METODI ANALITICI E NUMERICI PER L'INGEGNERIA
Corso di Laurea in Ingegneria Meccanica
Prof. S. Perotto

A.A. 2024 – 2025
Politecnico di Milano
Dr. N. Ferro, E. Temellini

Introduzione a `Matlab`

1 L'ambiente Matlab

Matlab® è un software per il calcolo scientifico, particolarmente sviluppato per quanto riguarda la gestione ed elaborazione di vettori e matrici, e adatto a trattare numerosi problemi di interesse in applicazioni scientifiche e ingegneristiche. Ulteriori pacchetti mettono a disposizione dell'utente metodi per l'analisi statistica, il machine learning, tecniche per l'imaging e molto altro ancora. Matlab® è un software commerciale a pagamento; maggiori informazioni si possono trovare all'indirizzo `http://www.mathworks.it/`.
GNU Octave è un linguaggio ad alto livello, principalmente pensato per il calcolo scientifico. Octave non possiede una interfaccia grafica di default come Matlab® , ma è possibile installarla separatamente. Senza interfaccia l'inseriremento dei comandi avviene da linea di comando. Octave è essenzialmente compatibile con Matlab® , salvo per alcune piccole differenze che saranno affrontate durante il corso. È un software gratuito che è possibile scaricare all'indirizzo `http://www.gnu.org/software/octave/`.

1.1 Nozioni di base ed Help in linea

Matlab® è un linguaggio interpretato, ovvero ogni linea di un programma Matlab® viene letta, interpretata ed eseguita sul momento.
Matlab® ha un'interfaccia grafica interattiva e una linea di comando (prompt `>>`) sulla quale si possono scrivere dei comandi. Ad esempio, per uscire da Matlab® , scrivere:

```
>> quit
```

Matlab® offre un help in linea molto completo. Comporre

```
>> help comando
```

per avere una spiegazione sul modo di utilizzo di un comando Matlab® .

```
>> lookfor argomento
```

per avere una lista di comandi Matlab® inerenti un certo argomento

```
>> helpwin
```

per aprire un help interattivo dei comandi matlab disponibili, classificati per argomenti.

1.2 Interfaccia Grafica di Matlab®

È costituita principalmente da quattro ambienti. Nel **workspace** sono rappresentate tutte le varibili memorizzate, il loro valore e tipo. La finestra **current directory** rappresenta una finestra sulla cartella in cui si sta lavorando e mostra tutti i file presenti nella cartella stessa. La **command history** contiene lo storico di tutti i comandi digitati. L'ambiente principale è la **command window** in cui vengono inseriti i comandi. Per quanto riguarda Octave , a meno che non si installi una interfaccia grafica, c'è solo la command window.

Matlab® è l'acronimo di **Mat**rix **Lab**oratory, per cui tutte le variabili in Matlab® sono considerate matrici. In particolare, gli scalari sono considerati matrici 1×1, i vettori riga sono matrici $1 \times n$, i vettori colonna sono matrici $n \times 1$, dove n è la lunghezza del vettore. Octave si comporta nello stesso modo.

2 Dichiarazione di variabili

Matlab® permette di creare e inizializzare variabili molto facilmente. La dichiarazione di variabili in Matlab® segue le seguenti regole:

- tutte le variabili sono *matrici*;
- non si dichiara il *tipo* di variabile.

`>>`	`a=5`	*variabile scalare*	(1×1)
`>>`	`b=[4  6]`	*vettore riga*	(1×2)
`>>`	`c=[-5; 2]`	*vettore colonna*	(2×1)
`>>`	`d=[2 3; -1 7]`	*matrice quadrata*	(2×2)

2.1 Assegnazione di scalari

Cominciamo con l'assegnare il valore 2.45 alla variabile `a`:

```
>> a = 2.45
 a  =
    2.45
```

assegniamo ora il valore 3.1 alla variabile `A`. Osserviamo che Matlab® fa distinzione tra le lettere maiuscole e le lettere minuscole.

```
>> A = 3.1
 A  =
    3.1
```

Le variabili sono sovrascrivibili, cioè se ora assegniamo ad `A` un nuovo valore:

```
>> A = 7.2
 A  =
    7.2
```

il precedente valore 3.1 viene definitivamente perso. Osserviamo che possiamo far seguire un comando da una virgola, senza rilevare nessuna differenza; tale virgola è però necessaria per separare più comandi scritti sulla stessa linea.

```
>> a = 1.2,
 a  =
    1.2
>> a = 1.7, a = 2.45
 a  =
    1.7
 a  =
    2.45
```

Se invece si fa seguire il comando da un punto e virgola, Matlab® non visualizzerà sulla finestra di comando il risultato dell'operazione; il punto e virgola può essere usato per separare due comandi sulla stessa riga.

```
>> a = 1.2;
>> a = 1.7; a = 2.45
 a  =
    2.45
```

Per sapere quali sono le variabili dell'utente attualmente in memoria si utilizza il comando:

```
>> who
 Your variables are:
  A   a
```

Per sapere quali sono le variabili in memoria definite dall'utente è anche possibile utilizzare il comando `whos`. Quest'ultimo, a differenza di `who`, mostra anche la dimensione, l'occupazione di memoria in numero di bytes e il tipo della variabile.

```
>> whos
  Name      Size             Bytes  Class     Attributes

  A         1x1                  8  double
  a         1x1                  8  double
```

Le variabili possono essere cancellate utilizzando il comando `clear`. Possiamo ad esempio cancellare la variabile `A` digitando:

```
>> clear A
```

o tutte le variabili con il comando clear:

```
>> clear
```

Per ripulire la finestra grafica dalle istruzioni precedenti è possibile usare il comando

```
>> clc
```

oppure

```
>> home
```

La seguente tabella riassume le principali funzioni di gestione dell'ambiente e la loro azione:

comando	azione
`clear`	Cancella tutte le variabili
`clear var`	Cancella la variabile `var`
`clc`	Cancella tutte le istruzioni a schermo e blocca la barra di scorrimento
`home`	Cancella tutte le istruzioni a schermo
`help istruzione`	Fornisce le funzionalità e le modalità d'uso di `istruzione`
`who`	Elenca le variabili in memoria
`whos`	Elenca le variabili, il loro tipo e la dimensione di memoria occupata
`size (a)`	Fornisce le dimensioni di `a`

Variabili predefinite. Alcune variabili, proprie di Matlab® , non necessitano di alcuna definizione. Tra queste ricordiamo:

- `>> pi`: il numero $\pi = 3.141592653589793\ldots$;
- `>> i, j`: l'unità immaginaria $\sqrt{-1}$ (entrambe le varabili predefinite la contengono);
- `>> exp(1)`: il numero di Nepero $e = 2.718281828459046\ldots$

Attenzione! le variabili predefinite possono essere ridefinite, ovvero:

```
>> a = 5 + 2 * i
 a  =
    5.0000 + 2.0000i
>> i = 2
 i  =
    2
>> a = 5 + 2 * i
 a =
    9
```

2.2 Istruzione `format`

I numeri reali sono visualizzati con solo quattro cifre decimali; tuttavia, la rappresentazione interna contiene sempre 16 cifre significative.
Il comando `format` permette di modificare il formato di visualizzazione dei risultati ma NON la precisione con cui i calcoli vengono condotti. Il comando ha la sintassi

```
>> format tipo
>> 1/7
```

e produce i risultati elencati nella seguente tabella, in base al `tipo` usato

`tipo`	
`rat`	1/7
`short`	0.1429
`short e`	$1.4286\text{e}-01$
`short g`	0.142856
`long`	0.142857142857143
`long e`	$1.428571428571428\text{e}-01$
`long g`	0.142857142857143

Il comando `format`, senza ulteriori specifiche, seleziona automaticamente il formato più conveniente per la classe delle variabili in uso. In Octave il comando `format` si usa esattamente nello stesso modo; è pero possibile che fornisca risultati leggermente diversi da quelli di Matlab® .

2.3 Assegnazione di vettori

Riportiamo alcuni metodi per l'inserimento di vettori.

- Si possono costruire vettori riga elencando gli elementi separati da uno spazio o da una virgola, come nella tabella seguente

`comando`	risultato
`>> b = [ 1 5 9 4 ]`	b =(1, 5, 9, 4)
`>> c = [ 1, 5, 9, 4 ]`	c =(1, 5, 9, 4)

- Per costruire dei vettori riga con elementi equispaziati, il comando generico è

  ```
  >> b = [ primo elemento : passo: ultimo elemento ]
  ```

  Se non indicato, il passo di incremento è pari a 1.

