Relazione Elaborato Calcolo Numerico

De Luca Riccardo 7076138 - riccardo.deluca2@edu.unifi.it

Feri Alessandro 7081870 - alessandro.feri1@edu.unifi.it

Anno accademico 2023/2024

Esercizio 1

$$D = \frac{25\,f(x) \;-\; 48\,f\big(x-h\big) \;+\; 36\,f\big(x-2h\big) \;-\; 16\,f\big(x-3h\big) \;+\; 3\,f\big(x-4h\big)}{12\,h}$$

Espandiamo con Taylor:

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

Definiamo:

$$S = 25\,f(x) - 48\,f(x-h) + 36\,f(x-2h) - 16\,f(x-3h) + 3\,f(x-4h)$$

Sostituendo le espansioni:

$$\begin{aligned} S = {} & 25\,f(x) \\ & -48\left[f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x)\right] \\ & +36\left[f(x) - 2hf'(x) + \frac{(2h)^2}{2}f''(x) - \frac{(2h)^3}{6}f^{(3)}(x) + \frac{(2h)^4}{24}f^{(4)}(x)\right] \\ & -16\left[f(x) - 3hf'(x) + \frac{(3h)^2}{2}f''(x) - \frac{(3h)^3}{6}f^{(3)}(x) + \frac{(3h)^4}{24}f^{(4)}(x)\right] \\ & +3\left[f(x) - 4hf'(x) + \frac{(4h)^2}{2}f''(x) - \frac{(4h)^3}{6}f^{(3)}(x) + \frac{(4h)^4}{24}f^{(4)}(x)\right] + O(h^5) \end{aligned}$$

Raccogliendo per potenze di h:

Termine in $f(x)$:

$$25 - 48 + 36 - 16 + 3 = 0$$

Termine in $hf'(x)$:

$$48 - 72 + 48 - 12 = 12$$

Gli altri termini di grado maggiore si annullano e S diventa:

$$S = 12hf'(x) + O(h^5)$$

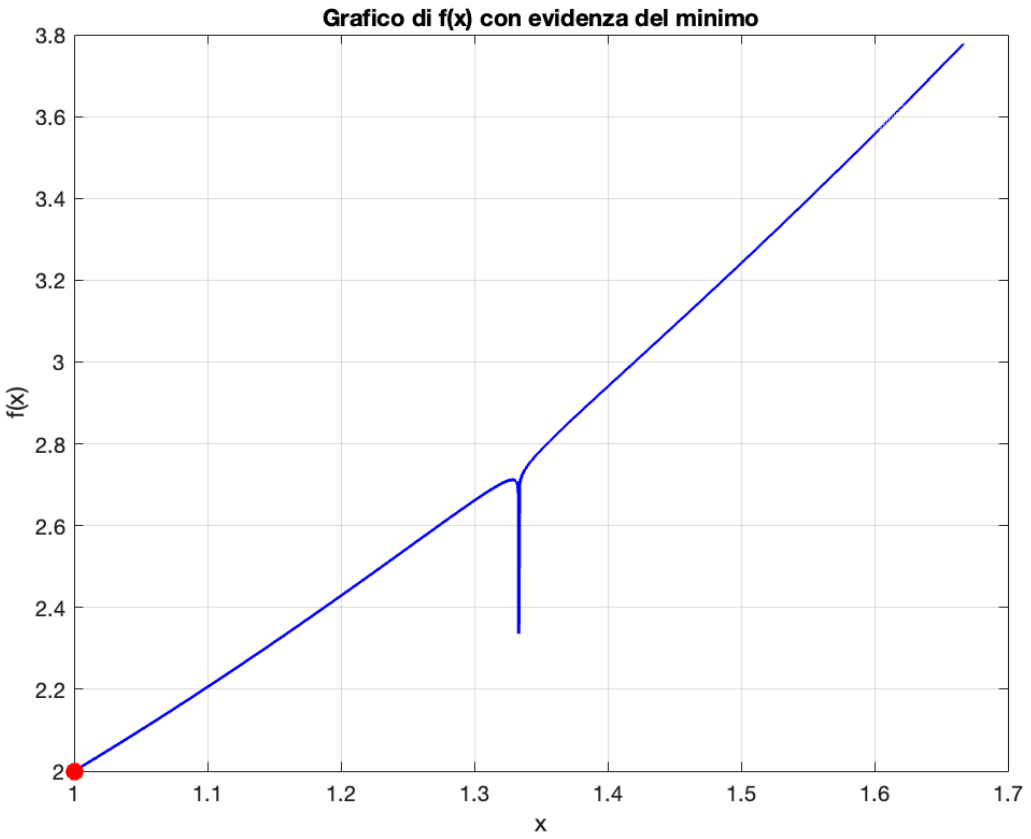
Infine si ha:

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

Esercizio 2

```
x = linspace(1, 5/3, 100001);
f = 1 + x.^2 + log(abs(3*(1-x) + 1)) / 80;
[min_f, idx_min] = min(f);
x_min = x(idx_min);
```

```
disp(['Il minimo della funzione si verifica in x = ', num2str(x_min), ' con
valore f(x) = ' , num2str(min_f)]);
figure;
plot(x, f, 'b', 'LineWidth', 1.5);
hold on;
plot(x_min, min_f, 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'r'); xlabel('x');
ylabel('f(x)');
title('Grafico di f(x) con evidenza del minimo');
grid on;
x_asintoto = 4/3;
epsilon = logspace(-10, -1, 100);
x_left = x_asintoto - epsilon;
f_left = 1 + x_left.^2 + log(abs(3*(1-x_left) + 1)) / 80;
x_right = x_asintoto + epsilon;
f_right = 1 + x_right.^2 + log(abs(3*(1-x_right) + 1)) / 80;
disp(['Limite sinistro (x → 4/3⁻): ', num2str(f_left(end))]);
disp(['Limite destro (x → 4/3⁺): ', num2str(f_right(end))]);
```



Se ci limitiamo a calcolare il minimo della funzione, sui punti campionati nell'intervallo $[1, 5/3]$, MATLAB indica un minimo in $x = 1$, con $f(1) = 2$. Questo non riflette il comportamento reale della funzione, poiché in prossimità di $x = 4/3$ si ha un asintoto verticale e la funzione tende a meno infinito. L'uso dell aritmetica floating-point, unito al fatto

che $x = 4/3$ è esattamente un punto singolare, impedisce di “vedere” il reale andamento verso $-\infty$.

