Elaborato esame Calcolo Numerico

Matlab Release: 23.2.0.2380103 (September 11, 2023)

Lorenzo Bartolini
7073016
lorenzo.bartolini8@edu.unifi.it

04/06/2024

Esecizio 1. Dimostrare che:

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

Per prima cosa sviluppiamo le singole funzioni con il polinomio di Taylor centrato nel punto x e otteniamo le seguenti:

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

Pertanto, sostituendo nell'equazione di partenza e svolgendo i calcoli otteniamo:

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

Esercizio 2. La funzione

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

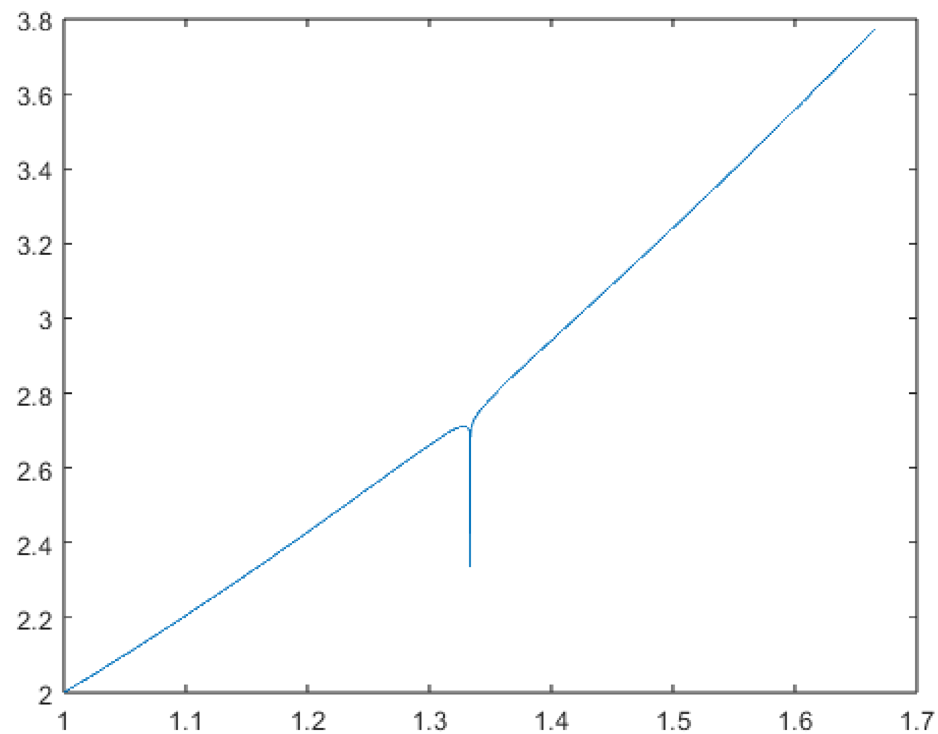
ha un asintoto in $x = 4/3$, in cui tende a $-\infty$. Graficarla in Matlab, utilizzando

$$\boldsymbol{x} = \texttt{linspace}(1, 5/3, 100001)$$

(in modo che il floating di $4/3$ sia contenuto in $\boldsymbol{x}$) e vedere dove si ottiene il minimo. Commentare i risultati ottenuti.

Listing 1: Codice Esercizio 2

```
x = linspace(1, 5/3, 100001);
y = f(x);

plot(x, y);

function y=f(x)
    y=1+x.^2+log(abs(3*(1-x)+1))/80;
end
```



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

La cancellazione numerica è la manifestazione del mal condizionamento della somma algebrica in caso di addendi di segno discorde.

Siano: $\tilde{x_1} = x_1(1+\epsilon_1)$, $\tilde{x_2} = x_2(1+\epsilon_2)$ gli addendi perturbati, allora $\tilde{y} = y(1+\epsilon_y)$ sarà il risultato perturbato.

Sia $y = x_1 + x_2$ il risultato esatto della somma.

Allora:
$\tilde{y} = \tilde{x_1}+\tilde{x_2} = x_1(1+\epsilon_1)+x_2(1+\epsilon_2) = (x_1+x_2)+x_1\epsilon_1+x_2\epsilon_2 = y+x_1\epsilon_1+x_2\epsilon_2 =$
$= y + y\epsilon_y = y(1+\epsilon_y)$
$\rightarrow |\epsilon_y| \leq \frac{|x_1|+|x_2|}{|x_1+x_2|} * max\{|\epsilon_1|, |\epsilon_2|\}$

Il numero di condizionamento della somma è $k = \frac{|x_1|+|x_2|}{|x_1+x_2|}$, che non è limitato superiormente se x_1 e x_2 sono quasi opposti. Il mal condizionamento si manifesta utilizzando un'aritmetica finita: partendo da addendi con cifre significative corrette si può ottenere un risultato con molte meno cifre significative corrette.
Si consideri il seguente esempio:
$x1 = 1.23456786$ e $x2 = -1.23456785$
$y = x1 + x2 = 0.00000001$
Procediamo considerando i dati di ingresso perturbati:
$\tilde{x1} = x1 + 10^{-7} = 1.23456787$
$\tilde{x2} = x2 - 10^{-7} = -1.23456786$
Quindi, il risultato perturbato sarà:
$\tilde{y} = \tilde{x1} + \tilde{x2} =$
Pertanto possiamo calcolare l'errore sui dati di uscita ottenendo:
$err_y = \frac{10^{-7}+10^{-7}}{0.00000001} = 2$
Come abbiamo già discusso in precedenza, osserviamo un errore sui dati di uscita molto maggiore di quello sui dati di ingresso.

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

Listing 2: Bisezione

```
function [x, i, flag] = bisezione(f, a, b, tolx)
%
%   x=bisezione(f,a,b,tolx) restituisce una approssimazione
    della
%       radice di f(x)=0 con il metodo di bisezione
%
%   Input:
%        f - identificatore della function della funzione
%      a,b - estremi dell'intervallo
%     tolx - tolleranza accettata
%
%   Output:
%         x - approssimazione dello zero della funzione
```

```
%          i - numero di iterazioni necessarie
%       flag - vale 1 se l'errore e' minore della tolleranza

flag = 0;
fa = feval(f, a);
fb = feval(f, b);

if fa*fb > 0
    error("Gli estremi devono essere di segno opposto");
end

n_max = ceil(log2(b-a)-log2(tolx));

for i=1:n_max
    x = (a+b)/2;
    fx = feval(f, x);

    if (abs(fx)*abs(b-a))/abs(fb-fa) <= tolx
        flag = 1;
        return
    end

    if fa*fx<0
        b=x;
        fb=fx;
    else
        a=x;
        fa=fx;
    end
end
end
```

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

Listing 3: Newton

```
function [x, i] = newton(fun, deriv, x0, tolx, maxiter)
%
% [x, i] = newton( fun, deriv, x0, tolx , maxiter )
%
% Metodo di Newton per determinare una approssimazione della
    radice di f(x)=0
%   Input:
%        fun - funzione in input
%        deriv - derivata della funzione fun in input
%        x0 - punto iniziale
%       tolx - tolleranza
```

```
%     maxiter - numero massimo di iterazioni
%
%  Output:
%         x - approssimazione dello zero della funzione
%      i - valore che indica il numero di iterazioni richieste per
%      trovare lo zero; vale -1 se la derivata si annulla o se la
%      tolleranza non    soddisfatta entro maxit iterazioni
%

x=x0;
j = -1;
for i=1:maxiter
    f1x = feval(deriv, x0);
    if f1x == 0
        break
    end
    x = x0 - feval(fun, x0)/f1x;
    err = abs(x-x0);
    if err <= tolx
        j=i;
        break
    end
    x0 = x;
end

i=j;
return;

end
```

Listing 4: Secanti

```
function [x, i] = secanti(fun, x0, x1, tolx, maxiter)
%
%   [x,i]=secanti(fun,x0,x1,tol,itmax) restituisce una approssimazione
%       dello zero della funzione con il metodo delle secanti
%   Input:
%           fun - funzione
%         x0,x1 - punti iniziali
%           tol - tolleranza
%         itmax - numero massimo di iterazioni
%   Output:
%           x - approssimazione dello zero della funzione
%           i - numero iterazioni necessarie

fx0 = feval(fun, x0);
```

```
fx1 = feval(fun, x1);
for i=1:maxiter
    x = x1 - ((x0-x1)*fx1)/(fx0-fx1);
    err = abs(x-x1);
    if abs(err)<= tolx
        return
    end
    x0=x1;
    fx0=fx1;
    x1=x;
    fx1=feval(fun, x1);
end

return;
end
```

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

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

per $tol = 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$, partendo da $x_0 = 1$ (e $x_1 = 0.9$ per il metodo delle secanti). Per il metodo di bisezione, usare l'intervallo di confidenza iniziale $[-0.1, 1]$. Tabulare i risultati, in modo da confrontare il costo computazionale di ciascun metodo.

