

Elaborato di  
**Calcolo Numerico**  
Anno Accademico 2016/2017

Gabriele Puliti - 5300140 - *gabriele.puliti@stud.unifi.it*  
Luca Passaretta - 5436462 - *luca.passaretta@stud.unifi.it*

September 3, 2017



# Capitoli

<b>1</b>	<b>Capitolo 1</b>	<b>6</b>
1.1	Esercizio 1 . . . . .	6
1.2	Esercizio 2 . . . . .	6
1.3	Esercizio 3 . . . . .	6
1.4	Esercizio 4 . . . . .	6
1.5	Esercizio 5 . . . . .	7
1.6	Esercizio 6 . . . . .	8
1.7	Esercizio 7 . . . . .	8
1.8	Esercizio 8 . . . . .	8
1.9	Esercizio 9 . . . . .	9
1.10	Esercizio 10 . . . . .	9
1.11	Esercizio 11 . . . . .	9
1.12	Esercizio 12 . . . . .	10
1.13	Esercizio 13 . . . . .	10
<b>2</b>	<b>Capitolo 2</b>	<b>10</b>
2.1	Esercizio 1 . . . . .	10
2.2	Esercizio 2 . . . . .	10
2.3	Esercizio 3 . . . . .	11
2.4	Esercizio 4 . . . . .	11
2.5	Esercizio 5 . . . . .	12
2.6	Esercizio 6 . . . . .	12
2.7	Esercizio 7 . . . . .	13
2.8	Esercizio 8 . . . . .	14
2.9	Funzioni MatLab Usate . . . . .	14
2.9.1	Metodo Newton per $\sqrt{\alpha}$ . . . . .	14
2.9.2	Metodo Newton per $\sqrt[n]{\alpha}$ . . . . .	14
2.9.3	Metodo delle secanti per $\sqrt[n]{\alpha}$ . . . . .	14
2.9.4	Metodo di Newton Modificato . . . . .	15
2.9.5	Metodo di accelerazione di Aitken . . . . .	15
2.9.6	Metodo delle secanti . . . . .	16
2.9.7	Metodo della bisezione . . . . .	16
2.9.8	Metodo delle corde . . . . .	17
<b>3</b>	<b>Capitolo 3</b>	<b>17</b>
3.1	Esercizio 1 . . . . .	17
3.2	Esercizio 2 . . . . .	18
3.3	Esercizio 3 . . . . .	18
3.4	Esercizio 4 . . . . .	18
3.5	Esercizio 5 . . . . .	19
3.6	Esercizio 6 . . . . .	19
3.7	Esercizio 7 . . . . .	19
3.8	Esercizio 8 . . . . .	20
3.9	Esercizio 9 . . . . .	20
3.10	Esercizio 10 . . . . .	20
3.11	Esercizio 11 . . . . .	21
3.12	Esercizio 12 . . . . .	21
3.13	Esercizio 13 . . . . .	22
3.14	Esercizio 14 . . . . .	22
3.15	Esercizio 15 . . . . .	23
3.16	Esercizio 16 . . . . .	24
3.17	Esercizio 17 . . . . .	25
3.18	Esercizio 18 . . . . .	25
3.19	Esercizio 19 . . . . .	25
3.20	Esercizio 20 . . . . .	26
3.21	Esercizio 21 . . . . .	27
3.22	Funzioni MatLab Usate . . . . .	27

3.22.1	Algoritmo di fattorizzazione LU con pivoting parziale . . . . .	27
3.22.2	Risoluzione sistema lineare con funzione di ingresso già fattorizzata LU . . . . .	28
3.22.3	Risoluzione diagonale matrice LDLT di un sistema lineare . . . . .	28
3.22.4	Risoluzione di sistemi di equazioni non lineari mediante Newton . . . . .	28
<b>4</b>	<b>Capitolo 4</b>	<b>29</b>
4.1	Esercizio 1 . . . . .	29
4.2	Esercizio 2 . . . . .	29
4.3	Esercizio 3 . . . . .	30
4.4	Esercizio 4 . . . . .	30
4.5	Esercizio 5 . . . . .	31
4.6	Esercizio 6 . . . . .	32
4.7	Esercizio 7 . . . . .	32
4.8	Esercizio 8 . . . . .	32
4.9	Esercizio 9 . . . . .	32
4.10	Esercizio 10 . . . . .	32
4.11	Funzioni MatLab Usate . . . . .	32
4.11.1	Differenze divise . . . . .	32
4.11.2	Horner Generalizzato . . . . .	33
4.11.3	Funzione di valutazione . . . . .	33
4.11.4	Ascisse Equispaziate . . . . .	33
4.11.5	Chebyshev . . . . .	33
4.11.6	valutazione Spline . . . . .	34
4.11.7	Sistema sovradeterminato . . . . .	34
<b>5</b>	<b>Grafici</b>	<b>i</b>



# 1 Capitolo 1

## 1.1 Esercizio 1

Sapendo che il metodo iterativo è convergente a  $x^*$  allora per definizione si ha:

$$\lim_{k \rightarrow +\infty} x_k = x^*$$

inoltre per definizione di  $\Phi$  si calcola il limite:

$$\lim_{k \rightarrow +\infty} \Phi(x_k) = \lim_{k \rightarrow +\infty} x_{k+1} = x^*$$

infine ipotizzando che la funzione  $\Phi$  sia uniformemente continua, è possibile calcolare il limite:

$$\lim_{k \rightarrow +\infty} \Phi(x_k) = \Phi\left(\lim_{k \rightarrow +\infty} x_k\right) = \Phi(x^*)$$

dai due limiti si ha la tesi:

$$\Phi(x^*) = x^*$$

## 1.2 Esercizio 2

Dal momento che le variabili intere di 2 byte in Fortran vengono gestite in Modulo e Segno, la variabile **numero** inizializzata con:

```
integer*2 numero
```

varia tra  $-32768 \leq \text{numero} \leq 32767$  ( $-2^{15} \leq \text{numero} \leq 2^{15} - 1$ ).

Durante la terza iterazione del primo ciclo for si arriva al valore massimo rappresentabile tramite gli interi a 2 byte; alla quarta iterazione si avrà quindi la somma del **numero** in modulo e segno:

$$(32767)_{10} + (1)_{10} = (011111111111111)_{2,MS} + (000000000000001)_{2,MS} = (100000000000000)_{2,MS} = (-32768)_{10}$$

Nel secondo ciclo for, durante la quinta iterazione, al **numero** viene sottratto 1:

$$(-32768)_{10} - (1)_{10} = (100000000000000)_{2,MS} - (000000000000001)_{2,MS} = (011111111111111)_{2,MS} = (32767)_{10}$$

Da cui si spiega l'output del codice.

## 1.3 Esercizio 3

Per definizione si ha che la precisione di macchina  $u$ , per arrotondamento e' data da:

$$u = \frac{1}{2}b^{1-m}$$

Se  $b = 8, m = 5$  si ha:

$$u = \frac{1}{2} \cdot 8^{1-5} = \frac{1}{2} \cdot 8^{-4} = 1,2 \cdot 10^{-4}$$

## 1.4 Esercizio 4

Il codice seguente:

```
1 format long e;  
2  
3 h=zeros(12,1);  
4 f=zeros(12,1);  
5  
6 for i=1:12  
7     h(i)= power(10,-i);  
8 end  
9  
10 for j=1:12
```

```

11     f(j)=lim(0,v(j));
12 end
13
14 f
15
16 function p=lim(x,y)
17     p=(exp(x+y) - exp(x))/y;
18 end

```

restituisce questo risultato (assumendo che  $f(x) = e^x$  e  $x_0 = 0$ ):

h	$\Psi_h(0)$
$10^{-1}$	1.051709180756477e+00
$10^{-2}$	1.005016708416795e+00
$10^{-3}$	1.000500166708385e+00
$10^{-4}$	1.000050001667141e+00
$10^{-5}$	1.000005000006965e+00
$10^{-6}$	1.000000499962184e+00
$10^{-7}$	1.000000049433680e+00
$10^{-8}$	9.999999939225290e-01
$10^{-9}$	1.000000082740371e+00
$10^{-10}$	1.000000082740371e+00
$10^{-11}$	1.000000082740371e+00
$10^{-12}$	1.000088900582341e+00

si può notare che al diminuire del valore  $h$ , la funzione  $\Psi_h(0)$  approssima con maggior precisione il valore  $f'(0)$ , come si può vedere dal plot 1.

## 1.5 Esercizio 5

Per dimostrare le due uguaglianze è necessario sviluppare in serie di Taylor  $f(x)$  fino al secondo ordine:

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

Da cui possiamo sostituire con i valori di  $x = x + h$  e  $x = x - h$ :

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2 f''(x_0)}{2} + O(h^2)$$

$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2 f''(x_0)}{2} + O(h^2)$$

Andando a sostituire questi valori si ottiene, nel primo caso:

$$\begin{aligned}
 & \frac{f(x_0 + h) - f(x_0 - h)}{2h} = \\
 &= \frac{(f(x_0) + hf'(x_0) + \frac{h^2 f''(x_0)}{2} + O(h^2)) - (f(x_0) - hf'(x_0) + \frac{h^2 f''(x_0)}{2} + O(h^2))}{2h} = \\
 &= \frac{2hf'(x_0) + O(h^2)}{2h} = f'(x_0) + O(h^2)
 \end{aligned}$$

nel secondo caso:

$$\begin{aligned}
 & \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} = \\
 &= \frac{f(x_0) + hf'(x_0) + \frac{h^2 f''(x_0)}{2} + O(h^2) - 2f(x_0) + f(x_0) - hf'(x_0) + \frac{h^2 f''(x_0)}{2} + O(h^2)}{h^2} = \\
 &= \frac{h^2 f''(x_0) + O(h^2)}{h^2} = f''(x_0) + O(h^2)
 \end{aligned}$$

Abbiamo quindi dimostrato che:

$$\begin{aligned}
 & \frac{f(x_0 + h) - f(x_0 - h)}{2h} = f'(x_0) + O(h^2) \\
 & \frac{f(x_0 + h) - 2f(x_0) + f(x_0 - h)}{h^2} = f''(x_0) + O(h^2)
 \end{aligned}$$

## 1.6 Esercizio 6

Il codice MatLab, indicando con  $x=x_n$  e  $r=\epsilon$ :

```
1 format longEng
2 format compact
3
4 conv=sqrt(2);
5 x=[2,1.5];
6 r=[x(1)-conv,x(2)-conv];
7
8 for i= 2:7
9     x(i+1) = (x(i)*x(i-1)+2)/(x(i)+x(i-1))
10 end
11
12 for i=3:8
13     r(i)=x(i)-conv;
14 end
15
16 x
17 r
```

restituisce i valori:

n	$x_n$	$\epsilon$
0	2.000000000000000e+000	585.786437626905e-003
1	1.500000000000000e+000	85.7864376269049e-003
2	1.42857142857143e+000	14.3578661983335e-003
3	1.41463414634146e+000	420.583968367971e-006
4	1.41421568627451e+000	2.12390141496321e-006
5	1.41421356268887e+000	315.774073555986e-012
6	1.41421356237310e+000	0.00000000000000e+000
7	1.41421356237310e+000	0.00000000000000e+000

I valori indicano che per valori di  $n$  superiori a 5 l'errore, indicato con  $\epsilon$ , è dell'ordine di  $10^{-12}$ .

## 1.7 Esercizio 7

Sapendo che la rappresentazione del numero è stata fatta usando l'arrotondamento, la precisione di macchina si calcola:

$$u = \frac{b^{1-m}}{2}$$

il cui valore sappiamo essere pari a:

$$u \approx 4.66 \cdot 10^{-10}$$

dato che stiamo cercando il numero di cifre binarie allora si deve avere  $b = 2$ , è quindi possibile ricavare  $m$ :

$$m = 1 - \log_2(4.66 \cdot 10^{-10}) \approx 31.99$$

possiamo pertanto affermare che servono 32 cifre dedicate alla mantissa per rappresentare il numero con precisione macchina  $4.66 \cdot 10^{-10}$ .

## 1.8 Esercizio 8

Sapendo che la mantissa in decimale è calcolabile tramite la funzione:

- $m = 1 - \log_{10} u$  (troncamento)
- $m = 1 - \log_{10}(2 \cdot u)$  (arrotondamento)

e che la precisione di macchina assuma un valore accettabilmente piccolo in modo tale che il  $\log_{10} u \approx 1$ , cioè  $0 \leq u < 1$ , allora è possibile scrivere:

- $m = 1 - \log_{10} u \approx -\log_{10} u$  (troncamento)
- $m = 1 - \log_{10} 2u = 1 - \log_{10} 2 - \log_{10} u \approx -\log_{10} u$  (arrotondamento)



## 1.9 Esercizio 9

```
1 x=0;
2 delta = 1/10;
3 while x~=1,x=x+delta, end
```

Il valore di delta è uguale a  $\frac{1}{10}$ , la rappresentazione binaria di questo numero però non è esatta. Si tratta di una rappresentazione periodica e quindi in decimale sarà circa 0.0999. Prendendo  $delta \approx 0.9$  vedremo che per  $i = 10$   $x \approx 0.999$ , mentre per  $i = 11$  si avrà  $x \approx 1,0989$ . Per questo motivo la condizione  $x \approx 1$  non si avvererà mai e il programma non terminerà.

