Elaborato Calcolo Numerico

Custodi Alessandro 7084103
alessandro.custodi@edu.unifi.it

Matassini Cosimo 7083831
cosimo.matassini@edu.unifi.it

Anno Accademico 2023/2024

Esercizio 1

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5)$$
$$f(x-2h) = f(x) - 2hf'(x) + 2h^2f''(x) - \frac{4h^3}{3}f'''(x) + \frac{2h^4}{3}f^{(4)}(x) + O(h^5)$$
$$f(x-3h) = f(x) - 3hf'(x) + \frac{9h^2}{2}f''(x) - \frac{9h^3}{2}f'''(x) + \frac{27h^4}{8}f^{(4)}(x) + O(h^5)$$
$$f(x-4h) = f(x) - 4hf'(x) + 8h^2f''(x) - \frac{32h^3}{3}f'''(x) + \frac{32h^4}{3}f^{(4)}(x) + O(h^5)$$

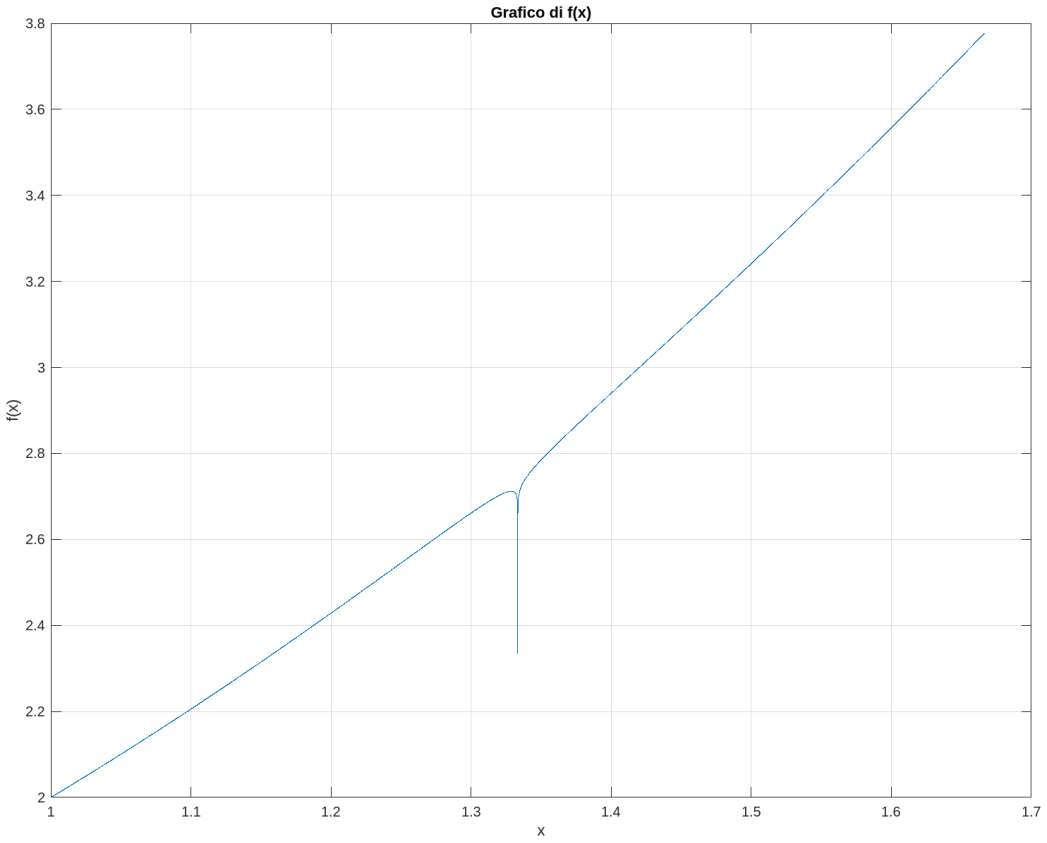
$$25f(x) - 48f(x-h) + 36f(x-2h) - 16f(x-3h) + 3f(x-4h) =$$

$$\begin{aligned}
&25f(x) - 48f(x) + 48hf'(x) - 24h^2f''(x) + 8h^3f'''(x) - 2f^{(4)}(x) + \\
&+ 36f(x) - 72hf'(x) + 72h^2f''(x) - 48h^3f'''(x) + 24 - 2f^{(4)}(x) - \\
&- 16f(x) + 48hf'(x) - 72h^2f''(x) + 72h^3f'''(x) - 54 - 2f^{(4)}(x) + \\
&+ 3f(x) - 12hf'(x) + 24h^2f''(x) - 32h^3f'''(x) + 32 - 2f^{(4)}(x) + O(h^5) = \\
&12hf'(x) + O(h^5)
\end{aligned}$$

$$\frac{12f'(x)h + O(h^5)}{12h} = f'(x) + O(h^4)$$

Esercizio 2

```
x = linspace(1, 5/3, 100001);
f = @(x) 1 + x.^2 + log(abs(3*(1 - x) + 1))/80;
plot(x, f(x))
xlabel('x')
ylabel('f(x)')
title('Grafico di f(x)')
grid on
```



Considerando la funzione calcolata nell'intervallo $[1, \frac{5}{3}]$, utilizzando il comando `min(f(x))` vengono restituiti i valori $x = 1$ e $y = 2$, che possiamo verificare graficamente (da non confondere però con il comportamento reale della funzione); se invece proviamo a calcolare la funzione in $\frac{4}{3}$, Matlab restituisce il valore 2.3272, molto più alto di quello calcolato precedentemente.

L'utilizzo di un aritmetica finita non ci permette di ottenere ulteriori informazioni riguardo all'asintoto verticale in $x = \frac{4}{3}$.

Esercizio 3

La cancellazione numerica è la manifestazione del malcondizionamento della somma algebrica quando, dati gli addendi x_1 e x_2, $x_1 \approx -x_2$. Infatti, il numero di condizionamento k risulta essere molto grande, in quanto il denominatore tende a zero:

$$k = \frac{|x_1| + |x_2|}{|x_1 + x_2|} \approx \frac{|x_1| + |x_1|}{|x_1 + x_2|} = \frac{2|x_1|}{|x_1 + x_2|} >> 1$$

Se ad esempio consideriamo $x_1 = 1.000$ e $x_2 = -1.001$ (valori quasi opposti), la somma algebrica $y = x_1 + x_2$ risulta essere $-0.001 = -10^{-3}$. Considerando invece come dati perturbati $\tilde{x_1} = 1.000$ e $\tilde{x_2} = -0.999$, otteniamo $\tilde{y} = \tilde{x_1} + \tilde{x_2} = 0.001 = 10^{-3}$.
Il numero di condizionamento risulta essere $k = \frac{1.000+1.001}{10^{-3}} = 2.001 * 10^3$: l'errore sui dati viene amplificato molto.
$\epsilon_1 = \frac{\tilde{x_1} - x_1}{x_1} = \frac{0}{1} = 0$
$\epsilon_2 = \frac{\tilde{x_2} - x_2}{x_2} \approx 2 * 10^{-3}$
L'errore sui dati iniziali: $\epsilon_x = max\{|\epsilon_1|, |\epsilon_2|\} = 2 * 10^{-3}$
L'errore sul risultato: $|\epsilon_y| = |\frac{\tilde{y} - y}{y}| = |\frac{10^{-3} - (-10^{-3})}{-10^{-3}}| = 2$
Ci sono 3 ordini di grandezza tra l'errore sui dati iniziali e l'errore sul risultato.

Esercizio 4

```
1  function [x, iterazioni] = bisezione(f, a, b, tol,
       itmax)
2  %
3  % [x, iterazioni] = bisezione(f, a, b, tol, itmax)
4  %
5  % calcola approssimazione della radice di
6  % f(x) con tolleranza tol
7  %
8  % Input:
9  %   f - funzione da cui calcolare la radice
10 %   a, b - punti iniziali
```

```
%   tol - tolleranza richiesta
%   itmax - numero iterazioni max
%
% Output:
%   x - approssimazione della soluzione
%   iterazioni - numero delle iterazioni eseguite

if nargin < 3
  error('numero di argomenti in ingresso errato');
elseif nargin == 3
  tol = 1e-6;
  itmax = ceil(log2(b - a) - log2(tol));
elseif nargin == 4
  itmax = ceil(log2(b - a) - log2(tol));
end

if(b <= a)
  error('intervallo iniziale errato');
end

if tol <= 0
  error('tolleranza non valida');
end

iterazioni = 0;

fa = f(a);
fb = f(b);

if fa == 0
  x = a;
  return;
end
if fb == 0
  x = b;
  return;
end

if fa * fb > 0
  error('intervallo non accettabile');
end

for iterazioni = 1:itmax
  x = (a + b) / 2;
  fx = f(x);
  fxderivata = abs(fb - fa) / (b - a);
```

```
57
58    if abs(fx) / abs(fxderivata) <= tol
59      break;
60    elseif fa * fx > 0
61      a = x;
62      fa = fx;
63    else
64      b = x;
65      fb = fx;
66    end
67  end
68  return
```

