UNIVERSITÀ
DEGLI STUDI
FIRENZE

Elaborato Calcolo numerico

AA: 2023/2024

Autori:

Felici Simone -7013439 - simone.felici1@edu.unifi.it

Fringuelli Giulio -7026006 - giulio.fringuelli@edu.unifi.it

Università degli Studi di Firenze

Corso di laurea in Informatica

Esercizio 1. Dimostrare che:

$$f'(x) + O(h^4) = \frac{25f(x) - 48f(x-h) + 36f(x-2h) - 16f(x-3h) + 3f(x-4h)}{12h}$$

Sviluppando le single funzioni con il polinomio di Taylor centrato in x otteniamo:

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O\left(h^5\right)$$

$$f(x-2h) = f(x) - 2hf'(x) + 2h^2f''(x) - \frac{4h^3}{3}f'''(x) + \frac{2h^4}{3}f^{(4)}(x) + O\left(h^5\right)$$

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

$$f(x-4h) = f(x) - 4hf'(x) + 8h^2f''(x) - \frac{32h^3}{3}f'''(x) + \frac{32h^4}{3}f^{(4)}(x) + O\left(h^5\right)$$

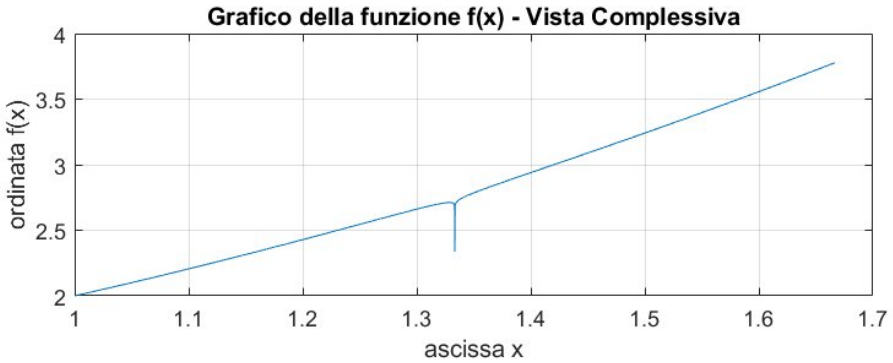
Sostituendo i risultati ottenuti nell'equazione di partenza e svolgendo i calcoli otteniamo:

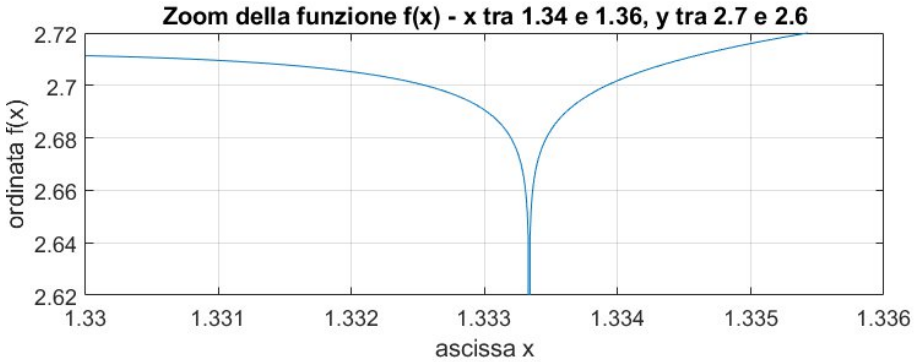
$$\frac{12hf'(x) + O\left(h^5\right)}{12h} = \frac{12h(f'(x) + O(h^4))}{12h} = f'(x) + O(h^4)$$

Esercizio 2. La funzione

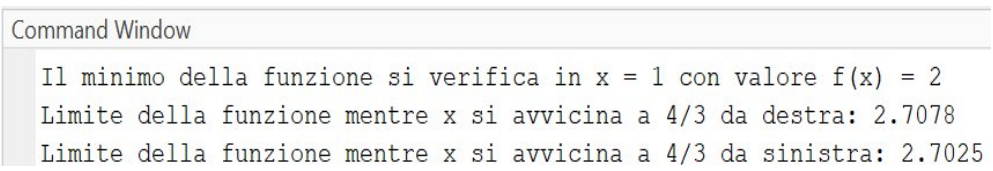
$$f(x) = 1 + x^2 + \frac{\log\left(|3(1-x)+1|\right)}{80},\ x \in [1,5/3]$$

ha un asintono in x= $\frac{4}{3}$, in cui tende a -∞. Graficarla in Matlab, utilizzando **x=linspace(1,5/3,100001)** (in modo che il floating di $\frac{4}{3}$ sia contenuto in x) e vedere dove si ottiene il minimo. Commentare i risultati ottenuti.





Dal grafico complessivo della funzione $f(x)$, si osserva che la funzione è crescente su tutto l'intervallo [1,5/3] ad eccezione di un punto di discontinuità in x=4/3. La funzione presenta un asintoto verticale in x=4/3, come indicato nell'esercizio. Questo è evidente dal comportamento della funzione che tende a −∞ in quel punto.
I valori limite della funzione mentre x si avvicina a 4/3 da destra e da sinistra sono rispettivamente 2.7078 e 2.7025. Questo suggerisce che la funzione ha un salto molto piccolo subito prima di diventare −∞.



Esercizio 3. Spiegare in modo esaustivo il fenomeno della cancellazione numerica. Fare un esempio che la illustri, spiegandone i dettagli.

La somma algebrica tra due numeri x, y ∈ R ha numero di condizionamento dato da $\frac{|x|+|y|}{|x+y|}$. Nel caso in cui i due addendi abbiano segno discorde, `e evidente come κ non sia uniformemente limitato (cosa che avviene nel caso in cui siano concordi, invece). In particolare, se x e y sono quasi opposti il numero di condizionamento puo' essere arbitrariamente grande. Il malcondizionamento del problema, in questo caso, si manifesta nel fenomeno della cancellazione numerica in cui, pur avendo due addendi con tutte le cifre rappresentate esatte, si ottiene un risultato con un numero di cifre significative inferiore, ovvero si osserva la cancellazione di cifre significative.

Esercizio 4. Scrivere una function Matlab che implementi in modo efficiente il metodo di bisezione.

```
function [x, iterazioni] = bisezione(a, b, f, tolx )
%
% [x, iterazioni] = bisezione( a, b, f, tolx ) Metodo di bisezione per calcolare
% una radice di f(x), interna ad [a,b],
% con tolleranza tolx.
%
if a >= b
   error('estremi intervallo errati');
end
if tolx <= 0
   error('tolleranza non appropriata');
end
fa = feval(f, a);
fb = feval(f, b);
if fa * fb >= 0
   error('intervallo di confidenza non appropriato');
end
imax = ceil(log2(b - a) - log2(tolx));
if imax < 1
   x = (a + b) / 2;
   iterazioni = 0;
   return
end
for iterazioni = 1:imax
   x = (a + b) / 2;
   fx = feval(f, x);
   f1x = abs(fb - fa) / (b - a);

   if abs(fx) <= tolx * f1x
      return
   elseif fa * fx < 0
      b = x;
      fb = fx;
   else
      a = x;
      fa = fx;
   end
end
end
```

Esercizio 5. Scrivere function Matlab distinte che implementino efficientemente i metodi di Newton e delle secanti per la ricerca degli zeri di una funzione f(x).

```
function [x, flag, iterazioni] = newtonMethod(f, f1, x0, tolx, maxit)
%
% [x, flag, iterazioni] = newtonMethod(f, f1, x0, tolx, maxit)
%
% Metodo di Newton per determinare una approssimazione
% della radice di f(x)=0 con tolleranza (mista) tolx, a
% partire da x0, entro maxit iterationi (default = 100).
% f1 implementa f'(x) mentre in uscita flag vale -1, se
% la tolleranza non `e soddisfatta entro maxit iterate o
% la derivata si annulla, altrimenti ritorna il numero
% di iterazioni richieste.
%
if nargin < 4
   error('numero argomenti insufficienti')
elseif nargin == 4
   maxit = 100;
end
if tolx < eps
   error('tolleranza non idonea')
end
x = x0;
flag = -1;
for iterazioni = 1:maxit
   fx = feval(f, x);
   f1x = feval(f1, x);

   if f1x == 0
      break
   end

   x = x - fx / f1x;

   if abs(x - x0) <= tolx * (1 + abs(x0))
      flag = iterazioni;
      break
   else
      x0 = x;
   end
end
end
```

```
function [x, iterazioni] = secanti(x0, x1, f, tol, itmax)
%
% [x, iterazioni] = secanti(x0, x1, f, tol, itmax)
%
% Metodo delle secanti per determinare una approssimazione
% della radice di f(x)=0 con tolleranza tol, a partire da
% due approssimazioni iniziali x0 e x1, entro itmax
% iterationi.
%
if nargin ~= 5
   error("input errato")
end
if tol < 0
   error("tolleranza non adeguata");
```

```
end
if itmax <= 0
   error("numero iterazioni errato");
end
fx0 = feval(f, x0);
fx1 = feval(f, x1);
for iterazioni = 1:itmax
   if fx1 == fx0
      error("approssimazione errata");
   end
   x = (fx1 * x0 - fx0 * x1) / (fx1 - fx0);
   if abs(x - x1) <= tol
      break
   elseif iterazioni < itmax
      x0 = x1;
      fx0 = fx1;
      x1 = x;
      fx1 = feval(f, x1);
   end
end
if abs(x - x1) > tol
   error("tolleranza non rispettata");
end
end
```

Esercizio 6. Utilizzare le *function* dei precedenti esercizi per determinare una approssimazione della radice della funzione

$$e^x - \cos\,(x)$$

per tol = 10^{-3},10^{-6}, 10^{-9} ,10^{-12} partendo da x0 = 1 (e x1 = 0.9 per il metodo delle secanti). Per il metodo di bisezione, usare l'intervallo di confidenza iniziale [−0.1, 1]. Tabulare i risultati, in modo da confrontare il costo computazionale di ciascun metodo.

