



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

April 28, 2018

# Capitoli

<b>1</b>	<b>Capitolo 1</b>	<b>1</b>
1.1	Esercizio 1 . . . . .	1
1.2	Esercizio 2 . . . . .	1
1.3	Esercizio 3 . . . . .	2
1.4	Esercizio 4 . . . . .	3
1.5	Esercizio 5 . . . . .	3
1.6	Esercizio 6 . . . . .	4
<b>2</b>	<b>Capitolo 2</b>	<b>6</b>
2.1	Esercizio 1 . . . . .	6
2.2	Esercizio 2 . . . . .	8
2.3	Esercizio 3 . . . . .	11
2.4	Esercizio 4 . . . . .	14
2.5	Esercizio 5 . . . . .	15
<b>3</b>	<b>Capitolo 3</b>	<b>18</b>
3.1	Esercizio 1 . . . . .	18
3.2	Esercizio 2 . . . . .	19
3.3	Esercizio 3 . . . . .	21
3.4	Esercizio 4 . . . . .	23
3.5	Esercizio 5 . . . . .	24
3.6	Esercizio 6 . . . . .	26
3.7	Esercizio 7 . . . . .	27
3.8	Esercizio 8 . . . . .	29
3.9	Esercizio 9 . . . . .	30
3.10	Esercizio 10 . . . . .	31
3.11	Esercizio 11 . . . . .	32
<b>4</b>	<b>Capitolo 4</b>	<b>34</b>
4.1	Esercizio 1 . . . . .	34
4.2	Esercizio 2 . . . . .	34
4.3	Esercizio 3 . . . . .	35
4.4	Esercizio 4 . . . . .	36
4.5	Esercizio 5 . . . . .	37
4.6	Esercizio 6 . . . . .	37
4.7	Esercizio 7 . . . . .	38
4.8	Esercizio 8 . . . . .	39
4.9	Esercizio 9 . . . . .	40
4.10	Esercizio 10 . . . . .	40
<b>5</b>	<b>Capitolo 5</b>	<b>43</b>
5.1	Esercizio 1 . . . . .	43
5.2	Esercizio 2 . . . . .	43
5.3	Esercizio 3 . . . . .	44
5.4	Esercizio 4 . . . . .	44
5.5	Esercizio 5 . . . . .	45
<b>6</b>	<b>Capitolo 6</b>	<b>46</b>

# 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.

Spesso, per avere un'idea di quanto è l'ordine di grandezza di  $\epsilon$  si scrive:

$$|\epsilon_x| \approx \frac{1}{2}10^{-k}$$

infatti :

$$|\epsilon_x| \approx \frac{1}{2}10^{-4.8739} = 6.6849e - 06 = |\epsilon_x|$$

## 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  $h = 10^{-j}$ ,  $j = 1, \dots, 10$ ,  $f(x) = x^4$  e  $x = 1$  :

```
1 format long e;
2
3 h = zeros(10,1);
4 f = zeros(10,1);
5
6 for j = 1:10
7     h(j) = power(10,-j);
8     f(j) = teta(1,h(j));
9 end
10
11 f
12
13 function val = teta(x,h)
14     sum = x+h;
15     dif = x-h;
16     val = (power(sum,4) - power(dif,4))/2*h;
17 end
```

restituisce i seguenti valori:

$h$	$\theta_h(1)$
$10^{-1}$	$4.040000000000003e-02$
$10^{-2}$	$4.000400000000004e-04$
$10^{-3}$	$4.00003999999724e-06$
$10^{-4}$	$4.00000039999230e-08$
$10^{-5}$	$4.00000000403680e-10$
$10^{-6}$	$3.99999999948489e-12$
$10^{-7}$	$4.000000000115022e-14$
$10^{-8}$	$4.000000003445692e-16$
$10^{-9}$	$4.000000108916879e-18$
$10^{-10}$	$4.000000330961484e-20$

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 l'errore inizia a crescere. Mostriamo l'andamento relativo nel seguente plot:



Figure 1: 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.

$$\begin{aligned} \varepsilon_1 &= \frac{((x(1 + \varepsilon_x) + y(1 + \varepsilon_y))(1 + \varepsilon_a) + z(1 + \varepsilon_z))(1 + \varepsilon_b) - (x + y + z)}{x + y + z} \approx \\ &\approx \frac{x(1 + \varepsilon_x + \varepsilon_a + \varepsilon_b) + y(1 + \varepsilon_y + \varepsilon_a + \varepsilon_b) + z(1 + \varepsilon_z + \varepsilon_b) - x - y - z}{x + y + z} \leq \\ &\leq \left| \frac{3 \cdot x \cdot \varepsilon_m + 3 \cdot y \cdot \varepsilon_m + 2 \cdot z \cdot \varepsilon_m}{x + y + z} \right| \leq \left| \frac{3 \cdot \varepsilon_m \cdot (x + y + z)}{x + y + z} \right| = 3 \cdot |\varepsilon_m| \end{aligned}$$

2. Seguendo gli stessi procedimenti del punto precedente possiamo scrivere:

$$\begin{aligned} \varepsilon_2 &= \frac{(x(1 + \varepsilon_x) + (y(1 + \varepsilon_y) + z(1 + \varepsilon_z))(1 + \varepsilon_a))(1 + \varepsilon_b) - (x + y + z)}{x + y + z} = \\ &= \dots \leq \left| \frac{2 \cdot x \cdot \varepsilon_m + 3 \cdot y \cdot \varepsilon_m + 3 \cdot z \cdot \varepsilon_m}{x + y + z} \right| \leq \left| \frac{3 \cdot \varepsilon_m \cdot (x + y + z)}{x + y + z} \right| = 3 \cdot |\varepsilon_m| \end{aligned}$$

Otteniamo quindi che i valori degli errori  $\varepsilon_1$  e  $\varepsilon_2$  sono  $\leq 3 \cdot |\varepsilon_m|$ .

## 1.5 Esercizio 5

Il seguente codice MatLab:

```
1 function [x,count] = Es_5(delta)
2     x = 0;
3     count = 0;
4     while x ~= 1
5         x = x + delta;
6         count = count + 1;
7     end
8     x, count
9 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  $\text{count}$  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  $\delta = [0,000011]_2$ . 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:

$$\begin{aligned} \left[\frac{1}{20}\right]_{10} &= [0,000011]_2 \\ [0,000011]_2 + [0,000011]_2 + \underbrace{\dots}_{6\text{ volte}} + [0,000011]_2 + [0,000011]_2 &= \\ &= [0.100011]_2 \approx [0.546875]_{10} \neq [1.00000]_{10} \end{aligned}$$

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  $\text{conv} = \sqrt{3} \approx 1.73205080756888e + 000$ :

```
1 format longEng
2
3 conv = sqrt(3)
4 x = [3];
5 r = [x(1)-conv];
6
7 for i = 1:5
8     x(i+1) = (x(i)+(3/x(i)))/2;
9     r(i+1) = x(i+1)-conv;
10 end
11
12 x, r
```

restituisce i seguenti valori:

$k$	$x_k$	$\epsilon_k$
$k = 0$	$x_0 = 3.00000000000000e + 000$	$\epsilon_0 = 1.26794919243112e + 000$
$k = 1$	$x_1 = 2.00000000000000e + 000$	$\epsilon_1 = 267.949192431123e - 003$
$k = 2$	$x_2 = 1.75000000000000e + 000$	$\epsilon_2 = 17.9491924311228e - 003$
$k = 3$	$x_3 = 1.73214285714286e + 000$	$\epsilon_3 = 92.0495739800131e - 006$
$k = 4$	$x_4 = 1.73205081001473e + 000$	$\epsilon_4 = 2.44585018904786e - 009$
$k = 5$	$x_5 = 1.73205080756888e + 000$	$\epsilon_5 = 0.00000000000000e + 000$

I calcoli indicano che per valori di  $k$  superiori a 4, l'errore assoluto indicato con  $\epsilon$ , è dell'ordine di  $10^{-9}$ , 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  $\text{conv} = \sqrt{3} \approx 1.73205080756888e + 000$ :

```
1 format longEng
2
3 conv = sqrt(3);
4 x = [3,2];
5 r = [x(1)-conv,x(2)-conv];
6
7 for i = 2:7
```

```

8      x(i+1) = (3+(x(i-1)*x(i)))/(x(i-1)+x(i));
9      r(i+1) = x(i+1)-conv;
10 end
11
12 x, r

```

restituisce i valori:

$k$	$x_k$	$\epsilon_k$
$k = 0$	$x_0 = 3.000000000000000e + 000$	$\epsilon_0 = 1.26794919243112e + 000$
$k = 1$	$x_1 = 2.000000000000000e + 000$	$\epsilon_1 = 267.949192431123e - 003$
$k = 2$	$x_2 = 1.800000000000000e + 000$	$\epsilon_2 = 67.9491924311229e - 003$
$k = 3$	$x_3 = 1.73684210526316e + 000$	$\epsilon_3 = 4.79129769428077e - 003$
$k = 4$	$x_4 = 1.73214285714286e + 000$	$\epsilon_4 = 92.0495739797911e - 006$
$k = 5$	$x_5 = 1.73205093470604e + 000$	$\epsilon_5 = 127.137164351865e - 009$
$k = 6$	$x_6 = 1.73205080757226e + 000$	$\epsilon_6 = 3.37863070853928e - 012$
$k = 7$	$x_7 = 1.73205080756888e + 000$	$\epsilon_7 = 222.044604925031e - 018$

I calcoli indicano che per valori di  $k$  superiori a 6 incluso, l'errore assoluto indicato con  $\epsilon$ , è dell'ordine di  $10^{-12}$ , 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:

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

–  $x = 1$

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

–  $x = 2$

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

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 radici 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**:



