Elaborato di

Calcolo Numerico

Anno Accademico 2017/2018

Mattia D'Autilia - 5765968 mattia.dautilia@stud.unifi.it
Yuri Bacciarini - 5654547 yuri.bacciarini@stud.unifi.it

May 15, 2018

Capitoli

1 Capitolo 1

1.1 Esercizio 1

Volendo conoscere quanto un errore influenzi il risultato quando $x = 0$, si definisce l'errore relativo:

$$\epsilon_x = \frac{\tilde{x} - x}{x}$$

da cui :

$$\tilde{x} = x(1 + \epsilon_x), \text{ e quindi } \frac{\tilde{x}}{x} = 1 + \epsilon_x$$

ovvero l'errore relativo deve essere comparato a 1: un errore relativo vicino a zero indicherà che il risultato approssimato è molto vicino al risultato esatto,mentre un errore relativo uguale a 1 indicherà la totale perdita di informazione.

Con $x = e \approx 2.7183 = \tilde{x}$, l'errore relativo è quindi $\epsilon_x = \frac{2.7183 - e}{e} = 6.6849e - 06$

Il numero di cifre significative k corrette all'interno di $\tilde{x}$ si definisce con la formula :

$$k = -\log(2|\epsilon_x|)$$

In questo caso il risultato del calcolo è $k = 4.8739$, che è abbastanza vicino alla realtà di $k = 5$ cifre significative corrette.

1.2 Esercizio 2

Partiamo da :

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Usando gli sviluppi di Taylor fino al secondo ordine otteniamo:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(\xi_x)h^3$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(\mu_x)h^3$$

Al numeratore otteniamo

$$f(x+h) - f(x-h) = 2f'(x)h + \frac{1}{6}(f'''(\xi_x)h^3 + f'''(\mu_x)h^3)$$

La relazione iniziale diventa

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} - \frac{1}{12}(f'''(\xi_x) + f'''(\mu_x))h^2$$

Abbiamo quindi verificato, usando gli sviluppi di Taylor fino al secondo ordine con resto in forma di Lagrange, se f $\in C^3$ risulta

$$f'(x) = \phi_h(x) + O(h^2)$$

dove

$$\phi_h(x) = \frac{f(x+h) - f(x-h)}{2h}$$

1.3 Esercizio 3

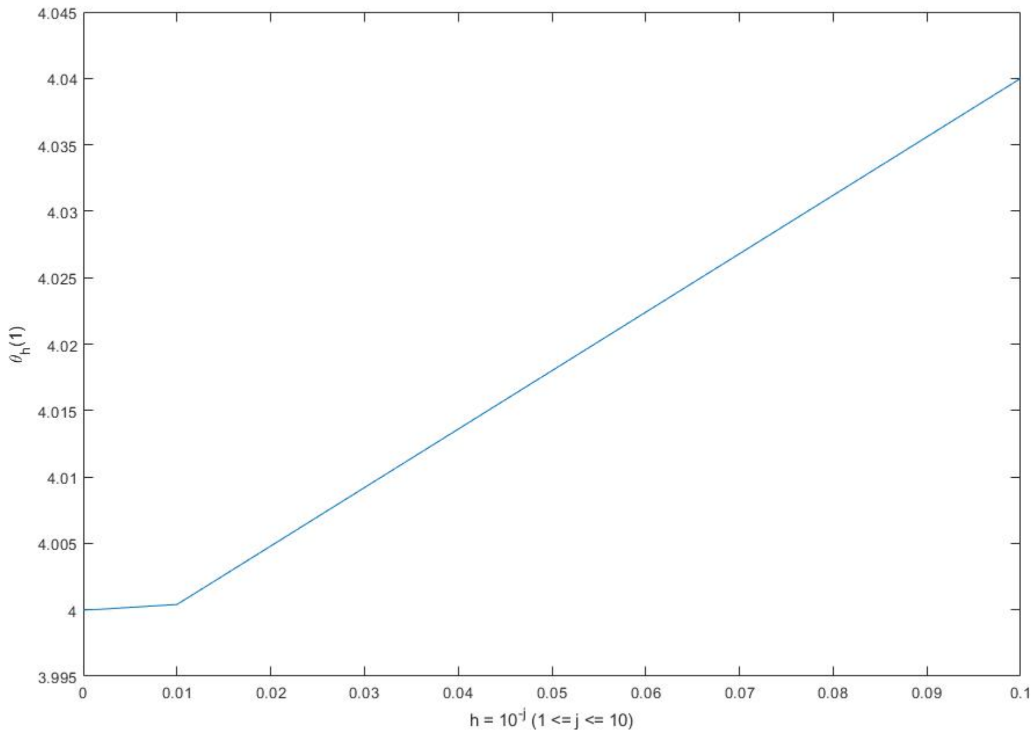
Il seguente codice MatLab, riguarda la funzione $\theta_h(x) = \frac{f(x+h)-f(x-h)}{2h}$, indicando con $h = 10^{-}j$, $j = 1, ..., 10$, $f(x) = x^4$ e $x = 1$:

```
% Soluzione Cap_1 Es_3

format long e;

% -h: vettore contenente i valori 10^-j.
% -f: vettore contenente i valori della funzione teta.

h = zeros(10,1);
f = zeros(10,1);

% Iterazione per il calcolo dei valori della funzione.

for j = 1:10
    h(j) = power(10,-j);
    f(j) = teta(1,h(j));
end

plot(h,f);

% val = teta(x,h)
%   Funzione che calcola il valore di teta.
%
% Input:
%   -x;
%   -h.
%
% Ouput:
%   -val.

function val = teta(x,h)
    sam = x+h;
    dif = x-h;
    val = (power(sam,4) - power(dif,4))/(2*h);
end
```

restituisce i seguenti valori:

h	$\theta_h(1)$
10^{-1}	$4.040000000000002e+00$
10^{-2}	$4.000400000000004e+00$
10^{-3}	$4.000003999999723e+00$
10^{-4}	$4.000000039999230e+00$
10^{-5}	$4.000000000403681e+00$
10^{-6}	$3.999999999948489e+00$
10^{-7}	$4.000000000115023e+00$
10^{-8}	$4.000000003445692e+00$
10^{-9}	$4.000000108916879e+00$
10^{-10}	$4.000000330961484e+00$

Si vede che i valori di $\theta_h(1)$ diminuiscono fino ad $h = 10^{-6}$, in cui si ha il minimo valore di $\theta_h(1)$, dopodichè inizia a crescere. Mostriamo l'andamento relativo nel seguente plot:



Andamento della funzione $\theta_h(1)$

1.4 Esercizio 4

Le due espressioni in aritmetica finita vengono scritte tenendo conto dell'errore di approssimazione sul valore reale:

1. $(x \oplus y) \oplus z \equiv fl(fl(fl(x) + fl(y)) + fl(z)) = ((x(1+\varepsilon_x) + y(1+\varepsilon_y))(1+\varepsilon_a) + z(1+\varepsilon_z))(1+\varepsilon_b)$
2. $x \oplus (y \oplus z) \equiv fl(fl(x) + fl(fl(y) + fl(z))) = (x(1+\varepsilon_x) + (y(1+\varepsilon_y) + z(1+\varepsilon_z))(1+\varepsilon_a))(1+\varepsilon_b)$

Indichiamo con $\varepsilon_x, \varepsilon_y, \varepsilon_z$ i relativi errori di x, y, z e con $\varepsilon_a, \varepsilon_b$ gli errori delle somme e per calcolare l'errore relativo delle due espressioni consideriamo $\varepsilon_m = max\{|\varepsilon_x|, |\varepsilon_y|, |\varepsilon_z|, |\varepsilon_a|, |\varepsilon_b|\}$.
Dalla definizione di errore relativo si ha quindi:

1.
$$\begin{aligned}
\varepsilon_1 &= \frac{((x(1+\varepsilon_x) + y(1+\varepsilon_y))(1+\varepsilon_a) + z(1+\varepsilon_z))(1+\varepsilon_b) - (x+y+z)}{x+y+z} \approx \\
&\approx \frac{x(1+\varepsilon_x+\varepsilon_a+\varepsilon_b) + y(1+\varepsilon_y+\varepsilon_a+\varepsilon_b) + z(1+\varepsilon_z+\varepsilon_b) - x - y - z}{x+y+z} \leq \\
&\leq \left| \frac{\varepsilon_b(x+y+z) + \varepsilon_a(x+y) + x\varepsilon_x + y\varepsilon_y + z\varepsilon_z}{(x+y+z)} \right| \leq \\
&\leq \frac{|\varepsilon_b||x+y+z| + |\varepsilon_a||x+y| + |x||\varepsilon_x| + |y||\varepsilon_y| + |z||\varepsilon_z|}{|x+y+z|} \leq \\
&\leq \varepsilon_m \frac{|x+y+z| + |x+y| + (|x|+|y|+|z|)}{|x+y+z|} = \\
&= \varepsilon_m \left(1 + \frac{|x+y|}{|x+y+z|} + \frac{|x|+|y|+|z|}{|x+y+z|}\right)
\end{aligned}$$

2. Tenendo presente che $x \oplus (y \oplus z) = (y \oplus z) \oplus x$, seguendo gli stessi procedimenti del punto precedente possiamo scrivere:
$$\begin{aligned}
\varepsilon_2 &= \frac{((y(1+\varepsilon_y) + z(1+\varepsilon_z))(1+\varepsilon_a) + x(1+\varepsilon_x))(1+\varepsilon_b) - (y+z+x)}{y+z+x} \approx \\
&\approx \frac{y(1+\varepsilon_y+\varepsilon_a+\varepsilon_b) + z(1+\varepsilon_z+\varepsilon_a+\varepsilon_b) + x(1+\varepsilon_x+\varepsilon_b) - y - z - x}{y+z+x} \leq \\
&\leq \left| \frac{\varepsilon_b(y+z+x) + \varepsilon_a(y+z) + y\varepsilon_y + z\varepsilon_z + x\varepsilon_x}{(y+z+x)} \right| \leq \\
&\leq \frac{|\varepsilon_b||y+z+x| + |\varepsilon_a||y+z| + |y||\varepsilon_y| + |z||\varepsilon_z| + |x||\varepsilon_x|}{|y+z+x|} \leq \\
&\leq \varepsilon_m \frac{|y+z+x| + |y+z| + (|y|+|z|+|x|)}{|y+z+x|} = \\
&= \varepsilon_m \left(1 + \frac{|y+z|}{|y+z+x|} + \frac{|y|+|z|+|x|}{|y+z+x|}\right)
\end{aligned}$$

Otteniamo quindi che i valori degli errori ε_1 e ε_2 sono condizionati rispettivamente, dai valori $\frac{|x+y|}{|x+y+z|}$ e $\frac{|y+z|}{|y+z+x|}$.

1.5 Esercizio 5

Il seguente codice MatLab:

```
% [x,count] = Es_5(delta)
% Funzione dichiarata nell'Es_5.
%
% Input:
%  —delta.
%
% Output:
%  —x;
%  —count.

function [x,count] = Es_5(delta)
    x = 0;
    count = 0;
    while x ~= 1
        x = x + delta;
        count = count + 1;
    end
end
```

restituisce i seguenti valori:

1. $delta = 1/16$
   Il valore di $delta = [0.0625]_{10}$ in binario si scrive $delta = [0,0001]_2$. Al passo 16, che sarà il valore di $count$ la rappresentazione di x sarà uguale a 1, e siccome l'unica condizione di uscita dello while è $x \; = 1$, il ciclo si arresterà.

2. $delta = 1/20$
   Il valore di $delta = [0,05]_{10}$ in binario si scrive $delta = [0,00\overline{0011}]_2$. A differenza del caso precedente, si può notare che la rappresentazione del valore di delta in binario è periodica. Al passo 10 la rappresentazione di x sarà diversa da 1, poichè la somma riguarda numeri periodici, e siccome l'unica condizione di uscita dello while è $x \; = 1$, il ciclo non si arresterà mai.
   Possiamo provarlo effettuando la somma in binario di:

$$\left[\frac{1}{20}\right]_{10} = \left[0,00\overline{0011}\right]_2$$

$$\left[0,00\overline{0011}\right]_2 + \left[0,00\overline{0011}\right]_2 + \underbrace{...}_{6volte} + \left[0,00\overline{0011}\right]_2 + \left[0,00\overline{0011}\right]_2 =$$

$$= [0.100011]_2 \approx [0.546875]_{10} \neq [1.00000]_{10}$$

che spiegherebbe il motivo del loop dello while.

1.6 Esercizio 6

1. Il seguente codice MatLab, riguarda la prima successione $x_{k+1} = (x_k + 3/x_k)/2$, indicando con $x = x_k$, $r = \epsilon$ e $conv = \sqrt{3} \approx 1.73205080756888e+000$:

```
% Soluzione Cap_1 Es_6 Prima successione.
%
% -conv: valore di convergenza;
% -x: vettore contenente il punto inziale e successivi punti della
% successione;
% -r: vettore contente gli errori assoluti di convergenza.

conv = sqrt(3);
x = [3];
r = [x(1)-conv];

% Iterazione della successione


for i = 1:5
    x(i+1) = (x(i)+(3/x(i)))/2;
    r(i+1) = x(i+1)-conv;
end
```

restituisce i seguenti valori:

k	x_k	ϵ_k
$k = 0$	$x_0 = 3.00000000000000$	$\epsilon_0 = 1.267949192431123e+00$
$k = 1$	$x_1 = 2.00000000000000$	$\epsilon_1 = 2.679491924311228e-01$
$k = 2$	$x_2 = 1.75000000000000$	$\epsilon_2 = 1.794919243112281e-02$
$k = 3$	$x_3 = 1.732142857142857$	$\epsilon_3 = 9.204957398001312e-05$
$k = 4$	$x_4 = 1.732050810014727$	$\epsilon_4 = 2.445850189047860e-09$
$k = 5$	$x_5 = 1.732050807568877$	$\epsilon_5 = 0.00000000000000e+000$

I calcoli indicano che per valori di k superiori a 5, l'errore assoluto indicato con ϵ, equivale a 0, cioè $\leq 10^{-12}$.

2. Il seguente codice MatLab, riguarda la seconda successione $x_{k+1} = (3 + x_{k-1}x_k)/(x_{k-1}x_k)$, indicando con $x = x_k$, $r = \epsilon$ e $conv = \sqrt{3} \approx 1.73205080756888e+000$:

```
% Soluzione Cap_1 Es_6 Seconda successione.
%
% -x: vettore contenente il punto inziale e successivi punti della
% successione;
% -r: vettore contente gli errori assoluti di convergenza.

conv = sqrt(3);
x = [3,2];
r = [x(1)-conv,x(2)-conv];

% Iterazione della successione

for i = 2:7
    x(i+1) = (3+(x(i-1)*x(i)))/(x(i-1)+x(i));
    r(i+1) = x(i+1)-conv;
end
```

restituisce i valori:

k	x_k	ϵ_k
$k=0$	$x_0 = 3.000000000000000$	$\epsilon_0 = 1.267949192431123e+00$
$k=1$	$x_1 = 2.000000000000000$	$\epsilon_1 = 2.679491924311228e-01$
$k=2$	$x_2 = 1.800000000000000$	$\epsilon_2 = 6.794919243112285e-02$
$k=3$	$x_3 = 1.736842105263158$	$\epsilon_3 = 4.791297694280772e-03$
$k=4$	$x_4 = 1.732142857142857$	$\epsilon_4 = 9.204957397979108e-05$
$k=5$	$x_5 = 1.732050934706042$	$\epsilon_5 = 1.271371643518648e-07$
$k=6$	$x_6 = 1.732050807572256$	$\epsilon_6 = 3.378630708539276e-12$
$k=7$	$x_7 = 1.732050807568877$	$\epsilon_7 = 2.220446049250313e-16$

I calcoli indicano che per valori di k superiori a 7 incluso, l'errore assoluto indicato con ϵ, è dell'ordine di 10^{-16}, cioè $\leq 10^{-12}$.

2 Capitolo 2

2.1 Esercizio 1

Studio analitico del polinomio $P(x) = x^3 - 4x^2 + 5x - 2$.

- **Zeri del polinomio**
  Prima di tutto si scompone il polinomio :

$$\begin{aligned} &x^3 - 4x^2 + 5x - 2 = \\ &= x^3 - 2x^2 + x - 2 - 2x^2 + 4x = \\ &= x(x^2 - 2x + 1) + 2(1 + x^2 - 2x) = \\ &= (x^2 - 2x + 1)(x - 2) = \\ &= (x - 1)^2(x - 2) \end{aligned}$$

  Quindi il polinomio si annulla $P(x) = 0$ per $(x - 1) = 0 \Rightarrow x = 1$ e $(x - 2) = 0 \Rightarrow x = 2$.

