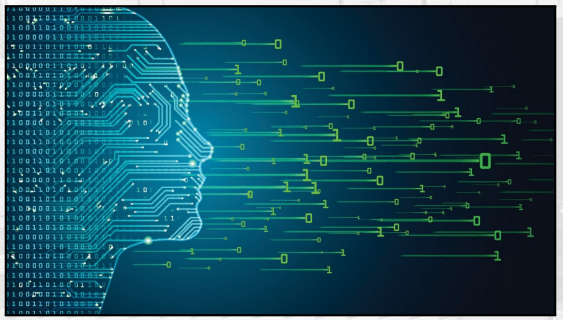
Programmazione Informatica
7. MATLAB, parte 2: Input/Output (I/O), strutture di controllo e funzioni

Riccardo Zamolo
rzamolo@units.it

Università degli Studi Trieste
Ingegneria Civile e Ambientale



A.A. 2020-21

- Finora abbiamo definito in maniera diretta, da prompt o su script, le variabili di ingresso al nostro problema, ed i risultati dei calcoli venivano visualizzati in termini di contenuto di variabili, una ad una.
- I dati in input e output in uno script possono essere forniti in maniera programmatica attraverso opportune funzioni di input/output (I/O), che permettono:
  - input e output formattati nel prompt;
  - output grafici (plot);
  - lettura o scrittura su file.
- La funzione **input** può essere utilizzata per acquisire un valore da tastiera attraverso il prompt. Il valore inserito può essere di qualsiasi tipo visto finora (numerico, logico, testuale), scalare o matriciale, anche in forma di espressione complessa:

  nome_variabile = input(messaggio)

  che mostra il testo riportato nel **messaggio** e assegna il valore inserito da tastiera, dopo aver premuto il tasto Invio, nella variabile **nome_variabile**:

```
>> H = input( 'Inserire altezza [metri]: ' ) ;
>> Inserire altezza [m]: 1.85
>> H
    1.8500

>> v = input( 'Inserire voti [/30]: ' ) ;
>> Inserire voti [/30]: [ 28 25 18 33 30 ]
>> v
    28    25    18    33    30
```

- La funzione **disp** può essere utilizzata per visualizzare nel prompt una variabile di qualsiasi tipo, ed equivale a richiamare una variabile direttamente nel prompt:

```
>> v = [ 1.5 2.6 3.7 ] ;
>> disp( v )
    1.5000    2.6000    3.7000
```

```
>> A = [ 1 2 3   ; ...
         4 5 6.5 ] ;
>> disp( A )
    1.0000    2.0000    3.0000
    4.0000    5.0000    6.5000
```

- La funzione **disp** non permette un output formattato, che è invece possibile ottenere con la funzione **fprintf**:

  fprintf(formato , x1 , ... , xN)

  dove **x1 , ... , xN** sono le **N** variabili da visualizzare e **formato** specifica la formattazione da dare all'output mediante gli operatori di formattazione:

  %(flags)(n_caratteri).(precisione)(conversione)

Tipo	(conversione)	Dettagli
intero	d	base 10
intero senza segno	u, o, x, X	basi 10 (u), 8 (o), 16 (x,X)
non intero	f, e, E	posizionale (f), esponenziale (e,E)
testo	c, s	caratteri (c) e stringhe (s)

- (conversione), esempi:

```
>> fprintf( '%d' , 99 ) ;
99
>> fprintf( '%o' , 63 ) ;
77
>> fprintf( '%X' , 255 ) ;
FF
```

```
>> fprintf( '%f' , pi ) ;
3.141593
>> fprintf( '%e' , pi ) ;
3.141593e+00
>> fprintf( '%c' , 'Testo' ) ;
Testo
```

- (precisione): numero di cifre decimali dopo il punto decimale.

```
>> fprintf( '%.1f' , 1.79612 ) ;
1.8
>> fprintf( '%.15' , pi ) ;
3.141592653589793
>> fprintf( '%.30' , 1+1/3 ) ;
1.333333333333333259318465024990
```

```
>> fprintf( '%.2e'  , pi*6.042e23 ) ;
1.90e+24
>> fprintf( '%.15e' , pi*6.042e23 ) ;
1.898150281298953e+24>>
```

- (n_caratteri): numero minimo di caratteri totali da impiegare.

```
>> fprintf( '%1d' , 12345 ) ;
12345
>> fprintf( '%8d' , 1 ) ;
       1
>> fprintf( '%8d' , 12345 ) ;
   12345
```

```
>> fprintf( '%12f' , pi ) ;
    3.141593
>> fprintf( '%12e' , pi ) ;
3.141593e+00
>> fprintf( '%14e' , pi ) ;
  3.141593e+00
```

- (flags): ulteriori possibilità di formattazione; "-" per allineare a sinistra e non a destra (default), "+" per visualizzare sempre il segno anche nel caso di numeri positivi, "0" per riempire gli spazi a sinistra con 0.

- Caratteri speciali:

Carattere	Forma	Risultato
apice singolo	''	'
percentuale	%%	%
barra rovesciata	\\	\
nuova linea	\n	a capo

```
>> fprintf( 'Pi greco con %d cifre decimali si scrive %.10f\n' , 10 , pi ) ;
Pi greco con 10 cifre decimali si scrive 3.1415926536
>>

>> r = 1.5 ;
>> fprintf( 'L''area del cerchio di raggio %.2f vale %.2f\n' , r , pi*r^2) ;
L'area del cerchio di raggio 1.50 vale 7.07
>>
```

- Con **printf** è possibile visualizzare in maniera formattata anche vettori:

```
>> v = 10 .^ (0:5) ;
>> fprintf( 'v è un vettore lungo %d e vale (' , length(v) ) ; ...
   fprintf( '%d,' , v ) ; ...
   fprintf( ')\n' ) ;
v è un vettore lungo 6 e vale (1,10,100,1000,10000,100000,)
>>
```

- Per dati in forma testuale si possono impiegare due tipi diversi:
  - **char**: singoli caratteri, definiti tra apici **'(carattere)'**;
  - **string**: stringhe = sequenze di caratteri, definite tra doppi apici **"(stringa)"**.

```
>> car = 'A' ;
>> class( car )
ans =
     'char'
```

```
>> str = "Università" ;
>> class( str )
ans =
     'string'
```

- Si possono definire vettori di tipo **char** o **string** concatenando opportunamente in riga o in colonna elementi di quel tipo, esattamente come per i tipi numerici:

```
>> car = [ 'U' 'n' 'i' 'v' ] ;
>> car = 'Univ'
car =
     'Univ'
>> size( car )
ans =
     1     4
>> [ car ; car ]
ans =
     2×4 char array
     'Univ'
     'Univ'
```

```
>> str = "Università" ;
>> size( str )
ans =
     1     1
>> [ str ; str ]
     2×1 string array
     "Università"
     "Università"
```

$\mathcal{NB}$: un vettore di **char** non corrisponde ad un tipo **string**.

- La codifica impiegata è UTF-8, corrispondente alla codifica ASCII per i primi 128 caratteri.

- Come per i tipi numerici, il nome del tipo può essere impiegato come funzione per effettuare la conversione verso altri tipi:

```
>> car = 'Univ' ;
>> str_car = string( car )
str_car =
     "Univ"
>> num_car = int16( car )
num_car =
  1×4 int16 row vector
    85   110   105   118
```

```
>> str = "Univ" ;
>> car_str = char( str )
car_str =
    'Univ'
>> num_str = int16( str )
 Error using int16
 Conversion to int16 from string is
 not possible.
>> num_str = int16( char( str ) )
num_str =
1×4 int16 row vector
85   110   105   118
```

- La funzione **sprintf** esegue le stesse operazioni di formattazione della funzione **fprintf**, ma l'output viene indirizzato in un vettore di **char** o in una stringa di tipo **string**, a seconda dei delimitatori impiegati per il formato:

```
>> car = sprintf( 'Pi = %.5f' , pi )
car =
    'Pi = 3.14159'
>> class( car )
ans =
     'char'
```

```
>> str = sprintf( "Pi = %.5f" , pi )
str =
     "Pi = 3.14159"
>> class( str )
ans =
     'string'
```

- Si possono combinare concatenazioni di vettori **char**, definiti direttamente o tramite **sprintf**, per definire un testo formattato complesso.