## 1.10 Esercizio 10

All'interno della radice può presentarsi un problema di overflow dato che la somma dei due quadrati potrebbe essere molto grande, tanto grande da poter superare il limite massimo rappresentabile dalla macchina:

$$realmax = (1 - b^{-m}) \cdot b^{b^s - \nu}$$

Per risolvere questo problema è necessario prendere il massimo valore tra le due variabili:

$$m = \max\{|x|, |y|\}$$

e moltiplicare e dividere per questo valore:

$$\sqrt{x^2 + y^2} = m \cdot \frac{\sqrt{x^2 + y^2}}{m} = m \cdot \sqrt{\frac{x^2 + y^2}{m^2}} = m \cdot \sqrt{\left(\frac{x}{m}\right)^2 + \left(\frac{y}{m}\right)^2}$$

In questo modo si eviterà il problema di overflow, il problema è ben condizionato dato che potenza e radice sono ben condizionate e grazie alla modifica proposta indicata sopra.

## 1.11 Esercizio 11

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

- $$\begin{aligned} fl(fl(fl(x) + fl(y)) + fl(z)) &= \\ &= ((x(1 + \varepsilon_x) + y(1 + \varepsilon_y))(1 + \varepsilon_a) + z(1 + \varepsilon_z))(1 + \varepsilon_b) \end{aligned}$$
- $$\begin{aligned} fl(fl(x) + fl(fl(y) + fl(z))) &= \\ &= (x(1 + \varepsilon_x) + (y(1 + \varepsilon_y) + z(1 + \varepsilon_z))(1 + \varepsilon_a))(1 + \varepsilon_b) \end{aligned}$$

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, 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:

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

- seguendo gli stessi procedimenti del punto precedente possiamo scrivere:

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

## 1.12 Esercizio 12

Sapendo che il numero di condizionamento del problema è dato da:

$$k = \left| f'_x \cdot \frac{x}{f(x)} \right|$$

Dato che la nostra funzione è  $f(x) = \sqrt{x}$  allora la derivata è data da  $f'(x) = \frac{1}{2\sqrt{x}}$ , sostituendo i valori otteniamo, come volevamo:

$$k = \left| \frac{1}{2\sqrt{x}} \cdot \frac{x}{\sqrt{x}} \right| = \left| \frac{1}{2} \right| = \frac{1}{2}$$

## 1.13 Esercizio 13

Il problema è che stiamo rappresentando dei numeri reali in un calcolatore, quindi la loro rappresentazione comporta delle approssimazioni. Nella riga 11 abbiamo calcolato e restituito in output il valore interno al logaritmo  $|3(1 - \frac{3}{4}) + 1|$  che teoricamente è zero, invece si ottiene  $2.220446049250313e - 16$ . Si può vedere che il codice MatLab:

```
1 format long;
2
3 x=linspace(2/3,2,1001);
4 y= [];
5
6 for i = 1:1001
7     y(i) = log(abs(3*(1-x(i))+1))/80 + x(i)^2 +1;
8 end
9
10 plot(x,y);
11 disp (abs(3*(1-4/3)+1))
```

calcola i valori della funzione ottenendo il grafico 2 e si può notare che l'asintoto verticale in  $x = \frac{4}{3}$  viene rappresentato come minimo della funzione  $f(x)$ .

## 2 Capitolo 2

Le funzioni usate nei codici seguenti sono in fondo al capitolo

### 2.1 Esercizio 1

```
1 x_0 = 3;
2 alpha = 3;
3 n = NewtonSqrt(alpha, x_0, 100, 0.0001);
```

La tabella delle approssimazioni è la seguente:

i	$x_i$
1	1.750000000000000e+00
2	1.732142857142857e+00
3	1.732050810014727e+00

### 2.2 Esercizio 2

La radice da approssimare in questo caso è  $\sqrt[3]{2}$  per gli  $n = 3, 4, 5$ . Il codice MatLab corrispondente è il seguente:

```
1 x_0 = 3;
2 alpha = 3;
3 disp('n=3');
4 n3 = NewtonSqrtN(3, alpha, x_0, 100, 0.0001);
```

```

5 disp('n=4');
6 n4 = NewtonSqrtN(4, alpha, x_0, 100, 0.0001);
7 disp('n=5');
8 n5 = NewtonSqrtN(5, alpha, x_0, 100, 0.0001);

```

L'output corrispondente è:

i	$n = 3$	$n = 4$	$n = 5$
1	1.631784138709347e+00	1.771797299323380e+00	1.943788863498140e+00
2	1.463411989089094e+00	1.463688102853308e+00	1.597060655491283e+00
3	1.442554125137959e+00	1.336940995805593e+00	1.369877122538772e+00
4	1.442249634601091e+00	1.316557487370408e+00	1.266284124539191e+00
5	1.442249570307411e+00	1.316074279204018e+00	1.246387399421677e+00
6	_____	1.316074012952573e+00	1.245731630753065e+00
7	_____	_____	1.245730939616284e+00

### 2.3 Esercizio 3

Per confrontare il metodo delle secanti con quello di Newton abbiamo creato il codice MatLab:

```

1 x_0 = 3;
2 alpha = x_0;
3 n= 2;
4
5 s = SecSqrtN(n, alpha, x_0, 100, 0.0001);

```

I risultati ottenuti dall'utilizzo del metodo delle secanti sono:

i	metodo di Newton	metodo delle secanti	$ newton - \sqrt{2} $	$ secanti - \sqrt{2} $
1	1.750000000000000e+00	1.736842105263158e+00	3.357864376269049e-01	3.226285428900628e-01
2	1.732142857142857e+00	1.732142857142857e+00	3.179292947697618e-01	3.179292947697618e-01
3	1.732050810014727e+00	1.732050934706042e+00	3.178372476416318e-01	3.178373723329468e-01
4	_____	1.732050807572255e+00	_____	3.178372451991598e-01

Si nota dalla tabella che  $|newton - \sqrt{2}| \approx |secanti - \sqrt{2}|$ , cioè che l'ordine di grandezza dell'errore assoluto tra i due metodi è lo stesso. Si può quindi affermare che l'uso del metodo di Newton o del metodo delle secanti è equivalente.

### 2.4 Esercizio 4

```

1 funct = @(x) (x-pi)^10;
2 dfunct = @(x) 10*(x-pi)^9;
3 disp('newton modificato funct 1');
4 NM11 = NewtonMod(funct, dfunct, 1, 3, 50, 1, 0)
5 NM12 = NewtonMod(funct, dfunct, 1, 3, 50, 0.1, 0)
6 NM13 = NewtonMod(funct, dfunct, 1, 3, 50, 0.01, 0)
7 NM14 = NewtonMod(funct, dfunct, 1, 3, 50, 0.001, 0)
8 NM15 = NewtonMod(funct, dfunct, 1, 3, 50, 0.0001, 0)
9 NM16 = NewtonMod(funct, dfunct, 1, 3, 50, 0.00001, 0)
10 NM17 = NewtonMod(funct, dfunct, 1, 3, 50, 0.000001, 0)
11 disp('aitken funct 1');
12 A11 = Aitken(funct, dfunct, 3, 50, 1)
13 A12 = Aitken(funct, dfunct, 3, 50, 0.1)
14 A13 = Aitken(funct, dfunct, 3, 50, 0.01)
15 funct2 = @(x) ((x-pi)^10)*(exp(1)^(2*x));
16 dfunct2 = @(x) (5+x-pi)*(x-pi)^9*2*exp(1)^(2*x);
17 disp('newton modificato funct 2');
18 NM21 = NewtonMod(funct2, dfunct2, 1, 3, 50, 1, 0)
19 NM22 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.1, 0)
20 NM23 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.01, 0)
21 NM24 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.001, 0)

```

```

22 NM25 = NewtonMod(func2, dfunc2, 1, 3, 50, 0.0001, 0)
23 NM26 = NewtonMod(func2, dfunc2, 1, 3, 50, 0.00001, 0)
24 NM27 = NewtonMod(func2, dfunc2, 1, 3, 50, 0.000001, 0)
25 disp('aitken func2 2');
26 A21 = Aitken(func2, dfunc2, 3, 50, 1)
27 A22 = Aitken(func2, dfunc2, 3, 50, 0.1)
28 A23 = Aitken(func2, dfunc2, 3, 50, 0.01)

```

Questo codice esegue i metodi di Newton, Newton modificato e Aitken per le funzioni date. L'output della  $f_1(x)$  rispetto alle diverse tolleranze sono:

tolx	Newton modificato	Aitken
1	3.014159265358979	3.141592653589755
0.1	3.014159265358979	3.141592653589789
0.01	3.057983607571556	3.141592653589789
0.001	3.133359078021984	_____
0.0001	3.140781835025782	_____
0.00001	3.140862916882183	_____
0.000001	3.140862916882183	_____

L'output della  $f_2(x)$  rispetto alle diverse tolleranze è:

tolx	Newton modificato	Aitken
1	3.014571920744193	3.137995613065127
0.1	3.014571920744193	3.141590324813442
0.01	3.059075504146139	3.141590324813442
0.001	3.132710330277176	_____
0.0001	3.140719503754643	_____
0.00001	3.140885428300940	_____
0.000001	3.140885428300940	_____

## 2.5 Esercizio 5

Il metodo di bisezione è applicabile in  $f$  se è:

1. continua nell'intervallo  $[a, b]$
2.  $f(a)f(b) < 0$

il metodo di bisezione non è possibile utilizzarlo a causa della seconda condizione dato che  $f_1(x) = (x - \pi)^{10}$  e  $f_2(x) = e^{2x}(x - \pi)^{10}$  sono sempre positive quindi non è possibile stabilire un intervallo  $[a, b]$  tale che  $f(a)f(b) < 0$ ,  $\forall a, b \in \mathbb{R}$

## 2.6 Esercizio 6

```

1 myf = @(x) (1-x-(1+cos(10*x))/2)*sin(x);
2 df = @(x) (5*sin(x)*sin(10*x)-cos(x)*(cos(10*x)/2 + 1)-1);
3
4 x0 = 0;
5 x1 = 1;
6 tol = logspace(-1,-10,10);
7 disp(tol);
8 imax = 50;
9 count = zeros(3,10);
10 for i=1:10
11     [temp, count(1,i)] = NewtonMod(myf,df,1,x0,imax,tol(i), 0);
12     [temp, count(2,i)] = secanti(myf,x0,x1,tol(i),imax);
13     [temp, count(3,i)] = corde(myf,feval(df,x0),x0,tol(i),imax);
14 end

```

```

15
16 disp('nella prima riga ci sono le iterazioni di newton modificato,');
17 disp('nella seconda riga ci sono il numero di iterazioni del metodo delle secanti');
18 disp('nella terza riga ci sono il numero di iterazioni del metodo delle corde');
19 count

```

Il numero di iterazioni effettuate all'interno dei 3 algoritmi, vengono salvate nella variabile count che restituisce i valori sotto forma di tabella:

$tol_x$	Newton	Secanti	Tangenti
$10^{-1}$	3	13	2
$10^{-2}$	4	15	8
$10^{-3}$	4	15	15
$10^{-4}$	4	16	22
$10^{-5}$	5	16	28
$10^{-6}$	5	16	35
$10^{-7}$	5	17	42
$10^{-8}$	5	17	48
$10^{-9}$	5	17	50
$10^{-10}$	6	17	50

## 2.7 Esercizio 7

```

1 format long
2
3 myf = @(x) (1-x-(1+cos(10*x)/2)*sin(x));
4 df = @(x) (5*sin(x)*sin(10*x)-cos(x)*(cos(10*x)/2 + 1)-1);
5
6 x0 = 0;
7 x1 = 1;
8 tol = logspace(-1,-10,10);
9 imax = 50;
10 count = zeros(1,10);
11 for i=1:10
12     [temp, count(1,i)] = bisection(myf,x0,x1,tol(i),imax);
13     disp('B');
14     i
15     temp
16 end
17
18 disp('nella riga ci sono il numero di iterazioni del metodo della bisezione');
19 count

```

Il numero di iterazioni del metodo di bisezione risultanti sono:

$tol_x$	bisezione
$10^{-1}$	0
$10^{-2}$	7
$10^{-3}$	10
$10^{-4}$	13
$10^{-5}$	16
$10^{-6}$	20
$10^{-7}$	21
$10^{-8}$	26
$10^{-9}$	30
$10^{-10}$	32

## 2.8 Esercizio 8

```
1 myfunct = @(x) (x-pi)^(10*x);
2 dfdx = @(x) 10*((x-pi)^(10*x))*(x/(x-pi)+log(x-pi));
3 x0 = 0;
4
5 y = NewtonMod(myfunct,dfdx, 1, x0, 5, 0.01, 1);
```

Il codice Matlab soprastante restituirà in output un valore  $\notin \mathbb{R}$ , più precisamente il valore è pari a  $-0.010239076641281 + 0.028100085752476i$  che  $\in \mathbb{C}$ . Questo significa che il metodo non potrà convergere partendo dal punto iniziale  $x_0$ .