Esercizio 3

La **cancellazione numerica** si verifica quando in un addizione o sottrazione di numeri molto vicini, ma di segno opposto, la precisione limitata del calcolatore provoca la perdita di cifre significative. Per esempio, consideriamo:

$$x = 1,000001 \quad e \quad y = -1,000000$$

La somma $x + y$ dovrebbe essere 0,000001, ma un calcolatore a precisione finita potrebbe restituire 0 o un valore errato, causando la cancellazione delle cifre significative.

Per valutare la sensibilità di un’operazione agli errori si utilizza il **numero di condizionamento**:

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

Se $x + y \approx 0$, il denominatore è molto piccolo e K diventa grande. Ciò indica che l’operazione è *mal condizionata* e dunque soggetta a possibili errori numerici rilevanti.

Esercizio 4

```
function [radice, iterazioni] = bisezione(funzione, estremoSx, estremoDx,
tolleranza, maxIter)
% bisezione - Approssima la radice di una funzione mediante il metodo della
bisezione.
%
%   [radice, iterazioni] = bisezione(funzione, estremoSx, estremoDx, tolleranza,
maxIter)
%
% Input:
%   funzione     - Funzione di cui cercare la radice (accetta vettori)
%   estremoSx    - Estremo sinistro dell'intervallo (deve essere < estremoDx)
%   estremoDx    - Estremo destro dell'intervallo
%   tolleranza   - Precisione richiesta (default: 1e-6)
%   maxIter      - Numero massimo di iterazioni (default:
ceil(log2(estremoDx-estremoSx)-log2(tolleranza)))
%
% Output:
%   radice       - Approssimazione della radice
%   iterazioni   - Numero di iterazioni effettuate
   if nargin < 3
       error('Sono necessari almeno 3 argomenti.');
   end
```

```
    if nargin < 4 || isempty(tolleranza)
        tolleranza = 1e-6;
    end
    if nargin < 5 || isempty(maxIter)
        maxIter = ceil(log2(estremoDx - estremoSx) - log2(tolleranza));
    end
    if estremoSx >= estremoDx
        error('Intervallo non valido: l''estremo sinistro deve essere minore di
quello destro.');
    end
    if tolleranza <= 0
        error('La tolleranza deve essere positiva.');
    end
    fSx = funzione(estremoSx);
    fDx = funzione(estremoDx);
    if fSx == 0
        radice = estremoSx;
        iterazioni = 0;
        return;
    end
    if fDx == 0
        radice = estremoDx;
        iterazioni = 0;
        return;
    end
    if fSx * fDx > 0
        error('La funzione non cambia segno nell''intervallo fornito.');
    end
    iterazioni = 0;
    radice = (estremoSx + estremoDx) / 2;
    while iterazioni < maxIter
        iterazioni = iterazioni + 1;
        radice = (estremoSx + estremoDx) / 2;
        fRadice = funzione(radice);
        derivataAppross = abs(fDx - fSx) / (estremoDx - estremoSx);
        if abs(fRadice) / abs(derivataAppross) <= tolleranza
            break;
        end
        if fSx * fRadice > 0
            estremoSx = radice;
            fSx = fRadice;
        else
            estremoDx = radice;
            fDx = fRadice;
        end
    end
end
```

Esercizio 5

Newton:

```
function [root, iter, n_eval] = newton(f, df, x0, tol, max_iter)
% newton - Metodo di Newton per trovare uno zero di f.
%
%   [root, iter, n_eval] = newton(f, df, x0, tol, max_iter)
%
% Input:
%   f       - handle della funzione (es. @(x) x.^2-2)
%   df      - handle della derivata di f (es. @(x) 2*x)
%   x0      - approssimazione iniziale
%   tol     - tolleranza per la convergenza (default 1e-16)
%   max_iter- numero massimo di iterazioni (default 1000)
%
% Output:
%   root    - l'approssimazione dello zero
%   iter    - numero di iterazioni eseguite
%   n_eval  - numero totale di valutazioni della funzione f (e di df se
necessario)
   if nargin < 4 || isempty(tol)
       tol = 1e-16;
   end
   if nargin < 5 || isempty(max_iter)
       max_iter = 1000;
   end
   root = x0;
   n_eval = 0;
   for iter = 1:max_iter
       fx = f(root);
       n_eval = n_eval + 1;
       dfx = df(root);
       n_eval = n_eval + 1;
       if dfx == 0
           error('La derivata si annulla in x = %f.', x);
       end
       root = root - fx/dfx;
       if abs(root - x0) <= tol * (1 + abs(x0))
           return;
       else
           x0 = root;
       end
   end
   warning('Il metodo di Newton non ha convertito in %d iterazioni.', max_iter);
return
```

Secanti:

```
function [soluzione, numIterazioni] = secanti(funzione, p0, p1, precisione,
maxIterazioni)
```

```
% secanti - Calcola un'approssimazione della radice di una funzione
%           utilizzando il metodo delle secanti.
%
%   [soluzione, numIterazioni] = secanti(funzione, p0, p1, precisione,
maxIterazioni)
%
% Input:
%   funzione      - Funzione per la quale trovare la radice (handle o funzione
anonima).
%   p0, p1        - Due approssimazioni iniziali.
%   precisione    - Tolleranza richiesta (default: 1e-15).
%   maxIterazioni - Numero massimo di iterazioni (default: 1000).
%
% Output:
%   soluzione     - Approssimazione della radice trovata.
%   numIterazioni - Numero di iterazioni eseguite.
%
   if nargin < 3
       error('Numero insufficiente di argomenti in ingresso.');
   end
   if nargin == 3
       precisione    = 1e-15;
       maxIterazioni = 1000;
   elseif nargin == 4
       maxIterazioni = 1000;
   end
   if precisione <= 0
       error('La precisione deve essere un valore positivo.');
   end
   if maxIterazioni <= 0
       error('Il numero massimo di iterazioni deve essere maggiore di zero.');
   end
   f_p0 = funzione(p0);
   f_p1 = funzione(p1);

   numIterazioni = 0;
   erroreCorrente = Inf;
   while numIterazioni < maxIterazioni && erroreCorrente >= precisione
       numIterazioni = numIterazioni + 1;

       if f_p1 == f_p0
           error('Impossibile proseguire: f(p0) e f(p1) risultano uguali.');
       end

       pNuovo = ( f_p1 * p0 - f_p0 * p1 ) / ( f_p1 - f_p0 );
       erroreCorrente = abs(pNuovo - p1);

       p0 = p1;
       f_p0 = f_p1;
       p1 = pNuovo;
       f_p1 = funzione(p1);
```

```
    end
    if erroreCorrente >= precisione
        error('La radice non è stata trovata entro il numero massimo di
iterazioni.');
    end
    soluzione = p1;
end
```

