Elaborato Calcolo numerico
Anno scolastico 2019/2020

Autore: Massimo Hong
Matricola:6365472

September 6, 2020

Contents

1 Esercizio1 **3**

2 Esercizio2 **4**

3 Esercizio3 **5**

4 Esercizio 4 **6**

5 Esercizio 5 **7**

6 Esercizio 6 **10**

7 Esercizio 7 **11**

8 Esercizio8 **13**

9 Esercizio9 **14**

10 Esercizio 10 **15**

11 Esercizio 11 **16**

12 Esercizio 12 **17**

13 Esercizio 13 **18**

14 Esercizio 14 **19**

15 Esercizio 15 **20**

16 Esercizio 16 **23**

17 Esercizio17 **25**

18 Esercizio 18 **26**

19 Esercizio19 **27**

20 Esercizio 20 **28**

21 Esercizio 21 **29**

22 Esercizio22 **30**

23 Esercizio 23 **31**

24 Esercizio 24 **32**

25 Esercizio 25 **34**

1 Esercizio1

Verificare che, per h sufficientemente piccolo, si ha

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

Per la dimostrazione utilizziamo il polinomio di taylor di grado n centrato in x_0 :

$$P_n(x;x_0) = \sum_{k=0}^{n} \frac{(x-x_0)^k}{k!} f^{(k)}(x_0)$$
$$R_n(x;x_0) = O(x-x_0)^{(n+1)}$$

Per cui possiamo calcolare $f(x+h)$ e $f(x-h)$ sviluppando il polinomio di Taylor centrato in x:

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

Sostituendo all'equazione iniziale otteniamo:

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

2 Esercizio2

Spiegare cosa calcola il seguente script Matlab: u = 1; while 1, if 1+u==1, break, end, u = u/2; end, u

Teoricamente questo script non dovrebbe mai terminare poichè dividendo per 2 il valore u, non si dovrebbe mai raggiungere lo 0, tuttavia il seguente script termina. Più precisamente, alla fine dell'esecuzione risulta che $u = 1.1102 \cdot 10^{-}16$, che è minore del valore della precisione di macchina $d = eps = 2.2204 \cdot 10^{-}16$(la metà), e quando andiamo a sommare ad un numero n un valore u<eps, n+u viene approssimato a n.
Un possibile utilizzo di questo codice è il calcolo della precisione di macchina.

3 Esercizio3

1. a = 1e20; b = 100; a-a+b
2. a = 1e20; b = 100; a+b-a

Le seguenti operazioni eseguono (algebricamente) la stessa operazione :

1. a= 1e20; b = 100; a-a+b
   Questo script restituisce il valore 100, in quanto $a - a = 0$ e $0 + 100 = 100$.
2. a= 1e20; b = 100; a+b-a
   Matlab ha il valore $eps = 2.2204 \cdot 10^{-}16$, che corrisponde a 16 cifre di precisione (a ha 20 cifre significative), per cui quando andiamo ad affettuare $a + b$, otterremo un valore che differisce da a solo per le ultime 2 cifre (che risultano essere fuori dall'intervallo rappresentabile) e di conseguenza viene approssimato ad a. Per cui l'espressione $a + b - a$ in Matlab è equivalente a $a - a$.

4 Esercizio 4

```
function xn = radn(x,n)
 %
 % xn=radn(n.x)
 % funzione Matlab che implementa il metodo di newton per il calcolo della
 % radice nesima di un numero positivo x
 % xn  valore in output che rappresenta la radice n-esima
 %
 format long e
 imax = 1000;
 tolx = eps;
 if x<0
     error('imposibile calcolare la radice di un numero negativo');
 end
 x0 = 1;
 xn = ((n-1)*x0+x/x0^(n-1))/n;

 i=1;
 while(i<imax) && (abs(xn-x0)>tolx)
     i=i+1;
     x0 = xn;
     xn = ((n-1)*x0+x/x0^(n-1))/n;
 end

 if abs(xn-x0)>tolx
     error('il metodo non converge');
 end
```

5 Esercizio 5

- Metodo di bisezione

```
function [x,i]= bisezione(f,a,b,tolx)
%
%   [x,i] bisezione(f,a,b,tolx)
%   metodo di bisezione per calcolare gli zeri di una funzione f (continua)
%   [a,b] intervallo interessato
%   x approssimazione della radice
%   i iterazioni necessarie
%   tolx tolleranza
%
    format long e
    fa = feval(f,a);
    fb = feval(f,b);
    if(fa*fb>0)
        error('dati in input invalidi');
    end
    imax = ceil(log2(b-a)-log2(tolx));

    for i= 1:imax
        x = (a+b)/2;
        fx = feval(f,x);

        if(abs(x-a)<= tolx*(1+abs(a)))
            break
        end

        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)
%
% x = newton(f,a,b,tolx) metodo per approssimazione tramite newton
% f funzione continua input
% f1 derivata prima della funzione f
% x0 ascissa di partenza
% tolx tolleranza
% x approssimazione della radice
% i iterazioni
%
    format long e;
    imax = 1000;
    x = x0;
    for i=1:imax
        fx = feval(f,x);
        f1x = feval(f1,x);
        x0 = x;
```

```
        x = x0-fx/f1x;
        if(abs(x-x0)<= tolx*(1+abs(x0)))
                break
        end
    end
    if(abs(x-x0)>tolx*(1+abs(x0)))
        error('il metodo non converge');
    end
end
```

- Metodo delle secanti

```
function [x,i] = secanti(f,x0,x1,tolx)
%
% x = secanti5(f,xprec,xi,tolx) metodo delle secanti per approssimazione
% della radice
% f funzione continua input
% f1 derivata prima della funzione f
% x0 ascissa di partenza
% x1 ascissa successiva a x0
% tolx tolleranza
% x approssimazione della radice
% i iterazioni
%
    format long e;
    fx0 = feval(f,x0);
    imax = 1000;
    for i=1:imax
        fx1 = feval(f,x1);
        dfx1 = (fx1-fx0)/(x1-x0);
        x = x1 -fx1/dfx1;
        if(abs(x-x1)<= tolx*(1+abs(x1)))
            break;
        end
        x0 = x1;
        x1 = x;
        fx0 = fx1;
    end
    if (abs(x-x1)>tolx*(1+abs(x1)))
        error('il metodo non converge');
    end
end
```

