Elaborato Calcolo Numerico

Alessio Santoro (7029440) - Lorenzo Campinoti (7030227)

A.A. 2022/2023

Nota: Per gli esercizi che prevedono delle *function* Matlab, si specifica nella relativa risposta al quesito i file tra gli allegati a cui essa si riferisce.

Esercizio 1

Si considera lo sviluppo delle funzioni $f(x-h), f(x+h), f(x+2h), f(x+3h)$:

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5)$$
$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5)$$
$$f(x+2h) = f(x) + 2hf'(x) + \frac{4h^2}{2}f''(x) + \frac{8h^3}{6}f^{(3)}(x) + \frac{16h^4}{24}f^{(4)}(x) + O(h^5)$$
$$f(x+3h) = f(x) + 3hf'(x) + \frac{9h^2}{2}f''(x) + \frac{27h^3}{6}f^{(3)}(x) + \frac{81h^4}{24}f^{(4)}(x) + O(h^5)$$

Si sostiuiscono le espressioni così trovate nella parte sinistra dell'equaziome iniziale e si ottiene la seguente espressione:

$$\begin{aligned}
&-\frac{1}{4}\left[f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5)\right] + \\
&-\frac{5}{6}\left[f(x)\right] + \\
&+\frac{3}{2}\left[f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5)\right] + \\
&-\frac{1}{2}\left[f(x) + 2hf'(x) + \frac{4h^2}{2}f''(x) + \frac{8h^3}{6}f^{(3)}(x) + \frac{16h^4}{24}f^{(4)}(x) + O(h^5)\right] + \\
&+\frac{1}{12}\left[f(x) + 3hf'(x) + \frac{9h^2}{2}f''(x) + \frac{27h^3}{6}f^{(3)}(x) + \frac{81h^4}{24}f^{(4)}(x) + O(h^5)\right]
\end{aligned}$$

Si procede a moltiplicare i coefficienti di ogni espressione e poi raccogliere i termini che contengono le derivate dello stesso ordine, una volta raccolti i temrini assumono i seguenti valori che, stando all'equazione iniziale dovranno poi essere sommati:

$$f(x)\left[-\frac{1}{4} - \frac{5}{6} + \frac{3}{2} - \frac{1}{2} + \frac{1}{12}\right] = 0 \tag{1}$$
$$f'(x)\cdot h\left[\frac{1}{4} + \frac{3}{2} - \frac{1}{2}2 + \frac{1}{12}3\right] = hf'(x) \tag{2}$$
$$f''(x)\cdot \frac{h^2}{2}\left[-\frac{1}{4} + \frac{3}{2} - \frac{1}{2}4 + \frac{1}{12}9\right] = 0 \tag{3}$$
$$f^{(3)}(x)\cdot \frac{h^3}{6}\left[-\frac{1}{4} + \frac{3}{2} - \frac{1}{2}8 + \frac{1}{12}27\right] = 0 \tag{4}$$
$$f^{(4)}(x)\cdot \frac{h^4}{24}\left[-\frac{1}{4} + \frac{3}{2} - \frac{1}{2}16 + \frac{1}{12}81\right] = 0 \tag{5}$$

Dalle espressioni (1)...(5) e dalle proprietà degli "O-grande" di moltiplicazione per una costante segue l'asserto.

Esercizio 2

La doppia precisione dello standard IEEE 754 è una rappresentazione in base binaria, in forma normalizzata $(1.f)$ che approssima per arrotondamento e occupa 64 bit, di cui 52 dedicati alla frazione (53 alla mantissa).
Si può dunque ottenere il valore della precisione di macchina (u) dalla seguente espressione, dove: $b = 2$ rappresenta la base, e $m = 53$ la mantissa:

$$u = \frac{1}{2} b^{1-m} = 2^{-53}$$

Invece `eps` è definito dalla stessa funzione `help` di Matlab come la distanza tra `1.0` e il maggior valore a doppia precisione successivo disponibile, ovvero 2^{-52}.
Si osserva infatti che, considerato il valore $x = 1 + u = 1 + 2^{-53} \neq 1$ e sia fl la funzione di floating, allora vale che $fl(x) = 1$, poichè $u = 2^{-53} < 2^{-52} =$`eps`.
Vi è dunque un errore di rappresentazione del valore x (ε_x), determinato dalla seguente espressione:

$$\varepsilon_x = \frac{|x - fl(x)|}{|x|} = \frac{|1 + 2^{-53} - 1|}{|1 + 2^{-53}|} = \frac{|2^{-53}|}{|1 + 2^{-53}|} < |2^{-53}| = u$$

Esercizio 3

La cancellazione numerica è quel fenomeno in cui, sommando in aritmetica finita due numeri quasi opposti si verifica la perdita di cifre signficative. Questo è dovuto all'espressione del numero di condizionamento della somma in aritmetica finita (k) che per due valori x e y è dato da:

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

Infatti, se $x \to -y$ allora $k \to \infty$ e la somma tra x e y risulta mal condizionata.

Esercizio 4

Sia $x^* \in \mathbb{R}$ il valore di cui si ricerca la radice sesta.
Per calcolarlo si definisce una funzione $f(x)$ come segue:

$$f(x) = x^6 - x^*$$

La cui derivata è:

$$f'(x) = 6x^5$$

La funzione $f(x)$ si annulla solo nella radice sesta di x^*, quindi avendo un'approsimazione iniziale x_0 si può applicare il metodo di Newton alla funzione $f(x)$ per ricercarne una radice che coinciderà con il valore cercato:

$$x_{i+1} = x_i + \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^6 - x^*}{6x_i^5} = \frac{1}{6}\left[5x_i + \frac{x^*}{x_i}\right]$$

La `function` che implementa il metodo presentato è contenuta nel file `radice.m` nella cartella `4`:

```
function root = radice(x)
%
%   root = radice(x)
%
%   Questa funzione calcola la radice sesta di un valore non
    negativo
%   attraverso il metodo iterativo di Newton utilizzando solo
    operazioni elementari
%
%   Input:
%       x: valore di cui si vuole calcolare la radice sesta
%   Output:
%       root; risultato del calcolo
if(x<0), error("Value x must be not negative");end
if(x==0)
    root = 0;
    return;
end

root = x;
er = 1;

while(er >= eps*(1+abs(x)))
    xi = (5*root+x/root^5)/6;
    er = abs(root - xi);
    root = xi;
end
return;
end
```

I dati sul confronto tra il risultato offerto dalla funzione e il valore `x^(1/6)` sono contentuti nel file `4_table.txt`:

x	radice(x)	x^(1/6)	errore
1e-10	0.021544	0.021544	3.4694e-18
1.1288e-09	0.032268	0.032268	6.9389e-18
1.2743e-08	0.048329	0.048329	6.9389e-18
1.4384e-07	0.072385	0.072385	1.3878e-17
1.6238e-06	0.10841	0.10841	4.1633e-17
1.833e-05	0.16238	0.16238	0
0.00020691	0.2432	0.2432	2.7756e-17
0.0023357	0.36425	0.36425	5.5511e-17
0.026367	0.54556	0.54556	0
0.29764	0.81711	0.81711	1.1102e-16
3.3598	1.2238	1.2238	2.2204e-16
37.927	1.833	1.833	0
428.13	2.7453	2.7453	4.4409e-16
4832.9	4.1118	4.1118	8.8818e-16
54556	6.1585	6.1585	0
6.1585e+05	9.2239	9.2239	1.7764e-15
6.9519e+06	13.815	13.815	1.7764e-15
7.8476e+07	20.691	20.691	3.5527e-15
8.8587e+08	30.99	30.99	3.5527e-15
1e+10	46.416	46.416	7.1054e-15

