Elaborato di

Calcolo Numerico

Anno Accademico 2017/2018

Moncini Federico - 5936828
Capecchi Tommaso - 5943118

August 30, 2018

Capitoli

1 Capitolo

1.1

Sia $x = e \approx 2.7183 = \tilde{x}$. Si calcoli il corrispondente errore relativo ϵ_x e il numero di cifre significative k con cui $\tilde{x}$ approssima x. Si verifichi che

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

Soluzione

L'errore relativo è definito come $|\epsilon_x| = \frac{|x-\hat{x}|}{|x|}$, dalla quale ricavo la $\hat{x}$ come segue:
$\hat{x} = x(1 + \epsilon_x)$ e dunque $\frac{\hat{x}}{x} = 1 + \epsilon_x$. Ciò ci suggerisce che l'errore relativo deve essere comparato a 1; esso sarà piccolo se si avvicina allo zero (ciò implica che il risultato approssimato sarà vicino al risultato esatto di un dato problema), viceversa un errore relativo vicino a 1 comporta una quasi totale perdita di informazione. Calcoliamo quindi l'errore relativo:

$$|\epsilon_x| = \frac{|e-2.7183|}{|e|} = 7.3576e - 06$$

Calcoliamo adesso il numero di cifre significative, ricavabili dalla seguente formula $k \approx -\log_{10}(2|\epsilon_x|)$

$$k \approx -\log_{10}(2|\epsilon_x|) \Longrightarrow k \approx 4.8322 \approx 5$$

Verifico ora che $|\epsilon_x| \approx \frac{1}{2}10^{-k}$. Infatti:

$$|\epsilon_x| \approx 5e - 06$$

1.2

Usando gli sviluppi di Taylor fino al secondo ordine con resto in forma di Lagrange, si verifichi che se $f \in C^3$, risulta

$$f'(x) = \phi_h(x) + O(h^2) \text{ dove } \phi_h(x) = \frac{f(x+h)-f(x-h)}{2h}.$$

Soluzione

$f'(x) = \phi_h(x) + O(h^2)$

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

Per mezzo degli sviluppi di Taylor fino al secondo ordine ottengo che:

$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + O[(x - x_0)^3$

Ricavo dunque $f(x + h)$ e $f(x - h)$:

$f(x + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x^2) + O(h^3)$

$f(x - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x^2) + O(h^3)$

Ottengo quindi:

$f'(x) = \frac{f(x_0)+hf'(x_0)+\frac{h^2}{2}f''(x_0)+O(h^3)-f(x_0)+hf'(x_0)-\frac{h^2}{2}f''(x_0)+O(h^3)}{2h} \Longrightarrow$

$f'(x) = \frac{2hf'(x_0)+O(h^3)}{2h} = f'(x_0) + O(h^2)$

1.3

Utilizzando Matlab, si costruisca una tabella dove, per $h = 10^{-j}, j = 1, ..., 10$ e per la funzione $f(x) = x^4$ si riporta il valore di $\theta_h(x)$ definito nell'Esercizio 1 in x = 1. Commentare i risultati ottenuti.

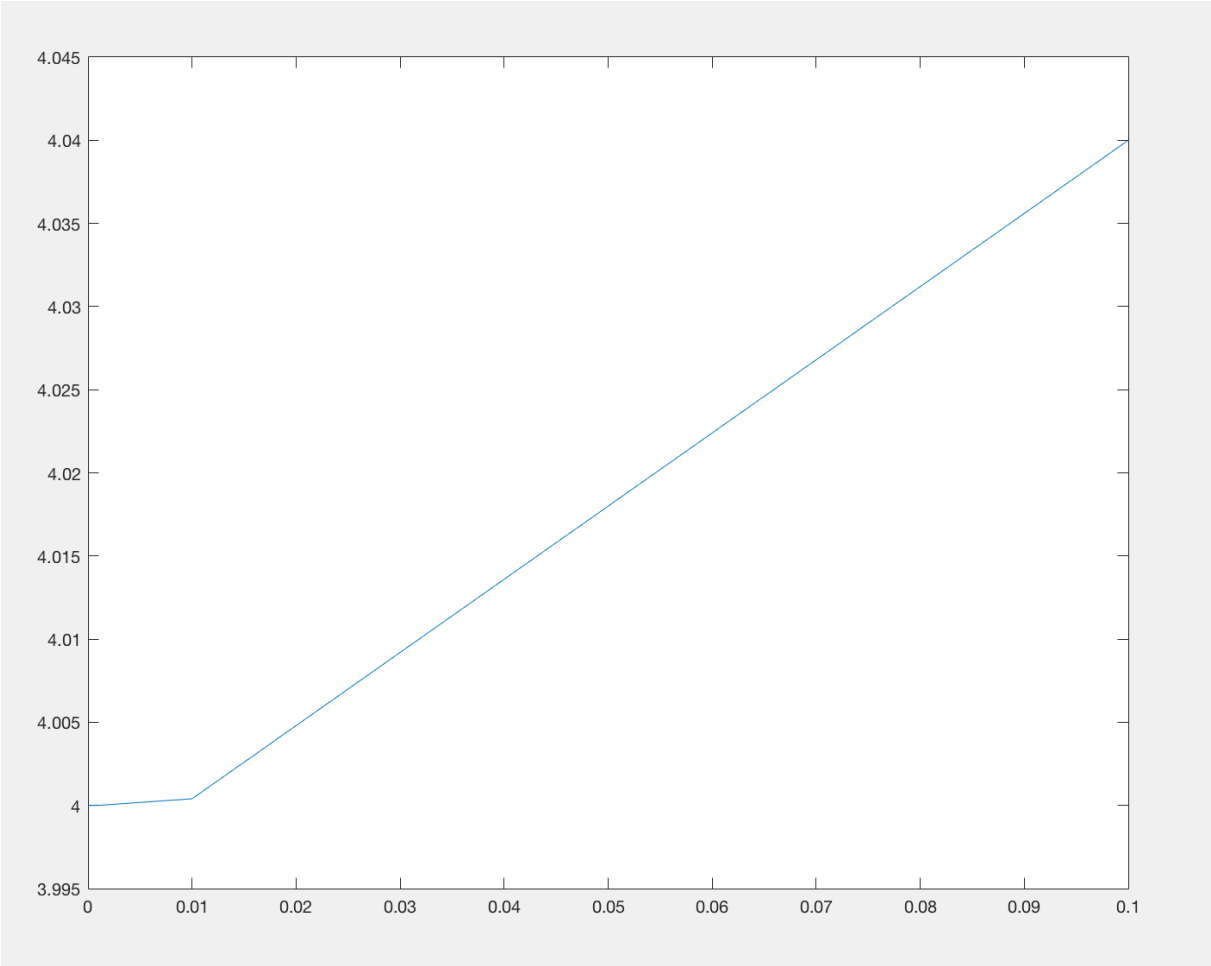
Soluzione

Il seguente codice riguarda la soluzione dell'esercizio 3 con la relativa tabella dei risultati:

```
format long e;

%   h = vettore contente valori di 10^-j per j = 1..10

h = zeros(10,1);

%   t = vettore contente valori di teta(x) = (f(x+h)-f(x-h))
   /2*h
%   con x = 1

t = zeros(10,1);

for i = 1:10
    h(i) = 10^-i;
    t(i) = computeTeta(1,h(i));
end

plot(h,t);

function rst = computeTeta(x,h)
        %   Calcola la funzione teta(x) = (f(x+h)-f(x-h))/2*h
        sum = x+h;
        diff = x-h;
        rst = (sum^4 - diff^4)/(2*h);
end
```

h	$\theta_h(1)$
10^{-1}	4.040000000000002
10^{-2}	4.000400000000004
10^{-3}	4.000003999999723
10^{-4}	4.000000039999230
10^{-5}	4.000000000403681
10^{-6}	3.999999999948489
10^{-7}	4.000000000115023
10^{-8}	4.000000003445692
10^{-9}	4.000000108916879
10^{-10}	4.000000330961484

Come si nota dalla tabella, la funzione $f(x) = x^4$ decresce fino a $i = 10^{-6}$ dopodiché cresce, come si puo notare anche dal seguente grafico:



Funzione $\theta_h(1)$

1.4

Si dia una maggiorazione del valore assoluto dell'errore relativo con cui $x + y + z$ viene approssimato dall'approssimazione prodotta dal calcolatore, ossia $(x \oplus y) \oplus z$ (supporre che non ci siano problemi di overflow o di underflow). Ricavare l'analoga maggiorazione anche per $x \oplus (y \oplus z)$ tenendo presente che $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.

Soluzione

Ricavo la maggiorazione del valore assoluto dell'errore di $(x \oplus y) \oplus z$:

$$\epsilon = \frac{(1+\epsilon_2)[(1+\epsilon_1)(x(1+\epsilon_x)+y(1+\epsilon_y))+z(1+\epsilon_z)]-(x+y+z)}{x+y+z} =$$

$$\frac{(1+\epsilon_2)\{[(1+\epsilon_1)(x+x\epsilon_x+y+y\epsilon_y)]+z+z\epsilon_z\}-x-y-z}{x+y+z} =$$

$$\frac{(1+\epsilon_2)\{x+x\epsilon_x+y+y\epsilon_y+x\epsilon_1+x\epsilon_x\epsilon_1+y\epsilon_1+y\epsilon_y\epsilon_1+z+z\epsilon_z\}-x-y-z}{x+y+z}$$

$$\frac{(1+\epsilon_2\{x(1+\epsilon_x+\epsilon_1+\epsilon_x\epsilon_1)+y(1+\epsilon_y+\epsilon_1+\epsilon_y\epsilon_1)+z(1+\epsilon_z)\}-x-y-z}{x+y+z}$$

$$\frac{(x+x\epsilon_2)(1+\epsilon_x+\epsilon_1+\epsilon_x\epsilon_1)+(x+y\epsilon_2)(1+\epsilon_y+\epsilon_1+\epsilon_y\epsilon_1)+(z+z\epsilon_2)(1+\epsilon_z)-x-y-z}{x+y+z}$$

$$\leq |\frac{x\epsilon_x+y\epsilon_y+z\epsilon_z+x\epsilon_1+y\epsilon_1+x\epsilon_2+y\epsilon_2+z\epsilon_2}{x+y+z}| = |\frac{x\epsilon_x+y\epsilon_y+z\epsilon_z+\epsilon_1(x+y)+\epsilon_2(z+y+z)}{x+y+z}|$$

$$\leq \frac{|x||\epsilon_x|+|y||\epsilon_y|+|z||\epsilon_z|+|\epsilon_1||x+y|+|\epsilon_2||x+y+z|}{|x+y+z|}$$

$$\leq \frac{\epsilon_m(|x|+|y|+|z|)+|x+y|+|x+y+z|}{|x+y+z|} = \epsilon_m(\frac{|x|+|y|+|z|}{|x+y+z|} + |\frac{|x+y|}{|x+y+z|} + 1)$$

Per ricavare la maggiorazione del valore assoluto dell'errore di $x \oplus (y \oplus z)$ è necessario scambiare al posto della x la lettera z, e si otterrà l'analoga maggiorazzione:

$$\epsilon_m(\frac{|x|+|y|+|z|}{|x+y+z|} + |\frac{|z+y|}{|x+y+z|} + 1)$$

Otteniamo quindi che i valori degli errori ε_1 e ε_2 sono condizionati rispettivamente, dai valori $\frac{|x+y|}{|x+y+z|}$ e $\frac{|y+z|}{|y+z+x|}$.

1.5

Eseguire le seguenti istruzioni in Matlab:

$$\text{x} = 0;\ \text{count} = 0;$$
$$while x \sim 1,\ \text{x} = \text{x} + \text{delta},\ \text{count} = \text{count} + 1,\ \text{end}$$

dapprima ponendo $delta = \frac{1}{16}$ e poi ponendo $delta = \frac{1}{20}$. Commentare i risultati ottenuti e in particolare il non funzionamento del secondo caso.

Soluzione

```
x = 0;
count = 0;
delta = 1/20;
while x ~= 1
    x = x+delta;
    count = count +1;
end
```

• Analizziamo il caso in cui $delta = \frac{1}{16}$

Il programma termina correttamente poiche $delta = \frac{1}{16} = 0.0625$ e dopo 16 iterazioni il valore raggiunta da x è proprio 1, che rispecchia la condizione di uscita del ciclo while.

•Analizziamo il caso in cui $delta = \frac{1}{20}$

In questo caso il programma non termina poiche il controllo sul ciclo while non viene verificato correttamente; infatti $delta = \frac{1}{20} = [0.05]_{10}$, che rappresentato in base 2 risulta $[0.00\overline{0011}]_2$. Questo fa si che l'operazione di somma ad ogni iterazione riguardi numeri periodici che essendo approssimati non raggiungeranno ma il valore x = 1, dunque il ciclo non terminerà mai. Una soluzione protrebbe essere quella di sostituire il controllo del while con uno piu efficiente, come: abs(x-1) > eps

1.6

Verificare che entrambe le seguenti successioni convergono a $\sqrt{3}$, (riportare le successive approssimazioni in una tabella a due colonne, una per ciascuna successione),

$$x_{k+1} = \frac{(x_k + \frac{3}{x_k})}{2}, x_0 = 3;$$

$$x_{k+1} = \frac{3 + x_{k-1} x_x}{x_{k-1} + x_k}, x_0 = 3; x_1 = 2$$

Per ciascuna delle due successioni, dire quindi dopo quante iterazioni si ottiene un'approssimazione con un errore assoluto minore o uguale a 10^{-12} in valore assoluto.

Soluzione

- $x_{k+1} = \frac{(x_k + \frac{3}{x_k})}{2}, x_0 = 3;$

```
% x = valore verso il quale la successione deve convergere
% x0 = punto iniziale della successione
%tol = tolleranza richiesta
%err = errore commesso ad ogni iterazione
%count = contatore relativo alle iterazioni effettuate

x = sqrt(3);
x0 = 3;
tol = 10e-12;
err = x0-x;
count =0;
while err >= tol
    x0 = (x0+(3/x0))/2;
    err = x0-x;
    count = count+1;
end
```

Il codice precedente restituisce i seguenti risultati:

k	x_k	ϵ_k
0	3.000000000000000	$\epsilon_0 = 1.267949e+00$
1	2.00000000000000	$\epsilon_1 = 2.679492e-01$
2	1.750000000000000	$\epsilon_2 = 1.794919e-02$
3	1.732143000000000	$\epsilon_3 = 9.204957e-05$
4	1.732051000000000	$\epsilon_4 = 2.445850e-09$
5	1.732051000000000	$\epsilon_5 = 0e+00$

Si vede quindi che per k >= 5 si ha un errore assoluto che equivale a 0, ovvero una quantita minore o uguale di 10^{-12}

•$x_{k+1} = \frac{3+x_{k-1}x_x}{x_{k-1}+x_k}, x_0 = 3; x_1 = 2$

```
% x = valore verso il quale la successione deve convergere
% x0 = punto iniziale della successione
% x1 = secondo punto della successione
% tol = tolleranza richiesta
% err0 = errore iniziale
% err1 = errore al passo 1
% count = contatore relativo alle iterazioni effettuate

x = sqrt(3);
x0 = 3;
x1 = 2;
tol = 10e-12;
err0 = x0-x;
err1 = x1-x;
count = 1;
while err1 >= tol
    temp = x1;
    x1 = (3+(x0*x1))/(x0+x1);
    x0 = temp;
    err1 = x1-x;
    count = count+1;
    fprintf(Errore end
```

Il codice precedente restituisce i seguenti risultati:

k	x_k	ϵ_k
0	3.000000000000000	$\epsilon_0 = 1.267949e+00$
1	2.00000000000000	$\epsilon_1 = 2.679492e-01$
2	1.80000000000000	$\epsilon_2 = 6.794919e-02$
3	1.736842000000000	$\epsilon_3 = 4.791298e-03$
4	1.732143000000000	$\epsilon_4 = 9.204957e-05$
5	1.7320510000000	$\epsilon_5 = 1.271372e-07$
6	1.7320510000000	$\epsilon_6 = 3.378631e-12$

Si vede quindi che per k >= 6 si ha un errore assoluto che equivale a 0, ovvero una quantita minore o uguale di 10^{-12}

2 Capitolo

2.1

Determinare analiticamente gli zeri del polinomio $P(x) = x^3 - 4x^2 + 5x - 2$ e la loro molteplicità. Dire perchè il metodo di bisezione è utilizzabile per approssimarne uno a partire dall'intervallo di confidenza *[a , b] = [0 , 3]*. A quale zero di P potrà tendere la successione generata dal metodo di bisezione a partire da tale intervallo? Costruire una tabella in cui si riportano il numero di iterazioni e di valutazioni di P richieste per valori decrescenti della tolleranza tolx.

Studio analitico del polinomio $P(x) = x^3 - 4x^2 + 5x - 2$.