- Scrivere uno script che fornisca un output testuale di tutti i caratteri ASCII (esclusi i primi 32) formattati in una tabella di 16 righe e seguendo l'ordine per colonne. L'output va inteso come una matrice di caratteri, separando le colonne dei caratteri richiesti con k spazi.
- Il numero totale di caratteri richiesti è $128 - 32 = 96$ e quindi il numero totale di colonne riempite dai caratteri richiesti sarà $96/16 = 6$.

```
% Matrice di caratteri senza spazi di separazione tra le colonne
%   preparazione dei caratteri e dimensioni
caratteri = char( 32 : 127 ) ;     % Caratteri richiesti
n_car     = length( caratteri ) ; % Numero totale di caratteri
righe     = 16 ;                   % Numero di righe
colonne   = n_car / righe ;        % Numero di colonne
%   reshape dei caratteri in matrice righe x colonne, senza spazi
M_ASCII   = reshape( caratteri , righe , colonne ) ;

% Matrice finale con separazione tra le colonne
%   preparazione della matrice finale
k = 2 ;                                  % Numero di spazi tra le colonne
colonne_tot = (colonne-1) * (1+k) + 1 ; % Numero totale di colonne
M    = zeros( righe , colonne_tot ) ;   % Matrice finale, numeri
M    = char( M ) ;                      % Matrice finale, char
M(:) = ' ' ;                            % Riempimento con spazi
%   assegnazione finale nella matrice con gli spazi tra le colonne
vj = 1 : (1+k) : colonne_tot ;          % Indici colonne da riempire
M( : , vj ) = M_ASCII ;                 % Assegnazione matriciale

% Visualizzazione risultato
disp( M ) ;
```

- Dall'esecuzione del precedente script si ottiene:

```
    0  @  P  `  p
!   1  A  Q  a  q
"   2  B  R  b  r
#   3  C  S  c  s
$   4  D  T  d  t
%   5  E  U  e  u
&   6  F  V  f  v
'   7  G  W  g  w
(   8  H  X  h  x
)   9  I  Y  i  y
*   :  J  Z  j  z
+   ;  K  [  k  {
,   <  L  \  l  |
-   =  M  ]  m  }
.   >  N  ^  n  ~
/   ?  O  _  o
```

	MSB					
	010	011	100	101	110	111
LSB (dec)	(32)	(48)	(64)	(80)	(96)	(112)
0000 (0)	SP	0	@	P	`	p
0001 (1)	!	1	A	Q	a	q
0010 (2)	"	2	B	R	b	r
0011 (3)	#	3	C	S	c	s
0100 (4)	$	4	D	T	d	t
0101 (5)	%	5	E	U	e	u
0110 (6)	&	6	F	V	f	v
0111 (7)	'	7	G	W	g	w
1000 (8)	(	8	H	X	h	x
1001 (9)	)	9	I	Y	i	y
1010 (10)	*	:	J	Z	j	z
1011 (11)	+	;	K	[	k	{
1100 (12)	,	<	L	\	l	|
1101 (13)	-	=	M	]	m	}
1110 (14)	.	>	N	^	n	~
1111 (15)	/	?	O	_	o	DEL

- Scrivere uno script che, acquisita da tastiera una sequenza di caratteri, renda maiuscole tutte le lettere minuscole (non accentate).
- I codici ASCII delle lettere minuscole vanno da 97 (`'a'`) a 122 (`'z'`).
- La distanza tra una lettera maiuscola e la relativa lettera minuscola è sempre 32.

```
% Input da tastiera
caratteri = input( 'Inserire un vettore di caratteri: ' ) ;

% Conversione in char nel caso che l'input sia di tipo string
caratteri = char( caratteri ) ;

% Vettore Booleano dei soli caratteri minuscoli
minuscoli = ( 97 <= caratteri ) & ( caratteri <= 122 ) ;

% Sostituzione con i relativi caratteri maiuscoli
caratteri( minuscoli ) = caratteri( minuscoli ) - 32 ;

% Output
disp( caratteri ) ;
```

- Dall'esecuzione del precedente script, inserendo `'Sequenza di N caratteri in INPUT'` da tastiera, si ottiene:

```
Inserire un vettore di caratteri: 'Sequenza di N caratteri in INPUT'
SEQUENZA DI N CARATTERI IN INPUT
```

- In MATLAB sono disponibili le 2 strutture di controllo di base: selezione (**if**) e iterazione (**while**), più altre 2 strutture supplementari, ossia selezione multipla (**switch**) e iterazione con conteggio (**for**):

```
if condizione (logico/Booleano)
  % Caso di condizione vera
  istruzioni
else
  % Caso di condizione falsa
  istruzioni
end

switch espressione
  case espr1
    % Caso espressione = espr1
    istruzioni
  case espr2
    % Caso espressione = espr2
    istruzioni
  ...
  otherwise
    % Caso complementare
    istruzioni
end
```

```
while condizione
  % Caso di condizione vera
  istruzioni
end

for indice = valori
  % indice = valore scalare
  istruzioni
end
```

- Se necessario, all'interno dei cicli iterativi **while** e **for** è possibile passare direttamente all'iterazione successiva, saltando le istruzioni rimanenti, o uscire del tutto dal ciclo attraverso i comandi **continue** e **break**, rispettivamente.

- Scrivere uno script che acquisisca da tastiera due interi `n` ≥ 0 e `b` > 1 e determini le cifre di `n` in base `b`, formattando opportunamente il risultato.
- Utilizzeremo le funzioni `input` e `fprintf` per l'I/O dei dati e l'algoritmo delle divisioni intere successive per eseguire il calcolo utilizzando il ciclo `while`:

```
% Input dei dati
n = input('Inserire il numero intero positivo n: ') ;
b = input('Inserire la base b>1: ') ;

% Algoritmo delle divisioni successive
q = n ; % Quoziente di lavoro
a = 0 ; % Vettore delle cifre calcolate
i = 0 ; % Indice di iterazione
while q > 0
  i    = i + 1 ;
  a(i) = mod( q , b ) ;      % resto della divisione
  q    = ( q - a(i) ) / b ; % quoziente
end

% Output formattato
fprintf( '(%d)_10 = (' , n ) ; ...
fprintf( '%d' , a(end:-1:1) ) ; ...
fprintf( ')_%d\n' , b ) ;
```

- Dall'esecuzione del precedente script con `n` $= 63$ e `b` $= 2$ si ottiene:

```
Inserire il numero intero positivo n: 63
Inserire la base b>1: 2
(63)_10 = (111111)_2
>>
```

- Scrivere uno script che acquisisca da tastiera due interi `n` ≥ 0 e `b` > 1 e determini le cifre di `n` in base `b`. Verificare al momento dell'inserimento la validità dei dati stessi.

```
% Input dei dati con check
input_invalido = true ;
while input_invalido
    n  = input('Inserire il numero intero positivo n: ') ;
    c1 = numel(n) == 1 ;      % Valore scalare
    c2 = isnumeric(n) ;       % Valore numerico
    c3 = n == floor( n ) ;    % Valore intero
    c4 = n >= 0 ;             % Valore positivo
    input_invalido = ~( c1 & c2 & c3 & c4 ) ;
    if input_invalido
        fprintf('Valore invalido!\n') ;
    end
end

input_invalido = true ;
while input_invalido
    b = input('Inserire la base b>1: ') ;
    input_invalido = ~( numel(b)==1  && ...
                        isnumeric(b) && ...
                        b==floor(b)  && ...
                        b>1 ) ;
    if input_invalido ; fprintf('Valore invalido!\n') ; end
end

% Algoritmo delle divisioni successive
...
```

- Scrivere uno script che acquisisca da tastiera
  - un vettore di coefficienti $\mathbf{a} = (a_0, a_1, \ldots, a_n)$
  - un intervallo $[b, c]$
  - un intero N

  e calcoli il polinomio

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\ &= a_0 + x(a_1 + x(\cdots a_{n-1} + x(a_n))), \end{aligned}$$

  sfruttando l'ultima forma, su N valori di x equidistanziati in $[b, c]$.
- Utilizzeremo un ciclo `for` per calcolare esplicitamente $p(x)$:

```
% Input dei dati
a  = input('Inserire il vettore dei coefficienti: ') ;
bc = input('Inserire l''intervallo [b,c] in forma di vettore: ') ;
N  = input('Inserire il numero di valori di x: ') ;
n  = length(a) - 1 ; % grado del polinomio

% Vettore riga degli N valori di x da b=bc(1) a c=bc(2)
x = linspace( bc(1) , bc(2) , N ) ;

% Calcolo del polinomio
p = zeros( 1 , N ) ; % inizializzazione polinomio a 0
for i = n : -1 : 0
     p = a(i+1) + x .* p ; % gli indici dei vettori partono sempre da 1
end
```

- Il risultato finale si troverà nel vettore riga `p`.

