Elaborato di

Calcolo Numerico

Anno Accademico 2019/2020

Niccolò *Piazzesi* - 6335623 - *niccolo.piazzesi@stud.unifi.it*
Pietro *Bernabei* - 6291312 - *pietro.bernabei@stud.unifi.it*

Contents

List of Figures

List of Tables

1 Capitolo 1

1.1 Esercizio 1

Sia $f(x)$ una funzione sufficientemente regolare e sia $h > 0$ una quantità abbastanza "piccola". Possiamo sviluppare i termini $f(x-h)$ e $f(x+h)$ mediante il polinomio di Taylor:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4)$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + O(h^4)$$

Sostituiamo i termini nell'espressione iniziale:

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

$$= \frac{f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + O(h^4) - 2f(x) + f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4)}{h^2} =$$

$$= \frac{h^2 f''(x) + O(h^4)}{h^2} = f''(x) + O(h^2)$$

1.2 Esercizio 2

Eseguendo lo script si ottiene $u = 1.1102 \cdot 10^{-16} = \frac{\epsilon}{2}$, dove ϵ è la precisione di macchina. Il controllo interno ci dice che si esce dal ciclo solamente quando u diventa talmente piccolo che la somma $1 + u$ viene percepita dal calcolatore come uguale a 1. Questo avviene se $u < \epsilon$, e la prima iterazione in cui il controllo risulta vero è proprio quando $u == \frac{\epsilon}{2}$. Il codice può quindi essere utilizzato per calcolare la precisione di macchina di un calcolatore, moltiplicando per 2 il valore di u restituito.

1.3 Esercizio 3

Quando si esegue $a - a + b$ il risultato è 100 mentre quando si esegue $a + b - a$ si ottiene 0. La differenza dei risultati è dovuta al fenomeno della cancellazione numerica:

- nel primo caso la sottrazione avviene sullo stesso numero $a = 10^{20}$. Sottrare un numero da se stesso ha sempre risultato esatto 0 e quindi il risultato finale è corretto
- nel secondo caso la sottrazione avvientе tra i termini $a + b = 10^{20} + 100$ e $a = 10^{20}$. a+b ha le prime 18 cifre in comune con a e , a causa degli errori di approssimazione, le ultime tre cifre vengono cancellate dalla sottrazione, dando 0 come risultato finale.

2 Capitolo 2

2.1 Esercizio 4

```
function x1=radn(x, n)
%
% x1=radn(n.x)
% funzione Matlab che implementa il metodo di newton per il calcolo della
% radice n-esima di un numero positivo x
%
format long e
imax=1000;
tolx=eps;
if x<=0
    error('valore in ingresso errato');
end
x1=x/2;
for i=1:imax
    x0=x1;
    fx=x0^n-x;
    fx1=(n)*x0^(n-1);
    x1=x0-fx/fx1;
    if abs(x1-x0)<=tolx
        break
    end

end
if abs(x1-x0)>tolx
    error('metodo non converge')
end
```

2.2 Esercizio 5

- Metodo di bisezione

```
function [x,i] = bisezione(f,a,b,tolx)
%bisez
%[x,i]=bisezione(f, a, b,  tolx, maxit)
%Pre: f continua in [a,b]
%  Applica il metodo di bisezione per il calcolo della
%  radice dell'equazione f(x)=0
%  f         -funzione
%  a, b      - estremi dell'intervallo
%
% tolx       -tolleranza
% restituisce in x l'approssimazione della radice e in i il numero di
   iterazioni
% VEDI ANCHE: newton, corde, secanti, aitken, newtonmod
    format long e
    fa = feval(f,a);
    fb = feval(f,b);
    if(fa * fb > 0 )
        error('gli estremi  hanno lo stesso segno');
    end
    x0=a;
    imax =  ceil(log2(b-a) - log2(tolx));
    for i = 1:imax
        x = (a+b)/2;
```

```
        fx = feval(f,x);
        if abs(x-x0) <= tolx*(1+abs(x0))
            break
        end
        x0=x;
        if fa*fx<0
            b = x;
            fb = fx;
        else
            a = x;
            fa = fx;
        end
    end

end
```

- Metodo di Newton

```
function [x,i] = newton( f, f1, x0, tolx, maxit )
%newton
%[x,i]=newton(f,f1, x0, tolx, maxit)
%Pre: f derivabile
%  Applica il metodo di newton per il calcolo della
%  radice dell'equazione f(x)=0
%  f          -funzione
%  f1         -derivata di f
%  x0         -approssimazione iniziale
% tolx        -tolleranza
% maxit       -numero massimo di iterazioni(default=100)
% restituisce in x l'approssimazione della radice e in i il numero di
   iterazioni
% VEDI ANCHE: bisezione, corde, secanti, aitken, newtonmod
       if nargin<4
              error('numero argomenti insufficienti');
       elseif nargin==4
               maxit = 100;
       end
       if tolx<eps
              error('tolleranza non idonea');
       end
       x = x0;
       for i = 1:maxit
              fx = feval( f, x );
              f1x = feval( f1, x );
              x = x - fx/f1x;
              if abs(x-x0)<=tolx*(1+abs(x0))
                     break;
              else
                     x0 = x;
              end
       end
       if abs(x-x0) > tolx*(1+abs(x0))
              error('metodo non converge');
       end
end
```

- Metodo delle secanti

```
function [x, i]=secanti(f,x0,x1,tolx,maxit)
%secanti
%[x,i]=secanti(f, x0, x1, tolx, maxit)
%
%  Applica il metodo delle secanti per il calcolo della
%  radice dell'equazione f(x)=0
%  f           -funzione
%  x0          -approssimazione iniziale
%  x1          -seconda approssimazione iniziale
% tolx         -tolleranza
% maxit        -numero massimo di iterazioni(default=100)
% restituisce in x l'approssimazione della radice e in i il numero di iterazioni
% VEDI ANCHE: bisezione, newton, corde, aitken, newtonmod

  format long e
  if nargin<4
    error('numero argomenti insufficienti');
  elseif nargin==4
    maxit = 100;
  end
  i=0;
  f0=feval(f,x0);
  for i=1:maxit
      f1=feval(f,x1);
      df1=(f1-f0)/(x1-x0);
      x=x1-(f1/df1);
      if abs(x1-x0)<=tolx*(1+abs(x0))
        break;
      end
      x0=x1;
      x1=x;
      f0=f1;

  end
  if abs(x-x0) > tolx*(1+abs(x0))
    error('metodo non converge');
  end
end
```