```

1 % bisezione(f, a, b, tolx)
2 % Metodo di bisezione.
3 %
4 % Input:
5 % f: la funzione;
6 % a: estremo sinistro dell'intervallo di confidenza;
7 % b: estremo destro dell'intervallo di confidenza;
8 % tolx: la tolleranza desiderata;
9 % Output :
10 % x: radici della funzione
11
12 function x = bisezione(f, a, b, tolx)
13     imax = ceil( log2(b-a) - log2(tolx) );
14     fa = feval(f, a);
15     fb = feval(f, b);
16     ib = 0;
17     while ( ib<imax )
18         x = (a+b)/2;
19         fx = feval(f, x);
20         flx = abs( (fb-fa)/(b-a) );
21         if abs(fx)<=tolx*flx
22             break
23         elseif fa*fx<0
24             b = x;
25             fb = fx;
26         else
27             a = x;
28             fa = fx;
29         end
30         ib = ib+1;
31     end
32     x
33 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:

```

1 p = inline('x^3-4*x^2+5*x-2');
2 tolx = 10^-1;
3 tx = [];
4 xb = [];
5 j = 1;
6 while tolx>eps
7     tx(j) = tolx;
8     xb(j) = bisezione(p, 0, 3, tolx);
9     tolx = tolx/10;
10    j = j+1;
11 end
12
13 xb

```

restituisce i seguenti valori:

$tol_x$	Bisezione	Num. Iterazioni
$10^{-1}$	$\tilde{x} = 1.5000$	$ib = 0$
$10^{-2}$	$\tilde{x} = 1.9922$	$ib = 6$
$10^{-3}$	$\tilde{x} = 2.0010$	$ib = 9$
$10^{-4}$	$\tilde{x} = 2.0001$	$ib = 13$
$10^{-5}$	$\tilde{x} = 2.0000$	$ib = 16$
$10^{-6}$	$\tilde{x} = 2.0000$	$ib = 19$
$10^{-7}$	$\tilde{x} = 2.0000$	$ib = 23$
$10^{-8}$	$\tilde{x} = 2.0000$	$ib = 26$
$10^{-9}$	$\tilde{x} = 2.0000$	$ib = 29$
$10^{-10}$	$\tilde{x} = 2.0000$	$ib = 33$
$10^{-11}$	$\tilde{x} = 2.0000$	$ib = 36$
$10^{-12}$	$\tilde{x} = 2.0000$	$ib = 39$
$10^{-13}$	$\tilde{x} = 2.0000$	$ib = 43$
$10^{-14}$	$\tilde{x} = 2.0000$	$ib = 46$
$10^{-15}$	$\tilde{x} = 2.0000$	$ib = 49$

Dalla tabella si può notare che la successione generata dal metodo di bisezione, a partire dall'intervallo  $[0,3]$ , 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

```

1  % newton(f, f1, x0, imax, tolX)
2  % Metodo di Newton generico.
3  %
4  % Input:
5  % f: la funzione;
6  % f1: la derivata della funzione;
7  % x0: l'approssimazione iniziale;
8  % imax: il numero massimo di iterazioni;
9  % tolX: la tolleranza desiderata;
10 % Output :
11 % x0: radici della funzione.
12
13 function x0 = newton(f, f1, x0, imax, tolX)
14     in = 0;
15     vai = true;
16     while ( in<imax ) && vai
17         in = in+1;
18         fx = feval(f, x0);
19         f1x = feval(f1, x0);
20         if f1x==0
21             vai=false;
22             in=in-1;
23             break
24         end
25         x1 = x0 - fx/f1x;
26         vai = abs(x1-x0)>tolX;
27         x0 = x1;
28     end
29     in
30 end

```

- Metodo delle Corde

```

1 % corde(f, f1, x0, imax, tolx)
2 % Metodo delle corde.
3 %
4 % Input:
5 % f: la funzione;
6 % f1: la derivata della funzione;
7 % x0: l'approssimazione iniziale;
8 % imax: il numero massimo di iterazioni;
9 % tolx: la tolleranza desiderata;
10 % Output :
11 % x0: radici della funzione.
12
13 function x0 = corde(f, f1, x0, imax, tolx)
14     flx0 = feval(f1, x0);
15     ic = 0;
16     vai = true;
17     while ( ic<imax ) && vai
18         ic = ic+1;
19         fx = feval(f, x0);
20         if flx0==0
21             vai=false;
22             ic=ic-1;
23             break
24         end
25         x1 = x0-fx/flx0;
26         vai = abs(x1-x0)>tolx;
27         x0 = x1;
28     end
29     ic
30 end

```

- Metodo delle Secanti

```

1 % secanti(f, f1, x0, imax, tolx)
2 % Metodo delle secanti.
3 %
4 % Input:
5 % f: la funzione;
6 % f1: la derivata della funzione;
7 % x0: l'approssimazione iniziale;
8 % imax: il numero massimo di iterazioni;
9 % tolx: la tolleranza desiderata;
10 % Output :
11 % x0: radici della funzione
12
13 function x0 = secanti(f, f1, x0, imax, tolx)
14     is = 1;
15     fx0 = feval(f, x0);
16     flx = feval(f1, x0);
17     if flx==0
18         vai=false;
19         is=is-1;
20     else
21         x1 = x0-fx0/flx;
22         vai = abs(x1-x0)>tolx;
23     end
24     while ( is<imax ) && vai

```

```

25     is = is+1;
26     fx1 = feval(f, x1);
27     if (fx1-fx0)==0
28         vai=false;
29         is=is-1;
30         break
31     end
32     x2 = (fx1*x0-fx0*x1)/(fx1-fx0);
33     vai = abs(x2-x1)>tolx;
34     fx0 = fx1;
35     x0 = x1;
36     x1 = x2;
37 end
38 is
39 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  $x_0 = 3$ :

```

1  p = inline('x^3-4*x^2+5*x-2');
2  pd = inline('3*x^2-8*x+5');
3  tol_x = 10^-1;
4  tx = [];
5  xn = [];
6  xc = [];
7  xs = [];
8  j = 1;
9  while tol_x>eps
10     tx(j) = tol_x;
11     xn(j) = newton(p, pd, 3, 1000, tol_x);
12     xc(j) = corde(p, pd, 3, 1000, tol_x);
13     xs(j) = secanti(p, pd, 3, 1000, tol_x);
14     tol_x = tol_x/10;
15     j = j+1;
16 end
17
18 xn, xc, xs

```

restituisce i seguenti valori:

$tol_x$	<i>Newton</i>		<i>Corde</i>		<i>Secanti</i>	
$10^{-1}$	$\tilde{x} = 2.0043$	$in = 4$	$\tilde{x} = 2.2764$	$ic = 3$	$\tilde{x} = 2.1375$	$is = 4$
$10^{-2}$	$\tilde{x} = 2.0000$	$in = 5$	$\tilde{x} = 2.0552$	$ic = 12$	$\tilde{x} = 2.1375$	$is = 6$
$10^{-3}$	$\tilde{x} = 2.0000$	$in = 6$	$\tilde{x} = 2.0067$	$ic = 27$	$\tilde{x} = 2.0010$	$is = 7$
$10^{-4}$	$\tilde{x} = 2.0000$	$in = 6$	$\tilde{x} = 2.0001$	$ic = 44$	$\tilde{x} = 2.0000$	$is = 8$
$10^{-5}$	$\tilde{x} = 2.0000$	$in = 7$	$\tilde{x} = 2.0000$	$ic = 62$	$\tilde{x} = 2.0000$	$is = 9$
$10^{-6}$	$\tilde{x} = 2.0000$	$in = 7$	$\tilde{x} = 2.0000$	$ic = 79$	$\tilde{x} = 2.0000$	$is = 9$
$10^{-7}$	$\tilde{x} = 2.0000$	$in = 7$	$\tilde{x} = 2.0000$	$ic = 96$	$\tilde{x} = 2.0000$	$is = 9$
$10^{-8}$	$\tilde{x} = 2.0000$	$in = 7$	$\tilde{x} = 2.0000$	$ic = 113$	$\tilde{x} = 2.0000$	$is = 10$
$10^{-9}$	$\tilde{x} = 2.0000$	$in = 8$	$\tilde{x} = 2.0000$	$ic = 131$	$\tilde{x} = 2.0000$	$is = 10$
$10^{-10}$	$\tilde{x} = 2.0000$	$in = 8$	$\tilde{x} = 2.0000$	$ic = 148$	$\tilde{x} = 2.0000$	$is = 10$
$10^{-11}$	$\tilde{x} = 2.0000$	$in = 8$	$\tilde{x} = 2.0000$	$ic = 165$	$\tilde{x} = 2.0000$	$is = 10$
$10^{-12}$	$\tilde{x} = 2.0000$	$in = 8$	$\tilde{x} = 2.0000$	$ic = 182$	$\tilde{x} = 2.0000$	$is = 11$
$10^{-13}$	$\tilde{x} = 2.0000$	$in = 8$	$\tilde{x} = 2.0000$	$ic = 199$	$\tilde{x} = 2.0000$	$is = 11$
$10^{-14}$	$\tilde{x} = 2.0000$	$in = 8$	$\tilde{x} = 2.0000$	$ic = 217$	$\tilde{x} = 2.0000$	$is = 11$
$10^{-15}$	$\tilde{x} = 2.0000$	$in = 8$	$\tilde{x} = 2.0000$	$ic = 233$	$\tilde{x} = 2.0000$	$is = 11$

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  $\alpha$  se la convergenza della successione dipende in modo critico dalla vicinanza di  $x_0$  ad  $\alpha$ . Il procedimento è *globalmente convergente* quando la convergenza non dipende da quanto  $x_0$  è vicino ad  $\alpha$ . Per i metodi a convergenza locale la scelta del punto di innesco è cruciale.

E' possibile utilizzare  $x_0 = 5/3$  come punto di innesco, in quanto essendo tutti e tre i metodi (**Newton, Corde e Secanti**) localmente convergenti, più la differenza con la radice è minore più velocemente converge.

Notiamo infatti che le distanze tra il nuovo punto di innesco ( $x_0 = \frac{5}{3}$ ) e le due radici del polinomio ( $x_1 = 1$  e  $x_2 = 2$ ) è in entrambi i casi minore rispetto alle distanze tra il vecchio punto di innesco ( $x_0 = 3$ ) e le stesse radici.

$$|\alpha_1 - x_0| = |2 - 5/3| = 0, \bar{3} \leq 1 = |2 - 3| = |\alpha_1 - x|$$

$$|\alpha_2 - x_0| = |1 - 5/3| = 0, \bar{6} \leq 2 = |1 - 3| = |\alpha_2 - x|$$

## 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