Listing 5: Codice esercizio 6

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

tolx = 1e-3;
disp("1e-3");
[x, i] = bisezione(f, -0.1, 1, tolx);
disp("Bisezione " + x + " Iterazioni: " + i);
[x, i] = newton(f, df, 1, tolx, 1000);
disp("Newton " + x + " Iterazioni: " + i);
[x, i] = secanti(f, 1, 0.9, tolx, 1000);
disp("Secanti " + x + " Iterazioni: " + i);

tolx = 1e-6;
disp("1e-6");
[x, i] = bisezione(f, -0.1, 1, tolx);
disp("Bisezione " + x + " Iterazioni: " + i);
[x, i] = newton(f, df, 1, tolx, 1000);
disp("Newton " + x + " Iterazioni: " + i);
[x, i] = secanti(f, 1, 0.9, tolx, 1000);
disp("Secanti " + x + " Iterazioni: " + i);

tolx = 1e-9;
disp("1e-9");
```

```
[x, i] = bisezione(f, -0.1, 1, tolx);
disp("Bisezione " + x + " Iterazioni: " + i);
[x, i] = newton(f, df, 1, tolx, 1000);
disp("Newton " + x + " Iterazioni: " + i);
[x, i] = secanti(f, 1, 0.9, tolx, 1000);
disp("Secanti " + x + " Iterazioni: " + i);

tolx = 1e-12;
disp("1e-12");
[x, i] = bisezione(f, -0.1, 1, tolx);
disp("Bisezione " + x + " Iterazioni: " + i);
[x, i] = newton(f, df, 1, tolx, 1000);
disp("Newton " + x + " Iterazioni: " + i);
[x, i] = secanti(f, 1, 0.9, tolx, 1000);
disp("Secanti " + x + " Iterazioni: " + i);
```

Risultati:

$tol = 10^{-}3$

Metodo	Errore di approssimazione	Iterazioni
Bisezione	0.00097656	9
Newton	2.8423e-09	5
Secanti	1.1522e-06	6

$tol = 10^{-}6$

Metodo	Errore di approssimazione	Iterazioni
Bisezione	9.5367e-07	19
Newton	3.5748e-17	6
Secanti	3.8242e-16	8

$tol = 10^{-}9$

Metodo	Errore di approssimazione	Iterazioni
Bisezione	9.3132e-10	29
Newton	3.5748e-17	7
Secanti	3.8242e-16	8

$tol = 10^{-}12$

Metodo	Errore di approssimazione	Iterazioni
Bisezione	9.0949e-13	39
Newton	3.5748e-17	7
Secanti	-6.1673e-17	9

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

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

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

Listing 6: Newton modificato

```
function [x, i] = newtonModificato(fun, deriv, x0, molt, tolx, maxiter)
%
% [x, i] = newton( fun, deriv, x0, molt, tolx , maxiter )
%
% Metodo di Newton per determinare una approssimazione della radice di f(x)=0
%   Input:
%        fun - funzione in input
%        deriv - derivata della funzione fun in input
%        x0 - punto iniziale
%       molt - molteplicita' della radice
%      tolx - tolleranza
%     maxiter - numero massimo di iterazioni
%
%  Output:
%         x - approssimazione dello zero della funzione
%      i - valore che indica il numero di iterazioni richieste per
%      trovare lo zero; vale -1 se la derivata si annulla o se la
%      tolleranza non    soddisfatta entro maxit iterazioni
%
    for i=1:maxiter
        deriv_x0 = feval(deriv,x0);

         if deriv_x0==0
            break
         end

        x = x0 - molt*(feval(fun, x0)/deriv_x0);
        err = abs(x-x0);
        if err <= tolx
            return
        end
        x0 = x;
    end
return
```

Listing 7: Codice esercizio 7

```
f = @(x) exp(x) - cos(x) + sin(x) - x*(x+2);
deriv = @(x) exp(x) + sin(x) + cos(x) - 2*x - 2;
molt = 5;
maxiter = 1000;

tolx = 1e-3;
disp("1e-3");
[x, i] = bisezione(f, -0.1, 1, tolx);
disp("Bisezione " + x + " Iterazioni: " + i);
[x, i] = newton(f, deriv, 1, tolx, maxiter);
disp("Newton " + x + " Iterazioni: " + i);
[x, i] = newtonModificato(f, deriv, 1, molt, tolx, maxiter);
disp("Newton modificato " + x + " Iterazioni: " + i);
[x, i] = secanti(f, 1, 0.9, tolx, maxiter);
disp("Secanti " + x + " Iterazioni: " + i);

tolx = 1e-6;
disp("1e-6");
[x, i] = bisezione(f, -0.1, 1, tolx);
disp("Bisezione " + x + " Iterazioni: " + i);
[x, i] = newton(f, deriv, 1, tolx, maxiter);
disp("Newton " + x + " Iterazioni: " + i);
[x, i] = newtonModificato(f, deriv, 1, molt, tolx, maxiter);
disp("Newton modificato " + x + " Iterazioni: " + i);
[x, i] = secanti(f, 1, 0.9, tolx, maxiter);
disp("Secanti " + x + " Iterazioni: " + i);

tolx = 1e-9;
disp("1e-9");
[x, i] = bisezione(f, -0.1, 1, tolx);
disp("Bisezione " + x + " Iterazioni: " + i);
[x, i] = newton(f, deriv, 1, tolx, maxiter);
disp("Newton " + x + " Iterazioni: " + i);
[x, i] = newtonModificato(f, deriv, 1, molt, tolx, maxiter);
disp("Newton modificato " + x + " Iterazioni: " + i);
[x, i] = secanti(f, 1, 0.9, tolx, maxiter);
disp("Secanti " + x + " Iterazioni: " + i);

