Introduzione all'uso di MATLAB ®
per il Calcolo Scientifico

Paola Causin, Stefano Micheletti, Riccardo Sacco

Dipartimento di Matematica "F. Brioschi",
Politecnico di Milano, Via Bonardi 9, 20133 Milano, Italy

20 ottobre 2000

Prefazione

Queste note nascono dall'esperienza di vari anni nell'insegnamento del Calcolo Numerico agli Allievi Ingegneri del Politecnico di Milano e raccolgono in maniera sistematica il materiale presentato in un corso svolto nell'estate 2000 per i Docenti del Dipartimento di Matematica del Politecnico di Milano.

Il lavoro è in particolar modo mirato a fornire un supporto didattico di facile consultazione sia per i Docenti che intendono avvalersi di `MATLAB` ® nel proprio insegnamento, sia per gli Studenti che potranno riconoscere in `MATLAB` ® uno strumento versatile e di grande utilità nell'ambito dei propri Corsi di Laurea.

Indice

1 Introduzione **4**

2 Per iniziare ... **7**

3 Algebra lineare **11**
3.1 Operazioni su scalari 23
3.2 Operazioni su vettori 24
3.2.1 Operazioni elemento per elemento su vettori 25
3.3 Operazioni su matrici 26
3.3.1 Operazioni elemento-elemento su matrici 27
3.4 Funzioni intrinseche definite per vettori e matrici 27

4 La programmazione in `MATLAB` **38**
4.1 M-files: Functions e Scripts 39
4.1.1 Functions 39
4.1.2 Script 41
4.2 Operatori logici e di confronto 42
4.3 Operazioni di input-output 44
4.4 Costrutti sintattici fondamentali 46
4.4.1 Ciclo `for` 46
4.4.2 Ciclo `while` 47
4.5 Le istruzioni `if`, `else` e `elseif` 47
4.5.1 L'istruzione `break` 48

5 La grafica in `MATLAB` **50**
5.1 Grafici in due dimensioni 50
5.2 Rappresentazione di curve in tre dimensioni 56
5.3 Rappresentazione di superfici in tre dimensioni 57

6 Esempi **61**
6.1 Esempio 1: Calcolo di integrali in una dimensione 61
6.2 Esempio 2: Visualizzazione di un vettore trasformato per roto-traslazione . 64
6.3 Esempio 3: Generazione e visualizzazione di successioni per ricorrenza . . . 68
6.4 Esempio 4: Approssimazione di una funzione con polinomi di Taylor. . . . 73
6.5 Esempio 5: Approssimazione di una funzione in serie di Fourier 77

7 Risoluzione numerica di equazioni differenziali ordinarie **79**

8 Trasformata Rapida di Fourier (FFT) **93**

9 Il toolbox `pde` **96**
9.1 Draw Mode 99
9.2 Boundary Mode 100
9.3 PDE Mode 101
9.4 Mesh Mode 101
9.5 Solve PDE 102
9.6 Plot Solution 103

1 Introduzione

MATLAB ® è un **linguaggio di programmazione** ad alto livello per calcolo scientifico, ovvero un linguaggio dotato di una potenza espressiva molto estesa e vicina alla sintassi umana.

Il nome MATLAB ®, abbreviazione di *Matrix laboratory*, corrisponde all'obiettivo originale di fornire un'interfaccia elementare ed immediata all'uso dei più diffusi pacchetti software per la risoluzione numerica di problemi di Algebra Lineare, quali le librerie LINPACK e EISPACK.

La struttura dati di base in MATLAB ® è costituita infatti dalla **matrice**: ciò significa che durante l'elaborazione ogni quantità (indicata nel seguito come *variabile*) viene trattata dall'ambiente di calcolo come una matrice di dimensione $m \times n$ (uno scalare reale è dunque memorizzato come una matrice 1×1).

Grazie a questo approccio *matrix-oriented* in MATLAB ® non è necessario dichiarare esplicitamente all'inizio del processo di calcolo una variabile matrice in termini delle sue dimensioni e della natura dei suoi coefficienti (numeri interi, reali o complessi, eventualmente in singola o doppia precisione), a differenza di quanto accade nei moderni linguaggi di programmazione (ad esempio, il `Fortran90` o il `C`), dove la fase di dichiarazione delle variabili è obbligatoria. Ciò costituisce una notevole semplificazione per l'utente sia nell'uso interattivo dell'ambiente di calcolo sia nella stesura di programmi.

Inoltre in MATLAB ® è già predefinito un ampio insieme di matrici elementari (matrice identità, matrice nulla, matrice unità..), di operazioni fra matrici (somma, moltiplicazione...) e di operatori algebrici di uso comune, quali ad esempio il calcolo del determinante o del rango di una matrice; si osservi a questo proposito che un'efficiente implementazione di queste operazioni può influire in modo considerevole sui tempi di calcolo.

Oltre alla semplicità della gestione delle variabili, un altro punto di forza di MATLAB ® è costituito dalle cosiddette *built-in functions*. Queste sono delle funzioni primitive (cioè già implementate all'interno di MATLAB ®) che possono essere richiamate dall'utente per risolvere in modo rapido ed efficiente problemi di elevata complessità, quali ad esempio:

- calcolo di autovalori ed autovettori di una matrice,
- risoluzione di sistemi lineari,
- fattorizzazioni (LU, QR) e decomposizione ai valori singolari (SVD) di una matrice.

Pur essendo stato concepito originariamente come risolutore di problemi di algebra lineare, MATLAB ® è diventato nella versione attuale un software interattivo estremamente *user-friendly* in grado di trattare in una logica generale e flessibile problemi di calcolo scientifico della più svariata provenienza. Utilizzando le familiari notazioni matematiche è infatti possibile definire in MATLAB ® il problema di interesse, affrontarne la risoluzione ed infine analizzare i risultati ottenuti con gli strumenti grafici.

A compendio ed integrazione del software di base sono state sviluppate in MATLAB ® delle **estensioni** del pacchetto originario: i `toolboxes` e `Simulink`. Un `toolbox` è una collezione omogenea di funzioni MATLAB ® (i cosiddetti M-files che verranno introdotti nel

paragrafo 4) specializzate nella risoluzione di una classe specifica di problemi di interesse applicativo. Un elenco parziale (ogni anno vengono proposti sempre nuovi aggiornamenti) di settori scientifici per i quali esiste un corrispondente `toolbox` è il seguente:

- teoria dei sistemi e del controllo;
- trattamento di segnali e di immagini;
- reti neurali;
- logica *fuzzy*;
- ottimizzazione;
- equazioni alle derivate parziali (**pde**);
- statistica;
- calcolo simbolico;
- finanza.

`Simulink` è un pacchetto interattivo per la simulazione di sistemi dinamici non lineari. Operando all'interno di un ambiente grafico è possibile schematizzare un sistema in termini di relazioni ingresso-uscita in un diagramma a blocchi e analizzarne dinamicamente le caratteristiche. `Simulink` permette di studiare sistemi dinamici continui e discreti, a multi-ingresso e multi-uscite. Per applicazioni specifiche (tipicamente, nel settore delle comunicazioni o del trattamento di segnali) esistono in questo ambito librerie di blocchi predefiniti (i cosiddetti `Blocksets`).

Per una descrizione dettagliata del software di base di `MATLAB` ®, di tutte le estensioni disponibili e per una raccolta aggiornata di guide per un uso elementare ed avanzato, si può fare riferimento al sito ufficiale di `MATLAB` ®:

`www.mathworks.com`

e ai numerosi siti degli utenti nelle università e nell'industria.

Nota. In questo lavoro si presuppone l'utilizzo di una qualunque versione di `MATLAB` ® a partire dalla 5.0. (Si osservi che il toolbox `pde` è disponibile solo dalla versione 5.3.0). La maggior parte del contenuto di questa dispensa è tuttavia compatibile anche con versioni di `MATLAB` ® precedenti (tipicamente a partire dalla 4.2).

Il sommario di queste note all'uso di `MATLAB` ® è il seguente. Nel paragrafo 2 si fornisce una breve descrizione della procedura di avvio di `MATLAB` ®, dei comandi e istruzioni di base e dell'interfaccia fra l'ambiente di calcolo ed il sistema operativo, con particolare riferimento a `Windows`. Nel paragrafo 3 vengono introdotte la dichiarazione e la manipolazione delle matrici e alcuni dei comandi principali che operano su di esse. Nel paragrafo 4 si descrivono le potenzialità dell'ambiente di calcolo come linguaggio di programmazione vero e proprio. Nel paragrafo 5 viene fornita una panoramica dei comandi principali che permettono il tracciamento di grafici bi- e tridimensionali. Nel paragrafo 6 sono presentati alcuni esempi relativi ad argomenti trattati nel corso di *Elementi di Analisi Matematica*

(A) e di Geometria del primo anno di studi di Ingegneria del Nuovo Ordinamento. Precisamente, si affrontano: il calcolo integrale, le successioni per ricorrenza, l'approssimazione di funzioni con gli sviluppi in serie di Taylor e di Fourier e le trasformazioni di vettori nel piano. Gli ultimi tre paragrafi trattano argomenti più specifici; in particolare, nel paragrafo 7 si presenta il pacchetto **ode suite** per la risoluzione di sistemi di equazioni differenziali ordinarie, nel paragrafo 8 viene brevemente descritto l'algoritmo della Trasformata Rapida di Fourier (FFT) mentre nel paragrafo 9 si illustra il toolbox **pde** per la risoluzione di sistemi di equazioni differenziali alle derivate parziali. Ogni paragrafo è corredato di numerosi esempi allo scopo di agevolare la comprensione degli argomenti discussi. [1]

[1] D'ora in poi ometteremo per semplicità il simbolo di Copyright nella scritta `MATLAB` ®

2 Per iniziare ...

In questo paragrafo descriviamo brevemente come avviare MATLAB, i comandi di base per familiarizzarsi con il suo uso e le modalità di interazione tra l'ambiente di calcolo e il sistema operativo all'interno del quale esso è installato (assumiamo nel seguito di lavorare nei sistemi Windows o Unix).

Per lanciare MATLAB da ambiente Windows basta cliccare due volte con il *mouse* sull'icona corrispondente. In ambiente Unix si deve digitare il comando matlab (eventualmente con il percorso completo del file eseguibile, che tipicamente si trova nel direttorio /usr/local/matlab/bin) dal *prompt* di sistema e quindi dare il comando di invio.
All'avvio viene visualizzata la seguente schermata

```
Commands to get started: intro, demo, help help
Commands for more information: help, info, subscribe

>>
```

Per entrare in confidenza con l'ambiente di lavoro è utile lanciare il comando demo che illustra le potenzialità del software attraverso significativi esempi numerici e casi test. È altresì essenziale fare costante riferimento all'uso dell'help che permette di ottenere informazioni dettagliate su qualsiasi comando. Ad esempio, la seguente istruzione consente di conoscere in dettaglio la sintassi e le funzioni del comando sqrt per il calcolo della radice quadrata di un numero:

```
>> help sqrt

 SQRT   Square root.
    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

    See also SQRTM.
```

Il modo più immediato per interagire con MATLAB consiste nello scrivere direttamente l'istruzione sulla linea di commando e farla seguire dal carattere di invio per mandarla in esecuzione. Si consideri ad esempio l'assegnazione del valore 3 alla variabile a:

```
>> a = 3

a =

     3
```

Questa modalità di interazione caratterizza MATLAB come un **linguaggio interprete** (o **interprete di comandi**). Tale prerogativa rende il colloquio con la macchina estremamente semplice e naturale; lo svantaggio di questo approccio risiede nel fatto che talune operazioni possono risultare più lente nell'esecuzione rispetto alla loro corrispondente implementazione mediante un linguaggio compilato (ovvero un linguaggio che preveda la traduzione del programma sorgente in istruzioni macchina direttamente eseguibili dal computer).

È naturalmente possibile anche un livello di interazione con MATLAB più sofisticato di quello appena descritto. Esso consiste nell'esecuzione di un insieme di istruzioni successive, più o meno articolate, memorizzate sotto formato di file di testo, detto `M-file`. Nel caso in cui l'M-file riceva parametri in ingresso e restituisca parametri in uscita esso diventa una `function`, in caso contrario esso si chiama *script*. Una descrizione dettagliata di entrambe le modalità d'uso di un M-file sarà fornita nel paragrafo 4.1.

Durante la sessione di lavoro è possibile richiamare i comandi precedentemente digitati utilizzando i tasti $\leftarrow \rightarrow \uparrow \downarrow$. Per esempio premendo una volta il tasto $\uparrow$ viene richiamato l'ultimo comando immesso. Premendo ripetutamente tale tasto vengono richiamati ad uno ad uno i comandi digitati, richiamandoli "storicamente" dall'ultimo fino eventualmente al primo. Il tasto $\downarrow$ ripercorre invece i comandi in direzione inversa partendo dal primo fino all'ultimo. Inoltre, immettendo i primi caratteri di un'istruzione già precedentemente digitata e poi premendo il tasto $\uparrow$, viene completata la riga con l'ultima istruzione che inizia con quegli stessi caratteri. Durante l'immissione di un'istruzione i tasti $\leftarrow$ e $\rightarrow$ permettono di riposizionare sulla linea il cursore e di modificare il testo scritto. Alternativamente, operando con il tasto sinistro del mouse sulla finestra di calcolo si possono selezionare parti di testo che è poi possibile copiare, tagliare ed incollare utilizzando il menu **Edit** della barra degli strumenti (oppure il tasto destro del mouse per gli utenti Windows).

Per salvare una cronaca completa della sessione di lavoro si deve eseguire *all'inizio* della sessione stessa il comando `diary`:

```
>> diary mywork.dat
>> a = 3;
>> help sqrt

 SQRT   Square root.
    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

    See also SQRTM.

>> diary off
```

L'istruzione `diary mywork` apre il file di testo `mywork.dat` nel quale viene trascritto interamente il flusso di istruzioni digitate successivamente. Con l'istruzione `diary off` si chiude il file `mywork.dat` che rimane salvato nel direttorio corrente. Si osservi che il file di testo creato dal comando `diary` **NON** permette di recuperare il contenuto delle variabili definite durante la sessione di lavoro. A tale scopo si deve invece salvare tutta l'area di memoria (o parte di essa) con il comando `save`:

```
>> x = 1;
>> a = 3;
>> z = sqrt(a)-x;
>> save areawork
>> save xzarea x z
```

Con la prima istruzione `save` viene creato il file binario `areawork.mat` contenente **tutte** le variabili attive in menoria (`x`, `a` e `z`), mentre la seconda istruzione `save` memorizza

le sole variabili `x` e `z` nel file binario `xzarea.mat`. Per ripristinare in una successiva sessione di lavoro le variabili così salvate basta utilizzare il comando `load` seguito dal nome del file che le contiene (senza l'estensione `.mat`).

Per uscire dall'ambiente `MATLAB` è infine sufficiente digitare l'istruzione `quit`.

A completamento di questo paragrafo introduttivo, descriviamo brevemente le modalità di interazione tra l'ambiente di calcolo e il sistema operativo.

