Elaborato Calcolo numerico 2021-22

Cristian Pernice,
63638474

Esercizio 1. Si considera lo sviluppo delle funzioni $f(x+2h)$, $f(x-2h)$, $f(x+h)$, $f(x-h)$

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

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

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

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

$$\frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} = \frac{-4hf'(x) + 16hf'(x) - \frac{8}{3}h^3f^{(3)}(x) + \frac{8}{3}f^{(3)}(x) + O(h^5)}{12h}$$

$$= f'(x) + O(h^4) \square$$

Esercizio 2. Lo standard IEEE754 a doppia precisione presenta le seguenti caratteristiche:

1- Base binaria
2- Corrisponde a 64 bit di cui 52 riservati alla mantissa
3- La rappresentazione è normalizzata $(1.f)$
4- La rappresentazione è per arrotondamento

Da queste caratteristiche possiamo ottenere i dati necessari per calcolare la precisione di macchina

Da (1) otteniamo $b = 2$, da (2) e (3) $m = 53$, infine da (4) otteniamo la precisione di macchina $u = \frac{1}{2}b^{1-m}$. Sostituendo b e m con i valori ottenuti, si ottiene il numero di precisione di macchina $u = 2^{-53}$.
La variabile eps restituisce la distanza tra 1.0 e il numero successivo rappresentabile con lo standard IEEE754 con precisione doppia; tale valore è uguale a 2^{-52}, infatti è possibile dimostrare la relazione esistente tra eps e la precisione di macchina calcolata in precedenza.
La rappresentazione in codifica IEEE754 del valore $x = 1 + 2^{-53}$, risulta essere $fl(x) = 1$, in quanto $2^{-53} < eps$. L'errore di rappresentazione è, dunque, uguale a

$$\varepsilon_x = \frac{|x| - |fl(x)|}{|x|} = \frac{\left|\left(1.\overbrace{000\ldots000}^{52}1 - 1.\overbrace{000\ldots000}^{52}\right)_2\right|}{\left|\left(1.\overbrace{000\ldots00}^{52}1\right)_2\right|} = \frac{0.\overbrace{000\ldots000}^{52}1_2}{1.\overbrace{000\ldots000}^{52}1_2} \leq 2^{-53} = u \square$$

Esercizio 3. Il risultato che si dovrebbe ottenere è $\left(1 + (10^{-14} - 1)\right) * 10^{14} = 1$, tuttavia il risultato restituito da Matlab corrisponde a 0.99920072216264l. Tale risultato è dovuto alla somma tra 1 e il risultato di $(10^{-14} - 1)$, ovvero la somma di due numeri quasi opposti, la quale risulta essere sempre mal condizionata, infatti $k = \frac{|1| + |10^{-14} - 1|}{|1 + (10^{-14} - 1)|} \approx \frac{2}{10^{-14}} \gg 1$.

Esercizio 4. La function radice implementata risolve, dato in input un numero x_0, l'equazione $x^2 - r = 0$. È possibile osservare che la risoluzione utilizzata è, in realtà, il metodo di Newton per la ricerca degli zeri della funzione $f(x) = x^2 - x_0$, infatti è possibile dimostrare che l'approssimazione x_{n+1} corrisponde a

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - r}{2x_n} = \frac{1}{2}(x_n + \frac{r}{x_n})$$

Per ottenere la massima precisione possibile si utilizza il criterio di arresto $\frac{|x_{n+1}-x_n|}{|x_{n+1}|} \leq 2^{-53}$, il quale corrisponde al criterio $\varepsilon_x \leq u$, e, successivamente, si ottiene $\bar{x}$ dividendo r per l'approssimazione ottenuta in modo da eliminare errori di approssimazione dell'ordine di grandezza molto piccoli.

Metodo di Newton per le radici quadrate

```
function x = radice(x)
%
%   x = radice(x)
%   Function che restituisce la radice quadrata di x utilizzando solo
%       operazioni algebriche elementari
%
%   Input:
%       x: numero non negativo di cui va calcolata la radice quadrata
%
%   Output:
%       x: risultato della radice quadrata

if x<0,    error("Parametro non valido! x < 0"); end
if x~=1 && x~=0
    rad = x;
    x = x/2;
    e = x;
    while e > abs(x)*2^-53
        x0 = x;
        x = (x0 + rad/x0)/2;
        e = abs(x-x0);
    end
    x = rad/x;
end
return
```

Per confrontare l'approssimazione ottenuta con la function sqrt di Matlab alle condizioni richieste si utilizza la seguente linea di comandi:

```
x = logspace(-10,10,20)
y = arrayfun(@(xi) radice(xi),x)
diff = y - arrayfun(@(xi) sqrt(xi),x)
```

il vettore restituito dalla differenza delle due funzioni per i venti punti equispaziati logaritmicamente nell'intervallo [1e-10,1e10] è un vettore di zeri.

Esercizio 5.

Metodo di Newton

```
function [x,n] = newton(f, f1, x0, tol, itmax)
%
%   x = newton(f, f1, x0, tolx, itmax)
%       Function che implementa il metodo di Newton per la risoluzione
%       dell'equazione f(x) = 0
%
%   Input:
%       f: stringa contenente la function che implementa f(x)
%       f1: stringa contenente la function che implementa f'(x)
%       x0: approssimazione iniziale della radice
%       tol: accuratezza richiesta
%       itmax: numero massimo di iterazioni richieste
%
%   Output:
%       x: approssimazione finale della radice
%       n: numero di iterazioni utilizzate
if tol < 0, error('Accuratezza non corretta.'),end
if itmax < 0, error('Numero di iterazioni non corretto.'),end
x = x0;
for n = 1:itmax
    fx = feval(f,x);
    f1x = feval(f1,x);
    if f1x == 0, break, end
    x0 = x;
    x = x0 - fx/f1x;
    if abs(x-x0) <= tol*(1+abs(x0)), break, end
end
if abs(x-x0) > tol*(1+abs(x0)), error('Il metodo non converge!'), end
return
```