Esercizio 6

```
clearvars; close all; clc
f = @(x) exp(x) - cos(x);
df = @(x) exp(x) + sin(x);
% Parametri iniziali per ciascun metodo:
tol_list = [1e-3, 1e-6, 1e-9, 1e-12];
max_iter = 100;
% Per il metodo della bisezione, usiamo l'intervallo iniziale [-0.1, 1]
a0 = -0.1;
b0 = 1;
% Per il metodo di Newton: x0 = 1
x0_newton = 1;
% Per il metodo delle secanti: x0 = 1 e x1 = 0.9
x0_sec = 1;
x1_sec = 0.9;
results = {};
for tol = tol_list
  % Metodo della Bisezione
  [root_bis, iter_bis] = bisezione(f, a0, b0, tol, max_iter);
  results = [results; {'Bisezione', tol, root_bis, iter_bis}];
   % Metodo di Newton
  [root_newton, iter_newton, n_eval_newton] = newton(f, df, x0_newton, tol,
max_iter);
  results = [results; {'Newton', tol, root_newton, iter_newton}];
   % Metodo delle Secanti
  [root_sec, iter_sec] = secanti(f, x0_sec, x1_sec, tol, max_iter);
  results = [results; {'Secanti', tol, root_sec, iter_sec}];
end
fprintf('Metodo      \tTolleranza\tRadice\t\t\tIterazioni\n');
fprintf('-----------------------------------------------------------------------
----------------------\n');
for i = 1:size(results,1)
  fprintf('%-10s\t%1.0e\t\t%1.6e\t\t%3d\n', ...
      results{i,1}, results{i,2}, results{i,3}, results{i,4});
end
```

Tolleranza 10^{-3}

Metodo	Radice	Iterazioni
Bisezione	$9.765625e^{-4}$	9
Newton	$2.842335e^{-9}$	5
Secanti	$1.152202e^{-6}$	6

Tolleranza 10^{-6}

Metodo	Radice	Iterazioni
Bisezione	$9.536743e^{-7}$	19
Newton	$3.574791e^{-17}$	6
Secanti	$2.094874e^{-16}$	8

Tolleranza 10^{-9}

Metodo	Radice	Iterazioni
Bisezione	$9.313226e^{-10}$	29
Newton	$3.574791e^{-17}$	7
Secanti	$2.094874e^{-16}$	8

Tolleranza 10^{-12}

Metodo	Radice	Iterazioni
Bisezione	$9.094871e^{-13}$	39
Newton	$3.574791e^{-17}$	7
Secanti	$-1.255712e^{-17}$	9

Dall'analisi dei dati emerge che il metodo di Newton è il più efficiente in termini di numero di iterazioni, raggiungendo la convergenza rapidamente in tutti i casi testati. Il metodo delle secanti si dimostra anch'esso molto efficace, con un numero di iterazioni leggermente superiore rispetto a Newton, ma senza la necessità di calcolare la derivata. Il metodo della bisezione, pur essendo il più stabile, risulta il più lento, richiedendo un numero significativamente maggiore di iterazioni per ottenere la stessa precisione.

Esercizio 7

```
clearvars; close all; clc
f  = @(x) exp(x) - cos(x) + sin(x) - x.*(x+2);
```

```
df = @(x) exp(x) + sin(x) + cos(x) - (2*x + 2);  % derivata di f(x)
tol_list = [1e-3, 1e-6, 1e-9, 1e-12];
max_iter = 200;
% Dati iniziali:
a0 = -0.1;      % Bisezione: estremo sinistro
b0 = 1;         % Bisezione: estremo destro
x0_newton = 1;  % Newton: approssimazione iniziale
x0_sec = 1;     % Secanti: primo punto
x1_sec = 0.9;   % Secanti: secondo punto
x0_newt_mod = 1;% Newton modificato: approssimazione iniziale
m_newt_mod = 5; % Newton modificato: molteplicità della radice
results = {};
for tol = tol_list
   % Metodo della Bisezione
   try
       [root_bis, iter_bis] = bisezione(f, a0, b0, tol);
       results = [results; {'Bisezione', tol, root_bis, iter_bis}];
   catch ME
       disp(['Err: ', ME.message]);
       results = [results; {'Bisezione', tol, "Non Converge", "-"}];
   end
   % Metodo di Newton
   try
       [root_newton, iter_newton, n_eval_newton] = newton(f, df, x0_newton, tol,
max_iter);
       results = [results; {'Newton', tol, root_newton, iter_newton}];
   catch ME
       disp(['Err: ', ME.message]);
       results = [results; {'Newton', tol, "Non Converge", "-"}];
   end
   % Metodo delle Secanti
   try
       [root_sec, iter_sec] = secanti(f, x0_sec, x1_sec, tol, max_iter);
       results = [results; {'Secanti', tol, root_sec, iter_sec}];
   catch ME
       disp(['Err: ', ME.message]);
       results = [results; {'Secanti', tol, "Non Converge", "-"}];
   end
   % Metodo di Newton Modificato
   try
       [root_newt_mod, iter_newt_mod, n_eval_newt_mod] = newton_mod(f,
m_newt_mod, df, x0_newt_mod, tol, max_iter);
       results = [results; {'Newton Modificato', tol, root_newt_mod,
iter_newt_mod, n_eval_newt_mod}];
   catch ME
       disp(['Err: ', ME.message]);
       results = [results; {'Newton Modificato', tol, "Non Converge", "-"}];
   end
end
fprintf('Metodo            \tTolleranza\tRadice\t\tIterazioni\n');
fprintf('-------------------------------------------------------------------
```

```
----------------------\n');
for i = 1:size(results,1)
   metodo = results{i,1};
   toll   = results{i,2};
   radice = results{i,3};
   iterazioni = results{i,4};
   if isnumeric(radice)
       radice_str = sprintf('%1.6e', radice);
   else
       radice_str = radice;
   end
   if isnumeric(iterazioni)
       iter_str = sprintf('%d', iterazioni);
   else
       iter_str = iterazioni;
   end
   fprintf('%-18s\t%1.0e\t\t%-15s\t%-15s\n', metodo, toll, radice_str,
iter_str);
end
```

