Elaborato di

Calcolo Numerico

Anno Accademico 2016/2017

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

October 5, 2017

Capitoli

1 Capitolo 1

1.1 Esercizio 1

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

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

inoltre per definizione di Φ si calcola il limite:

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

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

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

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 $-32768 \leq$ `numero` ≤ 32767 ($-2^{15} \leq$ `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:

$$(32767)_{10}+(1)_{10} = (0111111111111111)_{2,MS}+(0000000000000001)_{2,MS} = (1000000000000000)_{2,MS} = (-32768)_{10}$$

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

$$(-32768)_{10}-(1)_{10} = (1000000000000000)_{2,MS}-(0000000000000001)_{2,MS} = (0111111111111111)_{2,MS} = (32767)_{10}$$

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:

```
format long e;

h=zeros(12,1);
f=zeros(12,1);

for i=1:12
    h(i)= power(10,-i);
end

for j=1:12
```

```
    f(j)=lim(0,h(j));
end

f

function p=lim(x,y)
    p=(exp(x+y) - exp(x))/y;
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 l'errore inizia a crescere. Mostriamo l'andamento relativo nel seguente plot:

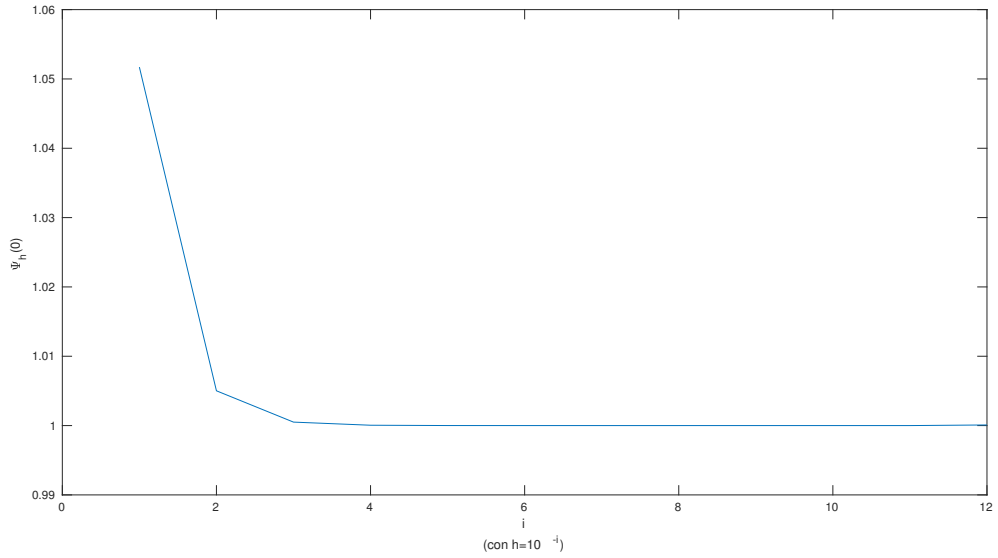


Figure 1: Andamento della funzione $\Psi_h(0)$

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:

$$\frac{f(x_0 + h) - f(x_0 + h)}{2h} =$$

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

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

nel secondo caso:

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

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

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

Abbiamo quindi dimostrato che:

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

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

1.6 Esercizio 6

Il codice MatLab, indicando con x=x_n e r=ϵ:

```
format longEng

conv=sqrt(2);
x=[2,1.5];
r=[x(1)-conv,x(2)-conv];

for i= 2:6
    x(i+1) = (x(i)*x(i-1)+2)/(x(i)+x(i-1));
end

for i=3:7
    r(i)=x(i)-conv;
end

x
r
```

restituisce i valori:

n	x_n	ϵ
0	2.00000000000000e+000	585.786437626905e-003
1	1.50000000000000e+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 $log_{10}u >> 1$ 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 delta 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:

$$\left[\frac{1}{10}\right]_{10} = \left[0, \overline{00011}\right]_2$$

$$\left[0, \overline{00011}\right]_2 + \left[0, \overline{00011}\right]_2 + \underbrace{...}_{6volte} + \left[0, \overline{00011}\right]_2 + \left[0, \overline{00011}\right]_2 =$$

$$= [1, 00010]_2 \approx [1.0625]_{10} \neq [1.0000]_{10}$$

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:

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

- seguendo gli stessi procedimenti del punto precedente possiamo scrivere:

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

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 $f'(x) = \frac{1}{2 \cdot \sqrt{x}}$, 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 $\left|3(1-\frac{4}{3})+1\right|$ 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:

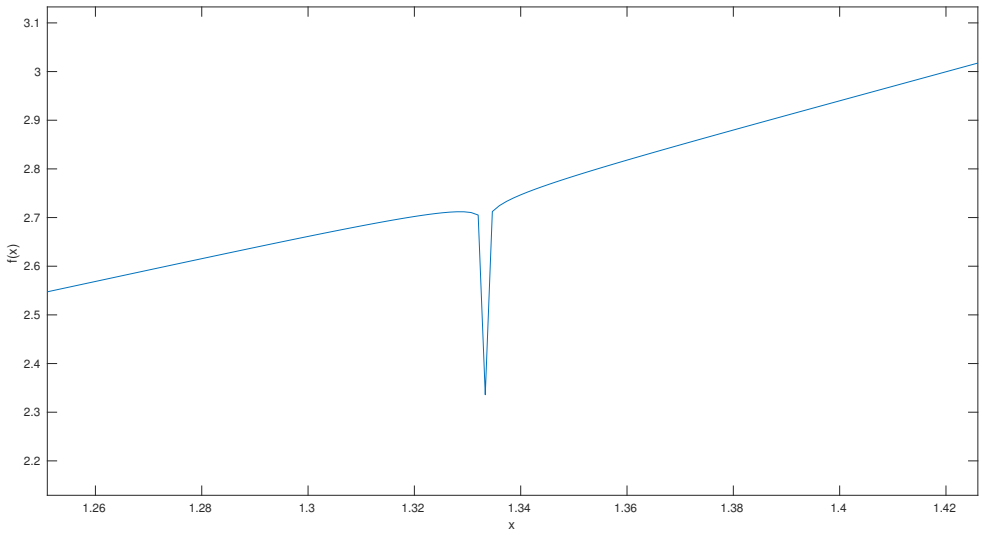


Figure 2: `Plot MatLab della funzione` $f(x) = \frac{ln(|3(1-x)+1|)}{80} + x^2 + 1$

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 :
   1.333333333333333
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

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:

$$x = \sqrt{\alpha}$$
$$x^2 = \left(\sqrt{\alpha}\right)^2$$
$$x^2 - \alpha = 0$$

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 = NewtonSqrt(alpha, x_0, imax, tol):

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

```
function y = NetwtonSqrt(alpha, x0, imax, tol)
    format long e
    x = [x0,(x0+alpha/x0)/2];
    i = 2;
    while(i < imax) && (abs(x(i)-x(i-1))>tol)
        x(i+1) = (x(i) + alpha/x(i))/2;
        disp(x(i+1));
        i = i+1;
    end
    y = x(i);
end
```

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

i	x_i
1	1.750000000000000e+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:

$$x = \sqrt[n]{\alpha}$$
$$x^n = \left(\sqrt[n]{\alpha}\right)^n$$
$$x^n - \alpha = 0$$

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

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

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

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 = NewtonSqrt(n, alpha, x_0, imax, tol). Lo script MatLab corrispondente ai casi $n = 3, 4, 5$ è il seguente:

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

```
function y = NetwtonSqrtN(n, alpha, x0, imax, tol)
    format long e
    x = [x0, (((n-1)*x0^n + alpha)/ x0^(n-1)) / n];
    i = 2;
    while(i < imax) && (abs(x(i)-x(i-1))>tol)
        x(i+1) = (((n-1)*x(i)^n + alpha)/ (x(i)^(n-1)) / n);
        disp(x(i+1));
        i = i+1;
    end
    y = x(i);
end
```

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:

$$x = \sqrt{\alpha}$$
$$x^2 = \left(\sqrt{\alpha}\right)^2$$
$$x^2 - \alpha = 0$$

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 = SecSqrt(alpha, x_0, imax, tol):

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

```
function x = SecSqrt(x0,alpha,imax,tol)
   x1 = NewtonSqrt(alpha,x0,1,0.00000001);
   x = [x0,x1,(alpha+x1*x0)/(x1+x0)];
```

```
    i = 3;
    while(i < imax) && (abs(x(i)-x(i-1))>tol)
        x(i+1) = (alpha + x(i)*x(i-1))/(x(i)+x(i-1));
        disp(x(i+1));
        i = i+1;
    end
    y = x(i);
end
```

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). Scritte le Function, le abbiamo usate nel seguente script:

```
funct = @(x) (x-pi)^10;
dfunct = @(x) 10*(x-pi)^9;

disp('metodo newton funct 1');
for i=0:5
    N1(i+1) = Newton(funct, dfunct, 5, 50, 10^(-i));
end

disp('newton modificato funct 1');
for i =0:5
    NM1(i+1) = NewtonMod(funct, dfunct, 1, 5, 50, 10^(-i));
end

disp('aitken funct 1');
for i = 0:5
    A1(i+1) = Aitken(funct, dfunct, 5, 50, 10^(-i));
end


funct = @(x) ((x-pi)^10)*(exp(1)^(2*x));
dfunct = @(x) (5+x-pi)*(x-pi)^9*2*exp(1)^(2*x);

disp('metodo newton funct 2');
for i=0:5
    N2(i+1) = Newton(funct, dfunct, 5, 50 ,10^(-i));
end

disp('newton modificato funct 2');
for i=0:5
    NM2(i+1) = NewtonMod(funct, dfunct, 1, 5, 50, 10^(-i));
end

disp('aitken funct 2');
for i=0:5
```

```
    A2(i+1) = Aitken(funct, dfunct, 5, 50, 10^(-i));
end
```

```
function [x, i] = Newton(fx,dfx, x0, imax, tolx)
fx0 = feval(fx,x0);
dfx0 = feval(dfx,x0);
x = x0 - fx0/dfx0;
i=0;
while (i<imax) & (abs(x-x0)>tolx)
    i=i+1;
    x0=x;
    fx0=feval(fx,x0);
    dfx0=feval(dfx,x0);
    x=x0-fx0/dfx0;
end

if abs(x-x0)>tolx
    disp('il metodo non converge');

end
```

```
function x = NewtonMod(fx, dfx, m, x0, imax, tol)
fx0 = feval(fx,x0);
dfx0 = feval(dfx,x0);
x = x0 - m * fx0 / dfx0;
i = 0;
while((i < imax) & abs(x-x0) > tol)
    i = i+1;
    x0=x;
    fx0 = feval(fx, x0);
    dfx0 = feval(dfx, x0);
    x = x0 - m * fx0 / dfx0;
end
if abs(x-x0)>tol
    disp('il metodo non converge');
end
```

```
function y = Aitken( fx, dfx, x0, imax, tol )
    i = 0;
    x=x0;
    diverror=1;
    while((i < imax) && diverror)
        i=i+1;
        x0=x;
        fx0 = feval(fx,x0);
        dfx0 = feval(dfx,x0);
        x1 = x0 - fx0/dfx0;
        fx0 = feval (fx,x1);
        dfx0 = feval (dfx, x1);
        x = x1 - fx0/dfx0;
        if (x-2*x1+x0 == 0)
            disp('divisione per zero');
            diverror = 0;
            break;
        end
        x = (x*x0-x1^2)/(x-2*x1+x0);
        diverror = abs(x-x0)>tol;
    end
```