Esercizio 5

Newton:

```
1   function [x, iterazioni] = newton(f, df, x0, tol,
        imax)
2   %
3   % [x, iterazioni] = newton(f, df, x0, tol, imax)
4   %
5   %
6   % Input:
7   %   f - funzione da cui ricavare la radice
8   %   df - derivata della funzione f
9   %   x0 - punto di partenza
10  %   tol - tolleranza richiesta
11  %   imax - numero di iterazioni max
12  %
13  % Output:
14  %   x - approssimazione della radice
15  %   iterazioni - numero delle iterazioni eseguite
16
17  if nargin < 3
18    error('numero di argomenti insufficiente');
19  elseif nargin == 3
20    tol = 10e-16;
21    imax = 1000;
22  elseif nargin == 4
23    imax = 1000;
24  end
25
26  x = x0;
27
```

```
for iterazioni = 1:imax
  fx = f(x);
  fxderivata = df(x);
  if fxderivata == 0
    break;
  end
  x = x - fx/fxderivata;
  if abs(x - x0) <= tol * (1 + abs(x0))
    return;
  else
    x0 = x;
  end
end

warning('soluzione non trovata nelle iterazioni massime disponibili');
return
```

Secanti:

```
function [x, iterazioni] = secanti(f, x0, x1, tol, itmax)
%
% [x, iterazioni] = secanti(f, xo, x1, tol, itmax)
%
% Calcola una approssimazione della radice di f(x)
% con tolleranza tol
%
% Input:
%   f - funzione da cui ricavare la radice;
%   x0, x1 - punti iniziali;
%   tol - accuratezza richiesta
%   itmax - numero massimo di iterazioni
%
% Output:
%   x - approssimazione della soluzione
%   iterazioni - numero delle iterazioni eseguite

if nargin < 3
  error('numero di argomenti in ingresso errato')
elseif nargin == 3
  tol = 10e-16;
  itmax = 1000;
elseif nargin == 4
  itmax = 1000;
end

```

```
if tol <= 0
  error('tolleranza errata')
end
if itmax <= 0
  error('numero di iterazioni massimo errato')
end

f0 = f(x0);
f1 = f(x1);
for iterazioni = 1:itmax
  if f1 == f0
    error('impossibile eseguire il metodo');
  end
  x = (f1 * x0 - f0 * x1) / (f1 - f0);
  if abs(x - x1) <= tol
    break
  elseif iterazioni < itmax
  x0 = x1;
  f0 = f1;
  x1 = x;
  f1 = f(x1);
  end
end
if abs(x - x1) > tol
  error('soluzione non trovata nelle iterazioni
     massime disponibili');
end
return
```

Esercizio 6

```
f = @(x) exp(x) - cos(x);
df = @(x) exp(x) + sin(x);

xstar = zeros(3, 0);
iterazioni = zeros(3, 0);

metodo  = ["Bisezione"; "Newton"; "Secanti"];

[xstar(1, 1), iterazioni(1, 1)] = bisezione(f, -0.1,
    1, 10.^(-3), 1000);
[xstar(2, 1), iterazioni(2, 1)] = newton(f, df, 1,
   10.^(-3), 1000);
```

```
[xstar(3, 1), iterazioni(3, 1)] = secanti(f, 1, 0.9,
    10.^(-3), 1000);

tolleranza3 = table(metodo, xstar, iterazioni);

[xstar(1, 1), iterazioni(1, 1)] = bisezione(f, -0.1,
    1, 10.^(-6), 1000);
[xstar(2, 1), iterazioni(2, 1)] = newton(f, df, 1,
   10.^(-6), 1000);
[xstar(3, 1), iterazioni(3, 1)] = secanti(f, 1, 0.9,
    10.^(-6), 1000);

tolleranza6 = table(metodo, xstar, iterazioni);

[xstar(1, 1), iterazioni(1, 1)] = bisezione(f, -0.1,
    1, 10.^(-9), 1000);
[xstar(2, 1), iterazioni(2, 1)] = newton(f, df, 1,
   10.^(-9), 1000);
[xstar(3, 1), iterazioni(3, 1)] = secanti(f, 1, 0.9,
    10.^(-9), 1000);

tolleranza9 = table(metodo, xstar, iterazioni);

[xstar(1, 1), iterazioni(1, 1)] = bisezione(f, -0.1,
    1, 10.^(-12), 1000);
[xstar(2, 1), iterazioni(2, 1)] = newton(f, df, 1,
   10.^(-12), 1000);
[xstar(3, 1), iterazioni(3, 1)] = secanti(f, 1, 0.9,
    10.^(-12), 1000);

tolleranza12 = table(metodo, xstar, iterazioni);
```

Table 1: Tolleranza 10^{-3}

Metodo	xstar	iterazioni
Bisezione	0.00097656	9
Newton	2.8423e-09	5
Secanti	1.1522e-06	6

Table 2: Tolleranza 10^{-6}

Metodo	xstar	iterazioni
Bisezione	9.5367e-07	19
Newton	3.5748e-17	6
Secanti	2.0949e-16	8

Table 3: Tolleranza 10^{-9}

Metodo	xstar	iterazioni
Bisezione	9.3132e-10	29
Newton	3.5748e-17	7
Secanti	2.0949e-16	8

Table 4: Tolleranza 10^{-12}

Metodo	xstar	iterazioni
Bisezione	9.0949e-13	39
Newton	3.5748e-17	7
Secanti	-1.2557e-17	9

Utilizzando il comando `table()` di Matlab, abbiamo tabulato i risultati ottenuti con i metodi di bisezione, Newton e secanti per le diverse tolleranze. La colonna xstar rappresenta il valore approssimato della radice, mentre la colonna iterazioni rappresenta il numero di iterazioni necessarie per ottenere il risultato. Da qui possiamo vedere che il miglior metodo, in tutti e 4 i casi, è Newton e il peggiore è il metodo di bisezione. Anche secanti è molto efficiente, arrivando a convegenza in poche iterazioni e con una tolleranza molto minore rispetto a quella indicata di partenza, in tutti i 4 casi.

Esercizio 7

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

metodo  = ["Bisezione"; "Newton"; "Newton modificato"; "Secanti"];

xstar = zeros(4, 0);
iterazioni = zeros(4, 0);

[xstar(1, 1), iterazioni(1, 1)] = bisezione(f, -0.1, 1, 10.^(-3), 1000);
[xstar(2, 1), iterazioni(2, 1)] = newton(f, df, 1, 10.^(-3), 1000);
[xstar(3, 1), iterazioni(3, 1)] = newtonMod(f, 5, df, 1, 10.^(-3), 1000);
[xstar(4, 1), iterazioni(4, 1)] = secanti(f, 1, 0.9, 10.^(-3), 1000);

tolleranza3 = table(metodo, xstar, iterazioni);

xstar = zeros(4, 0);
iterazioni = zeros(4, 0);

[xstar(1, 1), iterazioni(1, 1)] = bisezione(f, -0.1, 1, 10.^(-6), 1000);
[xstar(2, 1), iterazioni(2, 1)] = newton(f, df, 1, 10.^(-6), 1000);
[xstar(3, 1), iterazioni(3, 1)] = newtonMod(f, 5, df, 1, 10.^(-6), 1000);
[xstar(4, 1), iterazioni(4, 1)] = secanti(f, 1, 0.9, 10.^(-6), 1000);

tolleranza6 = table(metodo, xstar, iterazioni);

xstar = zeros(4, 0);
iterazioni = zeros(4, 0);

[xstar(1, 1), iterazioni(1, 1)] = bisezione(f, -0.1, 1, 10.^(-9), 1000);
[xstar(2, 1), iterazioni(2, 1)] = newton(f, df, 1, 10.^(-9), 1000);
[xstar(3, 1), iterazioni(3, 1)] = newtonMod(f, 5, df, 1, 10.^(-9), 1000);
```

```
[xstar(4, 1), iterazioni(4, 1)] = secanti(f, 1, 0.9,
    10.^(-9), 1000);

tolleranza9 = table(metodo, xstar, iterazioni);

xstar = zeros(4, 0);
iterazioni = zeros(4, 0);

[xstar(1, 1), iterazioni(1, 1)] = bisezione(f, -0.1,
    1, 10.^(-12), 1000);
[xstar(2, 1), iterazioni(2, 1)] = newton(f, df, 1,
   10.^(-12), 1000);
[xstar(3, 1), iterazioni(3, 1)] = newtonMod(f, 5, df
   , 1, 10.^(-12), 1000);
[xstar(4, 1), iterazioni(4, 1)] = secanti(f, 1, 0.9,
    10.^(-12), 1000);

tolleranza12 = table(metodo, xstar, iterazioni)
```

Table 5: Tolleranza 10^{-3}

Metodo	xstar	iterazioni
Bisezione	0.0375	3
Newton	0.0039218	25
Newton modificato	-	-
Secanti	0.005576	33

Table 6: Tolleranza 10^{-6}

Metodo	xstar	iterazioni
Bisezione	0.003125	5
Newton	-	-
Newton modificato	-	-
Secanti	-0.0010403	61

Table 7: Tolleranza 10^{-9}

Metodo	xstar	iterazioni
Bisezione	0.0011163	31
Newton	-	-
Newton modificato	-	-
Secanti	-0.001075	89

Table 8: Tolleranza 10^{-12}

Metodo	xstar	iterazioni
Bisezione	0.0011163	32
Newton	-	-
Newton modificato	-	-
Secanti	-0.0010751	123

Il metodo di Newton modificato è stato implementato aggiungendo ai parametri della funzione il valore m, molteplicità della radice, e aggiornati in maniera opportuna i controlli di consistenza sul numero di argomenti. In questo caso la molteplicità della radice è 5.
Stavolta il metodo di bisezione risulta essere il migliore in termini di iterazioni. Il metodo di Newton converge solo con tolleranza di 10^{-3}, mentre il metodo di Newton modificato non converge in nessuno dei 4 casi. Entrambi i metodi, quando non convergono, restituiscono il messaggio di errore "derivata uguale a zero". Il metodo delle secanti converge sempre, ma è il peggiore in termini di iterazioni.

Esercizio 8