  Alcuni esempi sono raccolti nella tabella sottostante: [1]

`comando`	risultato
`>> d = [ 1 :  1 :  4 ]`	d = (1, 2, 3, 4)
`>> g = [ 1 :  4 ]`	g = (1, 2, 3, 4)
`>> r = [ 1 :  0.1 :  2 ]`	r = (1, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2)
`>> s = [ 0 :  -0.3 :  -1 ]`	s = (0, - 0.3, - 0.6, - 0.9)

  Si può utilizzare anche il comando `linspace` per definire un vettore riga di elementi equispaziati. Al comando devono essere forniti come parametri di ingresso i due estremi dell'intervallo e il numero di elementi del vettore.

  ```
  >> p = linspace( primo elemento,  ultimo elemento, numero elementi)
  ```

[1] Le parentesi quadre sono state indicate per chiarezza e uniformità con i comandi presentati di seguito. In questo specifco caso, non sono tuttavia necessarie.

Nell'esempio il vettore `p` è costituito da 6 elementi equispaziati nell'intervallo $[0, 5]$:

```
>> p = linspace( 0, 5, 6)
 p  =
    0    1    2    3    4    5
```

- I vettori colonna vengono costruiti elencando gli elementi separati dal punto e virgola, come nell'esempio:

```
>> q = [ 1; 2; 3; 4 ]
  q  =
    1
    2
    3
    4
```

2.4 Assegnazione di matrici

È buone norma utilizzare per il nome delle variabili per le matrici le lettere maiuscole. Combinando i comandi visti per definire i vettori riga e colonna si ottengono immediatamente le istruzioni per definire le matrici, per esempio:

```
>> H = [ 1 2 3 ; 2 4 7 ; 1 4 3 ]
  H  =
    1    2    3
    2    4    7
    1    4    3
```

3 Operazioni su vettori e su matrici

- **Trasposizione** L'operazione di trasposizione si realizza aggiungendo un apice al nome della variabile:

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

  Naturalmente, applicando l'operazione di trasposizione a tutte le modalità viste per definire dei vettori riga, si ottengono dei vettori colonna.

- **Operazioni algebriche** Le principali operazioni algebriche tra matrici e vettori sono elencate nella tabella sottostante. Dati un vettore di n componenti e una matrice $n \times n$,

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

operazione	azione
`a + b`	Somma di due vettori della stessa dimensione
`H + G`	Somma di due matrici della stessa dimensione
`a - b`	Sottrazione di due vettori della stessa dimensione
`H - G`	Sottrazione di due matrici della stessa dimensione
`H * G`	Prodotto algebrico righe per colonne. Il risultato è una matrice $n \times n$
`G * a'`	Prodotto algebrico righe per colonne. Il risultato è un vettore $n \times 1$
`a * G`	Prodotto algebrico righe per colonne. Il risultato è un vettore $1 \times n$
`a' * b`	Prodotto algebrico righe per colonne. Il risultato è una matrice $n \times n$
`a * b'`	Prodotto algebrico righe per colonne. Il risultato è uno scalare
`3 * G`	Prodotto di uno scalare per una matrice
`3 * b`	Prodotto di uno scalare per un vettore
`cross(a,b)`	Prodotto vettoriale. `a` e `b` devono necessariamente avere solo tre elementi

Osserviamo che nel prodotto righe per colonne le dimensioni dei vettori o delle matrici utilizzate devono essere compatibili.

- **Operazioni puntate (o elemento per elemento)** In Matlab® sono definite anche delle operazioni particolari che agiscono sulle matrici e sui vettori elemento per elemento.
  - Somma o sottrazione di un valore scalare ad ogni elemento di una matrice (vettore). La matrice (vettore) risultante ha le stesse dimensioni della matrice (vettore) di partenza.

```
>> f = [ 1 2 3 ; 2 4 6 ; 3 6 9 ];
>> l = f + 3
  l =
    4    5    6
    5    7    9
    6    9    12
```

  - Prodotto elemento per elemento di due matrici (vettori). Date due matrici (vettori) il loro prodotto elemento per elemento è una matrice (vettore) delle stesse dimensioni e i cui elementi sono il prodotto degli elementi di posto corrispondente nelle matrici (vettori) iniziali. (Attenzione alla compatibilità: in questo caso le due matrici (vettori) devono avere le stesse dimensioni!). Per esempio

```
>> b = [2 5 3 2];
>> c = [1 2 4 2];
>> f = b .* c
 f  =
    2  10  12  4
```

    mentre se i due vettori non hanno le stesse dimensioni si ottiene un messaggio di errore, come nel seguente esempio

```
>> b = [2 5 3 2];
>> a = [ 3 5 6];
>> f = a .* b
    ??? Error using ==> times
    Matrix dimensions must agree
```

  - Divisione elemento per elemento di due matrici (vettori). Date due matrici (vettori) la loro divisione elemento per elemento è una matrice (vettore) delle stesse dimensioni e i cui elementi sono il rapporto degli elementi di posto corrispondente nelle matrici (vettori) iniziali.

Per esempio

```
>> b = [2 5 3 2];
>> c = [1 2 4 2];
>> format rat
>> f = b ./ c
 f  =
    2  5/2  3/4  1
```

– Elevamento a potenza di una matrice (vettore) elemento per elemento. La matrice (vettore) risultante ha le stesse dimensioni della matrice (vettore) di partenza e i suoi elementi sono ottenuti elevando alla potenza richiesta gli elementi della matrice (vettore) di partenza. Per esempio

```
>> a = [1 2 3 4];
>> b = a.^2
  b =
    1    4    9    16
```

Si osservi che tutte le operazioni elemento per elemento sono contraddistite dal "." che precede il simbolo dell'operazione, con l'eccezione dell'operazione di somma e sottrazione.

Nella tabella sottostante sono riassunte le operazioni "elemento per elemento". Dati

```
>> a = [ 1 3 5 6 ];
>> b = [ 4 6 4 8 ];
```

`operazione`	azione
`b + 3`	Somma lo scalare 3 a tutti gli elementi di `b`
`a - 3`	Sottrae lo scalare 3 a tutti gli elementi di `a`
`a .* b`	Moltiplica gli elementi di `a` e `b` di posto corrispondente
`a ./ b`	Divide ogni elemento di `a` per il corrispondente elemento di `b`
`a .^ n`	Eleva alla potenza `n` tutti gli elementi di `a`

Sottolineiamo ancora una volta la differenza tra le consuete operazioni algebriche e quelle effettuate elemento per elemento. Nel caso di due matrici (di dimensioni compatibili) si ha:

`>> C = A + B`	somma tra matrici, $C_{ij} = A_{ij} + B_{ij}$
`>> C = A * B`	prodotto tra matrici, $C_{ij} = \sum_k A_{ik} B_{kj}$
`>> C = A^3`	elevamento a potenza di una matrice (`C = A*A*A`)

`>> C = A .* B`	prodotto elemento per elemento, $C_{ij} = A_{ij} B_{ij}$
`>> C = A ./ B`	divisione elemento per elemento, $C_{ij} = \frac{A_{ij}}{B_{ij}}$
`>> C = A.^3`	elevamento a potenza, $C_{ij} = A_{ij}^3$

4 Funzioni matematiche elementari

Le funzioni matematiche elementari restituiscono matrici o vettori della stessa dimensione della variabile cui è applicata la funzione. Data una matrice `A`:

funzione	azione
abs(A)	Valore assoluto degli elementi di A
sqrt(A)	Radice quadrata degli elementi di A
exp(A)	Funzione esponenziale di ogni elemento di A
log(A)	Logaritmo naturale di ogni elemento di A
log10(A)	Logaritmo in base 10 di ogni elemento di A
log2(A)	Logaritmo in base 2 di ogni elemento di A
sin(A)	Seno di ogni elemento di A (con A in radianti)
cos(A)	Coseno di ogni elemento di A (con A in radianti)
tan(A)	Tangente di ogni elemento di A (con A in radianti)
asin(A)	Arco seno di ogni elemento di A (in radianti)
acos(A)	Arco coseno di ogni elemento di A (in radianti)
atan(A)	Arco tangente di ogni elemento di A (in radianti)
sinh(A)	Seno iperbolico di ogni elemento di A
cosh(A)	Coseno iperbolico di ogni elemento di A
tanh(A)	Tangente iperbolica di ogni elemento di A