```
    if(diverror)
        disp('Il metodo non converge');
    end
    y = x;
end
```

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}	3.150212685926121	3.150212685926121	3.141592653589783
10^{-5}	3.150212685926121	3.150212685926121	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}	3.154537411727535	3.154537411727535	3.141592492095279
10^{-4}	3.154537411727535	3.154537411727535	3.141592516390607
10^{-5}	3.154537411727535	3.154537411727535	3.141592516390607

Abbiamo poi riportato il plot matlab relativo ai precedenti valori:

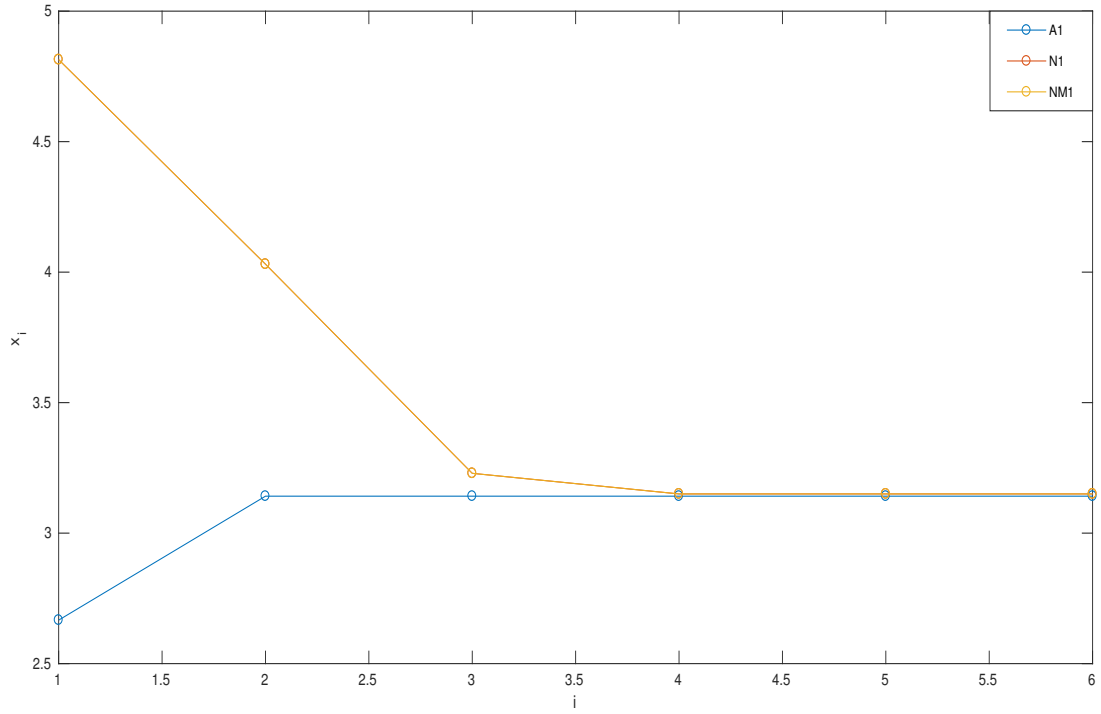


Figure 3: Andamento del calcolo degli zeri della funzione $f_1(x) = (x - \pi)^{10}$ al variare della tolleranza

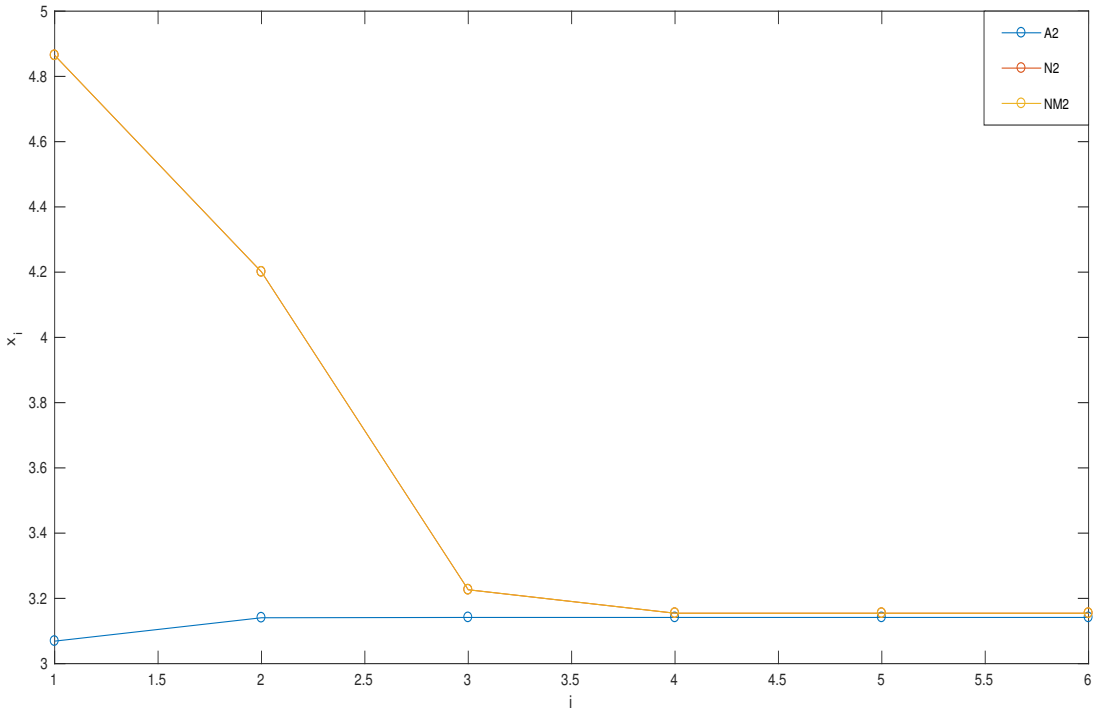


Figure 4: Andamento del calcolo degli zeri della funzione $f_2(x) = (x-\pi)^{10} \cdot e^{2 \cdot x}$ al variare della tolleranza

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:

```
f = @(x) (1-x-(1+cos(10*x)/2)*sin(x));
df = @(x) (5*sin(x)*sin(10*x)-cos(x)*(cos(10*x)/2 + 1)-1);

x0 = 0;
x1 = 1;
tol = logspace(-1,-10,10);
imax = 100;
iter = zeros(10,3);
res = zeros(10,3);
for i=1:10
    [res(i,1), iter(i,1)] = Newton(f,df,x0,imax,tol(i));
    [res(i,2), iter(i,2)] = secanti(f,df,x0,imax, tol(i));
    [res(i,3), iter(i,3)] = corde(f,df,x0,imax, tol(i));
end

```

```
disp('La prima colonna sono riferite al metodo di Newton (iterazioni e risultato)');
disp('La seconda colonna sono riferite al metodo delle secanti (iterazioni e risultato)')
    ;
disp('La terza colonna sono riferite al metodo delle corde (iterazioni e risultato)');
iter
res
```

```
function [x, i] = secanti(f,df,x0,imax, tol)
fx = feval(f,x0);
f1x = feval (df, x0);
x = x0 - fx/f1x;
i = 0;
while (i<imax) & (abs(x-x0)>tol)
    i = i+1;
    fx0 = fx;
    fx = feval(f,x);
    x1 = (fx*x0-fx0*x)/(fx-fx0);
    x0 = x;
    x = x1;
end
if (abs(x-x0)>tol), disp('il metodo non converge'), end
end
```

```
function [x,i]=corde(fx,dfx,x0,imax,tol)
fx0 = feval(fx,x0);
dfx0 = feval(dfx,x0);
x = x0 - fx0/dfx0;
i=0;
while (i<imax) & (abs(x-x0)>tol)
    i=i+1;
    x0=x;
    fx0=feval(fx,x0);
    x=x0-fx0/dfx0;
end

if abs(x-x0)>tol
    disp('il metodo non converge');
end
```

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 tramite il plot MatLab:

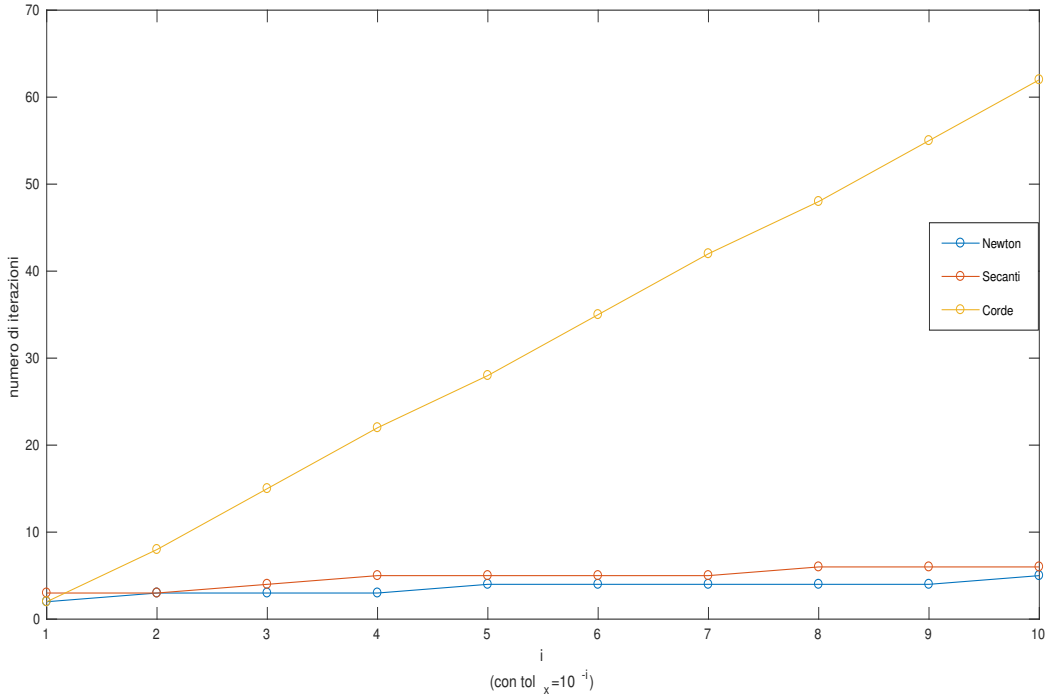


Figure 5: `Andamento del numero delle iterazioni al decrescere della tolleranza per i metodi Newton, Secanti, Corde`

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:

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

questo significa che il problema è ben condizionato.

2.7 Esercizio 7

Lo script usato è il seguente:

```
format long

f = @(x) (1-x-(1+cos(10*x)/2)*sin(x));
df = @(x) (5*sin(x)*sin(10*x)-cos(x)*(cos(10*x)/2 + 1)-1);

tol = logspace(-1,-10,10);

for i=1:10
    [res, iterB(i,1)] = bisect(f,0,1,tol(i));
end

```

```
disp('il numero di iterazioni del metodo di bisezione con tolleranza decrescente da
    10^(-1) a 10^(-10) e'' il seguente');
iterB

for i=1:10
    iter(i,4) = iterB(i);
end

plot(iter)
xlabel('10^{-x}');
ylabel('numero di iterazioni');
```

```
function [x,i]=bisect(f,a,b,tol)
    fa = feval (f,a);
    fb = feval (f,b);
    x = (a+b)/2;
    fx = feval(f,x);
    imax = ceil (log2(b-a) - log2(tol));
    for i=2:imax
        f1x = abs((fb-fa)/(b-a));
        if abs(fx)<=tol*f1x
            break
        elseif fa*fx<0
            b=x;
            fb=fx;
        else
            a=x;
            fa=fx;
        end
        x = (a+b)/2;
        fx = feval(f,x);
    end
end
```

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 nel seguente plot MatLab:

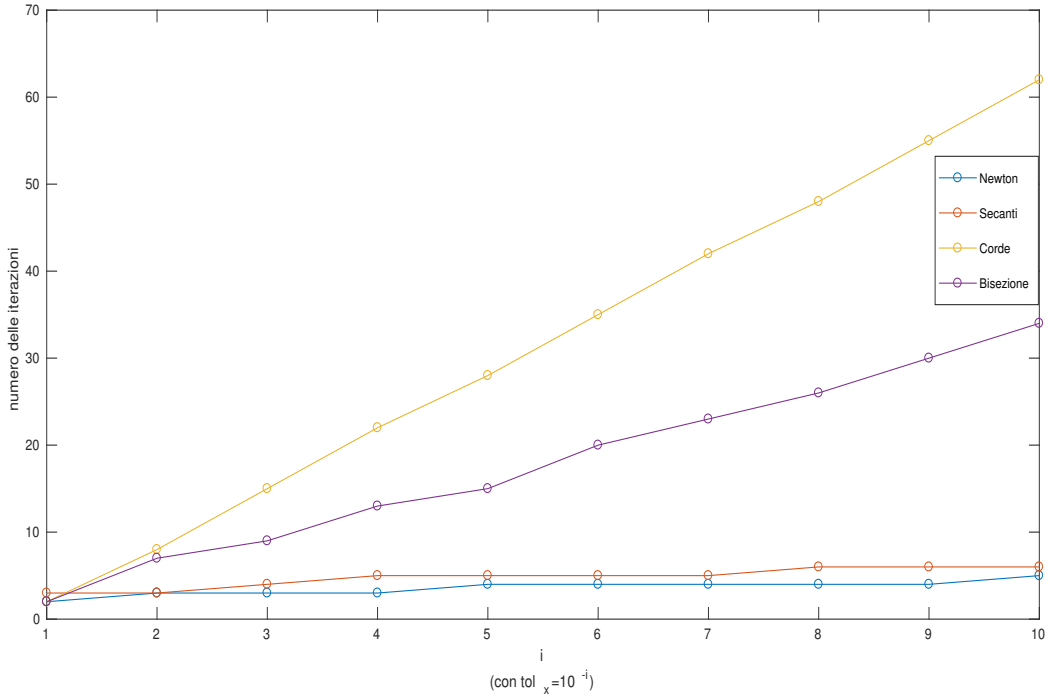


Figure 6: `Aggiunta dell'andamento del metodo di Bisezione rispetto ai precedenti metodi`

2.8 Esercizio 8

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