- Metodo delle corde

```
function [x,i] = corde( f, f1, x0, tolx, maxit )
%corde
%[x,i]=corde(f,f1, x0, tolx, maxit)
%Pre: f derivabile
%  Applica il metodo delle corde per il calcolo della
%  radice dell'equazione f(x)=0
%  f           -funzione
%  f1          -derivata di f
%  x0          -approssimazione iniziale
% tolx         -tolleranza
% maxit        -numero massimo di iterazioni(default=100)
% restituisce in x l'approssimazione della radice e in i il numero di iterazioni
% VEDI ANCHE: bisezione, newton, secanti, aitken, newtonmod

    format long e
    if nargin<4
```

```
            error('numero argomenti insufficienti');
    elseif nargin==4
            maxit = 100;
    end
    if tolx<eps
            error('tolleranza non idonea');
    end
    f1x = feval(f1, x0);
    x = x0;
    for i = 1:maxit
            fx = feval( f, x );
            if fx==0
                    break;
            end
            x = x - fx/f1x;
            if abs(x-x0)<=tolx*(1+abs(x0))
                    break;
            else
                    x0 = x;
            end
    end
    if abs(x-x0) > tolx*(1+abs(x0))
        error('metodo non converge');
     end
end
```

2.3 Esercizio 6

Eseguendo lo script es6.msi ottengono i risultati contenuti nella tabella ?? e nella figura ??. Come si può notare, il metodo di newton e il metodo delle secanti convergono molto più rapidamente del metodo di bisezione e del metodo delle corde.

Metodo	tolleranza= 10^{-3}	tolleranza= 10^{-6}	tolleranza= 10^{-9}	tolleranza= 10^{-12}
bisezione	0.739257812500000	0.739085197448730	0.739085133187473	0.739085133215667
newton	0.739085133385284	0.739085133215161	0.739085133215161	0.739085133215161
corde	0.739567202212256	0.739084549575213	0.739085132739254	0.739085133215737
secanti	0.739085133215001	0.739085133215161	0.739085133215161	0.739085133215161

Table 1: valori approssimati da bisezione, newton, secanti, corde

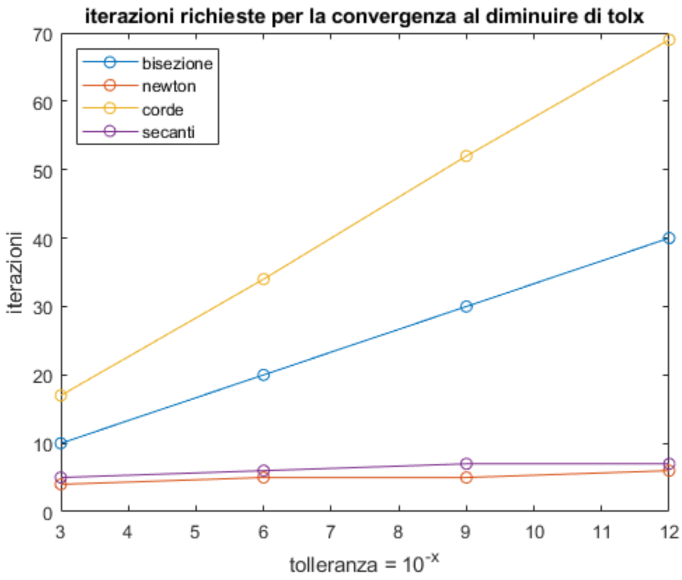


Figure 1: iterazioni richieste

2.4 Esercizio 7

Le nuove funzioni utilizzate in questo esercizio sono:

- Metodo di Newton modificato

```
function [x, i] = newtonmod( f, f1, x0, m, tolx, maxit )
%NEWTONMOLT
%[x,i]=Newtonmolt(f,f1,x0,m,tolx,maxit)
%  Pre: f derivabile
%  Applica il metodo di Newton per il calcolo della
%  radice (di molteplicita' nota r) dell'equazione f(x)=0
%  f        -funzione
%  f1       -derivata di f
%  x0       -approssimazione iniziale
%  m        -molteplicita' della radice
% tolx      -tolleranza
% maxit     -numero massimo di iterazioni(default=100)
% restituisce in x l'approssimazione della radice e in i il numero di
   iterazioni
% VEDI ANCHE: bisezione, newton, secanti, corde, aitken

    format long e
    if nargin<5
           error('numero argomenti insufficienti');
    elseif nargin==5
            maxit = 100;
    end
    if tolx<eps
           error('tolleranza non idonea');
    end
    x = x0;
    for i = 1:maxit
           fx = feval( f, x );
           f1x = feval( f1, x );
           if fx==0
                  break;
```

```
            end
            x = x - m*(fx/f1x);
            if abs(x-x0)<=tolx*(1+abs(x0))
                    break;
            else
                    x0 = x;
            end
    end

end
```

- Metodo delle accelerazioni di Aitken

```
function [x, i] = aitken( f, f1, x0, tolx, maxit )
%aitken
%[x,i]=aitken(f,f1, x0, tolx, maxit)
%  Pre: f derivabile
%  Applica il metodo di accelerazione di aitken per il calcolo della
%  radice (di molteplicita' incognita) dell'equazione f(x)=0
%  f          -funzione
%  f1         -derivata di f
%  x0         -approssimazione iniziale
% tolx        -tolleranza
% maxit       -numero massimo di iterazioni(default=100)
% restituisce in x l'approssimazione della radice e in i il numero di
   iterazioni
% VEDI ANCHE: bisezione, newton, secanti, corde, newtonmod
format long e
if nargin<4
        error('numero argomenti insufficienti');
elseif nargin==4
         maxit = 100;
end
if tolx<eps
        error('tolleranza non idonea');
end
fx = feval(f,x0);
f1x = feval(f1,x0);
x= x0-fx/f1x;
for i = 1:maxit
    x0 = x;
    fx = feval( f, x0 );
    f1x = feval( f1, x0 );
    x1 = x0 - fx/f1x;
    fx = feval( f, x1 );
    f1x = feval( f1, x1 );
    x = x1 -fx/f1x;
    x = (x*x0-x1^2)/(x-2*x1+x0);
    if abs(x-x0)<=tolx
            break;
    end
end
if abs(x-x0) > tolx*(1+abs(x0))
        error('metodo non converge');
end
end
```

La radice nulla della funzione $f(x) = x^2 tan(x)$ ha molteplicità m = 3, in quanto 0 annulla due volte il termine x^2 e una volta il termine $tan(x)$.