Esercizio 5

Il seguente codice è contenuto nel file `newtonMethod.m` nella cartella `5` e rappresenta il metodo di Newton:

```
function [x,n] = newtonMethod(f,df, x0, tol)
%
%   x = newtonMethod(f,df,x0,tol, itmax)
%
%   Ricerca la radice di una funzione di cui è nota la derivata a
    partire
%   da un approssimazione iniziale mediante il metodo di Newton
%
%   Input:
%       f: funzione di cui si ricercano le radici
%       df: derivata della funzione f
%       x0: approssimazione iniziale della radice
%       tol: errore assoluto ammissibile
%   Output:
%       x: approssimazione della radice di f
%       n: numero di iterazioni eseguite

%controllo valori input
if nargin ~= 4, error("Missing arguments"); end
if tol<0, error("Invalid arguments: tolerance must be non negative
    "); end
x = x0;
fx = feval(f,x);
dfx =  feval(f,x);

x = x0- fx/dfx;

n = 1;
while abs(x-x0) > tol*( 1 + abs(x0))

    x0 = x;

    fx = feval(f,x0);
    dfx = feval(df, x0);

    if dfx==0
        error("Value of derivative function is 0, invalid first
            approximation");
    end
    n = n+1;
    x = x0 - fx/dfx; %calcolo effettivo
end
return
end
```

Il seguente codice è contenuto nel file `secantsMethod.m` nella cartella `5` e rappresenta il metodo delle secanti:

```
function [x,i] = secantsMethod(f, x0, x1, tol)
%
%   x = secantsMethod(f,df,x0,tol, itmax)
%
%   Ricerca la radice di una funzione di cui è nota la derivata a
    partire
%   da un approssimazione iniziale mediante il metodo delle secanti
%
%   Input:
%       f: funzione di cui si ricercano gli 0
%       x0: prima approssimazione iniziale della radice
%       x1: seconda approssimazione iniziale della radice
%       tol: errore assoluto ammissibile
%   Output:
%       x: approssimazione della radice di f
%       i: numero di iterazioni eseguite

%controllo valori input
if nargin ~= 4, error("Missing arguments"); end
if tol<0, error("Invalid arguments: tolerance must be non negative
    "); end

fx0 = feval(f,x0);
fx1= feval(f,x1);

x = x1-(fx1*(x1-x0))/(fx1-fx0);

i = 1;
while abs(x-x1) > tol*( 1 + abs(x1))
    i = i+1;

    x0 = x1;
    x1 = x;
    fx0 = fx1;
    fx1 = feval(f,x1);
    if fx0 == fx1, error("Invalid initial approximations: "+ ...
            "function assume same value in different points");end
    x = x1-(fx1*(x1-x0)/(fx1-fx0));
end
return
end
```

Esercizio 6

Nel file `6_result.txt` è contenuta la tabella dei risultati delle funzioni precedentemente mostrate:

```
Tolleranza    Ris. Newton    Iterazoni Newton    Ris. secanti    Iterazioni secanti
    ----------    -----------    ----------------    ------------    ------------------

      0.001         0.73909              8              0.7391              4
      1e-06         0.73909              9             0.73909              6
      1e-09         0.73909             10             0.73909              7
      1e-12         0.73909             10             0.73909              7
```

Per entrambi i metodi, la parte più costosa computazionalmente è la valutazione funzionale, dato che tutte le altre operazioni che vengono svolte sono operazioni elementari.
Il metodo di Newton esegue due valutazioni in ogni iterazione.
Sia n il numero di iterazioni, il costo computazionale del metodo di Newton è dato da $2(n+1)$.
Il metodo delle secanti esegue due valutazioni iniziali e poi una per ogni iterazione, quindi il suo costo computazionale per n iterazioni è dato da $n+2$.

Tolleranza	Iterazioni Newton	Costo Newton	Iterazioni secanti	Costo secanti
10^{-3}	8	16	4	6
10^{-6}	9	18	6	8
10^{-9}	10	20	6	8
10^{-12}	10	20	7	9

Esercizio 7

La seguente tabella fornisce i risultati dell'utilizzo delle funzioni precedenti per calcolare la radice della funzione $f(x) = [x - cos(x)]^5$:

tolleranza	Newton ris.	Newton iter.	Secant ris.	Secant iter.
10e-3	0.74512	18	0.73015	26
10e-6	0.73909	49	0.73908	70
10e-9	0.73909	80	0.73909	115
10e-12	0.73909	111	0.73909	159

Dopo aver sviluppato la *function* `modifiedNewtonMethod.m` si sono riscontrati i seguenti risultati:

tolleranza	risultato Newton modificato	numero di iterazioni
1e-3	0.73909	22
1e-6	0.73909	23
1e-9	0.73909	24
1e-12	0.73909	24

Come atteso, i metodi di Newton e delle secanti sono più lenti a causa del metodo di Newton modificato, a causa della natura multipla della radice.
Infatti il metodo di Newton e quello delle secanti hanno convergenza quadratica nel caso di radici a molteplicità 1, ma solo lineare nel caso di radici multiple.
La modifica che abbiamo fatto, ovvero $x_{i+1} = x_i - m \cdot \frac{f(x_i)}{f'(x_i)}$, nonostante richieda che la molteplcità m della radice sia nota, ripristina la convergenza quadratica del metodo di Newton.
I rislutati sono contentuti nel file `table_7.txt` e si mostra di seguito il codice della funztion del metodo di Newton modificato:

```
function [x,n] = modifiedNewtonMethod(f,df, m, x0, tol)
%
%   x = newtonMethod(f,df,x0,tol, itmax)
%
%   Ricerca la radice di una funzione di cui è nota la derivata a
    partire
%   da un approssimazione iniziale mediante il metodo di Newton
%
%   Input:
%       f: funzione di cui si ricercano le radici
%       df: derivata della funzione f
%       m: molteplicità (nota) della radice
%       x0: approssimazione iniziale della radice
%       tol: errore assoluto ammissibile
%   Output:
%       x: approssimazione della radice di f
%       n: numero di iterazioni eseguite

%controllo valori input
if nargin ~= 5, error("Missing arguments"); end
if tol<0, error("Invalid arguments: tolerance must be non negative
    "); end
x = x0;
fx = feval(f,x);
dfx =  feval(f,x);
```

```
x = x0- m*fx/dfx;

n = 1;
while abs(x-x0) > tol*( 1 + abs(x0))

    x0 = x;

    fx = feval(f,x0);
    dfx = feval(df, x0);

    if dfx==0
        error("Value of derivative function is 0, invalid first
            approximation");
    end
    n = n+1;
    x = x0 - m*fx/dfx; %calcolo effettivo
end
return
end
```

Esercizio 8

Il codice della *function* è contenuto nel file `mialu.m` nella cartella `8`:

```
function x = mialu(A,b)
%
% x = mialu(A,b)
%
% presa in input una matrice ed un vettore calcola la soluzione del
% corrispondente sistema lineare utilizzando 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("La matrice non è quadrata");
end
if n ~= length(b)
    error("la lunghezza del vettore dei termini noti " + ...
        "non è coerente con quella della matrice");
end
p = (1:n).';
for i = 1:n
    [mi, ki] = max(abs(A(i:n,i)));
    if mi == 0
        error("la matrice è 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
return;
end
```

