A.A 2022/2023

Elaborato Calcolo Numerico

Autori:

Damiano Salvi 7032930

Lorenzo Fiordiponti 7025845

Esercizio 1. Verificare che:

$$-\frac{1}{4}f(x-h)-\frac{5}{6}f(x)+\frac{3}{2}f(x+h)-\frac{1}{2}f(x+2h)+\frac{1}{12}f(x+3h)=hf'(x)+O(h^5)$$

Soluzione:

Per verificare l'equazione dobbiamo scrivere gli sviluppi di Taylor

$$f(x-h)=f(x)-hf'(x)+\frac{1}{2}h^2f''(x)-\frac{1}{6}h^3f'''(x)+\frac{1}{24}h^4f''''(x)+O(h^5)$$

$$f(x+h)=f(x)+hf'(x)+\frac{1}{2}h^2f''(x)+\frac{1}{6}h^3f'''(x)+\frac{1}{24}h^4f''''(x)+O(h^5)$$

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

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

Sostituendo al primo membro dell'equazione gli sviluppi qui sopra si ottiene:

$$-\frac{1}{4}f(x)+\frac{h}{4}f'(x)-\frac{1}{8}h^2f''(x)+\frac{1}{24}h^3f'''(x)-\frac{1}{96}h^4f''''(x)+O(h^5)$$

$$\frac{3}{2}f(x)+\frac{3}{2}hf'(x)+\frac{3}{4}h^2f''(x)+\frac{1}{4}h^3f'''(x)+\frac{1}{16}h^4f''''(x)+O(h^5)$$

$$-\frac{1}{2}f(x)-hf'(x)-h^2f''(x)-\frac{2}{3}h^3f'''(x)-\frac{1}{3}h^4f''''(x)+O(h^5)$$

$$\frac{1}{12}f(x)+\frac{1}{4}hf'(x)+\frac{3}{8}h^2f''(x)+\frac{3}{8}h^3f'''(x)+\frac{9}{32}h^4f''''(x)+O(h^5)$$

Considerando anche

$$-\frac{5}{6}f(x)$$

Si ottiene, sommando membro a membro:

$$-\tfrac{5}{6}f(x)-\tfrac{1}{4}f(x)+\tfrac{3}{2}f(x)-\tfrac{1}{2}f(x)+\tfrac{1}{12}f(x)=0$$

$$\frac{1}{4}hf'(x)+\frac{3}{2}hf'(x)-hf'(x)+\frac{1}{4}hf'(x)=hf'(x)$$

$$-\frac{1}{8}h^2f''(x)+\frac{3}{4}h^2f''(x)-h^2f''(x)+\frac{3}{8}h^2f''(x)=\ 0$$

$$\frac{1}{24}h^3f'''(x) + \frac{1}{4}h^3f'''(x) - \frac{2}{3}h^3f'''(x) + \frac{3}{8}h^3f'''(x) = 0$$

$$-\frac{1}{96}h^4f''''(x) + \frac{1}{16}h^4f''''(x) - \frac{1}{3}h^4f''''(x) + \frac{9}{32}h^4f''''(x) = 0$$

cioè:

$$hf'(x) + O(h^5)$$

Che è equivalente al termine sinistro dell'esercizio.

Esecizio 2. Matlab utilizza la doppia precisione IEEE. Stabilire, pertanto, il nesso tra la variabile eps e la precisione di macchina di questa aritmetica.

Soluzione:

Sappiamo che la precisione di macchina u, secondo lo standard IEEE, si calcola come

$$u = \frac{1}{2}b^{1-m}$$

Dato che lo standard utilizza la rappresentazione per arrotondamento. Data la base binaria (*b=2*) e m = 53 cifre per la mantissa, si ha

$$\frac{1}{2}b^{1-m}$$

Sostituendo

$$\frac{1}{2}2^{1-53} = 2^{-1}2^{1-53}$$

$$2^{-53} \approx 1{,}110223e-16$$

La funzione **eps** di MatLab, che contiene la precisione di macchina, viene invece calcolata per troncamento, quindi con la seguente formula

$$u = b^{1-m}$$

Sostituendo

$$2^{1-53} = 2^{-52}$$
$$2^{-52} \approx 2.220446e-16$$

La precisione di macchina è definita come il massimo errore relativo dovuto alla rappresentazione in aritmetica finita di un numero reale. Si può quindi vedere come la precisione di macchina calcolata per arrotondamento sia più precisa rispetto al calcolo per troncamento.

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

Soluzione:

La cancellazione numerica è la perdita di cifre significative nella somma, in aritmetica finita, di numeri quasi opposti. Questo fenomeno è dovuto al mal condizionamento della somma algebrica quando i due numeri da sommare sono di segno discorde. Infatti, sappiamo che per la somma il numero K di condizionamento è dato da

$$K = \frac{|x| + |y|}{|x + y|}$$

È evidente che:

- Se xy > 0 ne segue che |x|+|y| = |x+y| e quindi K = 1
- Se invece $x \approx -y$ allora |x+y|$\ll$ |x|+|y| e quindi $K \gg 1$; ne segue quindi che la somma è mal condizionata.

Un esempio che esemplifica il malcondizionamento della somma potrebbe essere:

```
format long e
(1+(1e-14-1))*1e14
```

Eseguendo lo script si ottiene infatti:

$$ans = 9.992007221626409e-01 \approx 0.9992007221626$$

Che però non è corretto, dato che il risultato atteso è 1. Il motivo di questa discordanza è motivato dal mal condizionamento, spiegato sopra.

Esercizio 4. Scrivere una function Matlab, radice(x) che, avendo in ingresso un numero x non negativo, calcoli 6√x utilizzando solo operazioni algebriche elementari, con la massima precisione possibile. Confrontare con il risultato fornito da x^(1/6) per 20 valori di x, equispaziati logaritmicamente nell'intervallo [1e-10,1e10], tabulando i risultati in modo che si possa verificare che si è ottenuta la massima precisione possibile.

Soluzione:

Per calcolare la radice di n-esima di un numero non negativo A possiamo utilizzare il metodo iterativo $x_{k+1} = \frac{1}{n}((n-1)x_k + \frac{A}{x_k^{n-1}})$ che parte da un'approssimazione della radice positiva e scelta a priori.

Nel nostro caso, per la radice sesta si utilizza $x_{k+1} = \frac{1}{6}\left(5x_k + \frac{A}{x_k^5}\right)$

È possibile vedere che il limite di questa sequenza converge alla radice $\lim_{k\to\infty} x_k = \sqrt[6]{A}$

```
function radice = radice(r)
%
% radice = radice(x)
%
% Metodo che calcola la radice quadrata del numero reale in input utilizzando un
% metodo iterativo con sole operazioni elementari e che parte da una
% approssimazione iniziale. Questo metodo calcola la radice con la massima
% precisione possibile
%
% Input: r = radicando
% Output: radice = radice quadrata di r
%
if(r<0), error("La radice sesta di un numero negativo non è prevista dalla
funzione"); end
if(r == 0)
    radice = 0;
    return
end
radice = r;
errore = 1;
while (errore >= eps*(1+abs(r)))
    radiceProv = (radice * 5 + r/radice^5)/6;
    errore = abs((radice - radiceProv));
    radice = radiceProv;
end
return
end
```

x	radice(x)	x^(1/6)
1.000000e-10	2.154435e-02	2.154435e-02
1.128838e-09	3.226799e-02	3.226799e-02
1.274275e-08	4.832930e-02	4.832930e-02
1.438450e-07	7.238509e-02	7.238509e-02
1.623777e-06	1.084146e-01	1.084146e-01
1.832981e-05	1.623777e-01	1.623777e-01
2.069138e-04	2.432008e-01	2.432008e-01
2.335721e-03	3.642533e-01	3.642533e-01
2.636651e-02	5.455595e-01	5.455595e-01
2.976351e-01	8.171103e-01	8.171103e-01
3.359818e+00	1.223825e+00	1.223825e+00
3.792690e+01	1.832981e+00	1.832981e+00
4.281332e+02	2.745342e+00	2.745342e+00
4.832930e+03	4.111829e+00	4.111829e+00
5.455595e+04	6.158482e+00	6.158482e+00
6.158482e+05	9.223851e+00	9.223851e+00
6.951928e+06	1.381500e+01	1.381500e+01
7.847600e+07	2.069138e+01	2.069138e+01
8.858668e+08	3.099046e+01	3.099046e+01
1.000000e+10	4.641589e+01	4.641589e+01

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). Per tutti i metodi, utilizzare come criterio di arresto

$$|x_{n+1} - x_n| \leq tol(1 + |x_n|),$$

essendo *tol* una opportuna tolleranza specificata in ingresso. Curare particolarmente la robustezza del codice.

Soluzione:

Metodo di Newton

```
function [r,it] = newton(r0,f,f1,tol,itmax)
%
% [r,i] = newton(x0,f,f1,tol,itmax)
%
% Partendo dalla approssimazione iniziale x0, questa funzione restituisce
% una radice della funzione utilizzando il metodo di Newton
%
% Input:
%       r0 = approssimazione iniziale della radice
%       f = funzione su cui si applica il metodo di Newton
%       f1 = derivata prima di f
%       tol = tolleranza del metodo
%       itmax = numero massimo di iterazioni
% Output:
%       r = approssimazione della radice
%       it = numero di iterazioni svolte dal metodo
%
if tol<0
 error('Errore tolleranza: la tolleranza deve essere maggiore di 0.');
end
if itmax<=0
 error('Errore numero iterazioni: le iterazioni devono essere maggiore di 0.');
end
r = r0;
for it=1:itmax
 r0 = r;
 fx = feval(f,r0);
 f1x = feval(f1,r0);
 if (f1x==0)
 error('Errore derivata prima uguale a 0: scegliere una diversa approssimazione
iniziale' );
 end
 r = r0-fx/f1x;
 if(abs(r-r0)<=tol*(1+abs(r0)))
 disp('Il metodo converge.');
 break
 end
end
if (abs(r-r0)>tol*(1+abs(r0)))
 disp('Il metodo non converge.');
end
end
```

Metodo delle secanti

```
function [x,i] = secanti(x0,x,f,tolx,itmax)
%
% [x,i] = secanti(x0,x,f,tolx,itmax)
%
% A partire da due approssimazioni iniziali, il metodo calcola la radice
% della funzione con il metodo delle secanti
%
%
% Input:
%       x0 = approssimazione iniziale
%       x = approssimazione iniziale
%       f = funzione di cui si vuole trovare la radice
%       tolx = valore di tolleranza
%       itmax = numero massimo di iterazioni
%
% Output:
%       x = radice della funzione
%       i = numero di iterazioni
%
if itmax<=0
 error('Errore tolleranza: la tolleranza deve essere maggiore di 0.');
end
if tolx<0
 error('Errore numero iterazioni: le iterazioni devono essere maggiore di 0.');
end
fx = feval(f,x0);
for i=1:itmax
 fx0 = fx;
 fx = feval(f,x);
 if (fx-fx0==0)
 error('Errore: Denominatore uguale a zero.');
 end
 x1 = (fx*x0-fx0*x)/(fx-fx0);
 x0 = x;
 x = x1;
 if(abs(x-x0)<=tolx*(1+abs(x0)))
 disp('Il metodo converge');
 break
 end
end
if not(abs(x-x0)<=tolx*(1+abs(x0)))
 disp('Il metodo non converge.');
end
end
```

Esercizio 6. Utilizzare le *function* del precedente esercizio per determinare una approssimazione della radice della funzione

$$f(x) = x - \cos(x)$$

Per tol = $10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$ partendo da $x_0 = 0$ (e $x_1 = 0.1$ per il metodo delle secanti).