- **Zeri del polinomio**
  Per trovare gli zeri del polinomio occorre scomporlo nel seguente modo:

$$x^3 - 4x^2 + 5x - 2 =$$
$$= (x-2)(x^2 - 2x + 1) =$$
$$= (x-2)(x-1)^2$$

  Da cui si deduce che $P(x) = 0$ per $(x-2) = 0 \Rightarrow x = 2$ e $(x-1) = 0 \Rightarrow x = 1$.

- **Molteplicità**
  I valori di x precedentemente calcolati vengono definiti come *radici* del polinomio. Si dice che a è una radice di *P(x)* con *molteplicità* n se e solo se *P(x)* è divisibile per $(x-a)^n$, ma non è divisibile per $(x-a)^{n-1}$. Inoltre si dice che x ha *molteplicità esatta* $n \geq 1$, se:

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

  •$x = 2$

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

  •$x = 2$

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

In questo caso la radice $x = 2$ viene definita *semplice* in quanto ha molteplicità $m = 1$, mentre la radice $x = 1$ si definisce *multipla* data la molteplicità $m = 2$.

Il requisito per poter applicare il *metodo di bisezione* in un intervallo $[a, b]$ è che sia $f(a)f(b) < 0$ in modo da garantire l'esistenza di almeno uno zero. Per il polinomio $P(x)$ è possibile applicare il *metodo di bisezione* nell'intervallo $[0, 3]$ poichè il requisito è soddisfatto. In questo caso si ha: $P(0) * P(3) = (-2) * (4) = -8$.

Il seguente codice MatLab, riguarda il **Metodo di bisezione**:

```
function y = Bisezione_Es1(a,b,tol,f)
    %y = Bisezione_Es1(a,b,tol,f) calcola l'approssimazione
       di una radice
    %della funzione f con tolleranza tol sull'errore, nell'
       intervallo [a,b]
    fa = feval(f,a);
    fb = feval(f,b);
    if fa*fb >= 0
        warning(Metodo non applicabile);
    end
    y = (a+b)/2;
    fy = feval(f,y);
    max = ceil(log2(b-a)-log2(tol));
    for i = 1:max
        f1x = abs((fb-fa)/(b-a)); %calcolo la derivata prima
           per sostituire il controllo fx ==0
        if abs(fy)<= tol*f1x
            break;
        elseif fa*fy < 0
            b = y;
            fb = fy;
```

```
        else
            a = y;
            fa = fy;
        end
        y = (a+b)/2;
        fy = feval(f,y);
    end
end
```

Nel seguente codice Matlab viene applicato il *metodo di bisezione* al polinomio $P(x)$ sull'intervallo $[0, 3]$, con una tolleranza iniziale pari a $10^{-}1$, la quale viene decremntata di un fattore 10 ad ogni iterazione:

```
%soluzione esercizio1 capitolo 2
a = 0;
b =3;
f = @(x) x^3-4*x^2+5*x-2;
f1 = @(x) x^2+2;
tol = 1e-1;
tolx = [];
rstApprox = [];
index = 1;
while tol>eps
    tolx(index) = tol;
    rstApprox(index) = Bisezione_Es1(a,b,tol,f);
    tol = tol/10;
    index = index+1;
end
%si nota che la radice verso il quale tende la successione
   generata
%dal metodo di bisezione, nell'intervallo [0,3] tende a 2;
```

Da i seguenti valori riportati dall'esecuzione del codice è possibile notare che la successione generata dal metodo di *bisezione* converge alla radice $x = 2$:

tol_x	*Bisezione*	*Num. Iterazioni*
10^{-1}	$\tilde{x} = 1.500000000000000$	$ib = 0$
10^{-2}	$\tilde{x} = 1.992187500000000$	$ib = 6$
10^{-3}	$\tilde{x} = 2.000976562500000$	$ib = 9$
10^{-4}	$\tilde{x} = 2.000061035156250$	$ib = 13$
10^{-5}	$\tilde{x} = 1.999992370605469$	$ib = 16$
10^{-6}	$\tilde{x} = 2.000000953674316$	$ib = 19$
10^{-7}	$\tilde{x} = 2.000000059604645$	$ib = 23$
10^{-8}	$\tilde{x} = 1.999999992549419$	$ib = 26$
10^{-9}	$\tilde{x} = 2.000000000931323$	$ib = 29$
10^{-10}	$\tilde{x} = 2.000000000058208$	$ib = 33$
10^{-11}	$\tilde{x} = 1.999999999992724$	$ib = 36$
10^{-12}	$\tilde{x} = 2.000000000000910$	$ib = 39$
10^{-13}	$\tilde{x} = 2.000000000000057$	$ib = 43$
10^{-14}	$\tilde{x} = 1.999999999999993$	$ib = 46$
10^{-15}	$\tilde{x} = 2.000000000000001$	$ib = 49$

2.2

Completare la tabella precedente riportando anche il numero di iterazioni e di valutazioni di P richieste dal metodo di Newton, dal metodo delle corde e dal metodo delle secanti (con secondo termine della successione ottenuto con Newton) a partire dal punto $x_0 = 3$. Commentare i risultati riportati in tabella. E‘ possibile utilizzare $x_0 = \frac{(}{5}3)$ come punto di innesco?

Abbiamo visto come il polinomio $P(x) = x^3 - 4x^2 + 5x - 2$, in $P(x) = 0$ presenta due radici, una con molteplicità multipla $x = 1$ e una con molteplicità semplice $x = 2$.

Di seguito sono riportati tre codici MatLab, rispettivamente:

- **Metodo di Newton**

```
1  function [y,i] = newtonSolve(x0,itmax,fx,f1x,tol)
2      %Funzione che ritorna l'approssimazione della radice
3      %della funzione fx data in input usando il metodo di
          Newton
4      %INPUT:
5      %x0 = punto iniziale dell'iterazione
6      %itmax = numero di iterazioni massime
7      %fx = funzione data in input
8      %f1x = derivata della funzione
9      %tol = tolleranza usata per accettare l'
          approssimazione
10     %OUTPUT:
11     %y = risultato approssimazione
12     %i = numero di iterazioni compiute
13
14     tempFx = feval(fx,x0);
15     tempF1x = feval(f1x,x0);
16     y = x0-(tempFx/tempF1x);
17     i = 1;
18     while (i < itmax) && (abs(y-x0) > tol)
19         i = i+1;
20         x0 = y;
21         tempFx = feval(fx,x0);
22         tempF1x = feval(f1x,x0);
23         y = x0-(tempFx/tempF1x);
24     end
```

```
    if abs(y-x0) > tol
        warning('La funzione non converge alla radice');
    end
end
```

- Metodo delle Corde

```
function [y,i] = cordeSolve(x0,itmax,fx,f1x,tol)
    %Funzione che ritorna l'approssimazione della radice
    %della funzione fx data in input usando il metodo
       delle Corde
    %INPUT:
    %x0 = punto iniziale dell'iterazione
    %itmax = numero di iterazioni massime
    %fx = funzione data in input
    %f1x = derivata della funzione
    %tol = tolleranza usata per accettare l'
       approssimazione
    %OUTPUT:
    %y = risultato approssimazione
    %i = numero di iterazioni compiute

    tempFx = feval(fx,x0);
    f1x = feval(f1x,x0);
    y = x0-(tempFx/f1x);
    i = 1;
    while (i < itmax) && (abs(y-x0) > tol)
        i = i+1;
        x0 = y;
        tempFx = feval(fx,x0);
        y = x0-(tempFx/f1x);
    end
    if abs(y-x0) > tol
        warning('La funzione non converge alla radice');
    end
end
```

- Metodo delle Secanti

```
function [y,i] = secantiSolve(x0,itmax,fx,f1x,tol)
```

```
    %Funzione che ritorna l'approssimazione della radice
    %della funzione fx data in input usando il metodo
       delle Secanti
    %INPUT:
    %x0 = punto iniziale dell'iterazione
    %itmax = numero di iterazioni massime
    %fx = funzione data in input
    %f1x = derivata della funzione
    %tol = tolleranza usata per accettare l'
       approssimazione
    %OUTPUT:
    %y = risultato approssimazione
    %i = numero di iterazioni compiute
    tempFx = feval(fx,x0);
    f1x = feval(f1x,x0);
    y = x0-(tempFx/f1x);
    i = 1;
    while(i<itmax) && (abs(y-x0) > tol)
        i = i+1;
        tempFx = feval(fx,x0);
        f1x = feval(fx,y);
        temp = y;
        y = ((x0*f1x)-(y*tempFx))/(f1x-tempFx);
        x0 = temp;
    end
    if abs(y-x0) > tol
        warning('La funzione non converge alla radice');
    end
end
```

Il seguente codice MatLab, riguarda il polinomio $P(x) = x^3 - 4x^2 + 5x - 2$, sul quale vengono eseguiti il metodo di Newton, il metodo delle Corde e il metodo delle Secanti (con secondo termine della successione ottenuto con Newton), valore di $tol_x = 10^{-1}$ che decresce ad ogni passaggio, pd che indica la derivata del polinomio, numero di iterazioni massime 1000 e punto di partenza $x_0 = 3$:

```
% Soluzione Cap_2 Es_2.
%
% -f: polinomio;
```

```
% -f1x: derivata prima del polinomio;
% -tol: tolleranza;
% -t: vettore contenente i valori di tolleranza ad ogni passo
   ;
% -n: vettore contenente i valori del metodo di newton ad
   ogni passo;
% -c: vettore contenente i valori del metodo delle corde ad
   ogni passo;
% -s: vettore contenente i valori del metodo delle secanti ad
    ogni passo.

f = @(x) x^3-4*x^2+5*x-2;
f1x = @(x) 3*x^2-8*x+5;
tol = 10^-1;
t = [];
n = [];
c = [];
s = [];
j = 1;

% Iterazione fino a una tolleranza di 10^(-17)

while tolx>10^(-17)
    t(j) = tolx;
    n(j) = newtonSolve(3, 1000, f, f1x, tol);
    c(j) = cordeSolve(3, 1000, f, f1x, tol);
    s(j) = secantiSolve(3, 1000, f, f1x, tol);
    tol = tol/10;
    j = j+1;
end
```

restituisce i seguenti valori:

tol_x	*Newton*		*Corde*		*Secanti*	
10^{-1}	$\tilde{x} = 2.004$	$n = 4$	$\tilde{x} = 2.276$	$c = 3$	$\tilde{x} = 2.137$	$s = 4$
10^{-2}	$\tilde{x} = 2.000$	$n = 5$	$\tilde{x} = 2.055$	$c = 12$	$\tilde{x} = 2.010$	$s = 6$
10^{-3}	$\tilde{x} = 2.000$	$n = 6$	$\tilde{x} = 2.006$	$c = 27$	$\tilde{x} = 2.000$	$s = 7$
10^{-4}	$\tilde{x} = 2.000$	$n = 6$	$\tilde{x} = 2.000$	$c = 44$	$\tilde{x} = 2.000$	$s = 8$
10^{-5}	$\tilde{x} = 2.000$	$n = 7$	$\tilde{x} = 2.000$	$c = 62$	$\tilde{x} = 2.000$	$s = 9$
10^{-6}	$\tilde{x} = 2.000$	$n = 7$	$\tilde{x} = 2.000$	$c = 79$	$\tilde{x} = 2.000$	$s = 9$
10^{-7}	$\tilde{x} = 2.000$	$n = 7$	$\tilde{x} = 2.000$	$c = 96$	$\tilde{x} = 2.000$	$s = 9$
10^{-8}	$\tilde{x} = 2.000$	$n = 7$	$\tilde{x} = 2.000$	$c = 113$	$\tilde{x} = 2.000$	$s = 10$
10^{-9}	$\tilde{x} = 2.000$	$n = 8$	$\tilde{x} = 2.000$	$c = 131$	$\tilde{x} = 2.000$	$s = 10$
10^{-10}	$\tilde{x} = 2.000$	$n = 8$	$\tilde{x} = 2.000$	$c = 148$	$\tilde{x} = 2.000$	$s = 10$
10^{-11}	$\tilde{x} = 2.000$	$n = 8$	$\tilde{x} = 2.000$	$c = 165$	$\tilde{x} = 2.000$	$s = 10$
10^{-12}	$\tilde{x} = 2.000$	$n = 8$	$\tilde{x} = 2.000$	$c = 182$	$\tilde{x} = 2.000$	$s = 11$
10^{-13}	$\tilde{x} = 2.000$	$n = 8$	$\tilde{x} = 2.000$	$c = 199$	$\tilde{x} = 2.000$	$s = 11$
10^{-14}	$\tilde{x} = 2.000$	$n = 8$	$\tilde{x} = 2.000$	$c = 217$	$\tilde{x} = 2.000$	$s = 11$
10^{-15}	$\tilde{x} = 2.000$	$n = 8$	$\tilde{x} = 2.000$	$c = 233$	$\tilde{x} = 2.000$	$s = 11$
10^{-16}	$\tilde{x} = 2.000$	$n = 8$	$\tilde{x} = 2.000$	$c = 243$	$\tilde{x} = 2.000$	$s = 11$

Dalla tabella si nota che il metodo di Newton effettua meno iterazioni rispetto ai metodi delle Corde e delle Secanti, poiche converge quadraticamente alla radice, a discapito di un elevato costo computazionale (ad ogni passo della iterazione si deve valutare la derivata prima). Il metodo delle Corde è piu efficiente rispetto al metodo di Newton dal punto di vista dei costi computazionali, poiche non richiede la valutazione della derivata prima ad ogni iterazione, a discapito di un ordine di convergenza minore (converge solo linearmente). Infine il metodo delle Secanti ha lo stesso costo computazionale del metodo delle corde, pero ha un ordine di convergenza maggiore di quest'ultimo (circa pari a 1.618 per radici semplici). Nel caso in cui si utilizzi $x0 = \frac{(}{3}5)$ come punto di innesco si nota che calcolando la molteplicità di $P(x)4$ questa risulta zero, dunque non si avrebbe convergenza.

2.3

Costruire una seconda tabella analoga alla precedente relativa ai metodi di Newton, di Newton modificato e di accelerazione di Aitken applicati alla funzione polinomiale P a partire dal punto di innesco $x_0 = 0$. Commentare i risultati riportati in tabella.

Abbiamo visto come il polinomio $P(x) = x^3 - 4x^2 + 5x - 2$, in $P(x) = 0$ presenta due radici, una con molteplicità multipla $x = 1$ e una con molteplicità semplice $x = 2$.

Di seguito sono riportati tre codici Matlab, rispettivamente:

- **Metodo di Newton**

```
function [y,i] = newtonSolve(x0,itmax,fx,f1x,tol)
    %Funzione che ritorna l'approssimazione della radice
    %della funzione fx data in input usando il metodo di
       Newton
    %INPUT:
    %x0 = punto iniziale dell'iterazione
    %itmax = numero di iterazioni massime
    %fx = funzione data in input
    %f1x = derivata della funzione
    %tol = tolleranza usata per accettare l'
       approssimazione
    %OUTPUT:
    %y = risultato approssimazione
    %i = numero di iterazioni compiute

    tempFx = feval(fx,x0);
    tempF1x = feval(f1x,x0);
    y = x0-(tempFx/tempF1x);
    i = 1;
    while (i < itmax) && (abs(y-x0) > tol)
        i = i+1;
        x0 = y;
        tempFx = feval(fx,x0);
        tempF1x = feval(f1x,x0);
        y = x0-(tempFx/tempF1x);
    end
    if abs(y-x0) > tol
```

```
        warning('La funzione non converge alla radice');
    end
end
```

- **Metodo di Newton modificato**

```
function [y,i] = newtonMod(x0,itmax,fx,f1x,m,tol)
    %Funzione che ritorna l'approssimazione della radice
    %della funzione fx data in input usando il metodo di
       Newton modificato
    %INPUT:
    %x0 = punto iniziale dell'iterazione
    %itmax = numero di iterazioni massime
    %fx = funzione data in input
    %f1x = derivata della funzione
    %tol = tolleranza usata per accettare l'
       approssimazione
    %m = molteplicita della radice
    %OUTPUT:
    %y = risultato approssimazione
    %i = numero di iterazioni compiute

    tempFx = feval(fx,x0);
    tempF1x = feval(f1x,x0);
    y = x0-m*(tempFx/tempF1x);
    i = 1;
    while (i < itmax) && (abs(y-x0) > tol)
        i = i+1;
        x0 = y;
        tempFx = feval(fx,x0);
        tempF1x = feval(f1x,x0);
        y = x0-m*(tempFx/tempF1x);
    end
    if abs(y-x0) > tol
        warning('La funzione non converge alla radice');
    end
end
```

- **Metodo di Aitken**

```
function [y,i] = Aitken(x0,fx,f1x,itmax,tol)
    %Funzione che ritorna l'approssimazione della radice
    %della funzione fx data in input usando il metodo di
       accelerazione
    %di Aitken
    %INPUT:
    %x0 = punto iniziale dell'iterazione
    %itmax = numero di iterazioni massime
    %fx = funzione data in input
    %f1x = derivata della funzione
    %tol = tolleranza usata per accettare l'
       approssimazione
    %OUTPUT:
    %y = risultato approssimazione
    %i = numero di iterazioni compiute
    i = 0;
    y = x0;
    flag = 1;
    while (i<itmax) && flag
        i = i+1;
        x0 = y;
        tempFx = feval(fx,x0);
        tempF1x = feval(f1x,x0);
        x1 = x0-(tempFx/tempF1x);
        tempFx = feval(fx,x1);
        tempF1x = feval(f1x,x1);
        y = x1-(tempFx/tempF1x);
        if(y-(2*x1)+x0) == 0
            flag = 0;
            i = i-1;
            break;
        end
        y = ((y*x0)-x1^2)/((y-(2*x1))+x0);
        flag = abs(y-x0)>tol;
    end
    if flag
        warning('Tolleranza non raggiunta');
    end
```

```
37 end
```

I seguenti tre seguenti script Matlab servono per eseguire i precedenti metodi. Lo script per eseguire il metodo di *Newton* con molteplicità $m = 1$ coincide con la versione normale dello stesso. Tutti gli script richiamano le rispettive funzioni con una approssimazione iniziale pari a $x = 0$, un numero massimo di iterazioni pari a $itmax = 1000$ ed una tol_x iniziale pari a 10^-1, la quale viene decrementata di un fattore 10 ad ogni iterazione. Al metodo di *Newton modificato* viene inoltre passato come parametro il valore della molteplicità pari a $m = 2$.