- **Molteplicità**
  I valori di x precedentemente calcolati vengono definiti come *radici* del polinomio. Si dice che a è una radice di *P(x)* con *molteplicità* n se e solo se *P(x)* è divisibile per $(x - a)^n$, ma non è divisibile per $(x - a)^{n-1}$.
  Inoltre si dice che x ha *molteplicità esatta* $n \geq 1$, se:

$$f(x) = f'(x) = ... = f^{(n-1)}(x) = 0, f^{(n)}(x) \neq 0.$$

  - $x = 1$

$$\begin{aligned} P(1) &= 1 - 4 + 5 - 2 = 0 \\ P'(1) &= 3x^2 - 8x + 5 = 3 - 8 + 5 = 0 \\ P''(1) &= 6x - 8 = 6 - 8 \neq 0 \Rightarrow \textit{ molteplicità } n = 2 \end{aligned}$$

  - $x = 2$

$$\begin{aligned} P(2) &= 8 - 16 + 10 - 2 = 0 \\ P'(2) &= 3x^2 - 8x + 5 = 12 - 16 + 5 = 1 \neq 0 \Rightarrow \textit{ molteplicità } n = 1 \end{aligned}$$

  Quindi è che con $x = 1$, la radice viene definita *multipla* in quanto il polinomio viene annullato 2 volte, con molteplicità $n = 2$; invece con $x = 2$, la radice viene definita *semplice* in quanto il polinomio viene annullato 1 volta, con molteplicità $n = 1$.

Il **metodo di bisezione** è utilizzabile per approssimarne uno delle due radice a partire dall'intervallo di confidenza *[a,b]=[0,3]* se e solo se il polinomio dato $P(x) = 0$ definito e continuo nell'intervallo di confidenza *[a,b]=[0,3]*, tale che $P(a) * P(b) < 0$, è allora possibile calcolarne un'approssimazione in *[a,b]*.

$$\begin{aligned} P(a) &= P(0) = -2 \\ P(b) &= P(3) = 4 \\ P(a) * P(b) &= -2 * 4 = -8 < 0 \end{aligned}$$

Essendo il polinomio continuo, le ipotesi sono rispettate. Infatti entrambe le radici $x \in \{1, 2\}$, appartengono all'intervallo di confidenza *[a,b]=[0,3]*.

Il seguente codice MatLab, riguarda il **Metodo di bisezione**:

```matlab
% x = bisezione(f, a, b, tolx)
%   Metodo di bisezione.
%
% Input:
%   -f: la funzione;
%   -a: estremo sinistro dell'intervallo di confidenza;
%   -b: estremo destro dell'intervallo di confidenza;
%   -tolx: la tolleranza desiderata;
%
% Output :
%   -x: radici della funzione

function x = bisezione(f, a, b, tolx)
    imax = ceil( log2(b-a) - log2(tolx) );
    fa = feval(f, a);
    fb = feval(f, b);
    ib = 0;
    while ( ib<imax )
        x = (a+b)/2;
        fx = feval(f, x);
        f1x = abs( (fb-fa)/(b-a) );
        if abs(fx)<=tolx*f1x
            break;
        elseif fa*fx<0
            b = x;
            fb = fx;
        else
            a = x;
            fa = fx;
        end
        ib = ib+1;
    end
end
```

Il seguente codice MatLab, riguarda il polinomio $P(x) = x^3 - 4x^2 + 5x - 2$, su quale viene eseguito il metodo di bisezione, con intervallo di confidenza *[a,b]=[0,3]* e valore di $tol_x = 10^{-1}$ che decresce ad ogni passaggio:

```matlab
% Soluzione Cap_2 Es_1.
%
% -p: polinomio;
% -tolx: tolleranza;
% -tx: vettore contenente i valori di tolleranza ad ogni passo;
% -xb: vettore contenente i valori del metodo di bisezione ad ogni passo.

p = @(x) x^3-4*x^2+5*x-2;
tolx = 10^-1;
tx = [];
xb = [];
j = 1;

% Iterazione fino a una tolleranza di circa 10^(-16) (eps)

while tolx>eps
    tx(j) = tolx;
    xb(j) = bisezione(p, 0, 3, tolx);
    tolx = tolx/10;
```

```
    j = j+1;
end
```

restituisce i seguenti valori:

tol_x	*Bisezione*	*Num. Iterazioni*
10^{-1}	$\tilde{x} = 1.500000000000000$	$ib = 0$
10^{-2}	$\tilde{x} = 1.992187500000000$	$ib = 6$
10^{-3}	$\tilde{x} = 2.000976562500000$	$ib = 9$
10^{-4}	$\tilde{x} = 2.000061035156250$	$ib = 13$
10^{-5}	$\tilde{x} = 1.999992370605469$	$ib = 16$
10^{-6}	$\tilde{x} = 2.000000953674316$	$ib = 19$
10^{-7}	$\tilde{x} = 2.000000059604645$	$ib = 23$
10^{-8}	$\tilde{x} = 1.999999992549419$	$ib = 26$
10^{-9}	$\tilde{x} = 2.000000000931323$	$ib = 29$
10^{-10}	$\tilde{x} = 2.000000000058208$	$ib = 33$
10^{-11}	$\tilde{x} = 1.999999999992724$	$ib = 36$
10^{-12}	$\tilde{x} = 2.000000000000910$	$ib = 39$
10^{-13}	$\tilde{x} = 2.000000000000057$	$ib = 43$
10^{-14}	$\tilde{x} = 1.999999999999993$	$ib = 46$
10^{-15}	$\tilde{x} = 2.000000000000001$	$ib = 49$

Dalla tabella si può notare che la successione generata dal metodo di bisezione, a partile dall'intevallo [0,3], tende alla radice $x = 2$.

2.2 Esercizio 2

Abbiamo visto come il polinomio $P(x) = x^3 - 4x^2 + 5x - 2$, in $P(x) = 0$ presenta due radici, una con molteplicità multipla $x = 1$ e una con molteplicità semplice $x = 2$.

Di seguito sono riportati tre codici MatLab, rispettivamente:

- **Metodo di Newton**

```
% x0 = newton(f, f1, x0, imax, tolx)
%   Metodo di Newton generico.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%
% Output :
%   -x0: radici della funzione.

function x0 = newton(f, f1, x0, imax, tolx)
    in = 0;
    control = true;
    iszero = false;
    while ( in<imax ) && control
        in = in+1;
        fx = feval(f, x0);
        f1x = feval(f1, x0);
        if f1x == 0
            iszero = true;
            in = in-1;
            break
        end
        x1 = x0 - fx/f1x;
        if abs(x1-x0)>tolx
            control = true;
        else
            control = false;
        end
        x0 = x1;
    end
    if iszero
        fprintf('Con tol = %e il metodo di Newton non converge entro la tolleranza
            ed il numero di passi richiesti\n', tolx);
    else
        fprintf('Con tol = %e il metodo di Newton converge a %f dopo %d passi\n',
            tolx, x0, in);
    end
end
```

- **Metodo delle Corde**

```
% x0 = corde(f, f1, x0, imax, tolx)
%   Metodo delle corde.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
```

```
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%
% Output :
%   -x0: radici della funzione.

function x0 = corde(f, f1, x0, imax, tolx)
    f1x0 = feval(f1, x0);
        ic = 0;
    control = true;
    iszero = false;
    while ( ic<imax ) && control
        ic = ic+1;
        fx = feval(f, x0);
        if f1x0 == 0
            iszero = true;
            ic = ic-1;
            break;
        end
        x1 = x0-fx/f1x0;
        if abs(x1-x0)>tolx
            control = true;
        else
            control = false;
        end
        x0 = x1;
    end
    if iszero
        fprintf('Con tol = %e il metodo delle Corde non converge entro la tolleranza ed il numero di passi richiesti\n', tolx);
    else
        fprintf('Con tol = %e il metodo delle Corde converge a %f dopo %d passi\n', tolx, x1, ic);
    end
end
```

- **Metodo delle Secanti**

```
% x0 = secanti(f, f1, x0, imax, tolx)
%   Metodo delle secanti.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%
% Output :
%   -x0: radici della funzione

function x0 = secanti(f, f1, x0, imax, tolx)
    is = 1;
    fx0 = feval(f, x0);
    f1x = feval(f1, x0);
    iszero = false;
    if f1x == 0
       control = false;
```

```
            iszero = true;
            is = is-1;
        else
            x1 = x0-fx0/f1x;
            if abs(x1-x0)>tolx
                control = true;
            else
                control = false;
            end
        end
        while ( is<imax ) && control
            is = is+1;
            fx1 = feval(f, x1);
            if (fx1-fx0)==0
                iszero = true;
                is = is-1;
                break
            end
            x2 = (fx1*x0-fx0*x1)/(fx1-fx0);
            if abs(x2-x1)>tolx
                control = true;
            else
                control = false;
            end
            fx0 = fx1;
            x0 = x1;
            x1 = x2;
        end
        if iszero
            fprintf('Con tol = %e il metodo delle Secanti non converge entro la tolleranza ed il numero di passi richiesti\n', tolx);
        else
            fprintf('Con tol = %e il metodo delle Secanti converge a %f dopo %d passi\n', tolx, x1, is);
        end
end
```

Il seguente codice MatLab, riguarda il polinomio $P(x) = x^3 - 4x^2 + 5x - 2$, sul quale vengono eseguiti il metodo di Newton, il metodo delle Corde e il metodo delle Secanti (con secondo termine della successione ottenuto con Newton), valore di $tol_x = 10^{-1}$ che decresce ad ogni passaggio, pd che indica la derivata del polinomio, numero di iterazioni massime 1000 e punto di partenza $x_0 = 3$:

```
% Soluzione Cap_2 Es_2.
%
% -p: polinomio;
% -pd: derivata prima del polinomio;
% -tolx: tolleranza;
% -tx: vettore contenente i valori di tolleranza ad ogni passo;
% -xn: vettore contenente i valori del metodo di newton ad ogni passo;
% -xc: vettore contenente i valori del metodo delle corde ad ogni passo;
% -xs: vettore contenente i valori del metodo delle secanti ad ogni passo.

p = @(x) x^3-4*x^2+5*x-2;
pd = @(x) 3*x^2-8*x+5;
tolx = 10^-1;
tx = [];
xn = [];
xc = [];
```

```
xs = [];
j = 1;

% Iterazione fino a una tolleranza di 10^(−17)

while tolx>10^(−17)
    tx(j) = tolx;
    xn(j) = newton(p, pd, 3, 1000, tolx);
    xc(j) = corde(p, pd, 3, 1000, tolx);
    xs(j) = secanti(p, pd, 3, 1000, tolx);
    tolx = tolx/10;
    j = j+1;
end
```

restituisce i seguenti valori:

tol_x	*Newton*		*Corde*		*Secanti*	
10^{-1}	$\tilde{x} = 2.004347826086958$	$in = 4$	$\tilde{x} = 2.27636384963989$	$ic = 3$	$\tilde{x} = 2.13750571037003$	$is = 4$
10^{-2}	$\tilde{x} = 2.000037320395596$	$in = 5$	$\tilde{x} = 2.05521103455135$	$ic = 12$	$\tilde{x} = 2.01075076951071$	$is = 6$
10^{-3}	$\tilde{x} = 2.000000002785312$	$in = 6$	$\tilde{x} = 2.00669955933657$	$ic = 27$	$\tilde{x} = 2.00098797530078$	$is = 7$
10^{-4}	$\tilde{x} = 2.000000002785312$	$in = 6$	$\tilde{x} = 2.00068271918546$	$ic = 44$	$\tilde{x} = 2.00002087492682$	$is = 8$
10^{-5}	$\tilde{x} = 2.000000000000000$	$in = 7$	$\tilde{x} = 2.00006162697522$	$ic = 62$	$\tilde{x} = 2.00000004118549$	$is = 9$
10^{-6}	$\tilde{x} = 2.000000000000000$	$in = 7$	$\tilde{x} = 2.00000636579868$	$ic = 79$	$\tilde{x} = 2.00000004118549$	$is = 9$
10^{-7}	$\tilde{x} = 2.000000000000000$	$in = 7$	$\tilde{x} = 2.00000065763381$	$ic = 96$	$\tilde{x} = 2.00000004118549$	$is = 9$
10^{-8}	$\tilde{x} = 2.000000000000000$	$in = 7$	$\tilde{x} = 2.0000000679392$	$ic = 113$	$\tilde{x} = 2.00000000000172$	$is = 10$
10^{-9}	$\tilde{x} = 2.000000000000000$	$in = 8$	$\tilde{x} = 2.00000000614137$	$ic = 131$	$\tilde{x} = 2.00000000000172$	$is = 10$
10^{-10}	$\tilde{x} = 2.000000000000000$	$in = 8$	$\tilde{x} = 2.00000000063446$	$ic = 148$	$\tilde{x} = 2.00000000000172$	$is = 10$
10^{-11}	$\tilde{x} = 2.000000000000000$	$in = 8$	$\tilde{x} = 2.00000000006554$	$ic = 165$	$\tilde{x} = 2.00000000000172$	$is = 10$
10^{-12}	$\tilde{x} = 2.000000000000000$	$in = 8$	$\tilde{x} = 2.00000000000677$	$ic = 182$	$\tilde{x} = 2.000000000000000$	$is = 11$
10^{-13}	$\tilde{x} = 2.000000000000000$	$in = 8$	$\tilde{x} = 2.0000000000007$	$ic = 199$	$\tilde{x} = 2.000000000000000$	$is = 11$
10^{-14}	$\tilde{x} = 2.000000000000000$	$in = 8$	$\tilde{x} = 2.00000000000006$	$ic = 217$	$\tilde{x} = 2.000000000000000$	$is = 11$
10^{-15}	$\tilde{x} = 2.000000000000000$	$in = 8$	$\tilde{x} = 2.00000000000001$	$ic = 233$	$\tilde{x} = 2.000000000000000$	$is = 11$
10^{-16}	$\tilde{x} = 2.000000000000000$	$in = 8$	$\tilde{x} = 2.00000000000000$	$ic = 243$	$\tilde{x} = 2.000000000000000$	$is = 11$

Si vede da questi risultati che i metodi di Newton e delle Secanti convergono molto velocemente alla soluzione, mentre il metodo delle Corde, seppur convergendo, richiede molti più passi d'iterazione. Tuttavia, osservando il tempo d'esecuzione impiegato dai tre metodi per eseguire un singolo step, si deduce che i metodi quasi-Newton (Corde e Secanti) hanno un tempo di esecuzione medio per step inferiore a quello del metodo di Newton: infatti, in media, un passo d'iterazione del metodo delle secanti dura circa $\frac{1}{2}$ rispetto a quello di Newton e quello delle corde $\frac{1}{4}$. Quindi, in questo caso, il metodo più efficiente sembra essere quello delle secanti, che combina un'alta convergenza con un basso tempo di esecuzione.

La scelta del **valore di innesco** x_0 è importante. Un metodo *converge localmente* ad α se la convergenza della successione dipende in modo critico dalla vicinanza di x_0 ad α. Il procedimento è *globalmente convergente* quando la convergenza non dipende da quanto x_0 è vicino ad α. Per i metodi a convergenza locale la scelta del punto di innesco è cruciale.
Anche se essendo tutti e tre i metodi (**Newton, Corde e Secanti**) localmente convergenti, quindi più la differenza con la radice è minore più velocemente convergono, se viene utilizzato come punto di innesco $x_0 = 5/3$, i metodi non convergono in quanto tale punto di innesco è una radice della funzione derivata, cioè $f'(\frac{5}{3}) = 0$.

2.3 Esercizio 3

Abbiamo visto come il polinomio $P(x) = x^3 - 4x^2 + 5x - 2$, in $P(x) = 0$ presenta due radici, una con molteplicità multipla $x = 1$ e una con molteplicità semplice $x = 2$.