È anzitutto possibile eseguire alcune istruzioni proprie del sistema operativo (ad esempio copia di un file, lista di file, ...) direttamente dal *prompt* di `MATLAB` anteponendo all'istruzione desiderata il carattere '!'. Ad esempio per gli utenti Windows l'istruzione

```
>> !copy c:\prova1\work1.m c:\prova2\work2.m
```

permette di fare una copia del file `work1.m` e di chiamarla `work2.m`. È necessario specificare il percorso completo, come in questo caso, qualora il file da copiare si trovi in un direttorio che non è quello corrente o qualora si voglia collocare il nuovo file in un direttorio diverso da quello corrente (il comando `pwd` permette di conoscere il direttorio corrente di lavoro). Analogamente per gli utenti Unix

```
>> !cp /u/tom/work1.m /u/tom/work2.m
```

copia il file `work1.m` e chiama la copia `work2.m`.

È inoltre importante conoscere le procedure di inizializzazione che `MATLAB` attiva al suo avvio. Una volta digitato il comando `matlab`, viene eseguito automaticamente il file `matlabrc.m` che si trova per default nel direttorio `c:\ matlab \ toolbox \ local`. Inoltre, se esiste in tale direttorio, viene eseguito anche il file `startup.m`. L'esecuzione del file `matlabrc.m` imposta il `path` di `MATLAB` lanciando il file `pathdef.m` e i principali parametri per la grafica.

Il `path` (in italiano, percorso) è un elenco di direttori nei quali si indica a `MATLAB` di ricercare i file eseguibili, detti M-files. In generale, quando dal prompt viene digitata una sequenza di caratteri seguita da invio, ad esempio

```
>> apple
```

`MATLAB` opera nel seguente modo:

- controlla se `apple` è *una variabile* correntemente in uso, altrimenti ...
- controlla se `apple` è *una funzione built-in*, altrimenti ...
- controlla se nel direttorio corrente esiste un M-file chiamato `apple.m`, altrimenti ...
- controlla se il file `apple.m` è presente in qualche direttorio specificato nel path, partendo dall'inizio della lista dei direttori indicati e fermandosi appena trova un file con tale nome, altrimenti ...
- viene visualizzato il messaggio di errore:

```
>> apple
??? Undefined function or variable 'apple'.
```

Per visualizzare il `path` impostato da `pathdef.m` basta digitare il comando

```
>> path
```

Il file `startup.m` può essere utilizzato dall'utente per definire particolari parametri o variabili all'inizio dell'elaborazione. Ad esempio, il seguente file `startup.m` (generato con un qualsiasi editore di testo)

```
addpath c:\mydir -end
```

aggiunge all'avvio al `path` di default il direttorio `c:\ mydir` (collocando il nome di tale direttorio in fondo alla lista di direttori contenuta nel file `pathdef.m`) permettendo così a `MATLAB` di eseguire programmi creati dall'utente e residenti in `c:\ mydir`.

Nota. Si osservi che, se si sta lavorando in un ambiente condiviso da altri utenti, come accade normalmente nelle aule informatizzate delle università, *non* è permesso modificare il file `startup.m`. È però ugualmente possibile aggiungere un direttorio al path, limitatamente alla sessione di lavoro corrente, digitando *dalla linea di comando* l'istruzione:

```
addpath c:\mydir -end
```

Gli utenti Windows possono anche aggiungere un direttorio al path preesistente selezionando l'icona della barra degli strumenti rappresentata qui a destra. Nella finestra che appare (si veda la Figura 1) si deve prima selezionare l'opzione "Path" e poi, nel menu a tendina che appare, selezionare "Add to Path". Si deve quindi digitare nello spazio predisposto il nome del direttorio da aggiungere e attivare l'opzione "back" che posiziona il nuovo direttorio in fondo alla lista attuale.

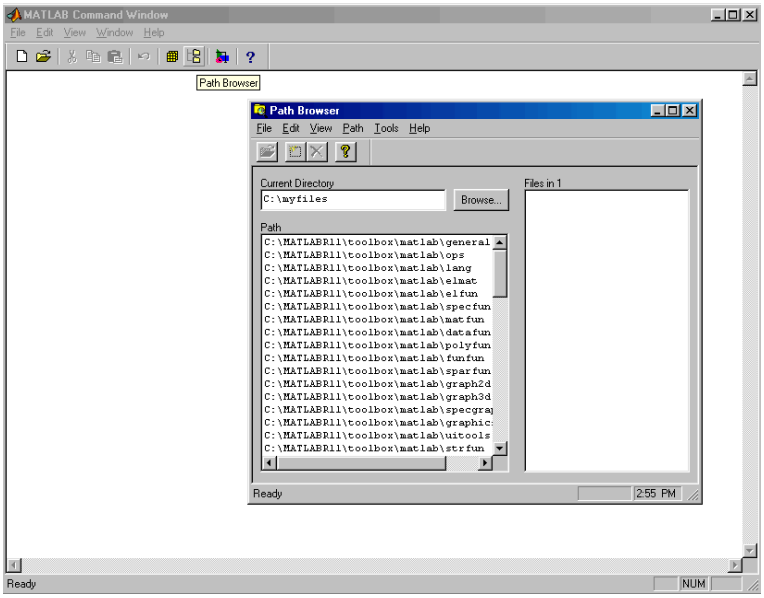


Figura 1: Aggiunta di un direttorio al path di ricerca di `MATLAB`

3 Algebra lineare

Nell'introduzione si è accennato al fatto che l'elemento di base di MATLAB è la matrice. Vedremo in questo paragrafo quali sono i diversi tipi di matrici supportati e le principali operazioni che si possono compiere su di esse. Fondamentalmente esiste un unico tipo di matrice, la più generale matrice ad elementi complessi, di dimensione $m \times n$ (cioè costituita da m righe e n colonne, con $m \neq n$ eventualmente). Può essere utile in molti casi riferirsi ai casi particolari seguenti:

$$\begin{array}{lcl} \text{Lo scalare} & \Rightarrow & m = n = 1. \\ \text{Il vettore riga} & \Rightarrow & m = 1, n > 1. \\ \text{Il vettore colonna} & \Rightarrow & m > 1, n = 1. \\ \text{La matrice quadrata} & \Rightarrow & m = n. \end{array} \tag{1}$$

La dichiarazione di ogni variabile è automatica nel momento in cui se ne assegna il valore. Non è quindi necessario dichiarare la variabile stessa a priori, come avviene ad esempio in Fortran o in C. Ad esempio se volessimo dichiarare e assegnare alla variabile s il valore 10 digitiamo la seguente istruzione dalla riga di comando

```
>> s = 10

s =

    10
```

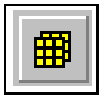
Come si osserva, MATLAB risponde con un'eco che conferma l'esito positivo dell'assegnazione. Ogni qualvolta lo si desidera possiamo conoscere qual è lo stato dell'ambiente di lavoro, il cosiddetto *Workspace*, tramite il comando whos

```
>> whos
  Name        Size              Bytes  Class

  s           1x1                   8  double array

Grand total is 1 elements using 8 bytes
```

Le informazioni restituite dal comando hanno il significato seguente: ogni riga si riferisce ad una variabile diversa e per ciascuna variabile vengono elencati:

- Name Il nome della variabile.
- Size Il numero di righe-colonne $m \times n$.
- Bytes L'occupazione di memoria.
- Class Classe o tipo della variabile. Può assumere i valori double, sparse, struct, cell, char, <nome-classe> (si rimanda al comando class per una trattazione dei tipi non elementari come struct, cell...).

Gli utenti Windows possono in alternativa selezionare dalla barra degli strumenti l'icona rappresentata qui a destra, che apre una finestra, detta *Workspace Browser*, che mostra le stesse informazioni del comando whos (si veda la Figura 2).



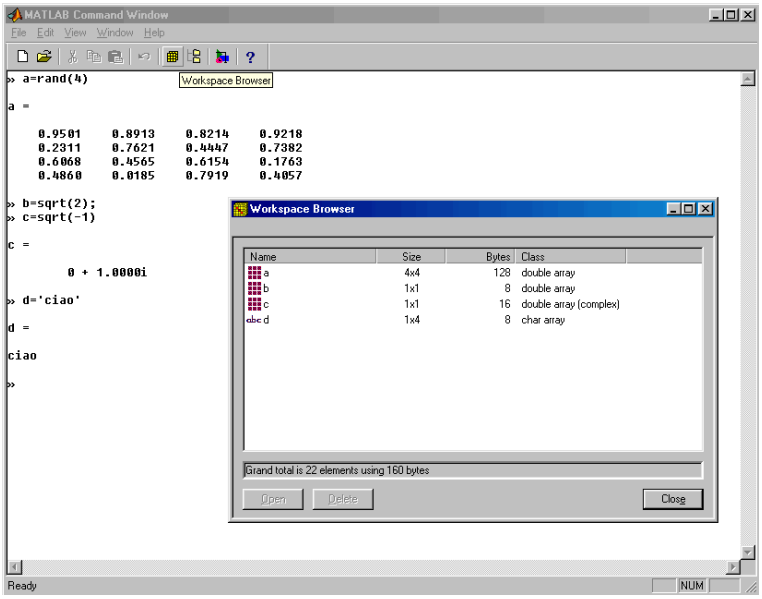


Figura 2: Uso equivalente del comando `whos` mediante icona

Osservazione La variabile `s` non è una variabile intera, come ci si sarebbe potuto aspettare; anzi, tale classe non esiste nemmeno. Infatti, tutte le operazioni vengono svolte lavorando con la cosiddetta *doppia precisione*. Ciò significa che il calcolatore considera ogni quantità numerica come un numero reale e ogni variabile reale è rappresentata internamente con 64 bit, cui corrispondono circa 16 cifre significative decimali. Rimandiamo a [3], Cap.2, per una trattazione più approfondita di questi fondamentali aspetti che sono tipico ambito del Calcolo Numerico.

Osservazione In alcuni casi può essere utile dichiarare una variabile vuota. Ciò si ottiene con la sintassi seguente

```
>> V = []

V =

     []

>> whos
  Name      Size            Bytes  Class

  V         0x0                 0  double array
  s         1x1                 8  double array

Grand total is 1 elements using 8 bytes
```

Si osservi che la variabile `V` non occupa alcuno spazio in memoria ma è comunque dichiarata, per cui ogni riferimento successivo ad essa è legittimo. Nel paragrafo 4.5.1 vedremo un esempio significativo in cui è comodo dichiarare una *variabile vuota*, ossia dichiarare l'esistenza di tale variabile senza però assegnare ad essa un valore.

Per chiarire l'osservazione precedente sulla precisione con cui vengono effettuati i calcoli illustriamo brevemente il comando `format`. Esso serve per impostare il formato di

visualizzazione dei risultati, mentre non modifica la precisione dei calcoli che, come detto, sono sempre condotti in doppia precisione. In particolare sono disponibili le seguenti opzioni:

- `short` virgola fissa con 5 cifre (è il formato di default)

```
>> format
>> pi

ans =

    3.1416
```

- `long` virgola fissa con 15 cifre

```
>> format long
>> pi

ans =

   3.14159265358979
```

- `short e` virgola mobile con 5 cifre

```
>> format short e
>> pi

ans =

   3.1416e+00
```

- `long e` virgola mobile con 15 cifre

```
>> format long e
>> pi

ans =

     3.141592653589793e+00
```

- `rat` razionale (rapporto di due numeri interi)

```
>> format rat
>> pi

ans =

   355/113

>> 6/4
```

```
ans =

     3/2
```

- `bank` formato per operazioni bancarie (vengono visualizzate solo due cifre decimali)

```
>> format bank
>> pi

ans =

          3.14
```

Un'altra opzione utile è il simbolo di continuazione. Normalmente infatti quando premiamo il tasto d'invio `MATLAB` interpreta la linea di comando e la esegue. A volte la linea di comando può essere lunga fino a tendere a oltrepassare i limiti fisici imposti dalla finestra su cui stiamo lavorando. In ogni caso, indipendentemente dalla lunghezza della linea, possiamo continuare a editarla andando a capo con l'invio senza provocare l'esecuzione terminando la linea stessa con tre o più punti. Ad esempio

```
>> x = 1;
>> x = x +
??? x = x +
          |
Missing operator, comma, or semi-colon.

>> x = x + ...
       1

x =

      2
```

Per default, `MATLAB` produce sempre una eco di tutte le assegnazioni e operazioni effettuate. Per inibire tale modalità occorre terminare la riga di comando con il carattere punto-e-virgola " ; ", che risulta praticamente indispensabile quando si lavora con matrici e vettori per evitare il susseguirsi di lunghe schermate di valori numerici. Nell'assegnazione di una variabile si può naturalmente fare riferimento ad una variabile definita precedentemente, come ad esempio

```
>> t = s + 1

t =

    11
```

Le variabili di classe `char`, cioè quelle che in molti contesti sono dette sinteticamente *stringhe* si dichiarano con la seguente sintassi

```
>> u = 'ciao'

u =

ciao

>> whos
  Name      Size          Bytes  Class

  s         1x1               8  double array
  t         1x1               8  double array
  u         1x4               8  char array

Grand total is 6 elements using 24 bytes
```

Si osservi in particolare la terza riga dell'output del comando `whos`: La variabile `u` è una variabile di tipo `char array` e in particolare un vettore riga a quattro componenti ciascuna delle quali occupa uno spazio di due bytes.

Nel caso in cui non si assegni esplicitamente il risultato di una operazione ad una variabile, `MATLAB` assegna automaticamente il risultato stesso alla variabile di default `ans` (answer)

```
>> 3.5

ans =

    3.5000
```

Tale variabile può essere poi usata in successive operazioni ma è buona norma salvarne il valore in una variabile definita dall'utente per non correre il rischio che `ans` venga sovrascritta da un altro valore.

Le variabili possono essere cancellate con il comando `clear`. Da solo, o anche con la sintassi equivalente `clear all`, esso cancella tutte le variabili presenti nel workspace mentre per ottenere una azione selettiva è necessario specificare il nome delle variabili separandone il nome con uno o più spazi.

```
>> clear s u
>> whos
  Name      Size          Bytes  Class

  ans       1x1               8  double array
  t         1x1               8  double array

Grand total is 2 elements using 16 bytes
```

Oltre alla variabile `ans` in `MATLAB` sono predefinite le varibili `pi = 3.1416, i = 0 + 1.0000i, j = 0 + 1.0000i, eps = 2.2204e-16`. La variabile `pi` è ovviamente una approssimazione di π, mentre `i` e `j` rappresentano entrambe l'unità immaginaria $\sqrt{-1}$: `MATLAB` è infatti in grado di operare in campo complesso, quando è il caso. Ad esempio:

```
>> sqrt(-4)

ans =

        0 + 2.0000i
```

La radice quadrata di (-2) è definita solo in campo complesso e MATLAB dà come risultato la prima radice. Per definire un numero complesso, ad esempio il numero $c = 2+3i$, basta quindi scrivere

```
>> c = 2+3*i

c =

   2.0000 + 3.0000i

>> whos
  Name      Size          Bytes  Class

  ans       1x1              16  double array (complex)
  c         1x1              16  double array (complex)
  t         1x1               8  double array

Grand total is 3 elements using 40 bytes
```

Infine, eps è il cosiddetto epsilon-macchina. Precisamente esso è il più piccolo numero rappresentabile nell'unità aritmetica del calcolatore tale che la somma "macchina" con il numero 1 restituisca ancora un valore maggiore di 1. In pratica eps è un indice della qualità dell'approssimazione che si può ottenere con un dato software/hardware di calcolo. Si rimanda a [3], Cap.2, per ulteriori dettagli sulla definizione precisa dell'epsilon-macchina e sulle implicazioni del suo valore numerico nel calcolo scientifico.