Tolleranza 10^{-3}

Metodo	Radice	Iterazioni
Bisezione	$3.750000e^{-2}$	3
Newton	$3.927574e^{-03}$	25
Newton Modificato	Non Converge	-
Secanti	$5.576032e^{-3}$	33

Tolleranza 10^{-6}

Metodo	Radice	Iterazioni
Bisezione	$3.125000e^{-3}$	5
Newton	Non Converge	-
Newton Modificato	Non Converge	-
Secanti	$-1.040327e^{-3}$	61

Tolleranza 10^{-9}

Metodo	Radice	Iterazioni
Bisezione	$1.116270e^{-3}$	31
Newton	Non Converge	-
Newton Modificato	Non Converge	-

Secanti	$-1.075035e^{-3}$	89

Tolleranza 10^{-12}

Metodo	Radice	Iterazioni
Bisezione	$1.116270e^{-3}$	32
Newton	Non Converge	-
Newton Modificato	Non Converge	-
Secanti	$-1.0750351e^{-3}$	123

Il metodo di Newton converge solo per una tolleranza, per tolleranze più strette non converge, indicando che la derivata della funzione causa problemi numerici, come divisioni per valori prossimi a zero o iterazioni instabili. Il metodo di Newton modificato non converge per nessuna tolleranza, mostrando che anche la sua variante non riesce a gestire il comportamento della derivata.

Il metodo delle secanti si dimostra più efficace, riuscendo a convergere in tutti i casi, anche se con un numero di iterazioni superiore rispetto al metodo di Newton nei casi in cui quest'ultimo riusciva a convergere. La bisezione, come previsto, è il metodo più lento in termini di numero di iterazioni, ma è anche il più affidabile, garantendo sempre la convergenza.

Esercizio 8

```
function x = mialu(A, b)
% mialu - Risolve il sistema lineare Ax = b usando la fattorizzazione LU con
pivoting parziale.
%
%   x = mialu(A, b)
%
% Input:
%   A   - matrice n x n
%   b   - vettore termini noti
%
% Output:
%   x   - soluzione del sistema lineare Ax = b
   [n, m] = size(A);
   if n == 1
       x = b / A;
       return;
   end
   if n ~= m
       error('La matrice A deve essere quadrata.');
   end
   if size(b, 1) ~= n || size(b, 2) ~= 1
```

```
        error('Il vettore b deve essere colonna e avere dimensione compatibile
con A.');
    end
    L = eye(n);
    U = A;
    P = eye(n);

    for k = 1:n-1
        [pivot_val, pivot_index] = max(abs(U(k:n, k)));
        pivot_index = pivot_index + k - 1;

        if pivot_val == 0
            error('La matrice A è singolare.');
        end
        if pivot_index ~= k
            U([k, pivot_index], :) = U([pivot_index, k], :);
            P([k, pivot_index], :) = P([pivot_index, k], :);
            if k > 1
                L([k, pivot_index], 1:k-1) = L([pivot_index, k], 1:k-1);
            end
        end
        L(k+1:n, k) = U(k+1:n, k) / U(k, k);
        U(k+1:n, :) = U(k+1:n, :) - L(k+1:n, k) * U(k, :);
    end
    if U(n, n) == 0
        error('La matrice A è singolare.');
    end
    b_perm = P * b;

    y = zeros(n, 1);
    for i = 1:n
        y(i) = b_perm(i) - L(i, 1:i-1) * y(1:i-1);
    end

    x = zeros(n, 1);
    for i = n:-1:1
        x(i) = (y(i) - U(i, i+1:n) * x(i+1:n)) / U(i, i);
    end
end
```

Inserendo la matrice $A = [1, 2, 3; 0, 0, 1]$ (non quadrata) con un vettore b, si ottiene l'errore "`La matrice A deve essere quadrata.`". Inserendo la matrice $A = [2, 4, 8; 1, 2, 4; 3, 6, 12]$ (singolare) con un vettore b , si ottiene l'errore "`La matrice A è singolare.`". Inserendo una matrice quadrata con il vettore $b = [5; 6; 7; 8]$ (dimensione non compatibile), si ottiene l'errore "`Il vettore b deve essere colonna e avere dimensione compatibile con A.`". Inserendo la matrice $A = [2, 1, 3; 4, 5, 6; 7, 8, 10]$ (quadrata e non singolare) e il vettore $b = [3; 2; 1]$, si ottiene il vettore soluzione:

$$x = \begin{bmatrix} -8.3333 \\ -1.3333 \\ 7.0000 \end{bmatrix}$$

Esercizio 9

```
function x = mialdl(A, b)
% mialdl - Risolve il sistema lineare Ax = b per una matrice A simmetrica e
definita positiva utilizzando la fattorizzazione LDL^T (senza pivoting).
%
%   x = mialdl(A, b)
%
% Input:
%   A   - matrice n x n (simmetrica, definita positiva)
%   b   - vettore colonna di dimensione n
%
% Output:
%   x   - soluzione del sistema lineare Ax = b
   [n, m] = size(A);
   if n ~= m
       error('La matrice A deve essere quadrata.');
   end
   if ~isequal(A, A')
       error('La matrice A non e'' simmetrica.');
   end
   if ~isequal(size(b), [n, 1])
       error('Il vettore b deve essere colonna e avere dimensione compatibile
con A.');
   end
   L = eye(n);
   D = zeros(n, 1);
   for k = 1:n
       somma = 0;
       for j = 1:k-1
           somma = somma + L(k,j)^2 * D(j);
       end
       D(k) = A(k,k) - somma;
       if D(k) <= 0
           error('La matrice A non e'' definita positiva (valore diagonale <=
0).');
       end
       for i = k+1:n
           somma = 0;
           for j = 1:k-1
               somma = somma + L(i,j)*L(k,j)*D(j);
           end
           L(i,k) = (A(i,k) - somma) / D(k);
       end
   end
```

```
    y = zeros(n,1);
    for i = 1:n
        s = 0;
        for j = 1:i-1
            s = s + L(i,j)*y(j);
        end
        y(i) = (b(i) - s);
    end
    z = y ./ D;
    x = zeros(n,1);
    for i = n:-1:1
        s = 0;
        for j = i+1:n
            s = s + L(j,i)*x(j);
        end
        x(i) = (z(i) - s);
    end
end
```