- **Script metodo di Newton**

```
1  %script esecuzione newton
2  tolx = 10^-1;
3  tol = 1e-1;
4  fx = @(x) x^3-4*x^2+5*x-2;
5  f1x = @(x) 3*x^2-8*x+5;
6  x0 = 0;
7  itmax = 1000;
8  rstApprox = [];
9  numIt = [];
10 index = 1;
11 while tol>eps
12         [rstApprox(index),numIt(index)] = newtonSolve(x0,
               itmax,fx,f1x,tol);
13     index = index+1;
14     tol = tol/10;
15 end
```

- **Script metodo di Newton modificato**

```
1  %soluzione esercizio3, capitolo 2 Newton Modificato
2  fx = @(x) x^3-4*x^2+5*x-2;
3  f1x = @(x) 3*x^2-8*x+5;
4  tol = 1e-1;
5  m = 2;
6  rstApprox = [];
7  numIt = [];
```

```
index = 1;
itmax = 2000;
x0=0;
while tol>eps
    [rstApprox(index),numIt(index)] = newtonMod(x0,itmax,
       fx,f1x,m,tol);
    index = index+1;
    tol = tol/10;
end
```

- **Script metodo di Aitken**

```
%soluzione esercizio3, capitolo 2 Aitken
fx = @(x) x^3-4*x^2+5*x-2;
f1x = @(x) 3*x^2-8*x+5;
tol = 1e-1;
rstApprox = [];
numIt = [];
index = 1;
itmax = 2000;
x0=0;
while tol>eps
    [rstApprox(index),numIt(index)] = Aitken(x0,fx,f1x,
       itmax,tol)
    index = index+1;
    tol = tol/10;
end
```

Dalla successiva tabella si nota che il metodo di *Aitken* e di *Newton* modificato con molteplicità $m = 2$ richiedono meno passi di iterazione e convergono piu velocemente rispetto ai metodi di *Newton* e di *Newton* modificato con $m = 1$.

tol_x	*Newton*		*NewtonMod* $m = 2$		*Aitken*	
10^{-1}	$\tilde{x} = 0.89598571514$	$in = 4$	$\tilde{x} = 0.99988432620$	$inm_2 = 3$	$\tilde{x} = 1.001987314688$	$ia = 2$
10^{-2}	$\tilde{x} = 0.99289408417$	$in = 8$	$\tilde{x} = 0.99999999331$	$inm_2 = 4$	$\tilde{x} = 1.000000989315$	$ia = 3$
10^{-3}	$\tilde{x} = 0.99910626211$	$in = 11$	$\tilde{x} = 0.99999999331$	$inm_2 = 4$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-4}	$\tilde{x} = 0.99994409459$	$in = 15$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-5}	$\tilde{x} = 0.99999301150$	$in = 18$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-6}	$\tilde{x} = 0.99999912650$	$in = 21$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-7}	$\tilde{x} = 0.99999994598$	$in = 25$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-8}	$\tilde{x} = 1.00000000137$	$in = 29$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-9}	$\tilde{x} = 1.00000000137$	$in = 29$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-10}	$\tilde{x} = 1.00000000137$	$in = 29$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-11}	$\tilde{x} = 1.00000000137$	$in = 29$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-12}	$\tilde{x} = 1.00000000137$	$in = 29$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-13}	$\tilde{x} = 1.00000000137$	$in = 29$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-14}	$\tilde{x} = 1.00000000137$	$in = 29$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$
10^{-15}	$\tilde{x} = 1.00000000137$	$in = 29$	$\tilde{x} = 0.99999999331$	$inm_2 = 5$	$\tilde{x} = 0.999999998201$	$ia = 4$

2.4

Definire una procedura iterativa basata sul metodo di Newton per approssimare $\sqrt{\alpha}$, per un assegnato $\alpha > 0$. Costruire una tabella dove si riportano le successive approssimazioni ottenute e i corrispondenti errori assoluti (usare l'approssimazione Matlab di $\sqrt{\alpha}$ per il calcolo dell'errore) nel caso in cui $\sqrt{\alpha} = 5$ partendo da $x_0 = 5$.

Dato che $\sqrt{\alpha}$ è la radice ricercata, occorre quindi trovare una funzione $f(x)$ che abbia uno *zero* in appunto $\sqrt{\alpha}$. La funzione ricercata in questo caso è $f(x) = x^2 - \alpha$, la quale ha due radici semplici in $\pm\sqrt{\alpha}$ e ha derivata $f'(x) = 2x$.

L'iterazione del metodo di Newton utilizzando questa funzione diventa :

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - \alpha}{2x_i} =$$

$$= \frac{2x_i^2 - x_i^2 + \alpha}{2x_i} = \frac{x_i^2 + \alpha}{2x_i} =$$

$$= \frac{1}{2}\left(x_i + \frac{\alpha}{x_i}\right), \quad i = 0, 1, 2, ...$$

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

```
1  function [xi,erri] = newtonSolveEs4(alpha,x0,tol,itmax)
2      %Funzione che ritorna l'approssimazione della radice
3      %della funzione fx data in input usando il metodo di
           Newton
4      %INPUT:
5      %x0 = punto iniziale dell'iterazione
6      %itmax = numero iterazioni massime da effettuare
7      %fx = funzione data in input
8      %f1x = derivata della funzione
9      %tol = tolleranza usata per accettare l'approssimazione
10     %OUTPUT:
11     %xi = vettore contente le approssimazioni al passo i-
           esimo
12     %erri = vettore contente gli errori al passo i-esimo
```

```
    xi = [];
    erri = [];
    i = 2;
    xi(i-1) = x0;
    erri(i-1) = abs(x0-sqrt(x0));
    xi(i) = 0.5*(x0+(alpha/x0));
    erri(i) = abs(xi(i)-sqrt(x0));
    while i < itmax && abs(x0-xi(i)) > tol
        x0 = xi(i);
        i=i+1;
        xi(i) = 0.5*(x0+(alpha/x0));
        erri(i) = abs(xi(i)-sqrt(5));
    end
end
```

Il seguente codice MatLab, riguarda la chiamata della funzione definita precedentemente, con $\alpha = x_0 = 5$, con numero di passi massimi $itmax = 10$ e indice di tolleranza $tol_x = eps$:

```
err = [];
index = 1;
rstApprox = [];
x0 = 5;
itmax = 10;
alpha = 5;
[rstApprox,err] = newtonSolveEs4(alpha,x0,eps,itmax);
```

restituisce i seguenti valori:

i	x_i	$E_{ass} = \epsilon_i = \lvert x_i - \sqrt{\alpha}\rvert \quad \alpha = 5$
$i = 0$	$x_0 = 5$	$\lvert\epsilon_0\rvert = 2.763932022500210$
$i = 1$	$x_1 = 3$	$\lvert\epsilon_1\rvert = 0.763932022500210$
$i = 2$	$x_2 = 2.333333333333334$	$\lvert\epsilon_2\rvert = 0.097265355833544$
$i = 3$	$x_3 = 2.238095238095238$	$\lvert\epsilon_3\rvert = 0.002027260595448$
$i = 4$	$x_4 = 2.236068895643363$	$\lvert\epsilon_4\rvert = 9.181435736138610e-07$
$i = 5$	$x_5 = 2.236067977499978$	$\lvert\epsilon_5\rvert = 1.882938249764266e-13$
$i = 6$	$x_6 = 2.236067977499790$	$\lvert\epsilon_6\rvert = 0$
$i = 7$	$x_7 = 2.236067977499790$	$\lvert\epsilon_7\rvert = 0$

2.5

Definire una procedura iterativa basata sul metodo delle secanti sempre per approssimare $\sqrt{\alpha}$, per un assegnato $\alpha > 0$. Completare la tabella precedente aggiungendovi i risultati ottenuti con tale procedura partendo da $x_0 = 5$ e $x_1 = 3$. Commentare i risultati riportati in tabella.

Come visto nel precedente esercizio occorre utilizzare la funzione $f(x) = x^2 - \alpha$
L'iterazione del metodo delle Secanti utilizzando questa funzione diventa:

$$x_{i+1} = \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})} =$$

$$= \frac{(x_i^2 - \alpha)x_{i-1} - (x_{i-1}^2 - \alpha)x_i}{x_i^2 - \alpha - x_{i-1}^2 + \alpha} =$$

$$= \frac{x_i^2 x_{i-1} - \alpha x_{i-1} - x_{i-1}^2 x_i + \alpha x_i}{x_i^2 - x_{i-1}^2} =$$

$$= \frac{x_i x_{i-1}(x_i - x_{i-1}) + \alpha(x_i - x_{i-1})}{(x_i - x_{i-1})(x_i + x_{i-1})} =$$

$$= \frac{(x_i - x_{i-1})(x_i x_{i-1} + \alpha)}{(x_i - x_{i-1})(x_i + x_{i-1})} =$$

$$= \frac{x_i x_{i-1} + \alpha}{x_i + x_{i-1}}, \quad i = 0, 1, 2, ...$$

L'implementazione del metodo delle secanti in Matlab è la seguente:

```
1  function [xi,erri] = secantiSolveEs5(alpha,x0,x1,itmax,tol)
2      %Funzione che ritorna l'approssimazione della radice
3      %della funzione fx data in input usando il metodo delle
           Secanti
4      %INPUT:
5      %x0 = punto iniziale dell'iterazione
6      %itmax = numero di iterazioni massime
7      %fx = funzione data in input
8      %f1x = derivata della funzione
9      %tol = tolleranza usata per accettare l'approssimazione
10     %OUTPUT:
11     %y = risultato approssimazione
12     %i = numero di iterazioni compiute
```

```
    xi = [];
    erri = [];
    i=1;
    xi(i) = x0;
    erri(i) = abs(x0-sqrt(alpha));
    i=i+1;
    xi(i) = x1;
    erri(i) = abs(x1-sqrt(alpha));
    while(i<itmax) && (abs(x1-x0) > tol)
        i = i+1;
        xi(i) = ((x1*x0)+alpha)/(x1+x0);
        erri(i) = abs(xi(i)-sqrt(5));
        x0 = xi(i-1);
        x1 = xi(i);
    end
end
```

Il seguente codice MatLab, riguarda la chiamata della funzione definita precedentemente, con $\alpha = x_0 = 5$, con $x_1 = 3$, con numero di passi massimi $imax = 100$ e indice di tolleranza $tol_x = eps$:

```
fx = @(x) x^2-5;
err = [];
rstApprox = [];
x0 = 5;
x1 = 3;
alpha = 5;
[rstApprox,err] = secantiSolveEs5(alpha,x0,x1,itmax,eps);
```

restituisce i seguenti valori:

i	x_i	$E_{ass} = \epsilon_i = \lvert x_i - \sqrt{\alpha}\rvert \quad \alpha = 5$
$i = 0$	$x_0 = 5$	$\lvert\epsilon_0\rvert = 2.763932022500210$
$i = 1$	$x_1 = 3$	$\lvert\epsilon_1\rvert = 0.763932022500210$
$i = 2$	$x_2 = 2.500000000000000$	$\lvert\epsilon_2\rvert = 0.263932022500210$
$i = 3$	$x_3 = 2.272727272727273$	$\lvert\epsilon_3\rvert = 0.36659295227483$
$i = 4$	$x_4 = 2.238095238095238$	$\lvert\epsilon_4\rvert = 0.00202760595448$
$i = 5$	$x_5 = 2.236084452975048$	$\lvert\epsilon_5\rvert = 1.647547525829296e - 05$
$i = 6$	$x_6 = 2.236067984964863$	$\lvert\epsilon_6\rvert = 7.465073448287285e - 09$
$i = 7$	$x_7 = 2.236067977499817$	$\lvert\epsilon_7\rvert = 2.753353101070388e - 14$
$i = 8$	$x_8 = 2.236067977499790$	$\lvert\epsilon_8\rvert = 4.440892098500626e - 16$
$i = 9$	$x_9 = 2.236067977499790$	$\lvert\epsilon_9\rvert = 0$
$i = 10$	$x_{10} = 2.236067977499790$	$\lvert\epsilon_{10}\rvert = 0$

3 Capitolo

3.1

Scrivere una function Matlab per la risoluzione di un sistema lineare con matrice dei coefficienti triangolare inferiore a diagonale unitaria. Inserire un esempio di utilizzo.

Soluzione

Il seguente codice riguarda la risoluzione di un sistema triangolare inferiore a diagonale unitaria:

```
1  function x = trisolveInf(A,b)
2      %x = trisolveInf(A,b);
3      %La funzione restituisce la soluzione del sistema lineare
           Ax = b
4      %INPUT:
5      %A = matrice triangolare inferiore a diagonale unitaria
6      %b = vettore termini noti
7      %OUTPUT:
8      %x = vettore soluzione
9      x = b;
10     [m,n] = size(A);
11     if m~=n
12         error('Matrice non quadrata');
13     end
14     for i=1:n
15         if A(i,i) ~= 1
16             error(Matrice non a diagonale unitaria);
17         end
18     end
19     for i = 1:n
20         for j = 1:i-1
21             x(i) = x(i)-A(i,j)*(x(j));
22         end
23         x(i) = x(i)/A(i,i);
24     end
25 end
```

Un esempio di utilizzo è dato dalla seguente matrice A, e vettore dei termini noti b:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

E si ottiene il seguente vettore soluzione: $x = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

3.2

Utilizzare l'Algoritmo 3.6 del libro per stabilire se le seguenti matrici sono sdp o no,

$$A_1 = \begin{bmatrix} 1 & -1 & 2 & 2 \\ -1 & 5 & -14 & 2 \\ 2 & -14 & 42 & 2 \\ 2 & 2 & 2 & 65 \end{bmatrix}, A_2 = \begin{bmatrix} 1 & -1 & 2 & 2 \\ -1 & 6 & -17 & 3 \\ 2 & -17 & 48 & -16 \\ 2 & 3 & -16 & 4 \end{bmatrix}$$

Soluzione

L'algoritmo utilizzato per risolvere l'esercizio è quello della fattorizzazione LDL^T. Questo perchè grazie al Teorema 3.6 del libro, è noto che una matrice è sdp se e solo se questa è fattorizzabile LDL^T.
Si riporta dunque il codice Matlab di tale fattorizzazione:

```
1  function A = LDLTFatt(A)
2  %y = LDLTFatt(A)
3  %Funzione che restituisce la matrice A fattorizzata con il
       metodo LDLT
4  %dove L matrice triangolare inferiore a diagonale unitaria, D
       matrice diagonale,
5  %ed LT matrice trasposta di L
6  %INPUT:
7  %A = matrice sdp da fattorizzare
8  %OUTPUT:
9  %A = matrice A fattorizzata LDLT
10     n = size(A,1);
11     if A(1,1) <= 0
12         error('Matrice non sdp');
13     end
14     A(2:n,1) = A(2:n,1)/A(1,1);
15     for j = 2:n
16         v = (A(j,1:j-1).').* diag(A(1:j-1,1:j-1));
17         A(j,j) = A(j,j) - A(j,1:j-1)*v;
18         if A(j,j) <= 0
19             error ('Matrice non sdp');
20         end
21         A(j+1:n,j) = (A(j+1:n,j)-A(j+1:n,1:j-1)*v)/A(j,j);
22     end
```

```
23 end
```

• A_1

Si richiama il precedente Algoritmo con la matrice A_1 come input. Si ottiene il seguente risultato:

$$A_1 = \begin{bmatrix} 1 & -1 & 2 & 2 \\ -1 & 4 & -14 & 2 \\ 2 & -3 & 2 & 2 \\ 2 & 1 & 5 & 7 \end{bmatrix}$$

• A_2

Si richiama il precedente Algoritmo con la matrice A_2 come input. In questo caso però la matrice non risulta essere fattorizzabile LDL^T e dunque l'esecuzione stamperà il seguente errore:

Error using LDLTFatt (line 19)
Matrice non sdp

3.3

Scrivere una function Matlab che, avendo in ingresso un vettore **b** contenente i termini noti del sistema lineare $\mathbf{Ax} = \mathbf{b}$ con A sdp e l'output dell'Algoritmo 3.6 del libro (matrice A riscritta nella porzione triangolare inferiore con i fattori L e D della fattorizzazione LDL^t di A), ne calcoli efficientemente la soluzione.