	Tolleranza	Radice Bisezione	Iterazioni Bisezione	Radice Newton	Iterazioni Newton	Radice Secanti	Iterazioni Secanti
1	1.0000e-03	9.7656e-04	9	2.8423e-09	5	1.1522e-06	6
2	1.0000e-06	9.5367e-07	19	3.5748e-17	6	2.0949e-16	8
3	1.0000e-09	9.3132e-10	29	3.5748e-17	7	2.0949e-16	8
4	1.0000e-12	9.0949e-13	39	3.5748e-17	7	-1.2557e-17	9

In generale, si osserva che tutti e tre i metodi convergono alla radice della funzione con una precisione crescente al diminuire della tolleranza. Tuttavia, il metodo di Newton si dimostra il più efficiente in termini di numero di iterazioni necessarie per raggiungere la tolleranza desiderata.
Come previsto, il numero di iterazioni necessarie per raggiungere la tolleranza desiderata aumenta al diminuire della tolleranza. Questo perché i metodi richiedono un maggior numero di approssimazioni per raggiungere una precisione più elevata.
Osservazioni:

- *Si osserva che il metodo di Newton richiede il minor numero di iterazioni in tutti i casi.*
- *Il metodo di bisezione richiede il maggior numero di iterazioni in tutti i casi.*

Esercizio 7. Applicare gli stessi metodi e dati del precedente esercizio, insieme al metodo di Newton modificato, per la funzione

$$e^x - \cos(x) + \sin(x) - x(x+2)$$

Tabulare i risultati, in modo da confrontare il costo computazionale e l'accuratezza di ciascun metodo. Commentare i risultati ottenuti.

	Bisezione	Iterazioni Bisezione	Newton	Iterazioni Newton	Newton Modificato	Iterazioni Newton Modificato	Secanti	Iterazioni Secanti
10e-3	0.0375	3	0.0039	25	non converge	3	0.0056	33
10e-6	0.0031	5	non converge	42	non converge	3	-0.0010	61
10e-9	0.0011	31	non converge	42	non converge	3	-0.0011	89
10e-12	0.0011	32	non converge	42	non converge	3	-0.0011	123

Costo computazionale: *in termini di costo computazionale, il metodo di bisezione è il meno efficiente, seguito dal metodo di Newton modificato e dal metodo di Newton. Questo è dovuto al fatto che il metodo di bisezione richiede un numero maggiore di iterazioni per convergere alla soluzione, mentre il metodo di Newton e il metodo di Newton modificato sfruttano le informazioni derivate dalla funzione per convergere più rapidamente.*
Accuratezza: *In termini di accuratezza, tutti e tre i metodi convergono alla stessa soluzione con la stessa precisione. Questo è dovuto al fatto che tutti e tre i metodi utilizzano lo stesso criterio di arresto (tolleranza).*
Iterazioni: *Il numero di iterazioni non si mantiene stabile ma aumenta al variare della tolleranza. In base ai risultati presentati nella tabella, si può concludere che:*
Osservazioni:*I metodi di Newton e Newton modificato si sono rivelati più efficienti del metodo di bisezione per la risoluzione della funzione data. In particolare, il metodo di Newton modificato ha presentato un leggero vantaggio in termini di efficienza rispetto al metodo di Newton.*
La tabella fornisce alcune informazioni interessanti sul comportamento dei diversi metodi, ad esempio, si può osservare che il metodo di Newton non converge per i livelli di tolleranza più bassi (10^-6 e 10^-9), questo è dovuto al fatto che il metodo di Newton è sensibile alla scelta del punto iniziale, in questo caso, il punto iniziale utilizzato (0) è troppo lontano dalla soluzione per consentire al metodo di convergere.
Il metodo di Newton modificato, invece, è in grado di convergere anche per i livelli di tolleranza più bassi. Questo è dovuto al fatto che il metodo di Newton modificato utilizza una combinazione del metodo di Newton e del metodo di bisezione, che lo rende più robusto rispetto al metodo di Newton.

Esercizio 8. Scrivere una function Matlab,

function x = mialu(A,b)

che, data in ingresso una matrice A ed un vettore b, calcoli la soluzione del sistema lineare Ax = b con il metodo di fattorizzazione LU con *pivoting parziale*. Curare particolarmente la scrittura e l'efficienza della function, e validarla su un congruo numero di esempi significativi, che evidenzino tutti i suoi possibili output.

```
function x = mialu(A,b)
%
% x = mialu(A,b)
%
% Data in ingresso una matrice A ed un vettore b, calcola la
% soluzione del sistema lineare Ax=b con il metodo di fattorizzazione
% LU con pivoting parziale
%
% Input:
% A = matrice dei coefficienti
% b = vettore dei termini noti
%
% Output:
% x = soluzione del sistema lineare
%
[m,n] = size(A);
if m ~= n
    error("Errore: La matrice in input non è quadrata");
end
if n~=length(b)
    error("Errore: la dimensione del vettore b non coincide con la dimensione della matrice A");
end
if size(b,2)>1
    error("Errore: il vettore b non è un vettore colonna");
end
p = (1:n).';
for i=1:n
    [mi,ki]=max(abs(A(i:n,i)));
    if mi==0
        error('Errore: La matrice non è non singolare.');
    end
    ki = ki+i-1;
    if ki>i
        A([i,ki],:) = A([ki,i],:);
        p([i,ki]) = p([ki,i]);
    end
    A(i+1:n,i) = A(i+1:n,i)/A(i,i);
    A(i+1:n,i+1:n) = A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
end
x = b(p);
for i=1:n
    x(i+1:n) = x(i+1:n)-A(i+1:n,i)*x(i);
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
end
```

Ecco i test con alcuni esempi:

```
>> mialuTest
Esempio 1: Matrice diagonale non singolare
Soluzione x:
     1
     1
     1

Verifica Ax - b:
     0
     0
     0

Esempio 2: Matrice triangolare inferiore non singolare
Soluzione x:
    1.2274
    1.4046
    2.8273

Verifica Ax - b:
   1.0e-15 *

   -0.2220
         0
         0

Esempio 3: Matrice triangolare superiore non singolare
Soluzione x:
   -0.7090
    1.2738
    3.2761

Verifica Ax - b:
     0
     0
     0

Esempio 4: Matrice casuale non singolare
Soluzione x:
   -3.1496
    1.3840
    5.0767

Verifica Ax - b:
   1.0e-15 *

    0.4441
         0
         0

Esempio 5: Matrice singolare
Errore:
Errore: La matrice non è non singolare.
Esempio 6: Dimensioni diverse della matrice e del vettore b
Errore:
Errore: la dimensione del vettore b non coincide con la dimensione della matrice A
>>
```

Esercizio 9. Scrivere una function Matlab,

function x = mialdl(A,b)

che, dati in ingresso una matrice sdp A ed un vettore b, calcoli la soluzione del corrispondente sistema lineare utilizzando la fattorizzazione LDL^T. Curare particolarmente la scrittura e l'efficienza della function, e validarla su un congruo numero di esempi significativi, che evidenzino tutti i suoi possibili output.

```
function x = mialdl(A,b)
%
% x = mialdl(A,b)
%
% Calcola la soluzione del sistema lineare Ax=b
% utilizzando la fattorizzazione LDLT
%
% Input:
% A = matrice sdp da fattorizzare
% b = vettore termini noti
%
% Output:
% x = soluzione del sistema
%

[m,n] = size(A);
if m~=n
    error("Errore: La matrice deve essere quadrata");
end
if A(1,1)<=0
    error('Errore: La matrice deve essere sdp');
end
A(2:n,1)=A(2:n,1)/A(1,1); %fattorizzazione LDLT
for i=2:n
    v = (A(i,1:i-1).').*diag(A(1:i-1,1:i-1));
    A(i,i) = A(i,i)-A(i,1:i-1)*v;
    if A(i,i)<=0
        error("Errore: La matrice deve essere sdp");
    end
    A(i+1:n,i) = (A(i+1:n,i)-A(i+1:n,1:i-1)*v)/A(i,i);
end
x=b;
for i=1:n
    x(i+1:n) = x(i+1:n)-(A(i+1:n,i)*x(i));
end
x = x./diag(A);
for i=n:-1:2
    x(1:i-1) = x(1:i-1)-A(i,1:i-1).'*x(i);
end
end
```

