

# Laboratorio 2

## Funzioni, Plot e aritmetica floating point

---

©2009 - Questo testo (compresi i quesiti ed il loro svolgimento) è coperto da diritto d'autore. Non può essere sfruttato a fini commerciali o di pubblicazione editoriale. Non possono essere ricavati lavori derivati. Ogni abuso sarà punito a termine di legge dal titolare del diritto. This text is licensed to the public under the Creative Commons Attribution-NonCommercial-NoDerivs2.5 License (<http://creativecommons.org/licenses/by-nc-nd/2.5/>)

---

### 1 Definizione di funzioni

Matlab<sup>®</sup> e Octave mettono a disposizione una serie di funzioni matematiche “standard” (`sin`, `cos`, `exp`, ...), ma consentono all’utente anche di definire le proprie funzioni.

In altre parole, è possibile scrivere un frammento di codice che prenda in ingresso una variabile  $x$  e restituisca il valore  $f(x)$ , ad esempio  $f(x) = x^4 + 3 \log(x)$ , e che venga eseguito tramite il comando `nome_funzione(x)`<sup>1</sup>.

Vengono forniti quattro diversi modi per definire una funzione:

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

In questo laboratorio ci occupiamo solo dei primi tre casi. Il quarto, che prevede l’utilizzo di un file esterno, verrà illustrato nelle prossime esercitazioni.

#### 1.1 Il comando `eval`

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

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

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

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

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

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

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

---

<sup>1</sup>è buona norma di programmazione dare nomi significativi alle funzioni!

```

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

```

## 1.2 Il comando inline

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

### 1.3 Anonymous functions ( @ )

Questo terzo modo può essere pensato come una “contrazione” del comando `inline`<sup>2</sup>. La sintassi è la seguente:

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

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

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

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

**Esercizio 1: Valutare tramite i tre metodi appena descritti le funzioni:**

- $f(x) = x \sin(x) + \left(\frac{1}{2}\right)^{\sqrt{x}}$
- $f(x) = x^4 + \log(x^3 + 1)$

sul vettore `x=[0 1 2 3]` .

## 2 Disegnare grafici di funzione

### 2.1 il comando `plot`

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

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

---

<sup>2</sup>in realtà i due meccanismi sono diversi: le `anonymous function` usano il concetto di `function handle` @, che non approfondiremo in questo corso

Nel nostro caso, il comando:

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

produce il grafico in Figura 1.

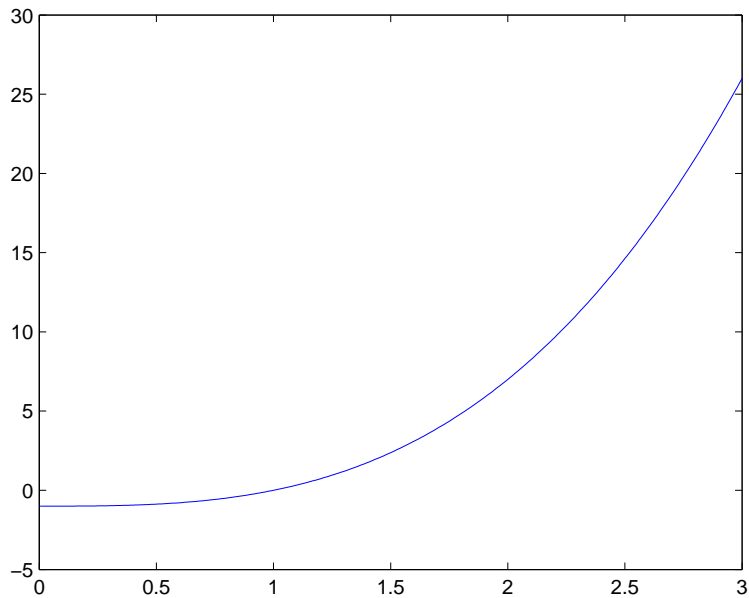


Figura 1:  $x^3 - 1$  per  $0 \leq x \leq 3$

Un ulteriore comando `plot` disegna il nuovo grafico sulla stessa finestra, cancellando quello precedente. Per modificare questo comportamento, così che i nuovi grafici siano sulla stessa figura, si usa il comando `hold on`. Si torna allo stato precedente con `hold off` mentre `hold` da solo cambia tra i due stati.