Metodo delle secanti

```
function [x,n] = secanti(f, x0, x1, tol, itmax)
%
%   x = secanti(f, f1, x0, x1, tol, itmax)
%       Function che implementa il metodo delle secanti per la risuluzione
%       dell'equazione f(x) = 0
%
%   Input:
%       f: stringa contente la function che implementa la funzione f(x)
%       f1: stringa contente la function che implementa la function f'(x)
%       x0: approssimazione iniziale della radice
%       x1: approssimazione iniziale della radice
%       tol: accuratezza richiesta
%       itmax: numero massimo di iterazioni
%
%   Output:
%       x: approssimazione finale della radice
%       n: numero di iterazioni
if tol < 0, error('Accuratezza non corretta.'),end
if itmax < 0, error('Numero di iterazioni non corretto.'),end
fx0 = feval(f,x0);
x = x1;
for n = 0:itmax
    if abs(x-x0) <= tol*(1+abs(x0)), break, end
    fx = feval(f,x);
    if fx == fx0, break, end
    x1 = (fx*x0-fx0*x)/(fx-fx0);
    x0 = x;
    fx0 = fx;
    x = x1;
end
if abs(x-x0) > tol*(1+abs(x0)), error('Il metodo non converge'), end
return
```

Metodo di Steffensen

```
function [x,n] = steffensen(f, x0, tol, itmax)
%
%   x = steffensen(f,x0,tol,itmax)
%       Function che implementa il metodo di Steffensen per
%       la risoluzione dell'equazione f(x) = 0
%
%   Input:
%       f: function che implementa la funzione f(x)
%       x0: approssimazione iniziale della radice
%       tol: accuratezza richiesta
%       itmax: numero massimo di iterazioni richieste
%
%   Output:
%       x: approssimazione finale della radice
%       n: numero di iterazioni
if tol < 0, error('Accuratezza non corretta.'),end
if itmax < 0, error('Numero di iterazioni non corretto.'),end

x = x0;
for n = 1:itmax
    x0 = x;
    fx = feval(f,x0);
    gx = feval(f,x0+fx);
    if fx == gx, break, end
    x = x0 - (fx^2)/(gx-fx);
    if abs(x-x0) <= tol*(1+abs(x0)), break, end
end
if abs(x-x0) > tol*(1+abs(x0)), error('Il metodo non converge!'), end
return
```

Esercizio 6.

tol		Newton	Secanti	Steffensen
$1e-3$	x	$5.946116463605413e-01$	$5.946184776717106e-01$	$5.946116811419248e-01$
	n	3	3	4
$1e-6$	x	$5.946116440568356e-01$	$5.946116440568420e-01$	$5.946116440568371e-01$
	n	4	5	5
$1e-9$	x	$5.946116440568356e-01$	$5.946116440568355e-01$	$5.946116440568356e-01$
	n	5	6	6
$1e-12$	x	$5.946116440568356e-01$	$5.946116440568355e-01$	$5.946116440568356e-01$
	n	5	6	6

Si osserva ora la tabella del numero di iterazioni per metodo:

tol		Newton	Secanti	Steffensen
$1e-3$	n	3	3	4
	$n\ it.$	6	4	8
$1e-6$	n	4	5	5
	$n\ it.$	8	6	10
$1e-9$	n	5	6	6
	$n\ it.$	10	7	12
$1e-12$	n	5	6	6
	$n\ it.$	10	7	12

La tabella dimostra che, per tutti i valori di tol testati, il metodo delle secanti utilizza il minor numero di valutazioni, mentre il metodo di Steffensen utilizza il maggior numero di valutazioni tra i tre.

Esercizio 7.
Si nota preliminarmente che la funzione presenta una molteplicità $m > 1$, diversamente dalla funzione dell'esercizio precedente. Si ottengono i seguenti risultati:

tol		Newton	Secanti	Steffensen
$1e-3$	x	$5.969479343078770e-01$	$5.991437227787726e-01$	$5.973965526716525e-01$
	n	13	18	18
$1e-6$	x	$5.946140179818806e-01$	$5.946156634766229e-01$	$5.946143706771916e-01$
	n	30	41	35
$1e-9$	x	$5.946116464662755e-01$	$5.946116487688006e-01$	$non\ converge$
	n	47	65	
$1e-12$	x	$5.946116440592810e-01$	$5.946116440610055e-01$	$non\ converge$
	n	64	90	

Si osserva ora la tabella del numero di iterazioni per metodo:

tol		Newton	Secanti	Steffensen
1e-3	n	13	16	18
	$n\ it.$	26	17	36
1e-6	n	30	41	35
	$n\ it.$	60	42	70
1e-9	n	47	65	$non\ converge$
	$n\ it.$	94	66	
1e-12	n	64	90	$non\ converge$
	$n\ it.$	128	91	

Avendo una sottrazione al denominatore, il metodo di Steffensen si interrompe quando $f\big(x_n + f(x_n)\big) \approx f(x_n)$, in quanto, preso n sufficientemente grande, $fl\left(f\big(x_n + f(x_n)\big)\right) = fl\big(f(x_n)\big)$; ciò non garantisce che il metodo converga alla radice desiderata generando errori.

Esercizio 8.

Risoluzione dei sistemi lineari mediante fattorizzazione LU con pivoting

```
function b = mialu(A,b)
%
%   b = mialu(A,b)
%
%   Metodo di risoluzione di sistemi lineari tramite fattorizzazione LU
%   con pivoting parziale per la risoluzione di sistemi lineari
%
%   Input:
%       A: matrice quadrata nonsingolare
%       b: vettore dei termini noti
%
%   Output
%       x: vettore delle incognite
%

n = size(A,1);
if n ~= size(A,2), error('La matrice non è quadrata.'), end
if n ~= size(b), error('La dimensione di b non coincide con la dimensione della
matrice.'), end
[A,p] = miafattlup(A,n);
b = b(p);
for i = 2:n
    b(i:n) = b(i:n)-A(i:n,i-1)*b(i-1);
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
return
```

```
function [A,p] = miafattlup(A,n)
%
%   [A,p] = miafattlup(A,n)
%
%   Metodo di fattorizzazione LU con pivoting parziale per la risoluzione
%   di sistemi lineari
%
%   Input:
%       A: matrice quadrata nonsingolare
%       n: dimensione della matrice A
%
%   Output:
%       A: matrice contenente i fattori L e U
%       p: vettore di permutazione
%

p = (1:n).';
for i = 1:n
    [mi,ki] = max(abs(A(i:n,i)));
    if mi == 0, error('Matrice 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
return
```

Le matrici dei coefficienti negli esempi seguenti sono state generate casualmente con il comando `randi([-5,5],4)`, come vettore delle soluzioni è stato generato un vettore x con il comando `x = (1:4).'` e il vettore dei termini noti b è stato ottenuto dal prodotto `A*x`.
L'obiettivo è confrontare la soluzione del sistema con la soluzione del sistema data dal metodo mialu.