Soluzione

Si vuole quindi risolvere il sistema Ax = b con A = LDL^t. Il seguente codice risolve tale sistema lineare efficientemente:

```
1  function x = solveLDLT(A,b)
2      %x = solveLDLT(A,b)
3      %Restituisce il vettore soluzione del sistema lineare Ax
          = b
4      %fattorizzando la matrice A = LDLT
5      %Per la soluzione finale ci riconduciamo a risolvere i
          seguenti sistemi
6      %lineari: 1) Ly = b 2) Dz = y 3) LTx = z
7      %INPUT:
8      %A = matrice sdp
9      %b = vettore termini noti
10     %OUTPUT:
11     %x = vettore soluzione
12     A = LDLTFatt(A);
13     x = trisolveInf(tril(A,-1)+eye(length(A)),b); %
          restituisce la soluzione della matrice triangolare
          inferiore (tril(A,-1)) a diagonale unitaria eye(length(
          A)
14     x = x ./ (diag(A))';
15     x = trisolveSup((tril(A,-1)+eye(length(A)))',x);
16 end
```

Si fattorizza la matrice A di modo che questa venga riscritta nella porzione triangolare inferiore con i fattori L e D della fattorizzazione LDL^t. Dopo di che per risolvere il sistema Ax = b, ci riconduciamo a risolvere i seguenti sistemi lineari:

- $Ly = b$

Si impiega l'algoritmo di risoluzione per matrici triangolari inferiori

precedentemente utilizzato. Vale la pena notare che per estrarre la parte triangolare inferiore a diagonale unitaria si utilizza la funzione tril((A,-1) che restituisce sotto matrice strettamente inferiore rispetto alla diagonale e la si concatena con una matrice identità che permette cosi di ricostruire una matrice triangolare inferiore a diagonale unitaria.

•$Dz = y$

Si calcola la soluzione del sistema dividendo ogni elemento di x (soluzione del precedente sistema) con il vettore riga contenente gli elementi della diagonale di A

•$L^t = z$

Infine per ottenere la soluzione finale si ricava dalla matrice A la sotto matrice triangolare inferiore a diagonale unitaria con la stessa tecnica precedentemente utilizzata, calcolandone però la trasposta di modo da ricavare la sotto matrice triangolare superiore. Si invoca il seguente algoritmo per la sua risoluzione:

```
1  function x = trisolveSup(A,b)
2  %La funzione restituisce la soluzione del sistema lineare Ax
     = b
3  %INPUT:
4  %A = matrice triangolare superiore a diagonale unitaria
5  %b = vettore termini noti
6  %OUTPUT:
7  %x = vettore soluzione
8      x = b;
9      [m,n] = size(A);
10     if m~=n
11         error('Matrice non quadrata');
12     end
13     if A(n,n) == 0
14         error('Matrix not upper triangular');
15     else
16         x(n) = x(n) /A(n,n);
17     end
18     for i = n-1:-1:1
19         for j = n:-1:i+1
20             x(i) = x(i) - A(i,j)*x(j);
21         end
```

```
22          x(i) = x(i)/A(i,i);
23      end
24  end
```

3.4

Scrivere una function Matlab che, avendo in ingresso un vettore **b** contenente i termini noti del sistema lineare $\mathbf{Ax} = \mathbf{b}$ con A sdp e l'output dell'Algoritmo 3.7 del libro (matrice A riscritta con la fattorizzazione LU con pivoting parziale e il vettore **p** delle permutazioni), ne calcoli efficientemente la soluzione.

Soluzione

Con il seguente codice Matlab si ricava la fattorizzazione A = LU con pivoting parziale

```
function [A,p] = LUPivoting(A)
    %[A,p] = LUPivoting(A)
    %Restituisce la matrice A fattorizzata LU con pivoting
       parziale ed il
    %vettore p delle permutazioni
    %INPUT:
    %A = matrice sdp
    [m,n] = size(A);
    if m~=n
        error('Matrice non quadarata');
    end
    p = 1:n;
    for i = 1:n-1
        [mi,ki] = max(abs(A(i:n,i)));
        if mi == 0
            error('Matrice non singolare');
        end
        ki = ki+i-1;
        if ki> i
            A([i ki],:) = A([ki i],:);
            p([i ki]) = p([ki i]);
        end
        A(i+1:n,i) = A(i+1:n,i)/A(i,i);
        A(i+1:n,i+1:n) = A(i+1:n,i+1:n) -A(i+1:n,i)*A(i,i+1:n
           );
    end
end
```

Si vuole risolvere il sistema Ax = b con A = LU. Per farlo utilizziamo il codice:

```
function x = LUPivotingSolve(A,b)
    %x = LUPivotingSolve(A,b)
    %Calcola il vettore soluzione del sistema lineare Ax = b
       con matrice
    %fattorizzata LU con pivoting.
    %INPUT:
    %A = matrice
    %b = vettore termini noti
    %OUTPUT:
    %x = vettore soluzione
    [A,p] = LUPivoting(A);
    x = trisolveInf(tril(A,-1)+eye(length(A)),b);
    x = trisolveSup(triu(A),x);
end
```

La soluzione sarà data quindi dalla risoluzione dei due sistemi lineari:

•$Ly = b$

Si ricava dalla matrice A la sotto matrice triangolare inferiore a diagonale unitaria e si risolve tale sistema per mezzo dell'algoritmo di risoluzione **trisolveInf(A,b)**

•$Ux = y$

Si ricava dalla matrice A la sotto matrice triangolare superiore e si risolve tale sistema per mezzo dell'algoritmo di risoluzione **trisolveSup(A,b)**

3.5

Inserire alcuni esempi di utilizzo delle due function implementate per i punti 3 e 4, scegliendo per ciascuno di essi un vettore $\hat{x}$ e ponendo $\mathbf{b} = \mathbf{A}\hat{x}$. Riportare $\hat{x}$ e la soluzione $\mathbf{x}$ da essi prodotta. Costruire anche una tabella in cui, per ogni esempio considerato, si riportano il numero di condizionamento di A in norma 2 (usare cond di Matlab) e le quantità $\|r\|/\|b\|$ e $\|x-\hat{x}\|/\|\hat{x}\|$.

Soluzione

Si riporta in seguito il codice utilizzato per la risoluzione del problema. Questo contiene le chiamate delle funzioni precedentemente descritte nell'esercizio 3 e 4:

```
%Soluzione esercizio 5

%Esempio esercizio 3: LDLT
xc = ([64 8 13])';
A = [11 4 -2;6 7 -1;5 1 12];
b = A * xc;


x = solveLDLT(A,b);
condA = cond(A);
normaR = norm((A*x)-b);
normaRB = normaR/norm(b);
errRel = norm(x-xc)/norm(xc);

%Esempio esercizio 4: LU
xc1 = ([64 8 13])';
A1 = [11 4 -2;6 7 -1;5 1 12];
b1 = A1 * xc1;

x1 = LUPivotingSolve(A1,b1);
condA1 = cond(A1);
normaR1 = norm((A1*x1)-b1);
normaRB1 = normaR1/norm(b1);
errRel1 = norm(x1-xc1)/norm(xc1);
```

- Esempio esercizio 3 ($A = LDL^t$)

$$A = \begin{bmatrix} 11 & 4 & -2 \\ 6 & 7 & -1 \\ 5 & 1 & 12 \end{bmatrix}, \hat{x} = \begin{bmatrix} 64 \\ 8 \\ 13 \end{bmatrix}, A\hat{x} = b = \begin{bmatrix} 710 \\ 427 \\ 484 \end{bmatrix}$$

Dalla fattorizzazione A = LDL^t ottengo:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5455 & 1 & 0 \\ 0.4545 & -0.4634 & 1 \end{bmatrix}, D = \begin{bmatrix} 11 & 0 & 0 \\ 0 & 3.7273 & 0 \\ 0 & 0 & 8.9268 \end{bmatrix},$$

$$L^t = \begin{bmatrix} 1 & 0.5455 & 0.4545 \\ 0 & 1 & -0.4634 \\ 0 & 0 & 1 \end{bmatrix}$$

Dalla quale ricavo la soluzione:

$$x = \begin{bmatrix} 44.4945 \\ 19.9863 \\ 20.1284 \end{bmatrix}$$

• Esempio esercizio 4 (A = LU con pivoting)

$$A = \begin{bmatrix} 11 & 4 & -2 \\ 6 & 7 & -1 \\ 5 & 1 & 12 \end{bmatrix}, \hat{x} = \begin{bmatrix} 64 \\ 8 \\ 13 \end{bmatrix}, A\hat{x} = b = \begin{bmatrix} 710 \\ 427 \\ 484 \end{bmatrix}$$

Dalla fattorizzazione A = LU con pivoting, si ottiene:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5455 & 1 & 0 \\ 0.4545 & -0.1698 & 1 \end{bmatrix}, U = \begin{bmatrix} 11 & 4 & -2 \\ 0 & 4.8182 & 0.0909 \\ 0 & 0 & 12.9245 \end{bmatrix}$$

In questo caso non vengono effettuati scambi tra le righe, dunque il vettore b rimane invariato. Si ottiene quindi il risultato:

$$x = \begin{bmatrix} 64 \\ 8.000000000000004 \\ 13.000000000000004 \end{bmatrix}$$

Di seguito si riporta in una tabella il numero di condizionamento di A in norma 2 e le quantità $\|r\|/\|b\|$ e $\|x - \hat{x}\|/\|\hat{x}\|$.

A	$K_2(A)$	$\frac{\|r\|}{\|b\|}$	$\frac{\|x-\hat{x}\|}{\|\hat{x}\|}$
$A = LDL^t$	4.1947	0.1931	0.3644
$A = LU$	4.1947	5.9241e-17	7.6363e-17

3.6

Sia $A = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}$ con $\epsilon = 10^{-13}$. Definire L triangolare inferiore a diagonale unitaria e U triangolare superiore in modo che il prodotto LU sia la fattorizzazione LU di A e, posto b = Ae, con $e = (1,1)^T$, confrontare l'accuratezza della soluzione che si ottiene usando il comando $U\backslash(L\backslash b)$ (Gauss senza pivoting) e il comando $A\backslash b$(Gauss con pivoting),

Soluzione

Con il seguente codice si effettua la fattorizzazione A = LU:

```
function A = luFactorization(A)
    % A = luFactorization(A) restuisce la matrice A
       fattorizzata LU
    % INPUT:
    %A = matrice nxn nonsingolare
    %A = matrice A = LU
    m = size(A,1);
    n = size(A,2);
    if m ~= n
        error ('Matrice non quadrata');
    end
    for i = 1:n-1
        if A(i,i) == 0
            error('Matrice non fattorizzabile LU')
        end
        A(i+1:n,i) = A(i+1:n,i)/A(i,i);
        A(i+i:n,i+i:n) = A(i+1:n, i+1: n) - A(i+1:n,i)* A(i,i
           +1:n);
    end
    return
end
```

Si richiama inoltre il seguente codice Matlab per effettuare la chiamata alla funzione $luFactorization(A)$, ricavare quindi la sottomatrice triangolare inferiore a diagonale unitaria (L) e la sottomatrice triangolare superiore (U); si assegna a b la quantità Ae ed infine si confrontano i risultati usando prima Gauss senza pivoting e poi Gauss con pivoting.

```
%Soluzione es6 Capitolo 3
```

```
2
3  A = [10^-13 1; 1 1];
4  A = luFactorization(A);
5  L = tril(A,-1)+eye(length(A));
6  U = triu(A);
7  e = ([1 1])';
8  b = A*e;
9  gSp = U\(L\b); %Gauss senza pivoting
10 gP = A\b; %Gauss con pivoting
```

Dalla fattorizzazione si ottiene:

$$A = \begin{bmatrix} 1.0000e-13 & 1 \\ 1.0000e+13 & -1.0000e+13 \end{bmatrix}, L = \begin{bmatrix} 1 & 0 \\ 1.0000e+13 & 1 \end{bmatrix},$$
$$U = \begin{bmatrix} 1.0000e-13 & 1 \\ 0 & -1.0000e+13 \end{bmatrix}$$

Calcolando il vettore b = Ae si ottiene:

$$b = \begin{bmatrix} 1.000000000000100 \\ 1 \end{bmatrix}$$

Infine si ottengono i due vettori rispettivamente di Gauss senza pivoting e Gauss con pivoting:

$$gSp = \begin{bmatrix} 0 \\ 1.000000000000100 \end{bmatrix}, gp = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Dunque si nota come Gauss con pivoting sia piu accurato rispetto a Gauss senza pivoting.

3.7

Scrivere la function Matlab specifica per la risoluzione di un sistema lineare con matrice dei coefficienti $A \in R^{nxn}$ bidiagonale inferiore a diagonale unitaria di Toeplitz, specificabile con uno scalare α. Sperimentarne e commentarne le prestazioni (considerare il numero di condizionamento della matrice) nel caso in cui $n = 12$ e $\alpha = 100$ ponendo dapprima $b = (1, 101, ...101)^T$ (soluzione esatta $\hat{x} = (1, ..., 1)^T$) e quindi b = 0.1 * $(1, 101, ..., 101)^T$) (soluzione esatta $\hat{x} = (0.1, ..., 0.1)^T$).

Soluzione

Si riporta in seguito il codice utilizzato per la risoluzione di un sistema lineare Ax = b, con $A \in R^{nxn}$ bidiagonale inferiore a diagonale unitaria di Toeplitz, che ha la seguente forma:

$$A = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & \alpha & 1 \end{bmatrix}$$

```
1  function x = trisolveInfAlphaToeplitz(dim,b,alpha)
2      %x = trisolveInfAlphaToeplitz(dim,b,alpha);
3      %La funzione crea una matrice bidiagonale inferiore a
          diagonale unitaria
4      %di Toeplitz ed infine restituisce la soluzione del
          sistema lineare Ax = b
5      %INPUT:
6      %dim = dimensione della matrice
7      %b = vettore termini noti
8      %alpha = scalare sotto la diagonale
9      %OUTPUT:
10     %x = vettore soluzione
11
12     A = eye(dim);
13     %zeros(1,dim) crea un vettore riga di dim elementi; eye(
          dim-1) zeros(1,dim-1)'
14     %crea una matrice dim-1*dim-1 con un 1 sulla diagonale, e
           infine si concatena con la matrice dimxdim
15     B = alpha*[zeros(1,dim);eye(dim-1) zeros(1,dim-1)'];
```

```
    A = A+B;

    x = b;
    [m,n] = size(A);
    if m~=n
        error('Matrice non quadrata');
    end
    for i=1:n
        if A(i,i) ~= 1
            error(Matrice non a diagonale unitaria);
        end
    end
    for i = 1:n
        for j = 1:i-1
            x(i) = x(i)-A(i,j)*(x(j));
        end
        x(i) = x(i)/A(i,i);
    end
end
```

Il seguente codice invece riguarda la soluzione dell'esercizio:

```
%Soluzione esercizio7 cap3
b = ones(1,12)';
b(2:12) = b(2:12)*101;
b1 = b*0.1;

x = trisolveInfAlphaToeplitz(12,b,100);
x1 = trisolveInfAlphaToeplitz(12,b1,100);


dim = 12;
A = eye(dim);
B = 100*[zeros(1,dim);eye(dim-1) zeros(1,dim-1)']; %zeros(1,
   dim) crea un vettore riga di dim elementi; eye(dim-1) zeros
   (1,dim-1)' crea una matrice dim-1*dim-1 con un 1 sulla
   diagonale, e infine si concatena con la matrice dimxdim
A = A+B;
%Condizionamento matrice
condA = norm(A,Inf)* norm(inv(A),Inf);
```

```
16 %Condizionamento del risultato caso 2
17 x1esatto = (ones(1,12)*0.1)';
18 b1esatto = A*x1esatto;
19 errAbsB = abs(b1esatto-b1);
20 condB = norm(errAbsB,Inf)/norm(b1esatto,Inf);
21 errX = condA*condB;
```

Si ottengono quindi i due vettori soluzione:

$$x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, x1 = \begin{bmatrix} 1.0000000000000000e-01 \\ 1.0000000000000014e-01 \\ 9.999999999985931e-02 \\ 1.000000000140702e-01 \\ 9.999999859298470e-02 \\ 1.000001407015319e-01 \\ 9.998592984681665e-02 \\ 1.014070153183369e-01 \\ -4.070153183368319e-02 \\ 1.417015318336832e+01 \\ -1.406915318336832e+03 \\ 1.407016318336832e+05 \end{bmatrix}$$

Analizzando il condizionamento si nota che la matrice risulta essere mal condizionata, in quanto $condA = 1.0202e+24$ ovvero $k(A) >> 1$.
Ricordiamo che l'errore commesso sulla soluzione risulta:

$$\frac{\|\triangle x\|}{\|x\|} \leq \|A\| \cdot \|A^{-1}\|(\frac{\|\triangle b\|}{\|b\|} + \frac{\|\triangle A\|}{\|A\|}).$$

Considerando l'errore commesso sulla soluzione nel primo caso ($b = (1, 101, ...101)^T$) si ha che il vettore risultato coincide con il vettore esatto che viene fornito dal testo dell'esercizio. Di conseguenza l'errore commesso è pari a zero. Nel secondo caso invece (b = 0.1 * $(1, 101, ..., 101)^T$)) la soluzione non coincide con la soluzione esatta fornita dal testo; calcolando l'errore sul risultato, questo risulta essere amplificato dal mal condizionamento della matrice:

$$errX = 1.7943e+08$$

3.8

Scrivere una function che, dato un sistema lineare sovradeterminato Ax = b, con $A \in R^{m+n}$, m > n, rank(A) = n e $b \in R^m$, preso come input b e l'output dell'Algoritmo 3.8 del libro (matrice A riscritta con la parte significativa di R e la parte significativa dei vettori di Householder normalizzati con la prima componente unitaria), ne calcoli efficientemente la soluzione nel senso dei minimi quadrati.