Tabulare i risultati, in modo da confrontare il costo computazionale di ciascun metodo.

Soluzione:

Nella seguente tabella i risultati (*i* indica il numero di iterazioni)

tol	Newton		Secanti	
	Risultato	i	Risultato	i
10^{-3}	0.739085133385284	3	0.739085133345758	3
10^{-6}	0.739085133215161	4	0.739085133215161	4
10^{-9}	0.739085133215161	4	0.739085133215161	4
10^{-12}	0.739085133215161	5	0.739085133215161	5

Costo computazionale:

- Newton: Sono necessarie 2 valutazioni funzionali per ogni iterazione -> *2i*;
- Secanti: Per il metodo delle secanti è necessaria 1 valutazione funzionale per ogni iterazione, più una valutazione iniziale -> *i + 1*

tol	Newton		Secanti	
	valutazione	i	valutazione	i
10^{-3}	6	3	4	3
10^{-6}	8	4	5	4
10^{-9}	8	4	5	4
10^{-12}	10	5	6	5

Inoltre con Newton è necessario calcolare la derivata prima.

Esercizio 7. Utilizzare le *function* del precedente esercizio per determinare una approssimazione della radice della funzione

$$f(x) = [x - \cos(x)]^5$$

Per tol = $10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$ partendo da $x_0 = 0$ (e $x_1 = 0.1$ per il metodo delle secanti).

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

Soluzione:

nella tabella sottostante i risultati delle varie funzioni

tol	Newton		Secanti		Newton Modificato	
	Soluzione raggiunta	i	Soluzione raggiunta	i	Soluzione raggiunta	i
10^{-3}	0.553653192405491	34	0.554479696664797	34	0.733079688985905	2
10^{-6}	0.692368176589544	6980	0.692367424057885	6979	0.739085133538309	4
10^{-9}	0.727356991168904	1701866	0.727356990400689	1701865	0.739085133538309	4
10^{-12}	0.73613961797603	424430882	0.736139617975257	424430881	0.739085133214857	65

Si può notare immediatamente come i metodi di Newton e Secanti siano estremamente più lenti del metodo di newton modificato, questo perché la funzione data ha molteplicità maggiore di uno, questo rende la convergenza del metodo di Newton lineare; se però utilizziamo l'accelerazione di Aitken per il metodo di Newton possiamo ripristinare la convergenza quadratica.
Si può notare anche come la radice raggiunta dal metodo di newton modificato sia più precisa già dalle prime iterazioni.
Siccome conosciamo la molteplicità della radice avremmo potuto utilizzare

$$x_{n+1} = x_n - m * \frac{f(x_n)}{f'(x_n)}$$

Invece del metodo di Aitken.

```
function [x,i]=newtonModificato(f,df,x0,tolx,itmax)
% [x,i]=newtonModificato(f,df,x0,tolx,nmax)
%
%   Esegue il metodo di accelerazione Aitken, una modifica del metodo
%   di Newton per il calcolo di f(x)=0 in presenza di radici multiple,
%   senza conoscerne la molteplicità
%
%   [x,i]=newtonModificato(x0,f,df,tolx,nmax)
%
% Input:
%       x0 = il punto iniziale
%       f = funzione di cui valutare uno zero
```

```
%       df = la derivata della funzione f
%       tolx = tolleranza per la radice
%       itmax = limite superiore al numero di iterazioni
%
% Output:
%       x = la soluzione trovata
%       i = il numero di iterazioni impiegate per ottenere la soluzione
%
%
%

if tolx <= 0 || itmax <= 0
    error('Errore: tolx e itmax devono essere positivi.')
end


i=0;
err=tolx+1;
x=x0;
while (i<itmax && err>tolx)
   x0=x;
   fx0=feval(f,x0);
   dfx0=feval(df,x0);

   if dfx0 == 0
       error('Errore: la derivata e` zero.')
   end


   tolf=tolx*abs(dfx0);

   if abs(fx0)<=tolf
        break
   end

   x1=x0-(fx0/dfx0);
   err=abs(x1-x0);

   if err<tolx
       break
   end

   fx1=feval(f,x1);
   dfx1=feval(df,x1);
   tolf=tolx*abs(dfx1);

   if dfx1 == 0
       error('Errore: la derivata e` zero.')
   end

   if abs(fx1)<=tolf
       break
```

```
   end

   x2=x1-(fx1/dfx1);
   err=abs(x2-x1);

   if err<tolx
       break
   end

   x=(x2*x0-(x1)^2)/(x2-2*x1+x0);
   err=abs(x0-x);
   i=i+1;

end
```

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 due esempi non banali, generati casualmente, di cui sia nota la soluzione.

Soluzione:

```
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
```

Validazione:

```
>> A = randi([-6,6],6)

A =

     4    -3     6     4     2     3
     5     1     0     6     3    -6
    -5     6     4     2     3    -3
     5     6    -5    -6    -1    -6
     2    -4    -1     5     2    -5
    -5     6     5     6    -4     4

>> x = randi([-6,6],6,1)

x =

     3
    -2
     6
    -6
    -1
    -2

>> b = A*x

b =

    22
   -14
   -12
    22
   -14
   -37

>> mialu(A,b)

ans =

    3.0000
   -2.0000
    6.0000
   -6.0000
   -1.0000
   -2.0000
```

```
>> A = randi([-15,15],5)

A =

     8     5     5    -5     0
     9     6   -10     3     6
   -10     8   -12    -9    12
     0    -7     0     8    14
    -2     6    14    -8     1

>> x = randi([-15,15],5,1)

x =

   -11
   -11
    -8
    11
    -8

>> b=A*x

b =

  -238
  -100
   -77
    53
  -252

>> mialu(A,b)

ans =

  -11.0000
  -11.0000
   -8.0000
   11.0000
   -8.0000
```

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 funtion, e validarla su due esempi non banali, generati casualmente, di cui sia nota la soluzione.

Soluzione:

```
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
```

Soluzione:

```
>> A = randi([-8,8], 4)

A =

    -1    -5     4     3
    -2     0     4    -6
     5    -1    -4    -6
     5     2     3     0

>> d = randi([5,30], 4,1)

d =

    29
    13
    20
    10

>> A = tril(A,-1) + triu (A',1) + diag (d)

A =

    29    -2     5     5
    -2    13    -1     2
     5    -1    20     3
     5     2     3    10

>> x = randi ([-8,8], 4, 1)

x =

     4
    -4
     0
     3

>> b = A*x

b =

   139
   -54
    33
    42

>> mialdl(A,b)

ans =

    4.0000
   -4.0000
    0.0000
    3.0000
```

```
>> A=randi([-8,8],5)

A =

     5    -7    -6    -6     3
     7    -4     8    -1    -8
    -6     1     8     7     6
     7     8     0     5     7
     2     8     5     8     3

>> d=randi([6,21],5,1)

d =

    18
    17
    12
    16
     8

>> A=tril(A,-1)+triu(A',1)+diag(d)

A =

    18     7    -6     7     2
     7    17     1     8     8
    -6     1    12     0     5
     7     8     0    16     8
     2     8     5     8     8

>> x=randi([-10,21],5,1)

x =

    12
    -9
    -2
    -9
    -7

>> b=A*x

b =

    88
  -199
  -140
  -188
  -186

>> mialdl(A,b)

ans =

   12.0000
   -9.0000
   -2.0000
   -9.0000
   -7.0000
```

Esercizio 11. Scrivere una *function* Matlab,

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

che, data in ingresso la matrice A m x n, con $m \geq n = rank(A)$, 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*. Validare la *function* `miaqr` su due esempi non banali, generati casualmente, confrontando la soluzione ottenuta con quella calcolata con l'operatore Matlab \

Soluzione:

```
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
```

Validazione:

```
>> A = randi([-15,15], 8 , 5)

A =

    -2   -12     0    11     4
     5     0     6    -8    -1
     6    14    12    10    -5
     8    -5    14    -8    10
    -7     3     1    13     3
     6    -9   -11    -5     2
     5     8   -11    -9    13
   -10    -8    -8    -8    -7

>> b = randi([-15,15], 8, 1)

b =

     8
     8
    -4
     2
   -13
   -14
     1
     9

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

x =

   -0.6004
   -0.1458
    0.4611
   -0.5437
   -0.0071


nr =

   19.5503

>> x = A\b

x =

   -0.6004
   -0.1458
    0.4611
   -0.5437
   -0.0071
```

```
>> A = randi([-20,18],10,6)

A =

    12    -1    -1    17    -7    -9
     7    -3    17     1   -13     9
    -8     5    -7   -15   -11     9
    17     7     2   -15     4    -6
   -19     9   -12   -10    -2     2
    -3   -10     9    12    -7   -18
    -6     6   -11   -11    12   -18
     9     5    -1    11     2     0
    11   -14     7   -11     1    10
   -13   -16    14    16    15    16

>> b=randi([-20,18],10,1)

b =

   -15
     2
    -2
   -20
    -7
   -14
    10
    -8
     0
   -14

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

x =

   -0.0906
   -0.1659
   -0.3541
   -0.1969
   -0.0383
    0.1093


nr =

   32.1823

>> x=A\b

x =

   -0.0906
   -0.1659
   -0.3541
   -0.1969
   -0.0383
    0.1093
```

