0
−1
−2
−1
0
1
2
k=3
1
0
−1
−2
−1
0
1
2
k=4
1
0
−1
−2
−1
0
1
2

Le capacità grafiche di MATLAB consentono di produrre grafici di funzioni in più variabili, ossia rappresentazioni grafiche di array a più indici. Così come una funzione di una variabile $y = f(x)$ definisce una curva nel piano, una funzione di due variabili indipendenti $z = f(x, y)$ definisce una superficie nello spazio tridimensionale.

Esempio **9 (Un semplice grafico di superficie)** Consideriamo la funzione

$$z = x(1 - x)y(1 - y),$$

nel dominio rettangolare $0 \leq x \leq 1$, $0 \leq y \leq 1$. Per costruirne il grafico della superficie dobbiamo innanzitutto definire la griglia di valori (x, y) nei quali valuteremo la nostra funzione $z = f(x, y)$. Per costruire la griglia di valori possiamo utilizzare la funzione `meshgrid` nella forma

```
>> n=5;m=5;
>> x=linspace(0,1,n);
>> y=linspace(0,1,m);
>> [X,Y]=meshgrid(x,y)
X =
```

```
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
Y =
         0         0         0         0         0
    0.2500    0.2500    0.2500    0.2500    0.2500
    0.5000    0.5000    0.5000    0.5000    0.5000
    0.7500    0.7500    0.7500    0.7500    0.7500
    1.0000    1.0000    1.0000    1.0000    1.0000
```

La funzione costruisce due matrici $m \times n$

$$X = \begin{pmatrix} x(1) & x(2) & \dots & x(n) \\ x(1) & x(2) & \dots & x(n) \\ \vdots & \vdots & & \vdots \\ x(1) & x(2) & \dots & x(n) \end{pmatrix} \quad Y = \begin{pmatrix} y(1) & y(1) & \dots & y(1) \\ y(2) & y(2) & \dots & y(2) \\ \vdots & \vdots & & \vdots \\ y(m) & y(m) & \dots & y(m) \end{pmatrix}$$

Per creare il grafico della superficie è sufficiente aggiungere le istruzioni

```
>> Z = X.*(1-X).*Y.*(1-Y);
>> surf(X,Y,Z);
>> xlabel('x');ylabel('y');zlabel('z');
```

Si noti come la matrice Z avrà la proprietà che $Z(i,j) = f(X(i,j), Y(i,j))$.

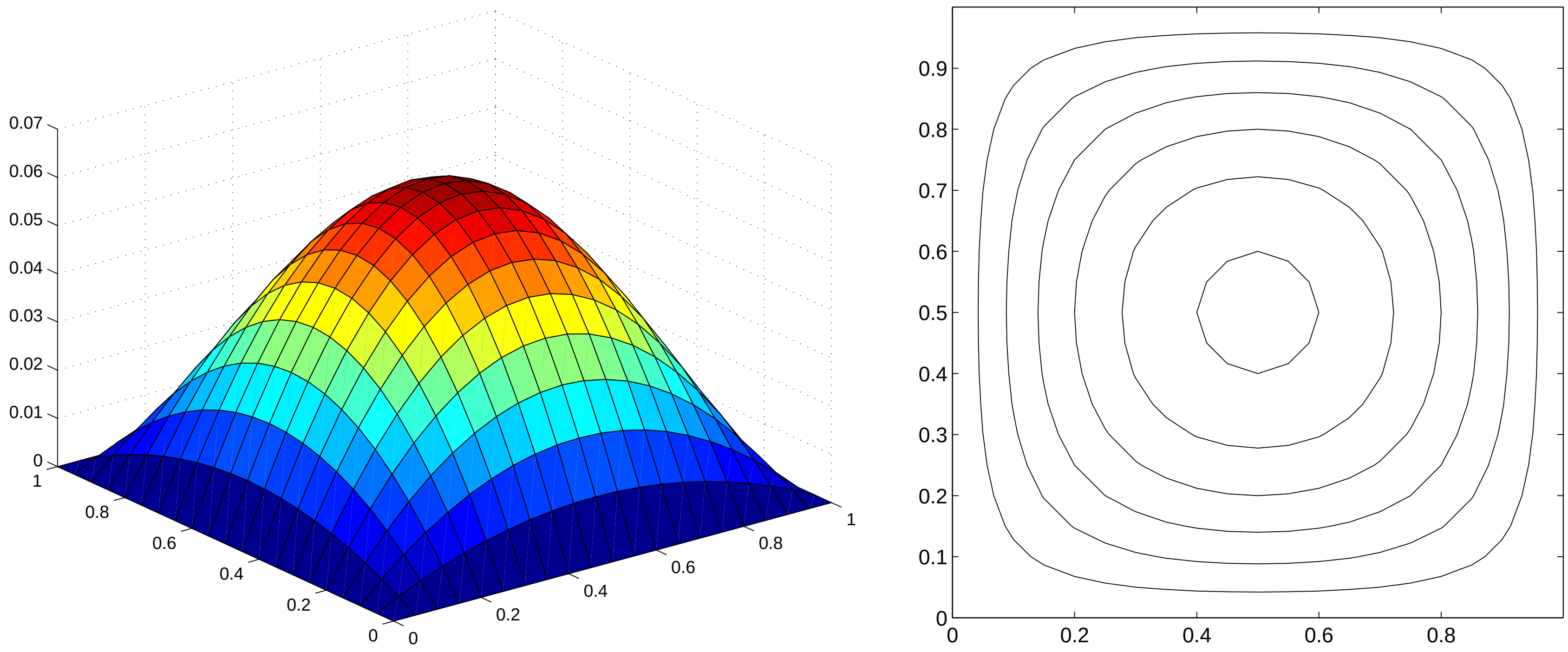


Grafico di superficie e curve di livello di $f(x,y) = x(1-x)y(1-y)$.

La funzione principale è quindi `surf` che ha la seguente sintassi

`surf(`*Matrice1*, *Matrice2*, *Matrice3*`)`

dove *Matrice1* e *Matrice2* sono matrici quadrate contenenti i valori delle variabili x e y, mentre *Matrice3* è una matrice quadrata tale che l'elemento di indici i, j è dato da $f(x(i), y(j))$. Esistono inoltre altre forme per utilizzare la funzione e si rimanda al solito all'help in linea.

Il grafico di superfici può essere personalizzato in diversi modi. Oltre alle opzioni precedentemente discusse, ossia `axis`, `title`, `xlabel`, `ylabel` e `zlabel`, è possibile cambiare il colore, l'angolo di visualizzazione e realizzare grafici di tipo differente. Spesso risulta utile avere una rappresentazione bidimensionale della superficie, analoga a quella comunemente utilizzata nella realizzazione di mappe topografiche. Utilizzando le matrici X, Y e Z costruite in precedenza è sufficiente sostituire l'istruzione `surf` con

```
>> contour(X,Y,Z);
```

In Figura possiamo confrontare il risultato grafico di `contour` con quello ottenuto in precedenza con la funzione `surf`. Il grafico mostra le curve di livello della funzione, ossia le curve sulle quali $z = f(x, y)$ risulta costante.

Funzione	**Significato**
`view`	cambia l'orientamento del grafico
`colormap`	cambia il colore del grafico
`shading`	cambia l'ombreggiatura del grafico
`mesh`	disegna un grafico a griglia
`contour`	disegna un grafico a curve di livello
`surf`	disegna un grafico di superficie

Alcune funzioni per grafici di superfici

Script e function files

Le successioni di comandi viste negli esempi precedenti possono essere memorizzate direttamente in un file di testo. In questo modo si crea uno *script* o *m-file* MATLAB (in quanto è obbligatorio assegnare al file l'estensione ".m"). Scrivendo il nome del file nella finestra principale di MATLAB si ottiene lo stesso risultato che si sarebbe ottenuto scrivendo uno ad uno i comandi elencati nello script. In sostanza uno script è un vero e proprio programma che ci consente di memorizzare e organizzare istruzioni MATLAB al fine di realizzare progetti di una certa complessità.

Esempio **10 (Disegnare figure piane con MATLAB)** Supponiamo di volere disegnare nel piano una poligonale chiusa. La funzione `plot` si presta bene a questo scopo in quanto raccorda tramite segmenti di retta in modo ordinato i vertici dei vettori argomento. Per disegnare una poligonale chiusa ad n vertici il primo e l'ultimo vertice memorizzati devono coincidere, quindi dovremo memorizzare $n+1$ vertici. Ad esempio nel caso di un quadrato di lato unitario avremo bisogno di memorizzare $4+1$ vertici

```
>> x=[0 1 1 0 0];
>> y=[0 0 1 1 0];
>> plot(x,y)
>> axis('square')
```

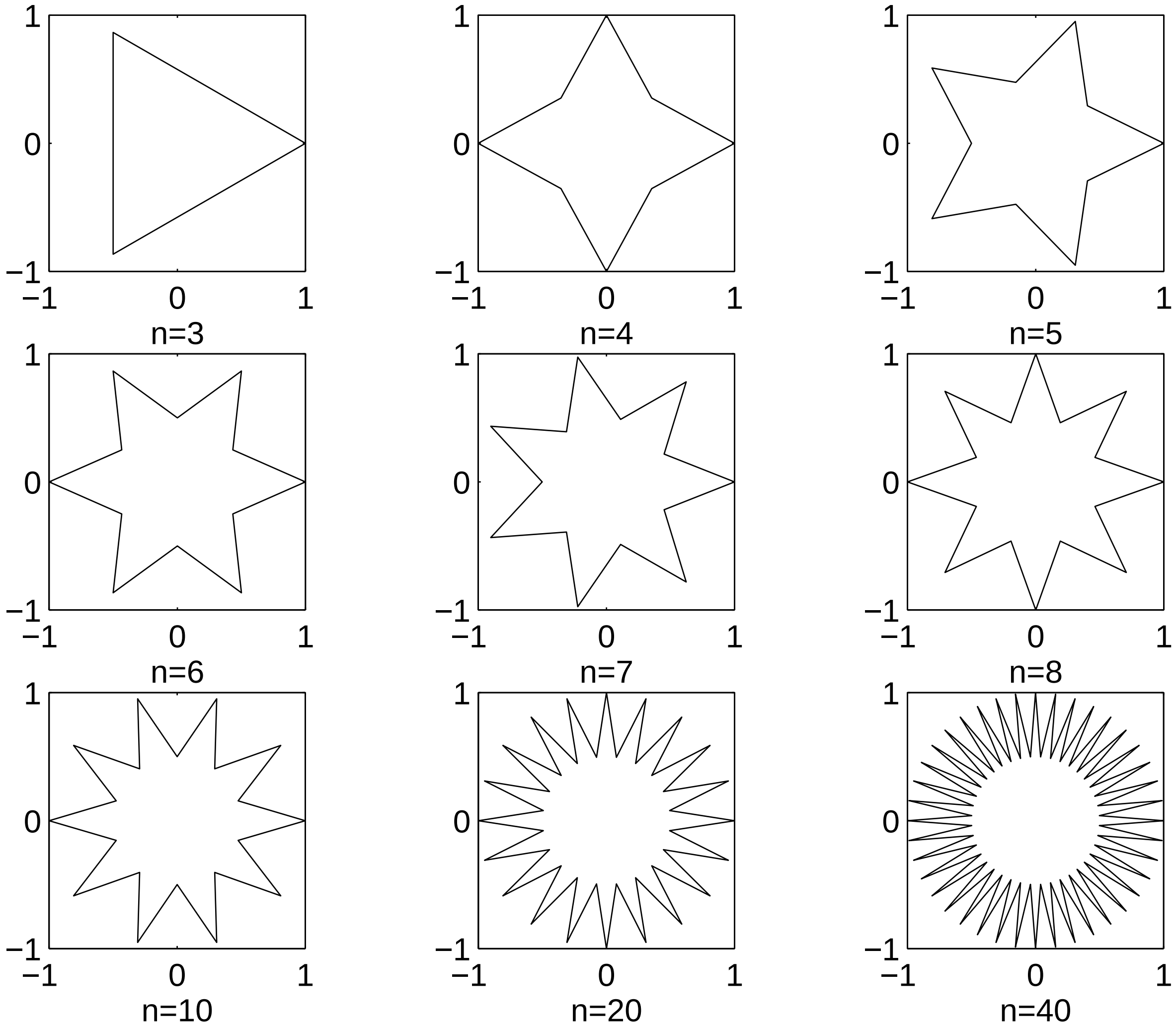
L'istruzione `axis('square')` fa sì che la figura venga visualizzata utilizzando la stessa scala per entrambi gli assi. Se ora vogliamo realizzare il progetto più complesso di disegnare una stella con un numero prefissato di punte possiamo memorizzare la successione dei comandi nello script `stella.m`

```
% stella.m
% Disegna una stella a n punte
%
n=5;a=0.5;
t=linspace(0,2*pi,n+1);
s=linspace(pi/n,2*pi-pi/n,n);
x(1:2:2*n+1)=cos(t);
x(2:2:2*n)=a*cos(s);
y(1:2:2*n+1)=sin(t);
y(2:2:2*n)=a*sin(s);
plot(x,y);
axis('square');
```

e scrivere nella finestra di comando di MATLAB

```
>> stella
```

per ottenere il risultato desiderato.