Di seguito sono riportati tre codici MatLab, rispettivamente:

- **Metodo di Newton**

```matlab
% x0 = newton(f, f1, x0, imax, tolx)
%   Metodo di Newton generico.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%
% Output :
%   -x0: radici della funzione.

function x0 = newton(f, f1, x0, imax, tolx)
    in = 0;
    control = true;
    iszero = false;
    while ( in<imax ) && control
        in = in+1;
        fx = feval(f, x0);
        f1x = feval(f1, x0);
        if f1x == 0
            iszero = true;
            in = in-1;
            break
        end
        x1 = x0 - fx/f1x;
        if abs(x1-x0)>tolx
            control = true;
        else
            control = false;
        end
        x0 = x1;
    end
    if iszero
        fprintf('Con tol = %e il metodo di Newton non converge entro la tolleranza
            ed il numero di passi richiesti\n', tolx);
    else
        fprintf('Con tol = %e il metodo di Newton converge a %f dopo %d passi\n',
            tolx, x0, in);
    end
end
```

- **Metodo di Newton modificato**

```matlab
% x0 = newtonMod(f, f1, x0, m, imax, tolx)
%   Metodo di Newton modificato.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
```

```matlab
%   -x0: l'approssimazione iniziale;
%   -m: la molteplicita della radice;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%
% Output :
%   -x0: radici della funzione

function x0 = newtonMod(f, f1, x0, m, imax, tolx)
    inm = 0;
    control = true;
    iszero = false;
    while ( inm<imax ) && control
        inm = inm+1;
        fx = feval(f, x0);
        f1x = feval(f1, x0);
        if f1x == 0
            iszero = true;
            inm = inm-1;
            break
        end
        x1 = x0 - m*(fx/f1x);
        if abs(x1-x0)>tolx
            control = true;
        else
            control = false;
        end
        x0 = x1;
    end
    if iszero
        fprintf('Con tol = %e il metodo di Newton Modificato non converge entro la tolleranza ed il numero di passi richiesti\n', tolx);
    else
        fprintf('Con tol = %e il metodo di Newton Modificato converge a %f dopo %d passi\n', tolx, x1, inm);
    end
end
```

- **Metodo di Aitken**

```matlab
% x0 = aitken(f, f1, x0, imax, tolx)
%   Metodo di accelerazione di Aitken.
%
% Input:
%   -f: la funzione
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%
% Output :
%   -x0: radici della funzione

function x0 = aitken(f, f1, x0, imax, tolx)
    ia = 0;
        control = true;
    iszero = false;
    while ( ia<imax ) && control
        ia = ia+1;
```

```
        fx = feval(f, x0);
        f1x = feval(f1, x0);
        if f1x == 0
            iszero = true;
            ia = ia-1;
            break
        end
        x1 = x0 - fx/f1x;
        fx = feval(f, x1);
        f1x = feval(f1, x1);
        if f1x == 0
            iszero = true;
            ia = ia-1;
            break
        end
        x2 = x1 - fx/f1x;
        if (x2-2*x1+x0)==0
            iszero = false;
            ia = ia-1;
            break
        end
        x3 = (x2*x0-x1^2)/(x2-2*x1+x0);
        if abs(x3-x0)>tolx
            control = true;
        else
            control = false;
        end
        x0 = x3;
    end
    if iszero
        fprintf('Con tol = %e il metodo di Aitken non converge entro la tolleranza
            ed il numero di passi richiesti\n', tolx);
    else
        fprintf('Con tol = %e il metodo di Aitken converge a %f dopo %d passi\n',
            tolx, x3, ia);
    end
end
```

Il seguente codice MatLab, riguarda il polinomio $P(x) = x^3 - 4x^2 + 5x - 2$, sul quale vengono eseguiti il metodo di Newton, il metodo di Newton modificato (con molteplicità $m = 1$ per la radice $x = 2$ e $m = 2$ per la radice $x = 1$) e il metodo di Aitken, valore di $tol_x = 10^{-1}$ che decresce ad ogni passaggio, pd che indica la derivata del polinomio, numero di iterazioni massime 1000 e punto di partenza $x_0 = 0$:

```
% Soluzione Cap_2 Es_3.
%
% -p: polinomio;
% -pd: derivata prima del polinomio;
% -tolx: tolleranza;
% -xn: vettore contenente i valori del metodo di newton ad ogni passo;
% -xnm: vettore contenente i valori del metodo di newton modificato m=1 ad ogni passo;
% -xa: vettore contenente i valori del metodo di Aitken ad ogni passo;

p = @(x) x^3-4*x^2+5*x-2;
pd = @(x) 3*x^2-8*x+5;
tolx = 10^-1;
xn = [];
xnm = [];
xa = [];
```

```
j = 1;

% Iterazione fino a una tolleranza di circa 10^(-16) (eps)

while tolx>eps
    xn(j) = newton(p, pd, 0, 1000, tolx);
    xnm(j) = newtonMod(p, pd, 0, 2, 1000, tolx);
    xa(j) = aitken(p, pd, 0, 1000, tolx);
    tolx = tolx/10;
    j = j+1;
end
```

restituisce i seguenti valori:

tol_x	*Newton*		*NewtonMod* $m = 2$		*Aitken*	
10^{-1}	$\tilde{x} = 0.895985715144219$	$in = 4$	$\tilde{x} = 0.999884326200135$	$inm_2 = 3$	$\tilde{x} = 1.00198731468863$	$ia = 2$
10^{-2}	$\tilde{x} = 0.992894084175596$	$in = 8$	$\tilde{x} = 0.999999993312949$	$inm_2 = 4$	$\tilde{x} = 1.00000098931511$	$ia = 3$
10^{-3}	$\tilde{x} = 0.999106262111969$	$in = 11$	$\tilde{x} = 0.999999993312949$	$inm_2 = 4$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-4}	$\tilde{x} = 0.999944094597924$	$in = 15$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-5}	$\tilde{x} = 0.999993011508562$	$in = 18$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-6}	$\tilde{x} = 0.999999126508267$	$in = 21$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-7}	$\tilde{x} = 0.999999945981418$	$in = 25$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-8}	$\tilde{x} = 1.00000000137933$	$in = 29$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-9}	$\tilde{x} = 1.00000000137933$	$in = 29$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-10}	$\tilde{x} = 1.00000000137933$	$in = 29$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-11}	$\tilde{x} = 1.00000000137933$	$in = 29$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-12}	$\tilde{x} = 1.00000000137933$	$in = 29$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-13}	$\tilde{x} = 1.00000000137933$	$in = 29$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-14}	$\tilde{x} = 1.00000000137933$	$in = 29$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$
10^{-15}	$\tilde{x} = 1.00000000137933$	$in = 29$	$\tilde{x} = 0.999999993312949$	$inm_2 = 5$	$\tilde{x} = 0.999999998201194$	$ia = 4$

Si vede da questi risultati che i metodi di Newton modificato con molteplicità $m = 2$ e di Aitken convergono molto velocemente alla soluzione (arrotondando $\tilde{x}_{nm2} = 0.999999993312949 \approx 1$ e $\tilde{x}_a = 0.999999998201194 \approx 1$), mentre il metodo di Newton e di Newton modificato con $m = 1$ (tale valore di molteplicità rende identici i valori restituiti, quindi non lo abbiamo calcolato), seppur convergendo (arrotondando $\tilde{x}_n = 1.00000000137933 \approx 1$), richiedono più passi d'iterazione.

2.4 Esercizio 4

Essendo $\sqrt{\alpha}$ la radice ricercata, dobbiamo innanzitutto trovare una funzione $f(x)$ che abbia uno zero in $x = \sqrt{\alpha}$. La funzione più semplice di questo tipo è $f(x) = x - \sqrt{\alpha}$, ma ovviamente, dato che si sta tentando di approssimare $\sqrt{\alpha}$ stessa, non è verosimile utilizzare il valore esatto per il calcolo dell'approssimazione. Quindi si utilizza la funzione $f(x) = x^2 - \alpha$, che ha radici semplici in $x = \sqrt{\alpha}$ e in $x = -\sqrt{\alpha}$, ovvero $f(\pm\sqrt{\alpha}) = 0$. La derivata prima di questa funzione è $f'(x) = 2x$.

L'iterazione del metodo di Newton utilizzando questa funzione diventa :

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - \alpha}{2x_i} =$$

$$= \frac{2x_i^2 - x_i^2 + \alpha}{2x_i} = \frac{x_i^2 + \alpha}{2x_i} =$$

$$= \frac{1}{2}\Big(x_i + \frac{\alpha}{x_i}\Big), \quad i = 0, 1, 2, \ldots$$

Il seguente codice MatLab, riguarda l'implementazione del **metodo di Newton per il calcolo** $\sqrt{\alpha}$:

```
% [xn,exn] = newtonSqrtAlpha(alpha, x0, imax, tolx)
%   Metodo di Newton ottimizzato per l'approssimazione della radice
%   quadrata.
%
% Input:
%   -alpha: l'argomento della radice quadrata;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata.
%
% Output:
%   -xn: vettore radici;
%   -exn: vettore errore.

function [xn,exn] = newtonSqrtAlpha(alpha, x0, imax, tolx)
    xn = [];
    exn = [];
    i = 1;
    xn(i) = x0;
    exn(i) = x0-sqrt(alpha);
    i = i+1;
    x = (x0+alpha/x0)/2;
    xn(i) = x;
    exn(i) = x-sqrt(alpha);
    while( i<imax ) && ( abs(x-x0)>tolx )
        i = i+1;
        x0 = x;
        x = (x0+alpha/x0)/2;
        xn(i) = x;
        exn(i) = x-sqrt(alpha);
    end
end
```

Il seguente codice MatLab, riguarda la chiamata della funzione definita precedentemente, con $\alpha = x_0 = 5$, con numero di passi massimi $imax = 100$ e indice di tolleranza $tol_x = eps$:

```
% Soluzione Cap_2 Es_4
```

```
%
% Tolleranza di circa 10^(-16) (eps)

[xn,exn] = newtonSqrtAlpha(5, 5, 100, eps);
```

restituisce i seguenti valori:

i	x_i	$E_{ass} = \epsilon_i = \|x_i - \sqrt{\alpha}\| \quad \alpha = 5$
$i = 0$	$x_0 = 5$	$\|\epsilon_0\| = 2.763932022500210e + 00$
$i = 1$	$x_1 = 3$	$\|\epsilon_1\| = 7.639320225002102e - 01$
$i = 2$	$x_2 = 2.333333333333333e + 00$	$\|\epsilon_2\| = 9.726535583354368e - 02$
$i = 3$	$x_3 = 2.238095238095238e + 00$	$\|\epsilon_3\| = 2.027260595448332e - 03$
$i = 4$	$x_4 = 2.236068895643363e + 00$	$\|\epsilon_4\| = 9.181435736138610e - 07$
$i = 5$	$x_5 = 2.236067977499978e + 00$	$\|\epsilon_5\| = 1.882938249764265e - 13$
$i = 6$	$x_6 = 2.236067977499790e + 00$	$\|\epsilon_6\| = 0$
$i = 7$	$x_7 = 2.236067977499790e + 00$	$\|\epsilon_7\| = 0$

2.5 Esercizio 5

Come precedentemente visto nell'Esercizio 2.4 si utilizzerà la funzione $f(x) = x^2 - \alpha$, che ha radici semplici in $x = \sqrt{\alpha}$ e in $x = -\sqrt{\alpha}$, ovvero $f(\pm\sqrt{\alpha}) = 0$. La derivata prima di questa funzione è $f'(x) = 2x$.

L'iterazione del metodo delle Secanti utilizzando questa funzione diventa :

$$
\begin{aligned}
x_{i+1} &= \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})} = \\
&= \frac{(x_i^2 - \alpha)x_{i-1} - (x_{i-1}^2 - \alpha)x_i}{x_i^2 - \alpha - x_{i-1}^2 + \alpha} = \\
&= \frac{x_i^2 x_{i-1} - \alpha x_{i-1} - x_{i-1}^2 x_i + \alpha x_i}{x_i^2 - x_{i-1}^2} = \\
&= \frac{x_i x_{i-1}(x_i - x_{i-1}) + \alpha(x_i - x_{i-1})}{(x_i - x_{i-1})(x_i + x_{i-1})} = \\
&= \frac{(x_i - x_{i-1})(x_i x_{i-1} + \alpha)}{(x_i - x_{i-1})(x_i + x_{i-1})} = \\
&= \frac{x_i x_{i-1} + \alpha}{x_i + x_{i-1}}, \quad i = 0, 1, 2, \ldots
\end{aligned}
$$

Il seguente codice MatLab, riguarda l'implementazione del **metodo delle Secanti per il calcolo** $\sqrt{\alpha}$:

```
% [xs,exs] = secantiSqrtAlpha(alpha, x0, x1, imax, tolx)
%   Metodo delle Secanti ottimizzato per l'approssimazione della radice
%   quadrata.
%
% Input:
%   -alpha: l'argomento della radice quadrata;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata.
%
% Output:
%   -xs: vettore radici;
%   -exs: vettore errore.

function [xs, exs] = secantiSqrtAlpha(alpha, x0, x1, imax, tolx)
    x = x1;
    xs = [];
    exs = [];
    i = 1;
    xs(i) = x0;
    exs(i) = x0-sqrt(alpha);
    i = i+1;
    xs(i) = x;
    exs(i) = x-sqrt(alpha);
    while ( i<imax ) && ( abs(x-x0)>tolx )
        i = i+1;
        x1 = (x*x0 + alpha)/(x + x0);
        x0 = x;
        x = x1;
        xs(i) = x;
        exs(i) = x-sqrt(alpha);
    end
end
```

Il seguente codice MatLab, riguarda la chiamata della funzione definita precedentemente, con $\alpha = x_0 = 5$, con $x_1 = 3$, con numero di passi massimi $imax = 100$ e indice di tolleranza $tol_x = eps$:

```
% Soluzione Cap_2 Es_5
%
% Tolleranza di circa 10^(-16) (eps)

[xs, exs] = secantiSqrtAlpha(5, 5, 3, 100, eps);
```

restituisce i seguenti valori:

i	x_i	$E_{ass} = \epsilon_i = \lvert x_i - \sqrt{\alpha}\rvert \quad \alpha = 5$
$i = 0$	$x_0 = 5$	$\lvert\epsilon_0\rvert = 2.763932022500210e + 00$
$i = 1$	$x_1 = 3$	$\lvert\epsilon_1\rvert = 7.639320225002102e - 01$
$i = 2$	$x_2 = 2.500000000000000e + 00$	$\lvert\epsilon_2\rvert = 2.639320225002102e - 01$
$i = 3$	$x_3 = 2.272727272727273e + 00$	$\lvert\epsilon_3\rvert = 3.665929522748312e - 02$
$i = 4$	$x_4 = 2.238095238095238e + 00$	$\lvert\epsilon_4\rvert = 2.027260595448332e - 03$
$i = 5$	$x_5 = 2.236084452975048e + 00$	$\lvert\epsilon_5\rvert = 1.647547525829296e - 05$
$i = 6$	$x_6 = 2.236067984964863e + 00$	$\lvert\epsilon_6\rvert = 7.465073448287285e - 09$
$i = 7$	$x_7 = 2.236067977499817e + 00$	$\lvert\epsilon_7\rvert = 2.753353101070388e - 14$
$i = 8$	$x_8 = 2.236067977499790e + 00$	$\lvert\epsilon_8\rvert = 4.440892098500626e - 16$
$i = 9$	$x_9 = 2.236067977499790e + 00$	$\lvert\epsilon_9\rvert = 0$
$i = 10$	$x_{10} = 2.236067977499790e + 00$	$\lvert\epsilon_{10}\rvert = 0$

Si può notare come la *convergenza superlineare* sia leggermente più lenta rispetto alla *convergenza quadratica* del metodo di Newton visto nell'esercizio precedente (2.4).

