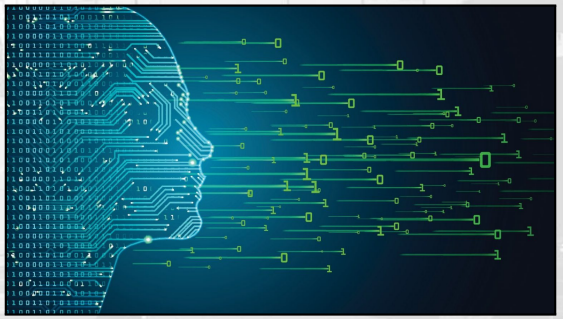
Programmazione Informatica
E. Esercizi

Riccardo Zamolo
rzamolo@units.it

Università degli Studi Trieste
Ingegneria Civile e Ambientale



A.A. 2020-21

- Scrivere uno script che, acquisite da tastiera le coordinate cartesiane (x_i, y_i) di 3 punti nel piano, determini i 3 coefficienti a_i della parabola

$$y = a_1 + a_2 x + a_3 x^2$$

che passa per i 3 punti dati. Inoltre, si forniscano in output:

  - i tre coefficienti a_i;
  - il grafico della parabola su un intervallo $[\bar{x} - \Delta x, \bar{x} + \Delta x]$ dove

$$\bar{x} = \frac{x_{min} + x_{max}}{2}$$

è il punto medio tra gli estremi minimo x_{min} e massimo x_{max} dei valori x_i forniti, mentre $\Delta x = x_{max} - x_{min}$ è la distanza tra questi estremi.

- *Input coordinate:* memorizziamo le coordinate dei 3 punti in una matrice `xy` di dimensione 3×2 (una riga per ogni punto), che possiamo già inizializzare:

```
xy = zeros(3,2) ; % inizializzazione non strettamente necessaria
```

Usiamo `input` per acquisire le coordinate dei 3 punti da tastiera:

```
xy(1,1) = input('Inserire la coordinata x del primo punto: ') ;
xy(1,2) = input('Inserire la coordinata y del primo punto: ') ;
...
xy(3,2) = input('Inserire la coordinata y del terzo punto: ') ;
```

- Potremmo compattare i 6 precedenti input da tastiera in un unico ciclo **for** che itera per ognuno dei 3 punti attraverso la variabile **punto** che assumerà i valori $\{1, 2, 3\}$. Per fare questo bisogna creare due variabili testuali **messaggio_x** e **messaggio_y** che cambiano per ognuno dei punti, utilizzando la funzione **sprintf** per formattare i messaggi impiegando la variabile contatore **punto**:

```
for punto = 1 : 3
  messaggio_x = sprintf('Inserire la coordinata x del punto %d: ', punto) ;
  messaggio_y = sprintf('Inserire la coordinata y del punto %d: ', punto) ;
  xy(punto,1) = input( messaggio_x ) ;
  xy(punto,2) = input( messaggio_y ) ;
end
```

- Potremmo compattare a sua volta il corpo del precedente ciclo **for** in un altro ciclo **for** che itera sulle 2 coordinate x e y attraverso la variabile **coord** che assumerà i valori $\{1, 2\}$. Introduciamo la variabile **caratteri_coord = 'xy'** che conterrà i caratteri corrispondenti alle 2 coordinate indicizzate dalla variabile contatore **coord**:

```
caratteri_coord = 'xy' ;
for punto = 1 : 3
  for coord =  1 : 2
    messaggio = sprintf('Inserire la coordinata %c del punto %d: ', ...
                        caratteri_coord(coord), punto) ;
    xy(punto,coord) = input( messaggio ) ;
  end
end
```

- *Calcolo dei coefficienti:* l'equazione della parabola deve valere per ognuno dei 3 punti (x_i, y_i), ottenendo il sistema lineare

$$\begin{cases} a_1 + a_2 x_1 + a_3 x_1^2 = y_1 \\ a_1 + a_2 x_2 + a_3 x_2^2 = y_2 \\ a_1 + a_2 x_3 + a_3 x_3^2 = y_3 \end{cases} \quad \Rightarrow \quad \mathbf{Va} = \mathbf{y}$$

dove $\mathbf{a} = (a_1, a_2, a_3)^T$ è il vettore colonna dei coefficienti incogniti, $\mathbf{y} = (y_1, y_2, y_3)^T$ è il vettore colonna delle 3 coordinate y e $\mathbf{V}$ è la matrice 3×3

$$\mathbf{V} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix}$$

Scriviamo la matrice $\mathbf{V}$ in termini di operazioni vettoriali utilizzando il vettore colonna `x = xy(:,1)` delle coordinate x ed il vettore riga `e = 0:2` degli esponenti $\{0, 1, 2\}$ della parabola.

```
x = xy(:,1) ;
e = 0 : 2 ;
V = x .^ e ;
```