```
1 function x = mialu(A,b)
2 %
3 % x = mialu(A,b)
```

```
4  %
5  % Metodo di fattorizzazione LU con pivoting parziale
6  %
7  % Input:
8  %   A: matrice n x n
9  %   b: vettore dei termini noti
10 %
11 % Output:
12 %   x: soluzione del sistema Ax = b
13 %
14
15 [m, n] = size(A);
16 if m ~= n
17   error('La matrice non e quadrata!');
18 end
19 [m, o] = size(b);
20 if m ~= n || o ~= 1
21   error('Dimensione del vettore e della matrice non
        compatibili')
22 end
23 P = (1:n)';
24
25 for i = 1:n-1
26   [mi, ki] = max(abs(A(i:n, i)));
27   if mi == 0
28     error('Matrice singolare');
29   end
30   ki = ki+i-1;
31   if ki > i
32     A([i ki], :) = A([ki i], :);
33     P([i ki]) = P([ki i]);
34   end
35   A(i+1:n, i) = A(i+1:n, i) / A(i,i);
36   A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) *
        A(i, i+1:n);
37 end
38
39 x = b(P);
40 for i = 2:n
41   x(i:n) = x(i:n) - A(i:n, i-1) * x(i-1);
42 end
43 for i = n:-1:1
44   x(i) = x(i) / A(i,i);
45   x(1:i-1) = x(1:i-1) - A(1:i-1,i) * x(i);
46 end
47 return
```

Inserendo la matrice `A = [1 0 2; 0 0 2]`, con un qualsiasi vettore b, si ottiene il messaggio di errore `La matrice non e quadrata!`
Inserendo la matrice `A = [1 2 3 ; 2 4 6 ; 3 6 9]`, con un qualsiasi vettore b, si ottiene il messaggio di errore `Matrice singolare`
Inserendo il vettore `b = [1; 2; 3; 4]` (con una matrice quadrata non singolare) si ottiene il messaggio di errore `Dimensione del vettore e della matrice non compatibili`
Inserendo la matrice `A = [1 2 3; 4 5 6; 7 8 10]`, che è quadrata e non singolare, e il vettore `b=[1; 2; 3]`, si ottiene il vettore soluzione `x = [-0.3333; 0.6667; 0]`

Esercizio 9

```
function x = mialdl(A,b)
%
% x = mialdl(A,b)
%
% Risolve Ax = b con fattorizzazione LDLt
%
% Input:
%   A: matrice n x n
%   b: vettore dei termini noti
%
% Output:
%   x: soluzione del sistema Ax = b

[m, n] = size(A);
if m ~= n
  error('La matrice non e quadrata!');
end
[m, o] = size(b);
if m ~= n || o ~= 1
  error('Dimensione del vettore e della matrice non
     compatibili')
end

if A(1,1) <= 0
 error('Matrice non sdp');
end

A(2:n, 1) = A(2:n, 1) / A(1,1);
for j = 2:n
  v = (A(j, 1:j-1).') .* diag(A(1:j-1, 1:j-1));
```

```
30    A(j, j) = A(j, j) - A(j, 1:j-1) * v;
31    if A(j, j) <= 0
32      error('Matrice non sdp');
33    end
34    A(j+1:n, j) = (A(j+1:n, j) - A(j+1:n, 1:j-1) * v)
        / A(j, j);
35  end
36
37  d = diag(A);
38  if ~all(d > 0)
39    error('Matrice non sdp');
40  end
41  x = b(:);
42  for i = 2:n
43    x(i:n) = x(i:n) - A(i:n, i-1) * x(i-1);
44  end
45  x = x./d;
46  for i = n:-1:2
47    x(1:i-1) = x(1:i-1) - A(i, 1:i-1)' * x(i);
48  end
49  return
```

Inserendo la matrice `A = [1 0 2; 0 0 2]`, con un qualsiasi vettore b, si ottiene il messaggio di errore `La matrice non e quadrata!`
Inserendo il vettore `b = [1; 2; 3; 4]` (con una matrice di dimensione nxn, con n diverso da 4) si ottiene il messaggio di errore `Dimensione del vettore e della matrice non compatibili`
Inserendo la matrice `A = [1 2 3; 4 5 6; 7 8 -10]`, con un qualsiasi vettore colonna di lunghezza 4, si ottiene il messaggio di errore `Matrice non sdp`
Inserendo la matrice `A = [4 1 1; 1 2 0; 1 0 3]` e il vettore `b=[1; 2; 3]`, si ottiene il vettore soluzione `x = [-0.3158, 1.1579, 1.1053]`

Esercizio 10

```
1  function [x, nr] = miaqr(A, b)
2  %
3  % [x, nr] = miaqr(A,b)
4  %
5  % Esegue la fattorizzazione QR di A
6  % restituendo la soluzione ai minimi quadrati del
     sistema
7  % e la norma del corrispondente vettore residuo
```

```
%
% Input:
%   A: matrice m x n
%   b: vettore dei termini noti
%
% Output:
%   x: soluzione del sistema Ax = b
%   nr: norma vettore residuo

[m, n] = size(A);
[k, o] = size(b);
if m ~= k || o ~= 1
  error('Dimensione del vettore e della matrice non compatibili')
end

for i = 1:n
  alfa = norm(A(i:m, i));
  if alfa == 0
    error('Matrice non a rango massimo');
  end
  if A(i,i) >= 0
    alfa = -alfa;
  end
  v1 = A(i,i) - alfa;
  A(i,i) = alfa;
  A(i+1:m, i) = A(i+1:m, i) / v1;
  beta = -v1 / alfa;
  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));
  b(i:m) = b(i:m) - (beta * [1 A(i+1:m, i)'] * b(i:m)) * [1; A(i+1:m, i)];
end

x = b(:);
for i = n:-1:1
  x(i) = x(i) / A(i,i);
  x(1:i-1) = x(1:i-1) - A(1:i-1, i) * x(i);
end
nr = norm(x(n+1:m));
x = x(1:n);
return
```

Inserendo la matrice `A = [1 2; 3 4; 5 6]` e il vettore `b = [1; 2; 3]`, si ottiene il messaggio di errore `Dimensione del vettore e della matrice non compatibili`

Inserendo la matrice `A = [1 2 3 4; 4 5 6 7; 7 8 9 10];` e il vettore `b = [1; 2; 3]`, si ottiene il messaggio di errore `Matrice non a rango massimo`

A = [1 2; 3 4; 5 6]; b = [3; 4; 5]
Inserendo la matrice `A = [1 2; 3 4; 5 6]` e il vettore `b = [3; 4; 5]`, si ottiene il vettore soluzione `x = [-2.0000; 2.5000]` e la norma del corrispondente vettore residuo `nr = 3.5527e-15`

Esercizio 11

Matrice	**Numero di condizionamento**
A_1	1
A_2	11.356
A_3	191.06
A_4	2167.9
A_5	22819
A_6	2.3418e+05
A_7	2.3771e+06
A_8	2.3995e+07
A_9	2.4147e+08
A_{10}	2.4254e+09
A_{11}	2.4332e+10
A_{12}	2.4391e+11
A_{13}	2.4437e+12
A_{14}	2.4473e+13
A_{15}	2.4501e+14

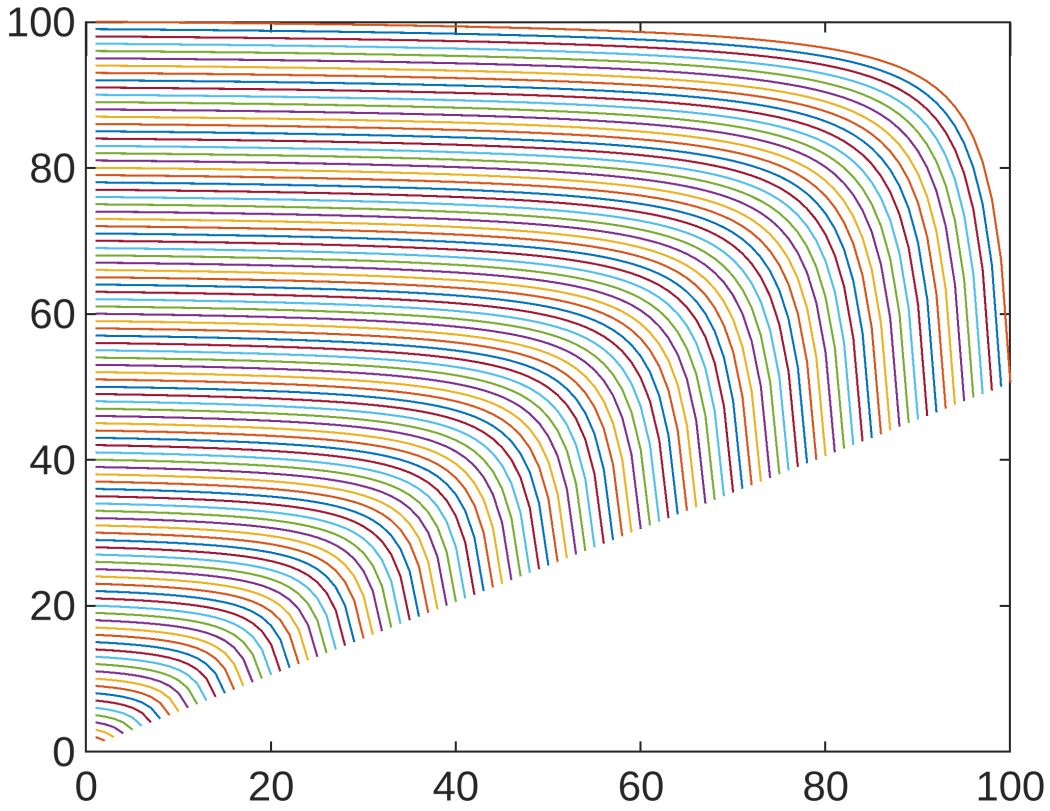
Come possiamo vedere dalla tabella, il numero di condizionamento delle matrici (utilizzando la funzione `cond` con una matrice in ingresso) cresce molto rapidamente, ma nonostante questo, tutti i valori delle soluzioni dei vari vettori x_i $i = 1, \cdots, n$ calcolati utilizzando la funzione `mialu`, non si discostano molto da 1. Non possiamo comunque escludere il fatto che, aumentando la dimensione della matrice e del vettore dei coefficenti, il vettore soluzione possa contenere valori che si discostano da 1 in maniera più sostanziale di quelli che abbiamo visto.

Esercizio 12

Con il seguente codice Matlab sono state calcolate tutte le 100 matrici A_n e graficato (in un unico grafico) gli elementi del fattore D rispetto all'indice diagonale