Il carattere % serve per introdurre un commento all'interno dello script, MATLAB ignora il contenuto alla destra del carattere % fino alla linea successiva. Il commento all'inizio dello script file è particolarmente importante in MATLAB, infatti richiamando il comando `help` seguito dal nome dello script otteniamo come risposta il commento inserito all'inizio dello script stesso

```
>> help stella
 stella.m
 Disegna una stella a n punte
```

Una caratteristica degli script è quella di non avere parametri in ingresso modificabili. Ad esempio se vogliamo modificare i valori di a e n dobbiamo modificare ogni volta lo script. Un'altra particolarità consiste nel fatto che tutte le variabili usate nello script vengono automaticamente messa nella memoria di lavoro di MATLAB.

Una funzione MATLAB risponde esattamente alle precedenti due necessità, ossia è uno script al quale è possibile passare parametri in ingresso ed ottenerne in uscita, inoltre le variabili utilizzate durante l'esecuzione della funzione vengono automaticamente cancellate dalla memoria di MATLAB al termine della stessa A differenza di uno script standard la sintassi di una funzione richiede che la prima riga abbia la struttura

function *Parametri in uscita* = *Nomefunzione*(*Parametri in Ingresso*)

generalmente seguita da alcune righe di commento, che possono ancora essere visualizzate tramite `help` *Nomefunzione*. Altre regole importanti da ricordare nella costruzione di funzioni consistono nell'obbligo di salvare la funzione come script file `Nomefunzione.m` e che nel corpo della funzione devono essere assegnati dei valori ai *Parametri in Uscita*. Ad esempio una funzione MATLAB che rende più versatile il precedente script è data da

```
function [xv,yv]=puntistella(ap,np)
% Sintassi [xv,yv]=puntistella(ap,np)
% Costruisce due vettori xv e yv contenenti i vertici
% di una stella a np punte. Il parametro ap, 0 < ap < 1,
```

```
% consente di modificare la lunghezza delle punte
%
t=linspace(0,2*pi,np+1);
s=linspace(pi/np,2*pi-pi/np,np);
xv(1:2:2*np+1)=cos(t);
xv(2:2:2*np)=ap*cos(s);
yv(1:2:2*np+1)=sin(t);
yv(2:2:2*np)=ap*sin(s);
```

Uno script che utilizza la funzione ora scritta è per esempio

```
% stella2.m
% Disegna una stella a n punte
%
n=5;a=0.5;
[x,y]=puntistella(a,n);
plot(x,y);
axis('square');
```

Si noti come i parametri formali `ap, np` assumano i valori assegnati ad `a` e `n` al momento della chiamata della funzione, così come i parametri formali `xv, yv` forniscono i vettori risultato alle variabili `x` e `y`.

Va rilevato inoltre che, al fine di poter usare una funzione all'interno di uno script, la funzione dovrà trovarsi nella stessa directory dello script oppure in una directory tra quelle predefinite da MATLAB (si scriva **`help path`** per maggiori informazioni al riguardo). Le stesse osservazioni valgono quando si desidera eseguire uno script dalla finestra di comando di MATLAB. Segnaliamo infine come dalla versione 5 di MATLAB sia possibile scrivere e memorizzare più funzioni direttamente all'interno dello stesso script file.

MATLAB come linguaggio di programmazione

Anche gli algoritmi più semplici richiedono l'esecuzione ripetuta di istruzioni e l'esecuzione condizionata di alcune parti. La costruzione ripetuta di blocchi di codice in MATLAB viene eseguita tramite cicli. Esistono due diversi modi per realizzare cicli, il ciclo incondizionato `for...end` ed il ciclo condizionato `while...end`. La sintassi del primo è la seguente

```
for Indice=Espressione
    blocco di istruzioni
end
```

Dove *Indice* è una quantità che assume diversi valori a seconda di *Espressione*. Esistono numerose possibilità a questo proposito, e senza avere la pretesa si essere esaustivi ci limiteremo a considerare alcuni esempi.

Esempio **11 (Valore medio di vettori e matrici)** Consideriamo il semplice problema del calcolo del valore medio di un vettore o di una matrice. Dato un vettore x di n componenti il valore medio è definito come

$$m = \frac{1}{n} \sum_{i=1}^{n} x(i).$$

Analogamente nel caso di una matrice $m \times n$ il valore medio sarà

$$M = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} A(i,j).$$

Nel caso di un vettore tale risultato può essere ottenuto con la seguente funzione

```
function m=mediav(x)
% Sintassi m=mediav(x)
% Calcola la media del vettore x
%
n=length(x);
somma=0;
for i=1:n
   somma=somma+x(i);
end
m=somma/n;
```

Si noti come la variabile "i" assuma i valori $1, 2, \ldots, n$ specificati dalla notazione due punti alla destra dell'uguale. In modo analogo potremo utilizzare un qualunque vettore per assegnare valori diversi dell'indice i all'interno del ciclo (valori decrescenti oppure non interi sono possibili).

Ad esempio il ciclo `for` precedente può esser sostituito da

```
for xi=x
   somma=somma+xi;
end
```

dove l'indice `xi` assume il valore degli elementi di x. Potremo utilizzare la precedente funzione nel seguente modo

```
>> x=[1 2 3 4 5 6];
>> mediav(x)
ans=
    3.5000
```

Non è obbligatorio, ma è fortemente consigliato, scrivere il blocco di istruzioni all'interno del ciclo allineandolo con l'espressione che governa il ciclo. Questo modo di procedere, che utilizzeremo anche in seguito, consente di evidenziare bene le parti di codice che vengono eseguite nei cicli o sotto opportune condizioni. In particolare l'uso del ciclo `for` può essere evitato utilizzando la funzione MATLAB `sum` nella forma `somma=sum(x)` che assegna alla variabile `somma` la somma degli elementi del vettore x. Si confronti la funzione `mediav` con la funzione `mean` predefinita in MATLAB.

In maniera analoga per calcolare la media di una matrice potremo utilizzare la seguente funzione

```
function M=mediam(A)
% Sintassi M=mediam(A)
% Calcola la media della matrice A
%
[m,n]=size(A);
somma=0;
for i=1:m
  for j=1:n
     somma=somma+A(i,j);
  end
end
```

```
M=somma/(m*n);
```

I due cicli for sono “annidati” l’uno all’interno dell’altro ed il blocco principale di istruzioni viene eseguito mn volte. Per utilizzarla si puó scrivere

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

Si consideri infine la seguente versione alternativa della precedente funzione

```
function M=mediam2(A)
% Sintassi M=mediam2(A)
% Calcola la media della matrice A
[m,n]=size(A);
somma=0;
for v=A
   somma=somma+mediav(v);
```

```
end
M=somma/n;
```

In questo caso il ciclo è eseguito n volte, ossia il numero di colonne della matrice A, e l'indice v assume il valore delle righe della matrice A. Chiaramente entrambe le funzioni saranno corrette anche se applicate ad un vettore. Si noti che l'uso della funzione `mean` su una matrice restituisce un vettore riga contenete il valore medio delle colonne della matrice. Per ottenere il valor medio della matrice A dovremo scrivere `mean(mean(A))`. In modo analogo agisce anche la funzione `sum` se applicata ad una matrice.

Gli esempi precedenti hanno la caratteristica di ripetere un blocco di istruzioni un numero prefissato di volte. In molte circostanze si ha la necessità di ripetere un certo numero di operazioni diverse volte a seconda che una certa condizione sia verificata oppure no. In questo caso si utilizza il costrutto

```
while Condizione
   blocco di istruzioni
end
```

Dove *Condizione* è un'espressione che MATLAB valuta numericamente e che viene interpretata come vera se diversa da zero.

Esempio 12 (Calcolo di successioni di numeri interi) Il problema che consideriamo è quello del calcolo della seguente successione di numeri interi generata a partire da a_1 numero intero intero

$$a_{n+1} = 3a_n - 1, \qquad n \geq 1$$

Ad esempio per $a_1 = 9$ la successione fornisce i termini $9, 26, 77, 230, 689, \ldots$, e continua a crescere indefinitamente. Chiaramente se volessimo generare i primi 20 elementi della successione potremmo utilizzare un ciclo `for` come in precedenza. Vogliamo arrestare il calcolo della successione quando $a(n) > a^*$ assegnato. Una soluzione possibile è contenuta nella seguente funzione

```
function x=successione(a1,as)
% Sintassi x=successione(a1,as)
% Calcola la successione
%
```

```
%       a(n+1)= 3*a(n)-1
%
% partendo da a(1)=a1, fino a quando a(n) < as
%
a(1)=a1;
n=1;
while a(n) < as
    a (n+1) = 3*a(n)-1;
    n = n+1;
end
```

Quanto scritto merita alcune osservazioni. Innanzitutto il ciclo `while` viene eseguito solo se la *Condizione* $a(n)$ sia minore di as risulti verificata, in MATLAB questo corrisponde all'istruzione `a(n) < as`. Va detto che MATLAB restituisce il valore numerico zero se la condizione risulta falsa ed uno se risulta vera. Attenzione a non confondere l'operatore relazionale == che confronta il valore della variabile a sinistra dell'operatore con quello della variabile a destra, con l'operatore di assegnazione = che invece assegna il valore della quantità alla destra dell'operatore alla variabile a sinistra dello stesso (vedi Tabella).

Operatore	**Significato**
<	minore
<=	minore o uguale
==	uguale
>=	maggiore o uguale
>	maggiore
~=	non uguale

Operatori relazionali in MATLAB

Consideriamo ora la successione in cui, partendo da x_1 intero positivo (che in questo caso assumeremo maggiore di due) è definita come

$$x_{n+1} = \begin{cases} x_n/2 + 1 & \text{se } x_n \text{ è pari} \\ 3x_n - 1 & \text{se } x_n \text{ è dispari} \end{cases} \qquad n \geq 1.$$

Ad esempio per $x_1 = 9$ la successione fornisce i termini $9, 26, 14, 8, 5, 14, 8, 5, 14, \ldots$ ed ha un andamento oscillante in cui da un certo indice in poi si ha la sequenza di numeri $14, 8, 5$ che si ripete. Chiaramente questo avviene ogni volta che il numero 14 è generato dalla successione. Vogliamo allora arrestare il calcolo della successione non appena viene raggiunto il valore 14. Avremo in questo caso la funzione

```
function x=successione2(x1)
% Sintassi x=successione2(x1)
% Calcola la successione
%
%           | x(n)/2+1 se x(n) e' pari
%   x(n) = <
%           | 3*x(n)-1 se x(n) e' dispari
%
% partendo da x(1)=x1, fino a quando x(n)=14
%
x(1)=x1;
n=1;
```

```
while x(n) ~= 14
    if rem(x(n),2) == 0
        x(n+1) = x(n)/2+1;
    else
        x(n+1) = 3*x(n)-1;
    end
    n = n+1;
end
```

Una caratteristica della precedente funzione è l'uso della funzione MATLAB `rem` per controllare se $x(n)$ è pari o dispari. Tale funzione usata nella forma `rem(a,b)` con a e b interi restituisce il resto (remainder) della divisione b/a. La verifica viene effettuata tramite il costrutto di confronto `if...else...end` la cui sintassi generale è

```
if Condizione1
      blocco di istruzioni
elseif Condizione2
      blocco di istruzioni
...
else
      blocco di istruzioni
end
```

Dove il primo blocco di istruzioni sarà eseguito solo se la *Condizione1* risulta essere verificata, il secondo solo se la *Condizione1* risulta essere falsa e la *Condizione 2* vera e così via. Il blocco di istruzioni che segue `else` sarà eseguito soltanto se nessuna delle precedenti condizioni risulti essere vera.

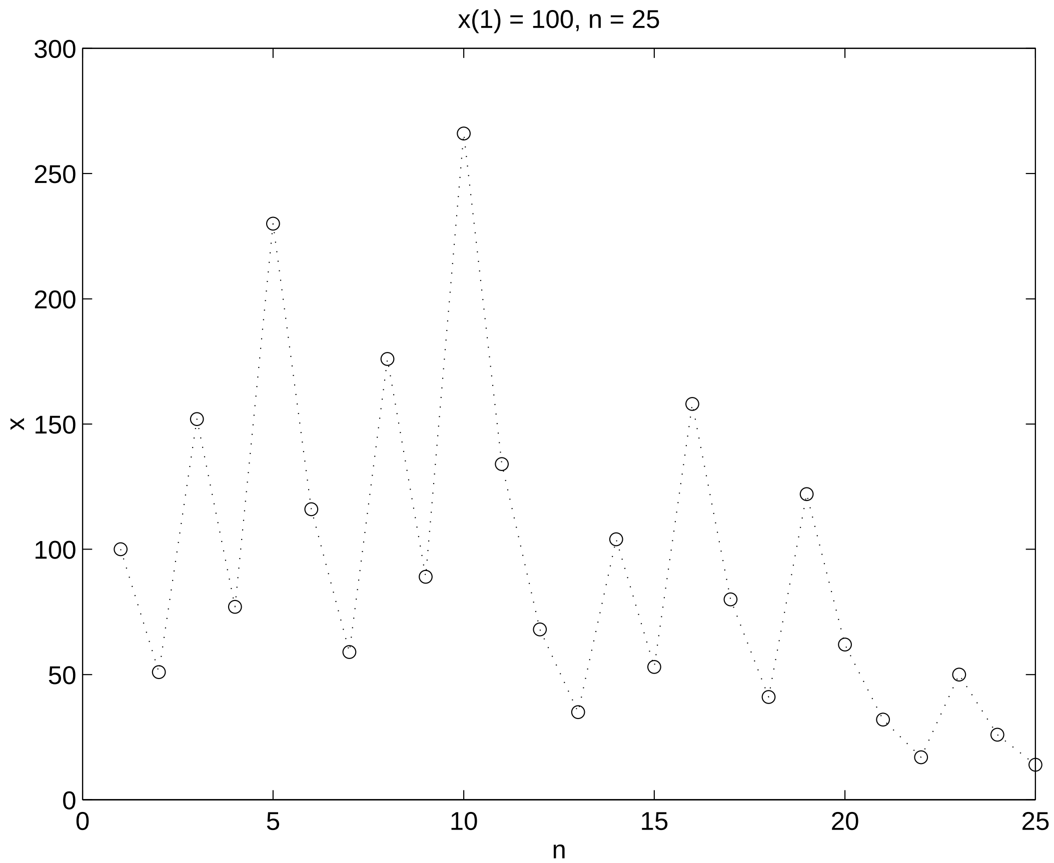


Grafico generato con il comando `plot(successione2(100),':o')`.