Si faccia inoltre attenzione al fatto che tutte le variabili, anche quelle predefinite, possono essere sovrascritte. In particolare possono essere ridefinite le variabili i e j che, per tradizione, sono spesso utilizzate come variabili indice (a valori interi) in fase di programmazione. Nel caso una variabile predefinita fosse sovrascritta, per ripristinare il valore originale si usa clear

```
>> i = 2

i =

     2

>> clear i
>> i

ans =

        0 + 1.0000i
```

La rappresentazione dei numeri in MATLAB è codificata secondo lo standard internazionale IEEE. Tale standard permette anche di gestire delle situazioni numeriche che, da un

punto di vista matematico, non sono definite in modo rigoroso. In particolare operazioni come 0.0/0.0 e $\infty - \infty$ danno come risultato `NaN` *Not-a-Number* mentre operazioni come 1.0/0.0 e e^{1000} danno come risultato `Inf`. Da una parte questa convenzione permette di evitare situazioni che sono pericolose in altri ambienti di programmazione, dove causano ad esempio il blocco della macchina o la generazione di un file di grandi dimensioni su cui viene riversato il contenuto della memoria (detto `core` file). D'altro canto lo svolgimento corretto dei calcoli dal punto di vista dell'utente è comunque compromesso nel caso intervengano esplicitamente `NaN` e `Inf` in operazioni successive, in quanto tali valori segnalano il verificarsi di una situazione critica e danno luogo a risultati non utilizzabili nella pratica.

Vediamo ora come è possibile definire dei vettori. La modalità più immediata consiste nell'elencare le singole componenti del vettore fra una coppia di parentesi quadre. Naturalmente questa procedura è comoda solo se il numero delle componenti è modesto.

```
>> b = [1 2 3 4]

b =

     1     2     3     4

>> whos
  Name      Size          Bytes  Class

  ans       1x1               8  double array
  b         1x4              32  double array
  c         1x1              16  double array (complex)
  t         1x1               8  double array

Grand total is 7 elements using 64 bytes
```

Si osservi che il vettore creato in questo modo è di tipo riga. In modo alternativo si possono separare le componenti del vettore con il carattere virgola ",". Usando invece il carattere punto-e-virgola si ottiene un vettore colonna.

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

b =

     1
     2
     3
     4

>> whos
Name      Size          Bytes  Class

  ans       1x1               8  double array
  b         4x1              32  double array
  c         1x1              16  double array (complex)
  t         1x1               8  double array
```

```
Grand total is 7 elements using 64 bytes
```

Esiste un modo rapido per creare dei vettori quando le componenti seguono una progressione aritmetica. In tale caso si usa il carattere due-punti ":" omettendo eventualmente le parentesi quadre. Il formato generale segue la sintassi `inizio:incremento:fine` dove `incremento` può anche essere un numero negativo e in tale caso `fine` dovrà essere minore di `inizio`. Nel caso in cui `incremento` sia uguale a 1 la sintassi si può semplificare in `inizio:fine`. Vediamo alcuni esempi

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

b =

     1     2     3     4     5     6     7     8     9    10

>>  b = [1:2:10]

b =

     1     3     5     7     9

>> b = [10:-3:0]

b =

    10     7     4     1

>>  b = [1:1.5:10]

b =

    1.0000    2.5000    4.0000    5.5000    7.0000    8.5000   10.0000
```

Si osservi che l'incremento può essere un numero non intero e che i valori della progressione che oltrepassano, inferiormente o superiormente, uno dei due estremi specificati, non vengono considerati. Come si vedrà nel paragrafo 5, un uso molto frequente di questo comando è la costruzione di un vettore che approssima un intervallo appartenente all'asse delle ascisse nello studio di funzioni per via grafica.

La costruzione delle matrici segue un formato simile. Ad esempio

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

m =

     1     2
     3     4
```

definisce una matrice quadrata di ordine 2. In generale, per la costruzione, si partiziona la matrice in vettori riga e si procede riga per riga, usando come separatore di ogni riga il punto-e-virgola. Vale naturalmente la regola che, in tutte le righe, il numero di componenti deve essere lo stesso.

```
>> m = [1 2; 3 4 5]
??? m = [1 2; 3 4 5]
                    |
All rows in the bracketed expression must have the same
number of columns.
```

Nel caso ciò non avvenisse MATLAB scrive un messaggio d'errore per avvisare l'utente di tale circostanza. Va sempre tenuto presente che, una volta definita, una matrice non è altro che una collezione di vettori riga ovvero colonna. Questo è importante sia in fase di costruzione che per l'accesso a insiemi di componenti della matrice stessa.

Vediamo ora come si estraggono gli elementi di un vettore o di una matrice. Cominciamo dai vettori.

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

b =

     1     2     3     4     5     6     7     8     9    10

>> b(5)

ans =

     5
```

Nell'esempio si è estratta la quinta componente del vettore b.

```
>> b(1:3)

ans =

     1     2     3
```

Ora si sono estratte le prime tre componenti. Si noti che si è fatto uso del carattere due-punti per costruire un vettore di indici da 1 a 3 con passo 1.

```
>> b([1:2,6,9:10])

ans =

     1     2     6     9    10
```

Qui si sono estratte le prime due componenti, la sesta, la nona e la decima. Si osservi l'uso combinato dei caratteri virgola e due-punti, unito a quello delle parentesi quadre, per costruire un vettore riga di indici.

Nel caso delle matrici, occorre invece utilizzare una coppia di indici, ciascuno dei quali può essere un vettore di interi.

```
>> m = [1:5; 6:10; 11:15; 16:20]

m =
```

```
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
```

Abbiamo definito una matrice di dimensioni 4×5 tramite il carattere due-punti. Vediamo alcuni esempi.
Estrazione di un elemento:

```
>> m(2,3)

ans =

     8
```

Estrazione di una riga:

```
>> m(1,:)

ans =

     1     2     3     4     5
```

Si osservi che il carattere due-punti da solo equivale ad esaurire tutte le colonne (in questo caso), dalla prima all'ultima.
Estrazione di una colonna:

```
>> m(:,3)

ans =

     3
     8
    13
    18
```

Estrazione di più righe e una sola colonna:

```
>> m([1:2,4],3)

ans =

     3
     8
    18
```

Estrazione di una riga e di più colonne:

```
>> m(2,[2,4])

ans =

     7     9
```

Estrazione di più righe e colonne:

```
>> m(2:4,1:2:5)

ans =

     6     8    10
    11    13    15
    16    18    20
```

Infine si osservi quest'ultimo comando

```
>> m(:)

ans =

     1
     6
    11
    16
     2
     7
    12
    17
     3
     8
    13
    18
     4
     9
    14
    19
     5
    10
    15
    20
```

Tutti gli elementi della matrice vengono estratti colonna per colonna e raccolti in un vettore colonna di dimensione $m \times n$.

Esempio: Calcolo dei determinanti dei minori principali di una matrice

In questo esempio si vogliono applicare le modalità di accesso agli elementi di una matrice al calcolo dei determinanti dei minori principali di una data matrice.

```
>> format rat
>> a = hilb(4)

a =

      1           1/2          1/3          1/4
     1/2          1/3          1/4          1/5
```

```
    1/3          1/4          1/5          1/6
    1/4          1/5          1/6          1/7

>> m1 = a(1,1);     dm1 = det(m1);
>> m2 = a(1:2,1:2); dm2 = det(m2);
>> m3 = a(1:3,1:3); dm3 = det(m3);
>> m4 = a(1:4,1:4); dm4 = det(m4);
```

Facciamo alcune osservazioni. Il primo comando `format` imposta la visualizzazione dei valori numerici e in particolare con l'opzione `rat` tutti i numeri appariranno in un formato razionale, cioè espressi come rapporto di due numeri interi. Tale formato è stato scelto in quanto la matrice poi costruita ben si presta ad essere visualizzata in questo modo. Per costruire la matrice si utilizza il comando `hilb` che costruisce la cosiddetta matrice di Hilbert. Questa è una matrice quadrata di dimensione arbitraria il cui elemento h_{ij} di indici (i, j) è definito come $h_{ij} = \frac{1}{i+j-1}$. Per l'estrazione dei vari minori si è utilizzato opportunamente il carattere due-punti per effettuare un accesso multiplo a righe e colonne. Infine per il calcolo del determinante si è utilizzato il comando `det`.

Osservazione
I comandi `det` e `hilb` sono esempi di quelle che in ambiente `MATLAB` sono chiamate *functions*. Rimandiamo al paragrafo 4 per una descrizione dettagliata delle functions. Qui basti dire che le variabili o i valori presenti all'interno delle parentesi tonde dopo il nome della function rappresentano i parametri sui cui essa opera stessa, mentre il risultato della function viene assegnato al parametro a sinistra dell'uguale. Ad esempio, il comando `a = hilb(n)` costruisce la matrice di Hilbert di dimensione pari al valore (intero) assunto dalla variabile `n` e lo assegna alla variabile `a`.

Procediamo con l'elaborazione delle matrici. In particolare vediamo come è possibile costruire le matrici a blocchi, cioè delle matrici che possono essere partizionate in sotto-strutture che a loro volta sono matrici. In altri termini, una matrice "standard" A è una matrice in cui i singoli blocchi costituenti sono gli elementi a_{ij}, cioè matrici particolari di dimensione 1; una matrice a blocchi è invece costituita da "macro elementi" che possono essere a loro volta matrici (rettangolari) di dimensione qualsiasi, con l'unico vincolo che la matrice risultante abbia tutte le righe (o colonne) con lo stesso numero di elementi. Ovviamente tutte le matrici sono standard ma spesso, l'applicazione stessa, fisica, ingegneristica, economica, etc., suggerisce in modo naturale una partizione in blocchi della matrice. La matrice seguente A è un esempio di matrice di ordine 5 partizionata in quattro blocchi che, procedendo da sinistra verso destra e dal basso verso l'alto, sono (sotto) matrici di dimensione 2×2, 2×3, 3×2 e 3×3.

$$A = \left[\begin{array}{cc|ccc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{array}\right]$$

Per assemblare una matrice a partire dai suoi blocchi si procede nel seguente modo

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

```
B =

     1     2
     3     4

>> C = [5 6; 7 8]

C =

     5     6
     7     8

>> A = [B, C]

A =

     1     2     5     6
     3     4     7     8

>> A = [B; C]

A =

     1     2
     3     4
     5     6
     7     8

>> A = [B, C; C, B]

A =

     1     2     5     6
     3     4     7     8
     5     6     1     2
     7     8     3     4
```

Si osservi che la sintassi è indipendente dalle dimensioni delle variabili B e C: è la stessa sia che esse siano matrici scalari 1×1 che matrici vere e proprie di dimensione maggiore. In quest'ultimo caso l'unica avvertenza è che le dimensioni delle matrici siano compatibili, cioè se due matrici sono affiancate devono avere lo stesso numero di righe, mentre se sono incolonnate una sopra l'altra devono avere lo stesso numero di colonne.

3.1 Operazioni su scalari

Le principali operazioni su variabili di tipo scalare, cioè matrici di dimensione 1, sono le consuete operazioni di somma, sottrazione, moltiplicazione, divisione e elevamento a potenza. Esse si ottengono in MATLAB con gli operatori "+", "−", "∗", "/"e "^". In più è definita l'operazione di divisione a sinistra "\", definita in modo tale che $a\backslash b = a^{-1}\, b$.

L'utilità di questa operazione si apprezzerà considerando la sua generalizzazione al caso delle matrici. Per i numeri complessi è definita l'operazione di coniugazione che si ottiene tramite l'operatore " ' ", ad esempio

```
>> (2 + 4*i)'

ans =
       2.0000 - 4.0000i
```

3.2 Operazioni su vettori

I vettori in MATLAB sono una rappresentazione degli enti vettoriali matematici che appartengono allo spazio delle ennuple di numeri reali $\mathbb{R}^n$ o complessi $\mathbb{C}^n$. Sono pertanto definite le operazioni di somma e sottrazione con l'effetto di operare elemento per elemento, " + " e " − " rispettivamente.

```
>> b = 1:4;
>> c = 2:5;
>> b+c

ans =

      3            5            7            9

>> c - b

ans =

      1            1            1            1
```

L'unica avvertenza è che i vettori abbiano le stesse dimensioni sia in termini di righe che di colonne. Non si posso quindi sommare o sottrarre vettori riga e colonna fra loro nemmeno se hanno lo stesso numero di componenti. Come caso particolare, una variabile scalare può essere sempre sommata o sottratta ad un vettore con il significato che essa viene sommata o sottratta a ciascuna delle componenti del vettore. Sono definite poi l'operazione di moltiplicazione per uno scalare, sempre con il significato di moltiplicare ogni elemento per lo scalare, e di moltiplicazione fra vettori. In quest'ultimo caso `a*b` rispetta la definizione di prodotto riga-colonna definito in generale per le matrici ed è quindi legittimo solo in due casi: quando `a` è un vettore riga e `b` un vettore colonna con lo stesso numero di componenti e il risultato è uno scalare, il cosiddetto *prodotto scalare* o quando `a` è un vettore colonna $m \times 1$ e `b` un vettore riga $1 \times n$ e il risultato è una matrice di dimensione $m \times n$, il cosiddetto *prodotto di Kronecker*.

```
>> b*c'

ans =

     40

>> b'*c
```

```
ans =

     2     3     4     5
     4     6     8    10
     6     9    12    15
     8    12    16    20
```

Nell'esempio si è fatto uso del carattere "' "che corrisponde all'operazione di trasposizione. Ricordiamo che il trasposto di un vettore riga è il vettore colonna che ha le stesse componenti. Nel caso di vettori a componenti complesse lo stesso carattere corrisponde all'operazione di coniugazione componente per componente. Nel caso in cui si volesse solo trasporre senza coniugare si usa ".' ".