Tolleranza	Newton	Newton modificato	Aitken
10^{-3}	$1.99400296195610{\cdot}10^{-3}$	$1.32348898008484{\cdot}10^{-23}$	$3.72603946110722{\cdot}10^{-24}$
10^{-6}	$1.34922220938115{\cdot}10^{-6}$	$1.32348898008484{\cdot}10^{-23}$	$3.72603946110722{\cdot}10^{-24}$
10^{-9}	$1.36940553054800{\cdot}10^{-9}$	0	$2.93579661656743{\cdot}10^{-39}$
10^{-12}	$1.38989077859525{\cdot}10^{-12}$	0	$2.93579661656743{\cdot}10^{-39}$

Table 2: valori approssimati da newton, newton modificato e aitken(dati raccolti in es7.m)

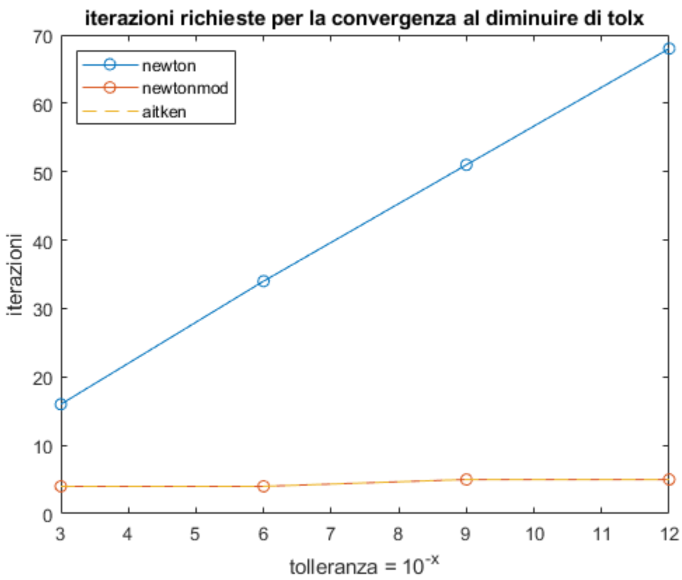


Figure 2: iterazioni richieste

Il metodo di newton classico perde la convergenza quadratica, essendo la radice cercata di molteplicità multipla. Il metodo di newton modificato e il metodo di aitken convergono molto più rapidamente e newton modificato riesce anche a trovare la radice esatta.

3 Capitolo 3

3.1 Esercizio 8

```
function [LU,p]=palu(A)
% [LU,p]=palu(A)
% funzione che dato in input matrice A restituisce matrice fattorizzata LU
% e il relativo vettore p di permutazione di LU con pivoting parziale di A
% input:
%   A= matrice di cui si vuole calcolare la fattorizzazione lu con pivoting
%   parziale
% output:
%   LU=matrice quadrata di dimensioni n*n, composta dalla matrice
%   triangolare superiore U e la matrice triangolare inferiore a diagonale
%   unitaria L
%   p= vettore di permutazione di dimensione n, generato dalla
%   fattorizzazione di A con pivoting parziale
%

[n,m]=size(A);
if(n~=m)
    error(matrice A non quadrata);
end
LU=A;
p=[1:n];
for i=1:n-1
    [mi,ki]=max(abs(LU(i:n,i)));
    if mi==0
        error('La matrice e'' non singolare')
    end
    ki=ki+i-1;
    if ki>i
        p([i ki])=p([ki i]);
       LU([i ki],:)= LU([ki i],:);
    end
    LU(i+1:n,i)=LU(i+1:n,i)/LU(i,i);
    LU(i+1:n,i+1:n)=LU(i+1:n,i+1:n)-LU(i+1:n,i)*LU(i,i+1:n);
end
return
end
```

3.2 Esercizio 9

```
function x=LUsolve(LU,p,b)
%
% funzione che risolve il sistema lineare LUx=b(p):
%input:
%   LU=matrice quadrata (n*n) fattorizzata LU, ottenuta attrarso la
%   fattorizzazione con pivoting parziale
%   p= vettore di permutazione per b, di dimensione n, con valori da (1 a
%   n)
%   b=vettore dei termini noti
%output:
%   x=vettore delle incognite calcolate
%
%
   [m,n]=size(LU);
```

```
    if(m~=n  ||  n~=length(b))  error('dati  incosistenti')
    else  if(min(abs(diag(LU)))==0)
                error(fattorizzazione errata);
        end
    end
     x=b(p);
     for  i=1:n-1
         x(i+1:n)=x(i+1:n)-(LU(i+1:n,i)*x(i));
     end
        x(n)=x(n)/LU(n,n);
        for  i=n-1:-1:1
            x(1:i)=x(1:i)-(LU(1:i,i+1)*x(i+1));
            x(i)=x(i)/LU(i,i);
        end
     return
end
```

3.3 Esercizio 10

i	Sigma	Norma
1	10^{-1}	8.9839e-15
2	10^{1}	1.4865e-14
3	10^{3}	1.3712e-12
4	10^{5}	1.2948e-10
5	10^{7}	5.3084e-09
6	10^{9}	1.0058e-06
7	10^{11}	8.5643e-05
8	10^{13}	0.0107
9	10^{15}	0.9814
10	10^{17}	4.1004e+03

Table 3: valori approssimati

Tabella che composta: i=indice dell'iterazione; Sigma=valore calcolato e usato dalla funzione linsis(), per introdurre un errore nella matrice generata A e nel suo vettore dei termini noti b, che cresce al crescere dell'interazione. Norma= valore della distanza tra il vettore x, soluzione del sistema lineare LU*x=b, il quale è affetto da errore, e il vettore xref, soluzione corretta del sistema. Da questa tabella quindi si può notare come all'incremento della interazione, e quindi della sigma, l'errore nelle soluzioni cresce, quasi proporzionalmente come sigma, con un fattore di 10^2;

3.4 Esercizio 11

```
function QR = myqr(A)
%   QR = myqr(A)
%   calcola la fattorizzazione QR di Householder della matrice A
%   Input:
%           A= matrice quadrata da fattorizzare
%
%   Output:
%           QR=matrice contenente le informazioni sui fattori Q e R della
%           fattorizzazione QR di A
%
    [m,n] = size(A);
    if  n > m
        error('Dimensioni errate');
    end
    QR = A;
```

```
    for i = 1:n
        alfa = norm(QR(i:m,i));
        if alfa == 0
            error('la matrice non ha rango massimo');
        end
        if QR(i,i) >= 0
            alfa = -alfa;
        end
        v1 = QR(i,i) -alfa;
        QR(i,i) = alfa;
        QR(i+1:m,i) = QR(i+1:m,i)/v1;
        beta = -v1/alfa;
        v = [1; QR(i+1:m,i)];
        QR(i:m,i+1:n) = QR(i:m,i+1:n) - (beta * v) * (v' * QR(i:m,i+1:n));
    end
end
```