Esempio 1

$$\begin{pmatrix} -5 & 3 & 3 & 4 \\ -2 & 1 & 4 & -2 \\ -5 & -5 & 0 & -2 \\ 0 & -5 & -4 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 26 \\ 4 \\ -23 \\ -10 \end{pmatrix}$$

Soluzione del sistema: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$

Soluzione mialu: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1.000000000000000e+00 \\ 2.000000000000000e+00 \\ 3.000000000000000e+00 \\ 4.000000000000000e+00 \end{pmatrix}$

Esempio 2

$$\begin{pmatrix} -5 & 0 & -2 & -5 \\ -5 & 3 & 2 & 3 \\ 2 & 2 & 1 & -2 \\ 1 & 3 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -31 \\ 19 \\ 1 \\ 8 \end{pmatrix}$$

Soluzione del sistema: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$

Soluzione mialu: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 9.999999999999998e-01 \\ 1.999999999999999e+00 \\ 3.000000000000000e+00 \\ 4.000000000000000e+00 \end{pmatrix}$

Esercizio 9.

Risoluzione dei sistemi lineari mediante fattorizzazione LDL^t

```
function b = mialdl(A,b)
%
%   b = mialdl(A,b)
%
%   Metodo di risoluzione di sistemi lineari tramite fattorizzazione LDL per la
risoluzione di sistemi lineari
%
%   Input:
%       A: Matrice dei coefficienti. Matrice nxn sdp.
%       b: Vettore dei termini noti
%
%   Output
%       x: Vettore delle incognite
%

n = size(A,1);
if n ~= size(A,2), error('La matrice non è quadrata.'), end
if n ~= size(b), error('La dimensione di b non coincide con la dimensione della
matrice.'), end
A = miafattldl(A,n);
for i = 2:n
    b(i:n) = b(i:n)-A(i:n,i-1)*b(i-1);
end
b = b./diag(A);
for i = n:-1:2
    b(1:i-1) = b(1:i-1)-A(i,1:i-1).'*b(i);
end
return
```

```
function A = miafattldl(A,n)
%
%   A = miafattldl(A,n)
%
%   Metodo di fattorizzazione LDL per la risoluzione di sistemi lineari
%
%   Input:
%       A: matrice quadrata sdp.
%       n: dimensione della matrice A
%
%   Output:
%       A: matrice contenente i fattori L e D
%

if A(1,1)<0, error('la matrice non è sdp.'), end
A(2:n,1) = A(2:n,1)/A(1,1);
for j=2:n
    v = (A(j,1:j-1).') .* diag(A(1:j-1,1:j-1));
    A(j,j) = A(j,j) - A(j,1:j-1)*v;
    if A(j,j)<0, error('la matrice non è sdp.'), end
    A(j+1:n,j) =(A(j+1:n,j)-A(j+1:n,1:j-1)*v)/A(j,j);
end
return
```

Le matrici dei coefficienti negli esempi sono state generate casualmente con i seguenti comandi:

```
A = randi([-5,5],4)
d = randi([15,30],4,1)
A = tril(A,-1) + triu(A', 1) + diag(d)
```

Da questi comandi si genera una matrice sdp.

Come vettore delle soluzioni è stato generato un vettore x con il comando `x = (1.4).'` e il vettore dei termini noti b è stato ottenuto dal prodotto `A*x`.

L'obiettivo è confrontare la soluzione del sistema con la soluzione del sistema data dal metodo mialdl.

Esempio 1

$$\begin{pmatrix} 25 & -5 & 5 & -2 \\ -5 & 20 & 0 & 2 \\ 5 & 0 & 29 & 4 \\ -2 & 2 & 4 & 16 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 22 \\ 43 \\ 108 \\ 78 \end{pmatrix}$$

Soluzione del sistema: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$

Soluzione mialdl: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1.000000000000000e+00 \\ 2.000000000000000e+00 \\ 3.000000000000000e+00 \\ 4.000000000000001e+00 \end{pmatrix}$

Esempio 2

$$\begin{pmatrix} 18 & 0 & 2 & 5 \\ 0 & 9 & 16 & 3 \\ 2 & 8 & 19 & -2 \\ 5 & 3 & -2 & 22 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 44 \\ 70 \\ 69 \\ 93 \end{pmatrix}$$

Soluzione del sistema: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$

Soluzione mialdl: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1.000000000000000e+00 \\ 2.000000000000000e+00 \\ 3.000000000000000e+00 \\ 3.999999999999999e+00 \end{pmatrix}$

Esercizio 10. Nella function linsis(n,k,simme) il parametro k è utilizzato per definire la variabile sigma
`sigma = 10^(-2*(1-k))/n`
la quale, a sua volta, è utilizzata per definire la matrice dei coefficienti
`A = q1*diag([sigma 2/n:1/n:1])*q2`
Per valori di k grandi si hanno valori di sigma grandi e, di conseguenza, valori degli elementi della matrice dei coefficienti grandi.
Si ricorda che il numero di condizionamento del problema ha valore $k(A) = \left||A|\right| * \left||A^{-1}|\right|$ e, su Matlab, si può ricavare con la function cond.
Generando A e b con la linea di comando linsis(10,1) otteniamo il numero di condizionamento pari a `c = cond(A)` = $9.999999999999993e+00$, ovvero una matrice ben condizionata: provando a risolvere il sistema con la function mialu, infatti, si ottiene un vettore delle soluzioni approssimativamente uguale a quello che si dovrebbe ottenere. Utilizzando, invece, la linea di comando linsis(10,10) otteniamo un numero di condizionamento pari a `c = cond(A)` = $7.119787475990509e+18 \gg 1$, ovvero una matrice mal condizionata: provando a risolvere il sistema con la function mialu, infatti, si ottiene un vettore delle soluzioni diverso da quello che si dovrebbe ottenere.

Esercizio 11. Nella function linsis(n,k,simme) se il parametro simme è definito si ottiene una matrice sdp.
Generando A e b con la linea di comando linsis(10,1,1) otteniamo il numero di condizionamento pari a `c = cond(A)` = $1.000000000000001e+01$, ovvero una matrice ben condizionata: provando a risolvere il sistema con la function mialdl, infatti, si ottiene un vettore delle soluzioni approssimativamente uguale a quello che si dovrebbe ottenere. Utilizzando, invece, la linea di comando linsis(10,10,1) otteniamo un numero di condizionamento pari a `c = cond(A)` = $1.037238529826488e+18 \gg 1$, ovvero una matrice mal condizionata: provando a risolvere il sistema con la function mialdl si ottiene il messaggio di errore generato dalla function quando la matrice in ingresso non è sdp; ciò non dovrebbe succedere in quanto la function linsis, quando ha il parametro simme definito, dovrebbe restituire una matrice sdp.