Esercizio 11. Data la function Matlab

```
function [A1,A2,b1,b2] = linsis1(n,simme)
%
%
rng(0);
If nargin == 2
      q2=q1';
else
      [q2,r1] = qr(rand(n));
end
A1=q1*diag([1 2/n:1/n:1)*q2;
A2=q1*diag([1e-10 2/n:1/n:1)*q2;
b1=sum(A1,2);
b2=sum(A2,2);
return
```

che crea sistemi lineari casuali di dimensione n con soluzione nota,

$A_1 x = b_1\ A_2 x = b_2\ x = (1, \dots, 1)^T\ \in\ R^n$,

risolvere, utilizzando la function mialu, i sistemi lineari generati da [A1,A2,b1,b2]=linsis(5). Commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Soluzione:

Il risultato ottenuto con la funzione linsis nel caso della matrice A2 è errato, mentre quello ottenuto per la matrice A1 è corretto.
Questo risultato è dovuto al differente condizionamento delle 2 matrici per come è stata definita la funzione.
A riprova di questo fatto con la funzione cond(A1), che misura il condizionamento della matrice, si hanno i valori 2.5 per A1 mentre 9.99*10^9 per A2

Si riportano i risultati ottenuti dalle 2 chiamate.

```
>> mialu(A1,b1)

ans =

   1.000000000000000
   1.000000000000000
   1.000000000000000
   1.000000000000000
   1.000000000000000
```

```
>> mialu(A2,b2)

ans =

   0.999999647657477
   1.000000446226050
   1.000000098875194
   1.000000207059384
   1.000000011600807
```

Esercizio 12. Risolvere, utilizzando la *funtion* mialdlt, i sistemi lineari generati da [A1,A2,b1,b2] = linsis(5,1). Commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Soluzione:

```
>> mialdl(A1,b1)

ans =

   1.000000000000000
   0.999999999999999
   1.000000000000000
   1.000000000000000
   1.000000000000000

>> mialdl(A2,b2)

ans =

   1.000000052293492
   1.000000058138758
   1.000000008150719
   1.000000058625536
   1.000000040588329
```

Come nell'esercizio precedente possiamo vedere che le soluzioni per A2 non sono precise in quanto il condizionamento di A2 è alto.

Esercizio 13. Utilizzare la function miaqr per risolvere, nel senso dei minimi quadrati, i sistemi lineari sovradterminati

A x = b, (D*A)x = (D*b), (D1*A)x = (D1*b),

definiti dai seguenti dati:

A= [1 3 2; 3 5 4; 5 7 6; 3 6 4; 1 4 2];

b= [15 28 41 33 22]';

D= diag(1:5);

D1= diag(pi*[1 1 1 1 1]).

Calcolare le corrispondenti soluzioni e residui, e commentare i risultati ottenuti.

Soluzione:

Ax = b

```
x =

           3.00000000000001
                        5.8
          -2.50000000000001


nr =

           1.26491106406736
```

I valori della soluzione sembrano essere buoni anche se per una minima alterazione dovuta alla rappresentazione dei numeri reali non di macchina.

(D*A)x = (D*b)

```
x =

         -0.602569986232244
            4.7016980266177
           1.75837540156039


nr =

           3.73515111234241
```

Il risultato è diverso se confrontato con i risultati ottenuti con l'operatore \. Inoltre in questo caso la norma euclidea è diversa da quella precedente.

(D1*A)x = (D1*b)

```
x =

           3.00000000000002
           5.80000000000001
          -2.50000000000002


nr =

           3.97383530631843
```

In questo caso i calori si discostano minimamente dai risultati del primo sistema lineare, invece la norma euclidea è poco più che triplicata.

Esercizio 14. Scrivere una *function* Matlab,

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

che implementi efficientemente il metodo di Newton per risolvere sistemi di equazioni nonlineari. Curare particolarmente il criterio di arresto, che deve essere analogo a quello usato nel caso scalare. La seconda variabile, se specificata, ritorna il numero di iterazioni eseguite. Prevedere opportuni valori di *default* per gli ultimi due parametri di ingresso.