Un esempio di utilizzo è contenuto nel file di testo `ex_8_mialu.txt`:

Esercizio 9

Il codice della *function* è contenuto nel file `mialdl.m` nella cartella `9`:

```
function x = mialdl(A,b)
%
% x = mialdl(A,b)
%
% presa in input una matrice ed un vettore calcola la soluzione del
% corrispondente sistema lineare utilizzando il metodo di
    fattorizzazione
% LDL
%
% Input:
% A = matrice dei coefficienti
% b = vettore dei termini noti
%
% Output:
% x = soluzione del sistema lineare
%
[m,n] = size(A);
if m ~= n
    error("la matrice non è quadrata");
end
if n ~= length(b)
    error("la lunghezza del vettore dei termini noti " + ...
        "non è coerente con quella della matrice");
end
if A(1,1) <= 0
    error("la matrice non è sdp");
end
% la matrice non è memorizzata in forma compressa! (cit. libro)
A(2:n,1) = A(2:n,1)/A(1,1);
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("la matrice non è 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 = 2:n
    x(i:n) = x(i:n) - A(i:n,i-1) * x(i-1);
end
x = x ./ diag(A);
for i = n-1:-1:1
    x(1:i) = x(1:i) - A(i+1,1:i) .'* x(i+1);
end
end
```

Un esempio di utilizzo è contenuto nel file di testo `9_mialdl.txt`:

```
% si genera una matice quadrata casuale
A = randi([-8,8],4)

A =

     0     0     2    -3
```

```
    -1     0    -2     7
    -1     5     5     6
    -3     5     1     1

% si generano i valori di una diagonale
d = randi([5,30],4,1)

d =

    21
    20
    10
    12

% si costruisce una matrice adeguata per la fattorizzazione LDL
A = tril(A,-1)+triu(A',1)+diag(d)

A =

    21    -1    -1    -3
    -1    20     5     5
    -1     5    10     1
    -3     5     1    12

% Si genera la soluzione, da confrontare dopo
x = randi([-8,8],4,1)

x =

     0
    -5
     6
    -5

% si calcolano i termini noti
b = A*x

b =

    14
   -95
    30
   -79

% si usa la funzione per calcolare la soluzione
mialdl(A,b)

ans =

    0.0000
```

```
   -5.0000
    6.0000
   -5.0000

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

A =

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

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

d =

    22
    15
    14
    30

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

A =

    22    -4     5    -8
    -4    15     0     1
     5     0    14     1
    -8     1     1    30

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

x =

    -8
     7
     7
     5

b = A*x

b =

  -209
   142
    63
   228

mialdl(A,b)
```

```
ans =

   -8.0000
    7.0000
    7.0000
    5.0000
```

Esercizio 10

La funzione è nel file `functions/miaqr.m` nella cartella `10`, mostrato di seguito insieme ad un esempio in cui viene applicato:

```
function [x,nr] = miaqr(A,b)
%
%   [x, nr] = miaqr(A,b)
%
%   Calcola la soluzione del sistema lineare sovradimensioanto Ax =
     b
%   nel senso dei minimi quadrati e restituisce la norma del
%   corrispondente vettore residuo
%
%   Input:
%       A:  matrice dei coefficienti del sistema
%       b:  vettore dei termini noti
%   Output:
%       x:  soluzione nel senso dei minimi quadrati
%       nr: norma del vettore resiudo
[m,n] = size(A);
if(n>=m), errror("Il sistema non è sovradimensionato"); end
if(m~=length(b)), error("Le dimensioni della matrice e del vettore
    " + ...
        "non sono compatibili");end
for i=1:n
    alfa = norm( A(i:m,i));
    if alfa==0,error("La matrice A non ha rango massimo");end
    if(A(i,i)>=0), alfa = -alfa; end
    v = A(i,i) - alfa;
    A(i,i) = alfa;
    A(i+1:m,i) = A(i+1:m,i)/v;
    beta = -v/alfa;
    A(i:m,i+1:n) = A(i:m,i+1:n)-(beta*[1;A(i+1:m,i)])*...
        ([1;A(i+1:m,i)]'*A(i:m,i+1:n));
end
for i=1:n
    v = [1;A(i+1:m,i)];
    beta = 2/(v'*v);
    b(i:end) = b(i:end)-(beta*(v'*b(i:end)))*v;
end
for i=n:-1:1
    b(i) = b(i)/A(i,i);
    b(1:i-1) = b(1:i-1)-A(1:i-1,i)*b(i);
end
x = b(1:n);
nr = norm(b(n+1:m));
return ;
end
```

```
>> A = randi([-20,20],7,4)

A =

    -6     6   -13    -1
    13    -2    -5    -3
   -20     2     5    -2
   -19    -8    11    -8
```

```
   -14    10   -17     0
     6   -13    18     0
    10     8    11    13

>> b = randi([-20,20],7,1)

b =

    12
     6
    -5
    13
     1
    -6
    18

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

x =

    0.5023
    2.5325
    1.1667
   -2.1687


nr =

   22.8572

>> A\b

ans =

    0.5023
    2.5325
    1.1667
   -2.1687

>> A = randi([-20,20],7,4)

A =

    15   -11    -8   -16
     2    14    17   -10
     5   -13    -3    -4
     4   -11   -13     4
   -12   -14    17   -10
    -8   -11    20     4
    -1    -3    -3     9
```

```
>> b = randi([-20,20],7,1)

b =

   -11
   -16
    -8
    -7
    -3
     0
   -17

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

x =

   -1.2544
    0.2774
   -0.6423
   -0.2978


nr =

   22.2472

>> A\b

ans =

   -1.2544
    0.2774
   -0.6423
   -0.2978

>>
```

Esercizio 11

Di seguito un esempio di applicazione di `mialu` per risolvere i sistemi generati da `linsis`:

```
[A1,A2,b1,b2]=linsis(5)

A1 =

    0.0659   -0.4423    0.2073   -0.5127    0.3531
    0.7016   -0.2493   -0.1158    0.1664   -0.0385
   -0.1391   -0.4272    0.4168    0.2575   -0.0030
    0.2598   -0.4140   -0.0020   -0.2632   -0.6674
    0.0654   -0.2921   -0.6037   -0.1323    0.5153


A2 =

   -0.2172   -0.0838    0.2868   -0.3463    0.3624
    0.3869    0.1493   -0.0275    0.3514   -0.0281
   -0.1833   -0.3713    0.4292    0.2834   -0.0016
   -0.0576   -0.0121    0.0871   -0.0767   -0.6569
   -0.1544   -0.0139   -0.5420   -0.0032    0.5225


b1 =

   -0.3287
    0.4645
    0.1050
   -1.0869
   -0.4475


b2 =

    0.0019
    0.8320
    0.1565
   -0.7163
   -0.1910

mialu(A1,b1)

ans =

    1.0000
    1.0000
    1.0000
    1.0000
```

```
    1.0000

mialu(A2,b2)

ans =

    1.0000
    1.0000
    1.0000
    1.0000
    1.0000
```