Ecco i risulati con alcuni esempi:

```
Command Window
>> mialdlTest
Esempio 1: Matrice sdp con soluzione esatta
Soluzione x:
    1.2537
   -0.4627
    1.0597

Verifica Ax - b:
   1.0e-15 *

    0.8882
         0
         0

Esempio 2: Matrice non sdp
Errore:
Matrice non sdp
Esempio 3: Dimensioni diverse della matrice e del vettore b
Errore:
Dimensione vettore termini noti non corretta
```

Esercizio 10. Scrivere una function Matlab,

function [x,nr] = miaqr(A,b)

che, data in ingresso la matrice A m × n, con m ≥ n = rank(A), ed un vettore b di lunghezza m, calcoli la soluzione del sistema lineare Ax = b nel senso dei minimi quadrati e, inoltre, la norma, nr, del corrispondente vettore residuo. Curare particolarmente la scrittura e l'efficienza della function, e validarla su un congruo numero di esempi significativi, che evidenzino tutti i suoi possibili output.

```
function [x,nr] = miaqr(A,b)
% [x,nr] = miaqr(A,b)
%
% La funzione calcola la fattorizzazione QR del sistema lineare
% sovraddimensionato e restituisce, oltre alla fattorizzazione, la norma
% euclideiana del vettore residuo
%
% Input:
% A = matrice da fattorizzare
% b = vettore dei termini noti
%
% Output:
% x = soluzione del sistema Ax=b
% nr = norma del vettore residuo
%
[m,n] = size(A);
if(n>m)
   error('Errore: il sistema in input non è sovradeterminato.');
end
```

```
k = length(b);
if k~=m
    error('Errore: La dimensione della matrice e del vettore non coincidono.');
end
for i = 1:n
    a = norm(A(i:m,i),2);
    if a==0
        error('Errore: La matrice non ha rango massimo.');
    end
    if A(i,i)>=0
        a = -a;
    end
    v1 = A(i,i)-a;
    A(i,i) = a;
    A(i+1:m,i) = A(i+1:m,i)/v1;
    beta = -v1/a ;
    A(i:m,i+1:n) = A(i:m,i+1:n)-(beta*[1;A(i+1:m,i)])*...
        ([1;A(i+1:m,i)]'*A(i:m,i+1:n));
end
for i=1:n
    v = [1;A(i+1:m,i)];
    beta = 2/(v'*v);
    b(i:k) = b(i:k)-(beta*(v'*b(i:k)))*v;
end
for j=n:-1:1
    b(j) = b(j)/A(j,j);
    b(1:j-1) = b(1:j-1)-A(1:j-1,j)*b(j);
end
x = b(1:n);
nr = norm(b(n+1:m));
end
```

Ecco i risultati con alcuni esempi:

```
>> miaqrTest
Esempio 1: Matrice a rango massimo
Soluzione x:
   -0.0000
    0.5000
    0.0000

Norma del vettore residuo:
   4.4409e-16

Esempio 2: Matrice non a rango massimo
Errore:
Matrice non a rango massimo
Esempio 3: Dimensioni diverse della matrice e del vettore b
```

Esercizio 11. Risolvere i sistemi lineari, di dimensione n

$$A_n x_n = b_n \qquad n = 1,\ldots\ldots 15$$

In cui

$$A_n = \begin{pmatrix} 1 & 1 & \cdots & \cdots & 1 \\ 10 & \ddots & \ddots & \ddots & \vdots \\ 10^2 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 & 1 \\ 10^{n-1} & \cdots & 10^2 & 10 & 1 \end{pmatrix} \in \mathbb{R}^n, \qquad b_n = \begin{pmatrix} n-1+\frac{10^1-1}{9} \\ n-2+\frac{10^2-1}{9} \\ n-3+\frac{10^3-1}{9} \\ \vdots \\ n-n+\frac{10^n-1}{9} \end{pmatrix}$$

la cui soluzione è il vettore $x_n = (1,\ldots,1)^T \in \mathbb{R}^n$, utilizzando la *function mialu*. Tabulare e commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Ecco tabulati i risultati:

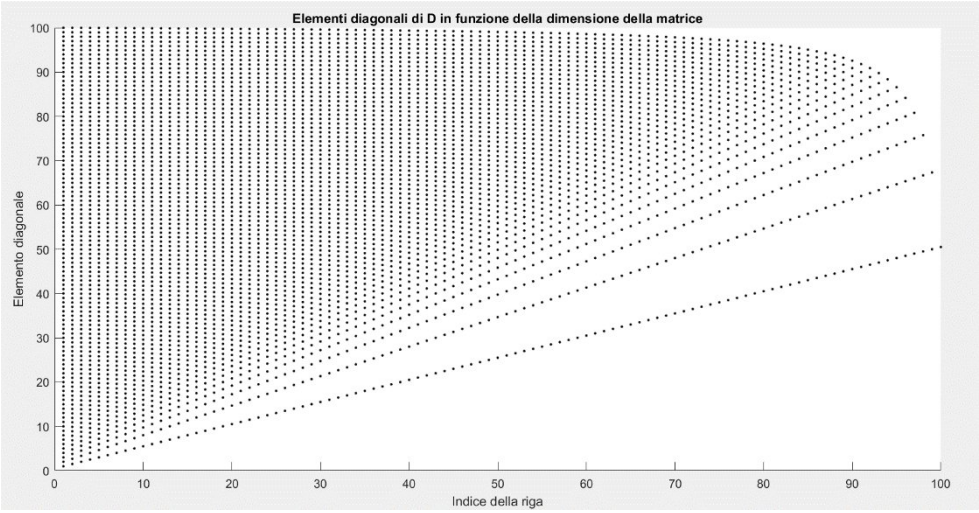
Dimensione n	Condizionamento	Errore Assoluto	Errore Relativo
1	1.0000000000e+00	0.0000000000e+00	0.0000000000e+00
2	1.1356388279e+01	1.6063934761e+02	1.1358917202e+02
3	1.9105814295e+02	1.6338757603e+03	9.4331861001e+02
4	2.1679039517e+03	1.6492347225e+04	8.2461736127e+03
5	2.2819019427e+04	1.6514554698e+05	7.3855333844e+04
6	2.3418238977e+05	1.6517449519e+06	6.7432205289e+05
7	2.3770673699e+06	1.6517806232e+07	6.2431439278e+06
8	2.3995046019e+07	1.6517848624e+08	5.8399413863e+07
9	2.4146532579e+08	1.6517853535e+09	5.5059511783e+08
10	2.4253547305e+09	1.6517854093e+10	5.2234040993e+09
11	2.4331913082e+10	1.6517854156e+11	4.9803204161e+10
12	2.4391001447e+11	1.6517854163e+12	4.7682937737e+11
13	2.4436635652e+12	1.6517854167e+13	4.5812284739e+12
14	2.4472650172e+13	1.6517854164e+14	4.4145822175e+13
15	2.4501220264e+14	1.6517876984e+15	4.2648974984e+14

L'analisi dei risultati suggerisce che, nonostante il condizionamento elevato della matrice A, la precisione numerica degli algoritmi utilizzati sembra mantenere gli errori assoluti a livelli relativamente costanti mentre gli errori relativi diminuiscono dopo un certo punto. Tuttavia, è importante tenere conto del condizionamento elevato della matrice, che può aumentare la sensibilità agli errori numerici, specialmente quando si risolvono sistemi lineari con n grandi.

Esercizio 12. Fattorizzare usando la *function* ***miadlt***, le matrici sdp

$$A_n = \begin{pmatrix} n & -1 & \cdots & \cdots & -1 \\ -1 & \ddots & \ddots & \ddots & \vdots \\ -1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -1 & -1 \\ -1 & \cdots & \cdots & -1 & n \end{pmatrix} \in \mathbb{R}^n, \quad n = 1,\ldots\ldots\ldots 100$$

Graficare, in un unico grafico, gli elementi diagonali del fattore D, rispetto all'indice diagonale.



