

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 29, 2017

Capitoli

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

3.22	Funzioni MatLab Usate	28
3.22.1	Risoluzione sistema lineare	28
3.22.2	Risoluzione diagonale matrice LDLT di un sistema lineare	28
3.22.3	Risoluzione di sistemi di equazioni non lineari mediante Newton	28
4	Capitolo 4	30
4.1	Esercizio 1	30
4.2	Esercizio 2	30
4.3	Esercizio 3	31
4.4	Esercizio 4	31
4.5	Esercizio 5	32
4.6	Esercizio 8	32
4.7	Esercizio 9	33
4.8	Esercizio 10	33
4.9	Funzioni MatLab Usate	33
4.9.1	Differenze divise	33
4.9.2	Horner Generalizzato	34
4.9.3	Funzione di valutazione	34
4.9.4	Ascisse Equispaziate	34
4.9.5	Chebyshev	34
4.9.6	valutazione Spline	35
4.9.7	Sistema sovradeterminato	35
5	Capitolo 5	36
5.1	Esercizio 5.1	36
5.2	Esercizio 5.2	36
5.3	Esercizio 5.3	36
5.4	Esercizio 5.4	37
5.5	Esercizio 5.5	38
5.6	Esercizio 5.6	39
6	Grafici	40

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 Φ si calcola il limite:

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

infine ipotizzando che la funzione Φ sia uniformemente continua, è possibile calcolare il limite:

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

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:

$$(32767)_{10} + (1)_{10} = (011111111111111)_{2,MS} + (0000000000000001)_{2,MS} = (1000000000000000)_{2,MS} = (-32768)_{10}$$

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

$$(-32768)_{10} - (1)_{10} = (1000000000000000)_{2,MS} - (0000000000000001)_{2,MS} = (011111111111111)_{2,MS} = (32767)_{10}$$

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,h(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 vede che i valori di $\Psi_h(0)$ diminuiscono fino ad $h = 10^{-8}$, in cui si ha il minimo valore di $\Psi_h(0)$. Dopodichè il margine di errore inizia a crescere come mostra la tabella e il relativo plot a pagina 40.

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:

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

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

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

restituisce i valori:

n	x_n	ϵ
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

I valori indicano che per valori di n superiori a 5 l'errore, indicato con ϵ , 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 :

$$m = 1 - \log_2(2 \cdot 4.66 \cdot 10^{-10}) = 1 - \log_2(9.32 \cdot 10^{-10}) = 1 - (-29.9999999) \approx 31$$

possiamo pertanto affermare che servono 31 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 $\log_{10}u \gg 1$ 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)

Possiamo fare l'esempio con i valori $b = 10$ e $u \approx 4.66 \cdot 10^{-10}$ ottenendo così:

- $m = 10.3316$ (troncamento)
- $m = 10.0306$ (arrotondamento)

che è una buona approssimazione di $-\log_{10}u = 9.33161$.

1.9 Esercizio 9

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

dato che il valore di $\delta = [0, 1]_{10}$ in binario si scrive $\delta = [0, \overline{00011}]_2$ allora si nota che la rappresentazione del valore di δ in binario è periodica. Al passo 10 la rappresentazione di x sarà diversa da 1, perchè somma di numeri periodici, essendo $x = 1$ l'unica condizione di uscita dello while il ciclo non si arresterà mai. Possiamo provarlo effettuando le somme binarie:

$$\begin{aligned} \left[\frac{1}{10}\right]_{10} &= [0, \overline{00011}]_2 \\ [0, \overline{00011}]_2 + [0, \overline{00011}]_2 + \underbrace{\dots}_{6\text{ volte}} + [0, \overline{00011}]_2 + [0, \overline{00011}]_2 &= \\ &= [1, 00010]_2 \approx [1.0625]_{10} \neq [1.0000]_{10} \end{aligned}$$

che spiegherebbe il motivo del loop dello while.

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:

- $fl(fl(fl(x) + fl(y)) + fl(z)) = ((x(1 + \varepsilon_x) + y(1 + \varepsilon_y))(1 + \varepsilon_a) + z(1 + \varepsilon_z))(1 + \varepsilon_b)$
- $fl(fl(x) + fl(fl(y) + fl(z))) = (x(1 + \varepsilon_x) + (y(1 + \varepsilon_y) + z(1 + \varepsilon_z))(1 + \varepsilon_a))(1 + \varepsilon_b)$

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, 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:

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

Otteniamo quindi che i valori degli errori ε_1 e ε_2 sono $\leq 3 \cdot \varepsilon_m$.

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:

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

1.13 Esercizio 13

Nella riga 11 abbiamo calcolato e restituito in output il valore interno al logaritmo $|3(1 - \frac{4}{3}) + 1|$ 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 xlabel('x');
12 ylabel('f(x)');
13 disp('valore interno al limite in x=4/3 : ');
14 disp(abs(3*(1-4/3)+1))
15 disp('la funzione calcolata in 4/3 ha in output:');
16 disp(log(abs(3*(1-(4/3))+1))/80 + (4/3)^2 +1);

```


calcola i valori della funzione ottenendo il grafico a pagina 41 e si può notare che l'asintoto verticale in $x = \frac{4}{3}$ non viene rappresentato come tale. Il problema è che stiamo rappresentando dei numeri reali in un calcolatore, quindi la loro rappresentazione comporta delle approssimazioni. In questo caso infatti abbiamo il valore di $4/3$ che è un numero periodico, ma il calcolatore lo dovrà rappresentare con un numero di cifre finite causando un errore di rappresentazione che in questo caso risulta rilevante. Si allega sotto l'output del codice soprastante:

```
>> es13
  il valore di 4/3 :
    1.3333333333333333
valore interno al limite in x=4/3 :
    2.220446049250313e-16
la funzione calcolata in 4/3 ha in output:
    2.327232110413813
```

2 Capitolo 2

Le funzioni usate nei codici seguenti sono in fondo al capitolo (pag. 13)

2.1 Esercizio 1

Per ricercare la radice quadrata di un numero è possibile sfruttare una modifica al problema delle radici di una funzione. Infatti partendo da $x = \sqrt{\alpha}$ è possibile svilupparla trovando una funzione $f(x)$ utilizzabile nel metodo di Newton:

$$\begin{aligned}x &= \sqrt{\alpha} \\ x^2 &= (\sqrt{\alpha})^2 \\ x^2 - \alpha &= 0\end{aligned}$$

possiamo quindi considerare $f(x) = x^2 - \alpha$ e $f'(x) = 2 \cdot x$. La procedura iterativa è definita quindi da:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - \alpha}{2 \cdot x_n} = x_n - \frac{x_n}{2} + \frac{\alpha}{2 \cdot x_n} = \frac{x_n}{2} + \frac{\alpha}{2 \cdot x_n} = \frac{1}{2} \cdot \left(x_n + \frac{\alpha}{x_n} \right)$$

che ci permette di implementare il seguente script matlab e la funzione $y = \text{NewtonSqrt}(\alpha, x_0, \text{imax}, \text{tol})$:

```
1 x_0 = 3;
2 alpha = 3;
3 n = NewtonSqrt(alpha, x_0, 100, 10^(-8));
```

Che restituisce in output valori che abbiamo rappresentato nella tabella seguente:

i	x_i
1	1.7500000000000000e+00
2	1.732142857142857e+00
3	1.732050810014727e+00
4	1.732050807568877e+00

2.2 Esercizio 2

E' possibile effettuare gli stessi passaggi dell'esercizio precedente, ricordandosi che la radice in questo caso non è quadrata ma ennesima:

$$\begin{aligned}x &= \sqrt[n]{\alpha} \\ x^n &= (\sqrt[n]{\alpha})^n \\ x^n - \alpha &= 0\end{aligned}$$

consideriamo quindi la funzione $f(x) = x^n - \alpha$ e $f'(x) = n \cdot x^{n-1}$. La procedura iterativa è definita quindi da:

$$\begin{aligned}x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^n - \alpha}{n \cdot x_n^{n-1}} = x_n - \frac{x_n}{n} + \frac{\alpha}{n \cdot x_n^{n-1}} = \\ &= \left((n-1) \cdot x_n - \frac{\alpha}{x_n^{n-1}} \right) \cdot \frac{1}{n} = \frac{\left((n-1) \cdot x_n^n + \alpha \right)}{n \cdot x_n^{n-1}}\end{aligned}$$

La radice da approssimare in questo caso ha grado ennesimo quindi sono necessarie delle modifiche alla funzione matlab usata nell'esercizio precedente che chiamiamo $y = \text{NewtonSqrt}(n, \alpha, x_0, \text{imax}, \text{tol})$. Lo script MatLab corrispondente ai casi $n = 3, 4, 5$ è il seguente:

```
1 x_0 = 3;
2 alpha = 3;
3 disp('n=3');
4 n3 = NewtonSqrtN(3, alpha, x_0, 100, 10^(-8));
5 disp('n=4');
6 n4 = NewtonSqrtN(4, alpha, x_0, 100, 10^(-8));
7 disp('n=5');
8 n5 = NewtonSqrtN(5, alpha, x_0, 100, 10^(-8));
```

Mostriamo l'output in forma tabellare con i che rappresenta le iterazioni del metodo e x_i i relativi risultati:

i	x_i con $n = 3$	x_i con $n = 4$	x_i con $n = 5$
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 ricercare la radice quadrata di un numero è possibile sfruttare una modifica al problema delle radici di una funzione. Infatti partendo da $x = \sqrt{\alpha}$ è possibile svilupparla trovando una funzione $f(x)$ utilizzabile per il metodo delle secanti:

$$x = \sqrt{\alpha}$$

$$x^2 = (\sqrt{\alpha})^2$$

$$x^2 - \alpha = 0$$

possiamo quindi considerare $f(x) = x^2 - \alpha$. La procedura iterativa è definita quindi da:

$$x_{n+1} = \frac{(x_n^2 - \alpha) \cdot x_{n-1} - (x_{n-1}^2 - \alpha)x_n}{x_n^2 - x_{n-1}^2} = \frac{\alpha \cdot (x_n - x_{n-1}) + x_n \cdot x_{n-1} \cdot (x_n - x_{n-1})}{(x_n + x_{n-1}) \cdot (x_n - x_{n-1})} = \frac{\alpha + x_n \cdot x_{n-1}}{x_n + x_{n-1}}$$

che ci permette di implementare il seguente script matlab e la funzione $y = \text{SecSqrt}(\alpha, x_0, \text{imax}, \text{tol})$:

```

1 disp('metodo delle secanti');
2 s = SecSqrt(3,3,100,10^(-8));
3 disp('metodo di newton');
4 n = NewtonSqrt(3,3,100,10^(-8));

```

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

i	metodo di Newton	metodo delle secanti	newton- $\sqrt{3}$	secanti- $\sqrt{3}$
1	1.750000000000000e+00	1.736842105263158e+00	1.794919243112281e-02	4.791297694280772e-03
2	1.732142857142857e+00	1.732142857142857e+00	9.204957398001312e-05	9.204957397979108e-05
3	1.732050810014727e+00	1.732050934706042e+00	2.445850189047860e-09	1.271371643518648e-07
4	1.732050807568877e+00	1.732050807572256e+00	0	3.378630708539276e-12
5	_____	1.732050807568877e+00	_____	2.220446049250313e-16

Dalla tabella possiamo dunque notare che il metodo di Newton arriva più velocemente al valore cercato rispetto al metodo delle secanti.

2.4 Esercizio 4

Abbiamo scritto delle funzioni MalLab che ci permettono di effettuare il raffronto tra i vari metodi (in ordine: metodo di Newton, metodo di Newton modificato, metodo di accelerazione di Aitken), è possibile vederle a pag. 13. Scritte le Function, le abbiamo usate nel seguente script:

```
1  funct = @(x) (x-pi)^10;
2  dfunct = @(x) 10*(x-pi)^9;
3
4  disp('metodo newton funct 1');
5  for i=0:5
6      N1(i+1) = Newton(funct, dfunct, 5, 50, 10^(-i));
7  end
8
9  disp('newton modificato funct 1');
10 for i =0:5
11     NM1(i+1) = NewtonMod(funct, dfunct, 1, 5, 50, 10^(-i));
12 end
13
14 disp('aitken funct 1');
15 for i = 0:5
16     A1(i+1) = Aitken(funct, dfunct, 5, 50, 10^(-i));
17 end
18
19
20 funct = @(x) ((x-pi)^10)*(exp(1)^(2*x));
21 dfunct = @(x) (5+x-pi)*(x-pi)^9*2*exp(1)^(2*x);
22
23 disp('metodo newton funct 2');
24 for i=0:5
25     N2(i+1) = Newton(funct, dfunct, 5, 50 ,10^(-i));
26 end
27
28 disp('newton modificato funct 2');
29 for i=0:5
30     NM2(i+1) = NewtonMod(funct, dfunct, 1, 5, 50, 10^(-i));
31 end
32
33 disp('aitken funct 2');
34 for i=0:5
35     A2(i+1) = Aitken(funct, dfunct, 5, 50, 10^(-i));
36 end
37
38 format long
39
40 N1
41 NM1
42 A1
43 N2
44 NM2
45 A2
```

Questo codice esegue i metodi di Newton, Newton modificato e Aitken per le funzioni date. Rispetto alla funzione $f_1(x) = (x - \pi)^{10}$ rappresentiamo l'output del codice precedente in forma tabellare:

tolx	Newton	Newton modificato	Aitken
10^0	4.814159265358979	4.814159265358979	2.666666666666667
10^{-1}	4.030463126315021	4.030463126315021	3.141592653589783
10^{-2}	3.229126031324792	3.229126031324792	3.141592653589783
10^{-3}	3.150212685926121	3.150212685926121	3.141592653589783
10^{-4}	il metodo non converge	il metodo non converge	3.141592653589783
10^{-5}	il metodo non converge	il metodo non converge	3.141592653589783

Rispetto alla funzione $f_2(x) = (x - \pi)^{10} \cdot e^{2 \cdot x}$ si hanno invece i valori:

tolx	Newton	Newton modificato	Aitken
10^0	4.864516114854061	4.864516114854061	3.068982105941251
10^{-1}	4.200823975656096	4.200823975656096	3.140645194956157
10^{-2}	3.226469748549500	3.226469748549500	3.141592492095279
10^{-3}	il metodo non converge	il metodo non converge	3.141592492095279
10^{-4}	il metodo non converge	il metodo non converge	3.141592516390607
10^{-5}	il metodo non converge	il metodo non converge	3.141592516390607

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 $f_1(x) = (x - \pi)^{10} > 0 \forall x$ e $f_2(x) = e^{2x}(x - \pi)^{10} > 0 \forall x$ 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

Per poter costruire una tabella abbiamo scritto il seguente codice MatLab (lo script usa function definite a parte che è possibile vedere a pagina 13):

```

1 f = @(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 imax = 100;
8 iter = zeros(10,3);
9 res = zeros(10,3);
10 for i=1:10
11     [res(i,1), iter(i,1)] = Newton(f,df,x0,imax,tol(i));
12     [res(i,2), iter(i,2)] = secanti(f,df,x0,imax, tol(i));
13     [res(i,3), iter(i,3)] = corde(f,df,x0,imax, tol(i));
14 end
15
16 disp('La prima colonna sono riferite al metodo di Newton (iterazioni e risultato)');
17 disp('La seconda colonna sono riferite al metodo delle secanti (iterazioni e risultato)');
18 ;
19 disp('La terza colonna sono riferite al metodo delle corde (iterazioni e risultato)');
20 iter
    res

```

Sotto forma tabellare rappresentiamo il numero di iterazioni effettuate dai 3 algoritmi (salvate nella matrice *iter*):

tol_x	Newton	Secanti	Corde
10^{-1}	2	3	2
10^{-2}	3	3	8
10^{-3}	3	4	15
10^{-4}	3	5	22
10^{-5}	4	5	28
10^{-6}	4	5	35
10^{-7}	4	5	42
10^{-8}	4	6	48
10^{-9}	4	6	55
10^{-10}	5	6	62

Questo risultato è possibile vederlo in maniera grafica a pagina 42.

Il numero di condizionamento del problema è dato da:

$$k = \frac{1}{|f'(x^*)|}$$

Per poterlo calcolare è necessario trovare la derivata della nostra funzione che è pari a:

$$f'(x) = -1 - \cos(x) + 5 \cdot \sin(10 \cdot x) - \frac{\cos^2(10 \cdot x)}{2}$$

Essendo la radice della nostra funzione $x^* = 0,488944$ il relativo numero di condizionamento è dato da:

$$k = \frac{1}{|f'(x^*)|} = \frac{1}{|f'(0,488944)|} = \frac{1}{|-4,27233|} = \frac{1}{4,27233} = 0,234064$$

questo significa che il problema è ben condizionato.

2.7 Esercizio 7

Lo script usato è il seguente:

```

1 format long
2
3 f = @(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 tol = logspace(-1,-10,10);
7
8 for i=1:10
9     [res, iterB(i,1)] = bisect(f,0,1,tol(i));
10 end
11
12 disp('il numero di iterazioni del metodo di bisezione con tolleranza decrescente da
13     10^(-1) a 10^(-10) e'' il seguente');
14 iterB
15
16 for i=1:10
17     iter(i,4) = iterB(i);
18 end
19
20 plot(iter)
21 xlabel('10^{-x}');
22 ylabel('numero di iterazioni');
```

Il numero di iterazioni del metodo di bisezione risultanti sono:

tol_x	bisezione
10^{-1}	2
10^{-2}	7
10^{-3}	10
10^{-4}	14
10^{-5}	17
10^{-6}	20
10^{-7}	24
10^{-8}	27
10^{-9}	30
10^{-10}	34

Si può vedere dal grafico l'andamento dei vari metodi usati a pagina 43.

2.8 Esercizio 8

Per determinare la radice della funzione data, abbiamo scritto il seguente script MatLab:

```

1 f = @(x) (x-pi)*(exp(1)^(10*x));
2 df = @(x) (exp(1)^(10*x))*(10*x-10*pi+1);
3
4 disp('Con punto iniziale x0=0');
5 [res,nit] = Newton(f,df, 0, 100, 10^(-2))
6 disp('Con punto iniziale x0=4');
7 [res,nit] = Newton(f,df, 4, 100, 10^(-2))

```

Con risultato in output:

```

Con punto iniziale x0=0
il metodo non converge
res =
-10.246212624496760
nit =
100
Con punto iniziale x0=4
res =
3.142020948244105
nit =
12

```

Si può vedere che il metodo non converge per il punto iniziale $x_0 = 0$, ma con punto iniziale diverso $x_0 = 4$ il metodo converge. Questo significa che il metodo può convergere localmente. Dobbiamo quindi studiare la convergenza locale della funzione $f(x)$. Tale convergenza risulta essere garantita solo in un intorno della radice, in questo caso in un intorno di π . Essendo la funzione $f(x) = (x - \pi) \cdot e^{10 \cdot x}$ la funzione di iterazione che definisce il metodo è determinata da:

$$\Phi(x) = x - \frac{(x - \pi) \cdot e^{10 \cdot x}}{e^{10 \cdot x} \cdot (10 \cdot x - 10 \cdot \pi + 1)} = x - \frac{(x - \pi)}{(10 \cdot x - 10 \cdot \pi + 1)}$$

Per convergere localmente si deve avere π punto fisso della funzione di iterazione $\Phi(x)$. si ha infatti che:

$$\Phi(\pi) = \pi - \frac{(\pi - \pi)}{(10 \cdot \pi - 10 \cdot \pi + 1)} = \pi - 0 = \pi$$

Questo conferma il fatto che la funzione $f(x)$ è convergente localmente in un intorno di π , confermando i risultati ottenuti dallo script precedente.

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

2.9.3 Metodo delle secanti per $\sqrt{\alpha}$

```
1 function x = SecSqrt(x0,alpha,imax,tol)
2     x1 = NewtonSqrt(alpha,x0,1,0.00000001);
3     x = [x0,x1,(alpha+x1*x0)/(x1+x0)];
4     i = 3;
5     while(i < imax) && (abs(x(i)-x(i-1))>tol)
6         x(i+1) = (alpha + x(i)*x(i-1))/(x(i)+x(i-1));
7         disp(x(i+1));
8         i = i+1;
9     end
10    y = x(i);
11 end
```

2.9.4 Metodo di Newton

```
1 function [x, i] = Newton(fx,dfx, x0, imax, tolx)
2 fx0 = feval(fx,x0);
3 dfx0 = feval(dfx,x0);
4 x = x0 - fx0/dfx0;
5 i=0;
6 while (i<imax) & (abs(x-x0)>tolx)
7     i=i+1;
8     x0=x;
9     fx0=feval(fx,x0);
10    dfx0=feval(dfx,x0);
```



```

11     x=x0-fx0/dfx0;
12 end
13
14 if abs(x-x0)>tolx
15     disp('il metodo non converge');
16
17 end

```

2.9.5 Metodo di Newton Modificato

```

1 function x = NewtonMod(fx, dfx, m, x0, imax, tol)
2 fx0 = feval(fx,x0);
3 dfx0 = feval(dfx,x0);
4 x = x0 - m * fx0 / dfx0;
5 i = 0;
6 while((i < imax) & abs(x-x0) > tol)
7     i = i+1;
8     x0=x;
9     fx0 = feval(fx, x0);
10    dfx0 = feval(dfx, x0);
11    x = x0 - m * fx0 / dfx0;
12 end
13 if abs(x-x0)>tol
14     disp('il metodo non converge');
15 end

```

2.9.6 Metodo di accelerazione di Aitken