## 2.9 Funzioni MatLab Usate

### 2.9.1 Metodo Newton per $\sqrt{\alpha}$

```
1 function y = NetwtonSqrt(alpha, x0, imax, tol)
2     format long e
3     x = (x0 + alpha/x0)/2;
4     i = 1;
5     while(i < imax) && (abs(x-x0)>tol)
6         x0=x;
7         i = i+1;
8         x = (x0 + alpha/x0)/2;
9         disp(x);
10    end
11    y = x;
12
13 end
```

### 2.9.2 Metodo Newton per $\sqrt[n]{\alpha}$

```
1 function y = NetwtonSqrtN(n, alpha, x0, imax, tol)
2     format long e
3     x = (((n-1)*x0^n + alpha)/ x0^(n-1)) / n;
4     i = 1;
5     while(i < imax) && (abs(x-x0)>tol)
6         x0=x;
7         i = i+1;
8         x = (((n-1)*x0^n + alpha)/ x0^(n-1)) / n;
9         disp(x);
10    end
11    y = x;
12 end
```

### 2.9.3 Metodo delle secanti per $\sqrt[n]{\alpha}$

```
1 function y = SecSqrtN(n, alpha, x0, imax, tol)
2     format long e
3     x1 = (x0 + alpha/x0)/2;
4     x = (f(x1,n, alpha) * x0 - f(x0,n, alpha)*x1) / (f(x1, n, alpha) - f(x0, n, alpha));
5     i = 1;
6     while(i < imax) && (abs(x-x0)>tol)
7         x0=x1;
8         x1=x;
9         i = i+1;
```

```

10         x = (f(x1,n, alpha) * x0 - f(x0,n, alpha)*x1 ) / (f(x1, n, alpha) - f(x0, n,
11             alpha));
12         disp(x)
13     end
14     y = x;
15 end
16
17 function y = f(x, n, alpha)
18     y = (x ^ n) - alpha;
19 end

```

#### 2.9.4 Metodo di Newton Modificato

```

1 function [y, i] = NewtonMod(f, df, m, x0, imax, tol, output)
2     format long;
3     i = 0;
4     x = x0;
5     vai=1;
6     while((i < imax) && vai)
7         i = i+1;
8         fx = feval(f, x0);
9         dfx = feval(df, x0);
10        if(dfx ~= 0)
11            x = x0 - m * fx / dfx;
12        else
13            break;
14        end
15        if(abs(x-x0)<tol)
16            vai = 0;
17        end
18        x0 = x;
19    end
20    if(output)
21        if(vai)
22            disp(['Errore] Tolleranza non calcolabile']);
23        else
24            disp(i);
25        end
26    end
27    y = x;
28 end

```

#### 2.9.5 Metodo di accelerazione di Aitken

```

1 function y = Aitken( f, df, x0, imax, tol )
2     format long;
3     i = 0;
4     vai=1;
5
6     while((i < imax) && vai)
7         x1 = NewtonMod(f, df, 1, x0, 1, tol, 0);
8         x2 = NewtonMod(f, df, 1, x1, 1, tol, 0);
9         i = i+1;
10        if((x0 - 2*x1 +x2) == 0)
11            disp(['Errore] Divisione per 0']);

```

```

12         vai = 0;
13         break;
14     end
15     x = (x2*x0 - x1^2)/(x0 - 2*x1 + x2);
16     if(abs(x-x0)<tol)
17         vai = 0;
18     end
19     x0 = x;
20 end
21 if(vai)
22     disp('[Errore] Tolleranza non calcolabile');
23     disp(i);
24 end
25 y = x;
26 end

```

### 2.9.6 Metodo delle secanti

```

1 function [x,nit,tolf]=secanti(f,x0,x1,tolx,maxit)
2     nit=0;
3     fx0=feval(f,x0);
4     err=abs(x1-x0);
5     while (nit<maxit & err>tolx)
6         fx1=feval(f,x1);
7         dfx1=(fx1-fx0)/(x1-x0);
8         tolf=tolx*abs(dfx1);
9         if abs(fx1)<=tolf
10             break
11         end
12         x2=x1-(fx1/dfx1);
13         err=abs(x2-x1);
14         x0=x1;
15         x1=x2;
16         fx0=fx1;
17         nit=nit+1;
18     end
19     x=x1;
20 end

```

### 2.9.7 Metodo della bisezione

```

1 function [sol,nit]=bisect(f,a,b,tolx,maxit);
2     nit=maxit;
3     j=0;
4     if(subs(f,a)*subs(f,b)>0)
5         disp('[Warning] il metodo non puo'' essere usato');
6     else
7         while(j<=maxit)
8             sol=(a+b)/2;
9             ff=subs(f,sol);
10            if(abs(ff)<=tolx)
11                nit=j;
12                break;
13            end
14            fa=subs(f,a);
15            fm=subs(f,sol);

```



```

16         if (fa*fm<=0)
17             b=sol;
18         else
19             a=sol;
20         end
21         j=j+1;
22     end
23 end

```

### 2.9.8 Metodo delle corde

```

1 function [x,nit]=corde(f,m,x0,tolx,maxit)
2     nit=0;
3     err=tolx+1;
4     x=x0;
5     while (nit<maxit && err>tolx)
6         fx=feval(f,x);
7         tolf=tolx*abs(m);
8         if abs(fx)<=tolf
9             break
10        end
11        x1=x-fx/m;
12        err=abs(x1-x);
13        x=x1;
14        nit=nit+1;
15    end
16 end

```

## 3 Capitolo 3

Le funzioni usate nei codici seguenti sono in fondo al capitolo

### 3.1 Esercizio 1

Una matrice  $L \in M_{n \times n}$  è definita triangolare inferiore se preso  $l_{i,j} \in L$  vale la proprietà:

$$l_{i,j} = 0 \quad \forall i < j \quad i, j \in [1, \dots, n]$$

Possiamo dimostrare facilmente che la somma di due matrici triangolari inferiori è ancora una matrice triangolare inferiore. Prendiamo due matrici  $L, K \in M_{n \times n}$  per definizione di triangolare inferiore si deve valere che:

$$l_{i,j} + k_{i,j} = 0 + 0 = 0 \quad \forall i < j \quad i, j \in [1, \dots, n]$$

che è la definizione di matrice triangolare inferiore, come volevasi dimostrare.

Dimostriamo ora che il prodotto di due matrici triangolari inferiori è ancora una matrice triangolare inferiore. Indichiamo con  $A \in M_{n \times n}$  la matrice risultante del prodotto delle 2 matrici  $L$  e  $K$ , gli elementi della nuova matrice,  $a_{i,j} \in A$ , sono calcolati come la somma del prodotto degli elementi delle due matrici:

$$a_{i,j} = \sum_{m=1}^n l_{i,m} \cdot k_{m,j} \quad \forall i, j \in [1, \dots, n].$$

questa somma può essere scritta anche:

$$\sum_{m=1}^n l_{i,m} \cdot k_{m,j} = \underbrace{\sum_{i < j} l_{i,m} \cdot k_{m,j}}_0 + \sum_{i \geq j} l_{i,m} \cdot k_{m,j}$$

da quest'ultima il valore  $0 = \sum_{i < j} l_{i,m} \cdot k_{m,j} = a_{i,j} \quad i, j \in [1, \dots, n]$  che è la definizione di matrice triangolare inferiore, come volevasi dimostrare.

(Allo stesso modo si può dimostrare per matrici triangolari superiori).

### 3.2 Esercizio 2

Una matrice triangolare inferiore  $L \in M_{n \times n}$  è detta a diagonale unitaria se i suoi elementi sulla diagonale sono pari a 1:

$$l_{i,i} = 1 \quad \forall i \in [1, \dots, n]$$

Prendiamo una seconda matrice  $K \in M_{n \times n}$  triangolare inferiore a diagonale unitaria, calcoliamo il prodotto tra  $K$  e  $L$ :

$$\sum_{m=1}^n l_{i,m} \cdot k_{m,j} = \underbrace{\sum_{i < j} l_{i,m} \cdot k_{m,j}}_0 + \underbrace{\sum_{i=j} l_{i,m} \cdot k_{m,i}}_1 + \sum_{i > j} l_{i,m} \cdot k_{m,j}$$

La risultante matrice assume valori:

- $\sum_{i < j} l_{i,m} \cdot k_{m,j} = 0 \quad \forall i, j \in [1, \dots, n]$
- $\sum_{i=j} l_{i,m} \cdot k_{m,j} = 1 \quad \forall i, j \in [1, \dots, n]$
- $\sum_{i > j} l_{i,m} \cdot k_{m,j} \in \mathbb{R} \quad \forall i, j \in [1, \dots, n]$

che non è altro che la definizione di matrice triangolare inferiore a diagonale unitaria, come volevamo dimostrare.

(Allo stesso modo si può dimostrare per matrici triangolari superiori).

### 3.3 Esercizio 3

Indichiamo con  $A \in M_{n \times n}$  una matrice triangolare inferiore con elementi sulla diagonale non nulli, tale matrice può essere scritta come:

$$A = D(I_n + U)$$

in cui  $D$  è una matrice diagonale dove  $\text{diag}(D) = \text{diag}(A)$ , la matrice  $I_n$  è la matrice identità e  $U$  è una matrice strettamente triangolare inferiore, cioè con diagonale nulla, e gli unici elementi non nulli sono gli stessi elementi della matrice  $A$ . Una matrice strettamente triangolare inferiore è anche una matrice nilpotente, questo significa che  $\exists n \in \mathbb{R}$  tale che  $U^n = 0_{n \times n}$ . Dobbiamo quindi dimostrare che  $A^{-1}$  è ancora una matrice triangolare inferiore, se  $A^{-1}$  è l'inversa  $A$  deve valere:

$$A \cdot A^{-1} = D(I_n + U) \cdot A^{-1} = I_n$$

$$A^{-1} = (I_n + U)^{-1} \cdot D^{-1}$$

Sappiamo che l'inversa di una matrice diagonale è ancora una matrice diagonale, quindi  $D^{-1}$  è diagonale. Per scoprire che tipo di matrice è  $(I_n + U)^{-1}$  è necessario sviluppare in serie:

$$(I_n + U)^{-1} = I_n - U + U^2 - \dots + (-1)^{n-1} U^{n-1}$$

che sono somme di matrici strettamente triangolari inferiori, questo implica che  $(I_n + U)^{-1}$  è di quel tipo. Abbiamo quindi dimostrato che anche  $A^{-1}$  è una matrice triangolare inferiore.

Nel caso in cui  $A$  sia una matrice triangolare inferiore a diagonale unitaria la dimostrazione non varia dato che gli elementi dell'inversa di  $D$  rimangono unitari nel processo di inversione.

(Allo stesso modo si può dimostrare per matrici triangolari superiori).

### 3.4 Esercizio 4

L'eliminazione nella prima colonna richiede  $n$  somme ed  $n$  prodotti per  $n - 1$  righe, quindi in totale  $(n + n)(n - 1) = 2n(n - 1)$  flops. L'eliminazione della seconda richiede  $n - 1$  somme ed  $n - 1$  prodotti per  $n - 2$  righe, quindi in totale  $[(n - 1) + (n - 1)](n - 2) = 2(n - 1)(n - 2)$  flops.

Procedendo per induzione si ha che il numero totale di operazioni é

$$\sum_{i=0}^n 2(n - i)(n - i + 1).$$

Operando la sostituzione  $j = n - i + 1$  si ha che la somma diviene :

$$2 \sum_{j=n+1}^1 j(j-1) = 2 \left( \sum_{j=1}^{n+1} j^2 + \sum_{j=1}^{n+1} j \right) = 2 \left( \sum_{j=1}^{n-1} j^2 + n^2 + (n+1)^2 + \sum_{j=1}^{n-1} j + n + n + 1 \right) =$$

$$= 2 \left( \frac{n \cdot (n-1) \cdot (2n-2)}{6} + \frac{n \cdot (n-1)}{2} + 2n^2 + 3n + 2 \right) = 2 \left( \frac{2n^3 - 4n^2 + 2n}{6} + \frac{n^2 - n}{2} + 2n^2 + 3n + 2 \right) \leq \frac{2}{3} \cdot n^3$$

quindi si ha che il numero di flop è  $\frac{2}{3} \cdot n^3$ , come volevamo dimostrare

### 3.5 Esercizio 5

L'algoritmo di fattorizzazione LU con pivoting parziale da noi implementato è il seguente:

```

1 function [L,U,P]=factLUP(A)
2 [m,n]=size(A);
3 if m~=n
4     error('[Errore] La matrice inserita A inserita non e'' quadrata');
5 end
6 L=eye(n);
7 P=eye(n);
8 U=A;
9 for k=1:n
10     [pivot, m]=max(abs(U(k:n,k)));
11     if pivot==0
12         error('[Attenzione] La matrice e'' singolare');
13     end
14     m=m+k-1;
15     if m~=k
16         U([k,m], :) = U([m, k], :);
17         P([k,m], :) = P([m, k], :);
18         if k >= 2
19             L([k,m], 1:k-1) = L([m,k], 1:k-1);
20         end
21     end
22     L(k+1:n,k)=U(k+1:n,k)/U(k,k);
23     U(k+1:n,:)=U(k+1:n,:)-L(k+1:n,k)*U(k,:);
24 end

```

### 3.6 Esercizio 6

Consideriamo la Matrice di ingresso  $A$  già fattorizzata LU con pivoting parziale:

```

1 function [b] = linLUP(L,U, P, b)
2     b = triInf(L,P*b);
3     b = triSup(U,b);
4 end

```

### 3.7 Esercizio 7

Per essere SDP una matrice  $A \in \mathbb{R}^{n \times n}$  deve sottostare a due proprietà:

- deve essere simmetrica, cioè  $A = A^T$ ;
- $\forall x \in \mathbb{R}^n$  tale che  $x \neq 0$  vale  $x^T A x > 0$

La matrice  $A$  essendo non singolare è anche SDP. Le matrici  $AA^T$  e  $A^T A$  per essere SDP devono dimostrare le proprietà sopra:

- simmetriche:

$$(AA^T)^T = (A^T)^T A^T = AA^T$$

$$(A^T A)^T = A^T (A^T)^T = A^T A.$$

- definite positive:

$$x^T AA^T x = xx^T AA^T xx^T = x(A^T x)^T (x^T A)^T x^T = \underbrace{(x^T A^T x)}_{>0} \cdot \underbrace{(x^T Ax)}_{>0} > 0$$

$$x^T A^T Ax = xx^T A^T Axx^T = x(Ax)^T (x^T A^T)^T x^T = \underbrace{(x^T Ax)}_{>0} \cdot \underbrace{(x^T A^T x)}_{>0} > 0$$

possiamo quindi affermare che le matrici  $AA^T$  e  $A^T A$  sono SDP.