Esercizio 13. Utilizzare la *function miaqr* per risolvere, nel senso dei minimi quadrati, il sistema

$$Ax = b$$

in cui $A = \begin{pmatrix} 7 & 2 & 1 \\ 8 & 7 & 8 \\ 7 & 0 & 7 \\ 4 & 3 & 3 \\ 7 & 0 & 10 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$

dove viene minimizzata la seguente norma *pesata* del residuo $r = (r_1,\ldots\ldots,r_5)^T$

$$\rho_\omega^2 := \sum_{i=1}^{5} \omega_i r_i^2$$

con $\omega_1 = \omega_2 = 0.5, \quad \omega_3 = 0.75, \quad \omega_4 = \omega_5 = 0.25$

Dettagliare l'intero procedimento, calcolando, in uscita, anche ρω.

```
Command Window
>> Esercizio13
Soluzione x:
    0.1531
   -0.1660
    0.3185

Norma pesata del residuo (ρ ω): 1.0627
```

Esercizio 14. Scrivere una function Matlab,

[x,nit] = newton(fun,x0,tol,maxit)

che implementi efficientemente il metodo di Newton per risolvere sistemi di equazioni nonlineari. Curare particolarmente il criterio di arresto. La seconda variabile, se specificata, ritorna il numero di iterazioni eseguite. Prevedere opportuni valori di default per gli ultimi due parametri di ingresso (rispettivamente, la tolleranza per il criterio di arresto, ed il massimo numero di iterazioni). La function fun deve avere sintassi: [f,jacobian]=fun(x), se il sistema da risolvere `e f(x)=0.

```
function [x, nit] = newton(fun, x0, tol, maxit)
%
%   [x, nit] = newton(fun, x0, tol, maxit)
%
%   Questo metodo risolve sistemi non lineari di equazioni attraverso l'uso
%   del metodo di Newton. Può restituire inoltre il numero di iterazioni
%   eseguite
%
%   Input:
%   fun     [f,jacobian]=fun(x)
%   x0      vettore delle approssimazioni iniziali
%   tol     tolleranza
%   maxit   massimo numero iterazioni
%
%   Output:
%   x       vettore delle soluzioni
%   nit     numero di iterazioni svolte
%
% Verifica degli input e assegnazione dei valori di default
if nargin < 2
    error('Numero di input errato');
elseif nargin == 2
    tol = 1e-6;  % Tolleranza di default
    maxit = 100; % Massimo numero di iterazioni di default
elseif nargin == 3
    maxit = 100; % Massimo numero di iterazioni di default
end
if tol <= 0
    error('Errore: La tolleranza in input non può essere minore o uguale a 0.');
elseif maxit <= 0
    error('Errore: Il massimo numero di iterazioni deve essere maggiore di 0.');
end
x = x0;
for nit = 1:maxit
    x0 = x;
    % Valutazione della funzione e del suo jacobiano
    [f, jac] = feval(fun, x0);
    % Risoluzione del sistema lineare jac * delta = -f
    delta = risLU(jac, -f);
    % Aggiornamento della soluzione
    x = x0 + delta;
    % Criterio di arresto
    if norm(delta) <= tol * (1 + norm(x0))
        disp('Tolleranza desiderata raggiunta.');
        return
    end
end
```

```
% Messaggio se il metodo non è convergente entro il massimo numero di iterazioni
if norm(x - x0) > tol * (1 + norm(x0))
    disp('Il metodo non è convergente.');
end
end
```

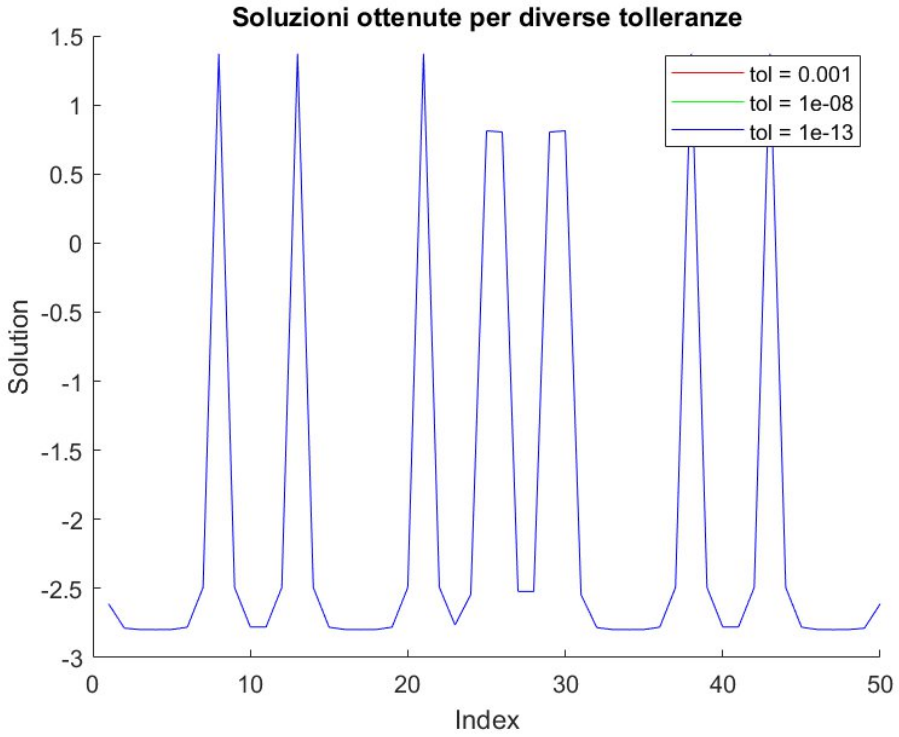
Esercizio 15. Usare la function del precedente esercizio per risolvere, a partire dal vettore iniziale nullo, il sistema nonlineare derivante dalla determinazione del punto stazionario della funzione:

$$f(x) = \frac{1}{2}x^T Q x + e^T\left[\cos(\alpha x) + \beta \exp(-x)\right], \qquad e = (1,\ldots\ldots 1)^T \in \mathbb{R}^{50}$$

$$Q = \begin{pmatrix} 4 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 4 \end{pmatrix} \in \mathbb{R}^{50x50}, \qquad \alpha = 2, \quad \beta = -1.1$$

utilizzando tolleranze tol = 1e-3, 1e-8, 1e-13 (le function cos e exp sono da intendersi in modo vettoriale). Graficare la soluzione e tabulare in modo conveniente i risultati ottenuti.

I grafici si sovrappongono esattamente, cioè è lo stesso per ongi tolleranza usata



	Tolleranza	Numero di Iterazioni
1	1.0000e-03	699
2	1.0000e-08	701
3	1.0000e-13	702

Esercizio 16. Costruire una function, lagrange.m, avente la stessa sintassi della function spline di Matlab, che implementi, in modo vettoriale, la forma di Lagrange del polinomio interpolante una funzione.

```
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
%Controllo che le componenti del vettore X siano distinte
if length(unique(X)) ~= n
 error('Le ascisse non sono distinte');
end
YQ = zeros(size(XQ));
for i=1:n
 YQ = YQ + Y(i) * lin(XQ, X, i);
end
end


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
end
```

Esercizio 17. Costruire una function, newton.m, avente la stessa sintassi della function spline di Matlab, che implementi, in modo vettoriale, la forma di Newton del polinomio interpolante una funzione.

```
function YQ = newton(X, Y, XQ)
%
% YQ = newton(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
%Controllo che le componenti del vettore X siano distinte
if length(unique(X)) ~= length(X)
 error('Le ascisse non sono distinte');
end
df = divdif(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
end


function df = divdif(x, f)
%
% df = divdif(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. Costruire una function, newton.m, avente la stessa sintassi della function spline di Matlab, che implementi, in modo vettoriale, la forma di Newton del polinomio interpolante una funzione.

```
function yy = hermite(xi,fi,f1i,xx)
%   yy = hermite(xi,fi,f1i,xx)
%   function che implementa la forma di hermite del polinomio
%   interpolante una funzione
%
%   input
%   xi      vettore delle coordinate x
%   fi      valori della funzione alle coordinate x
%   f1i     valori delle derivate della funzione nei punti x
%   xx      vettore dei punti in cui calcolare il polinomio
%
%   output
%   yy      polinomio interpolante in forma di Newton
%controllo sui dati in input
n= length(xi);
if n~= length(fi) || n<= 0 || length(f1i)~= n
   error("Le dimensioni dei dati in input non coincidono");
end
if length(unique(xi))~=n
   error("I valori delle ascisse non sono distinte tra di loro");
end
fi = repelem (fi,2);
for i= 1: length(f1i)
   fi(i*2) = f1i(i);
end
%calcolo delle differenze divise
n=length(xi)-1;
for j=(2*n+1): -2: 3
   fi(j)= (fi(j)- fi(j-2))/(xi((j+1)/2) - xi((j-1)/2));
end
for j= 2: 2*n+1
   for i=(2*n+2): -1: j+1
      fi(i)=(fi(i)- fi(i-1)) / ( xi(round(i/2))-xi(round((i-j)/2)));
   end
end
%calcolo del polinomio interpolante
n=length(fi)-1;
yy= fi(n+1)*ones(size(xx));
for i = n : -1:1
   yy= yy .*(xx- xi(round(i/2)))+fi(i) ;
end
return
end
```

Esercizio 19. Si consideri la seguente base di Newton,

$$\omega_i(x) = \prod_{j=0}^{i-1}(x - x_j) \quad i = 0,.....n$$

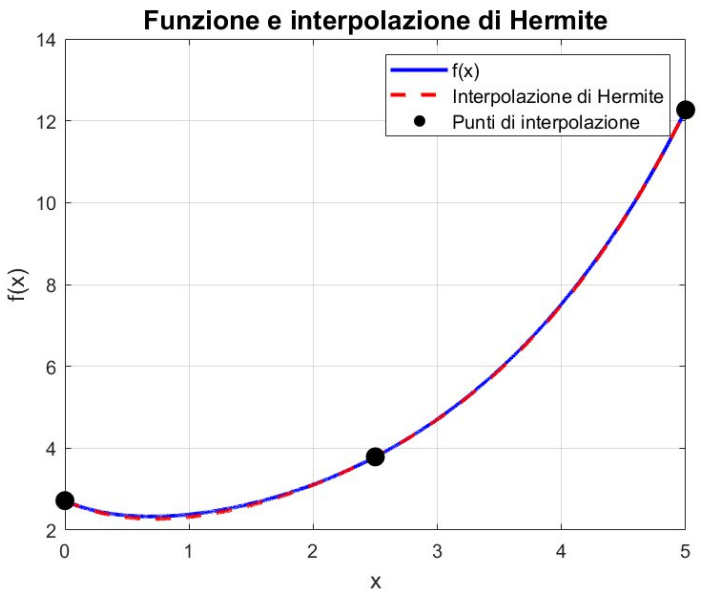
con x0, . . . , xn ascisse date (non necessariamente distinte tra loro), ed un polinomio rappresentato rispetto a tale base,

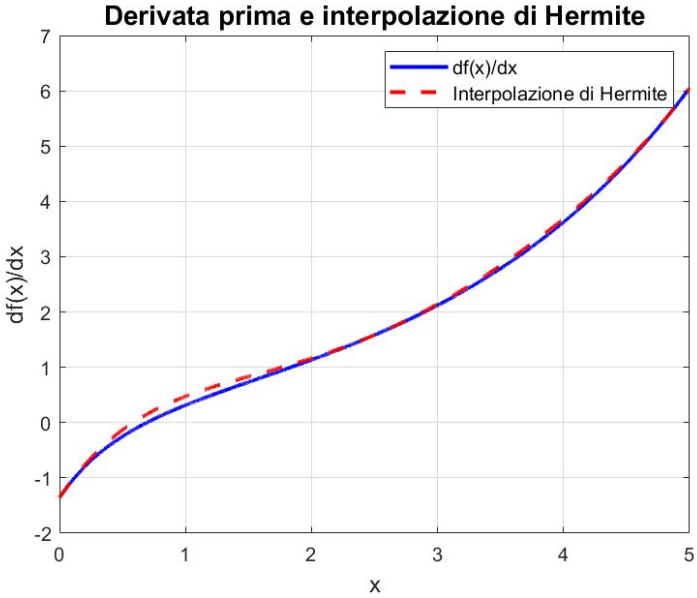
$$p(x) = \sum_{i=0}^{n} a_i \omega_i(x).$$

Derivare una modifica dell' algoritmo di Horner per calcolarne efficientemente la derivata prima.