Esercizio 12.

Risoluzione dei sistemi lineari sovradeterminati

```
function [x,nr] = miaqr(A,b)
%
%   [x,nr] = miaqr(A,b)
%
%   Metodo di risoluzione dei sistemi lineari sovradeterminati (m > n =
%   rank(A)).
%
%   Input:
%       A: Matrice dei coefficienti. Matrice mxn con m >= n = rank
%       b: Vettore dei termini noti. Vettore di dimensione m
%
%   Output:
%       x: Vettore delle incognite. Vettore di dimensione n
%       nr: norma del vettore residuo
%

A = miafattqr(A);
[m,n] = size(A);

for i = 1:n
      vtb = [zeros(i-1,1); 1 ; A(i+1:m,i)].'*b;
    vtv = [zeros(i-1,1); 1 ; A(i+1:m,i)].'*[zeros(i-1,1); 1 ; A(i+1:m,i)];
      b = b - (2/vtv)*vtb*[zeros(i-1,1); 1 ; A(i+1:m,i)];
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
```

```
function A = miafattqr(A)
%
%   A = miafattqr(A)
%
%   Metodo di fattorizzazione QR per matrici mxn con m > n = rank(A)
%
%   Input:
%       A: matrice mxn da fattorizzare
%   Output:
%       A: matrice mxn contenente i fattori Q e R
%

[m,n] = size(A);
if m < n, error('Dati inseriti non corretti: m < n'), end
for i = 1:n
    alfa = norm(A(i:m,i));
    if alfa == 0, error('Dati inseriti non corretti: la matrice non ha rango
massimo'), end
    if A(i,i) >= 0, alfa = -alfa; end
    v1 = A(i,i) - alfa;
    A(i,i) = alfa;
    A(i+1:m,i) = A(i+1:m,i)/v1;
    beta = -v1/alfa;
    A(i:m,i+1:n) = A(i:m,i+1:n) - (beta*[1; A(i+1:m,i)])*([1
A(i+1:m,i).']*A(i:m,i+1:n));
end
return
```

Le matrici dei coefficienti e i vettori dei termini noti negli esempi sono state generate casualmente con i seguenti comandi:

```
A = randi([-5,5],5,3)
b = randi([-10,10],5,1)
```

Esempio 1

$$\begin{pmatrix} 2 & 3 & 2 \\ -5 & 3 & -5 \\ 4 & -1 & -2 \\ 5 & 2 & -5 \\ 2 & -4 & -4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \\ -4 \\ 9 \\ -10 \end{pmatrix}$$

Soluzione data dall'operatore \: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4.313385953608250e-01 \\ 2.559197809278351e+00 \\ 6.399001288659793e-02 \end{pmatrix}$

Soluzione miaqr: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4.313385953608246e-01 \\ 2.559197809278350e+00 \\ 6.399001288659799e-02 \end{pmatrix}$ $\qquad nr = 4.215941457270728e+00$

Esempio 2

$$\begin{pmatrix} -1 & 0 & -2 \\ -1 & -1 & 2 \\ 3 & 2 & 2 \\ 3 & 2 & -4 \\ -3 & 3 & -4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 10 \\ -3 \\ 2 \\ -6 \end{pmatrix}$$

Soluzione data dall'operatore \: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4.808030996829873e-01 \\ -1.958436069038394e+00 \\ -1.849242691088413e-01 \end{pmatrix}$

Soluzione miaqr: $\mathrm{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4.808030996829871e-01 \\ -1.958436069038393e+00 \\ -1.849242691088412e-01 \end{pmatrix}$ $\quad nr = 9.663882786111307e+00$

Dagli esempi si nota che le soluzioni date dall'operatore \ coincidono approssimativamente alle soluzioni date dal metodo miaqr.

Esercizio 13.

Dato $A = \begin{pmatrix} 1 & 3 & 2 \\ 3 & 5 & 4 \\ 5 & 7 & 6 \\ 3 & 6 & 4 \\ 1 & 4 & 2 \end{pmatrix}$ e $b = \begin{pmatrix} 15 \\ 28 \\ 41 \\ 33 \\ 22 \end{pmatrix}$, dal comando miaqr(A,b) si ottiene

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3.000000000000008e+00 \\ 5.800000000000001e+00 \\ -2.500000000000008e+00 \end{pmatrix}, nr = 1.264911064067357e+00$$

Data la matrice diagonale D = diag(1:5) al comando miaqr(D*A,D+b) si ottiene, invece,

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -6.025699862322442e-01 \\ 4.701698026617703e+00 \\ 1.758375401560388e+00 \end{pmatrix}, nr = 3.735151112342407e+00$$

Si osserva che sistema lineare $DAx = Db$ è rappresentabile come segue

$$\begin{cases} x_1 + 3x_2 + 2x_3 \\ 6x_1 + 10x_2 + 8x_3 \\ 15x_1 + 21x_2 + 18x_3 \\ 12x_1 + 24x_2 + 16x_3 \\ 5x_1 + 20x_2 + 10x_3 \end{cases} = \begin{cases} 15 \\ 56 \\ 123 \\ 132 \\ 110 \end{cases} \rightarrow \begin{cases} x_1 + 3x_2 + 2x_3 \\ 2(3x_1 + 5x_2 + 4x_3) \\ 3(5x_1 + 7x_2 + 6x_3) \\ 4(3x_1 + 6x_2 + 4x_3) \\ 5(x_1 + 4x_2 + 2x_3) \end{cases} = \begin{cases} 15 \\ 2 * 28 \\ 3 * 41 \\ 4 * 33 \\ 5 * 22 \end{cases} \rightarrow \begin{cases} x_1 + 3x_2 + 2x_3 \\ 3x_1 + 5x_2 + 4x_3 \\ 5x_1 + 7x_2 + 6x_3 \\ 3x_1 + 6x_2 + 4x_3 \\ x_1 + 4x_2 + 2x_3 \end{cases} = \begin{cases} 15 \\ 28 \\ 41 \\ 33 \\ 22 \end{cases}$$

$$\rightarrow Ax = b$$

La soluzione del sistema lineare $DAx = Db$ dovrebbe, dunque, risultare uguale alla soluzione di $Ax = b$, tuttavia il comando miaqr(D*A,D*b) restituisce un risultato diverso rispetto a quello restituito dal comando miaqr(A,b); ciò succede perché la function miaqr restituisce la soluzione del sistema lineare nel senso dei minimi quadrati, ovvero la soluzione per cui il quadrato della norma euclidea del vettore residuo $\left|\left|r\right|\right|_2^2 = \left|\left|Ax - b\right|\right|_2^2$ è minima. Osserviamo che per

$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3.000000000000008e+00 \\ 5.800000000000001e+00 \\ -2.500000000000008e+00 \end{pmatrix}$, otteniamo la norma del vettore residuo $\left|\left|r\right|\right| = \left|\left|DAx - Db\right|\right| = 5.276362383309186e+00$, ovvero un valore maggiore rispetto alla norma restituita dalla function miaqr(D*A,D*b).