### 3.8 Esercizio 8

Se la matrice  $A$  ha rango massimo significa che la matrice è invertibile, di conseguenza il suo determinante è non nullo che implica che la matrice è nonsingolare. Dalla dimostrazione dell'esercizio precedente (**Esercizio 7**) si ha quindi che se la matrice ha rango massimo allora  $A^T A$  è SDP.

### 3.9 Esercizio 9

La matrice  $A \in \mathbf{R}^{n \times n}$  può essere scritta come:

$$A = \frac{A}{2} + \frac{A}{2} + \frac{A^T}{2} - \frac{A^T}{2} = \frac{A + A^T}{2} + \frac{A - A^T}{2} \equiv A_s + A_a$$

si ha quindi che  $A_s \equiv \frac{1}{2} \cdot (A + A^T)$  e  $A_a \equiv \frac{1}{2} \cdot (A - A^T)$ .

Possiamo inoltre dimostrare che preso un  $x \in \mathbf{R}^n$  risulta:

$$\begin{aligned} x^T Ax &= x^T (A_s + A_a)x = x^T A_s x + x^T A_a x = x^T A_s x + x^T \frac{(A - A^T)}{2} x = \\ &= x^T A_s x + \underbrace{\frac{1}{2}(x^T Ax - x^T A^T x)}_{=0} = x^T A_s x \end{aligned}$$

il termine  $\frac{1}{2}(x^T Ax - x^T A^T x) = \frac{1}{2}(x^T Ax - (Ax)^T x)$  è pari a zero e possiamo vederlo tramite la sostituzione  $y = Ax$ :

$$x^T Ax - (Ax)^T x \underbrace{=}_{y=Ax} x^T y - y^T x = 0$$

dato che  $x^T y = y^T x$  allora la loro differenza non può essere altro che zero.

### 3.10 Esercizio 10

Prendiamo  $i \in [1, \dots, n]$  colonne della matrice, possiamo vedere che l'algoritmo esegue  $i - 1$  somme di 2 prodotti quindi  $2(i - 1)$  e in più esegue un'operazione di sottrazione e una di divisione che equivale a 2 flop. Queste operazioni vengono eseguite per  $n - i$  volte, cioè per ogni colonna della matrice, il che significa che il numero di flop sono:

$$\begin{aligned} \sum_{i=1}^n 2(i - 1)(n - i) &= 2 \sum_{i=1}^n (i \cdot n - i^2 - n) = 2(n \sum_{i=1}^n i - \sum_{i=1}^n i^2 - n^2) = \\ &= 2\left(n \frac{n(n - 1)}{2} + n^2 - \frac{n(n - 1)(2n - 1)}{6} - n^2\right) = 2\left(\frac{n^3}{2} - \frac{n^2}{2} - \frac{2n^3}{6} + \frac{n^2}{6} + \frac{2n^2}{6} - \frac{n}{6}\right) = \\ &= n^3 - n^2 - \frac{2n^3}{3} + \frac{n^2}{3} + \frac{2n^2}{3} - \frac{n}{3} \approx \frac{n^3}{3} \end{aligned}$$

quindi l'algoritmo di fattorizzazione  $LDL^T$  ha un costo di  $\frac{n^3}{3} flop$ .

### 3.11 Esercizio 11

L'algoritmo che abbiamo scritto è il seguente:

```
1 function [L,D] = factLDLT(A)
2     [m,n]=size(A);
3     if m~=n
4         error('[Errore] La matrice non e'' quadrata')
5     end
6     if A(1,1)<=0
7         error('[Errore] La matrice non e'' SDP')
8     end
9     A(2:n,1)=A(2:n,1)/A(1,1);
10    for j=2:n
11        v = (A(j,1:(j-1)))'.*diag(A(1:(j-1),1:(j-1)));
12        A(j,j) = A(j,j)-A(j,1:(j-1))*v;
13        if A(j,j)<=0
14            error('[Errore] La matrice non e'' SDP');
15        end
16        A((j+1):n,j)=(A((j+1):n,j)-A((j+1):n,1:(j-1))*v)/A(j,j);
17    end
18    if nargin==1
19        L=A;
20    else
21        for j=1:n
22            for i=1:n
23                if i==j
24                    D(i,j) = A(i,j);
25                    L(i,j) = 1;
26                end
27                if i>j
28                    D(i,j) = 0;
29                    L(i,j) = A(i,j);
30                end
31                if i<j
32                    D(i,j) = 0;
33                    L(i,j) = 0;
34                end
35            end
36        end
37    end
38 end
```

### 3.12 Esercizio 12

Il codice MatLab corrispondente alla risoluzione di un sistema lineare con matrice di ingresso fattorizzata LDLT è il seguente:

```
1 function [x] = linLDL(L,D, x)
2     x = triInf(L,x');
3     x = linDiag(diag(D),x');
4     x = triSup(L',x');
5 end
```

### 3.13 Esercizio 13

Per verificarlo abbiamo usato il seguente codice MatLab:

```
1 A1 = [1,1,1,1;1,2,2,2;1,2,3,3;1,2,3,4]
2 [L1,D1] = factLDLT(A1)
3 A2 = [1,1,1,1;1,2,2,2;1,2,3,3;1,2,3,2]
4 [L2,D2] = factLDLT(A2)
```

Che restituisce l'output:

```
1 A1 =
2     1     1     1     1
3     1     2     2     2
4     1     2     3     3
5     1     2     3     4
6 L1 =
7     1     0     0     0
8     1     1     0     0
9     1     1     1     0
10    1     1     1     1
11 D1 =
12     1     0     0     0
13     0     1     0     0
14     0     0     1     0
15     0     0     0     1
16 A2 =
17     1     1     1     1
18     1     2     2     2
19     1     2     3     3
20     1     2     3     2
21 Error using factLDLT (line 14)
22 [Errore] La matrice non e' SDP
23 Error in es13 (line 6)
24 [L2,D2] = factLDLT(A2)
```

L'output è molto chiaro, la seconda matrice  $A_2$  non può essere fattorizzata  $LDL^T$  di conseguenza non è  $SDP$ .

### 3.14 Esercizio 14

In entrambi i casi abbiamo usato la matrice  $A \in M^{3 \times 3}$  con elementi:

$$A = \begin{pmatrix} 15 & -3 & 2 \\ -4 & 9 & 2 \\ 6 & 0 & 10 \end{pmatrix}$$

e il vettore dei termini noti  $b \in \mathbb{R}^3$  con valori:

$$b = (3.2, 2.3, 3.1)^T$$

Usando il codice MatLab sottostante è possibile risolvere questi 2 esempi:

```
1 format longEng
2 format compact
3
4 % 3.5 3.6
5 A=[15,-3,2;-4,9,2;6,0,10]
6 [L,U,P] = factLUP(A)
7 b = [3.2,2.3,3.1]';
8 [x] = linLUP(L,U,P,b)
9 r=A*x -b
10
11
```

```

12 % 3.11 3.12
13 A=[15,-3,2;-4,9,2;6,0,10]
14 [L,D] = factLDLT(A)
15 b = [3.2,2.3,3.1]';
16 [x] = linLUP(L,D*L',eye(3),b)
17 r=A*x -b

```

Il codice sopra restituisce l'output:

```

1 A =
2   15.00000000000000e+000   -3.00000000000000e+000    2.00000000000000e+000
3   -4.00000000000000e+000    9.00000000000000e+000    2.00000000000000e+000
4    6.00000000000000e+000    0.00000000000000e+000   10.00000000000000e+000
5 L =
6    1.00000000000000e+000    0.00000000000000e+000    0.00000000000000e+000
7   -266.666666666667e-003    1.00000000000000e+000    0.00000000000000e+000
8   400.000000000000e-003   146.341463414634e-003    1.00000000000000e+000
9 U =
10   15.00000000000000e+000   -3.00000000000000e+000    2.00000000000000e+000
11    0.00000000000000e+000    8.20000000000000e+000    2.53333333333333e+000
12    0.00000000000000e+000    0.00000000000000e+000    8.82926829268293e+000
13 P =
14    1.00000000000000e+000    0.00000000000000e+000    0.00000000000000e+000
15    0.00000000000000e+000    1.00000000000000e+000    0.00000000000000e+000
16    0.00000000000000e+000    0.00000000000000e+000    1.00000000000000e+000
17 x =
18   260.220994475138e-003
19   337.016574585635e-003
20   153.867403314917e-003
21 r =
22   888.178419700125e-018
23    0.00000000000000e+000
24   444.089209850063e-018
25 A =
26   15.00000000000000e+000   -3.00000000000000e+000    2.00000000000000e+000
27   -4.00000000000000e+000    9.00000000000000e+000    2.00000000000000e+000
28    6.00000000000000e+000    0.00000000000000e+000   10.00000000000000e+000
29 L =
30    1.00000000000000e+000    0.00000000000000e+000    0.00000000000000e+000
31   -266.666666666667e-003    1.00000000000000e+000    0.00000000000000e+000
32   400.000000000000e-003   201.680672268908e-003    1.00000000000000e+000
33 D =
34   15.00000000000000e+000    0.00000000000000e+000    0.00000000000000e+000
35    0.00000000000000e+000    7.93333333333333e+000    0.00000000000000e+000
36    0.00000000000000e+000    0.00000000000000e+000    7.27731092436975e+000
37 x =
38   245.496535796767e-003
39   364.665127020785e-003
40   162.702078521940e-003
41 r =
42   -286.143187066974e-003
43    325.404157043880e-003
44   -444.089209850063e-018

```

### 3.15 Esercizio 15

```

1 v = [1,1,1,1,1,1,1,1,1,1];
2 A = (diag(v*(-100),-1)+eye(10));
3
4 norm(A,1)
5 norm(A,inf)
6

```

```
7 cond(A,1)
```

Il codice MatLab sopra restituisce in output:

```
1 ans =
2   101.000000000000e+000
3 ans =
4   101.000000000000e+000
5 Warning: Matrix is close to singular or badly scaled.
6 Results may be inaccurate. RCOND = 9.801980e-21.
7 > In cond (line 46)
8   In es15 (line 7)
9 ans =
10  102.020202020202e+018
```

Analiticamente abbiamo che  $\|A\|_{\infty} = \|A\|_1 = 101$ , come é possibile verificare attraverso l'istruzione `norm` di Matlab. Per quanto riguarda il calcolo del numero di condizionamento  $k_{\infty}(A)$  abbiamo che, andando a calcolare l'inversa, gli elementi della matrice crescono di 2 ordini di grandezza per ogni riga, arrivando ad avere valori prossimi a  $10^{18}$ . Questo implica che  $\|A^{-1}\|_{\infty} > 10^{20}$  per cui possiamo affermare che il problema é malcondizionato. La verifica con l'istruzione `cond` di Matlab restituisce un warning in cui avverte che  $RCOND = 9.801980e - 21$ , confermando quanto appena detto.