Ad esempio, i comandi per disegnare sullo stesso grafico la funzione  $f(x) = 3 - x^2$  sono:

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

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

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

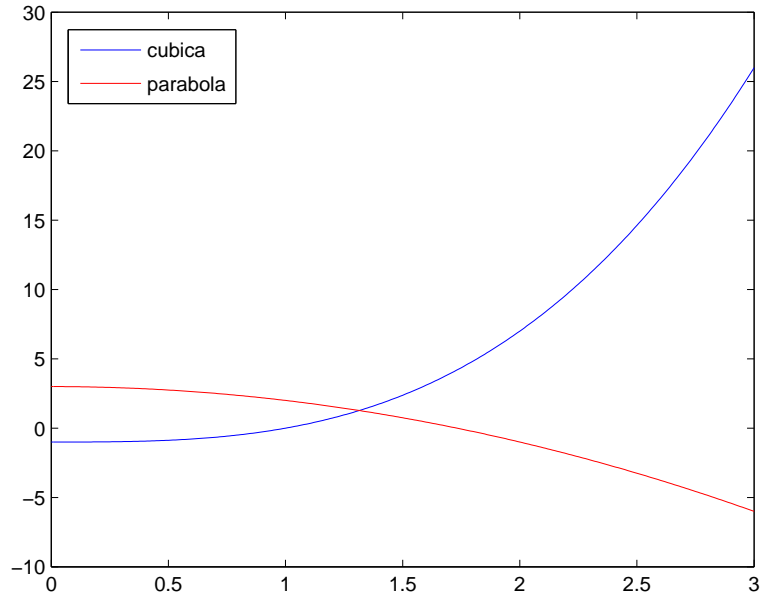


Figura 2: grafico di  $x^3 - 1$  e  $3 - x^2$  per  $0 \leq x \leq 3$

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

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

Nota: in Matlab® se si vuole cancellare un grafico dalla finestra mentre `hold on` è attivato, si può entrare in modalità `edit plot` (menù `tools/edit plot`, quindi selezionare il grafico desiderato col mouse e cancellarlo premendo il tasto `cancel`).

### Esercizio 2: Disegnare il grafico della funzione

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

per  $0 \leq x \leq 6$ . Sovrapporre a questo grafico quelli delle due rette che limitano l'andamento di tale funzione, disegnate con linea tratteggiata.

**Suggerimento** Le due rette in questione sono  $y = -x + 5$  e  $y = x - 1$ .

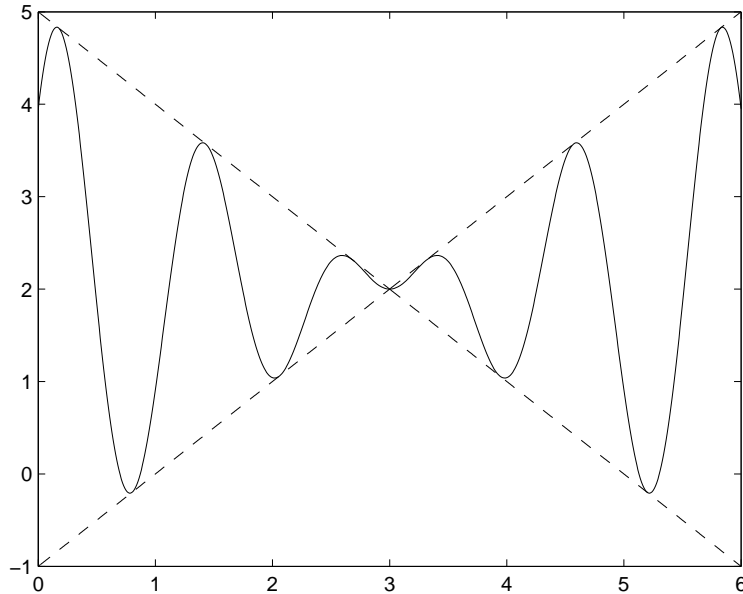


Figura 3: soluzione dell'esercizio 2

## 2.2 Grafici in scala logaritmica

In molte aree scientifiche vengono usati grafici in scala logaritmica/semilogaritmica. Matlab® e Octave forniscono a tale proposito i comandi `semilogy`, `semilogx` e `loglog`, che sono l'equivalente di `plot` ma tracciano un grafico rispettivamente con l'asse delle ordinate in scala logaritmica, con l'asse delle ascisse logaritmico ed entrambi in scala logaritmica.