Gli operatori relazionali che abbiamo visto nel precedente esempio possono essere combinati tra di loro tramite operatori logici. In Tabella sono riportati gli operatori logici in MATLAB e la loro “azione”.

Operatore	Significato
&	and
—	or
∼	not
xor	or esclusivo

a	b	a & b	a | b	∼a	xor(a,b)
0	0	0	0	1	0
1	0	0	1	0	1
0	1	0	1	1	1
1	1	1	1	0	0

Operatori logici in MATLAB e loro azione

Poiché a priori non sappiamo se la successione fornirà sempre il valore 14 per evitare che il precedente ciclo si trasformi in un ciclo infinito o più semplicemente per evitare la costruzione di vettori eccessivamente lunghi potremo modificare la funzione `successione` sostituendola con la seguente

```
function x=successione2b(x1,N)
% Sintassi x=successione2b(x1,N)
% Calcola al piu' N termini della successione
%
```

```
%           | x(n)/2+1 se x(n) e' pari
%   x(n) = <
%           | 3*x(n)-1 se x(n) e' dispari
%
% partendo da x(1)=x1, fino a quando x(n)=14
%
x(1)=x1;
n=1;
while (x(n) ~= 14)
    if rem(x(n),2) == 0
         x(n+1) = x(n)/2+1;
    else
         x(n+1) = 3*x(n)-1;
    end
    if n > N
       break ;
    end
    n = n+1;
end
```

Il comando `break` consente di uscire in maniera forzata da un ciclo ed evita in questo caso che siano calcolati più di N termini della successione. Quando il comando `break` è eseguito MATLAB salta automaticamente all'istruzione `end` che termina il ciclo. Un comando che svolge una funzione analoga in MATLAB è `return`. La differenza è che `return` interrompe l'esecuzione della funzione e ritorna al programma da cui tale funzione era stata chiamata. Lo stesso risultato si può ottenere tramite la funzione

```
function x=successione2c(x1,N)
% Sintassi x=successione2c(x1,N)
% Calcola al piu' N termini della successione
%
%           | x(n)/2+1 se x(n) e' pari
%   x(n) = <
%           | 3*x(n)-1 se x(n) e' dispari
%
% partendo da x(1)=x1, fino a quando x(n)=14
% Il parametro N e' opzionale
%
```

```
if nargin==1, N=500; end
x=zeros(1,N);
x(1)=x1;
n=1;
while (x(n) ~= 14) & (n < N)
    if rem(x(n),2) == 0
        x(n+1) = x(n)/2+1;
    else
        x(n+1) = 3*x(n)-1;
    end
    n = n+1;
end
x=x(1:n);
```

Il ciclo sarà eseguito fintantoché l'istruzione `(x(n) ~= 14) & (n < N)` restituisce il valore uno (vero), ossia fino a che entrambe le condizioni `x(n) ~= 14` e `n < N` risultino entrambe verificate. Altri aspetti della precedente funzione meritano di essere evidenziati. Innanzitutto l'uso della variabile interna MATLAB `nargin` che all'inerno di una funzione assume un valore pari al numero di parametri passati in ingresso alla funzione stessa. Tale variabile interna consente quindi di costruire funzioni con un numero di parametri variabile, caratteristica come alle funzioni interne di MATLAB. Questo significa che l'istruzione

```
>> x=successione2b(100);
```

assumerà $N = 500$ in quanto avendo passato alla funzione un solo parametro `nargin` risulterà uguale a uno. Si noti inoltre come una breve istruzione `if...else ...end` possa essere scritta su un'unica linea a patto di utilizzare una virgola come separatore tra la condizione ed il blocco di istruzioni.

Un secondo aspetto interessante è dato dall'inizializzazione del vettore x tramite l'istruzione `x=zeros(1,N)`. Questo permette a MATLAB di non aggiungere in memoria un nuovo elemento al vettore x, rendendo così più efficiente la gestione automatica della memoria: nel caso di aggiunte progressive devono intervenire continuamente le routine di gestione della memoria. L'ultima riga della funzione `x=x(1:n)` serve invece ad eliminare dalla memoria di MATLAB la parte di vettore che non abbiamo utilizzato.

In particolare va evidenziato come in MATLAB Anche gli operatori logici e relazionali possono essere applicati a vettori e matrici. Ad esempio avremo

```
>> x=1:5;
>> y=5:-1:1;
>> z = x > y
z =
     0     0     0     1     1
```

In alcune circostanze questa notazione compatta risulta particolarmente utile, ad esempio per individuare gli elementi non nulli di una matrice o di un vettore.

Osservazione 1 La caratteristica principale di MATLAB consiste nella possibilià di eseguire operazioni vettoriali. L'uso efficiente di MATLAB è strettamente legato alla capacità dell'utente di sfruttare tale caratteristica. Un ciclo di istruzioni di tipo `for ...end` o `while ...end` comporta l'esecuzione ripetuta in forma sequenziale di un blocco di istruzioni. È buona regola prima di scrivere un ciclo cercare di vedere se è possibile evitarlo tramite un uso opportuno di istruzioni vettoriali.

Inoltre va notato che MATLAB, come linguaggio di programmazione, è un linguaggio interpretato e solo recentemente sono stati introdotti opportuni compilatori. L'efficienza e la velocità saranno quindi ridotte rispetto agli usuali programmi scritti in linguaggi ad alto livello, come il C o il Fortran, e compilati.

L'ultimo costrutto MATLAB che andremo ad analizzare è una struttura di confronto analoga a `if...else...end`. Tale struttura nella forma `switch...case..` risulta particolarmente utile qualora si debbano eseguire numerose scelte di tipo esclusivo (se una è verificata le altre sono false). La sintassi del comando è la seguente

```
switch Espressione
    case Valore1
         blocco di istruzioni
    case Valore2
         blocco di istruzioni
      ...
    otherwise
         blocco di istruzioni
end
```

I blocchi di istruzioni sono eseguiti solo se l'*Espressione* assume il corrispondente *Valore*. L'ultimo blocco di istruzioni sarà eseguito solo nel caso in cui *Espressione* non abbia assunto nessuno dei precedenti valori. Il blocco di istruzioni che corrisponde alla scelta tra $x(n)$ pari e dispari diventerá

```
z = rem(x(n),2) ;
switch z
   case 0
      x(n+1) = x(n)/2+1;
   case 1
```

```
        x(n+1) = 3*x(n)-1;
    otherwise
        disp('Non hai inserito un numero intero');
end
```

L'istruzione `disp` consente di visualizzare una stringa di testo sullo schermo del calcolatore.

Input/Output essenziale

La sintassi della funzione `disp` è la seguente

`disp(`*stringa di caratteri*`)`

dove *stringa di caratteri* rappresenta un array di stringhe di tipo vettore o matrice, dove ogni stringa è racchiusa tra apici. Ad esempio potremo scrivere

```
>> disp('Oggi e'' lunedi'')
oggi e' lunedi'
>> disp(['Gennaio ','Febbraio ','Marzo'])
Gennaio Febbraio Marzo
```

Osserviamo che per utilizzare i simboli di apostrofo/accento all'interno della stringa si deve utilizzare il simbolo '' come delimitatore per evitare conflitti con il segnale di inizio e fine della stringa. Nel secondo caso l'argomento è un vettore riga di stringhe, che vengono così concatenate. Se vogliamo costruire vettori colonna è importante ricordarsi che le stringhe devono avere tutte la stessa dimensione

```
>> disp(['Gennaio ';'Febbraio';'Marzo   '])
Gennaio
Febbraio
Marzo
```

cosa che può essere facilmente ottenuta inserendo un opportuno numero di spazi. In molte circostanze si ha inoltre la necessità di rappresentare valori numerici in uscita. Vediamo due differenti possibilità, la prima tramite il seguente esempio

```
>> x=10;
>> stringa = ['x = ', num2str(x)];
>> disp(stringa)
x = 10
```

In questo caso abbiamo utilizzato la funzione `num2str` che consente di convertire una variabile numerica in una stringa.
Un modo più versatile di costruire stringhe di caratteri per l'output di testo con formato è fornito dalle funzioni `fprintf` e `sprintf` mutuate direttamente dal linguaggio C. La sintassi della prima è del tipo

`fprint(`*Fid*, *Formato*, *Variabili*`)`

dove *Formato* è una stringa di testo che tramite l'uso di caratteri speciali indica il tipo di formato dell'output, *Variabili* è una lista opzionale di variabili

separate da una virgola e che hanno un corrispondente all'interno della stringa *Formato*. Infine *Fid* è un identificatore opzionale del file al quale l'output è inviato.
La funzione `sprintf` ha invece la sintassi

Stringa = `sprint`(*Formato*, *Variabili*)

dove in questo caso l'output viene indirizzato su una stringa di testo.
Potremo quindi scrivere per esempio

```
>> x=10;y=5.5;
>> fprintf('x = %d e y= %f\n',x,y);
x = 10 e y= 5.500000
>> stringa=sprintf('x = %d e y= %f\n',x,y);
>> disp(stringa)
x = 10 e y= 5.500000
```

La stringa di formato contiene codici che specificano il tipo di variabile che deve essere convertita in stringa e rappresentata sullo schermo del calcolatore o su file. il formato `%d` serve a visualizzare un numero intero, mentre il formato `%f` è utilizzato per i numeri reali. In Tabella sono riportati alcuni descrittori di formato nella loro forma base e in Tabella la loro azione sulla rappresentazione di alcuni valori numerici.

Codice di formato	**Azione**
`%s`	formato stringa
`%d`	formato senza parte frazionaria (intero)
`%f`	formato numero decimale
`%e`	formato in notazione scientifica
`%g`	formato in forma compatta usando `%f` o `%e`
`\n`	inserisce carattere di ritorno a capo
`\t`	inserisce carattere di tabulazione

Principali codici di formato in `fprintf` e `sprintf`

Valore	**%6.3f**	**%6.3e**	**%6d**
2	2.000	2.000e+00	2
0.02	0.020	2.000e-02	2.000000e-02
200	200.000	2.000e+02	200
sqrt(2)	1.414	1.414e+00	1.414214e+00
sqrt(0.02)	0.141	1.414e-01	1.414214e-01

Azione di alcuni descrittori di formato in `fprintf` e `sprintf`

La principale differenza tra le funzioni MATLAB `fprintf` e `sprintf` e le equivalenti versioni in C, è data dalla possibilità di uso vettoriale, come vedremo nel prossimo esempio.

Esempio 13 (Salvare una semplice tabella di valori) Abbiamo già visto nell'Esempio 4, come costruire una semplice tabella di valori per le funzioni seno e coseno. Uno script file che realizza tale tabella utilizzando una migliore visualizzazione è il seguente

```
% tabcossin.m
% Realizza una tabella di valori di coseno e seno
%
n=input('Inserisci il numero di valori ? : ');
x=linspace(0,pi,n);
c=cos(x);
s=sin(x);
disp('----------------------------------------');
fprintf('k\t x(k)\t cos(x(k))\t cos(x(k))\n');
disp('----------------------------------------');
fprintf('%d\t %3.2f\t %6.5f\t %6.5f\n',[1:n;x;c;s]);
```

che produce il seguente output

```
>> tabcossin
>> Inserisci il numero di valori ? : 5
```

```
----------------------------------------
k         x(k)    cos(x(k))        sin(x(k))
----------------------------------------
1         0.00    1.00000          0.00000
2         0.79    0.70711          0.70711
3         1.57    0.00000          1.00000
4         2.36    -0.70711         0.70711
5         3.14    -1.00000         0.00000
```

La funzione `input` è stata utilizzata per assegnare un valore al numero di punti n. La sintassi della funzione è

Variabile=`input`(*Stringa di caratteri*)

ed attende un ingresso da tastiera di un'espressione MATLAB da assegnare a *Variabile*. Il valore assegnato potrà quindi essere di tipo scalare, vettore oppure matrice utilizzando la sintassi standard di MATLAB. Si veda **help input** per l'assegnazione in ingresso di stringhe di caratteri.
Si noti l'uso vettoriale di **fprintf**. Bisogna prestare attenzione nel caso

in cui si utilizzi una matrice come variabile in output, il comando legge la matrice per colonne, applicando ad ogni elemento il corrispondente descrittore di formato. La visualizzazione avviene naturalmente per righe e questo porta ad una sorta di 'trasposizione' della matrice.
Il modo più semplice di salvare una copia della precedente tabella consiste nell'utilizzo del comando `diary` che trasferisce su file tutto ciò che compare nella finestra di comando di MATLAB. La sintassi del comando è

`diary` *Nomefile*

per attivare la copia sul file di testo chiamato *Nomefile* di tutto ciò che comparirà sulla finestra di comando fino a quando non verrà dato il comando `diary off` per ripristinare la situazione originale. Ad esempio il seguente

```
% salvatabella.m
% Esempio di uso della funzione diary
%
diary tabella.txt
tabcossin
```

```
diary off
```

costruisce sul file di testo chiamato `tabella.txt` la tabella così come visualizzata in precedenza.