4.1 Funzioni per la gestione di vettori e matrici

Nella tabella successiva è riportato un elenco delle piu importanti funzioni Matlab® e Octave per la gestione di vettori e matrici, con alcuni esempi di applicazione. Alcuni comandi sono indifferentemente utilizzabili sia per i vettori che per le matrici. In caso contrario è esplicitamente dichiarato. Data una matrice A e un vettore b: [2]

funzione	Azione
size(A)	Restituisce un vettore di due elementi, il primo è il numero di righe di A, il secondo il numero di colonne di A
size(A,1)	Restituisce il primo elemento di size(A), cioè il numero di righe di A
size(A,2)	Restituisce il secondo elemento di size(A), cioè il numero di colonne di A
length(b)	Restituisce la lunghezza del vettore b (equivale a max(size(b)))
max(b)	Restituisce il più grande elemento di b
min(b)	Restituisce il più piccolo elemento di b
max(A)	Restituisce un vettore riga contenente il massimo di ogni colonna di A
min(A)	Restituisce un vettore riga contenente il minimo di ogni colonna di A
sum(b)	Restituisce uno scalare pari alla somma di tutti gli elementi di b
sum(A)	Restituisce un vettore riga i cui elementi sono la somma degli elementi di colonna di A
diag(A)	Estrae la diagonale principale di A
diag(A,1)	Estrae la sopradiagonale di ordine 1 di A
diag(A,-1)	Estrae la sottodiagonale di ordine 1 di A
diag(A,k)	Estrae la sopra/sottodiagonale di ordine k di A; se la dimensione di A è n, k può variare da -n+1 a n-1
diag(b)	Costruisce una matrice quadrata diagonale con gli elementi di b sulla diagonale principale
diag(b,k)	Costruisce una matrice quadrata con gli elementi di b sulla sopra/sottodiagonale di ordine k.
tril(A)	Crea una matrice triangolare inferiore con elementi coincidenti con i corrispondenti elementi di A
triu(A)	Crea una matrice triangolare superiore con elementi coincidenti con i corrispondenti elementi di A

[2] In versioni precedenti di Matlab® , l'applicazione di length(A) ad una matrice potrebbe dare risultati diversi: in generale, il suo utilizzo è consigliabile solo con vettori.

funzione	Azione
A(1,1)	Estrae l'elemento di posto (1,1) di A
A(:,1)	Estrae la prima colonna di A
A(1,:)	Estrae la prima riga di A
A(1:3,:)	Estrae le prime tre righe di A
A([1,3],:)	Estrae la prima e la terza riga di A
A(1:3,1:3)	Estrae la sottomatrice costituita dalle prime tre righe e tre colonne di A
A(2:4,[1,3])	Estrae la sottomatrice costituita dalle righe 2, 3, 4 e dalle colonne 1, 3 di A

Il comando `diag(M,k)`, dove `M` è una matrice, costruisce un vettore colonna avente per elementi gli elementi della diagonale `k`-esima di `M`, dove `k` $= 0$ indica la diagonale principale, `k` > 0 indica la `k`-esima sopradiagonale mentre `k` < 0 indica la `k`-esima sottodiagonale; se `k` viene omesso, si suppone `k`=0.
Osserviamo infine che il comando `diag(v,k)`, dove `v` è un vettore, costruisce una matrice di dimensione (`length(v)+k`)x(`length(v)+k`) con gli elementi di `v` sulla diagonale `k`-esima e gli altri elementi nulli.

Merita una descrizione a parte, la funzione `norm`. Essa ha due parametri in input: un vettore `v` e un numero intero `n` oppure `inf`. Se viene passato solo il vettore, calcola la norma 2 (*norma euclidea*) di `v`, definita come

$$||\mathtt{v}||_2 = \sqrt{\sum_{i=1}^{\mathtt{length(v)}} \mathtt{v}_i^2}.$$

Se viene passato un numero intero `n`, calcola la norma n di `v` definita come

$$||\mathtt{v}||_n = \left(\sum_{i=1}^{\mathtt{length(v)}} |\mathtt{v}_i|^n \right)^{\frac{1}{n}}.$$

Infine se viene passato come secondo argomento `inf`, calcola la norma infinito di `v` definita come

$$||\mathtt{v}||_\infty = \max_{1 \le i \le \mathtt{length(v)}} |\mathtt{v}_i|.$$

operazione	azione
norm(v)	Calcola la norma euclidea (norma 2) di v
norm(v,1)	Calcola la norma 1 di v
norm(v,2)	Calcola la norma euclidea (norma 2) di v
norm(v,inf)	Calcola la norma infinito di v

4.2 Funzioni per definire vettori o matrici particolari

Dati `Nrighe` e `Ncolonne` il numero di righe e il numero di colonne della variabile vettoriale che vogliamo costruire, la tabella seguente riassume i comandi per la costruzione di matrici particolari:

funzione	azione
zeros(Nrighe, Ncolonne)	Costruisce una matrice (vettore) di tutti 0
zeros(Nrighe)	Costruisce una matrice quadrata di tutti 0
ones(Nrighe, Ncolonne)	Costruisce una matrice (vettore) di tutti 1
ones(Nrighe)	Costruisce una matrice quadrata di tutti 1
eye(Nrighe)	Costruisce una matrice quadrata con elementi
eye(Nrighe, Ncolonne)	Costruisce una matrice di dimensioni fornite come sottomatrice della matrice identità
rand(Nrighe, Ncolonne)	Costruisce una matrice (vettore) di elementi casuali compresi tra 0 ed 1
rand(Nrighe)	Costruisce una matrice quadrata di elementi casuali compresi tra 0 ed 1

4.3 Ulteriori operazioni matriciali

Sia `A` una matrice; Matlab® permette di effettuare numerose operazioni su una matrice, tra le quali ricordiamo le più ricorrenti:

`>> C = A'`	trasposta di `A`, $C_{ij} = A_{ji}$
`>> C = inv(A)`	inversa di `A` (matrici quadrate), $C = A^{-1}$. **Warning**: L'utilizzo del comando **inv** comporta il calcolo esplicito della matrice inversa. Questo algoritmo è particolarmente oneroso, quindi deve essere utilizzato solo se necessario.
`>> d = det(A)`	determinante di `A` (matrici quadrate)
`>> r = rank(A)`	rango di `A`
`>> nrm = norm(A)`	norma 2 di `A`
`>> cnd = cond(A)`	numero di condizionamento (in norma 2) di `A`
`>> v = eig(A)`	autovalori (e autovettori) di `A` (matrici quadrate)

Soluzione di sistemi lineari di ridotta dimensione.
Sia `A` una matrice quadrata di dimensione $n \times n$ e `b` un vettore di dimensione n, allora il vettore `x`, soluzione del sistema lineare `Ax = b` può essere calcolato mediante l'istruzione

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

Warning: se si è interessati soltanto alla soluzione `x` del sistema e non al calcolo dell'inversa `inv(A)`, l'istruzione precedente non è ottimale. È preferibile, invece, utilizzare il comando *backslash*

```
>> x = A \ b
```

che risolve il sistema lineare con algoritmi altamente efficienti.

5 Script

La scrittura di programmi da linea di comando è pratica perché immediata ed interattiva, ma è comoda solo quando il numero di istruzioni è limitato. Nel caso si voglia scrivere codice lungo e soprattutto si voglia avere la possibiltà di riutilizzare in futuro quanto già scritto, è utile salvare i comandi in un file di *script*. Matlab® ed Octave utilizzano dei semplici file di testo non formattato con estensione `.m` per memorizzare le istruzioni; file di questo tipo sono detti *M–files*. A questo scopo apriamo l'*editor*[3] di Matlab® o Octave utilizzando il comando `edit`. Dopo aver aperto l'editor, si scrivono i comandi come se fossero nel prompt dei comandi:

Listing 1: Esempio di script

```
x = [0:0.01:1];
f = @(x) sin (x) .* cos (x);
plot (x, f (x), 'b.-');
grid on;
f (0.2)
```

Salviamo il file con il nome `test_plot.m`. Il file verrà salvato nella directory di lavoro corrente, il cui percorso è mostrato nella barra degli strumenti, oppure tramite il comando `pwd`. Tale directory può essere cambiata utilizzando il comando `cd` (si digiti `help cd` per maggiori chiarimenti) oppure tramite l'apposita casella disponibile nell'interfaccia grafica (sia per Matlab® che per QtOctave).
Si possono eseguire i comandi contenuti nel file scrivendo semplicemente il nome del file dalla riga di comando [4] oppure cliccando sul pulsante `Run` situato nella barra degli strumenti dell'editor (identificato dall'icona a forma di triangolo verde).

[3] Matlab® dispone di un proprio editor, mentre Octave può aprire un editor presente nel sistema oppure anch'esso un editor dedicato, se si utilizza un'interfaccia grafica come *QtOctave*. L'editor di Matlab® o QtOctave può essere aperto anche cliccando sulla apposita icona nella barra in alto. In realtà può essere utilizzato qualsiasi editor che possa salvare testo non formattato: ne esistono tantissimi e molti sanno riconoscere la sintassi di Matlab® e Octave.