tolx = 1e-12;
disp("1e-12");
[x, i] = bisezione(f, -0.1, 1, tolx);
disp("Bisezione " + x + " Iterazioni: " + i);
[x, i] = newton(f, deriv, 1, tolx, maxiter);
disp("Newton " + x + " Iterazioni: " + i);
[x, i] = newtonModificato(f, deriv, 1, molt, tolx, maxiter);
disp("Newton modificato " + x + " Iterazioni: " + i);
[x, i] = secanti(f, 1, 0.9, tolx, maxiter);
disp("Secanti " + x + " Iterazioni: " + i);
```

Risultati:

$tol = 10^-3$

Metodo	Errore di approssimazione	Iterazioni
Bisezione	0.0375	3
Newton	0.0039218	25
Newton Modificato	2.6016e-05	3
Secanti	0.005577	33

$tol = 10^-6$

Metodo	Errore di approssimazione	Iterazioni
Bisezione	0.003125	5
Newton	-8.0477e-05	-1
Newton Modificato	2.6016e-05	3
Secanti	0.0013347	48

$tol = 10^-9$

Metodo	Errore di approssimazione	Iterazioni
Bisezione	0.0011163	31
Newton	-8.0477e-05	-1
Newton Modificato	2.6016e-05	3
Secanti	0.0014114	57

$tol = 10^-12$

Metodo	Errore di approssimazione	Iterazioni
Bisezione	0.0011163	32
Newton	-8.0477e-05	-1
Newton Modificato	2.6016e-05	3
Secanti	0.0014114	57

Si noti come il metodo di Newton nelle ultime tre esecuzioni, ovvero con tolleranza sempre minore, si blocchi a causa della derivata prima che si azzera. Inoltre è interessante osservare l'efficienza di questi metodi; In particolare il metodo di bisezione e quello delle secanti sono i peggiori, invece il metodo di Newton modificato è in grado di raggiungere il risultato ottimale in pochissime iterazioni.

Esercizio 9. Scrivere una *function* Matlab,

`function x = mialdl(A,b)`

che, dati in ingresso una matrice sdp A ed un vettore $\boldsymbol{b}$, calcoli la soluzione del corrispondente sistema lineare utilizzando la fattorizzazione $LDL^\top$. Curare particolarmente la scrittura e l'efficienza della *function*, e validarla su un congruo numero di esempi significativi, che evidenzino tutti i suoi possibili *output*.

Listing 8: mialdl

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

%Controlli di consistenza
[m, n] = size(A);
if m ~= n
    error('Matrice non quadrata');
end
if length(b) ~= n
    error('Dimensione vettore termini noti non corretta');
end
if A(1,1) <= 0
    error('Matrice non sdp');
end
% Fattorizzazione LDLt di A
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('Matrice non sdp');
    end
    A(j+1:n, j) = (A(j+1:n, j) - A(j+1:n, 1:j-1) * v) / A(j, j);
end
% risoluzione sistema Ax = b
% Controllo che tutti i valori sulla diagonale siano positivi
d = diag(A);
```

```
if ~all(d > 0)
    error('Matrice non sdp');
end
x = b(:);
for i = 2:n % Lx1 = b
    x(i:n) = x(i:n) - A(i:n, i-1) * x(i-1);
end
x = x./d; % Dx2  = x1
for i = n:-1:2   % Ltx = x2
    x(1:i-1) = x(1:i-1)  - A(i, 1:i-1)' * x(i);
end

return;
end
```

$$\text{A} = \begin{bmatrix} 4 & 2 & 4 \\ 2 & 10 & 5 \\ 4 & 5 & 6 \end{bmatrix} \text{ b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \rightarrow x = \begin{bmatrix} -1.0833 \\ -0.3333 \\ 1.5000 \end{bmatrix}$$

$$\text{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \text{ b} = \begin{bmatrix} 4 \\ 2 \end{bmatrix} \rightarrow \text{Errore: Matrice non sdp}$$

Esercizio 10. Scrivere una *function* Matlab,

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

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

Listing 9: miaqr

```
function [x, nr] = miaqr(A, b)
%
%   [x,nr] = miaqr(A,b)
%
% Esegue la fattorizzazione QR di A e restituisce la soluzione ai minimi
% quadrati del sistema lineare e la norma del corrispondente vettore
% residuo
%
% Input:
%   A: matrice m x n
%   b: vettore dei termini noti
% Output:
%   x: soluzione del sistema Ax = b
%   nr: norma vettore residuo
%
```

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

    b(i:m) = b(i:m) - (beta * [1 A(i+1:m, i)'] * b(i:m)) * [1; A(i+1:m, i)];
end
% Risoluzione sistema Ax=b
x = b(:);
for i = n:-1:1
    x(i) = x(i) / A(i,i);
    x(1:i-1) = x(1:i-1)   - A(1:i-1, i) * x(i);
end
% Norma del vettore residuo
nr = norm(x(n+1:m));

x = x(1:n);

return;
end
```

$$\text{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad \text{b} = \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} \rightarrow x = \begin{bmatrix} -6.0000 \\ 6.5000 \end{bmatrix}$$

$$\text{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \quad \text{b} = \begin{bmatrix} 4 \\ 9 \end{bmatrix} \rightarrow \text{Errore: Matrice non a rango massimo}$$

Esercizio 11. Risolvere i sistemi lineari, di dimensione n,

$$A_n \boldsymbol{x}_n = \boldsymbol{b}_n, \qquad n = 1, \ldots, 15,$$

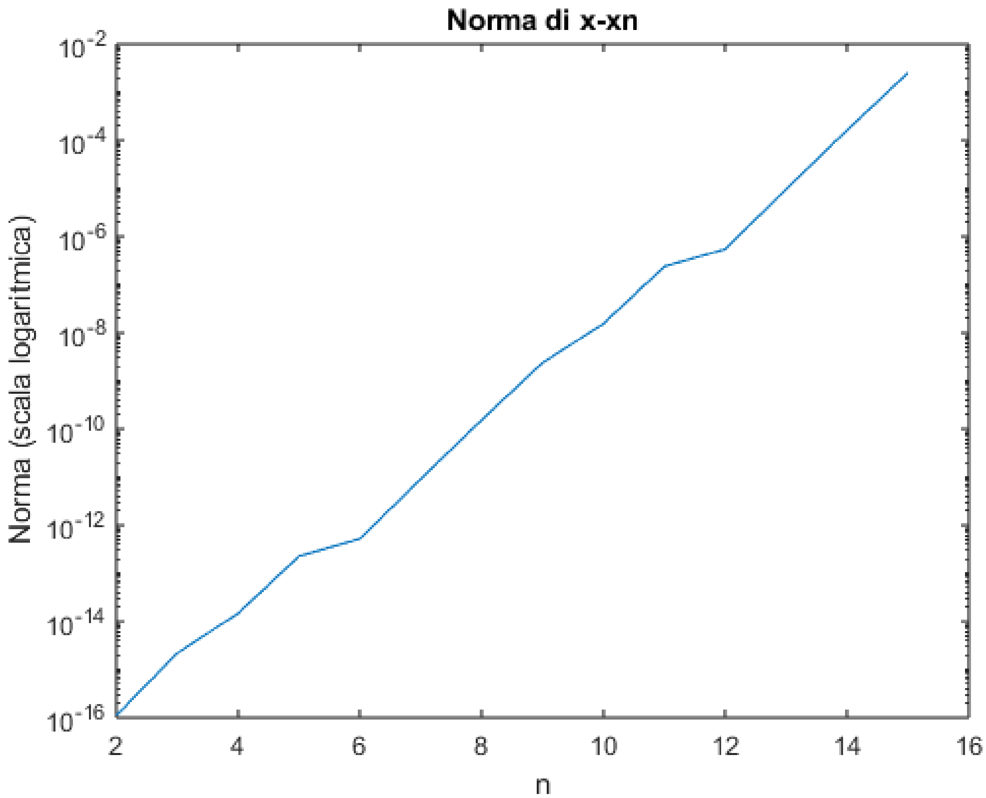
in cui

$$A_n = \begin{pmatrix} 1 & 1 & \ldots & \ldots & 1 \\ 10 & \ddots & \ddots & & \vdots \\ 10^2 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 & 1 \\ 10^{n-1} & \ldots & 10^2 & 10 & 1 \end{pmatrix} \in \mathbb{R}^{n \times n}, \qquad \boldsymbol{b}_n = \begin{pmatrix} n-1+\frac{10^1-1}{9} \\ n-2+\frac{10^2-1}{9} \\ n-3+\frac{10^3-1}{9} \\ \vdots \\ 0+\frac{10^n-1}{9} \end{pmatrix} \in \mathbb{R}^n,$$

la cui soluzione è il vettore $\boldsymbol{x}_n = (1, \ldots, 1)^\top \in \mathbb{R}^n$, utilizzando la *function* `mialu`. Tabulare e commentare l'accuratezza dei risultati ottenuti, dandone spiegazione esaustiva.

Listing 10: Codice esercizio 11