Un modo sicuramente più versatile di registrare dati su file è fornito dal comando `save`. Il seguente script costruisce un file di testo `tabella.dat` contenente solo i valori della precedente tabella (non tutta la sessione di lavoro come prima)

```
% salvavalori.m
% Esempio di output su file con save
%
n=input('Inserisci il numero di valori ? : ');
x=linspace(0,pi,n);
c=cos(x);
s=sin(x);
save tabella.dat 1:n x c s -ascii
```

Il comando `save` consente di registrare alcune variabili presenti nella memoria di MATLAB su file secondo la sintassi

`save` *Nomefile Elenco Variabili Formato*

dove *Formato* è un parametro opzionale. Se tale parametro è omesso il file è salvato in formato binario e qualora il file non abbia estensione a questo è aggiunta l'estensione `.mat`. Il formato `-ascii` consente di salvare file in modalità testo. Potremo leggere la precedente tabella tramite il seguente

```
% visualizzavalori.m
% Esempio di input da file con load
%
load tabella.dat
A=tabella;
disp('----------------------------------------');
fprintf('k\t x(k)\t cos(x(k))\t cos(x(k))\n');
disp('----------------------------------------');
```

```
fprintf('%d\t %3.2f\t %6.5f\t %6.5f\n',A');
```

dove abbiamo usato il comando

`load` *Nomefile Formato*

per leggere il file. Se vogliamo leggere un file di testo che non ha estensione allora l'uso del formato `-ascii` è obbligatorio anche in lettura, altrimenti è opzionale. Si noti che MATLAB crea una matrice contenente i valori precedentemente registrati ed a questa assegna il nome del file in lettura. Nell'esempio precedente l'istruzione `load tabella.dat` legge i valori presenti nel file `tabella.dat` e li assegna ad una matrice chiamata `tabella`. La riga successiva `A=tabella` assegna la matrice di valori alla variabile `A`.

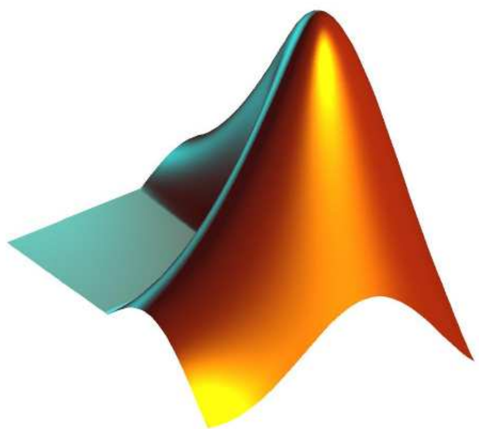
Segnaliamo infine come sia possibile salvare i grafici prodotti in MATLAB in file utilizzando diversi formati grafici. La sintassi base del comando

`print` *Opzioni Nomefile*

consente di salvare il contenuto della attuale finestra grafica sul file *Nomefile* utilizzando un particolare formato grafico specificato in *Opzioni*. In Tabella sono riportati i formati grafici principali. Ad esempio, in questo testo è stato usato il formato PostScript, che risulta particolarmente adatto per la stampa di figure. Il seguente semplice script realizza la figura tridimensionale che MATLAB utilizza come logo, tramite il comando`logo` e la salva in formato jpeg nel file mlogo.jpg

```
% salvalogo.m
% Esempio di output grafico su file
logo
printf -djpeg mlogo.jpg
```

Opzione	**Formato grafico**
`-dps`	PostScript bianco e nero
`-deps`	Encapsulated PostScript bianco e nero
`-dpsc`	PostScript a colori
`-depsc`	Encapsulated PostScript a colori
`-dgif`	Immagine GIF
`-dbmp`	Immagine Bitmap
`-djpeg`	Immagine JPEG compressa



Il logo di MATLAB salvato nel file `mlogo.jpeg`.

Numeri macchina o la macchina dà i numeri ?

Fino a questo momento non ci siamo occupati di come MATLAB, o più in generale un calcolatore, esegue le operazioni che noi gli abbiamo richiesto. Abbiamo già accennato all'inizio di questo capitolo ad alcuni aspetti caratteristici del calcolo numerico, quali ad esempio il fatto di utilizzare approssimazioni al posto di valori esatti. Per capire cosa vi sia all'origine di tutto ciò dobbiamo parlare di come i numeri vengono rappresentati all'interno della memoria di un calcolatore.

Numeri binari

Il sistema numerico cui siamo abituati è detto sistema numerico in base 10. Al contrario i calcolatore usano generalmente un sistema numerico in base 2. Come mai fino a questo momento non ce ne siamo accorti ? Le operazioni in MATLAB infatti venivano eseguite apparentemente nel nostro sistema numerico. Ad esempio

```
>> 57*13
ans=
     741
```

Questo non significa che il calcolatore usa un sistema in base 10, infatti quello che il calcolatore effettua è la conversione i nostri dati in ingresso in base 2 per poi eseguire le operazioni richieste sempre in base 2 e infine riconverte il risultato in base 10 per la visualizzazione (altrimenti saremmo costretti ad introdurre e ad osservare lunghe sequenze di cifre binarie: cosa che era normale nei primi calcolatori).
Nella rappresentazione dei numeri in base 10 se N è un intero positivo allora esistono $k+1$ cifre $d_0, d_1, \ldots, d_k$ appartenenti all'insieme $\{0, 1, 2, \ldots, 8, 9\}$ tali che N può essere scritto come

$$N = (d_k \times 10^k) + (d_{k-1} \times 10^{k-1}) + \ldots + (d_1 \times 10^1) + (d_0 \times 10^0).$$

Il numero N è quindi rappresentato in base 10 come

$$N = (d_k d_{k-1} \ldots d_1 d_0)_{10}.$$

Abbiamo quindi che

$$741 = (7 \times 10^2) + (4 \times 10^1) + (1 \times 10^0) = (741)_{10},$$

in quanto la base 10 è quella che utilizziamo comunemente per rappresentare i numeri.
L'idea precedente di espandere un numero scrivendolo come somma di prodotti di potenze in una prefissata base, può essere generalizzata. Se ad esempio consideriamo potenze di 2 invece che potenze di 10 avremo

$$\begin{aligned} 741 &= (1 \times 2^9) + (0 \times 2^8) + (1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) \\ &+ (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0). \end{aligned}$$

Che corrisponde al calcolo $741 = 512 + 128 + 64 + 32 + 4 + 1$.
In generale quindi, dato N numero intero positivo e B base intera positiva potremo affermare che esistono $j+1$ cifre $a_j, a_{j-1}, \dots, a_1, a_0$ nell'insieme $\{0, 1, \dots, B-2, B-1\}$ tali che N in base B ha la rappresentazione

$$\begin{aligned} N &= (a_j \times B^j) + (a_{j-1} \times B^{j-1}) + \ldots + (a_1 \times B^1) + (a_0 \times B^0) \qquad (1) \\ &= (a_j a_{j-1} \ldots a_1 a_0)_B. \end{aligned}$$

Con queste notazioni il calcolo effettuato in precedenza mostra che

$$741 = (1011100101)_2.$$

Tipicamente il numero di cifre necessarie per rappresentare un numero in base 2 è superiore al numero di cifre necessarie in base 10. Questo è dovuto al fatto che le potenze di due crescono molto più lentamente delle potenze di dieci. Solitamente per semplicità le parentesi $(\ldots)_{10}$ vengono omesse nel caso in cui la base è 10.

Esempio 14 (Conversione di un numero da base 10 a base qualunque) Per convertire un numero N da base 10 a base B intera possiamo procedere nel seguente modo. Se dividiamo l'espressione (1) per B otteniamo

$$\frac{N}{B} = (a_j \times B^{j-1}) + (a_{j-1} \times B^{j-2}) + \ldots + (a_1 \times B^0) + \frac{a_0}{B} = N_0 + \frac{a_0}{B},$$

dove N_0 è un intero positivo minore di N. In particolare $N = BN_0 + a_0$ perciò l'ultima cifra della rappresentazione a_0 è il resto intero della divisione

N/B. Per ottenere la penultima cifra della rappresentazione basterà quindi considerare il resto intero della divisione N_0/B. Infatti

$$\frac{N_0}{B} = (a_j \times B^{j-2}) + (a_{j-1} \times B^{j-3}) + \ldots + (a_2 \times B^0) + \frac{a_1}{B} = N_1 + \frac{a_1}{B}.$$

Procedendo in questo modo siamo in grado di costruire all'indietro tutte le cifre che compongono la rappresentazione del numero. Il processo chiaramente si arresterà alla cifra a_j per la quale $N_j = 0$. Ad esempio

$$\begin{aligned}
741 &= 2 \times 370 + 1, & a_0 &= 1\\
370 &= 2 \times 185 + 0, & a_1 &= 0\\
185 &= 2 \times 92 + 1, & a_2 &= 1\\
92 &= 2 \times 46 + 0, & a_3 &= 0\\
46 &= 2 \times 23 + 0, & a_4 &= 0\\
23 &= 2 \times 11 + 1, & a_5 &= 1\\
11 &= 2 \times 5 + 1, & a_6 &= 1\\
5 &= 2 \times 2 + 1, & a_7 &= 1\\
2 &= 2 \times 1 + 0, & a_8 &= 0\\
1 &= 2 \times 0 + 1, & a_9 &= 1.
\end{aligned}$$

ossia $741 = (a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0)_2 = (1011100101)_2$. Una funzione MATLAB che realizza il precedente algoritmo è la seguente

```
function a=convint(N,B)
% Sintassi a=convint(N,B)
% Funzione che converte un numero da base 10 a base qualunque
%
% a: vettore contenente le cifre del numero convertito
% N: numero intero da convertire
% B: base in cui deve essere convertito
%
a=zeros(1,100);
k=0;
while (N~=0) & (k <= 100)
   k=k+1;
   a(k)=rem(N,B);
   N=fix(N/B);
end
a=a(k:-1:1);
```

Nella funzione le cifre vengono memorizzate nel vettore $a(j)$, si noti che non essendo ammissibile usare l'indice 0 in MATLAB, avremo la relazione `a(k+1)`$= a_k$, $k = 1, \ldots, j$ tra gli elementi del vettore e le cifre che caratterizzano il numero. L'istruzione `fix` nella forma `fix(a)` con `a` numero reale restituisce il più grande numero intero n tale che $n \leq a$. Questa operazione come vedremo in seguito è chiamata arrotondamento per difetto. Avremo quindi che

```
>> convint(741,2)
ans =
     1     0     1     1     1     0     0     1     0     1
```

Possiamo utilizzare la precedente funzione per costruire una semplice tabella di conversione dei primi N numeri interi in base 2

```
% base2.m
% realizza una semplice tabelle di conversione dei primi N
% numeri interi da base 10 a base 2
%
```

```
N=15;
disp('-------------------');
fprintf('Base 10\t Base 2\n');
disp('-------------------');
for i=1:N
  a=convint(i,2);
  sa = num2str(a);
  fprintf('%2d \t %6s\n',i,sa');
end
```

Lo script fornisce il seguente risultato dal quale si vede chiaramente come 3, 7 e 15 siano i massimi numeri rappresentabili rispettivamente con due, tre e quattro cifre binarie

```
>> base2
-------------------
Base 10  Base 2
-------------------
 1             1
```

2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

In modo analogo è possibile costruire algoritmi per convertire un numero da base qualunque in base 10. Come per le usuali operazioni aritmetiche su interi in base 10 è possibile definire operazioni equivalenti che agiscono su interi rappresentati in una differente base.

Osservazione 2 Sostanzialmente tutti i moderni calcolatori utilizzano una rappresentazione dei numeri in base 2. Parole come *bit*, *byte* e *word* indicano numeri binari di diversa “lunghezza” (numero di cifre). Un bit, o *binary digit*, indica il singolo valore 1 o 0. Un byte è formato da un gruppo di 8 bit e una word invece dipende dal tipo di calcolatore utilizzato (quando si parla di calcolatori a 32 bit o a 64 bit, si fa riferimento alla lunghezza di una word). Da un punto di vista tecnico una word rappresenta la più piccola unità di memoria indirizzabile da un calcolatore e solitamente la lunghezza di una word è direttamente proporzionale alla velocità ed alla potenza di calcolo di un computer.
La maggior parte dei calcolatori utilizza 16 bit (2 byte) o 32 bit (4 byte) per memorizzare numeri interi. Questo indipendentemente dal numero di cifre necessarie a rappresentare il numero. In pratica anche se per rappresentare 741 abbiamo visto sono necessari 10 bit, il calcolatore né utilizzerà comunque 16 o 32 a seconda del tipo. Alla base di questo apparente spreco c’è la necessità di avere architetture semplici ed efficienti. In ogni caso un calcolatore che utilizza 16 bit per un intero avrà un limite per il valore intero massimo rappresentabile dato da $2^{16}-1=65535$. Esistono varie strategie per la memorizzazione di interi con segno. Un modo molto semplice è

basato sulla traslazione del precedente intervallo di valori rappresentabili con 16 bit da $[0, 2^{16} - 1] = [0, 65535]$ a $[-2^{15}, 215 - 1] = [-32768, 32767]$.