```
for n = 1:100
  An = ones(n) .* -1 + diag(ones(1, n) * n + 1);
  factorizedA = mialdlt(An);
  plot ((1:n), diag(factorizedA));
  hold on
end
hold off
```

La funzione mialdlt è una porzione della funzione `mialdl` definita in precedenza, che restituisce solamente la matrice inserita come argomento fattorizzata LDLt.



Esercizio 13

```
omega = [0.5; 0.5; 0.75; 0.25; 0.25];
B = diag(sqrt(omega));
A = [7 2 1; 8 7 8; 7 0 7; 4 3 3; 7 0 10]
b = [1; 2; 3; 4; 5]
[x, nr] = miaqr(B*A, B*b);
```

Dal codice matlab sopra riportato, otteniamo `x = [0.1531; -0.1660; 0.3185]` e `nr = 1.5940`

Esercizio 14

Il nome del file (e quindi della funzione) è stato rinominato in newtonSis.m per evitare di avere due file con lo stesso nome.

```
function [x, nit] = newtonSis(fun, x0, tol, maxit)
```

```
2  %
3  % [x, nit] = newtonSis(fun, x0, tol, maxit)
4  %
5  % Metodo di newton per la risoluzione di sistemi di
       equazioni non lineari
6  %
7  % Input:
8  %   fun: [f, jacobian] = fun(x) se il sistema da
       risolvere e f(x)=0
9  %      f: gradiente di una funzione f(x) di cui
       vogliamo approssimare una radice
10 %      jacobian: matrice Hessiana di f(x);
11 %   x0: vettore valori iniziali
12 %   tol: tolleranza
13 %   maxit: numero massimo di iterazioni
14 %
15 % Output:
16 %   x: soluzione del sistema
17 %   nit: numero di iterazioni eseguite
18
19 if nargin < 2
20   error('Numero di argomenti insufficiente');
21 elseif nargin == 2
22   tol = 1e-6;
23   maxit = 1000;
24 elseif nargin == 3
25   maxit = 1000;
26 elseif maxit <= 0 || tol <= 0
27   error('Dati in ingresso errati');
28 end
29
30 x0 = x0(:);
31 nit = maxit;
32 for i = 1:maxit
33   [f, jacobian] = fun(x0);
34   delta = mialum(jacobian, -f);
35   x = x0 + delta;
36   if norm(delta ./ (1 + abs(x0)), inf) <= tol
37     nit = i;
38     break
39   end
40   x0 = x;
41 end
42 return
43
44 function x = mialum(A, b)
```

```
% x = mialum(A, b);
%
% Risolve il sistema lineare Ax = b con
   fattorizzazione LU senza pivoting parziale.
%
%   Input:
%   A - matrice dei coefficienti
%   b - vettore dei termini noti
%
% Output:
%   x - vettore soluzione

[m, n] = size(A);
if m ~= n
  error('La matrice non e quadrata!');
end
[m, o] = size(b);
if m ~= n || o ~= 1
  error('Dimensione del vettore e della matrice non
     compatibili')
end

for i = 1:n-1
  if A(i, i) == 0
    error('Matrice singolare');
  end
  for j = i+1:n
    A(j, i) = A(j, i) / A(i, i);
    A(j, i+1:n) = A(j, i+1:n) - A(j, i) * A(i, i+1:n
       );
  end
end

x = b(:);
for i = 2:n
  x(i:n) = x(i:n) - A(i:n, i-1) * x(i-1);
end

for i = n:-1:1
  x(i) = x(i) / A(i, i);
  x(1:i-1) = x(1:i-1) - A(1:i-1, i) * x(i);
end
return
```

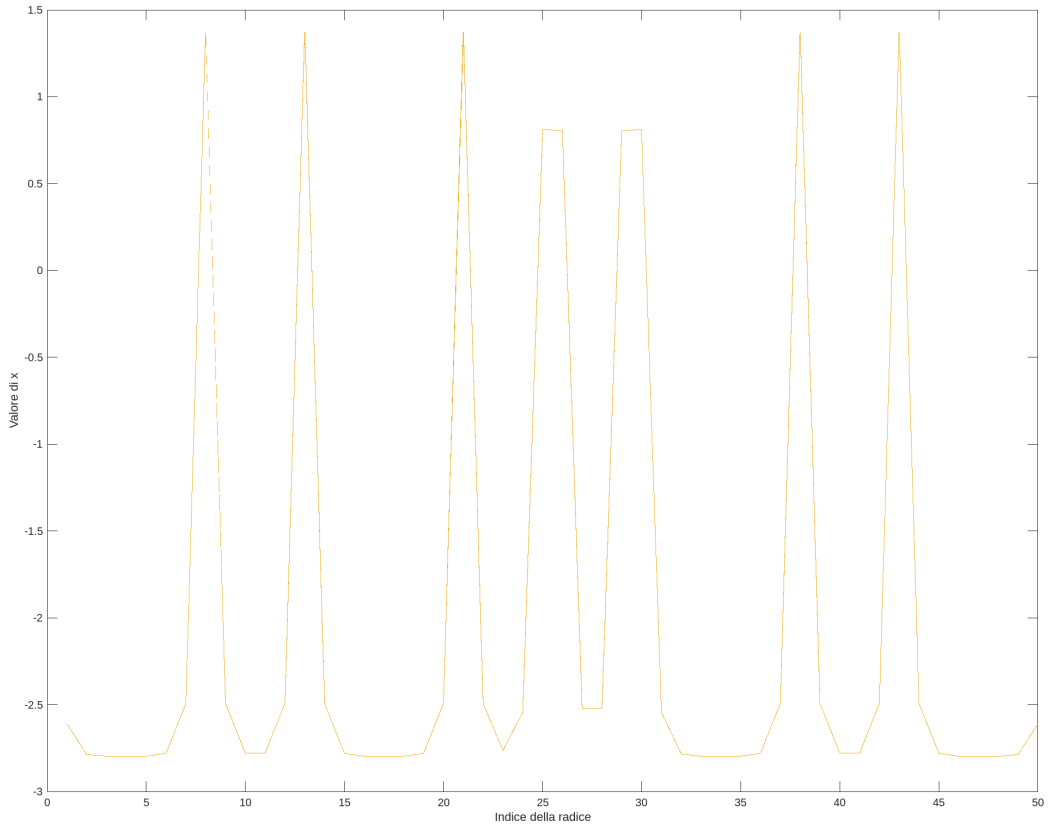
Esercizio 15

```
function [f, jacobian] = fun(x)
%
% [f, jacobian] = fun(x);
%
% Calcola il gradiente e la matrice Hessiana di f(x)
%
% Input:
%   x - vettore delle ascisse
%
% Output:
%   f - gradiente della funzione f(x)
%   jacobian - matrice Hessiana di f(x)

x = x(:);
n = length(x);
Q = 4 * eye(n) + diag(ones(n-1, 1), 1) + diag(ones(n-1, 1), -1);
e = ones(n, 1);
alfa = 2;
beta = -1.1;
grad = @(x) Q * x - alfa * e .* sin(alfa * x) - beta * e .* exp(-x);
Jac = @(x) Q - alfa^2 * diag(e .* cos(alfa * x)) + beta * diag(e .* exp(-x));
f = grad(x);
jacobian = Jac(x);
return
```

Utilizzando la funzione qui sopra, possiamo risolvere il sistema nonlineare e graficare i risultati ottenuti, con anche il numero di iterazioni necessarie per ottenere le soluzioni.