3.2.1 Operazioni elemento per elemento su vettori

MATLAB estende le proprietà delle operazioni tipo somma e sottrazione, che per definizione agiscono elemento per elemento, anche alle operazioni di moltiplicazione, divisione, divisione a sinistra e elevamento a potenza. Tale effetto si ottiene facendo precedere il carattere che identifica il tipo di operazione desiderato dal carattere punto ". ". Ad esempio l'operazione ".* " corrisponde alla moltiplicazione elemento-elemento. In generale dati i vettori $a, b \in \mathbb{R}^n$, si ha che a .op b è il vettore c di $\mathbb{R}^n$ le cui componenti sono $c_i = a_i$ op b_i, $i = 1, \cdots, n$ ove op può essere "$*$", "/", "\"o " ^". Il vincolo è che i due vettori operandi abbiano lo stesso numero di componenti riga e colonna. Ad esempio

```
>> a = 1:3;
>> b = a;
>> a.*b

ans =

     1     4     9
```

genera un vettore la cui componente i-esima è $a_i * b_i$. In modo simile

```
>> a./b

ans =

     1     1     1

>> a.^b

ans =

     1     4    27


>> a.\(a.*a)

ans =
```

```
     1     2     3
```

L'uso di tali funzioni è molto frequente in quanto permette, ad esempio, di valutare una data funzione matematica su un insieme di punti. Ad esempio per valutare la funzione $y = 3x^2 + 1$ per i valori di x corrispondenti ai primi 10 numeri naturali diamo i seguenti comandi

```
>> x = 1:10;
>> y = 3*x.^2 + 1

y =

     4    13    28    49    76   109   148   193   244   301
```

Si osservi che le variabili scalari sono sempre compatibili con i vettori e che non è necessario usare per queste il "punto". Si intuisce come le operazioni elemento-elemento siano molto sfruttate per lo studio di una funzione su un dato intervallo tramite visualizzazione grafica, come si vedrà nel seguito.

3.3 Operazioni su matrici

Per quanto riguarda le operazioni su matrici si ha che la somma e la differenza agiscono già per definizione a livello di elemento per cui l'unico vincolo è che gli addendi abbiano lo stesso numero di righe e colonne. Ad esempio

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

ans =

     2     3     4
     6     7     8
    10    11    12

>> A-B

ans =

     0     1     2
     2     3     4
     4     5     6
```

La moltiplicazione fra matrici segue la regola usuale di moltiplicazione righe-colonne per cui se A e B sono due matrici di ordine $m \times n$ e $n \times p$ rispettivamente, il prodotto $A * B$ è la matrice C di dimensione $m \times p$ e di elementi $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.

```
>> A*B

ans =
```

```
    14     14     14
    32     32     32
    50     50     50
```

Speciale significato ha la divisione che, ricordiamo, *non* è definita sulle matrici. `MATLAB` ci permette però di eseguire le seguenti operazioni di divisione e divisione a sinistra `A/B` = `A*`inv(`B`) e `A\B` = inv(`A*B`).

```
A = hilb(3);
>> B = A + 1;
>> A/B

ans =

    0.7000    2.4000   -3.0000
   -0.3000    3.4000   -3.0000
   -0.3000    2.4000   -2.0000

>> A\B

ans =

    4.0000    3.0000    3.0000
  -24.0000  -23.0000  -24.0000
   30.0000   30.0000   31.0000
```

L'elevamento a potenza X^y è definito in particolare se X è una matrice quadrata e y è uno scalare intero e corrisponde alla moltiplicazione ripetuta di X con se stessa y volte.

```
>> A^2

ans =

    1.3611    0.7500    0.5250
    0.7500    0.4236    0.3000
    0.5250    0.3000    0.2136
```

Per altri usi di " ^ "si veda l'help del comando `mpower`.

3.3.1 Operazioni elemento-elemento su matrici

Valgono considerazioni analoghe a quelle fatte a proposito dei vettori. Si usano le operazioni "puntate".

Per informazioni dettagliate su tutte le operazioni si rimanda all'help di `ops`.

3.4 Funzioni intrinseche definite per vettori e matrici

In questa paragrafo illustriamo alcune delle principali funzioni disponibili che permettono di manipolare o agire su vettori e matrici.

- `eye(n)` costruisce la matrice identità di ordine n, cioè la matrice che ha elementi 1 sulla diagonale principale e 0 altrove.

- `ones(m,n)` costruisce una matrice o un vettore di dimensione $m \times n$ i cui elementi sono tutti 1.

- `zeros(m,n)` genera un vettore o una matrice con elementi tutti 0.

- `rand(m,n)` costruisce una matrice di ordine $m \times n$ i cui elementi sono numeri pseudo-casuali generati con una distribuzione uniforme di probabilità nell'intervallo (0,1).

Tutti questi comandi accettano anche un solo parametro di ingresso, ad esempio n, con il significato che le matrici generate sono quadrate di ordine n.

Indichiamo nel seguito con `M` e `v` una matrice $m \times n$ e un vettore di $\mathbb{R}^q$ rispettivamente, ad esempio

```
>> M = [1 2 4; 4 5 6; 7 8 9];
>> v = [100; 200; 300];
```

- `diag(v)` costruisce la matrice quadrata diagonale i cui elementi sono le componenti del vettore `v`

```
>> diag(v)

ans =

   100     0     0
     0   200     0
     0     0   300
```

- `diag(v,i)` costruisce una matrice in cui gli elementi della diagonale i-esima coincidono con le componenti del vettore `v`

```
>> diag(v,2)

ans =

     0     0   100     0     0
     0     0     0   200     0
     0     0     0     0   300
     0     0     0     0     0
     0     0     0     0     0
```

  Si osservi che le diagonali sono numerate assegnando alla diagonale principale il valore 0, alla sopradiagonale il valore 1 etc., alla prima sottodiagonale il valore -1 e così via. Il riferimento alla diagonale principale può essere omesso, come visto sopra.

- `diag(M,i)` estrae la diagonale i-esima dalla matrice `M` e genera un corrispondente vettore colonna.

```
>> diag(M,-1)

ans =

     4
     8
```

- `size(M)` fornisce le dimensioni m e n della matrice `M`

```
>> size(M)

ans =

     3     3
```

- `size(v)` fornisce le dimensioni del vettore `v`

```
>> size(v)

ans =

     3     1
```

- `length(v)` restituisce la lunghezza del vettore `v`. È equivalente al comando `max(size(v))`, ovvero restituisce la massima fra le dimensioni di `v`

```
>> length(v)

ans =

      3
```

- `tril(M,i)` costruisce la matrice triangolare inferiore estratta da `M` a partire dalla i-esima diagonale

```
>> tril(M)

ans =

     1     0     0
     4     5     0
     7     8     9

>> tril(M,1)

ans =

     1     2     0
     4     5     6
     7     8     9
```

- `triu(M,i)` costruisce la matrice triangolare superiore estratta da `M` a partire dalla i-esima diagonale

```
>> triu(M)

ans =

     1     2     4
     0     5     6
     0     0     9

>> triu(M,-1)

ans =

     1     2     4
     4     5     6
     0     8     9
```

Esempio: Costruzione di una matrice con struttura per diagonali

Si voglia costruire la seguente *matrice tridiagonale*

$$A = \begin{bmatrix} 4 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 4 \end{bmatrix}.$$

Utilizziamo opportunamente i comandi visti

```
>> A = 4*eye(4) + diag(ones(1,3),1) + diag(ones(1,3),-1)

A =

     4     1     0     0
     1     4     1     0
     0     1     4     1
     0     0     1     4
```

Riportiamo nel seguito un elenco di *built-in functions* di grande utilità nel trattamento di vettori e matrici in Algebra Lineare. Per i dettagli sulle definizioni e proprietà rimandiamo a [3], Capitolo 1.

- `max(v), min(v)` calcola il massimo (minimo) valore delle componenti di un dato vettore `v` (considerando le componenti con il loro segno, non in modulo)
- `[m,i] = max(v), [m,i] = min(v)` calcola il massimo (minimo) valore `m` delle componenti di `v` e l'indice `i` della componente ad essa associato.

```
>> [m,i] = min(v)

m =

   100


i =

     1
```

- `sum(v)` calcola la somma degli elementi di `v`

```
>> sum(v)

ans =

   600
```

- `prod(v)` calcola il prodotto degli elementi di `v`

```
>> prod(v)

ans =

     6000000
```

- `sort(v)` ordina gli elementi di `v` in ordine crescente

```
>> sort([1 4 3 2])

ans =

     1     2     3     4
```

- `max(M)`, `min(M)` calcola il massimo (minimo) valore degli elementi di una data matrice `M` per ogni colonna e genera un vettore riga

```
>> min(M)

ans =

     1     2     4
```

- `[m,i] = max(M)`, `[m,i] = min(M)` calcola il massimo (minimo) valore `m` degli elementi di `M` lungo ogni colonna e l'indice di riga corrispondente. Sia `m` che `i` sono vettori riga.

```
>> [m,i] = max(M)

m =

     7     8     9


i =

     3     3     3
```

- `sum(M)` costruisce un vettore riga di cui la generica componente j-esima è la somma degli elementi della colonna j-esima di `M`

```
>> sum(M)

ans =

    12    15    19
```

- `prod(M)` calcola il prodotto degli elementi di ciascuna colonna

```
>> prod(M)

ans =

    28    80   216
```

- `sort(M)` ordina gli elementi delle colonne di `M` in ordine crescente
- `det(M)` calcola il determinante della matrice `M`

```
>> det(M)

ans =

    -3
```

- `rank(M)` calcola il rango della matrice `M` cioè il massimo numero di colonne linearmente indipendenti

```
>> rank(M)

ans =

     3
```

- `null(M)` calcola una base ortonormale del nucleo della matrice `M`, indicato in Algebra Lineare come ker(`M`) e definito come l'insieme dei vettori $\mathbf{x}$ non nulli tali che $M\mathbf{x} = \mathbf{0}$. Se $M \in \mathbb{R}^{m\times n}$ si ha

$$\texttt{rank(M)} + \dim(\texttt{null(M)}) = n.$$

  Nel caso in cui `M` sia non singolare (ovvero `det(M)` $\neq 0$) l'insieme `null(M)` è vuoto.

```
>> null(M)

ans =

   Empty matrix: 3-by-0
```

Consideriamo invece la matrice singolare `N` per la quale si ha dim(`null(N)`)=1.

```
>> N = [1 2 -3; 4 -2 -2; -2 5 -3];
>> det(N)

ans =

    0

>> null(N)

ans =

    0.5774
    0.5774
    0.5774
```

In effetti, la verifica del calcolo del rango di `N` fornisce

```
>> rank(N)

ans =

     2
```

- `orth(M)` calcola una base ortonormale dell'immagine della matrice `M` cioè dell'insieme dei vettori $\mathbf{y}$: $\mathbf{y} = M\mathbf{x}$ che si ottengono al variare di $\mathbf{x}$. In altre parole `orth(M)` è lo spazio generato dalle colonne di `M`.

```
>> orth(M)

ans =

    0.2532    0.9345    0.2504
    0.5159    0.0885   -0.8520
    0.8184   -0.3449    0.4597

>> orth(N)

ans =
```

```
   -0.3629    0.5315
    0.3251    0.8420
   -0.8733    0.0926
```

- `eig(M)` calcola gli autovalori della matrice `M`

```
>> eig(M)

ans =

   16.5038
   -1.6163
    0.1125
```

- `[V,D] = eig(M)` calcola gli autovalori e gli autovettori di `M`. Precisamente, si ha che $\texttt{V}^{-1}$ `M V = D`, dove `V` è la matrice le cui colonne sono gli autovettori di `M` mentre `D` è la matrice diagonale degli autovalori di `M`.

```
>> [V,D] = eig(M)

V =

   -0.2757  -0.8408   0.5560
   -0.5181   0.0176  -0.7862
   -0.8096   0.5411   0.2697

D =

   16.5038        0        0
         0  -1.6163        0
         0        0   0.1125
```

- `trace(M)` calcola la traccia di `M` cioè la somma degli elementi sulla diagonale principale

```
>> trace(M)

ans =

    15
```

- `inv(M)` calcola la matrice inversa di `M`

```
>> inv(M)

ans =

    1.0000   -4.6667    2.6667
   -2.0000    6.3333   -3.3333
    1.0000   -2.0000    1.0000
```

- `M\v` risolve il sistema lineare $M\mathbf{x} = \mathbf{v}$ con il metodo di eliminazione di Gauss

```
>> M\v

ans =

  -33.3333
   66.6667
    0.0000
```

Osservazione Si fa presente che la risoluzione di un sistema lineare del tipo $A\mathbf{x} = \mathbf{b}$ equivale formalmente a calcolare la soluzione $\mathbf{x}$ come $\mathbf{x} = A^{-1}\mathbf{b}$. Tuttavia, da un punto di vista pratico si consiglia *vivamente* di usare la sintassi `MATLAB x = A\b`. Si veda, ad esempio, [3], Cap.3, per le motivazioni di questa scelta.

- `norm(v)` calcola la norma 2 del vettore `v`: $\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{q} v_i^2}$

```
>> norm(v)

ans =

  374.1657
```

- `norm(v,1)` calcola la norma 1 di `v`: $\|\mathbf{v}\|_1 = \sum_{i=1}^{q} |v_i|$

```
>> norm(v,1)

ans =

   600
```

- `norm(v,inf)` calcola la norma infinito di `v`: $\|\mathbf{v}\|_\infty = \max_{1 \le i \le q} |v_i|$

```
>> norm(v,inf)

ans =

   300
```

- `norm(M)` calcola la norma 2 della matrice `M` (è equivalente al comando `norm(M,2)`): $\|M\|_2 = \sqrt{\rho(M^T M)}$ dove, indicati con $\{\lambda_i(X)\}_{i=1}^{n}$ gli autovalori di una generica matrice $X \in \mathbb{R}^{n \times n}$, si definisce *raggio spettrale* di X la quantità $\rho(X) = \max_{1 \le i \le n} |\lambda_i(X)|$.

```
>> norm(M)

ans =

   17.0048
```

- `norm(M,1)` calcola la norma 1 di `M`: $\|M\|_1 = \max_{1\le j\le n}\sum_{i=1}^{m}|m_{ij}|$ cioè il massimo valore delle somme per colonna dei moduli degli elementi di `M`

  ```
  >> norm(M,1)

  ans =

      19
  ```

- `norm(M,inf)` calcola la norma infinito di `M`: $\|M\|_1 = \max_{1\le i\le m}\sum_{j=1}^{n}|m_{ij}|$ cioè il massimo valore delle somme per righe dei moduli degli elementi di `M`

  ```
  >> norm(M,inf)

  ans =

      24
  ```

- `cond(M)` calcola il numero di condizionamento in norma 2 della matrice `M`. Sono anche disponibili `cond(M,1)` e `cond(M,inf)` per il calcolo del numero di condizionamento rispetto alle norme 1 e infinito.

Infine il comando `flops` visualizza il numero di operazioni floating point effettuate a partire dall'inizio della sessione di lavoro, intendendo con ciò le principali operazioni di somma, sottrazione, moltiplicazione, etc. Ogni somma e sottrazione richiede 1 flop; la moltiplicazione e la divisione richiedono entrambe 1 flop se il risultato è reale e 6 flops se è complesso; le funzioni elementari richiedono 1 flop se il risultato è reale mentre richiedono più flops se esso è un numero complesso. Per dare un'idea degli ordini di grandezza si ha che, se A e B sono matrici reali $n \times n$

- $A + B$ richiede n^2 flops
- $A * B$ richiede $2n^3$ flops
- $LU(A)$ (fattorizzazione LU di A) richiede circa $2n^3/3$ flops

Per reinizializzare il contatore si usa il comando `flops(0)`. Ciò è utile qualora si voglia stimare la complessità computazionale di un particolare algoritmo o insieme di espressioni: è sufficiente far precedere l'algoritmo dal comando `flops(0)` e terminare con `flops`

```
>> flops(0)
```

Espressioni .....