Numeri reali e numeri macchina

Abbiamo discusso la rappresentazione dei numeri interi in un calcolatore e abbiamo accennato al fatto di come, per questioni pratiche di progettazione, questa ponga dei limiti al più grande ed al più piccolo intero rappresentabile. Limitazioni ancora più forti si incontrano quando invece di numeri interi si considerano generici numeri reali. Innanzitutto non possiamo adottare direttamente la precedente strategia, in quanto i numeri binari non hanno parti frazionarie utilizzabili per rappresentare la parte decimale di un numero. Un problema aggiuntivo inoltre è dato dalla necessità di un numero infinito di cifre per rappresentare un generico numero reale.
L'idea di base è quella di memorizzare il numero tramite una rappresentazione binaria equivalente alla sua rappresentazione in notazione scientifica. Esistono molti diversi modi per rappresentare un numero in notazione scientifica, ad esempio

$$12,34 = 12.34 \times 10^0 = 1,234 \times 10^1 = 1234 \times 10^{-2} = \ldots$$

L'unica differenza è sostanzialmente nella posizione della virgola che delimita la parte decimale e nella potenza di 10 per cui il numero è moltiplicato. La rappresentazione precedente può essere applicata ad un qualunque numero reale x nella forma

$$x = \pm q \times 10^n,$$

dove il numero reale q è detto *mantissa* e l'intero n *esponente*.
In questo modo possiamo rappresentare tutti i numeri reali nella forma

$$x = \pm(0, d_1 d_2 \ldots d_{k-1} d_k \ldots)_{10} \times 10^n,$$

dove le cifre $d_1, d_2, \ldots, d_{k-1}, d_k, \ldots$ avranno valori in $\{0, 1, \ldots, 8, 9\}$ tali che

$$\begin{aligned} q &= (0, d_1 d_2 \ldots d_{k-1} d_k \ldots)_{10} \\ &= (d_1 \times 10^{-1}) + (d_2 \times 10^{-2}) + \ldots + (d_{k-1} \times 10^{-k+1}) + (d_k \times 10^{-k}) + \ldots \end{aligned}$$

In pratica il numero è caratterizzato

- dalla base utilizzata (10 per esempio);

- dalla mantissa q;

- dall'esponente n.

Se imponiamo che $1/10 \leq q < 1$, o equivalentemente che $d_1 \neq 0$, e se richiediamo che le cifre di q non siano definitivamente uguali a 9, allora sia q che l'esponente n risultano univocamente determinati.
Per esempio nel caso precedente avremo $12,34 = 0,1234 \times 10^2$ dove le parentesi $(\ldots)_{10}$ sono state omesse per semplicità.
Lo stesso tipo di rappresentazione potrà chiaramente essere adottato anche utilizzando basi B diverse da 10. Una rappresentazione equivalente in base B, con B intero positivo, del numero reale x sarà

$$\begin{aligned} x &= \pm q \times B^m, \\ q &= (0, a_1 a_2 \ldots a_{k-1} a_k \ldots)_B \qquad (2) \\ &= (a_1 \times B^{-1}) + (a_2 \times B^{-2}) + \ldots + (a_{k-1} \times B^{-k+1}) + (a_k \times B^{-k}) + \ldots \end{aligned}$$

dove le cifre $a_1, a_2 \ldots, a_{k-1}, a_k, \ldots$ appartengono all'insieme di valori $\{0, 1, \ldots, B-2, B-1\}$ ed m numero intero è l'esponente. La rappresentazione (3) è detta *rappresentazione in virgola mobile* o *floating point* in base B del numero reale x. Se imponiamo la condizione $a_1 \neq 0$ o equivalentemente $1/B \leq q \leq 1$ la rappresentazione è detta normalizzata. La codifica di $x = 0$ può variare (per esempio si assumono uguali a zero sia la mantissa q che l'esponente t).
Avremo quindi che in base 2 la rappresentazione in virgola mobile normalizzata di $12,34$ sarà

$$12,34 = (0,10001)_2 \times 2^6.$$

Infatti

$$\begin{aligned}(0,10001)_2 &= (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (0 \times 2^{-4}) + (1 \times 2^{-5}) \\ &= 0,5 + 0,313 = 0,5313\end{aligned}$$

da cui $0.5313 \times 2^6 = 12,34$. Tipicamente i calcolatori utilizzano una rappresentazione in virgola mobile normalizzata in base 2 dei numeri reali utilizzando un numero finito di cifre. Questo significa che non è il numero reale x che viene memorizzato nel calcolatore ma una sua approssimazione $\tilde{x}$ ottenuta utilizzando un numero finito k di cifre

$$\begin{aligned} x &\approx \tilde{x} = \pm\tilde{q} \times 2^t, \\ \tilde{q} &= (0, b_1 b_2 \dots b_{k-1} b_k)_2 \end{aligned}$$

Quindi un calcolatore utilizza solo un piccolo sottoinsieme dei numeri reali e ogni volta che deve rappresentare un numero che non appartiene a questo insieme dovrà sceglierne uno che lo approssimi all'interno dell'insieme a disposizione*. Tali numeri costituiscono un insieme finito e sono generalmente chiamati *numeri macchina* o *numeri floating point*.
Più in generale un insieme di numeri macchina in base B aventi un numero finito s di cifre sarà formato dai numeri reali x nella forma

$$x = \pm(0, a_1 a_2 \dots a_{s-1} a_s)_B \times B^m, \tag{3}$$

*Formalmente i numeri rappresentabili formano un sottoinsieme finito dell'insieme dei numeri razionali.

e sarà caratterizzato da un intervallo di valori interi $[L, U]$ per l'esponente m. Indichiamo con $F(B, s, L, U)$ tale insieme. È facile vedere che tale insieme è finito ed ha un valore assoluto massimo e un valore assoluto minimo ottenuti prendendo tutte le s cifre uguali a $B-1$ e l'esponente rispettivamente uguale ad U ed a L. In particolare si noti come la rappresentazione di $F(B, s, L, U)$ sulla retta reale non sarà continua.

Osservazione 3 I calcolatori attuali utilizzano 32 o 64 bit per rappresentare i numeri macchina. Tali rappresentazioni sono comunemente dette in *singola precisione* e *doppia precisione*. Un numero macchina in singola precisione utilizza 23 bit per la mantissa e 8 bit per l'esponente. Un numero in doppia precisione ha invece 53 bit per mantissa e 11 bit per l'esponente. In Tabella sono riportati approssimativamente gli intervalli di valori rappresentabili in singola e doppia precisione. Se in MATLAB proviamo a scrivere

```
>> realmax
ans =
  1.7977e+308
```

```
>> realmin
ans =
  2.2251e-308
```

otteniamo in risposta il più grande ed il più piccolo numero macchina diverso da zero rappresentabile in doppia precisione. In particolare ad un numero maggiore di `realmax` MATLAB associa lo speciale valore `Inf` (“infinito”).

```
>> 2*realmax
ans =
  Inf
```

Si noti che l'equivalente operazione prodotta su `realmin` fornisce un risultato inatteso

```
>> realmin/2
ans =
  1.1125e-308
```

ossia un numero più piccolo di `realmin`. MATLAB infatti cambia in questo caso il tipo di rappresentazione utilizzando parte dei bit normalmente utilizzati dalla mantissa per l'esponente. Tale rappresentazione non sarà però nella forma normalizzata utilizzata dagli altri valori ed è per questo detta *denormal*. Si noti che questo comporta una perdita di cifre significative rispetto agli altri numeri macchina. Se tale perdita supera un certo livello allora MATLAB restituisce semplicemente il valore 0 come in quest'ultimo esempio

```
>> realmin/1e+16
ans =
      0
```

Tipo	**Intervallo**
singola precisione	$-3.40 \times 10^{38} \leq x \leq -1.18 \times 10^{-38}$ 0 $1.18 \times 10^{-38} \leq x \leq 3.40 \times 10^{38}$
doppia precisione	$-1.80 \times 10^{308} \leq x \leq -2.23 \times 10^{-308}$ 0 $2.23 \times 10^{-308} \leq x \leq 1.80 \times 10^{308}$

Intervalli approssimati dei valori rappresentabili in singola e doppia precisione

Sbagliando si impara: analizziamo l'errore

Abbiamo visto alcuni tipi di errori che intervengono nell'esecuzione di algoritmi al calcolatore e dovuti al modo in cui i numeri reali vengono memorizzati: errori di *overflow*, *undeflow* e *arrotondamento*. Come vedremo esistono numerosi altri tipi di errore che intervengono nell'ambito del calcolo scientifico. Per esempio possono sorgere errori nell'esecuzione di operazioni aritmetiche utilizzando un numero finito di cifre, nell'utilizzo di funzioni polinomiali al posto di funzioni trigonometriche, approssimando di integrali tramite somme finite, utilizzando dati sperimentali poco accurati e via dicendo.

Errore assoluto e relativo

Consideriamo il problema da un punto di vista generale e studiamo la relazione tra una quantità approssimata $\widehat{x}$ e il suo corrispondente valore esatto x. La qualità della nostra approssimazione potrà essere misurata tramite l'*errore assoluto*

$$E_a(\widehat{x}) = |\widehat{x} - x|,$$

oppure tramite l'*errore relativo*

$$E_r(\widehat{x}) = \frac{|\widehat{x} - x|}{|x_{\text{ref}}|},$$

dove x_{ref} è una quantità di riferimento, tipicamente $x_{\text{ref}} = x$. Consideriamo il seguente esempio.

Esempio 15 (Calcolo approssimato del fattoriale di un numero) Dato un numero intero n il fattoriale di tale numero, indicato con $n!$ è definito ricorsivamente come

$$n! = \begin{cases} n(n-1)! & n \geq 1, \\ 1 & n = 0. \end{cases}$$

Un'approssimazione del fattoriale è data dalla seguente funzione

$$S(n) = \sqrt{2\pi n}\left(\frac{n}{\exp(1)}\right)^n.$$

detta approssimazione di Stirling.
Il seguente script MATLAB calcola l'errore assoluto e l'errore relativo a questa approssimazione per i primi 10 numeri interi

```
% fattoriale.m
%
```

```
% Tabella dell'errore assoluto e relativo per
% la formula di Stirling nel calcolo di n!
%
clc;
e=exp(1);
s='----------------------------------------------------------';
disp(s)
fprintf('n\t\t n!\t\t S(n)\t Ea\t\t Er\n')
disp(s)
nfact=1;
for n=1:10
   nfact = n*nfact;
   s = sqrt(2*pi*n)*((n/e)^n);
   Ea = abs(nfact - s);
   Er = Ea/nfact;
   fprintf('%2d\t %7d\t %10.2f\t %3.2e\t %3.2e\n',...
      n,nfact,s,Ea,Er);
end
```

Dove abbiamo utilizzato il comando MATLAB `clc` per cancellare il contenuto della finestra di comando e posizionare l'inizio del nuovo output nell'angolo in alto a sinistra dello schermo. Si noti anche l'uso di output con formato tramite il descrittore di formato unito ad un numero che specifica il numero di cifre in output in quel dato formato. Ad esempio `%2d` indicherà un intero a 2 cifre e `%10.2f` un numero macchina in virgola mobile a con 10 cifre di cui 2 decimali. Il risultato fornisce la seguente Tabella

```
----------------------------------------------------------------
n                n!               S(n)    Ea               Er
----------------------------------------------------------------
 1                1               0.92    7.79e-02         7.79e-02
 2                2               1.92    8.10e-02         4.05e-02
 3                6               5.84    1.64e-01         2.73e-02
 4               24              23.51    4.94e-01         2.06e-02
 5              120             118.02    1.98e+00         1.65e-02
 6              720             710.08    9.92e+00         1.38e-02
 7             5040            4980.40    5.96e+01         1.18e-02
 8            40320           39902.40    4.18e+02         1.04e-02
```

9	362880	359536.87	3.34e+03	9.21e-03
10	3628800	3598695.62	3.01e+04	8.30e-03

Si noti come l'errore assoluto aumenta all'aumentare di n a differenza dell'errore relativo che invece diminuisce. In particolare l'errore assoluto non fornisce indicazioni precise sulla qualità dell'approssimazione (per $n = 10$ l'errore assoluto è 300000 volte maggiore che per $n = 1$). Al contrario l'errore relativo risulta un buon indicatore della qualità dell'approssimazione in quanto non è influenzato dall'ordine di grandezza delle quantità che si stanno misurando, anche se non fornisce indicazioni sulla distanza effettiva che intercorre tra la quantità approssimata e quella esatta. In pratica è la conoscenza di entrambe gli errori che fornisce indicazioni precise sul comportamento di un'approssimazione numerica.