```
format long;
x0 = zeros(50, 1);
x = zeros(50, 3);
iterazioni = zeros(3, 1);

[x(:, 1), iterazioni(1)] = newtonSis(@fun, x0, 10.^(-3));
[x(:, 2), iterazioni(2)] = newtonSis(@fun, x0, 10.^(-8));
[x(:, 3), iterazioni(3)] = newtonSis(@fun, x0, 10.^(-13));

```

```
plot(1:50, x(:, 1));
hold on
plot(1:50, x(:, 2));
hold on
plot(1:50, x(:, 3));
hold off
xlabel('Indice della radice');
ylabel('Valore di x');
```



Tolleranza	Iterazioni
10^{-3}	699
10^{-8}	701
10^{-13}	702

Esercizio 16

```
function YQ = lagrange(X, Y, XQ)
%
% YQ = lagrange(X, Y, XQ)
%
% Calcola il polinomio interpolante in forma di
   Lagrange definito dalle
% coppie (Xi, Yi) nei punti del vettore XQ
%
% Input:
%   (X,Y): dati del problema
%   XQ: vettore in cui calcolare il polinomio
%
% Output:
%   YQ: polinomio interpolante in forma di Lagrange

n = length(X);
if length(Y) ~= n || n <= 0
  error('Dati inconsistenti');
end
if length(unique(X)) ~= n
  error('Ascisse non distinte');
end
YQ = zeros(size(XQ));
for i=1:n
  YQ = YQ + Y(i) * lin(XQ, X, i);
end
return
```

```
function L = lin(x, xi, i)
%
% L = lin(x, xi, i)
%
% Calcola il polinomio di base di Lagrange in
   funzione degli argomenti
% passati
%
% Input:
%   x: vettore in cui calcolare il polinomio
%   xi: vettore ascisse
%
% Output:
%   L: polinomio di base di Lagrange

L = ones(size(x));
```

```
n = length(xi) - 1;
xii = xi(i);
xi = xi([1:i-1, i+1:n+1]);
for k=1:n
  L = L.*(x - xi(k))/(xii - xi(k));
end
return
```

Esercizio 17

Per lo stesso motivo dell'esercizio 14, il nome del file è stato rinominato in newton0.m per evitare di avere file con lo stesso nome.

```
function YQ = newton0(X, Y, XQ)
%
% YQ = newton0(X, Y, XQ)
%
% Calcola il polinomio interpolante in forma di
   Newton definito dalle
% coppie (Xi, Yi) nei punti del vettore XQ
%
% Input:
% (X,Y): dati del problema
% XQ: matrice in cui calcolare il polinomio
%
% Output:
%   YQ: Polinomio interpolante in forma di Newton

if length(X) ~= length(Y) || length(X) <= 0
  error('Dati errati');
end

if length(unique(X)) ~= length(X)
  error('Ascisse non distinte');
end
df = difdiv(X, Y);
n = length(df) - 1;
YQ = df(n+1) * ones(size(XQ));
for i = n:-1:1
  YQ = YQ.*(XQ - X(i)) + df(i);
end
return
```

```
function df = difdiv(x, f)
```

```
%
% df = difdiv(x, f)
%
% Calcola le differenze divise sulle coppie (xi, fi)
%
% Input:
%   x: vettore delle ascisse
%   f: vettore delle ordinate
% Output:
%   df: vettore delle differenze divise

n = length(x);
if length(f) ~= n
  error('Dati errati');
end
n = n-1;
df = f;
for j=1:n
  for i = n+1:-1:j+1
    df(i) = (df(i) - df(i-1))/(x(i) - x(i-j));
  end
end
return
```

Esercizio 18

```
function yy = hermite(xi, fi, f1i, xx)
%
% yy = hermite(xi, fi, f1i, xx)
%
% Calcola il polinomio interpolante di Hermite definito dalle
% coppie (xi, yi) nei punti del vettore xx
%
% Input:
%   (xi, fi, f1i): dati del problema
%   xx: vettore in cui calcolare il polinomio
%
% Output:
%   yy: polinomio interpolante di Hermite

if length(fi) ~= length(xi) || length(xi) <= 0 || length(xi) ~= length(f1i)
  error('Dati inconsistenti');
```

```
end

if length(unique(xi)) ~= length(xi)
  error('Le ascisse non sono distinte');
end

fi = repelem(fi, 2);
for i = 1:length(f1i)
  fi(i*2) = f1i(i);
end
df = difdivHermite(xi, fi);
n = length(df)-1;
yy = df(n+1) * ones(size(xx));
for i = n:-1:1
  yy = yy.*(xx - xi(round(i/2))) + df(i);
end
return
```

```
function df = difdivHermite(X, Y)
%
% df = difdivHermite(X, Y)
%
% Calcola le differenze divise di Hermite sulle
   coppie (xi, fi)
%
% Input:
%   X: vettore delle ascisse
%   Y: vettore delle ordinate e delle derivate della
   forma [f(0) f'(0) f(1)...]
%
% Output:
%   df: vettore delle differenze divise di Hermite
%
n = length(X)-1;
df = Y;
for i = (2*n+1):-2:3
  df(i) = (df(i)-df(i-2))/(X((i+1)/2)-X((i-1)/2));
end
for j = 2:2*n+1
  for i = (2*n+2):-1:j+1
    df(i) = (df(i)-df(i-1))/(X(round(i/2))-X(round((
       i-j)/2)));
  end
end
return
```

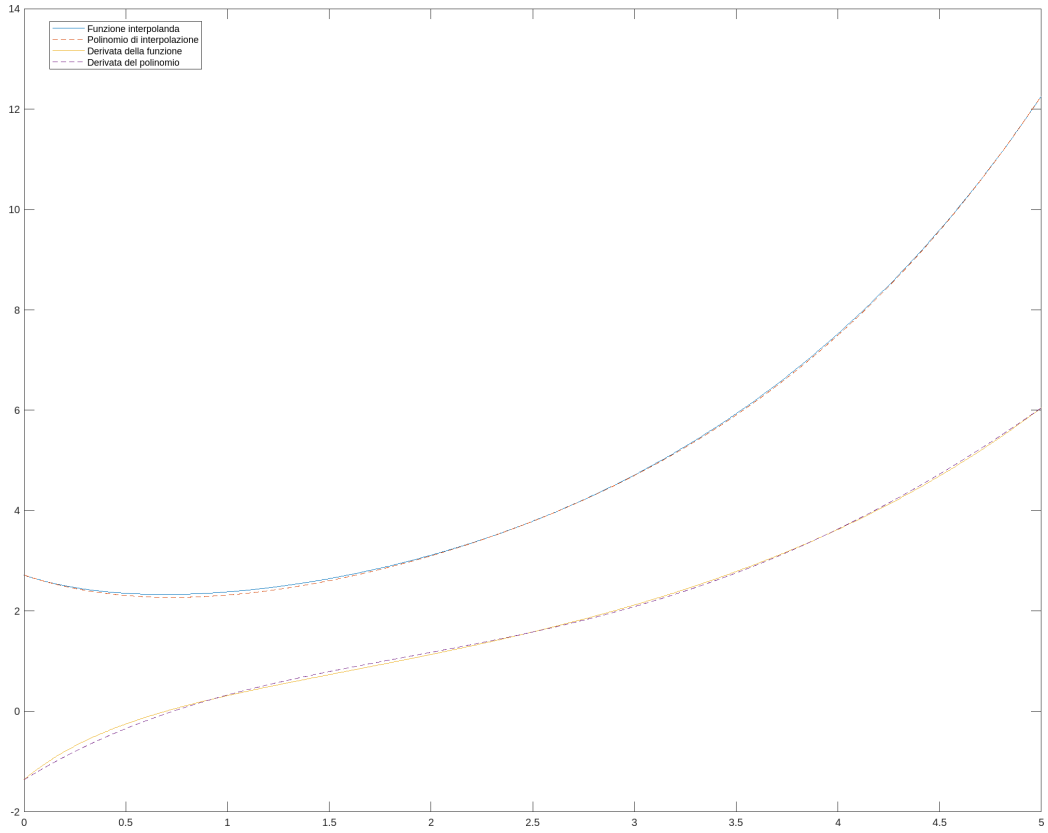
Esercizio 19

```
function dP = hornerDerivata(ascisse, coefficienti, xi)
%
% dP = hornerDerivata(ascisse, coefficienti, xi)
%
% Calcola la derivata di un polinomio in forma di Newton in un punto specifico
%
% Input:
%   ascisse - Vettore di ascisse [x0, x1, ..., xn]
%   coefficienti - Vettore dei coefficienti [a0, a1, ..., an]
%   xi - Ascissa su cui valutare la derivata
% Output:
%   dP - Valore della derivata del polinomio in xi
if nargin < 3
    error("Numero di parametri insufficienti");
end
n = length(coefficienti);
if n ~= length(ascisse)
    error("Dimensione degli input errata");
end
P = coefficienti(n);
dP = 0;
for k = n-1:-1:1
  dP = dP .* (xi - ascisse(k)) + P;
  P = P .* (xi - ascisse(k)) + coefficienti(k);
end
return
```

Esercizio 20

```
f = @(x) (exp(x/2 + exp(-x)));
df = @(x) (0.5*exp(exp(-x) - x/2).*(-2 + exp(x)));

x = linspace(0, 5, 1000);
xi = [0 2.5 5];
fi = f(xi);
f1i = df(xi);

plot(x, f(x), "DisplayName", "Funzione interpolanda");
```

```
10 hold on
11
12 plot(x, hermite(xi, fi, f1i, x),  "--","DisplayName
      ", "Polinomio di interpolazione");
13 hold on
14
15 plot(x, df(x), "DisplayName", "Derivata della
      funzione");
16 hold on
17
18 xiRaddoppiato = repelem(xi, 2);
19 fi = repelem(fi, 2);
20 for i = 1:length(f1i)
21   fi(i*2) = f1i(i);
22 end
23 dd = difdivHermite(xi, fi);
24
25 plot(x, hornerDerivata(xiRaddoppiato, dd, x), "--","
      DisplayName", "Derivata del polinomio");
26 hold off
27
28 legend("Location", "Best");
```



Esercizio 21