```
norms = (1:15);
for n=1:15
    % soluzione reale
    xn = ones(1, n).';

    An = ones(n);
    for i=1:n
        v = [ones(1, i) 10.^(1:n-i)].';
        An(:, i) = v;
    end
    bn = ones(1, n).*n - (1:n) + (10.^(1:n)-1)/9;
    bn = bn.';

    x = mialu(An, bn);
    norms(i) = norm(x-xn);
end

semilogy((1:15), norms);
title("Norma di x-xn");
xlabel("n");
ylabel("Norma (scala logaritmica)");
```



Si noti che all'aumentare di n l'errore commesso rispetto alla soluzione reale cresce esponenzialmente a causa del fenomeno della cancellazione numerica e a causa degli errori di arrotondamento dato che il calcolatore lavora in aritmetica finita.

Esercizio 12. Fattorizzare, utilizzando la *function* `mialdlt`, le matrici sdp

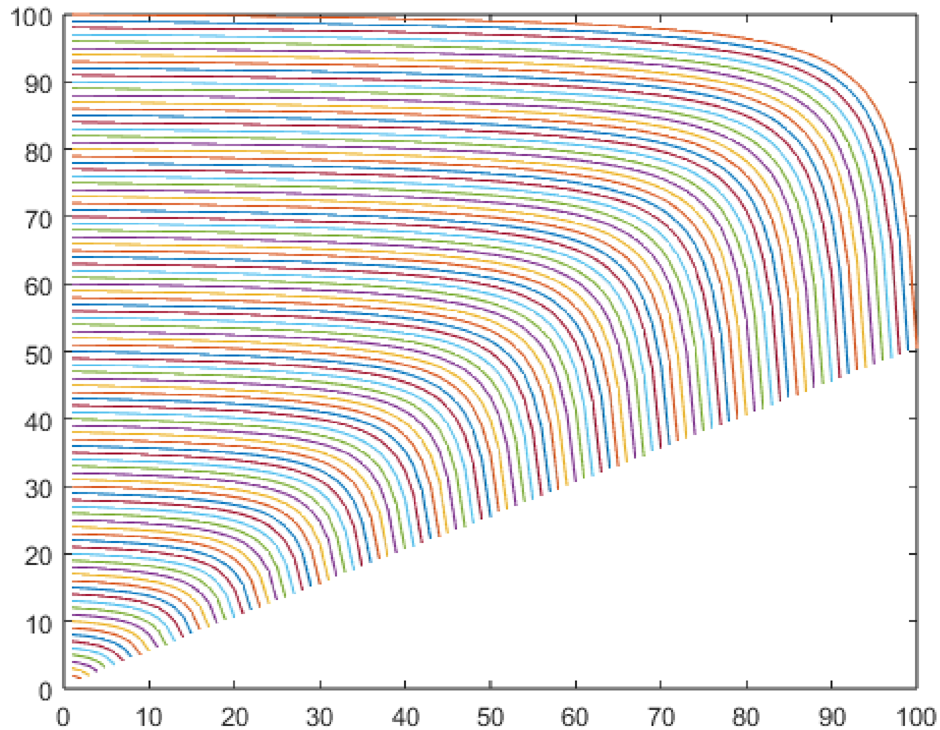
$$A_n = \begin{pmatrix} n & -1 & \dots & -1 \\ -1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -1 \\ -1 & \dots & -1 & n \end{pmatrix} \in \mathbb{R}^{n\times n}, \qquad n = 1, \dots, 100.$$

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

Listing 11: Codice esercizio 12

```
N=100;

for n=1:N
```

```
    An = ones(n).*-1;
    An = An + diag(ones(1, n)*n + 1);

    LDLt = mialdlt(An);

    plot((1:n), diag(LDLt));
    hold on
end
hold off
```



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

$$A\boldsymbol{x} = \boldsymbol{b},$$

in cui

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

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

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

con

$$\omega_1 = \omega_2 = 0.5, \quad \omega_3 = .75, \quad \omega_4 = \omega_5 = 0.25.$$

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

Listing 12: Calcolo della norma pesata

```
% Norma del vettore residuo
w = [0.5 0.5 0.75 0.25 0.25].';
r = A*x - b;
nr = sqrt(sum(w.*(r.^2)));
```

E' stata usata la funzione del precedente esercizio per risolvere il sistema dato.
La funzione è stata modificata affinchè calcolasse la norma indicata.
Risultati:

Risultato x	Norma
[0.1645, -0.1326, 0.3392]	1.4815

Esercizio 14. Scrivere una *function* Matlab,

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

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

Listing 13: Newton

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

```
%   [x, nit] = newton(fun, x0, tol, maxit)
%
% Metodo di newton per la risoluzione di sistemi di equazioni non lineari
%
% Input:
%   fun: forma [f, jacobian] = fun(x) se il sistema da risolvere    f(x)=0
%   x0: vettore valori iniziali
%   tol: tolleranza
%   maxit: numero massimo di iterazioni
%
% Output:
%   x: soluzione del sistema
%   nit: numero di iterazioni eseguite
%
% Criterio d'arresto: |Xn+1 - Xn| <= tol * (1 + |Xn|)

% Controlli di consistenza
if tol <= 0
    error('Tolleranza non valida');
end
if maxit <= 0
    error('Numero di iterazioni non valido');
end
%Valori di default per i parametri in ingresso
if  nargin == 3
    tol = 1e-3;
    maxit = 1000;
else if nargin == 4
        maxit = 1000;
    end
end
x = x0;
for i=1:maxit
    x0 = x;
    f, jacobian = feval(fun, x0);

    b = -f;
    A = jacobian;

    x = x0 + mialu(A, b); % Fattorizzazione e aggiornamento di xn+1

    % Controllo sul criterio di arresto
    if norm(x - x0, 1) <= tol * (1 + norm(x0, 1))
        break;
    end
end
nit = i;
```

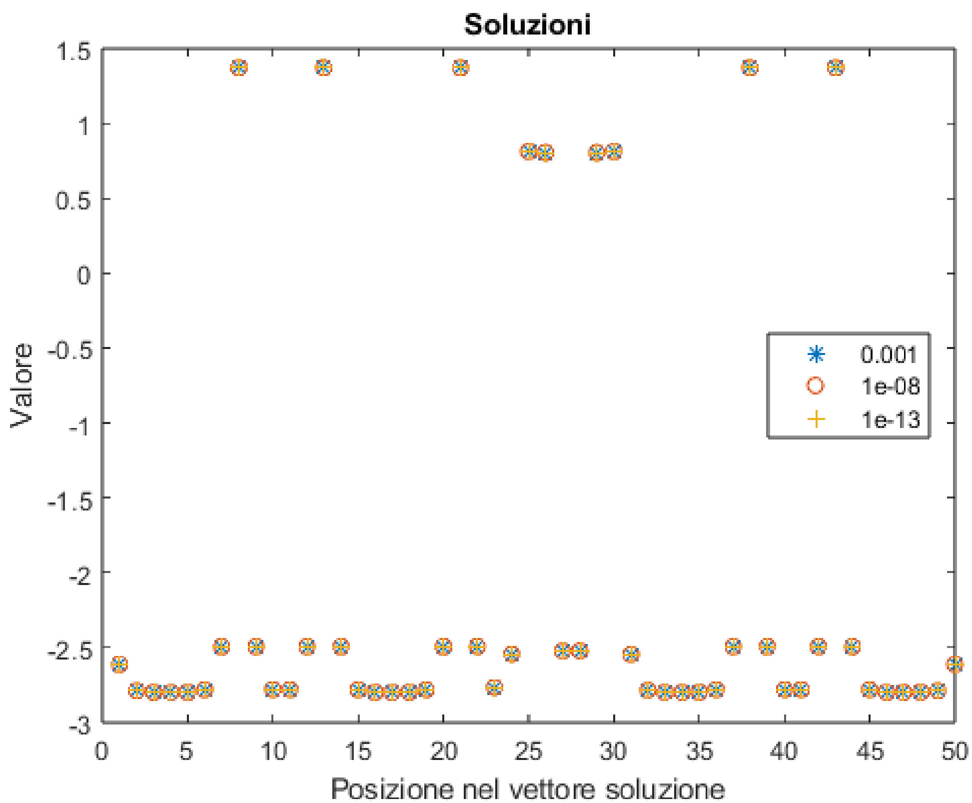
```
if norm(x - x0, 1) > tol * (1 + norm(x0, 1))
    disp('Tolleranza non raggiunta');
end
return
end
```

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:

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

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

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



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

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

Listing 14: Lagrange

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