Arrotondamento e troncamento

Si consideri ora un generico numero reale positivo rappresentato in virgola mobile normalizzata in base B

$$x = 0, a_1 a_2 \dots a_{k-1} a_k \dots \times B^m.$$

e si indichi con $fl(x)$ il numero macchina più vicino ad x all'interno dell'insieme $F(B, s, L, U)$. In pratica $fl(x)$ può essere pensato come il valore che il calcolatore memorizza al posto di x. Quale è l'errore relativo che si commette approssimando x con $fl(x)$?
Se escludiamo la possibilità di overflow o undeflow, ossia assumiamo che $B^L < |x| < B^U$ avremo che

$$fl(x) = 0, \tilde{a}_1 \tilde{a}_2 \ldots \tilde{a}_{s-1} \tilde{a}_s \times B^m,$$

con $\tilde{a}_i = a_i$, $i = 1, \ldots, s-1$. L'*errore di arrotondamento* che commettiamo sarà determinato dalla scelta del valore $\tilde{a}_s$. Nel caso di approssimazione per troncamento avremo $\tilde{a}_s = a_s$, mentre nel caso di approssimazione per arrotondamento $\tilde{a}_s$ è ottenuta arrotondando il numero $a_s, a_{s+1} a_{s+2} \ldots$ all'intero più vicino.
Quindi la differenza tra il numero esatto e la sua rappresentazione sarà del tipo

$$x - fl(x) = 0,\, 0 \ldots 0\, d_s d_{s+1} \ldots \leq \frac{1}{2} B^m,$$

con $d_s < B/2$ nel caso di arrotondamento e $d_s < B$ nel caso di troncamento. Di conseguenza avremo

$$|x - fl(x)| < CB^{m-s},$$

con $C = 1/2$ per l'arrotondamento e $C = 1$ per il troncamento. Utilizzando infine la disuguaglianza $x > B^{m-1}$ otteniamo

$$\frac{|fl(x) - x|}{|x|} < CB^{1-s}.$$

La quantità CB^{1-s} è detta *precisione di macchina*. Va precisato che tutti i calcolatori moderni utilizzano l'arrotondamento nella rappresentazione di un numero reale. In base 2 quindi la precisione di macchina sarà $eps = 2^{-s}$. La costante MATLAB `eps` ha il valore della precisione di macchina utilizzata dallo stesso MATLAB. Ad esempio su un comune personal a 32 bit avremo

```
>> eps
eps =
   2.2204e-16
```

Una conseguenza della rappresentazione dei numeri reali con un numero finito di cifre sarà la costante presenza di errori di arrotondamento durante una qualunque operazione aritmetica.

Esempio **16 (Operazioni con $F(10,3,-9,9)$)** Consideriamo il caso di operazioni effettuate con l'insieme di numeri macchina $F(B,s,L,U) = F(10,3,-9,9)$. Ogni elemento di tale insieme avrà la forma

$$fl(x) = \pm 0, d_1 d_2 d_3 \times 10^m$$

con $-9 \leq m \leq 9$ e $1 \leq d_i \leq 9$, $d_1 \neq 0$. Si consideri l'operazione di somma di due numeri appartenenti all'insieme di numeri macchina considerato, ad esempio $x = 15,4$ ed $y = 3,55$ che saranno rappresentati esattamente come

$$x = fl(x) = 0,154 \times 10^2, \qquad y = fl(y) = 0,355 \times 10^1.$$

La somma esatta di tali numeri è $x + y = 18,95$ e non appartiene più all'insieme dei numeri macchina utilizzato in quanto le cifre significative necessarie a rappresentarlo sono 4 e non 3. Nel caso di arrotondamento avremo quindi il risultato approssimato

$$fl(x+y) = 0,19 \times 10^2.$$

■

Esempio 17 (Calcolo di radici di polinomi di secondo grado) La formula comunemente utilizzata per calcolare le radici del polinomio di secondo grado

$$p(x) = ax^2 + bx + c,$$

è data da

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{4}$$

A causa degli errori di arrotondamento l'utilizzo di questa formula al calcolatore può fornire risultati imprecisi. Ad esempio si consideri

$$a = 1, \qquad b = -54.32, \qquad c = 0.1,$$

che fornisce le radici (esatte per le prime 11 cifre)

$$x_+ = 54.3218158995, \qquad x_- = 0.0018410049576.$$

In questo caso $b^2 = 2950.7 \gg 4ac = 0.4$. Se utilizziamo l'insieme dei numeri macchina con quattro cifre ($s = 4$) per calcolare tali radici tramite la (4) otteniamo

$$\sqrt{b^2 - 4ac} = \sqrt{(-54.32)^2 - 0.4000} = \sqrt{2951 - 0.4000} = \sqrt{2951} = 54.32.$$

In altre parole dopo ogni operazione abbiamo arrotondato il risultato alle prime quattro cifre significative. Avremo quindi

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{54.32 + 54.32}{2.000} = \frac{108.6}{2.000} = 54.30.$$

Tale valore contiene un errore percentuale solo dello 0.04 che può essere considerato un buon risultato dato il numero limitato di cifre a disposizione. Se ora calcoliamo la seconda radice abbiamo

$$x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a} = \frac{54.32 - 54.32}{2.000} = \frac{0.000}{2.000} = 0,$$

che comporta un errore del 100 percento !. Utilizzando più cifre decimali l'effetto degli errori di arrotondamento nel calcolo di $\sqrt{b^2 - 4ac}$ vengono ridotti ma ciò nonostante non si eliminerebbe il problema dovuto alla perdita di accuratezza nella sottrazione di due numeri quasi uguali. Tale tipo di errore è comunemente chiamato *errore di cancellazione*.

Una possibile soluzione al problema è quella di costruire un algoritmo opportuno che minimizzi gli errori di cancellazione dovuti agli effetti dell'arrotondamento. L'algoritmo consigliato in questo caso è quello di calcolare prima

$$q = \frac{1}{2}\left(b + \mathrm{sign}(b)\sqrt{b^2 - 4ac}\right),$$

dove

$$\mathrm{sign}(b) = \begin{cases} 1 & \text{se } b > 0, \\ -1 & \text{se } b < 0, \end{cases}$$

e successivamente le radici tramite le relazioni

$$x_+ = \frac{q}{a}, \qquad x_- = \frac{c}{q}.$$

In questo caso il calcolo di x_- fornisce

$$x_- = \frac{2c}{-b + \sqrt{b^2 + 4ac}} = \frac{0.2000}{54.32 + 54.32} = \frac{0.2000}{108.6} = 0.001842,$$

che comporta un errore dello 0.05 percento analogo a quello del calcolo di x_+. Si noti infine che le due formulazioni considerate per il calcolo delle radici sono algebricamente equivalenti, la differenza nel risultato è dovuta solo agli effetti dell'arrotondamento. ▪

Esempio 18 (Calcolo dell'esponenziale come limite) Dall'analisi è noto il limite fondamentale

$$e = \exp(1) = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n.$$

Questo implica che all'aumentare di n il valore di

$$f(n) = \left(1+\frac{1}{n}\right)^n,$$

aumenta indefinitamente di accuratezza nell'approssimare il numero e. Possiamo costruire un semplice script MATLAB che calcola per diversi valori di n il valore di $f(n)$ e l'errore commesso nell'approssimazione di e

```
% approxexp.m
%
```

```
%
e=exp(1);
disp(sprintf('n\t f(n)\t\t Errore'));
for k=0:2:16
   n=10^k;
   fn=(1+1/n)^n;
   errore=abs(e-fn);
   fprintf('%2.1e\t %11.10f\t %e\n',n,fn,errore);
end
```

Lo script fornisce il seguente risultato

```
>> approxexp
n        f(n)            Errore
1.0e+00  2.0000000000    7.182818e-01
1.0e+02  2.7048138294    1.346800e-02
1.0e+04  2.7181459268    1.359016e-04
1.0e+06  2.7182804691    1.359363e-06
1.0e+08  2.7182817983    3.011169e-08
```

1.0e+10	2.7182820532	2.247757e-07
1.0e+12	2.7185234960	2.416676e-04
1.0e+14	2.7161100341	2.171794e-03
1.0e+16	1.0000000000	1.718282e+00

L'errore quindi diminuisce ed ha un minimo per $n = 10^8$ per poi aumentare al crescere di n. Si noti in particolare come l'errore nel calcolo di $f(n)$ è il risultato di solo tre operazioni matematiche, ossia $1/n$, $1+1/n$ e $(1+1/n)^n$. Se l'errore commesso nel calcolo di $1/n$ è piccolo in generale, altrettanto non si può dire per gli errori relativi introdotti dalle restanti due operazioni. Quindi è vero che accumulazione degli errori di arrotondamento in milioni di addizioni e sottrazioni può avere risultati disastrosi, ma come questo esempio mostra, errori catastrofici possono essere causati anche da poche semplici operazioni. ∎

Ordine di accuratezza

Se consideriamo le due successioni $\{1/n\}_{n\in\mathbf{N}}$ ed $\{1/n^2\}_{n\in\mathbf{N}}$ è chiaro che entrambe convergono a zero quando $n \to \infty$. La velocità di convergenza delle sue successioni sarà però differente, nel senso che la seconda successione andrà a zero più rapidamente della prima. In molte situazioni è importante avere una nozione precisa sulla velocità di convergenza di una successione di valori. A questo fine richiamiamo le seguenti definizioni

Definizione 1 *La funzione $f(x)$ è detta un* o-piccolo *della funzione $g(x)$ per x che tende a x_0 e denotata con $f(x) = o(g(x))$ se esiste una funzione $k(x) \geq 0$ tale che*

$$|f(x)| \leq k(x)|g(x)|, \qquad \lim_{x \to x_0} k(x) = 0.$$

In tal caso si dirà che $f(x)$ è trascurabile rispetto a $g(x)$ per x che tende a x_0.

In sostanza la precedente definizione significa che il rapporto f/g tende a zero (è un infinitesimo) per x che tende a x_0. La notazione $f(x) = o(1)$ indicherà che $f(x)$ è infinitesima per $x \to x_0$.

Avremo quindi $x^4 = o(x^2)$ per $x \to 0$ ed anche $x^4 + x^3 = o(x^2)$ per $x \to 0$.

Definizione 2 *La funzione $f(x)$ è detta un* O-grande *della funzione $g(x)$ per x che tende ad x_0 e denotata con $f(x) = O(g(x))$ se esiste una costante $C > 0$ tale che*

$$|f(x)| \leq C|g(x)|,$$

per x in un intorno di x_0.

In altre parole il rapporto tra le due funzioni f/g si mantiene limitato in un intorno di x_0. In particolare $f(x) = O(1)$ se e soltanto se la funzione $f(x)$ è limitata in un intorno di x_0.

Ad esempio date le funzioni $f(x) = x^3/(1+x)$ e $g(x) = x^2$, avremo per $x \to 0$

$$\frac{x^3}{1+x} = O(x^2),$$

poiché $x^3/(1+x) \leq x^3/x = x^2$ per $x \geq 0$. In pratica la notazione O-grande consente di descrivere il comportamento di una funzione in termini di funzioni elementari note (x^n, $x^{1/n}$, a^x, $\log_a x$, ecc.).

In modo simile si definisce

Definizione 3 *La successione* $\{x_n\}_{n\in\mathbf{N}}$ *è detta essere un O-grande della successione* $\{y_n\}_{n\in\mathbf{N}}$ *e denotata con* $x_n = O(y_n)$ *se esistono due costanti* C *e* N *tali che*

$$|x_n| \le C|y_n|, \qquad n \ge N.$$

Ad esempio la successione

$$\frac{n^2-1}{n^3} = O\left(\frac{1}{n}\right),$$

poiché $(n^2-1)/n^3 \le n^2/n^3 = 1/n$ per $n \ge 1$.

Si consideri ora un'approssimazione numerica $\widehat{f}_h(x)$ della funzione $f(x)$ tale che $\widehat{f}_h(x) \to f(x)$ per $h \to 0$. L'errore in $\widehat{f}_h(x)$ dipende dal punto x e da un parametro numerico h che caratterizza l'algoritmo usato per ottenere l'approssimazione. L'errore sarà caratterizzato da

$$E_h(x) = f(x) - \widehat{f}_h(x).$$

In generale $E_h(x)$ non sarà noto in quanto il valore di $f(x)$ non è noto (altrimenti non avremmo avuto la necessità di costruire una sua approssimazione).

Definizione 4 *La funzione $\hat{f}_h(x)$ è un'approssimazione di ordine $\alpha > 0$ di $f(x)$ se l'errore $E_h(x)$ soddisfa*

$$E_h(x) = O(h^\alpha), \quad \textit{per } h \textit{ sufficientemente piccolo}. \tag{5}$$

In tale caso

$$f(x) = \hat{f}_h(x) + O(h^\alpha).$$

Ad esempio se consideriamo l'espansione in serie di Taylor

$$e^x = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + \ldots$$

potremo scrivere (in questo caso $h = x$)

$$e^x \;=\; 1 + x + \frac{x^2}{2!} + O(x^3),$$

$$e^x \;=\; 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + O(x^{n+1}),$$

che caratterizzeranno rispettivamente approssimazioni di ordine 3 ed ordine n della funzione esponenziale.

Una procedura pratica per individuare l'ordine di accuratezza di una approssimazione nel caso in cui l'errore possa essere scritto come

$$E(h,x) = Ch^{\alpha} + o(h^{\alpha}), \tag{6}$$

è la seguente. Supponiamo di avere calcolato due valori approssimati $\widehat{f}_{h_1}(x)$ e $\widehat{f}_{h_2}(x)$ in corrispondenza di due diversi valori h_1 e h_2 di h.