[4] Il file deve trovarsi nella cartella in cui stiamo lavorando

```
>> test_plot
ans =
    0.1947
```

All'interno dello script è possibile utilizzare variabili già presenti in memoria. Inoltre, le variabili create (o modificate) durante l'esecuzione dell'M–file vengono assegnate nello spazio di lavoro, come se le istruzioni fossero state battute nel prompt dei comandi.
Notiamo infine che è possibile inserire *commenti* all'interno degli script Matlab® utilizzando il simbolo `%`:

Listing 2: Esempio di commenti in Matlab®

```
% un esempio di commento
a = 2;

b = a+1 % un altro commento. I caratteri dopo il '%' vengono ignorati
```

I commenti possono occupare sia un'intera riga sia la parte finale di una riga. In qualunque caso, i caratteri che seguono il simbolo `%` saranno ignorati da Matlab® quando lo script sarà eseguito.

6 Definizione di funzioni

Matlab® e Octave mettono a disposizione una serie di funzioni matematiche "standard" (`sin`, `cos`, `exp`, ...), ma consentono all'utente anche di definire le proprie funzioni. In altre parole, è possibile scrivere un frammento di codice che prenda in ingresso una variabile x e restituisca il valore $f(x)$, ad esempio $f(x) = x^4 + 3\log(x)$, e che venga eseguito tramite il comando `nome_funzione(x)` [5].
Vengono forniti quattro diversi modi per definire una funzione:

1. tramite il comando `eval`;
2. tramite il comando `inline`;
3. tramite `anonymous functions`, con il comando `@`;
4. tramite `.m file`;

6.1 Il comando `eval`

Supponiamo di voler implementare la funzione che restituisca il valore $f(x) = x^3 - 1$. Un modo di procedere è scrivere le operazioni che devono essere eseguite all'interno di una stringa di caratteri che chiamiamo `cubica` (notare l'uso di `.^`):

```
>> cubica='x.^3-1'
```

poi dichiarare il vettore dei valori di `x` su cui vogliamo valutare $f(x)$:

```
>> x=[0:1:3];
```

ed infine il comando `eval` ci permette di valutare la funzione memorizzata nella stringa `cubica` in corrispondenza dei valori contenuti nel vettore `x`:

```
>> eval(cubica)
ans =
    -1     0     7    26
```

È importante osservare che in questo caso la variabile utilizzata per definire la stringa `cubica` deve necessariamente avere lo stesso nome del vettore di punti che vogliamo valutare, cioè `x`. In caso contrario si otterrà un errore, come nel seguente esempio:

[5] è buona norma di programmazione dare nomi significativi alle funzioni!

```
>> clear x
>> t=[0:1:3]
t =
     0     1     2     3
>> eval(cubica)
??? Error using ==> eval
Undefined function or variable 'x'.
```

6.2 Il comando `inline`

Il secondo metodo per definire una funzione utilizza il comando `inline`. La sintassi è:

```
>> x=[0:1:3];
>> cubica = inline('x.^3-1','x');
>> cubica(x)
ans =
    -1     0     7    26
```

Il comando `inline` quindi prende in ingresso una stringa contenente la definizione della funzione e una stringa contenente il nome della variabile in ingresso a tale funzione. Va notato che `cubica` adesso non è una stringa, come nel caso di `eval`, ma un oggetto di tipo `function`:

```
>> cubica = inline('x.^3-1','x')
cubica =
     Inline function:
     cubica(x) = x.^3-1
```

e quindi sono consentite valutazioni della funzione anche su variabili che non si chiamino `x`:

```
>> t=[0:1:4];
>> cubica(t)
ans =
    -1     0     7    26    63
```

Tramite il comando `inline` è anche facile definire funzioni di più variabili:

```
>> cubicaR2 = inline('x.^3+y.^3','x','y');
>> cubicaR2(2,4)
ans =
    72
```

6.3 Anonymous functions (`@`)

Questo terzo modo può essere pensato come una "contrazione" del comando `inline` [6] . La sintassi è la seguente:

```
>> cubica = @(x) x.^3-1
cubica =
    @(x) x.^3-1
>> cubica(x)
ans =
    -1     0     7    26
```

[6] In realtà i due meccanismi sono diversi: le `anonymous function` usano il concetto di `function handle @`, che non approfondiremo in questo corso.

Dopo il carattere speciale `@` si indica fra parentesi la variabile in ingresso, e poi si scrive l'operazione che deve essere eseguita. Anche in questo caso è possibile valutare `cubica` su variabili che non si chiamano `x`, ed è facile dichiarare funzioni di più variabili:

```
>> t = [4 5];
>> cubica(t)
ans =
    63   124

>> cubicaR2 = @(x,y) x.^3+y.^3
cubicaR2 =
    @(x,y) x.^3+y.^3
>> cubicaR2(2,4)
ans =
    72
```

Le funzioni anonime, introdotte nella versione 7 (R14) di Matlab® , permettono di trattare funzioni in maniera molto naturale, senza bisogno di utilizzare un comando dedicato alla valutazione come `eval` o `feval`. In generale la sintassi è

```
f = @(lista argomenti) espressione;
```

e per valutare la `f` basta scrivere `f(argomento)`. Lo stesso vale per funzioni di più variabili, ad esempio `f = @(x,p) log(x) - p;`. Infine, le funzioni anonime permettono di definire funzioni a partire da comandi Matlab® *qualsiasi*, incluse le *function* definite da un m-file. Ad esempio , la funzione

```
rho = @(A) max(abs(eig(A)));
```

calcola il raggio spettrale di una matrice `A`.

6.4 Definizione di funzioni tramite `.m` files

Come abbiamo già visto, in Matlab® è possibile eseguire degli script chiamando nel prompt dei comandi il nome di un `.m` file presente nella cartella di lavoro. Ad esempio, una volta raccolta una serie di comandi nel file `miofile.m`, digitando nel prompt l'istruzione

```
>> miofile
```

Matlab® eseguirà fedelmente tutti i comandi presenti nel file. Le caratteristiche degli script sono quelle di non avere parametri in ingresso modificabili e di poter lavorare solo su variabili globali presenti in memoria.

Una funzione Matlab® risponde esattamente a queste limitazioni, 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 al termine della stessa. A differenza di uno script standard, la sintassi di una funzione richiede che la prima riga del file abbia la struttura

```
function [Output1,Output2,...] = nomefunzione(Input1,Input2,...)
```

dove le parentesi quadre possono essere omesse se l'output è uno solo. È buona norma inserire alcune righe di commento (ovvero precedute dal carattere `%`) dopo la definizione, che possono essere visualizzate tramite il comando

```
>> help nomefunzione
```

Successivamente saranno inserite tutte le istruzioni necessarie all'esecuzione della funzione. Il file dovrà essere salvato con il nome della funzione e con l'estensione `.m` e dovrà essere visibile a Matlab® nell'area di lavoro (ovvero deve essere posizionato nella cartella corrente).

Nel seguente file, che chiameremo `det2.m`, si implementa una funzione che calcola il determinante di una matrice quadrata di dimensione 2:

Listing 3: Script `det2.m`

```
function [det]=det2(A)
%DET2 calcola il determinante di una matrice quadrata di ordine 2
[n,m]=size(A);
if n==m
    if n==2
        det=A(1,1)*A(2,2)-A(2,1)*A(1,2);
    else
        error('Solo matrici 2x2');
    end
else
    error('Solo matrici quadrate');
end
return
```

dove si utilizza la funzione `error('...')` per interrompere l'esecuzione del programma e visualizzare una stringa di errore nel prompt dei comandi.