Inserendo la matrice $A = [1, 0, 2; 0, 0, 2]$ (non quadrata) con un qualsiasi vettore b, si ottiene l'errore: "`La matrice A deve essere quadrata.`". Inserendo il vettore $b = [1; 2; 3; 4]$ con una matrice di dimensione $n \times n$ e $n \neq 4$, si ottiene l'errore: "`Il vettore b deve essere colonna e avere dimensione compatibile con A.`". Inserendo la matrice $A = [2, -1, 0; -1, 0, 1; 0, 1, 3]$ con un qualsiasi vettore colonna, si ottiene l'errore: "`La matrice A non e'' definita positiva (valore diagonale <= 0).`". Inserendo la matrice $A = [4, 1, 2; 0, 2, 3; 1, 0, 3]$ (non simmetrica) si ottiene l'errore "`La matrice A non e'' simmetrica.`". Inserendo la matrice $A = [4, 1, 1; 1, 2, 0; 1, 0, 3]$ (simmetrica e definita positiva) e il vettore $b = [1; 2; 3]$, si ottiene il vettore soluzione:

$$x = \begin{bmatrix} -0.3158 \\ 1.1579 \\ 1.1053 \end{bmatrix}$$

Esercizio 10

```
function [x, nr] = miaqr(A, b)
% miaqr - Risolve il sistema lineare Ax = b nel senso dei minimi quadrati
utilizzando la fattorizzazione QR con Householder. Restituisce la norma del
residuo.
%
%   [x, nr] = miaqr(A, b)
%
% Input:
%   A : matrice m x n, con m >= n e rank(A) = n.
%   b : vettore colonna di lunghezza m.
%
% Output:
```

```
%   x  : soluzione in minimi quadrati di Ax = b.
%   nr : norma del vettore residuo, ||b - A*x||.
    [m, n] = size(A);
    if length(b) ~= m
        error('Dimensione del vettore b non compatibile con la matrice A.');
    end
    if m < n
        error('La matrice A deve avere m >= n per la fattorizzazione QR.');
    end
    if rank(A) < n
        error('rank(A) < n: impossibile eseguire la fattorizzazione QR per un
sistema ben determinato.');
    end
    R = A;
    Q = eye(m);
    for k = 1:n
        x = R(k:m, k);
        alpha = norm(x, 2);
        if x(1) < 0
            alpha = -alpha;
        end
        v = x;
        v(1) = v(1) + alpha;
        if norm(v, 2) > 0
            v = v / norm(v, 2);
            Hk = eye(m);
            Hk(k:m, k:m) = Hk(k:m, k:m) - 2*(v * v');
            R = Hk * R;
            Q = Q * Hk';
        end
    end
    R_upper = R(1:n, 1:n);
    Qt_b = Q' * b;
    y = Qt_b(1:n);
    x = R_upper \ y;
    r = b - A*x;
    nr = norm(r);
end
```

Inserendo la matrice $A = [1, 2; 3, 4; 5, 6]$ con il vettore $b = [1; 2]$ (che ha dimensione incompatibile), si ottiene l'errore "`Dimensione del vettore b non compatibile con la matrice A.`". Inserendo la matrice $A = [1, 2, 3; 4, 5, 6]$ (con $m < n$) e un vettore b, si ottiene l'errore "`La matrice A deve avere m >= n per la fattorizzazione QR.`". Inserendo la matrice $A = [1, 2, 3; 2, 4, 6; 3, 6, 9]$ (che ha colonne linearmente dipendenti, quindi $\text{rank}(A) < n$), si ottiene l'errore "`rank(A) < n: impossibile eseguire la fattorizzazione QR per un sistema ben determinato.`". Inserendo la matrice $A = [1, 2; 3, 4; 5, 6]$ (dove $m > n$ e $\text{rank}(A) = n$), e il vettore $b = [1; 2; 3]$, si ottiene la soluzione:

$$x = \begin{bmatrix} -0.0000 \\ 0.5000 \end{bmatrix}$$

e la norma del residuo $nr = 0$

Esercizio 11

```
close all; clearvars; clc;
results = zeros(15, 2);
for n = 1:15
    [A, b] = genera_matrice_vettore(n);
    condizionamento_2 = cond(A);
    display(num2str(condizionamento_2));

    x_exact = ones(n, 1);

    x_computed = mialu(A, b);
end

function [A, b] = genera_matrice_vettore(n)
    A = zeros(n, n);
    for i = 1:n
        for j = 1:n
            A(i, j) = 10^max(i - j, 0);
        end
    end

    b = zeros(n, 1);
    for i = 1:n
        b(i) = n - i + (10^i - 1) / 9;
    end
end
```

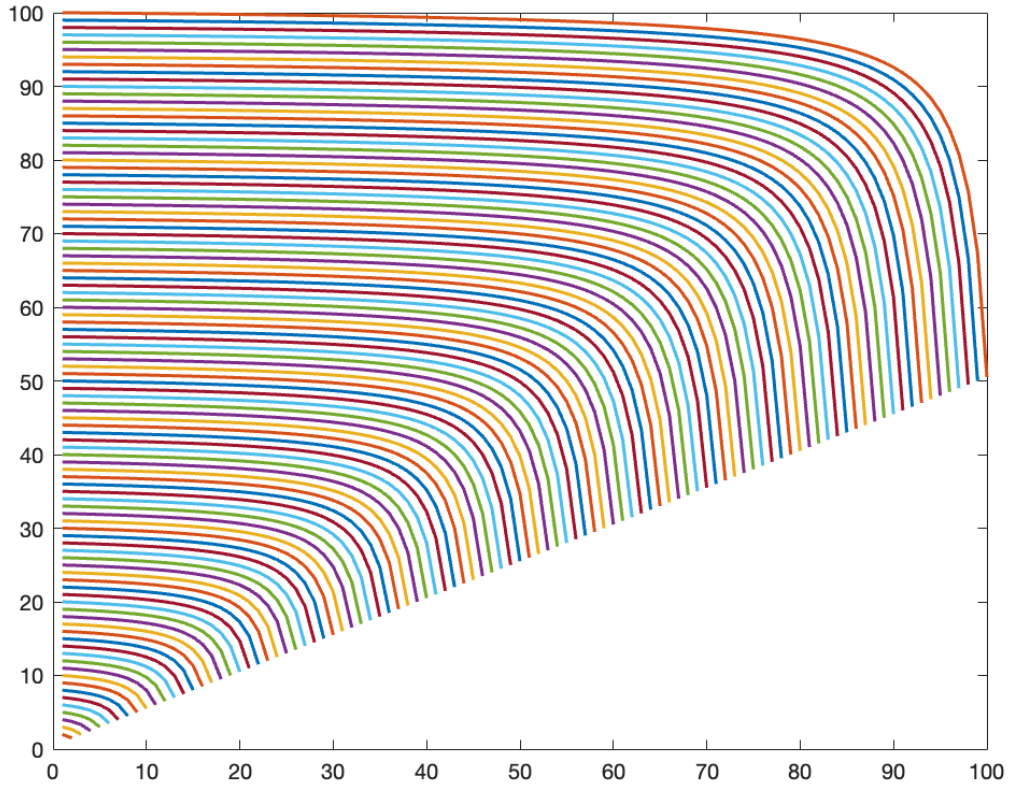
Matrice	Numero di condizionamento
A_1	1.0000×10^0
A_2	1.1356×10^1
A_3	1.9106×10^2
A_4	2.1679×10^3
A_5	2.2819×10^4
A_6	2.3418×10^5
A_7	2.3771×10^6