```
%
% Input:
%   (X,Y): dati del problema
%   XQ: vettore in cui calcolare il polinomio
%
% Output:
%   YQ: polinomio interpolante in forma di Lagrange

n = length(X);
if length(Y) ~= n || n <= 0
    error('Dati inconsistenti');
end
%Controllo che le componenti del vettore X siano distinte
if  length(unique(X)) ~= n
    error('Le ascisse non sono distinte');
end

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

Listing 15: Lin

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

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

```
end
```

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

Listing 16: Newton

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

if length(X) ~= length(Y) ||   length(X) <= 0
    error('Dati errati');
end
%Controllo che le componenti del vettore X siano distinte
if length(unique(X)) ~= length(X)
    error('Le ascisse non sono distinte');
end

df = diffdiv(X, Y);
n = length(df) - 1;
YQ = df(n+1) * ones(size(XQ));
for i = n:-1:1
    YQ = YQ.*(XQ - X(i   )) + df(i);
end

return;
end
```

Listing 17: Differenze divise

```
function df = diffdiv(x, f)
%
%   df = divdif(x, f)
%
% Calcola le differenze divise sulle coppie (xi, fi)
```

```
%
% Input:
%   x: vettore delle ascisse
%   f: vettore delle ordinate
% Output:
%   df: vettore delle differenze divise

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

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

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

che implementi, in modo <u>vettoriale</u>, il polinomio interpolante di Hermite.

Listing 18: Hermite

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

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

```

%Controllo che le componenti del vettore xi siano distinte
if length(unique(xi)) ~= length(xi)
    error('Le ascisse non sono distinte');
end

%Vettore con valori di f e derivata prima di f: [f(0) f'(0) f(1) f'(1)...]
fi = repelem(fi, 2);
for i = 1:length(f1i)
    fi(i*2) = f1i(i);
end
df = diffdiv(xi, fi);
n = length(df)-1;
yy = df(n+1) * ones(size(xx));
for i = n:-1:1
    yy = yy.*(xx - xi(round(i/2))) + df(i);
end
return;
end
```

Esercizio 19. Si consideri la seguente base di Newton,

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

con $x_0, \dots, x_n$ ascisse date (non necessariamente distinte tra loro), ed un polinomio rappresentato rispetto a tale base,

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

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

Listing 19: Horner per calcolo della Derivata

```
function dy = hornerDerivata(x, ai, xi)
%
% dy = hornerDerivata(x, ai, xi)
%      Calcola la derivata prima nel punto x
%       del polinomio p(x) dove ai sono i coefficienti della base di Newton
%
% Input:
%   x: Ascissa su cui valutare la derivata
%   ai: Coefficienti del pominomio
%   xi: Ascisse su cui valutare la base di Newton
%
% Output:
```

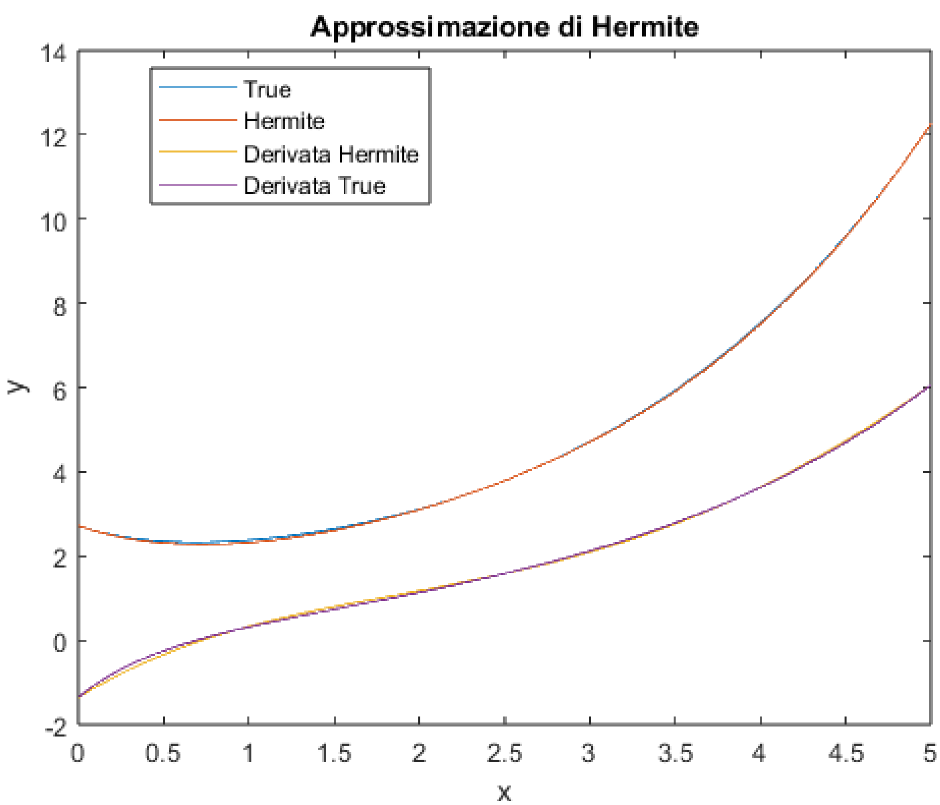
```
%   dy: Derivata prima calcolata nel punto x

n = length(ai);
if n ~= length(xi)-1
    error('Dimensione degli input non consistente');
end

dy = 0;
b = ai(n);

for k = n-1:-1:1
    dy = b + (x - xi(k)) * dy;
    b = ai(k) + (x - xi(k)) * b;
end

return;
end
```

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

Listing 20: Codice esercizio 20

```
fun = @(x) (exp(x/2 + exp(-x)));
funPrime = @(x) (.5*exp(exp(-x) - x/2).*(-2 + exp(x)));

xi = [0 2.5 5];
fi = fun(xi);
f1i = funPrime(xi);

x = linspace(0, 5, 1000);

yHermite = hermite(xi, fi, f1i, x);
yTrue = fun(x);

plot(x, yTrue, "DisplayName", "True");
hold on

plot(x, yHermite, "DisplayName", "Hermite");
hold on

dfTrue = funPrime(x);

% Calcolo dei vettori raddoppiati
xiRaddoppiato = repelem(xi, 2);
```

```
fi = repelem(fi, 2);
for i = 1:length(f1i)
    fi(i*2) = f1i(i);
end
dd = ddHermite(xi, fi);

plot(x, hornerDerivata(x, dd, xiRaddoppiato), "DisplayName",
    "Derivata Hermite");
hold on

plot(x, dfTrue, "DisplayName", "Derivata True");
hold off

title("Approssimazione di Hermite");
xlabel("x");
ylabel("y");
legend("Location", "Best");
```



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

Listing 21: Chebyshev

```
function x = chebyshev(n, a, b)
%
% x = chebyshev(n, a, b)
%
% Calcola le n+1 ascisse di Chebyshev sull'intervallo [a, b]
%
% Input:
% n: numero di ascisse che vogliamo calcolare
% [a, b]: intervallo in cui vengono calcolate le ascisse di
    Chebyshev
%
% Output:
% x: ascisse di Chebyshev calcolate sull'intervallo [a, b]
%
if a >= b || n <= 0
    error('Dati errati');
end
x = (a+b)/2 + ((b-a)/2) * cos((2*[n:-1:0] + 1)/((2*(n+1)))*
    pi);

return;
end
```