```

1  % newton(f, f1, x0, imax, tolX)
2  % Metodo di Newton generico.
3  %
4  % Input:
5  % f: la funzione;
6  % f1: la derivata della funzione;
7  % x0: l'approssimazione iniziale;
8  % imax: il numero massimo di iterazioni;
9  % tolX: la tolleranza desiderata;
10 % Output :
11 % x0: radici della funzione.
12
13 function x0 = newton(f, f1, x0, imax, tolX)
14     in = 0;
15     vai = true;
16     while ( in<imax ) && vai
17         in = in+1;
18         fx = feval(f, x0);
19         f1x = feval(f1, x0);
20         if f1x==0
21             vai=false;
22             in=in-1;
23             break
24         end
25         x1 = x0 - fx/f1x;
26         vai = abs(x1-x0)>tolX;
27         x0 = x1;

```

```

28     end
29     in
30 end

```

#### • Metodo di Newton modificato

```

1  % newtonMod(f, f1, x0, m, imax, tolx)
2  % Metodo di Newton modificato.
3  %
4  % Input:
5  % f: la funzione;
6  % f1: la derivata della funzione;
7  % x0: l'approssimazione iniziale;
8  % m: la molteplicità della radice;
9  % imax: il numero massimo di iterazioni;
10 % tolx: la tolleranza desiderata;
11 % Output :
12 % x0: radici della funzione
13
14 function x0 = newtonMod(f, f1, x0, m, imax, tolx)
15     inm = 0;
16     vai = true;
17     while ( inm<imax ) && vai
18         inm = inm+1;
19         fx = feval(f, x0);
20         f1x = feval(f1, x0);
21         if f1x==0
22             vai=false;
23             inm=inm-1;
24             break
25         end
26         x1 = x0 - m*(fx/f1x);
27         vai = abs(x1-x0)>tolx;
28         x0 = x1;
29     end
30     inm
31 end

```

#### • Metodo di Aitken

```

1  % aitken(f, f1, x0, imax, tolx)
2  % Metodo di accelerazione di Aitken.
3  %
4  % Input:
5  % f: la funzione
6  % f1: la derivata della funzione;
7  % x0: l'approssimazione iniziale;
8  % imax: il numero massimo di iterazioni;
9  % tolx: la tolleranza desiderata;
10 % Output :
11 % x0: radici della funzione
12
13 function x0 = aitken(f, f1, x0, imax, tolx)
14     ia = 0;
15     vai = true;
16     while ( ia<imax ) && vai
17         ia = ia+1;
18         fx = feval(f, x0);
19         f1x = feval(f1, x0);

```

```

20         if f1x==0
21             vai=false;
22             ia=ia-1;
23             break
24         end
25         x1 = x0 - fx/f1x;
26         fx = feval(f, x1);
27         f1x = feval(f1, x1);
28         if f1x==0
29             vai=false;
30             ia=ia-1;
31             break
32         end
33         x2 = x1 - fx/f1x;
34         if (x2-2*x1+x0)==0
35             vai=false;
36             ia=ia-1;
37             break
38         end
39         x3 = (x2*x0-x1^2)/(x2-2*x1+x0);
40         vai = abs(x3-x0)>tolx;
41         x0 = x3;
42     end
43     ia
44 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  $x_0 = 0$ :

```

1  p = inline('x^3-4*x^2+5*x-2');
2  pd = inline('3*x^2-8*x+5');
3  tolx = 10^-1;
4  xn = [];
5  xnm1 = [];
6  xnm2 = [];
7  xa = [];
8  j = 1;
9  while tolx>eps
10     xn(j) = newton(p, pd, 0, 1000, tolx);
11     xnm1(j) = newtonMod(p, pd, 0, 1, 1000, tolx);
12     xnm2(j) = newtonMod(p, pd, 0, 2, 1000, tolx);
13     xa(j) = aitken(p, pd, 0, 1000, tolx);
14     tolx = tolx/10;
15     j = j+1;
16 end
17
18 xn, xnm1, xnm2, xa

```

restituisce i seguenti valori:

$tol_x$	<i>Newton</i>		<i>NewtonMod m = 1</i>		<i>NewtonMod m = 2</i>		<i>Aitken</i>	
$10^{-1}$	$\tilde{x} = 0.8960$	$in = 4$	$\tilde{x} = 0.8960$	$inm_1 = 4$	$\tilde{x} = 0.9999$	$inm_2 = 3$	$\tilde{x} = 1.0020$	$ia = 2$
$10^{-2}$	$\tilde{x} = 0.9929$	$in = 8$	$\tilde{x} = 0.9929$	$inm_1 = 8$	$\tilde{x} = 1.0000$	$inm_2 = 4$	$\tilde{x} = 1.0000$	$ia = 3$
$10^{-3}$	$\tilde{x} = 0.9991$	$in = 11$	$\tilde{x} = 0.9991$	$inm_1 = 11$	$\tilde{x} = 1.0000$	$inm_2 = 4$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-4}$	$\tilde{x} = 0.9999$	$in = 15$	$\tilde{x} = 0.9999$	$inm_1 = 15$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-5}$	$\tilde{x} = 1.0000$	$in = 18$	$\tilde{x} = 1.0000$	$inm_1 = 18$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-6}$	$\tilde{x} = 1.0000$	$in = 21$	$\tilde{x} = 1.0000$	$inm_1 = 21$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-7}$	$\tilde{x} = 1.0000$	$in = 25$	$\tilde{x} = 1.0000$	$inm_1 = 25$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-8}$	$\tilde{x} = 1.0000$	$in = 29$	$\tilde{x} = 1.0000$	$inm_1 = 29$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-9}$	$\tilde{x} = 1.0000$	$in = 29$	$\tilde{x} = 1.0000$	$inm_1 = 29$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-10}$	$\tilde{x} = 1.0000$	$in = 29$	$\tilde{x} = 1.0000$	$inm_1 = 29$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-11}$	$\tilde{x} = 1.0000$	$in = 29$	$\tilde{x} = 1.0000$	$inm_1 = 29$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-12}$	$\tilde{x} = 1.0000$	$in = 29$	$\tilde{x} = 1.0000$	$inm_1 = 29$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-13}$	$\tilde{x} = 1.0000$	$in = 29$	$\tilde{x} = 1.0000$	$inm_1 = 29$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-14}$	$\tilde{x} = 1.0000$	$in = 29$	$\tilde{x} = 1.0000$	$inm_1 = 29$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$
$10^{-15}$	$\tilde{x} = 1.0000$	$in = 29$	$\tilde{x} = 1.0000$	$inm_1 = 29$	$\tilde{x} = 1.0000$	$inm_2 = 5$	$\tilde{x} = 1.0000$	$ia = 4$

Si vede da questi risultati che i metodi di Newton modificato con molteplicità  $m = 2$  e di Aitken convergono molto velocemente alla soluzione, mentre il metodo di Newton e di Newton modificato con  $m = 1$  (tale valore di molteplicità rende identici i valori restituiti), seppur convergendo, 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 :