```
function x = chebyshev(n, a, b)
%
% x = chebyshev(n, a, b)
%
% calcola le n+1 ascisse di Chebyshev sull'
   intervallo [a, b]
%
% Input:
%   n: numero di ascisse che vogliamo calcolare
%   a, b: intervallo in cui vengono calcolate le
   ascisse di Chebyshev
%
% Output:
%   x: ascisse di Chebyshev calcolate sull'
   intervallo [a, b]
```

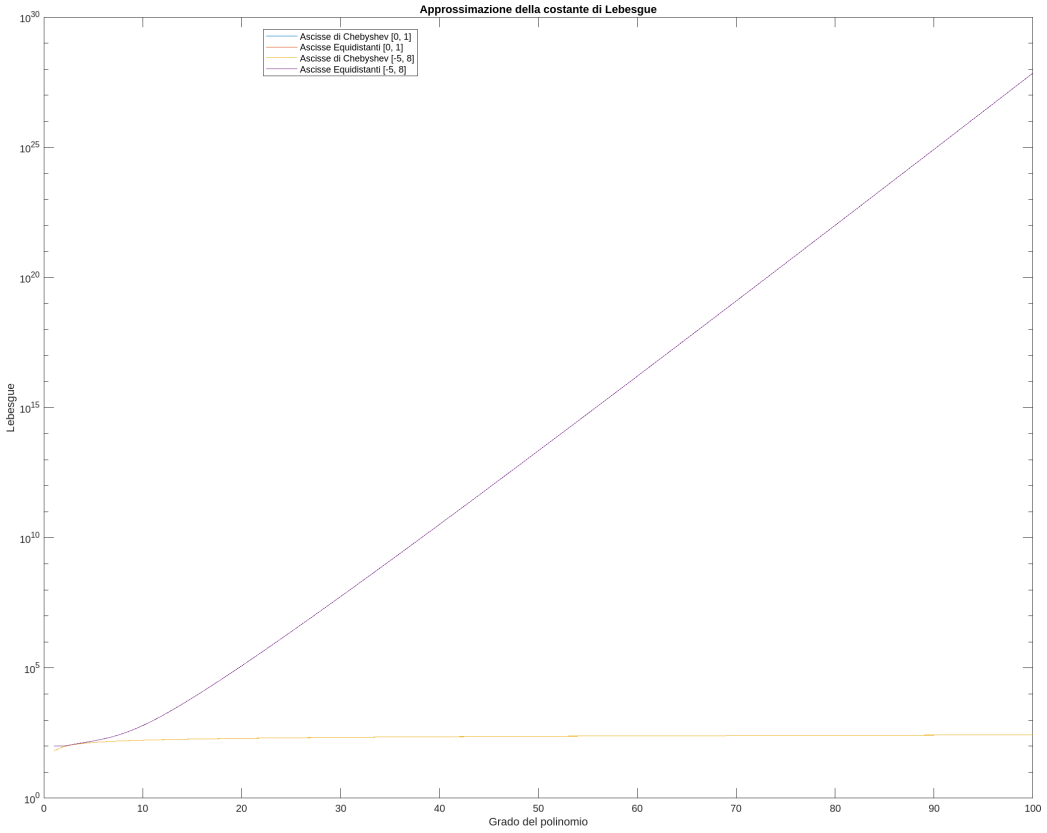
```
13
14 if a >= b || n <= 0
15   error('Dati errati');
16 end
17
18 x = (a+b)/2 + ((b-a)/2) * cos((2*[n:-1:0] + 1)/((2*(
     n+1)))*pi);
19 return
```

Esercizio 22

```
1  function ll = lebesgue(a,b, nn, type)
2  %
3  % ll = lebesgue(a, b, nn, type)
4  %
5  % restituisce le approssimazioni della costante di
     Lebesgue
6  % sull'intervallo [a, b] per i polinomi di grado
     specificato da nn
7  %
8  % Input:
9  %   a, b: intervallo
10 %   nn: vettore contenente i gradi dei polinomi per
     l'approssimazione
11 %   type: ascisse equidistanti (= 0), chebyshev (=
     1)
12 %
13 % Output:
14 %   ll: approssimazioni della costante di Lebesgue
15
16 max=10001;
17 x=linspace(a, b, max);
18 ll = nn;
19 for i=1:length(nn)
20   if type == 0
21     xi = linspace(a, b, nn(i));
22   elseif type == 1
23     xi = chebyshev(nn(i), a, b);
24   end
25   leb = zeros(1, max);
26   for j=1:nn(i)
27     leb = leb + abs(lin(x, xi, j));
28   end
29   ll(i) = norm(leb);
```

```
end
return
```

```
nn = (1:100);

semilogy(nn, lebesgue(0, 1, nn, 1), "DisplayName", "
   Ascisse di Chebyshev [0, 1]");
hold on

semilogy(nn, lebesgue(0, 1, nn, 0), "DisplayName", "
   Ascisse Equidistanti [0, 1]");
hold on

semilogy(nn, lebesgue(-5, 8, nn, 1), "DisplayName",
   "Ascisse di Chebyshev [-5, 8]");
hold on

semilogy(nn, lebesgue(-5, 8, nn, 0), "DisplayName",
   "Ascisse Equidistanti [-5, 8]");
hold off

title("Approssimazione della costante di Lebesgue");
xlabel("Grado del polinomio");
ylabel("Lebesgue");
legend("Location", "Best");
```

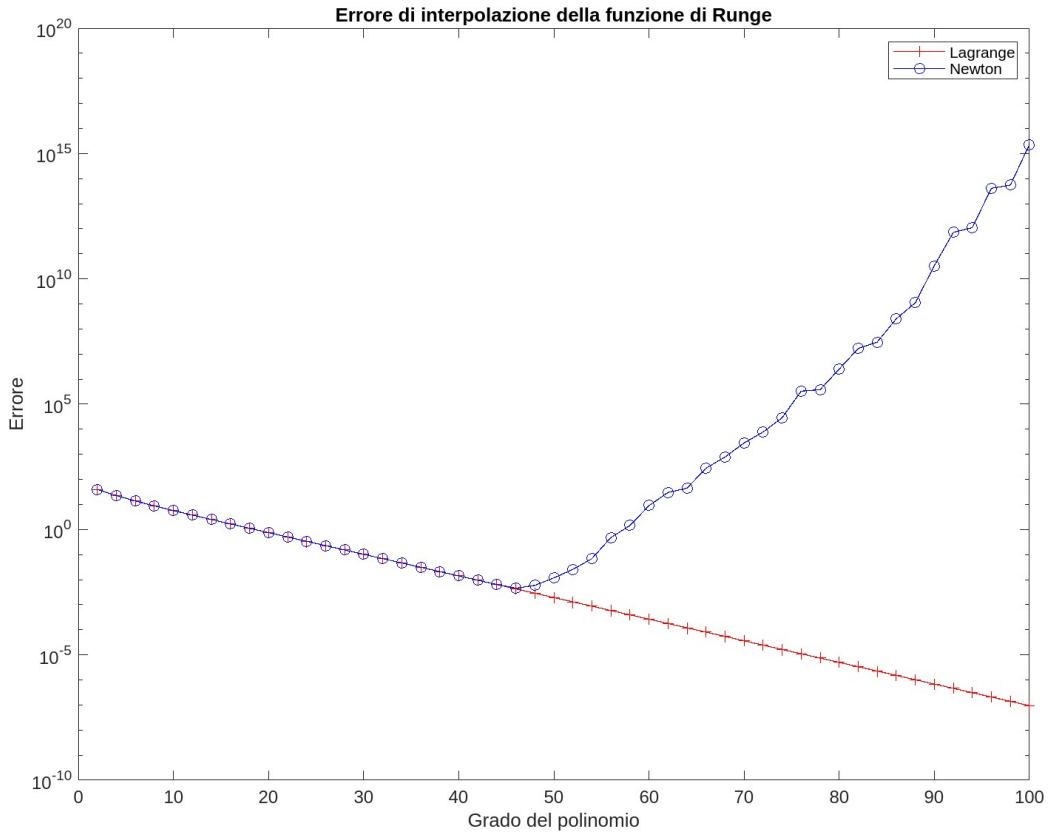


Dal grafico in figura possiamo osservare la crescita ottimale della costante di Lebesgue (logaritmica) utilizzando le ascisse di Chebyshev. Le ascisse equidistanti, al contrario, fanno crescere la costante in maniera esponenziale. Inoltre si può osservare che l'intervallo [a, b] non influisce sulla costante, infatti le linee per i due intervalli si sovrappongono perfettamente.

Esercizio 23

```
f = @(x) (1./(1+x.^2));

x = linspace(-5, 5, 10001);
y = f(x);

normLagr = (1:50);
normNewt = (1:50);

```

```
for n=1:50
  xCheb = chebyshev(2*n, -5, 5);
  yCheb = f(xCheb);

  yLag = lagrange(xCheb, yCheb, x);
  yNewt = newton0(xCheb, yCheb, x);

  normLagr(n) = norm(y-yLag);
  normNewt(n) = norm(y-yNewt);
end

semilogy((2:2:100), normLagr, 'r-+', 'DisplayName','
   Lagrange');
hold on
semilogy((2:2:100), normNewt, 'b-o', 'DisplayName','
   Newton');
hold off
title("Errore di interpolazione della funzione di
   Runge con ascisse di Chebyscev");
xlabel("Grado del polinomio");
ylabel("Errore di Interpolazione");
legend
```



Per quanto riguarda l'interpolazione di Lagrange abbiamo un errore che diminuisce man mano che il grado del polinomio aumenta. Le ascisse di Chebyshev riducono significativamente il fenomeno di Runge. Il metodo di newton, invece, inizialmente sembra avere lo stesso comportamento di Lagrange, ma da un certo punto in poi, ha un errore che aumenta con il grado del polinomio. L'interpolazione di Lagrange, sembrerebbe quindi più efficiente e accurata per gradi di polinomio elevati quando si utilizzano le ascisse di Chebyshev.

Esercizio 24