Esercizio 22. Costruire una function Matlab, con sintassi

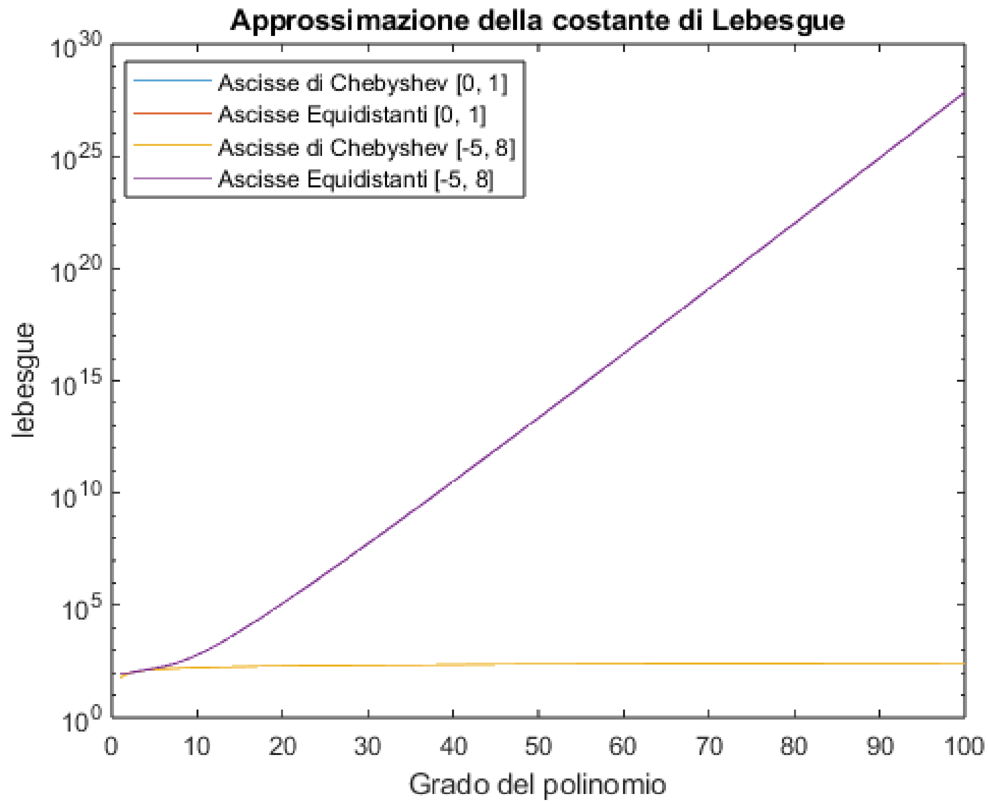
`ll = lebesgue( a, b, nn, type ),`

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

Listing 22: Lebesgue

```
function ll = lebesgue(a,b, nn, type)
%
% ll = lebesgue(a, b, nn, type) restituisce le
    approssimazioni della
% costante di Lebesgue sull'intervallo [a, b] per i polinomi
     di grado
% specificato dal vettore nn
%  Input:
%    a, b: intervallo
```

```
%    nn: vettore contenente i gradi dei polinomi per l'approssimazione
%    type: ascisse equidistanti=0, chebyshev=1
%
%  Output:
%    ll: approssimazioni della costante di Lebesgue
%
max=10001;
x=linspace(a, b, max);
ll = nn;
for i=1:length(nn)
    if type == 0
        xi = linspace(a, b, nn(i));
    elseif type == 1
        xi = chebyshev(nn(i), a, b);
    end

    leb = zeros(1, max);
    for j=1:nn(i)
        leb = leb + abs(lin(x, xi, j));
    end

    ll(i) = norm(leb);
end
return;
end
```



Dal grafico in figura possiamo osservare la crescita ottimale della costante di Lebesgue utilizzando le ascisse di Chebyshev al contrario di quelle equidistanti. Inoltre osserviamo che l'intervallo non influisce, infatti le linee per i due intervalli si sovrappongono perfettamente.

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

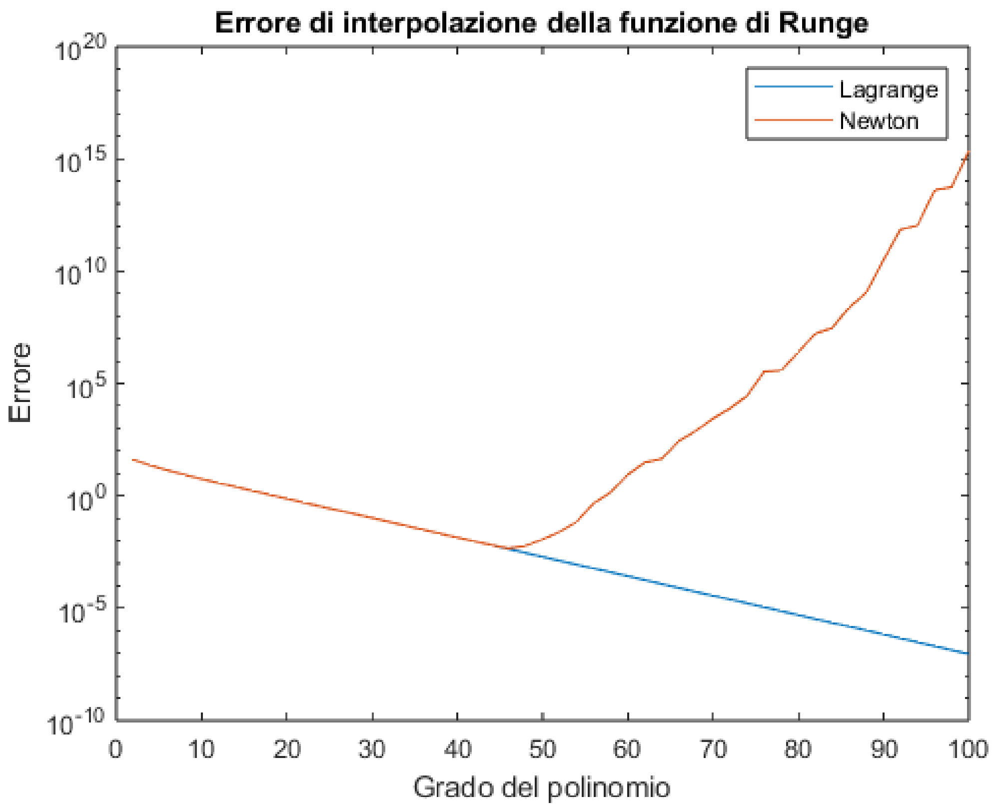
$$f(x) = \frac{1}{1+x^2}, \qquad x \in [-5, 5],$$

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

Listing 23: Codice esercizio 23

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

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

```
normLagrange = (1:50);
normNewton = (1:50);

for n=1:50
    xCheby = chebyshev(2*n, -5, 5);
    yCheby = f(xCheby);

    yLagrange = lagrange(xCheby, yCheby, x);
    yNewton = newton(xCheby, yCheby, x);

    normLagrange(n) = norm(y-yLagrange);
    normNewton(n) = norm(y-yNewton);
end

semilogy((2:2:100), normLagrange,'DisplayName','Lagrange');
hold on
semilogy((2:2:100), normNewton,'DisplayName','Newton');
hold off
title("Errore di interpolazione della funzione di Runge");
xlabel("Grado del polinomio");
ylabel("Errore");
legend
```



Possiamo osservare che l'errore cresce per n grande nel caso si utilizzi il polinomio di Newton. Questo avviene a causa della propagazione dell'errore nel calcolo incrementale del polinomio al contrario del caso di Lagrange.

