Elaborato di

Calcolo Numerico

Anno Accademico 2016/2017

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

September 2, 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 $-32768 \le$ `numero` ≤ 32767 ($-2^{15} \le$ `numero` $\le 2^{15} - 1$).
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,v(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 può notare che al diminuire del valore h, la funzione $\Psi_h(0)$ approssima con maggior precisione il valore $f'(0)$, come si può vedere dal plot 1.

1.5 Esercizio 5

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

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

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

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

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

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

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

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

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

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
format compact

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

for i= 2:7
    x(i+1) = (x(i)*x(i-1)+2)/(x(i)+x(i-1))
end

for i=3:8
    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
7	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:

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

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

1.8 Esercizio 8

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

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

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

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

1.9 Esercizio 9

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

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

1.10 Esercizio 10

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

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

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

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

e moltiplicare e dividere per questo valore:

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

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

1.11 Esercizio 11

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

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

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:

- $$\varepsilon_1 = \frac{((x(1 + \varepsilon_x) + y(1 + \varepsilon_y))(1 + \varepsilon_a) + z(1 + \varepsilon_z))(1 + \varepsilon_b) - (x + y + z)}{x + y + z} \approx$$
  $$\approx \frac{x(1 + \varepsilon_x + \varepsilon_a + \varepsilon_b) + y(1 + \varepsilon_y + \varepsilon_a + \varepsilon_b) + z(1 + \varepsilon_z + \varepsilon_b) - x - y - z}{x + y + z} \leq$$
  $$\leq \frac{3 \cdot x \cdot \varepsilon_m + 3 \cdot y \cdot \varepsilon_m + 2 \cdot z \cdot \varepsilon_m}{x + y + z} \leq \frac{3 \cdot \varepsilon_m \cdot (x + y + z)}{x + y + z} = 3 \cdot \varepsilon_m$$
- seguendo gli stessi procedimenti del punto precedente possiamo scrivere:
  $$\varepsilon_2 = \frac{(x(1 + \varepsilon_x) + (y(1 + \varepsilon_y) + z(1 + \varepsilon_z))(1 + \varepsilon_a))(1 + \varepsilon_b) - (x + y + z)}{x + y + z} =$$
  $$= \ldots \leq \frac{2 \cdot x \cdot \varepsilon_m + 3 \cdot y \cdot \varepsilon_m + 3 \cdot z \cdot \varepsilon_m}{x + y + z} \leq \frac{3 \cdot \varepsilon_m \cdot (x + y + z)}{x + y + z} = 3 \cdot \varepsilon_m$$

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

Il problema è che stiamo rappresentando dei numeri reali in un calcolatore, quindi la loro rappresentazione comporta delle approssimazioni. Nella riga 11 abbiamo calcolato e restituito in output il valore interno al logaritmo $\left|3(1 - \frac{3}{4}) + 1\right|$ 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 disp (abs(3*(1-4/3)+1))
```

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

2 Capitolo 2

Le funzioni usate nei codici seguenti sono in fondo al capitolo

2.1 Esercizio 1

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

La tabella delle approssimazioni è la seguente:

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

2.2 Esercizio 2

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

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

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

l'output corrispondente è:

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

2.3 Esercizio 3

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

```
x_0 = 3;
alpha = x_0;
n= 2;

s = SecSqrtN(n, alpha, x_0, 100, 0.0001);
```

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

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

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

2.4 Esercizio 4

```
funct = @(x) (x-pi)^10;
dfunct = @(x) 10*(x-pi)^9;
disp('newton modificato funct 1');
NM11 = NewtonMod(funct, dfunct, 1, 3, 50, 1, 0)
NM12 = NewtonMod(funct, dfunct, 1, 3, 50, 0.1, 0)
NM13 = NewtonMod(funct, dfunct, 1, 3, 50, 0.01, 0)
NM14 = NewtonMod(funct, dfunct, 1, 3, 50, 0.001, 0)
NM15 = NewtonMod(funct, dfunct, 1, 3, 50, 0.0001, 0)
NM16 = NewtonMod(funct, dfunct, 1, 3, 50, 0.00001, 0)
NM17 = NewtonMod(funct, dfunct, 1, 3, 50, 0.000001, 0)
disp('aitken funct 1');
A11 = Aitken(funct, dfunct, 3, 50, 1)
A12 = Aitken(funct, dfunct, 3, 50, 0.1)
A13 = Aitken(funct, dfunct, 3, 50, 0.01)
funct2 = @(x) ((x-pi)^10)*(exp(1)^(2*x));
dfunct2 = @(x) (5+x-pi)*(x-pi)^9*2*exp(1)^(2*x);
disp('newton modificato funct 2');
NM21 = NewtonMod(funct2, dfunct2, 1, 3, 50, 1, 0)
NM22 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.1, 0)
NM23 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.01, 0)
NM24 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.001, 0)
```