A_8	2.3995×10^7
A_9	2.4147×10^8
A_{10}	2.4254×10^9
A_{11}	2.4332×10^{10}
A_{12}	2.4391×10^{11}
A_{13}	2.4437×10^{12}
A_{14}	2.4473×10^{13}
A_{15}	2.4501×10^{14}

Dalla tabella si nota che il numero di condizionamento delle matrici cresce rapidamente all'aumentare di n, indicando che il sistema diventa progressivamente più sensibile a perturbazioni nei dati. Nonostante ciò, la funzione per gli n da 1 a 15 calcola il risultato corretto (ovvero 1).

Esercizio 12

```
close all; clearvars; clc;
for dim = 1:100
   A = -ones(dim) + diag(ones(1, dim) * dim + 1);

   LDT = mialdlt(A);

   plot(1:dim, diag(LDT), 'LineWidth', 1.5);
   hold on;
end
hold off;
```



Dall'analisi del grafico si osserva che tutti gli elementi diagonali del fattore D sono sempre positivi, confermando che la matrice è definita positiva. Inoltre, i valori seguono un andamento regolare: iniziano con valori più elevati per gli indici iniziali della diagonale e decrescono progressivamente all'aumentare dell'indice, evidenziando la simmetria delle matrici.

Esercizio 13

```
A = [7 2 1; 8 7 8; 7 0 7; 4 3 3; 7 0 10];
b = [1; 2; 3; 4; 5];
omega = [0.5; 0.5; 0.75; 0.25; 0.25];
B = diag(sqrt(omega));
[x, nr] = miaqr(B*A, B*b);
```

Dal codice sopra otteniamo:

$$x = [0.1531, -0.1660, 0.3185]$$
$$nr = 1.5940$$

Esercizio 14

La funzione è stata nominata $newton2$ per evitare conflitti con la funzione $newton$ definita precedentemente

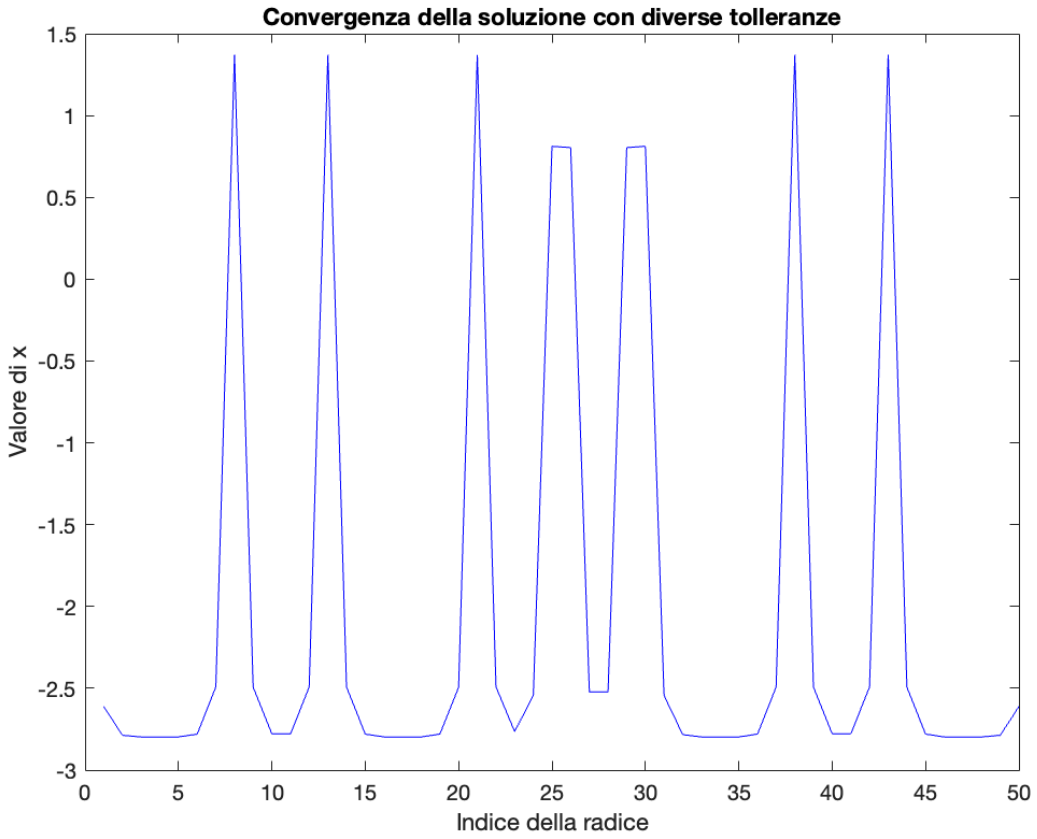
```
function [x, nit] = newton2(fun, x0, tol, maxit)
% newton2 - Risolve un sistema di equazioni non lineari f(x)=0
%          tramite il metodo di Newton multivariato.
%
%   [x, nit] = newton2(fun, x0, tol, maxit)
%
% Input:
%   fun   - funzione che, dato x, restituisce [f, J]
%           dove f è il vettore delle equazioni, J la Jacobiana
%   x0    - vettore colonna iniziale (condizione iniziale)
%   tol   - tolleranza per il criterio di arresto (default 1e-6)
%   maxit - numero massimo di iterazioni (default 100)
%
% Output:
%   x     - soluzione approssimata
%   nit   - numero di iterazioni effettuate
   if nargin < 3 || isempty(tol)
       tol = 1e-6;
   end
   if nargin < 4 || isempty(maxit)
       maxit = 100;
   end

   x0 = x0(:);
   nit = maxit;

   for i = 1:maxit
       [fval, J] = fun(x0);
       delta = mialum(J, -fval);
       x = x0 + delta;
       if norm(delta ./ (1 + abs(x0)), inf) <= tol
           nit = i;
           break
       end
       x0 = x;
   end
   warning('Il metodo di Newton non è convergente entro %d iterazioni.', maxit);
end

function x = mialum(A, b)
% mialum - Risolve il sistema lineare Ax = b con fattorizzazione LU senza
% pivoting parziale
%
%   x = mialum(A, b)
%
% Input:
%   A   - matrice n x n
%   b   - vettore termini noti
%
% Output:
%   x   - soluzione del sistema lineare Ax = b
```

```
    [n, m] = size(A);
    if n ~= m
        error('La matrice A deve essere quadrata.');
    end
    if size(b, 1) ~= n || size(b, 2) ~= 1
        error('Il vettore b deve essere colonna e avere dimensione compatibile
con A.');
    end
    for i = 1:n-1
        if A(i, i) == 0
            error("La matrice è singolare")
        end
        for j = i+1:n
            A(j, i) = A(j, i) / A(i, i);
            A(j, i+1:n) = A(j, i+1:n) - A(j, i) * A(i, i+1:n);
        end
    end
    x = b(:);
    for i = 2:n
        x(i:n) = x(i:n) - A(i:n, i - 1) * x(i-1);
    end
    for i = n:-1:1
        x(i) = x(i) / A(i, i);
        x(1:i-1) = x(1:i-1) - A(1:i-1, i) * x(i);
    end
end
```