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

Listing 24: Spline Cubica Naturale

```
function yy = spline0(x, y, xx)
%
%  yy = spline0(x, y, xx)
%
%  Calcola la spline cubica naturale interpolante e
%  restituisce il valore assunto dalla spline sulle ascisse xx
%
% Input:
%   x - vettore delle ascisse di interpolazione
```

```
%   y - vettore dei valori della funzione assunti sulle ascisse
%       interpolanti
%   xx - vettore delle ascisse dove si calcola il valore della spline
%
% Output:
%   yy - vettore delle ordinate calcolate sulle ascisse
%

n = length(x);

% Controlli di consistenza
if   length(y) ~= n
    error('Dati errati');
end

n = n-1;
h(1:n) = x(2:n+1) - x(1:n);
b = h(2:n-1)./(h(2:n-1) + h(3:n)); % phi
c = h(2:n-1)./(h(1:n-2) + h(2:n-1)); % csi
a(1:n-1) = 2;
df = ddspline(x, y, 3);

m = tridia(a, b, c, 6*df); % risoluzione del sistema tridiagonale
m = [0, m, 0];

yy = zeros(size(xx));

j = 1;
for i=2:n+1
    ri  = y(i-1) - (h(i-1)^2)/6 * (m(i-1));
    qi = (y(i) - y(i-1))/h(i-1) - h(i-1)/6*(m(i) - m(i-1));
    while j <= length(xx) && xx(j) <= x(i)
        yy(j) = ((xx(j) - x(i-1))^3 * m(i) + (x(i) - xx(j))^3 * m(i-1))/ ...
             (6*h(i-1)) + qi*(xx(j) - x(i-1)) + ri;
        j = j+1;
    end
end

return;
end
```

Listing 25: Tridia

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

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

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

return;
end
```

Listing 26: Differenze Divise

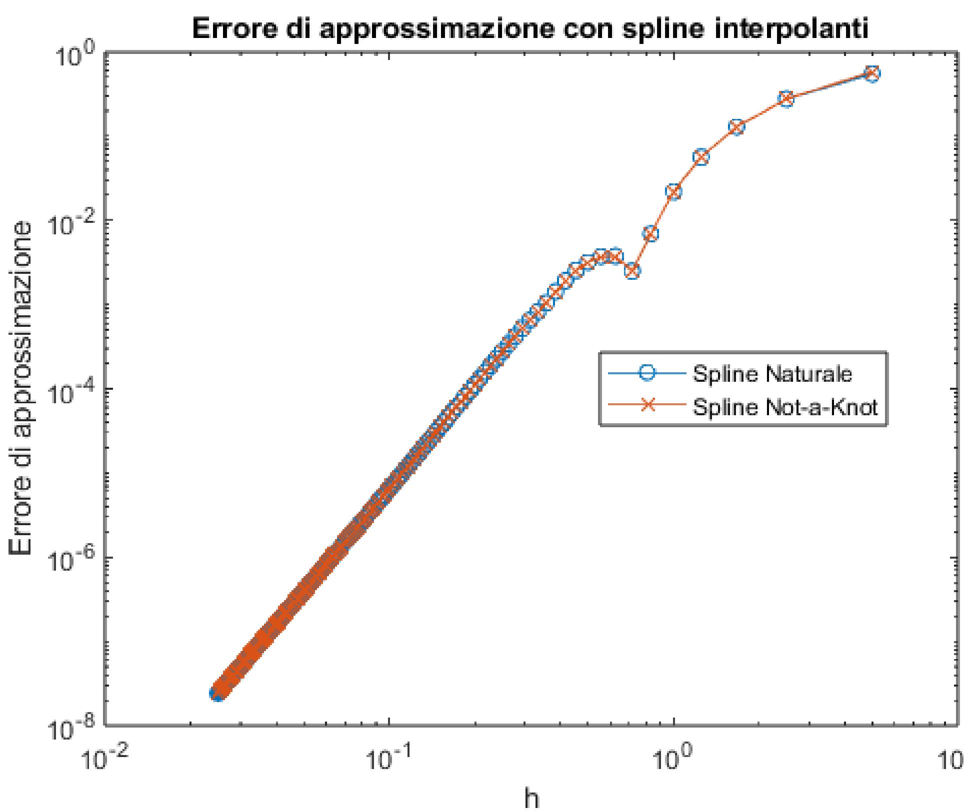
```
function df = ddspline(x, y, it)
%
%   df = ddspline(x, y, it)
%
% Calcola le differenze divise sulle coppie (xi, fi)
% terminando alla it-esima iterazione
%
% Input:
%   x - vettore delle ascisse
%   y - vettore delle ordinate
% Output:
%   df - vettore delle differenze divise
%

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

```
df = df(1, it:n+1);

return;
end
```

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

$$\Delta = \left\{ x_i = -10 + i\frac{20}{n},\ i = 0, \ldots, n \right\}, \qquad n = \texttt{4 : 4 : 800},$$

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

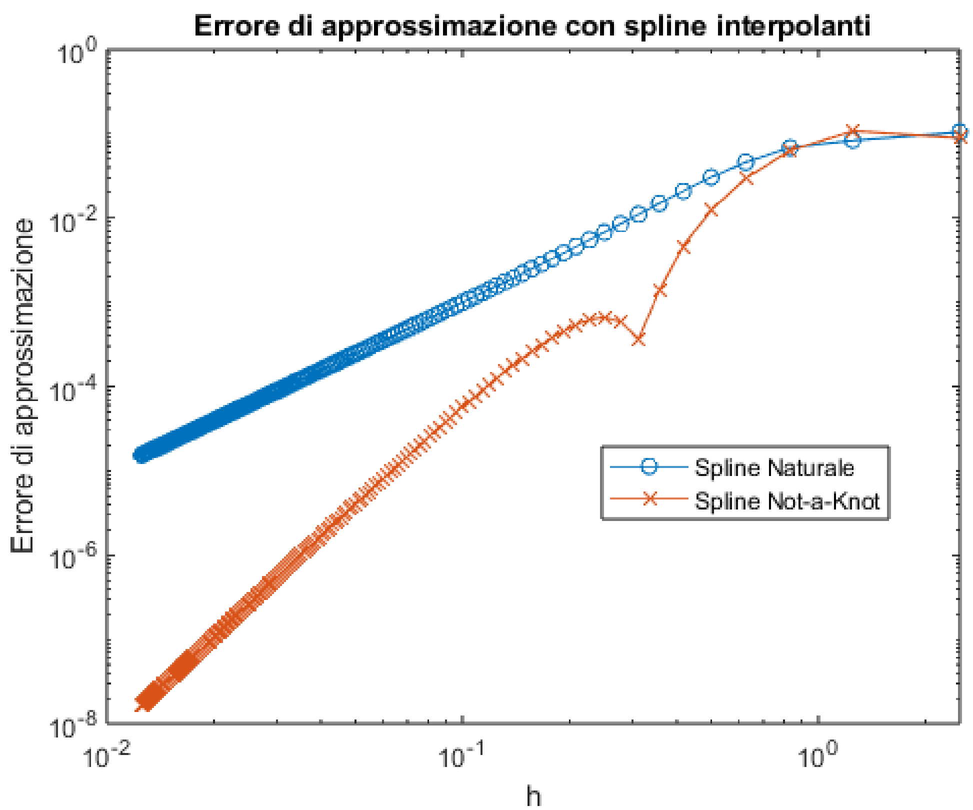


Possiamo osservare una decrescita esponenziale. Si noti che sulle ascisse è rappresentato il valore $h = 20/n$.

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

$$\Delta = \left\{ x_i = i\frac{20}{n},\ i = 0, \dots, n \right\}, \qquad n = \mathtt{4:4:800},$$

rispetto alla distanza $h = 10/n$ tra le ascisse. Utilizzare 10001 punti equispaziati nell'intervallo $[0, 10]$ per ottenere la stima dell'errore. Che tipo di decrescita si osserva? Confrontare e discutere i risultati ottenuti, rispetto a quelli del precedente esercizio.