Soluzione:

```
function [x,nit] = newton(fun,jacobian,x0,tol,maxit)
%
% [x,nit] = newton(fun,jacobian,x0,tol,maxit)
%
% Questo metodo risolve sistemi non lineare di equazioni attraverso l'uso
% del metodo di Newton. Può restituire inoltre il numero di iterazioni
% eseguite
%
% Input:
%       fun = vettore delle funzioni
%       jacobian = matrice jacobiana di fun
%       x0 = vettore delle approssimazioni iniziali
%       tol = tolleranza specificata
%       maxit = quante iterazioni il metodo potrà fare al massimo, se non
%       specificata maxit = 1000
%
% Output:
%       x = vettore delle soluzioni
%       nit = numero di iterazioni svolte
%
if(nargin == 3)
    tol = eps;
end
if(nargin == 4)
    maxit = 1000;
end
if (tol<0)
 error('Errore: La tolleranza in input non può essere minore di 0.');
end
if (maxit<=0)
 error('Errore: Il massimo numero di iterazioni non può essere minore di 0.');
end
x = x0;
for nit=1:maxit
 x0 = x;
 fx0 = feval(fun,x0);
 fj = feval(jacobian,x0);
 x = x0-(fj\fx0);
 if (norm(x-x0)<=tol*(1+norm(x0)))
    disp('Tolleranza desiderata raggiunta.');
    break
 end
end
if not(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:

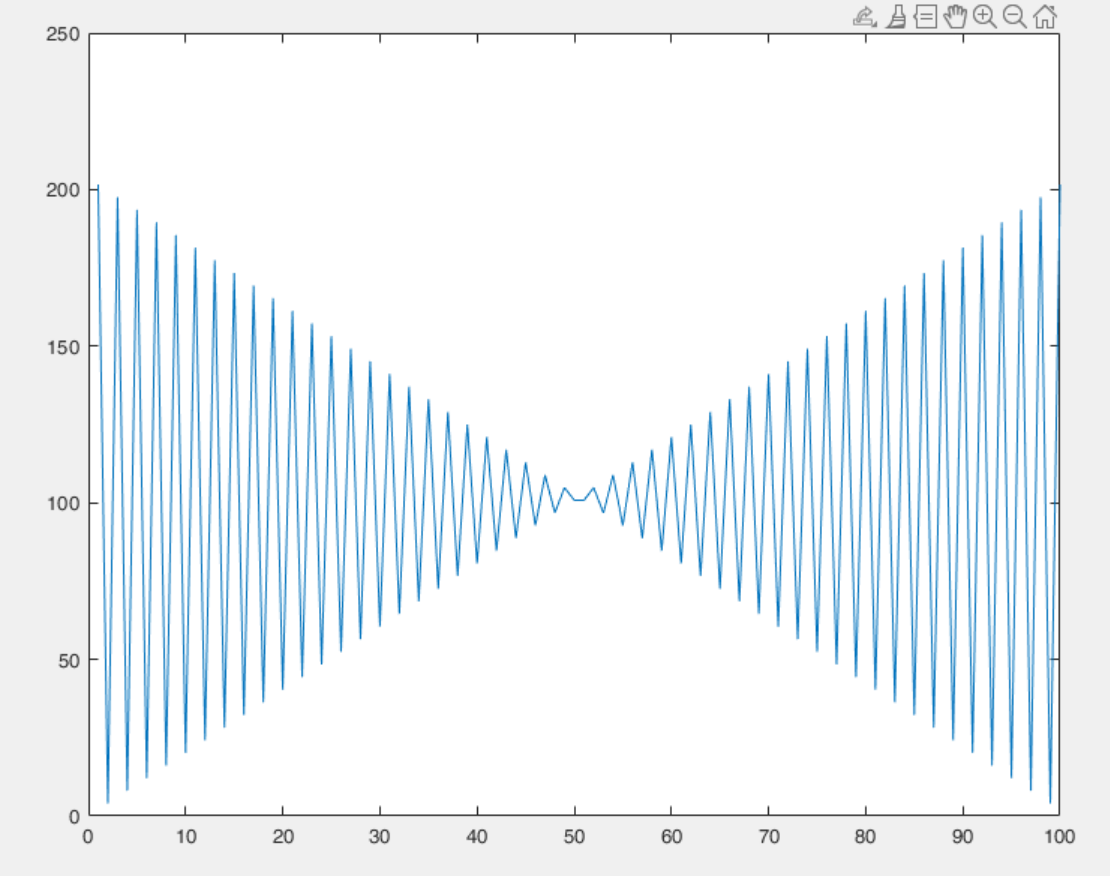
$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^\top Q\boldsymbol{x} - \boldsymbol{e}^\top \left[\sin\left(\frac{\pi}{2}\boldsymbol{x}\right) + \boldsymbol{x}\right], \qquad \boldsymbol{e} = \frac{1}{100}\begin{pmatrix} 1 \\ 2 \\ \vdots \\ 100 \end{pmatrix} \in \mathbb{R}^{100},$$

$$Q = \begin{pmatrix} 2 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 2 \end{pmatrix} \in \mathbb{R}^{100\times 100}, \qquad \sin\left(\frac{\pi}{2}\boldsymbol{x}\right) = \begin{pmatrix} \sin\left(\frac{\pi}{2}x_1\right) \\ \sin\left(\frac{\pi}{2}x_2\right) \\ \vdots \\ \sin\left(\frac{\pi}{2}x_{100}\right) \end{pmatrix}.$$

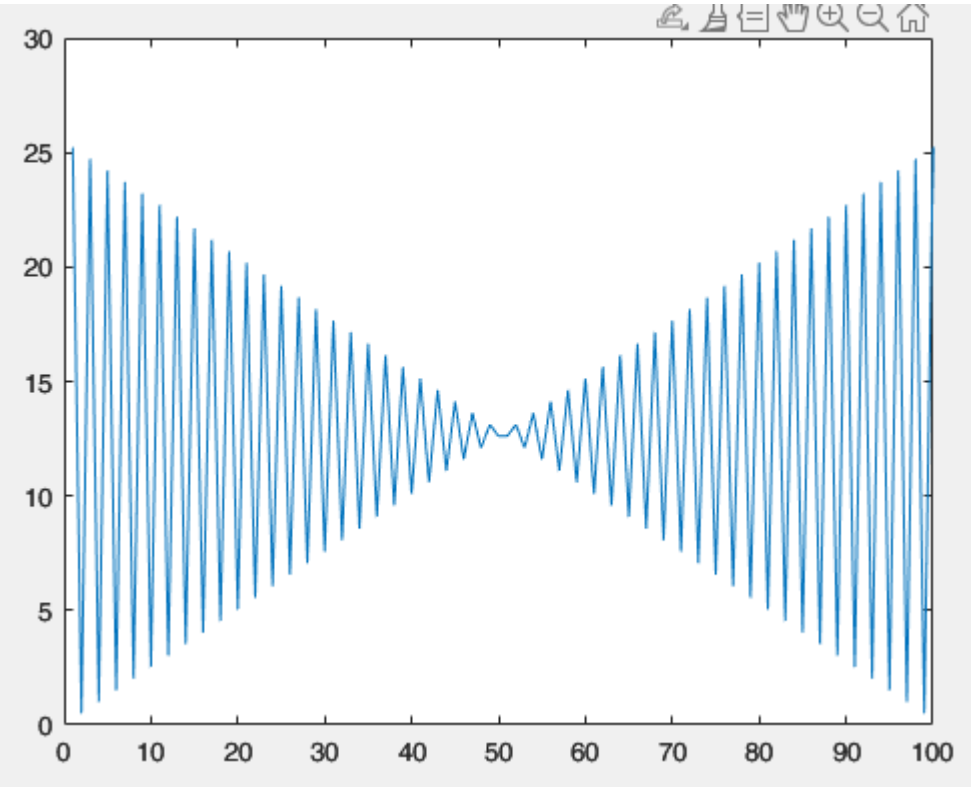
utilizzando tolleranze `tol = 1e-3, 1e-8, 1e-13`. Graficare la soluzione e tabulare in modo conveniente i risultati ottenuti.

Soluzione:

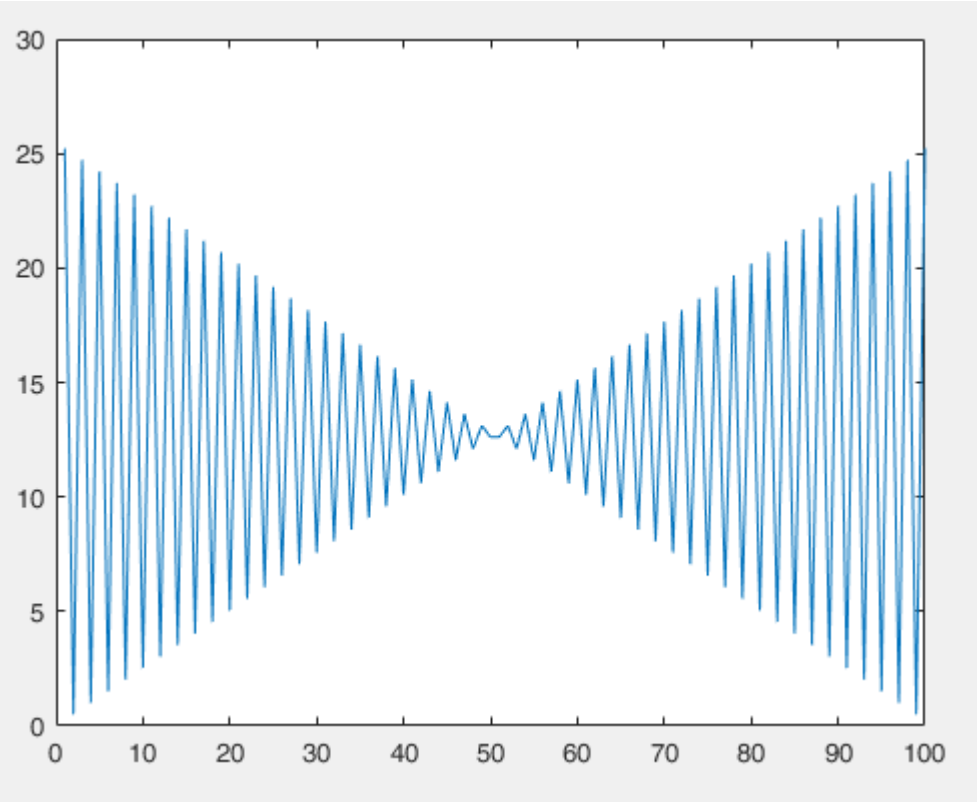
Tol = 1e-03



Tol = 1e-08



Tol = 1e-13



Tol: 1e-3

Indice	Valore
1	201.612479893568
2	4.03224959787136
3	197.580230295697
4	8.06449919574276
5	193.547980697825
6	12.0967487936144
7	189.515731099954
8	16.1289983914861
9	185.483481502082
10	20.161247989358
11	181.45123190421
12	24.1934975872299
13	177.418982306338
14	28.2257471851017
15	173.386732708466
16	32.2579967829736
17	169.354483110595
18	36.2902463808453
19	165.322233512723
20	40.3224959787171
21	161.289983914851
22	44.3547455765887
23	157.25773431698
24	48.3869951744603
25	153.225484719108
26	52.4192447723319
27	149.193235121236
28	56.4514943702037
29	145.160985523364
30	60.4837439680755
31	141.128735925493
32	64.5159935659472
33	137.096486327621
34	68.548243163819
35	133.064236729749
36	72.5804927616906
37	129.031987131878
38	76.6127423595621
39	124.999737534006
40	80.6449919574337
41	120.967487936135
42	84.6772415553052
43	116.935238338263
44	88.7094911531767
45	112.902988740392
46	92.7417407510481
47	108.87073914252
48	96.7739903489194
49	104.838489544649
50	100.806239946791

1e-8

Indice	Valore
1	25.2244874301087
2	0.504489748602265
3	24.7199976815064
4	1.00897949720452
5	24.2155079329042
6	1.51346924580675
7	23.7110181843019
8	2.01795899440899
9	23.2065284356997
10	2.52244874301122
11	22.7020386870975
12	3.02693849161344
13	22.1975489384953
14	3.53142824021566
15	21.6930591898931
16	4.03591798881787
17	21.1885694412908
18	4.54040773742008
19	20.6840796926886
20	5.04489748602228
21	20.1795899440864
22	5.5493872346245
23	19.6751001954842
24	6.05387698322671
25	19.170610446882
26	6.55836673182891
27	18.6661206982798
28	7.06285648043109
29	18.1616309496776
30	7.56734622903327
31	17.6571412010755
32	8.07183597763545
33	17.1526514524733
34	8.57632572623763
35	16.6481617038711
36	9.0808154748398
37	16.1436719552689
38	9.58530522344197
39	15.6391822066668
40	10.0897949720441
41	15.1346924580646
42	10.5942847206463
43	14.6302027094624
44	11.0987744692485
45	14.1257129608603
46	11.6032642178506
47	13.6212232122581
48	12.1077539664528
49	13.1167334636559
50	12.612243715055

1e-13

Indice	Valore
1	25.2244799412148
2	0.504489598824269
3	24.7199903423905
4	1.00897919764855
5	24.2155007435663
6	1.51346879647284
7	23.711011144742
8	2.01795839529715
9	23.2065215459177
10	2.52244799412145
11	22.7020319470934
12	3.02693759294573
13	22.1975423482691
14	3.53142719177003
15	21.6930527494448
16	4.03591679059432
17	21.1885631506205
18	4.54040638941861
19	20.6840735517962
20	5.04489598824292
21	20.1795839529719
22	5.54938558706722
23	19.6750943541476
24	6.05387518589152
25	19.1706047553233
26	6.55836478471582
27	18.666115156499
28	7.06285438354013
29	18.1616255576747
30	7.56734398236444
31	17.6571359588504
32	8.07183358118875
33	17.152646360026
34	8.57632318001305
35	16.6481567612017
36	9.08081277883736
37	16.1436671623774
38	9.58530237766168
39	15.6391775635531
40	10.089791976486
41	15.1346879647288
42	10.5942815753103
43	14.6301983659045
44	11.0987711741346
45	14.1257087670802
46	11.6032607729589
47	13.6212191682559
48	12.1077503717832
49	13.1167295694316
50	12.6122399706075

VALUTAZIONI FUNZIONALI	1e-3	1e-8	1e-13
	225	772	1128

Si è deciso di tabulare i risultati da 1 a 50 invece che da 1 a 100 perché i risultati dopo il cinquantesimo si ripetono in questo modo: x_1 == x_100, x_2 == x_99 ecc…

Questo lo possiamo vedere anche dai grafici precedenti.

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.
N.B.: il risultato dovrà avere le stesse dimensioni del dato di ingresso; questo vale anche per gli esercizi a seguire.

Soluzione:

```
function l = lagrange(x,y,xq)
%
% l = lagrange(x,y,xq)
%
% Implementa, in modo vettoriale, la frma di Lagrange del polinomio
% interpolante una funzione
%
% Input:
%       x = vettore delle ascisse di iterpolazione
%       y = valori della funzione nelle ascisse di interpolazione
%       xq = vettore contenente i punti in cui calcolare il polinomio
% Output:
%       p = polinomio interpolante nella forma di Lagrange
%
n = length(x);
if length(y) ~= n
    error('Errore: Le dimensioni di x e y non corrispondono.');
end
if length(unique(x)) ~= n
    error('Errore: Alcune ascisse sono uguali.');
end
l = zeros(size(xq));
for j = 1:n
    L = ones(size(xq));
    for i = 1:n
        if i ~= j
            L = L .* (xq - x(i)) / (x(j) - x(i));
        end
    end
    l = l + y(j) * L;
end
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.

Soluzione:

```
function yq = newton(x,y,xq)
%
% yq = newton(x,y,xq)
%
% Implementa la forma di Newton del polinomio interpolante in modo
% vettoriale
%
% Input:

%      x = vettore ascisse interpolazione
%      y = valori della funzione alle ascisse di interpolazione
%      xq = punti su cui si vuole calcolare la funzione
%
% Output:

%      q = valori approssimati della funzione
%
n=length(x);
if length(unique(x))~=n
error("Errore: le ascisse di interpolazione non possono essere uguali tra
loro");
end
if length(y)~=n
error("Errore: le dimensioni del vettore delle ascisse interpolanti e quello
delle immagini corrispondenti sono diverse");
end
% calcolo differenze divise
diffDiv=y;
for j=1:n-1
for i=n:-1:j+1
diffDiv(i)=(diffDiv(i)-diffDiv(i-1))/(x(i)-x(i-j));
end
end
n = length(diffDiv)-1;
yq = diffDiv(n+1)*ones(size(xq));
for k=n:-1:1
yq = yq.*(xq-x(k))+diffDiv(k);
end
end
```

Esercizio 18. Costruire una *function*, `hermite.m`, avente sintassi

```
yy = hermite( xi, fi, f1i, xx )
```

che implementi, in modo vettoriale, il polinomio interpolante di Hermite.