Risolviamo rispetto al vettore dei coefficienti $\mathbf{a}$ mediante l'operatore backslash \ applicato al vettore colonna delle y, ossia `y = xy(:,2)`:

```
y = xy(:,2) ;
a = V\y ;
```

- *Output testuale dei coefficienti*: usiamo **fprintf** per visualizzare i coefficienti nella forma **ai = valore**, utilizzando per esempio 2 cifre decimali:

```
for i = 1 : 3
  fprintf( 'a%d = %.2f\n' , i , a(i) ) ;
end
```

Eventualmente si può sfruttare l'input vettoriale a **fprintf** per compattare ancora di più il precedente ciclo **for** (ricordiamo l'ordine per colonne nelle matrici):

```
fprintf( 'a%d = %.2f\n' , [ 1:3 ; a' ] ) ;
```

- *Output grafico:* calcoliamo l'intervallo $[\bar{x} - \Delta x, \bar{x} + \Delta x]$ che utilizzeremo per il plot:

```
x_min  = min(x) ;
x_max  = max(x) ;
x_med  = ( x_min + x_max ) / 2 ;
delta  = x_max - x_min ;
x_plot = linspace( x_med-delta , x_med+delta , 1000 )' ; % vettore colonna
```

e calcoliamo i valori della parabola per i valori di x in **x_plot**:

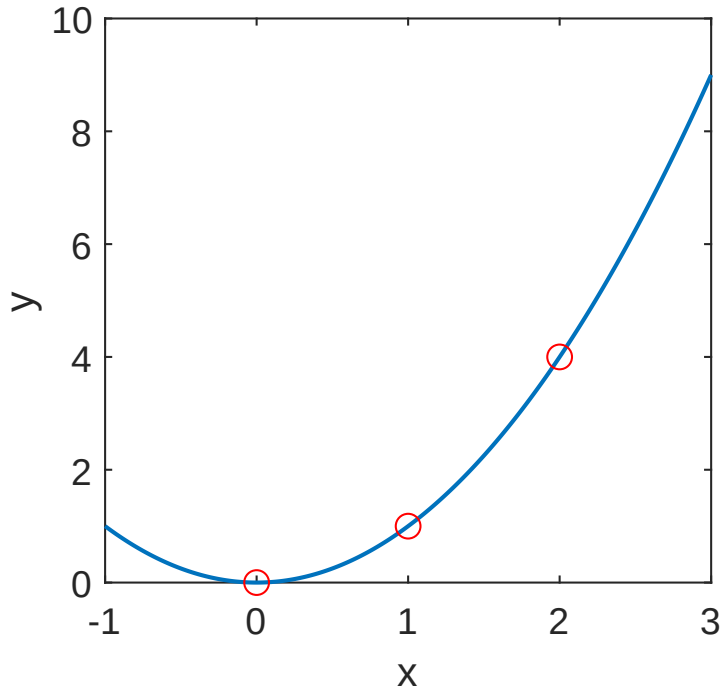
```
y_plot = ( x_plot .^ e ) * a ;
```

(alternativamente possiamo utilizziamo una qualsiasi delle forme viste a lezione per il calcolo vettoriale di un polinomio).

- Eseguiamo il plot della parabola, per esempio specificando un determinato spessore di linea. Visualizziamo anche i tre punti inseriti, utilizzando per esempio dei marker rossi a forma di cerchio:

```
plot( x_plot , y_plot , 'LineWidth' , 1 ) ;
hold on ;
plot( x , y , 'ro' ) ;
xlabel('x') ;
ylabel('y') ;
```

```
Inserire la coordinata x del punto 1: 0
Inserire la coordinata y del punto 1: 0
Inserire la coordinata x del punto 2: 1
Inserire la coordinata y del punto 2: 1
Inserire la coordinata x del punto 3: 2
Inserire la coordinata y del punto 3: 4
a1 = 0.00
a2 = 0.00
a3 = 1.00
```



- Dato lo script del precedente esercizio, modificarlo in maniera tale da acquisire da tastiera un numero arbitrario n di punti per interpolare un generico polinomio di grado $n-1$:

$$y = a_1 + a_2 x + a_3 x^2 + \cdots + a_n x^{n-1}$$

- *Input coordinate:* utilizzando l'ultima forma compatta per gli input, le modifiche necessarie ad acquisire un numero arbitrario di punti sono minime:

```
n = input('Inserire il numero di punti:  ')  ;
xy = zeros(n,2) ; % inizializzazione non strettamente necessaria
caratteri_coord = 'xy' ;
for punto = 1 : n
  ...
```

- *Calcolo dei coefficienti:* sarà necessario modificare solo il secondo estremo del vettore degli esponenti `e`:

```
e = 0 : (n-1) ;
```

- *Output testuale dei coefficienti:* sarà necessario modificare solo il secondo estremo del vettore degli indici:

```
fprintf( 'a%d = %.2f\n' , [ 1:n ; a' ] ) ;
```