```

1 function y = Aitken( fx, dfx, x0, imax, tol )
2     i = 0;
3     x=x0;
4     diverror=1;
5     while((i < imax) && diverror)
6         i=i+1;
7         x0=x;
8         fx0 = feval(fx,x0);
9         dfx0 = feval(dfx,x0);
10        x1 = x0 - fx0/dfx0;
11        fx0 = feval (fx,x1);
12        dfx0 = feval (dfx, x1);
13        x = x1 - fx0/dfx0;
14        if (x-2*x1+x0 == 0)
15            disp('divisione per zero');
16            diverror = 0;
17            break;
18        end
19        x = (x*x0-x1^2)/(x-2*x1+x0);
20        diverror = abs(x-x0)>tol;
21    end
22    if(diverror)
23        disp('Il metodo non converge');
24    end
25    y = x;
26 end

```

2.9.7 Metodo delle secanti

```
1 function [x, i] = secanti(f,df,x0,imax, tol)
2 fx = feval(f,x0);
3 flx = feval (df, x0);
4 x = x0 - fx/flx;
5 i = 0;
6 while (i<imax) & (abs(x-x0)>tol)
7     i = i+1;
8     fx0 = fx;
9     fx = feval(f,x);
10    x1 = (fx*x0-fx0*x)/(fx-fx0);
11    x0 = x;
12    x = x1;
13 end
14 if (abs(x-x0)>tol), disp('il metodo non converge'), end
15 end
```

2.9.8 Metodo delle corde

```
1 function [x,i]=corde(fx,dfx,x0,imax,tol)
2 fx0 = feval(fx,x0);
3 dfx0 = feval(dfx,x0);
4 x = x0 - fx0/dfx0;
5 i=0;
6 while (i<imax) & (abs(x-x0)>tol)
7     i=i+1;
8     x0=x;
9     fx0=feval(fx,x0);
10    x=x0-fx0/dfx0;
11 end
12
13 if abs(x-x0)>tol
14     disp('il metodo non converge');
15 end
```

2.9.9 Metodo della bisezione

```
1 function [x,i]=bisection(f,a,b,tol)
2     fa = feval (f,a);
3     fb = feval (f,b);
4     x = (a+b)/2;
5     fx = feval(f,x);
6     imax = ceil (log2(b-a) - log2(tol));
7     for i=2:imax
8         flx = abs((fb-fa)/(b-a));
9         if abs(fx)<=tol*flx
10             break
11         elseif fa*fx<0
12             b=x;
13             fb=fx;
14         else
15             a=x;
16             fa=fx;
17         end
18         x = (a+b)/2;
19         fx = feval(f,x);
20     end
21 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:

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

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

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

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 segue:

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

questa somma può essere scritta anche:

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

gli elementi $a_{i,j} = \sum_{i < j} (l_{i,m} \cdot k_{m,j})$, con indici $i < j \quad \forall i, j \in [1, \dots, n]$, sono pari a zero che è la definizione di matrice triangolare inferiore.

(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:

$$\begin{aligned} (I_n + U)^{-1} &= \sum_{i=0}^n (-U)^i = \underbrace{(-U)^0}_{I_n} + (-U)^1 + (-U)^2 + \dots + (-U)^{n-1} + \underbrace{(-U)^n}_{0_{n \times n}} = \\ &= I_n - U + U^2 - \dots + (-1)^{n-1} \cdot U^{n-1} + 0_{n \times n} = I_n - U + U^2 - \dots + (-1)^{n-1} \cdot U^{n-1} \end{aligned}$$

che sono somme di matrici 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 la successione fino alla prima riga si ottiene la sommatoria:

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

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

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

quindi si ha che il numero di flop è circa $\frac{2}{3} \cdot n^3$, 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('La matrice 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('La matrice inserita 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

```

è possibile vedere il funzionamento di questa function nell'esercizio 14 a pagina 22.

3.6 Esercizio 6

Supponendo che in ingresso si abbiano le matrici di fattorizzazione LU con pivoting parziale di una matrice quadrata non singolare qualsiasi $A \in \mathbb{R}^{n \times n}$, rispettivamente:

- L matrice triangolare inferiore
- U matrice triangolare superiore
- P matrice delle permutazioni

e il vettore dei termini noti b , è possibile scrivere la function linLUP che risolve sistemi lineari:

```

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

```

Abbiamo sfruttato le function triInf e triSup, la loro implementazione si trova a fine capitolo a pagina 28.

3.7 Esercizio 7

Per dimostrare che la matrice $A \in \mathbb{R}^{n \times n}$ sia SDP deve sottostare a due proprietà:

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

Le matrici AA^T e $A^T A$ per essere SDP devono dimostrare le proprietà sopra:

- proprietà di simmetria:

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

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

la proprietà è quindi confermata;

- definita positiva:

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

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

anche questa proprietà è confermata.

Tenendo conto che la matrice A è non singolare se le sue righe sono linearmente indipendenti allora per ogni vettore x non nullo la combinazione lineare $\sum_{i=1}^n (x_i \cdot A_i)$ deve essere un vettore non nullo e vale anche l'inverso, possiamo quindi affermare che le matrici $A A^T$ e $A^T A$ sono simmetriche definite positive.

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:

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

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

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

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:

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

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

3.11 Esercizio 11

L'algoritmo di fattorizzazione LDL^T da noi implementato è il seguente:

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

è possibile vedere il funzionamento di questa function nell'esercizio 22.

3.12 Esercizio 12

Supponendo che in ingresso si abbiano le matrici di fattorizzazione LDL^T di una qualsiasi matrice $M \in \mathbb{R}^{n \times n}$ SDP, rispettivamente:

- L matrice triangolare inferiore a diagonale unitaria
- D matrice diagonale con elementi diagonali positivi

e b vettore dei termini noti, è possibile scrivere la function linLDL che risolve sistemi lineari:

```
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
```

Abbiamo sfruttato le function triInf, triSup e linDiag, la cui implementazione si trova a fine capitolo (pagina 28).

3.13 Esercizio 13

Per verificarlo abbiamo usato il seguente codice MatLab:

```
1 format short
2 disp('matrice A1');
3 A1 = [1,1,1,1;1,2,2,2;1,2,3,3;1,2,3,4]
4 [L1,D1] = factLDLT(A1)
5 disp('matrice A2');
6 A2 = [1,1,1,1;1,2,2,2;1,2,3,3;1,2,3,2]
7 [L2,D2] = factLDLT(A2)
```

Che restituisce l'output:

```
>> es13
matrice A1
A1 =
     1     1     1     1
     1     2     2     2
     1     2     3     3
     1     2     3     4

L1 =
     1     0     0     0
     1     1     0     0
     1     1     1     0
     1     1     1     1

D1 =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

matrice A2
A2 =
     1     1     1     1
     1     2     2     2
     1     2     3     3
     1     2     3     2

Error using factLDLT (line 15)
La matrice non e' SDP
Error in es13 (line 7)
[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 shortE
2
3 % 3.5 3.6
4 disp('Vettore residuo con fattorizzazione LU con pivoting parziale:');
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 % 3.11 3.12
12 disp('Vettore residuo con fattorizzazione LDLT');
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: La soluzione ottenuta

```

>> es14
Vettore residuo con fattorizzazione LU con pivoting parziale:
r =
    8.8818e-16
         0
    4.4409e-16
Vettore residuo con fattorizzazione LDLT:
r =
   -2.8614e-01
    3.2540e-01
   -4.4409e-16

```

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 = [1,1,1,1,1,1,1,1,1,1];
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

```

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);
9 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 tol=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 Risoluzione sistema lineare

```

1 function [x] = triInf(A, b)
2     x = b;
3     for j=1:length(A)
4         if A(j,j) ~= 0
5             x(j) = x(j)/A(j,j);
6         else
7             error('La matrice e'' singolare')
8         end
9         for i=j+1:length(A)
10            x(i) = x(i)-A(i,j)*x(j);
11        end
12    end
13 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

```

3.22.2 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

```

3.22.3 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 7 e 5, 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 11 e 9 non è molto rilevante. Infatti già con 4 ascisse abbiamo un'interpolazione quasi perfetta. Nelle figure 8 6 12 10 e' 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 13 grafico errori runge not-a-knot 15 grafico not-a-knot funzione 14 grafico errori not-a-knot funzione 16

4.6 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)

```

4.7 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);
```

4.8 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 17 per il grafico

4.9 Funzioni MatLab Usate

4.9.1 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-i));  
5         end  
6     end  
7 end
```

4.9.2 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
```

4.9.3 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
```

4.9.4 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
```

4.9.5 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
```

4.9.6 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
```

4.9.7 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 Capitolo 5

5.1 Esercizio 5.1

```
1 function [In] = trapeziComp(f,a,b,n)
2     h = (b-a)/n;
3     In = 0;
4     for i=1:n-1
5         In = In + f(a+i*h);
6     end
7     In = (h/2)*(2*In + f(a) + f(b));
8 end
```

```
1 function [In] = simpsonComp(f,a,b,n)
2     h = (b-a)/n;
3     In = f(a)-f(b);
4     for i=1:n/2
5         In = In + 4*f(a+(2*i-1)*h)+2*f((a+2*i*h));
6     end
7     In = In*(h/3);
8 end
```

5.2 Esercizio 5.2

```
1 format long
2 F = @(x) x*exp(1)^-x*cos(2*x);
3 y = (3*(exp(1)^(-2*pi) - 1) - 10*pi*exp(1)^(-2*pi))/25;
4 nmax = 8;
5 err = zeros(nmax,2);
6 rap = zeros(nmax-1,2);
7 for i=1:8
8     err(i,1) = abs(y - trapeziComp(F,0,2*pi,2^i));
9     err(i,2) = abs(y - simpsonComp(F,0,2*pi,2^i));
10    if i>1
11        rap(i-1,:) = err(i,:)./err(i-1,:);
12    end
13 end
14
15 semilogy([1:8],err);
16 plot([2:nmax],rap);
```

Al posto di riportare i dati in una tabella abbiamo ritenuto più opportuno mostrare l'andamento dell'errore mediante l'uso di grafici 19 20. L'andamento del rapporto tra gli errori è scorrelato per i primi 2^5 sottointervalli, ma dopo si stabilizza con un rapporto costante.