- Abbiamo visto che le funzioni servono a scrivere una volta sola una serie di istruzioni che necessitano di essere riutilizzate più volte nell'esecuzione di un programma. La funzione viene quindi semplicemente *chiamata* col suo nome quando è necessario.
- Il tipo più semplice di funzione in MATLAB è la *funzione anonima*: viene definita quando si ritiene più opportuno all'interno di uno script e non necessita di essere salvata in un file .m separato:

  `nome_funzione = @(x1,...,xN)(istruzione)`

  dove `x1,...,xN` sono gli `N` argomenti in input, `(istruzione)` è il corpo della funzione e dev'essere costituito da un'istruzione sola.
- L'output della funzione anonima coincide quindi con il valore dell'istruzione stessa: non è quindi possibile assegnare più di un output (in maniera diretta). L'output può ovviamente essere di qualsiasi tipo, sia scalare che vettoriale/matriciale:

```
% Definizione
f = @(x) (1+x-x.^2) ./ (x+7*x.^2) ;

% Utilizzo (scalare)
y = ( f(1) + f(2) ) / f(3) ;

% Utilizzo (vettoriale)
v_x = linspace( 0 , 1 , 1000 ) ;
v_y = f(v_x) ;
```

```
% Definizione
f = @(x,y) sqrt( x .^ 2 + y .^ 2 ) ;

% Utilizzo (scalare)
z = f(1,2) / f(2,1) ;

% Utilizzo (matriciale)
v_x = linspace( 0 , 1 , 1000 ) ;
v_y = linspace( 0 , 1 , 1000 )' ;
v_z = f(v_x,v_y) ;
```

Esercizi su strutture di controllo e funzioni anonime (Parte2_Es6.m)

- Considerato il problema dell'esercizio `Parte2_Es4.m`, si utilizzi una funzione anonima per raccogliere le operazioni (singole) comuni a tutti e due i cicli `while`.
- Le operazioni (singole) in comune sono relative ai controlli sulla dimensione (scalare), sul tipo (numerico) e sul valore (intero) del dato inserito da tastiera.

```
% Funzione anonima che verifica se il dato è uno scalare intero
is_scalare_intero = @(x) numel(x)==1  && ...
                         isnumeric(x) && ...
                         x==floor(x) ;

% Input dei dati con check
input_invalido = true ;
while input_invalido
  n = input('Inserire il numero intero positivo n: ') ;
  input_invalido = ~( is_scalare_intero(n) && n>=0 ) ;
  if input_invalido ; fprintf('Valore invalido!\n') ; end
end

input_invalido = true ;
while input_invalido
  b = input('Inserire la base b>1: ') ;
  input_invalido = ~( is_scalare_intero(b) && b>1 ) ;
  if input_invalido ; fprintf('Valore invalido!\n') ; end
end

% Algoritmo delle divisioni successive...
```

- Scrivere uno script che determini un'approssimazione dello zero della funzione $f(x) = \sin(2x) - x$ mediante il metodo della bisezione a partire dall'intervallo $[\pi/4, \pi/2]$. Definire $f(x)$ mediante una funzione anonima.
- Il metodo della bisezione suddivide iterativamente un intervallo $[a, b]$ in due metà e scegliendo l'intervallo dove $f(x)$ cambia segno:

```
% Definizione della funzione anonima
f = @(x) sin(2*x) - x ;

% Intervallo di partenza
ab   = [ pi/4 pi/2 ] ;

% Valori della funzione agli estremi dell'intervallo
f_ab = f( ab ) ;

% Bisezione con 50 iterazioni
for i = 1 : 50
  xm = ( ab(1) + ab(2) ) / 2 ; % punto medio dell'intervallo
  f_xm = f( xm ) ;             % f( punto medio )
  % Caso di funzione nulla nel punto medio: uscita
  if f_xm == 0
    ab = [ xm xm ] ; % assegnazione intervallo puntiforme
    f_ab = [ 0 0 ] ; % assegnazione valori nulli agli estremi
    break ;          % uscita dal ciclo for
  else
    % Nel caso di funzione non nulla nel punto medio, bisogna decidere quale
    % semi-intervallo scegliere
    if sign( f_ab(1) ) ~= sign( f_xm ) % f cambia segno nell'intervallo sx
```

(Parte2_Es7.m)

```
      % Teniamo il semi-intervallo sinistro
      ab(2)   = xm ;
      f_ab(2) = f_xm ;
    else
      % Teniamo il semi-intervallo destro
      ab(1)   = xm ;
      f_ab(1) = f_xm ;
    end
  end
end

% Output risultati
x = ( ab(1) + ab(2) ) / 2 ; % punto medio
f_x = f( x ) ;              % f( punto medio )
disp( x ) ;
disp( f_x ) ;
```

- Questo metodo di ricerca dello zero di una funzione è più efficiente del metodo impiegato nell'esercizio `Parte1_Es8.m`: ad ogni iterazione la lunghezza $l = b - a$ dell'intervallo $[a, b]$ viene dimezzato.
- All'iterazione i si avrà perciò $l_i = l_0/2^i$. Sarà perciò sufficiente scegliere un numero di iterazioni tale che l'intervallo finale sia $[a, a + a\epsilon]$, ossia quello minimo rappresentabile in doppia precisione. Si ha perciò:

$$l_i = l_0/2^i = (a + a\epsilon) - a = a\epsilon \approx \epsilon$$

poichè $a \in [\pi/4, \pi/2] \approx [0.78, 1.57]$ e sappiamo già che lo zero è $a \approx 1$.

- Risolvendo la precedente relazione rispetto ad i si ottiene:

$$i = \frac{\log l_0 - \log \epsilon}{\log 2} \approx 52$$

dove $l_0 = \pi/2 - \pi/4 = \pi/4$ e $\epsilon = 2^{-52}$ in doppia precisione.
Si potrebbe quindi incorporare questo calcolo direttamente nello script prima del ciclo **for**:

```
% Stima del numero di iterazioni necessarie
i_max = ceil( ( log( ab(2)-ab(1) ) - log(eps) ) / log(2) ) ;
```

- Dall'esecuzione dello script con 50 iterazioni (e **format long**) si ottiene:

```
0.947747133516990
2.220446049250313e-16
```

mentre con 51 iterazioni si ottiene:

```
0.947747133516990
0
```

coincidente proprio con il risultato esatto $x = 0.947747133516990$.

- Le funzioni anonime sono molto comode perchè si definiscono agevolmente e direttamente all'interno di uno script. Sono però limitate dal fatto di poter essere costituite da un'istruzione sola. Per questo sono tipicamente usate per implementare funzioni matematiche analitiche di una o più variabili, non troppo complesse.
- Una funzione MATLAB (non anonima) dev'essere definita in un file `nome_funzione.m` separato, che deve quindi avere lo stesso nome della funzione stessa:

```
function [y1,...,yN] = nome_funzione(x1,...,xM)
  istruzioni che assegnano y1,...,yN
end
```

  dove `x1,...,xM` sono gli `M` argomenti in input, `y1,...,yN` sono gli `N` argomenti in output e le istruzioni racchiuse nella funzione non hanno più nessuna limitazione.
- Quando una funzione viene richiamata in uno script o da prompt, MATLAB ricerca i relativi file funzione .m in una lista di percorsi indicati nella variabile `path`, oltre che nell'attuale percorso di lavoro (`pwd`). Per questo motivo tutti i file funzione definiti dall'utente sono tipicamente collocati nella stessa cartella dello script principale, che è un file .m anch'esso.

- Quando però i file funzione sono numerosi risulta più comodo organizzarli in sottocartelle del percorso di lavoro, che vanno aggiunte al **path**. L'aggiunta al **path** di tutte le sottocartelle viene eseguita tramite

```
addpath( genpath( pwd ) ) ;
```

  che andrà messo in testa nello script principale.
- Tutti gli elementi visti precedentemente, ossia variabili/matrici di tutti i tipi e funzioni anonime, erano definiti in una zona di memoria detta *base workspace*, accessibile dal prompt in maniera diretta (richiamando o definendo una variabile o una funzione anonima) oppure attraverso i comandi **who/whos, workspace**. Tutte le variabili/funzioni anonime definite nel base workspace sono quindi *visibili* in questo contesto.
- Una funzione definita in un file funzione .m non lavora nel base workspace ma definisce uno spazio diverso e separato. Gli elementi definiti all'interno di una funzione come questa sono detti *locali* poichè sono visibili solo all'interno della funzione stessa e non all'esterno di essa.
- Una variabile definita dentro una funzione, per esempio, è una *variabile locale* e non è visibile nel base workspace, richiamandola per esempio dal prompt. Allo stesso modo, una variabile definita nel base workspace non è visibile all'interno della funzione. L'unico modo per scambiare dati con una funzione è farlo attraverso gli argomenti `x1,...,xM` in input e `y1,...,yN` in output, che sono quindi delle variabili locali.