- Scrivere una funzione che calcoli il fattoriale doppio $n!!$ di un intero n:

$$n!! = \begin{cases} n(n-2)(n-4)\cdots 5 \cdot 3 \cdot 1 & \text{se } n \text{ dispari} \\ n(n-2)(n-4)\cdots 6 \cdot 4 \cdot 2 & \text{se } n \text{ pari} \end{cases}$$

- Versione con n scalare:

```
function m = double_factorial(n)
  m = prod( n:-2:1 ) ;
end
```

- Versione con n vettoriale: bisogna scrivere esplicitamente un ciclo for per ogni componente del vettore n

```
function m = double_factorial(n)
  m = 0*n ;
  for i = 1 : length(n)
    m(i) = prod( n(i):-2:1 ) ;
  end
end
```

- Data la serie di MacLaurin di una funzione $f(x)$ ed una sua approssimazione $\tilde{f}(x)$ ottenuta troncandola ad un numero finito $i_{max}+1$ di termini:

$$f(x) = \sum_{i=0}^{+\infty} \frac{f^{(i)}(0)}{i!} x^i \quad , \qquad \tilde{f}(x) = \sum_{i=0}^{i_{max}} \frac{f^{(i)}(0)}{i!} x^i = \sum_{i=0}^{i_{max}} a_i x^i$$

  scrivere una funzione che prenda in ingresso
  - un vettore `x` di valori di x,
  - un function handle ad una funzione `derivata_i` che restituisce la derivata i-èsima in $x = 0$, cioè $f^{(i)}(0)$,
  - un intero i_{max},

  e calcoli $\tilde{f}(x)$. Si utilizzi la funzione `calcola_polinomio( a , x )` che calcola un polinomio per un vettore `x` di valori x dati i suoi coefficienti a_i nel vettore `a`; si utilizzi anche la funzione `factorial(n)` che calcola il fattoriale di un vettore di interi `n`.

- Scriviamo intanto la definizione della funzione con gli input e l'output richiesti:

```
% Input: x, vettore
%        derivata_i, function handle della derivata i-èsima di f in 0
%        i_max, massimo numero di termini della serie
% Output: y, serie MacLaurin di f troncata al termine i_max, valutata in x
function y = serie_MacLaurin( x , derivata_i , i_max )
```

- Date le funzioni a disposizione, restano solo da calcolare i coefficienti a_i del polinomio $\tilde{f}$ per $i = 0, \ldots, i_{max}$:

$$a_i = \frac{f^{(i)}(0)}{i!}$$

e li calcoliamo in maniera vettoriale:

```
  i = 0 : i_max ;
  a = derivata_i( i ) ./ factorial( i ) ;
```

- Noto il vettore dei coefficienti `a`, è sufficiente chiamare la funzione messa a disposizione per valutare il polinomio con quei coefficienti per i valori di x del vettore `x`:

```
  y = calcola_polinomio( a , x ) ;
end
```

- Data la funzione `serie_MacLaurin(x, derivate_i, i_max)` del precedente esercizio, che calcola per un vettore `x` di valori x la serie di MacLaurin di una funzione $f(x)$ troncata al termine `i_max`, definire la funzione `derivate_i` che, dato un vettore `i` di valori i, restituisce le corrispindenti derivate i-èsime in $x = 0$, cioè $f^{(i)}(0)$. Considerare i seguenti casi per f:
- $f(x) = e^x \Rightarrow f^{(i)}(x) = e^x \Rightarrow f^{(i)}(0) = 1$. Conviene implementarla come funzione anonima (attenzione all'uscita vettoriale nel caso di costante):

```
derivate_ex = @(i) 1 + 0*i ;
```

- $f(x) = (1-x)^{-1} \Rightarrow f^{(i)}(x) = i!(1-x)^{-(i+1)} \Rightarrow f^{(i)}(0) = i!$. Conviene ancora implementarla come funzione anonima:

```
derivate_geom = @(i) factorial(i) ;
```

- $f(x) = \sin x \quad \Rightarrow \quad f^{(i)}(x) = \sin x, \cos x, -\sin x, -\cos x, \sin x, \ldots$
$\Rightarrow \quad f^{(i)}(0) = 0, 1, 0, -1, 0, 1, \ldots$ Conviene implementarla come funzione esterna:

```
function di = derivate_sin(i)
  di = 0*i ;
  di(2:4:end) =  1 ;
  di(4:4:end) = -1 ;
end
```