Esercizio 14.

Risoluzione di sistemi non lineari

```
function [x,nit] = newton(fun, jacobian, x0, tol, maxit)
%
%   [x,nit] = newton(fun, jacobian, x0, tol, maxit)
%       Function che implementa il metodo di Newton per la risoluzione di
%       sistemi non lineari
%
%   Input:
%       fun: vettore contenente le funzioni non lineari
%       jacobian: matrice contente la jacobiana di fun
%       x0: vettore contenente le approssimazioni iniziali delle radici
%       tol: accuratezza richiesta
%       maxit: numero massimo di iterazioni richieste
%
%   Output:
%       x: vettore contenente le radici
%       nit: numero di iterazioni utilizzate
%
if nargin == 3
    tol = 1e-13;
    maxit = 15;
end
if nargin == 4, maxit = 15; end
if tol < 0, error('Dati errati: accuratezza negativa.'), end
if maxit <= 0, error('Dati errati: numero di iterazioni <= 0.'), end
n = size(jacobian,1);
if n ~= size(jacobian,2), error('Dati errati: jacobiana non quadrata'), end
x = x0;

for nit = 1:maxit
    fx = -feval(fun,x);
    jx = feval(jacobian,x);
    fx = mialu(jx, fx);
    x0 = x;
    x = x0+fx;
    if norm(fx) <= tol*(1+norm(x0)), break, end
end
if norm(fx) > tol*(1+norm(x0)), error('Il metodo non converge.'), end
return
```

Esercizio 15.

tol		$f_1(x)$
1e-3	x_1	$1.000000007127784e+00$
	x_2	$2.000000000000000e+00$
	n	7
1e-8	x_1	1
	x_2	2
	n	8
1e-13	x_1	1
	x_2	2
	n	9

tol		$f_2(x)$
1e-3	x_1	$9.999998150012226e-01$
	x_2	$9.999997068384456e-01$
	x_3	$1.000000239080166e+00$
	n	5
1e-8	x_1	1
	x_2	1
	x_3	1
	n	7
1e-13	x_1	1
	x_2	1
	x_3	1
	n	7

Esercizio 16.

Lagrange

```
function YQ = lagrange(X,Y,XQ)
%   YQ = lagrange(X,Y,XQ)
%
%       Calcola il polinomio interpolante in forma di Lagrange definito
%       dalle coppie (X(i),Y(i)) nei punti del vettore XQ
%

n = length(X);
if length(Y)~=n, error('Dati errati: i vettori X e Y hanno lunghezze diverse'); end
if length(unique(X))~=n, error('Dati errati: le componenti del vettore X non sono
distinte'); end

YQ = zeros(size(XQ));
for i=1:n
    YQ = YQ + Y(i)*lin(XQ,X,i);
end
return
```

```
function L = lin(XQ,X,i)
%
%   Calcola il polinomio Lin(x)
%

L = ones(size(XQ));
n = length(X) - 1;
Xi = X(i);
X=X([1:i-1,i+1:n+1]);
for k=1:n
    L = L.*(XQ-X(k))/(Xi-X(k));
end
return
```

Esercizio 17.

Newton

```
function YQ = newton(X,Y,XQ)
%
%   YQ = newton(X,Y,XQ)
%       Calcola il polinomio interpolante in forma di Newton definito
%       dalle coppie (X(i),Y(i)) nei punti del vettore XQ
%       Input:
%           X: vettore delle ascisse
%           Y: vettore delle ordinate
%           XQ: matrice in cui calcolare il polinomio
%           interpolante
%       Output:
%           YQ: vettore delle ordinate del polinomio

if length(X) ~= length(Y), error('Dati errati: dimensioni di X e Y non coincidono.');
end
if length(unique(X)) ~= length(X) , error('Dati errati: le componenti del vettore X
non sono distinte'); end
df = divdif(X,Y);
n = length(df)-1;
YQ = df(n+1)*ones(size(XQ));
for i=n:-1:1
    YQ = YQ.*(XQ-X(i)) + df(i);
end
return
```

```
function df = divdif(X,Y)
%
%   dY = divdif(X,Y)
%       Calcola la differenza divisa sulle coppie [xi,fi]
%       Input:
%           X: vettore delle ascisse
%           Y: vettore delle ordinate
%       Output:
%           dY: vettore delle differenze divise

n = length(X)-1;
df = Y;
for j=1:n
    for i=n+1:-1:j+1
        df(i)=(df(i)-df(i-1))/(X(i)-X(i-j));
    end
end
return
```

Esercizio 18.

Hermite

```
function YQ = hermite(X,Y,XQ,dY)
%
%   YQ = hermite(X,Y,XQ,dY)
%       Calcola il polinomio interpolante in forma di Hermite definito
%       dalle coppie (X(i),Y(i)) nei punti del vettore XQ
%       Input:
%           X: vettore delle ascisse
%           Y: vettore delle ordinate
%           XQ: matrice in cui calcolare il polinomio
%           interpolante
%           dY: vettore contenente i valori della derivata per i punti X
%       Output:
%           YQ: vettore delle ordinate del polinomio

if length(X) ~= length(Y), error('Dati errati: dimensioni di X e Y non coincidono.');
end
if length(unique(X)) ~= length(X) , error('Dati errati: le componenti del vettore X
non sono distinte'); end
if length(X) ~= length(dY), error('Dati errati: dimensioni di X e Y non
coincidono.'); end
df = divdifher(X,Y,dY);
n = length(df)-1;
YQ = df(n+1)*ones(size(XQ));
for i=n:-1:1
    YQ = YQ.*(XQ-X(round(i/2))) + df(i);
end
return
```

```
function df = divdifher(X,Y,dY)
%
%   dY = divdif(X,Y)
%       Calcola la differenza divisa sulle coppie [xi,fi] per il polinomio
%       di Hermite
%   Input:
%       X: vettore delle ascisse
%       Y: vettore delle ordinate
%       dY: vettore contenente i valori della derivata per i punti X
%
%   Output:
%       df: vettore delle differenze divise

n = length(X)-1;
df = repelem(Y,2);
for i=n+1:-1:1
    df(i*2) = dY(i);
end
for i=2*n+1:-2:3
    df(i) = (df(i)-df(i-2))/(X((i+1)/2)-X((i-1)/2));
end
for j = 2:2*n+1
    for i = (2*n+2):-1:j+1
        df(i) = (df(i)-df(i-1))/(X(round(i/2))-X(round((i-j)/2)));
    end
end
return
```

Esercizio 19.