Soluzione

Si riporta l'Algoritmo 3.8 del libro, relativo alla fattorizzazione QR di Householder:

```
function A = QRFatt(A)
    %A =QRFatt(A)
    %Calcola la fattorizzazione QR della matrice A
    %INPUT:
    %A = matrice da fattorizzare
    %OUTPUT:
    %A = matrice fattorizzata
    [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
        v1 = A(i,i)-alpha;
        A(i,i) = alpha;
        A(i+1:m,i) = A(i+1:m,i)/v1;
        beta = -v1/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
```

Mentre la function utilizzata per risolvere il sistema nel senso dei minimi quadrati è la seguente:

```
function x = SolveLeastSquares(A,b,m,n)
    %x = SolveLeastSquares(A,b)
    %Risolve il sistema sovradeterminato Ax = b nel senso dei
        minimi
    %quadrati
    %INPUT:
    %A = matrice fattorizzata QR
    %b = vettore termini noti
    %OUTPUT:
    %x = soluzione nel senso dei minimi quadrati, ovvero Rx =
        g1

    qT = eye(m);
    for i=1:n
        qT = [eye(i-1) zeros(i-1,m-i+1); zeros(i-1,m-i+1)' (
           eye(m-i+1)-(2/norm([1;A(i+1:m,i)],2)^2)*([1; A(i+1:
           m,i)]*[1 A(i+1:m, i)']))]*qT;
    end
    R = triu(A(1:n, :));
    qTb = qT(1:n,:)*b';
    x = trisolveSup(R,qTb);
end
```

Si vuole quindi calcolare la soluzione del sistema lineare $\hat{R}$ x = b .Per fare ciò si ricostruire la matrice Q^t a partire dalla matrice QR riscritta sui vettori di Householder. Quindi si ricava la matrice $\hat{R}$ per mezzo della funzione *triu* che ci restituisce la sotto matrice triangolare superiore di A, ed infine si ricava il vettore g1, moltiplicando le prime n componenti di Q^t per il vettore colonna b. Si richiama dunque la funzione per la risoluzione della matrice triangolare superiore con parametri R (cioè $\hat{R}$) e qTb (cioè g1).

3.9

Inserire due esempi di utilizzo della function implementata per il punto 8 e confrontare la soluzione ottenuta con quella fornita dal comando A\b

Soluzione

Nel seguente codice si mostrano due esempi per l'esercizio 8:

```
%Soluzione esercizio 9 cap3
% esempio 1
A1 = [3 2 1; 1 2 3; 1 2 1; 2 1 2];
b1 = [10 10 10 10];
[m1,n1] = size(A1);
x1 = A1\b1';
A1 = QRFatt(A1);
rst1 = SolveLeastSquares(A1,b1,m1,n1);

% esempio 2
A2 = [1 1 1; 1 2 4; 1 −1 1; 1 −2 4];
b2 = [1 1 1 2];
[m2,n2] = size(A2);
x2 = A2\b2';
A2 = QRFatt(A2);
rst2 = SolveLeastSquares(A2,b2,m2,n2);
```

•ESEMPIO 1

$$A1 = \begin{bmatrix} 3 & 2 & 1 \\ 1 & 2 & 3 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}, b1 = \begin{bmatrix} 10 \\ 10 \\ 10 \\ 10 \end{bmatrix}, x1 = A1\backslash b1 = \begin{bmatrix} 1.000000000000002e+00 \\ 2.800000000000000e+00 \\ 1.399999999999998e+00 \end{bmatrix}$$

$$rst1 = \begin{bmatrix} 1.399999999999999e+00 \\ 2.800000000000000e+00 \\ 1.400000000000001e+00 \end{bmatrix}$$

•ESEMPIO 2

$$A2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \end{bmatrix}, b2 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 2 \end{bmatrix}, rst2 = \begin{bmatrix} 0.8333333333333331e-01 \\ -0.2000000000000001e-01 \\ 1.666666666666668e-01 \end{bmatrix}$$

$$x2 = A2\backslash b2 = \begin{bmatrix} 0.8333333333333335e-01 \\ -0.2000000000000000e-01 \\ 1.6666666666666666e-01 \end{bmatrix}$$

3.10

Scrivere una function che realizza il metodo di Newton per un sistema non lineare (prevedere un numero massimo di iterazioni e utilizzare il criterio di arresto basato sull'incremento in norma euclidea). Utilizzare la function costruita al punto 4 per la risoluzione del sistema lineare ad ogni iterazione.

Soluzione

Per la risoluzione dei sistemi non lineari, ovvero formato da equazioni del tipo

$$F(y) = 0, F : \Omega \subseteq R^n \to R^n$$

si utilizza il metodo di Newton, cioè un metodo *iterarivo* definito da

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

partendo da un'approssimazione x^0 assegnata. Inoltre $J_F(x)$ rappresenta la matrice Jacobiana, formata dalle derivate parziali:

$$J_F(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) & \dots & \frac{\partial f_1}{\partial x_n}(x) \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1}(x) & \dots & \frac{\partial f_n}{\partial x_n}(x) \end{bmatrix}$$

Di seguito si riporta il codice relativo alla soluzione dei sistemi non lineari per mezzo del metodo iterativo di Newton

```
1  function [x,norma,j] = solveNewtonNonLineare(F,x,J,tol,itmax)
2      %[x,norma] = solveNewtonNonLineare(F,x,J,tol,imax)
3      %Risolve il sistema non lineare utilizzando il metodo di
           Newton
4      %INPUT:
5      %F= sistema non lineare
6      %x = punto/i iniziali
7      %J = matrice Jacobiana
8      %tol = tolleranza sull'errore
9      %itmax = iterazioni massime
10     %OUTPUT:
11     %x = soluzione del sistema non lineare F
12     %norma = norma dell'ultimo incremento
13     j = 0;
14     xPrev = 0;
```

```
15      n = length(x);
16      while(j<itmax) && (norm(x-xPrev)>tol)
17          j=j+1;
18          xPrev = x;
19          evaluation = -feval(F,x);
20          b = evaluation(1:n)';
21          x = x+LUPivotingSolve(J,b);
22      end
23      norma = norm(x-xPrev);
24  end
```

Ci riconduciamo quindi a risolvere il sistema lineare formato da 2 equazioni:
•$J_F(x^k)d^k = -F(x^k)$
•$x^{k+1} = x^k + d^k$

Pertanto la risoluzione di tale sistema lineare si riconduce alla risoluzione di una successione di sistemi lineare, dove ad ogni passo si necessita la fattorizzazione della matrice Jacobiana per mezzo dell'algoritmo di fattorizzazione LU con pivoting.

3.11

Verificato che la funzione $f(x_1, x_2) = x_1^2 + x_2^3 - x_1 x_2$ ha un punto di minimo relativo in $(1/12,\ 1/6)$, costruire una tabella in cui si riportano il numero di iterazioni eseguite, e la norma euclidea dell'ultimo incremento e quella dell'errore con cui viene approssimato il risultato esatto utilizzando la function sviluppata al punto precedente per valori delle tolleranze pari a 10^{-t}, con $t = 3, 6$. Utilizzare $(1/2,\ 1/2)$ come punto di innesco. Verificare che la norma dell'errore è molto più piccola di quella dell'incremento (come mai?)

Soluzione

Ricapitolando abbiamo che:

- $F(x) = 0, \quad F = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 - x_2 \\ 3x_2^2 - 1 \end{bmatrix} = f$
- $x_1 = \frac{1}{2}$
- $x_2 = \frac{1}{2}$
- $J_F = \begin{bmatrix} 2 - x_2 & 2x_1 - 1 \\ 3x_2^2 - 1 & 6x_2 - x_1 \end{bmatrix}$
- $itmax = 1000$
- $tol = 10^{-t}, t = [3, 6]$

Si richiama quindi il seguente codice Matlab:

```
function [x,norma,j] = solveNewtonNonLineare(F,x,J,tol,itmax)
    %[x,norma] = solveNewtonNonLineare(F,x,J,tol,imax)
    %Risolve il sistema non lineare utilizzando il metodo di
       Newton
    %INPUT:
    %F= sistema non lineare
    %x = punto/i iniziali
    %J = matrice Jacobiana
    %tol = tolleranza sull'errore
    %itmax = iterazioni massime
    %OUTPUT:
    %x = soluzione del sistema non lineare F
    %norma = norma dell'ultimo incremento
    j = 0;
    xPrev = 0;
```

```
15      n = length(x);
16      while(j<itmax) && (norm(x-xPrev)>tol)
17          j=j+1;
18          xPrev = x;
19          evaluation = -feval(F,x);
20          b = evaluation(1:n)';
21          x = x+LUPivotingSolve(J,b);
22      end
23      norma = norm(x-xPrev);
24  end
```

Il codice produce i seguenti risultati

$tol = 10^{-t}$	it	$\|norma\|$	$\|err\|$
$tol = 10^{-3}$	17	8.543066834330485e-04	0.003794501517081
$tol = 10^{-6}$	51	9.788751414570120e-07	4.480819013409465e-06

La norma dell'ultimo incremento è molto minore della norma dell'errore sull'approssimazione del risultato. Questo avviene grazie all'ordine di convergenza del metodo di Newton per sistemi non lineari (che è 2, infatti il metodo ha convergenza quadratica), che consente all'approssimazione del risultato di convergere rapidamente verso la soluzione esatta.

4 Capitolo 4

4.1

Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di grado n in forma di Lagrange.
La forma della function deve essere del tipo: `y = lagrange( xi, fi, x)`

Il seguente codice Matlab implementa la function richiesta.

```
1  function y = lagrange(xi,fi,x)
2      %y = lagrange(xi,fi,x) calcola il polinomio interpolante
          la coppia di
3      %dati ascissa-ordinata.
4      %INPUT:
5      %xi = ascisse
6      %fi = ordinate
7      %x = vettore contenente i valori della x nel quale voglio
           calcolare il
8      %polinomio
9      %OUTPUT:
10     %y = vettore contente i valori del polinomio interpolante
           le coppie
11     %dei dati
12     n = length(xi);
13     L = ones(n,length(x));
14     for i = 1:n
15         for j = 1:n
16             if(i~=j)
17                 L(i,:) = L(i,:).*(x-xi(j))/(xi(i)-xi(j));
18             end
19         end
20     end
21     y = 0;
22     for i =1:n
23         y = y +fi(i)*L(i,:);
24     end
25 end
```

4.2

Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di grado n in forma di Newton.
La forma della function deve essere del tipo: `y = newton( xi, fi, x)`

Il seguente codice Matlab implementa la function richiesta.

```
function y = newton(xi,fi,x)
    % y = newton(xi,fi,x)
    % Calcola il polinomio interpolante le coppie di dati (xi
       ,fi)
    % sui punti del vettore x.
    % INPUT:
    % xi = vettore contenente le ascisse di interpolazione su
        cui calcolare la differenza divisa
    % fi = vettore contenente i valori assunti dalla funzione
        nei corrispondenti punti xi
    % x = vettore di punti sui quali valutare il polinomio
       interpolante
    % OUTPUT:
    % y = vettore contenente il valore del polinomio
       interpolante

    n = length(xi)-1; %grado del polinomio interpolante
    for j = 1:n
        for i = n+1:-1:j+1
            fi(i) = (fi(i) - fi(i-1))/(xi(i) - xi(i-j));
        end
    end
    y = fi(n+1)*ones(size(x));
    for i = n:-1:1
        y = (y.*(x-xi(i))+fi(i));
    end
end
```

4.3

Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di Hermite.
La forma della function deve essere del tipo: `y = hermite( xi, fi, f1i, x)`

Il seguente codice Matlab implementa la function richiesta.

```
function y = hermite(xi,fi,f1i,x)
    %y = hermite(xi,fi,f1i,x) calcola il polinomio interpolante le coppie
    %di dati ascissa ordinata per mezzo del polinomio di Hermite
    %INPUT:
    %xi = ascisse
    %fi = ordinate
    %f1i = derivata prima
    %x = valori della x
        %OUTPUT:
    %y = vettore contenente il valore del polinomio interpolante calcolato
    % sulle x.

    n = length(xi)-1;
    xH = zeros(2*n+2,1); %ascisse interpolazione di Hermite, sono 2n+2
    xH(1:2:2*n+1) = xi;
    xH(2:2:2*n+2) = xi;
    fH = zeros(2*n+2,1); %vettore delle 2n+2 differenze divise
    fH(1:2:2*n+1) = fi;
    fH(2:2:2*n+2) = f1i;
    pHermGrade = length(xH)-1;
    for i = pHermGrade:-2:3
        fH(i) = (fH(i)-fH(i-2))/(xH(i)-xH(i-1));
    end
    for i = 2:pHermGrade
        for j = pHermGrade+1:-1:i+1
            fH(j) = (fH(j)-fH(j-1))/(xH(j)-xH(j-i));
```

```
        end
    end
    %Algoritmo di horner
    y = fH(pHermGrade+1)*ones(size(x));
    for i=pHermGrade:-1:1
        y = y.*(x-xH(i))+fH(i);
    end
end
```

4.4

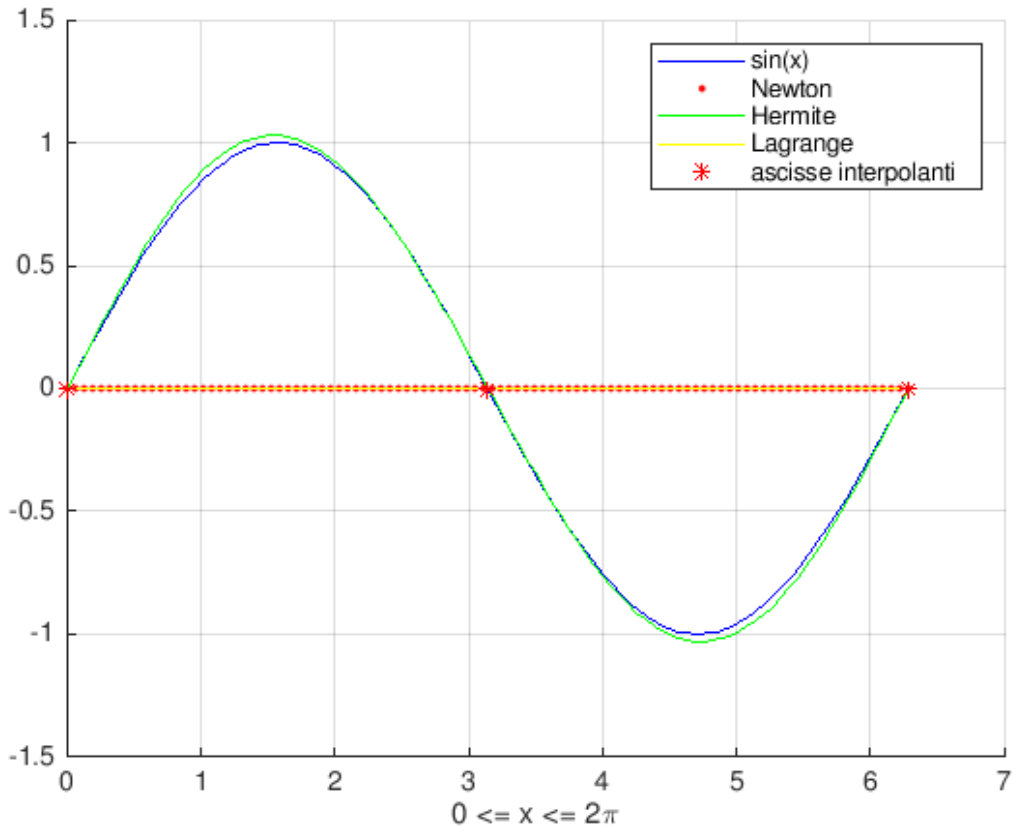
Utilizzare le functions degli esercizi precedenti per disegnare l'approssimazione della funzione $\sin(x)$ nell'intervallo $[0, 2\pi]$, utilizzando le ascisse di interpolazione $x_i = i\pi$, $i = 0, 1, 2$.