Supponiamo di voler disegnare in scala  $y$ -logaritmica sull'intervallo  $0 \leq x \leq 10$  il grafico delle funzioni  $y = e^x$  e  $y = e^{2x}$ . I comandi necessari sono (stavolta disegniamo il secondo grafico in verde, con linea continua, indicando i punti con degli asterischi, vedi figura 4):

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

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

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

- la griglia:

```
>> grid on
```

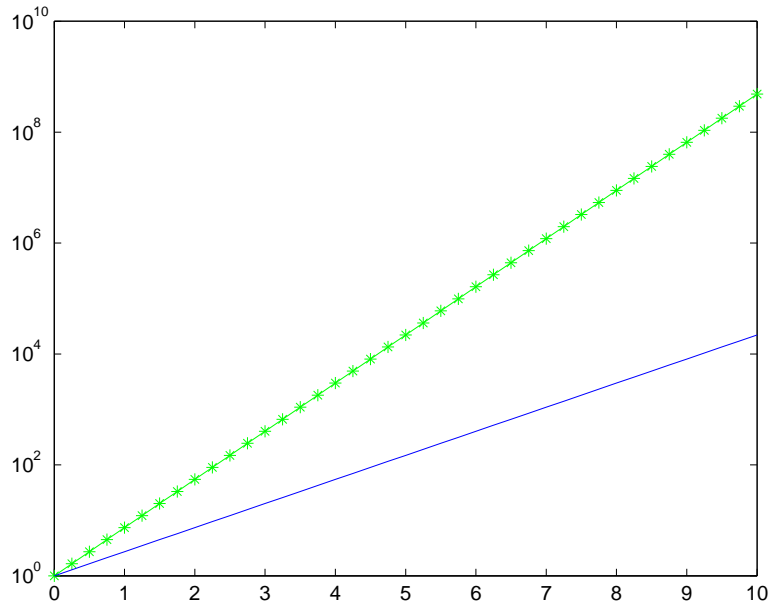


Figura 4: grafico in scala  $y$ -logaritmica di  $y = e^x$  e  $y = e^{2x}$

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

- il titolo del grafico:

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

- i titoli degli assi:

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

- la legenda:

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

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

**Esercizio 3:** Cosa ci si aspetta di ottenere disegnando il grafico  $x$ -logaritmico di  $f(x) = (\log(x))^2$  sull'intervallo  $0.1 \leq x \leq 10$ ? Disegnare il grafico e verificare.

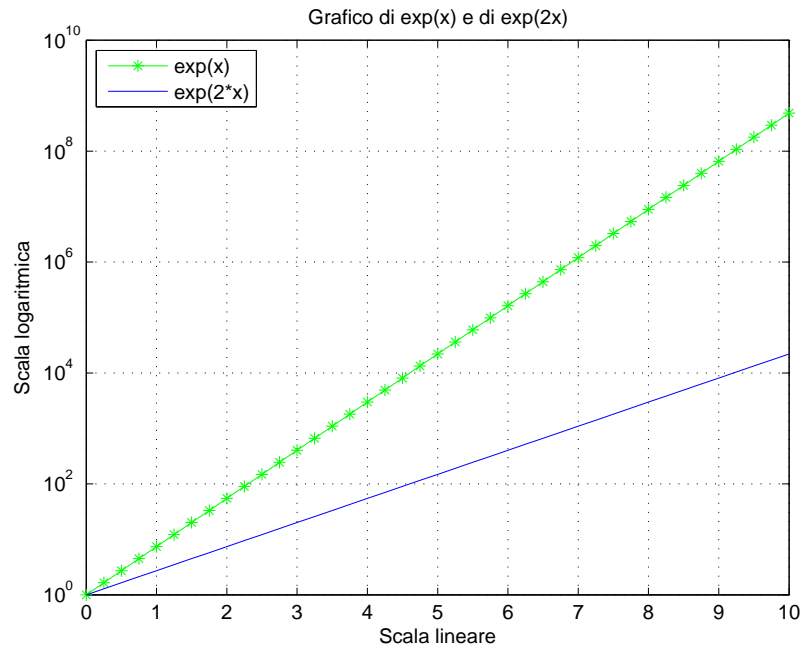


Figura 5: grafico in scala  $y$ -logaritmica di  $y = e^x$  e  $y = e^{2x}$ , con titoli e legenda

### 3 Aritmetica finita del calcolatore

È importante notare che l'insieme dei numeri utilizzabili da un calcolatore è un particolare sottoinsieme finito dei numeri razionali che chiamiamo  $\mathbb{F}$ .

Per memorizzare ogni numero è disponibile una quantità limitata di bit, più precisamente 64; di conseguenza i numeri che costituiscono l'aritmetica del calcolatore hanno necessariamente un numero finito di cifre decimali. In particolare, i numeri irrazionali come  $\pi$  oppure i numeri razionali periodici come  $1/3$  sono sostituiti da una loro approssimazione con un numero finito di cifre decimali.

Se in  $\mathbb{F}$  sono contenuti solo numeri con un numero finito di cifre decimali, attorno ad ogni elemento  $f$  di  $\mathbb{F}$  esiste un piccolo intervallo  $I_f$  “vuoto”, che contiene solo  $f$  stesso e nessun altro elemento di  $\mathbb{F}$ .