Chebyshev

```
function X = chebyshev(n,a,b)
%
%   X = chebyshev(n,a,b)
%       Calcola le ascisse di Chebyshev relativo al calcolo del polinomio
%       interpolante
%
%   Input:
%       n: grado del polinomio
%       a: estremo sinistro dell'intervallo preso in considerazione
%       b: estremo destro dell'intervallo preso in considerazione
%
%   Output:
%       X: vettore contenete le ascisse di Chebyshev

if n <= 0, error('Dati errati: n<=0'), end
X = (a+b)/2 + ((b-a)/2)*cos( pi * (2*(n:-1:0)+1)./(2*(n+1)) );
return
```

Esercizio 20.

Spline0

```
function YQ = spline0(X,Y,XQ)
%
%   YQ = spline0(X,Y,XQ)
%       Calcola la spline cubica naturale interpolante una funzione
%
%   Input:
%       X: vettore delle ascisse
%       Y: vettore delle ordinate
%       XQ: matrice contenente i punti delle ascisse in cui calcolare il polinomio
%           interpolante
%   Output:
%       YQ: vettore dei valori interpolati

n = length(X);
nq = length(XQ);
if n ~= length(Y), error('Dati errati: dimensioni di X e Y non coincidono.'); end
if length(unique(X)) ~= length(X) , error('Dati errati: le componenti del vettore X
non sono distinte'); end
phi = ((X(3:n-1)-X(2:n-2))./(X(4:n)-X(2:n-2)));
xi = ((X(3:n-1)-X(2:n-2))./(X(3:n-1)-X(1:n-3)));
df = divdifp(X,Y,3).';
m = [0; tridia(phi,2*ones(n-2,1),xi,6*df); 0];
df = divdifp(X,Y,2).';
j = 1;
YQ = ones(size(XQ));
for i=2:n
    hi = X(i) - X(i-1);
    qi = df(i-1) - (m(i)-m(i-1))*hi/6;
    ri = Y(i-1) - m(i-1)*(hi^2)/6;
    while j <=nq && XQ(j) <= X(i)
        YQ(j) = ((XQ(j)-X(i-1))^3*m(i)+(X(i)-XQ(j))^3*m(i-1))/(6*hi);
        YQ(j) = YQ(j) + qi*(XQ(j)-X(i-1)) + ri;
        j = j+1;
    end
end
```

```
while j <= nq && XQ(j) > X(n)
        YQ(j) = ((XQ(j)-X(n-1))^3*m(n)+(X(n)-XQ(j))^3*m(n-1))/(6*hi);
        YQ(j) = YQ(j) + qi*(XQ(j)-X(n-1)) + ri;
        j = j+1;
end
return
```

```
function df = divdifp(X,Y,nit)
%
%   df = divdifp(X,Y,nit)
%       Restituisce le differenze divise fermandosi all'iterazione nit
%
if nit < 0, error('Dati errati: numero iterazioni < 0'), end
n = length(X);
if length(Y) ~= n,error('Dati errati: dimensione dei vettori X e Y non coincidono');
end
df = Y;
for j=1:nit-1
    for i=n:-1:j+1
        df(i)=(df(i)-df(i-1))/(X(i)-X(i-j));
    end
end
df=df(nit:n);
return
```

```
function b = tridia(l,d,u,b)
%
%   df= tridia(phi,xi,df)
%       Function che determina il vettore delle incognite per un sistema
%       lineare date in ingresso le diagonali significative della matrice
%       dei coefficienti e il vettore delle incognite.
%

n = length(d);
for i=1:n-1
    if d(i) == 0, error('Matrice non fattorizzabile LU'),end
    l(i) = l(i)/d(i);
    d(i+1) = d(i+1)-l(i)*u(i);
end
for i=2:n
    b(i) = b(i) - l(i-1)*b(i-1);
end
b(n) = b(n)/d(n);
for i=n-1:-1:1
    b(i) = (b(i)-b(i+1)*u(i))/d(i);
end
return
```

Esercizio 21.

Ascisse equidistanti

n	Lagrange	Newton	Hermite
4	$2.403815580286168e-01$	$7.070135746606335e-01$	$4.998681947544071e-01$
8	$2.473586065593149e-01$	$2.473586065593149e-01$	$6.264604673549260e-01$
16	$2.107551869554572e+00$	$2.107551869557100e+00$	$1.075622145122779e+02$
24	$3.641003656212056e+01$	$3.641003656210778e+01$	$3.307374064298416e+04$
32	$7.052964658243415e+02$	$7.052964658223567e+02$	$1.257038491150360e+07$
40	$1.446715032522146e+04$	$1.446715060998881e+04$	$6.301867406233521e+09$

n	Spline naturale	Spline Matlab
4	$7.013574660633484e-01$	$7.070135746606334e-01$
8	$2.460753323436626e-01$	$2.471061445541360e-01$
16	$3.089074684813309e-02$	$3.089073253328101e-02$
24	$5.372462774453357e-03$	$5.372462710659165e-03$
32	$1.314616578641292e-03$	$1.314616578444783e-03$
40	$4.338554349913037e-04$	$4.338554349905266e-04$

Ascisse di Chebyshev

n	Lagrange	Newton	Hermite
4	$4.020169252071493e-01$	$4.020169252071493e-01$	$2.672042285524814e-01$
8	$1.708356260402806e-01$	$1.708356260402808e-01$	$7.245750049018529e-02$
16	$3.261358359847166e-02$	$3.261358359847255e-02$	$3.895373220544007e-03$
24	$6.948423587493924e-03$	$6.948423587493591e-03$	$1.949941176364010e-03$
32	$1.401747194729241e-03$	$1.401747194755498e-03$	$3.039532720372281e+01$
40	$2.894607646986014e-04$	$2.894607647125902e-04$	$4.551874800612261e+05$

n	Spline naturale	Spline Matlab (not-a-knot)
4	$3.300896163075316e-01$	$3.592478392850814e-01$
8	$1.337508670981575e-01$	$1.338711311519075e-01$
16	$1.362664984216511e-02$	$1.362666088573783e-02$
24	$3.537101277692467e-03$	$3.537101189318159e-03$
32	$2.756837617875929e-03$	$2.756837617649999e-03$
40	$1.372501951093952e-03$	$1.372501951093286e-03$