- Per scrivere una funzione, apriamo un nuovo script. Scriviamo per esempio una funzione, che chiameremo **fattoriale**, per calcolare il fattoriale di un intero utilizzando un ciclo **for**:

fattoriale.m

```
function m = fattoriale( n )
  m = 1 ;          % m è una variabile locale (l'output della funzione)
  for i = 2 : n    % i è una variabile locale
    m = m * i ;
  end
end
```

- Salveremo quindi lo script con lo stesso nome utilizzato per la funzione, quindi **fattoriale.m**.
- La funzione così definita può essere richiamata dovunque, al prompt o all'interno di altre funzioni:

```
>> fattoriale(10)
ans =
     3628800
```

coefficiente_taylor.m

```
function y = coefficiente_taylor( x )
  y = 1 / fattoriale( x ) ;
end
```

- **m,n,i** sono quindi variabili locali, visibili (e quindi utilizzabili) solo all'interno della funzione **fattoriale**. Sono quindi libero di usare **m,n,i** come nomi per altre variabili all'esterno della funzione **fattoriale**, per esempio nel base workspace o in un'altra funzione, senza preoccuparmi di eventuali conflitti.

- All'interno di un file funzione .m va sempre definita la funzione omonima in testa. Eventualmente si possono definire altre funzioni dopo la funzione principale oppure dentro la funzione principale, che potranno essere utilizzate solo all'interno della funzione principale stessa e non altrove:

`somma_quadrati.m`

```
function c = somma_quadrati( a , b )
  c = quadrato( a ) + quadrato( b ) ;
end

function y = quadrato( x )
  y = x^2 ;
end
```

`somma_quadrati.m`

```
function c = somma_quadrati( a , b )
  function y = quadrato( x )
    y = x^2 ;
  end
  c = quadrato( a ) + quadrato( b ) ;
end
```

- In questo caso la funzione `quadrato` non può essere quindi richiamata al prompt o all'interno di altre funzioni.
- Nel caso di funzione con output multipli (`N` > 1) l'assegnazione degli output al momento della chiamata alla funzione avviene nuovamente con la notazione tra parentesi quadre, come nella definizione della funzione:

```
[o1,..,oN] = nome_funzione(i1,..,iM)
```

ed è possibile non assegnare degli output usando il simbolo $\sim$:

```
[~,o2,~,..,oN] = nome_funzione(i1,..,iM)
```

- Scrivere una funzione che prenda in input due numeri n e b e fornisca in output il quoziente q ed il resto r della divisione intera di n per b:

$$n = q \cdot b + r$$

- Si hanno due numeri in output: si può scegliere di usare due argomenti in output, uno per ogni numero:

quoziente_resto.m

```
function [ q , r ] = quoziente_resto( n , b )
  r = mod( n , b ) ;
  q = ( n - r ) / b ;
end
```

oppure si possono organizzare i due numeri in unico vettore in output:

quoziente_resto_vettoriale.m

```
function qr_vettore  = quoziente_resto_vettoriale( n , b )
  qr_vettore = [ 0 0 ] ; % Preparazione vettore output
  r = mod( n , b ) ;
  qr_vettore(2) = r ;              % Assegnazione nel vettore output
  qr_vettore(1) = ( n - r ) / b ; % Assegnazione nel vettore output
end
```

- L'assegnazione degli output nella chiamata alle due funzioni sarà quindi diversa:

```
>> [ quoziente , resto ] = quoziente_resto( 123 , 7 )
quoziente =
      17
resto =
       4
```

mentre nel caso vettoriale:

```
>> q_r_vett = quoziente_resto_vettoriale( 123 , 7 )
q_r_vett =
      17      4
```

- Nel primo caso si potrà decidere quali output assegnare e quali no:

```
% Assegnazione del primo argomento in output
>> [ q , ~ ] = quoziente_resto( 123 , 7 )
q =
      17
>> q = quoziente_resto( 123 , 7 )
q =
      17

% Assegnazione del secondo argomento in output
>> [ ~ , r ] = quoziente_resto( 123 , 7 )
r =
       4
```

- Scrivere una funzione che calcoli il fattoriale di un numero n in maniera ricorsiva:

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1 = n(n-1)!$$

fattoriale_ricorsivo.m

```
function fatt = fattoriale_ricorsivo( n )
  if n > 1
    fatt = n * fattoriale_ricorsivo( n-1 ) ;
  else
    fatt = 1 ;
  end
end
```

- Nel caso di funzioni ricorsive, cioè che chiamano se stesse, ogni nuova chiamata alla funzione definisce uno spazio diverso per gli elementi locali (variabili, funzioni, ecc.).
- Nel caso della precedente funzione `fattoriale_ricorsivo`, ogni sua chiamata ricorsiva predispone delle nuove variabili locali `n` e `fatt` diverse dalle chiamate precedenti, anche se il nome è evidentemente lo stesso.

- Scrivere una funzione che prenda in input un vettore di coefficienti $\mathbf{a} = (a_0, a_1, \ldots, a_n)$ ed un vettore $\mathbf{x}$ e calcoli in maniera ricorsiva il polinomio

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \\ &= a_0 + x(a_1 + x(\cdots a_{n-1} + x(a_n))), \end{aligned}$$

  sfruttando quindi l'ultima forma, per tutti i valori di x del vettore $\mathbf{x}$.

polinomio_ricorsivo.m

```
function p = polinomio_ricorsivo(a, x, k)
  n = length(a) - 1 ;
  if k < n
    p = a(k+1) + x .* polinomio_ricorsivo(a, x, k+1) ;
  else
    p = a(n+1) + 0*x ;
  end
end
```

- La chiamata alla funzione dovrà quindi inizare da $k = 0$:

```
p_x = polinomio_ricorsivo(coefficienti, vettore_x, 0) ;
```

- Si può controllare che le tre implementazioni per il calcolo di un polinomio (`Parte1_Es4.m`, `Parte2_Es5.m` e la presente) coincidano a parità di input.

- Scrivere una funzione che prenda in input un intero N e calcoli in maniera vettoriale la seguente somma:

$$e_N = \sum_{i=0}^{N} \frac{1}{i!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{N!}$$

somma_eN.m

```
function eN = somma_eN(N)
  i = 1 : N ;
  fattoriale_i = [ 1 cumprod( i ) ] ;
  eN = sum( 1 ./ fattoriale_i ) ;
end
```

- Si visualizzi poi la differenza $\Delta = e - e_N$ per $N = 5, 10, 15, 20$:

```
for N = 5 : 5 : 20
  Delta = exp(1) - somma_eN(N) ;
  fprintf('N = %2d, Delta = %.15f\n', N, Delta) ;
end
```

che produce il seguente output:

```
N =  5, Delta = 0.001615161792379
N = 10, Delta = 0.000000027312661
N = 15, Delta = 0.000000000000051
N = 20, Delta = 0.000000000000000
```

in quanto $\lim_{N \to \infty} e_N = e$.

- A volte è molto utile poter passare in input ad una funzione (oppure ottenere in output da una funzione, ma è raro) un'altra funzione.
- Si pensi per esempio ad una funzione `trova_zeri` che calcola gli zeri di una funzione `f` non nota a priori. In questo caso bisogna essere in grado di passare `f` (la funzione di cui cerchiamo gli zeri) a `trova_zeri` (la funzione che li calcola) al pari di una qualsiasi variabile.
- In MATLAB ciò si può fare attraverso variabili di tipo `function_handle` dette appunto *function handle* che servono a riferirsi a funzioni.
- Quando si definisce una funzione anonima con la sintassi già vista

  `nome_funzione = @(x1,...,xN)(istruzione)`

  `nome_funzione` risulta essere una variabile di tipo `function_handle` che punta alla funzione definita con quel nome:

```
>> f = @(x) sin(x) / x
f =
  function_handle with value:
    @(x)sin(x)/x
```

Possiamo convincerci che a questo punto `f` è una function handle nel base workspace, poichè definita da prompt, richiamando il comando `whos`:

```
>> whos
  Name        Size                Bytes  Class                Attributes
  f           1×1                    32  function_handle
```

- Una variabile di tipo **function_handle** può quindi essere passata ad una funzione alla pari di una qualsiasi variabile, e permette di richiamare quella funzione puntata dal suo function handle:
- Esempio: definiamo da prompt la funzione anonima

```
>> f = @(x) sin(x) / x ;
```

il cui function handle è **f**. Possiamo quindi passare **f** ad un'altra funzione anonima **is_positive** che in questo caso verifica se il valore della funzione passata tramite function handle, valutata in **x**, è positiva:

```
>> is_positive = @( f_handle , x ) f_handle(x)>0 ;
>> is_positive( f , pi/2 )
ans =
  logical
   1
```

- Si può anche passare una funzione anonima direttamente, senza preventiva definizione, come argomento di funzione:

```
>> is_positive( @(x)( sin(x)/x ) , pi/2 )
ans =
  logical
   1
```

- Analogamente, possiamo passare un function handle ad una funzione esterna definita in un file funzione .m:

is_positive.m

```
function y = is_positive( f_handle , x )
  y = f_handle(x) > 0 ;
end
```

e richiamarla in maniera identica ai precedenti casi.