```
function derivative_value = horner_polynomial_derivative(x_nodes, coefficients, xi)
% Calcola la derivata di un polinomio in forma di Newton in un punto specifico
% Input:
%   x_nodes - Vettore di ascisse [x0, x1, ..., xn]
%   coefficients - Vettore dei coefficienti [a0, a1, ..., an]
%   xi - Punto in cui valutare la derivata
% Output:
%   derivative_value - Valore della derivata del polinomio in x_point
    n = length(coefficients) - 1;

    % Preallocazione dei vettori per P e P'
    P = coefficients(n+1);
    P_prime = 0;

    for k = n:-1:1
        P_prime = P_prime * (xi - x_nodes(k)) + P;
        P = P * (xi - x_nodes(k)) + coefficients(k);
    end

    derivative_value = P_prime;
end
```

Esercizio 20. Utilizzando le function degli esercizi 18 e 19, calcolare il polinomio interpolante di Hermite la funzione f(x) = exp(x/2 + exp(−x)) sulle ascisse equidistanti {0, 2.5, 5}. Graficare il grafico della funzione interpolanda e del polinomio interpolante nell'intervallo [0, 5], e quello della derivata prima della funzione interpolanda, e della derivata prima del polinomio interpolante, verificando graficamente le condizioni di interpolazione per entrambi.





Esercizio 21. Costruire una function Matlab che, specificato in ingresso il grado n del polinomio interpolante, e gli estremi dell'intervallo [a, b], calcoli le corrispondenti ascisse di Chebyshev

```
function x = chebyshev (n ,a ,b )
%
% Genera n+1 coordinate di Chebyschev nell ' intervallo [a,b]
%
% Input :
% n: numero di coordinate da generare (n +1)
% a: estremo inferiore dell ' intervallo
% b: estremo superiore dell ' intervallpo
% Output :
% x: vettore contenente le coordinate
if n <= 0
error (" il grado del polinomio deve essere maggiore di zero ") ;
end
if a >= b
error ("l ' estremo inferiore dell ' intervallo non pu `o essere " +
...
" minore o coincidente con quello maggiore ") ;
end
x = cos ( (2*( n : -1:0) +1) *pi ./ (2*( n +1) ) ) ;
x = x * (b - a) /2 + (a+b ) /2;
end
```

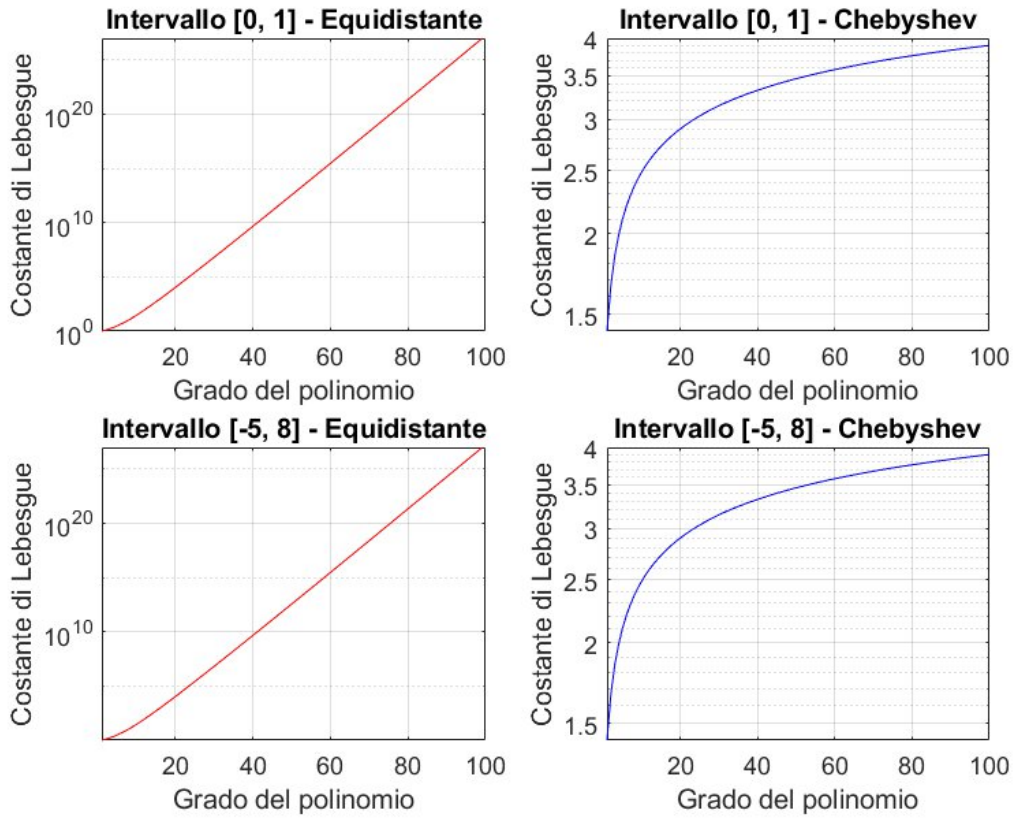
Esercizio 22. Costruire una function Matlab, con sintassi

ll = lebesgue(a, b, nn, type),

che approssimi la costante di Lebesgue per l'interpolazione polinomiale sull'intervallo [a,b], per i polinomi di grado specificato nel vettore nn, utilizzando ascisse equidistanti, se type=0, o di Chebyshev, se type=1 (utilizzare 10001 punti equispaziati nell'intervallo [a,b] per ottenere ciascu na componente di ll). Graficare opportunamente i risultati ottenuti, per nn=1:100, utilizzando [a,b]=[0,1] e [a,b]=[-5,8]. Commentare i risultati ottenuti.