```
f = @(x) (x-pi)*(exp(1)^(10*x));
df =  @(x) (exp(1)^(10*x))*(10*x-10*pi+1);

disp('Con punto iniziale x0=0');
[res,nit] = Newton(f,df, 0, 100, 10^(-2))
disp('Con punto iniziale x0=4');
[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 $x0 = 0$, ma con punto iniziale diverso $x0 = 4$ 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.

3 Capitolo 3

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:

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

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:

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

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:

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

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 $i < j \ \forall i, j \in [1, .., n]$, 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:

$$l_{i,i} = 1 \qquad \forall i \in [1, .., n]$$

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:

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

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 $diag(D) = diag(A)$, 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:

$$(I_n + U)^{-1} = \sum_{i=0}^{n}(-U)^n = \underbrace{(-U)^0}_{I_n} + (-U)^1 + (-U)^2 + ... + (-U)^{n-1} + \underbrace{(-U)^n}_{0_{n\times n}} =$$

$$= I_n - U + U^2 - ... + (-1)^{n-1} \cdot U^{n-1} + 0_{n\times n} = I_n - U + U^2 - ... + (-1)^{n-1} \cdot U^{n-1}$$

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 :

$$2\cdot \sum_{j=n+1}^{1} j(j-1) = 2\cdot\Big(\sum_{j=1}^{n+1}(j^2-j)\Big) = 2\cdot\Big(\sum_{j=1}^{n+1}(j^2) - \sum_{j=1}^{n+1}(j)\Big) = 2\cdot\Big(\sum_{j=1}^{n-1}(j^2) + n^2 + (n+1)^2 - \sum_{j=1}^{n-1}(j) - n - n - 1\Big) =$$

$$= 2\cdot\Big(\frac{n\cdot(n-1)\cdot(2n-1)}{6} + 2\cdot n^2 + 2\cdot n + 1 - \frac{n\cdot(n-1)}{2} - 2\cdot n - 1\Big) = 2\cdot\Big(\frac{2\cdot n^3 - 3\cdot n^2 + n}{6} + 2\cdot n^2 - \frac{n^2-n}{2}\Big) =$$

$$= 2\cdot\Big(\frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6} + 2\cdot n^2 - \frac{n^2}{2} + \frac{n}{2}\Big) = 2\cdot\Big(\frac{n^3}{3} + n^2 + \frac{2}{3}\cdot n\Big) \approx \frac{2}{3}\cdot n^3$$

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
```

```
L=eye(n);
P=eye(n);
U=A;
for k=1:n
    [pivot, m]=max(abs(U(k:n,k)));
    if pivot==0
        error('La matrice inserita e'' singolare');
    end
    m=m+k-1;
    if m~=k
        U([k,m], :) = U([m, k], :);
        P([k,m], :) = P([m, k], :);
        if k >= 2
            L([k,m],1:k-1) = L([m,k],1:k-1);
        end
    end
    L(k+1:n,k)=U(k+1:n,k)/U(k,k);
    U(k+1:n,:)=U(k+1:n,:)-L(k+1:n,k)*U(k,:);
end
```

è possibile vedere il funzionamento di questa function nell'esercizio 14 a pagina 23.

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:

```
function [x] = linLUP(L,U, P, b)
    x = triInf(L,P*b);
    x = triSup(U,x);
end
```

`Calcolo matrice trinagolare inferiore`

```
function [x] = triInf(A, b)
    x = b;
    for j=1:length(A)
        if A(j,j) ~= 0
            x(j) = x(j)/A(j,j);
        else
            error('La matrice e'' singolare')
        end
        for i=j+1:length(A)
            x(i) = x(i)-A(i,j)*x(j);
        end
    end
end
```

`Calcolo matrice trinagolare superiore`

```
function [b] = triSup(A, b)
    for j=size(A):-1:1
        if A(j,j)==0
            error('[Attenzione] La matrice non e'' singolare')
```

```
        else
            b(j)=b(j)/A(j,j);
        end
        for i=1:j-1
            b(i)=b(i)-A(i,j)*b(j);
        end
    end
end
```

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 $x^T Ax > 0$

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

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

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 $i \in [1, ..., n]$ 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:

$$\sum_{i=1}^{n} 2(n-i)(i-1) = 2 \cdot \sum_{i=1}^{n} (i \cdot n - n - i^2 + i) = 2 \cdot \left[(n+1) \sum_{i=1}^{n} i - n^2 - \sum_{i=1}^{n} i^2 \right] =$$

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

$$= 2 \cdot \left[n^3 \cdot \left(\frac{1}{2} - \frac{1}{3} \right) + n^2 \cdot \left(-1 + \frac{1}{6} + \frac{1}{2} \right) + n \cdot \left(1 - \frac{1}{2} - \frac{1}{6} \right) \right] = \frac{2}{6} n^3 + \frac{2}{3} n^2 + \frac{2}{3} n \approx \frac{1}{3} n^3$$

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:

```
function [L,D] = factLDLT(A)
    [m,n]=size(A);
    if m~=n
        error('La matrice non e'' quadrata')
    end
    if A(1,1)<=0
        error('La matrice non e'' SDP')
    end
    A(2:n,1)=A(2:n,1)/A(1,1);
    for j=2:n
        v = (A(j,1:(j-1))').*diag(A(1:(j-1),1:(j-1)));
        A(j,j) = A(j,j)-A(j,1:(j-1))*v;
        if A(j,j)<=0
            error('La matrice non e'' SDP');
        end
        A((j+1):n,j)=(A((j+1):n,j)-A((j+1):n,1:(j-1))*v)/A(j,j);
    end
    D=diag(diag(A));
    L=tril(A,-1)+eye(size(A));
end
```

è possibile vedere il funzionamento di questa function nell'esercizio 23.

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:

```
function [x] = linLDLT(L,D, b)
    x = linLUP(L,D*L',eye(size(D)),b);
end
```

3.13 Esercizio 13

Per verificarlo abbiamo usato il seguente codice MatLab:

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

Nel primo caso abbiamo usato la matrice $A \in M^{3\times 3}$ con elementi:

$$A = \begin{pmatrix} 0 & -3 & 8 \\ -1 & 8 & 7 \\ 1 & 3 & 0 \end{pmatrix}$$

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

$$b = \begin{pmatrix} 3.1416 \\ 1.1618 \\ 2.7183 \end{pmatrix}$$

Nel secondo caso abbiamo usato la matrice $A \in M^{3\times 3}$ con elementi:

$$A = \begin{pmatrix} 14 & 5 & 2 \\ 5 & 8 & 1 \\ 2 & 1 & 4 \end{pmatrix}$$

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

$$b = \begin{pmatrix} 3.1416 \\ 1.1618 \\ 2.7183 \end{pmatrix}$$

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

```
format shortE

% 3.5 3.6
disp('Vettore residuo con fattorizzazione LU con pivoting parziale:');
A=[0,-3,8;-1,8,7;1,3,0];
[L,U,P] = factLUP(A);
b = [3.1416,1.1618,2.7183]';
[x] = linLUP(L,U,P,b);
r=A*x -b

% 3.11  3.12
disp('Vettore residuo con fattorizzazione LDLT:');
A=[14,5,2;5,8,1;2,1,4];
[L,D] = factLDLT(A);
b = [3.1416,1.1618,2.7183]';
[x] = linLDLT(L,D,b);
r=A*x -b
```

Il codice sopra restituisce l'output:

```
>> es14
Soluzione e vettore residuo con fattorizzazione LU con pivoting parziale:
x =
   2.4692e+00
   8.3024e-02
   4.2383e-01
r =
   8.8818e-16
            0
            0
Soluzione e vettore residuo con fattorizzazione LDLT:
x =
   1.4465e-01
  -2.1765e-02
   6.1269e-01
r =
            0
  -4.4409e-16
            0
```

3.15 Esercizio 15

Per confrontare i risultati tra il nostro procedimento di calcolo di $k_\infty(A)$ e quello della function cond di MatLab, abbiamo scritto il seguente script:

```
format shortE
v = [1,1,1,1,1,1,1,1,1,1];
A = (diag(v*(-100),-1)+eye(10))
disp('Il numero di condizionamento k senza la function cond e'':');
disp('con norma infinito');
normAinf = norm(A,inf);
normAIinf = norm(inv(A),inf);
kinf = normAinf*normAIinf
disp('con norma 1');
```

```
normA1 = norm(A,1);
normAI1 = norm(inv(A),1);
k1 = normA1*normAI1
disp('Il numero di condizionamento k con la funzione cond e'':');
disp('con norma infinito');
cond(A,inf)
disp('con norma 1');
cond(A,1)
```

Che restituisce come output:

```
>> es15
A =
     1     0     0     0     0     0     0     0     0     0
  -100     1     0     0     0     0     0     0     0     0
     0  -100     1     0     0     0     0     0     0     0
     0     0  -100     1     0     0     0     0     0     0
     0     0     0  -100     1     0     0     0     0     0
     0     0     0     0  -100     1     0     0     0     0
     0     0     0     0     0  -100     1     0     0     0
     0     0     0     0     0     0  -100     1     0     0
     0     0     0     0     0     0     0  -100     1     0
     0     0     0     0     0     0     0     0  -100     1
Il numero di condizionamento k senza la function cond e':
con norma infinito
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  9.801980e-21.
> In es15 (line 9)
kinf =
   1.0202e+20
con norma 1
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  9.801980e-21.
> In es15 (line 13)
k1 =
   1.0202e+20
Il numero di condizionamento k con la funzione cond e':
con norma infinito
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  9.801980e-21.
> In cond (line 46)
  In es15 (line 18)
ans =
   1.0202e+20
con norma 1
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  9.801980e-21.
> In cond (line 46)
  In es15 (line 20)
ans =
   1.0202e+20
```

Dall'output possiamo vedere che k_∞ che per k_1 risultano uguali con valore pari a $1.0202 \cdot 10^{20}$. Inoltre sia cond che inv restituiscono come warning:

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  9.801980e-21.
```

che significa che il problema dell'inversione della matrice può non essere accurato perchè la matrice è mal condizionata. Dobbiamo ora dimostrare che $k_\infty(A) = k_1(A)$, cioè:

$$\left|\left|A\right|\right|_\infty \cdot \left|\left|A^{-1}\right|\right|_\infty = \left|\left|A\right|\right|_1 \cdot \left|\left|A^{-1}\right|\right|_1$$