Funzioni più flessibili Una funzione Matlab® può modificare il suo comportamento a seconda del numero di variabili in input o output che riceve: per questo scopo, le funzioni `nargin` e `nargout` contano rispettivamente il numero di parametri di input (output) che sono stati passati alla funzione. Ad esempio, la semplice funzione:

```
function c = test_nargin(a, b)
if (nargin == 1)
    c = a .^ 2;
elseif (nargin == 2)
    c = a + b;
end
```

modifica il suo comportamento a seconda che le venga passato un solo parametro in input (ne calcola il quadrato) o due (ne calcola la somma). Per differenziare il comportamento della funzione è standard l'uso della struttura `if() ... elseif()... end`. Ad esempio, la semplice funzione:

```
function [out1, out2] = test_nargout(a, b)
if (nargout == 1)
    out1 = a * b;
elseif (nargout == 2)
    out1 = a + b;
    out2 = a - b;
end
```

differenzia il suo comportamento a seconda del numero di parametri di output che richiediamo. Infatti:

```
>> a = test_nargout(3,4)
a =
    12
>> [b,c] = test_nargout(3,4)
b =
     7
c =
    -1
```

7 Output su monitor

La funzione `disp` permette di visualizzare una stringa di caratteri (o, più in generale, una matrice di stringhe), racchiusa tra apici. Ad esempio potremmo scrivere:

```
>> disp('Ciao a tutti')
Ciao a tutti
>> disp('Oggi e'' martedi'' ')   % doppio apice per visualizzarne uno
Oggi e' martedi'
```

In molte circostanze si ha la necessità di rappresentare come stringa valori di varibili numeriche. La prima possibilità è quella di convertire una variabile numerica in stringa tramite la funzione `num2str`:

```
>> x = 10;
>> stringa = ['La variabile x vale : ', num2str(x)];
>> disp(stringa)
La variabile x vale : 10
```

mentre un modo molto più versatile (e consigliato) consiste nell'utilizzare le funzioni `fprintf` e `sprintf`. La sintassi della prima è:

```
fprintf('Formato' , Variabili)
```

e scrive su video il valore delle `Variabili` indicate, utilizzando il `Formato` definito, che altro non è che una stringa che contiene i caratteri che si vogliono visualizzare e, nelle posizioni in cui si vuole venga inserito il valore delle `Variabili`, deve essere indicato uno dei formati di visualizzazione preceduti dal carattere `%` . I formati di visualizzazione sono riassunti nella seguente tabella:

Codice di formato	Azione
`%s`	stringa
`%d`	numero intero
`%f`	numero in virgola fissa (es. 1234.5678)
`%e`	numero notazione scientifica (es. $1.2345678e+003$)
`%g`	numero in notazione compatta (sceglie tra `%f` e `%e`)
`\n`	inserisce carattere ritorno a capo
`\t`	inserisce carattere di tabulazione

Ad esempio:

```
>> x = 10; y = 5.5;
>> fprintf('x vale %d mentre y vale %f \n' , x, y);
x vale 10 mentre y vale 5.500000
```

8 Disegnare grafici di funzione

8.1 il comando `plot`

Per disegnare il grafico di una funzione reale di variabile reale si utilizza il comando `plot`.
La sintassi di `plot` prevede che si diano in ingresso al comando:

- due vettori `x` e `y` (delle stesse dimensioni) contenenti le ascisse e le ordinate dei punti del grafico (il grafico viene disegnato unendo tali punti con dei segmenti).
- i comandi opzionali con cui specificare le caratteristiche del grafico (tipo di linea, colore, spessore, colore dello sfondo...)

Nel nostro caso, il comando:

```
>> x=[0:0.01:3];
>> plot(x,cubica(x))
```

produce il grafico in Figura 1.

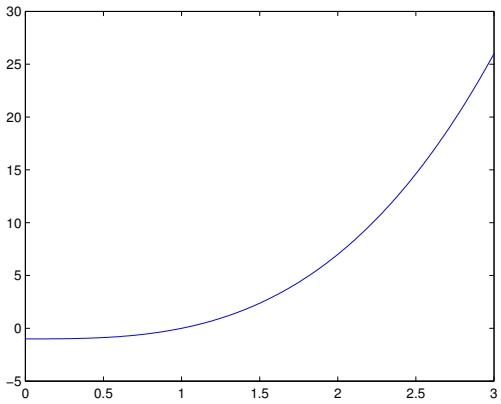


Figure 1: $x^3 - 1$ per $0 \leq x \leq 3$

Un ulteriore comando `plot` disegna il nuovo grafico sulla stessa finestra, cancellando quello precedente. Per modificare questo comportamento, così che i nuovi grafici siano sulla stessa figura, si usa il comando `hold on`. Si torna allo stato precedente con `hold off` mentre `hold` da solo cambia tra i due stati.
Ad esempio, i comandi per disegnare sullo stesso grafico la funzione $f(x) = 3 - x^2$ sono:

```
>> parabola = @(x) 3-x.^2;
>> hold on;
>> plot(x,parabola(x))
```

Per migliorare la leggibilità dei grafici sono utili i comandi di stile grafico: ad esempio, si può far disegnare il secondo grafico in rosso (Figura 2). Per aprire una nuova finestra grafica si usa il comando `figure`.

```
>> figure;
>> plot(x,cubica(x));
>> hold on;
>> plot(x,parabola(x),'r')
```

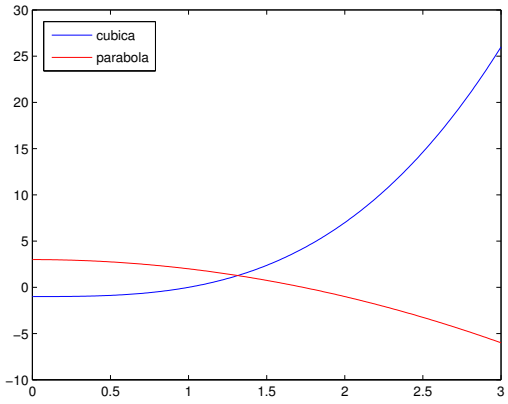


Figure 2: grafico di $x^3 - 1$ e $3 - x^2$ per $0 \leq x \leq 3$

Digitando `help plot` si ottiene una panoramica sulla formattazione che si può assegnare ad un grafico. Ad esempio questi sono i comandi per definire colore, stile della linea e stile dei punti:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
		v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

Ad esempio, volendo disegnare il grafico della funzione

$$f(x) = 2 + (x - 3)\sin(5(x - 3))$$

per $0 \leq x \leq 6$, sovrapponendo a questo grafico quelli delle due rette ($y = -x + 5$ e $y = x - 1$) che limitano l'andamento di tale funzione, disegnate con linea tratteggiata, si possono usare i seguenti comandi:

```
>> x=[0:0.01:6];
>> f=@(x) 2+(x-3).*sin(5*(x-3));
>> r1=@(x) -x+5;
>> r2=@(x) x-1;
>> plot(x,f(x),'k');
>> hold on
>> plot(x,r1(x),'--k')
>> plot(x,r2(x),'--k')
```

e ottenere il grafico riportato di seguito.

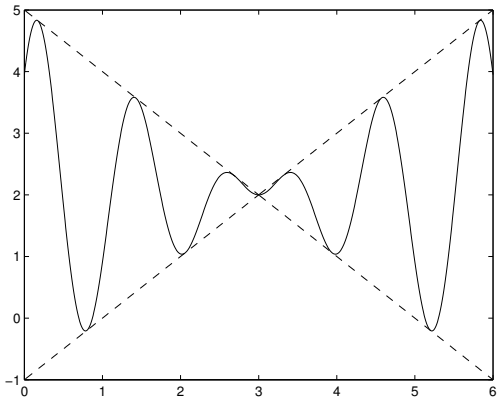


Figure 3: grafico di $f(x) = 2 + (x-3)\sin(5(x-3))$ e delle rette $y = -x + 5$, $y = x - 1$

8.2 Grafici in scala logaritmica

In molte aree scientifiche vengono usati grafici in scala logaritmica/semilogaritmica. Matlab® e Octave forniscono a tale proposito i comandi `semilogy`, `semilogx` e `loglog`, che sono l'equivalente di `plot` ma tracciano un grafico rispettivamente con l'asse delle ordinate in scala logaritmica, con l'asse delle ascisse logaritmico ed entrambi in scala logaritmica.
Supponiamo di voler disegnare in scala y-logaritmica sull'intervallo $0 \leq x \leq 10$ il grafico delle funzioni $y = e^x$ e $y = e^{2\,x}$. I comandi necessari sono (stavolta disegnamo il secondo grafico in verde, con linea continua, indicando i punti con degli asterischi, vedi figura 4):

```
>> x=[0:0.1:10];
>> semilogy(x,exp(x))
>> hold on;
>> semilogy(x,exp(2*x),'-*g')
```

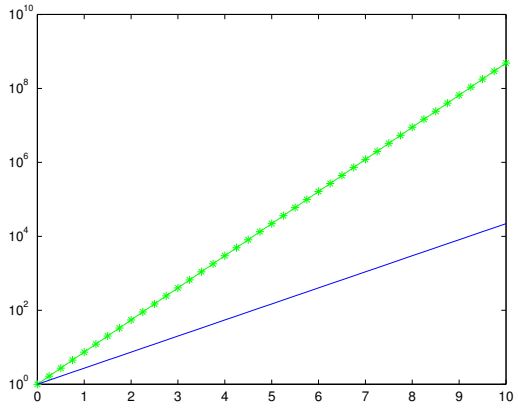


Figure 4: grafico in scala y-logaritmica di $y = e^x$ e $y = e^{2x}$

I grafici ottenuti sono delle rette, dal momento che stiamo tracciando $\log(y) = log(e^x) = x$, cioè una retta. La funzione $y = e^{2x}$ risulta essere una retta con pendenza doppia, poiché $\log(e^{2\,x}) = 2\,x$.