Il risltato sembra essere corretto, ma se si sottrae le soluzioni ad un vettore composto di soli 1, si può osservare l'errore nella risoluzione.
Nel sistema $A_1x = b_1$ l'errore è nell'ordine di 10^{-15} mentre nel secondo sistema $A_2x = b_2$ l'ordine di errore è di 10^{-6}.
L'errore molto maggiore nel secondo sistema è dovuto al mal condizionamento della matrice dei coefficienti.

```
mialu(A1,b1)-[1 1 1 1 1]'

ans =

   1.0e-15 *

         0
   -0.1110
         0
         0
    0.2220

mialu(A2,b2)-[1 1 1 1 1]'

ans =

   1.0e-06 *

    0.3523
   -0.4462
   -0.0989
   -0.2071
   -0.0116

>> cond(A1)

ans =

    2.5000

>> cond(A2)
```

```
ans =

   1.0000e+10

>> cond(b1)

ans =

     1

>> cond(b2)

ans =

     1

>>
```

Esercizio 12

Similemente a quanto si è ottenuto per l'esercizio precedente, si può osservare come i risultati ottenuti dalla funzione `mialdl` siano accurati con un oridne di grandezza dell'errore di 10^{-15} per il primo sistema e 10^{-6} per il secondo. Anche in questo caso la differenza è dovuta alla differenza del condizionamento delle due matrici A_1 e A_2.

```
[A1,A2,b1,b2] = linsis(5,1)

A1 =

    0.7625    0.0003    0.1094    0.0730    0.1782
    0.0003    0.7442    0.0084    0.1772    0.1085
    0.1094    0.0084    0.6419    0.0332   -0.1291
    0.0730    0.1772    0.0332    0.8375   -0.1198
    0.1782    0.1085   -0.1291   -0.1198    0.8139


A2 =

    0.5197   -0.2696    0.0716   -0.1992   -0.0102
   -0.2696    0.4441   -0.0337   -0.1254   -0.1010
    0.0716   -0.0337    0.6360   -0.0092   -0.1585
   -0.1992   -0.1254   -0.0092    0.5324   -0.3310
   -0.0102   -0.1010   -0.1585   -0.3310    0.6677


b1 =

    1.1234
    1.0385
    0.6638
    1.0011
    0.8517


b2 =

    0.1124
   -0.0855
    0.5063
   -0.1323
    0.0670

>> x1 = mialdl(A1,b1)

x1 =

    1.0000
```

```
    1.0000
    1.0000
    1.0000
    1.0000

>> x2 = mialdl(A2,b2)

x2 =

    1.0000
    1.0000
    1.0000
    1.0000
    1.0000

>> x1 - [1 1 1 1 1]'

ans =

   1.0e-15 *

         0
   -0.4441
   -0.1110
   -0.1110
         0

>> x2 - [1 1 1 1 1]'

ans =

   1.0e-06 *

    0.3138
    0.3488
    0.0489
    0.3518
    0.2435

>>
```

Esercizio 13

Di seguito si mostra come sono stati assegnati i valori richiesti, le soluzioni trovate con la funzione `miaqr` sono confrontate con il risultato dell'operatore \.

```
>> 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])

D1 =

    3.1416         0         0         0         0
         0    3.1416         0         0         0
         0         0    3.1416         0         0
         0         0         0    3.1416         0
         0         0         0         0    3.1416

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

x =

    3.0000
    5.8000
   -2.5000


nr =

    1.2649

>> A\b

ans =

    3.0000
    5.8000
   -2.5000

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

x =

   -0.6026
    4.7017
    1.7584


nr =
```

```
    3.7352

>> (D*A)\(D*b)

ans =

   -0.6026
    4.7017
    1.7584

>> [x,nr] = miaqr(D1*A,D1*b)

x =

    3.0000
    5.8000
   -2.5000


nr =

    3.9738

>> (D1*A)\(D1*b)

ans =

    3.0000
    5.8000
   -2.5000
```

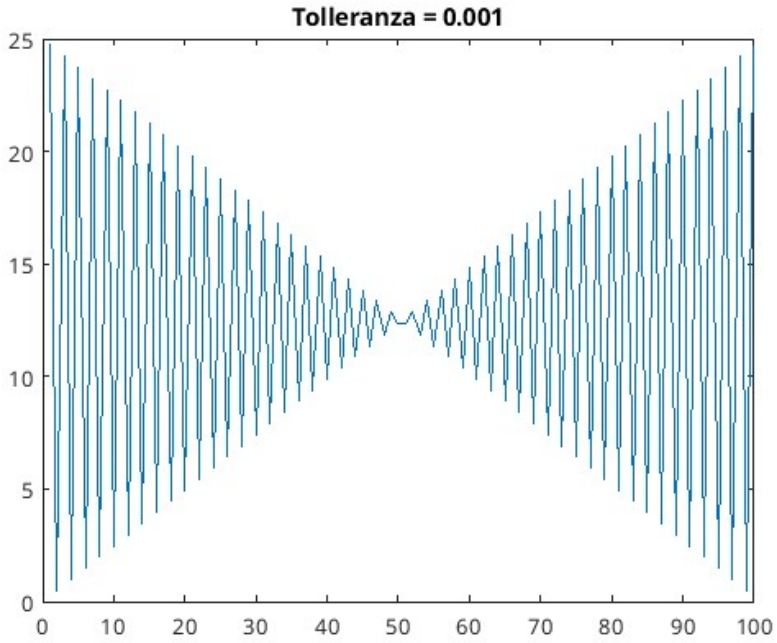
Si osserva come le soluzioni siano coerenti, ma la norma del vettore residuo aumenta, negli ultimi due sistemi è quasi il triplo che nel primo.