Soluzione:

```
function yy = hermite(xi,fi,f1i,xx)
%
% yy = hermite(xi,fi,f1i,xx)
%
% Questa funzione implementa il polinomio interpolante di Hermite in
% modo vettoriale
%
% Input:
%       xi = vettore ascsse di interpolazione
%       fi = valori della funzione in corrispondenza delle ascisse di
%           interpolazione
%       f1i = vettore contenente le derivate prime su X
%       xx = vettore che contiene i punti su cui vogliamo
%            approssimare la funzione in input
%
% Output:
%       yy = valori approssimati della funzione usando il polinomio
%         interpolante
%
n = length(xi);
if length(fi)~=n
    error("Errore: le dimensioni del vettore delle ascisse interpolanti e quello
delle immagini corrispondenti non coincidono");
end
if length(f1i)~=n
    error("Errore: le dimensioni del vettore delle ascisse interpolanti e quello
delle derivate corrispondenti non coincidono");
end
if length(unique(xi))~=n
    error("Errore: le ascisse di interpolazione non sono distinte tra di loro");
end
n=length(xi);
diffDiv(1:2:2*n-1) = fi;
diffDiv(2:2:2*n)=f1i;
x = repelem(xi,2);
%differenze divise per hermite
n=(length(diffDiv)/2)-1;
for i=2*n+1:-2:3
    diffDiv(i) = (diffDiv(i)-diffDiv(i-2))/(x(i)-x(i-1));
end
for j=2:2*n+1
    for i=(2*n+2):-1:j+1
        diffDiv(i) = (diffDiv(i)-diffDiv(i-1))/(x(i)-x(i-j));
    end
end
n=length(diffDiv)-1;
yy = diffDiv(n+1)*ones(size(xx)); %algoritmo di horner
for i=n:-1:1
    yy = yy.*(xx-x(i))+diffDiv(i);
end
end
```

Esercizio 19. 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.

Soluzione:

```
function x = chebyshev(n,a,b)
%
% x = chebyshev(n,a,b)
%
% Calcola le ascisse di Chebyshev nell'intervallo [a,b]
% per un polinomio interpolante di grado n
%
% Input:
% n = grado del polinomio
% a = estremo sx dell'intervallo
% b = estremo dx dell'intervallo
%
% Output:
% x = vettore contenente le ascisse di Chebyshev
%
if(n<0)
error("Errore: il grado del polinomio non puo' essere negativo o
nullo");
end
if(a>=b)
error("Errore: l'estremo sx dell'intervallo non puo' essere maggiore
dell'estremo
dx");
end
x = (a+b)/2+((b-a)/2)*cos(pi*(2*(n:-1:0)+1)./(2*(n+1)));
end
```

Esercizio 20. Costruire una function Matlab, con sintassi

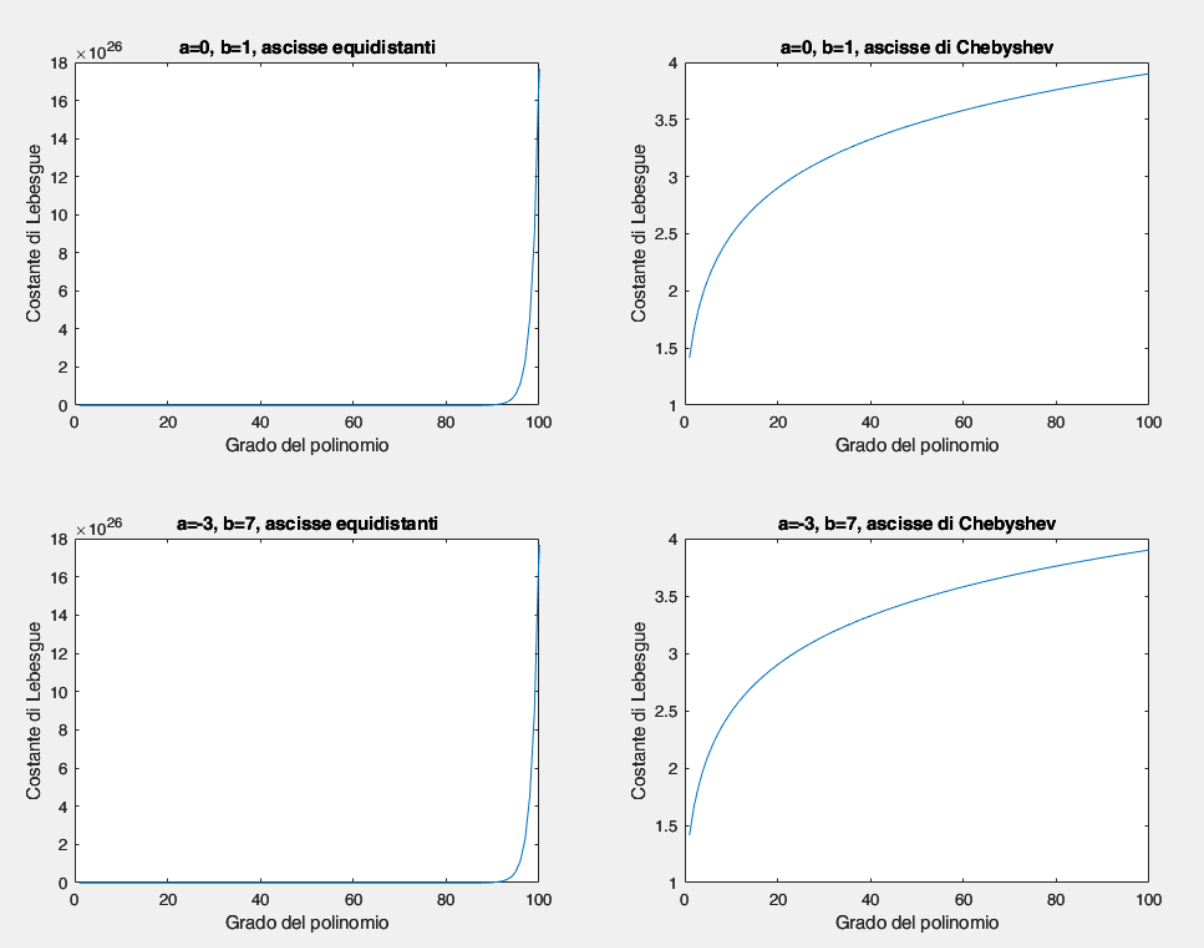
`ll = lebesgue( a, b, nn, type ),`

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 ciascuna componente di ll). Graficare i risultati ottenuti, per nn=1:100, utilizzando [a,b]=[0,1] e [a,b]=[-3,7]. Giustificare i risultati ottenuti.

Soluzione:

```
function ll = lebesgue(a, b, nn, type)
%
%   Questa funzione calcola la costante di Lebesgue per l'interpolazione
%   polinamiale
%
%   Input:
%           a,b = inizio e fine intervallo
%           nn = vettore con specificato il grado dei polinomi
%           type = specifica che tipo di ascisse di interpolazione usare
%                   se 0 utilizza le ascisse equispaziate nell'intervallo [a,b]
%                    altrimenti se 1 utilizza le ascisse di chebyshev
%   Output:
%           ll = vettore delle costanti di Lebesgue per ogni grado
%           specificato in input
%

    if a >= b
        error('Errore: a deve essere minore di b.');
    end
    if any(mod(nn, 1) ~= 0) || any(nn < 0)
        error('Errore: nn deve contenere solo numeri interi non negativi.');
    end
    if type ~= 0 && type ~= 1
        error('Errore: type deve essere 0 o 1.');
    end

ll = ones(numel(nn), 1);
xq = linspace(a, b, 10001);


for j = 1:numel(nn)
    if type == 0
    % Ascisse equidistanti
    x = linspace(a, b, nn(j)+1);
    else
    % Ascisse di Chebyshev
    x = chebyshev(nn(j), a, b);
    end
    lin = lebesgue_function(x,xq);
    ll(j) = max(abs(lin));
end
end

function lin = lebesgue_function(x,xq)

n = length(x);
```

```
if length(unique(x)) ~= n
    error('Errore: Alcune ascisse sono uguali.');
end

lin = zeros(size(xq));

for j = 1:n
    L = ones(size(xq));
    for i = 1:n
        if i ~= j
            L = L .* (xq - x(i)) / (x(j) - x(i));
        end
    end
    lin = lin + abs(L);
end
end
```