Possiamo calcolare $\left|\left|A\right|\right|_\infty$ che ovviamente sarà pari a 101 dato che gli unici 2 valori ottenuti dalla somma dei valori assoluti degli elementi riga della matrice sono 1 e 101, da cui possiamo affermare che il massimo tra i due è 101. Per $\left|\left|A\right|\right|_1$ i valori delle somme dei valori assoluti degli elementi colonna della matrice sono 1 e 101, come per la precedente norma otteniamo quindi il valore massimo 101. Sappiamo quindi che $\left|\left|A\right|\right|_\infty = \left|\left|A\right|\right|_1$. Rimane da dimostrare che $\left|\left|A^{-1}\right|\right|_\infty = \left|\left|A^{-1}\right|\right|_1$. Per farlo possiamo andare a calcolare la matrice inversa $A^{-1} = \frac{1}{\det(A)} A^*$, in cui A^* è la matrice aggiunta calcolata come segue:

$$a^*_{1,1} = \det|A_{1,1}| = 1 = a^*_{2,2} = a^*_{3,3} = ... = a^*_{10,10}$$
$$a^*_{2,1} = \det|A_{1,2}| = 10^2 = a^*_{3,2} = a^*_{4,3} = ... = a^*_{10,9}$$
$$a^*_{3,1} = \det|A_{1,3}| = 10^4 = a^*_{4,2} = a^*_{5,3} = ... = a^*_{10,8}$$
$$.$$
$$.$$
$$.$$
$$a^*_{10,1} = \det|A_{1,10}| = 10^{18}$$
$$a^*_{1,2} = \det|A_{2,1}| = 0 = a^*_{1,3} = ... = a^*_{1,10}$$

(Abbiamo usato la notazione $A_{n,m}$ per la matrice che si ottiene a partire da A eliminando la riga n e la colonna m). Per i valori al di sopra della diagonale possiamo dire che sono in valore assoluto $<< 1$. Dato che $\det(A) = 1$ allora $A^{-1} = A^*$:

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 10^2 & 1 & . & . & . & . & . & . & . & . \\ 10^4 & 10^2 & 1 & . & . & . & . & . & . & . \\ 10^6 & 10^4 & 10^2 & 1 & . & . & . & . & . & . \\ 10^8 & 10^6 & 10^4 & 10^2 & 1 & . & . & . & . & . \\ 10^{10} & 10^8 & 10^6 & 10^4 & 10^2 & 1 & . & . & . & . \\ 10^{12} & 10^{10} & 10^8 & 10^6 & 10^4 & 10^2 & 1 & . & . & . \\ 10^{14} & 10^{12} & 10^{10} & 10^8 & 10^6 & 10^4 & 10^2 & 1 & . & . \\ 10^{16} & 10^{14} & 10^{12} & 10^{10} & 10^8 & 10^6 & 10^4 & 10^2 & 1 & . \\ 10^{18} & 10^{16} & 10^{14} & 10^{12} & 10^{10} & 10^8 & 10^6 & 10^4 & 10^2 & 1 \end{bmatrix}$$

Notiamo che la somma dei valori della riga 10 e della colonna 1 della matrice inversa sono identici, inoltre sono anche i valori ottenuti effettuando $\left|\left|A^{-1}\right|\right|_\infty$ e $\left|\left|A^{-1}\right|\right|_1$. Possiamo quindi affermare che $\left|\left|A^{-1}\right|\right|_\infty = \left|\left|A^{-1}\right|\right|_1$.

3.16 Esercizio 16

Abbiamo implementato il codice seguente per poter rispondere alle domande dell'esercizio:

```
v = [1,1,1,1,1,1,1,1,1,1];
A = (diag(v*(-100),-1)+eye(10));
b = [1 ,-99*ones(1,9)]';
c = 0.1*[1,-99*ones(1,9)]';

x = ones(10,1);
y = 0.1*x;

format shortE

rx = A*x -b
ry = A*y -c

x(1)=b(1);
for i=2:10
    x(i)=b(i)+100*x(i-1);
end
x=x(:)

y(1)=c(1);
for i=2:10
    y(i)=c(i)+100*y(i-1);
```

```
23 end
24 y=y(:)
25
26 rx = A*x -b
27 ry = A*y -c
```

Possiamo confermare che le soluzioni x e y dei sistemi lineari $A \cdot x = b$ e $A \cdot y = c$ sono giuste dato che calcolando i loro residui otteniamo:

```
>> es16
rx =
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
ry =
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

Nel passo successivo si usa la serie di istruzioni forniteci dall'esercizio. Nel caso del vettore x si perviene alla stessa soluzione precedentemente fornita dall'esercizio. Invece nel caso del vettore y abbiamo una propagazione degli errori nella soluzione trovata, che si può vedere a partire dall'elemento y_7. Possiamo vederlo dall'output:

```
x =
     1
     1
     1
     1
     1
     1
     1
     1
     1
     1
y =
   1.0000e-01
   1.0000e-01
   1.0000e-01
   1.0000e-01
   1.0000e-01
   1.0000e-01
   9.9996e-02
   9.9641e-02
   6.4114e-02
  -3.4886e+00
```

Si vede che c'è una perturbazione sulla soluzione che è possibile spiegare andando a studiare la seguente disuguaglianza:

$$\frac{||\Delta x||}{||x||} \leq k(A) \cdot \left(\frac{||\Delta c||}{||c||} + \frac{||\Delta A||}{||A||} \right) = k(A) \cdot \left(\frac{||\Delta c||}{||0.1 \cdot b||} + \frac{||\Delta A||}{||A||} \right) =$$

$$= k(A) \cdot \left(\frac{||\Delta c||}{10^{-1} \cdot ||b||} + \frac{||\Delta A||}{||A||} \right) = k(A) \cdot \left(10 \cdot \frac{||\Delta c||}{||b||} + \frac{||\Delta A||}{||A||} \right)$$

Considerato che:

- $\frac{||\Delta x||}{||x||}$ può essere assimilato ad una sorta di errore relativo sul risultato
- $\frac{||\Delta A||}{||A||}$ e $\frac{||\Delta c||}{||c||}$ possono essere assimilati ai corrispondenti errori relativi sui dati in ingresso

dato che il numero di condizionamento del problema è $k(A) = 1.0202 \cdot 10^{20}$, che è $>> 1$ (calcolato nell'esercizio precedente a pag. 25), allora la matrice è mal condizionata.

3.17 Esercizio 17

L'algoritmo di fattorizzazione QR, mediante metodo di householder, da noi implementato è il seguente:

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

è possibile vedere il funzionamento di questa function nell'esercizio 19 a pagina 28.

3.18 Esercizio 18

Supponendo che in ingresso si abbiano:

- la matrice $A \in M^{n \times m}$ (con $n > m \in \mathbb{N}$) già fattorizzata QR
- il vettore dei termini noti $b \in \mathbb{R}^n$

è possibile scrivere la function solveQRH che risolve sistemi lineari sovradeterminati:

```
function [x] = solveQRH( A, b )
    [m,n] = size(A);
    Qt = factQRH(A);
    x = TriSup(triu(A(1:n, :)), Qt(1:n, :)*b);
end
```

è possibile vedere il funzionamento di questa function nell'esercizio 19 a pagina 28.

3.19 Esercizio 19

Il codice usato è:

```
format shortE

A = [3,2,1;1,2,3;1,2,1;2,1,2]
b = [6;6;4;4]
```

```
5
6  x = solveQRH(A,b)
7
8  r = A*x-b
9
10 disp('Norma di r : ')
11 norm(r,2)^2
```

che ha generato questo risultato:

```
>> es19
A =
     3     2     1
     1     2     3
     1     2     1
     2     1     2
b =
     6
     6
     4
     4
x =
   3.6762e+00
  -1.0057e+01
   8.2286e+00
r =
  -6.8571e+00
   2.2476e+00
  -1.2210e+01
   9.7524e+00
Norma di r :
ans =
   2.9625e+02
```

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:

```
1  format shortE
2
3  x(1)=1;
4  x(2)=1;
5  imax=1000;
6  tolx=0.0001;
7
8  F = @(x) [x(2) - cos(x(1)); x(1)*x(2)-1/2];
```

```
J = @(x) [sin(x(1)),1 ; x(2), x(1)];

[x] = nonLinearNewton(F, J, x, imax, tolx , 1);
```

```
function [x] = nonLinearNewton(F,J, x, imax, tolx, out)
    i=0;
    xold = x+1;
    while (i< imax )&&( norm (x-xold )> tolx )
        i=i+1;
        xold =x;
        [L,U,P] = factLUP(feval(J,x));
        x=x+linLUP(L,U, P, -feval(F,x));
        if out
            disp ('iterata:');
            disp(i);
            disp('norma dell''incremento:');
            disp(norm(x-xold));
            disp('valori di x:');
            disp(x);
        end
    end
end
```

si ottengono i risultati:

```
>> es20
iterata:
     1
norma dell'incremento:
   5.0007e-01
valori di x:
ans =
   7.4577e-01
   7.5423e-01
iterata:
     2
norma dell'incremento:
   3.1450e-01
valori di x:
ans =
   5.5311e-01
   8.6530e-01
iterata:
     3
norma dell'incremento:
   9.2853e-02
valori di x:
ans =
   6.0420e-01
   8.2406e-01
iterata:
     4
norma dell'incremento:
   1.0230e-02
valori di x:
ans =
   6.0996e-01
   8.1968e-01
iterata:
     5
norma dell'incremento:
   1.2686e-04
valori di x:
ans =
   6.1003e-01
   8.1963e-01
```

Che in forma tabellare sono rappresentati da:

i	x_1	x_2	Norma incremento
1	0.7458	0.7542	0.50007
2	0.5531	0.8653	0.31450
3	0.6042	0.8241	0.09285
4	0.6100	0.8197	0.01023
5	0.6100	0.8196	0.00013

3.21 Esercizio 21

Dobbiamo studiare il minimo della funzione a più variabili $f(x_1, x_2) = x_1^4 + x_1 \cdot (x_1 + x_2) + (1 + x_2)^2$, per fare questo è necessario trovare i punti stazionari del gradiente della funzione f:

$$\nabla f \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \underline{0}$$

Con ∇f calcolato come segue:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 4 \cdot x_1^3 + 2 \cdot x_1 + x_2 \\ x_1 + 2 \cdot (1 + x_2) \end{pmatrix}$$

A questo punto è possibile usufruire della function MatLab nonLinearNewton implementata precedentemente per trovare la soluzione di questo sistema lineare. Per poterla usare è però necessario trovare la matrice jacobiana di ∇f (che non è altro che la matrice hessiana di f):

$$J\nabla f = \begin{pmatrix} \frac{\partial(\nabla f_1)}{\partial x_1} & \frac{\partial(\nabla f_1)}{\partial x_2} \\ \frac{\partial(\nabla f_2)}{\partial x_1} & \frac{\partial(\nabla f_2)}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 12x_1^2 + 2 & 1 \\ 1 & 2 \end{pmatrix}$$

Abbiamo quindi scritto lo script:

```
format shortE

x(1)=0;
x(2)=0;
imax=100;
tolx=0.0001;

F = @(x) [4*x(1)^3 + 2*x(1) + x(2); x(1)+2*(1+x(2))];
J = @(x) [12*x(1)^2+2,1 ; 1, 2];

[x] = nonLinearNewton(F, J, x, imax, tolx , 0);

disp ('Il minimo ottenuto e''');
disp (x);
disp ('F(x): ');
disp (x(1)^4+x(1)*(x(1)+x(2))+(1+x(2))^2);
```

che restituisce in output il risultato:

```
>> es21
Il minimo ottenuto e'
   4.3981e-01   4.3981e-01
  -1.2199e+00  -1.2199e+00
F(x):
  -2.5732e-01
```

$\underline{x^*} = \begin{pmatrix} 4.3981e-01 & -1.2199e+00 \end{pmatrix}$ rappresenta quindi il punto di minimo della funzione $f(x_1, x_2) = x_1^4 + x_1 \cdot (x_1 + x_2) + (1 + x_2)^2$. Andando a sostituire i valori si ottiene:

$$f(\underline{x^*}) = (0,43981)^4 + (0,43981) \cdot (0,43981 - 1,2199) + (1 - 1,2199)^2 = -0.25732$$

4 Capitolo 4

4.1 Esercizio 1

Il seguente script MatLab utilizza 2 function: diffDiv e hornerGen, di cui è possibile trovarne l'implementazione a pagina **??**:

```
function [pval] = newtonHor(xi, fi, xval)
  dd = diffDiv(xi, fi);
  pval = hornerGen(xi,dd,xval);
end
```

```
function [fi] = diffDiv(xi, fi, n)
  if nargin == 2, n=length(xi); end
  for i=1:n-1
    for j=n:-1:i+1
      fi(j) = (fi(j) - fi(j-1))/(xi(j)-xi(j-i));
    end
  end
end
```

```
function [p] = hornerGen(xi, dd, xval)
  n=length(dd);
  for i=1:length(xval)
    p(i)=dd(n);
    for k=n-1:-1:1
      p(i)=p(i)*(xval(i)-xi(k))+dd(k);
    end
  end
end
```

è possibile vedere il funzionamento di questa function nell'esercizio successivo (pag. 32).

4.2 Esercizio 2

Il codice MatLab usato è il seguente:

```
format longE

f1 = @(x) 1./(1.+x.^2);
a=-5;
b=5;
ferrors1 = zeros(2,10);
[errorsA1, plotsA1, l1] = evaluatePoli(f1,a,b,20,10,0,100);
[errorsC1, plotsC1] = evaluatePoli(f1,a,b,20,10,1,100);
disp('errori con ascisse equidistanti');
ferrors1(1,:) = max(abs(errorsA1'))';
disp(ferrors1(1,:));
disp('errori con ascisse di chebyshev');
ferrors1(2,:) = max(abs(errorsC1'))';
disp(ferrors1(2,:));

n_p = 5;

plotting1 = cat(2,plotsA1(n_p,:)',plotsC1(n_p,:)');
plotting1 = cat(2,plotting1,f1(l1)');

figure(1);
plot(l1,plotting1');

```

```
n_p = 3;

plotting1 = cat(2,plotsA1(n_p,:)',plotsC1(n_p,:)');
plotting1 = cat(2,plotting1,f1(l1)');

figure(4);
plot(l1,plotting1');

f2 = @(x) x.*sin(x);
a=0;
b=pi;
ferrors2 = zeros(2,10);
[errorsA2, plotsA2, l2] = evaluatePoli(f2,a,b,20,10,0,100);
[errorsC2, plotsC2] = evaluatePoli(f2,a,b,20,10,1,100);
disp('errori con ascisse equidistanti');
ferrors2(1,:) = max(abs(errorsA2'))';
disp(ferrors2(1,:));
disp('errori con ascisse di chebyshev');
ferrors2(2,:) = max(abs(errorsC2'))';
disp(ferrors2(2,:));

n_p = 1;

plotting2 = cat(2,plotsA2(n_p,:)',plotsC2(n_p,:)');
plotting2 = cat(2,plotting2,f2(l2)');
figure(2);
plot(l2,plotting2');

n_p = 2;

plotting2 = cat(2,plotsA2(n_p,:)',plotsC2(n_p,:)');
plotting2 = cat(2,plotting2,f2(l2)');
figure(3);
plot(l2,plotting2');
```

```
% funct -> funzione da interpolare
% a,b -> intervallo di interpolazione
% maxn -> limite massimo del grado di interpolazione della funzione
% n_steps -> numero di polinomi interpolanti da calcolare
% cheb_asc -> = 1 ascisse calcolate con chebyshev | = 0 ascisse calcolate
% con ascisse equispaziate
% plot_steps -> numero di nodi
function [errors, plots, l] = evaluatePoli(funct, a, b, maxn, n_steps, cheb_asc,
    plot_steps)
    errors = zeros(n_steps,plot_steps);
    plots = zeros(n_steps,plot_steps);
    l = linspace(a,b,plot_steps);
    steps = linspace(2, maxn, n_steps);
    for i=1:n_steps
        if cheb_asc == 0
            ascisse = eqAscisse(a, b, steps(i));
        elseif cheb_asc == 1
            ascisse = cheby(a, b, steps(i));
        end

        fInt = newtonHor(ascisse, funct(ascisse), l);
        errors(i,:) = funct(l)-fInt;
        plots(i,:) = fInt;
```

```
    end
end
```

```
function [ptx] = eqAscisse(a, b, n)
  h = (b-a)/n;
  ptx = zeros(n+1, 1);
  for i=1:n+1
    ptx(i) = a +(i-1)*h;
  end
end
```

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

Con $f(x) = \frac{1}{1+x^2}$ gli errori sono:

n	$AscisseEquispaziate$	$Chebyshev$
2	$6.459699748665507 \cdot 10^{-1}$	$6.005718959661196 \cdot 10^{-1}$
4	$4.382728746134097 \cdot 10^{-1}$	$4.019561301268590 \cdot 10^{-1}$
6	$6.164015686420363 \cdot 10^{-1}$	$2.641051307764343 \cdot 10^{-1}$
8	1.045078216378144	$1.700656314789976 \cdot 10^{-1}$
10	1.915434269679889	$1.090256419757498 \cdot 10^{-1}$
12	3.611701597804764	$6.902642915187807 \cdot 10^{-2}$
14	7.189298472071057	$4.608936896629207 \cdot 10^{-2}$
16	$1.401353448336841 \cdot 10$	$3.258023221036738 \cdot 10^{-2}$
18	$2.750677081179504 \cdot 10$	$2.212383276248159 \cdot 10^{-2}$
20	$5.840669034602058 \cdot 10$	$1.500717452472311 \cdot 10^{-2}$

Il grafico ottenuto per questa funzione con n=6 è:

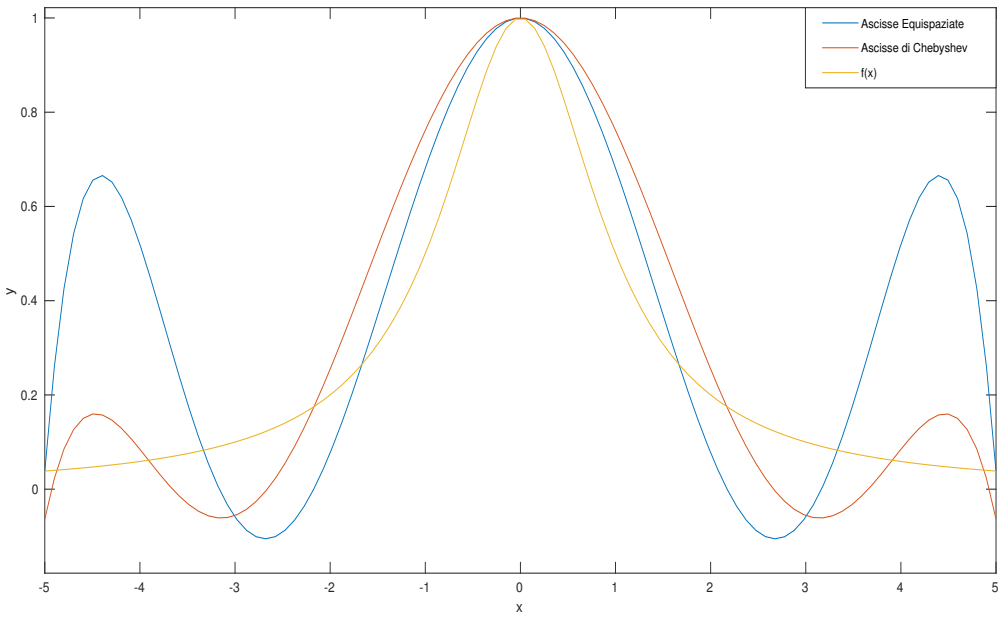


Figure 7: `Confronto tra il grafico reale della funzione e le rispettive interpolazioni`

Il grafico ottenuto per questa funzione con n=12 è:

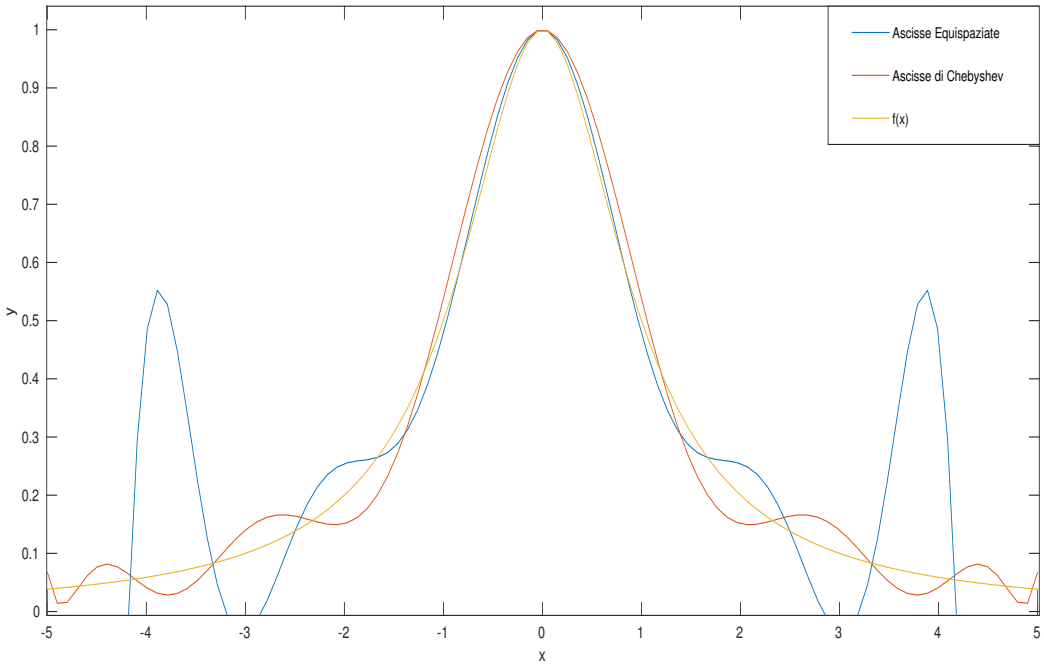


Figure 8: `Confronto tra il grafico reale della funzione e le rispettive interpolazioni`

Nel caso in cui $f(x) = x \cdot sin(x)$ gli errori sono:

i	$AscisseEquispaziate$	$Chebyshev$
2	$6.381422078133654 \cdot 10^{-1}$	$4.370888339682153 \cdot 10^{-1}$
4	$4.127250365828502 \cdot 10^{-2}$	$2.286331189343005 \cdot 10^{-2}$
6	$1.343141368041964 \cdot 10^{-3}$	$4.779185850743994 \cdot 10^{-4}$
8	$2.575400820072765 \cdot 10^{-5}$	$5.331729223456705 \cdot 10^{-6}$
10	$3.237670194583542 \cdot 10^{-7}$	$3.688279803792938 \cdot 10^{-8}$
12	$2.843434354291019 \cdot 10^{-9}$	$1.733829746441984 \cdot 10^{-10}$
14	$1.873067152768915 \cdot 10^{-11}$	$5.891953591685706 \cdot 10^{-13}$
16	$1.261490911730334 \cdot 10^{-13}$	$3.417152747496221 \cdot 10^{-15}$
18	$8.933132011890166 \cdot 10^{-14}$	$1.776356839400250 \cdot 10^{-15}$
20	$1.768307722471718 \cdot 10^{-13}$	$2.327007791852781 \cdot 10^{-15}$

Il grafico per questa funzione con n=2 è il seguente:

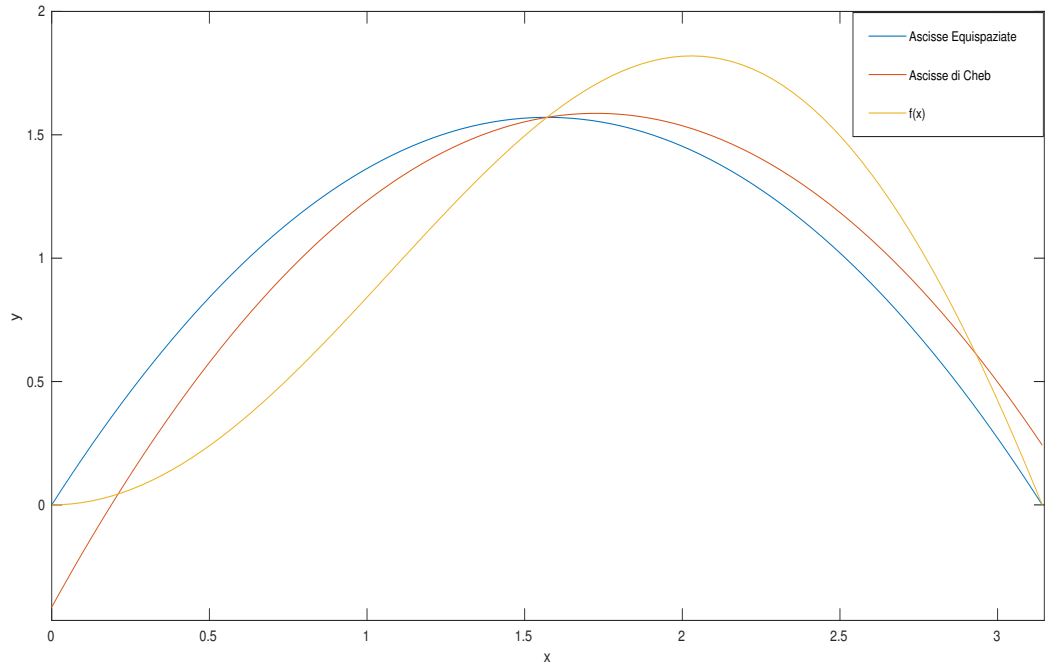


Figure 9: `Confronto tra il grafico reale della funzione e le rispettive interpolazioni`

Il grafico per questa funzione con n=4 è il seguente:

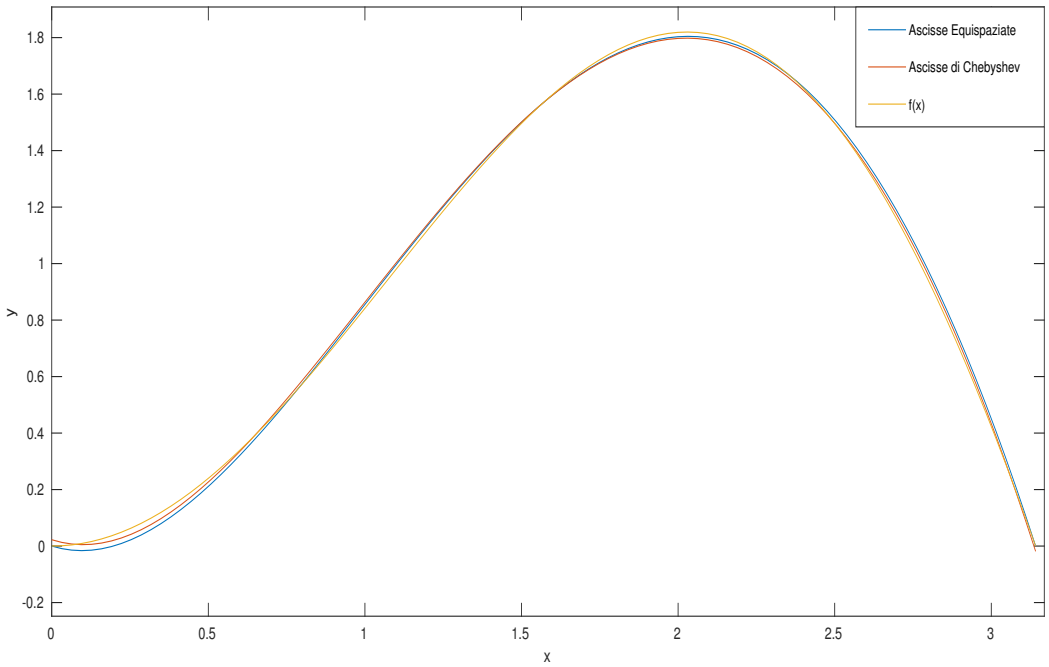


Figure 10: `Confronto tra il grafico reale della funzione e le rispettive interpolazioni`

Si può vedere dai 2 grafici che le due interpolazioni all'aumentare di n si avvicinano al reale grafico di $f(x) = x \cdot sin(x)$. Con $n = 20$ si arriva ad un errore relativo dell'ordine di 10^{-13} e 10^{-15}, cioè un'interpolazione ottima della funzione originaria.

4.3 Esercizio 3