- Metodo delle corde

```
function [x,i] = corde(f,f1,x0,tolx)
%
% function [x,i] = corde(f,f1,x0,tolx)
% Calcolo della radice di un numero utilizzando
% il metodo delle corde
% f funzione continua input
% f1 derivata prima della funzione f
% x0 ascissa di partenza
% tolx tolleranza
% x approssimazione della radice
% i iterazioni
%
    format long e;
    if(tolx < eps)
```

```
15          error(tolleranza non idonea);
16      end
17
18      f1x = feval(f1,x0);
19      x=x0;
20      maxit = 1000;
21      for i = 1:maxit
22              fx = feval(f,x);
23              if fx==0
24                      break;
25              end
26              x = x - fx/f1x;
27              if abs(x-x0)<=tolx*(1+abs(x0))
28                      break;
29              else
30                      x0 = x;
31              end
32      end
33
34      if abs(x-x0) > tolx*(1+abs(x0))
35         error('metodo non converge');
36      end
37  end
```

6 Esercizio 6

Dai risultati ottenuti possiamo affermare che il metodo di Newton sia il più efficiente (in termini di iterazioni), seguito dal metodo delle secanti e della bisezione (che ha una crescita lineare) e infine, dal metodo delle corde. Per quanto riguarda il costo computazionale, l'operazione che è decisamente più costosa è la valutazione funzionale delle ascisse tramite feval.

- Bisezione
  Per il metodo della bisezione la funzione è eseguita : una volta all'interno del ciclo e due volte fuori. Avremo quindi: 2+ n.iterazioni.
- Newton
  Per il metodo di Newton la funzione è eseguita due volte nel ciclo : 2 * n.iterazioni.
- corde
  Per il metodo delle corde funzione è eseguita una volta dentro il ciclo e una fuori : 1 + n.iterazioni.
- secanti
  Per il metodo delle secanti la funzione è eseguita una volta dentro il ciclo e una fuori : 1 + n.iterazioni.

Per tutte e 4 le funzioni, si ha un costo lineare in n.

Tolleranza	Iterazioni	Newton
10^{-3}	4	7.390851333852840e-01
10^{-6}	5	7.390851332151607e-01
10^{-9}	5	7.390851332151607e-01
10^{-12}	6	7.390851332151607e-01

Tolleranza	Iterazioni	Secanti
10^{-3}	4	7.390851121274639e-01
10^{-6}	5	7.390851332150012e-01
10^{-9}	6	7.390851332151607e-01
10^{-12}	6	7.390851332151607e-01

Tolleranza	Iterazioni	Bisezione
10^{-3}	10	7.392578125000000e-01
10^{-6}	20	7.390851974487305e-01
10^{-9}	30	7.390851331874728e-01
10^{-12}	40	7.390851332156672e-01

Tolleranza	Iterazioni	Corde
10^{-3}	17	7.395672022122561e-01
10^{-6}	34	7.390845495752126e-01
10^{-9}	52	7.390851327392538e-01
10^{-12}	69	7.390851332157368e-01

7 Esercizio 7

La molteplicità m di f(x) = $x^2 tan(x)$ è m=3. Sostituendo a x il valore zero otteniamo: $(0)^2 * tan(0)$; 0 annulla due volte il primo termine del prodotto mentre annulla una volta il secondo termine, in quanto tan(0)=0;

- Metodo di newtonModificato

```
function [x,i]= newtonModificato(f,f1,x0,m,tolx)
%
% x = newtonModificato(f,a,b,tolx) metodo per approssimazione tramite newton
% f funziona continua passata come parametro in input
% f1 derivata prima di f1
% m molteplicita' della radice
% x0 punto di parteza
% tolx tolleranza
% x valore in uscita che rappresenta l'approssimazione della radice
% i numero di iterazioni
%
    format long e;
    imax = 1000;
    x = x0;
    for i=1:imax
        fx = feval(f,x);
        if(fx == 0)
            break;
        end
        f1x = feval(f1,x);
        x0 = x;
        x = x0-m*(fx/f1x);
        if(abs(x-x0)<= tolx*(1+abs(x0)))
                break
        end
    end
    if(abs(x-x0)>tolx*(1+abs(x0)))
        error('il metodo non converge');
    end
end
```

- Metodo di Aitken

```
function [x,i]= aitken(f,f1,x0,tolx)
%
% x = newton(f,a,b,tolx) metodo per approssimazione tramite newton
% f funziona continua passata come parametro in input
% f1 derivata prima di f1
% x0 punto di parteza
% tolx tolleranza
% x valore in uscita che rappresenta l'approssimazione della radice
%
%
    format long e;
    imax = 1000;
    x = x0;
    for i=1:imax
        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*(1+abs(x0)))
                break
        end
    end
    if(abs(x-x0)>tolx*(1+abs(x0)))
        error('il metodo non converge');
    end
end
```

Tolleranza	Iterazioni	Newton
10^{-3}	16	1.994002961956096e-03
10^{-6}	34	1.349222209381150e-06
10^{-9}	51	1.369405530548002e-09
10^{-12}	68	1.389890778595252e-12

Tolleranza	Iterazioni	Newton Modificato
10^{-3}	4	6.617444900424221e-24
10^{-6}	4	6.617444900424221e-24
10^{-9}	5	0
10^{-12}	5	0

Tolleranza	Iterazioni	Newton Modificato
10^{-3}	3	-1.570796335324655e+00
10^{-6}	4	-1.570796356741072e+00
10^{-9}	148	-1.570796314458764e+00
10^{-12}	Il metodo non converge	Il metodo non converge

Il metodo di Aitken non è corretto, per cui non è stato preso in considerazione per i paragoni. Possiamo notare che il metodo di newton modificato è più efficiente rispetto al metodo di newton normale, che inizia a "faticare" quando aumenta la molteplicità della radice.