5.3 Esercizio 5.3

```
1 f = @(x) x*exp(-x)*cos(2*x);
2
3 [In,px] = simpsonAda(f,0,2*pi,10^-5, 5);
4
5 disp(In);
6 disp(px);
7
8 [In,px] = trapeziAda(f,0,2*pi,10^-5,3);
9
10 disp(In);
```

```
11 disp(px);
```

```
1 function [In] = simpsonComp(f,a,b,n)
2     h = (b-a)/n;
3     In = f(a)-f(b);
4     for i=1:n/2
5         In = In + 4*f(a+(2*i-1)*h)+2*f((a+2*i*h));
6     end
7     In = In*(h/3);
8 end
```

```
1 function [In,pt] = simpsonAda(f, a, b, tol)
2     pt=5;
3     h = (b-a)/6;
4     m = (a+b)/2;
5     m1 = (a+m)/2;
6     m2 = (m+b)/2;
7     In1 = h*(feval(f, a) + 4*feval(f, m) + feval(f, b));
8     In = In1/2 + h*(2*feval(f, m1) + 2*feval(f, m2) - feval(f, m));
9     err = abs(In-In1)/15;
10    if err>tol
11        [intSx, ptSx] = simpsonAdattativaRicorsiva(f, a, m, tol/2, 1);
12        [intDx, ptDx] = simpsonAdattativaRicorsiva(f, m, b, tol/2, 1);
13        In = intSx+intDx;
14        pt = pt+ptSx+ptDx;
15    end
16 end
```

5.4 Esercizio 5.4

```
1 function [xn, i, err] = jacobi(A, b, x0, tol, nmax)
2     D = diag(diag(A));
3     J = -inv(D)*(A-D);
4     q = D\b;
5     xn = J*x0 + q;
6     i = 1;
7     err(i) = norm(xn-x0)/norm(xn);
8     while (i<=nmax && err(i)>tol)
9         x0 = xn;
10        xn = J*x0+q;
11        i = i+1;
12        err(i) = norm(xn-x0)/norm(xn);
13    end
14    if i>nmax
15        disp('Jacobi non converge nel numero di iter fissato');
16    end
17 end
```

```
1 function [xn, i, err] = gaussSeidel(A, b, x0, tol, nmax)
2     D=diag(diag(A));
3     L=tril(A)-D;
4     U=triu(A)-D;
5     DI=inv(D+L);
6     GS=-DI*U;
7     b1=(D+L)\b;
8     xn=GS*x0+b1;
```



```

9      i=1;
10     err(i)=norm(xn-x0,inf)/norm(xn);
11
12     while(err(i)>tol && i<=nmax)
13         x0=xn;
14         xn=GS*x0+b1;
15         i=i+1;
16         err(i)=norm(xn-x0,inf)/norm(xn);
17     end
18     if i>nmax
19         error('Gauss-Seidel non converge nel numero di iterazioni fissato');
20     end
21     i=i-1;
22 end

```

5.5 Esercizio 5.5

```

1  A = [-4,2,1;1,6,2;1,-2,5];
2  b = [1,2,3]';
3  x0 = [0,0,0]';
4
5  [z,j,jerr] = jacobi(A,b,x0,1.e-3,25)
6  [y,i,gerr] = gaussSeidel(A,b,x0,25,1.e-3)

```

Il sistema $Ax = b$ è formato dagli elementi:

$$A = \begin{pmatrix} -4 & 2 & 1 \\ 1 & 6 & 2 \\ 1 & -2 & 5 \end{pmatrix}$$

$$b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

partendo dal vettore iniziale

$$x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

il metodo di Jacobi mi restituisce in output il vettore:

$$\begin{pmatrix} -0.02682 \\ 0.1201 \\ 0.6534 \end{pmatrix}$$

con il metodo di Gauss-Seidel si ottiene invece il vettore:

$$\begin{pmatrix} -0.02688 \\ 0.12 \\ 0.6532 \end{pmatrix}$$

Oltre a calcolare il vettore risultante, le due funzioni usate nel codice mi dicono anche quante iterazioni avvengono:

Metodo	Iterazioni
Jacobi	12
Gauss-Seidel	8

5.6 Esercizio 5.6

```
1 H = [0,0,0,0,0;1,0,1,0,0;1,1,0,0,0;0,1,0,0,0;0,1,0,0,0];
2 p=0.85;
3
4
5 [n,m] = size(H);
6 if(n~=m), error('Matrice non quadrata'); end
7 s = sum(H);
8 S=zeros(n,n);
9 for i=1:n
10     if s(i)~=0
11         S(:,i)=H(:,i)/s(i);
12     else
13         S(:,i)=(1/n);
14     end
15 end
16 A= eye(n) - p*S;
17 b = ((1-p)/n).*ones(n,1);
18 tols= logspace(-1,-10,10);
19 iters = zeros(10,3);
20
21 for i=1:10
22     v=zeros(n,4);
23     [v(:,1),iters(i,1)]=PotenzePR(S,p,tols(i));
24     [v(:,2),iters(i,2)]=jacobi(A,b,ones(n,1), tols(i), 10000);
25     [v(:,3),iters(i,3)]=gaussSeidel(A,b,ones(n,1), tols(i), 10000);
26 end
27 plot(iters)
```

Nel grafico 18 é mostrato l'andamento dei vari metodi numerici per il calcolo dell'autovettore. Si nota come al crescere della tolleranza i metodi di Jacobi e delle Potenze non divergano sostanzialmente, mentre il metodo di Gauss-Seidel mostra una maggior efficienza anche per valori di tolleranza ridotti.