Possiamo osservare in entrambi i casi una decrescita esponenziale. Al contrario rispetto all'esercizio precedente però le due Spline decrescono diversamente, la Spline Naturale infatti commette un errore maggiore rispetto a quella Not-a-Knot.

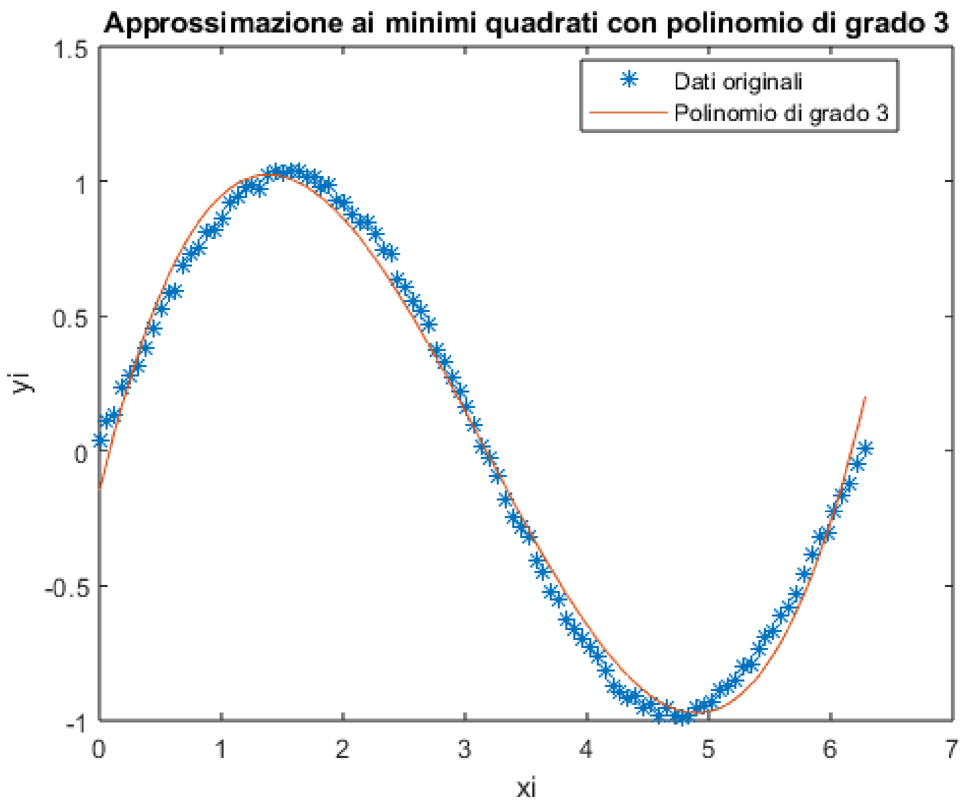
Esercizio 27. Calcolare i coefficienti del polinomio di approssimazione ai minimi quadrati di grado 3 per i seguenti dati:

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

Graficare convenientemente i risultati ottenuti.

Listing 27: Codice esercizio 27

```
rng(0);

xi = linspace(0, 2*pi, 101);
yi = sin(xi) + rand(size(xi)) * 0.05;

n = length(xi);

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

a = V \ yi';
yi_fit = polyval(a, xi);

figure;
plot(xi, yi, '*', 'DisplayName', 'Dati originali');
hold on;
plot(xi, yi_fit, '-', 'DisplayName', 'Polinomio di grado 3');
legend("Location", "Best");
xlabel('xi');
ylabel('yi');
title('Approssimazione ai minimi quadrati con polinomio di grado 3');
```



Esercizio 28. 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).

Listing 28: Pesi Newton-Cotes

```
function c = pesiNewtonCotes(n)
%
% c = pesiNewtonCotes(n)
%
% Function che restituisce i pesi della quadratura
% della formula di Newton-Cotes di grado n
%
% Input:
% n- Grado della formula
%
% Output:
% c- Pesi della quadratura
%
```

```
% Controlli di consistenza
if n < 1 || n > 9 || n == 8
    error("Input errato");
end

c = zeros(1, n+1);

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

return;
end
```

Grado	Pesi
1	$\frac{1}{2}, \frac{1}{2}$
2	$\frac{1}{3}, \frac{4}{3} \frac{1}{3}$
3	$\frac{3}{8}, \frac{9}{8} \frac{9}{8}, \frac{3}{8}$
4	$\frac{14}{45}, \frac{64}{45} \frac{8}{15}, \frac{64}{45}, \frac{14}{45}$
5	$\frac{95}{288}, \frac{125}{96} \frac{125}{144}, \frac{125}{96}, \frac{14}{45}, \frac{95}{288}$
6	$\frac{41}{140}, \frac{54}{35} \frac{27}{140}, \frac{68}{35}, \frac{27}{140}, \frac{54}{35}, \frac{41}{140}$
7	$\frac{108}{355}, \frac{810}{559}, \frac{343}{640}, \frac{649}{536}, \frac{649}{536}, \frac{343}{640}, \frac{810}{559}, \frac{108}{355}$
9	$\frac{130}{453}, \frac{419}{265}, \frac{23}{212}, \frac{307}{158}, \frac{213}{367}, \frac{213}{367}, \frac{307}{158}, \frac{23}{212}, \frac{419}{265}, \frac{130}{453}$

Esercizio 29. Scrivere una function Matlab,

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

che implementi la formula composita di Newton-Cotes di grado `k` su `n+1` ascisse equidistanti, con `n` multiplo pari di `k`, in cui:

- `fun` è la funzione integranda (che accetta input vettoriali);
- `[a,b]` è l'intervallo di integrazione;
- `k, n` come su descritti;
- `If` è l'approssimazione dell'integrale ottenuta;
- `err` è la stima dell'errore di quadratura.

Le valutazioni funzionali devono essere fatte tutte insieme in modo vettoriale, senza ridondanze.

Listing 29: Composita

```
function [If, err] = composita(fun, a, b, k, n)
%
% [If, err] = composita(fun, a, b, k, n)
%
% Function che calcola l'approssimazione dell'integrale
% ritornando errore di quadratura
%
% Input:
% fun- identificatore della function che calcoli la funzione integranda
% a, b- intervallo di integrazione
% k- grado formula di Newton-Cotes
% n- (n+1) ascisse equidistanti con n multiplo pari di k
%
% Output:
% If- approssimazione dell'integrale
% err- stima dell'errore di quadratura

% Controlli di consistenza
if k < 1
    error("Grado inserito errato");
end
if a > b
    error("Intervallo errato");
end
if mod(n/k, 2) ~= 0
    error("n deve essere un multiplo pari di k");
end

u=1;
if (mod(k,2) == 0)
    u=2;
end

c = pesiNewtonCotes(k);
x = linspace(a, b, (n+1)+(k-1)*n);
fx = feval(fun, x);
hf = (b-a)/((n+1)+(k-1)*n);
h = (b-a)/((n/2+1)+(k-1)*n/2);

I = 0;
for i=0:n/2-1
    tmp = (i*2*k:2:(i+1)*2*k)+1;
    I = I + h*sum(fx( tmp ).*c);
end

If = 0;
for i=0:n-1
    If = If + hf*sum(fx((i*k:(i+1)*k)+1).*c);
end
```

```
err = (If-I)/(2^(k+u) - 1);
return;
end
```

Esercizio 30. Calcolare l'espressione del seguente integrale:

$$I(f) = \int_0^1 e^{3x} \mathrm{d}x.$$

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

k (n=12)	Approssimazione I	Stima dell'errore	Errore vero
1	5.9030	0.1123	0.4588
2	6.1074	0.0157	0.2545
3	6.1899	0.0109	0.1719
6	6.2747	$3.3252 * 10^{-4}$	0.0871
Risultato esatto	6.3618	—	—