8 Esercizio8

```
function [LU,p] = palu(A)
%
%function [LU,p] = paly(A)
%Fattorizzazione LU con pivoting parziale di una matrice
%A non singolare data in input
% Input:
%   -A: matrice non singolare da fattorizzare
%Output:
%   -LU matrice che contiene informazioni dei fattori L e U
%   -p : vettore di permutazione
%
    [m,n] = size(A);
    if(n ~= m)
        error(la matrice A non è quadrata);
    end
    p=[1:n];
    LU = A;
    for i = 1 : (n-1)
       [mi, ki] = max(abs(LU(i:n, i)));
       if (mi == 0)
           error(La matrice non è singolare);
       end
       ki = ki+i-1;
       if(ki>i)
           LU([i,ki],:) = LU([ki,i], :);
           p([i,ki]) = p([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
```

9 Esercizio9

```
function x = lusolve(LU,p,b)
%
%function x = lusolve(LU,p,b)
%risoluzione sistema lineare lux=b(p)
%Input:
%   -LU matrice ottenuta tramite function palu
%   -p: vettore di permutazione ottenuta tramite function palu
%   -b: vettore termini noti del sistema Ax = b
%Output:
%   -x: soluzione del sistema lineare lux = b(p)

    [m,n] = size(LU);
    if(m~=n || n~=length(b))
        error(dati inconsistenti);
    elseif ( min(abs(diag(LU))) == 0 )
        error(fattorizzazione errata);
    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
```

10 Esercizio 10

- linsis

```
function [A,b] = linsis(n,k,simme)
%
%  [A,b] = linsis(n,k,simme)  Crea una matrice A nxn ed un termine noto b,
%                             in modo che la soluzione del sistema lineare
%                             A*x=b sia x = [1,2,...,n]^T.
%                             k non ve lo dico a cosa serve.
%                             simme, se specificato, crea una matrice
%                             simmetrica e definita positiva.
%
    sigma = 10^(-2*(1-k))/n;
    rng(0);
    [q1,r1] = qr(rand(n));
    if nargin==3
        q2 = q1';
    else
        [q2,r1] = qr(rand(n));
    end
    A = q1*diag([sigma 2/n:1/n:1])*q2;
    x = [1:n]';
    b = A*x;
    disp(sigma);
    return
end
```

- Script

```
n = 10;
xref = (1:10)';
for i = 1:10
[A,b] = linsis(n,i);
[LU,p] = palu(A);
x = lusolve(LU,p,b);
disp(norm(x-xref))
end
```

Iterazione	Sigma	Norma
1	$0.1000 = 10^{-1}$	8.9839e-15
2	10	1.4865e-14
3	$1000 = 10^{3}$	1.3712e-12
4	$100000 = 10^{5}$	1.2948e-10
5	$10000000 = 10^{7}$	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

Possiamo notare che all'aumentare delle iterazioni il valore di sigma aumenta incrementa con un fattore di 10^2, mentre la norma cresce all'incirca di 10^1. Questo aumento della norma è causato dall'incremento del valore di condizionamento della matrice.

11 Esercizio 11

```
function QR = myqr(A)
%
%function QR = myqr(A)
%fattorizzazione di una matrice A mxn
%Input:
%   —A matrice m x n con m >= n
%Output:
%   —QR Matrice che contiene informazioni riguardanti i fattori Q e R
%
    [m,n] = size(A);
    if(n > m)
        error(Dimensioni della matrice A non sono appropriate);
    end
    QR = A;
    for i = 1 : n
        alfa = norm(QR(i : n ,i));
        if (alfa == 0)
            error(Matrice non ha rango massimo!);
        end
        if(QR(i,i) >= 0)
            alfa = —alfa;
        end
        v1 = 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
    return
end
```

12 Esercizio 12

```
function x = qrsolve(QR,b)
%
%function x = qrsolve(QR,b)
%risoluzione sistema lineare QRx=b
%Input:
%   -QR Matrice ottenuta dalla funzione myqr(A)
%   -b: vettore termini noti del sistema Ax = b
%Output:
%   -x: soluzione del sistema lineare QRx = b
%

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

13 Esercizio 13

Eseguendo lo script:

```
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);
disp(QR);
sol = qrsolve(QR,b);
disp(sol);
```

Otteniamo:
La Matrice QR:

$$\begin{bmatrix} -6.7082 & -8.6461 & -11.1803 \\ 0.1297 & -1.1155 & -2.9881 \\ 0.3892 & -0.0827 & 1.0351 \\ 0.3892 & -0.0827 & -0.8037 \\ 0.6487 & -0.5222 & -0.4346 \end{bmatrix}$$

Soluzione del sistema lineare:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

14 Esercizio 14

Eseguendo lo scrpit:

```
format long e;
A = rot90(vander(1:10));
A = A(:,1:8);
x = (1:8)';
b = A*x;

disp(A\b);
disp('Condizionamento matrice Vandermonde');
disp(cond(A));
disp('Condizionamento dopo moltiplicazione');
disp(cond(A'*A));
disp((A'*A)\(A'*b));
```

Le operazioni:
- $A\backslash b$: da come risultato il vettore x, che rappresenta la soluzione del sistema lineare $Ax = b$; Questa operazione da lo stesso risultato di inv(A)*b se la matrice A ha rango massimo ed è non singolare.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$

- $(A' * A)\backslash(A' * b)$: poichè stiamo lavorando con una matrice mal condizionata, il vettore x risultante presenta un errore di approssimazione;

$$\begin{bmatrix} 3.5759 \\ -3.4624 \\ 9.5151 \\ -1.2974 \\ 7.9574 \\ 4.9125 \\ 7.2378 \\ 7.9765 \end{bmatrix}$$

Le due espressioni sopra elencate eseguono la stessa operazione : risoluzione del sistema lineare $Ax = b$; La differenza fra i risultati è data dal fatto che la matrice di Vandermonde è malcondizionata : quando si eseguono moltiplicazioni fra matrici mal condizionate, il coefficiente di condizionamento aumenta e di conseguenza influenza il risultato ottenuto. In questo esempio abbiamo che il condizionamento di A = 1.542832727600791e+09. Nel primo caso, quando andiamo ad effettuare $A\backslash b$, il condizionamento non è rilevante fino al punto di influenzare il risultato ottenuto. Nel secondo caso invece, dopo la moltiplicazione della matrice A con la sua trasposta, il valore di condizionamento risulta = 4.489735065328789e+18.Questo valore di condizionamento introdurrà un errore quando viene eseguita l'operazione successiva.

15 Esercizio 15

- Ascisse equidistanti

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

- Ascisse di Chebyshev

```
function x = chebyshev(a,b,n)
% function x = chebyshev(a,b,n);
%

x = (a+b)/2 +((b-a)/2)*cos((2*(0:n)+1)*(pi/(2*n+2)));
x = flip(x);
end
```

- Utilizzando il polinomio di newton

```
f = @(x)(cos((pi*x.^2)/2));
x = linspace(-1, 1, 1000);
erroriEquidistanti = zeros(1, 40);
erroriChebyshev = zeros(1, 40);
for n = 1:40
    xequi = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    fequi = newton(xequi,f(xequi),x);
    fchebyshev = newton(xcheby,f(xcheby),x);
    erroriEquidistanti(n) = max(abs(f(x) - fequi));
    erroriChebyshev(n) = max( abs(f(x) - fchebyshev));
end
semilogy(erroriEquidistanti);
hold on;
semilogy(erroriChebyshev);
legend(equidistanti,chebyshev);
xlabel(ascisse di interpolazione);
ylabel(errore commesso);
```

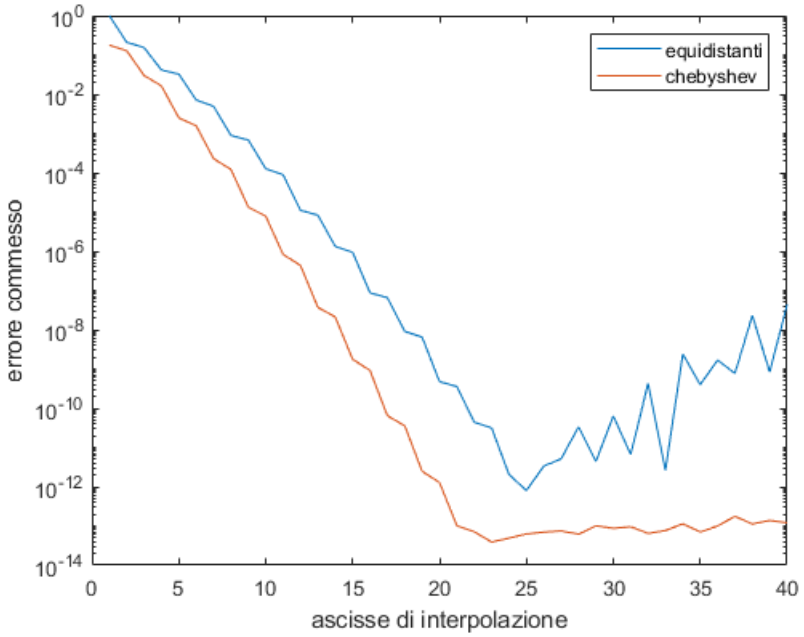

Figure 1: Grafico esercizio 15 con newton

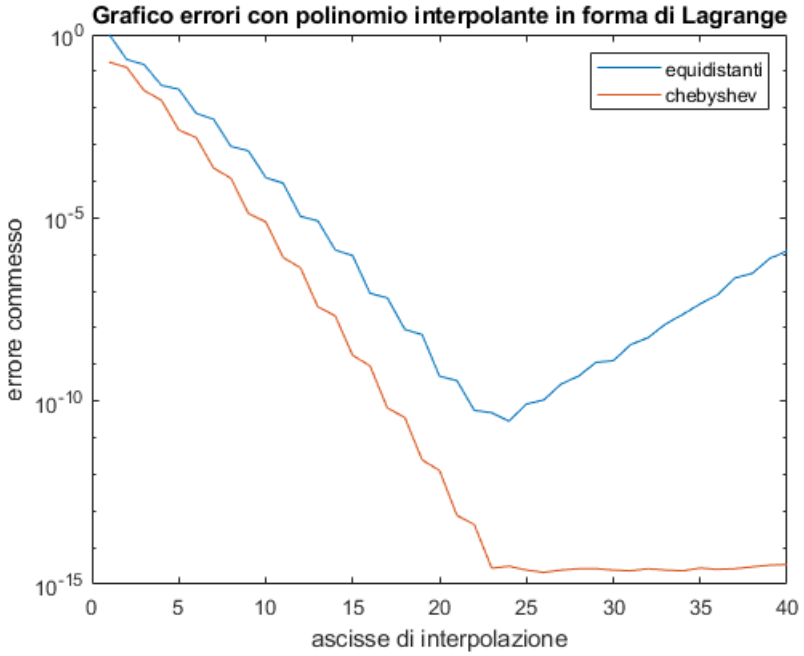

Figure 2: Grafico esercizio 15 con Lagrange

Possiamo notare dal grafico 1 che l'errore nel caso delle ascisse di chebyshev è costantemente minore rispetto a quello delle ascisse equidistanti. Dopo n = 23 il grafico dell'errore commesso utilizzando le ascisse di chebyshev presenta un andamento costante. Per le ascisse equidistanti (utilizzando i metodo di newton) invece, l'errore raggiunge il valore minimo a n =25 , e in seguito inizia a salire con un andamento non costante.
Possiamo notare che il grafico 2 è molto simile a quello precedente per quanto riguarda le ascisse di chebyshev. Mentre per le ascisse equidistanti, l'errore dopo aver raggiunto il suo valore minimo non presenta l'andamento a "zig zag".
Al crescere del numero delle ascisse di interpolazione l'errore commesso scegliendo le ascisse di chebyshev rimane più o meno costante in entrambi i casi. Questo è dovuto al fatto che la costante di Lebesgue

equivale a circa $\frac{2}{\pi} \log n$, e risulta quindi avere una crescita ottimale, mentre per le ascisse equidistanti invece si ha una crescita esponenziale al crescere di n.

16 Esercizio 16

- Hermite