Esercizio 22.

$$f(x) = \sin(2\pi x)$$

n	Spline naturale	Spline Matlab (not-a-knot)
5	$8.965080522515101e-03$	$6.479355623625926e-02$
10	$4.472573474414432e-04$	$2.649339337953582e-03$
15	$8.328505616517212e-05$	$3.635863745360401e-04$
20	$2.567926759022843e-05$	$8.751628083206175e-05$
25	$1.053441357834473e-05$	$2.886551146477623e-05$
30	$5.065905102186186e-06$	$1.164157031628699e-05$
35	$2.723736119314424e-06$	$5.397637143467193e-06$
40	$1.590316647259726e-06$	$2.772329771229265e-06$
45	$9.939815188708678e-07$	$1.539902200371757e-06$
50	$6.519610061817005e-07$	$9.099142061486098e-07$

$$g(x) = \cos(2\pi x)$$

n	Spline naturale	Spline Matlab (not-a-knot)
5	$9.517411707796586e-02$	$1.809815027162931e-02$
10	$2.040036777593446e-02$	$3.485340157968198e-03$
15	$8.812831778400421e-03$	$7.878291718896868e-04$
20	$4.907953097219586e-03$	$2.605526893537657e-04$
25	$3.126576412744497e-03$	$1.088800921688060e-04$
30	$2.165789528203788e-03$	$5.307587114833368e-05$
35	$1.588774878993338e-03$	$2.883449454615938e-05$
40	$1.215209344729962e-03$	$1.697286640600648e-05$
45	$9.595011039276180e-04$	$1.062633267234769e-05$
50	$7.768338363960403e-04$	$6.986130921871059e-06$

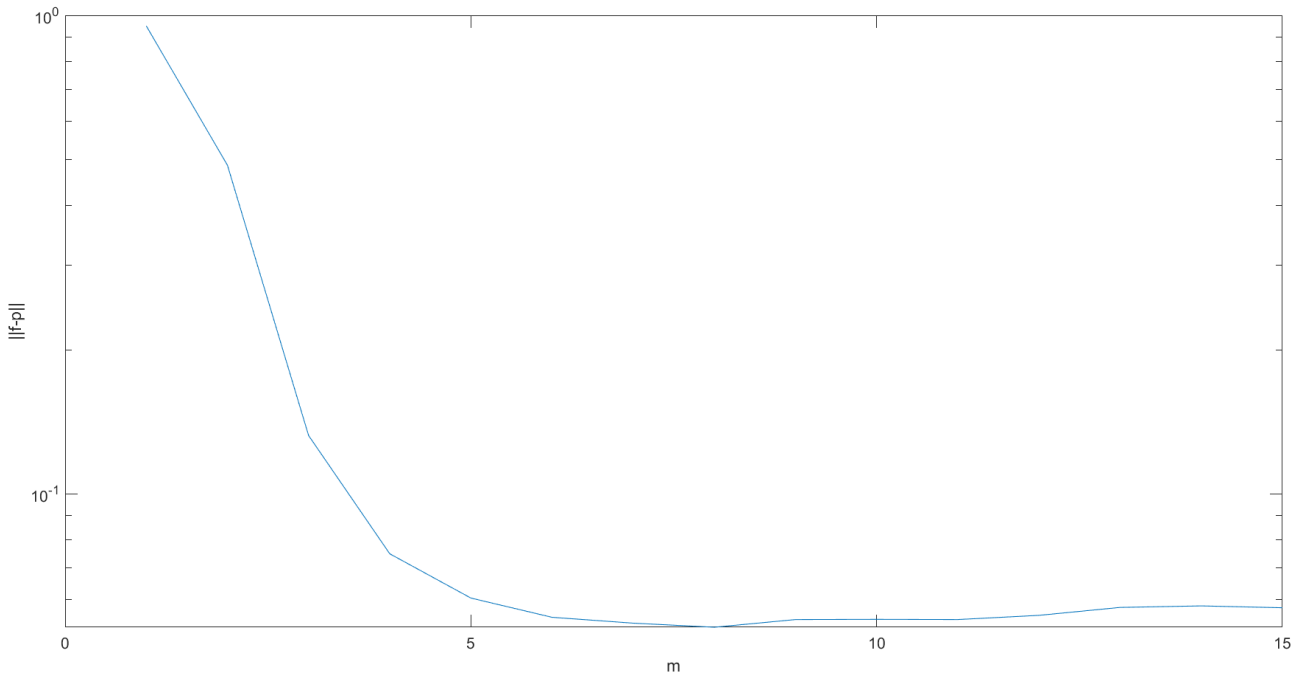
Dai risultati ottenuti è possibile osservare che, per la funzione $f(x) = \sin(2\pi x)$, la spline cubica naturale presenta un errore massimo minore rispetto alla spline cubica not-a-knot, mentre per la funzione $g(x) = \cos(2\pi x)$ succede l'opposto. Osserviamo che le condizioni aggiuntive della spline cubica naturale sono $s_3''(0) = s_3''(1) = 0$, risultati in linea con $f''(0) = -\sin(0) = 0 = -\sin(2\pi) = f''(1)$, ma non con $g''(0) = -\cos(0) = -1 = -\cos(2\pi) = g''(1)$.

Esercizio 23.

Codice utilizzato:

```
f = @(x)sin(pi.*x.^2)
for i = 1:15
       p = polyfit(x,fx,i)
       y = polyval(p,x)
       err(i) = norm(y-fx, Inf)
end
semilogy(m,err)
```

È stata utilizzata la function polyfit per calcolare i polinomi di approssimazione ai minimi quadrati.



Esercizio 24.

c_{in}

```
function c = cin(n)
%
%   c = cin(n)
%       Function che restituisce i pesi della quadratura della formula di
%       Newton-Cotes di grado n
%
%   Input:
%       n: grado della formula di quadratura
%   Output:
%       c: vettore dei coefficienti della quadratura della formula

if n <= 0, error('Grado non corretto'), end
c = zeros(1,n);
for i = 0:n
    d = i - [0:i-1 i+1:n];
    den = prod(d);
    a = poly([0:i-1 i+1:n]);
    a = [a./((n+1):-1:1) 0];
    num = polyval(a,n);
    c(i+1) = num/den;
end
return
```

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	65/45	8/15	65/45	14/45					
5	32/97	125 /96	125 /144	125 /144	125 /96	32/97				
6	41 /140	54/35	27 /140	68/35	27 /140	54/35	41 /140			
7	108 /355	71/49	82 /153	224 /185	224 /185	82 /153	71/49	108 /355		
9	33 /115	419 /265	23 /212	307 /158	65 /112	65 /112	307 /158	23 /212	419 /265	33 /115

Esercizio 25.

$$I(f) = \int_0^1 e^{3x}\, dx = \frac{1}{3}[e^{3x}]_0^1 = \frac{1}{3}(e^3 - 1)$$

n	$E_n(f)$
1	$-4.180922820531278e + 00$
2	$-1.402032263607653e - 01$
3	$-6.409819710703601e - 02$
4	$-1.827410064641377e - 03$

5	$-1.038246005103716e-03$
6	$-1.990061007006716e-05$
7	$-1.226107634177964e-05$
9	$-1.049225657467900e-07$

Esercizio 26.