```
function ll = lebesgue (a , b , nn , type )
% ll = lebesgue (a, b, nn , type )
%
% Approssima la costante di Lebesgue per l' interpolazione
% polinomiale sull intervallo [a,b], per i polinomi di
% grado specificato nel vettore nn utilizzando le ascisse
% equidistanti se type `e uguale a 0 e quelle di chebyshev
% se type `e uguale a uno
%
% Input :
% a,b: intervalli sui quali calcolare le ascisse
% nn: grado dei polinomi
% type : se 0 usa le ascisse equidistante
% se 1 usa le ascisse di Chebyshev
%
% Output :
% ll: stima della costante di Lebesgue
if a >= b
    error ("l ' estremo inferiore dell ' intervallo non pu `o essere minore o coincidente con quello
maggiore ") ;
end
n = length ( nn ) ;
ll = ones ( length (n ));
```

```
xq = linspace (a , b , 10001) ;
for i = 1: n
   if type == 0
      x = linspace (a , b , nn (i) +1) ;
   elseif type == 1
      x = chebyshev ( nn (i) , a , b );
   else
      error (" il valore di type pu `o essere soltanto 0 o 1") ;
   end
   L = zeros ( size ( xq )) ;
   m = length (x);
   for k = 1: m
      Lkn = ones ( size ( xq ));
      for j = 1: m
         if k ~= j
            Lkn = Lkn .* (( xq - x(j)) /( x(k ) - x( j)));
         end
      end
      L = L + abs ( Lkn ) ;
   end
   ll (i) = max ( abs (L ));
end
end
```

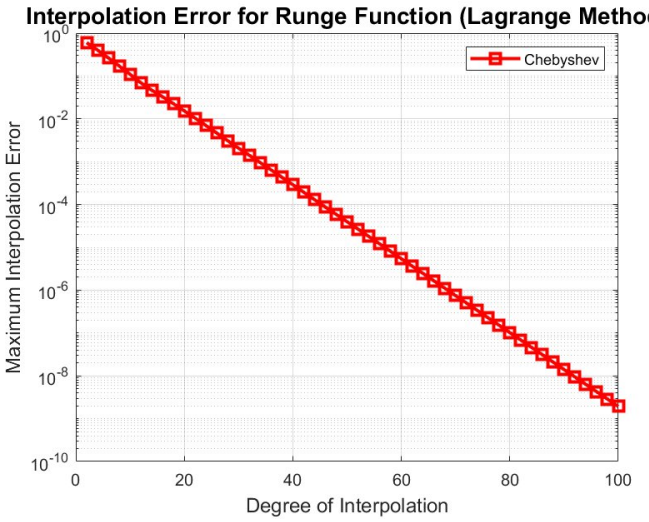


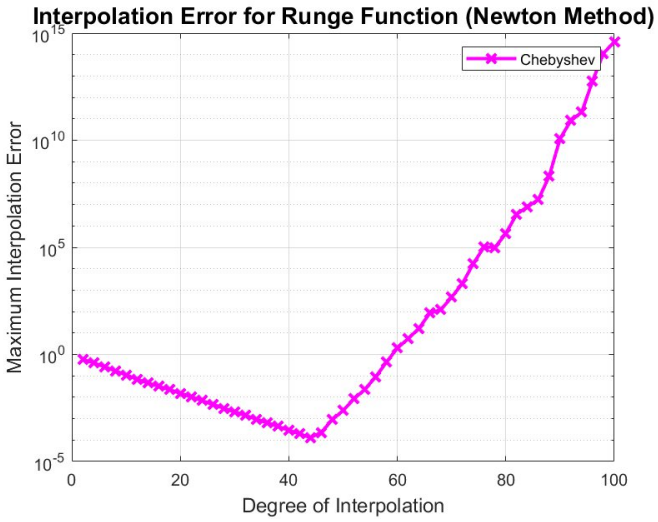
Questi dati sono coerenti con i valori attesi della costante di Lebesgue in quanto è noto che indicando con Λ_n la costante di Lebesgue, questa cresce esponenzialmente se si utilizzano ascisse equidistanti. Inoltre è anche noto che: $\Lambda_n \geq O(log_2(n))$ e che le ascisse di Chebyshev sono ottimali e t.c: $\Lambda_n \approx \frac{2}{\pi} log_2(n)$. Come ci aspettavamo, la costante di Lebesgue è indipendente dall'intervallo [a,b] scelto e con la scelta delle ascisse di Chebyshev cresce in maniera quasi ottimale, cioè ≈ log n

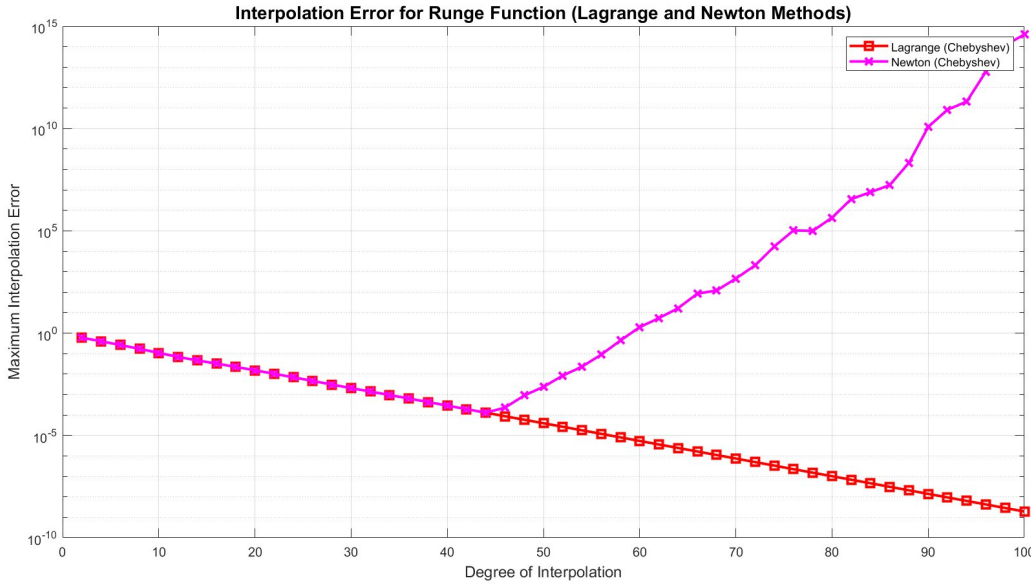
Esercizio 23. Utilizzando le function degli esercizi 16 e 17, graficare (in semilogy) l'andamento errore di interpolazione (utilizzare 10001 punti equispaziati nell'intervallo per ottenerne la stima) per la funzione di Runge,

$$f(x) = \frac{1}{1+x^2} \quad x \in [-5,5]$$

utilizzando le ascisse di Chebyshev, per i polinomi interpolanti di grado n=2:2:100. Commentare i risultati ottenuti.







L'interpolazione di Lagrange *sembra essere stabile per i polinomi di grado crescente. Questo si riflette in un errore di interpolazione che diminuisce progressivamente man mano che il grado del polinomio aumenta.L e ascisse di Chebyshev riducono significativamente il fenomeno di Runge, migliorando la qualità dell'interpolazione. Infatti, le ascisse di Chebyshev sono note per minimizzare il massimo errore di interpolazione rispetto ai punti equispaziati, specialmente per funzioni con comportamento oscillatorio come la funzione di Runge.*

Il metodo di Newton *sembra soffrire di problemi di stabilità numerica con l'aumento del grado del polinomio. Anche se le ascisse di Chebyshev migliorano l'interpolazione, il metodo di Newton non sembra trarne pieno vantaggio, forse a causa della struttura del metodo stesso che può essere più*

sensibile agli errori numerici accumulati.
L'interpolazione di Lagrange, sebbene teoricamente equivalente a quella di Newton, si dimostra più efficiente e accurata per gradi di polinomio elevati quando si utilizzano le ascisse di Chebyshev. Questo si riflette in un errore di interpolazione che diminuisce costantemente.

Quindi:

- ***Metodo di Lagrange:*** *Adatto per polinomi di alto grado con ascisse di Chebyshev per ottenere un errore di interpolazione basso e stabile.*
- ***Metodo di Newton:*** *Sebbene teoricamente valido, può essere meno stabile numericamente per polinomi di alto grado, portando a errori di interpolazione elevati in presenza di fenomeni come l'accumulo di errori di arrotondamento.*

I risultati sottolineano l'importanza di considerare la stabilità numerica nei metodi di interpolazione, specialmente quando si utilizzano polinomi di alto grado e ascisse specifiche come quelle di Chebyshev

Esercizio 24. Costruire una function, spline0.m, avente la stessa sintassi della function spline di Matlab, che calcoli la spline cubica interpolante naturale i punti (xi,fi).

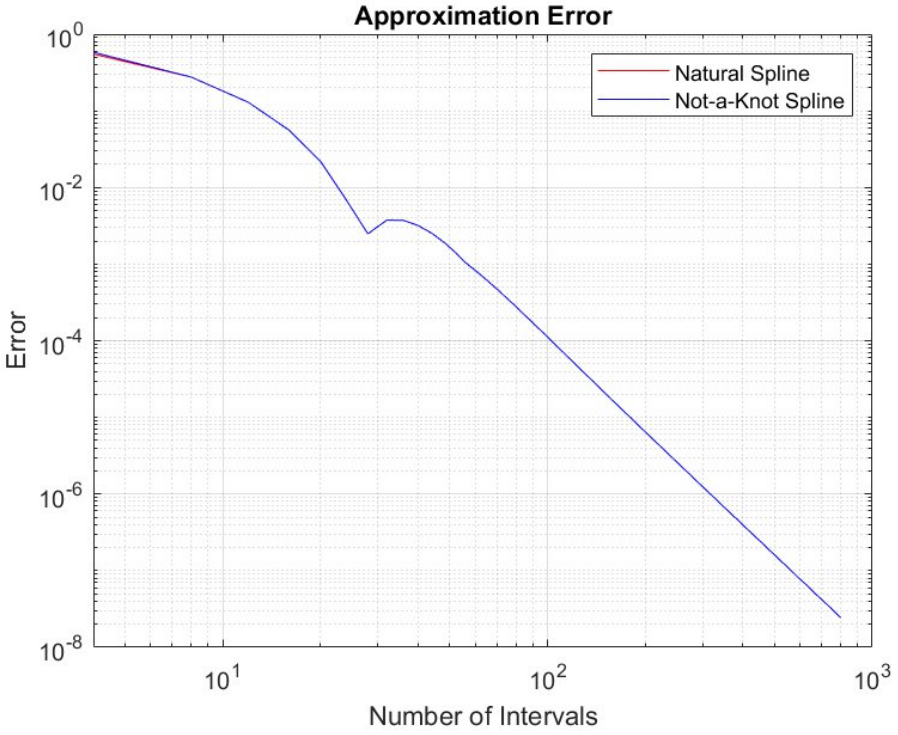
```
function YQ = spline0(X, Y, XQ)
%
% YQ = spline0(X, Y, XQ)
%
% La function calcola la spline cubica naturale interpolante e
% restituisce il valore assunto dalla spline sulle ascisse XQ
%
% Input:
% X: vettore delle ascisse di interpolazione
% Y: vettore dei valori della funzione assunti sulle ascisse
% interpolanti
% XQ: vettore delle ascisse dove si calcola il valore della spline
%
% Output:
% YQ: vettore delle ordinate calcolate sulle ascisse
%
n = length(X);
% Controlli di consistenza
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)); % phi
c = h(2:n-1)./(h(1:n-2) + h(2:n-1)); % csi
a(1:n-1) = 2;
df = ddspline(X, Y, 3);
m = tridia(a, b, c, 6*df); % risoluzione del sistema tridiagonale
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
end