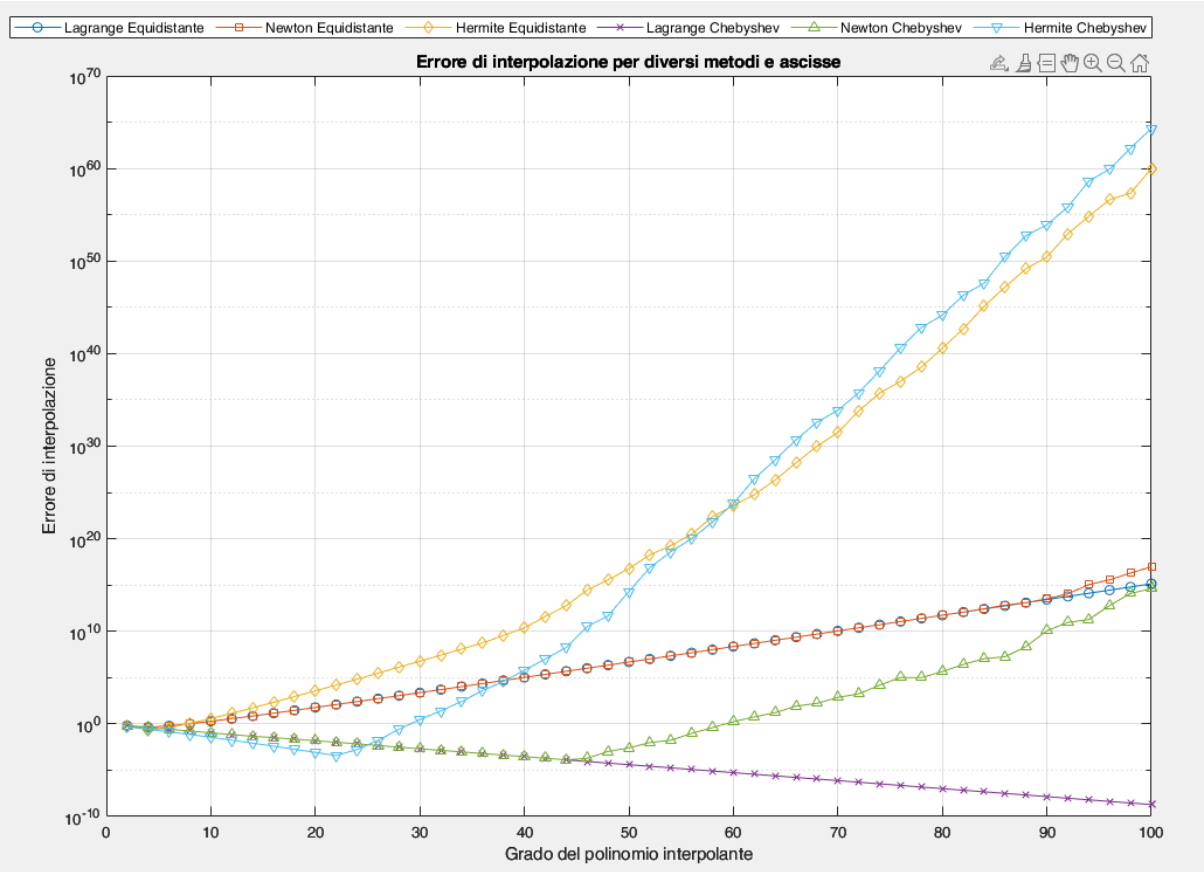
Come ci aspettavamo, la costante di Lebesgue è indipendente dall'intervallo [a,b] scelto e con la scelta delle ascisse di Chebyshev cresce in maniera quasi ottimale, cioè $\approx \log n$

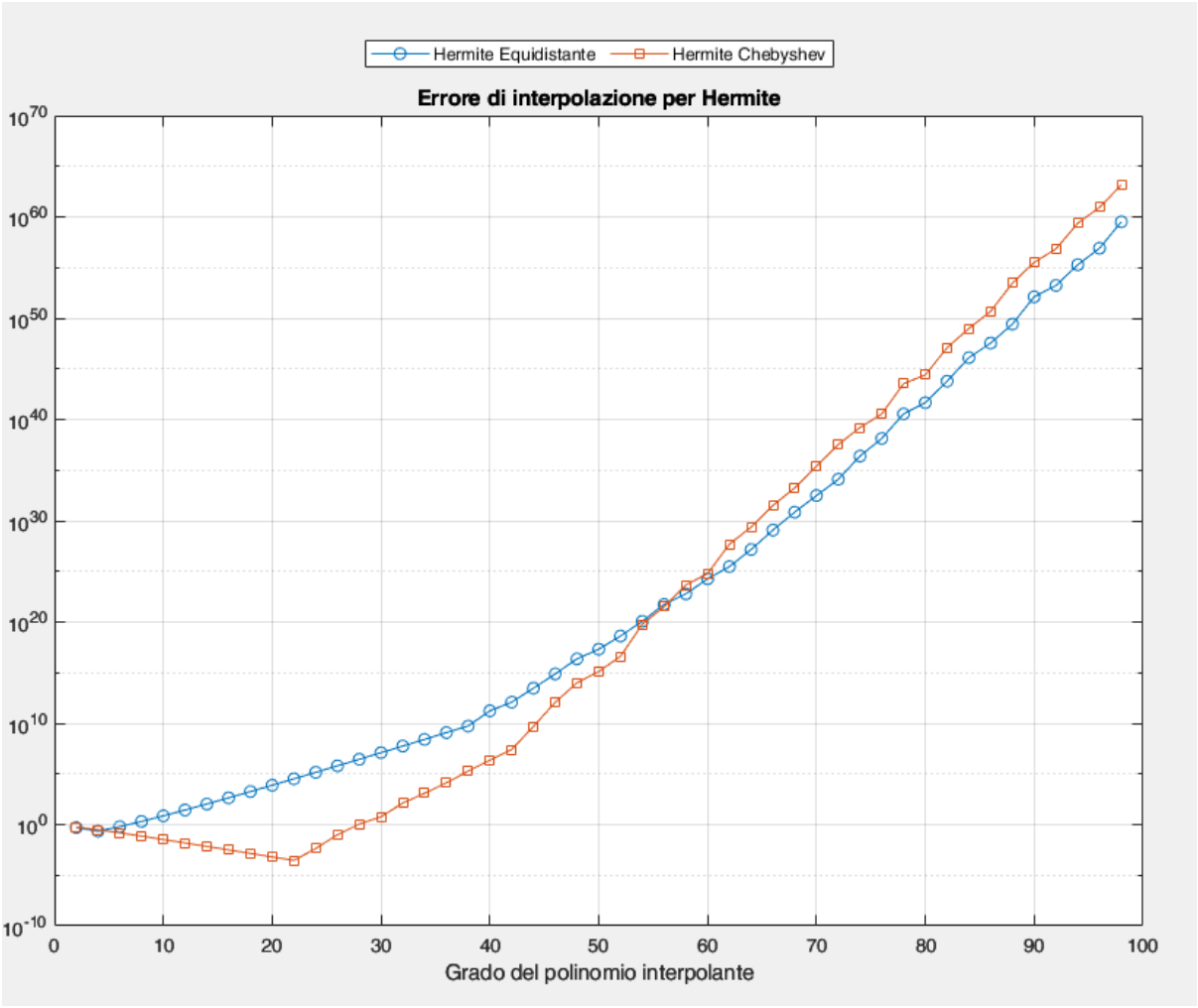
Esercizio 21. Utilizzando le function dei precedenti esercizi, graficare (in semilogy) l'andamento errore di interpolazione (utilizzare 10001 punti equispaziati nell'intervallo per ottenerne la stima) per la funzione di Runge,

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

utilizzando sia le ascisse equidistanti che di Chebyshev, per i polinomi interpolanti di grado nn=2:2:100. Graficare l'errore di interpolazione anche per i polinomi interpolanti di Hermite di grado nn=3:2:99, sia utilizzando ascisse equidistanti che ascisse di Chebyshev, nell'intervallo considerato.

Soluzione:




Hermite Equidistante
Hermite Chebyshev
Errore di interpolazione per Hermite
10^{70}
10^{60}
10^{50}
10^{40}
10^{30}
10^{20}
10^{10}
10^{0}
10^{-10}
0
10
20
30
40
50
60
70
80
90
100
Grado del polinomio interpolante

Esercizio 22. Costruire una *function*, `myspline.m`, avente sintassi

`yy = myspline( xi, fi, xx, type )`
dove type=0 calcola la spline cubica interpolante naturale i punti (xi,fi), e type≠0 calcola quella not-a-knot (default).

Soluzione:

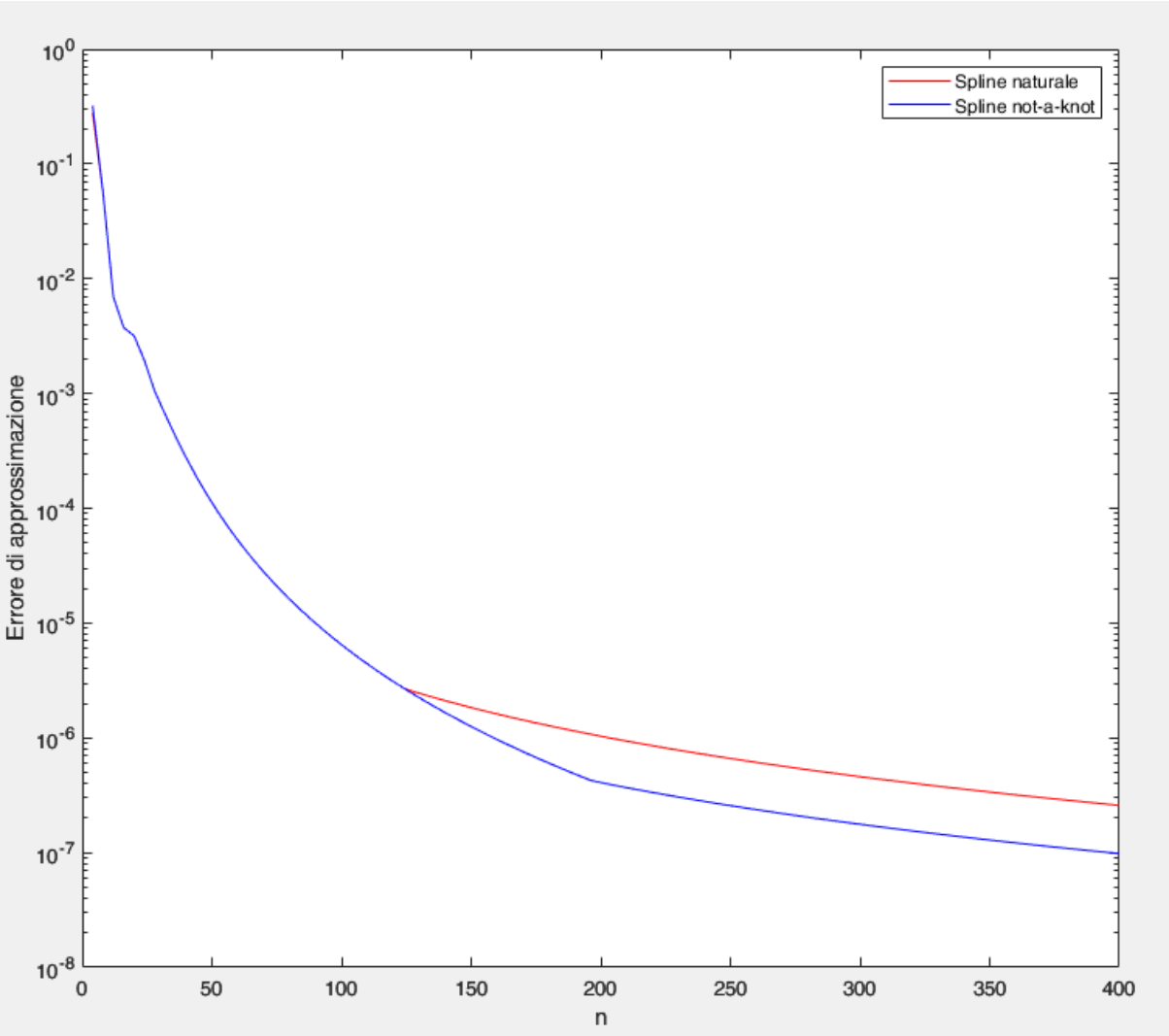
```
function yy = myspline ( xi , fi , xx , type)
%
% yy = myspline ( xi , fi , xx , tipo )
%
% Calcola la spline cubica interpolante naturale o not a knot a seconda del
% valore "tipo"
% Input :
%
%   xi = vettore delle ascisse di interpolazione
%   fi = vettore dei valori della funzione nelle ascissedi interpolazione
%   xq = punti in cui si vuole calcolare il polinomio interpolante
%   tipo = tipo di spline, 0 per naturale oppure 1 per not a knot
%
% Output :
%   yy = valori che la spline assume nei punti xx
if nargin < 4
error ("Errore: numero degli argomenti errato") ;
elseif length ( xi ) ~= length ( fi )
error ("Errore: i due vettori devono avere la stessa dimensione") ;
elseif length ( xi ) ~= length ( unique ( xi ) )
error ("Errore: le ascisse devono essere distinte tra loro" ) ;
elseif size ( xi , 2) > 1 || size ( fi , 2) > 1
error ("Errore: vettore colonna non valido" )
elseif isempty ( xx )
error ("Errore: partizione assegnata vuota") ;
end
n = length ( xi ) - 1;
epsilon = zeros (n -1 ,1) ;
q = zeros (n -1 ,1) ;
for i = 2 : n
hi = ( xi ( i ) - xi (i -1) ) ;
hi1 = ( xi ( i +1) - xi ( i ) ) ;
q (i -1) = hi / ( hi + hi1 ) ;
epsilon (i -1) = hi1 / ( hi + hi1 ) ;
end
l = size ( xi ) ;
diff_div = fi ;
l = l -1;
for j = 1 : 2
for i = l +1 : -1 : j +1
diff_div ( i ) = ( diff_div ( i ) - diff_div (i -1) ) /( xi ( i ) - xi (i -
j ) ) ;
end
end
diff_div = diff_div (3: l +1) ; % Calcolo le differenze divise

a = 2* ones (n -1 ,1) ;
if type == 0
m = tridia (a , q , epsilon , diff_div * 6) ;
m = [0; m ; 0];
```

```
else
      diff_div = diff_div * 6;

    a(1) = 2 - q(1);
    epsilon(1) = epsilon(1) - q(1);
    diff_div_1 = diff_div(1);
    diff_div(1) = (1 - q(1)) * diff_div(1);
    q(1) = 0;

    a(n-1) = 2 - epsilon(n-1);
    q(n-1) = q(n-1) - epsilon(n-1);
    diff_div_n = diff_div(n-1);
    diff_div(n-1) = (1 - epsilon(n-1)) * diff_div(n-1);
    epsilon(n-1) = 0;

    m = tridia(a, epsilon, q, diff_div);

    m0 = diff_div_1 - m(1) - m(2);
    mn = diff_div_n - m(n-1) - m(n-2);
    m = [m0; m; mn];
end

yy = zeros ( length ( xx ) , 1) ;
for j = 1 : length ( xx )
for i = 2 : length ( xi )
if (( xx ( j ) >= xi (i -1) && xx ( j ) <= xi ( i ) ) || xx ( j ) < xi (1) )
hi = xi ( i ) - xi (i -1) ;
ri = fi (i -1) - hi ^2/6* m (i -1) ;
qi = ( fi ( i ) - fi (i -1) ) / hi - hi /6*( m ( i ) -m (i -1) ) ;
yy ( j ) =(( xx ( j ) - xi (i -1) ) ^3* m ( i ) +( xi ( i ) - xx ( j ) ) ^3* m (i
-1) ) /(6* hi ) + qi*( xx ( j ) - xi (i -1) ) + ri ;
break
end
end
end
return ;
end



function x = tridia (a , b , c , g )
% x = tridia (a , b , c , g )
%
% Risolve il sistema lineare fattorizzabile LU
% Input :
%   a = Vettore diagonale principale
%   b = Vettore sovradiagonale
%   c = Vettore sottodiagonale
%   g = Vettore dei termini noti
%
% Output :
%   x = Vettore soluzione
n = length ( g ) ;
x = g ;
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 23. Graficare, utilizzando il formato semilogy, l'errore di approssimazione utilizzando le spline interpolanti naturale e not-a-knot per approssimare la funzione di Runge sull'intervallo [-5,5],
utilizzando una partizione $\Delta = \{-5 = x0 < \cdots < xn = 5\}$, con ascisse equidistanti e n=4:4:400.
Utilizzare 10001 punti equispaziati nell'intervallo [−5, 5] per ottenere la stima dell'errore.

Soluzione:



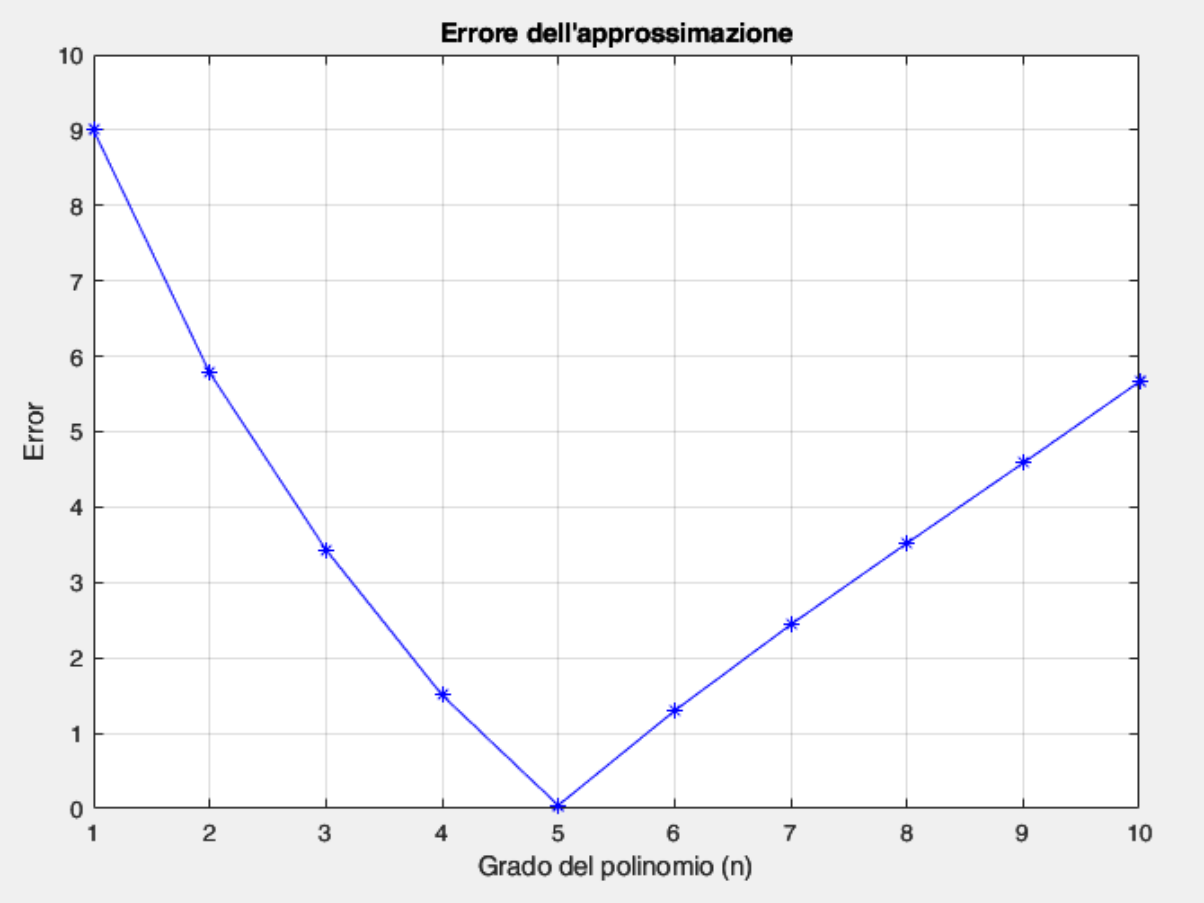
Esercizio 24. E' noto che un fenomeno fisico evolve come y = xn, con n incognito. Il file data.mat, reperibile a:
https://drive.google.com/file/d/14u2Pjnl_BZN1GWEFCZd0tqKCObGGI-M5/view?usp=share_link
contiene 1000 coppie di dati (xi, yi), in cui la seconda componente è affetta da un errore con distribuzione Gaussiana a media nulla e varianza "piccola". Utilizzando un opportuno polinomio di approssimazione ai minimi quadrati, stimare il grado n. Argomentare il procedimento seguito, graficando la norma del residuo rispetto a valori crescenti di n. E' richiesto il codice Matlab dell'algoritmo che avrete implementato (potete utilizzare, se lo ritenete opportuno, la function polyfit di Matlab).

Soluzione:

```
data = load('data.mat').data;

x = data(:,1);
y = data(:,2);

% Inizializza vettore errore
error = zeros(1,15);

for m = 1:15

%{
    Calcola i coefficienti del polinomio di approssimazione ai minimi quadrati
    che meglio approssima i dati ordine decrescente di grado.
%}
    p = polyfit(x, y, m);

  % Calcola il vettore di valori del polinomio approssimante sui punti x dei
dati.
    y_fit = horner(p, x);


    % Calcola l'errore di approssimazione tra i dati originali e quelli
approssimati
    error(m) = norm(x.^m - y_fit);
end