- Data la serie di Taylor di una funzione $f(x)$, troncata ad un numero finito N_t di termini:

$$\tilde{f}(x) = \sum_{i=0}^{N_t-1} \frac{f^{(i)}(x_0)}{i!}(x-x_0)^i = \sum_{i=0}^{N_t-1} a_i(x-x_0)^i$$

scrivere una funzione che prenda in ingresso
  - un cell array `cell_df` dei function handle delle derivate di $f(x)$, nell'ordine $\{f(x), f'(x), f''(x), \ldots\}$,
  - un vettore `x` di valori di x,
  - il centro x_0 dell'espansione

e calcoli $\tilde{f}(x)$. Si utilizzi la funzione `calcola_polinomio( a , x )` che calcola un polinomio per un vettore `x` di valori x dati i suoi coefficienti a_i nel vettore `a`; si utilizzi anche la funzione `factorial(n)` che calcola il fattoriale di `n`.

```
function y = serie_Taylor(cell_df, x, x0)
  Nt = length(cell_df) ;
  a  = zeros(Nt, 1) ;
  for i = 0 : Nt-1 % i è l'indice della serie, parte da 0
    ix_v = i+1 ;   % ix_v è l'indice per accedere a vettori/cell array
    df_i = cell_df{ix_v} ; % df_i è un function handle
    a(ix_v) = df_i(x0) / factorial(i) ;
  end
  y = calcola_polinomio(a, x-x0) ;
end
```

- Data una funzione continua $f(x)$ definita dal suo function handle `f`, scrivere una funzione esterna `f_costante` che prende in ingresso `f`, una costante c ed un intervallo $[a, b]$ e calcoli un valore di x tale per cui $f(x) = c$. Si usi la funzione `metodo_secanti(g, ab)` che restituisce uno zero della funzione `g` a partire dall'intervallo `ab`. Si assume per ipotesi che $f(a) < c < f(b)$ o $f(b) < c < f(a)$.
- Poniamo la condizione cercata $f(x) = c$ nella forma $f(x) - c = 0$ in maniera tale da poter cercare uno zero di $g(x) := f(x) - c$ con la funzione `metodo_secanti` messa a disposizione. Implementiamo g come funzione anonima dentro la funzione `f_costante`:

```
function xc = f_costante( f , c , ab )
  g = @(x) f(x) - c ;
```

  e poi richiamiamo `metodo_secanti`:

```
  xc = metodo_secanti(g, ab) ;
end
```

- La funzione appena implementata può quindi essere usata per calcolare la funzione inversa f^{-1} di una funzione f, ossia quella funzione f^{-1} tale che $f^{-1}(f(x)) = x$, sotto opportune ipotesi.

- Il teorema di Lagrange dice che per una funzione $f(x)$ continua e derivabile in un intervallo $[a, b]$ esiste almeno un punto $x_R \in [a, b]$ tale per cui:

$$f'(x_R)(b - a) = f(b) - f(a)$$

- Scrivere una funzione esterna che prenda in ingresso
  - due function handle `f` e `df` relativi ad una funzione continua e derivabile $f(x)$ e alla sua derivata $f'(x)$,
  - un intervallo $[a, b]$ nella forma di due ingressi separati `a` e `b`,

  e calcoli il punto x_R del teorema. Si faccia ancora uso della funzione `metodo_secanti(g, ab)` che restituisce uno zero della funzione `g` a partire dall'intervallo `ab`.
- Si procede analogamente all'esercizio precedente, definendo una nuova funzione $g(x) := f'(x)(b-a) - \big(f(b) - f(a)\big)$ della quale trovare lo zero:

```
function xR = punto_Lagrange( f , df , a , b )
  g = @(x) df(x)*(b-a) - ( f(b) - f(a) ) ;
  xR = metodo_secanti(g, [a b]) ;
end
```

- Si verifichi la soluzione analitica $x_R = e - 1$ nel caso di $f(x) = \log x$ nell'intervallo $[1, e]$:

```
xR = punto_Lagrange( @(x) log(x) , @(x) 1 ./ x , 1 , exp(1) ) ;
errore = xR - (exp(1)-1) ;
disp( errore ) ;
```

- Data la funzione
$$f(x) = e^{-x}(x^2 + x + 1)$$
  - si disegni il grafico $y = f(x)$ nell'intervallo $ab = [-1, 7]$;
  - si calcolino tutti i punti di estremo locale ed i flessi nell'intervallo ab, utilizzando la funzione **`zeri_completi(g, ab)`** che fornisce il vettore di tutti gli zeri della funzione `g` nell'intervallo `ab`;
  - si diagrammino i punti di estremo e di flesso sul grafico della funzione.
- Scriviamo le funzioni anonime per $f(x)$ e per le sue derivate fino a f'':
$$f'(x) = e^{-x}(-x^2 + x) \quad , \qquad f''(x) = e^{-x}(x^2 - 3x + 1)$$

```
f   = @(x) exp(-x) .* (  x.^2 +   x + 1 ) ; % operazioni element-wise con .
df  = @(x) exp(-x) .* ( -x.^2 +   x     ) ;
ddf = @(x) exp(-x) .* (  x.^2 - 3*x + 1 ) ;
```

- Disegniamo il grafico di f nell'intervallo dato, linea nera (**`'k'`**):

```
ab = [-1 7] ;
x = linspace(ab(1), ab(2), 1000 ) ;
plot( x , f(x) , 'k' , 'LineWidth' , 1 ) ; % spessore linea 1 > default=0.5
hold on ;
```