In altre parole, la distanza fra  $f$  e il suo elemento successivo di  $\mathbb{F}$  non è infinitamente piccola, ma è un valore ben determinato (per quanto “molto piccolo”). Tale distanza si definisce *epsilon macchina*, e si indica tramite  $\epsilon(f)$ .

Ovviamente esistono diversi modi per memorizzare un numero in 64 bit (virgola fissa, virgola mobile, etc), e il valore di  $\epsilon(f)$  e l'estensione di  $I_f$  dipendono dal metodo di memorizzazione che si decide di utilizzare.

Matlab<sup>®</sup> e Octave utilizzano lo standard IEEE 754; secondo questo particolare formato la distanza  $\epsilon(f)$  non è costante, ma dipende da  $f$ , e l'intervallo  $I_f$  non è necessariamente simmetrico

In Matlab<sup>®</sup> e Octave esiste la variabile predefinita `eps` che rappresenta l'epsilon macchina del numero 1, cioè il più piccolo numero che sommato al numero 1 fornisce un numero maggiore di 1. In particolare:

```
>> eps
```

```
ans =
```

```
2.2204e-16
```

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

```
>> format long e
```

```
>> a = 1+eps
```

```
a =
```

```
1.0000000000000000e+00
```

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

```
>> a - 1
```

```
ans =
```

```
2.220446049250313e-16
```

Adesso eseguiamo lo stesso procedimento sommando al numero 1 il valore  $\text{eps}/2$  e otteniamo:

```
>> format long e
```

```
>> b = 1+eps/2
```

```
b =
```

```
1
```

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

```
>> b - 1
```

```
ans =
```

```
0
```

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

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

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

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

```
ans =
```

```
0
```

```
>> 1 + eps(1) - 1
```

```
ans =
```

```
2.220446049250313e-16
```

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

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

```
ans =
```

```
1.136868377216160e-13
```

Osserviamo infine che la funzione built-in `eps( )` di Matlab<sup>®</sup> di cui abbiamo parlato, non è implementata in Octave mentre la variabile predefinita `eps` è la stessa. Infatti in Octave è implementata come built-in un'altra funzione `eps( )`:

```
octave:> n = 2
octave:> eps(n)
ans =
```

```
2.2204e-16  2.2204e-16
2.2204e-16  2.2204e-16
```

produce una matrice quadrata di dimensione `n` di valori tutti coincidenti con la variabile `eps`. Come ultima considerazione, possiamo identificare l'elemento più piccolo e più grande di  $\mathbb{F}$  con le variabili predefinite `realmin` e `realmax` di Matlab<sup>®</sup> e Octave :

```
>> realmin
```

```
ans =
```

```
2.225073858507201e-308
```

```
>> realmax
```

```
ans =
```

```
1.797693134862316e+308
```

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

```
>> a = 1e400
```

```
a =
```

```
Inf
```

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

```
>> a = 1e-310
```

```
a =
```

```
9.999999999999969e-311
```

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

```
>> eps(realmin)
```

```
ans =
```

```
4.940656458412465e-324
```

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

```
>> a = 1e-325
```

```
a =
```

```
0
```

## 4 Esempi di errori numerici

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

```
>> a = [0:0.1:1];
>> b=linspace(0,1,11);
```

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

```
>> c = abs( a-b )
>> c =
```

```
1.0e-16 *
```

```
Columns 1 through 7
```

```
0 0 0 0.5551 0 0 0
```

```
Columns 8 through 11
```

```
0 0 0 0
```

Ripetiamo lo stesso procedimento per altri due vettori **e** ed **f** e valutiamo la loro differenza **g**:

```
>> e = [0:0.25:4];
>> f=linspace(0,4,17);
>> g = abs( a-b )
>> g =
```

```
Columns 1 through 12
```

```
0 0 0 0 0 0 0 0 0 0 0 0
```

```
Columns 13 through 17
```

```
0 0 0 0 0
```

Cosa succede di diverso nei due esempi? Il problema nasce dal fatto che il passo usato nel primo esempio (0.1) è un numero periodico in base 2:

$$(0.1)_{10} = (0.000\overline{1100})_2$$

quindi nella sua rappresentazione floating point (in  $\mathbb{F}$ ) è necessariamente troncato; questo comporta degli errori che si propagano in modo diverso a seconda della funzione utilizzata per costruire il vettore. Al contrario nel secondo esempio non si presenta questo problema, perché il passo 0.25 non è un numero periodico in base 2:

$$(0.25)_{10} = (0.01)_2$$

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

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

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

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

mentre ovviamente:

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

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