```
function [y,df] = hermite( xi, fi, f1i, xx )
%
% [y,df] = hermite( xi, fi, f1i, xx ) Calcola il valore del polinomio
% interpolante di Hermite sulle
% ascisse xi. I vettori fi e f1i
% contengono i corrispondenti valori della f e della sua ferivata prima.
% Se specificato, df contiene il vettore delle differenze divise.
%
% controlli sui dati di ingresso
%
    m = length(xi);
    if m~=length(fi) || m~=length(f1i), error(dati inconsistenti), end
    for i = 1:m-1
        if any( find(xi(i+1:m)==xi(i)) ), error(ascisse non distinte), end
    end
    n = 2*m-1; % grado del polinomio interpolante
    x = zeros(n+1,1);
    df = x;
    x(1:2:n) = xi(:);
    x(2:2:n+1) = xi(:);
    df(1:2:n) = fi(:);
    df(2:2:n+1) = f1i(:);
    for i = n:-2:3 % seconda colonna della tabella
        df(i) = ( df(i)-df(i-2) )/( x(i)-x(i-1) );
    end
    for i = 2:n % colonne successive della tabella
        for j = n+1:-1:i+1
            df(j) = ( df(j)-df(j-1) )/( x(j)-x(j-i) );
        end
    end
    % calcolo il polinomio interpolante nelle ascisse prescritte
    %
    y = df(n+1)*ones(size(xx));
    for k = 0:n-1
        y = y.*( xx-x(n-k) ) +df(n-k);
    end
    return
end
```

- Script es. 16

```
f = @(x)(cos((pi*x.^2)/2));
f1 = @(x)(-pi*x.*sin((pi*x.^2)/2));
x = linspace(-1, 1, 101);
erroriEquidistanti = zeros(1, 20);
erroriChebyshev = zeros(1,  20);
for n = 1:20
    xequi = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    fequi = hermite(xequi,f(xequi),f1(xequi),x);
    fchebyshev = hermite(xcheby,f(xcheby),f1(xcheby),x);
    erroriEquidistanti(n) = max(abs(f(x) - fequi));
    erroriChebyshev(n) = max( abs(f(x) - fchebyshev));
end
semilogy(erroriEquidistanti);
```

```
15 hold on;
16 semilogy(erroriChebyshev);
17 legend(equidistanti,chebyshev);
18 xlabel(ascisse di interpolazione);
19 ylabel(errore commesso);
```

Eseguendo lo script si ottiene il seguente grafico:

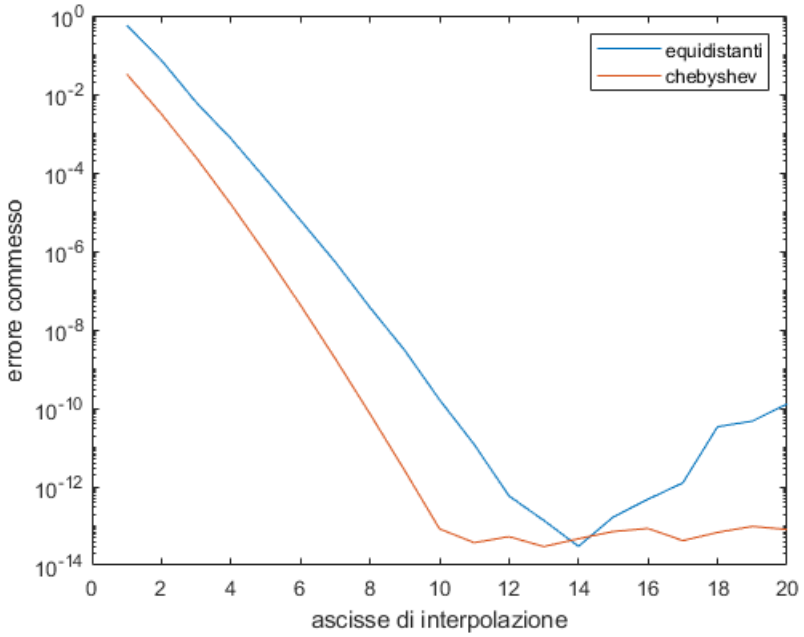

Figure 3: Grafico esercizio 16 con Hermite

Si può osservare che dopo n = 10 il grafico dell'errore per le ascisse di chebyshev presenta un andamento costante. Per le ascisse equidistanti utilizzando il polinomio di Hermite l'errore raggiunge il valore minimo a n = 14 ed in seguito torna a salire. Possiamo affermare che in generale l'approssimazione tramite ascisse di chebyshev è più precisa.

17 Esercizio17