6 Grafici

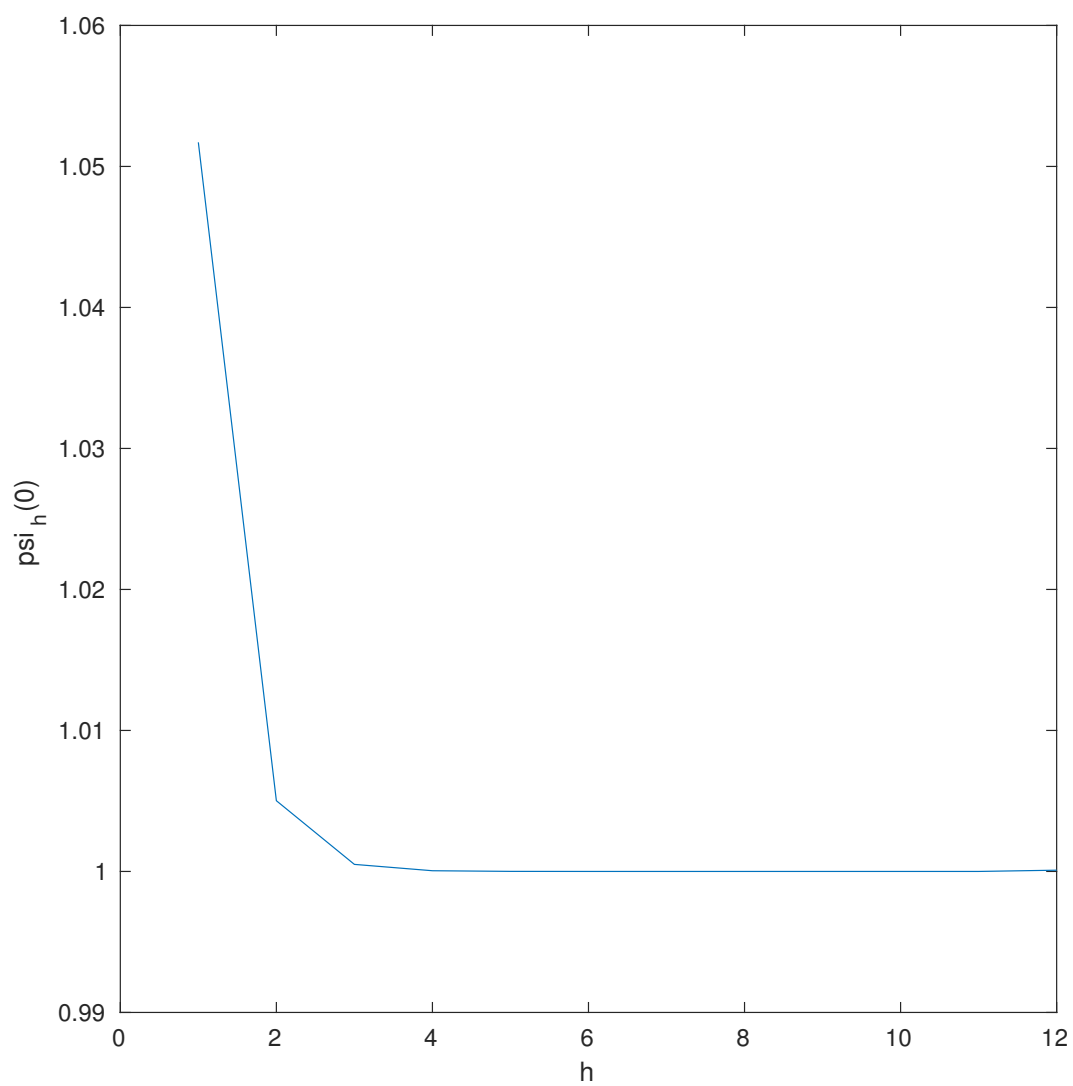


Figure 1: Andamento della funzione $\Psi_h(0)$

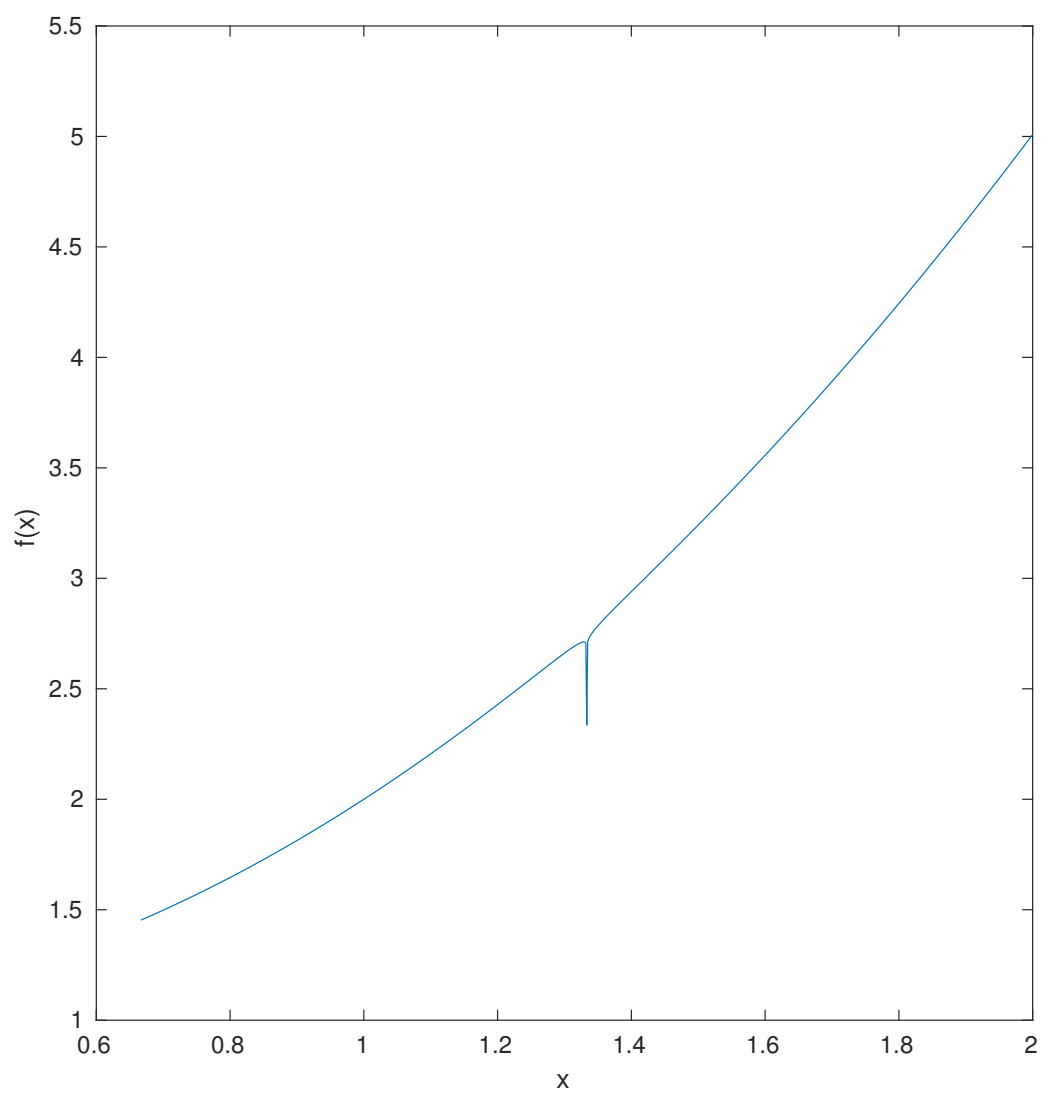


Figure 2: Plot MatLab della funzione $f(x) = \frac{\ln(3(1-x)+1)}{80} + x^2 + 1$

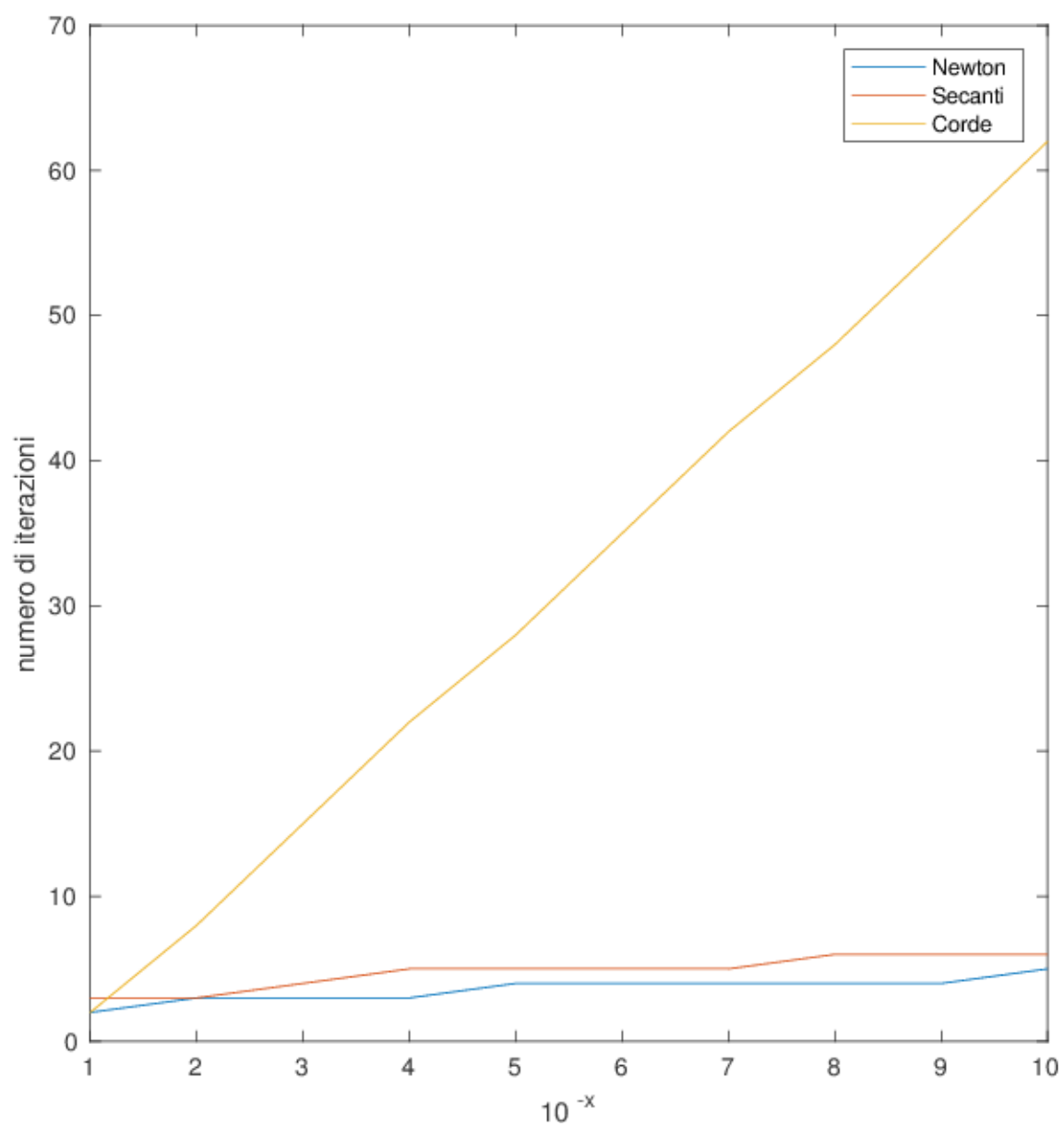


Figure 3: Andamento del numero delle iterazioni al decrescere della tolleranza per i metodi Newton, Secanti, Corde

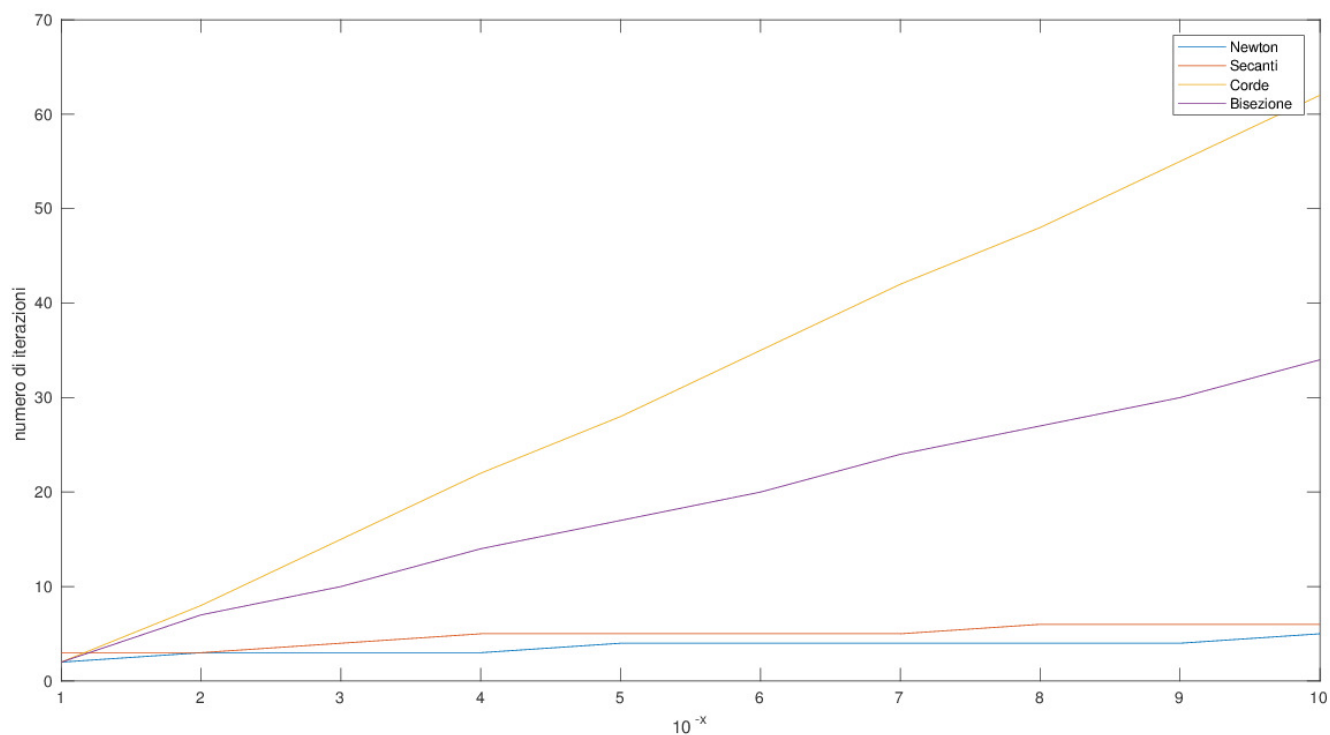


Figure 4: Aggiunta dell'andamento del metodo di Bisezione rispetto ai precedenti metodi