Possiamo aggiungere molti dettagli al disegno (vedi figura 5):

- la griglia:

  ```
  >> grid on
  ```

  Il comando `grid on` traccia la griglia, `grid off` la rimuove mentre `grid` da solo cambia tra i due stati.

- il titolo del grafico:

  ```
  >> title('Grafico di exp(x) e di exp(2x)')
  ```

- i titoli degli assi:

  ```
  >> xlabel('Scala lineare')
  >> ylabel('Scala logaritmica')
  ```

- la legenda:

  ```
  legend('exp(x)','exp(2*x)','Location','NorthWest')
  ```

  Il comando `legend` attribuisce le stringhe di testo che gli sono passate ai grafici disegnati da plot, nello stesso ordine (prima stringa con il primo grafico, seconda stringa con il secondo grafico etc.). Alcune particolari stringhe di testo, come `'Location'` che abbiamo appena utilizzato, sono interpretate da `legend` come comandi che permettono di modificare aspetto e posizione della legenda. Ad esempio `'Location','NorthWest'` indica di porre la legenda in alto a sinistra (*NordOvest* in una carta geografica con il nord in alto). È possibile anche modificare a mano la legenda utilizzando il mouse nella finestra del grafico (menù `insert`).

N.B. le etichette degli assi riportano sempre il valore della variabile, non il suo logaritmo: ad esempio, per `semilogy`, l'asse delle ordinate indica i valori di y e **non** di $log(y)$.

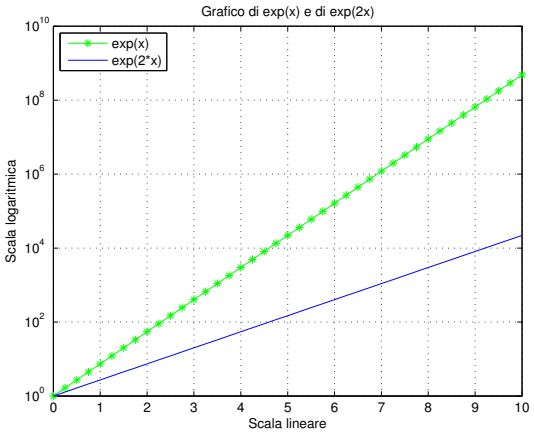


Figure 5: grafico in scala y-logaritmica di $y = e^x$ e $y = e^{2x}$, con titoli e legenda

9 Istruzioni di controllo

Matlab® offre, come altri linguaggi di programmazione, alcuni cicli di controllo e istruzioni condizionali.

Ciclo `for`

Se si vuole eseguire delle istruzioni in sequenza per ciascun valore

$$i = m,\ m+1,\ \ldots,\ n$$

di una variabile i tra i limiti m e n, si può utilizzare l'istruzione `for`. Ad esempio, per calcolare il prodotto scalare `ps` tra due vettori `x` e `y` di dimensione `n` si può utilizzare:

```
>> ps = 0;
>> for i = 1:n;
>>    ps = ps + x(i)*y(i);
>> end;
```

Questo ciclo è equivalente al prodotto matriciale $x^T y$ (anche se computazionalmente molto meno efficiente); supponendo che i vettori siano stati definiti come vettori colonna:

```
>> ps = x'*y;
```

Ciclo `while`

Se si vuole eseguire un'istruzione fintantoché una certa espressione logica è vera, si utilizza l'istruzione `while`. Ad esempio, lo stesso calcolo eseguito precedentemente con un ciclo `for` può essere eseguito in modo analogo con:

```
>> ps = 0;
>> i = 0;
>> while (i < n);
>>    i = i + 1;
>>    ps = ps + x(i)*y(i);
>> end;
```

Istruzione condizionale `if` / `if-else`

Se si vuole eseguire un'istruzione soltanto se una certa espressione logica è vera, si utilizza `if`. Ad esempio, se si vuole calcolare la radice quadrata di una variabile scalare `r` soltanto se essa è non negativa :

```
>> if (r >= 0)
>>   radice = sqrt(r);
>> end;
```

Se si vuole che una diversa istruzione venga eseguita se l'espressione logica è falsa, si utilizza il costrutto `if-else`. Ad esempio:

```
>> if (r >= 0)
>>   radice = sqrt(r);
>> else
>>   disp('r ha valore negativo');
>> end;
```

Gli *operatori logici* a disposizione sono:

Operatore	Azione logica
`&`	*and*
`|`	*or*
`~`	*not*

Operatore	Azione logica
`==`	*equal to*
`~=`	*not qual to*
`>=`	*greater than*
`<=`	*less than*

10 Aritmetica finita del calcolatore

È importante notare che l'insieme dei numeri utilizzabili da un calcolatore è un particolare sottoinsieme finito dei numeri razionali che chiamiamo $\mathbb{F}$. Per memorizzare ogni numero è disponibile una quantita limitata di bit (64 bit); di conseguenza i numeri che costituiscono l'aritmetica del calcolatore hanno necessariamente un numero finito di cifre decimali. In particolare, i numeri irrazionali come π oppure i numeri razionali periodici come $1/3$ sono sostituiti da una loro approssimazione con un numero finito di cifre decimali.
Se in $\mathbb{F}$ sono contenuti solo numeri con un numero finito di cifre decimali, attorno ad ogni elemento f di $\mathbb{F}$ esiste un piccolo intervallo I_f "vuoto", che contiene solo f stesso e nessun altro elemento di $\mathbb{F}$. In altre parole, la distanza fra f e il suo elemento successivo di $\mathbb{F}$ non è infinitamente piccola, ma è un valore ben determinato ("molto piccolo"). Tale distanza si definisce *epsilon macchina*, e si indica tramite $\epsilon(f)$.
In Matlab® e Octave esiste la variabile predefinita `eps` che rappresenta l'epsilon macchina del numero `1`, cioè il più piccolo numero che sommato al numero `1` fornisce un numero maggiore di `1`. In particolare:

```
 >> eps

ans =

   2.2204e-16
```

Memorizziamo in una variabile `a` il valore `1 + eps`, osservando che anche con il formato di visualizzazione `format long e` la variabile `a` sembra ancora essere uguale al numero `1`.

```
 >> format long e
>> a = 1+eps

a =

     1.000000000000000e+00
```

Proviamo ora a sottrarre ad `a` il numero `1` e riotteniamo esattamente il valore di `eps`:

```
 >> a - 1

ans =

     2.220446049250313e-16
```

Adesso eseguiamo lo stesso procedimento sommando al numero `1` il valore `eps/2` e otteniamo:

```
 >> format long e
>> b = 1+eps/2

b =
     1
```

Apparentemente non c'è nessuna differenza con il caso precedente, ma se adesso sottraiamo `1` da `b` otteniamo:

```
 >> b - 1

ans =
      0
```

Come è stato accenato in precedenza, l'epsilon macchina non è lo stesso per ogni numero di $\mathbb{F}$ e dipende dalla tecnica usata per gestire i 64 bit disponibili per la rappresentazione di ogni numero. In particolare in Matlab® esiste la funzione `eps( )` che prende in ingresso un numero e restituisce il corrispondente epsilon macchina. Per esempio:

comando	risultato
>> eps(1)	2.2204e-16
>> eps(10)	1.7764e-15
>> eps(100)	1.4211e-14
>> eps(1000)	1.1369e-13
...	...