```
function y = splinenat( xi, fi, x )
%
% y = splinenat( xi, fi,x ) Calcola il vettore degli mi per il calcolo di una
% spline cubica naturale interpolante i punti (xi,fi).
%
%
    m = length(xi);
    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)));

    len = length(x);
    y = zeros(len,1);

    for i = 1 : len
        index=find(x(i)>=xi(1:n),1,'last');
        if x(i) < xi(1)
            index = 1;
        end
        if x(i) > xi(m)
            index = n;
        end
        num = (((x(i)-xi(index)).^3).*mi(index+1)+((xi(index+1)-x(i)).^3).*mi(index));
        den = (6*hi(index));
        pq = q(index).*(x(i)-xi(index));
        y(i) = num/den + pq +r(index);
    end
    return

end
```

18 Esercizio 18

```
f = @(x)(cos((pi*(x.^2))/2));
x = linspace(-1,1, 1000);
x = x(:);
erroriEquidistanti= zeros(1, 40);
erroriChebyshev = zeros(1, 40);

for n = 4:100
    xlin = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    fequi = splinenat(xlin,f(xlin),x);
    fchebyshev = splinenat(xcheby,f(xcheby),x);
    erroriEquidistanti(n) = max(abs(f(x) - fequi));
    erroriChebyshev(n) = max( abs(f(x) - fchebyshev));
end

semilogy(erroriEquidistanti);
hold on;
semilogy(erroriChebyshev);
xlabel('ascisse di interpolazione');
ylabel('errore commesso');
legend({'equidistanti', 'chebyshev'});
```

Eseguendo lo script si ottiene il seguente grafico:

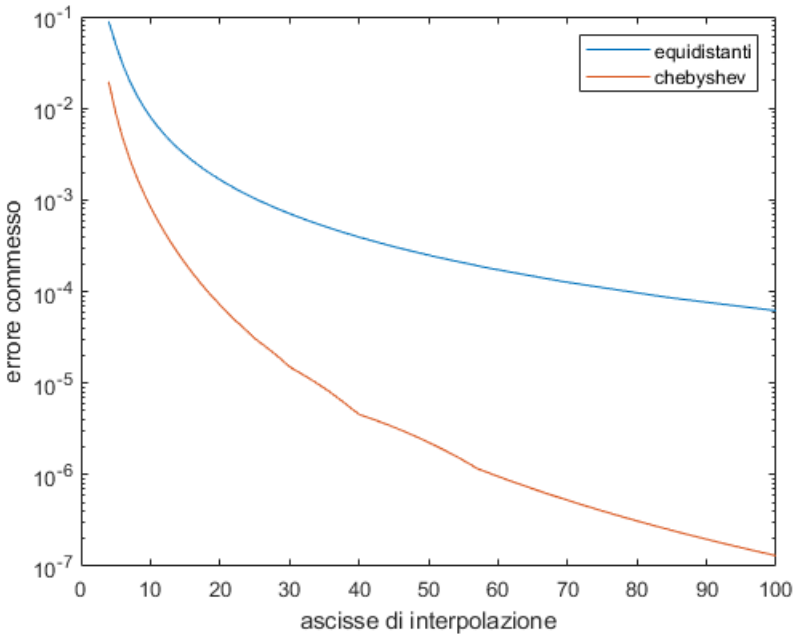

Figure 4: Grafico esercizio 18

19 Esercizio19

```
f = @(x)(cos((pi*x.^2)/2));
x = linspace(-1, 1, 1000);
erroriEquidistanti = zeros(1, 100);
erroriChebyshev = zeros(1, 100);
for n = 4:100
    xequi = linspace(-1, 1, n+1);
    xcheby = chebyshev(-1,1,n+1);
    fequi = spline(xequi,f(xequi),x);
    fchebyshev = spline(xcheby,f(xcheby),x);
    erroriEquidistanti(n) = max(abs(f(x) - fequi));
    erroriChebyshev(n) = max( abs(f(x) - fchebyshev));
end
semilogy(erroriEquidistanti);
hold on;
semilogy(erroriChebyshev);
legend(equidistanti,chebyshev);
xlabel(ascisse di interpolazione);
ylabel(errore commesso);
```

Eseguendo lo script si ottiene il seguente grafico:

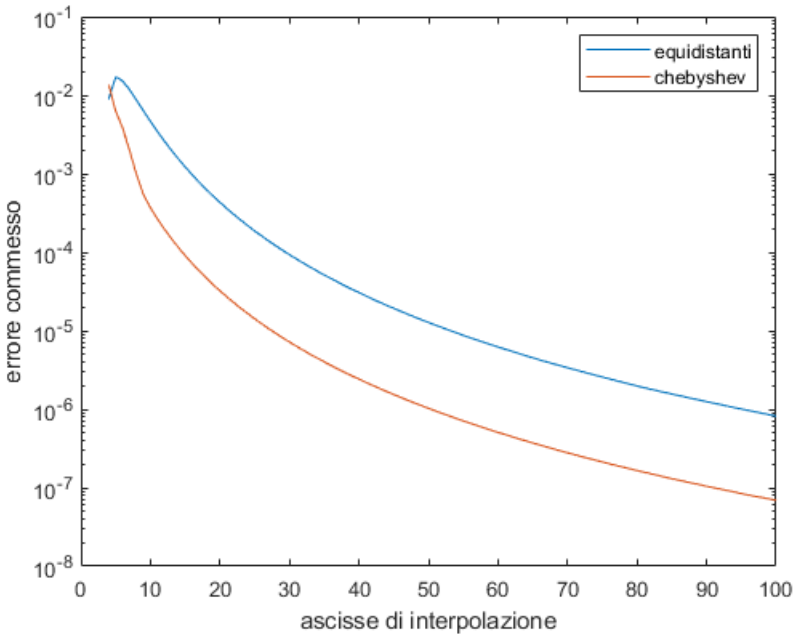

Figure 5: Grafico esercizio 19

Confrontando i due grafici possiamo notare che la funzione spline di Matlab commette un errore minore rispetto a splinenat.

- Asisse di Chebyshev I grafici presentano un'andamento molto simile formando una curva che parte da 10^{-2} a 10^{-7}, essendo l'errore commesso dalla funzione spline leggermente minore.
- Ascisse equidistanti Dai grafici si può osservare che utilizzando ascisse equidistanti la funzione splinenat commente un'errore notevolmente maggiore rispetto alla funzione spline di Matlab: nel caso della splinenat l'errore varia da 10^{-1} a 10^{-4},mentre nel caso della spline di matlab l'errore varia da circa 10^{-2} a 10^{-6}.

20 Esercizio 20

Polinomio di approssimazione ai minimi quadrati