Il seguente codice Matlab contiene le chiamate alle funzioni degli esercizi precedenti:
$y = newton(xi, fi, x)$;
$y = lagrange(xi, fi, x)$;
$y = hermite(xi, fi, f1i, x)$;
calcolate fornendo in input le ascisse $0, \pi, 2\pi$ e la loro immagine attraverso f=$\sin(x)$. Nel caso di Hermite anche la loro immagine attraverso f'=$\cos(x)$.

```
1  f = @(x) sin(x); % la funzione
2  xi = zeros(3,1); % vettore contenente le ascisse interpolanti
       la funzione
3  di = zeros(3,1); % vettore contenente i valori delle derivate
4                   % della funzione relativa alle ascisse
                       interpolanti
5  for j = 1:length(xi)
6      xi(j) = (j-1)*pi;
7      di(j) = cos(xi(j));
8  end
9  fi = f(xi);
10 x = linspace(0,2*pi);
11 pN = newton(xi,fi,x);
12 pL = lagrange(xi,fi,x);
13 pH = hermite(xi,fi,di,x);
14
15 figure;
16 hold on;
17 grid on;
18 fplot(f, [0, 2*pi], 'b');
19 plot(x, pN, 'r.', 'Markersize', 7);
20 plot(x, pH, 'g', 'Markersize', 7);
21 plot(x, pL, 'y', 'Markersize', 7);
22 plot(xi, fi, 'r*');
23 xlabel('0 <= x <= 2\pi');
24 legend('sin(x)', 'Newton', 'Hermite', 'Lagrange', 'ascisse
```

```
    interpolanti');
```

Nella figura sottostante é riportata l'approssimazione della funzione $\sin(x)$ tramite l'utilizzo delle funzioni di interpolazione:



Essendo $f_i = 0$ per tutte le ascisse x_i, sia il polinomio interpolante di Lagrange, che quello di Newton in realtá sono la retta $y = 0$.

4.5

Scrivere una function Matlab che implementi la *spline* cubica interpolante (naturale o *not-a-knot*, come specificato in ingresso) delle coppie di dati assegnate. La forma della function deve essere del tipo: `y = spline3( xi, fi, x, tipo)`

I seguenti codici Matlab, contengono la soluzione al problema dato:

```
function y = spline3(xi,fi,x,tipo)
    % y = spline3(xi,fi,x,tipo) calcola la spline cubica
       specificata dal
    % parametro tipo
    %INPUT:
    %xi = n+1 nodi di interpolazione
    %fi = valori della funzione calcolata in xi
    %x = vettore di valori nel quale voglio calcolare la
       spline
    %tipo = specifica il tipo di spline cubica tra naturale o
        not-a-knot
    %OUPTUT:
    %y = vettore contenente i risultati della valutazione nei
        punti di x
    phi = zeros(1,length(xi)-2);
    eps = zeros(1,length(xi)-2);
    for i=1:length(xi)-2
        hi = xi(i+1)-xi(i);
        h1 = xi(i+2)-xi(i+1);
        phi(i) = hi/(hi+h1);
        eps(i) = h1/(hi+h1);
    end
    diffDiv = diffDivise(xi,fi);
    if tipo == 0
        %cubica naturale
        mi = solveSplineNat(phi,eps,diffDiv);
        mi = [0;mi;0];
    else
        %cubica not a knot
        mi = solveSplineNaK(phi,eps,diffDiv);
    end
```

```
28      s = createSpline(mi,fi,xi);
29      y = evaluateSpline(s,xi,x);
30  end
```

```
function diffDiv = diffDivise(xi,fi)
    % fi = diffDivise(xi,fi) calcola le differenze divise
    % INPUT:
    % xi = nodi di interpolazione
    % fi = valori calcolati nei nodi di interpolazione
    % OUTPUT:
    % diffDiv = vettore contente le differenze divise
    n=length(xi)-1;

    for j=1:n-1
        for i=n+1:-1:j+1
            fi(i)=(fi(i)-fi(i-1))/(xi(i)-xi(i-j));
        end
    end

    diffDiv=fi(3:length(fi))';
end
```

```
function xi = solveSplineNat(b,c,xi)
    % solveSplineNat(b,c,xi) calcola la soluzione
    % del sistema lineare Ax = b dove A fattorizzabile LU
    % INPUT:
    % b = contiene le phi
    % c = contiene le eps
    % xi = contiene le differenze divise
    xi = 6*xi;
    a = ones(length(xi),1)*2;
    n = length(xi);
    for i = 1:n-1
        b(i) = b(i)/a(i);
        a(i+1) = a(i+1)- b(i)*c(i);
    end
    xi(2:n) = xi(2:n) - b(1:n-1) .* xi(1:n-1);
    xi(n) = xi(n)/a(n);
    for i = 1:n-1
        xi(n-i) = (xi(n-i)- c(n-i)*xi(n-i+1))/a(n-i);
    end
end
```

```
function xi = solveSplineNaK(phi,eps,xi)
    % mi = solveSplineNaK(phi,eps,xi) calcola la soluzione
    % del sistema lineare Ax = b dove A fattorizzabile LU
    % INPUT:
    % b = contiene le phi
    % c = contiene le eps
    % xi = contiene le differenze divise

    xi = [6*xi(1);6*xi;6*xi(length(xi))];
    n = length(eps)+1;
    a = zeros(n+1,1);
    b = zeros(n,1);
    c = zeros(n,1);
    a(1) = 1;
    c(1) = 0;
    b(1) = phi(1)/a(1);
    a(2) = 2-phi(1);
    c(2) = eps(1)-phi(1);
    b(2) = phi(2)/a(2);
    c(3) = eps(2);
    a(3) = 2-(b(2)*c(2));
    for i = 4:n-1
        b(i-1) = phi(i-1) / a(i-1);
        a(i) = 2 - b(i-1) * c(i-1);
        c(i) = eps(i-1);
    end

    b(n-1) = (phi(n-1)-eps(n-1))/a(n-1);
    a(n) = 2-eps(n-1)-b(n-1)*c(n-1);
    c(n) = eps(n-1);

    b(n) = 0;
    a(n+1) = 1;
    xi(2:n+1) = xi(2:n+1) - b(1:n) .* xi(1:n);
    xi(n+1) = xi(n+1)/a(n+1);
    for i = n:-1:1
        xi(i) = (xi(i)- c(i)*xi(i+1))/a(i);
    end
```

```
39      xi(1) = xi(1)-xi(2)-xi(3);
40      xi(n+1) = xi(n+1)-xi(n)-xi(n-1);
41  end
```

```
function s = createSpline(mi,fi,xi)
    % s = createSpline(mi,fi,xi) calcola i polinomi che
       costituiscono la
    % spline cubica
    % INPUT:
    % mi = derivata seconda della spline spline cubica
       calcolata in xi
    % xi = nodi di interpolazione
    % fi = valori della funzione calcolata in xi
    n = length(xi);
    s = sym('x',[n-1 1]);
    syms x;
    for i = 2:n
        hi = xi(i)-xi(i-1);
        ri = fi(i-1)-((hi^2)/6)*mi(i-1);
        qi = ((fi(i)-fi(i-1))/hi)-(hi/6) * (mi(i)-mi(i-1));
        s(i-1) = (((x-xi(i-1))^3)*mi(i)+((xi(i)-x)^3)*mi(i-1)
           )/(6*hi);
        s(i-1) = s(i-1)+qi*(x-xi(i-1))+ri;
    end
end
```

```
function x = evaluateSpline(s,xi,x)
    % y = evaluateSpline(s,xi,x) valuta la spline s nei punti
    % contenuti in x
    % INPUT:
    % s = vettore contenente i polinomi che costituiscono la
       spline
    % xi = n+1 nodi di interpolazione
    % x = valori nel quale voglio valutare la spline
    n = length(xi) - 1;
    z = 1;
    k = 1;
    for i = 1:n
        isInt = 1;
        while k <= length(x) && isInt
            if x(k) >= xi(i) && x(k)<= xi(i+1)
                k = k+1;
```

```
            else
                isInt = 0;
            end
        end
        x(z:k-1) = subs(s(i),x(z:k-1));
        z = k;
    end
end
```

4.6

Scrivere una function Matlab che implementi il calcolo delle ascisse di Chebyshev per il polinomio interpolante di grado n, su un generico intervallo $[a, b]$.

La function deve essere del tipo: `xi = ceby( n, a, b )`

Il seguente codice Matlab implementa la function richiesta.

```
function xi = ceby(n,a,b)
    % xi = ceby(n,a,b)
    % Calcola le ascisse di Chebyshev per il polinomio
    % interpolante di grado n, trasformate in [a,b]
    % INPUT:
    % n =  numero di intervalli desiderati tra a e b che
       definiscono la
    % partizione delle ascisse interpolanti
    % a = estremo sinistro
    % b = estremo destro
    % OUTPUT:
    % xi = vettore ascisse di Chebyshev

    xi = cos( ((2*[0:n]+1)*pi) / (2*n+2) );
    xi = ((a+b)+(b-a)*xi)/2;
end
```

4.7

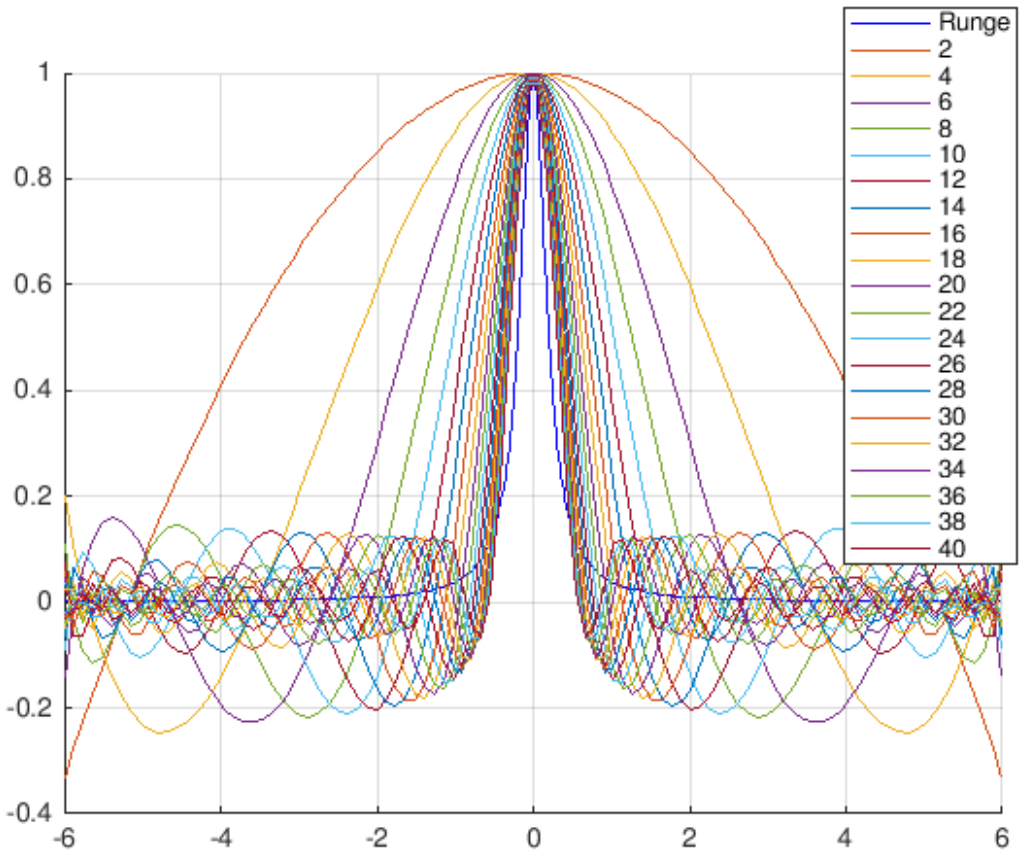
Utilizzare le function degli Esercizi 4.1 e 4.6 per graficare l'approssimazione della funzione di Runge sull'intervallo $[-6, 6]$, per $n = 2, 4, 6, \ldots, 40$. Stimare numericamente l'errore commesso in funzione del grado n del polinomio interpolante.

Il seguente codice Matlab implementa la soluzione al problema dato:

```
%Soluzione esercizio 7 capitolo 4
f = @(x) 1./(1+25 * x.^2); %funzione di Runge
a = -6;
b = 6;
```

```
n = 2;
xi = [];
err = zeros(20,1);
i = 1;
hold on;
grid on;
x = linspace(a,b);
fplot(f, [a,b], 'b','Markersize', 7);
while n <= 40
    xi = ceby(n,a,b);
    fi = f(xi);
    y = lagrange(xi,fi,x);
    err(i) = norm(f(x)-y, inf);
    plot(x,y);
    n = n+2;
    i = i+1;
end
legend('Runge','2','4','6','8','10','12','14','16','18','20',
   '22','24','26','28','30','32','34','36','38','40');
hold off
```

Il grafico seguente mostra i polinomi interpolanti di grado n, calcolati utilizzando come punti di interpolazione quelli corrispondenti alle n ascisse di Chebyshev. Si ricorda la funzione di Runge scritta come: $f(x) = \frac{1}{1+25x^2}$.


Runge
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-6
-4
-2
0
2
4
6

Abbiamo quindi calcolato l'errore al variare di n (con f = funzione di Runge e $p_n(x)$ = il suo polinomio interpolante di grado n) come segue:

$$||err|| \approx ||f(x) - p_n(x)||_{\text{inf}}$$

Nella seguente tabella si riportano gli errori calcolati:

n	$\|\|err\|\|$
2	0.9244
4	0.8717
6	0.8217
8	0.7757
10	0.7262
12	0.6866
14	0.6464
16	0.6025
18	0.5568
20	0.5291
22	0.5000
24	0.4696
26	0.4384
28	0.4067
30	0.3747
32	0.3427
34	0.3110
36	0.2855
38	0.2713
40	0.2570

Grazie alla scelta delle ascisse di Chebyshev come punti di interpolazione, l'errore diminuisce all'aumentare di n.

4.8

Relativamente al precedente esercizio, stimare numericamente la crescita della costante di Lebesgue.