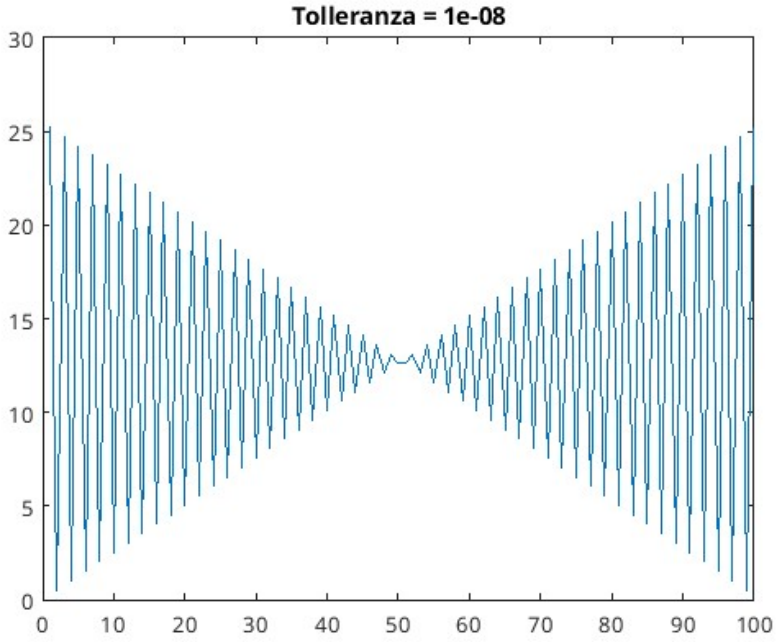
Esercizio 14

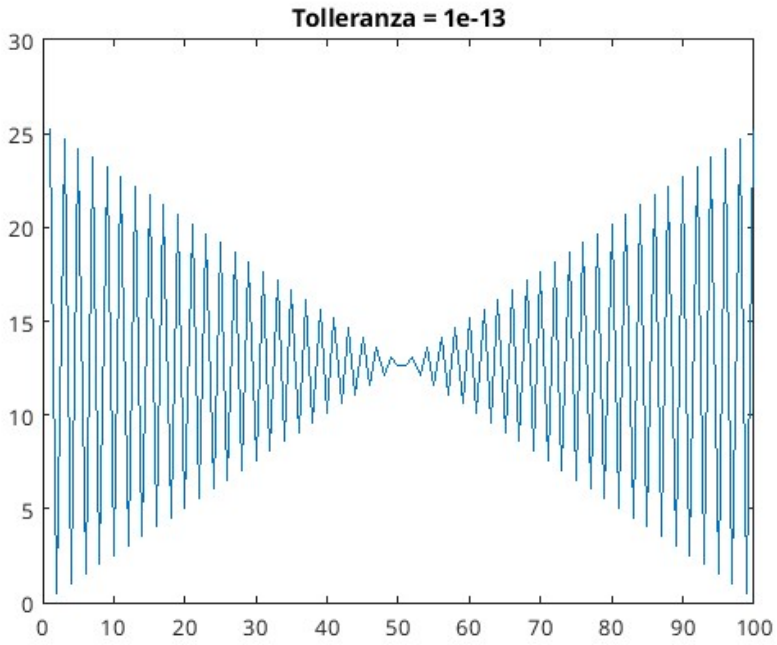
La funzione è contenuta nel file `newton.m` nella cartella `14`.

```
function [x,nit] = newton(fun,jacobian,x0,tol,maxit)
%
%   [x,nit] = newton(fun,jacobian,x0,tol,maxit)
%
%   Utilizza il metodo di Newton per risolvere sistemi di equazioni
%   nonlineari
%
%   Input:
%       fun: vettore delle funzioni del sistema
%       jacobian: matrice jacobiana di fun
%       x0: approssimazione iniziale
%       tol: tolleranza aspettata
%       maxit: numero massimo di iterazioni ammesso
%   Output:
%       x: approssimazione della funzione
%       nit: numero delle iterazioni del metodo

if(nargin<4), tol = eps;    end
if(nargin<5), maxit = 1e3;  end
if(tol<=0),   error("La tolleranza deve essere positiva");
              end
if(maxit<=0), error("Il numero di iterazioni deve essere positivo")
    ;end
x = x0;
nit = 0;
while(nit<maxit&&norm(x-x0)<=tol*(1+norm(x0)))
    x0 = x;
    fx0 = feval(fun,x0);
    jx0 = feval(jacobian,x0);
    x = x0 - fx0/jx0;
end
if(nit == maxit)
    disp("Il numero di iterazioni specificato non ha permesso " +
        ...
        "di raggiungere la tolleranza desiderata");
end
return;
end
```

Esercizio 15






Tolleranza = 1e-13
30
25
20
15
10
5
0
0
10
20
30
40
50
60
70
80
90
100

Esercizio 16

Riportiamo in seguito la funzione `lagrange.m` nella cartella `16` che implementa il metodo di interpolazione polinomiale con il metodo di Lagrange.

```
function l = lagrange(x,y,xq)
% l = lagrange(x,y,xq)
%
%   Implementa in modo vettoriale la forma di lagrange del
    polinomio
%   interpolante di una funzione
%
%   Input:
%       x:  ascisse di interpolazione
%       y:  valori della funzione sulle ascisse di interpolazione
%       xq: punti in cui calcolare il polinomio
%   Output:
%       l:  polinomio di lagrange calcolato

n = length(x);
if n ~= length(y)
    error("il numero di punti sulle ascisse x non è coerente con" +
         ...
        " il numero di quelli sulle ordinate");
end
if n ~= length(unique(x))
    error("ad una stessa ascissa non possono corrispondere più
        punti")
end
l = zeros(size(xq));
for k = 1:n
    Lkn = ones(size(xq));
    for j = 1:n
        if k ~= j
            Lkn = Lkn .* ((xq - x(j))/(x(k) - x(j)));
        end
    end
    l = l + y(k)*Lkn;
end
return;
end
```

Esercizio 17

Riportiamo in seguito la funzione `newton.m` nella cartella `17` che implementa il metodo di interpolazione polinomiale con il metodo di Newton.

```
function l = newton(x,y,xq)
% l = newton(x,y,xq)
%
%   Implementa in modo vettoriale, la forma di Newton del polinomio
%   interpolante una funzione
%
%   Input:
%       x:  vettore contenente le ascisse di interpolazione
%       y:  valori assunti dalla funzione sulle ascisse di
%           interpolazione
%       xq: punti su cui si vuole calcolare la funzione
%   Output:
%       l:  approsimazione dei valori della funzione secondo
%           il polinomio interpolante

n = length(x);
if n ~= length(y)
    error("il numero di punti sulle ascisse x non è coerente con il
         numero" + ...
        " di quelli sulle ordinate");
end
if n ~= length(unique(x))
    error("ad una stessa ascissa non possono corrispondere più
        punti")
end
f = y; %differenze divise
for k = 1:n-1
    for r = n:-1:k+1
        f(r) = (f(r) - f(r-1)) ./ (x(r) - x(r-k));
    end
end
l = ones(size(xq)) * f(n);
for k = n-1:-1:1
    l = l .* (xq - x(k)) + f(k);
end
end
```

Esercizio 18

Riportiamo in seguito la funzione `hermite.m` nella cartella `18` che implementa il metodo di interpolazione polinomiale con il metodo di Hermite.

```
function yy = hermite( xi, fi, f1i, xx )
%   Implementa in modo vettoriale il polinomio interpolante di
    Hermite
%
%   Input:
%       xi:     vettore delle ascisse di interpolazione
%       fi:     valori assunti dalla funzione sulle ascisse
%               di interpolazione
%       f1i:    valori assunti dalla derivata della funzione sulle
    ascisse
%               di interpolazione
%       xx:     vettore di ascisse su cui si vuole calcolare il
    polinomio
%   Output:
%       yy:     valori assunti dal polinomio sui punti specificati

n = length(xi);
if n ~= length(fi)
    error("il numero di punti sulle ascisse xi " + ...
        "non è coerente con il numero di quelli sulle ordinate fi")
            ;
end
if n ~= length(f1i)
    error("il numero di punti sulle ascisse xi non è coerente" +
        ...
        " con il numero di quelli sulle ordinate fi"); % ---
end
if n ~= length(unique(xi))
    error("le ascisse di interpolazione non sono tutte distinte");
end
x = repelem(xi,2);

%differenze divise
f = zeros(2 * n, 1);
f(1:2:end) = fi;
f(2:2:end) = f1i;
% algortimo 4.2 libro
n = length(f)/2-1;
for i = (2*n-1):-2:3
    f(i) = (f(i)-f(i-2))/(x(i)-x(i-1));
end
for j = 2:2*n-1
    for i = (2*n+2):-1:j+1
        f(i) = (f(i)-f(i-1))/(x(i)- x(i-j));
    end
end
%algoritmo di horner
n = length(f)-1;
yy = f(n+1)*ones(size(xx));
for i = n:-1:1
    yy = yy .* (xx-x(i))+f(i);
end
end
```

Esercizio 19

Riportiamo in seguito la funzione `chebyshev.m` nella cartella `19` che genera n ascisse di Chebyshev su un intervallo a,b.