Esercizio 15

```
clc; clearvars; close all;
tolleranze = [1e-3, 1e-8, 1e-13];
iterazioni = zeros(length(tolleranze), 1);
x0 = zeros(50, 1);
x = zeros(50, length(tolleranze));
for i = 1:length(tolleranze)
    [x(:, i), iterazioni(i)] = newton2(@fun, x0, tolleranze(i), 1000);
end
fprintf('Tolleranza\tIterazioni\n');
for i = 1:length(tolleranze)
    fprintf('10^{%d}\t\t%d\n', log10(tolleranze(i)), iterazioni(i));
end
figure;
colors = {'r', 'g', 'b'};
hold on;
for i = 1:length(tolleranze)
    plot(1:50, x(:, i), colors{i}, 'DisplayName', ...
        sprintf('tol = 10^{%d}', log10(tolleranze(i))));
end
hold off;
```

```
xlabel('Indice della radice');
ylabel('Valore di x');
title('Convergenza della soluzione con diverse tolleranze');
legend('Location', 'best');

function [f, jacobian] = fun(x)
% fun - Calcola il gradiente e l'Hessiana di f(x)
   x = x(:);
   n = length(x);
   Q = 4 * eye(n) + diag(ones(n-1, 1), 1) + diag(ones(n-1, 1), -1);
   e = ones(n, 1);
   alpha = 2;
   beta = -1.1;
   grad = @(x) Q * x - alpha * e .* sin(alpha * x) - beta * e .* exp(-x);

   Jac = @(x) Q - alpha^2 * diag(e .* cos(alpha * x)) + beta * diag(e .*
exp(-x));

   f = grad(x);
   jacobian = Jac(x);
end
```



Tolleranze	Interazioni

10^{-3}	700
10^{-8}	701
10^{-13}	702

Esercizio 16

```
function yy = lagrange(x, y, xx)
% lagrange - Interpolazione polinomiale di Lagrange
%
%   yy = lagrange(x, y, xx)
%
% Input:
%   x  - Vettore dei nodi di interpolazione
%   y  - Vettore dei valori corrispondenti ai nodi
%   xx - Vettore di punti nei quali valutare l'interpolazione
%
% Output:
%   yy - Vettore contenente i valori interpolati in xx
   n = length(x);
   L = ones(n, length(xx));

   for i = 1:n
       for j = 1:n
           if i ~= j
               L(i, :) = L(i, :) .* (xx - x(j)) / (x(i) - x(j));
           end
       end
   end
   yy = sum(y(:) .* L, 1);

   yy = reshape(yy, size(xx));
end
```

Esercizio 17

La funzione è stata nominata $newton3$ per evitare conflitti con la funzione $newton$ definita precedentemente

```
function YQ = newton3(X, Y, XQ)
% newton3 - Interpolazione polinomiale di Newton
%
%     YQ = newton3(X, Y, XQ)
%
% Input:
```

```
%   X  - Vettore dei nodi di interpolazione
%   Y  - Vettore dei valori corrispondenti ai nodi
%   XQ - Vettore di punti nei quali valutare l'interpolazione
%
% Output:
%   yy - Vettore contenente i valori interpolati in xx
   if length(X) ~= length(Y) || length(X) <= 0
          error('I vettori X e Y devono avere la stessa dimensione e non essere
vuoti.');
   end

   if length(unique(X)) ~= length(X)
          error('Le ascisse in X devono essere distinte.');
   end
   df = difdiv(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 = difdiv(x, f)
% difdiv - Calcola le differenze divise di Newton per i nodi (x, f)
%
%     df = difdiv(x, f)
%
% Input:
%   x   - vettore delle ascisse
%     f   - vettore delle ordinate
%
% Output:
%     df  - vettore delle differenze divise
   n = length(x);
   if length(f) ~= n
          error('Dati errati');
   end
   n = n-1;
   df = f;
   for j=1:n
          for i = n+1:-1:j+1
                df(i) = (df(i) - df(i-1))/(x(i) - x(i-j));
          end
   end
return
```