### 3.16 Esercizio 16

```
1 v = [1,1,1,1,1,1,1,1,1];
2 A = (diag(v*(-100),-1)+eye(10));
3
4 b = [1 -99*ones(1,9)]';
5 c = 0.1*[1 -99*ones(1,9)]';
6 x = ones(10,1);
7 y = 0.1*x;
8
9 r =A*x -b;
10 r = A*y -c;
11
12 xx(1)=b(1);
13 for i=2:10
14     xx(i)=b(i)+100*xx(i-1);
15 end
16 xx=xx(:)
17 yy(1)=c(1);
18
19 for i=2:10
20     yy(i)=c(i)+100*yy(i-1)
21 end
22 yy=yy(:)
```

I risultati ottenuti sono questi:

```
1 xx =
2   1.0000e+000
3   1.0000e+000
4   1.0000e+000
5   1.0000e+000
6   1.0000e+000
7   1.0000e+000
8   1.0000e+000
9   1.0000e+000
10  1.0000e+000
11  1.0000e+000
12 yy =
13  100.0000e-003
```



```

14 100.0000e-003
15 100.0000e-003
16 100.0000e-003
17 100.0000e-003
18 100.0000e-003
19 99.9964e-003
20 99.6411e-003
21 64.1140e-003
22 -3.4886e+000

```

### 3.17 Esercizio 17

```

1 function A = factQRH(A)
2     [m,n] = size(A);
3     for i=1:n
4         alpha = norm(A(i:m, i));
5         if alpha==0
6             error('[Errore] La matrice A non ha rango massimo')
7         end
8         if A(i,i)>=0
9             alpha = -alpha;
10        end
11        v = A(i,i) - alpha;
12        A(i,i) = alpha;
13        A(i+1:m,i) = A(i+1:m,i)/v;
14        beta = -v/alpha;
15        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:
16            n));
17    end
end

```

### 3.18 Esercizio 18

```

1 function [x] = solveQRH( A, b )
2     [m,n] = size(A);
3     Qt = eye(m);
4     for i=1:n
5         Qt= [eye(i-1) zeros(i-1,m-i+1); zeros(i-1, m-i+1)' (eye(m-i+1)-(2/norm([1; A(i+1:
6             m, i)], 2)^2)*([1; A(i+1:m, i)]*[1 A(i+1:m, i)]'))]*Qt;
7     end
8     x = TriSup(triu(A(1:n, :)), Qt(1:n, :)*b);
end

```

### 3.19 Esercizio 19

Il codice usato è:

```

1 format shortEng
2 format compact
3
4 A = [3,2,1;1,2,3;1,2,1;2,1,2]
5
6 b = [6;6;4;4]
7
8 x = solveQRH(A,b)
9
10 r = A*x-b

```

```

11
12 disp('Norma di r : ')
13 norm(r,2)^2

```

che ha generato questo risultato:

```

1 A =
2     3.0000e+000     2.0000e+000     1.0000e+000
3     1.0000e+000     2.0000e+000     3.0000e+000
4     1.0000e+000     2.0000e+000     1.0000e+000
5     2.0000e+000     1.0000e+000     2.0000e+000
6 b =
7     6.0000e+000
8     6.0000e+000
9     4.0000e+000
10    4.0000e+000
11 x =
12     3.6762e+000
13    -10.0571e+000
14     8.2286e+000
15 r =
16    -6.8571e+000
17     2.2476e+000
18    -12.2095e+000
19     9.7524e+000
20 Norma di r :
21 ans =
22    296.2536e+000

```

### 3.20 Esercizio 20

La funzione data è

$$F(x_1, x_2) = \begin{cases} x_2 - \cos(x_1) \\ x_1 x_2 - 1/2 \end{cases}$$

Vogliamo trovare  $F(x_1, x_2) = 0$  partendo da  $x_1(0) = 1$ ,  $x_2(0) = 1$   
Troviamo quindi il Jacobiano della funzione:

$$J = \begin{pmatrix} \sin(x_1) & 1 \\ x_1 & x_2 \end{pmatrix}$$

Applicando il metodo di Newton si va a risolvere:

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

Usando il codice MatLab dell'esercizio successivo (**Esercizio 21**), si ottengono i risultati sotto:

$$x_1 = 0.6100 \text{ e } x_2 = 0.8196$$

con la rispettiva tabella:

i	$x_1$	$x_2$	Norma incremento
1	0.7458	0.7542	0.5001
2	0.5531	0.8653	0.3145
3	0.6042	0.8241	0.0929
4	0.6100	0.8197	0.0102
5	0.6100	0.8196	0.00013

### 3.21 Esercizio 21

Un punto stazionario  $(\hat{x}_1, \hat{x}_2)$  è tale per cui  $J(\hat{x}_1, \hat{x}_2) = 0$ . Il sistema lineare è definito quindi da:

$$F(\underline{x}) = \underline{0} \text{ con } F = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 4x_1^3 + 2x_1 + x_2 \\ x_1 + 2x_2 - 2 \end{pmatrix}.$$

Il Jacobiano della funzione è quindi dato da:

$$J = \begin{pmatrix} 12x_1^2 + 2 & 1 \\ 1 & 2 \end{pmatrix}$$

da cui si ottiene il minimo della funzione:

$$\min f(x_1, x_2) \approx -0.2573$$

Il codice matlab per il calcolo del minimo è:

```
1 format short
2 format compact
3
4 x(1)=1;
5 x(2)=1;
6 imax=1000;
7 tolX=0.0001;
8
9 F = @(x) [x(2) - cos(x(1)); x(1)*x(2)-1/2];
10 J = @(x) [sin(x(1)), 1 ; x(2), x(1)];
11
12 [x] = nonLinearNewton(F, J, x, imax, tolX , 1);
13
14 disp ('Usando l''algoritmo di Newton la radice ottenuta e'': ')
15 disp (x);
16 disp ('F(x): ');
17 disp ([x(2) - cos(x(1)), x(1)*x(2)-1/2]);
```

### 3.22 Funzioni MatLab Usate

#### 3.22.1 Algoritmo di fattorizzazione LU con pivoting parziale

```
1 function [L,U,P]=factLUP(A)
2 [m,n]=size(A);
3 if m~=n
4     error('[Errore] La matrice inserita A inserita non e'' quadrata');
5 end
6 L=eye(n);
7 P=eye(n);
8 U=A;
9 for k=1:n
10     [pivot, m]=max(abs(U(k:n,k)));
11     if pivot==0
12         error('[Attenzione] La matrice e'' singolare');
13     end
14     m=m+k-1;
15     if m~=k
16         U([k,m], :) = U([m, k], :);
17         P([k,m], :) = P([m, k], :);
18         if k >= 2
19             L([k,m], 1:k-1) = L([m,k], 1:k-1);
20         end
21     end
22     U(k,m)=0;
23 end
```

```

21     end
22     L(k+1:n,k)=U(k+1:n,k)/U(k,k);
23     U(k+1:n,:)=U(k+1:n,:)-L(k+1:n,k)*U(k,:);
24 end

```

### 3.22.2 Risoluzione sistema lineare con funzione di ingresso già fattorizzata LU

```

1 function [b] = triInf(A, b)
2     for j=1:length(A)
3         if A(j,j) ~= 0
4             b(j) = b(j)/A(j,j);
5         else
6             error('[Attenzione]La matrice e'' singolare')
7         end
8         for i=j+1:length(A)
9             b(i) = b(i)-A(i,j)*b(j);
10        end
11    end
12 end

```

```

1 function [b] = triSup(A, b)
2     for j=size(A):-1:1
3         if A(j,j)==0
4             error('[Attenzione] La matrice non e'' singolare')
5         else
6             b(j)=b(j)/A(j,j);
7         end
8         for i=1:j-1
9             b(i)=b(i)-A(i,j)*b(j);
10        end
11    end
12 end

```

### 3.22.3 Risoluzione diagonale matrice LDLT di un sistema lineare

```

1 function [d] = linDiag(d,x)
2     n = size(d);
3     for i = 1:n
4         d(i) = x(i)/d(i);
5     end
6 end

```

### 3.22.4 Risoluzione di sistemi di equazioni non lineari mediante Newton

```

1 function [x] = nonLinearNewton(F,J, x, imax, tolX, out)
2     i=0;
3     xold = x+1;
4     while (i< imax )&&( norm (x-xold )> tolX )
5         i=i+1;
6         xold =x;
7         [L,U,P] = factLUP(feval(J,x));
8         x=x+linLUP(L,U, P, -feval(F,x));
9         if out
10             disp(norm(x-xold));
11             disp(x);

```

```

12         end
13     end
14 end

```

## 4 Capitolo 4

Le funzioni usate nei codici seguenti sono in fondo al capitolo

### 4.1 Esercizio 1

```

1 function [pval] = newtonHor(xi, fi, xval)
2     dd = diffDiv(xi, fi);
3     pval = hornerGen(xi,dd,xval);
4 end

```

### 4.2 Esercizio 2

Il codice MatLab usato è il seguente:

```

1 Rungef = @(x) 1./(1.+x.^2);
2 a=-5; b=5;
3 res_err = zeros(4,10);
4 [errors, plots, l] = evaluatePoli(Rungef, a,b, 20, 10, 0, 100);
5 plots = cat(2,plots', Rungef(l)');
6 res_err(1,:) = max(abs(errors'))';
7 plot(l,plots');
8 lgd=legend('2','4','6','8','10','\infty');
9 title(lgd, 'Ascisse');
10 plot(l,errors')
11 lgd=legend('2','4','6','8','10');
12 title(lgd, 'Ascisse');
13 [errors, plots, l] = evaluatePoli(Rungef, a,b, 20, 10, 1, 100);
14 res_err(2,:) = max(abs(errors'))';
15 plots = cat(2,plots', Rungef(l)');
16 plot(l,plots');
17 lgd=legend('2','4','6','8','10','12','14','16','18','20','\infty');
18 title(lgd, 'Ascisse');
19 plot(l,errors')
20 lgd=legend('2','4','6','8','10','12','14','16','18','20');
21 title(lgd, 'Ascisse');
22
23 Sinf = inline('x.*sin(x)');
24 a=0; b=pi;
25 max_n = 20;
26 [errors, plots, l] = evaluatePoli(Sinf, a,b, max_n, 10, 0, 100);
27 res_err(3,:) = max(abs(errors'))';
28
29 plots = cat(2,plots', Sinf(l)');
30 lgd=legend('2','4','6','8','10','12','14','16','18','20','\infty');
31 title(lgd, 'Ascisse');
32 plot(l,plots');
33 plot(l,errors')
34 lgd=legend('2','4','6','8','10','12','14','16','18','20');
35 title(lgd, 'Ascisse');
36 [errors, plots, l] = evaluatePoli(Sinf, a,b, max_n, 10, 1, 100);
37 res_err(4,:) = max(abs(errors'))';

```

```

38
39 plots = cat(2,plots', Sinf(l)');
40 plot(l,plots');
41 lgd=legend('2','4','6','8','10','12','14','16','18','20','\infty');
42 title(lgd, 'Ascisse');
43 plot(l,errors')
44 lgd=legend('2','4','6','8','10','12','14','16','18','20');
45 title(lgd, 'Ascisse');

```

che genera questi risultati:

<i>RungeEq</i>	<i>RungeCheb</i>	<i>SinEq</i>	<i>SinCheb</i>
0.646	0.4371	0.6381	0.4371
0.4383	0.02286	0.04127	0.02286
0.6164	0.0004779	0.001343	0.0004779
1.045	$5.332 \cdot 10^{-6}$	$2.575 \cdot 10^{-5}$	$5.332 \cdot 10^{-6}$
1.915	$3.688 \cdot 10^{-8}$	$3.238 \cdot 10^{-7}$	$3.688 \cdot 10^{-8}$
3.612	$1.734 \cdot 10^{-10}$	$2.843 \cdot 10^{-9}$	$1.734 \cdot 10^{-10}$
7.189	$5.892 \cdot 10^{-13}$	$1.873 \cdot 10^{-11}$	$5.892 \cdot 10^{-13}$
14.01	$3.417 \cdot 10^{-15}$	$1.261 \cdot 10^{-13}$	$3.417 \cdot 10^{-15}$
27.51	$1.776 \cdot 10^{-15}$	$8.933 \cdot 10^{-14}$	$1.776 \cdot 10^{-15}$
58.41	$2.327 \cdot 10^{-15}$	$1.768 \cdot 10^{-13}$	$2.327 \cdot 10^{-15}$

Possiamo vedere la differenza tra 5 e 3, nella prima all'aumentare delle ascisse la funzione interpolata degenera, mentre nella seconda già con  $n=5$  abbiamo una buona interpolazione. Nel caso della seconda funzione possiamo vedere che la differenza tra 9 e 7 non è molto rilevante. Infatti già con 4 ascisse abbiamo un'interpolazione quasi perfetta. Nelle figure 6 4 10 8 è possibile vedere l'andamento dell'errore per i vari metodi di interpolazione.