```
>> flops
```

Vediamo alcuni esempi

```
>> flops(0)
>> M+M;
>> flops

ans =

     9

>> flops(0)
>> M*M;
>> flops

ans =

    54

>> flops(0)
>> sin(pi);
>> flops

ans =

     1

>> flops(0)
>> sqrt(-2);
>> flops

ans =

    10
```

4 La programmazione in MATLAB

Oltre ad utilizzare MATLAB in modo interattivo richiamando funzioni "preconfezionate", l'utente può anche sfruttare le potenzialità dell'ambiente di calcolo come linguaggio di programmazione vero e proprio. In questo paragrafo ci occuperemo quindi di descrivere brevemente gli elementi di base di programmazione in ambiente MATLAB.

Come già accennato, la differenza fondamentale tra MATLAB e gli altri linguaggi comunemente impiegati (C, Fortran, ecc...) consiste nel fatto che MATLAB è un *linguaggio interpretato* e non compilato: non viene cioè creato un file eseguibile a partire dal sorgente, ma il codice viene eseguito riga per riga, interpretando ogni singola istruzione. Ciò implica che MATLAB non è certamente adatto per scrivere codici a cui sono richieste prestazioni elevate in termini di tempo di calcolo. È disponibile un traduttore integrato nell'ambiente di calcolo che consente di generare un codice C partendo da un sorgente MATLAB: questo permette ad esempio di creare delle librerie (fra cui le *dll libraries* di Windows) dedicate ad un certo argomento (per maggiori informazioni si rimanda alla documentazione presente nel sito `www.mathworks.com`).

Qualunque editor di testo può essere impiegato per scrivere le righe di codice, tuttavia gli utenti Windows dispongono di un editor integrato direttamente in MATLAB di cui è sempre consigliabile fare uso. Esso consente in modo immediato di scrivere, salvare, modificare i file da eseguire. Per iniziare la stesura di un nuovo file, bisogna selezionare con il mouse dall'ambiente di calcolo la voce **File** e dal menu a tendina che compare la voce **Nuovo** e successivamente l'opzione **M-file**, come mostrato in Figura 3. A questo punto è possibile comporre il testo del file come in un qualunque editor, come mostrato in Figura 4.

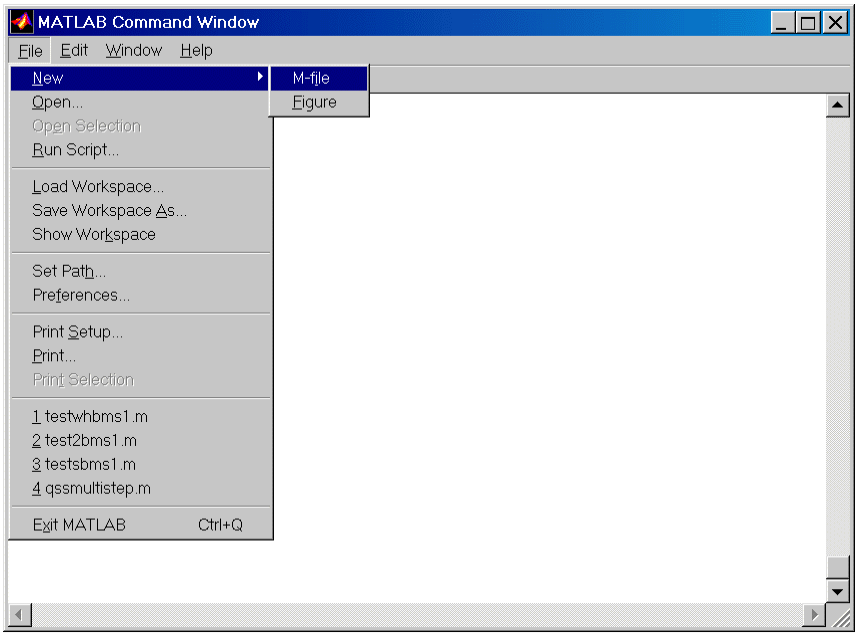


Figura 3: Apertura di un nuovo file con l'editor integrato di MATLAB

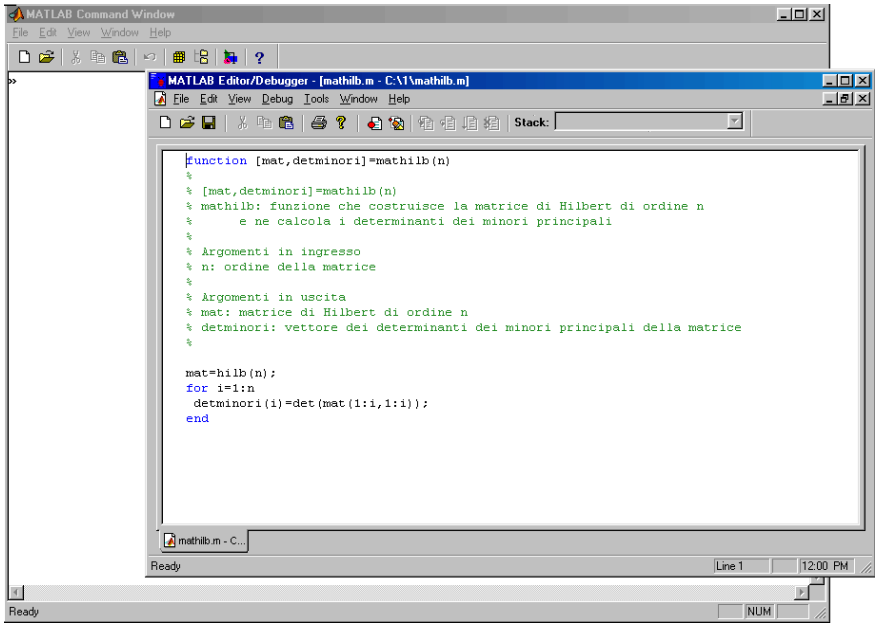


Figura 4: Stesura di una file .m: oltre a tutte le funzioni di un normale editor di testo, l'editor integrato offre alcune funzioni specifiche, fra cui ricordiamo l'utilità di debug

4.1 M-files: Functions e Scripts

I files che contengono codice `MATLAB` vengono chiamati *M-files* e la loro estensione *deve* essere *.m*. Gli *M-files* si dividono in due categorie, le *functions* e gli *scripts*. Vediamo in dettaglio quali sono le principali differenze tra questi due tipi di file.

4.1.1 Functions

Le caratteristiche principali delle functions sono:

- accettano argomenti in ingresso e possono restituire argomenti in uscita;
- le variabili interne alla function sono locali.

Supponiamo ad esempio di voler scrivere una function che costruisca la matrice di Hilbert di ordine `n` e ne calcoli i determinanti dei minori principali. Per prima cosa generiamo con l'editor di `MATLAB` il file `mathilb.m`. La prima riga della function costituisce l'intestazione della function stessa e ne definisce il nome e gli argomenti di input e di output:

```
function [mat,detminori]=mathilb(n)
```

Le variabili `mat` e `detminori` sono i parametri in uscita mentre la variabile `n` è il parametro in ingresso. In generale l'intestazione di una function ha la seguente sintassi:

function [out1, out2, out3, ...]= Nome-function(in1, in2, in3, ...)

Per semplificare l'utilizzo, è bene che il file abbia nome *Nome-function*, ovvero lo stesso nome della function in esso contenuta; inoltre, come già anticipato esso *deve* avere estensione .m.
Subito dopo l'intestazione è buona norma inserire alcune righe di commento. Tali righe iniziano con il carattere `%` e sono mostrate da `MATLAB` quando viene richiamato l'help della funzione. Nel nostro caso aggiungiamo le seguenti righe:

```
function [mat,detminori]=mathilb(n)
%
% [mat,detminori]=mathilb(n)
% mathilb: funzione che costruisce la matrice di Hilbert di ordine n
% e ne calcola i determinanti dei minori principali
%
% Argomenti in ingresso
% n: ordine della matrice
%
% Argomenti in uscita
% mat: matrice di Hilbert di ordine n
% detminori: vettore dei determinanti dei minori principali della matrice
%
```

Il risultato dell'istruzione:

```
>> help mathilb
```

è il seguente:

```
%
% [mat,detminori]=mathilb(n)
% mathilb: funzione che costruisce la matrice di Hilbert di ordine n
%      e ne calcola i determinanti dei minori principali
%
% Argomenti in ingresso
% n: ordine della matrice
%
% Argomenti in uscita
% mat: matrice di Hilbert di ordine n
% detminori: vettore dei determinanti dei minori principali della matrice
%
```

La parte di commenti è seguita dal corpo della funzione. La function completa è la seguente (spiegheremo nel seguito il significato preciso di ciascuna istruzione, per ora pensiamo alla function come ad una "scatola nera" che riceve in ingresso dei dati e restituisce delle quantità calcolate):

```
function [mat,detminori]=mathilb(n)
%
% [mat,detminori]=mathilb(n)
% mathilb: funzione che costruisce la matrice di Hilbert di ordine n
%      e ne calcola i determinanti dei minori principali
%
% Argomenti in ingresso
% n: ordine della matrice
%
% Argomenti in uscita
% mat: matrice di Hilbert di ordine n
% detminori: vettore dei determinanti dei minori principali della matrice
%
```

```
mat=hilb(n);
for i=1:n
 detminori(i)=det(mat(1:i,1:i));
end
```

Proviamo ad eseguire dal prompt di `MATLAB` la funzione `mathilb` (con `n=5`, ad esempio) digitando l'istruzione:

```
>> [a,determinanti]=mathilb(5);
```

Le variabili `a` e `determinanti`, che prima dell'esecuzione della function `mathilb` erano indefinite, ora contengono, rispettivamente, la matrice di Hilbert di ordine 5 ed il vettore dei determinanti dei suoi minori principali. Verifichiamo quanto detto con il comando `whos`:

```
>> whos
  Name               Size          Bytes    Class

  a                   5x5            200    double array
  determinanti        5x1             40    double array

Grand total is 30 elements using 240 bytes
```

Osserviamo che la variabile `i` è una variabile *locale* alla sola funzione `mathilb` e pertanto, quando il controllo viene restituito al prompt, di essa non rimane traccia in memoria e dunque non è riportata dal comando `whos` (si dice quindi che la sua *visibilità* è locale).

4.1.2 Script

Riscriviamo ora l'esempio precedente sotto forma di *script* e salviamolo con il nome `mathilb2.m`

```
% script per il calcolo dei determinanti dei minori principali
% della matrice di Hilbert di ordine n
n=5;
a=hilb(n);
for i=1:n
 determinanti(i)=det(a(1:i,1:i));
end
```

Osserviamo questa volta che non sono più presenti parametri in ingresso e in uscita come per le function. Proviamo ad eseguire lo script, semplicemente richiamandone il nome:

```
>> mathilb2
```

Analogamente a prima controlliamo quali sono le variabili attive dopo l'esecuzione del programma:

```
>> whos
  Name               Size          Bytes  Class
```

```
a                5x5           200  double array
i                1x1             8  double array
determinanti     5x1            40  double array
n                1x1             8  double array

Grand total is 32 elements using 256 bytes
```

Come si può notare, in questo caso *tutte* le variabili usate dal programma sono ancora attive dopo l'esecuzione del programma stesso, si dice cioè che la loro visibilità è globale. A loro volta, anche tutte le variabili definite dal prompt di MATLAB *prima* di eseguire lo script sono visibili dallo script stesso e possono essere utilizzate.

Quale è la logica alla base della scelta fra una function o uno script? La function corrisponde ad uno stile di programmazione modulare che riconduce la risoluzione di un problema di arbitraria complessità alla risoluzione di sottoproblemi di complessità inferiore. Lo script è invece una sorta di "quaderno degli appunti" dove scrivere in modo estemporaneo brevi programmi che possono essere modificati senza dover digitare di nuovo ad ogni modifica tutte le istruzioni dal prompt. È in questo senso che esso deve essere sfruttato. Qualora invece si voglia procedere alla stesura di un codice complesso, destinato a utenti che non necessariamente debbano conoscere tutti i dettagli dell'implementazione, è sicuramente consigliabile utilizzare delle functions.

Vediamo ora più in dettaglio le principali istruzioni e strutture sintattiche che permettono di scrivere un programma compiuto.

4.2 Operatori logici e di confronto

Introduciamo i fondamentali operatori di confronto e la loro codifica, osservando che in MATLAB tali operatori effettuano un confronto elemento per elemento fra due matrici (il confronto tra due scalari è quindi un caso particolare):

operatore	significato
==	uguale
<=	minore o uguale
>=	maggiore o uguale
<	minore stretto
>	maggiore stretto
~=	diverso

Tabella 1: Operatori di confronto

Nota. Si faccia attenzione alla differenza tra operatore (=) che *assegna* il valore a destra del simbolo alla variabile a sinistra del simbolo stesso e l'operatore(==) che invece *confronta* il valore a destra con quello a sinistra del simbolo.
Osserviamo inoltre che in MATLAB, analogamente al linguaggio C, non esistono i valori logici vero e falso (true-false), ma essi sono rappresentati da qualsiasi valore numerico diverso da zero (true) e dal valore zero (false).
Esempi.

```
>> 1==9

ans =

     0

>> a=hilb(4);
>> b=mathilb(4);
>> a==b

ans =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1

>> c=mathilb(4); c(1,:)=100;
>> a==c

ans =

     0     0     0     0
     1     1     1     1
     1     1     1     1
     1     1     1     1


>> 1~=1

ans =

     0

>> 1~=0

ans =

     1

>> 1>=9

ans =

     0
```

Gli operatori logici e la loro codifica in MATLAB sono riassunti nella seguente tabella:

Operatore	codifica	valore assunto
a AND b	&	vero quando a,b entrambe sono vere, falso in ogni altro caso
a OR b	|	falso quando a,b sono entrambe false, vero in ogni altro caso
NOT a	$\sim$	falso se l'espressione a è vera e viceversa

Tabella 2: Operatori logici

Nota. Qualora si usi una tastiera italiana, il simbolo $\sim$ si ottiene premendo AltGr+126.

Osserviamo che gli operatori logici hanno un livello di precedenza inferiore sia rispetto agli operatori aritmetici, che hanno precedenza massima, sia rispetto agli operatori di confronto che hanno un livello di precedenza intermedio. L'ordine delle precedenze può essere alterato facendo uso delle parentesi tonde, di cui è comunque sempre consigliabile fare uso per ragioni di leggibilità e chiarezza.