I seguenti codici Matlab contengono il calcolo della costante di Lebesgue in funzione di n. La costante di Lebesgue è definita come segue:

$$\Lambda_n = ||\lambda_n|| \quad con\ \lambda_n(x) = \sum_{i=0}^{n} |L_{(i,n)}(x)|$$

```
%Soluzione esercizio 8 capitolo 4
a = -6;
b = 6;
n = 2;
constLeb = zeros(20,1);
i = 1;
while n <= 40
    xi = ceby(n,a,b);
    constLeb(i) = computeLeb(xi);
    n = n+2;
    i = i+1;
end
```

```
function y = computeLeb(xi)
    %y = computeLeb(xi) calcola la costante di Lebesgue
       relativa alle
    %ascisse in xi
    %INPUT:
    %xi = vettore delle ascisse
    %OUTPUT
    %y = costante di Lebesgue
    n = length(xi);
    x = linspace(xi(1),xi(end),10000);
    L = ones(n,length(x));
    sumL = 0;
    for i = 1:n
        for j = 1:n
            if(i~=j)
                L(i,:) = L(i,:).*(x-xi(j))/(xi(i)-xi(j));
```

```
            end
        end
        sumL = sumL + abs(L(i,:));
        y = norm(sumL,inf);
    end
end
```

Nella seguente tabella viene mostrata come varia la costante di Lebesgue in funzione di n. Come si può notare, grazie alle ascisse di Chebyshev, si ha una crescita logaritmica della costante al variare del grado n del polinomio:

n	$lebesgue$
2	1.2500
4	1.5702
6	1.7825
8	1.9416
10	2.0687
12	2.1747
14	2.2655
16	2.3450
18	2.4156
20	2.4792
22	2.5370
24	2.5900
26	2.6386
28	2.6843
30	2.7266
32	2.7662
34	2.8036
36	2.8391
38	2.8718
40	2.9022

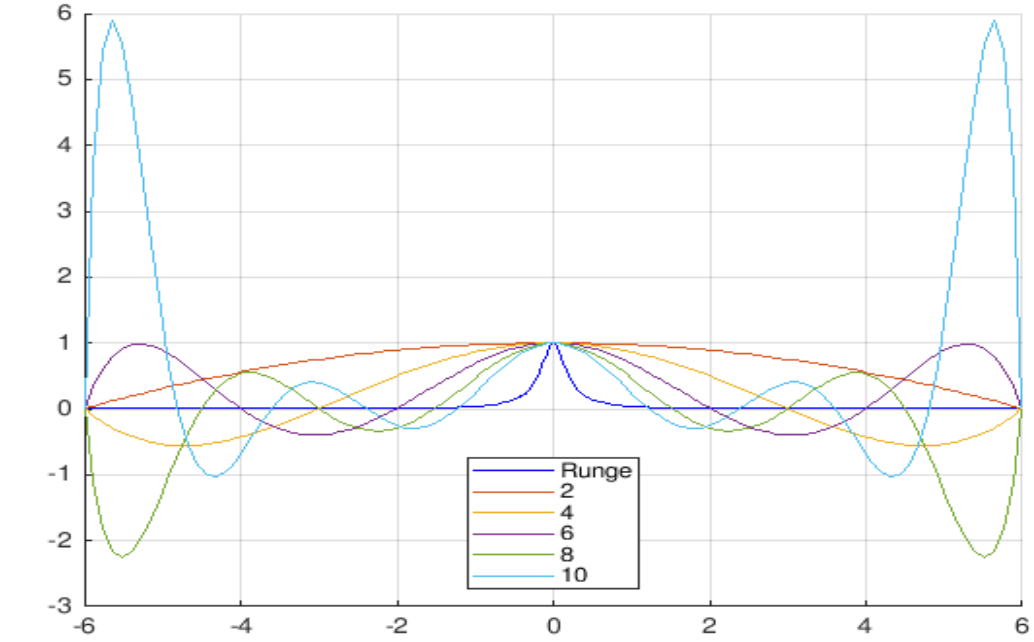
4.9

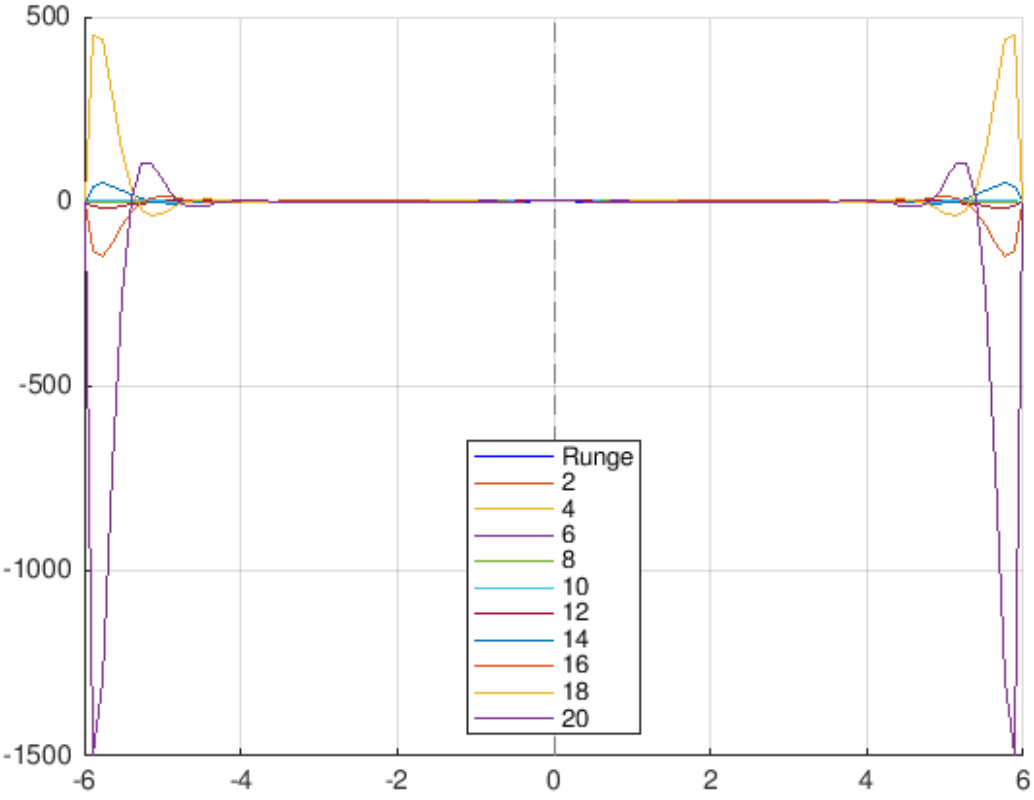
Utilizzare la function dell'Esercizio 4.1 per approssimare la funzione di Runge sull'intervallo $[-6, 6]$, su una partizione uniforme di $n + 1$ ascisse per $n = 2, 4, 6, \ldots, 40$. Stimare le corrispondenti costanti di Lebesgue.

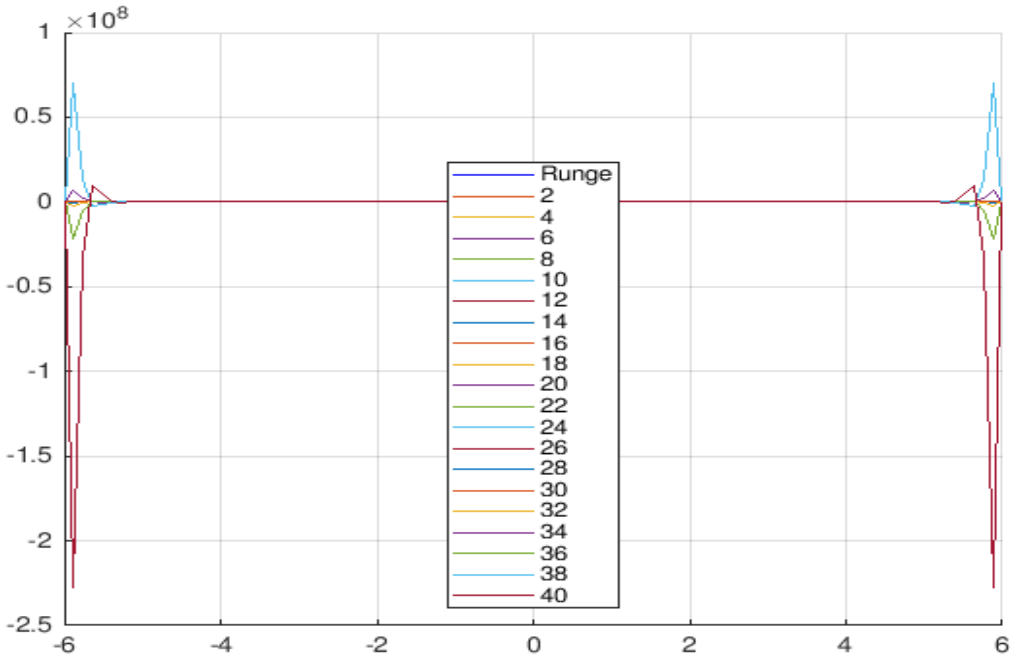
Il seguente codice Matlab contiene la soluzione al problema dato:

```
%Soluzione esercizio 9 capitolo 4
f = @(x) 1./(1+25 * x.^2); %funzione di Runge
a = -6;
b = 6;
n = 2;
xi = [];
err = zeros(20,1);
constLeb = zeros(20,1);
i = 1;
hold on;
grid on;
x = linspace(a,b);
fplot(f, [a,b], 'b','Markersize', 7);
while n <= 40
    xi = linspace(a,b,n+1);
    fi = f(xi);
    y = lagrange(xi,fi,x);
    err(i) = norm(f(x)-y, inf);
    constLeb(i) = computeLeb(xi);
    plot(x,y);
    n = n+2;
    i = i+1;
end
legend({'Runge','2','4','6','8','10','12','14','16','18','20'
   ,'22','24','26','28','30','32','34','36','38','40'},'
   Location','south');
hold off
```

Di seguito i grafici mostrano i polinomi interpolanti di Lagrange al variare del grado N con $N = 2, 4, 6, \ldots, 40$.







Nella tabella è riportato come varia la *costante di Lebesgue*, al variare del grado n del polinomio interpolante. Come si può vedere, all'aumentare di n l'errore aumenta a causa della scelta delle ascisse equispaziate.

n	$lebesgue$
2	0.9342
4	0.8566
6	0.9846
8	2.2590
10	5.8960
12	16.3788
14	49.2750
16	147.6550
18	450.3933
20	$1.5025e+03$
22	$5.0066e+03$
24	$1.6654e+04$
26	$5.5282e+04$
28	$1.8307e+05$
30	$6.0468e+05$
32	$1.9918e+06$
34	$6.5422e+06$
36	$2.1426e+07$
38	$6.9960e+07$
40	$2.2774e+08$

4.10

Stimare, nel senso dei minimi quadrati, posizione, velocitá iniziale ed accelerazione relative ad un moto rettilineo uniformemente accelerato per cui sono note le seguenti misurazioni dele coppie $(tempo, spazio)$:
$(1, 2.9)$ $(1, 3.1)$ $(2, 6.9)$ $(2, 7.1)$ $(3, 12.9)$ $(3, 13.1)$ $(4, 20.9)$ $(4, 21.1)$ $(5, 30.9$

La legge che descrive il fenomeno del moto rettilineo uniformemente accelerato si puó scrivere in forma polinomiale come segue:

$$y = s(t) = x_0 + v_0 t + a_0 t^2 \qquad \text{con } a_0 = \frac{1}{2}a$$

Il cui grado è n = 2. Il sistema ha soluzione se si ha almeno n+1 punti distinti. In questo caso il problema é ben posto poiché i punti distinti sono 5>3.
Si vuole quindi stimare nel senso dei minimi quadrati: posizione, velocità iniziale, ed accelerazione, che equivale alla risoluzione del sistema lineare sovradeterminato:

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

con V matrice di tipo *Vandermonde* (la trasposta di una matrice di tipo Vandermonde), $\underline{a}$ vettore delle incognite e $\underline{y}$ il vettore dei valori misurati.
Tale sistema si risolve mediante fattorizzazione QR. La matrice V è scritta come segue:

$$V = \begin{bmatrix} x_0^0 & x_0^1 & \cdots & x_0^m \\ x_1^0 & x_1^1 & \cdots & x_1^m \\ \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^m \end{bmatrix}$$

```
1 %Soluzione esercizio 10 cap 4
2
3 %FORMULA MOTO RETTILINEO UNIFORMEMENTE ACCELERATO
4 %x(t) = x0+v0*t+1/2*a*t^2
5 %Il sistema ha soluzione se ho almeno n+1 punti distinti
6 %In questo caso il problema è ben posto poiche i punti
     distinti sono 5>3
7 value = 1;
8 V = zeros(10,3);
```

```
for i = 1:10
    V(i, 1:3) = value.^(0:2);
    r = mod(i,2);
    if r == 0
        value = value + 1;
    end
end
y = [2.9 3.1 6.9 7.1 12.9 13.1 20.9 21.1 30.9 31.1];
V = QRFatt(V);
[m,n] = size(V);
x = SolveLeastSquares(V,y,m,n);
```

Le soluzioni, calcolate, al problema dato sono :

$$x_0 = 1,\ v_0 = 1,\ a_0 = 1$$

5 Capitolo

5.1

Scrivere una function Matlab che implementi la formula composita dei trapezi su $n+1$ ascisse equidistanti nell'intervallo $[a,\ b]$, relativamente alla funzione implementata da **fun(x)**. La function deve essere del tipo: **If = trapcomp(n, a, b, fun)**.

Il codice riportato in seguito implementa la formula *composita dei trapezi* su $n+1$ ascisse equidistanti definite sull'intervallo $[a,b]$, relativa alla generica funzione $fun(x)$ e ne restituisce l'approssimazione del relativo integrale.

```
1  function If = trapcomp(n,a,b,fun)
2      %If = trapcomp(n,a,b,fun) calcola l'integrale tra a e b
          della funzione
3      %fun applicando il metodo dei trapezi composito con n
          sottointervalli
4      %INPUT:
5      %n = numero dei sottointervalli
6      %a = estremo sinistro
7      %b = estremo destro
8      %fun = funzione da integrare
9      %OUTPUT:
10     %If = approssimazione dell'integrale di fun
11     x = linspace(a,b,n+1);
12     f = feval(fun,x);
13     If = ((b-a)/n)*(sum(f)-0.5*(f(1)+f(end)));
14 end
```

5.2

Scrivere una function Matlab che implementi la formula composita di Simpson su $2n + 1$ ascisse equidistanti nell'intervallo $[a, b]$, relativamente alla funzione implementata da **fun(x)**. La function deve essere del tipo: **If = simpcomp(n, a, b, fun)**.

Il codice riportato in seguito implementa la formula *composita di Simpson* su $2n+1$ ascisse equidistanti definite sull'intervallo $[a,b]$, relativa alla generica funzione $fun(x)$ e ne restiusce l'approssimazione del relativo integrale.

```
1  function If = simpcomp(n,a,b,fun)
2      %If = simpcomp(n,a,b,fun) calcola l'integrale tra a e b
          della funzione
3      %fun applicando il metodo dei Simpson composito con 2n
          sottointervalli
4      %INPUT:
5      %n = numero dei sottointervalli (pari)
6      %a = estremo sinistro
7      %b = estremo destro
8      %fun = funzione da integrare
9      %OUTPUT:
10     %If = approssimazione dell'integrale di fun
11     r = mod(n,2);
12     if r ~= 0
13         error('Il numero di sottointervalli non è pari');
14     end
15     x = linspace(a,b,n+1);
16     f = feval(fun,x);
17     If = 4*sum(f(2:2:n))+2*sum(f(1:2:n+1))-(f(1)+f(end));
18     If = If*((b-a)/(3*n));
19 end
```

5.3

Scrivere una function Matlab che implementi la formula composita dei trapezi adattativa nell'intervallo *[a, b]*, relativamente alla funzione implementata da **fun(x)**, e con tolleranza **tol**. La function deve essere del tipo: **If = trapad(a, b, fun, tol)**.

Il codice riportato in seguito implementa la formula *adattiva dei trapezi* sull'intervallo *[a,b]*, relativa alla generica funzione fun(x), con una tolleranza tol e ne restiusce l'approssimazione del relativo integrale.

```
1  function If = trapad(a,b,fun,tol,fa,fb)
2      %If = trapad(a,b,fun,tol[,fa,fb]) calcola l'integrale
          della funzione
3      %fun utilizzando la formula dei trapezi adattiva nell'
          intervallo [a,b]
4      %con tolleranza tol
5      %INPUT:
6      %a = estremo sinistro
7      %b = estremo destro
8      %fun = funzione da integrare
9      %tol = tolleranza sull'errore
10     if nargin == 4
11         fa = feval(fun,a);
12         fb = feval(fun,b);
13     end
14     x = (a+b)/2;
15     fx = feval(fun,x);
16     I1 = ((b-a)*(fa+fb))/2;
17     If = (I1+(b-a)*fx)/2;
18     err = abs(I1-I)/3;
19     if err > tol
20         t1 = trapad(a,x,fun,tol/2,fa,fx);
21         t2 = trapad(x,b,fun,tol/2,fx,fb);
22         If = t1+t2;
23     end
24 end
```

5.4

Scrivere una function Matlab che implementi la formula composita di Simpson adattativa nell'intervallo *[a, b]*, relativamente alla funzione implementata da **fun(x)**, e con tolleranza **tol**. La function deve essere del tipo: **If = simpad(a, b, fun, tol)**.

Il codice riportato in seguito implementa la formula *composita di Simpson adattiva* sull'intervallo *[a,b]* relativa alla generica funzione fun(x), con una tolleranza *tol* e ne restuisce l'approssimazione del relativo integrale.

```
1  function If = simpad(a,b,fun,tol,fa,fb,fc)
2      %If = trapad(a,b,fun,tol[,fa,fb,fc]) calcola l'integrale
          della funzione
3      %fun utilizzando la formula di Simpson adattiva nell'
          intervallo [a,b]
4      %con tolleranza tol
5      %INPUT:
6      %a = estremo sinistro
7      %b = estremo destro
8      %fun = funzione da integrare
9      %tol = tolleranza sull'errore
10     c = (a+b)/2;
11     if nargin <= 4
12         fa = feval(fun,a);
13         fb = feval(fun,b);
14         fc = feval(fun,c);
15     end
16     x1 = (a+c)/2;
17     x2 = (b+c)/2;
18     f1 = feval(fun,x1);
19     f2 = feval(fun,x2);
20     h = (b-a)/6;
21     I1 = h*(fa+(4*fc)+fb);
22     If = (0.5*h)*(fa+(4*f1)+(2*fc)+(4*f2)+fb);
23     err = abs(I1-If)/15;
24     if err > tol
25         t1 = simpad(a,c,fun,tol/2,fa,fb,f1);
```

```
26          t2 = trapad(c,b,fun,tol/2,fc,fb,f2);
27          If = t1+t2;
28      end
29  end
```

5.5

Calcolare quante valutazioni di funzione sono necessarie per ottenere una approssimazione di

$$I(f) = \int_0^1 \exp(-10^6 x)dx$$

che vale 10^-6 in doppia precisione IEEE, con una tolleranza 10^-9, utilizzando le functions dei precedenti esercizi. Argomentare quantitativamente la risposta.