```
f = @(x)(cos((pi*x.^2)/2));
fpert = @(x)(f(x) + 10^(-3)*rand(size(x)));
xi = -1 + 2*(0:10^4)/10^4;
fxi = f(xi);
fpi = fpert(xi);
errors=zeros(1, 20);
for m = 1:20
    a = polyfit(xi, fpi, m);
    y = polyval(a,xi);
    errors(m) = max(abs(y-fxi));
end
semilogy(errors);

xlabel('grado polinomio');
ylabel('errore di interpolazione');
```

Eseguendo lo script si ottiene il seguente grafico:

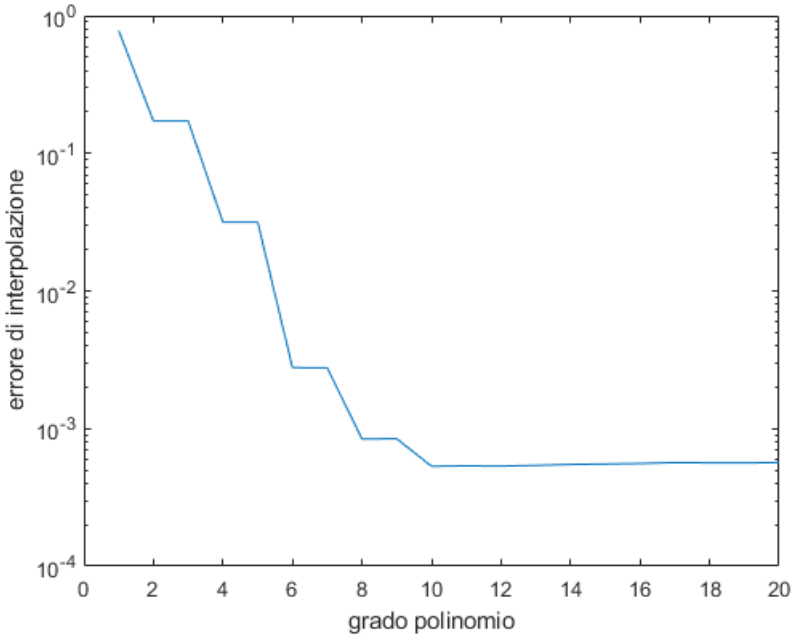

Figure 6: Grafico esercizio 20

Si può osservare che l'errore decresce quasi esponenzialmente fino a circa 10^{-3}, e quando il grado del polinomio m>=10, il grafico presenta un andamento costante.

21 Esercizio 21

```
function c = pesiNewtonCotes(n)
%
%function c = pesiNewtonCotes(n)
% funzione che calcola i pesi di quadratura di newton cotes di grado n.
%

    format rat;
    c = zeros(1,n);
    for i = 1 : n+1
        j = (0:n);
        j(i)=[];
        f = @(t)(prod(t-j) /prod(i-1-j));
        c(i) = integral(f, 0, n, 'ArrayValued', true);
    end

    return
end
```

Di seguito sono riportati i valori dei pesi delle formule di newton cotes di grado n = 1....7 .

grado/cnk	0	1	2	3	4	5	6	7
1	1/2	1/2						
2	1/3	4/3	1/3					
3	3/8	9/8	9/8	3/8				
4	14/45	64/45	8/15	64/45	14/45			
5	95/288	125/96	125/144	125/144	12/96	95/288		
6	41/140	54/35	27/140	68/35	27/140	54/35	41/140	
7	1073/3527	810/559	343/640	649/536	649/536	343/640	810/559	1073/3527

22 Esercizio22

```
condiz = zeros(50,1);
for i = 1 : 50
    condiz(i) = (1/i)*sum(abs(pesiNewtonCotes(i)));
end
semilogy(condiz);
ylabel(condizionamento);
xlabel(grado formula newton cotes);
```

Eseguendo lo script si ottiene il seguente grafico:

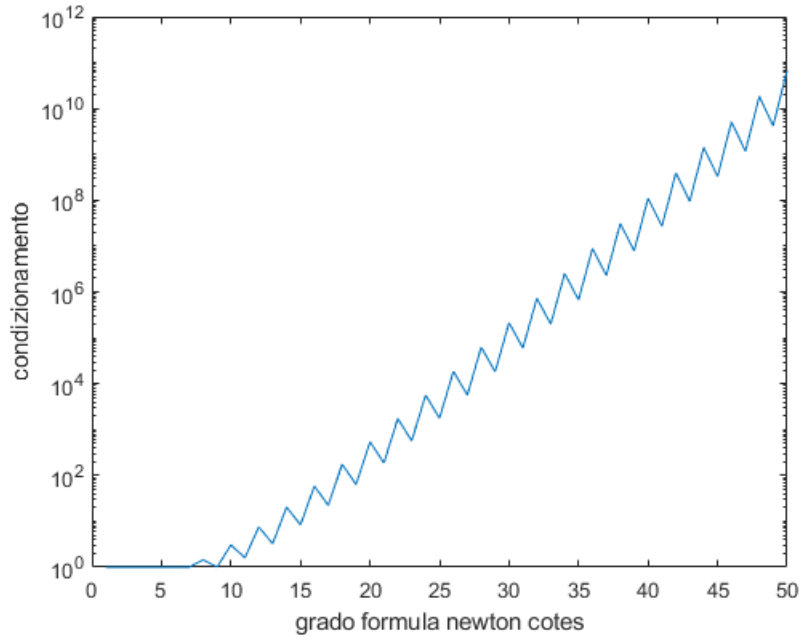

Figure 7: Grafico esercizio 22

23 Esercizio 23