3.5 Esercizio 12

```
function x = qrsolve(QR, b)
%
%
%   x = qrSolve(QR, b)
%   risolve il sistema QR*x=b nel senso dei minimi quadrati.
%   Input:
%           QR=matrice contenente le informazioni Q e R della
%           fattorizzazione di una matrice quadrata A
%           b=termine noto del sistema lineare
%   Output:
%           x=vettore delle soluzioni del sistema lineare
%

[m,n] = size(QR);
k = length(b);
if k ~= m
    error('Dati inconsistenti');
end
x=b(:);
for i = 1:n
    v=[1; QR(i+1:m,i)];
    beta = 2/(v'* v);
    x(i:m) = x(i:m) - beta*(v'*x(i:m))*v;
end
x=x(1:n);
for  j = n:-1:1
    if QR(j,j)==0
        error('Matrice singolare');
    end
    x(j) = x(j) / QR(j,j);
    x(1:j-1) = x(1:j-1) - QR(1:j-1,j)*x(j);
end
return
end
```

3.6 Esercizio 13

```
A= [1 ,  2 ,  3;  1  2  4;  3  4  5;  3  4  6;  5  6  7];
b=[14  17  26  29  38];
QR=myqr(A);
ris=qrsolve(QR,b);
disp(ris);
```

Il risultato finale è $ris = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$

3.7 Esercizio 14

A\b	(A'*A)\(A'*b)
1.0000	3.5759
2.0000	-3.4624
3.0000	9.5151
4.0000	-1.2974
5.0000	7.9574
6.0000	4.9125
7.0000	7.2378
8.0000	7.9765

Table 4: valori approssimati

L'espressione $A\backslash b$ risolve in matlab, il sistema di equazioni lineari nella forma matriciale A*x=b per x. L'espressione (A'*A) \ (A'*b) impiega lo stesso operatore \, quindi risolve il sistema delle equazioni lineare delle due parentesi. La Matrice A viene calcolata usando la funzione vander() che genera una matrice di tipo Vandermonde, la quale è mal condizionata. Usando la funzione cond() sulla matrice A si ottiene una condizionamento pari a: 1.5428e+09. Eseguendo poi la moltiplicazione della prima parentesi tonda, il condizionamento è pari a: 4.4897e+18. Andando cosi ad eseguire una divisione tra una matrice mal condizionata e un vettore, il risultato presenta degli errori.

4 Capitolo 4

4.1 Esercizio 15

Eseguendo il codice es15.m si ottengono i seguenti risultati:

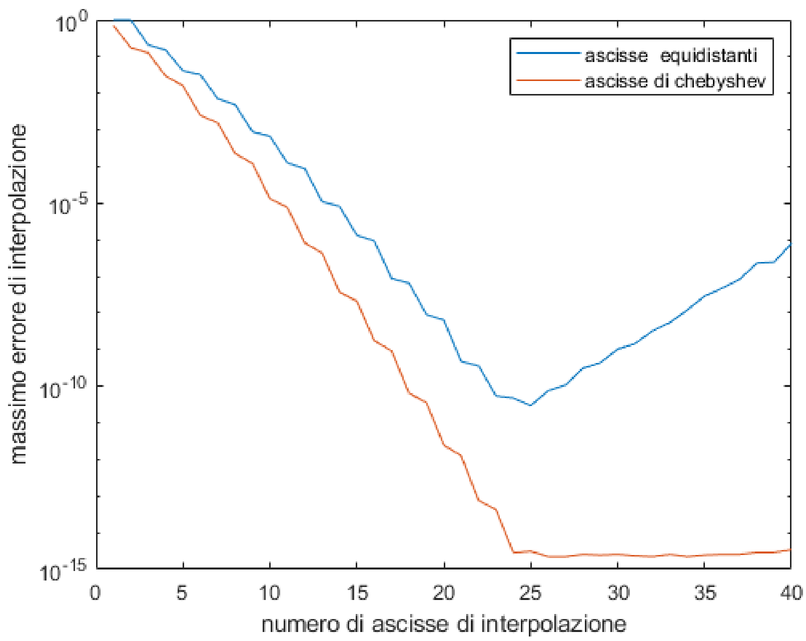


Figure 3: risultati interpolazione

Per le ascisse di chebyshev, si ha una decrescita esponenziale dell'errore massimo per $n \leq 25$ per poi assestarsi a circa $2 \cdot 10^{-15}$ per n successivi. Per quanto riguarda le ascisse equidistanti invece, si può notare come l'errore massimo torni a crescere esponenzialmente per $n > 25$. I risultati confermano il mal condizionamento del problema di interpolazione polinomiale quando vengono usate ascisse d'interpolazione equidistanti,

4.2 Esercizio 16

Eseguendo es16.m si ottiene:

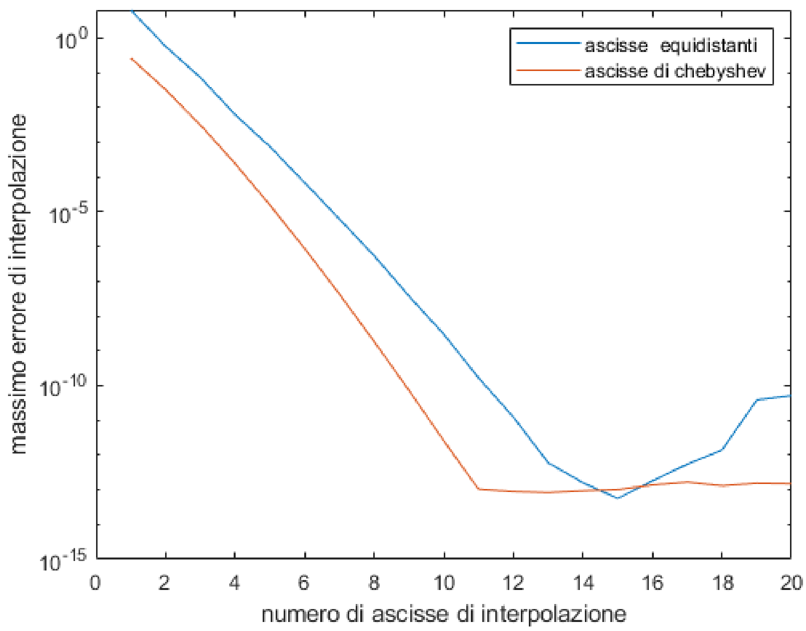


Figure 4: risultati interpolazione hermite

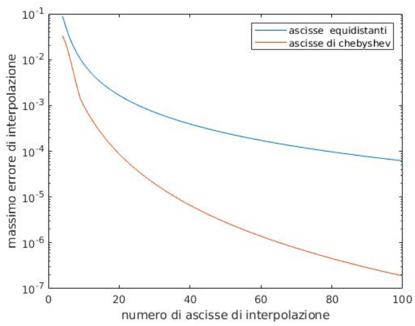
Si può notare come, rispetto all'interpolazione classica, l'errore decresca più rapidamente per $n \leq 15$. Anche in questo caso l'errore commesso usando le ascisse di chebyshev è migliore in confronto al caso delle ascisse equidistanti(eccetto per $n = 15$).