- Aggiungiamo due curve tratteggiate nere agli estremi sinistro e destro:

```
x_sx = linspace(ab(1)-.1, ab(1)   , 100 ) ;
x_dx = linspace(ab(2)   , ab(2)+2 , 100 ) ;
plot( x_sx , f(x_sx) , '--k' , 'LineWidth' , 1 ) ; % spessore linea 1
plot( x_dx , f(x_dx) , '--k' , 'LineWidth' , 1 ) ; % spessore linea 1
```

- Calcoliamo i punti di estremo locale ed i flessi cercando gli zeri di f' e f'' mediante la funzione **zeri_completi** e li diagrammiamo utilizzando dei marker distinti (croce rossa per gli estremi, cerchio blu per i flessi). La funzione **f** è implementata in maniera da accettare ingressi vettoriali:

```
zeri_df = zeri_completi( df , ab ) ;
plot( zeri_df , f(zeri_df) , 'xr' ) ;

zeri_ddf = zeri_completi( ddf , ab ) ;
plot( zeri_ddf , f(zeri_ddf) , 'ob' ) ;
```

- Richiamiamo il vettore **zeri_df** per vedere che valori sono stati ottenuti:

```
>> zeri_df
zeri_df =
     0.0000     1.0000
```

che corrispondono ai valori esatti $\{0, 1\}$ degli zeri di $f'(x) = e^{-x}x(1-x)$.

- Richiamiamo il vettore **zeri_ddf** per controllarne i valori:

```
>> zeri_ddf
zeri_ddf =
     0.3820     2.6180
```

che corrispondono ai valori esatti $\frac{3\pm\sqrt{5}}{2}$ degli zeri di $f''(x)$:

```
>> (3+[-1 1]*sqrt(5))/2
ans =
     0.3820     2.6180
```

- *Parte 2.* Per ogni punto di estremo e di flesso, disegnare il polinomio approssimante di Taylor, troncato al secondo ordine, utilizzando la funzione `serie_Taylor(cell_df, x, x0)` dell'esercizio `ParteE_Es6`. Ciò significa che nei punti di estremo il polinomio approssimante sarà la parabola $\tilde{f} = f(x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2$, mentre nei punti di flesso sarà la retta $\tilde{f} = f(x_0) + f'(x_0)(x - x_0)$.
- Per poter utilizzare la funzione `serie_Taylor(cell_df, x, x0)` messa a disposizione è necessario creare il cell array `cell_df` dei function handle `f`, `df`, `ddf` relativi alla funzione ed alle sue prime due derivate $f(x), f'(x), f''(x)$:

```
cell_df = { f , df , ddf } ;
```

- Disegniamo gli approssimanti per ogni punto di estremo con un ciclo `for` che itera su tutti gli elementi del vettore `zeri_df`, ossia su tutti gli zeri di f'. Nel vettore `x_plot` specifichiamo l'insieme dei valori di x, che andranno centrati in x_0, per i quali disegnare l'approssimante, calcolato mediante `serie_Taylor`:

```
x_plot = linspace(-.5, .5, 100) ;
for x0 = zeri_df
  x = x0 + x_plot ;
  y = serie_Taylor(cell_df, x, x0) ;
  plot( x , y , 'r' ) ;  % plot con linea rossa
end
```

- Per accertarsi se un punto di estremo è di minimo o di massimo, è sufficiente controllare il segno di f'':

```
>> ddf(zeri_df)
ans =
     1.0000   -0.3679
```

che stanno ad indicare che il primo punto di estremo è di minimo (`1.0000`>0) mentre il secondo punto è di massimo (`-0.3679`<0).

- In maniera analoga, disegniamo gli approssimanti per ogni punto di flesso, ossia per ogni zero di f'', cioè per ogni elemento del vettore **`zeri_ddf`**:

```
x_plot = linspace(-1, 1, 100) ;
for x0 = zeri_ddf
  x = x0 + x_plot ;
  y = serie_Taylor(cell_df, x, x0) ;
  plot( x , y , 'b' ) ;  % plot con linea blu
end
```