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

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

```

1 % newtonSqrtAlpha(alpha, x0, imax, tolX)
2 % Metodo di Newton ottimizzato per l'approssimazione della radice
3 % quadrata.
4 %
5 % Input:
6 % alpha: l'argomento della radice quadrata;
7 % x0: l'approssimazione iniziale;
8 % imax: il numero massimo di iterazioni;
9 % tolX: la tolleranza desiderata.
10 % Output:
11 % xn: vettore radici;
12 % exn: vettore errore.
13
14 function [xn,exn] = newtonSqrtAlpha(alpha, x0, imax, tolX)
15     format long e;
16     xn = [];

```



```

17     exn = [];
18     i = 1;
19     xn(i) = x0;
20     exn(i) = x0-sqrt(alpha);
21     i = i+1;
22     x = (x0+alpha/x0)/2;
23     xn(i) = x;
24     exn(i) = x-sqrt(alpha);
25     while( i<imax ) && ( abs(x-x0)>tolx )
26         i = i+1;
27         x0 = x;
28         x = (x0+alpha/x0)/2;
29         xn(i) = x;
30         exn(i) = x-sqrt(alpha);
31     end
32 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  $tol_x = eps$  :

```

1 [xn,exn] = newtonSqrtAlpha(5, 5, 100, eps);
2
3 xn, exn

```

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 :

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

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

```
1 % secantiSqrtAlpha(alpha, x0, x1, imax, tolX)
2 % Metodo delle Secanti ottimizzato per l'approssimazione della radice
3 % quadrata.
4 %
5 % Input:
6 % alpha: l'argomento della radice quadrata;
7 % x0: l'approssimazione iniziale;
8 % imax: il numero massimo di iterazioni;
9 % tolX: la tolleranza desiderata.
10 % Output:
11 % xs: vettore radici;
12 % exs: vettore errore.
13
14 function [xs, exs] = secantiSqrtAlpha(alpha, x0, x1, imax, tolX)
15     format long e;
16     x = x1;
17     xs = [];
18     exs = [];
19     i = 1;
20     xs(i) = x0;
21     exs(i) = x0-sqrt(alpha);
22     i = i+1;
23     xs(i) = x;
24     exs(i) = x-sqrt(alpha);
25     while ( i<imax ) && ( abs(x-x0)>tolX )
26         i = i+1;
27         x1 = (x*x0 + alpha)/(x + x0);
28         x0 = x;
29         x = x1;
30         xs(i) = x;
31         exs(i) = x-sqrt(alpha);
32     end
33 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  $tol_x = eps$  :

```
1 [xs, exs] = secantiSqrtAlpha(5, 5, 3, 100, eps);
2
3 xs, exs
```

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$
$i = 2$	$x_2 = 2.500000000000000e + 00$	$ \epsilon_2  = 2.639320225002102e - 01$
$i = 3$	$x_3 = 2.272727272727273e + 00$	$ \epsilon_3  = 3.665929522748312e - 02$
$i = 4$	$x_4 = 2.238095238095238e + 00$	$ \epsilon_4  = 2.027260595448332e - 03$
$i = 5$	$x_5 = 2.236084452975048e + 00$	$ \epsilon_5  = 1.647547525829296e - 05$
$i = 6$	$x_6 = 2.236067984964863e + 00$	$ \epsilon_6  = 7.465073448287285e - 09$
$i = 7$	$x_7 = 2.236067977499817e + 00$	$ \epsilon_7  = 2.753353101070388e - 14$
$i = 8$	$x_8 = 2.236067977499790e + 00$	$ \epsilon_8  = 4.440892098500626e - 16$
$i = 9$	$x_9 = 2.236067977499790e + 00$	$ \epsilon_9  = 0$
$i = 10$	$x_{10} = 2.236067977499790e + 00$	$ \epsilon_{10}  = 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

```
1 % triangolareInferiore(A, b)
2 % Metodo per la risoluzione di una matrice tringolare inferiore.
3
4 % Input:
5 % A: matrice triangolare inferiore;
6 % b: vettore dei termini noti.
7 % Output:
8 % x: vettore delle soluzioni del sistema.
9
10 function x = triangolareInferiore(A,b)
11     x = b;
12     if ~ismatrix(A)
13         error(A non e' una matrice);
14     end
15     [n,m] = size(A);
16     if (n~=m)
17         error(A non e' una matrice quadratica);
18     end
19     for j=1:n
20         if (A(j,j)~=1)
21             error(A non ha coefficienti diagonali unitari)
22         end
23     end
24     if (~isvector(x))
25         error(b non e' un vettore);
26     end
27     vectorSize = size(x);
28     if (vectorSize~=n)
29         error(Il vettore deve avere endfor j=1:nfor i = j+1:nx(i) =
                x(i)-A(i,j)*x(j);endendend
```

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

```
1 A = [1 2 0;2 1 0;2 2 1];
2 b = [2 2 2];
3
4 x = triangolareInferiore(A,b);
5 x
```

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  $LDL^t$

```
1 % fattorizzazioneLDLt(A)
2 % Metodo per la fattorizzazione LDLt di una matrice.
3
4 % Input:
5 % A: matrice sdp da fattorizzare.
6 % Output:
7 % A: matrice riscritta L, D e Lt.
8
9 function A = fattorizzazioneLDLt(A)
10 [m,n]=size(A);
11 if m~=n
12     error(La matrice non e' quadrata!);
13 end
14 if A(1,1)<=0
15     error(La matrice non e' SDP);
16 end
17 A(2:n,1) = A(2:n,1)/A(1,1);
18 for j = 2:n
19     v = (A(j,1:j-1).') .* diag(A(1:j-1,1:j-1));
20     A(j,j) = A(j,j) - A(j,1:j-1)*v;
21     if A(j,j)<=0
22         error(la matrice non e' SDP)
23     end
24     A(j+1:n,j) = (A(j+1:n,j) - A(j+1:n,1:j-1)*v)/A(j,j);
25 end
26 end
```

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

```
1 A1 = [1 -1 2 2; -1 5 -14 2; 2 -14 42 2; 2 2 2 65];
2 LDLt1 = fattorizzazioneLDLt(A1);
3 LDLt1
4
5 A2 = [1 -1 2 2; -1 6 -17 3; 2 -17 48 -16; 2 3 -16 4];
6 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  $LDL^t$ :

The screenshot shows the MATLAB R2017b interface. The Editor window displays the script `fattorizzazioneLDL.m`. The script is paused at line 19, which is the `if` statement checking for symmetry: `if A(1,1)~=0`. The Command Window shows the error message: `error('La matrice non e' quadrata!');`. The Variables window shows the current state of variables: `A` is a 4x4 double matrix, `m` is 4, `n` is 4, and `v` is a 4x1 double vector.

The screenshot shows the MATLAB R2017b interface. The Editor window displays the script `fattorizzazioneLDL.m`. The script is paused at line 21, which is the `if` statement checking for positive definiteness: `if A(j,j)~=0`. The Command Window shows the error message: `error('La matrice non e' sdp');`. The Variables window shows the current state of variables: `A` is a 4x4 double matrix, `m` is 4, `n` is 4, and `v` is a 4x1 double vector.



### 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  $LDL^t x = b$

```

1 % risolutoreLDLt(LDLt,b)
2 % Metodo per la risoluzione di una matrice LDLt.
3
4 % Input:
5 % LDLt: matrice;
6 % b: vettore dei termini noti.
7 % Output:
8 % x: vettore delle soluzioni del sistema.
9
10 function x = risolutoreLDLt(LDLt, b)
11     LDLt = fattorizzazioneLDLt(LDLt);
12     L = tril(LDLt,-1)+eye(length(LDLt));
13     D = diag(diag(LDLt));
14     Lt = (tril(LDLt,-1)+eye(length(LDLt)))';
15     L, D, Lt
16     x1 = triangolareInferiore(L,b);
17     x2 = diagonale(D,x1);
18     x = triangolareSuperiore(Lt,x2);
19 end

```

il quale implementa in ordine:

1. **fattorizzazioneLDLt( $LDL^t$ )**  
Una funzione di fattorizzazione di una 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  $tril(DDL^t, k < 0)$  che restituisce gli elementi sotto la  $k$ -esima diagonale di  $DDL^t$ ;  $eye(n)$  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  $DDL^t$  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

```

1  % diagonale(D,b)
2  % Metodo per il calcolo del vettore incognite di una matrice diagonale.
3  %
4  % Input:
5  % D: matrice diagonale.
6  % b: vettore dei termini noti.
7  % Output:
8  % x: vettore delle soluzioni del sistema.
9
10 function x = diagonale(D,b)
11     x = b;
12     if ~ismatrix(D)
13         error(D non e' una matrice);
14     end
15     [n,m] = size(D);
16     if (n~=m)
17         error(D non e' una quadratica);
18     end
19     if (~isvector(x))
20         error(x non e' un vettore);
21     end
22     vectorSize = size(x,1);
23     if (vectorSize~=n)
24         error('Il vettore deve avere %i righe, invece ha %i righe', n,
                vectorSize);
25     end
26     for j=1:n
27         x(j) = x(j)/D(j,j);
28     end
29 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  $tril(DDL^t, k < 0)$  che restituisce gli elementi sotto la  $k$ -esima diagonale di  $DDL^t$ ;  $eye(n)$  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

```

1  % triangolareSuperiore(A, b)
2  % Metodo per la risoluzione di una matrice triangolare superiore.
3
4  % Input:

```



```

5 % A: matrice triangolare superiore;
6 % b: vettore dei termini noti.
7 % Output:
8 % x: vettore delle soluzioni del sistema.
9
10 function x = triangolareSuperiore(A,b)
11     x = b;
12     if ~ismatrix(A)
13         error(A non e' una matrice);
14     end
15     [n,m] = size(A);
16     if(n~=m)
17         error(A non e' una matrice quadratica);
18     end
19     if(~isvector(x))
20         error(b non e' un vettore);
21     end
22     vectorSize = size(x,1);
23     if(vectorSize~=n)
24         error('Il vettore deve avere %i riche, invece ha %i righe', n,
                vectorSize);
25     end
26     for j=n:-1:1
27         x(j) = x(j)/A(j,j);
28         for i = 1:j-1
29             x(i) = x(i)-A(i,j)*x(j);
30         end
31     end
32 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

```

1 % risolutoreLUPiv(LU,b)
2 % Metodo per la risoluzione di una matrice LUPiv.
3
4 % Input:
5 % LU: matrice;
6 % b: vettore dei termini noti;
7 % Output:
8 % x: vettore delle soluzioni del sistema.
9
10 function x = risolutoreLUpiv(LU, b)
11     [LU,p] = fattorizzazioneLUPiv(LU);
12     P=zeros(length(LU));
13     for i=1:length(LU)
14         P(i, p(i)) = 1;
15     end
16     Pb = P*b;
17     L = tril(LU,-1)+eye(length(LU));
18     U = triu(LU);
19     P, Pb, L, U
20     x1 = triangolareInferiore(L, Pb);
21     x = triangolareSuperiore(U, x1);
22 end

```

il quale implementa in ordine:

1. **fattorizzazioneLUpiv(LU,b)**

Una funzione di fattorizzazione di una 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

```
1 % [A, p] = fattorizzaLUpiv(A)
2 % Metodo per la fattorizzazione LU di una matrice.
3 %
4 % Input:
5 % A: matrice sdp da fattorizzare.
6 % Output:
7 % A: matrice riscritta L e U;
8 % p: vettore contenente l'informazione della matrice di permutazione P.
9
10 function [A, p] = fattorizzazioneLUpiv(A)
11     [m,n]=size(A);
12     if m~=n
13         error('La matrice non e' quadrata!');
14     end
15     p=[1:n];
16     for i=1:(n-1)
17         [aki, ki] = max(abs(A(i:n,i)));
18         if aki==0
19             error('La matrice e' singolare!');
20         end
21         ki = ki+i-1;
22         if ki>i
23             A([i,ki],:) = A([ki,i],:);
24             p([i,ki]) = p([ki,i]);
25         end
26         A((i+1):n,i) = A((i+1):n,i)/A(i,i);
27         A((i+1):n,(i+1):n) = A((i+1):n,(i+1):n)-A((i+1):n,i)*A(i,(i+1):n);
28     end
29 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 zero 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 zero 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:

```

1  LDLt = [3 2 -1; 2 7 7; -1 7 30];
2  xt1 = [4; 5; 3];
3  b1 = LDLt*xt1;
4  b1
5  x1 = risolutoreLDLt(LDLt,b1);
6  x1
7  r1 = LDLt*x1-b1;
8  k1 = cond(LDLt);
9  krb1 = norm(r1)/norm(b1);
10 kxtx1 = norm(x1 - xt1)/norm(xt1);
11 r1, k1, krb1, kxtx1
12
13 LU = [-23 5 -21 8; 0 0 5 7; 1 54 7 9; 0 -8 12 4];
14 xt2 = [2; 8; 3; 5];
15 b2 = LU*xt2;
16 b2
17 x2 = risolutoreLU piv(LU, b2);
18 x2
19 r2 = LU*x2-b2;
20 k2 = cond(LU);
21 krb2 = norm(r2)/norm(b2);
22 kxtx2 = norm(x2 - xt2)/norm(xt2);
23 r2, k2, krb2, kxtx2

```

**Esempio :** *risolutoreLDL<sup>t</sup>*

con i seguenti parametri di input :

$$A_1 = LDL^t = \begin{bmatrix} 3 & 2 & -1 \\ 2 & 7 & 7 \\ -1 & 7 & 30 \end{bmatrix} \quad \hat{x}_1 = \begin{bmatrix} 4 \\ 5 \\ 3 \end{bmatrix} \quad b_1 = LDL^t \hat{x}_1 = \begin{bmatrix} 19 \\ 64 \\ 121 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0,6667 & 1 & 0 \\ -0,3333 & 1,3529 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 5,6667 & 0 \\ 0 & 0 & 19,2941 \end{bmatrix} \quad L^t = \begin{bmatrix} 1 & 0,6667 & -0,3333 \\ 0 & 1 & 1,3529 \\ 0 & 0 & 1 \end{bmatrix}$$

Il risultato ottenuto è:

$$x_1 = \begin{bmatrix} 4,0000000000000000 \\ 4,9999999999999999 \\ 3,0000000000000000 \end{bmatrix} \quad r_1 = LDL^t * x_1 - b_1 = \begin{bmatrix} 0 \\ -7,1054e-15 \\ 0 \end{bmatrix}$$

**Esempio :** *risolutoreLU piv*

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:

$$x_2 = \begin{bmatrix} 2.0000000000000000 \\ 8.0000000000000000 \\ 3.0000000000000000 \\ 5.0000000000000001 \end{bmatrix} \quad r_2 = LU * x_2 - b_2 = \begin{bmatrix} 7.1054e - 15 \\ 7.1054e - 15 \\ 0 \\ -3.5527e - 15 \end{bmatrix}$$

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  $LU$

```

1 % A = fattorizzazioneLU(A)
2 % Fattorizzazione LU di una matrice nonsingolare con tutti i minori
3 % principali non nulli.
4 %
5 % Input:
6 % A: la matrice nonsingolare da fattorizzare LU.
7 % Output:
8 % A: la matrice riscritta con le informazioni dei fattori L ed U.
9
10 function A = fattorizzazioneLU(A)
11     [m,n]=size(A);
12     if m~=n
13         error('La matrice non e' quadrata!');
14     end
15     for i=1:n-1
16         if A(i,i)==0
17             error('La matrice non e' fattorizzabile LU!');
18         end
19         A(i+1:n,i) = A(i+1:n,i)/A(i,i);
20         A(i+1:n,i+1:n) = A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
21     end
22 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  $\text{tril}(A)$  che restituisce la parte triangolare superiore di  $A$ ) e successivamente vengono eseguiti i comandi  $U \setminus (L \setminus b)$  *Gauss senza pivoting* e  $A \setminus b$  *Gauss con pivoting*:

```

1 A = [10^(-13) 1 ; 1 1];
2 LU = fattorizzazioneLU(A);
3 L = tril(LU,-1)+eye(length(LU));
4 U = triu(LU);
5 LU = L*U;
6 L, U, LU
7
8 format long e;
9 e = [1; 1];
10 b = A*e;
11 Sp = U \ (L \ b);
12 Cp = A \ b;
13 b, Sp, Cp

```

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.

$$e = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad b = Ae = \begin{bmatrix} 1.0000000000000100 \\ 2 \end{bmatrix}$$

• **Gauss senza pivoting**

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

• **Gauss con pivoting**

$$Cp = A \setminus b = \begin{bmatrix} 1.0000000000000000 \\ 1.0000000000000000 \end{bmatrix}$$

### 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**

```

1 % triangolareInferioreMod(alpha, b)
2 % Metodo per la risoluzione di una matrice bidiagonale inferiore a diagonale
   unitaria di Toeplitz
3
4 % Input:
5 % alpha: valore ripetuto nella diagonale inferiore;
6 % b: vettore dei termini noti.
7 % Output:
8 % b: vettore delle soluzioni del sistema.
9
10 function b = triangolareInferioreMod(alpha,b)
11     if(~isvector(b))
12         error(b non e' un vettore);
13     end
14     n = size(b,1);
15     for i=2:n
16         b(i) = b(i) - alpha*b(i-1);
17     end
18 end

```

#### • Implementazione

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

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

```

1 b1 = [1; 101*ones(12,1)];
2 b2 = 0.1*[1; 101*ones(12,1)];
3 x1 = triangolareInferioreMod(100,b1);
4 x2 = triangolareInferioreMod(100,b2);
5 x1
6 x2
7
8 A = eye(12)+100*[ zeros(1, 12); eye(11) zeros(1, 11)'];
9 k = cond(A);
10 ninf = norm(A,inf);
11 n1 = norm(A,inf);
12 ninv = norm(A^-1, inf);

```

restituendo i seguenti valori:

$$x_1 = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \quad x_2 = \begin{bmatrix} 1.000000000000000e-01 \\ 1.000000000000001e-01 \\ 9.99999999985931e-02 \\ 1.000000000140702e-01 \\ 9.99999859298470e-02 \\ 1.000001407015318e-01 \\ 9.998592984681665e-02 \\ 1.014070153183368e-01 \\ -4.070153183368319e-02 \\ 1.417015318336832e+01 \\ -1.406915318336832e+03 \\ 1.407016318336832e+05 \end{bmatrix}$$

- **Studio condizionamento**  
Risulta

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

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

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

nel caso  $n = 12$ , si ha  $k_{\infty}(A) > 10^{24}$  quindi il problema è malcondizionato. Su tale matrice, la function *cond* restituisce *Inf*. La *norma*  $\infty$  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*  $\infty$  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  $\underline{b1}$  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**