3 Capitolo 3

3.1 Esercizio 1

Il seguente codice MatLab, contiene l'implementazione di una funzione per la risoluzione di un sistema lineare $Ax = b$ con A matrice triangolare inferiore :

- **Metodo matrice triangolare inferiore**

```
% x = triangolareInferiore(A, b)
%   Metodo per la risoluzione di una matrice tringolare inferiore a
%   diagonale unitaria, anche con elementi sulla parte superiore.
%
% Input:
%   -A: matrice triangolare inferiore;
%   -b: vettore dei termini noti.
%
% Output:
%   -x: vettore delle soluzioni del sistema.

function x = triangolareInferiore(A,b)
    x = b;
    if ~ismatrix(A)
        error('A non è una matrice');
    end
    [n,m] = size(A);
    if(n~=m)
        error('A non è una matrice quadrata');
    end
    for j=1:n
        if(A(j,j)~=1)
            error('A non ha coefficienti diagonali unitari')
        end
    end
    if(~isvector(x))
        error('b non è un vettore\n');
    end
    vectorSize = size(x);
    if(vectorSize~=n)
        error('Il vettore deve avere %i riche, invece ha %i righe', n, vectorSize);
    end
    for j=1:n
        for i = j+1:n
            x(i) = x(i)-A(i,j)*x(j);
        end
    end
end
```

Il seguente codice MatLab, contiene la chiamata della funzione precedente:

```
% Soluzione Cap_3 Es_1.
%
% Input:
%   -A: matrice a digonale unitaria;
%   -b: vettore dei termini noti;
%
% Output:
%   -x: vettore delle incognite.
```

```

A = [1 2 0;2 1 0;2 2 1];
b = [2 2 2];

x = triangolareInferiore(A,b);
```

con i seguenti parametri di input :

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 2 & 2 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}$$

restituendo il seguente vettore :

$$x = \begin{bmatrix} 2 \\ -2 \\ 2 \end{bmatrix}$$

3.2 Esercizio 2

Il seguente codice MatLab, contiene l'implementazione di una funzione per la *fattorizzazione* LDL^t di una matrice A

- **Metodo fattorizzazione** LDL^t

```
% A = fattorizzazioneLDLt(A)
%   Metodo per la fattorizzazione LDLt di una matrice.
%
% Input:
%   -A: matrice sdp da fattorizzare.
%
% Output:
%   -A: matrice riscritta L, D e Lt.

function A = fattorizzazioneLDLt(A)
    [m,n]=size(A);
    if m~=n
        error('La matrice non è quadrata!');
    end
    if A(1,1)<=0
        error('La matrice non è SDP');
    end
    A(2:n,1) = A(2:n,1)/A(1,1);
    for j = 2:n
        v = (A(j,1:j-1).') .* diag(A(1:j-1,1:j-1));
        A(j,j) = A(j,j) - A(j,1:j-1)*v;
        if A(j,j)<=0
            error('la matrice non è SDP')
        end
        A(j+1:n,j) = (A(j+1:n,j) - A(j+1:n,1:j-1)*v)/A(j,j);
    end
end
```

Il seguente codice MatLab, contiene la chiamata della funzione precedente :

```
% Soluzione Cap_3 Es_2.
%
% Input:
%   -A1: matrice prima fattorizzazione;
%   -A2: matrice seconda fattorizzazione.
%
% Output:
%   -LDLt1: matrice A1 fattorizzata LDLt;
%   -LDLt2: matrice A2 fattorizzata LDLt.

A1 = [1 -1 2 2; -1 5 -14 2;2 -14 42 2;2 2 2 65];
LDLt1 = fattorizzazioneLDLt(A1);


A2 = [1 -1 2 2; -1 6 -17 3; 2 -17 48 -16; 2 3 -16 4];
LDLt2 = fattorizzazioneLDLt(A2);
```

con i seguenti parametri di input :

1. $$A_1 = \begin{bmatrix} 1 & -1 & 2 & 2 \\ -1 & 5 & -14 & 2 \\ 2 & -14 & 42 & 2 \\ 2 & 2 & 2 & 65 \end{bmatrix}$$

2. $$A_2 = \begin{bmatrix} 1 & -1 & 2 & 2 \\ -1 & 6 & -17 & 3 \\ 2 & -17 & 48 & -16 \\ 2 & 3 & -16 & 4 \end{bmatrix}$$

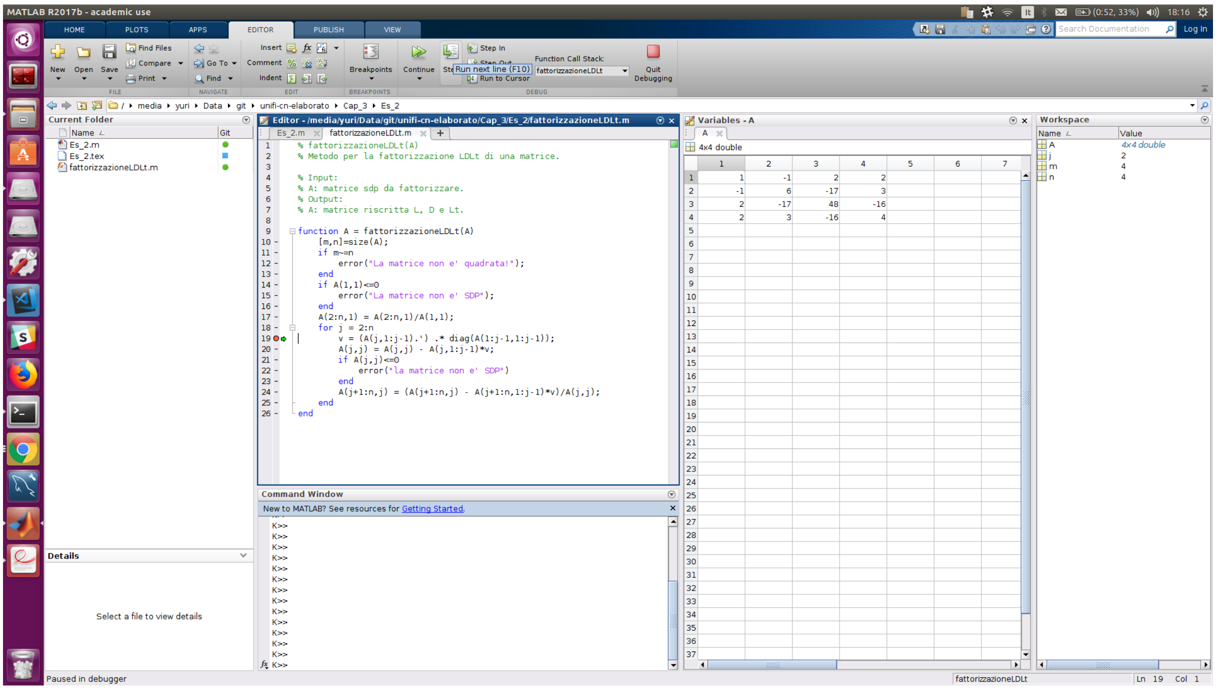
restituendo i seguenti risultati:

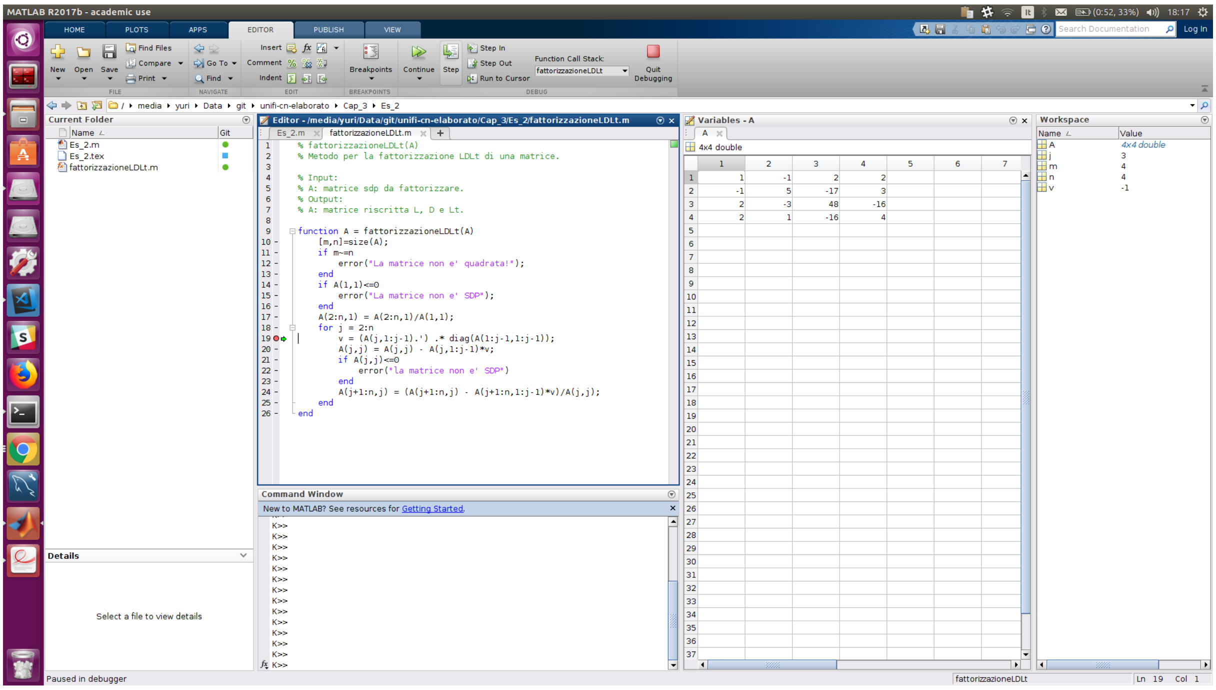
1. A_1 è fattorizzabile LDL^t

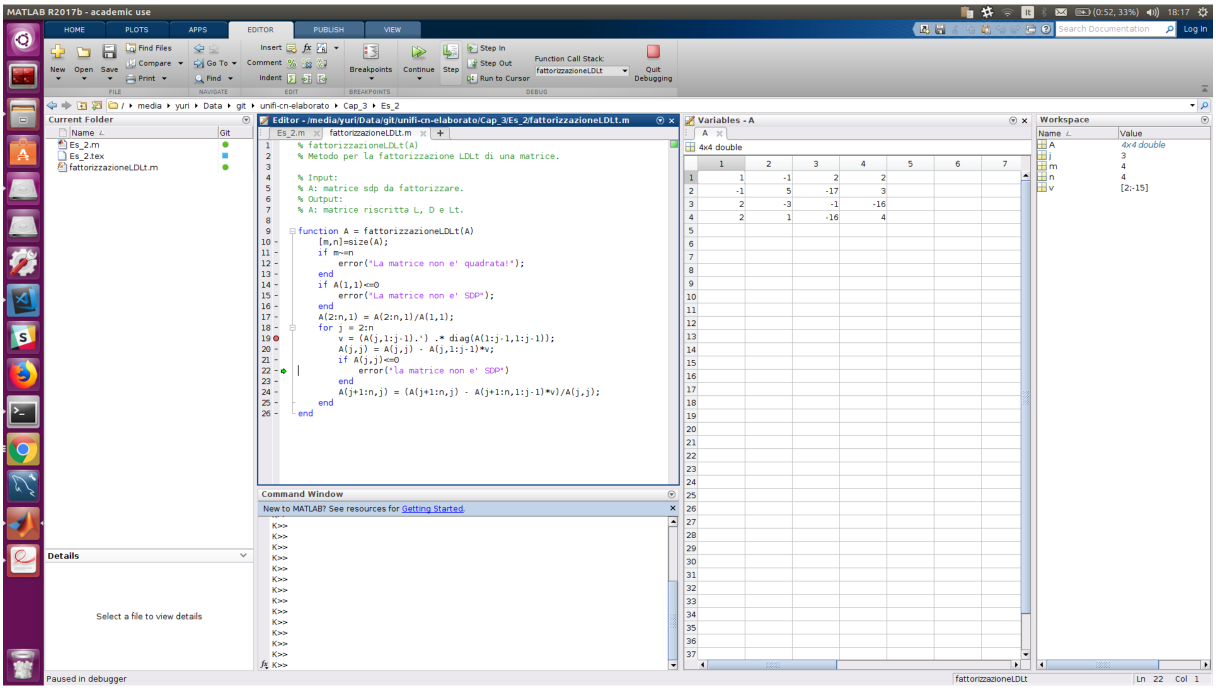
$$LDL_1^t = \begin{bmatrix} 1 & -1 & 2 & 2 \\ -1 & 4 & -14 & 2 \\ 2 & -3 & 2 & 2 \\ 2 & 1 & 5 & 7 \end{bmatrix}$$

   E' quindi *sdp*.

2. A_2 non è fattorizzabile LDL^t, quindi non è *sdp*. Dagli screenshot dell'esecuzione vediamo che al 2° ciclo il programma stabilisce che A_2 non è fattorizzabile LDL^t:







3.3 Esercizio 3

Per la risoluzione di un sistema lineare $Ax = b$ con $A = LDL^t$, viene chiamato il seguente codice in MatLab:

- **Metodo risoluzione** $LDL^t x = b$

```
% x = risolutoreLDLt(LDLt,b)
%   Metodo per la risoluzione di una matrice LDLt.
%
% Input:
%   -LDLt: matrice;
%   -b: vettore dei termini noti.
%
% Output:
%   -x: vettore delle soluzioni del sistema.

function x = risolutoreLDLt(LDLt, b)
    LDLt = fattorizzazioneLDLt(LDLt);
    x = triangolareInferiore(tril(LDLt,-1)+eye(length(LDLt)),b);
    x = diagonale(diag(LDLt),x);
    x = triangolareSuperiore((tril(LDLt,-1)+eye(length(LDLt)))',x);
end
```

il quale implementa in ordine:

1. **fattorizzazioneLDLt**(LDL^t)
   Una funzione di fattorizzazione di un matrice LDL^t passata come input e restituisce una matrice A riscritta con le informazioni di L, D e L^t (guarda es. 3.2).

2. **triangolareInferiore(L,b)**
   Una funzione per il calcolo del vettore incognite x_1 di una matrice triangolare inferiore a diagonale unitaria L (con l'utilizzo dei comandi *tril(*LDL^t*,k<0)* che restituisce gli elementi sotto la k-esima diagonale di LDL^t; *eye(n)* che restituisce una matrice di identità di grandezza n con tutti i valori $a_{i,j} = 0$ con $i \neq j$) passata come input insieme al vettore dei termini noti b del sistema (guarda es. 3.1).

3. **diagonale(D,**x_1**)**
   Una funzione per il calcolo del vettore incognite x_2 di una matrice diagonale D (con l'utilizzo del comando *diag(A)* due volte, in quando la prima mi restituisce un vettore colonna degli elementi diagonali principali di LDL^t e la seconda una matrice diagonale con tutti e solo gli elementi della diagonale $\neq 0$) passata come input insieme al vettore dei termini noti x_1:

   - **Metodo diagonale**

   ```
   % b = diagonale(D,b)
   %   Metodo per il cacolo del vettore incognite di una matrice diagonale.
   %
   % Input:
   %   -d: vettore diagonale della matrice.
   %   -b: vettore dei termini noti.
   %
   % Output:
   %   -b: vettore delle soluzioni del sistema.

   function b = diagonale(d,b)
       if(~isvector(b))
   ```

```
        error('b non è un vettore');
    end
    if(~isvector(d))
        error('d non è un vettore');
    end
    n = length(d);
    if(n~=length(b))
       error('d e b non hanno la stessa lunghezza');
    end
    for j=1:n
        b(j) = b(j)/d(j);
    end
end
```

4. **triangolareSuperiore(L^t,b)**
Una funzione per il calcolo del vettore incognite finale x del sistema lineare di una matrice triangolare superiore a diagonale unitaria L^t (con l'utilizzo dei comandi *tril(LDL^t,k<0)* che restituisce gli elementi sotto la k-esima diagonale di LDL^t; *eye(n)* che restituisce una matrice di identità di grandezza n con tutti i valori $a_{i,j} = 0$ con $i \neq j$; al tutto viene aggiunta (') per calcolarne la trasposta) passata come input insieme al vettore dei termini noti x_2:

- **Metodo matrice triangolare superiore**

```
% x = triangolareSuperiore(A, b)
%   Metodo per la risoluzione di una matrice tringolare superiore.
%
% Input:
%   -A: matrice triangolare superiore;
%   -b: vettore dei termini noti.
%
% Output:
%   -x: vettore delle soluzioni del sistema.

function x = triangolareSuperiore(A,b)
    x = b;
    if ~ismatrix(A)
        error('A non è una matrice');
    end
    [n,m] = size(A);
    if(n~=m)
        error('A non è una matrice quadratica');
    end
    if(~isvector(x))
        error('b non è un vettore');
    end
    vectorSize = size(x,1);
    if(vectorSize~=n)
        error('Il vettore deve avere %i righe, invece ha %i righe', n,
            vectorSize);
    end
    for j=n:-1:1
        x(j) = x(j)/A(j,j);
        for i = 1:j-1
            x(i) = x(i)-A(i,j)*x(j);
        end
    end
end
```

3.4 Esercizio 4

Per la risoluzione di un sistema lineare $Ax = b$ con $A = LU$, viene chiamato il seguente codice in MatLab:

- **Metodo risoluzione $LUx = b$ con pivoting**

```
% x = risolutoreLUPiv(LU,b)
%   Metodo per la risoluzione di una matrice LUPiv.
%
% Input:
%   -LU: matrice;
%   -b: vettore dei termini noti;
%
% Output:
%   -x: vettore delle soluzioni del sistema.

function x = risolutoreLUpiv(LU, b)
    [LU,p] = fattorizzazioneLUpiv(LU);
    b = b(p);
    x = triangolareInferiore(tril(LU,-1)+eye(length(LU)), b);
    x = triangolareSuperiore(triu(LU), x);
end
```

il quale implementa in ordine:

1. **fattorizzazioneLUpiv(LU,b)**
   Una funzione di fattorizzazione di un matrice LU passata come input che restituisce una matrice A riscritta con le informazioni di L e U insieme a un vettore p che indica le righe permutate :

   - **Metodo fattorizzazione LU con pivoting**

```
% [A, p] = fattorizzaLUpiv(A)
%   Metodo per la fattorizzazione LU di una matrice.
%
% Input:
%   -A: matrice sdp da fattorizzare.
%
% Output:
%   -A: matrice riscritta L e U;
%   -p: vettore contente l'informazione della matrice di permutazione P.

function [A, p] = fattorizzazioneLUpiv(A)
    [m,n]=size(A);
    if m~=n
        error('La matrice non è quadrata!');
    end
        p=[1:n];
    for i=1:(n-1)
        [aki, ki] = max(abs(A(i:n,i)));
        if aki==0
            error('La matrice è singolare!');
        end
        ki = ki+i-1;
        if ki>i
            A([i,ki],:) = A([ki,i],:);
            p([i,ki]) = p([ki,i]);
        end
        A((i+1):n,i) = A((i+1):n,i)/A(i,i);
        A((i+1):n,(i+1):n) = A((i+1):n,(i+1):n)-A((i+1):n,i)*A(i,(i+1):n);
```

```
    end
end
```

2. **triangolareInferiore(L,b)** Una funzione per il calcolo del vettore incognite x_1 di una matrice triangolare inferiore a diagonale unitaria A (con l'utilizzo dei comandi *tril(A,k<0)* che restituisce gli elementi sotto la k-esima diagonale di A; *eye(n)* che restituisce una matrice di identità (tutti i valori zeri a parte i termini diagonali) di grandezza n) passata come input insieme al vettore dei termini noti b del sistema, moltiplicato per la matrice di permutazione P calcolata $P * b$ (guarda es. 3.3).