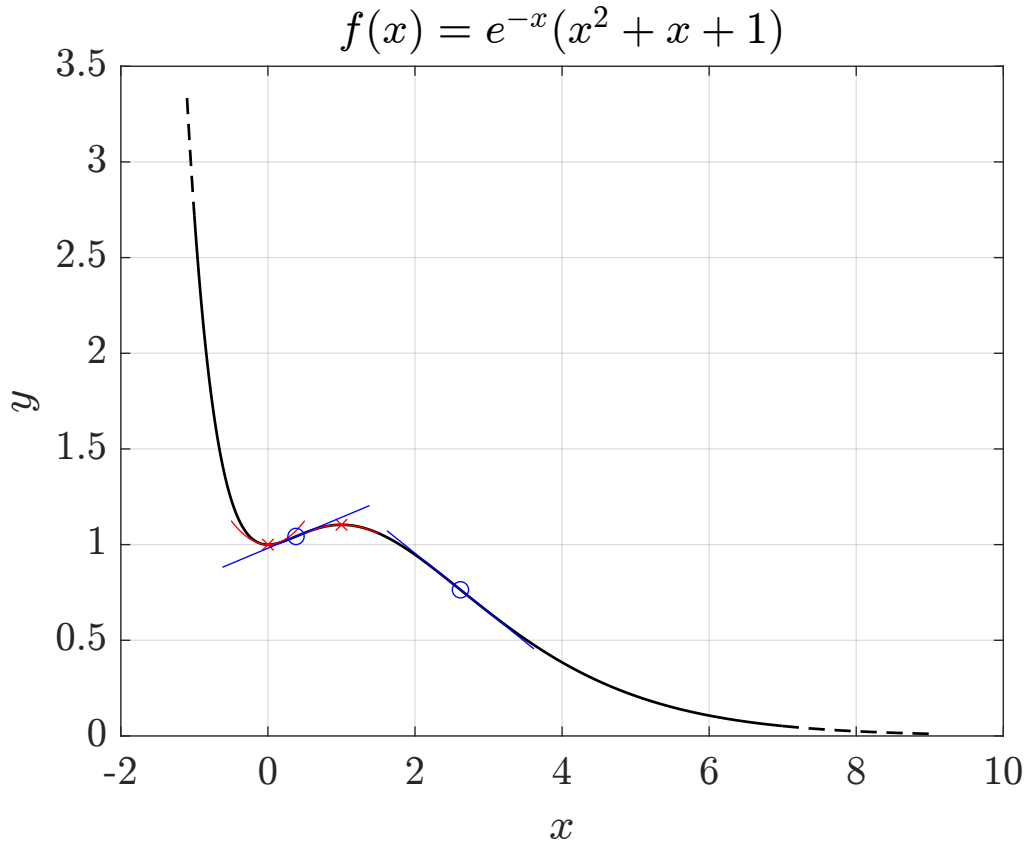
- Possiamo verificare la pendenza del grafico di f nei flessi:

```
>> df(zeri_ddf)
ans =
     0.1611   -0.3090
```

che stanno ad indicare che nel primo punto di flesso il grafico ha pendenza positiva ($\approx 16\%$) mentre nel secondo ha pendenza negativa ($\approx -31\%$)

- Eventualmente si può abbellire il grafico ed esportarlo:

```
xlabel( '$x$' , 'Interpreter' , 'latex') ;
ylabel( '$y$' , 'Interpreter' , 'latex') ;
title( '$f(x)=e^{-x}(x^2+x+1)$' , 'Interpreter' , 'latex') ;
assi = gca ;
assi.TickLabelInterpreter = 'latex' ;
assi.FontSize = 15 ;
grid on ;
exportgraphics( assi , 'plot.pdf' ) ;
```



- Data la funzione
$$f(x) = \frac{x^2+1}{x-1}$$
  - si disegni il grafico $y = f(x)$ nell'intervallo $ab = [-8, 10]$;
  - si calcolino tutti i punti di estremo locale nell'intervallo ab, utilizzando la funzione `zeri_completi(g, cd)` che fornisce il vettore di tutti gli zeri della funzione `g` nell'intervallo `cd`;
  - si diagrammino i punti di estremo sul grafico della funzione;
  - si calcolino e si disegnino gli asintoti $mx + q$ per $x \to \pm\infty$ utilizzando le funzioni `limite_meno_inf(g, epsilon)` e `limite_piu_inf(g, epsilon)` che calcolano $\lim_{x\to\pm\infty} g(x)$. Per `epsilon` utilizzare 10^{-15} nel calcolo di m e 10^{-6} nel calcolo di q.
- Scriviamo le funzioni anonime per $f(x)$ e per le sue derivate fino a f''. Conviene sottrarre e sommare 2 al numeratore per porre f nella forma:
$$f(x) = x + 1 + \frac{2}{x-1}$$
da cui:
$$f'(x) = 1 - \frac{2}{(x-1)^2} \quad , \qquad f''(x) = \frac{4}{(x-1)^3}$$

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

- Essendoci un asintoto verticale per x = `x_verticale` = 1, conviene suddividere l'intervallo dato $ab = [-8, 10]$ in due sottointervalli $[-8, 1]$ e $[1, 10]$ dove disegnare e fare i calcoli in maniera separata. Memorizziamo questi intervalli nelle righe della matrice `x_intervalli`:

```
ab = [-8 10] ;
x_verticale = 1 ;
x_intervalli = [    ab(1)    x_verticale ; ...
                x_verticale    ab(2)    ] ;
```