Composita

```
function [If1,err,nfeval] = composita(fun,a,b,n,tol)
%
%   [If,err,nfeval] = composita(fun,a,b,n,tol)
%       Function che implementa la formula di Newton Cotes di grado n
%       composita
%
%   Input:
%       fun: identificatore di una function che calcola la funzione
%       integranda
%       a: estremo sinistro dell'intervallo
%       b: estremo destro dell'intervallo
%       n: grado della formula di Newton-Cotes
%       tol: tolleranza richiesta
%
%   Output:
%       If: approssimazione dell'integrale
%       err: errore della quadratura
%       nfeval: numero di valutazioni richieste
%

if tol < 0, error('tolleranza in ingresso non corretta.'), end
if n <= 0, error('grado in ingresso non corretto.'), end
mu = 1 + mod(n,2);
c = cin(n);
k = n;
x = linspace(a,b,k+1);
fx = feval(fun,x);
h = (b-a)/k;
If1 = h*sum(fx(1:n+1).*c(1:n+1));
err = tol+eps;

while tol < err
    k = k*2;
    x = linspace(a,b,k+1);
    fx(1:2:k+1) = fx(1:1:k/2+1);
    fx(2:2:k) = feval(fun,x(2:2:k));
    h = (b-a)/k;
    If = 0;
    for i = 1:n+1
        If = If + h*sum(fx(i:n:k))*c(i);
    end
    If = If + h*fx(k+1)*c(n+1);
    err = abs(If-If1)/(2^(n+mu)-1);
    If1 = If;
end
nfeval = length(fx);
return
```

Esercizio 27.

tol	n	nfeval
10^{-2}	1	5
	2	33
	3	7
	4	9
	5	11
	6	13
	7	15
	9	19
10^{-3}	1	65
	2	65
	3	13
	4	33
	5	11
	6	13
	7	15
	9	19
10^{-4}	1	257
	2	129
	3	97
	4	129
	5	41
	6	49
	7	29
	9	19
10^{-5}	1	1025
	2	257
	3	193
	4	129
	5	161
	6	97
	7	113
	9	37
10^{-6}	1	2049
	2	513
	3	385
	4	257
	5	321
	6	193
	7	113
	9	37

Esercizio 28.

Formula adattativa dei trapezi

```
function [I2,nfeval] = adaptrap(f,a,b,tol,fa,fb)
%
%   [I2,nfeval] = adaptrap(f,a,b,fa,fb,tol)
%       Function che implementa la formula adattativa dei trapezi
%
%   Input:
%       f: identificatore di una function che calcola la funzione
%       integranda
%       a: estremo sinistro dell'intervallo
%       b: estremo destro dell'intervallo
%       tol: tolleranza richiesta
%       fa: valutazione funzionale di a
%       fb: valutazione funzionale di b
%
%   Output:
%       I2: approssimazione dell'integrale
%       nfeval: numero di valutazioni funzionali utilizzate
%

if tol < 0, error('Tolleranza non corretta'), end
nfeval = 0;
if a == b
    I2 = 0;
else
    if nargin <= 4
        fa = feval(f,a);
        fb = feval(f,b);
        nfeval = 2;
    end
    h = (b-a)/2;
    x1 = (a+b)/2;
    f1 = feval(f,x1);
    nfeval = nfeval + 1;
    I1 = h*(fa+fb);
    I2 = I1/2 + h*f1;
    e = abs(I2-I1)/3;
    if e > tol
        [I21,n1] = adaptrap(f,a,x1,tol/2,fa,f1);
        [I22,n2] = adaptrap(f,x1,b,tol/2,f1,fb);
        I2 = I21 + I22;
        nfeval = nfeval + n1 + n2;
    end
end
return
```

Esercizio 29.

Formula adattativa di Simpson

```
function [I4,nfeval] = adapsimp(f,a,b,tol,fa,fb,fm)
%
%   [I4,nfeval] = adapsimp(f,a,b,fa,fb,tol)
%       Function che implementa la formula adattativa di Simpson
%
%   Input:
%       f: identificatore di una function che calcola la funzione
%       integranda
%       a: estremo sinistro dell'intervallo
%       b: estremo destro dell'intervallo
%       tol: tolleranza richiesta
%       fa: valutazione funzionale di f(a)
%       fb: valutazione funzionale di f(b)
%       fm: valutazione funzionale di f((a+b)/2)
%
%   Output:
%       I4: approssimazione dell'integrale
%       nfeval: numero di valutazioni funzionali utilizzate
%
if tol < 0, error('Tolleranza non corretta'), end
nfeval = 0;
if a == b
    I4 = 0;
else
    if nargin <= 4
        fa = feval(f,a);
        fb = feval(f,b);
        fm = feval(f,(a+b)/2);
        nfeval = 3;
    end
    h = (b-a)/6;
    x1 = (a + (a+b)/2) / 2;
    f1 = feval(f,x1);
    x2 = (b + (a+b)/2) / 2;
    f2 = feval(f,x2);
    nfeval = nfeval + 2;
    I2 = h*(fa + 4*fm + fb);
    I4 = h*(fa + 4*f1 + 2*fm + 4*f2 + fb)/2;
    e = abs(I4-I2)/15;
    if e > tol
        [I41,n1] = adapsimp(f,a,(a+b)/2,tol/2,fa,fm,f1);
        [I42,n2] = adapsimp(f,(a+b)/2,b,tol/2,fm,fb,f2);
        I4 = I41+I42;
        nfeval = nfeval + n1 + n2;
    end
end
return
```

Esercizio 30.

tol	Formula adattativa trapezi	Formula adattativa Simpson
10^{-2}	303	85
10^{-3}	991	185
10^{-4}	3119	341
10^{-5}	10123	625
10^{-6}	31837	1081