3. **triangolareSuperiore(U,b)** Una funzione per il calcolo del vettore incognite finale x del sistema lineare di una matrice triangolare superiore a diagonale unitaria A (con l'utilizzo dei comandi *tril(A,k>0)* che restituisce gli elementi sopra la k-esima diagonale di A; *eye(n)* che restituisce una matrice di identità (tutti i valori zeri a parte i termini diagonali) di grandezza n) passata come input insieme al vettore dei termini noti $b = x_1$ (guarda es. 3.3).

3.5 Esercizio 5

Il seguente codice MatLab, contiene la chiamata delle due funzioni descritte negli esercizi precedenti, (*risolutoreLDLt* dell'es. 3.3 e risolutoreLUpiv dell'es. 3.4) per dimostrarne l'utilizzo tramite alcuni esempi:

```
% Soluzione Cap_3 Es_5.
%
% Input:
%   -LDLt: matrice da fattorizzare;
%   -xt1: vettore incognite.
%
% Ouput:
%   -b1: vettore termini noti;
%   -x1: vettore incognite;
%   -r1: vettore residuo
%   -k1: numero di condizionamento matrice;
%   -krb1: rapporto tra la norma vettore residuo e la norma vettore termini noti;
%   -kxtx1: rapporto tra la norma della differenzs tra x - ~x e la norma
%    del vettore ~x.

LDLt = [3 2 -1; 2 7 7; -1 7 30];
xt1 = [4; 5; 3];
b1 = LDLt*xt1;
x1 = risolutoreLDLt(LDLt,b1);
r1 = LDLt*x1-b1;
k1 = cond(LDLt);
krb1 = norm(r1)/norm(b1);
kxtx1 = norm(x1 - xt1)/norm(xt1);

% Input:
%   -LU: matrice da fattorizzare;
%   -xt2: vettore incognite.
%
% Ouput:
%   -b2: vettore termini noti;
%   -x2: vettore incognite;
%   -r2: vettore residuo
%   -k2: numero di condizionamento matrice;
%   -krb2: rapporto tra la norma vettore residuo e la norma vettore termini noti;
%   -kxtx2: rapporto tra la norma della differenzs tra x - ~x e la norma
%    del vettore ~x.

LU = [-23 5 -21 8;0 0 5 7; 1 54 7 9; 0 -8 12 4];
xt2 =[2; 8; 3; 5];
b2 = LU*xt2;
x2 = risolutoreLUpiv(LU, b2);
r2 = LU*x2-b2;
k2 = cond(LU);
krb2 = norm(r2)/norm(b2);
kxtx2 = norm(x2 - xt2)/norm(xt2);
```

Esempio : $risolutoreLDL^t$

con i seguenti parametri di input :

$$A_1 = LDL^t = \begin{bmatrix} 3 & 2 & -1 \\ 2 & 7 & 7 \\ -1 & 7 & 30 \end{bmatrix} \qquad \hat{x_1} = \begin{bmatrix} 4 \\ 5 \\ 3 \end{bmatrix} \qquad b_1 = LDL^t \hat{x_1} = \begin{bmatrix} 19 \\ 64 \\ 121 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0,6667 & 1 & 0 \\ -0,3333 & 1.3529 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 5.6667 & 0 \\ 0 & 0 & 19.2941 \end{bmatrix} \quad L^t = \begin{bmatrix} 1 & 0.6667 & -0.3333 \\ 0 & 1 & 1.3529 \\ 0 & 0 & 1 \end{bmatrix}$$

Il risultato ottenuto è:

$$x_1 = \begin{bmatrix} 4.00000000000000000 \\ 4.99999999999999999 \\ 3.00000000000000000 \end{bmatrix} \quad r_1 = LDL^t * x_1 - b_1 = \begin{bmatrix} 0 \\ -7.1054e-15 \\ 0 \end{bmatrix}$$

Esempio : *risolutoreLUpiv*

con i seguenti parametri di input :

$$A_2 = LU = \begin{bmatrix} -23 & 5 & -21 & 8 \\ 0 & 0 & 5 & 7 \\ 1 & 54 & 7 & 9 \\ 0 & -8 & 12 & 4 \end{bmatrix} \quad \hat{x_2} = \begin{bmatrix} 2 \\ 8 \\ 3 \\ 5 \end{bmatrix} \quad b_2 = LDL^t\hat{x_2} = \begin{bmatrix} -29 \\ 50 \\ 500 \\ -8 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -0.0435 & 1 & 0 & 0 \\ 0 & -0.1476 & 1 & 0 \\ 0 & 0 & 0.3877 & 1 \end{bmatrix} \quad U = \begin{bmatrix} -23 & 5 & -21 & 8 \\ 0 & 54.2174 & 6.0870 & 9.3478 \\ 0 & 0 & 12.8982 & 5.3793 \\ 0 & 0 & 0 & 4.92147 \end{bmatrix}$$

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad Pb = \begin{bmatrix} -29 \\ 500 \\ -8 \\ 50 \end{bmatrix}$$

Il risultato ottenuto è:

$$x_2 = \begin{bmatrix} 2.000000000000000 \\ 8.000000000000000 \\ 3.000000000000000 \\ 5.000000000000001 \end{bmatrix} \quad r_2 = LU * x_2 - b_2 = \begin{bmatrix} 7.1054e-15 \\ 7.1054e-15 \\ 0 \\ -3.5527e-15 \end{bmatrix}$$

La tabella sottostante contiene, per ogni esempio considerato, il numero di condizionamento di A, in *norma 2*, con l'utilizzo del comando *cond* e *norm* di Matlab :

A	$K_2(A)$	$\frac{\|r\|}{\|b\|}$	$\frac{\|x-\tilde{x}\|}{\|\tilde{x}\|}$
A_1	$k_1 = 20.0572$	$\frac{\|r_1\|}{\|b_1\|} = 5.1416e-17$	$\frac{\|x_1-\tilde{x_1}\|}{\|\tilde{x_1}\|} = 1.4043e-16$
A_2	$k_2 = 17.6716$	$\frac{\|r_2\|}{\|b_2\|} = 2.1173e-17$	$\frac{\|x_2-\tilde{x_2}\|}{\|\tilde{x_2}\|} = 1.0771e-16$

3.6 Esercizio 6

Il seguente codice MatLab, contiene l'implementazione di una funzione per la *fattorizzazione LU* di una matrice A:

- **Metodo fattorizzazione LU**

```
% A = fattorizzazioneLU(A)
%   Fattorizzazione LU di una matrice nonsingolare con tutti i minori
%   principali non nulli.
%
% Input:
%   -A: la matrice nonsingolare da fattorizzare LU.
%
% Output:
%   -A: la matrice riscritta con le informazioni dei fattori L ed U.

function A = fattorizzazioneLU(A)
        [m,n]=size(A);
    if m~=n
        error('La matrice non è quadrata!');
    end
    for i=1:n-1
        if A(i,i)==0
            error('La matrice non è fattorizzabile LU!');
        end
        A(i+1:n,i) = A(i+1:n,i)/A(i,i);
        A(i+1:n,i+1:n) = A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
    end
end
```

Il seguente codice Matlab, contiene la chiamata della funzione di *fattorizzazione LU* (L viene ricavata con l'utilizzo dei comandi *tril(A,k<0)* che restituisce gli elementi sotto la k-esima diagonale di A; *eye(n)* che restituisce una matrice di identità (tutti i valori zeri a parte i termini diagonali) di grandezza n), (U viene ricavata con l'utilizzo del comando *tril(A)* che restituisce la parte triangolare superiore di A) e successivamente vengono eseguiti i comandi $U\backslash(L\backslash b)$ *Gauss senza pivoting* e $A\backslash b$ *Gauss con pivoting*:

```
% Soluzione Cap_3 Es_6.
%
% Input:
%   -A: matrice da fattorizzare;
%   -e: vettore termini noti.
%
% Output:
%   -LU: fattorizzazione della matrice A;
%   -L: matrice triangolare inferiore di LU;
%   -U: matrice triangolare superiore di LU;
%   -b: vettore termini noti;
%   -Sp: Gauss senza pivoting;
%   -Cp: Gauss con pivoting.

A = [10^(-13) 1 ; 1 1];
LU = fattorizzazioneLU(A);
L = tril(LU,-1)+eye(length(LU));
U = triu(LU);

if A==L*U
    fprintf('La fattorizzazione LU è corretta');
```

```
else
    error('La fattorizzazione LU è errata');
end

e = [1; 1];
b = A*e;
Sp = U\(L\b);
Cp = A\b;
```

restituendo rispettivamente:

1. • **Fattorizzazione LU**

$$A = \begin{bmatrix} 10^{(-13)} & 1 \\ 1000 & -999 \end{bmatrix}$$

$$L * U = \begin{bmatrix} 1 & 0 \\ 10^{(13)} & 1 \end{bmatrix} * \begin{bmatrix} 10^{(-13)} & 1 \\ 0 & -10^{(13)} \end{bmatrix} = \begin{bmatrix} 10^{(-13)} & 1 \\ 1 & 1 \end{bmatrix} = LU$$

2.

$$e = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad b = Ae = \begin{bmatrix} 1.000000000000100 \\ 2 \end{bmatrix}$$

• **Gauss senza pivoting**

$$Sp = U\backslash(L\backslash b) = \begin{bmatrix} 0.999200722162641 \\ 1 \end{bmatrix}$$

• **Gauss con pivoting**

$$Cp = A\backslash b = \begin{bmatrix} 1.000000000000000 \\ 1.000000000000000 \end{bmatrix}$$

3.7 Esercizio 7

Il seguente codice MatLab, contiene l'implementazione di una funzione per la risoluzione di un sistema lineare $Ax = b$ con la seguente tipologia di matrice A :

$$A = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ \alpha & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \alpha & 1 \end{bmatrix}$$

- **Metodo matrice triangolare inferiore modificato**

```matlab
% b = triangolareInferioreMod(alpha, b)
%   Metodo per la risoluzione di una matrice bidiagonale inferiore a diagonale unitaria di Toeplitz
%
% Input:
%   -alpha: valore ripetuto nella diagonale inferiore;
%   -b: vettore dei termini noti.
%
% Output:
%   -b: vettore delle soluzioni del sistema.

function b = triangolareInferioreMod(alpha,b)
    if(~isvector(b))
        error(b non e' un vettore);
    end
    n = size(b,1);
    for i=2:n
        b(i) = b(i) - alpha*b(i-1);
    end
end
```

- **Implementazione**
  Il seguente codice MatLab contiene la chiamata della funzione precedentemente definita con i rispettivi valori di input(con $n = 12$, $A \in \mathbb{R}^{12X12}$, $b_1 \in \mathbb{R}^{12}$ e $b_2 \in \mathbb{R}^{12}$):

$$A = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 100 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 100 & 1 \end{bmatrix} \quad b_1 = \begin{bmatrix} 1 \\ 101 \\ \vdots \\ \vdots \\ 101 \end{bmatrix} \quad b_2 = 0.1 * \begin{bmatrix} 1 \\ 101 \\ \vdots \\ \vdots \\ 101 \end{bmatrix}$$

```matlab
% Soluzione Cap_3 Es_7.
%
% Input:
%   -b1: vettore termini noti.
%
% Output:
%   -x1: vettore delle incognite.
%
% Input:
%   -b2: vettore termini noti.
%
% Output:
```

```
%   −x2: vettore delle incognite.

b1 = [1; 101*ones(12,1)];
b2 = 0.1*[1; 101*ones(12,1)];
x1 = triangolareInferioreMod(100,b1);
x2 = triangolareInferioreMod(100,b2);

% Input:
%   −A: matrice.
%
% Output:
%   −k: numero di condizionamento matrice A;
%   −ninf: norma infinito matrice A;
%   −n1: norma 1 matrice A;
%   −ninv: norma infinito inversa matrice A.

A = eye(12)+100*[ zeros(1, 12); eye(11) zeros(1, 11)'];
k = cond(A);
ninf = norm(A,inf);
n1 = norm(A,1);
ninv = norm(A^−1, inf);
```

restituendo i seguenti valori:

$$x_1 = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \quad x_2 = \begin{bmatrix} 1.000000000000000e-01 \\ 1.000000000000014e-01 \\ 9.999999999985931e-02 \\ 1.000000000140702e-01 \\ 9.999999859298470e-02 \\ 1.000001407015318e-01 \\ 9.998592984681665e-02 \\ 1.014070153183368e-01 \\ -4.070153183368319e-02 \\ 1.417015318336832e+01 \\ -1.406915318336832e+03 \\ 1.407016318336832e+05 \\ -1.407015308336832e+07 \end{bmatrix}$$

- **Studio condizionamento**
  Risulta

$$\|A\|_\infty = \max_{i=1,\dots,n} \sum_{j=1}^{n} |a_{i,j}| = 101$$

$$\|A^{-1}\|_\infty = \max_{i=1,\dots,n} \sum_{j=1}^{n} |a_{i,j}| = \sum_{j=1}^{n} |a_{n,j}| = \sum_{s=0}^{n-1} 10^{2s} = \frac{10^{2n}-1}{10^2-1} = \frac{10^{2n}-1}{99}$$

$$quindi \quad k_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = 101 \frac{10^{2n}-1}{99} > 10^{2n}$$

  nel caso $n = 12$, si ha $k_\infty(A) > 10^{24}$ quindi il problema è malcondizionato. Su tale matrice, la function *cond* restiruisce *Inf*. La *norma* ∞ su una matrice è la somma delle righe, la *norma* 1 è la somma massima delle colonne; nella matrice A tutte le colonne, come tutte le righe, hanno somma 101 quindi $\|A\|_\infty = \|A\|_1 = 101$. Nella matrice A^{-1}, la *norma* ∞ considera l'n-esima riga mentre la *norma* 1 la prima colonna, in ogni caso, $\|A\|_\infty = \|A\|_1 = \frac{10^{24}-1}{99}$. Quindi $k_\infty(A) > 10^{24}$.

Considerando il vettore $\underline{b2}$ come una perturbazione di $\underline{b1}$ si ha:

$$\Delta\underline{b1} = \underline{b2} - \underline{b1} = \begin{bmatrix} -0.9 \\ -90.9 \\ \vdots \\ -90.9 \end{bmatrix}$$

segue

$$\frac{\|\Delta\underline{b1}\|}{\|\underline{b1}\|} \approx \frac{\sqrt{0.9 + 9 * 90.9^2}}{\sqrt{1 + 9 * 101^2}} \approx 1.$$

Quindi:

$$\frac{\|\Delta\underline{x}\|}{\|\underline{x}\|} \leq k(A)\left(\frac{\|\Delta\underline{b1}\|}{\|\underline{b1}\|} + \frac{\|\Delta A\|}{\|A\|}\right) = k(A)\frac{\|\Delta\underline{b1}\|}{\|\underline{b1}\|} \approx 10^{24}$$

ovvero a fronte di una perturbazione del vettore $\underline{\text{b1}}$ di 0.1, si ha un errore sul risultato dell'ordine di 10^{24}.