Possiamo osservare che `eps(1)` coincide con il valore della variabile predefinita `eps`. In particolare è molto importante notare che all'aumentare del numero in considerazione, il corrispondente epsilon macchina aumenta. Questo significa che i numeri che costituiscono l'insieme $\mathbb{F}$ non sono equispaziati, ma sono piu addensati intorno ai numeri piccoli e si diradano man mano che il loro valore aumenta. Proviamo questa affermazione con un esempio:

```
 >> format long
 >> 1000 + eps(1) -1000

ans =

     0

>> 1 + eps(1) - 1

ans =
     2.220446049250313e-16
```

da cui deduciamo che il calcolatore non è in grado di interpretare il numero `eps(1)=2.2204e-16` come un incremento diverso da zero quando viene sommato al numero `1000`, mentre lo riconosce come numero non nullo quando viene sommato ad `1`. Il più piccolo incremento del numero `1000` riconosciuto dal calcolatore è il corrispondente epsilon macchina:

```
>> 1000 + eps(1000) - 1000

ans =
     1.136868377216160e-13
```

Come ultima considerazione, possiamo identificare l'elemento più piccolo e più grande di $\mathbb{F}$ con le variabili predefinite `realmin` e `realmax` di Matlab® e Octave :

```
>> realmin

ans =
    2.225073858507201e-308

>> realmax

ans =
    1.797693134862316e+308
```

Assegnando ad una variabile un numero maggiore di `realmax`, Matlab® non è in grado di interpretare l'istruzione di assegnazione:

```
 >> a = 1e400

a =

   Inf
```

ma non viene prodotto un messaggio di errore. Differentemente, è ancora possibile assegnare ad una variabile un valore inferiore di `realmin` e Matlab® è ancora in grado di interpretare correttamente l'istruzione:

```
>> a = 1e-310

a =

    9.999999999999969e-311
```

Il più piccolo numero in assoluto riconoscibile da Matlab® è l'epsilon macchina di `realmin`:

```
 >> eps(realmin)

ans =
    4.940656458412465e-324
```

e assegnando ad una variabile inferiore ad `eps(realmin)` Matlab® interpreta l'istruzione come assegnazione di un valore nullo:

```
 >> a = 1e-325

a =
     0
```

11 Esempi di errori numerici

Vediamo alcuni esempi di errori numerici che ci permettono di comprendere meglio che tipo di problemi sono legati all'aritmetica finita del calcolatore. In base alle conoscenze acquisite fino ad ora, è possibile definire un vettore di elementi equispaziati "a mano" oppure con il comando `linspace`:

```
 >> a = [0:0.01:1];
 >> b=linspace(0,1,101);
 >> whos
  Name      Size             Bytes  Class     Attributes

  a         1x101              808  double
  b         1x101              808  double
```

Osserviamo che i due vettori `a` e `b` sono costituiti dallo stesso numero di elementi e anche gli estremi del loro intervallo di definizione `[0,1]` sono gli stessi. Proviamo a fare un grafico della differenza, in valore assoluto tra ogni elemento di posto corrispondente dei due vettori. Definiamo il vettore

```
 >> c = abs( a-b );
 >> plot(c, '*-')
```

e osserviamo il grafico del vettore `c` in figura 6.

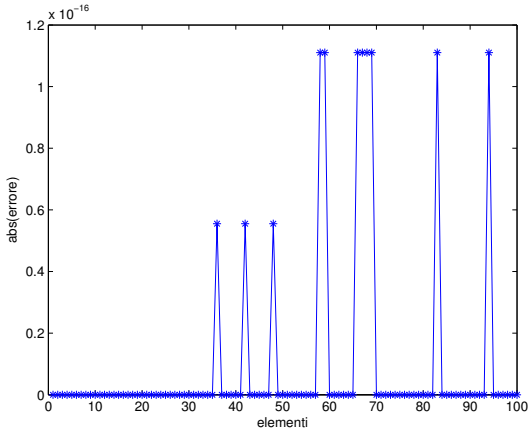


Figure 6: Grafico del vettore c

Si può notare che parecchi elementi sono affetti da errore e in particolare l'errore aumenta verso l'estremo di destra dell'intervallo.

Un altro esempio è il fenomeno noto come *cancellazione di cifre significative*. Calcolando con Matlab® l'espressione $((x+1)-1)/x$ per $x = 10^{-15}$, il cui risultato è ovviamente 1, si ottiene:

```
>> x=1e-15
x =
   1.0000e-15
>> ((1+x)-1)/x
ans =
    1.1102
```

Il motivo di questo fenomeno è che la somma fra numeri di $\mathbb{F}$ in valore assoluto molto diversi non è precisa, perché nelle conversioni che il calcolatore esegue per sommare i due numeri potrebbero perdersi alcune cifre significative. Infatti:

```
>> (1+x)-1
ans =
   1.1102e-15
```

mentre ovviamente:

```
>> (1-1)+x
ans =
   1.0000e-15
```

quindi possiamo concludere che in generale la somma non è associativa in $\mathbb{F}$. In generale è meglio evitare di sommare numeri molto diversi fra loro, e se è necessario farlo, cercare perlomeno di riordinare gli addendi dal più grande al più piccolo.

12 Messaggi d'errore in `Matlab`

Quando si scrive del codice sorgente in un qualunque linguaggio di programmazione, gli errori sono sempre dietro l'angolo. Questo vale anche in Matlab® , che però cerca di aiutare il debugging fornendo dei messaggi di errore che aiutino il più possibile a trovare il bug all'interno del proprio codice. È quindi estremamente utile imparare a leggere questi messaggi, in modo da capire su quale parte del codice concentrarsi nella ricerca dell'errore. In questo documento elencheremo i messaggi d'errore più comuni, mostrando le probabili cause che possono generare tali messaggi.

12.1 Errori di sintassi

Gli errori di sintassi sono gli errori che Matlab® trova nel codice ancor prima di eseguirlo. Questi errori infatti non consentono l'esecuzione del sorgente in quanto Matlab® non riesce a capire come interpretare quello che è stato scritto. Gli esempi più comuni sono:

- `Error: Expression or statement is incorrect--possibly unbalanced (, {, or [.`
  Questo errore significa che non tutte le parentesi che sono state aperte hanno la corrispondente parentesi chiusa. La causa può essere sia una parentesi chiusa mancante che una parentesi aperta di troppo. Ad esempio:

  ```
  >> v (1
   v (1
       |
  Error: Expression or statement is incorrect--possibly unbalanced (, {, or [.
  ```

  Si noti che Matlab® tende a suggerire la posizione in cui ritiene che la parentesi vada inserita (utilizzando il simbolo `|` nelle riga sopra al messaggio di errore).

- `Error: Unbalanced or misused parentheses or brackets.`
  Simile al caso precedente, questo errore di solito significa che c'è una parentesi chiusa in più:

  ```
  >> v (1))
   v (1))
        |
  Error: Unbalanced or misused parentheses or brackets.
  ```

  Anche in questo caso, Matlab® prova a dare un suggerimento su quale possa essere la parentesi incriminata.

- `Error: Expression or statement is incomplete or incorrect.`
  Questo errore signfica che l'istruzione data è in qualche modo incompleta. Ad esempio:

  ```
  >> a = 1 + 3 +
   a = 1 + 3 +
              |
  Error: Expression or statement is incomplete or incorrect.
  ```

- `Error: Unexpected MATLAB operator.`
  In questo caso invece l'istruzione data contiene un operatore che Matlab® non riconosce. Per operatore si intende un qualunque simbolo che agisce su uno o più argomenti per generare un risultato. Gli esempi più semplici di operatori sono gli operatori aritmetici (`+`, `-`, `*` e `/`), ma anche il simbolo `:` nell'istruzione `x = 1:10` è un operatore. Se viene usato un operatore che Matlab® non conosce, viene stampato a video l'errore sopra, poiché Matlab® non sa come generare il risultato a partire dagli operandi. Ad esempio:

```
>> x = 1 +/ 2
 x = 1 +/ 2
        |
Error: Unexpected MATLAB operator.
```

Matlab® non conosce l'operatore `+/` e non sa quindi come generare il valore da salvare in `x` a partire dai valori `1` e `2`.

È importante notare che se tali comandi sono inseriti in uno script Matlab® , il messaggio d'errore conterrà anche **la riga e la colonna** che indicano dove all'interno del file .m Matlab® ha trovato l'errore riportato. Se ad esempio salviamo i comandi:

```
a = 1;
b = 2;

c = a +/ b
```

nel file `prova.m` e cerchiamo di eseguirlo, otteniamo:

```
>> prova
Error: File: prova.m Line: 4 Column: 8
Unexpected MATLAB operator.
```

L'errore è effettivamente alla colonna 8 della quarta riga. Si noti inoltre che lo script **non viene eseguito**. Dopo la chiamata allo script, infatti, le variabili `a` e `b` non sono state definite: si possono ad esempio a dare i comandi

```
clear all
prova
whos
```

per vedere immediatamente che non c'è traccia di tali variabili.