- Per ottenere il function handle **f_handle** di una funzione non anonima, cioè definita esternamente in un file funzione **nome_funzione.m**, comprese le funzioni native di MATLAB, bisogna utilizzare il simbolo **@**:

```
f_handle = @nome_funzione
```

```
>> f = @sinc
f =
  function_handle with value:
    @sinc
```

my_sinc.m

```
function y = my_sinc( x )
  y = sin( x ) / x ;
end
```

```
>> f = @my_sinc
f =
  function_handle with value:
    @my_sinc
```

- I function handle delle funzioni non anonime possono essere usati in maniera del tutto identica a quelli delle funzioni anonime.

- Il metodo delle secanti è un metodo per la ricerca degli zeri di una funzione reale $f(x)$. Date due ascisse x_1, x_2 ed i due corrispondenti valori della funzione $f_1 = f(x_1), f_2 = f(x_2)$, si ricerca lo zero della retta che passa per questi due punti:

$$\alpha f_1 + (1-\alpha) f_2 = 0 \quad \Rightarrow \quad \alpha = \frac{-f_2}{f_1 - f_2} = \frac{\bar{x} - x_2}{x_1 - x_2}$$

  da cui si ottiene la soluzione

$$\Delta = \bar{x} - x_2 = -f_2 \frac{x_1 - x_2}{f_1 - f_2}$$

  procedendo poi iterativamente a partire dalle due nuove ascisse $x_2, \bar{x}$.
- Scrivere una funzione che prenda in input una funzione f, due ascisse di partenza $x_1, x2$ e determini un'approssimazione di uno zero di f mediante il metodo delle secanti. Si suppone che f abbia uno zero che possa essere trovato a partire da $x_1, x2$.
- A meno che f sia essa stessa una retta, dovremo definire un criterio per arrestare il procedimento iterativo: per esempio quando $|\Delta| < tol$ per un valore $tol > 0$ fornito (distanza assoluta). Ovviamente non possiamo spingerci al di sotto dell'epsilon di macchina: $tol > \epsilon \cdot x$ dove x è lo zero $f(x) = 0$.

metodo_secanti.m

```
% Input: f, function handle della funzione f
%        x = [ x1 x2 ], vettore delle due ascisse iniziali
%        tol, tolleranza assoluta sulla distanza tra le due ascisse finali
% Output: z, zero di f, f(z)=0

function z = metodo_secanti(f, x, tol)
  fx = f( x ) ;
  Delta = x(2) - x(1) ;
  while abs( Delta ) > tol
    % Calcolo del Delta dalla formula delle secanti
    Delta = -fx(2) * ( x(1)-x(2) ) / ( fx(1)-fx(2) ) ;

    % Calcolo della nuova x e della nuova f
    new_x = x(2) + Delta ;
    new_f = f( new_x ) ;

    % Spostamento dei vecchi valori da posizione 2 a posizione 1
     x(1) =  x(2) ;
    fx(1) = fx(2) ;

    % Aggiornamento dei nuovi valori in posizione 2
     x(2) = new_x ;
    fx(2) = new_f ;

    % Verifica caso che new_x sia lo zero esatto: uscita dal ciclo while
    if new_f == 0 ; break ; end
  end

  % Assegnazione output
  z = x(2) ;
end
```

- Utilizzo con funzioni f anonime:

```
>> f_1   = @(x) sin(2*x) - x ;
>> I_1   = [ pi/4 pi/2 ] ;
>> tol_1 = 1d-15 ;
>> x1 = metodo_secanti(f_1, I_1, tol_1)
x1 =
     0.947747133516990
>> f_1( x1 )
ans =
     0
```

```
>> x2 = metodo_secanti(@(x) sin(x), [pi/4 3*pi/2], 1e-12) ;
>> x2 = metodo_secanti(@sin       , [pi/4 3*pi/2], 1e-12)
x2 =
     3.141592653589793
```

```
>> x3 = metodo_secanti(@(x) exp(x)+x, [0 1], 1e-13)
x3 =
    -0.567143290409784
```

- Utilizzo con funzione f definita in un file funzione .m:

log_plus_square.m

```
function y = log_plus_square( x )
  y = log(x) + x .^ 2 ;
end
```

```
>> x4 = metodo_secanti(@log_plus_square, [1 2], 1e-13)
x4 =
     0.652918640419205
```

- Il metodo di Newton è un metodo per la ricerca degli zeri di una funzione reale $f(x)$. Differisce dal metodo delle secanti per l'uso dell'approssimante lineare invece della retta secante. Data un'ascissa x_0 ed i due corrispondenti valori della funzione $f_0 = f(x_0)$ e della derivata prima $f'_0 = f'(x_0)$, si ricerca lo zero dell'approssimante lineare in x_0:

$$f(x) \approx f_0 + f'_0(x - x_0) = 0 \quad \Rightarrow \quad \Delta = x - x_0 = \frac{-f_0}{f'_0}$$

  procedendo poi iterativamente a partire dalla nuova ascissa x.
- Scrivere una funzione che prenda in input una funzione f e la funzione derivata f', un'ascissa di partenza x_0 e determini un'approssimazione di uno zero di f mediante il metodo di Newton. Si suppone che f abbia uno zero che possa essere trovato a partire da x_0.
- Utilizzeremo sempre la condizione $|\Delta| < tol$ per l'arresto del procedimento.
- Dovremo fare attenzione che la derivata f' che passiamo alla funzione dovrà essere corretta, cioè dev'essere effettivamente la derivata della f passata alla funzione, altrimenti il procedimento può divergere.

metodo_Newton.m

```
% Input: f,  function handle della funzione f
%        df, function handle della derivata di f, f'
%        x,  ascissa di partenza
%        tol, tolleranza assoluta sulla distanza tra le due ascisse finali
% Output: x, zero di f, f(x)=0

function x = metodo_Newton(f, df, x, tol)
  Delta = Inf ;
  while abs( Delta ) > tol
    % Calcolo del Delta dalla formula di Newton
    f0 = f(x) ;
    Delta = -f0 / df(x) ;

    % Verifica caso che f0 sia lo zero esatto: uscita dal ciclo while
    if f0 == 0 ; break ; end

    % Calcolo della nuova x
    x = x + Delta ;
  end
end
```

- Utilizzo, per semplicità solo con funzioni anonime:

```
>> x1 = metodo_Newton(@(x) sin(2*x)-x, @(x) 2*cos(2*x)-1, pi/2, 1d-15)
x1 =
    0.947747133516990

>> x2 = metodo_Newton(@sin, @cos, 3, 1d-15)
x2 =
    3.141592653589793
```

- Si può far uso dell'approssimante quadratica invece che lineare per trovare gli zeri di una funzione reale $f(x)$.Data un'ascissa x_0 ed i tre corrispondenti valori della funzione $f_0 = f(x_0)$, della derivata prima $f_0' = f'(x_0)$ e della derivata seconda $f_0'' = f''(x_0)$, si ricerca lo zero dell'approssimante quadratico in x_0:

$$f(x) \approx f_0 + f_0'(x - x_0) + \frac{f_0''}{2}(x - x_0)^2 = 0$$

$$\Rightarrow \quad \Delta = x - x_0 = \frac{-2f_0}{f_0' \pm \sqrt{(f_0')^2 - 2f_0 f_0''}}$$

  procedendo poi iterativamente a partire dalla nuova ascissa x.
- Scrivere una funzione che prenda in input una funzione f, la sua funzione derivata prima f' e la sua funzione derivata seconda f'', un'ascissa di partenza x_0 e determini un'approssimazione di uno zero di f mediante il metodo di Newton. Si suppone che f abbia uno zero che possa essere trovato a partire da x_0.
- Utilizzeremo sempre la condizione $|\Delta| < tol$ per l'arresto del procedimento.
- Dovremo fare attenzione che le derivate f' e f'' che passiamo alla funzione dovranno essere corrette, altrimenti il procedimento può divergere.

metodo_quadratico.m

```
% Input: f,   function handle della funzione f
%        df,  function handle della derivata prima di f, f'
%        ddf, function handle della derivata seconda di f, f"
%        x,  ascissa di partenza
%        tol, tolleranza assoluta sulla distanza tra le due ascisse finali
% Output: x, zero di f, f(x)=0

function x = metodo_quadratico(f, df, ddf, x, tol)
  Delta = Inf ;
  while abs( Delta ) > tol
    % Calcolo del Delta dalla formula quadratica
      f0   =   f(x) ;
     df0   =  df(x) ;
    ddf0   = ddf(x) ;
    radice = sqrt( df0^2 - 2*f0*ddf0 ) ;

    % Selezione del segno della radice
    if df0 > 0
      den = df0 + radice ;
    else
      den = df0 - radice ;
    end
    Delta = -2 * f0 / den ;

    % Verifica caso che f0 sia lo zero esatto: uscita dal ciclo while
    if f0 == 0 ; break ; end

    % Calcolo della nuova x
    x = x + Delta ;
  end
end
```

- Considerato il problema dell'esercizio `Parte2_Es4.m`, si utilizzi una funzione esterna, definita in un file funzione .m, per raccogliere le operazioni comuni a tutti e due i cicli `while`.
- Le operazioni in comune ai due cicli `while` sono quasi tutte, con l'eccezione del messaggio da visualizzare nel comando `input` e dell'ultima condizione nell'espressione per la variabile `input_invalido`, che sono diverse: la funzione che scriveremo assumerà quindi in input questi due elementi.