3.8 Esercizio 8

Il seguente codice MatLab, contiene l'implementazione di una funzione per la *fattorizzazione QR* di una matrice A:

- **Metodo fattorizzazione QR**

```
% A = fattorizzaQR(A)
%   Fattorizzazione QR di Householder per matrici mxn con m>=n.
%
% Input:
%   -A: la matrice da fattorizzare QR.
%
% Output:
%   -A: la matrice riscritta con le informazioni dei fattori Q ed R.

function A = fattorizzazioneQR(A)
    [m,n]=size(A);
    for i=1:n
        alfa = norm(A(i:m, i), 2);
        if alfa==0
            error('La matrice non ha rango massimo');
        end
        if(A(i,i))>=0
            alfa = -alfa;
        end
        v1 = A(i,i)-alfa;
        A(i,i) = alfa;
        A(i+1:m, i) = A(i+1:m, i)/v1;
        beta = -v1/alfa;
        A(i:m, i+1:n) = A(i:m, i+1:n)-(beta*[1; A(i+1:m, i)])*([1 A(i+1:m, i)']*A(i:m, i+1:n));
    end
end
```

Il seguente codice Matlab, contiene la chiamata della funzione di *fattorizzazione QR*,quindi viene ricostruita la matrice Q^t a partire dalle informazioni presenti nella matrice QR riscritta sui vettori di Householder. Viene allora moltiplicata Q^t per il vettore $b(= g)$, per risolvere infine il sistema lineare $\hat{R}x = g_1$, viene richiamato il metodo *triangolareSuperiore*($\hat{R}$,g_1) dove $\hat{R}$ viene estratto come parte triangolare superiore di QR con l'utilizzo del comando *triu(QR)* e g_1 è il vettore formato dalle prime n componenti di g:

- **Metodo risoluzione QR**

```
% x = risolutoreQR(A,b)
%   Risoluzione di un sistema lineare sovredeterminato del tipo Ax=b
%   tramite fattorizzazione QR di Householder della matrice dei
%   coefficienti.
%
% Input:
%   -A: matrice dei coefficienti mxn dove m>n;
%   -b: vettore dei termini noti.
%
% Output:
%   -b: vettore delle soluzioni del sistema lineare sovradeterminato.

function b = risolutoreQR(A, b)
    [m,n] = size(A);
    QR = fattorizzazioneQR(A);
```

```
    Qt = eye(m);
    for i=1:n
        Qt= [eye(i−1) zeros(i−1,m−i+1);zeros(i−1, m−i+1)' (eye(m−i+1) − (2/norm([1;
            QR(i+1:m, i)], 2)^2)∗([1; QR(i+1:m, i)]∗[1 QR(i+1:m, i)']))]∗Qt;
    end
    R = triu(QR(1:n, :));
    b = triangolareSuperiore(R, Qt(1:n, :)∗b);
end
```

3.9 Esercizio 9

Il seguente codice MatLab, contiene la chiamata della funzione descritta nell'esercizio precedente, (*risolutoreQR* dell'es. 3.8) e del comando $A\backslash b$, per dimostrarne l'utilizzo tramite alcuni esempi:

```
% Soluzione Cap_3 Es_9.

A1 = [3 2 1; 1 2 3; 1 2 1; 2 1 2];
b1 = [10; 10; 10; 10];
xqr1 = risolutoreQR( A1, b1 );
xab1 = A1\b1;

A2 = [9 -14 -3; 4 9 6; 33 4 12; 7 -23 4];
b2 = [12; -5; 9; -25];
xqr2 = risolutoreQR( A2, b2 );
xab2 = A2\b2;
```

Esempio

1. con input:

$$A_1 = \begin{bmatrix} 3 & 2 & 1 \\ 1 & 2 & 3 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix} \qquad b_1 = \begin{bmatrix} 10 \\ 10 \\ 10 \\ 10 \end{bmatrix}$$

   - *Fattorizzazione QR e* $A\backslash b$

$$x_{1(QR)} = \begin{bmatrix} 1.400000000000002 \\ 2.800000000000000 \\ 1.399999999999998 \end{bmatrix} \qquad x_{1(A\backslash b)} = \begin{bmatrix} 1.399999999999999 \\ 2.800000000000000 \\ 1.400000000000001 \end{bmatrix}$$

2. con input:

$$A_2 = \begin{bmatrix} 9 & -14 & -3 \\ 4 & 9 & 6 \\ 33 & 4 & 12 \\ 7 & -23 & 4 \end{bmatrix} \qquad b_2 = \begin{bmatrix} 12 \\ -5 \\ 9 \\ -25 \end{bmatrix}$$

   - *Fattorizzazione QR e* $A\backslash b$

$$x_{2(QR)} = \begin{bmatrix} 1.445540584346432 \\ 0.913813354327608 \\ -3.483370148462845 \end{bmatrix} \qquad x_{2(A\backslash b)} = \begin{bmatrix} 1.445540584346432 \\ 0.913813354327608 \\ -3.483370148462845 \end{bmatrix}$$

3.10 Esercizio 10

Per la risoluzione di sistemi nonlineari, ovvero del tipo

$$F(x) = 0 \quad F : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^n$$

con F costituita dalle *funzioni componenti*

$$F(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_n(x) \end{pmatrix} \qquad f_1 : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^n$$

ed x il vettore delle incognite che risolvono il sistema

$$x = \begin{pmatrix} x_1 \\ \cdots \\ x_n \end{pmatrix} \qquad \in \mathbb{R}^n$$

si utilizza il **metodo di Newton**, ovvero un *metodo iterativo* definito da

$$x^{k+1} = x^k - J_F(x^k)^{-1} F(x^k) \quad k = 0, 1, ...$$

partendo da un'approssimazione x^0 assegnata. $J_F(x)$ indica la **matrice Jacobiana**, ovvero la matrice delle derivate parziali:

$$J_F(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1}(x) & \cdots & \frac{\partial f_n}{\partial x_n}(x) \end{pmatrix}$$

Il seguente codice MatLab implementa la risoluzioni di sistemi nonlineari tramite l'utilizzo del **metodo di Newton**:

```
% x = newtonNonLin(f, J, x, imax, tol )
%   Metodo per la risoluzione di sistemi non lineari con il metodo di Newton.
%
% Input :
%   -F: sistema non lineare;
%   -J: Jacobiano;
%   -x: punto iniziale;
%   -imax: passi massimi;
%   -tol: tolleranza.
%
% Ouput :
%   -x: minimo relativo;
%   -nx: norma dell'ultimo incremento.

function [x,nx] = newtonNonLin(f, J, x, imax, tolx)
    i=0;
    xold=0;
    while(i<imax) && (norm(x-xold)>tolx)
        i=i+1;
        xold=x;
        val = -feval(f,x);
        b = [val(1);val(2)];
        x = x + risolutoreLUpiv(J, b);
    end
    nx = norm(x-xold);
end
```

In pratica, ogni passo dell'iterazione corrisponde a risolvere il seguente sistema lineare:

$$\begin{cases} J_F(x^k)d^k = -F(x^k) \\ x^{k+1} = x^k + d^k \end{cases}$$

dove il vettore temporaneo delle incognite d^k viene utilizzato per poter spezzare l'iterazione in due equazioni. Quindi la risoluzione del sistema nonlineare si riconduce alla risoluzione di una successione di sistemi lineari. Ovviamente, per ogni sistema lineare della successione sarà necessario fattorizzare LU la matrice Jacobiana.

3.11 Esercizio 11

Il seguente codice effettua la chiamata della funzione **newtonNonLin**, partendo dalla funzione $f = f(x_1, x_2) = x_1^2 + x_2^3 - x_1 x_2$, per risolvere il seguente sistema nonlineare con i relativi parametri di input:

$$F(x) = 0 \quad F = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 - x_2 \\ 3x_2^2 - x_1 \end{bmatrix} = f \quad \textit{con punto di innesco } x_1 = \frac{1}{2} \quad x_2 = \frac{1}{2}$$

$$J_F = \begin{bmatrix} 2 - x_2 & 2x_1 - 1 \\ 3x_2^2 - 1 & 6x_2 - x_1 \end{bmatrix}$$

$$imax = 100, \quad tolx = 10^{-t} \quad t = \{3, 6\}$$

```
% Soluzione Cap_3 Es_11.

min = [1/12;1/6];

x = [1/2;1/2];
f = @(x) x(1)^2+x(2)^3-x(1)*x(2);
F = @(x) [2*x(1)-x(2), 3*x(2)^2-x(1)];
J = [2-x(2), 2*x(1)-1; 3*x(2)^2-1, 6*x(2)-x(1)];

tolx = 10^-3;
[x1, nx1] = newtonNonLin(F, J, x, 100, tolx);
e1 = norm(min-x1);

tolx = 10^-6;
[x2,nx2] = newtonNonLin(F, J, x, 100, tolx);
e2 = norm(min-x2);
```

Qui di seguito è riportata una tabella con le seguenti informazioni (i numero di iterazioni eseguite, $\|nx_i\|$ norma euclidea dell'ultimo incremento e $\|e_1\|$ norma euclidea dell'errore con cui viene approssimato il risultato esatto):

$tol_x = 10^{-t}$	i	$\|nx_i\|$	$\|e_i\|$
10^{-3}	$i = 17$	$\|nx_1\| = 8.5430668343330485e - 04$	$\|e_1\| = 0.003794501517081$
10^{-6}	$i = 51$	$\|nx_2\| = 9.788751414570120e - 07$	$\|e_2\| = 4.4808190134094 65e - 06$

4 Capitolo 4

4.1 Esercizio 1

Il seguente codice MatLab contiene l'implementazione del calcolo del polinomio interpolante di grado n in forma di Lagrange. La forma della funzione è del seguente tipo: *y = lagrange(xi, fi, x)*

```
% y = lagrange(xi, fi, x)
%   Calcola il polinomio interpolante di grado n in forma di Lagrange, nei
%   punti x.
%
% Input:
%   -xi : vettore contenente le ascisse di interpolazione su cui calcolare
%   la differenza divisa;
%   -fi : vettore contenente i valori assunti dalla funzione in
%   corrispondenza dei punti xi.
%   -x : vettore contenente i valori su cui calcolare il polinomio
%   interpolante
%
% Output:
%   -y : vettore contenente il valore del polinomio interpolante calcolato
%   sulle x.

function [y] = lagrange(xi, fi, x)
    n = length(xi);
    m = length(x);
    y = zeros(m,1);
    for i = 1:m
        y(i) = 0;
        for j = 1:n
            range = [1:j-1, j+1:n];
            bl = prod(x(i) - xi(range))/prod(xi(j) - xi(range));
            y(i) = y(i) + fi(j) * bl;
        end
    end
end
```

4.2 Esercizio 2

Il seguente codice MatLab contiene l'implementazione del calcolo del polinomio interpolante di grado n in forma di Newton. La forma della funzione è del seguente tipo: *y = newton(xi, fi, x)*

```
% y = newton(xi, fi, x)
%   Calcola il polinomio interpolante di grado n in forma di Newton, nei
%   punti x.
%
% Input:
%   -xi : vettore contenente le ascisse di interpolazione su cui calcolare
%   la differenza divisa;
%   -fi : vettore contenente i valori assunti dalla funzione in
%   corrispondenza dei punti xi.
%   -x : vettore contenente i valori su cui calcolare il polinomio
%   interpolante
%
% Output:
%   -y : vettore contenente il valore del polinomio interpolante calcolato
%   sulle x.

function [y] = newton(xi, fi, x)
    n = length(xi)-1;
    for j = 1:n
        for i = n+1:-1:j+1
            fi(i) = (fi(i)-fi(i-1))/(xi(i)-xi(i-j));
        end
    end
    y = fi(n+1)*ones(size(x));
    for i = n:-1:1
        y = y.*(x-xi(i))+fi(i);
    end
end
```

4.3 Esercizio 3

Il seguente codice MatLab contiene l'implementazione del calcolo del polinomio interpolante di grado n in forma di Hermite. La forma della funzione è del seguente tipo: y = *newton(xi, fi, f1i, x)*

```
% y = hermite(xi, fi, f1i, x)
%   Calcola il polinomio interpolante di grado n in forma di Hermite, nei
%   punti x.
%
% Input:
%   -xi : vettore contenente le ascisse di interpolazione su cui calcolare
%   la differenza divisa;
%   -fi : vettore contenente i valori assunti dalla funzione in
%   corrispondenza dei punti xi.
%   -f1i : vettore contenente i valori assunti dalla derivata prima della
%   funzione in corrispondenza dei punti xi.
%   -x : vettore contenente i valori su cui calcolare il polinomio
%   interpolante
%
% Output:
%   -y : vettore contenente il valore del polinomio interpolante calcolato
%   sulle x.

function [y] = hermite(xi, fi, f1i, x)
    n = length(xi)-1;
    xh = zeros(2*n+2, 1);
    xh(1:2:2*n+1) = xi;
    xh(2:2:2*n+2) = xi;
    fh = zeros(2*n+2, 1);
    fh(1:2:2*n+1) = fi;
    fh(2:2:2*n+2) = f1i;
    nh = length(xh)-1;
    % Calcolo delle differenze divise
    for i = nh:-2:3
        fh(i) = (fh(i)-fh(i-2))/(xh(i)-xh(i-1));
    end
    for i = 2:nh
        for j = nh+1:-1:i+1
            fh(j) = (fh(j)-fh(j-1))/(xh(j)-xh(j-i));
        end
    end
    % Horner
    y = fh(nh+1)*ones(size(x));
    for i = nh:-1:1
        y = y.*(x-xh(i))+fh(i);
    end
    y = y.';
end
```

4.4 Esercizio 4

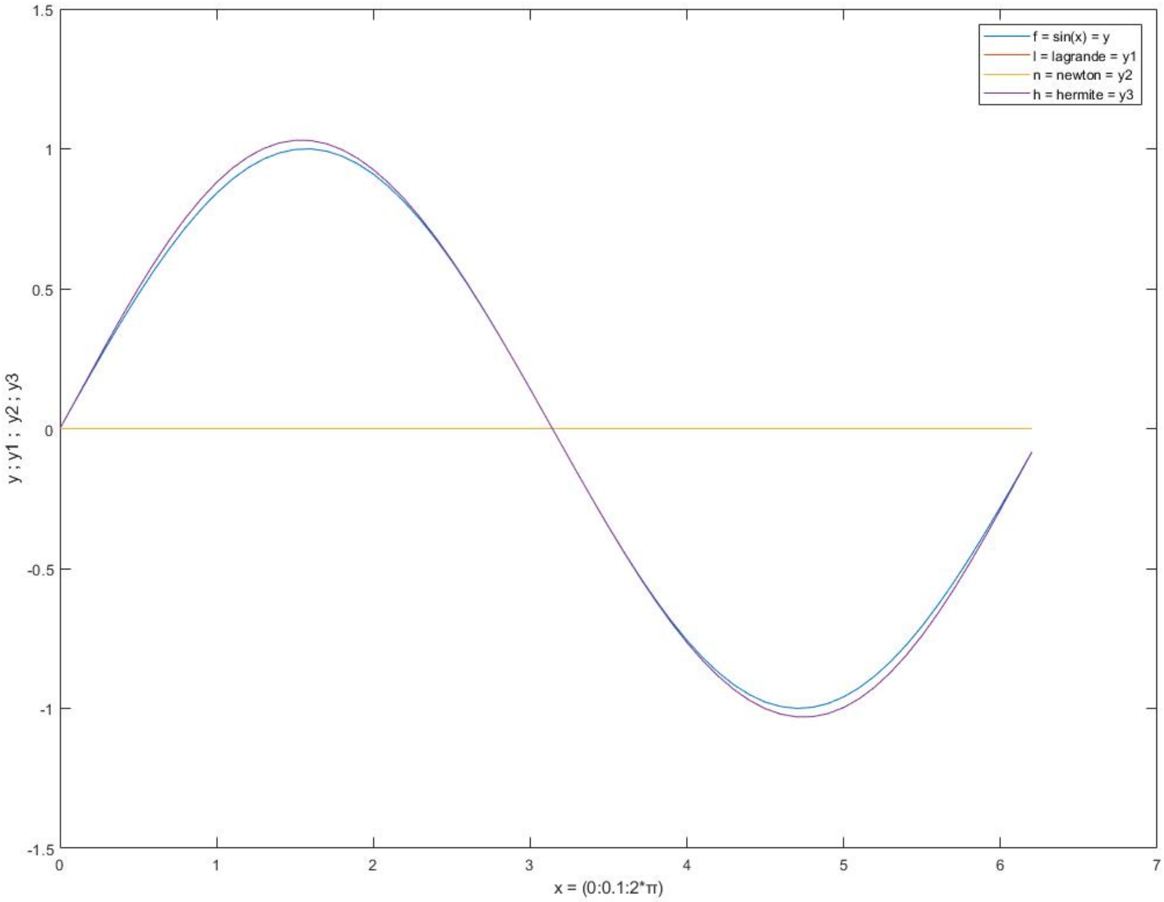
Il seguente codice MatLab contiene la chiamata rispettivamente delle funzioni implementate negli esercizi precedente (*es.1 y = lagrange(xi, fi, x)*, *es.2 y = newton(xi, fi, x)* e *es.3 y = hermite(xi, fi, f1i, x)*) con i seguenti valori di *Input* :

$$f_i = sin(x) \quad [0, 2\pi] \quad f_i' = cos(x)$$

$$x_i = i\pi \quad i = 0, 1, 2$$

```
% Soluzione Cap_4 Es_4.
%
% -xi: valori ascisse di interpolazione
% -fi: valori della funzione sin() sui punti di interpolazione xi;
% -f1i: valori derivata della funzione sull ascisse di interpolazione;
% -x: serie di punti;
% -y: valori della funzione sin() calcolati sui punti x;
% -y1: valori del polinomio di lagrange calcolati sui punti x;
% -y2: valori del polinomio di newton calcolati sui punti x;
% -y3: valori del polinomio di hermite calcolati sui punti x.

xi = zeros(3,1);
f1i = zeros(3,1);
for i = 0:length(xi)-1
    xi(i+1) = i*pi;
    f1i(i+1) = cos(xi(i+1));
end

fi = [0;0;0];
x = (0:0.1:2*pi);

y = sin(x);
y1 = lagrange(xi, fi, x);
y2 = newton(xi, fi, x);
y3 = hermite(xi, fi, f1i, x);

plot(x,y,x,y1,x,y2,x,y3);
legend('f','l','n','h')
```