```

1  % A = fattorizzaQR(A)
2  % Fattorizzazione QR di Householder per matrici mxn con m>=n.
3  %
4  % Input:
5  % A: la matrice da fattorizzare QR.
6  % Output:
7  % A: la matrice riscritta con le informazioni dei fattori Q ed R.
8
9  function A = fattorizzazioneQR(A)
10     [m,n]=size(A);
11     for i=1:n
12         alfa = norm(A(i:m, i), 2);
13         if alfa==0
14             error('La matrice non ha rango massimo');
15         end

```

```

16         if(A(i,i))>=0
17             alfa = -alfa;
18         end
19         v1 = A(i,i)-alfa;
20         A(i,i) = alfa;
21         A(i+1:m, i) = A(i+1:m, i)/v1;
22         beta = -v1/alfa;
23         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));
24     end
25 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

```

1  % x = risolutoreQR(A,b)
2  % Risoluzione di un sistema lineare sovredeterminato del tipo Ax=b
3  % tramite fattorizzazione QR di Householder della matrice dei
4  % coefficienti.
5  %
6  % Input:
7  % A: matrice dei coefficienti mxn dove m>n;
8  % b: vettore dei termini noti.
9  % Output:
10 % b: vettore delle soluzioni del sistema lineare sovradeterminato.
11
12 function b = risolutoreQR(A, b)
13     [m,n] = size(A);
14     QR = fattorizzazioneQR(A);
15     Qt = eye(m);
16     for i=1:n
17         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;
18     end
19     R = triu(QR(1:n, :));
20     Q = Qt';
21     R, Q, Qt, QR
22     b = triangolareSuperiore(R, Qt(1:n, :)*b);
23 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:

```

1 A1 = [3 2 1; 1 2 3; 1 2 1; 2 1 2];
2 b1 = [10; 10; 10; 10];
3 xqr1 = risolutoreQR( A1, b1 );
4 xab1 = A1\b1;
5 xqr1, xab1
6
7 A2 = [9 -14 -3; 4 9 6; 33 4 12; 7 -23 4];

```



```

8 | b2 = [12; -5; 9; -25];
9 | xqr2 = risolutoreQR( A2, b2 );
10 | xab2 = A2\b2;
11 | xqr2, xab2

```

### Esempio

1. con input:

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

• Fattorizzazione QR e  $A \backslash b$

$$x_{1(QR)} = \begin{bmatrix} 1.4000000000000002 \\ 2.8000000000000000 \\ 1.3999999999999998 \end{bmatrix} \quad x_{1(A \backslash b)} = \begin{bmatrix} 1.3999999999999999 \\ 2.8000000000000000 \\ 1.4000000000000001 \end{bmatrix}$$

2. con input:

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

• Fattorizzazione QR e  $A \backslash b$

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

### 3.10 Esercizio 10

Per la risoluzione di sistemi nonlineari, ovvero del tipo

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

con  $F$  costituita dalle *funzioni componenti*

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

ed  $x$  il vettore delle incognite che risolvono il sistema

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

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

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

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 risoluzione di sistemi non lineari tramite l'utilizzo del **metodo di Newton**:

```

1 % x = newtonNonLin(f, J, x, imax, tol)
2 % Metodo per la risoluzione di sistemi non lineari con il metodo di Newton.
3 %
4 % Input :
5 % F: sistema non lineare;
6 % J: Jacobiano;
7 % x: punto iniziale;
8 % imax: passi massimi;
9 % tol: tolleranza.
10 % Output :
11 % x: minimo relativo.
12
13 function x = newtonNonLin(f, J, x, imax, tol)
14     i=0;
15     xold=x;
16     while(i<imax) && (norm(x-xold)>tol)
17         i=i+1;
18         xold=x;
19         val = -feval(f,x);
20         b = [val(1);val(2)];
21         x = x + risolutoreLUpiv(J, b);
22         i, b
23     end
24 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 non lineare 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  $f = f(x_1, x_2) = x_1^2 + x_2^3 - x_1x_2$ , per risolvere il seguente sistema non lineare con i relativi parametri di input:

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

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

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

```

1 format short;
2
3 zero = [1/12;1/6];
4
5 x = [1/2;1/2];
6 f = inline('x(1)^2+x(2)^3-x(1)*x(2)');
7 F = inline('[2*x(1)-x(2), 3*x(2)^2-x(1)]');

```

```

8 J = [2-x(2), 2*x(1)-1; 3*x(2)^2-1, 6*x(2)-x(1)];
9
10 tol_x = 10^-3;
11 x1 = newtonNonLin(F, J, x, 100, tol_x);
12 n1 = norm(x1);
13 e1 = norm(zero-x1)
14 x1, n1
15
16 tol_x = 10^-6;
17 x2 = newtonNonLin(F, J, x, 100, tol_x);
18 n2 = norm(x2);
19 e2 = norm(zero-x2)
20 x2, n2

```

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

$tol_x = 10^{-t}$	$i$	$\ n\ $	$\ e\ $
$10^{-3}$	$i = 17$	$\ n_1\  = 0.190126478566088$	$\ e_1\  = 0.003794501517081$
$10^{-6}$	$i = 51$	$\ n_2\  = 0.186343470827848$	$\ e_2\  = 4.480819013409465e-06$

## 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:  $y = \text{lagrange}(xi, fi, x)$

```
1 % y = lagrange(xi, fi, x)
2 %   Calcola il polinomio interpolante di grado n in forma di Lagrange, nei
3 %   punti x.
4 %
5 % Input:
6 %   -xi : vettore contenente le ascisse di interpolazione su cui calcolare
7 %   la differenza divisa;
8 %   -fi : vettore contenente i valori assunti dalla funzione in
9 %   corrispondenza dei punti xi.
10 %   -x : vettore contenente i valori su cui calcolare il polinomio
11 %   interpolante
12 %
13 % Output:
14 %   -y : vettore contenente il valore del polinomio interpolante calcolato
15 %   sulle x.
16
17 function [y] = lagrange(xi, fi, x)
18     n = length(xi);
19     m = length(x);
20     y = zeros(m,1);
21     for i = 1:m
22         y(i) = 0;
23         for j = 1:n
24             range = [1:j-1, j+1:n];
25             bl = prod(x(i) - xi(range))/prod(xi(j) - xi(range));
26             y(i) = y(i) + fi(j) * bl;
27         end
28     end
29 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:  $y = \text{newton}(xi, fi, x)$

```
1 % y = newton(xi, fi, x)
2 %   Calcola il polinomio interpolante di grado n in forma di Newton, nei
3 %   punti x.
4 %
5 % Input:
6 %   -xi : vettore contenente le ascisse di interpolazione su cui calcolare
7 %   la differenza divisa;
8 %   -fi : vettore contenente i valori assunti dalla funzione in
9 %   corrispondenza dei punti xi.
10 %   -x : vettore contenente i valori su cui calcolare il polinomio
11 %   interpolante
12 %
13 % Output:
14 %   -y : vettore contenente il valore del polinomio interpolante calcolato
15 %   sulle x.
```

```

16
17 function [y] = newton(xi, fi, x)
18     n = length(xi)-1;
19     for j = 1:n
20         for i = n+1:-1:j+1
21             fi(i) = (fi(i)-fi(i-1))/(xi(i)-xi(i-j));
22         end
23     end
24     y = fi(n+1)*ones(size(x));
25     for i = n:-1:1
26         y = y.*(x-xi(i))+fi(i);
27     end
28 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:  $y = \text{newton}(xi, fi, fli, x)$

```

1  % y = hermite(xi, fi, fli, x)
2  %   Calcola il polinomio interpolante di grado n in forma di Hermite, nei
3  %   punti x.
4  %
5  % Input:
6  %   -xi : vettore contenente le ascisse di interpolazione su cui calcolare
7  %   la differenza divisa;
8  %   -fi : vettore contenente i valori assunti dalla funzione in
9  %   corrispondenza dei punti xi.
10 %   -fli : vettore contenente i valori assunti dalla derivata prima della
11 %   funzione in corrispondenza dei punti xi.
12 %   -x : vettore contenente i valori su cui calcolare il polinomio
13 %   interpolante
14 %
15 % Output:
16 %   -y : vettore contenente il valore del polinomio interpolante calcolato
17 %   sulle x.
18
19 function [y] = hermite(xi, fi, fli, x)
20     n = length(xi)-1;
21     xh = zeros(2*n+2, 1);
22     xh(1:2:2*n+1) = xi;
23     xh(2:2:2*n+2) = xi;
24     fh = zeros(2*n+2, 1);
25     fh(1:2:2*n+1) = fi;
26     fh(2:2:2*n+2) = fli;
27     nh = length(xh)-1;
28     % Calcolo delle differenze divise
29     for i = nh:-2:3
30         fh(i) = (fh(i)-fh(i-2))/(xh(i)-xh(i-1));
31     end
32     for i = 2:nh
33         for j = nh+1:-1:i+1
34             fh(j) = (fh(j)-fh(j-1))/(xh(j)-xh(j-i));
35         end
36     end
37     % Horner
38     y = fh(nh+1)*ones(size(x));