`input_con_check.m`

```
% Input: messaggio, vettore di char oppure stringa
%        condizione, function handle alla condizione che si vuole sull'input
% Output: x, variabile inserita da tastiera e verificata

function x = input_con_check( messaggio , condizione )
  input_invalido = true ;
  while input_invalido
    x = input( messaggio ) ; % Uso del messaggio in input
    input_invalido = ~( numel(x)==1   && ...
                        isnumeric(x)  && ...
                        x==floor(x)   && ...
                        condizione(x) ) ; % Condizione aggiuntiva
    if input_invalido ; fprintf('Valore invalido!\n') ; end
  end
end
```

```
n=input_con_check('Inserire il numero intero positivo n: ', @(x) x>=0 ) ;
b=input_con_check('Inserire la base b>1: ', @(x) x>1 ) ;
% Algoritmo delle divisioni successive...
```

- Limite di una funzione $\lim_{x \to x_0} f(x) = l$:

$$\forall \varepsilon > 0 \;\; \exists \delta > 0 \;\; : \;\; 0 < |x - x_0| < \delta \Rightarrow |f(x) - l| < \varepsilon$$

- Dal punto di vista operativo (ma un pò scorretto), ciò vuol dire che fissato un $\varepsilon > 0$ piccolo a piacere, potremo sempre trovare un δ sufficientemente piccolo tale per cui $|\varepsilon_f| < \varepsilon$, con $\varepsilon_f = f(x_0+\delta) - f(x_0+\delta/2)$, per il quale $f(x_0+\delta)$ e $f(x_0+\delta/2)$ sono ragionevolmente vicini al limite l.
- Non troveremo mai il valore esatto di l ma solo delle sue approssimazioni, più o meno buone, riducendo via via ε, facendo attenzione.
- Il precedente ragionamento può essere esteso anche nel caso di $x \to \pm\infty$, pensando di raddoppiare δ invece di dimezzarlo, con δ concorde al segno del $\pm\infty$ cercato.
- Possiamo tenere conto anche del caso $l = \pm\infty$: se nel procedimento $|f(x)| > F$ per un valore di soglia F sufficientemente grande, assegneremo un limite $\pm$`Inf` col giusto segno.

limite.m

```
% Input: f, function handle della funzione f(x)
%        x0, ascissa del limite lim f(x) x->x0
%        delta, salto iniziale ascissa x = x0 + delta
%        epsilon, tolleranza su f: |f(x_new)-f(x)| < epsilon
%        limite_puntuale, booleano: true se si vuole x->x0,
%                                   false per x->±Infinito
% Output: fx, il limite lim f(x) x->x0

function fx = limite( f , x0 , delta , epsilon , limite_puntuale )
  % Funzione anonima per calcolare f(x+step)
  f_x_plus_step = @(x, step) f(x+step) ;

  % Valori iniziali
  fx = f_x_plus_step(x0, delta) ; % f(x0+delta) iniziale
  epsilon_f = Inf ;               % differenza tra le f iniziale
  while abs( epsilon_f ) > epsilon
    if limite_puntuale
      delta = delta / 2 ; % per tendere a x0
    else
      delta = delta * 2 ; % per tendere a ±Infinito
    end
    fx_new = f_x_plus_step(x0, delta) ; % f(x0+delta)
    epsilon_f = fx_new - fx ;           % differenza tra le f
    fx = fx_new ; % spostamento di fx_new in fx per l'iterazione successiva
    % Caso di limite ±Infinito
    if abs(fx) > 1e6 % Soglia F
      if fx > 0 ; fx = +Inf ; else ; fx = -Inf ; end
    break ;
    end
  end
end
```

- Utilizzo nel caso di alcuni limiti notevoli:

```
% (1+1/x)^x -> e per x -> +Infinito
f = @(x) (1+1/x)^x ;
x0        = 1 ;      % x -> +Infinito, attenzione a non esagerare
delta     = 1 ;      % attenzione a non esagerare
epsilon   = 1e-6 ;   % non scendere sotto eps di macchina
flag_tipo = false ; % non puntuale, x -> +Infinito
l = limite( f , x0 , delta , epsilon , flag_tipo ) ;
disp(f) ;
disp(l) ;
```

```
>> @(x)(1+1/x)^x
  2.718281180370128
```

```
% sin(x)/x -> 1 per x -> 0
f = @(x) sin(x)/x ;
x0        = 0 ;      % x -> 0
delta     = pi/2 ;
epsilon   = 1e-12 ; % non scendere sotto eps di macchina
flag_tipo = true ;  % puntuale, x -> x0
l = limite( f , x0 , delta , epsilon , flag_tipo ) ;
disp(f) ;
disp(l) ;
```

```
>> @(x)sin(x)/x
  0.999999999999907
```

- La visualizzazione delle variabili vettoriali/matriciali per via numerica nella command window risulta impraticabile, oltre che scomdoda, quando le dimensioni non sono esigue ($M, N > 10$).
- In tal caso risulta più pratico ed immediato un output in forma di grafico 2D. Il caso più banale, ma importantissimo, è quello dello studio di una funzione di una variabile.
- Si utilizza la funzione `plot` che mette in forma grafica i dati che vogliamo visualizzare per mezzo di elementi grafici (linee, simboli, ecc.):

  `plot( x , y , formato )`

  dove `x` e `y` possono essere vettori o matrici della stessa dimensione; nel caso di matrici, viene visualizzata una curva per ciascuna coppia di colonne corrispondenti di `x` e `y`. `formato` è costituito da un stringa o da un vettore di caratteri che definiscono lo stile della curva:

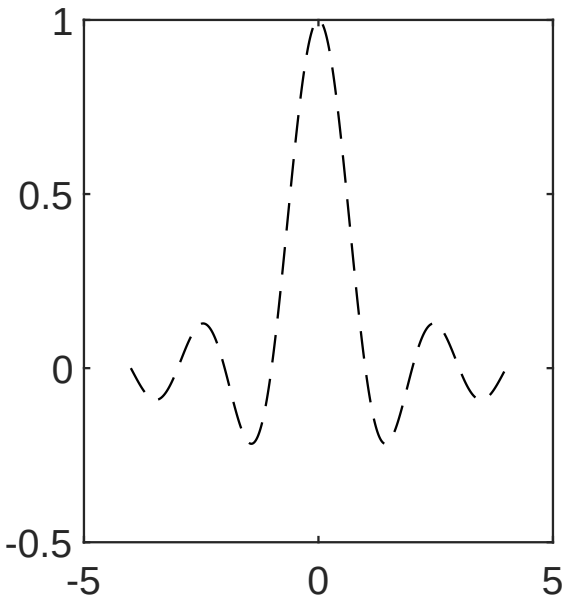
  `formato = '(colore)(tipo linea)(marker)'`

  in qualsiasi ordine, e non è necessario definire sempre tutte e tre le proprietà: le proprietà non definite vengono assunte di default (colore blu, linea continua senza marker; eccezione: marker senza linea):
- `(colore)` è definito con un carattere: `y,m,c,r,g,b,w,k`
- `(tipo linea)` è definito con: `- , -- , : , -.`
- `(marker)` è definito con un carattere: `o,+,*,.,x,s,d,^,v,>,<,p,h`

- Esempi:

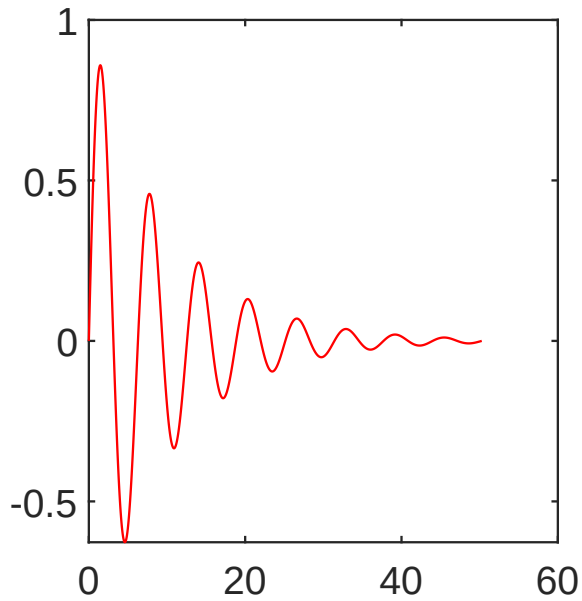
$f(x) = \sin \pi x / \pi x = \operatorname{sinc} x$ in $[-4, 4]$

```
x = linspace( -4 , 4 , 1000 ) ;
y = sinc( x ) ;
plot( x , y , '--k' ) ;
```



$f(x) = e^{-x/10} \sin x$ in $[0, 16\pi]$

```
x = linspace( 0 , 16*pi , 1000 ) ;
y = exp(-x) .* sin(x) ;
plot( x , y , 'r' ) ;
```



- Alcune funzioni/comandi utili:
  - `hold on`: per sovrascrivere sui grafici precedenti senza cancellarli;
  - `xlabel(txt)`, `ylabel(txt)`: per visualizzare il testo `txt` in corrispondenza degli assi. Esempio: `xlabel('Altezza [m]')`;
  - `xlim(l)`, `ylim(l)`: per restringere gli assi ai valori di `l`. Esempio: `xlim([-5 5])`.

- Tracciare il grafico e gli asintoti della funzione

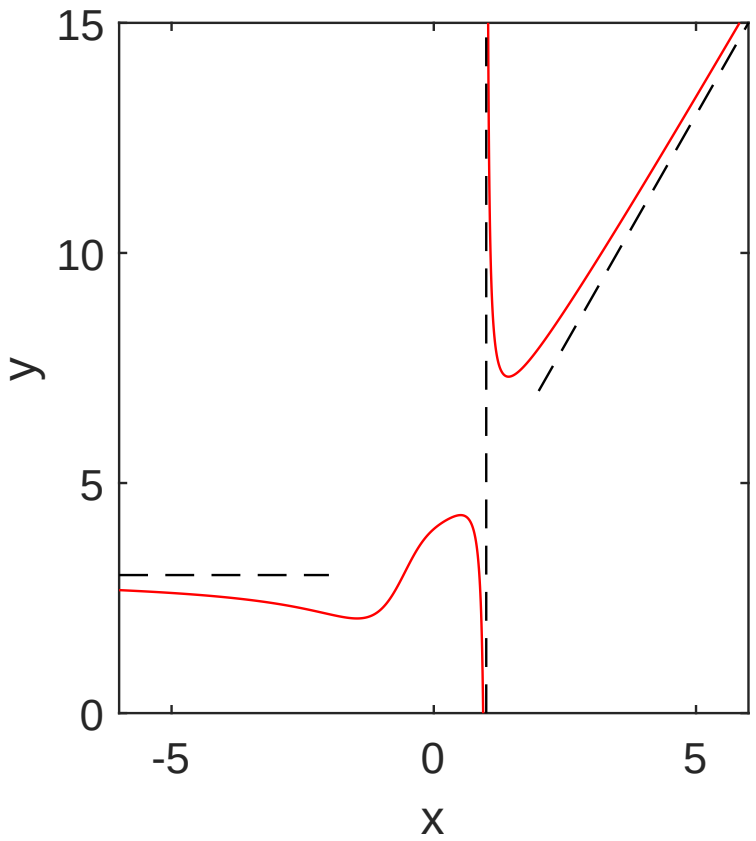
$$f(x) = \frac{x^4 + 3x^3 + 2x^2 - x - 4}{x^3 - 1} + x \tanh x$$

- Sappiamo che:
  - $Dom(f) = \mathbb{R} \setminus \{x_v = 1\} = (-\infty, 1) \cup (1, +\infty)$
  - $l = \lim_{x \to -\infty} f(x) = 3$
  - per $x \to +\infty$ la funzione segue l'asintoto obliquo $y = mx + q = 2x + 3$.

```
f = @(x) (x.^4+3*x.^3+2*x.^2-x-4) ./ (x.^3-1) + x.*tanh(x) ;

% Dominio
estremi_dominio = [ -6 6 ] ;
xv = 1 ;
N  = 1000 ;
dominio_sx = linspace( estremi_dominio(1) , xv , N ) ; % [ -6 , 1 ]
dominio_dx = linspace( xv , estremi_dominio(2) , N ) ; % [  1 , 6 ]
dominio    = [ dominio_sx , dominio_dx ] ;

% Funzione e asintoti
f_dom = f( dominio ) ;
l = limite( f             , -10 ,  -1 , 1e-10 , false ) ; % x → -Inf
s = limite( f             ,  xv , -.1 , 1e-10 , true  ) ; % x →  xv-
d = limite( f             ,  xv , +.1 , 1e-10 , true  ) ; % x →  xv+
m = limite( @(x) f(x)/x   ,  10 ,  +1 , 1e-10 , false ) ; % x → +Inf
q = limite( @(x) f(x)-2*x ,  10 ,  +1 , 1e-10 , false ) ; % x → +Inf
```




```
% Plot
%  funzione
plot( dominio , f_dom , 'r' ) ;
hold on ;

%  asintoto orizzontale x → -Inf
x = [ estremi_dominio(1) , -2 ] ;
y = l + 0*x ;
plot( x , y , '--k' ) ;

%  asintoto obliquo x → +Inf
x = [ 2 , estremi_dominio(2) ] ;
y = m*x + q ;
plot( x , y , '--k' ) ;

%  asintoto verticale x → xv
y = [ -1000 1000 ] ;
plot( [ xv xv ] , y , '--k' ) ;

xlabel('x') ;
ylabel('y') ;
ylim( [ 0 15 ] ) ;
```

- I limiti calcolati risultano essere `l` ≈ 3, `s=-Inf`, `d=Inf`, `m` ≈ 2 e `q` ≈ 3. Infatti la funzione si può scrivere anche nella seguente forma:

$$f(x) = \frac{2x^2 - 1}{x^3 - 1} + (1 + \tanh x)x + 3$$

- Considerando la seguente definizione di derivata di $f(x)$:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

definire una funzione che prenda in ingresso una funzione `f` ed un valore x_0 e ne calcoli la derivata in quel punto utilizzando la funzione `limite` utilizzata precedentemente.

```
% Funzione anonima che calcola la derivata di f in x0
d_dx = @(f,x0) limite( @(h) ( f(x0+h)-f(x0-h) ) / (2*h) , ...
                       0 , .1 , 1e-6 , true ) ;
```

- Utilizzo:

```
% d(5x)/dx = 5
>> d_dx( @(x) 5*x , 1 )
ans =
     5
% d(x^2)/dx = 2x
>> d_dx( @(x) x^2 , 1 )
ans =
     2.000000000000000
% d(sin x)/dx = cos x
>> d_dx( @sin , 0 )
ans =
     0.999999898274743
% d(atan x)/dx = 1 / ( 1+x^2 )
>> d_dx( @atan , 2 )
ans =
     0.200000286458568
```

- Considerata la funzione $f(x)$ dell'esercizio `Parte2_Es17`, si calcolino tutti i punti di minimo e di massimo relativo, mediante gli strumenti ed i metodi visti precedentemente
- Utilizzeremo il metodo delle secanti per trovari gli zeri della derivata di $f(x)$, che definiremo mediante la funzione `limite` come nell'esercizio precedente.
- Il metodo delle secanti ha bisogno di un intervallo di partenza sufficientemente vicino allo zero per poter convergere: per determinare questo intervallo per ognuno degli zeri di $f'(x)$ utilizzeremo il primo metodo visto, quello della ricerca del cambio del segno (`Parte1_Es8`):

`metodo_cambio_segno.m`

```
% Input: f, function handle della funzione f
%        ab = [ a b ], vettore degli estremi dell'intervallo I = (a,b)
%        N, numero di punti in cui suddividere l'intervallo
% Output: x12, matrice 2 x M delle ascisse entro le quali f cambia segno:
%              per l'i-èsimo zero, f cambia segno nell'intervallo definito
%              dalla colonna i-èsima di x12
function x12 = metodo_cambio_segno( f , ab , N )
  x = linspace( ab(1) , ab(2) , N ) ; % valori equidistanziati di x su I
  x = x(2:end-1) ;                    % interv. aperto: esclusione estremi

  segno_f = sign( f(x) ) ; % segno della funzione

  ix = find( segno_f(1:end-1) ~= segno_f(2:end) ) ; % Ricerca cambio segno

  x12 = [ x(ix) ; x(ix+1) ] ; % intervallini entro i quali f cambia segno
end
```