```
>> z=(3<4) & (8>5)

z =

     1

>> z=(3<4) | (4>6)

z =

     1

>> z=~(3<8 | 5>7)

z =

     0
```

4.3 Operazioni di input-output

Illustriamo una prima serie di istruzioni che consentono di gestire in modo immediato operazioni di input-output dalla finestra di calcolo. Definiamo tali istruzioni di "alto livello", perché non richiedono alcuna conoscenza delle regole di formattazione per l'acquisizione o la stampa delle variabili. La loro semplicità d'uso è tuttavia controbilanciata da una serie di limitazioni sulle funzioni che esse possono svolgere.
Introduciamo le seguenti istruzioni:

- `clc`: pulisce lo schermo e porta il cursore in alto a sinistra
- `disp`: stampa sullo schermo il contenuto di una variabile o di una stringa. Ad esempio:

```
>> disp('      pi greco '); disp(pi)
      pi greco
       3.1416
```

È necessario utilizzare un'istruzione `disp` *per ogni* variabile o stringa che si vuole stampare.

- `input`: acquisisce da tastiera un valore visualizzando un messaggio di richiesta e assegnando tale valore alla variabile specificata a sinistra. Ad esempio:

```
>> x=input('Numero di prove effettuate?   ')
Numero di prove effettuate?   10

x =

     10
```

- `pause`: interrompe l'esecuzione del programma ed attende che l'utente prema un qualsiasi tasto prima di continuare. Specificando `pause(n)` viene interrotta l'esecuzione del programma per un intervallo di `n` secondi.

Per gestire in modo più fine le operazioni di input-output esistono una serie di istruzioni che chiamiamo di "basso livello": esse richiedono la conoscenza di alcune regole di formattazione e delle convenzioni di nomenclatura dei flussi di dati in entrata ed uscita di dati (detti *streams*). Illustriamo qui di seguito le due istruzioni di più frequente uso:

- `fscanf`: legge da file secondo un formato specificato. La sintassi è:

  `var = fscanf` (identificatore file, formato lettura di ciascuna variabile)

  Ad esempio:

```
>> fid = fopen('c:/cerchio.txt','r');
>> raggio=fscanf(fid,'%f\n');
```

  Il comando `fopen` apre in lettura (`'r'`) il file `c:/cerchio.txt`. Nell'istruzione viene poi specificato tra apici il formato di lettura della variabile `raggio`, indicato con il simbolo `%`. Fra i formati di lettura ricordiamo:

  - %s : lettura di stringa
  - %d : lettura di variabile intera
  - %f : lettura di variabile in virgola fissa
  - %e : lettura di variabile in notazione esponenziale
  - %g : lettura

- `fprintf`: stampa su file secondo un formato specificato. La sintassi generale è:

  `fprintf` (identificatore file, formato scrittura di ciascuna variabile, elenco variabili)

  Il simbolo `\n` manda a capo la riga, facendo sì che la riga successiva venga scritta su una nuova linea. Ad esempio:

```
>> raggio=2; area=pi*raggio^2;
>> fid=fopen('c:/area.txt','w');
>> fprintf(fid,'L''area del cerchio di raggio %f vale %f\n',...
           raggio,area)
```

Il comando `fopen` apre in scrittura (`'w'`) un file che verrà chiamato `area.txt`. Dopo l'identificatore di formato nell'istruzione `fprintf` viene poi specificata tra apici la stringa da stampare (in cui osserviamo che l'apostrofo viene indicato con due apici), inserendo opportunamente le specificazioni di formato. Al primo specificatore di formato corrisponde la prima variabile elencata dopo gli apici, al secondo la seconda variabile e così via. L'istruzione `fprintf` permette anche di stampare in modo formattato sullo schermo, (detto *standard output*), che viene considerato alla stregua di un file normale, con l'unica particolarità di avere un identificatore `fid` posto per convenzione pari a 1. Ad esempio:

```
>> fid=1;
>> fprintf(fid,'L''area del cerchio di raggio %f vale %f\n',...
           raggio,area)

L'area del cerchio di raggio 2.000000 vale 12.566371
```

Notiamo che entrambe le funzioni `fscanf` e `fprintf` hanno una sintassi molto simile alle corrispondenti funzioni definite in linguaggio C. Per un panorama completo di tutte le istruzioni di input-output a basso livello è utile fare riferimento alla sezione dell'help "Low-level File I/O Functions" (`help iofun`).

4.4 Costrutti sintattici fondamentali

4.4.1 Ciclo `for`

L'istruzione `for` ripete per un *determinato* numero di volte un blocco di istruzioni . La forma generale dell'istruzione *for* è la seguente:

for <indice> = <start>:<step>:<end>
{blocco istruzioni} *end*

Esempio. Stampa di tutti i numeri dispari da 1 a 50:

```
for i=1:2:50
  disp(i)
end
```

Come abbiamo già osservato nel caso della costruzione di vettori, notiamo che, se non esplicitamente indicato, la variabile `step` vale 1.
È possibile anche definire cicli con incremento negativo: in tal caso `start` deve essere maggiore di `end` e `step`, se specificato, deve essere negativo.

Esempio. Stampa dei numeri da 1 a 2 con passo 0.1 in ordine inverso:

```
>> v=1:0.1:2;
>> for i= length(v):-1:1
      disp(v(i))
   end
```

Osserviamo infine che è possibile annidare più cicli `for`. Ad esempio per riprodurre il comportamento della built-in function `hilb` possiamo scrivere:

```
for i=1:n
   for j=1:n
      a(i,j)=1/(i+j-1);
   end
end
```

È sempre utile indentare opportunamente le righe di codice come in questo caso per rendere il listato più facilmente comprensibile.

4.4.2 Ciclo `while`

A differenza di quanto accade per l'istruzione `for`, l'istruzione `while` esegue un blocco di istruzioni un numero *indefinito* di volte fino al persistere di una certa condizione. Non è quindi noto a priori il numero di ripetizioni del blocco di istruzioni. La sintassi generale è:
while <condizione di persistenza nel ciclo >
{blocco istruzioni}
end

Esempio. Calcolo del massimo valore intero n tale che il suo fattoriale sia minore di `1e10`:

```
>> n=1;
>> while (prod(1:n)<1e10)
        n=n+1;
   end
>> disp(n)
```

Nota. Qualora, a causa di un errore di programmazione, il programma dovesse ripetere un numero infinito di volte il blocco di istruzioni perché la condizione di persistenza è sempre verificata, è possibile interrompere l'esecuzione del programma premendo contemporaneamente i tasti Ctrl+C.

4.5 Le istruzioni `if`, `else` e `elseif`

L'istruzione *if* controlla l'esecuzione di un determinato blocco di codice a seconda del valore (vero o falso) assunto da una certa espressione logica. La forma generale dell'istruzione *if* è la seguente:

if <expressione logica 1>
{blocco istruzioni 1}
elseif <expressione logica 2>
{blocco istruzioni 2}
else
{blocco istruzioni 3}
end

Esempio. Scriviamo un codice per il calcolo del fattoriale di un generico numero `n`:

```
n=input('Numero di cui calcolare il fattoriale >> ');
if n<0
  disp('n negativo')
elseif n==0
  fattoriale=1
else
  fattoriale=prod(1:n)
end
```

Si noti che le clausole `else` e `elseif` sono opzionali, cioè è possibile costruire blocchi più semplici di quello riportato sopra, ad esempio:

if <espressione>
    {blocco istruzioni}
end

oppure

if <espressione>
    {blocco istruzioni 1}
else
    {blocco istruzioni 2}
end

4.5.1 L'istruzione `break`

L'istruzione `break` termina forzatamente l'esecuzione di un ciclo `for` o di un ciclo `while`. Nel caso vi siano più cicli annidati `break` termina l'esecuzione del ciclo più interno.

Esempio. Consideriamo una reticolazione triangolare di un assegnato dominio $\overline{\Omega}$ del piano, come mostrato in Figura 5.

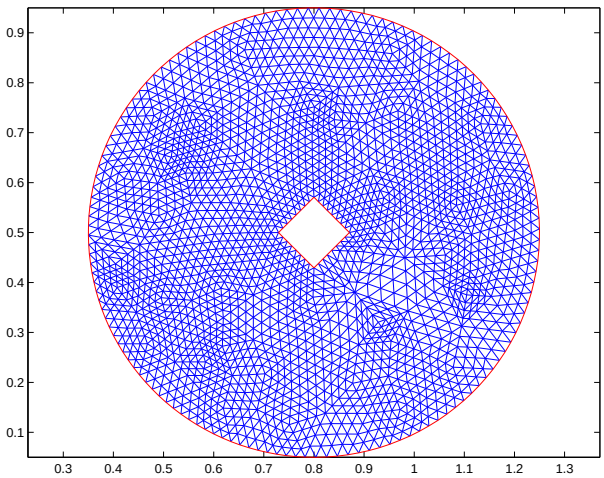


Figura 5: Reticolazione con elementi triangolari di un dominio $\overline{\Omega}$ in $\mathbb{R}^2$

Supponiamo di avere numerato i triangoli della reticolazione da 1 a `nelem` e di aver costruito una struttura di dimensioni `(nelem,3)` che chiamiamo `adiac` e che per ogni

triangolo contiene gli indici dei triangoli ad esso adiacenti. Assumiamo la convenzione per cui, se il $j$esimo lato del triangolo `ie` appartiene al bordo di $\overline{\Omega}$, il corrispondente valore in `adiac(ie,j)` è negativo.
Vogliamo ad esempio individuare gli indici dei triangoli che hanno *almeno* un lato appartenente al bordo; utilizziamo le seguenti righe di codice:

```
tr_bordo=[ ];
for ie=1:nelem
 for j=1:3
   if(adiac(ie,j)<0)
      tr_bordo=[tr_bordo;ie];
      break
   end
 end
end
```

In questo caso l'istruzione *break* permette di interrompere l'esecuzione del ciclo *for* più interno appena viene trovato in `adiac` un valore negativo (non ha senso infatti completare il ciclo più interno quando è stato stabilito che l'elemento ha almeno un lato di bordo!)

5 La grafica in MATLAB

MATLAB permette di creare in modo molto semplice grafici bidimensionali e tridimensionali e di corredare tali grafici con una serie di annotazioni utili a renderli chiari ed esplicativi.

5.1 Grafici in due dimensioni

Per iniziare, tracciamo nel piano xy il grafico di una funzione $f(x)$ per un intervallo assegnato di valori della variabile x utilizzando il comando `plot` nella sua forma più elementare. Supponiamo che sia assegnata la funzione $f(x) = \sin(x)$ nell'intervallo $[0, 2\pi]$:

```
>> x=0:pi/100:2*pi;
>> y=sin(x);
>> plot(x,y);
>> grid
```

Abbiamo usato il comando `plot` specificando dapprima il nome del vettore da rappresentare in ascissa (ovvero x) e poi quello da rappresentare in ordinata (ovvero y). Il comando `grid` traccia sul grafico una griglia di riferimento.

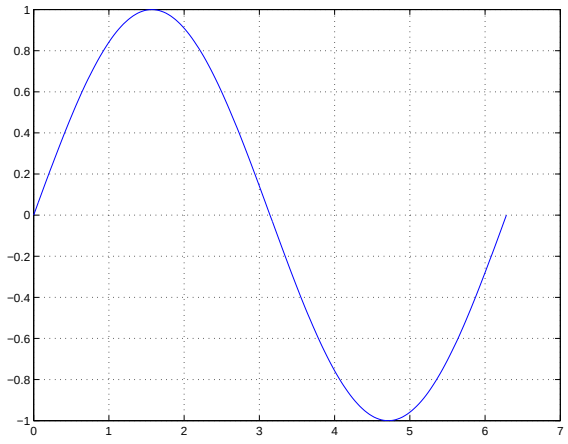


Figura 6: Grafico della funzione $y = \sin(x)$

Specificando invece solo il nome di una variabile, ad esempio `plot(x)`, viene rappresentata tale variabile in ordinata rispetto ad un'ascissa numerata progressivamente da 1 alla lunghezza del vettore stesso. È possibile rappresentare nello stesso grafico più di una curva che MATLAB contrassegna automaticamente con colori differenti (si veda la Figura 7):

```
>> y1=sin(x);
>> y2=sin(2*x);
>> y3=sin(3*x);
>> plot(x,y1,x,y2,x,y3)
```

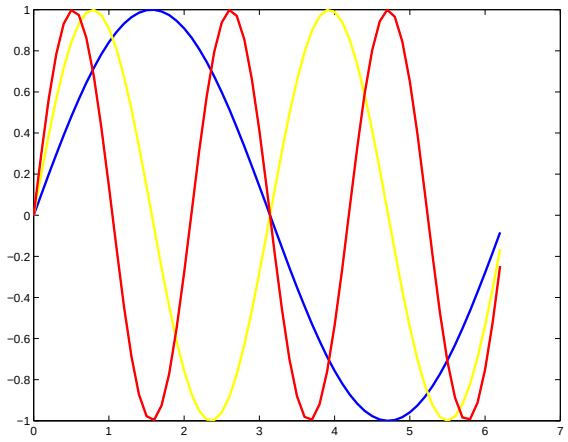


Figura 7: Grafici delle funzioni $y1 = \sin(x), y2 = \sin(2x), y3 = \sin(3x)$

Per scegliere lo stile della linea (*linestyle*), l'indicatore (*marker*) ed il colore (*linecolor*) con i quali ciascuna curva viene disegnata bisogna specificare tali opzioni utilizzando la sintassi `plot(x,y,'linecolor-linestyle-marker')`. Ad esempio:

```
>> plot(x,y,'r-.*');
```

rappresenta la curva in rosso (opzione `'r'`) con una linea tratto-punto (opzione `'-.'`) e identifica ogni coppia ascissa-ordinata discreta con un asterisco (opzione `'*'`).

- Le opzioni indicanti i colori sono `'r','g','b','w','k','y','c','m'`, che corrispondono rispettivamente a rosso, verde, blu, bianco, nero, giallo, azzurro chiaro e magenta.

- Le opzioni indicanti gli stili di linea sono `'-'` per linea piena, `'--'` per linea tratteggiata, `':'` per linea punteggiata, `'-.'` per linea tratto-punto e `'none'` per "nessuna linea".

- Fra i *marker* di linea più comuni ricordiamo infine quelli identificati dalle opzioni `'+','o','x','*'`.

Per maggiori informazioni su tutte le opzioni esistenti, digitare il comando `help plot`.

Osservazione. Per rappresentare la curva per punti discreti possiamo utilizzare un *marker* (si veda il risultato in Figura 8).