```
NM25 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.0001, 0)
NM26 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.00001, 0)
NM27 = NewtonMod(funct2, dfunct2, 1, 3, 50, 0.000001, 0)
disp('aitken funct 2');
A21 = Aitken(funct2, dfunct2, 3, 50, 1)
A22 = Aitken(funct2, dfunct2, 3, 50, 0.1)
A23 = Aitken(funct2, dfunct2, 3, 50, 0.01)
```

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

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

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

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

2.5 Esercizio 5

Il metodo di bisezione è applicabile in f se è:

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

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

2.6 Esercizio 6

```
myf = @(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);
disp(tol);
imax = 50;
count = zeros(3,10);
for i=1:10
    [temp, count(1,i)] = NewtonMod(myf,df,1,x0,imax,tol(i), 0);
    [temp, count(2,i)] = secanti(myf,x0,x1,tol(i),imax);
    [temp, count(3,i)] = corde(myf,feval(df,x0),x0,tol(i),imax);
end
```

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

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

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

2.7 Esercizio 7

```
format long

myf = @(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 = 50;
count = zeros(1,10);
for i=1:10
    [temp, count(1,i)] = bisect(myf,x0,x1,tol(i),imax);
    dips('B');
    i
    temp
end

disp('nella riga ci sono il numero di iterazioni del metodo della bisezione');
count
```

Il numero di iterazioni del metodo di bisezione risultanti sono:

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

2.8 Esercizio 8

```
myfunct = @(x) (x-pi)^(10*x);
dfdx =  @(x) 10*((x-pi)^(10*x))*(x/(x-pi)+log(x-pi));
x0 = 0;

y = NewtonMod(myfunct,dfdx, 1, x0, 5, 0.01, 1);
```

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

2.9 Funzioni MatLab Usate

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

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

end
```

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

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

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

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

```
        i = i+1;
        x = (f(x1,n, alpha) * x0 - f(x0,n, alpha)*x1 ) / (f(x1, n, alpha) - f(x0, n, alpha));
        disp(x)
    end
    y = x;
end


function y = f(x, n, alpha)
    y = (x ^ n) - alpha;
end
```

2.9.4 Metodo di Newton Modificato

```
function [y, i] = NewtonMod(f, df, m, x0, imax, tol, output)
    format long;
    i = 0;
    x = x0;
    vai=1;
    while((i < imax) && vai)
        i = i+1;
        fx = feval(f, x0);
        dfx = feval(df, x0);
        if(dfx ~= 0)
            x = x0 - m * fx / dfx;
        else
            break;
        end
        if(abs(x-x0)<tol)
            vai = 0;
        end
        x0 = x;
    end
    if(output)
        if(vai)
            disp('[Errore] Tolleranza non calcolabile');
        else
           disp(i);
        end
    end
    y = x;
end
```

2.9.5 Metodo di accelerazione di Aitken

```
function y = Aitken( f, df, x0, imax, tol )
    format long;
    i = 0;
    vai=1;

    while((i < imax) && vai)
        x1 = NewtonMod(f, df, 1, x0, 1, tol, 0);
        x2 = NewtonMod(f, df, 1, x1, 1, tol, 0);
        i = i+1;
        if((x0 - 2*x1 +x2) == 0)
```

```
            disp('[Errore] Divisione per 0');
            vai = 0;
            break;
        end
        x = (x2*x0 - x1^2)/(x0 - 2*x1 +x2);
        if(abs(x-x0)<tol)
            vai = 0;
        end
        x0 = x;
    end
    if(vai)
        disp('[Errore] Tolleranza non calcolabile');
        disp(i);
    end
    y = x;
end
```

2.9.6 Metodo delle secanti

```
function [x,nit,tolf]=secanti(f,x0,x1,tolx,maxit)
  nit=0;
  fx0=feval(f,x0);
  err=abs(x1-x0);
  while (nit<maxit & err>tolx)
      fx1=feval(f,x1);
      dfx1=(fx1-fx0)/(x1-x0);
      tolf=tolx*abs(dfx1);
      if abs(fx1)<=tolf
         break
      end
      x2=x1-(fx1/dfx1);
      err=abs(x2-x1);
      x0=x1;
      x1=x2;
      fx0=fx1;
      nit=nit+1;
  end
  x=x1;
end
```

2.9.7 Metodo della bisezione

```
function [sol,nit]=bisect(f,a,b,tolx,maxit);
    nit=maxit;
    j=0;
    if(subs(f,a)*subs(f,b)>0)
        disp('[Warning] il metodo non puo'' essere usato');
    else
        while(j<=maxit)
            sol=(a+b)/2;
            ff=subs(f,sol);
            if(abs(ff)<=tolx)
                nit=j;
                break;
        end
        fa=subs(f,a);
```

```
        fm=subs(f,sol);
        if(fa*fm<=0)
            b=sol;
        else
            a=sol;
        end
        j=j+1;
    end
end
```