% semilogy
figure;
semilogy(1:15, error, 'b*-');
xlabel('Grado del polinomio (n)');
ylabel('Error');
title("Errore dell'approssimazione");
grid on;

function y = horner(p, x)
    y = p(1);
    for i = 2:length(p)
        y = y .* x + p(i);
    end
end
```

Questo script carica un set di dati da un file .mat e tenta di approssimare i dati con polinomi di grado variabile, da 1 a 15. Per ogni grado del polinomio, calcola i coefficienti del

polinomio di approssimazione ai minimi quadrati tramite polyfit, e poi valuta il polinomio sui punti x utilizzando l'algoritmo di Horner. Successivamente, calcola l'errore di approssimazione tra i dati originali e quelli approssimati, utilizzando la norma euclidea.



Possiamo osservare come sia inutile incrementare oltre 5 il grado del polinomio di approssimazione ai minimi quadrati in quanto l'errore non diminuisce, anzi aumenta, all'aumentare del grado del polinomio.

Esercizio 25. Costruire una *function* Matlab che, dato in input n, restituisca i pesi della quadratura della formula di Newton-Cotes di grado n. Tabulare, quindi, i pesi delle formule di grado
1, 2,. . . , 7 e 9 (come numeri razionali).

Soluzione:

```
function c = coefNewtonCotes(n)
%
% c = coefNewtonCotes(n)
%
% Questa funzione calcola i pesi della quadratura della formula di Newton-Cotes di
% grado n
%
% Input:
% n = grado della formula di Newton-Cotes
%
% Output:
% c = vettore contenente i coefficienti
%
if n<1
error('Errore: inserito un grado minore di 1; deve essere almeno 1.');
end
c = zeros(1,n);
for i=0:n
d=i-[0:i-1 i+1:n];
k=1;
for j=1:n
k = k*d(j);
end
a=poly([0:i-1 i+1:n]);
a=[a./((n+1):-1:1) 0];
num=polyval(a,n);
c(i+1)=num/k;
end
end
```

Di seguito la tabella dei pesi della quadratura delle formula di Newton-Cotes

n	**i**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
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	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 26. 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 `e la funzione integranda (che accetta input vettoriali);
- [a,b] `e l'intervallo di integrazione;
- k, n come su descritti;
- If `e l'approssimazione dell'integrale ottenuta;
- err `e la stima dell'errore di quadratura.

Soluzione:

```
function [If, err] = composita(fun, a, b, k, n)
% [If, err] = composita(fun, a, b, k, n)
%
% Input:
% fun = funzione integranda (che accetta input vettoriali)
% a, b = estremi di integrazione
% k = grado della formula
% n = numero di sottointervalli
%
% Output:
% If = approssimazione dell'integrale
% err = stima dell'errore di quadratura

if ~isnumeric(k) || mod(k,1) ~= 0 || k <= 0
    error('Errore: k deve essere un intero positivo');
end

if b < a
    errore('Errore: gli estremi di integrazione non vanno bene')
end

if mod(n,2)==0
    mu = 2;
else
    mu = 1;
end

% Calcola i coefficienti della formula di Newton-Cotes
coef = coeffNewtonCotes(k);

% Calcola l'approssimazione dell'integrale
If = 0;
I1 = 0;
h = (b - a) / n;
h1 = (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.*coef);


    if i<n/2
        x1 = linspace(a + i*h1, a + (i+1)*h1, k+1);
```

```
        y1 = feval(fun, x1);
        I1 = I1 + h1/k*sum(y1.*coef);
    end
end


% Stima l'errore
err = abs((If-I1)/(2^(k+mu)-1));

end
```

Esercizio 27. Utilizzare la function composita per ottenere l'approssimazione dell'integrale

$$\int_0^1 (\sum_{i=1}^{5} i \cos 2\pi i x - e^i \sin 2(\pi i + 0.1)x) dx$$

con le formule composite di Newton-Cotes di grado k=1,2,3,6. Per tutte, utilizzare n=12.

Soluzione:

	Integrale	Errore
1	-0.0925980476169048	0.0824483332575294
2	-0.179803240495083	0.00701161675929261
3	-0.17847075804203	0.00135497414488415
6	-0.177446511092622	6.51467298937115e-07

Esercizio 28. Implementare la formula composita adattativa di Simpson.

Soluzione:

```
function [I, nfeval] = simpson(fun,a,b,tol,fa,fM,fb)
%
% [I, nfeval] = simpson(fun,a,b,tol,fa,f1,fb)
%
% Calcola l'approssimazione dell'integrale definito di f
% sull'intervallo [a,b] con la formula adattiva di simpson
%
% Input:
% fun = funzione integranda
% a = estremo inferiore dell'intervallo di integrazione
% b = estremo superiore dell'intervallo di integrazione
% tol = tolleranza richiesta
% [fa,fM,fb] = funzione calcolata negli estremi di integrazione e nel punto
% medio. Sono parametri opzionali che diminuiscono il numero di valutazioni
% necessarie
%
% Output:
% I: approssimazione dell'integrale
% nfeval = numero di valutazioni funzionali effettuate
%
if a>b
error("Errore: gli estremi dell'intervallo non vanno bene");
end
if tol<0
error("Errore: la tolleranza specificata è minore o uguale a 0");
end
x2= (a+b)/2;
if nargin == 4 %se è la prima chiamata i valori fa,fb,nfeval non ci saranno,
mentre alle chiamate successive sì, quindi possiamo evitare il calcolo
fa = feval(fun,a);
fM = feval(fun,x2);
fb = feval(fun,b);
nfeval=3;
end
h=b-a;
x1 = (a+x2)/2;
x3 = (x2+b)/2;
f1=feval(fun,x1);
f3=feval(fun,x3);
nfeval=nfeval+2;
Itemp=(h/6)*(fa+4*fM+fb);
I=(h/12)*(fa+4*f1+2*fM+4*f3+fb);
err=abs(I-Itemp)/15;
if err>tol
[Isinistra,nfevals]=simpson(fun,a,x2,tol/2,fa,f1,fM);
[Idestra,nfevald]=simpson(fun,x2,b,tol/2,fM,f3,fb);
I=Isinistra+Idestra;
nfeval=nfeval+nfevals+nfevald;
end
end
```

Esercizio 29. Implementare la formula composita adattativa di Newton-Cotes di grado k=4.

Soluzione:

```
function [I, nfeval] = NewtonCotesAdatt(fun, a, b, tol, fa, f2, fm, f3, fb)
%
% [I, nfeval] = NewtonCotesAdatt(fun, a, b, tol, fa, f2, f4, f5, fm)
%
% Calcola l'approssimazione dell'integrale definito di f
% sull'intervallo [a,b] con la formula adattiva di Newton-Cotes con k=4
%
% Input:
%       fun = funzione integranda
%       a = estremo inferiore dell'intervallo di integrazione
%       b = estremo superiore dell'intervallo di integrazione
%       tol = tolleranza richiesta
%       [fa,f2,f4,f5,fm] = valori calcolati negli estremi di integrazione e nei
punti
%       intermedi. Sono parametri opzionali che diminuiscono il numero di
valutazioni
%       necessarie
%
% Output:
%       I: approssimazione dell'integrale
%       nfeval = numero di valutazioni funzionali effettuate
%

if a > b
    error("Errore: gli estremi dell'intervallo non vanno bene");
end

if tol < 0
    error("Errore: la tolleranza specificata è minore o uguale a 0");
end

xm = (a+b)/2;
x2 = (a+xm)/2;
x3 = (xm+b)/2;
nfeval = 0;

if nargin == 4
    fa = feval(fun, a);
    fb = feval(fun, b);
    fm = feval(fun, xm);
    f2 = feval(fun, x2);
    f3 = feval(fun, x3);
    nfeval = nfeval + 5;
end

h = (b-a)/90;

x4 = (a+x2)/2;
x5 = (x2 + xm)/ 2;
x6 = (xm + x3)/ 2;
x7 = (x3 + b) / 2 ;
```

```
f4 = feval(fun, x4);
f5 = feval(fun, x5);
f6 = feval(fun, x6);
f7 = feval(fun, x7);
nfeval = nfeval + 4;

Itemp = h * (7*fa + 32*f2 + 12*fm + 32*f3 + 7*fb);
I = h/2 * (7*fa + 32*f4 + 12*f2 + 32*f5 + 14*fm + 32*f6 + 12*f3 + 32*f7 + 7*fb);
err = abs(I - Itemp) / 63;

if err > tol
    [I1, nfeval1] = NewtonCotesAdatt(fun, a, xm, tol / 2, fa, f4, f2, f5, fm);
    [I2, nfeval2] = NewtonCotesAdatt(fun, xm, b, tol / 2, fm, f6, f3, f7, fb);
    I = I1 + I2;
    nfeval = nfeval + nfeval1 + nfeval2;
end

end
```

Esercizio 30. Confrontare le formule adattative degli ultimi due esercizi, tabulando il numero di valutazioni funzionali effettuate, rispetto alla tolleranza tol = 1e-2, 1e-3, ..., 1e-9, per ottenere l'approssimazione dell'integrale

$$\int_{10^{-5}}^{1} x^{-1}\cos(\log x^{-1})dx \equiv \sin\sin\log 10^{5}$$

Costruire un'altra tabella, in cui si tabula l'errore vero (essendo l'integrale noto, in questo caso) rispetto a tol.

Soluzione:

Tolleranza	Errore	Errore vero (NC)	Errore vero (Simpson)
1.000000e-02		2.305180e-06	2.935280e-04
1.000000e-03		1.065361e-06	4.572239e-04
1.000000e-04		9.852604e-10	2.635476e-05
1.000000e-05		3.104168e-06	2.344814e-06
1.000000e-06		3.697328e-08	3.007626e-07
1.000000e-07		2.281509e-08	3.656476e-08
1.000000e-08		2.133584e-09	3.211098e-09
1.000000e-09		1.496632e-10	3.098395e-10

Tolleranza	Valutazioni funzionali (NC)	Valore calcolato (NC)	Valutazioni funzionali (Simpson)	Valore calcolato (Simpson)
1.000000e-02	153	-8.691346e-01	201	-8.694258e-01
1.000000e-03	193	-8.691333e-01	333	-8.695895e-01
1.000000e-04	257	-8.691323e-01	605	-8.691586e-01
1.000000e-05	369	-8.691354e-01	1061	-8.691346e-01
1.000000e-06	553	-8.691323e-01	1869	-8.691326e-01
1.000000e-07	793	-8.691323e-01	3277	-8.691323e-01
1.000000e-08	1177	-8.691323e-01	5921	-8.691323e-01
1.000000e-09	1753	-8.691323e-01	10589	-8.691323e-01