Mostriamo nei seguenti plot l'approssimazione della funzione $sin(x)$ tramite l'utilizzo delle funzioni di interpolazione, precedentemente elencate:


f = sin(x) = y
l = lagrande = y1
n = newton = y2
h = hermite = y3
y ; y1 ; y2 ; y3
x = (0:0.1:2*π)

4.5 Esercizio 5

Il seguente codice MatLab contiene l'implementazione della *spline* cubica interpolante (naturale o *not-a-knot*, come specificato in ingresso) delle coppie di dati assegnate. La forma della funzione è del tipo : $y = spline3(\ xi,\ fi,\ x,\ tipo\ )$.

```
% y = spline3(xi, fi, x, tipo)
%   Determina le espressioni degli n polinomi che formano una spline
%   cubica naturale o con condizioni not-a-knot e la valuta su una seria
%   di punti.
%
% Input:
%   -xi: vettore contenente gli n+1 nodi di interpolazione;
%   -fi: vettore contenente i valori assunti dalla funzione da
%   approssimare nei nodi in xi;
%   -x: vettore di m punti su cui si vuole valutare la spline.
%   -tipo: true se la spline implementa condizioni not-a-knot, false se
%   invece e' una spline naturale.
% Output:
%   -y: vettore di m valori contenente la valutazione dei punti in x
%   della spline (NaN se un punto non e' valutabile).

function [y] = spline3(xi, fi, x, tipo)
    phi = zeros(length(xi)-2, 1);
    xxi = zeros(length(xi)-2, 1);
    dd = zeros(length(xi)-2, 1);
    for i=2:length(xi)-1
        hi = xi(i) - xi(i-1);
        hi1 = xi(i+1) - xi(i);
        phi(i) = hi/(hi+hi1);
        xxi(i) = hi1/(hi+hi1);
        dd(i) = differenzaDivisa(xxi(i-1:i+1), fi(i-1:i+1));
    end
    if tipo
        mi = risolviSistSplineNotAKnot(phi, xxi, dd);
    else
        mi = risolviSistSplineNaturale(phi, xxi, dd);
    end
    s = esprSpline3(xi, fi, mi);
    y = valutaSpline(xi, s, x);
end
```

Da come si può vedere, all'interno del codice vengono richiamate le seguenti funzioni:

- **dd = differenzaDivisa(xi, fi)**

```
% dd = differenzaDivisa(xi, fi)
%   Calcola la differenza divisa relativa ad un set di ascisse.
%
% Input:
%   -xi: vettore contenente le ascisse su cui calcolare la differenza
%   divisa;
%       -fi: vettore contenente i valori assunti dalla funzione in
%   corrispondenza dei punti in xi.
%
% Output:
%   -dd: il valore della differenza divisa risultante.

function [dd] = differenzaDivisa(xi, fi)
```

```
    dd = 0;
        for i=1:length(xi)
        prod = 1;
        for j=1:length(xi)
            if j~=i
                prod = prod*(xi(i)-xi(j));
            end
        end
        dd = dd+fi(i)/prod;
        end
end
```

- **mi = risolviSistSplineNotAKnot(phi, xi, dd)**

```
% m = risolviSistSplineNotAKnot(phi, xi, dd)
%   Risoluzione del sistema lineare di una spline cubica con condizioni
%   not-a-knot per la determinazione dei fattori m_i necessari per la
%   costruzione dell'espressione della spline cubica not-a-knot.
%
% Input:
%   -phi: vettore dei fattori phi che definiscono la matrice dei
%   coefficienti (lunghezza n-1);
%   -xi: vettore dei fattori xi che definiscono la matrice dei
%   coefficienti (lunghezza n-1);
%       -dd: vettore delle differenze divise (lunghezza n-1).
%
% Output:
%   -m: vettore riscritto con gli n+1 fattori m_i calcolati.

function [m] = risolviSistSplineNotAKnot(phi, xi, dd)
    dd = [6*dd(1); 6*dd; 6*dd(length(dd))];
    n = length(xi);
    l = zeros(n+1, 1);
        u = zeros(n+2, 1);
    w = zeros(n+1, 1);
        u(1) = 1;
    w(1) = 0;
    l(1) = phi(1)/u(1);
    u(2) = 2-phi(1);
    w(2) = xi(1)-phi(1);
    l(2) = phi(2)/u(2);
        u(3) = 2-l(2)*w(2);
    for i = 4:n
        w(i-1) = xi(i-2);
        l(i-1) = phi(i-1)/u(i-1);
        u(i) = 2-l(i-1)*w(i-1);
    end
        w(n) = xi(n-1);
    l(n) = (phi(n)-xi(n))/u(n);
    u(n+1) = 2-xi(n)-l(n-1)*w(n-1);
    w(n+1) = xi(n);
    l(n+1) = 0;
    u(n+2) = 1;

    y = zeros(n+2, 1);
    y(1) = dd(1);
    for i=2:n+2
        y(i) = dd(i)-l(i-1)*y(i-1);
```

```
    end
    m = zeros(n+2, 1);
    m(n+2) = y(n+2)/u(n+2);
    for i = n+1:-1:1
        m(i) = (y(i)-w(i)*m(i+1))/u(i);
    end
    m(1) = m(1)-m(2)-m(3);
    m(n+2) = m(n+2)-m(n+1)-m(n);
end
```

- **mi = risolviSistSplineNaturale(phi, xi, dd)**

```
% m = risolviSistSplineNaturale(phi, xi, dd)
%   Risoluzione del sistema lineare di una spline cubica con condizioni
%   naturale per la determinazione dei fattori m_i necessari per la
%   costruzione dell'espressione della spline cubica naturale.
%
% Input:
%   -phi: vettore dei fattori phi che definiscono la matrice dei
%   coefficienti (lunghezza n-1);
%   -xi: vettore dei fattori xi che definiscono la matrice dei
%   coefficienti (lunghezza n-1);
%       -dd: vettore delle differenze divise (lunghezza n-1).
%
% Output:
%   -m: vettore riscritto con gli n+1 fattori m_i calcolati.

function [m] = risolviSistSplineNaturale(phi, xi, dd)
        dd = 6*dd;
    n = length(xi)+1;
    u = zeros(n-1, 1);
    l = zeros(n-2, 1);
    u(1) = 2;
    for i = 2:n-1
        l(i) = phi(i)/u(i-1);
        u(i) = 2-l(i)*xi(i-1);
    end

    y = zeros(n-1, 1);
    y(1) = dd(1);
    for i = 2:n-1
        y(i) = dd(i)-l(i)*y(i-1);
    end
    m = zeros(n-1, 1);
    m(n-1) = y(n-1)/u(n-1);
    for i = n-2:-1:1
        m(i) = (y(i)-xi(i)*dd(i+1))/u(i);
    end
    m = [0; m; 0];
end
```

- **s = esprSpline3(xi, fi, mi)**

```
% s = esprSpline3(xi, fi, mi)
%   Calcola le espressioni degli n polinomi costituenti una spline
%   cubica.
%
```

```
% Input:
%   -xi: vettore contenente gli n+1 nodi di interpolazione;
%   -fi: vettore contenente i valori assunti dalla funzione da
%   approssimare nei nodi in xi;
%   -mi: fattori m_i calcolati risolvendo il sistema lineare
%   corrispondente.
%
% Output:
%   -s: vettore contenente le espressioni degli n polinomi che
%   definiscono la spline cubica.

function [s] = esprSpline3(xi, fi, mi)
    s = sym('x', [length(xi)-1 1]);
        syms x;
    for i = 2:length(xi)
        hi = xi(i)-xi(i-1);
        ri = fi(i-1)-((hi^2)/6)*mi(i-1);
        qi = (fi(i)-fi(i-1))/hi-(hi/6)*(mi(i)-mi(i-1));
        s(i-1) = (((x - xi(i-1))^3)*mi(i) + ((xi(i) - x)^3)*mi(i-1))/(6*hi) + qi*(x
            - xi(i-1)) + ri;
    end
end
```

- **y = valutaSpline(xi, s, x)**

```
% sx = valutaSpline(xi, s, x)
%   Valuta una spline su una serie di punti.
%
% Input:
%   -xi: vettore contenente gli n+1 nodi di interpolazione;
%   -s: vettore contenente le espressioni degli n polinomi che
%   definiscono la spline;
%   -x: vettore di m punti su cui si vuole valutare la spline.
%
% Output:
%   -sx: vettore di m valori contenente la valutazione dei punti in x
%   della spline (NaN se un punto non e' valutabile).

function [sx] = valutaSpline(xi, s, x)
    sx = zeros(length(x), 1);
    for i=1:length(x)
        if x(i) < xi(1) || x(i) > xi(length(xi))
            sx(i)=NaN;
        else
            for j=1:length(xi)
            if x(i) >= xi(j-1) && x(i) <= xi(j)
                f = inline(s(j));
                sx(i)=f(x(i));
                break;
            end
            end
        end
    end
end
```

4.6 Esercizio 6

Il seguente codice MatLab contiene l'implementazione del calcolo delle ascisse di Chebyshev per il polinomio interpolante di grado n, su un generico intervallo $[a, b]$. La forma della funzione è del seguente tipo: *xi = ceby(n, a, b)*

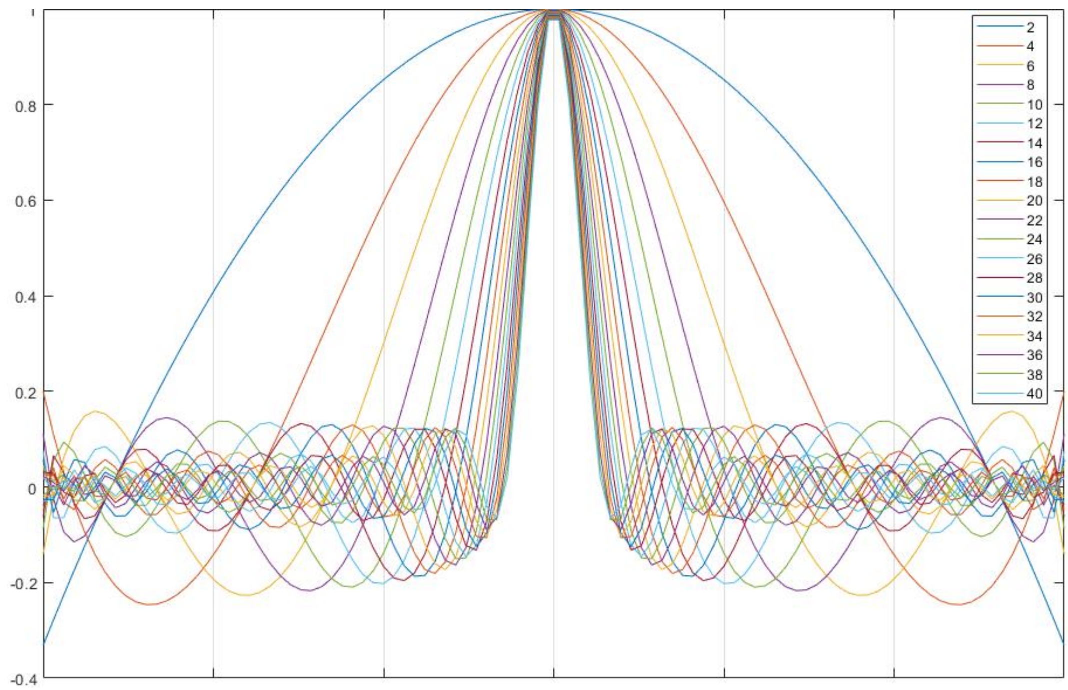
```
% xi = ceby(n, a, b)
% Calcola le ascisse di Chebyshev su un determinato intervallo.
%
% Input:
%   -a: l'estremo sinistro dell'intervallo;
%   -b: l'estremo destro dell'intervallo;
%   -n: il numero di ascisse che si vuole generare (n+1, da 0 a n).
%
% Output:
%   -xi: vettore contenente le ascisse di Chebyschev generate.

function xi = ceby(n, a, b)
    xi = zeros(n+1, 1);
    for i = 0:n
        xi(n+1-i) = (a+b)/2 + cos(pi*(2*i+1)/(2*(n+1)))*(b-a)/2;
    end
end
```

4.7 Esercizio 7

Il seguente codice MatLab contiene la soluzione del problema dell'Es.7 :

```
% Soluzione Cap_4 Es_7.

% Funzione di Runge
f = @(x) 1 ./ (1 + 25.*x.^2);
a = -6;
b = 6;
n = 2:2:40;

% valori nei quali mi interessa sapere il valore del polimonio interpolante
x = linspace(-6,6);

for i = 1:length(n)
    % Ascisse di Chebyshev
    xi = ceby(n(i),a,b);

    % Calcolo le fi nella funzione di Runge
    fi = f(xi);

    % Lagrange
    y = lagrange(xi,fi,x);

    % Plot
    plot(x, y);
    hold on

    norm(f(x) - y)
end
legend('2','4','6','8','10','12','14','16','18','20','22','24','26','28','30','32','34','36','38','40')
hold off
```

La seguente figura mostra il polinomio *p(x)*, interpolante le ascisse di Chebyshev, per la funzione di Runge *f(x)*, al variare del grado N del polinomio con $N = 2, 4, 6, 8...40$:



La seguente tabella mostra, la stima dell'errore al variare di N (grado del polinomio interpolante) tramite la norma euclidea della differenza tra i valori della funzione di *Runge* e quelle del polinomio di *Lagrange*.

N	$\|(f(x) - y\|$
2	65.1749
4	51.8929
6	44.2988
8	39.3066
10	35.6975
12	32.9190
14	30.7265
16	28.8977
18	27.3972
20	26.0781
22	24.9797
24	23.9751
26	23.1381
28	22.3469
30	21.6900
32	21.0518
34	20.5221
36	20.0008
38	19.5675
40	19.1448

4.8 Esercizio 8

Il seguente codice MatLab contiene la soluzione del problema dell'Es.8, tramite la chiamata della funzione *lebesque* il cui codice è riportato subito dopo :

```
% Soluzione Cap_4 Es_8.

% Funzione di Runge
f = @(x) 1 ./ (1 + 25.*x.^2);
a = -6;
b = 6;
n = 2:2:40;

% valori nei quali mi interessa sapere il valore del polimonio interpolante
x = linspace(-6,6);

for i = 1:length(n)
  % Ascisse di Chebyshev
  xi = ceby(n(i),a,b);
  lebesgue(xi)
end
```

```
function l = lebesgue(pts)
  rows_chebvand = length(pts);
  V_pts = gallery('chebvand',rows_chebvand,pts);
  M = max(5000,10*rows_chebvand);
  pts_leb = linspace(-1,1,M); %PUNTI TEST.
  V_leb = gallery('chebvand',rows_chebvand,pts_leb);
  l = norm(V_pts\V_leb,1);
end
```

Nella seguente tabella è riportato come varia la costante di Lebesgue λ al variare del grado N del polinomio di *Lagrange*:

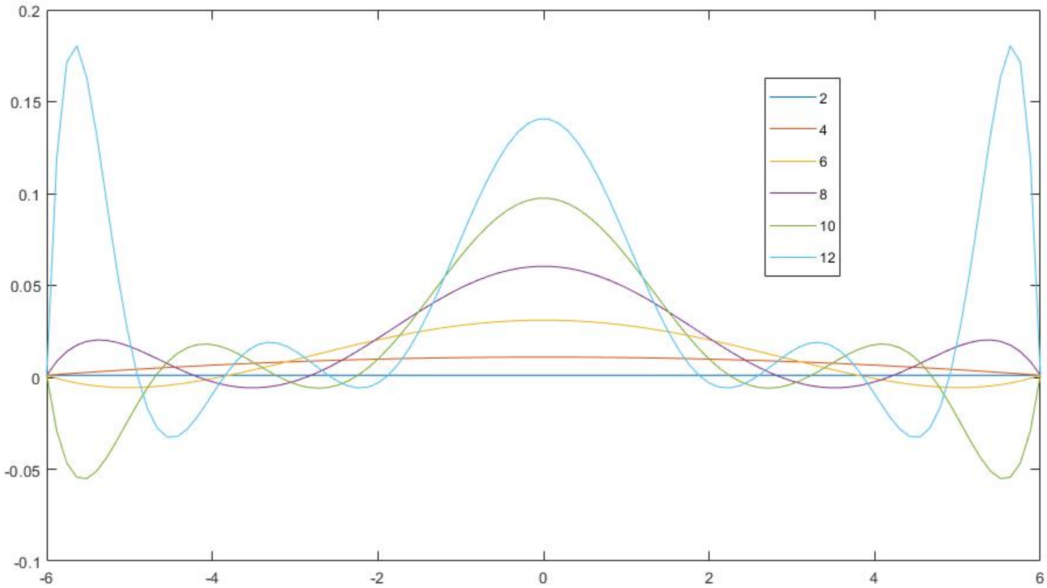
N	Λ
2	1.1554
4	1.4357
6	1.7132
8	1.9209
10	2.0493
12	2.1552
14	2.2460
16	2.3255
18	2.3962
20	2.4600
22	2.5763
24	3.3226
26	18.8308
28	20.9357
30	4.5380
32	64.2959
34	13.8721
36	7.6076
38	180.8400
40	19.5115