```
1 function YQ = spline0(X, Y, XQ)
2 %
3 % YQ = spline0(X, Y, XQ)
4 %
```

```
% La function calcola la spline cubica naturale interpolante e
% restituisce il valore assunto dalla spline sulle ascisse XQ
%
% Input:
%   X: vettore delle ascisse di interpolazione
%   Y: vettore dei valori della funzione assunti sulle ascisse interpolanti
%   XQ: vettore delle ascisse dove si calcola il valore della spline
%
% Output:
%   YQ: vettore delle ordinate calcolate sulle ascisse
%
n = length(X);
if length(Y) ~= n
  error('Dati errati');
end
n = n-1;
h(1:n) = X(2:n+1) - X(1:n);
b = h(2:n-1)./(h(2:n-1) + h(3:n));
c = h(2:n-1)./(h(1:n-2) + h(2:n-1));
a(1:n-1) = 2;
df = difdivSpline(X, Y, 3);
m = tridia(a, b, c, 6*df);
m = [0, m, 0];
YQ = zeros(size(XQ));
j = 1;
for i=2:n+1
  ri = Y(i-1) - (h(i-1)^2)/6 * (m(i-1));
  qi = (Y(i) - Y(i-1))/h(i-1) - h(i-1)/6*(m(i) - m(i-1));
  while j <= length(XQ) && XQ(j) <= X(i)
    YQ(j) = ((XQ(j) - X(i-1))^3 * m(i) + (X(i) - XQ(j))^3 * m(i-1))/(6*h(i-1)) + qi*(XQ(j) - X(i-1)) + ri;
    j = j+1;
  end
end
return
```

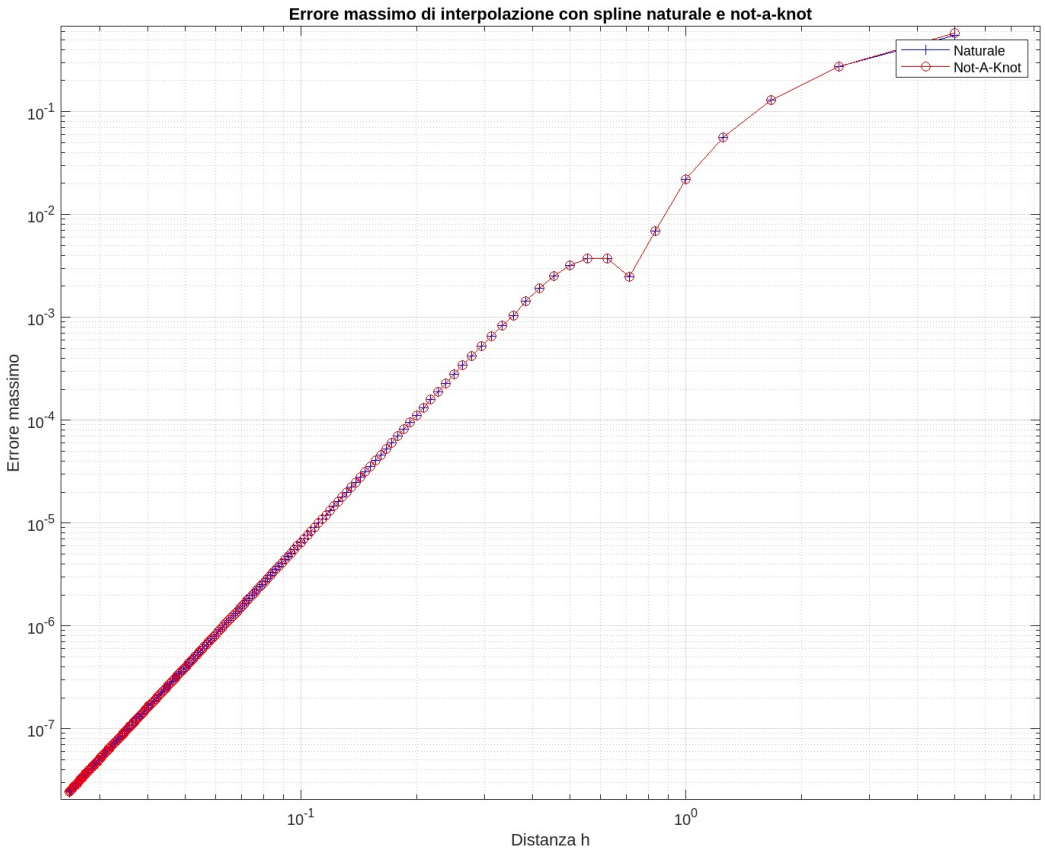
```
function x = tridia(a, b, c, x)
%
% x = tridia(a, b, c, x)
```

```
%
% risolve il sistema tridiagonale
% b(i)*x(i-1) + a(i)*x(i) + c(i)*x(i+1) = d(i), i =
   1...n
% con x(0)=x(n+1)=0

n = length(a);
for i = 1:n-1
  b(i) = b(i)/a(i);
  a(i+1) = a(i+1) - b(i)*c(i);
  x(i+1) = x(i+1) - b(i)*x(i);
end
  x(n) = x(n)/a(n);
for i = n-1:-1:1
  x(i) = (x(i) - c(i)*x(i+1))/a(i);
end
return
```

```
function df = difdivSpline(X, Y, it)
%
% df = difdivSpline(X, Y, it)
%
% Calcola le differenze divise sulle coppie (xi, fi)
% fermandosi alla it-esima iterazione
%
% Input:
%   x: vettore delle ascisse
%   f: vettore delle ordinate
%   it: numero di iterazioni
% Output:
%   df: vettore delle differenze divise

n = length(X);
if length(Y) ~= n
  error('Dati errati');
end
n = n-1;
df = Y;
for j=1:it-1
  for i = n+1:-1:j+1
    df(i) = (df(i) - df(i-1))/(X(i) - X(i-j));
  end
end
df = df(1, it:n+1);
return
```

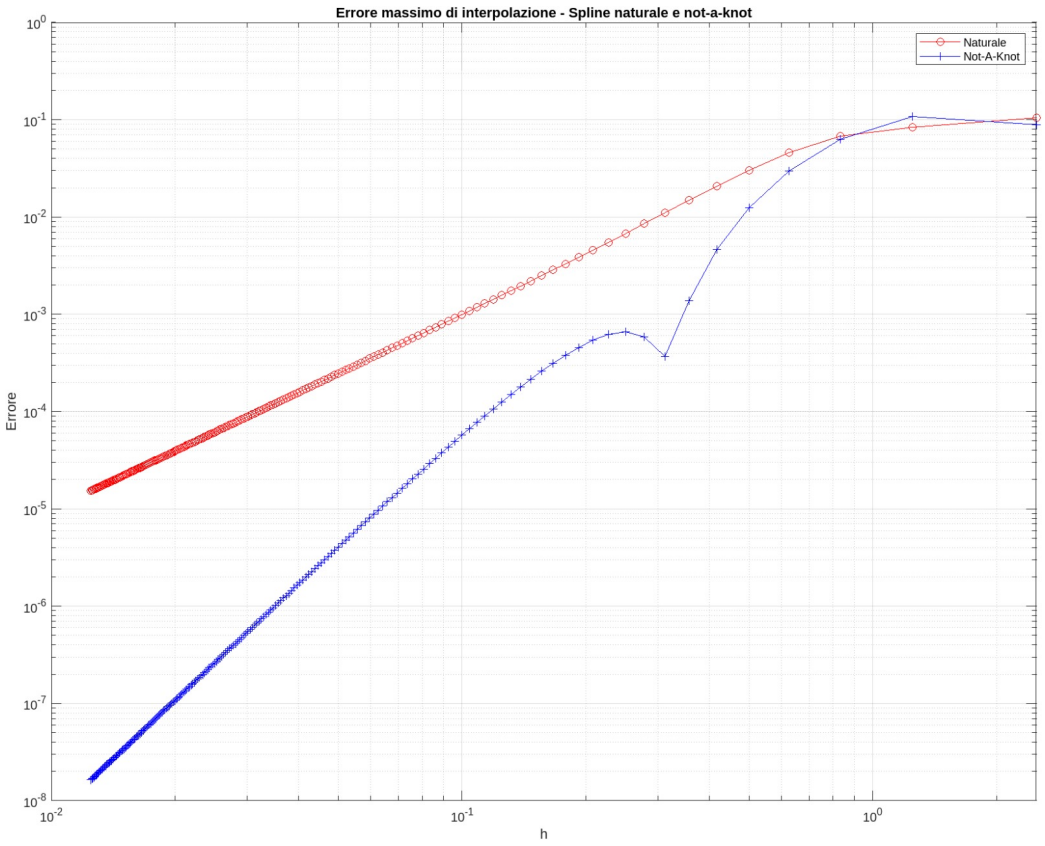
Esercizio 25