12.2 Errori di esecuzione

A differenza degli errori di sintassi, gli errori di esecuzione vengono segnalati solo quando i comandi vengono effettivamente eseguiti. Si tratta infatti di errori generati da comandi formalmente corretti che contengono però errori logici che non consentono la corretta esecuzione del comando. Vediamone alcuni:

- ```
  Error using  ***
  Matrix dimensions must agree.
  ```

  dove `***` indica una qualunque operazione fra `+`, `-`, `\`, `.*` e `./`. Questo errore indica che si è provato a fare un'operazione su operandi con dimensioni non compatibili rispetto a quell'operazione. Ad esempio in:

  ```
  >> A = rand (3, 4);
  >> B = rand (4, 3);
  >> A + B
  Error using  +
  Matrix dimensions must agree.
  ```

  la somma tra `A` e `B` non può essere calcolata, poiché la prima ha dimensione `3x4`, mentre la seconda ha dimensione `4x3`.
  Nel caso della moltiplicazione, l'errore è leggermente diverso:
  ```

```
>> C = rand (3, 3);
>> D = rand (4, 3);
>> C * D
Error using  *
Inner matrix dimensions must agree.
```

L'errore in questo caso parla di `Inner matrix dimensions`, in quanto per la moltiplicazione matriciale sono solo il numero di colonne della prima matrice e il numero di righe della seconda a definire se l'operazione è effettuabile o meno.

- `Index exceeds matrix dimensions.`
  Questo errore indica che si è provato ad accedere a un elemento in una posizione non presente nella matrice (o vettore), in quanto l'indice utilizzato è maggiore delle dimensioni della matrice stessa. Ad esempio:

```
>> v = [1 2 3];
>> v (4)
Index exceeds matrix dimensions.
```

  Un errore uguale negli effetti ma diverso nelle cause è:
  `Subscript indices must either be real positive integers or logicals.`
  In questo caso, l'indice utilizzato per accedere alla matrice non è un numero intero positivo:

```
>> v (0)
Subscript indices must either be real positive integers or logicals.

>> v (1.2)
Subscript indices must either be real positive integers or logicals.
```

- `Undefined function or variable '***'.`
  La variabile o la funzione usata non è mai stata definita. Nel caso delle variabili, controllate il vostro Workspace! Potreste aver cancellato per sbaglio la vostra variabile con i comandi `clear` o `clear all`. Se non è stata definita, ad esempio, una variabile `c` all'interno di una function `esempio.m`:

```
function [ b ] = esempio( a )
b=a+c;
end
```

  chiamandola dalla Command Window, otteniamo:

```
  >> esempio(4)
Undefined function or variable 'c'.
Error in esempio (line 3)
b=a+c;
```

  Si noti che questo errore compare anche quando nella chiamata a funzione c'è un **errore di battitura** nel nome della funzione! Si ricordi anche Matlab® distingue le lettere maiuscole dalle minuscole. Ad esempio:

```
>> A = rand (4, 4);
>> sums (A)
Undefined function or variable 'sums'.
>> Sum (A)
Undefined function or variable 'Sum'.
```

Il nome corretto della funzione è `sum`.

- `Not enough input arguments/Too many input arguments/Too many output arguments`
  Si è provato a chiamare la funzione con un numero di argomenti in ingresso o in uscita non appropriato. Si noti che questo stesso errore viene stampato quando si prova a usare una function come uno script:

  ```
  >> [V, D] = eig
  Error using eig
  Not enough input arguments.
  ```

  L'errore inverso (provare a eseguire uno script come una function, ossia chiamandolo dalla Command Window passando argomenti in ingresso) viene invece segnalato dal seguente errore:

  ```
  >> myscript (A)
  Attempt to execute SCRIPT myscript as a function
  ```

Come nel caso degli errori di sintassi, anche per gli errori di esecuzione viene segnalata la riga a cui l'errore è avvenuto nel caso in cui i comandi siano inseriti in uno script. Per esempio, salvando i comandi:

```
clc
clear

A = rand (3, 4);
B = rand (4, 3);

A - B
```

nel file `esempio_errori.m` e chiamandolo dalla Command Window, otteniamo:

```
>> esempio_errori.m
Error using  -
Matrix dimensions must agree.

Error in esempio_errori (line 7)
A - B
```

Si notino però due differenze fondamentali rispetto al caso trattato in precedenza:

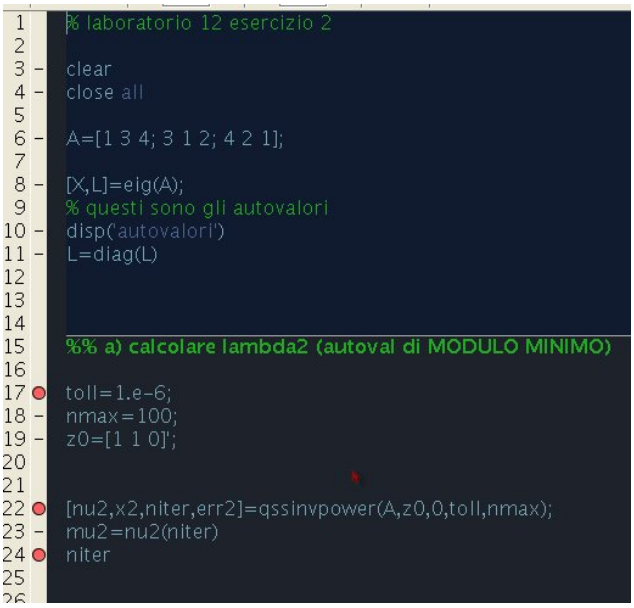
1. il messaggio di errore riporta la riga a cui tale errore è avvenuto ma non la colonna. Questo è ovvio, se pensiamo al tipo di errore: Matlab® sa solo che quell'istruzione ha dato un errore, ma non può dire quale parte dell'istruzione stessa ha generato tale errore, in quanto la scrittura è formalmente corretta
2. lo script **viene eseguito fino alla riga precedente a quella che dà errore!** Il comando `whos` dato dopo la chiamata allo script infatti stampa:

   ```
   >> whos
     Name      Size            Bytes  Class     Attributes
     A         3x4                96  double
     B         4x3                96  double
   ```

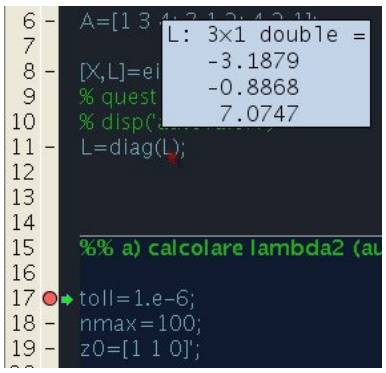
   Le due variabili sono dunque state create e sono rimaste in memoria.

13 Il debugger di `Matlab`

Il debugger di Matlab® è uno strumento estremamente utile, che consente di velocizzare di molto la ricerca degli errori all'interno di un codice Matlab® .
Infatti, quando un programma viene eseguito in modalità debug, è possibile arrestare l'esecuzione del codice alla riga che si desidera, e da lì in poi eseguire il programma riga per riga, visualizzando ogni volta che si desidera il contenuto di tutte le variabili presenti in memoria, e anche modificandone il valore.
Per arrestare l'esecuzione di uno script Matlab® ad una determinata riga è sufficiente cliccare sul trattino che compare a destra del numero di riga. Al posto del trattino comparirà un pallino rosso (*breakpoint*).



Se si esegue lo script il programma si fermerà prima di eseguire la riga evidenziata, indicando il punto di cui l'esecuzione si è fermata con una freccia. Il prompt di Matlab® diventa `K>>` ad evidenziare che si è in modalità debug. Si possono eseguire tutti i comandi Matlab® , ad esempio visualizzare o modificare i valori delle variabili in memoria. Si possono visualizzare i valori delle variabili anche nella finestra dello script, posizionando il mouse sopra la variabile desiderata.

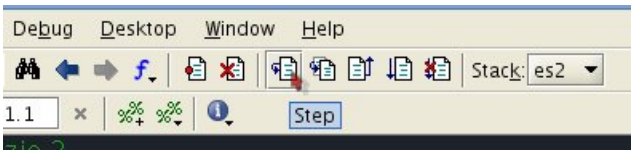


A questo punto di possono eseguire le diverse operazioni di debug :

- `step`: esegue la riga e si ferma nuovamente
- `step in`: prova ad eseguire la riga e, se la riga contiene una chiamata a funzione, si ferma alla prima riga della funzione chiamata, per eseguirne il debug

- `step out`: esce dalla funzione che è stata aperta da `step in`, finisce l'esecuzione della riga e si ferma alla riga successiva
- `continue`: continua fino al *breakpoint* successivo
- `dbquit`: esce dalla modalità debug ed interrompe l'esecuzione del programma.

Questi comandi sono accessibili tramite il menù "Debug", oppure sulla barra degli strumenti, oppure tramite scorciatoie da tastiera.



Per eliminare i *breakpoints* attivati, utilizzare il comando "clear breakpoints in all files" dal menù debug (oppure il tasto corrispondente sulla barra degli strumenti).

La modalità debug fornisce anche molti altri strumenti (*conditional breakpoints*, comando `keyboard`, come muoversi fra gli ambiti di visibilità quando più funzioni vengono chiamate una all'interno dell'altra, ...), per i quali si rimanda alla documentazione di Matlab® .