4.3 Esercizio 17

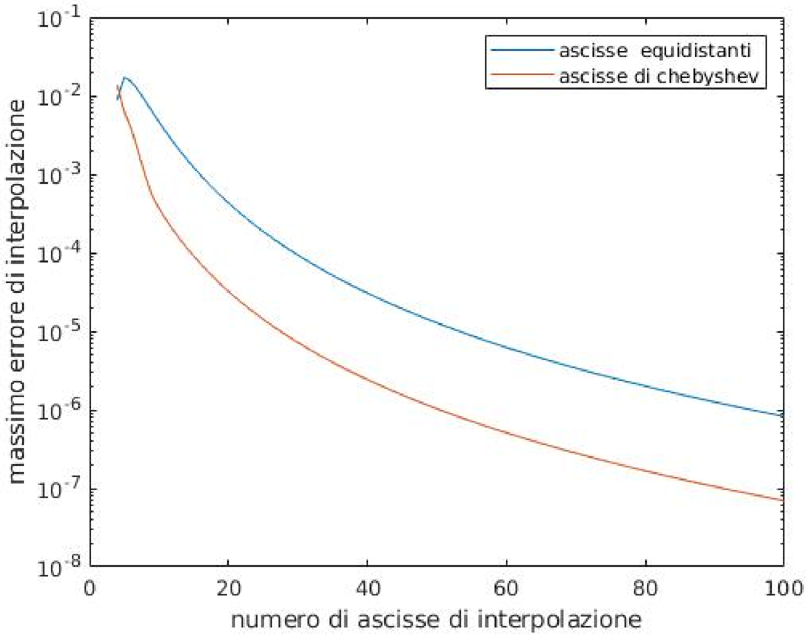
```
function output=splinenat(xi,fi,xq)
%
% output=splinenat(x,y,xq)
%funzione che calcola la spline cubica naturale.
%Input:
%   xi=vettore delle ascisse su cui calcolare la spline
%   fi=vettore dei valori di f(x), con x ascissa
%   xq= insieme delle ascisse di cui si vuole sapere il valore della spline
%Output:
%   output=vettore delle approssimazioni sulle ascisse xq
%

m = length(xi);
l=length(xq);

if m~=length(fi),
    error(dati errati);
end
for i = 1:m-1
    if any( find(xi(i+1:m)==xi(i)) ),
        error(ascisse non distinte),
    end
end
xi = xi(:);
```

```
fi = fi(:);
[xi,ind] = sort(xi);
fi = fi(ind);
% ordino le ascisse in modo crescente
hi = diff(xi);
n= m-1;
df = diff(fi)./hi;
hh = hi(1:n-1)+hi(2:n);
rhs = 6*diff(df)./hh;
phi = hi(1:n-1)./hh;
csi = hi(2:n)./hh;
% = 1-phi;
d= 2*ones(n-1,1);
phi = phi(2:n-1);
csi = csi(1:n-2);
mi = trisolve( phi, d, csi, rhs );
mi = [0;mi;0];
r=fi(1:n)-((hi(1:n).^2)/6).*mi(1:n);
q=df(1:n)-((hi(1:n)/6).*(mi(2:m)-mi(1:n)));
output=zeros(l,1);
for i=1:l
    indp=find(xq(i)>=xi(1:n),1,'last');
    indg=indp+1;
    output(i)=((((xq(i)-xi(indp)).^3).*mi(indg)+((xi(indg)-xq(i)).^3).*mi(
        indp))/(6*hi(indp)))+q(indp).*(xq(i)-xi(indp))+r(indp);
end
return
end
```

4.4 Esercizio 18



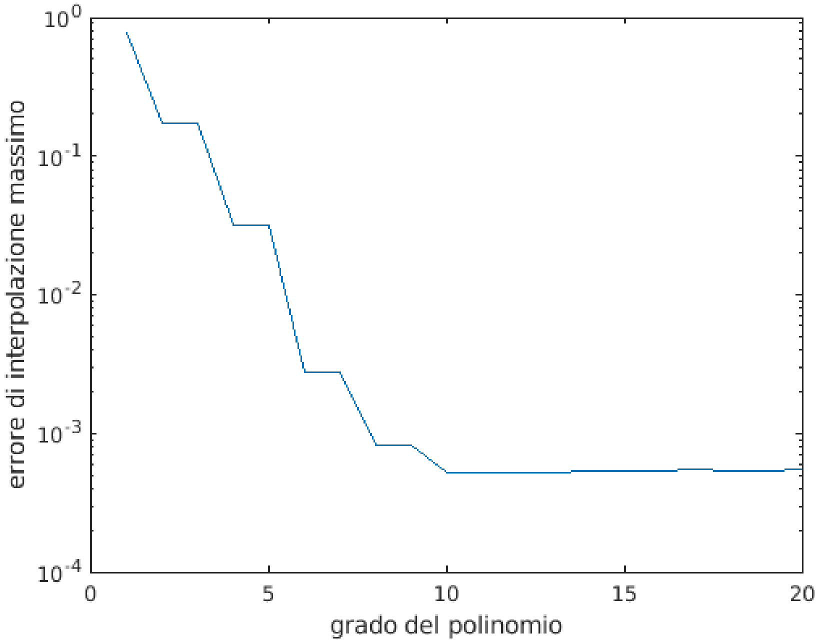
4.5 Esercizio 19



4.6 Esercizio 20

```
function y = minimiquadrati(xi, fi, m)
%
%
% y = minimiquadrati(xi, fi, m)
% calcola il valore del polinomio di approssimazione ai minimi quadrati di
    grado m
% sulle ascisse xi. fi contiene i valori approssimati di una funzione f
    valutata su xi
if length(unique(xi)) < m+1
    error('ascisse distinte non sufficienti');
end
fi = fi(:);
V = fliplr(vander(xi));
V = V(1:end, 1:m+1);
QR = myqr(V);
p = qrsolve(QR, fi);
y = p(m+1)*ones(size(xi));
for i = 0:m-1
    y = y.*xi+p(m-i);
end
end
```

Eseguendo es20.m si ottiene:



Si nota una decrescita delll'errore esponenziale fino a $m = 10$, dove si assesta tra 10^{-3} e 10^{-4}.