### 4.3 Esercizio 3

```

1 function [ m ] = moment(phi, xi, dd)
2     n = length(xi) + 1;
3     u = zeros(1, n - 1);
4     l = zeros(1, n - 2);
5     dd = 6 * dd;
6     u(1) = 2;
7     for i = 2 : n - 1
8         l(i) = phi(i) / u(i - 1);
9         u(i) = 2 - l(i) * xi(i - 1);
10    end
11    y = zeros(1, n - 1);
12    y(1) = dd(1);
13    for i = 2 : n - 1
14        y(i) = dd(i) - l(i) * y(i - 1);
15    end
16    m = zeros(1, n - 1);
17    m(n - 1) = y(n - 1) / u(n - 1);
18    for i = n - 2 : -1 : 1
19        m(i) = (y(i) - xi(i) * m(i + 1)) / u(i);
20    end
21    m = [0, m, 0];
22 end

```

### 4.4 Esercizio 4

```

1 function [ xx ] = eval(p, s, xx)
2     n=length(p) - 1;

```

```

3      k=1;
4      j=1;
5      for i = 1 : n
6          inInt = 1;
7          while j <= length(xx) && inInt
8              if xx(j) >= p(i) && xx(j) <= p(i + 1)
9                  j = j + 1;
10             else
11                 inInt = 0;
12             end
13         end
14         xx(k : j - 1) = subs(s(i), xx(k : j - 1));
15         k = j;
16     end
17 end

```

## 4.5 Esercizio 5

Il seguente listato valuta la spline naturale e quella not-a-knot per le funzioni date:

```

1  Rungef = @(x) 1./(1.+x.^2);
2  a=-5; b=5;
3  max_n = 20;
4  n_steps = 5;
5  [plots, l] = evalSpline(Rungef,a,b, max_n, n_steps, 0, 1000);
6  hold on;
7  grid on;
8  plot(l, plots);
9  hold off
10 [plots2, l] = evalSpline(Rungef,a,b, max_n, n_steps, 1, 1000);
11 hold on;
12 grid on;
13 plot(l, plots2);
14 hold off
15
16 error = plots' - plots2';
17 boxplot(error(:,1:5), 4:4:max_n);
18
19 Sinf = @(x) x.*sin(x);
20 a=0; b=pi;
21 [plots, l] = evalSpline(Sinf,a,b, max_n, n_steps, 0, 1000);
22 hold on;
23 grid on;
24 plot(l, plots);
25 hold off
26
27 [plots2, l] = evalSpline(Sinf,a,b, max_n, n_steps, 1, 1000);
28 hold on;
29 grid on;
30 plot(l, plots2);
31 hold off
32 error = plots' - plots2';
33
34 boxplot(error(:,1:5), 4:4:max_n);

```

con i risultanti grafici: grafico runge not-a-knot 11 grafico errori runge not-a-knot 13 grafico not-a-knot funzione 12 grafico errori not-a-knot funzione 14

#### 4.6 Esercizio 6

#### 4.7 Esercizio 7

#### 4.8 Esercizio 8

```
1 x=[0,0,0,3,4,3,2];
2 y=[1,2,5,2.1,1,2.2,0];
3 res = sisSov(x,y, 4)
```

#### 4.9 Esercizio 9

```
1 f1 = @(x,e,l) 5*x+ 2 +e*l;
2 f2 = @(x,e,l) 3*x^2 + 2*x +1 + e*l;
3 l=rand(1);
4 e= 0.1;
5 s= linspace(-1,1,10);
6 y1 = zeros(10,1);
7 y2 = zeros(10,1);
8
9 for i=1:10
10     l=rand(1);
11     y1(i) = f1(s(i),e,l);
12     y2(i) = f2(s(i),e,l);
13
14 end
15
16 res1 = sisSov(s,y1',1);
17 res2 = sisSov(s,y2',2);
```

#### 4.10 Esercizio 10

```
1 f1 = @(x,e,l) 5*x+ 2 +e*l;
2 l=rand(1);
3 e= 0.1;
4 s= linspace(-1,1,10);
5 y1 = zeros(10,1);
6 y = zeros(2,10);
7 for i=1:10
8     l=rand(1);
9     y1(i) = f1(s(i),e,l);
10 end
11
12 fit = sisSov(s, y1',1);
13 y(1,:)=polyval(fit, s);
14
15 invfit = sisSov(y1',s,1);
16 y(2,:)=polyval(invfit',y1);
17 plot(y1',y);
```

vedi 15 per il grafico

#### 4.11 Funzioni MatLab Usate

##### 4.11.1 Differenze divise



```

1 function [fi] = diffDiv(xi, fi)
2   for i=1:length(xi)-1
3     for j=length(xi):-1:i+1
4       fi(j) = (fi(j) - fi(j-1))/(xi(j)-xi(j-1));
5     end
6   end
7 end

```

#### 4.11.2 Horner Generalizzato

```

1 function [p] = hornerGen(xi, dd, xval)
2   n=length(dd);
3   for i=1:length(xval)
4     p(i)=dd(n);
5     for k=n-1:-1:1
6       p(i)=p(i)*(xval(i)-xi(k))+dd(k);
7     end
8   end
9 end

```

#### 4.11.3 Funzione di valutazione

```

1 function [errors, plots, l] = evaluatePoli(func, a, b, maxn, n_steps, cheb_asc,
2   plot_steps)
3   errors = zeros(n_steps, plot_steps);
4   plots = zeros(n_steps, plot_steps);
5   l = linspace(a, b, plot_steps);
6   steps = linspace(2, maxn, n_steps);
7   for i=1:n_steps
8     if cheb_asc == 0
9       ascisse = eqAscisse(a, b, steps(i));
10    elseif cheb_asc == 1
11      ascisse = cheby(a, b, steps(i));
12    end
13    fInt = newtonHor(ascisse, func(ascisse), l);
14    errors(i,:) = func(l)-fInt;
15    plots(i,:) = fInt;
16  end
17 end

```

#### 4.11.4 Ascisse Equispaziate

```

1 function [ptx] = eqAscisse(a, b, n)
2   h = (b-a)/n;
3   ptx = zeros(n+1, 1);
4   for i=1:n+1
5     ptx(i) = a +(i-1)*h;
6   end
7 end

```

#### 4.11.5 Chebyshev

```

1 function [xi] = cheby(a,b,n)
2   xi = zeros(n+1, 1);
3   for i=0:n
4     xi(n+1-i) = (a+b)/2 + cos(pi*(2*i+1)/(2*(n+1)))*(b-a)/2;
5   end
6 end

```

#### 4.11.6 valutazione Spline

```

1 function [plots, value_space] = evalSpline(funcnt, a, b, max_n, n_steps, nak, plot_steps)
2   value_space = linspace(a,b,plot_steps);
3   plots = zeros(n_steps+1,plot_steps);
4   ste = linspace(4,max_n,n_steps);
5   for i=1:n_steps
6     l = linspace(a,b,ste(i));
7     if nak
8       plots(i,:) = fnval(csapi(l,funcnt(l)), value_space);
9     else
10      plots(i,:) = eval(l,splineNat(l, funcnt(l)),value_space)';
11    end
12  end
13  plots(n_steps+1,:) = funcnt(value_space);
14 end

```

#### 4.11.7 Sistema sovradeterminato

```

1 function [y] = sisSov(x,y, m)
2   x=x';
3   if length(unique (x)) < m+1
4     error(['Errore] Non ci sono m ascisse distinte']);
5   end
6   V(:,m+1) = ones(length(x),1);
7   for j = m:-1:1
8     V(:,j) = x.*V(:,j+1);
9   end
10  y = V\y';
11  y=y';
12 end