```
function [ M ] = moment(f, x)
    n = length(x)-1;
    dd = 6*diffDiv(x, f, 3);
    h = diff(x);
    U = h(1:end-1)./( h(1:end-1)+h(2:end) );
    L = 5; L(1) = [];
    U(end) = [];
    D = 2*ones(1,n-1);

    for i = 2:n-1
        L(i-1) = L(i-1)/D(i-1);
        D(i) = 2-L(i-1)*U(i-1);
    end

    for i=2:n-2, dd(i) = dd(i) - L(i) * dd(i-1); end
    dd(1:end) = dd(1:end)./D(1:end);
    for i=n-3:-1:1, dd(i) = dd(i) - L(i) * dd(i+1); end

    M = [0,dd,0];
end
```

4.4 Esercizio 4

```
function [ sval ] = svalCubica( f, x, M, xval)
    n = length(x);
    nval = length(xval);
    sval = zeros(1, nval);
    r = zeros(1, n+1); q = r; h = q; k=1;
    for i = 2:n
        h(i) = x(i) - x(i-1);
        r(i) = f(i-1) - (h(i)^2)*M(i-1)/6;
        q(i) = (f(i) - f(i-1))/h(i) - h(i)*( M(i) - M(i-1) )/6;
    end
    sp = @(xval, i) (((xval(k)-x(i-1))^3) * M(i) + ((x(i)-xval(k))^3) * M(i-1))/(6*h(i))
        + q(i)*(xval(k)-x(i-1)) + r(i);
    t=2; k=1;
    while k <= nval
        if x(t-1) <= xval(k) && xval(k) <= x(t), sval(k) = sp(xval(k), t); k = k + 1;
        else, t = t + 1;
        end
    end
end
```

4.5 Esercizio 5

Il seguente script organizza il risultato dell'esercizio per fare il plot dei confronti:

```
fx = @(x) 1./(1.+x.^2);
xval = linspace(-5, 5);

temp = [];
```

```
temp2 = [];

temp = cat(2,temp,fx(xval));

for i = 4: 4: 20
    x = linspace(-5, 5, i);
    f = fx(x);

    M = moment(f, x);

    s = svalCubica(f, x, M, xval);

    ss = spline(x, f);
    ssval = ppval(ss, xval);

    temp = cat(1,temp,s);
    temp2 = cat(1, temp2, ssval);
end

figure(1);
plot(xval,cat(1,temp(1:2:4,:),temp2(2:2:3,:)));
legend('f(x)','spline cubica','not-a-knot');

fx = @(x) x.*sin(x);
xval = linspace(0, pi);

temp = [];
temp2 = [];

temp = cat(2,temp,fx(xval));

for i = 4: 4: 20
    x = linspace(-5, 5, i);
    f = fx(x);

    M = moment(f, x);

    s = svalCubica(f, x, M, xval);

    ss = spline(x, f);
    ssval = ppval(ss, xval);

    temp = cat(1,temp,s);

    temp2 = cat(1, temp2, ssval);

end

figure(2);
plot(xval,cat(1,temp(1:2:4,:),temp2(2:2:3,:)));
legend('f(x)','spline cubica','not-a-knot');
```

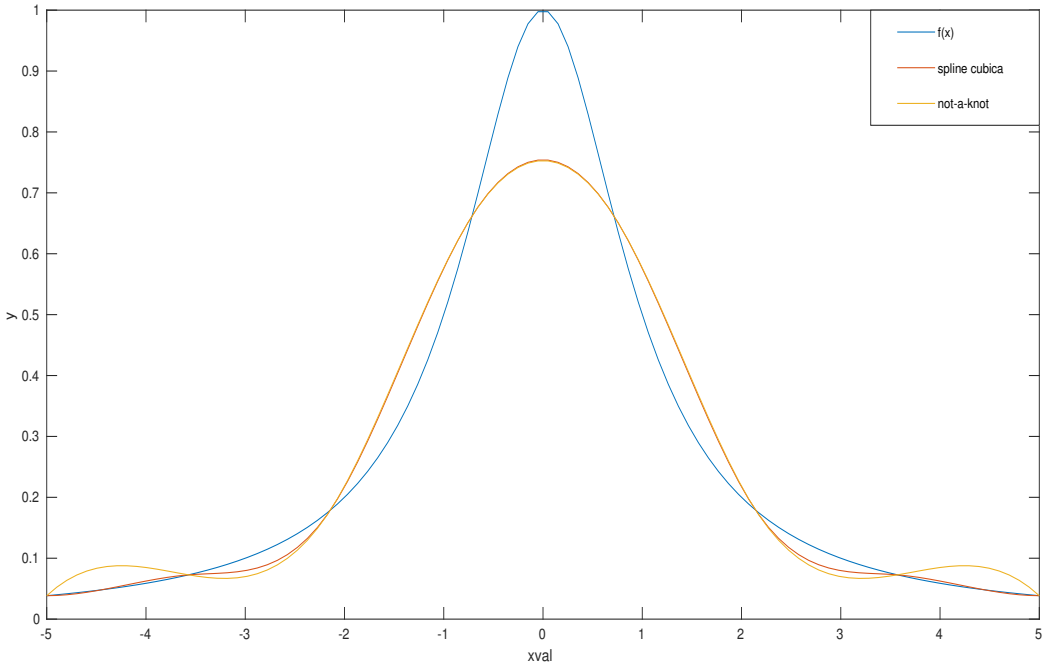


Figure 11: `Confronto tra spline e funzione reale`

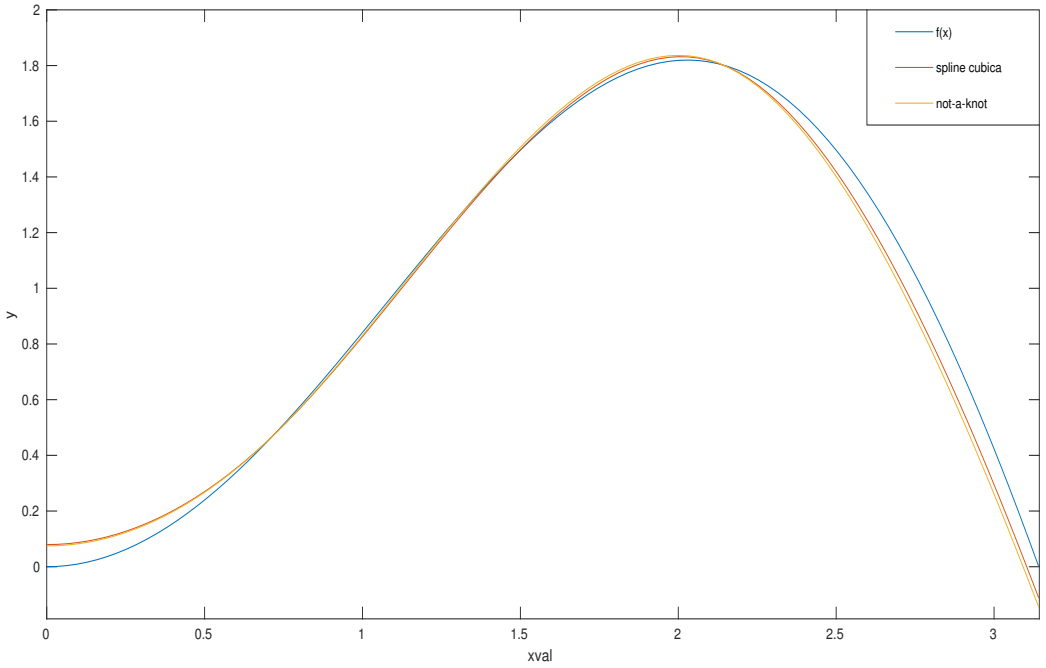


Figure 12: `Confronto tra spline e funzione reale`

4.6 Esercizio 6

4.7 Esercizio 7

4.8 Esercizio 8

Generalmente la legge che descrive un dato fenomeno è di tipo polinomiale con un determinato grado m:

$$y = \sum_{k=0}^{m} (a_k \cdot x^k)$$

Sapendo che (x_i, y_i) sono misure sperimentali è necessario estrapolare i vettore a_k che meglio approssima il polinomio. Per il calcolo di a_k abbiamo scritto la seguente function:

```
% m -> grado del polinomio
function [y] = polBetter(x,y, m)
  x=x';

  %controllo esistenza di m+1 ascisse distinte
  if length(unique (x)) < m+1
    error('[Errore] Non ci sono m ascisse distinte');
  end

  %creazione matrice vandermonde rettangolare
  V(:,m+1) = ones(length(x),1);
  for j = m:-1:1
    V(:,j) = x.*V(:,j+1);
  end

  %backslash
  y = V\y';
  y=y';
end
```

Abbiamo poi testato il suo funzionamento con il seguente script:

```
format shortE

x=[1,9,3,4,8,6,7];
disp(length(unique(x)));
y=[5,4,2,6,5,9,3];
polBetter(x,y, 4)

x=[3,0,3,4,0,6,3];
disp(length(unique(x)))
y=[5,4,2,6,5,9,3];
polBetter(x,y, 4)
```

Che restituisce in output:

```
>> es8
     7
ans =
   6.8251e-02  -1.4368e+00   1.0015e+01  -2.5426e+01   2.1728e+01
     4
Error using polBetter (line 7)
[Errore] Non ci sono m ascisse distinte
Error in es8 (line 11)
polBetter(x,y, 4)
```

Possiamo notare che il secondo insieme di dati sperimentali non ha un numero di ascisse distinte, infatti $4 > m + 1 = 5$ è falso.

4.9 Esercizio 9

Lo script che abbimo implementato è il seguente:

```
f1 = @(x,e,g) 5*x+ 2 +e*g;
f2 = @(x,e,g) 3*x^2 + 2*x +1 + e*g;
e= [0.1,0.2];
s= linspace(-1,1,10);
y1 = zeros(10,2);
y2 = zeros(10,2);

for j=1:2
for i=1:10
    y1(i,j) = f1(s(i),e(j),rand(1));
    y2(i,j) = f2(s(i),e(j),rand(1));
end
end

format shortE

disp('I coefficienti relativi al primo test con e=0.1');
res1(:,1) = polBetter(s,y1(:,1)',1);
disp(res1(:,1))
disp('I coefficienti relativi al primo test con e=0.2');
res1(:,2) = polBetter(s,y1(:,2)',1);
disp(res1(:,2))
disp('I coefficienti relativi al secondo test con e=0.1');
res2(:,1) = polBetter(s,y2(:,1)',2);
disp(res2(:,1))
disp('I coefficienti relativi al secondo test con e=0.2');
res2(:,2) = polBetter(s,y2(:,2)',2);
disp(res2(:,2))
```

Nello script abbiamo prima definito le funzioni da studiare e creato un vettore rappresentante la costante ϵ in cui il primo elemento è il caso $\epsilon = 0.1$ e il secondo è $\epsilon = 0.2$ Dunque nei due cicli annidati vengono calcolati gli elementi delle matrici $y_{i,1}$ e $y_{i,2}$ in cui nella prima colonna si hanno i valori della prima funzione con ϵ_1 e nella seconda con ϵ_2 (il valore di γ_i varia per ogni iterazione del ciclo in modo aleatorio). I quali vengono dati in input alla nostra funzione polBetter per effettuare i test richiesti, da cui si ricavano i coefficienti. Alla fine i risultati stampati sono:

```
>> es9
I coefficienti relativi al primo test con e=0.1
   5.0090e+00
   2.0441e+00
I coefficienti relativi al primo test con e=0.2
   5.0402e+00
   2.1043e+00
I coefficienti relativi al secondo test con e=0.1
   2.9703e+00
   2.0090e+00
   1.0562e+00
I coefficienti relativi al secondo test con e=0.2
   2.9691e+00
   2.0402e+00
   1.1170e+00
```

4.10 Esercizio 10

Per fare il confronto abbiamo scritto lo script:

```
1  f1 = @(x,e,l) 5*x+ 2 +e*l;
2  e= 0.1;
3  s= linspace(-1,1,10);
4  yi = zeros(10,1);
5  yd = zeros(10,1);
6  m=1;
7
8  % calcolo dei valori di y variabile indipendente
9  for i=1:10
10     yi(i) = f1(s(i),e,rand(1));
11 end
12
13 a = polBetter(s,yi',m);
14 % calcolo dei valori di y variabile dipendente
15 for i=1:10
16     for k=1:m+1
17         yd(i) = yd(i) + a(k)*s(i)^(k-1);
18     end
19 end
20
21 y = cat(2,yi,yd);
22 plot(s,y)
```

Il ciclo annidato a linea 15 calcola $y = \sum_{k=0}^{m}(a_k \cdot x^k)$ che trova i valori dipendenti dalle ascisse, il ciclo a linea 9 calcola i valori di y indipendenti. Il confronto tra le due rette è il seguente:

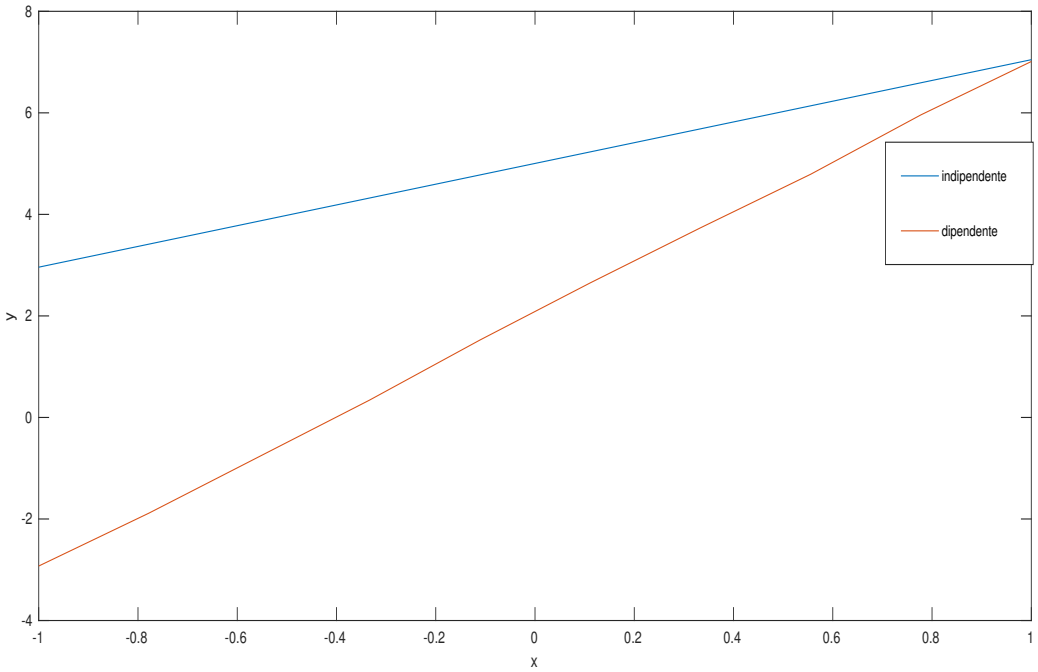


Figure 13: `Confronto tra y dipendente e indipendente`

5 Capitolo 5 e 6

5.1 Esercizio 5.1

Abbiamo implementato le seguenti function matlab:

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

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

Nell'esercizio successivo è possibile andare a vederne il funzionamento.

5.2 Esercizio 5.2

Il seguente codice calcola valori, errori e rapporti tra gli errori usando le 2 function precedenti:

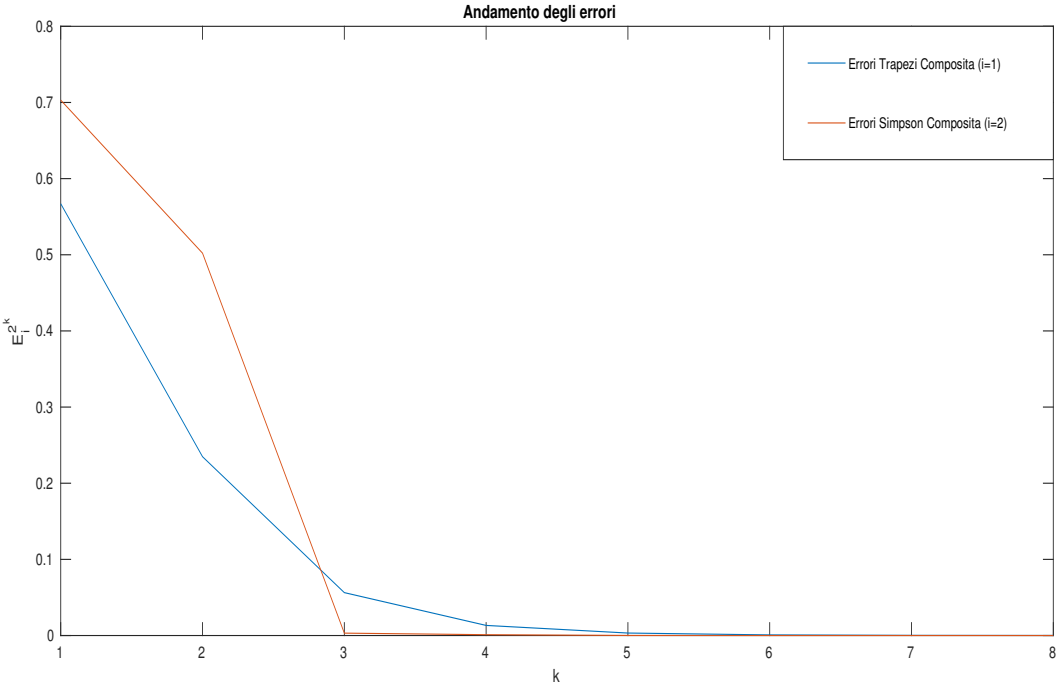
```
format longE

f = @(x) x*exp(-x)*cos(2*x);
I = (-10*pi*exp(-2*pi)+3*(exp(-2*pi)-1))/25;
kmax = 8;
error = zeros(kmax,2);
rap = zeros(kmax-1,2);

for k=1:kmax
    trap(k) = trapeziComposita(f,0,2*pi,2.^k);
    error(k,1) = abs(I-trap(k));
    simp(k) = simpsonComposita(f,0,2*pi,2.^k);
    error(k,2) = abs(I-simp(k));
    if k>=2
        rap(k-1,1) = error(k,1)/error(k-1,1);
        rap(k-1,2) = error(k,2)/error(k-1,2);
    end
end

figure(1);
plot((cat(1,I*ones(size(trap)),cat(1,trap,simp)))');
title('Valori ottenuti mediante le due formule');
legend('Valore vero','Trapezi Composita (i=1)','Simpson Composita (i=2)');
xlabel('k');
ylabel('I_i^{2^k}');