```
format short e;
f =@(x)tan(x);
expectedY = log(cos(1)/cos(1.1));
a = -1;
b = 1.1;
actualY = zeros(9,1);
errore = zeros(9,1);
for i = 1 : 9

    actualY(i) = round(newtonCotes(f,a,b,i),3,'significant');
    errore(i) = round(abs(expectedY - actualY(i)),3,'significant');
end

disp(actualY);
disp(errore);
```

Nella seguente tabella sono riportati i valori approssimati fino a 3 cifre significative in notazione scientifica .

grado	approssimazione	errore
1	4.28e-01	2.53e-01
2	2.13e-01	3.81e-02
3	1.96e-01	2.11e-02
4	1.80e-01	5.08e-03
5	1.79e-01	4.08e-03
6	1.76e-01	1.08e-03
7	1.76e-01	1.08e-03
8	1.75e-01	7.84e-05
9	1.75e-01	7.84e-05

24 Esercizio 24

- Simpson Composita

```
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)
%       f       funzione
%       a,b     estremi dell'intervallo
%       I       approssimazione integrale definito di f(x)
%
%
    format long e;

    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
```

- Trapezi Composita

```
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
%       f       funzione
%       a,b     estremi dell'intervallo
%       I       approssimazione integrale definito di f(x)
%
    format long e;
    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
```

- script

```
a = -1;
b = 1.1;
f = @(x)tan(x);
n = 100;

trapezi = zeros(1,n);
simpson = zeros(1,n);

for i = 1 : n
    trapezi(i) = trapecomp(f,a,b,2*i);  %2*i, deve essere pari
    simpson(i) = simpcomp(f,a,b,2*i);
end

trapezi = trapezi(:);
disp(Trapezi:  );
disp(trapezi);
disp(Simpson Composite:);
simpson = simpson(:);
disp(simpson);
```

Sono state riportate solo le prime 10 iterazioni.

Iterazioni	Trapezio	Simpson
2	2.664035584060345e-01	2.126315681335669e-01
4	2.034328044500163e-01	1.824425531313435e-01
6	1.884983466139722e-01	1.773334438860330e-01
8	1.827894088752250e-01	1.759082770169613e-01
10	1.800348035219603e-01	1.753928683822888e-01
12	1.785040157074719e-01	1.751725720719718e-01
14	1.775682181954108e-01	1.750665465192467e-01
16	1.769554131112014e-01	1.750107478565269e-01
18	1.765327096164695e-01	1.749792544399417e-01
20	1.762290375520298e-01	1.749604488953863e-01

$\int_{-1}^{1.1} tan(x)dx = log(\frac{cos(1)}{cos(1.1)}) = 0.174922$ (circa) , per cui risulta ovvio che l'approssimazione dell'integrale risulta più precisa utilizzando il metodo di Simpson.
Dal punto di vista computazionale la formula dei trapezi composita risulta leggermente meno costosa (tutti e due i metodi hanno costo lineare in n): entrambi eseguono una volta la funzione feval, ma il metodo dei trapezi composita calcola l'approssimazione dell'integrale eseguendo meno operazioni rispetto al metodo di Simpson.

25 Esercizio 25

- Simpson Adattiva

```
function [I2,points] = adapsim( f, a, b, tol, fa, f1, fb )
%
% [I2,points] = adapsim( f, a, b, tol )
%
    format long e;
    global points
    delta = 0.5; % ampiezza minima intervalli
    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+4*f1+fb );
    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
```

- Trapezi Adattiva

```
function [I2,points] = adaptrap( f, a, b, tol, fa, fb )
%
% [I2,points] = adaptrap( f, a, b, tol )
%
    format long e;
    global points
    delta = 0.5; % ampiezza minima intervalli
    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
```

- script

```
a = -1;
b = 1;
f = @(x)1/(1+10^2*x^2);
n = 5;
trapezi = zeros(1,n);
simpson = zeros(1,n);
pointsTrap = zeros(1,n);
pointsSim = zeros(1,n);

for i = 1 : n
    [trapezi(i),pointsT] = adaptrap(f,a,b,10^-(i+1));
    [simpson(i),pointsS] = adapsim(f,a,b,10^-(i+1));
    pointsTrap(i) = length(pointsT);
    pointsSim(i) = length(pointsS);
end

trapezi = trapezi(:);
disp(Trapezi:  );
disp(trapezi);
disp(Punti:  );
disp(pointsTrap);
disp(Simpson Composite:);
simpson = simpson(:);
disp(simpson);
disp(Punti:  );
disp(pointsSim);
```

Di seguito sono riportati i dati ottenuti:

Tolleranza	Trapezio	N.Punti	Simpson	N. punti
10^{-2}	2.955597117841284e-01	21	2.812976430626699e-01	17
10^{-3}	2.945853681850339e-01	93	2.812976430626699e-01	17
10^{-4}	2.942742008736351e-01	277	2.942593384196308e-01	41
10^{-5}	2.942301421648779e-01	793	2.942278097680047e-01	81
10^{-6}	2.942260196031783e-01	2693	2.942257646203842e-01	145

$\int_{-1}^{1} \frac{1}{1+10^2x^2}dx = 0.29423$ (circa).

Dai dati riportati in tabella risulta che entrambe le funzioni calcolano un'approssimazione molto precisa dell'integrale.
Inoltre,si nota immediatamente che la formula adattiva di Simpson calcola l'approssimazione dell'integrale

utilizzando un numero di punti nettamente inferiore rispetto a quella del Trapezio.Il costo computazionale in questo caso dipende dal numero di chiamate ricorsive effettuate (cioè di punti calcolati) , per cui possiamo affermare che la formula adattiva di Simpson è più efficiente rispetto a quella del Trapezio.