Figure 5: Runge Chebyshev

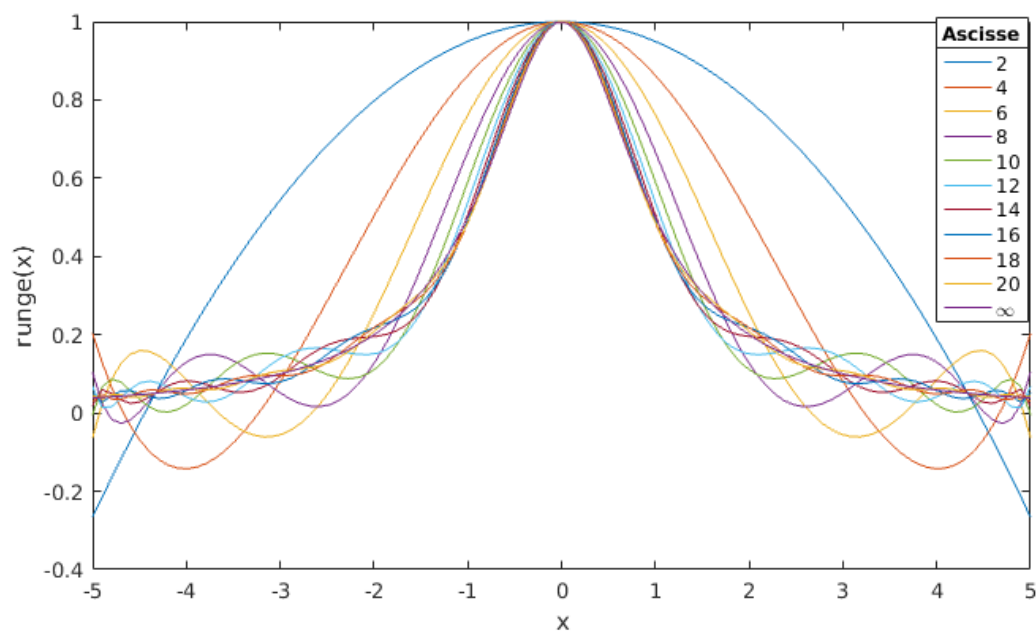


Figure 6: Runge Chebyshev Errori

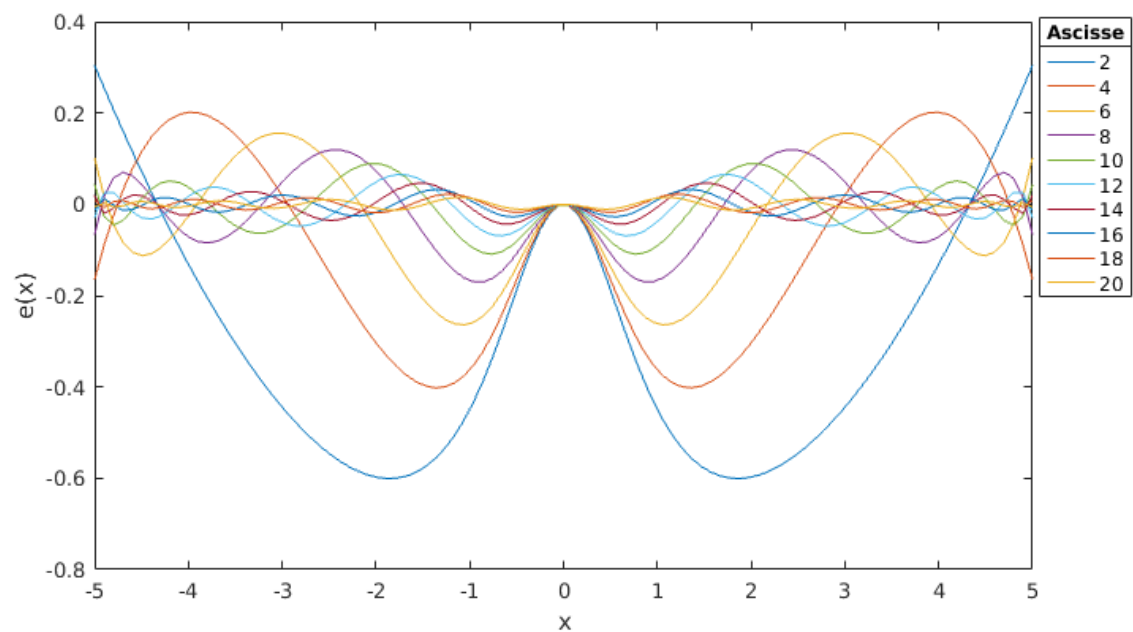


Figure 7: Runge con ascisse equidistanti

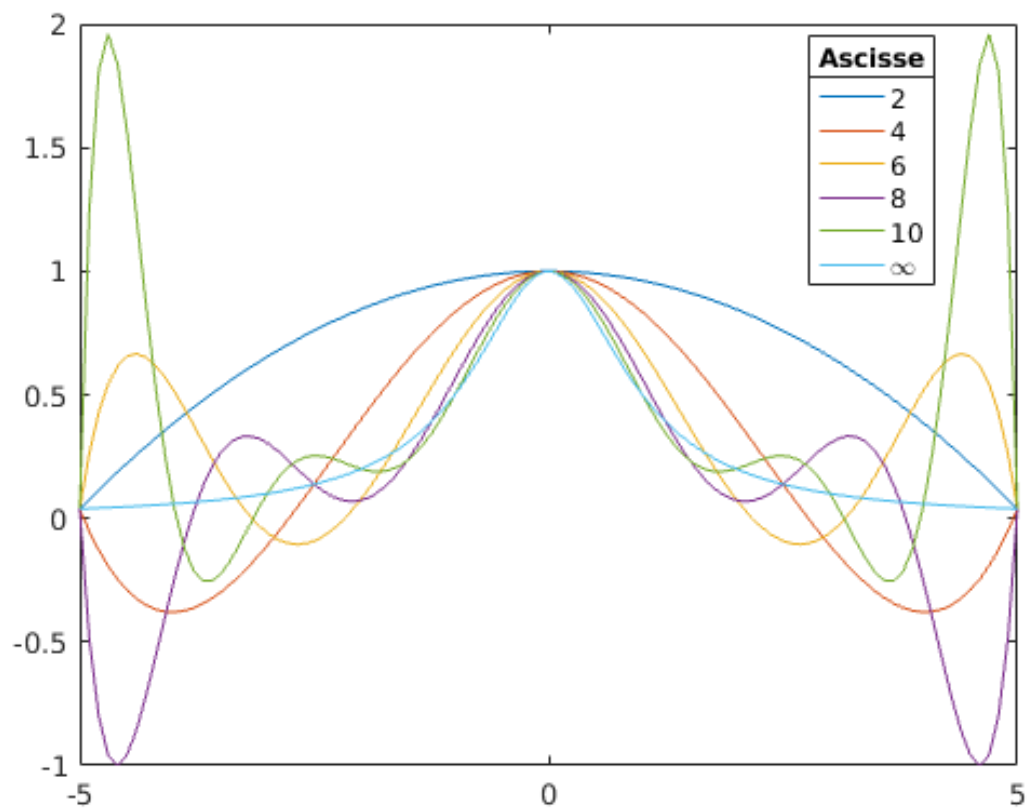


Figure 8: Runge con ascisse equidistanti errori

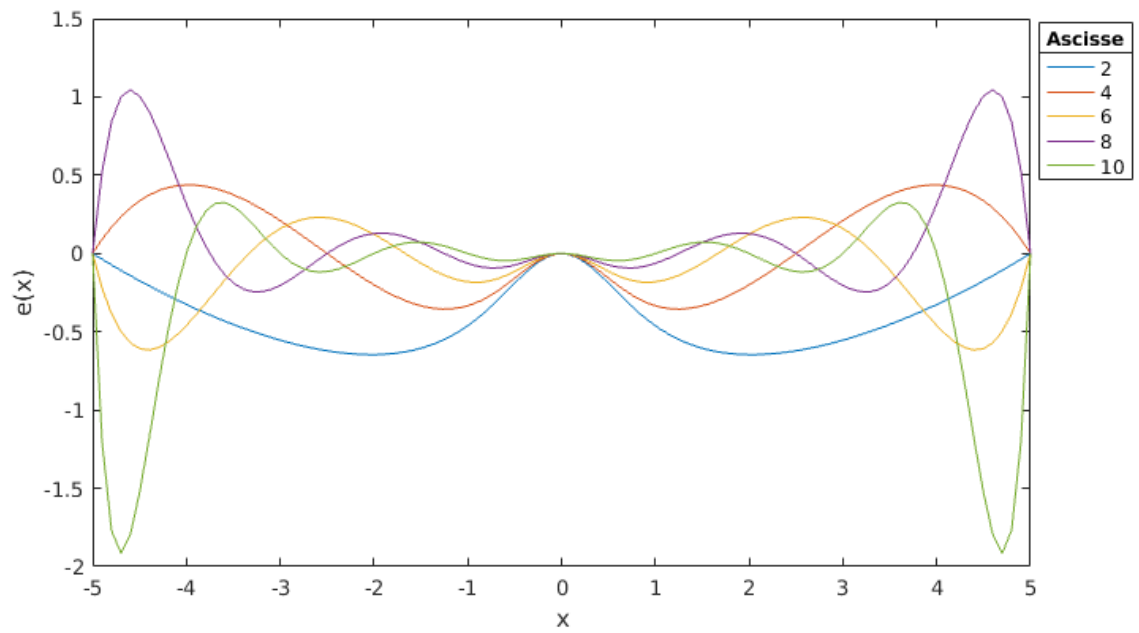


Figure 9: Funzione xsinx Chebyshev

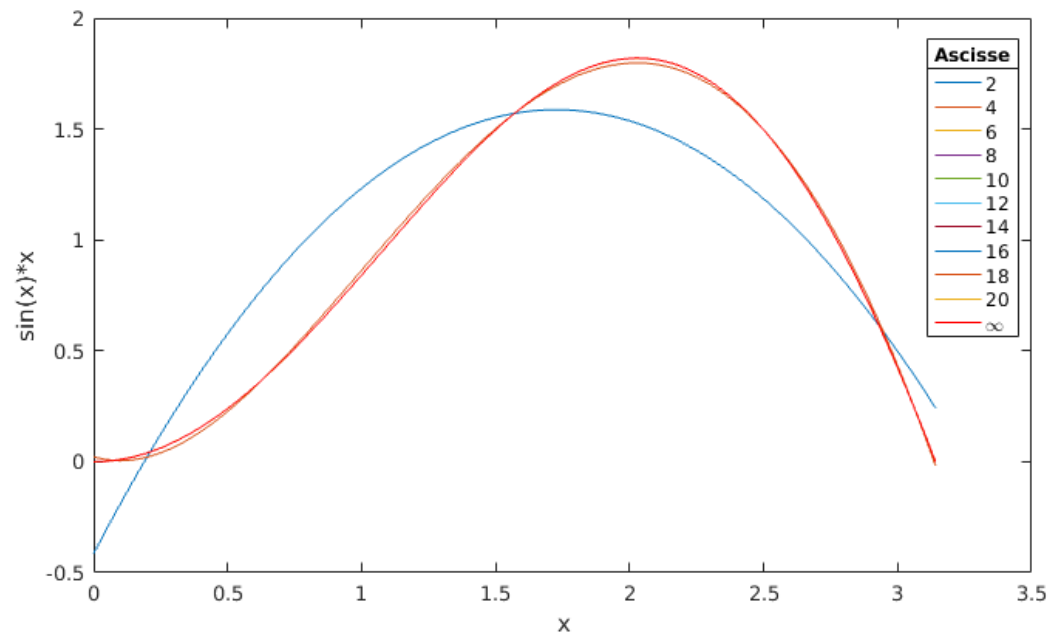