```

```

39     for i = nh:-1:1
40         y = y.*(x-xh(i))+fh(i);
41     end
42     y = y.';
43 end

```

#### 4.4 Esercizio 4

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

$$f_i = \sin(x) \quad [0, 2\pi] \quad f'_i = \cos(x) \\ x_i = i\pi \quad i = 0, 1, 2$$

```

1  xi = zeros(3,1);
2  fi = zeros(3,1);
3  fli = zeros(3,1);
4  for i = 0:length(xi)-1
5      xi(i+1) = i*pi;
6      fi(i+1) = sin(xi(i+1));
7      fli(i+1) = cos(xi(i+1));
8  end
9  x = [xi(1); 1; xi(2); 5; xi(3)];
10
11 y1 = lagrange(xi, fi, x);
12 plot(xi, fi, x, y1);
13
14 y2 = newton(xi, fi, x);
15 plot(xi, fi, x, y2);
16
17 y3 = hermite(xi, fi, fli, x);
18
19 % li plotto su due grafici diversi perche xi,fi altrimenti
20 % xi,fi e' indistinguibile dall ascissa del piano cartesiano
21 plot(xi, fi);
22 plot(x, y3);

```

Mostriamo nel seguente plot l'approssimazione della funzione tramite l'utilizzo delle funzioni di interpolazione, precedentemente elencate:

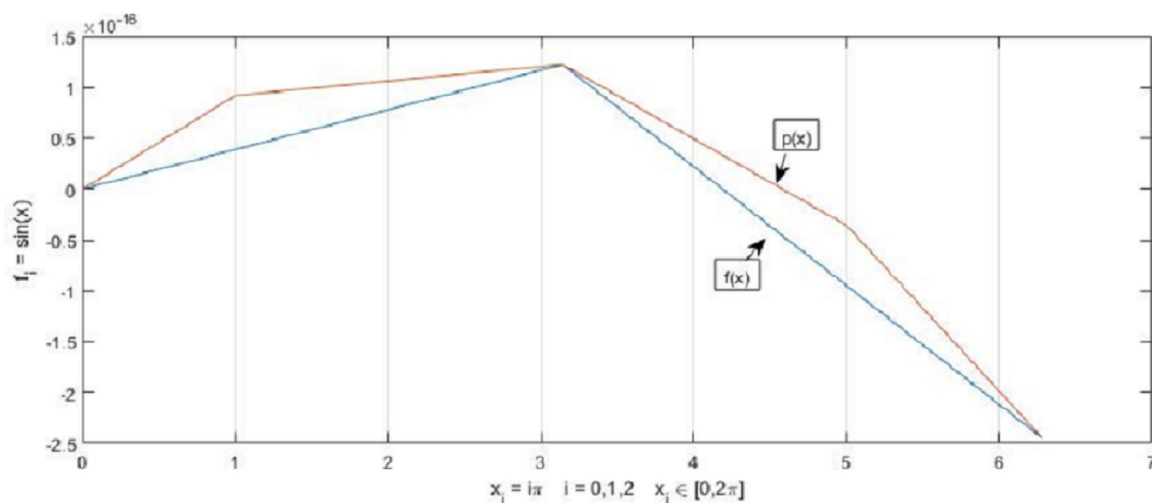


Figure 2: Interpolazione della funzione  $\sin(x)$

## 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 :  $y = \text{spline3}(xi, fi, x, tipo)$ .

```
1 % y = spline3(xi, fi, x, tipo)
2 %   Determina le espressioni degli n polinomi che formano una spline
3 %   cubica naturale o con condizioni not-a-knot e la valuta su una serie
4 %   di punti.
5 %
6 % Input:
7 %   -xi: vettore contenente gli n+1 nodi di interpolazione;
8 %   -fi: vettore contenente i valori assunti dalla funzione da
9 %   approssimare nei nodi in xi;
10 %   -x: vettore di m punti su cui si vuole valutare la spline.
11 %   -tipo: true se la spline implementa condizioni not-a-knot, false se
12 %   invece e' una spline naturale.
13 % Output:
14 %   -y: vettore di m valori contenente la valutazione dei punti in x
15 %   della spline (NaN se un punto non e' valutabile).
16
17 function [y] = spline3(xi, fi, x, tipo)
18     phi = zeros(length(xi)-2, 1);
19     xxi = zeros(length(xi)-2, 1);
20     dd = zeros(length(xi)-2, 1);
21     for i=2:length(xi)-1
22         hi = xi(i) - xi(i-1);
23         h1i = xi(i+1) - xi(i);
24         phi(i) = hi/(hi+h1i);
25         xxi(i) = h1i/(hi+h1i);
26         dd(i) = differenzaDivisa(xxi(i-1:i+1), fi(i-1:i+1));
27     end
28     if tipo
29         mi = risolviSistSplineNotAKnot(phi, xxi, dd);
30     else
31         mi = risolviSistSplineNaturale(phi, xxi, dd);
32     end
33     s = esprSpline3(xi, fi, mi);
34     y = valutaSpline(xi, s, x);
35 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:  $xi = \text{ceby}(n, a, b)$

```
1 % xi = ceby(n, a, b)
2 % Calcola le ascisse di Chebyshev su un determinato intervallo.
3 %
4 % Input:
5 %   -a: l'estremo sinistro dell'intervallo;
6 %   -b: l'estremo destro dell'intervallo;
7 %   -n: il numero di ascisse che si vuole generare (n+1, da 0 a n).
8 %
9 % Output:
```

```

10 % -xi: vettore contenente le ascisse di Chebyshev generate.
11
12 function xi = ceby(n, a, b)
13     xi = zeros(n+1, 1);
14     for i = 0:n
15         xi(n+1-i) = (a+b)/2 + cos(pi*(2*i+1)/(2*(n+1)))*(b-a)/2;
16     end
17 end

```

#### 4.7 Esercizio 7

```

1 % Funzione di Runge
2 f = @(x) 1 ./ (1 + 25.*x.^2);
3
4 a = -6;
5 b = 6;
6
7 n = 2:2:40;
8
9 % valori nei quali mi interessa sapere il valore del polinomio interpolante
10 x = linspace(-6,6);
11
12 for i = 1:length(n)
13     % Ascisse di Chebyshev
14     xi = ceby(n(i),a,b);
15
16     % Calcolo le fi nella funzione di Runge
17     fi = f(xi);
18
19     % Lagrange
20     y = lagrange(xi,fi,x);
21
22     norm(f(x) - y)
23 end

```

La seguente figura mostra il polinomio interpolante le ascisse di Chebyshev per la funzione di Runge al variare di  $N$  grado del polinomio con  $N = 2, 4, 6, 8, \dots, 40$



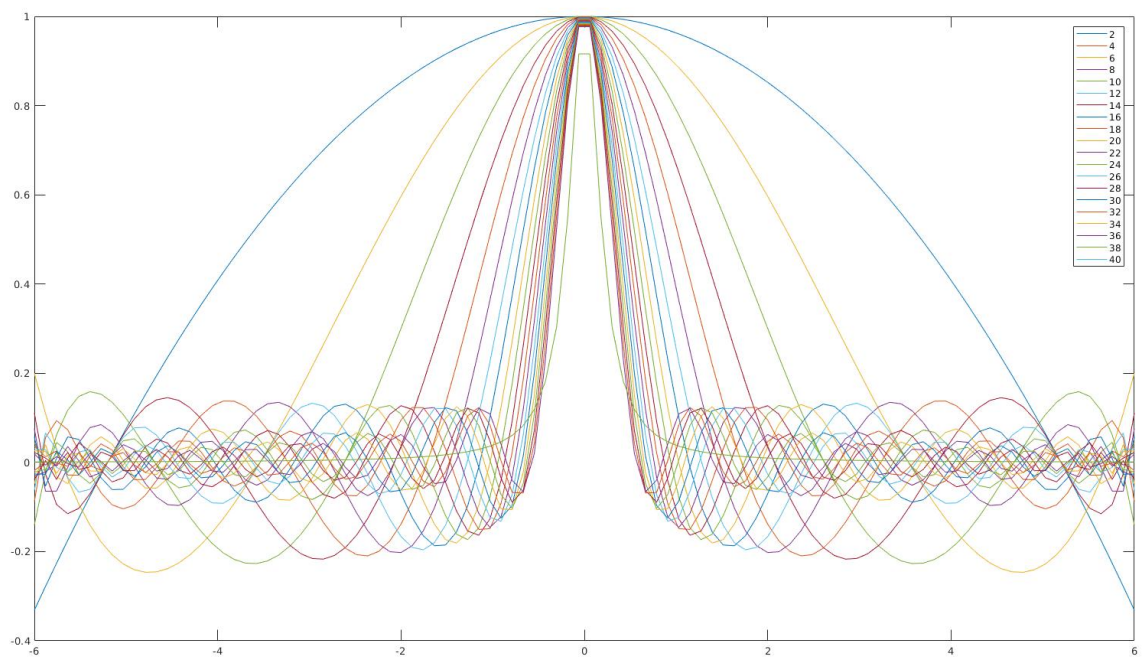


Figure 3: Polinomio di Lagrange al variare di N

La seguente tabella mostra invece la stima dell'errore al variare di N (grado del polinomio interpolante) tramite la norma euclidea della differenza tra le fi della funzione di Ruge 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