```
>> x=0:0.1:2*pi;
>> y=sin(x);
>> plot(x,y,'*')
```

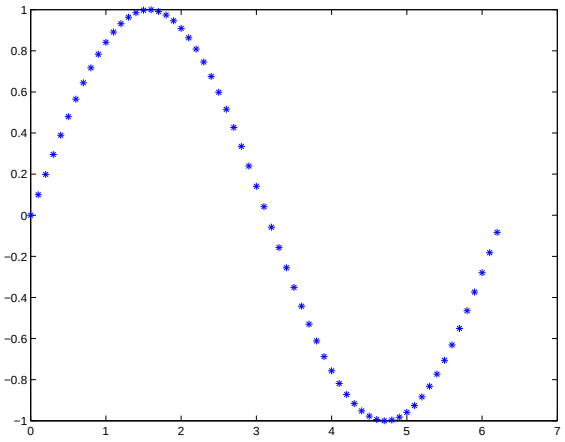


Figura 8: Grafico della funzione $y = \sin(x)$ rappresentato per valori discreti

Una volta tracciato un grafico è poi buona norma completarlo con un titolo e con delle etichette per gli assi, rispettivamente con i comandi `title, xlabel, ylabel`:

```
>> plot(x,y);
>> title('Funzione seno');
>> xlabel('x');
>> ylabel('y');
```

Inoltre, ad esempio quando sono presenti più curve sullo stesso grafico, può essere utile aggiungere una legenda che permetta di comprendere facilmente il significato di ciascuna curva. Una volta tracciato il grafico, il comando:

```
legend('stringa1','stringa2',...)
```

associa alla prima curva tracciata la voce della legenda '`stringa1`', alla seconda curva la voce della legenda '`stringa2`' e così via. Ad esempio si ottiene il risultato di Figura 9 con i comandi:

```
>> x=0:0.1:2*pi;
>> plot(x,y1,'o-',x,y2,'*-')
>> title('Funzioni trigonometriche')
>> legend('sin(x)','sin(2x)')
```

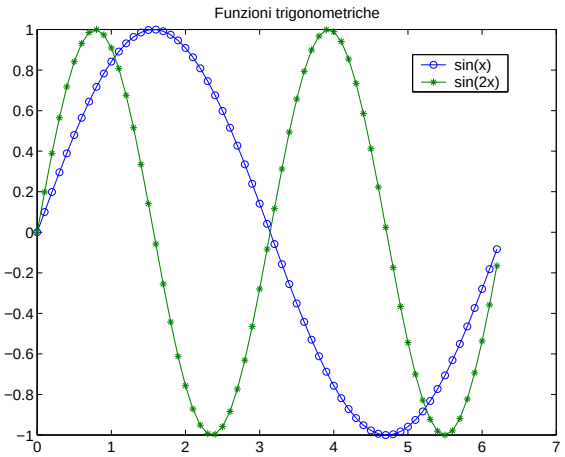


Figura 9: Uso dell'istruzione `legend` per aggiungere la legenda ad un grafico

Spesso si vuole aggiungere una curva ad un grafico già esistente. Una volta generato un certo grafico il comando `hold on`, oppure il solo comando `hold`, specifica a `MATLAB` di mantenere attivo tale grafico e di rappresentare *su di esso* le successive curve senza aprire ulteriori finestre. Una riscalatura automatica degli assi viene applicata quando necessario per includere interamente nella figura tutte le curve tracciate. Per rilasciare il grafico, chiudendo la corrispondente finestra, si utilizza il comando `hold off` oppure il solo `hold` (quest'ultimo funziona quindi come un *interruttore* acceso-spento). Ad esempio:

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

>> plot(x,y1,'r')
>> hold off

>> y2=sin(3+x);
>> plot(x,y2).
```

In questo modo le prime due curve `y,y1` vengono rappresentate nel medesimo grafico, mentre `y2` viene rappresentata su un grafico diverso.

Il comando `axis` permette di controllare gli estremi degli assi ed il loro aspetto. Specificando in ingresso un vettore riga le cui componenti sono `xmin, xmax, ymin, ymax` il grafico viene inquadrato in accordo con i nuovi limiti degli assi. Ciò risulta utile quando si vogliano evidenziare dei particolari che a causa delle diverse scale risultano invisibili nella rappresentazione standard, come nel seguente esempio dove si cerca l'intersezione fra le curve $y_1 = e^x$ e $y_2 = x + 3$.

```
>> x=0:0.01:10;
>> y1=exp(x);
>> y2=x+3;
>> plot(x,y1,x,y2)
>> axis([1 2 0 10])
```

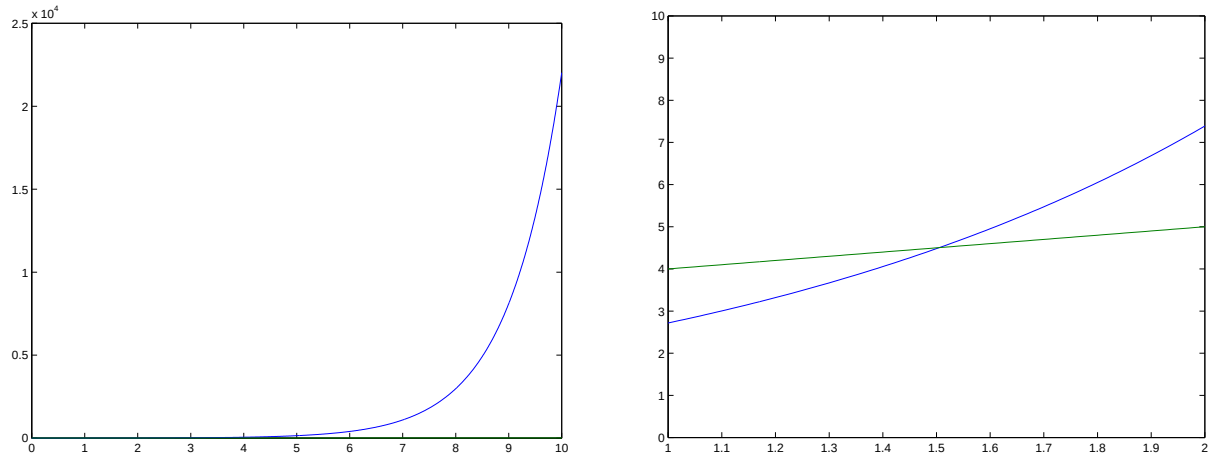


Figura 10: Nella figura a sinistra, a causa delle scale in gioco molto differenti, è impossibile riconoscere il punto di intersezione delle curve e^x ed $x + 3$, che invece è ben evidenziato riscalando gli assi come nella figura a destra

Si veda il risultato in Figura 10.

Per controllare l'aspetto degli assi esistono numerose altre opzioni, fra le quali citiamo:

- `axis('square')` che rende gli assi x ed y della stessa lunghezza, pur mantenendo su ciascuno di essi scale differenti.
- `axis('equal')` che suddivide gli assi x ed y utilizzando incrementi di ampiezza uguale.

Per le altre opzioni consultare l'help in linea (`help axis`).

In alcuni casi è indispensabile rappresentare i dati in scala logaritmica (in base 10) scegliendo se usare tale scala solo sull'asse x, sull'asse y oppure su entrambi gli assi. Nel primo caso si usa l'istruzione `semilogx`, nel secondo `semilogy` e nel terzo `loglog`, tutte con una sintassi analoga a `plot`. Ad esempio, per visualizzare in modo leggibile la funzione e^x per $x \in [0, 500]$ rappresentiamo le ordinate in scala logaritmica (diventa una retta in questo caso!):

```
>> x=1:500;
>> y=exp(x);
>> semilogy(x,y)
```

Osservazione. Se i dati in ingresso da plottare in scala logaritmica sono quantità negative, essi vengono ignorati (come avverte un *warning* che appare lanciando il comando di plottaggio).

Il comando `zoom` abilita (ancora con la logica di un interruttore acceso-spento) la possibilità di ingrandire un'area specifica di un grafico. Con il tasto sinistro del mouse si deve quindi delimitare un rettangolo intorno all'area interessata, eventualmente ripetendo più volte questa operazione. Per ogni pressione del tasto destro del mouse si torna invece al fattore di ingrandimento immediatamente precedente.

Osservazione. Talvolta è utile definire una funzione *in modo simbolico*, ovvero definire semplicemente l'espressione matematica della funzione $f = f(x)$ *senza che ad essa vengano associati dei valori numerici.* In questo caso si dice che la funzione è definita come stringa e la sua espressione deve essere racchiusa fra apici. Definiamo ad esempio come stringa la funzione tangente iperbolica:

```
>> fun='(exp(x)-exp(-x))./(exp(x)+exp(-x))'

fun =

(exp(x)-exp(-x))./(exp(x)+exp(-x))
```

Calcoliamo ora i valori della tangente iperbolica corrispondenti ad un certo insieme di valori della variabile x da cui essa dipende. Definiamo quindi un vettore x e valutiamo `'fun'` nei punti x_i di tale vettore con il comando `eval`: esso restituisce un vettore di valori numerici di dimensione `length(x)` la cui componente $i$esima è nel caso in esame il valore della tangente iperbolica nel punto x_i. Ad esempio per tracciare il grafico della funzione tangente iperbolica definita come stringa per valori di $x \in [-5, 5]$ con passo $\Delta x = 0.01$ possiamo scrivere:

```
>> x=-5:0.01:5;
>> plot(x,eval(fun))
>> grid
>> title('Tangente ipebolica');
```

Per modificare in modo più sofisticato l'aspetto di un grafico (ad esempio lo spessore delle linee, la grandezza ed il tipo dei caratteri utilizzati, il colore dei marker...) si possono utilizzare le istruzioni `get` e `set`. Si comprende il loro funzionamento pensando che a ciascun grafico venga associato un puntatore (detto *graphic handle*) che permette di accedere agli oggetti che costituiscono il grafico stesso (linee, titoli, assi..) e di manipolarne opportunamente le caratteristiche (dette *properties*). Supponiamo ad esempio di voler cambiare lo spessore della linea che traccia la curva e la grandezza del carattere nel titolo e nelle quote degli assi della Figura 11. Con l'istruzione `gca` si associa al grafico correntemente attivo il puntatore corrispondente (che chiamiamo in questo caso `ax`).

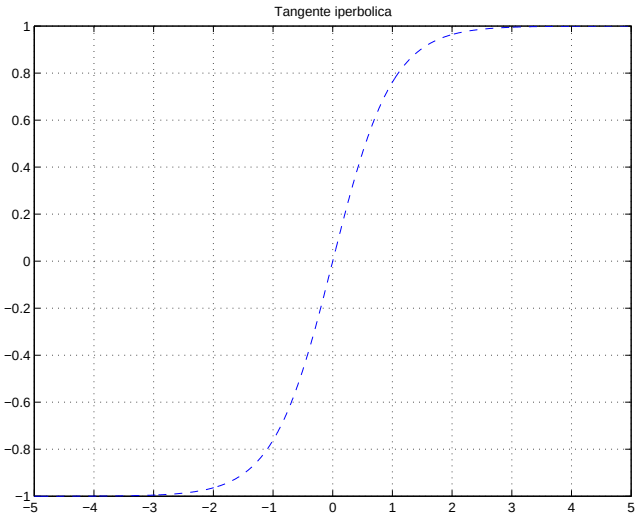


Figura 11: Grafico della funzione tangente iperbolica prima della manipolazione di alcune proprietà degli oggetti: alcuni particolari (ad esempio la grandezza del carattere del titolo) possono essere migliorati per rendere il grafico più chiaro

Utilizziamo ora il comando `get` che prende in ingresso il puntatore `ax` per conoscere il valore corrente delle proprietà del grafico:

```
>> ax=gca
>> get(ax)
```

Osserviamo che `ax` contiene l'indirizzo di una locazione di memoria come risulta dalla prima istruzione. Il contenuto della locazione di memoria *puntata* da `ax` viene estratto con il comando `get(ax)`.

Con il comando `set` modifichiamo quindi la proprietà originali. La sintassi generale è:

```
set(nome-puntatore,'Nome Proprieta1',Nuovo-Valore1,'Nome
Proprieta2',Nuovo-Valore2,'Nome Proprieta3',Nuovo-Valore3,...);
```

Ricordiamo che, qualora `Nuovo-Valore` sia una stringa di caratteri e non un valore numerico, essa deve essere racchiusa fra apici.

Ad esempio portiamo da 10 a 15 punti la dimensione dei caratteri nelle quote degli assi:

```
>> set(ax,'fontsize',15);
```

Portiamo anche il carattere del titolo da 10 a 15 punti: in questo caso il titolo *non* è un "oggetto elementare" come nel caso precedente, perché esso a sua volta è caratterizzato da varie proprietà quali la posizione, il tipo di carattere, il contenuto della stringa... (dal risultato del comando `get(ax)` osserviamo infatti che `Title` è a sua volta un puntatore) È quindi necessario prima accedere a tale puntatore e poi modificare la proprietà `'Fontsize'` ad esso relativa (si noti infatti che al comando `set` viene passato il puntatore al titolo che abbiamo chiamato `ti`):

```
>> ti=get(ax,'Title');
>> set(ti,'Fontsize',15);
```

Per finire, portiamo da 0.5 a 3 punti lo spessore della curva. Anche in questo caso le curve contenute nel grafico *non* sono oggetti elementari e vi si può accedere solo tramite un puntatore. Il campo di `h` da cui prelevare tale puntatore si chiama per definizione `'Children'`. Chiamiamo `li` il nuovo puntatore così ottenuto ed usiamolo in ingresso nel comando `set` per impostare lo spessore della linea:

```
>> li=get(ax,'Children');
>> set(li,'linewidth',3);
```

In Figura 12 è riportato il grafico di Figura 11 dopo le modifiche: il risultato è un grafico più leggibile e chiaro.