Figure 10: Funzione $x \sin x$ Chebyshev errori

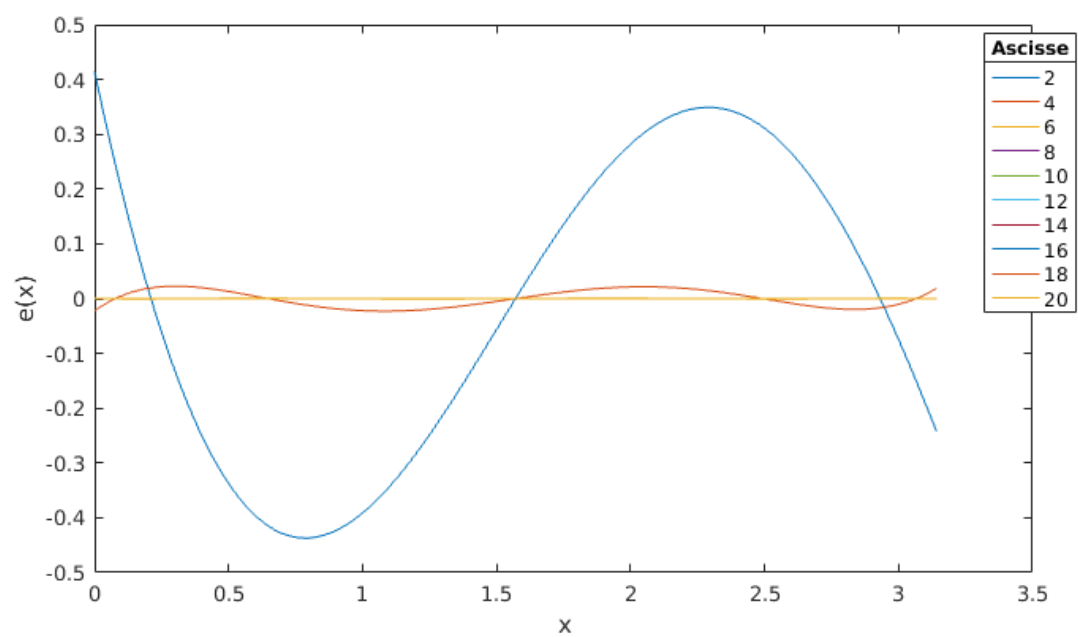


Figure 11: Funzione $x \sin x$ ascisse equidistanti

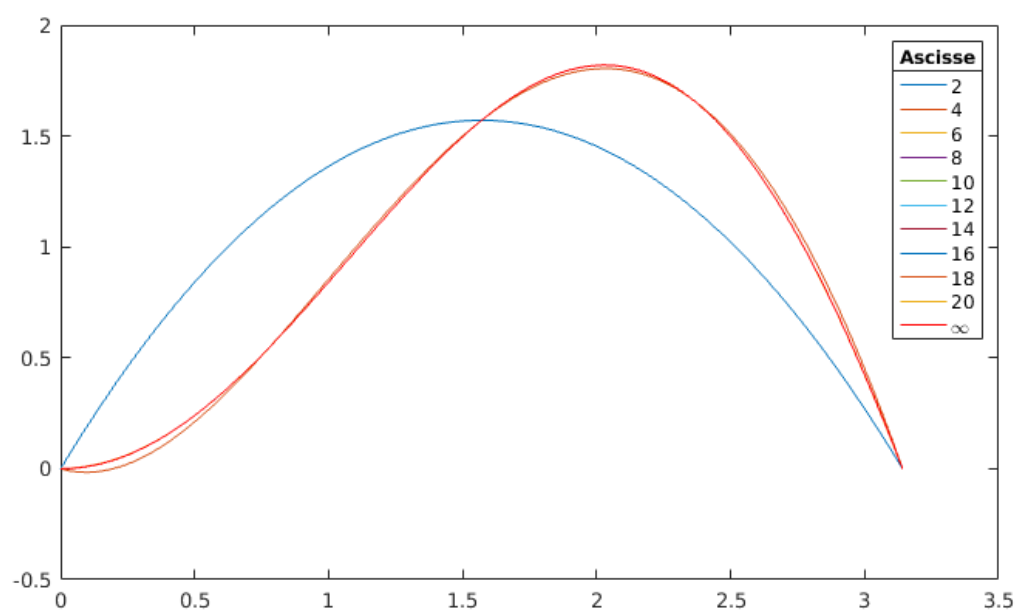


Figure 12: Funzione $x \sin x$ ascisse equidistanti errori

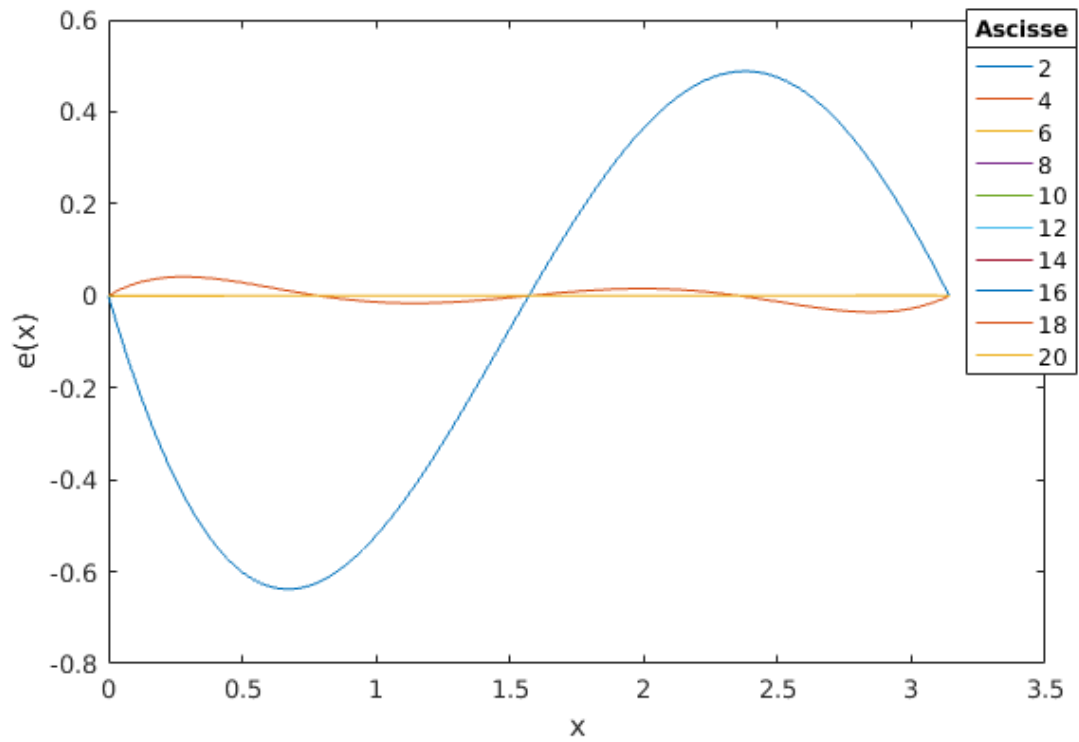


Figure 13: Runge not-a-knot

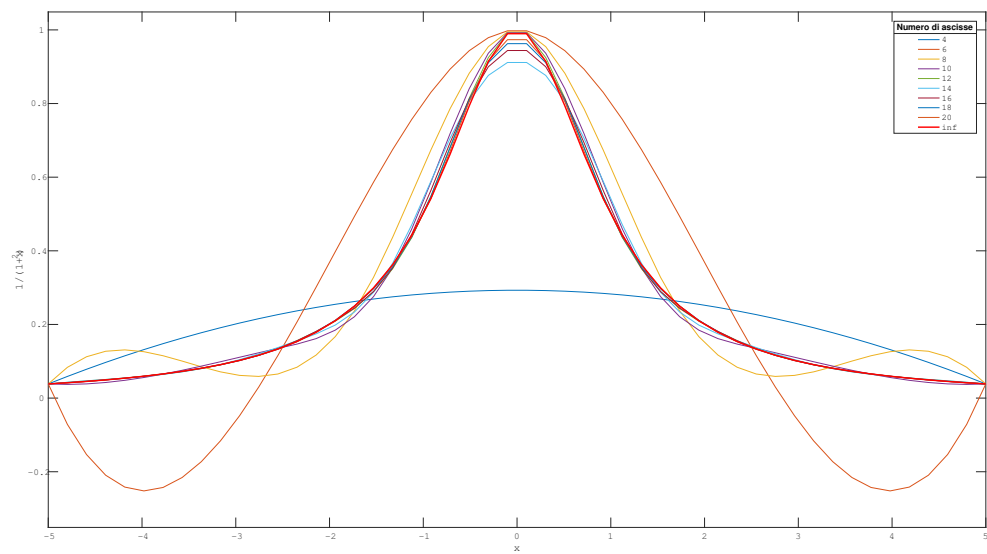


Figure 14: Funzione xsinx not-a-knot