```

1 function l=lebesgue(pts)
2   rows_chebvand=length(pts);
3   V_pts=gallery('chebvand',rows_chebvand,pts);

```

```

4 M=max(5000,10*rows_chebvand);
5 pts_leb=linspace(-1,1,M); %PUNTI TEST.
6 V_leb=gallery('chebvand',rows_chebvand,pts_leb);
7 l=norm(V_pts\V_leb,1);

1 addpath('..../Es_1');
2 addpath('..../Es_6');
3 addpath('..../Es_7');
4
5 % Funzione di Runge
6 f = @(x) 1 ./ (1 + 25.*x.^2);
7
8 a = -6;
9 b = 6;
10
11 n = 2:2:40;
12
13 % valori nei quali mi interessa sapere il valore del polinomio interpolante
14 x = linspace(-6,6);
15
16 for i = 1:length(n)
17     % Ascisse di Chebyshev
18     xi = ceby(n(i),a,b);
19     lebesgue(xi)
20 end

```

Nella seguente tabella è riportato come varia la costante di Lebesgue  $\lambda$  al variare di  $N$  grado del polinomio di Lagrange.

N	$\Lambda$
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

```

1 addpath('..../Es_1');
2 addpath('..../Es_6');
3
4 % Funzione di Runge
5 f = @(x) 1 ./ (1 + 25.*x.^2);

```

```

6
7 a = -6;
8 b = 6;
9
10 n = 2:2:40;
11
12 % valori nei quali mi interessa sapere il valore del polinomio interpolante
13 x = linspace(a,b);
14
15 for i = 1:length(n)
16     % Ascisse equidistanti
17     xi = linspace(a,b,n(i));
18
19     % Calcolo le fi nella funzione di Runge
20     fi = f(xi);
21
22     % Lagrange
23     y = lagrange(xi,fi,x);
24
25     % Plot
26     plot(x,y)
27     hold on
28
29     norm(f(x) - y)
30 end
31 legend('2','4','6','8','10','12','14','16','18','20','22','24','26','28','30','32','34','36','38','40')
32 hold off

```

#### 4.10 Esercizio 10

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

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

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  $i = 0, \dots, n$ :

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

quindi  $x_i = 5$  ascisse distinte 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:

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

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

in cui la matrice dei coefficienti  $V \in \mathbb{R}^{n+1 \times m+1}$  è una matrice di tipo *Vandermonde* (in realtà la trasposta di una matrice di tipo *Vandermonde*), il vettore  $\underline{a}$ , è il vettore da determinare e definisce il polinomio di approssimazione ai *minimi quadrati*, ed infine il vettore  $\underline{y}$  è il vettore dei *valori misurati*.

Quindi scambiando le incognite con i valori di Input abbiamo che :

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

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*:

```

1 A = [ ones(10,3)];
2 j = 2;
3 for i = 3:2:length(A)-1
4     A(i,2) = j;
5     A(i+1,2) = j;
6     A(i,3) = j^2;
7     A(i+1,3) = j^2;
8     j = j+1;
9 end
10 b = [2.9; 3.1; 6.9; 7.1; 12.9; 13.1; 20.9; 21.1; 30.9; 31.1];
11
12 x = risolutoreQR(A,b);
13 r = A*x-b;
14 n = norm(r)^2;
15 format long e
16 x, r, n

```

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

$$\underline{a} = \begin{bmatrix} 1.000000000000001e+00 \\ 1.000000000000002e+00 \\ 9.99999999999994e-01 \end{bmatrix} \quad \underline{r} = \begin{bmatrix} 1.000000000000023e-01 \\ -9.999999999999787e-02 \\ 1.000000000000032e-01 \\ -9.999999999999609e-02 \\ 1.000000000000014e-01 \\ -9.999999999999787e-02 \\ 1.000000000000014e-01 \\ -1.000000000000014e-01 \\ 9.999999999999787e-02 \\ -1.000000000000050e-01 \end{bmatrix} \quad \|r\|_2^2 = 1.000000000000009e-01$$

## 5 Capitolo 5

### 5.1 Esercizio 1

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

```
1 % If = trapcomp(n, a, b, fun)
2 %   Formula dei trapezi composta per l'approssimazione dell'integrale
3 %   definito di una funzione.
4 %
5 % Input:
6 %   -n: numero di sottointervalli sui quali applicare la formula dei
7 %       trapezi semplice.
8 %   -a: estremo sinistro dell'intervallo di integrazione;
9 %   -b: estremo destro dell'intervallo di integrazione;
10 %   -fun: la funzione di cui si vuol calcolare l'integrale;
11 %
12 % Output:
13 %   -If: l'approssimazione dell'integrale definito della funzione.
14
15 function [If] = trapcomp(n, a, b, fun)
16     h = (b-a)/n;
17     If = 0;
18     for i = 1:n-1
19         If = If+fun(a+i*h);
20     end
21     If = (h/2)*(2*If + fun(a) + fun(b));
22 end
```

### 5.2 Esercizio 2

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

```
1 % If = simpcomp(n, a, b, fun)
2 %   Formula di Simpson composta per l'approssimazione dell'integrale
3 %   definito di una funzione.
4 %
5 % Input:
6 %   -n: numero, pari, di sottointervalli sui quali applicare la formula di
7 %       Simpson semplice.
8 %   -a: estremo sinistro dell'intervallo di integrazione;
9 %   -b: estremo destro dell'intervallo di integrazione;
10 %   -fun: la funzione di cui si vuol calcolare l'integrale;
11 %
12 % Output:
13 %   -If: l'approssimazione dell'integrale definito della funzione.
14
15 function [If] = simpcomp(n, a, b, fun)
16     h = (b-a)/n;
17     If = fun(a)-fun(b);
18     for i=1:n/2
19         If = If + 4*fun(a+(2*i-1)*h)+2*fun((a+2*i*h));
20     end
```

```

21     If = If*(h/3);
22 end

```

### 5.3 Esercizio 3

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

```

1  % If = trapad(a, b, fun, tol)
2  %   Formula dei trapezi adattativa per l'approssimazione dell'integrale
3  %   definito di una funzione.
4  %
5  % Input:
6  %   -a: estremo sinistro dell'intervallo di integrazione;
7  %   -b: estremo destro dell'intervallo di integrazione;
8  %   -fun: la funzione di cui si vuol calcolare l'integrale;
9  %   -tol: la tolleranza entro la quale si richiede debba rientrare la
10 %   soluzione approssimata.
11 %
12 % Output:
13 %   -If: l'approssimazione dell'integrale definito della funzione;
14
15 function [If] = trapad(a, b, fun, tol)
16     h = (b-a)/2;
17     m = (a+b)/2;
18     If1 = h*(feval(fun, a) + feval(fun, b));
19     If = If1/2 + h*feval(fun, m);
20     err = abs(If-If1)/3;
21     if err>tol
22         IfSx = trapad(a, m, fun, tol/2);
23         IfDx = trapad(m, b, fun, tol/2);
24         If = IfSx+IfDx;
25     end
26 end

```

### 5.4 Esercizio 4

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

```

1  % If = simpad(a, b, fun, tol)
2  %   Formula di Simpson adattativa per l'approssimazione dell'integrale
3  %   definito di una funzione.
4  %
5  % Input:
6  %   -a: estremo sinistro dell'intervallo di integrazione;
7  %   -b: estremo destro dell'intervallo di integrazione;
8  %   -fun: la funzione di cui si vuol calcolare l'integrale;
9  %   -tol: la tolleranza entro la quale si richiede debba rientrare la
10 %   soluzione approssimata.
11 %
12 % Output:
13 %   -If: l'approssimazione dell'integrale definito della funzione;
14

```

```

15 function [If] = simpad(a, b, fun, tol)
16     h = (b-a)/6;
17     m = (a+b)/2;
18     m1 = (a+m)/2;
19     m2 = (m+b)/2;
20     If1 = h*(feval(fun, a) + 4*feval(fun, m) + feval(fun, b));
21     If = If1/2 + h*(2*feval(fun, m1) + 2*feval(fun, m2) - feval(fun, m));
22     err = abs(If-If1)/15;
23     if err>tol
24         IfSx = trapad(a, m, fun, tol/2);
25         IfDx = trapad(m, b, fun, tol/2);
26         If = IfSx+IfDx;
27     end
28 end

```

## 5.5 Esercizio 5

## 6 Capitolo 6