- Scriviamo un ciclo `for` che itera per ciascuno di questi (2) sottointervalli:

```
n_intervalli = size( x_intervalli , 1 ) ;
zeri_df_vettore = [] ;
for intervallo = 1 : n_intervalli
  x1 = x_intervalli( intervallo , 1 ) ;
  x2 = x_intervalli( intervallo , 2 ) ;
  x  = linspace(x1, x2, 1000) ;
  plot( x , f(x) , 'k' , 'LineWidth' , 1 ) ; % linea nera spessore 1
  hold on ;
  zeri_df = zeri_completi( df , [x1 x2] ) ;
  plot( zeri_df , f(zeri_df) , 'or' ) ; % cerchi rossi
  zeri_df_vettore = [ zeri_df_vettore zeri_df ] ;
end
```

dove all'interno del ciclo `for` svolgiamo in sequenza due operazioni:

- `plot` del grafico di f all'interno del sottointervallo identificato dagli estremi `x1` e `x2` presi dalla matrice `x_intervalli` creata precedentemente;
- calcolo e `plot` dei punti di estremo (zeri della derivata `df`) all'interno del medesimo sottointervallo [`x1`, `x2`] mediante la funzione `zeri_completi(df, [x1 x2])` messa a disposizione.

- Nel vettore **zeri_df_vettore** abbiamo concatenato gli zeri della derivata **df** calcolati separatamente per ciascun sottointervallo. Nel caso specifico, vi sono due zeri $1 \pm \sqrt{2}$, uno per ciascun sottointervallo. Verifichiamolo:

```
>> zeri_df_vettore
zeri_df_vettore =
   -0.4142    2.4142

>> 1+[-1 1]*sqrt(2)
ans =
   -0.4142    2.4142
```

- Calcolo degli asintoti obliqui $mx+q$ per $x \to \pm\infty$ utilizzando le funzioni **limite_meno_inf** e **limite_piu_inf**. Per calcolare i due m definiamo la funzione anonima per $f(x)/x$ con **@(x) f(x)./x**, mentre per calcolare i due q definiamo le corrispondenti funzioni anonime **@(x) f(x)-m*x**:

```
m_sx = limite_meno_inf( @(x) f(x) ./ x    , 1e-15 ) ;
q_sx = limite_meno_inf( @(x) f(x)-m_sx*x , 1e-6  ) ;
m_dx = limite_piu_inf(  @(x) f(x) ./ x    , 1e-15 ) ;
q_dx = limite_piu_inf(  @(x) f(x)-m_dx*x , 1e-6  ) ;
```

ottenendo in entrambi i casi il medesimo asintoto $x+1$ ($m = q = 1$), che si può immediatamente verificare dalla forma $f(x) = x + 1 + \frac{2}{x-1}$.

- I coefficienti m per $x \to \pm\infty$ si possono anche calcolare come $\lim_{x\to\pm\infty} f'(x)$ grazie alla regola di De l'Hôpital:

```
m_sx = limite_meno_inf( df , 1e-15 ) ;
m_dx = limite_piu_inf(  df , 1e-15 ) ;
```

- Disegno degli asintoti:

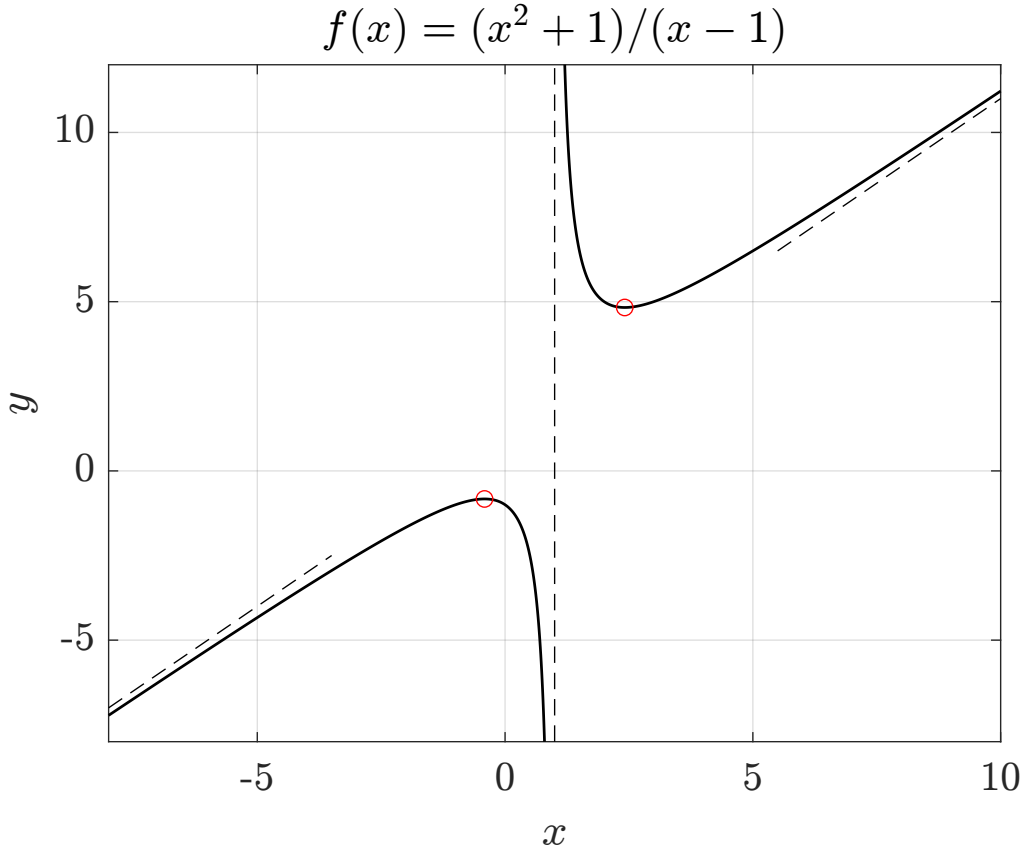
```
% Asintoto verticale in x=1
plot( [x_verticale x_verticale] , [-1000 1000] , '--k' ) ;

% Asintoto obliquo verso -infinito
x_med = ( ab(1) + x_verticale ) / 2 ;
x_sx = linspace( ab(1) , x_med , 1000 ) ;
plot( x_sx , m_sx*x_sx+q_sx , '--k' ) ;

% Asintoto obliquo verso +infinito
x_med = ( x_verticale + ab(2) ) / 2 ;
x_dx = linspace( x_med , ab(2) , 1000 ) ;
plot( x_dx , m_dx*x_dx+q_dx , '--k' ) ;
```