function df = ddspline(X, Y, it)
%
% df = ddspline(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
% 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
end

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
end
```

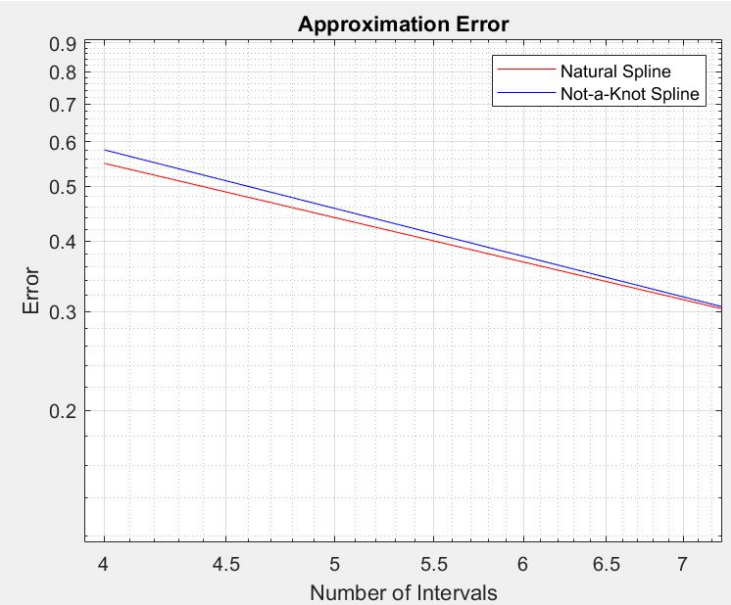
Esercizio 25. Graficare, utilizzando il formato loglog, l'errore di approssimazione utilizzando le spline interpolanti naturale e not-a-knot per approssimare la funzione di Runge sull'intervallo [−10, 10], utilizzando una partizione uniforme

$$\Delta = \left\{ x_i = -10 + i * \frac{20}{n}, i = 0,...n \right\} \; n = 4{:}4{:}800$$

rispetto alla distanza h = 20/n tra le ascisse. Utilizzare 10001 punti equispaziati nell'intervallo [−10, 10] per ottenere la stima dell'errore. Che tipo di decrescita si osserva?



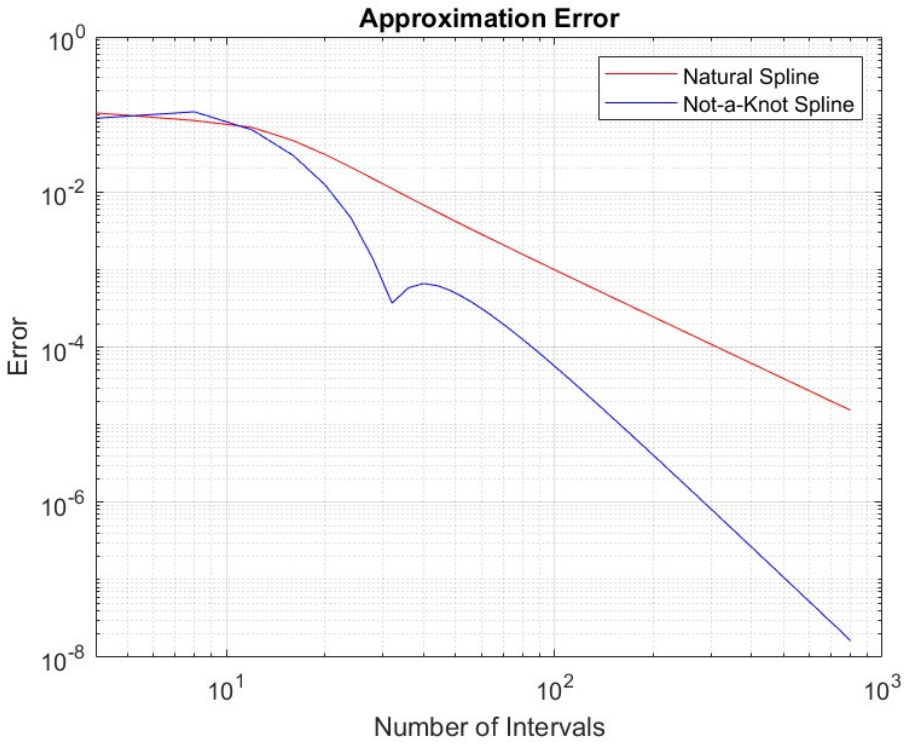
Nota: *per valori piccoli di n (e quindi valori grandi di h), le due spline hanno un errore iniziale leggermente diverso, con la spline naturale che mostra un errore leggermente inferiore, i due grafici si incontrano e proseguono in modo quasi identico, questo suggerisce che, per n più grandi (e quindi h più piccoli), l'errore di approssimazione delle due spline tende a diventare molto simile. La decrescita dell'errore di approssimazione per entrambe le spline segue una legge di potenza rispetto alla distanza h tra le ascisse, e che per valori sufficientemente piccoli di h, l'errore di approssimazione delle due tecniche diventa praticamente indistinguibile.*



Esercizio 26. Graficare, utilizzando il formato loglog, l'errore di approssimazione utilizzando le spline interpolanti naturale e not-a-knot per approssimare la funzione di Runge sull'intervallo [0, 10], utilizzando una partizione uniforme

$$\Delta = \left\{ x_i = i * \frac{20}{n}, i = 0,\ldots n \right\},\ n = 4{:}4{:}800$$

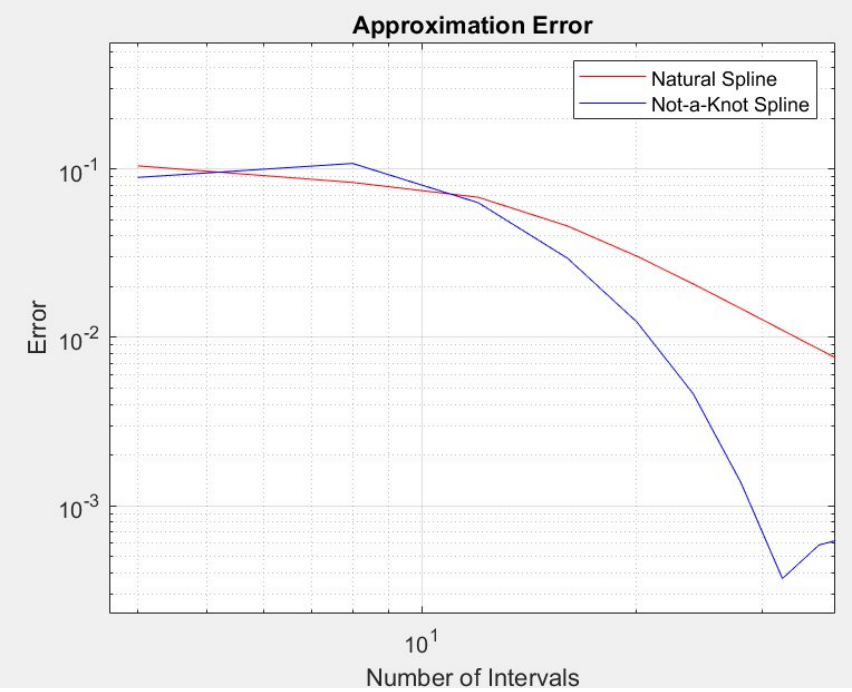
rispetto alla distanza h = 10/n tra le ascisse. Utilizzare 10001 punti equispaziati nell'intervallo [−10, 10] per ottenere la stima dell'errore. Che tipo di decrescita si osserva?



Spline Naturale: *La decrescita dell'errore per la spline naturale è regolare indicando che l'errore segue una legge di potenza . La pendenza della linea suggerisce che l'errore diminuisce costantemente man mano che n aumenta.*

Spline Not-A-Knot: *la decrescita dell'errore per la spline not-a-knot è più rapida inizialmente rispetto alla spline naturale, indicando una maggiore sensibilità a n. Tuttavia, la presenza di irregolarità nel grafico suggerisce che la decrescita dell'errore non è perfettamente uniforme.*

Quindi *questi risultati indicano che entrambi i metodi diventano più precisi con un aumento del numero di intervalli, ma il metodo naturale offre una decrescita più prevedibile rispetto al metodo not-a-knot.*



Esercizio 27. Calcolare i coefficienti del polinomio di approssimazione ai minimi quadrati di grado 3 per i seguenti dati:

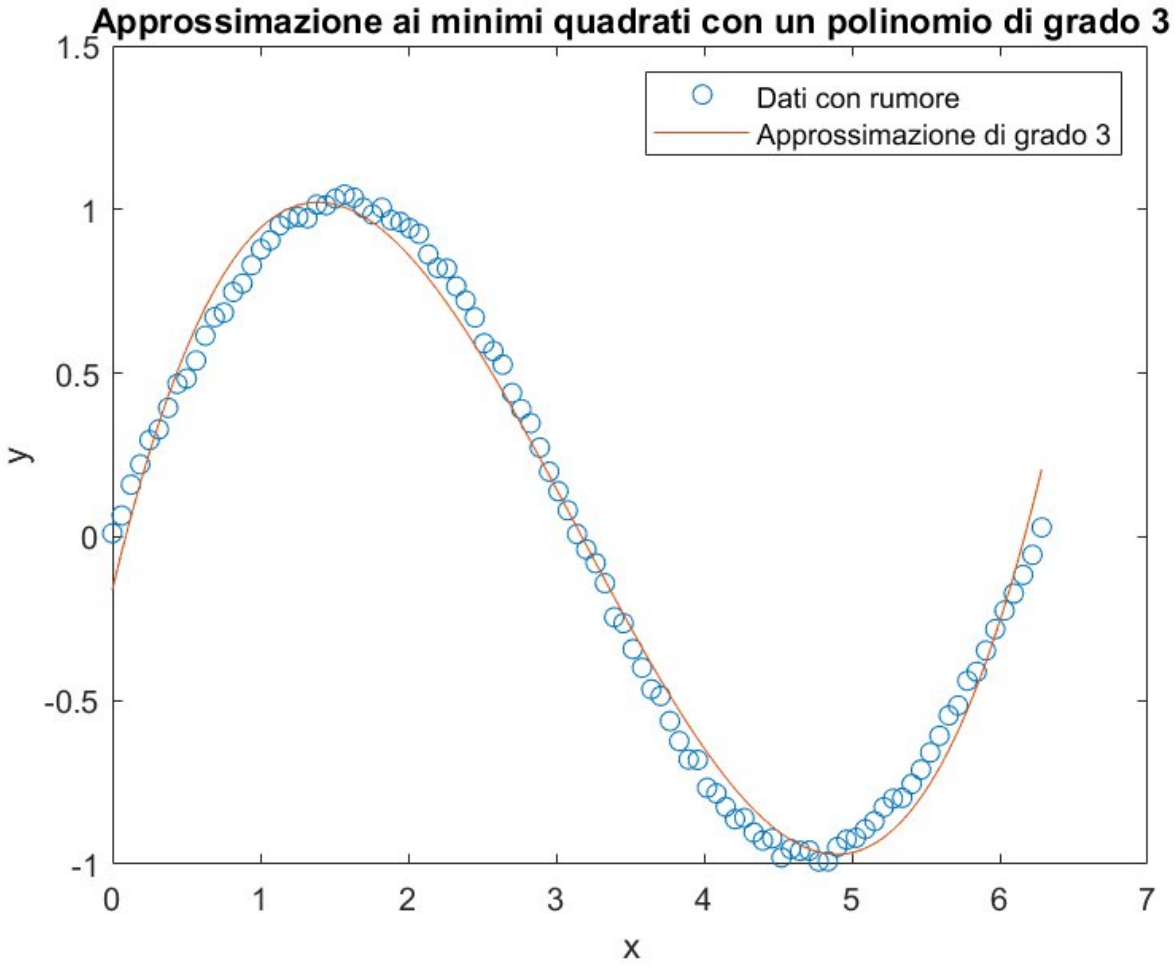
```
>> rng(0)
>> xi=linspace(0,2*pi,101);
>> yi=sin(xi)+rand(size(xi))*.05;
```

Graficare convenientemente i risultati ottenuti.

Ecco i coefficenti calcolati:

```
Coefficienti del polinomio di grado 3:
    0.0922   -0.8698    1.8827   -0.1617
```

Ed ecco il grafico richiesto:



Esercizio 28. Costruire una function Matlab che, dato in input n, restituisca i pesi della qua dratura della formula di Newton-Cotes di grado n. Tabulare, quindi, i pesi delle formule di grado 1, 2,. . . , 7 e 9 (come numeri razionali).

```
function w = newtonCotesWeights (n )
% w = newtoncotes_weights (n)
%
% Calcola i pesi della formula di quadratura di Newton - Cotes di
grado n
%
% Input :
% n: grado della formula di Newton - Cotes
% Output :
% w: vettore dei pesi dei coefficienti
if ~ isnumeric ( n) || n <= 0 || mod (n , 1) ~= 0
error (" Il grado n deve essere un numero intero positivo ") ;
end
x = 0: n;
w = zeros (1 , n + 1) ;
for i = 1: n +1
L = 1;
for j = 1: n +1
if j ~= i
L = conv (L , [1 , -x(j) ]) / ( x(i) - x(j) );
end
end
w(i ) = polyval ( polyint (L ) , n);
end
end
```

Si riportano inoltre i pesi delle formule di grado 1, 2,. . . , 7 e 9.

1	1/2	1/2	-	-	-	-	-	-	-	-
2	1/3	4/23	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	125/96	95/288	-	-	-	-
6	41/140	54/35	27/140	68/35	27/140	54/35	41/140	-	-	-
7	108/355	810/559	343/640	649/536	649/536	343/640	810/559	108/355	-	-
9	130/453	419/265	23/212	307/158	213/367	213/367	307/158	23/212	419/265	130/453

Esercizio 29. Scrivere una function Matlab,

[If,err] = composita(fun, a, b, k, n)

che implementi la formula composita di Newton-Cotes di grado k su n+1 ascisse equidistanti, con n multiplo pari di k, in cui:
-fun è la funzione integranda (che accetta input vettoriali);
-[a,b] è l'intervallo di integrazione;
-k, n come su descritti;
-If è l'approssimazione dell'integrale ottenuta;
-If è l'approssimazione dell'integrale ottenuta;
Le valutazioni funzionali devono essere fatte tutte insieme in modo vettoriale, senza ridondanze.

```
function [If , err ] = composita ( fun , a , b , k , n )
% [If , err] = composita ( fun , a, b, k, n )
%
% Calcola l' integrale della funzione fun mediante l' utilizzo
%della
% formula composita di Newton - Cotes di grado k su n intervalli
% equidistanti
%
% Input :
% fun : la funzione integranda
% a,b: intervalli di integrazione
% k: grado della formula di Newton - Cotes
% n: numero di intervalli
%
% Output :
% If: stima dell ' integrale
% err : stima dell ' errore di quadratura
if b < a
    error ("L ' intervallo specificato non `e corretto l ' estremo superiore non puo' essere minore di
quello inferiore ") ;
end
if mod (n ,2) ~=0 || mod (n ,k) ~=0
    error (" n deve essere un multiplo pari di k ") ;
end
if ( mod (k ,2) == 0)
    m =2;
else
    m =1;
end
w = newtonCotesWeights ( k);
If = 0;
Ie = 0;
h = (b -a) /n;
he = (b - a) /( n /2) ;
for i =0: n -1
    x = linspace (a +i*h , a +( i +1) *h , k +1) ;
    y = feval ( fun , x);
    If = If + (h/k) *sum (y .* w );
end
for i = 0: n /2 -1
    x = linspace (a +i*he , a +( i +1) *he , k +1) ;
    y = feval ( fun , x);
    Ie = Ie + ( he /k )* sum (y .* w);
end
err = abs (( If - Ie ) /(2^( k+m) -1) );
end
```

Esercizio 30. Calcolare l'espressione del seguente integrale:

$\int_0^1 e^{3x} dx$

Utilizzare la *function* del precedente esercizio per ottenere un'approssimazione dell' integrale per i valori k = 1, 2, 3, 6, e n = 12. Tabulare i risultati ottenuti, confrontando l'errore stimato con quello vero

k	Stima dell'integrale	Errore stimato	Errore vero
k	Stima dell'integrale	Errore stimato	Errore vero
1	6.3949	0.0141	0.0331
2	6.3619	8.5618e-06	8.6128e-06
3	6.3618	1.8429e-06	3.8287e-06
6	6.3618	6.1284e-14	1.2985e-12