5 Capitolo 5

5.1 Esercizio 21

```
function c = ncweights(n)
%
%
% c = nc-weights(n)
% calcola i pesi della formula di newton cotes di grado n;
%
if n<=0
    error('grado della formula non positivo');
end
c=zeros(1,floor(n / 2 + 1));
for j = 1:(ceil((n+1)/2))
    temp = (0:n);
    temp(j)=[];
    f = @(x)(prod(x-temp) /prod(j-1-temp));
    c(j) = integral(f, 0, n, 'ArrayValued', true);
end
c = [c flip(c)]; %sfrutto la simmetria dei pesi
if mod(n,2)==0
    %elimino la copia del valore centrale prodotta da flip(c) e che risulta di troppo per n pari
    c(n/2+1) = [];
end
return
end
```

Eseguendo lo script es21.m si ottiene:

n $\backslash c_{in}$	0	1	2	3	4	5	6	7
1	$\frac{1}{2}$	$\frac{1}{2}$						
2	$\frac{1}{3}$	$\frac{4}{3}$	$\frac{1}{3}$					
3	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{3}{8}$				
4	$\frac{14}{45}$	$\frac{64}{45}$	$\frac{8}{15}$	$\frac{64}{45}$	$\frac{14}{45}$			
5	$\frac{95}{288}$	$\frac{95}{288}$	$\frac{95}{288}$	$\frac{95}{288}$	$\frac{95}{288}$	$\frac{95}{288}$		
6	$\frac{41}{140}$	$\frac{54}{35}$	$\frac{27}{140}$	$\frac{68}{35}$	$\frac{27}{140}$	$\frac{54}{35}$	$\frac{41}{140}$	
7	$\frac{108}{355}$	$\frac{810}{559}$	$\frac{343}{640}$	$\frac{649}{536}$	$\frac{649}{536}$	$\frac{343}{640}$	$\frac{810}{559}$	$\frac{108}{355}$

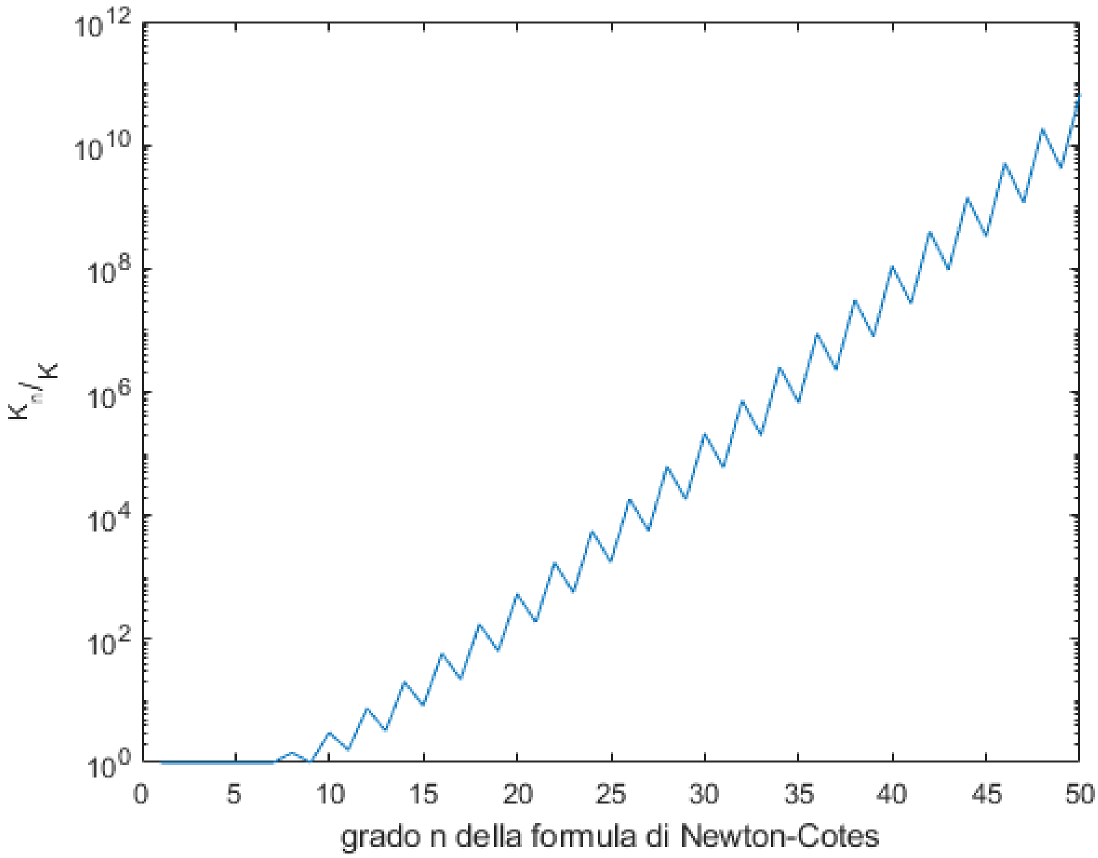
Table 5: pesi della formula di Newton-Cotes fino al settimo grado

5.2 Esercizio 22

Sappiamo che $k = (b-a)$ e $k_n = (b-a)\frac{1}{n}\sum_{i=0}^{n}|c_{in}|$. Il rapporto sarà dunque dato da:

$$\frac{k_n}{k} = \frac{(b-a)\frac{1}{n}\sum_{i=0}^{n}|c_{in}|}{b-a} = \frac{1}{n}\sum_{i=0}^{n}|c_{in}|$$

Calcolando $\frac{k_n}{k}$ per $n = 1, \ldots\ldots, 50$(es22.m) si ottiene:



5.3 Esercizio 23

```
function y = newtoncotes(f,a, b, n)
%
% y= newtoncotes(f,a,b, n)
% calcola l'approssimazione dell'integrale definito per la funzione f sull'
    intervallo [a, b],
% utilizzando la formula di newton cotes di grado n.
%

if a > b || n < 0
    error('dati inconsistenti');
end
xi = linspace(a, b, n+1);
fi = feval(f, xi);
h = (b-a) / n;
c = ncweights(n);
y = h*sum(fi.*c);
return
end
```

RISULTATI PER N DA 1 A 9(es23.m):

grado della formula	valore integrale	errore
1	$4,28 \cdot 10^{-1}$	$2,53 \cdot 10^{-1}$
2	$2,13 \cdot 10^{-1}$	$3,8 \cdot 10^{-2}$
3	$1,96 \cdot 10^{-1}$	$2,1 \cdot 10^{-2}$
4	$1,80 \cdot 10^{-1}$	$5 \cdot 10^{-3}$
5	$1,79 \cdot 10^{-1}$	$4 \cdot 10^{-3}$
6	$1,76 \cdot 10^{-1}$	$1 \cdot 10^{-3}$
7	$1,76 \cdot 10^{-1}$	$1 \cdot 10^{-3}$
8	$1,75 \cdot 10^{-1}$	0
9	$1,75 \cdot 10^{-1}$	0