Dalla precedente relazione (6) trascurando il termine $o(h^{\alpha})$ otteniamo

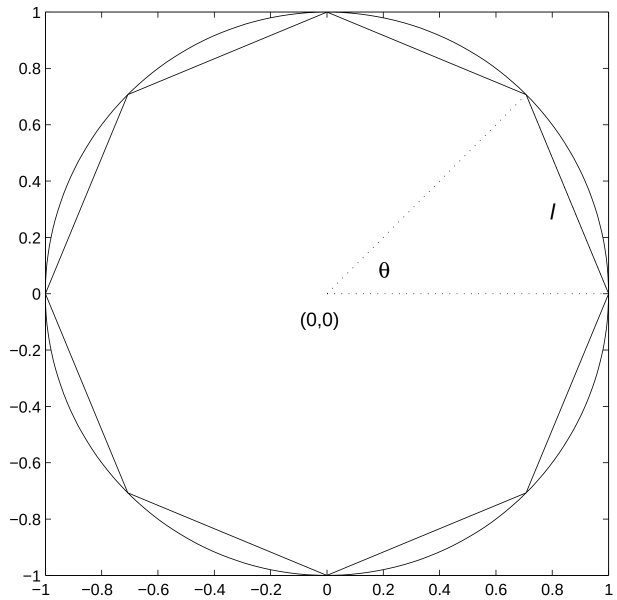
$$\frac{E_{h_1}(x)}{E_{h_2}(x)} \approx \left(\frac{h_1}{h_2}\right)^{\alpha},$$

e di conseguenza

$$\alpha \approx \frac{\log\left(E_{h_1}(x)/E_{h_2}(x)\right)}{\log\left(h_1/h_2\right)}.$$

In questo modo è possibile stimare α tramite un esperimento numerico.

Esempio **19 (Metodo di Archimede per il calcolo di π)** Il metodo di Archimede per il calcolo del numero π è basato sul calcolo del perimetro di una successione di poligoni inscritti in una circonferenza di diametro d assegnato.



Approssimazione di π tramite un poligono inscritto nella circonferenza per $n = 8$

Dato un poligono di n lati inscritto in una circonferenza di diametro d semplici considerazioni geometriche portano alle seguenti espressioni per l'angolo θ la cui corrispondente corda individua il lato l del poligono

$$\theta = \frac{2\pi}{n}, \qquad l = d \sin\left(\frac{1}{2}\theta\right).$$

Il perimetro del poligono sarà quindi $p_n = nl$ ossia

$$p_n = nd \sin\left(\frac{\pi}{n}\right).$$

In particolare avremo

$$\lim_{n\to\infty} p_n = d\pi.$$

Se utilizziamo p_n/d come approssimazione di π l'errore che commettiamo sarà

$$E_n = \pi - n \sin\left(\frac{\pi}{n}\right).$$

Chiaramente avremo $E_n \to 0$ per $n \to \infty$. Vogliamo calcolare con quale velocità $E_n \to 0$, ossia quale è l'ordine dell'approssimazione appena costruita. A questo scopo possiamo usare il semplice script MATLAB

```
% Archimede.m
%
%
fprintf(' n\t Errore\t\t alpha\n');
for k=2:6
   n=2^k;
   p=n*sin(pi/n);
   Errore1=abs(pi-p);
   fprintf('%2d\t %5.4e',n,Errore1);
   h1=1/n;
   if k > 2
      alpha=log(Errore1/Errore2)/log(h1/h2);
      fprintf('\t %8.5f\n',alpha);
   else
      fprintf('\n');
   end
   Errore2=Errore1;
   h2=h1;
end
```

Utilizzando la precedente funzione otteniamo

```
>> Archimede
 n   Errore        alpha
 4   3.1317e-01
 8   8.0125e-02   1.96660
16   2.0148e-02   1.99166
32   5.0442e-03   1.99791
64   1.2615e-03   1.99948
```

dalla quale si vede bene come $E_n = O(h^2) = O(1/n^2)$. ▪

Propagazione dell'errore

In questo paragrafo accenneremo brevemente a come l'errore si possa propagare per effetto di operazioni successive. Indichiamo con x ed y il valore esatto di due numeri e con $\widehat{x}$ ed $\widehat{y}$ due valori approssimati che contengono rispettivamente un errore ϵ_x ed ϵ_y

$$x = \widehat{x} + \epsilon_x, \qquad y = \widehat{y} + \epsilon_y.$$

Se consideriamo la somma avremo

$$x + y = (\widehat{x} + \epsilon_x) + (\widehat{y} + \epsilon_y) = (\widehat{x} + \widehat{y}) + (\epsilon_x + \epsilon_y).$$

Quindi nel caso dell'addizione l'errore nella somma è la somma degli errori negli addendi.

La propagazione dell'errore nel caso della moltiplicazione è più complessa. Infatti

$$xy = (\widehat{x} + \epsilon_x)(\widehat{y} + \epsilon_y) = \widehat{x}\widehat{y} + \widehat{x}\epsilon_y + \widehat{y}\epsilon_x + \epsilon_x\epsilon_y.$$

Quindi se i valori $\widehat{x}$ ed $\widehat{y}$ sono maggiori di uno in valore assoluto esiste la possibilità di amplificazione degli errori originali ϵ_x ed ϵ_y.

Se consideriamo l'errore relativo otteniamo dalla precedente relazione

$$\frac{xy - \widehat{x}\widehat{y}}{xy} = \frac{\widehat{x}\epsilon_y + \widehat{y}\epsilon_x + \epsilon_x\epsilon_y}{xy} = \frac{\widehat{x}\epsilon_y}{xy} + \frac{\widehat{y}\epsilon_x}{xy} + \frac{\epsilon_x\epsilon_y}{xy}.$$

Potremo supporre che $\widehat{x}$ ed $\widehat{y}$ siano buone approssimazioni nel senso che $\widehat{x}/x \approx 1$, $\widehat{y}/y \approx 1$ e $\epsilon_x \epsilon_y / xy \approx 0$. In queste ipotesi avremo che

$$\frac{xy - \widehat{x}\widehat{y}}{xy} \approx \frac{\epsilon_y}{y} + \frac{\epsilon_x}{x},$$

e di conseguenza l'errore relativo nel prodotto sarà approssimativamente dato dalla somma degli errori relativi in $\widehat{x}$ e $\widehat{y}$.

La precedente analisi dell'errore è detta *analisi dell'errore in avanti*. Tipicamente risulta molto difficile se non impossibile in situazioni realistiche dove si effettuano milioni di operazioni. Inoltre le ipotesi che si effettuano ad ogni passo per semplificare i calcoli conducono generalmente a stime molto imprecise sull'errore finale †. Un'approccio alternativo è dato dall'*analisi dell'errore all'indietro*. In questo caso si considera la soluzione approssimata calcolata come la soluzione esatta di un problema modificato. Ci si chiede poi quanto grande debba essere la modifica del problema originale

†si veda a titolo esemplificativo il famoso articolo di Goldstein e Von Neumann sull'analisi dell'errore in avanti per il metodo di eliminazione di Gauss, *Numerical inverting of Matrices of high order*, Bull. Amer. Math. Soc. 1947, pp.1021-1089.

per fornire tale soluzione. Più precisamente ci si chiede quale dovrà essere l'errore sui dati iniziali per produrre l'errore ottenuto nel risultato finale. In questo tipo di analisi il risultato finale sarà considerato accettabile se può essere interpretato come la soluzione esatta di un problema prossimo a quello originale.

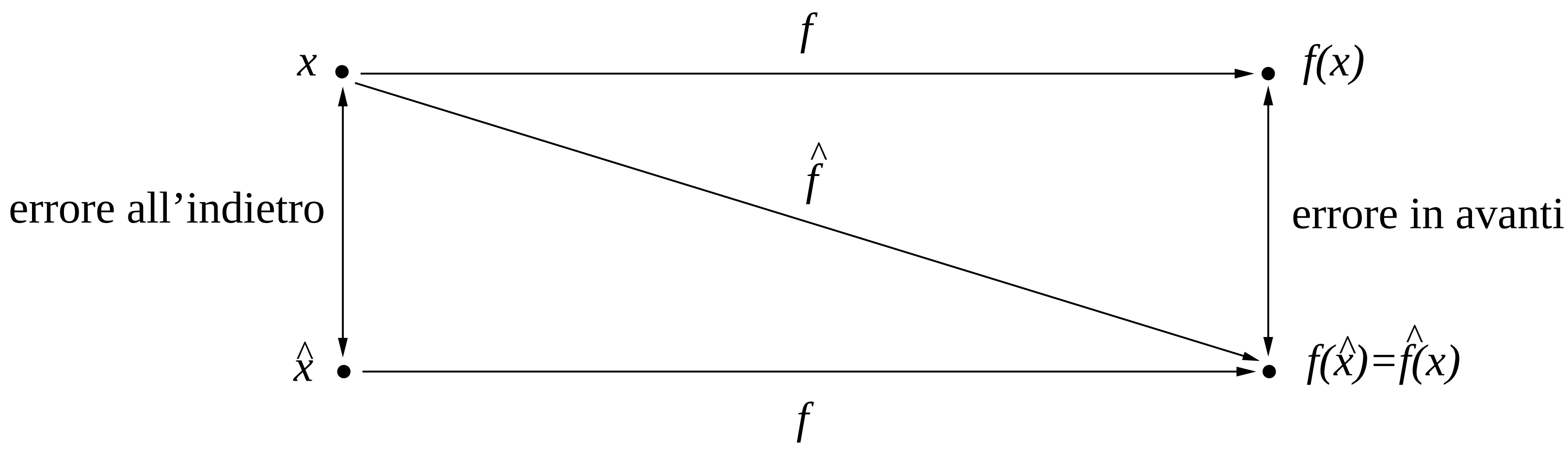


Illustrazione grafica dell'analisi dell'errore. I valori x e $f(x)$ indicano i dati esatti e la soluzione esatta del problema. La soluzione calcolata è $\widehat{f}(x)$. Il valore iniziale $\widehat{x}$ è tale che $f(\widehat{x}) = \widehat{f}(x)$.

Esempio **20 (Analisi dell'errore all'indietro)** Consideriamo il problema di analizzare la propagazione dell'errore nel caso dell'approssimazione

$$\widehat{f}(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!},$$

della funzione esponenziale $f(x) = e^x$. L'errore in avanti nel punto $x = 1$ calcolato con sette cifre decimali sarà dato da

$$f(x) - \widehat{f}(x) = e - \widehat{f}(1) = 2.718282 - 2.666667 = 0.051615.$$

Per determinare l'errore all'indietro nello stesso punto dobbiamo determinare il valore $\widehat{x}$ per la funzione $f(x)$ in modo tale che

$$f(\widehat{x}) = \widehat{f}(1)$$

ossia

$$e^{\widehat{x}} = 1 + 1 + \frac{1}{2} + \frac{1}{6}, \qquad \widehat{x} = \ln\left(\frac{8}{3}\right) = 0.980829.$$

L'errore all'indietro sarà quindi

$$x - \widehat{x} = 1 - 0.980829 = 0.019171.$$

Entrambi gli errori, in avanti ed all'indietro, forniscono una misura della bontà dell'approssimazione ottenuta. Non ha chiaramente senso confrontare numericamente i due diversi tipi di errore tra loro. Si noti inoltre come per valutare esattamente entrambi gli errori abbiamo utilizzato la conoscenza della soluzione esatta del problema cosa che in generale non sarà nota. Nella realtà si cercherà di ottenere una stima, accurata per quanto possibile, di tali errori. ▪

Spesso un errore iniziale viene propagato da una sequenza di calcoli. Una proprietà desiderabile in ogni metodo numerico è che un piccolo errore sui dati iniziali comporti in un piccolo errore nel risultato finale. Tipicamente un algoritmo con queste caratteristiche è detto *algoritmo stabile*. In caso contrario si parlerà di *algoritmo instabile*. Ad esempio, dal punto i vista dell'analisi all'indietro un algoritmo sarà stabile se il risultato prodotto è la soluzione esatta di un problema prossimo a quello originale. La stabilità di un algoritmo non garantirà però che la soluzione del problema sia anche accurata. L'accuratezza infatti si riferisce alla distanza che sussiste tra la soluzione calcolata e la soluzione vera del problema in esame. Esistono infatti problemi che sono particolarmente sensibili anche a piccole perturbazioni dei dati iniziali per i quali risulta intrinsicamente difficile il calcolo accurato di una soluzione. Tali problemi, per i quali la variazione relativa nella soluzione risulta molto maggiore della variazione nei dati iniziali, sono detti *problemi malcondizionati*. Più precisamente si può introdurre il seguente *numero di condizionamento* relativo al problema del calcolo di f nel punto x

$$\mu_f = \frac{|(f(\widehat{x}) - f(x))/f(x)|}{|(\widehat{x} - x)/x|},$$

dove $\widehat{x}$ è un punto vicino ad x. Per un problema malcondizionato tale numero sarà molto maggiore di uno.

In particolare se $\widehat{x} = x + h$ abbiamo per h sufficientemente piccolo

$$f(x + h) - f(x) \approx hf'(x).$$

Quindi l'errore relativo sarà

$$\frac{|f(x + h) - f(x)|}{|f(x)|} \approx \left| h\frac{f'(x)}{f(x)} \right|,$$

ed il numero di condizionamento

$$\mu_f = \frac{|hf'(x)/f(x)|}{|h/x|} \approx \left| x \frac{f'(x)}{f(x)} \right| .$$

Quindi l'errore assoluto e l'errore relativo possono essere molto maggiori o minori dell'errore sui dati iniziali a seconda delle proprietà della funzione f.

Esempio **21** **(Condizionamento nella valutazione di funzioni)** Nel caso in cui

$$f(x) = e^x,$$

abbiamo un errore assoluto he^x un errore relativo h e $\mu_f = |x|$. Il problema risulterà malcondizionato per valori grandi di x in valore assoluto.