```
f = @(x) 1 ./ (1 + x .^ 2);
a = -10;
b = 10;
x = linspace(a, b, 10001);
fx = f(x);
err_nat = zeros(1, 200);
err_nak = zeros(1, 200);
h = zeros(1, 200);

index = 1;
for n = 4:4:800
  xi = linspace(a, b, n+1);
  fi = f(xi);

  sn_i = spline0(xi, fi, x);

  snak_i = spline(xi, fi, x);

  err_nat(index) = max(abs(fx - sn_i));
  err_nak(index) = max(abs(fx - snak_i));

  h(index) = 20 / n;
  index = index + 1;
end

figure;
loglog(h, err_nat, 'b-+', h, err_nak, 'r-o');
xlabel('Distanza h');
ylabel('Errore massimo');
title('Errore massimo di interpolazione con spline
   naturale e not-a-knot');
legend('Naturale', 'Not-A-Knot');
grid on;
```



Con il diminuire di h, l'errore di approssimazione delle due spline sembra tendere a diventare sempre più simile, fino ad essere praticamente indistinguibile.

Esercizio 26

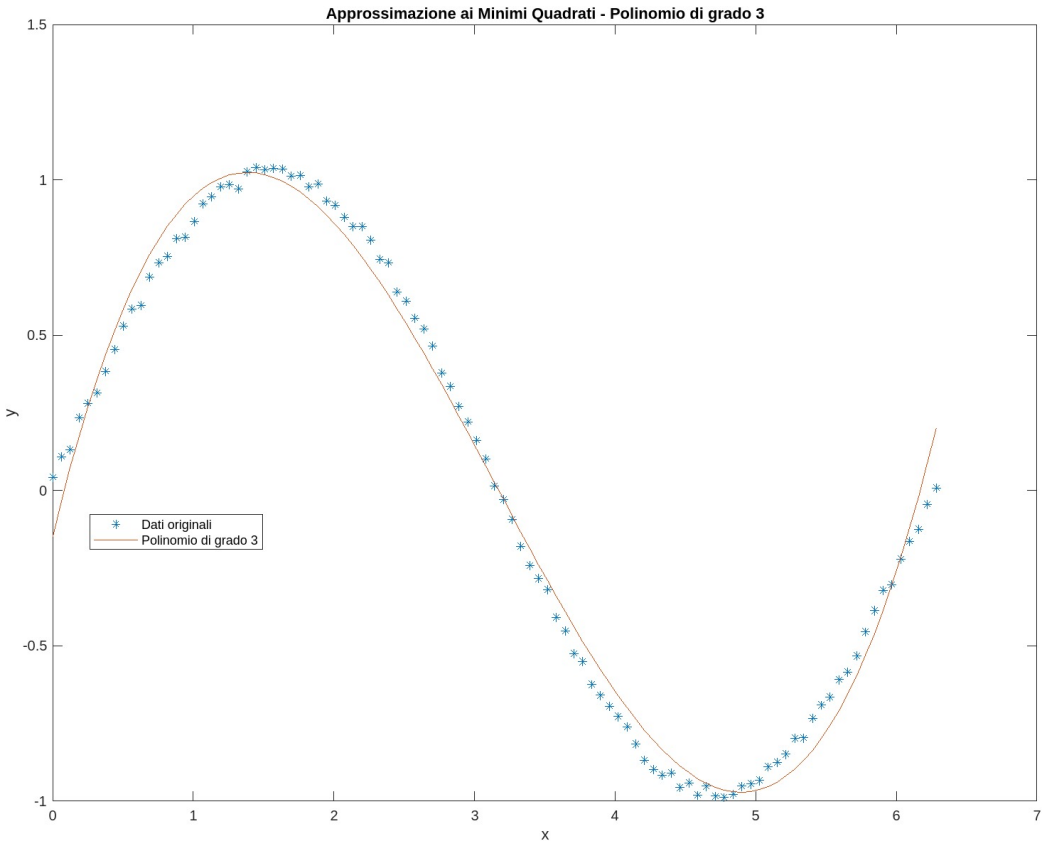


Il codice dell'esercizio 26 è equivalente a quello dell'esercizio 26, con la sola differenza che l'intervallo di interpolazione è stato modificato in $[0, 10]$ e la distanza $h = \frac{10}{n}$.
Stavolta, si può notare che la spline Not-a-Knot comporta una decrescita più rapida dell'errore all'aumentare del numero dei sottointervalli, mentre la spline naturale commette un errore maggiore rispetto a quella Not-a-Knot (e rispetto all'esercizio precedente).

Esercizio 27

```
1 rng(0)
2 xi = linspace(0, 2*pi, 101);
```

```
yi = sin(xi) + rand(size(xi))*.05;

n = length(xi);

coefficientsMatrix = zeros(n, 4);
for i = 1:n
  coefficientsMatrix(i, :) = [xi(i)^3, xi(i)^2, xi(i), 1];
end

a = coefficientsMatrix \ yi';
yi_pol = polyval(a, xi);

figure;
plot (xi, yi, "*", "DisplayName", "Dati originali");
hold on;
plot (xi , yi_pol , "-" , "DisplayName" , "Polinomio di grado 3");
legend ("Location" , "Best");
xlabel ("x");
ylabel ("y");
title ("Approssimazione ai Minimi Quadrati - Polinomio di grado 3");
```



Esercizio 28

```
function w = newtonCotesPesi(n)
%
% w = newtonCotesPesi(n)
%
% Function che restituisce i pesi della quadratura
% della formula di Newton-Cotes di grado n
%
% Input:
%   n: grado della formula
%
% Output:
%   w: pesi della quadratura

if n < 1 || n > 9 || n == 8
  error("Input errato");
```

```
end
w = zeros(1, n+1);
for i=0:n
  d = i - [0:i-1 i+1:n];
  den = prod(d);
  a = poly([0:i-1 i+1:n]);
  a = [a./((n+1):-1:1) 0];
  num = polyval(a, n);
  w(i+1) = num / den;
end
return
```

Grado	Pesi
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	$\frac{95}{288}, \frac{125}{96}, \frac{125}{144}, \frac{125}{96}, \frac{95}{288}$
6	$\frac{41}{140}, \frac{54}{35}, \frac{27}{140}, \frac{68}{35}, \frac{27}{140}, \frac{54}{35}, \frac{41}{140}$
7	$\frac{108}{355}, \frac{810}{559}, \frac{343}{640}, \frac{649}{536}, \frac{649}{536}, \frac{343}{640}, \frac{810}{559}, \frac{108}{355}$
9	$\frac{130}{453}, \frac{419}{265}, \frac{23}{212}, \frac{307}{158}, \frac{213}{367}, \frac{213}{367}, \frac{307}{158}, \frac{23}{212}, \frac{419}{265}, \frac{130}{453}$

Esercizio 29

```
function [If , err] = composita (fun, a, b, k, n)
%
% [If , err] = composita (fun, a, b, k, n)
%
% Input:
%   fun: funzione integranda
%   a,b: estremi sinistro e destro dell'intervallo
   di integrazione
%   k: grado della formula di quadratura composita
   di Newton-Cotes
```

```
9  %   n: numero di sottointervalli in cui suddividere
       l'intervallo di integrazione
10 % Output:
11 %   If: approssimazione dell'integrale ottenuta
12 %   err: stima dell'errore di quadratura
13
14 if a > b
15   error("Estremi intervallo non validi");
16 end
17 if k < 1
18   error("Grado k errato");
19 end
20 if(mod(n, k) ~= 0 || mod(n/2, 2)~= 0)
21   error("n deve essere un multiplo pari di k!");
22 end
23
24 tol = 1e-3;
25 mu = 1 + mod(k,2);
26 c = calcolaCoefficientiGrado(k);
27 x = linspace(a, b, n +1);
28 fx = feval(fun, x);
29 h = (b - a) / n;
30 If1 = h * sum(fx(1: k+1) .* c(1: k+1));
31 err = tol + eps;
32 while tol < err
33   n = n * 2;
34   x = linspace(a ,b , n +1);
35   fx(1:2:n+1) = fx(1:1:n/2+1);
36   fx(2:2:n) = fun(x(2:2:n));
37   h = (b - a)/ n;
38   If = 0;
39   for i = 1:k+1
40     If = If + h * sum(fx(i : k : n)) * c(i);
41   end
42   If = If + h * fx(n+1) * c(k+1);
43   err = abs(If - If1)/(2^(k + mu)-1);
44   If1 = If;
45 end
46 return
47
48 function coef=calcolaCoefficientiGrado(n)
49 if(n<=0)
50   error('Valore del grado della formula di Newton-
       Cotes non valido')
51 end
52 coef=zeros(n+1,1);
```

```
53 if (mod(n,2) == 0)
54   for i=0:n/2-1
55     coef(i+1)=calcolaCoefficienti(i,n);
56   end
57   coef(n/2+1)=n-sum(coef)*2;
58   coef((n/2)+1:n+1)=coef((n/2)+1:-1:1);
59 else
60   for i=0:round(n/2,0)-2
61     coef(i+1)=calcolaCoefficienti(i,n);
62   end
63   coef(round(n/2,0))=(n-sum(coef)*2)/2;
64   coef(round(n/2,0)+1:n+1)=coef(round(n/2,0):-1:1);
65 end
66 return
67
68 function cin=calcolaCoefficienti(i,n)
69 d=i-[0:i-1 i+1:n];
70 den=prod(d);
71 a=poly([0:i-1 i+1:n]);
72 a=[a./((n+1):-1:1) 0];
73 num=polyval(a,n);
74 cin=num/den;
75 return
```

Esercizio 30

`f = @(x)(exp(3*x))`
`intF = @(x)(exp(3*x)/3)`
Il valore dell'integrale calcolato con il comando `intF(1)-intF(0)` restituisce 6.36184564106256.

k	Approssimazione Integrale	Errore Stimato	Errore Vero
1	6.36391641954463	0.000887245471037801	0.00207077848206971
2	6.36184618011221	1.1533902044241e-06	5.39049648473622e-07
3	6.36184685336076	5.84856195072436e-07	1.21229820138069e-06
6	6.36184564106414	3.12444383576431e-12	1.57918123022682e-12