2.9.8 Metodo delle corde

```
function [x,nit]=corde(f,m,x0,tolx,maxit)
  nit=0;
  err=tolx+1;
  x=x0;
  while (nit<maxit && err>tolx)
      fx=feval(f,x);
      tolf=tolx*abs(m);
      if abs(fx)<=tolf
         break
      end
      x1=x-fx/m;
      err=abs(x1-x);
      x=x1;
      nit=nit+1;
  end
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:

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

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

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

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 la somma del prodotto degli elementi delle due matrici:

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

questa somma può essere scritta anche:

$$\sum_{m=1}^{n} l_{i,m} \cdot k_{m,j} = \underbrace{\sum_{i<j} l_{i,m} \cdot k_{m,j}}_{0} + \sum_{i\geq j} l_{i,m} \cdot k_{m,j}$$

da quest'ultima il valore $0 = \sum_{i<j} l_{i,m} \cdot k_{m,j} = a_{i,j} \quad i,j \in [1,..,n]$ che è la definizione di matrice triangolare inferiore, come volevasi dimostrare.
(Allo stesso modo si può dimostrare per matrici triangolari superiori).

3.2 Esercizio 2

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

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

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:

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

che sono somme di matrici strettamente triangolari inferiori, questo implica che $(I_n + U)^{-1}$ è di quel tipo. Abbiamo quindi dimostrato che anche A^{-1} è una matrice triangolare inferiore.
Nel caso in cui A sia una matrice triangolare inferiore a diagonale unitaria la dimostrazione non varia dato che gli elementi dell'inversa di D rimangono unitari nel processo di inversione.
(Allo stesso modo si può dimostrare per matrici triangolari superiori).

3.4 Esercizio 4

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

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

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

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

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

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

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

3.5 Esercizio 5

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

```
function [L,U,P]=factLUP(A)
[m,n]=size(A);
if m~=n
    error('[Errore] La matrice inserita A inserita non e'' quadrata');
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('[Attenzione] La matrice 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
```

3.6 Esercizio 6

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

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

3.7 Esercizio 7

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

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

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

- simmetriche:
$$(AA^T)^T = (A^T)^T A^T = AA^T$$
$$(A^TA)^T = A^T(A^T)^T = A^TA.$$

- definite positive:
$$x^TAA^Tx = xx^TAA^Txx^T = x(A^Tx)^T(x^TA)^Tx^T = (\underbrace{x^TA^Tx}_{>0})^T \cdot (\underbrace{x^TAx}_{>0})^T > 0$$
$$x^TA^TAx = xx^TA^TAxx^T = x(Ax)^T(x^TA^T)^Tx^T = (\underbrace{x^TAx}_{>0})^T \cdot (\underbrace{x^TA^Tx}_{>0})^T > 0$$

possiamo quindi affemrare che le matrici AA^T e A^TA sono SDP.

3.8 Esercizio 8

Se la matrice A ha rango massimo significa che la matrice è invertibile, di conseguenza il suo determinante è non nullo che implica che la matrice è nonsingolare. Dalla dimostrazione dell'esercizio precedente (**Esercizio 7**) si ha quindi che se la matrice ha rango massimo allora A^TA 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:
$$x^TAx = x^T(A_s + A_a)x = x^TA_sx + x^TA_ax = x^TA_sx + x^T\frac{(A - A^T)}{2}x =$$
$$= x^TA_sx + \underbrace{\frac{1}{2}(x^TAx - x^TA^Tx)}_{=0} = x^TA_sx$$
il termine $\frac{1}{2}(x^TAx - x^TA^Tx) = \frac{1}{2}(x^TAx - (Ax)^Tx)$ è pari a zero e possiamo vederlo tramite la sostituzione $y = Ax$:
$$x^TAx - (Ax)^Tx \underbrace{=}_{y=Ax} x^Ty - y^Tx = 0$$
dato che $x^Ty = y^Tx$ 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:
$$\sum_{i=1}^{n} 2(i-1)(n-i) = 2\sum_{i=1}^{n}(i \cdot n - i^2 - n) = 2(n\sum_{i=1}^{n} i - \sum_{i=1}^{n} i^2 - n^2) =$$
$$= 2\Big(n\frac{n(n-1)}{2} + n^2 - \frac{n(n-1)(2n-1)}{6} - n^2\Big) = 2\Big(\frac{n^3}{2} - \frac{n^2}{2} - \frac{2n^3}{6} + \frac{n^2}{6} + \frac{2n^2}{6} - \frac{n}{6}\Big) =$$
$$= n^3 - n^2 - \frac{2n^3}{3} + \frac{n^2}{3} + \frac{2n^2}{3} - \frac{n}{3} \approx \frac{n^3}{3}$$
quindi l'algoritmo di fattorizzazione LDL^T ha un costo di $\frac{n^3}{3}$ *flop*.

3.11 Esercizio 11

3.12 Esercizio 12

3.13 Esercizio 13

3.14 Esercizio 14

3.15 Esercizio 15

3.16 Esercizio 16

3.17 Esercizio 17

3.18 Esercizio 18

3.19 Esercizio 19

3.20 Esercizio 20

3.21 Esercizio 21

3.22 Funzioni MatLab Usate

3.22.1 Algoritmo di fattorizzazione LU con pivoting parziale