- Limitiamo gli estremi degli assi, soprattutto per l'asse y essendoci un asintoto verticale. Scegliamo in quest'ultimo caso l'intervallo $[-8, 12]$:

```
xlim(ab) ;
ylim([-8 12]) ;
```

- Eventualmente si può abbellire il grafico ed esportarlo:

```
xlabel( '$x$' , 'Interpreter' , 'latex') ;
ylabel( '$y$' , 'Interpreter' , 'latex') ;
title( '$f(x)=(x^2+1)/(x-1)$' , 'Interpreter' , 'latex') ;
assi = gca ;
assi.TickLabelInterpreter = 'latex' ;
assi.FontSize = 15 ;
grid on ;
exportgraphics( assi , 'plot.pdf' ) ;
```



- Per il calcolo approssimato della derivata di una funzione $g(x)$, si possono usare le due seguenti funzioni anonime, che prendono entrambe in ingresso il function handle `g` di $g(x)$ ed il valore `x` della x per la quale si vuole $g'(x)$:
  - Rapporto incrementale con passo fisso h:

```
h       = 1e-4 ;
d_dx_h = @(g,x) ( g(x+h) - g(x-h) ) / (2*h) ;
```

  - Limite approssimato del rapporto incrementale, facendo uso della funzione `limite`:

```
d_dx_lim = @(g,x) limite(@(h) ( g(x+h)-g(x-h) ) / (2*h) , ...
                                    0 , .1 , 1e-6 , true ) ;
```

- La prima implementazione (`d_dx_h`) accetta ingressi `x` vettoriali mentre la seconda (`d_dx_lim`) no; la seconda implementazione è però formalmente più corretta.
- Entrambe le precedenti funzioni possono essere pensate come degli operatori che applicati ad una particolare funzione `g` restituiscono un'approssimazione della funzione derivata di `g`.

- Scrivere una funzione esterna **derivata_n** che prenda in ingresso
  - un function handle **operatore_derivata_prima(g,x)** di uno dei due precedenti operatori derivata,
  - un function handle **f(x)** ad una funzione $f(x)$,
  - un intero **n**

  e fornisca in output il function handle della derivata n-èsima di $f(x)$, ossia $f^{(n)}(x)$.
- La derivata n-èsima si ottiene applicando ripetutamente n volte l'operatore derivata (prima) alla funzione f: $\frac{\mathrm{d}}{\mathrm{d}x}\frac{\mathrm{d}}{\mathrm{d}x}\cdots\frac{\mathrm{d}f(x)}{\mathrm{d}x}$. Utilizzeremo quindi un ciclo **for** che ripete n volte l'applicazione dell'operatore derivata (approssimato).

```
function f = derivata_n( operatore_derivata_prima , f , n )
  for derivata = 1 : n
    f = @(x) operatore_derivata_prima(f, x) ;
  end
end
```

- $\mathcal{NB}$: nel caso di n non piccolo, ad es. $n = 5$, l'uso dell'operatore **d_dx_lim** è molto inefficiente poichè ogni derivata richiede un limite di una funzione che è a sua volta definita da un limite e così via, portando ad una complessità esponenziale. Per il calcolo accurato delle derivate successive esistono altri metodi molto più efficienti ma un pò più complessi.