4.9 Esercizio 9

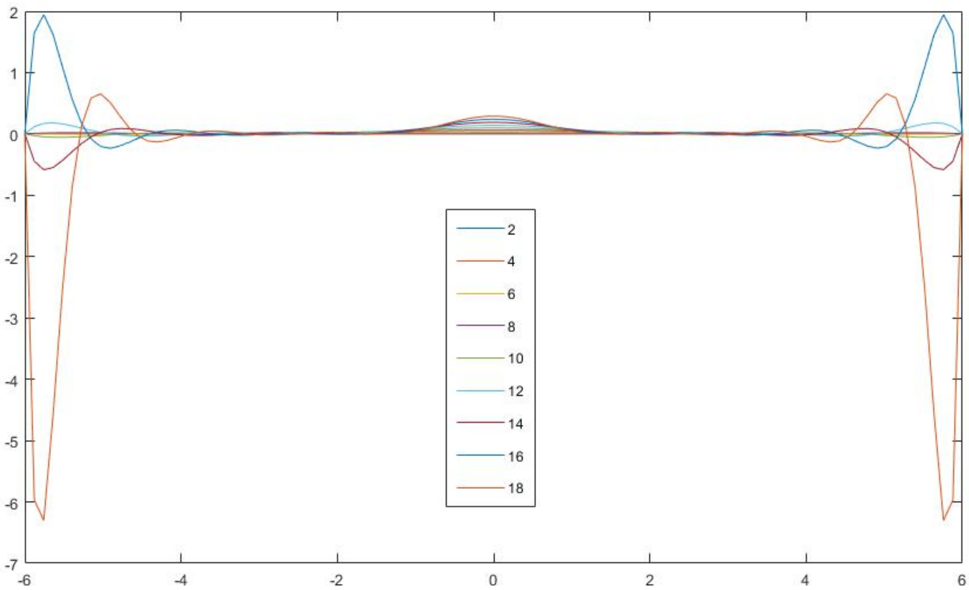
Il seguente codice MatLab contiene la soluzione del problema dell'Es.9 :

```
% Soluzione Cap_4 Es_9.

% Funzione di Runge
f = @(x) 1 ./ (1 + 25.*x.^2);
a = -6;
b = 6;
n = 2:2:40;

% valori nei quali mi interessa sapere il valore del polimonio interpolante
x = linspace(a,b);

for i = 1:length(n)
  % n+1 ascisse equidistanti in [a,b]
  xi = linspace(a,b,n(i)+1);

  % Calcolo le fi nella funzione di Runge
  fi = f(xi);

  % Lagrange
  y = lagrange(xi,fi,x);

  % Plot
  plot(x,y)
  hold on

  norm(f(x) - y)
end
legend('2','4','6','8','10','12','14','16','18','20','22','24','26','28','30','32','34','36','38','40')
hold off
```

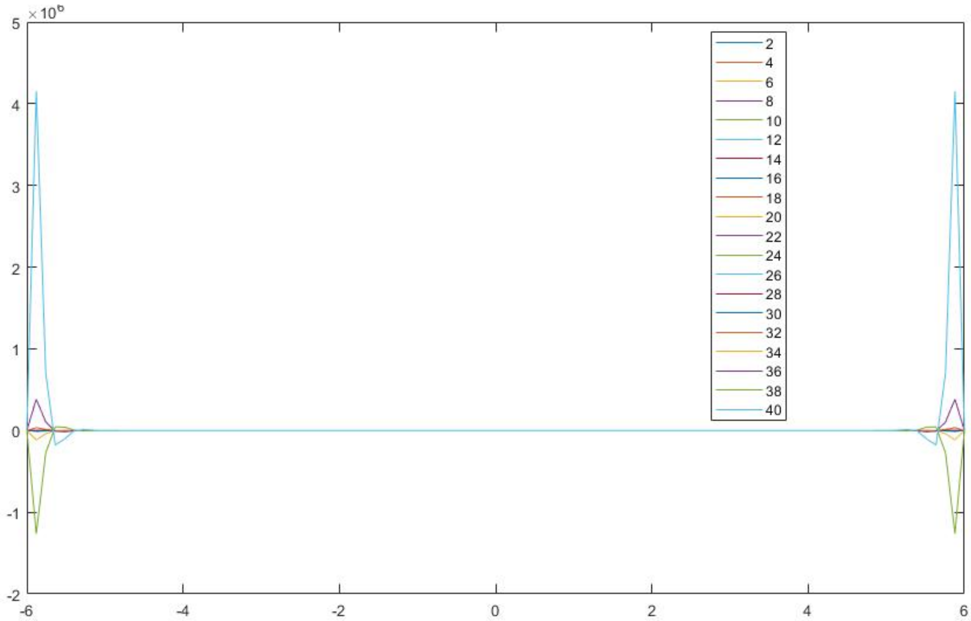
Le seguenti figura mostrano il polinomio di *Lagrange*, al variare del grado N del polinomio con $N = 2, 4, 6, 8...40$:



Polinomio di Lagrange con n° di ascisse $[2, 4, 6, 8, 10, 12]$



Polinomio di Lagrange con n° di ascisse $[2, 4, 6, 8, 10, 12, 14, 16, 18]$



Polinomio di Lagrange con n° di ascisse $[2, 4, 6, 8, .., 40]$

4.10 Esercizio 10

Il problema relativo ad un moto rettilineo uniformemente accelerato, in forma polinomiale è :

$$x(t) = x_0 + v_0 t + a_0 t^2 \quad con \quad a_0 = \frac{1}{2}a$$

il cui grado è $n = 2$.
Il problema è *ben posto*, cioè ammette soluzione ed è unica, se e solo se almeno $n + 1$ ascisse x_i delle coppie dei dati, sono tra loro distinte.
Nel nostro caso, abbiamo le seguenti coppie di dati $(tempo, spazio) = (x_i, y_i)$ per $i = 0, ..., n$:

$$(1, 2.9), (1, 3.1), (2, 6.9), (2, 7.1), (3, 12.9), (3, 13.1), (4, 20.9), (4, 21.1), (5, 30.9), (5, 31.1)$$

quindi $x_i = 5$ ascisse discinte che sono $\geq$ di $n + 1 = 2 + 1 = 3$, di conseguenza il problema risulta *ben posto*.
A questo punto possiamo stimare, nel senso dei *minimi quadrati*,posizione, velocità iniziale, ed accelerazione, che equivale alla risoluzione del sistema lineare determinato:

$$V\underline{a} = \underline{y}$$

$$V = \begin{bmatrix} x_0^0 & x_0^1 & \cdots & x_0^m \\ x_1^0 & x_1^1 & \cdots & x_1^m \\ \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^m \end{bmatrix} \qquad \underline{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix} \qquad \underline{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

in cui la matrice dei coefficienti $V \in \mathbb{R}^{n+1Xm+1}$ è una matrice di tipo *Vandermonde* (in realtà la trasposta di una matrice di tipo *Vandermonde*), il vettore $\underline{\text{a}}$, è il vettore da determinare e definisce il polinomio di approssimazione ai *minimi quadrati*, ed infine il vettore $\underline{\text{y}}$ è il vettore dei *valori misurati*.
Quindi scambiando le incognite con i valori di Input abbiamo che :

$$V = \begin{bmatrix} 1^0 & 1^1 & 1^2 \\ 1^0 & 1^1 & 1^2 \\ 2^0 & 2^1 & 2^2 \\ 2^0 & 2^1 & 2^2 \\ 3^0 & 3^1 & 3^2 \\ 3^0 & 3^1 & 3^2 \\ 4^0 & 4^1 & 4^2 \\ 4^0 & 4^1 & 4^2 \\ 5^0 & 5^1 & 5^2 \\ 5^0 & 5^1 & 5^2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 5 & 25 \end{bmatrix} \qquad \underline{a} = \begin{bmatrix} x_0 \\ v_0 \\ a_0 \end{bmatrix} \qquad \underline{y} = \begin{bmatrix} 2.9 \\ 3.1 \\ 6.9 \\ 7.1 \\ 12.9 \\ 13.1 \\ 20.9 \\ 21.1 \\ 30.9 \\ 31.1 \end{bmatrix}$$

ed il sistema lineare sovradeterminato da risolvere è :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 5 & 25 \end{bmatrix} \begin{bmatrix} x_0 \\ v_0 \\ a_0 \end{bmatrix} = \begin{bmatrix} 2.9 \\ 3.1 \\ 6.9 \\ 7.1 \\ 12.9 \\ 13.1 \\ 20.9 \\ 21.1 \\ 30.9 \\ 31.1 \end{bmatrix}$$

Tale sistema si risolve mediante fattorizzazione QR (possibile poichè tutte le ascisse sono distinti). Il seguente codice MatLab contiene la chiamata della funzione *risolutoreQR* con Input la matrice V e il vettore dei termini noti b:

```
% Soluzione Cap_4 Es_10.

A = [ ones(10,3)];
j = 2;
for i = 3:2:length(A)-1
    A(i,2) =  j;
    A(i+1,2) = j;
    A(i,3) = j^2;
    A(i+1,3) = j^2;
    j = j+1;
end
b = [2.9; 3.1; 6.9; 7.1; 12.9; 13.1; 20.9; 21.1; 30.9; 31.1];

x = risolutoreQR(A,b);
r = A*x-b;
n = norm(r)^2;
format long e
x, r, n
```

restituendo i seguenti risultati (vettore da determinare $\underline{a}$ e il vettore *residuo* $\underline{r}$ e la rispettiva *norma*):

$$\underline{a} = \begin{bmatrix} 1.000000000000001e+00 \\ 1.000000000000002e+00 \\ 9.999999999999994e-01 \end{bmatrix} \qquad \underline{r} = \begin{bmatrix} 1.000000000000023e-01 \\ -9.999999999999787e-02 \\ 1.000000000000032e-01 \\ -9.999999999999609e-02 \\ 1.000000000000014e-01 \\ -9.999999999999787e-02 \\ 1.000000000000014e-01 \\ -1.000000000000014e-01 \\ 9.999999999999787e-02 \\ -1.000000000000050e-01 \end{bmatrix} \qquad \|r\|_2^2 = 1.000000000000009e-01$$

5 Capitolo 5

5.1 Esercizio 1

Il seguente codice MatLab contiene l'implementazione della formula composita dei trapezi su *n+1* ascisse equidistanti nell'intervallo *[a,b]*, relativamente alla funzione implementata da *fun(x)*. La funzione deve essere del tipo : *If = trapcomp(n, a, b, fun)*.

```
% If = trapComp(n, a, b, fun)
%   Formula dei trapezi composita per l'approssimazione dell'integrale
%   definito di una funzione.
%
% Input:
%   -n: numero di sottointervalli sui quali applicare la formula dei
%       trapezi semplice.
%   -a: estremo sinistro dell'intervallo di integrazione;
%   -b: estremo destro dell'intervallo di integrazione;
%   -fun: la funzione di cui si vuol calcolare l'integrale;
%
% Output:
%   -If: l'approssimazione dell'integrale definito della funzione.

function [If] = trapComp(n, a, b, fun)
    h = (b-a)/n;
    If = 0;
    for i = 1:n-1
        If = If+fun(a+i*h);
    end
    If = (h/2)*(2*If + fun(a) + fun(b));
end
```

5.2 Esercizio 2

Il seguente codice MatLab contiene l'implementazione della formula composita di Simpson su *2n+1* ascisse equidistanti nell'intervallo *[a,b]*, relativamente alla funzione implementata da *fun(x)*. La funzione deve essere del tipo : *If = simpcomp(n, a, b, fun)*.

```
% If = simpComp(n, a, b, fun)
%   Formula di Simpson composita per l'approssimazione dell'integrale
%   definito di una funzione.
%
% Input:
%   -n: numero, pari, di sottointervalli sui quali applicare la formula di
%       Simpson semplice.
%   -a: estremo sinistro dell'intervallo di integrazione;
%   -b: estremo destro dell'intervallo di integrazione;
%   -fun: la funzione di cui si vuol calcolare l'integrale;
%
% Output:
%   -If: l'approssimazione dell'integrale definito della funzione.

function [If] = simpComp(n, a, b, fun)
    h = (b-a)/n;
    If = fun(a)-fun(b);
    for i=1:n/2
        If = If + 4*fun(a+(2*i-1)*h)+2*fun((a+2*i*h));
    end
    If = If*(h/3);
end
```

5.3 Esercizio 3

Il seguente codice MatLab contiene l'implementazione della formula composita dei trapezi adattiva nell'intervallo *[a,b]*, relativamente alla funzione implementata da *fun(x)*, e con tolleranza *tol*. La funzione deve essere del tipo : *If = trapad(a, b, fun, tol)*.

```
% If = trapAd(a, b, fun, tol)
%   Formula dei trapezi adattativa per l'approssimazione dell'integrale
%   definito di una funzione.
%
% Input:
%   -a: estremo sinistro dell'intervallo di integrazione;
%   -b: estremo destro dell'intervallo di integrazione;
%   -fun: la funzione di cui si vuol calcolare l'integrale;
%   -tol: la tolleranza entro la quale si richiede debba rientrare la
%   soluzione approssimata.
%
% Output:
%   -If: l'approssimazione dell'integrale definito della funzione;

function [If] = trapAd(a, b, fun, tol)
    h = (b-a)/2;
        m = (a+b)/2;
    If1 = h*(feval(fun, a) + feval(fun, b));
    If = If1/2 + h*feval(fun, m);
    err = abs(If-If1)/3;
    if err>tol
        IfSx = trapAd(a, m, fun, tol/2);
        IfDx = trapAd(m, b, fun, tol/2);
        If = IfSx+IfDx;
    end
end
```

5.4 Esercizio 4

Il seguente codice MatLab contiene l'implementazione della formula composita di Simpson adattiva nell'intervallo *[a,b]*, relativamente alla funzione implementata da *fun(x)*, e con tolleranza *tol*. La funzione deve essere del tipo : *If = simpad(a, b, fun, tol)*.

```matlab
% If = simpAd(a, b, fun, tol)
%   Formula di Simpson adattativa per l'approssimazione dell'integrale
%   definito di una funzione.
%
% Input:
%   -a: estremo sinistro dell'intervallo di integrazione;
%   -b: estremo destro dell'intervallo di integrazione;
%   -fun: la funzione di cui si vuol calcolare l'integrale;
%   -tol: la tolleranza entro la quale si richiede debba rientrare la
%   soluzione approssimata.
%
% Output:
%   -If: l'approssimazione dell'integrale definito della funzione;

function [If] = simpAd(a, b, fun, tol)
    h = (b-a)/6;
        m = (a+b)/2;
    m1 = (a+m)/2;
    m2 = (m+b)/2;
    If1 = h*(feval(fun, a) + 4*feval(fun, m) + feval(fun, b));
    If = If1/2 + h*(2*feval(fun, m1) + 2*feval(fun, m2) - feval(fun, m));
    err = abs(If-If1)/15;
    if err>tol
        IfSx = simpAd(a, m, fun, tol/2);
        IfDx = simpAd(m, b, fun, tol/2);
        If = IfSx+IfDx;
    end
end
```

5.5 Esercizio 5

Il seguente codice MatLab contiene la soluzione del problema dell'Es.5 :

```
% Soluzione Cap_5_Es_5
%
% -f: funzione;
% -a: estremo sinistro dell'intervallo;
% -b: estremo destro delli'intervallo;
% -tol: tolleranza.

f = @(x)(exp(-x * 10^(6)));
a = 0;
b = 1;
tol = 10^(-9);

ta = trapAd(a,b,f,tol);
sa = simpAd(a,b,f,tol);
```

restituendo i seguenti valori:

- **Formula dei Trapezi Adattiva**:

tol	$I = ta$	$num.iterazioni$
10^{-9}	$1.000000011252939e-06$	

- **Formula di Simpson Adattiva**:

tol	$I = sa$	$num.iterazioni$
10^{-9}	$1.000000016469981e-06$	

6 Capitolo 6