figure(2);
plot(error);
title('Andamento degli errori');
legend('Errori Trapezi Composita (i=1)','Errori Simpson Composita (i=2)');
```

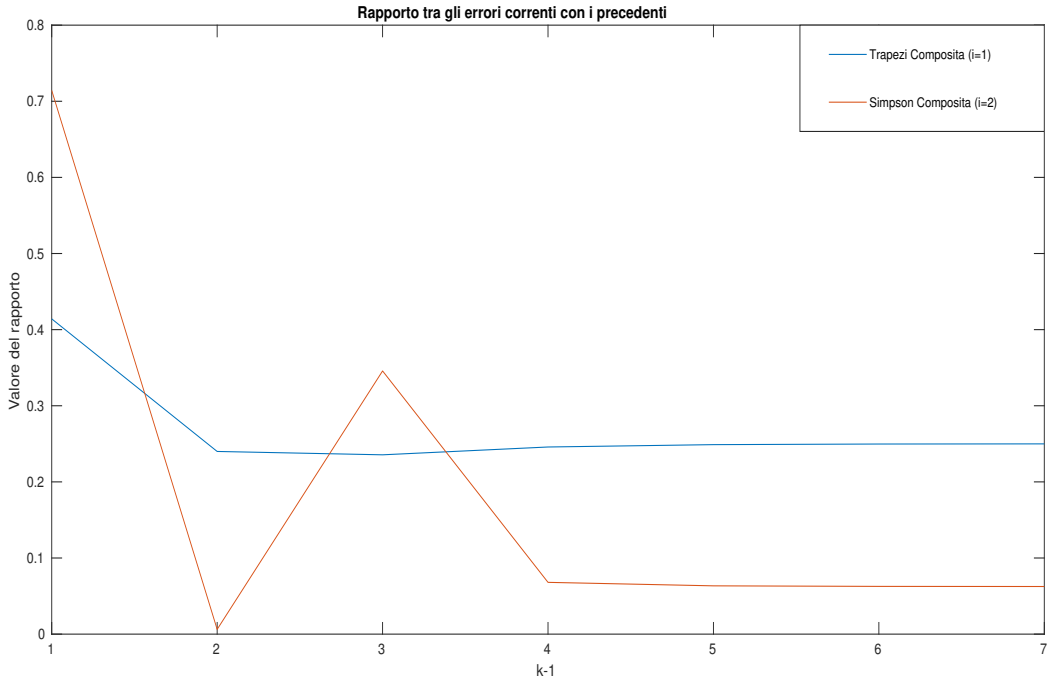
```
xlabel('k');
ylabel('E_i^{2^k}');

figure(3);
plot(rap);
title('Rapporto tra gli errori correnti con i precedenti');
legend('Trapezi Composita (i=1)','Simpson Composita (i=2)');
xlabel('k-1');
ylabel('Valore del rapporto');
```

Come richiesto riportiamo i valori in forma tabellare:

k	Errore Trapezi	Rapporto Trapezi	Errore Simpson	Rapporto Simpson
1	5.6706e-01	—	7.0308e-01	—
2	2.3483e-01	4.1412e-01	5.0212e-01	7.1417e-01
3	5.6353e-02	2.3997e-01	3.1390e-03	6.2514e-03
4	1.3274e-02	2.3556e-01	1.0853e-03	3.4574e-01
5	3.2632e-03	2.4583e-01	7.3810e-05	6.8010e-02
6	8.1229e-04	2.4892e-01	4.6819e-06	6.3431e-02
7	2.0285e-04	2.4973e-01	2.9360e-07	6.2710e-02
8	5.0699e-05	2.4993e-01	1.8365e-08	6.2551e-02

Lo script precedente genera i grafici sottostanti:





5.3 Esercizio 5.3

Abbiamo notato che i due metodi (in particolar modo la forma di Simpson) senza una condizione aggiuntiva rispetto a quella dell'errore è possibile che ciclino all'infinito. Inseriamo così un numero massimo di iterazioni della ricorsione negli argomenti della funzione:

```
function [I] = trapeziAdattiva(f, a, b, tol, iterMax)
  h=b-a;
  m=(a+b)/2;
  I1=0.5*h*(feval(f,a)+feval(f,b)); % metodo dei trapezi
  I=0.5*(I1+h*feval(f,m)); % metodo dei trapezi composito su 2 intervalli
  if (iterMax>0)&&(abs(I-I1)/3>tol), I=trapeziAdattiva(f,a,m,tol/2,iterMax-1)+
      trapeziAdattiva(f,m,b,tol/2,iterMax-1); end
end
```

```
function [I] = simpsonAdattiva(f, a, b, tol,iterMax)
  h=b-a;
  m=(a+b)/2;
  I2 = h*(f(a)+4*feval(f,m)+f(b))/6; % metodo di simpson
  I = (I2+h*feval(f,m))/6;  % metodo di simpson composito su 2 intervalli
  if (iterMax>0)&&(abs(I-I2)/15)>tol, I = simpsonAdattiva(f,a,m,tol/2, iterMax-1)+
      simpsonAdattiva(f,m,b,tol/2, iterMax-1); end
end
```

Con questi 2 metodi è possibile creare lo script di risoluzione seguente:

```
f = @(x) x*exp(-x)*cos(2*x);

[I] = trapeziAdattiva(f,0,2*pi,10^(-5),10);
disp('Formulazione adattiva con metodo dei trapezi:');
disp(I);

[I] = simpsonAdattiva(f,0,2*pi,10^(-3),10);
disp('Formulazione adattiva con metodo di Simpson');
disp(I);
```

Gli output che restituisce sono i seguenti:

```
>> es3
Formulazione adattiva con metodo dei trapezi:
   -122.123741436923e-003
Formulazione adattiva con metodo di Simpson
   -40.7063489337343e-003
```

5.4 Esercizio 5.4

```
function [x, k] = jacobi(A, b, x0, tol, nmax)
  k=0; n = size(A); n=n(1:1); temp = x0; x = x0;
  % implementazione del criterio d'arresto
  for i=1:n
      aux = 0;
      for j=1:n
          if j~=i, aux = aux + (A(i,j))*temp(j); end
      end
      x(i) = (b(i)-aux)/(A(i,i));
  end
  k = k+1;
  % ciclo principale
  while ((k<nmax)&&(norm(A*x-b)>tol*norm(b)))
      temp = x;
      for i=1:n
          aux = 0;
          for j=1:n
              if j~=i, aux = aux + (A(i,j))*temp(j); end
          end
          x(i) = (b(i)-aux)/A(i,i);
      end
      k=k+1;
  end
  if k>=nmax, disp('Jacobi non converge nel numero di iter fissato'); end
end
```

```
function [x, k] = gaussSeidel(A, b, x0, tol, nmax)
    k=0; n = size(A); n=n(1:1); x=zeros(n,1);
    % implementazione del criterio d'arresto
    for i=1:n
        aux = 0;
        for j=1:i-1, aux = aux + (A(i,j)*x(j)); end
        temp = 0;
        for j=i+1:n, temp = temp + (A(i,j)*x0(j)); end
        x(i) = (b(i)-aux-temp)/(A(i,i));
    end
    k = k+1;
    % ciclo principale
    while ((k<nmax)&&(norm(A*x-b)>tol*norm(b)))
       x0=x; x=zeros(n,1);
       for i=1:n
           aux = 0;
           for j=1:i-1, aux = aux + (A(i,j))*x(j); end
           temp = 0;
           for j=i+1:n, temp = temp + (A(i,j)*x0(j)); end
```

```
            x(i) = (b(i)-aux-temp)/A(i,i);
        end
        k=k+1;
    end
    if k>nmax, error('Gauss-Seidel non converge nel numero di iterazioni fissato'); end
end
```

5.5 Esercizio 5.5

Per testare le precedenti funzioni abbiamo implementato il seguente script:

```
A = [-4,2,1;1,6,2;1,-2,5];
b = [1,2,3]';
x0=[0;0;0];

[resJ,iterJ] = jacobi(A,b,x0,10^(-3),25);
disp('jacobi:');
disp(resJ);
disp('con iterazioni');
disp(iterJ);

[resG, iterG] = gaussSeidel(A,b,x0,10^(-3),25);
disp('gaussSeidel:');
disp(resG);
disp('con iterazioni');
disp(iterG);
```

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:

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

Come possiamo vedere dall'output:

```
>> es5
jacobi:
    -2.650738683127571e-02
     1.196586634087791e-01
     6.531825442386832e-01
con iterazioni
    11
gaussSeidel:
    -2.626826473042052e-02
     1.197856944260921e-01
     6.531679307165209e-01
con iterazioni
     8
```

Le nostre function restituiscono in output il vettore risultato e il numero di iterazioni che ha svolto.

5.6 Esercizio 5.6

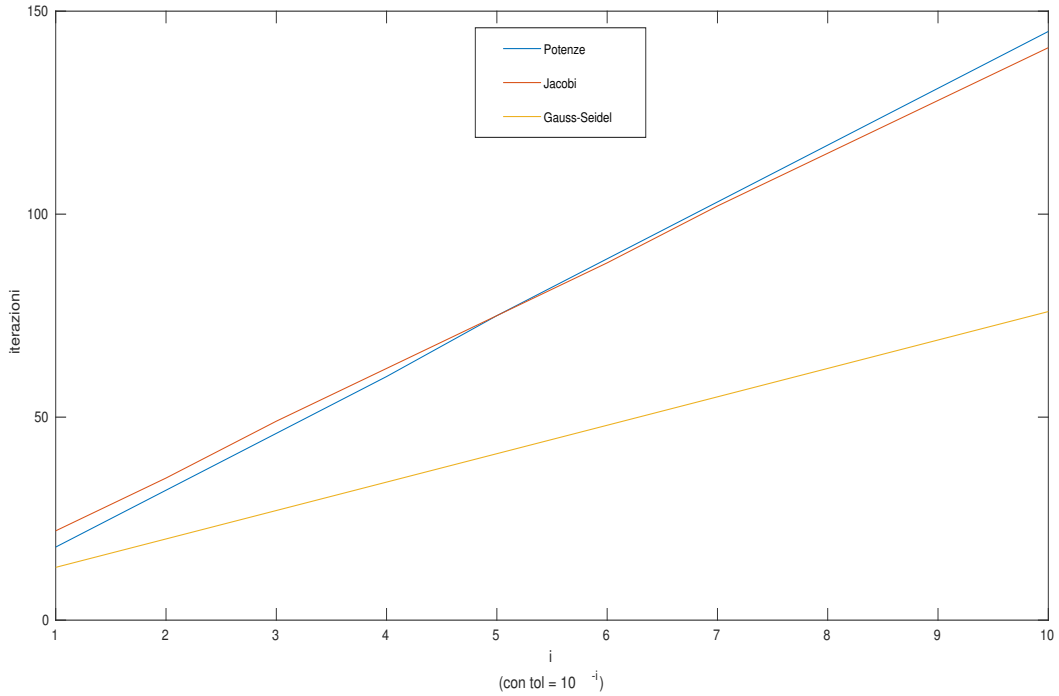
Per confrontare i vari metodi abbiamo usato il seguente:

```
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];
p=0.85;

[n,m] = size(H);
if(n~=m), error('Matrice non quadrata'); end
s = sum(H);
S=zeros(n,n);
for i=1:n
    if s(i)~=0, S(:,i)=H(:,i)/s(i);
    else, S(:,i)=(1/n); end
end
A = eye(n) - p*S;
b = ((1-p)/n).*ones(n,1);
tols = logspace(-1,-10,10);
iters = zeros(10,3);

for i=1:10
    v=zeros(n,4);
    [v(:,1),iters(i,1)]=PotenzePR(S,p,tols(i));
    [v(:,2),iters(i,2)]=jacobi(A,b,ones(n,1), tols(i), 10000);
    [v(:,3),iters(i,3)]=gaussSeidel(A,b,ones(n,1), tols(i), 10000);
end

plot(iters);
legend('Potenze','Jacobi','Gauss-Seidel');
xlabel('i (con tol = 10^{-i})');
ylabel('iterazioni');
```

```
function [x,i] = PotenzePR(H, p, tol)
  [m,n] = size(H);
  if m~=n
    error('Errore: H non quadrata');
  end
  delta = zeros(n, 1);
  for j=1:n
    for i=1:n
      delta(j) = delta(j) + H(i, j);
    end
  end
  for j=1:n
    for i=1:n
      if delta(j)==0
        H(i, j) = 0;
      else
        H(i, j) = H(i, j)/delta(j);
      end
    end
    if delta(j)==0
      delta(j)=1;
    else
      delta(j)=0;
    end
  end
  x = rand(n,1);
  x = x/norm(x);
```

```
28    imax = (log(tol)-log(2))/(log(p));
29    for i=1:imax
30      x = p*(H*x)+((1+p*(delta'*x -1))/n)*ones(n,1);
31    end
32  end
```

Nel quale mostriamo i grafici relativi dei 3 metodi usati confrontandoli rispetto al numero di iterazioni al variare della tolleranza:



Il metodo di Gauss-Seidell risulta il più efficiente in quanto per valori di tolleranza più piccoli converge in un numero minore delle iterazioni.