```

## 5 Grafici

Figure 1: Esercizio 1.4

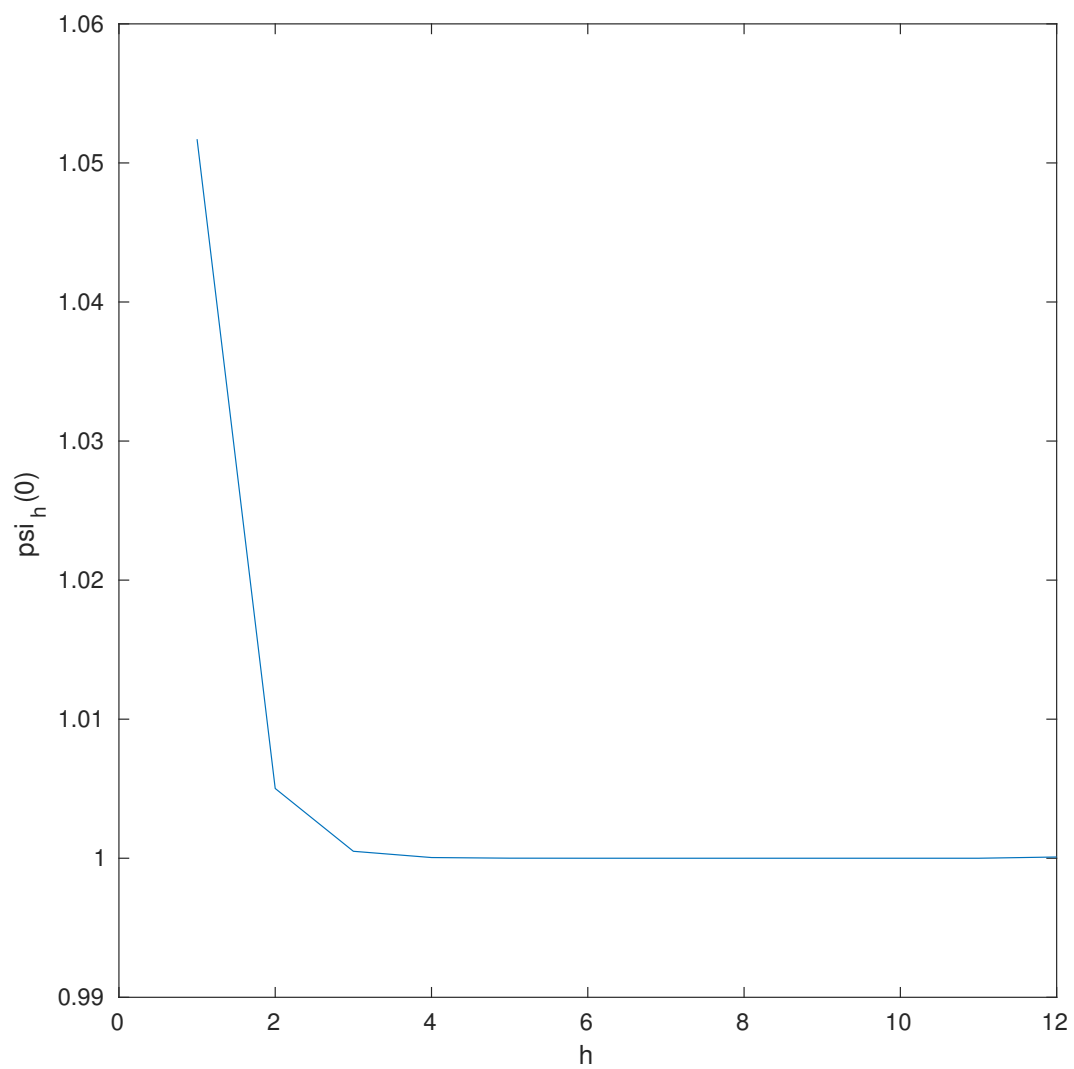


Figure 2: Esercizio 1.13

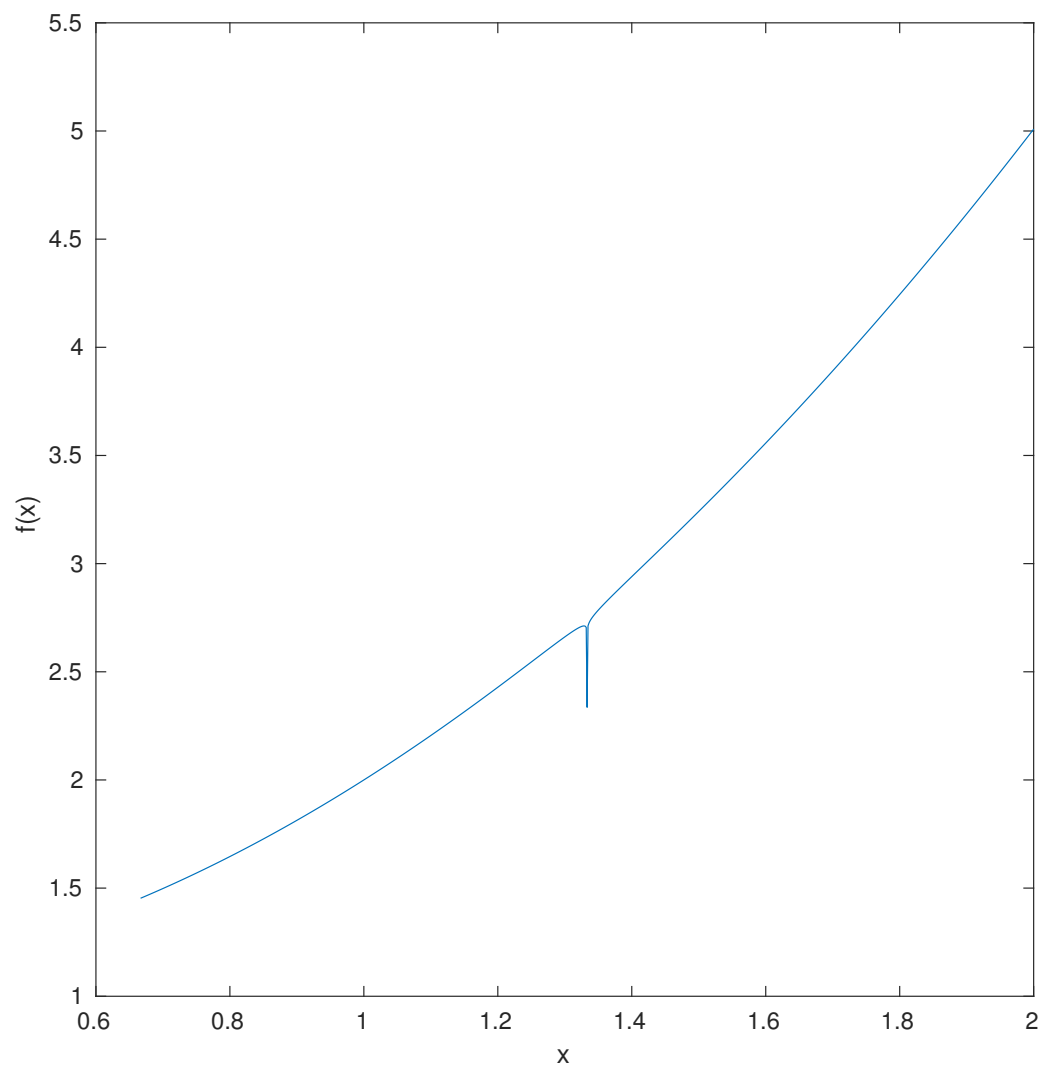


Figure 3: Runge Chebyshev

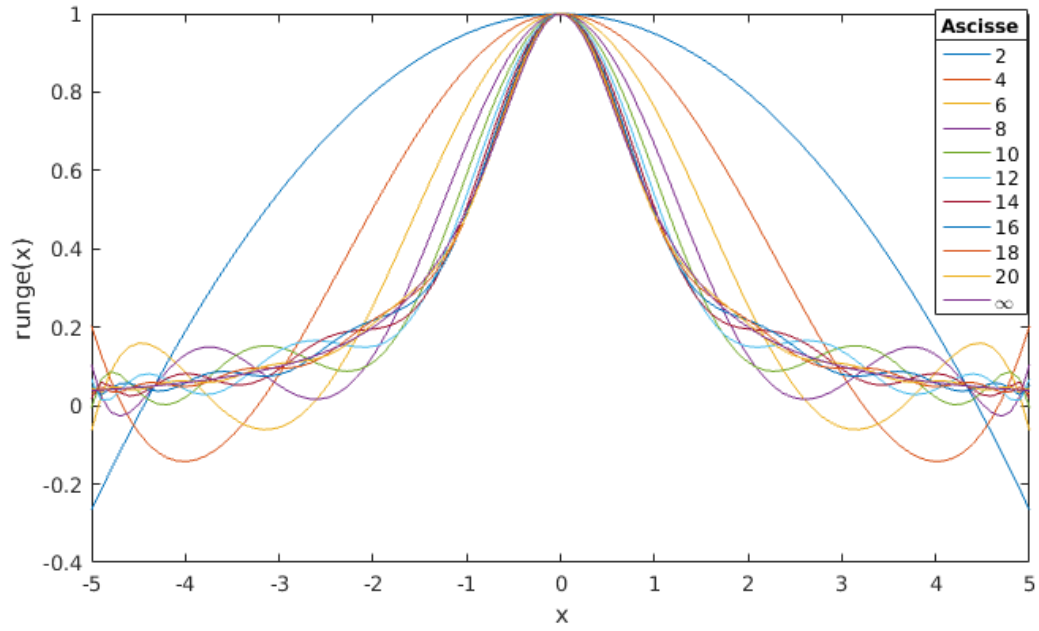


Figure 4: Runge Chebyshev Errori

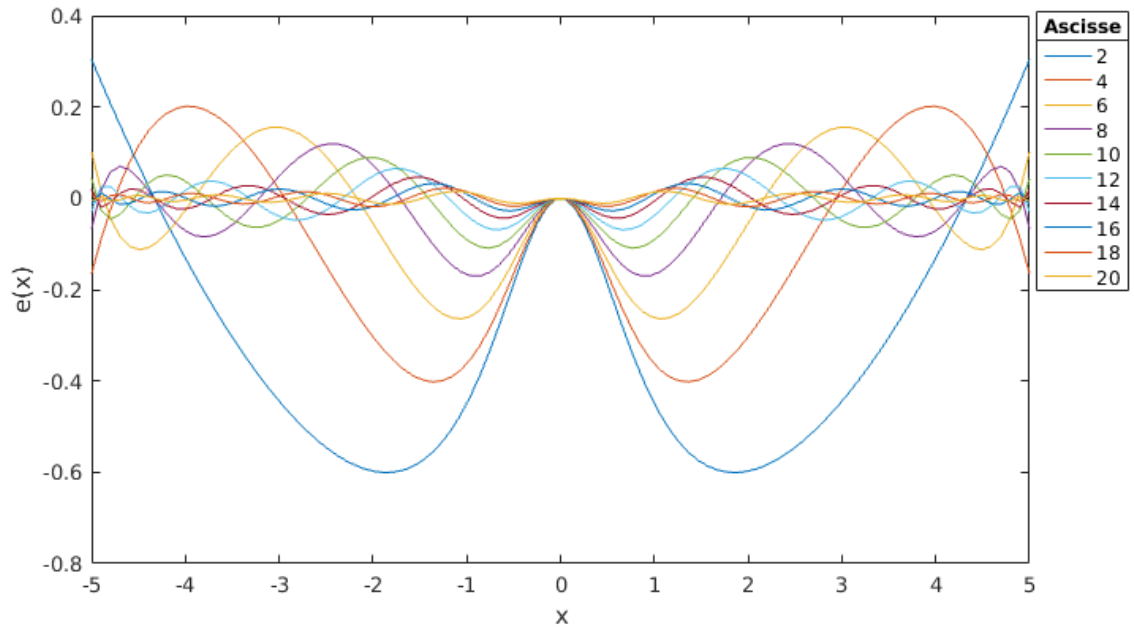


Figure 5: Runge con ascisse equidistanti

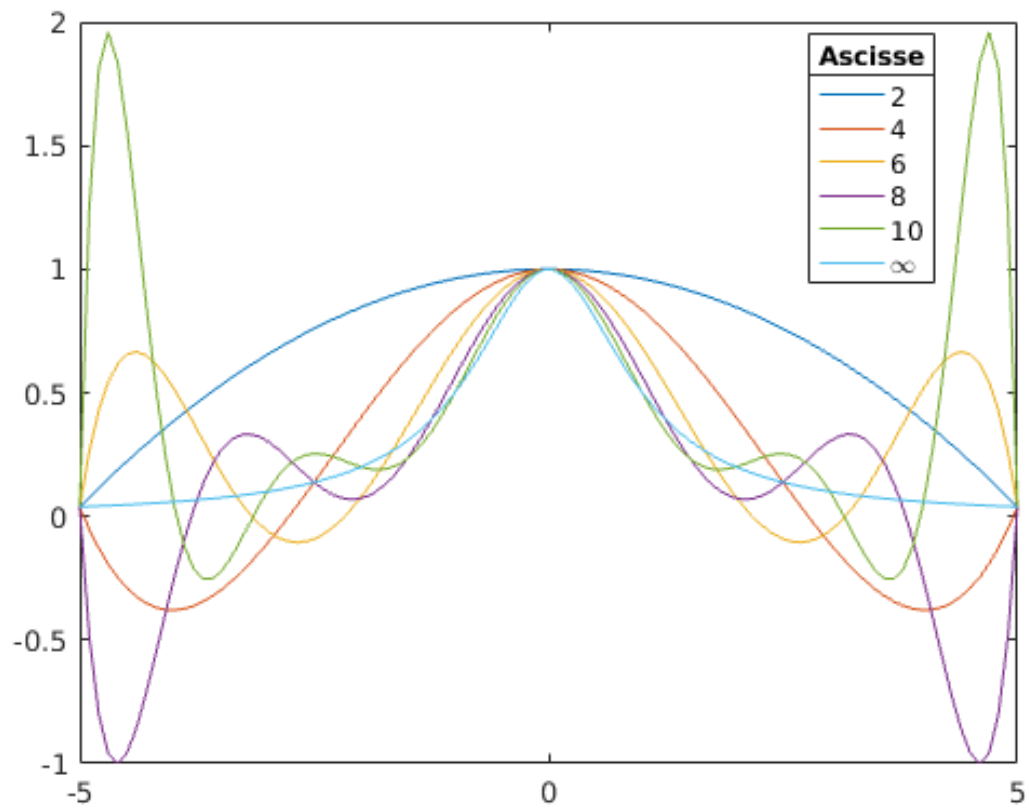


Figure 6: Runge con ascisse equidistanti errori

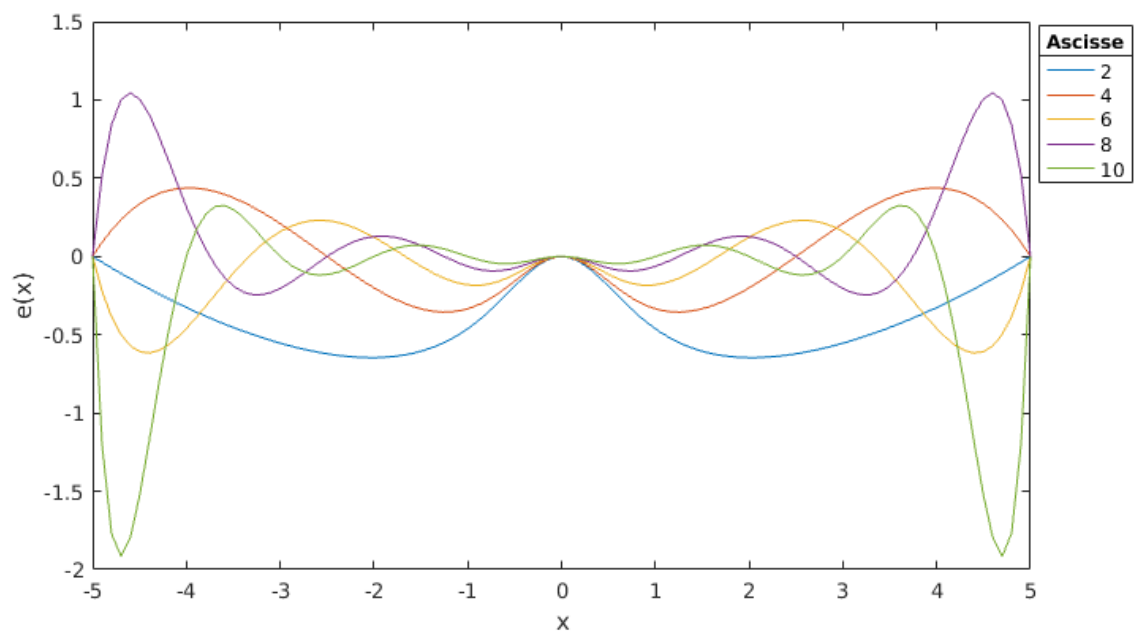


Figure 7: Funzione xsinx Chebyshev

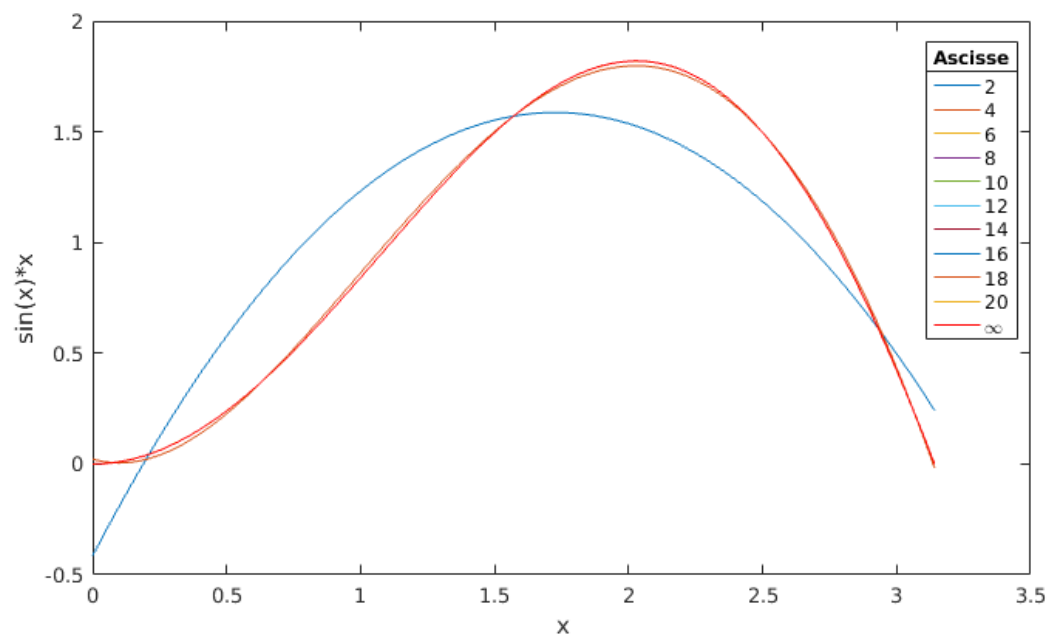


Figure 8: Funzione xsinx Chebyshev errori