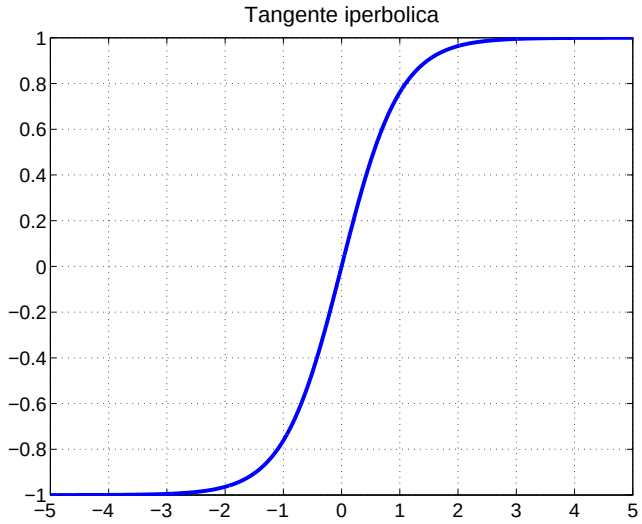


Figura 12: Grafico della funzione tangente iperbolica dopo la manipolazione di alcune proprietà tramite l'istruzione `set`

5.2 Rappresentazione di curve in tre dimensioni

Il comando `plot3`, con la sintassi
`plot3(x,y,z)`
analoga al semplice `plot`, permette di tracciare nello spazio curve i cui punti hanno coordinate rispettivamente assegnate tramite le componenti dei tre vettori `x,y,z`.
Ad esempio, per disegnare l'elica riprodotta in Figura 13:

```
>> x = 0:pi/50:10*pi;
>> plot3(sin(x),cos(x),x);
```

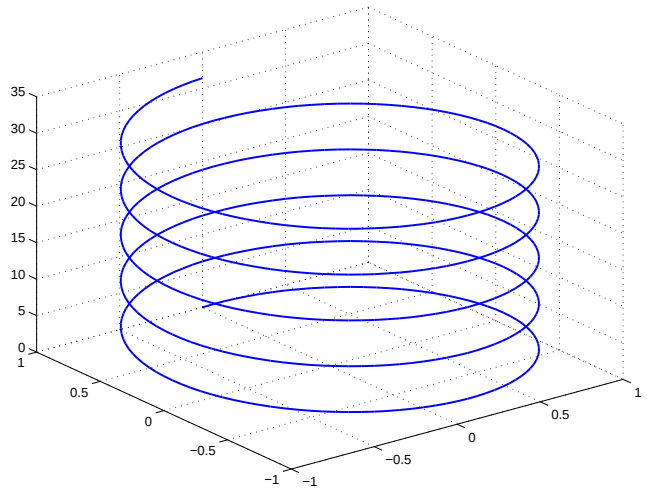


Figura 13: Uso del comando `plot3` per tracciare una curva nello spazio

5.3 Rappresentazione di superfici in tre dimensioni

Supponiamo ora di voler rappresentare nello spazio il grafico della funzione di due variabili $z = z(x, y)$. Le funzioni `mesh` e `surf` permettono di rappresentare superfici in tre dimensioni; la prima traccia solo lo "scheletro" della superficie, mentre la seconda, oltre a tracciare lo scheletro, riempie i quadrilateri da esso generati con facce poligonali colorate (si veda la Figura 14). A partire dai vettori x e y è necessario dapprima generare una griglia di base che individui tutti i nodi di coordinate x_i, y_j. A tale scopo l'istruzione `meshgrid` trasforma i vettori x e y rispettivamente nelle matrici X e Y che consistono nella ripetizione dei valori delle ascisse ad ordinata fissata (la matrice X) e delle ordinate ad ascissa fissata (la matrice Y).

```
>> x=-2:.2:2;
>> y=-2:.2:2;
>> [X,Y]=meshgrid(x,y);
```

Definiamo e rappresentiamo con `mesh` la funzione $z = xe^{-(x^2+y^2)}$ utilizzando la griglia di base generata con le matrici X ed Y:

```
>> Z = X.* exp(-(X.^2 +Y.^2));
>> mesh(X,Y,Z);
```

Con `surf`:

```
>> surf(X,Y,Z);
>> colormap(hsv);
```

È facoltativo specificare con `colormap` la mappa di colori con cui viene parametrizzato l'insieme dei valori assunti da $z = z(x, y)$. Esistono numerose mappe di colori, fra le quali citiamo `hot, cool, hsv, flag`; è conveniente stabilire di caso in caso la mappa che permette di ottenere il risultato più efficace. In Figura 14 sono rappresentati i grafici ottenuti rispettivamente con `mesh` e `surf`.

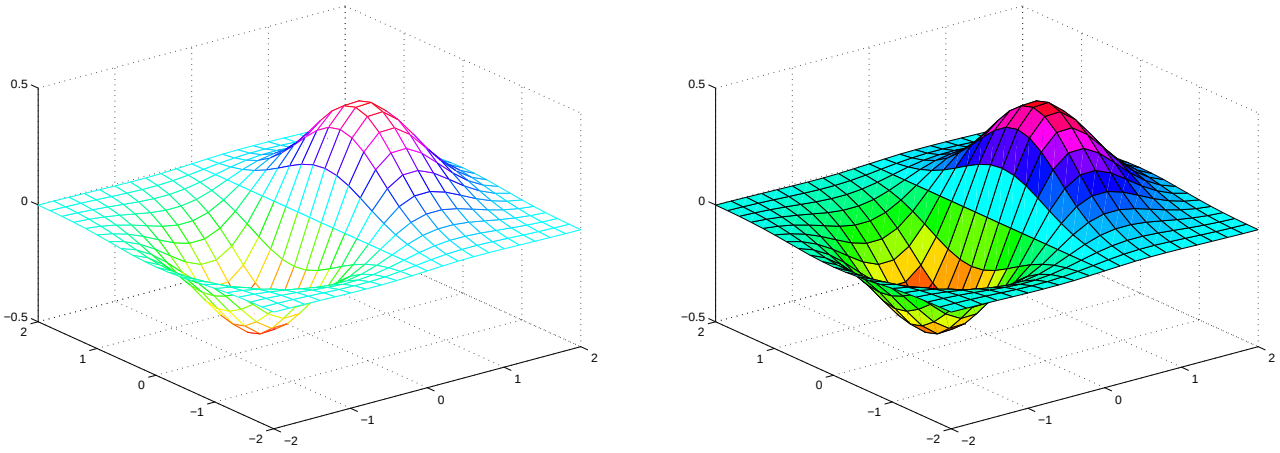


Figura 14: Rappresentazioni grafiche della superficie $z = xe^{-(x^2+y^2)}$ ottenute rispettivamente con `mesh` (a sinistra) e `surf` (a destra)

Nello studio delle superfici è utile poter rappresentare anche le curve di livello della superficie. La funzione `contour` traccia nel piano xy le curve di livello di $z = z(x, y)$. Fra le opzioni, è possibile sia fissare un numero prestabilito `N` di curve di livello da disegnare a livelli scelti automaticamente da `MATLAB` (utilizzando la sintassi `contour(X,Y,Z,N)`), sia fissare uno ad uno i valori delle curve di livello da tracciare definendo un vettore `v` che li elenchi (utilizzando in questo caso la sintassi `contour(X,Y,Z,v)`). Il comando

```
>> contour(X,Y,Z,20)
```

disegna 20 curve di livello, mentre il comando

```
>> v=[-0.4 -0.3 -0.2 0 0.2 0.3 0.4];
>> contour(X,Y,Z,v)
```

disegna `length(v)` curve di livello ai valori -0.4, -0.3, -0.2, 0, 0.2, 0.3, 0.4 (si veda la Figura 15).

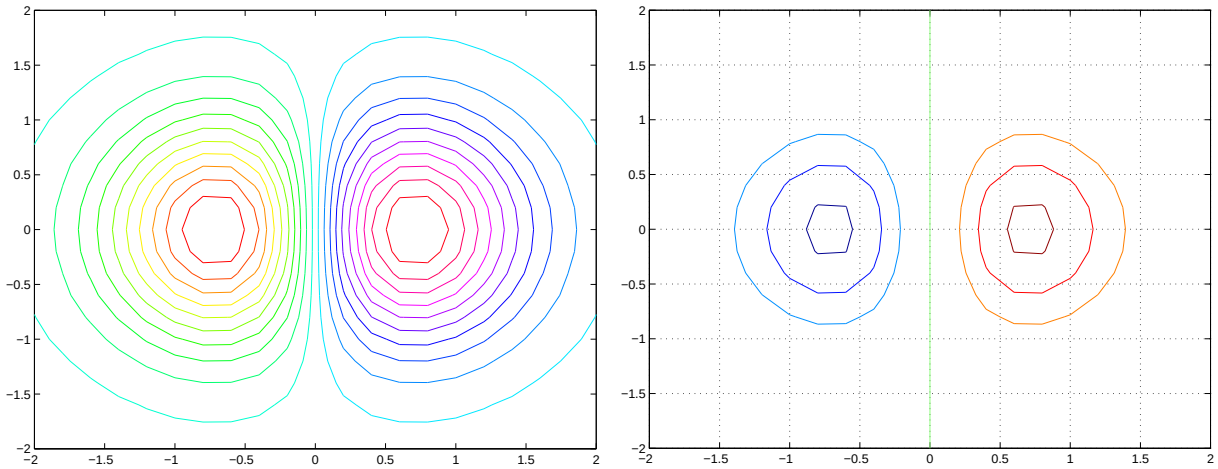


Figura 15: Curve di livello per la funzione $z = xe^{-(x^2+y^2)}$: a sinistra si è specificato di usare 20 curve (la cui quota viene scelta automaticamente da `MATLAB`), mentre a destra sono stati specificati ad uno ad uno i valori relativi alla quota di ciascuna da disegnare

In ultimo, data la funzione $z = z(x, y)$, possiamo calcolare *numericamente* le componenti `px` e `py` del gradiente con la funzione `gradient`. Rappresentiamo poi tale vettore nel piano xy in ogni punto in cui esso è definito con delle frecce che ne indicano modulo, direzione e verso utilizzando la funzione `quiver` (si veda il risultato in Figura 16 a sinistra):

```
>> [X,Y] = meshgrid(-2:.2:2, -2:.2:2);
>> Z = X.* exp(-(X.^2 + Y.^2));
>> [px,py] = gradient(Z,.2,.2);
>> quiver(X,Y,px,py)
```

Sovrapponendo al grafico realizzato con `quiver` le linee di livello ottenute con `contour` si osserva chiaramente che in ogni punto il vettore gradiente è perpendicolare alla curva di livello passante per quel punto (si veda la Figura 16 a destra).

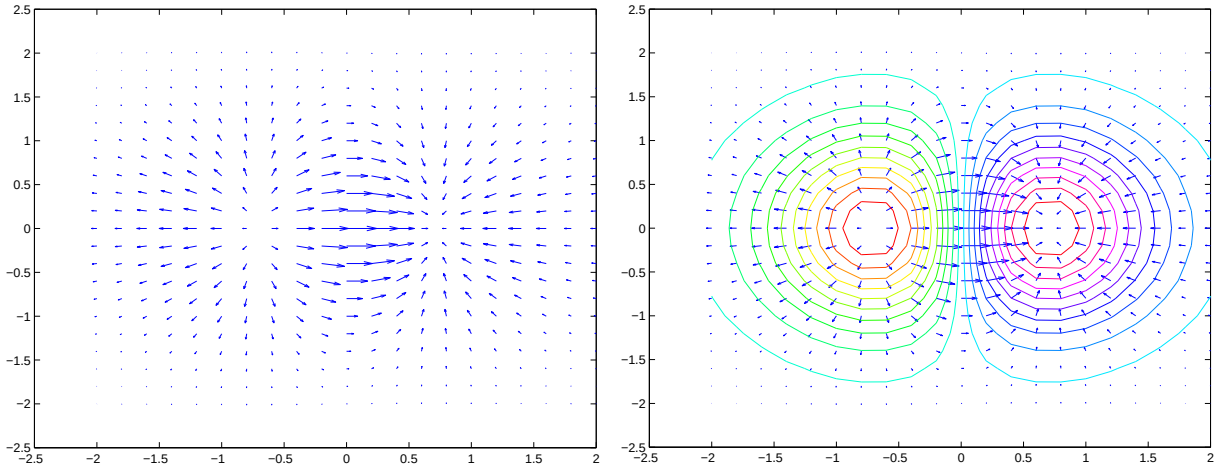


Figura 16: Rappresentazione vettoriale del gradiente della funzione $z = xe^{-(x^2+y^2)}$ (a sinistra) e rappresentazione vettoriale del gradiente di z cui sono sovrapposte le curve di livello (a destra)

Per una rappresentazione sintetica dei risultati è talvolta utile poter affiancare più figure diverse. Il comando `subplot(m,n,p)` permette di impostare una *matrice* costituita da m righe su ciascuna delle quali sono disposte n figure e restituisce il puntatore che permette di accedere alla $p$esima figura.
Ad esempio, riuniamo in una matrice di 2×2 figure i quattro risultati relativi ai grafici $3D$ precedentemente considerati (si veda la Figura 17):

```
>> [X,Y] = meshgrid(-2:.2:2, -2:.2:2);
>> Z = X.* exp(-(X.^2 + Y.^2));
>> [px,py] = gradient(Z,.2,.2);

>> subplot(2,2,1), mesh(X,Y,Z), title('mesh')
>> subplot(2,2,2), surf(X,Y,Z), title('surf')
>> subplot(2,2,3), contour(X,Y,Z,20), title('contour')
>> subplot(2,2,4), quiver(X,Y,px,py), title('quiver')
```

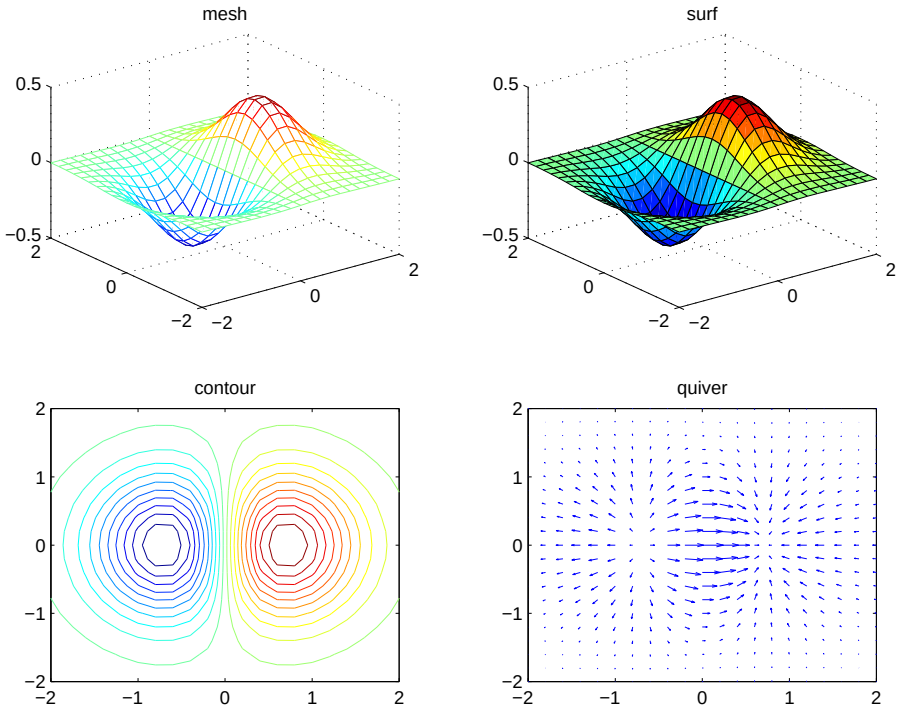


Figura 17: Rappresentazione sintetica di varie figure ottenuta tramite l'istruzione `subplot`

Le istruzioni `mesh, surf, contour` e `quiver` agiscono rispettivamente sulla figura specificata dall'ultimo valore indicato nei parametri di ingresso di `subplot`. Le figure vengono numerate progressivamente per righe, ovvero nella griglia 2×2 la prima figura della seconda riga è la numero 3. Ogni figura può essere singolarmente corredata di un titolo, di etichette degli assi e di legende.

Una volta realizzato un grafico, è possibile salvarne il contenuto in un file che può essere elaborato anche da altre applicazioni (ad esempio Word oppure LaTex) tramite il comando `print`: esistono numerosi formati ed opzioni collegati ad esso. Ad esempio ricordiamo:

- `print -dbmp256 nomefigura` salva la figura nel direttorio corrente in formato bitmap a 256 colori (estensione .bmp)
- `print -dpsc nomefigura` salva la figura nel direttorio corrente in formato Postscript a colori (estensione .ps)

Per altri formati specifici si consiglia di consultare l'help in linea (`help print`).