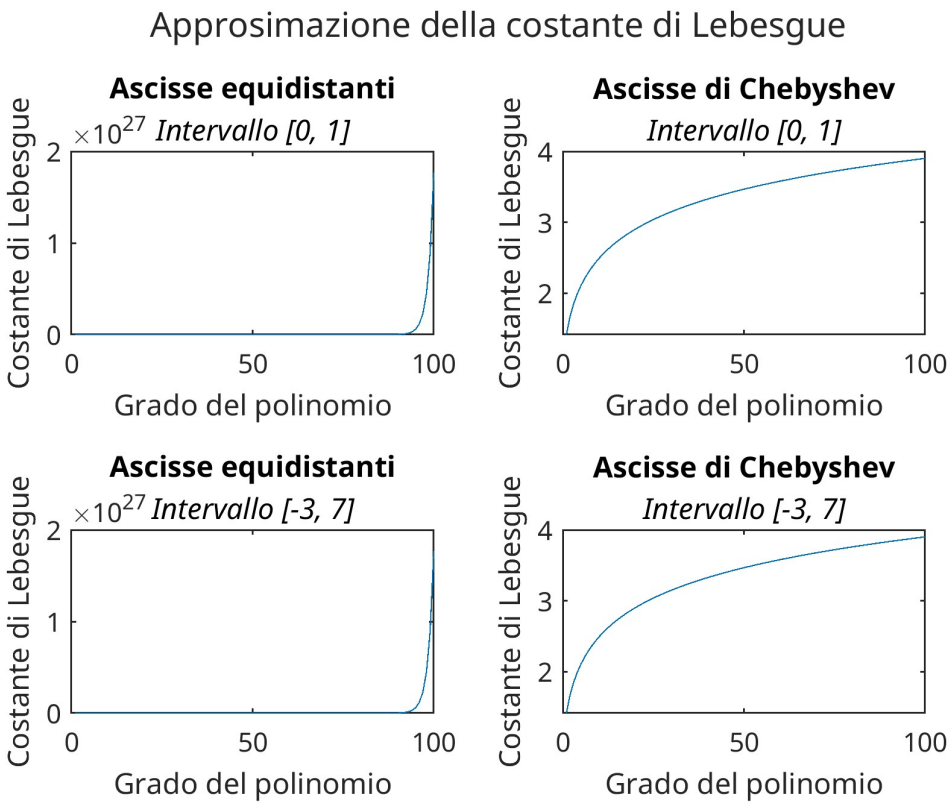
```
function x = chebyshev(n,a,b)
%
%   Genera n+1 coordinate di Chebyschev nell'intervallo [a,b]
%
%   Input:
%       n:  numero di coordinate da generare (n+1)
%       a:  estremo inferiore dell'intervallo
%       b:  estremo superiore dell'intervallpo
%   Output:
%       x:  vettore contenente le 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 può 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 20

Il codice della *function* è contenuto nel file `lebesgue.m` nella cartella `20`:

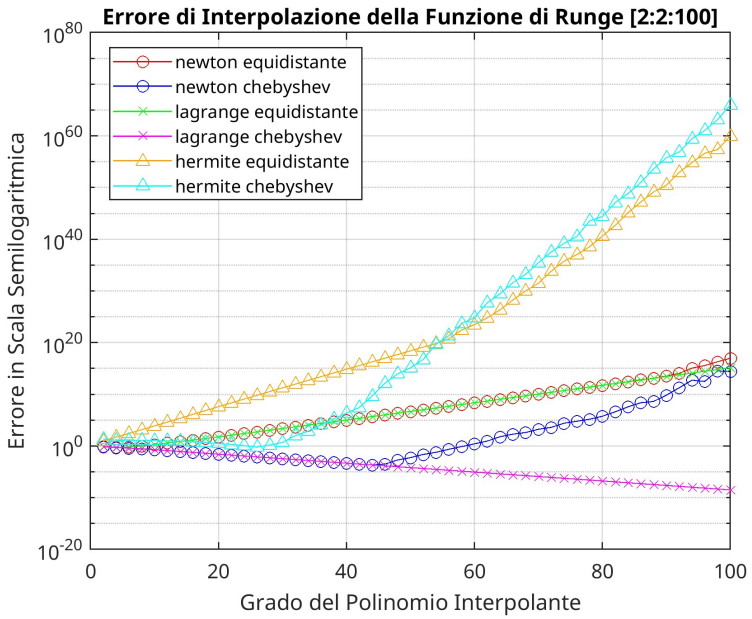
```
function ll = lebesgue(a, b, nn, type)
% ll = lebesgue(a, b, nn, type)
%
%   Approssima la costante di Lebesgue per l'interpolazione
%   polinomiale sull intervallo [a,b], per i polinomi di
%   grado specificato nel vettore nn utilizzando le ascisse
%   equidistanti se type è uguale a 0 e quelle di chebyshev
%   se type è 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 può 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 può 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
```

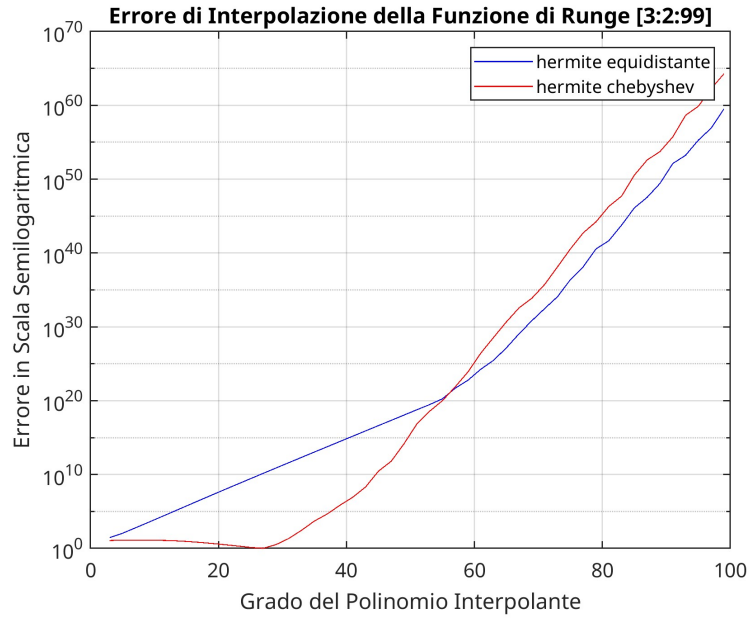
Si riportano inoltre i grafici dei risultati ottenuti per l'approssimazione della costante di Lebesgue su due intervalli distinti, si nota che il risultato è unicamente dipendente dalla scelta delle ascisse che operiamo e non dagli intervalli considerati.


Approsimazione della costante di Lebesgue
Ascisse equidistanti
Intervallo [0, 1]
×10^27
Costante di Lebesgue
Grado del polinomio
Ascisse di Chebyshev
Intervallo [0, 1]
Costante di Lebesgue
Grado del polinomio
Ascisse equidistanti
Intervallo [-3, 7]
×10^27
Costante di Lebesgue
Grado del polinomio
Ascisse di Chebyshev
Intervallo [-3, 7]
Costante di Lebesgue
Grado del polinomio

Esercizio 21

Grafici in scala semilogaritmica degli errori di interpolazione commessi dai vari algoritmi di interpolazione.





Esercizio 22

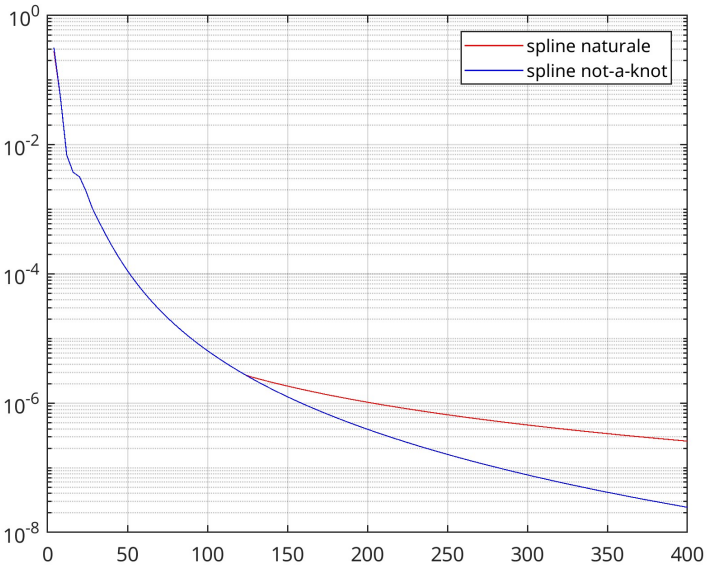
Si riporta il codice della funzione `myspline.m` contenuto nella cartella `22` che calcola la spline cubica o knot-a-knot:

```
function yy = myspline(xi, fi, xx, type )
%
%   Se type è uguale a 0 allora calcola la spline cubica
    interpolante
%   naturale i punti (xi(i),fi(i)), mentre se type è diverso da 0
    allora
%   calcola quella not-a-knot (default)
%
%   Input:
%       xi:     vettore delle ascisse di interpolazione
%       fi:     valori assunti dalla funzione nei rispettivi punti
    xi
%       xx:     ascisse su cui deve essere calcolata la spline
%       type:   valore che stabilisce il tipo di spline da creare
%   Output:
%       yy:     valori assunti dalla spline nei rispettivi punti xx

if nargin < 3, error("argomenti essenziali assenti"); end
if nargin == 3, type = 1; end
if size(xi) ~= size(fi), error("Le quantità di dati forniti per " +
     ...
        "l'interpolazione non corrispondono"); end
if length(xi) ~= length(unique(xi)), error("Le ascisse di " + ...
        "interpolazione devono essere tutte distinte tra loro");end

n = length(xi)-1;

h = zeros(1,n);
for i=1:n, h(i)=xi(i+1)-xi(i);end
phi = zeros(1,n);
xhi = zeros(1,n);
for i=1:n-1
    phi(i) = h(i)/(h(i)+h(i+1));
    xhi(i) = h(i+1)/(h(i)+h(i+1));
end

f = fi ; %differenze divise
for j = 1 : 2
    for i = n+1 : -1 : j +1
        f(i) = (f(i)-f(i-1))/(xi(i)-xi(i-j));
    end
end
f = f (3: n+1) ; % Calcolo le differenze divise

%definizione diagonali del sistema tridiagonale per trovare m0...mn
if type==0 %spline naturale
    a = 2*ones(n-1,1);  %size n-1
    b = xhi(1:n-2);     % n-2
    c = phi(2:n-1);      % n-2
    d = 6*f;
else %spline not-a-knot
    a = [1 2-phi(1) 2*ones(1,n-3) 2-xhi(n-1) 1];%size 2+n-3+2=n+1
    b = [0 xhi(i)-phi(i) xhi(2:n-1)];           % 2+n-2=n
    c = [phi(1:n-2) phi(n-1)-xhi(n-1) 0];       % n-2+2 = n
    d = 6*[f(1) f f(end)];
end
```

```
%risoluzione sistema tridiagonale
dim = length(d);
m = zeros(dim,1);
for i = 2:dim
    w = c(i-1)/a(i-1);
    a(i) = a(i)-w*b(i-1);
    d(i) = d(i)-w*d(i-1);
end

m(end) = d(end)/a(end);
for i = (dim-1):(-1):1
    m(i) = (d(i)-b(i)*m(i+1))/a(i);
end

if type == 0 %spline cubica
    m = [0 m' 0];
else % spline not-a-knot
    m(1) = m(1)-m(2)-m(3);
    m(n+1) = m(n+1)-m(n)-m(n-1);
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))
            r = fi(i-1)-h(i-1)^2/6*m(i-1) ;
            q = (fi(i)-fi(i-1))/h(i-1)-h(i-1)/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*h(i-1))+...
                q*(xx(j)-xi(i -1))+...
                r;
            break
        end
    end
end
return;
end
```

Esercizio 23

Si riporta il plot dell'errore di approssimazione ottenuto utilizzando le spline interpolanti naturale e not-a-knot per approssimare la funzione di Runge sull'intervallo [-5,5], con ascisse equidistanti e grado del polinomio 4:4:400.



Esercizio 24

Si hanno 1000 coppie di dati (x_i, y_i) che rappresentano un fenomeno fisico descritto da una potenza $y = x^n$. Si sa tuttavia che le coppie sono condizionate da un errore la cui distribuzione segue una gaussiana con media 0 e varianza "piccola". Si vogliono determinare i coefficienti $a_1, \ldots, a_m$ ignoti di un polinomio $p(x)$ di grado m che approssima i dati affetti da errore.

$$p(x) = \sum_{k=0}^{m} a_k x^k$$

$$p(x_i) = y_i, i = 1, \ldots, 1000$$

Il vettore dei valori attesi è dato dal prodotto matrice-vettore $V \cdot a$:

$$V = \begin{pmatrix} x_0^0 & x_0^1 & \ldots & x_0^m \\ x_1^0 & x_1^1 & \ldots & x_1^m \\ \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & \ldots & x_m^m \end{pmatrix}, a = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{pmatrix},$$

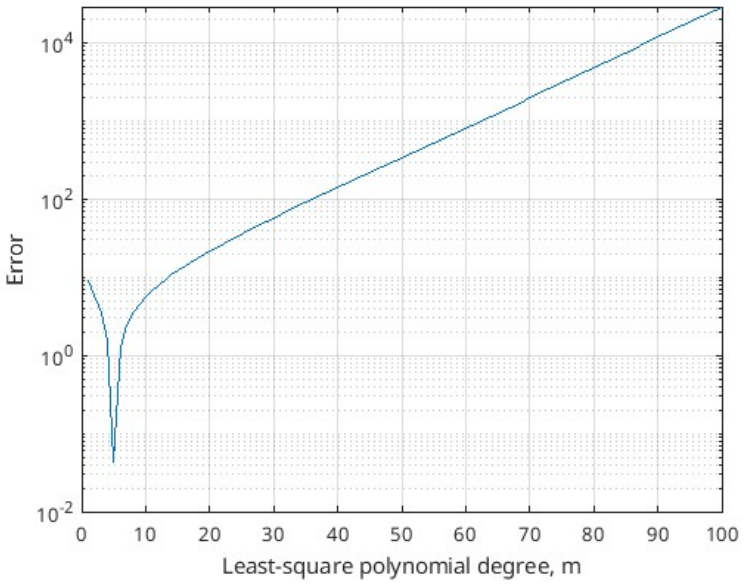
La matrice V ha rango massimo ed è fattorizzabile QR e quindi il sistema ha soluzione.