5.4 Esercizio 24

```
function I = trapecomp(f, a, b, n)
%
%
% I = trapecomp(f, a, b)
%
% Approssimazione dell'integrale definito di f(x) con estremi a e b,
% mediante la formula composita dei trapezi su n+1 ascisse equidistanti
if a==b
    I=0;
elseif n < 1 || n~=fix(n)
    error('numero di ascisse non valido');
else
    h=(b-a)/n;
    x=linspace(a, b, n+1);
    f = feval(f, x);
    I = h*(f(1)/2 + sum(f(2:n)) + f(n+1)/2);
end
return
end
```

```
function I = simpcomp(f, a, b, n)
    %myFun - Description
    %
    % I = simpcomp(f, a, b)
    %
    % Approssimazione dell'integrale definito di f(x) con estremi a e b,
    % mediante la formula composita di Simpson su n+1 ascisse equidistanti(
       n pari)
    if a==b
        I=0;
    elseif n < 2 || n/2 ~= fix(n/2)
        error('numero di ascisse non valido');
    else
        h=(b-a)/n;
        x=linspace(a, b, n+1);
        f = feval(f, x);
        I = (h/3) * (f(1) + f(n+1) + 4*sum(f(2:2:n)) + 2*sum(f(3:2:n-1)));
    end
    return
    end
```

Approssimando $\int_{-1}^{1.1} tan(x)dx$ con le due formule si ottiene:

intervalli\formula	trapezi composita	simpson composita
2	0.266403558406035	0.266403558406035
4	0.203432804450016	0.182442553131343
6	0.188498346613972	0.177333443886033
8	0.182789408875225	0.175908277016961
10	0.180034803521960	0.175392868382289
12	0.178504015707472	0.175172572071972
14	0.177568218195411	0.175066546519247
16	0.176955413111201	0.175010747856527
18	0.176532709616469	0.174979254439942
20	0.176229037552030	0.174960448895386

Table 6: risultati di es24.m

La formula composita di simpson converge più rapidamente ed è più precisa rispetto alla formula dei trapezi

5.5 Esercizio 25

```
function [I2, points] = adaptrap(f, a, b, tol, fa, fb)
%
%
% Syntax: [I2, points] = adaptrap(f, a, b, tol, fa, fb)
%
%
global points
delta = 0.5;
if nargin<=4
    fa = feval(f, a );
    fb = feval(f, b );
    if nargout==2
        points = [a fa; b fb];
    else
        points = [];
```

```
    end
end
h = b-a;
x1 = (a+b)/2;
f1 = feval(f, x1);
if ~isempty(points)
    points = [points; [x1 f1]];
end
I1 = .5*h*(fa+fb);
I2 = .5*(I1+h*f1);
e = abs(I2-I1)/3;
if e>tol || abs(b-a) > delta
    I2 = adaptrap(f, a, x1, tol/2, fa, f1) + adaptrap(f, x1, b, tol/2, f1,
        fb);
end
return
end
```

```
function [I2, points] = adapsim(f, a, b, tol, fa, f1, fb)
% [I2, points] = adapsim(f, a, b, tol, fa, f1, fb)
% Approssimazione dell'integrale definito di f(x) con estremi a e b,
% mediante la formula adattiva di simpson
global points
delta = 0.5;
x1 = (a+b)/2;
if nargin<=4
    fa = feval(f, a );
    fb = feval(f, b );
    f1 = feval(f, x1);
    if nargout==2
        points = [a fa;x1 f1; b fb];
    else
        points = [];
    end
end
h = (b-a)/6;
x2 = (a+x1)/2;
x3 = (x1+b)/2;
f2 = feval(f, x2);
f3 = feval(f, x3);
if ~isempty(points)
    points = [points; [x2 f2; x3 f3]];
end
I1 = h*(fa+fb+4*f1);
I2 = .5*h*(fa+4*f2+2*f1+4*f3+fb);
e = abs(I2-I1)/15;
if e>tol || abs(b-a) > delta
    I2 = adapsim(f, a, x1, tol/2, fa, f2, f1) + adapsim(f, x1, b, tol/2, f1
        , f3, fb);
end
return
end
```

Approssimando $\int_{-1}^{1} \frac{1}{1+10^2x^2}dx$ con le due formule si ottiene:

tolleranza\formula	trapezi adattiva	simpson adattiva
10^{-2}	0.295559711784128, punti = 21	0.281297643062670, punti = 17
10^{-3}	0.294585368185034 , punti = 93	0.281297643062670, punti = 17
10^{-4}	0.294274200873635, punti = 277	0.294259338419631, punti = 41
10^{-5}	0.294230142164878, punti = 793	0.294227809768005, punti = 81
10^{-6}	0.294226019603178, punti = 2692	0.294225764620384 , punti = 145

Table 7: risultati di es25.m

Per ciascuna formula, l'operazione che comporta maggior costo computazionale ad ogni chiamata è la valutazione funzionale dei punti di un sottointervallo. Poichè ogni punto viene valutato una sola volta, possiamo confrontare il costo delle due formule andando a vedere quanti punti aggiuntivi sono stati utilizzati. Osservando i dati riportati nella tabella **??**, è palese come la formula di simpson adattiva convergà più rapidamente rispetto alla formula dei trapezi adattiva.

6 Codici ausiliari

6.1 Esercizio 6

Listing 1: es6.m

```
f = @(x)(x-cos(x));
f1 = @(x)(1+sin(x));

x0 = 0;
x1 = 1;
x=zeros(4,4);
y= zeros(4, 4);
for i=3:3:12

    [x(1, i/3), y(1, i/3)] = bisezione(f, x0, x1, 10^(-i));
    [x(2, i/3), y(2, i/3)] = newton(f, f1, x0, 10^(-i));
    [x(3, i/3), y(3, i/3)] = corde(f, f1, x0, 10^(-i));
    [x(4,i/3), y(4, i/3)] = secanti(f, x0, x1, 10^(-i), 100);
end
row_names = {'bisezione', 'newton', 'corde', 'secanti'};
colnames = {'10^-3','10^-6','10^-9','10^-12'};
values = array2table(x,'RowNames',row_names,'VariableNames',colnames);
disp(values)
figure
plot([3, 6, 9, 12], y','o-')
title('iterazioni richieste per la convergenza al diminuire di tolx')
xlabel('tolleranza = 10^{-x}')
ylabel('iterazioni')
legend({'bisezione','newton','corde','secanti'},'Location','northwest')
```

6.2 Esercizio 7

Listing 2: es7.m

```
f = @(x)(x^2*tan(x));
f1 = @(x)(2*x*tan(x) +(x^2)/(cos(x)^2));
m = 3;
x0 = 1;
y= zeros(3, 4);
x=-1*ones(3,4);
for i=3:3:12
   [x(1, i/3), y(1, i/3)] = newton(f, f1, x0, 10^(-i));
   [x(2, i/3), y(2, i/3)] = newtonmod(f, f1, x0, m, 10^(-i));
   [x(3, i/3), y(3, i/3)] = aitken(f, f1, x0, 10^(-i));
end
disp(x);
disp(y);
row_names = {'newton', 'newton modificato', 'aitken'};
colnames = {'10^-3','10^-6','10^-9','10^-12'};
values = array2table(x,'RowNames',row_names,'VariableNames',colnames)

format
iterations = array2table(y,'RowNames',row_names,'VariableNames',colnames)
plot([3, 6, 9, 12], y(1,1:end)','-o');
hold on;
plot([3, 6, 9, 12], y(2,1:end)','-o');
plot([3, 6, 9, 12], y(3,1:end)','--')
```

```
title('iterazioni richieste per la convergenza al diminuire di tolx')
xlabel('tolleranza = 10^{-x}')
ylabel('iterazioni')
legend({'newton','newtonmod','aitken'},'Location','northwest')
```