Se consideriamo

$$f(x) = \cos(x),$$

avremo che l'errore assoluto sarà $h \sin(x)$ mentre quello relativo $h \tan(x)$ ed il numero di condizionamento $|x \tan(x)|$. Quindi piccole variazioni in

x vicino a $\pi/2$ forniranno grandi variazioni in $\cos(x)$ indipendentemente dall'algoritmo usato per il calcolo. Infatti

$$\cos(1.57079) = 0.63267949 \times 10^{-5},$$
$$\cos(1.57078) = 1.63267949 \times 10^{-5},$$

dove la variazione relativa della soluzione è 1.58 di fronte ad una variazione dei dati di 6.37×10^{-6}. ▪

Esempio 22 (Differenziazione numerica) In molte applicazioni risulta necessario valutare le derivate di una funzione. Se la funzione non è nota analiticamente ma solo per punti, oppure se il calcolo delle derivate risulta molto complesso allora si può utilizzare un calcolo approssimato delle derivate tramite *differenze finite*.

A questo scopo consideriamo la serie di Taylor di una funzione $f(x)$ nell'intorno del punto x

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(\xi),$$

con $\xi \in [x, x+h]$ ed h piccolo numero positivo detto *passo*.

Risolvendo la precedente equazione per $f'(x)$ possiamo scrivere

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h).$$

Per valori del passo h sempre più piccoli, la quantità

$$\frac{f(x+h) - f(x)}{h}, \tag{7}$$

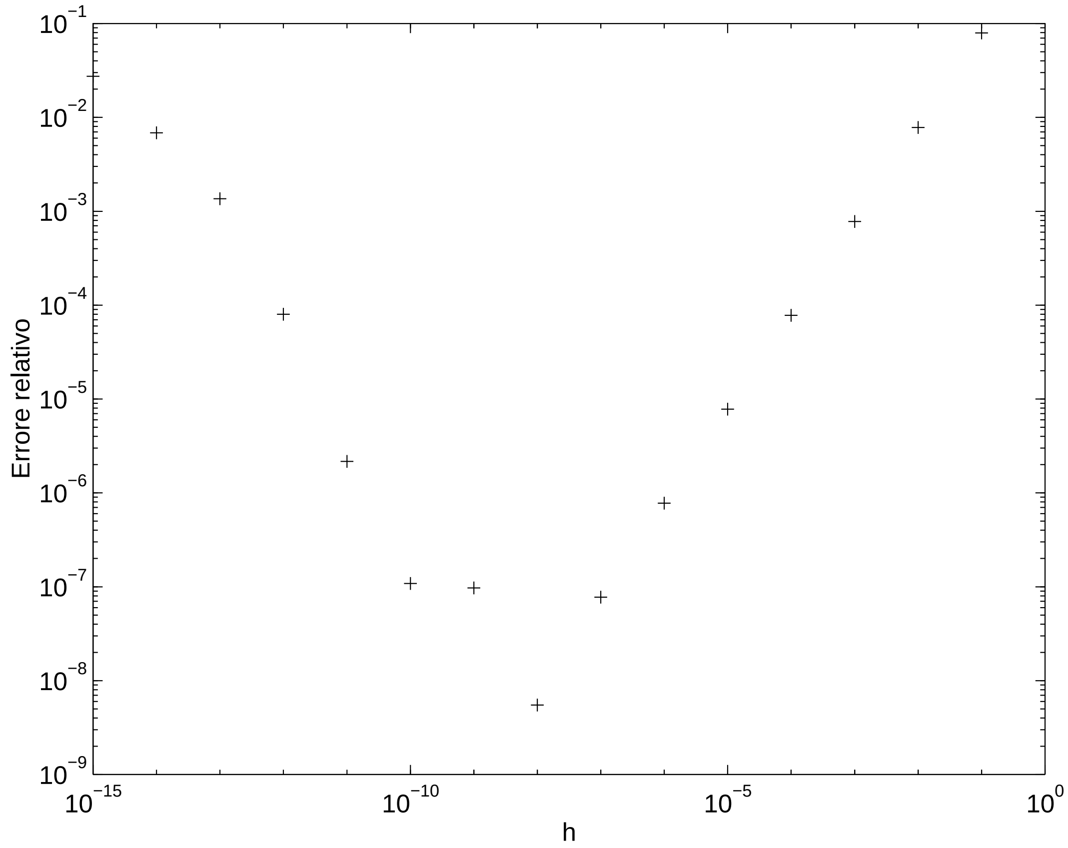
risulterà essere un'approssimazione sempre migliore di $f'(x)$.

Consideriamo quindi problema dell'approssimazione della derivata prima della funzione $f(x) = sin(x)$ tramite la relazione (7). Usando la seguente funzione

```
function sinfindiff(x)
% Sintassi sinfindiff(x)
%
h=10.^(-1:-1:-15);
Df=(sin(x+h)-sin(x))./h;
Errore=abs(Df-fp)/abs(cos(x));
```

```
loglog(h,Errore,'+');
xlabel('h');
ylabel('Errore relativo');
```

in $x = 1$ otteniamo il grafico riportato in Figura 22.



Errore relativo nell'approssimazione alle differenze finite della derivata prima di sin(x) in $x = 1$.

Abbiamo usato l'istruzione `loglog` che ha la stessa sintassi di `plot` per realizzare il grafico utilizzando una scala logaritmica sia sull'asse delle ascisse che su quello delle ordinate. Si consulti l'help anche per le funzioni analoghe `semilogx` e `semilogy`.

L'errore decresce linearmente come previsto per $10^{-8} < h < 1$ per poi crescere per valori di h minori di 10^{-8}. L'evidente perdita di accuratezza è dovuta agli errori di arrotondamento. Infatti il calcolatore non valuterà esattamente $f(x+h)$ ed $f(x)$ ma a numeratore calcolerà la differenza tra due quantità approssimate

$$\frac{\widehat{f}(x+h) - \widehat{f}(x)}{h} = \frac{(f(x+h)+\epsilon_1) - (f(x)+\epsilon_2)}{h} = \frac{f(x+h) - f(x)}{h} - \frac{\epsilon_1 - \epsilon_2}{h},$$

dove ϵ_1 ed ϵ_2 sono gli errori di arrotondamento.

Se supponiamo che ϵ_1 ed ϵ_2 siano minori della precisione di macchina `eps` potremo stimare l'errore assoluto che commettiamo tramite

$$\left|\frac{\widehat{f}(x+h)-\widehat{f}(x)}{h}-f'(x)\right| \leq \frac{2\mathtt{eps}}{h}+O(h).$$

Mentre il termine $O(h)$ converge linearmente a zero per $h \to 0$ il rapporto $2\mathtt{eps}/h$ tenderà a crescere indefinitamente. ∎

Riportiamo infine la seguente definizione

Definizione 5 *Supponiamo che ϵ_0 rappresenti un errore iniziale e ϵ_n l'errore dopo n passi di un certo algoritmo. Se*

$$\epsilon_n \approx n\epsilon_0,$$

la crescita dell'errore è detta lineare. Se

$$\epsilon_n \approx K^n \epsilon_0,$$

il comportamento dell'errore è detto esponenziale. In particolare se $K > 1$ l'errore cresce esponenzialmente ($\epsilon_n \to \infty$ per $n \to \infty$) mentre se $0 < K < 1$ l'errore decresce esponenzialmente ($\epsilon_n \to 0$ per $n \to \infty$).

Esercizi

Esercizio 1 Dato il vettore `z=50:-5:10` cosa restituiscono i seguenti comandi MATLAB ?

a) `length(z)`

b) `z'`

c) `z.*z(9:-1:1)`

d) `z(1:2:9)=ones(1,3)`

e) `z([3 1 7 5])=zeros(1,4)`

Quanti bytes utilizza MATLAB per memorizzare il vettore `z` ?

Esercizio 2 Per ciascuna funzione realizzare uno script MATLAB che disegna il grafico della funzione utilizzando un vettore con 101 elementi. Realizzare successivamente una tabella di 11 valori equispaziati della funzione utilizzando un sottovettore del precedente vettore.

a)

$$f(x) = \left(\frac{1 - x/24}{1 + x/24 + x^2/384} \right)^8, \qquad 0 \le x \le 1.$$

b)

$$f(x) = \begin{cases} (2 - x^2/2)^2, & x \in [-2, 0], \\ (x^2/2 + 2)^2, & x \in [0, 2]. \end{cases}$$

c)

$$f(x) = \begin{cases} \exp(-x^2), & -\pi \leq x \leq 0, \\ \cos(x), & 0 \leq x \leq \pi. \end{cases}$$

d)

$$f(x) = \begin{cases} \sqrt{1-(x-1)^2}, & 0 \leq x \leq 2, \\ \sqrt{1-(x-3)^2}, & 2 \leq x \leq 4, \\ \sqrt{1-(x-5)^2}, & 4 \leq x \leq 6, \\ \sqrt{1-(x-7)^2}, & 6 \leq x \leq 8. \end{cases}$$

Esercizio 3 Data la matrice `A=[1 2 3; 2 3 4; 3 4 5]`, cosa restituiscono i seguenti comandi MATLAB ?

a) `[p,q]=size(A)`

b) `A.*A'`

c) `A(1,2)=A(2,1)`

d) `A([3 1 2],2)=[1 2 3]`

e) `A(1,:)=A(2,:).*A(2,3:-1:1)`

Quanti bytes utilizza MATLAB per memorizzare la matrice `A` ?

Esercizio 4 Realizzare uno script MATLAB che disegna in nove sottofinestre nella stessa finestra grafica i grafici delle poligonali con $n = 3, 4, 5, 6, 8, 10, 12, 14,$ lati inscritte nella circonferenza $x^2 + y^2 = 1$.

Esercizio 5 Realizzare una funzione MATLAB chiamata `Ellisse(a,b,theta)` che, senza utilizzare nessun ciclo, produce il grafico dell'ellisse ruotata di un angolo θ secondo le equazioni

$$x(t) = \cos(\theta)\left[\frac{1}{2}(b-a) + \frac{1}{2}(a+b)\cos(t)\right] - \sin(\theta)\sqrt{ab}\sin(t)$$

$$y(t) \;=\; \sin(\theta)\left[\frac{1}{2}(b-a)+\frac{1}{2}(a+b)\cos(t)\right]+\cos(\theta)\sqrt{ab}\sin(t).$$

Esercizio 6 Realizzare il grafico di superficie e il grafico a curve di livello delle seguenti funzioni

a)

$$f(x,y)=\exp(-(x-1)^2-(y-1)^2)+\exp(-(x+1)^2-(y+1)^2), \quad (x,y)\in[-4,4]^2$$

b)

$$f(x,y)=\exp(-x^2-y^2)\sin(x^2+y^2), \quad (x,y)\in[-2,2]^2.$$

c)

$$f(x,y)=\frac{\sin(2\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}}, \quad (x,y)\in[-2,2]^2.$$

Esercizio 7 Scrivere i numeri interi 5, 21, 35 e 64 in base 2 e base 3.

Esercizio 8 *Convertire i numeri 0.4, 0.5, 1.5 in virgola mobile normalizzata in base 2.*

Esercizio 9 Quale è il più grande valore di n tale che $n!$ può essere rappresentato esattamente in $F(2, 24, -100, 100)$?

Esercizio 10 In aritmetica con quattro cifre significative eseguire la somma $x + y$ ed il prodotto xy dei seguenti numeri

$$x = 1.414, \quad y = 0.09125, \qquad\qquad x = 31.41, \quad y = 0.02718.$$

Esercizio 11 Si considerino le seguenti approssimazioni delle funzioni $\sin(x)$ e $\cos(x)$

$$s(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!}, \qquad c(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!}.$$

Realizzare uno script MATLAB che calcola gli errori relativi e assoluti delle precedenti approssimazioni per $x = -2, -2, 0, 1, 2$. Determinare empiricamente l'ordine di accuratezza delle approssimazioni.

Esercizio 12 Discutere la propagazione dell'errore in avanti nel caso della somma $x + y + z$ e del prodotto xyz di tre numeri $x = \widehat{x} + \epsilon_x$, $y = \widehat{y} + \epsilon_y$ e $z = \widehat{z} + \epsilon_z$.

Esercizio 13 Realizzare una funzione MATLAB che calcola il valore del polinomio $p(x) = (x - 1)^6$ utilizzando la formula

$$p(x) = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1,$$

e ne realizza il grafico in $[1 - \delta, 1 + \delta]$ per $\delta = .1, .01, .008, .007, .005, .003$. Cosa succede al diminuire di δ ? Spiegare il comportamento osservato.

Esercizio 14 Realizzare una funzione MATLAB che calcola il valore del seno iperbolico sinh(x) tramite la relazione

$$\text{sinh}(x) = \frac{e^x - e^{-x}}{2}.$$

La si confronti poi con la funzione MATLAB `sinh(x)` (che assumiamo come valore esatto del seno iperbolico) e si realizzi il grafico dell'errore assoluto e relativo per `x=10.^(-12:12)`. Quale è la causa di errore per valori piccoli di x ?

Esercizio 15 La costante γ di Eulero è definita come

$$\lim_{n\to\infty} \gamma_n, \qquad \gamma_n = \left[1 + \frac{1}{2} + \frac{1}{4} + \ldots \frac{1}{n} - \ln(n)\right].$$

Realizzare uno script MATLAB che esegue il calcolo della successione γ_n per `n=10.^(0:2:16)`. Discutere i risultati ottenuti dopo averli visualizzati graficamente.