- Determinazione degli intervalli di partenza:

```
% Definizione funzione
g = @(x) (x.^4+3*x.^3+2*x.^2-x-4) ./ (x.^3-1) + x.*tanh(x) ;

% Dominio di interesse
estremi_dominio = [ -6 6 ] ; % estremi
xv = 1 ;                     % valore di x che annulla il denominatore

% Derivata approssimata per determinare gli intervalli di partenza
h_approx     = 1e-3 ;
dg_dx_approx = @(x) ( g(x+h_approx)-g(x-h_approx) ) / (2*h_approx) ;

% Determinazione degli intervalli di partenza con metodo del cambio di segno
%  zeri a sx dell'asintoto verticale xv = 1
N     = 1000 ;
ab_sx = [ estremi_dominio(1) xv ] ; % intervallo
 z_sx = metodo_cambio_segno( dg_dx_approx , ab_sx , N ) ;

%  zeri a dx dell'asintoto verticale xv = 1
ab_dx = [ xv estremi_dominio(2) ] ; % intervallo
 z_dx = metodo_cambio_segno( dg_dx_approx , ab_dx , N ) ;

%  concatenazione zeri a sinistra e a destra dell'asintoto xv = 1
z = [ z_sx z_dx ] ;
```

- Richiamando a questo punto **z** da prompt otteniamo:

```
>> z
z =
   -1.4665    0.5095    1.4254
   -1.4595    0.5165    1.4304
```

- Calcolo dei punti di massimo/minimo con il metodo delle secanti:

```
% Derivata mediante limite
%  derivata per una generica funzione
 d_dx = @(f,x0) limite( @(h)(f(x0+h)-f(x0-h))/(2*h), 0, .1, 1e-6, true) ;
%  derivata della nostra funzione g
dg_dx = @(x) d_dx(g,x) ;

% Zeri della derivata mediante metodo delle secanti applicato a dg_dx
n_zeri = size(z, 2) ;
zeri   = zeros(1, n_zeri) ;
for i = 1 : n_zeri
  zeri(i) = metodo_secanti(dg_dx, z(:,i), 1e-6) ;
end
```

- Richiamando **zeri** da prompt otteniamo:

```
>> zeri
zeri =
   -1.4642    0.5127    1.4256
```

- Supponendo di avere già fatto il plot della funzione, possiamo aggiungervi i nuovi punti di minimo/massimo mediante dei marker:

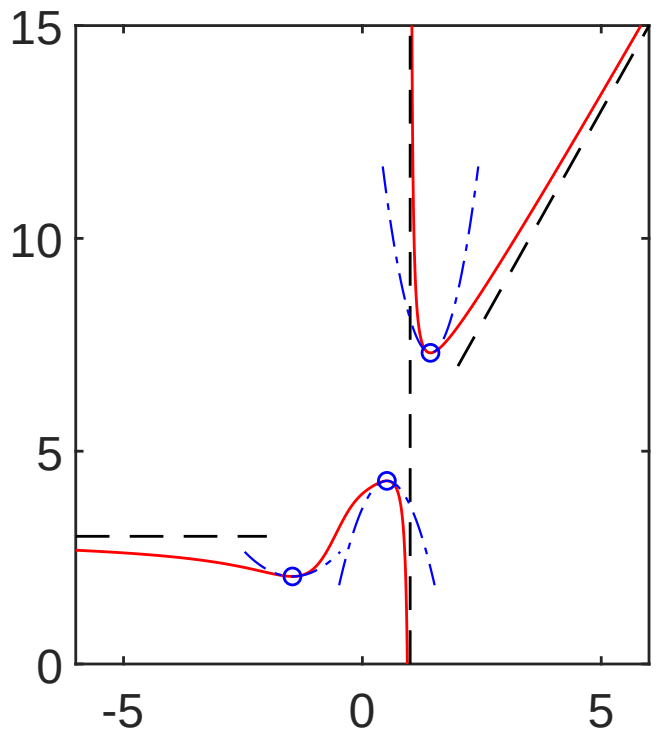
```
% Calcolo dei valori di massimo/minimo di g
g_minmax = g( zeri ) ;

% Plot con marker circolari blu:
plot( zeri , g_minmax , 'ob' ) ;
```

- Definiamo la derivata seconda per disegnare l'approssimante quadratico:

```
% Definizione derivata seconda
ddg_dx = @(x) d_dx(dg_dx,x) ;

% Plot dell'approssimante quadratico nei punti di estremo
x_quadratico = linspace(-1, 1, 400) ;
for i = 1 : n_zeri
  ddg_i = ddg_dx( zeri(i) ) ;
  x_i = x_quadratico + zeri(i) ;
  y_i = g_minmax(i) + (ddg_i/2) * x_quadratico .^ 2 ;
  plot( x_i , y_i , '-.b' ) ;
end
```



- Possiamo verificare direttamente dalla derivata seconda se un punto di estremo relativo è di massimo o di minimo:

```
>> ddg_dx( zeri(1) )
ans =
    1.1595 % > 0 ⇒ min relativo
```

ed essendo positiva conferma che il primo punto di estremo relativo, quello più a sinistra, è un minimo relativo.

- Quando si ha la necessità di input/output di dati di dimensioni non modeste, come ad esempio vettori o matrici, si possono impiegare operazioni di lettura/scrittura su file.
- Le funzioni più immediate per la lettura/scrittura di matrici su file sono `readmatrix` e `writematrix`:

```
matrice = readmatrix( nome_file )
writematrix( matrice , nome_file )
```

  dove `matrice` è la matrice in lettura/scrittura e `nome_file` è il nome del file da cui leggere o su cui scrivere. Vi sono due possibilità per il formato di file impiegato, che viene scelto in base all'estensione del `nome_file` specificato:
  - formato con delimitatore: estensioni `.txt`, `.dat` o `.csv`;
  - foglio elettronico Excel: estensioni `.xls`, `.xlsm` o `.xlsx`.
- `nome_file` può essere specificato utilizzando un indirizzo assoluto, ad esempio `'C:\risultati\file.txt'`, oppure utilizzando un indirizzo relativo rispetto alla cartella di lavoro, ad esempio `'file.txt'` per operare direttamente nella cartella di lavoro, oppure `'sottocartella\file.txt'` per operare nelle sottocartelle della cartella di lavoro.

```
>> x = linspace( 0 , 2*pi , 10 ) ;
>> y = sin( x ) ;
>> writematrix(y , 'sin_x.txt' ) ;
>> writematrix(y', 'sin_xT.txt' ) ;
```

```
>> x = linspace( 0 , 2*pi , 1000 )' ;
>> y = sin( x ) ;
>> writematrix([x y] ,'sin_x.xls' ) ;
>> M = readmatrix( 'sin_x.xls' ) ;
```

- Per l'input/output su file di variabili di tipo generico, non solo matriciali, esistono le funzioni `load` e `save` che permettono di leggere/scrivere non solo una variabile alla volta ma un insieme di variabili `v_1,...,v_N`:

  ```
  load( nome_file , v_1,...,v_N )
  save( nome_file , v_1,...,v_N )
  ```

  Senza ulteriori opzioni, il formato di default del `nome_file` è `.mat` (binario). La lista delle variabili `v_1,...,v_N` è opzionale, cioè se non viene specificata vengono considerate tutte le variabili:
  - in lettura (`load`), tutte le variabili presenti nel file `nome_file`;
  - in scrittura (`save`), tutte le variabili presenti nel workspace.
- Analogamente alle funzione `readmatrix` e `writematrix`, `nome_file` può essere specificato utilizzando un indirizzo assoluto oppure utilizzando un indirizzo relativo rispetto alla cartella di lavoro.
- `save(nome_file)` salva quindi su file tutte le variabili presenti nel workspace, `load(nome_file)` carica nel workspace tutte le variabili presenti nel file, con il nome delle variabili utilizzato al momento del salvataggio.

```
>> x = linspace( 0 , 2*pi , 1000 ) ;
>> y = sin( x ) ;
>> f = @(x) (x.^3-4*x)/tan(x) ;
>> z = f(x) ;
>> save('File_variabili') ;
>> save('File_xf', 'x','f') ;
```

```
>> load('File_variabili', 'y') ;
>> load('File_variabili') ;
```