6.3 Esercizio 15

Listing 3: es15.m

```
f = @(x)(cos((pi*x.^2)/2));
x = linspace(-1, 1, 100001);
linerrors = zeros(1, 40);
chebyerrors = zeros(1, 40);
for n = 1:40
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    ylin = lagrange(xlin,f(xlin),x);
    ycheby = lagrange(xcheby,f(xcheby),x);
    linerrors(n) = norm(abs(f(x) - ylin), inf);
    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
end
semilogy(linerrors);
hold on;
semilogy(chebyerrors);
xlabel('numero di ascisse di interpolazione');
ylabel('massimo errore di interpolazione');
legend({'ascisse  equidistanti', 'ascisse di chebyshev'},'Location','
    northeast');
```

6.4 Esercizio 16

Listing 4: es16.m

```
f = @(x)(cos((pi*x.^2)/2));
f1 = @(x)(-pi*x.*sin((pi*x.^2)/2));
x = linspace(-1, 1, 100001);
linerrors = zeros(1, 20);
chebyerrors = zeros(1, 20);
for n = 1:20
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1, n+1);
    ylin = hermite(xlin,f(xlin),f1(xlin),x);
    ycheby = hermite(xcheby,f(xcheby),f1(xcheby),x);
    linerrors(n) = norm(abs(f(x) - ylin), inf);
    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
end
semilogy(linerrors);
hold on;
semilogy(chebyerrors);
xlabel('numero di ascisse di interpolazione');
ylabel('massimo errore di interpolazione');
legend({'ascisse  equidistanti', 'ascisse di chebyshev'},'Location','
    northeast');
```

6.5 Esercizio 18

Listing 5: es18.m

```
f = @(x)(cos((pi*(x.^2))/2));
x = linspace(-1, 1, 100001);
linerrors = zeros(1, 40);
chebyerrors = zeros(1, 40);
for n = 4:100
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    xcheby(1)=-1;
    xcheby(n+1)=1;
    ylin = splinenat(xlin,f(xlin),x);
    ycheby = splinenat(xcheby,f(xcheby),x);
    ylin=ylin';
    ycheby=ycheby';
    linerrors(n) = norm(abs(f(x) - ylin), inf);
    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
end
semilogy(linerrors);
hold on;
semilogy(chebyerrors);
xlabel('numero di ascisse di interpolazione');
ylabel('massimo errore di interpolazione');
legend({'ascisse  equidistanti', 'ascisse di chebyshev'},'Location','
    northeast');
```

6.6 Esercizio 19

Listing 6: es19.m

```
f = @(x)(cos((pi*(x.^2))/2));
x = linspace(-1, 1, 100001);
linerrors = zeros(1, 40);
chebyerrors = zeros(1, 40);
for n = 4:100
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    ylin = spline(xlin,f(xlin),x);
    ycheby = spline(xcheby,f(xcheby),x);
    linerrors(n) = norm(abs(f(x) - ylin), inf);
    chebyerrors(n) = norm( abs(f(x) - ycheby), inf);
end
semilogy(linerrors);
hold on;
semilogy(chebyerrors);
xlabel('numero di ascisse di interpolazione');
ylabel('massimo errore di interpolazione');
legend({'ascisse  equidistanti', 'ascisse di chebyshev'},'Location','
    northeast');
```

6.7 Esercizio 20

Listing 7: es20.m

```
f = @(x)(cos((pi*x.^2)/2));
fp = @(x)(f(x) + 10^(-3)*rand(size(x)));
xi = -1 + 2*(0:10^4)/10^4;
fi = f(xi);
```

```
fpi = fp(xi);
errors=zeros(1, 20);
for m = 1:20
    y = minimiquadrati(xi, fpi, m);
    errors(m) = norm(abs(y-fi), inf);
end
semilogy(errors);
xlabel('grado del polinomio');
ylabel('errore di interpolazione massimo');
```

6.8 Esercizio 21

Listing 8: es21.m

```
for i = 1:7
    weights= rats(ncweights(i))
end
```

6.9 Esercizio 22

Listing 9: es22.m

```
rapp = zeros(1, 50);
for i = 1:50
    rapp(i) = sum(abs(ncweights(i)))/i;
end
semilogy(rapp);
xlabel('grado n della formula di Newton-Cotes');
ylabel('^{K_n}/_{K}');
```

6.10 Esercizio 23

Listing 10: es23.m

```
value = log(cos(1)/cos(1.1));
x = zeros(1,9);
errors=zeros(1, 9);
for i = 1:9
    x(i) = newtoncotes(@tan, -1,1.1, i);
    errors(i) = abs(value-x(i));
end
```

6.11 Esercizio 24

Listing 11: es24.m

```
a = -1;
b = 1.1;
n = 10;
itrap = zeros(1, n);
isimp = zeros(1, n);
for i = 1:n
    itrap(i) = trapecomp(@tan, a, b, i*2);
    isimp(i) = simpcomp(@tan, a, b, i*2);
end
integrali = [itrap; isimp];
```

```
row_names = {'trapezi composta', 'simpson composta'};
colnames = {'2','4','6','8','10','12','14','16','18','20'};
values = array2table(integrali,'RowNames',row_names,'VariableNames',
    colnames);
disp(values);
```

6.12 Esercizio 25

Listing 12: es25.m

```
format long e
f = @(x)(1/(1+100*x.^2));
a = -1;
b = 1;
itrap = zeros(1, 5);
trap_points = zeros(1, 5);
isimp = zeros(1, 5);
simp_points = zeros(1, 5);
for i = 1:5
    [itrap(i), points] = adaptrap(f, a, b, 10^(-i-1));
    trap_points(i) =  length(points);
    [isimp(i), points] = adapsim(f, a, b, 10^(-i-1));
    simp_points(i) = length(points);
end
integrali = [itrap; isimp];
npoints = [trap_points; simp_points];
row_names = {'trapezi adattiva', 'simpson adattiva'};
colnames = {'10^-2','10^-3','10^-4','10^-5','10^-6'};
values = array2table(integrali,'RowNames',row_names,'VariableNames',
    colnames);
npoints = array2table(npoints,'RowNames',row_names,'VariableNames',colnames
    );
disp(values);
format
disp(npoints);
```