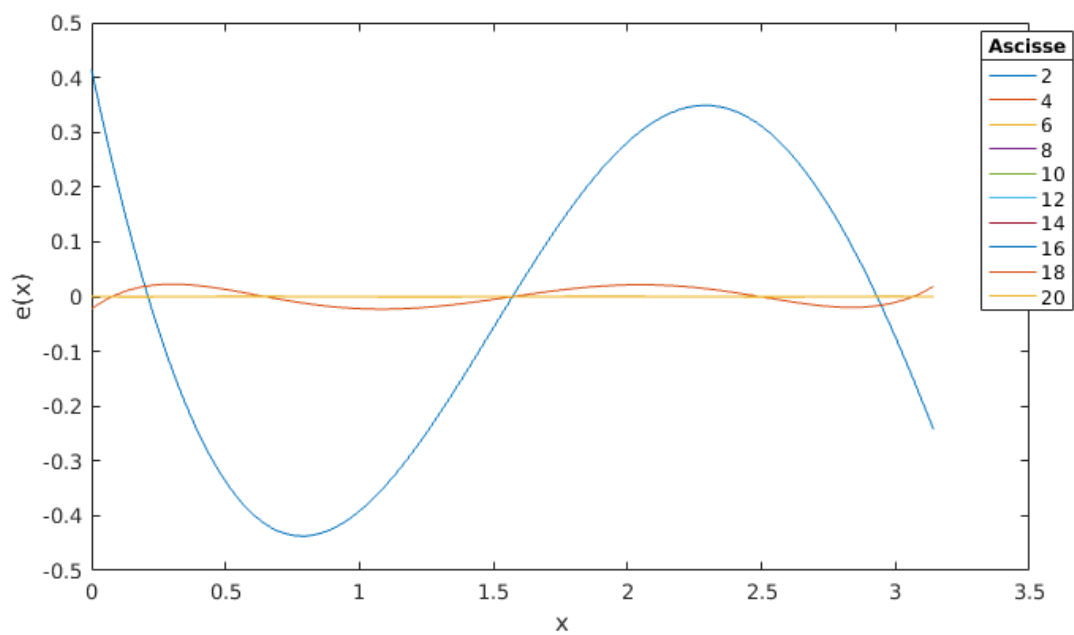


Figure 9: Funzione  $x \sin x$  ascisse equidistanti

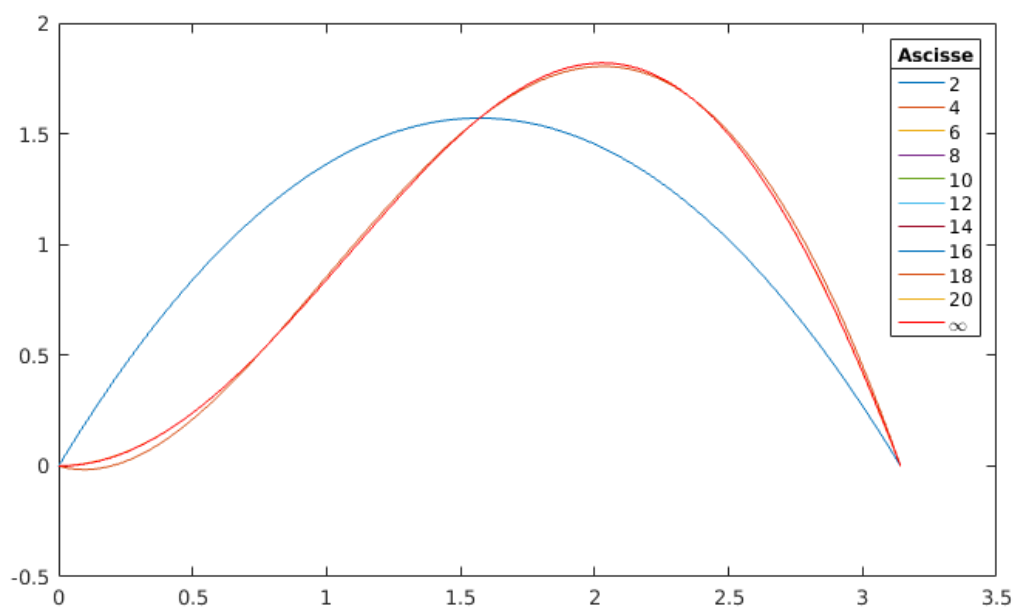


Figure 10: Funzione  $x \sin x$  ascisse equidistanti errori

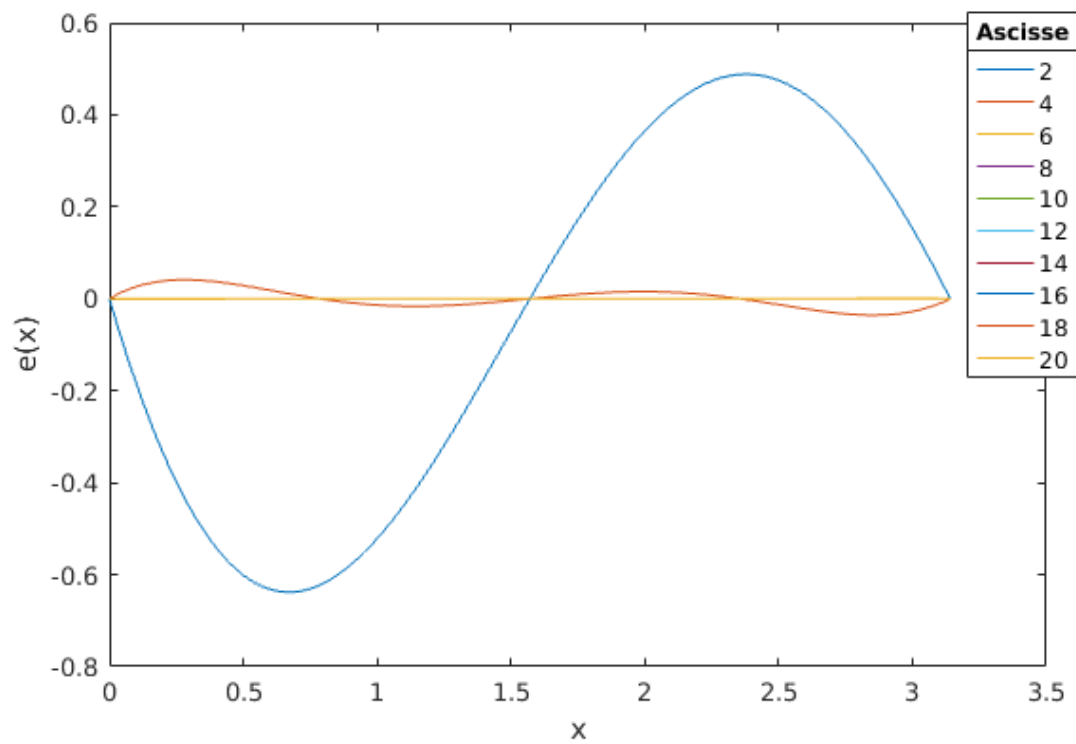




Figure 11: Runge not-a-knot

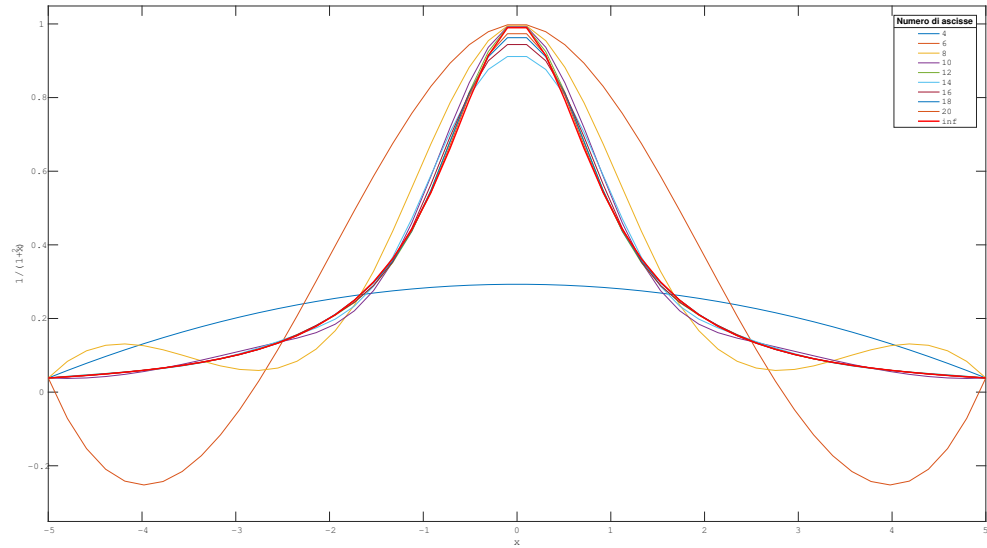


Figure 12: Funzione xsinx not-a-knot

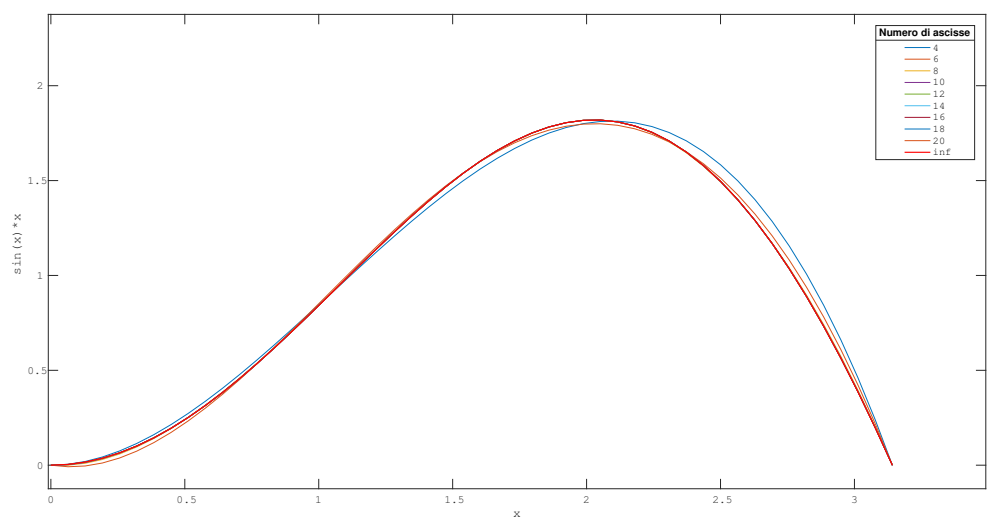


Figure 13: Errori Runge

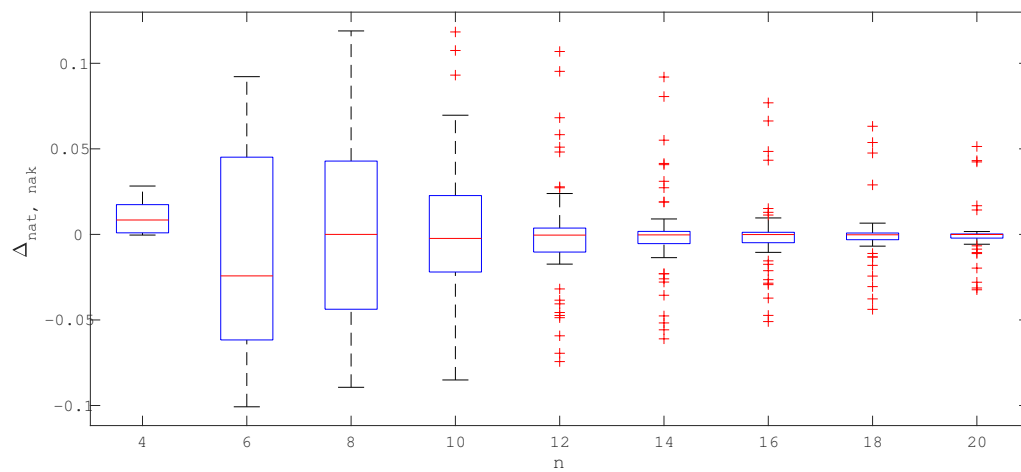


Figure 14: Errori funzione xsinx

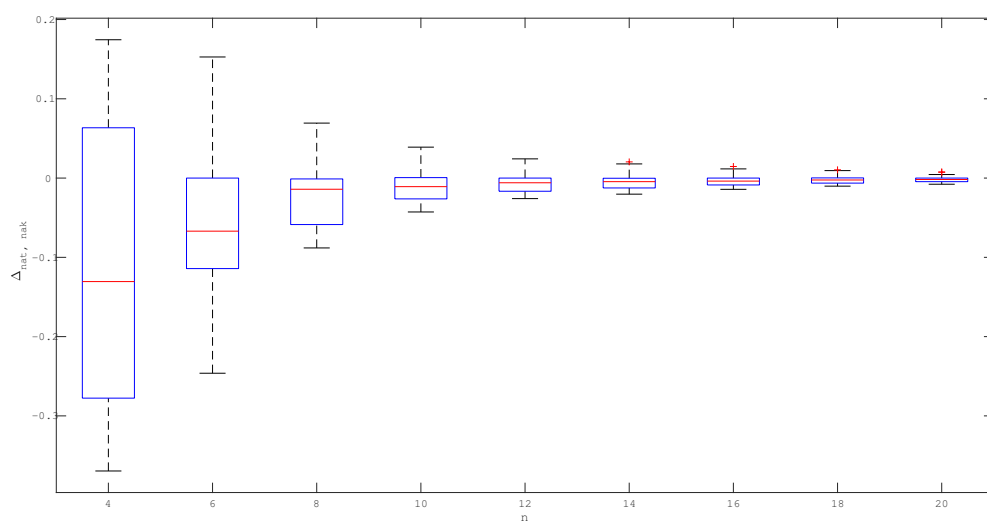


Figure 15: Esercizio 4.10