```
xi_yi = load("data.mat").data;

m_max = 100 ;

xi = xi_yi(:,1);
yi = xi_yi(:,2);
% m<=n
er = zeros(1,m_max);

for m = 1:m_max

    px = polyfit(xi,yi,m);

    % hormer algorithm with reverse loop
    y = px(1);
    for i = 2:length(px)
        y = y .* xi + px(i);
    end

    er(m) = norm(xi.^m - y);
end

% view
figure;
semilogy(1:m_max, er, "-");
xlabel("Least-square polynomial degree, m");
ylabel("Error");
grid on;
```

Nota: Durante il calcolo si usa una versione diversa dell'*algoritmo di Horner* per il calcolo di un polinomio. Il ciclo dell'algoritmo avviene in modo ascendente invece che discendente perchè la funzione `polyfit` restituisce i coefficienti del

polinomio secondo la potenza decrescente.

Questo script importa i dati (x_i, y_i) dal file `data.mat` e li salva in due diversi vettori. Viene scelto un valore (`m_max`) che viene utilizzato come massimo grado ammissibile per il polinomio di approssimazione ai minimi quadrati. All'interno di un ciclo, per ogni m viene utilizzata la funzione `polyfit` per calcolare i coefficenti di $p(x)$ di grado m e viene salvata nel vettore `er` l'errore trovato.
Tutti gli errori vengono mostrati graficati nella figura sottostante, che mostra come il polionomio che meglio approssima i dati (x_i, y_i) è quello di grado $m = 5$.



Esercizio 25

Si riporta il codice della funzione contenuta nel file `newtonCotesWeights.m` nella cartella `25`, la *function* fa utilizzo della funzione `conv` per la moltiplicazione tra polinomi.

```
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 riporta 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 26

Si riporta il codice della funzione contenuta nel file `composita.m` nella cartella `26` che implementa la formula composita di Newton-Cotes di grado k su n+1 ascisse equidistanti.

```
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 è corretto " + ...
        "l'estremo superiore non può 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 27

Si riporta le approssimazioni del seguente integrale e del relativo errore ottenute mediante la funzione composita implementata nel precedente esercizio.

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

k	Stima dell'integrale	Stima dell'errore
1	-0.0925980476169039	0.192379444267568
2	-0.179803240495078	0.00701161675929297
3	-0.178470758042028	0.00280027989942712
6	-0.177446511092635	6.51467298867454e-07

Esercizio 28

Si riporta il codice della funzione `simpsonadaptive.m` contenuto nella cartella 28 che calcola l'approssimazione dell'integrale di una funzione con il metodo di simpson adattivo:

```
function  [If,nval] = simpsonadaptive(fun,a,b,tol,fa,fm,fb)
%
%   [If,nval] = simpsonadaptive(fun,a,b,tol,fa,fm,fb)
%
%   Ricerca l'integrale definito di una funzione mediante la
%   formula adattativa di Simpson
%
%   Input:
%       fun:        fuzione integranda
%       a,b:        estremi dell'intervallo di integrazione
%       tol:        tolleranza dell'errore di quadratura
%       fa,fm,fb:   (opzionali) valutazioni funzionali negli
    estremi
%                   e nel punto mediano dell'intervallo di
    integrazione
%   Output:
%       If:         Stima dell'integrale
%       nval:       Numero di valutazioni funzionali effettuate
if tol<0, error("Tolleranza non positiva");end
if a>b, error("Intervallo non valido");end
xm = (a+b)/2;
if nargin ==4
    fa = feval(fun,a);
    fm = feval(fun,xm);
    fb = feval(fun,b);

    nval = 3;
else
    nval= 0;
end

h = (b-a)/2;

x1 = (a+xm)/2;
x2 = (b+xm)/2;

f1 = feval(fun,x1);
f2 = feval(fun,x2);
nval = nval +2;

I2 = h*(fa + 4*fm + fb)/3;
If = h*(fa + 4*f1 + 2*fm + 4*f2 + fb)/6;

err = abs(If-I2)/15;

if err>tol
    [IfL,navlL] = simpsonadaptive(fun,a,xm,tol/2,fa,f1,fm);
    [IfR,nvalR] = simpsonadaptive(fun,xm,b,tol/2,fm,f2,fb);

    If = IfL + IfR;
    nval = nval + nvalR + navlL;
end

return;
end
```

Esercizio 29

Si riporta il codice della funzione `newtoncotesadaptive.m` contenuto nella cartella `29` che calcola l'approssimazione dell'integrale di una funzione con il metodo di Newton-Cotes di grado 4 adattivo:

```
function [If,nval] = newtoncotesadaptive(fun,a,b,tol,fa,fm,fb,f1,f3
    )
%
%   [If,nval] = newtoncotes4(fun,a,b,tol,fa,fm,fb,f1,f3)
%
% Implementa la formula composita adattativa di Newton-Cotes di
    grado 4
% per calcolare il valore dell'integrale della funzione f nell'
    intervallo
% [a,b]
%   Input:
%       fun:        funzione integranda
%       a:          estremo sinistro dell'intervallo (a=x0)
%       b:          estremo destro dell'intervallo (b=x4)
%       tol:        tolleranza desiderata dell'errore
%       fa,fb,fm:   (opzionali) valori assunti dalla funzione
%                   nelle ascisse x0, x4, x2 (rispettivamente)
%   Output:
%       If:     stima dell'integrale
%       nval:    numero di valutazioni funzionali effettuate
if nargin<4, error("Argomenti essenziali mancanti");end
if tol<=0, error("Tolleranza non positiva");end
if b<a,error("Estremi dell'intervallo non validi");end

x2 = a+(b-a)/2;

if nargin==9

    f = [fa,f1,fm,f3,fb];
    nval = 0;
else
    x1 = (a+x2)/2;
    x3 = (x2+b)/2;

    x = [a x1 x2 x3 b];
    f = zeros(1,5);
    for i =1:5
        f(i) = feval(fun,x(i));
    end

    nval = 5;
end
h = (b-a)/4;

w = [14/45 64/45 8/15 64/45 14/45]';
If = h*(f*w);


he = (b-a)/8;
%sottointervallo sinistro
    x5 = (a+x1)/2;
    x6 = (x1+x2)/2;
    f5 = feval(fun,x5);
    f6 = feval(fun,x6);

    %vettore delle fi dell'intervallo sinistro
```

```
    fL = [f(1),f5,f(2),f6,f(3)];

    IfL = he*(fL*w);

%sottointervallo destro
    x7 = (x2+x3)/2;
    x8 = (x3+b)/2;
    f7 = feval(fun,x7);
    f8 = feval(fun,x8);

    fR = [f(3),f7,f(4),f8,f(5)];

    IfR = he*(fR*w);

% si contano insieme le valutazioni per entrambi gli intervalli
nval = nval+4;

%intervallo totale
If4 = IfL + IfR;

err = abs((If4-If)/63);

if(err<=tol)
    If = If4;
    return;
end

[IfL,nvalL] = newtoncotesadaptive(fun,a,x2,tol/2,f(1:3),f5,f6);
[IfR,nvalR] = newtoncotesadaptive(fun,x2,b,tol/2,f(3:5),f7,f8);

If = IfL+IfR;
nval = nval + nvalL+nvalR;
return;
end
```

Esercizio 30

Si riporta di seguito le tabelle richieste:

Numero di valutazioni funzionali		
Tolleranza	**Simpson**	**Newton-Cotes**
0.01	201	333
0.001	333	423
0.0001	605	567
1e-05	1061	819
1e-06	1869	1233
1e-07	3277	1773
1e-08	5921	2637
1e-09	10589	3933

Errore di Quadratura		
Tolleranza	**Simpson**	**Newton-Cotes**
0.01	0.000293528025694267	2.30517976307354e-06
0.001	0.000457223923412076	1.06536072985719e-06
0.0001	2.63547596148772e-05	9.85260983910052e-10
1e-05	2.34481367633599e-06	3.10416783566581e-06
1e-06	3.00762625249362e-07	3.69732803262579e-08
1e-07	3.65647581102024e-08	2.28150845993369e-08
1e-08	3.21109860923485e-09	2.13358331002667e-09
1e-09	3.09839820467062e-10	1.49663392789989e-10