Il seguente codice MatLab contiene la soluzione del problema dell'Es.5 :

```
%script es 5 capitolo 5
f = @(x) exp((-10^6)*x);
a = 0;
b = 1;
tol = 10^-9;
approx = 10^-6;
errTrapComp = zeros(100,1);
errSimpComp = zeros(100,1);
rstTrapComp = zeros(100,1);
rstTrapComp = zeros(100,1);
numIt = zeros(100,1);
i = 1;

%soluzione trapcomp
for n = 1000000:1000000:1000000000
    rstTrapComp(i) = trapcomp(n,a,b,f);
    errTrapComp(i) = abs(rst(i)-approx);
    if errTrapComp(i) < tol
        break;
    else
        i = i+1;
    end
end

if errTrapComp(i) > tol || i == length(errTrapComp)
    error('Tolleranza non raggiunta con il metodo dei Trapezi
        composito');
```

```
end


%soluzione simpcomp
i = 1;
for n = 100000:100000:1000000000
    rstSimpComp(i) = simpcomp(n,a,b,f);
    errSimpComp(i) = abs(rstSimpComp(i)-approx);
    if errSimpComp(i) < tol
        break;
    else
        i = i+1;
    end
end

if err(i) > tol || i == length(errSimpComp)
    error('Tolleranza non raggiunta con il metodo di Simpson
       composito');
end

%soluzione Simpson adattiva
[ISAD,numValSimpAd] = simpad(a,b,f,tol);

%soluzione Trapezi adattiva
[ITAD,numValTrapAD] = trapad(a,b,f,tol);
```

restituendo i seguenti valori:

Formula dei Trapezi Composita:

Per soddisfare la richiesta di avere un errore minore di una $tol = 10^-9$, occorre scegliere il giusto numero di sotto intervalli. Precedendo a *tentativi*, abbiamo iterato n partendo da 1000000 fino ad un massimo di 1000000000, con un passo di 1000000. Si può concludere che con 1000000 <= n <= 1000000000 la richiesta è soddisfatta.

Ultime due iterazioni:

tol	$num.val.funz. = n \quad (sottointervalli)$	$I = tc$	$E_1^{(n)}$
10^{-9}	9000000	$1.001028594957969e-06$	$1.028594957968679e-09$
10^{-9}	10000000	$1.000833194477503e-06$	$8.331944775031580e-10$

Formula dei Simpson Composita:

In questo caso si è usato un procedimento analogo al precedente.

Ultime due iterazioni:

tol	$num.val.funz. = n \quad (sottointervalli)$	$I = sc$	$E_2^{(n)}$
10^{-9}	1500000	$1.001041922369633e-06$	$1.041922369633001e-09$
10^{-9}	1600000	$1.000809844227888e-06$	$8.098442278876320e-10$

Formula dei Trapezi Adattiva:

In questo caso, come riportato nella tabella, il numero esatto di iterazioni per ottenere il risultato esatto è 25943.

tol	$num.val.funz.$	$I = ta$
10^{-9}	25943	$1.000000011252939e-06$

Formula di Simpson Adattiva:

Anche in questo caso il numero di iterazioni richieste è noto. Tale valore (349) mostra come la formula di *Simpson adattiva* sia molto più veloce rispetto a quella *adattiva dei trapezi*.

tol	$num.val.funz.$	$I = sa$
10^{-9}	349	$1.000000016469981e-06$

6 Capitolo

6.1

Scrivere una function Matlab che generi la matrice *sparsa* $n \times n$, con $n > 10$

$$A_n = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, \text{ con } a_{ij} = \begin{cases} 4 \text{ se } i = j \\ -1 \text{ se } i = j \pm 1 \\ -1 \text{ se } i = j \pm 10 \end{cases}$$

Utilizzare,a questo fine, la function Matlab spdiags.

Il seguente codice Matlab risolve il problema dato utilizzando la funzione `spdiags`:

```
function A = createSparseMatrix(n)
    %A = createSparseMatrix(n) crea una matrice sparsa
    %INPUT
    %n = grado della matrice
    %OUTPUT
    %A = matrice sparsa n*n in cui:
    %a(i,j) = 4 se i=j
    %a(i,j) = -1 se i=j+-1
    %a(i,j) = -1 se i=j+-10
    if n <= 10
        error('n deve essere maggiore di 10');
    end
    e = ones(n,1);
    A = [-1*e 4*e -1*e -1*e -1*e];
    A = spdiags(A,[-1:1,-10, 10],n,n);
end
```

6.2

Utilizzare il metodo delle potenze per calcolarne l'autovalore dominante della matrice A_n del precedente esercizio, con una approssimazione $tol = 10^{-5}$, partendo da un vettore con elementi costanti. Riempire, quindi, la seguente tabella:

n	*numero iterazioni effettuate*	*stima autovalore*
100		
200		
⋮		
1000		

Di seguito sono riportati i codici implementati. La funzione *potenze* calcola sia l'autovalore dominante della matrice A , sia il numero di iterazioni, *numIt* , impiegate.

```
%Soluzione es 2 capitolo 6
tol = 10^-5;
numIt = zeros(10,1);
autVal = zeros(10,1);
index = 1;
for i = 100:100:1000
    A = createSparseMatrix(i);
    v = ones(i,1);
    [autVal(index),numIt(index)] = potenze(A,tol,v);
    index = index+1;
end
```

```
function [lambda,numIt] = potenze(A,tol,x0,maxit)
    %[lambda,numIt] = potenze(A,tol,[x0,maxit]) calcola l'
       autovalore
    %dominante della matrice A.
    %INPUT:
    %A = matrice
    %tol = tolleranza richiesta
    %x0 = vettore iniziale
    %maxit = numero massimo di iterazioni
```

```
    %OUTPUT:
    %lambda = autovalore domaninate
    %numIt = numero di iterazioni effettuate
    n = size(A,1);
    if nargin <= 2
        x = rand(n,1);
    else
        x = x0;
    end
    x = x/norm(x);
    if nargin <= 3
        maxit = 100*n*max(round(-log(tol),1));
    end
    lambda = Inf;
    for i=1:maxit
        lambda0 = lambda;
        v = A*x;
        lambda = x'*v;
        err = abs(lambda-lambda0);
        if err <= tol
            break;
        end
        x = v/norm(v);
    end
    if err > tol
        warning('Tolleranza richiesta non raggiunta');
    else
        numIt = i;
    end
end
```

Nella seguente tabella é possibile visualizzare i risultati ottenuti:

n	*numero iterazioni effettuate*	*stima autovalore*
100	167	7.8224
200	420	7.8803
300	638	7.8916
400	721	7.8949
500	743	7.8964
600	824	7.8974
700	893	7.8976
800	868	7.8967
900	795	7.8957
1000	775	7.8954

6.3

Utilizzare il metodo di Jacobi per risolvere il sistema lineare

$$A_n \mathbf{x} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

dove A_n è la matrice definita nell'Esercizio 6.1, con tolleranza $tol = 10^{-5}$, e partendo dal vettore nullo. Graficare il numero di iterazioni necessarie, rispetto alla dimensione n del problema, con n che varia da 100 a 1000 (con passo 20).

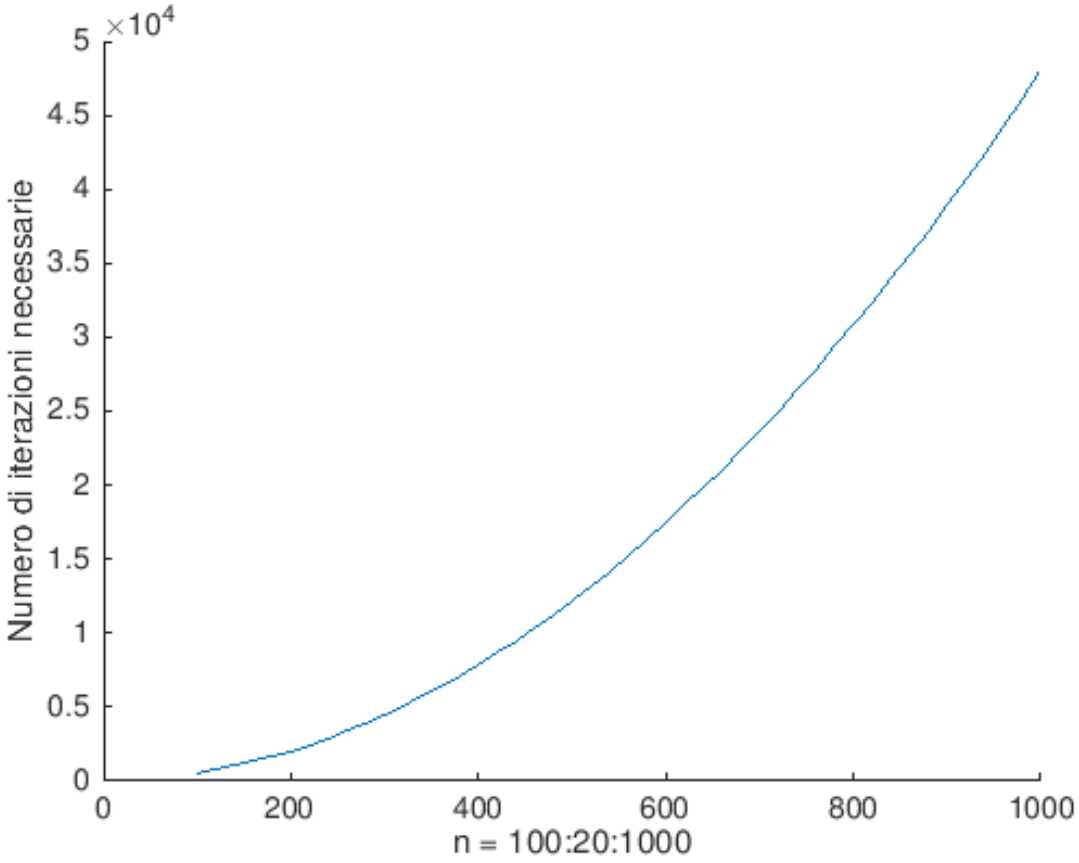
Di seguito si riportano i codici Matlab utilizzati. La funzione di *Jacobi* oltre ad effettuare il calcolo del vettore delle incognite della matrice A, restituisce anche il numero di iterazioni *numIt* impiegate e la norma infinito del residuo, al passo i-simo.

```
%Soluzione es 3 cap6
tol = 10^-5;
numIt = zeros(46,1);
index = 1;
for i = 100:20:1000
    A = createSparseMatrix(i);
    x0 = zeros(i,1);
    b = ones(i,1);
    [x,numIt(index)] = jacobi(A,b,tol,x0);
    index = index+1;
end
hold on;
plot(100:20:1000,numIt);
```

```
function [x,numIt,normRes] = jacobi(A,b,tol,x0)
    %[x,numIt,normRes] = jacobi(A,b,tol,[x0]) calcola la
       soluzione del sistema
    %lineare sparso Ax = b utilizzando il metodo iterativo di
        Jacobi con
    %tolleranza tol
    %INPUT:
    %A = matrice sparsa
```

```
    %b = vettore termini noti
    %tol = tolleranza richiesta
    %x0 = vettore iniziale
    %OUTPUT:
    %x = soluzione del sistema lineare sparso
    %numIt = numero di iterazioni effettuate
    n = length(b);
    if nargin <= 3
        x = zeros(n,1);
    else
        x = x0;
    end
    D = diag(A);
    maxit = 100*n*max(1,-log(tol));
    normRes = zeros(round(maxit),2);
    for i = 1:maxit
        r = b-(A*x);
        err = norm(r,inf);
        normRes(i,1) = i;
        normRes(i,2) = err;
        if err <= tol
            break;
        end
        x = x + r./D;
    end
    if err > tol
        warning('Tolleranza richesta non raggiunta');
    else
        numIt = i;
    end
end
```

Graficamente si puó osservare il seguente risultato, al variare di n:


×10^4
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
Numero di iterazioni necessarie
0
200
400
600
800
1000
n = 100:20:1000

6.4

Ripetere una procedura analoga a quella del precedente esercizio utilizzando il metodo di Gauss-Seidel.

Il codice Matlab utilizzato per realizzare il grafico é il seguente:

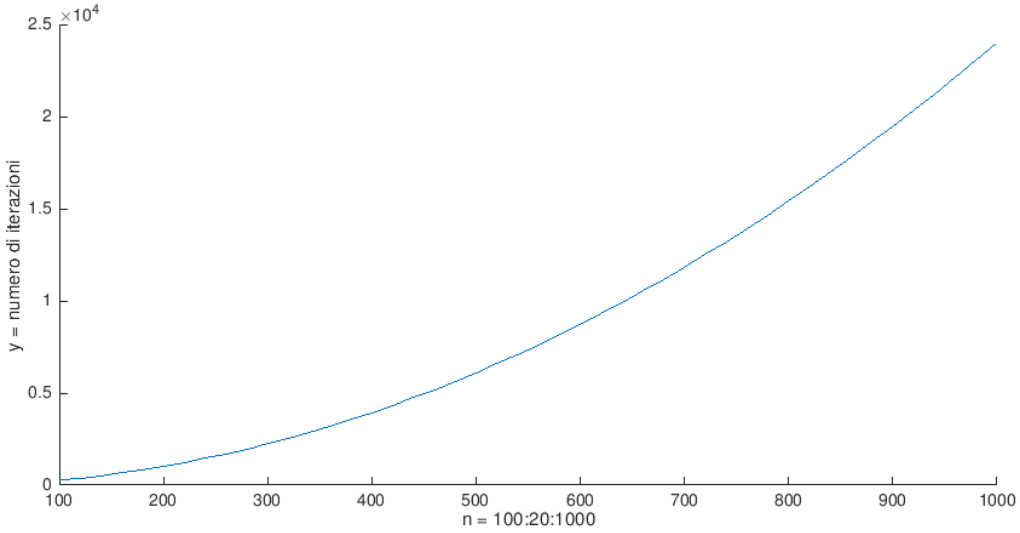
```
%Soluzione es 4 cap6
tol = 10^-5;
numIt = zeros(46,1);
index = 1;
for i = 100:20:1000
    A = createSparseMatrix(i);
    x0 = zeros(i,1);
    b = ones(i,1);
    [x,numIt(index)] = gaussSeidel(A,b,tol,x0);
    index = index+1;
end
hold on;
plot(100:20:1000,numIt);
```

```
function [x,numIt,normRes] = gaussSeidel(A,b,tol,x0)
        %[x,numIt,normRes] = gaussSeidel(A,b,tol,x0) risolve
           il sistema lineare Ax=b
        %con il metodo di Gauss-Seidel
    %INPUT:
    %A = matrice sparsa
    %b = vettore dei termini noti
    %tol = tolleranza richiesta
    %x0 = vettore iniziale
    %OUTPUT:
    %x = soluzione del sistema lineare sparso
    %numIt = numero di iterazioni
    n = length(b);
    if nargin <= 3
        x = zeros(n,1);
    else
        x = x0;
    end
    maxit = 100*max(1,-ceil(log(tol)))*n;
```

```
    err = inf;
    for i=1:maxit
        r = (A*x)-b;
        err = norm(r,inf);
        normRes(i,1) = i;
        normRes(i,2) = err;
        if err <= tol
            break;
        end
        r = trisolveInfGaussSeidel(A,r);
        x = x-r;
    end
    if err > tol
        warning('Tolleranza richiesta non raggiunta');
    else
        numIt = i;
    end
end
```

```
function x = trisolveInfGaussSeidel(A,b)
    %x = trisolveInfGaussSeidel(A,b)
    %La funzione restituisce la soluzione del sistema lineare
        Ax = b
    %INPUT:
    %A = matrice triangolare inferiore
    %b = vettore termini noti
    %OUTPUT:
    %x = vettore soluzione
    x = b;
    [m,n] = size(A);
    if m~=n
        error('Matrice non quadrata');
    end
    for i = 1:n
        x(i) = x(i)/A(i,i);
        x(i+1:n) = x(i+1:n)-x(i)*A(i+1:n,i);
    end
end
```

Grafico risultante:



6.5

Con riferimento al sistema lineare dell' Esercizio 6.3, con $n = 1000$, graficare la norma dei residui, rispetto all'indice di iterazione, generati dai metodi di Jacobi e Gauss-Seidel. Utilizzare il formato *semilogy* per realizzare il grafico, corredandolo di opportune *label*.

Il seguente codice Matlab é stato utilizzato per la risoluzione del problema:

```
%Soluzione esercizio 5 cap 6
tol = 10^-5;
A = createSparseMatrix(1000);
x0 = zeros(1000,1);
b = ones(1000,1);
[x,numIt,normRes] = jacobi(A,b,tol,x0);
[x2,numIt2,normRes2] = gaussSeidel(A,b,tol,x0);
hold on;
semilogy(normRes(:,1),normRes(:,2));
semilogy(normRes2(:,1),normRes2(:,2));
legend('Jacobi','Gauss-Seidel');
```

Il grafico seguente mostra la norma dei residui, rispetto all'indice di iterazione generati dai metodi di Jacobi (in azzurro) e Gauss-Seidel (in rosso):