```
function [L,U,P]=factLUP(A)
[m,n]=size(A);
if m~=n
    error('[Errore] La matrice inserita A inserita non e'' quadrata');
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('[Attenzione] La matrice 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
```

3.22.2 Risoluzione sistema lineare con funzione di ingresso già fattorizzata LU

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

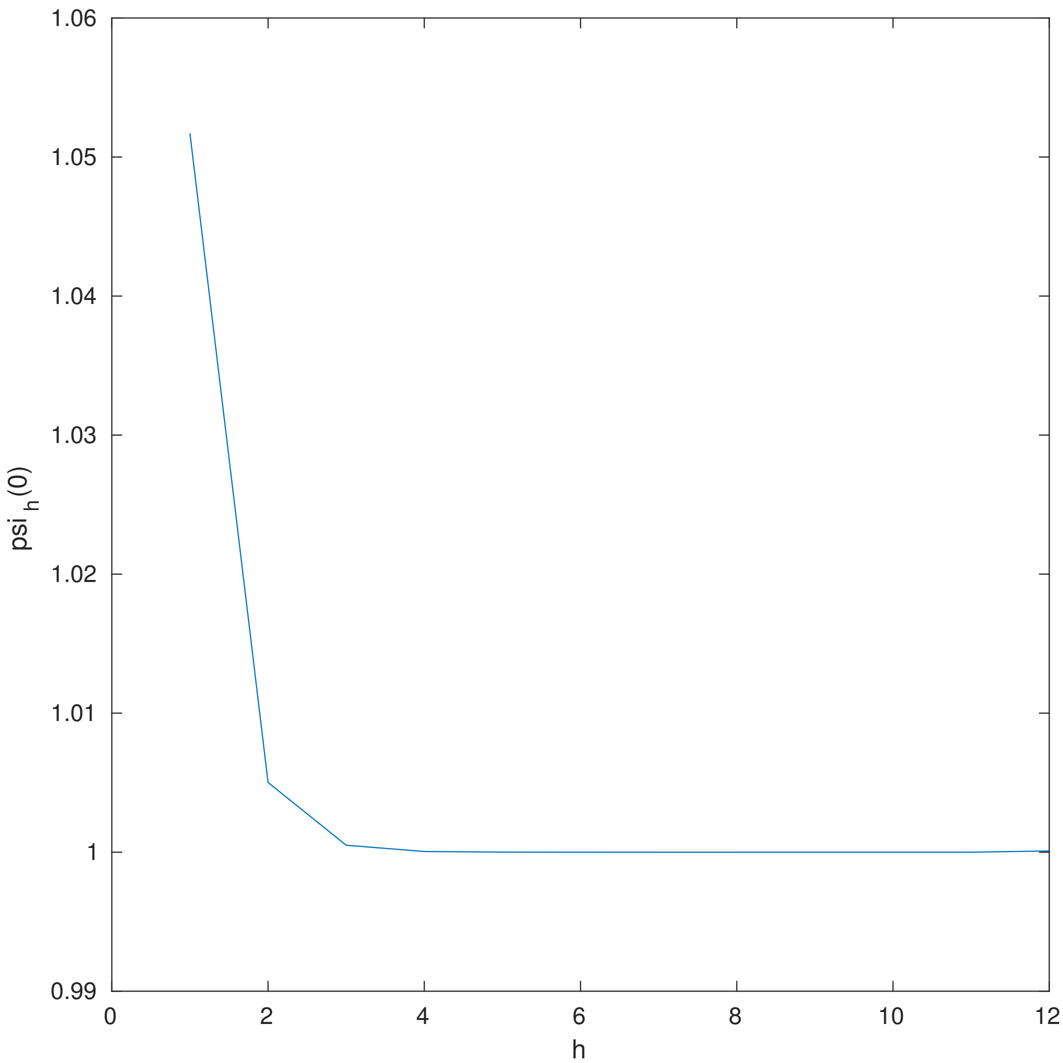
```
            b(i) = b(i)-A(i,j)*b(j);
        end
    end
end
```

```
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
```

4 Grafici

4.1 Esercizio 1.4

Figure 1: Esercizio 1.4



4.2 Esercizio 1.13

Figure 2: Esercizio 1.13