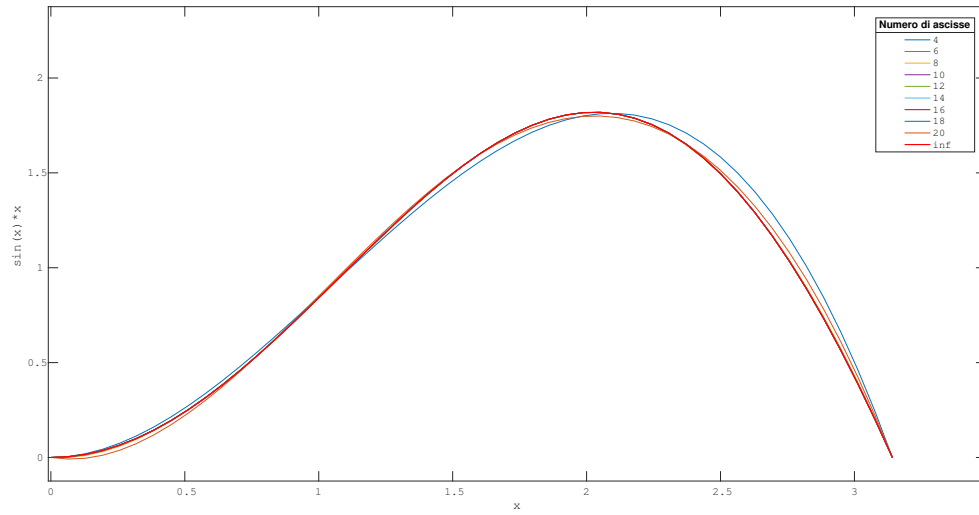


Figure 15: Errori Runge

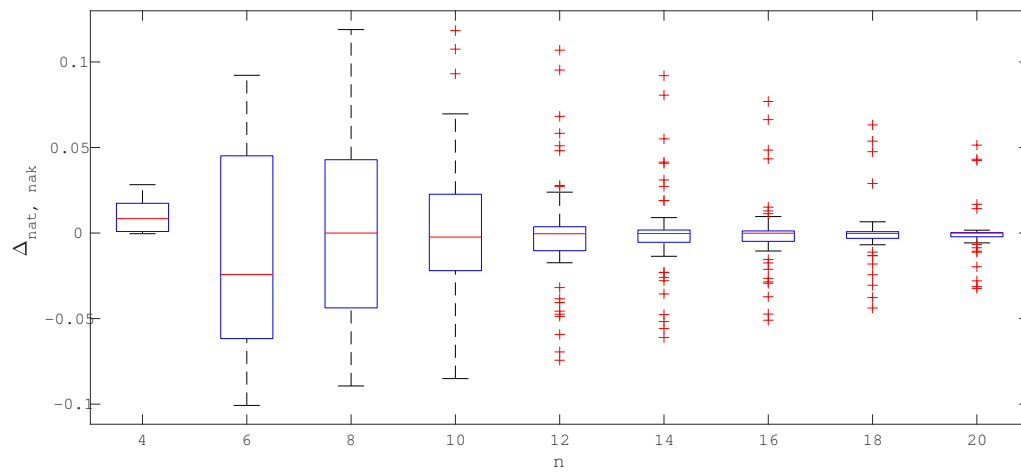


Figure 16: Errori funzione xsinx

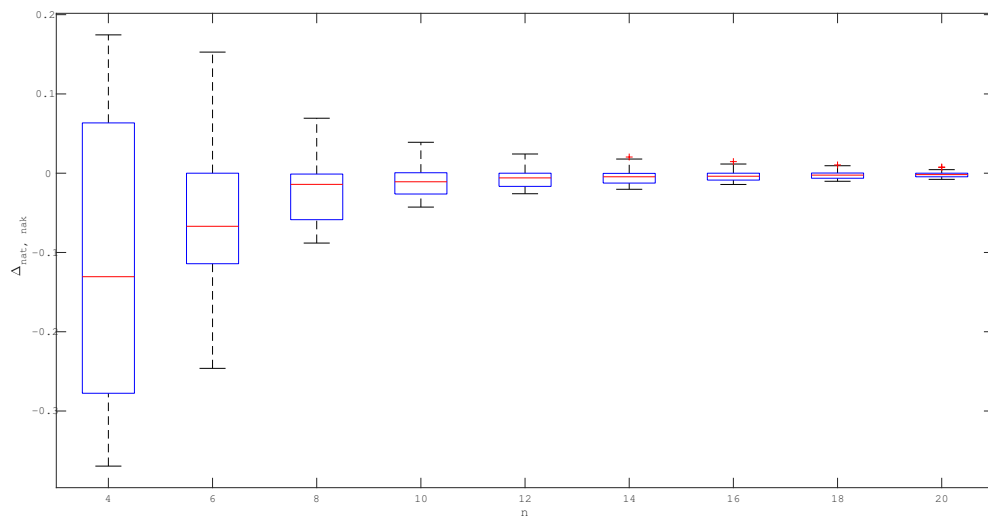


Figure 17: Esercizio 4.10

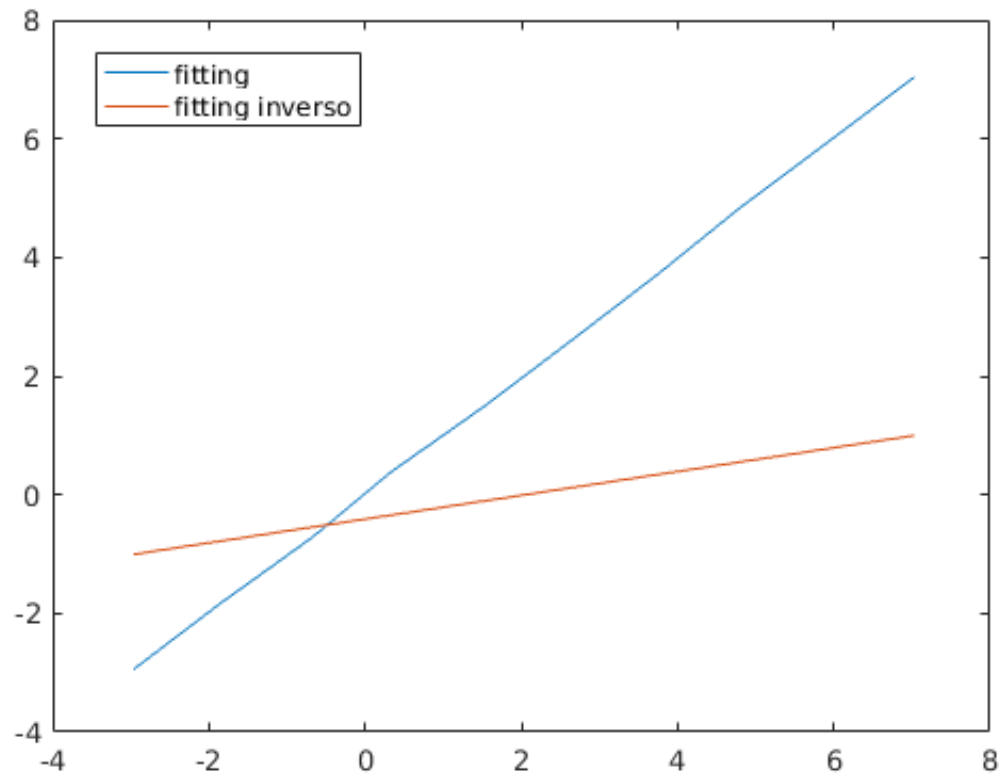


Figure 18: Esercizio 5.2

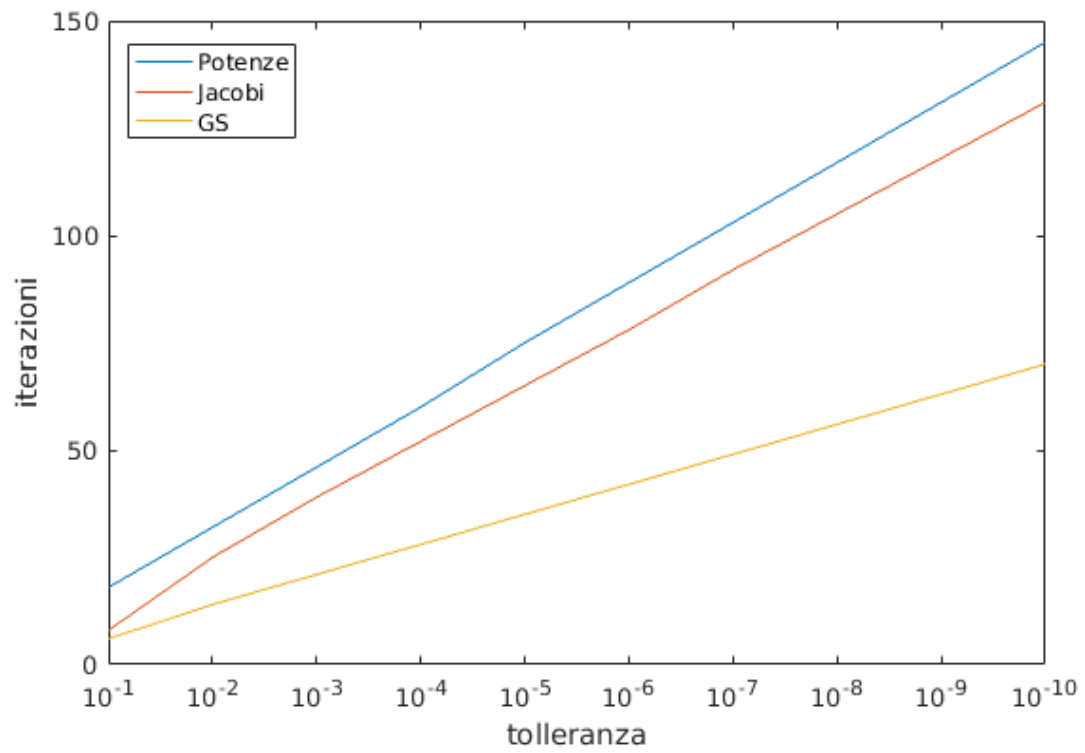


Figure 19: Esercizio 5.2

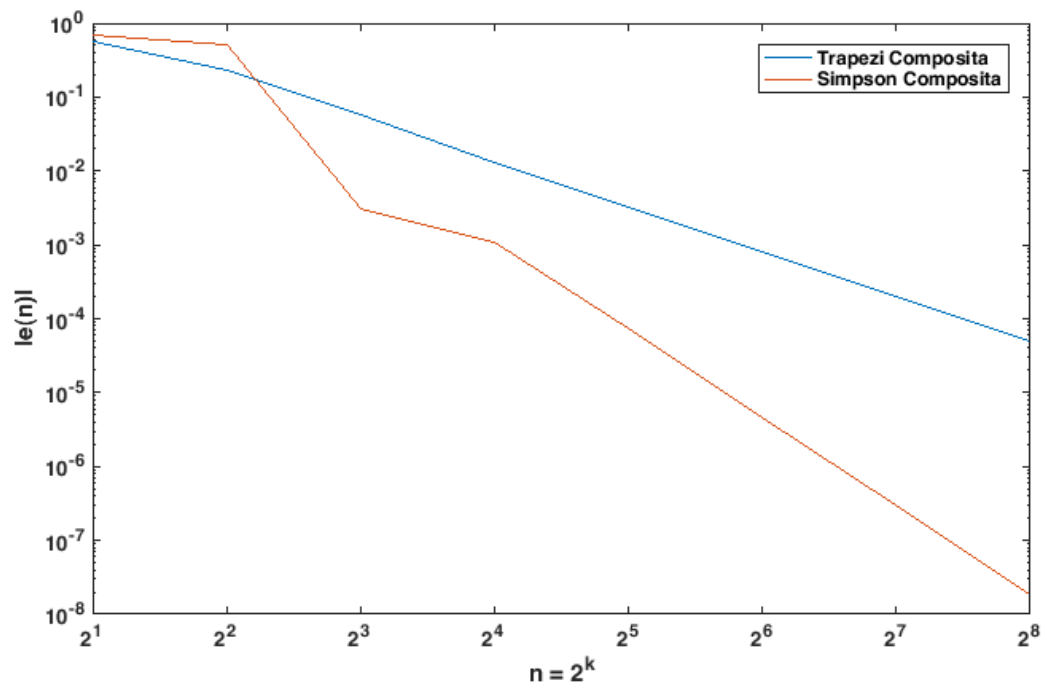


Figure 20: Esercizio 5.6