Esercizio 18

```
function yy = hermite(xi, fi, f1i, xx)
% hermite - funzione che calcola il polinomio interpolante di Hermite
```

```
%
%   yy = hermite(xi, fi, f1i, xx)
%
% Input:
%   xi  - vettore dei punti dati
%   fi  - valori della funzione in xi
%   f1i - valori delle derivate in xi
%   xx  - punti di valutazione
%
% Output:
%   yy  - valori interpolati in xx
   if length(fi) ~= length(xi) || length(xi) <= 0 || length(xi) ~= length(f1i)
       error("Dimensioni errate");
   end
   if length(unique(xi)) ~= length(xi)
       error("Ascisse non distinte")
   end

   fi = repelem(fi, 2);
   for i = 1:length(f1i)
       fi(i*2) = f1i(i);
   end
   df = difdivHermite(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
end

function df = difdivHermite(X, Y)
% difdivHermite - Calcola le differenze divise di Hermite sulle coppie (xi, fi)
%
%     df = difdivHermite(X, Y)
%
% Input:
%   X   - vettore delle ascisse
%     Y   - vettore delle ordinate e delle derivate della forma [f(0) f'(0)
f(1)...]
%
% Output:
%     df  - vettore delle differenze divise di Hermite
%
   n = length(X)-1;
   df = Y;
   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
end
```

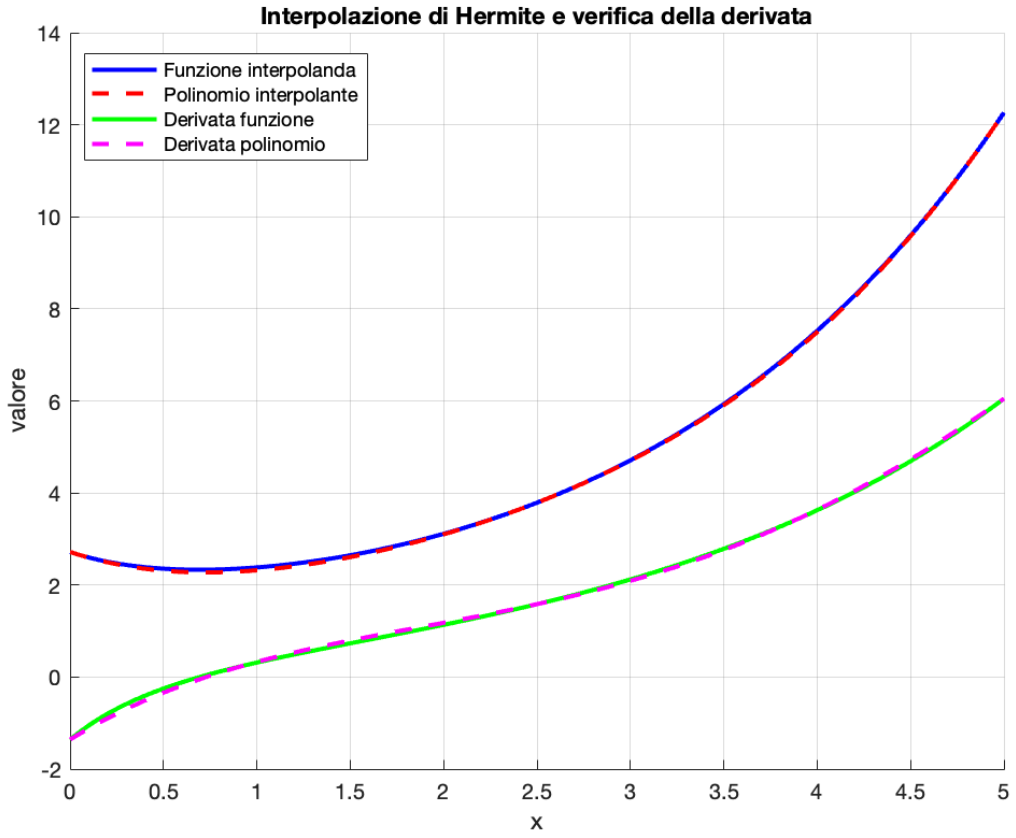
Esercizio 19

```
function [p, dp] = hornerDeriv(x, a, X)
% hornerDeriv - Valuta il polinomio in forma di Newton e la sua derivata
%
%   [p, dp] = hornerDeriv(x, a, X)
%
% Input:
%   x   - Ascissa in cui valutare il polinomio e la sua derivata
%   a   - Vettore dei coefficienti {a0, a1, ..., an} del polinomio in forma di
Newton
%   X   - Vettore dei nodi {x0, x1, ..., x_{n-1}} (di lunghezza n-1)
%
% Output:
%   p   - Valore del polinomio p(x)
%   dp  - Valore della derivata p'(x)
    if nargin < 3
        error('Numero di parametri insufficienti');
    end
    n = length(a);
    if length(X) ~= n - 1
        error('La lunghezza di X deve essere pari a n-1, dove n = length(a)');
    end
    p = a(n);
    dp = 0;

    for k = n-1:-1:1
        dp = dp * (x - X(k)) + p;
        p = p * (x - X(k)) + a(k);
    end
end
```

Esercizio 20

```
clc; clearvars; close all;
f  = @(x) exp(x/2 + exp(-x));
df = @(x) 0.5 * exp(exp(-x) - x/2) .* (-2 + exp(x));
xi  = [0, 2.5, 5];
fi  = f(xi);
f1i = df(xi);
x = linspace(0, 5, 200);
pvals = hermite(xi, fi, f1i, x);
xiH = repelem(xi, 2);
```

```
fiH = repelem(fi, 2);
for i = 1:length(f1i)
   fiH(2*i) = f1i(i);
end
dd = difdivHermite(xi, fiH);
XforHorner = xiH(1:end-1);
dvals = zeros(size(x));
for k = 1:length(x)
   [~, dvals(k)] = hornerDeriv(x(k), dd, XforHorner);
end
figure('Name','Interpolazione di Hermite - Funzione e Derivata');
hold on;
plot(x, f(x),    'b-', 'LineWidth', 2, 'DisplayName','Funzione interpolanda');
plot(x, pvals,   'r--','LineWidth', 2, 'DisplayName','Polinomio interpolante');
plot(x, df(x),   'g-', 'LineWidth', 2, 'DisplayName','Derivata funzione');
plot(x, dvals,   'm--','LineWidth', 2, 'DisplayName','Derivata polinomio');
hold off;
xlabel('x');
ylabel('valore');
title('Interpolazione di Hermite e verifica della derivata');
legend('Location','best');
grid on;
disp('Verifica delle condizioni di Hermite nei nodi:');
for i = 1:length(xi)
   [pVal_i, dVal_i] = hornerDeriv(xi(i), dd, XforHorner);
   fprintf('x = %.2f: f = %g, p = %g,  df = %g, p'' = %g\n', ...
       xi(i), f(xi(i)), pVal_i, df(xi(i)), dVal_i);
end
```



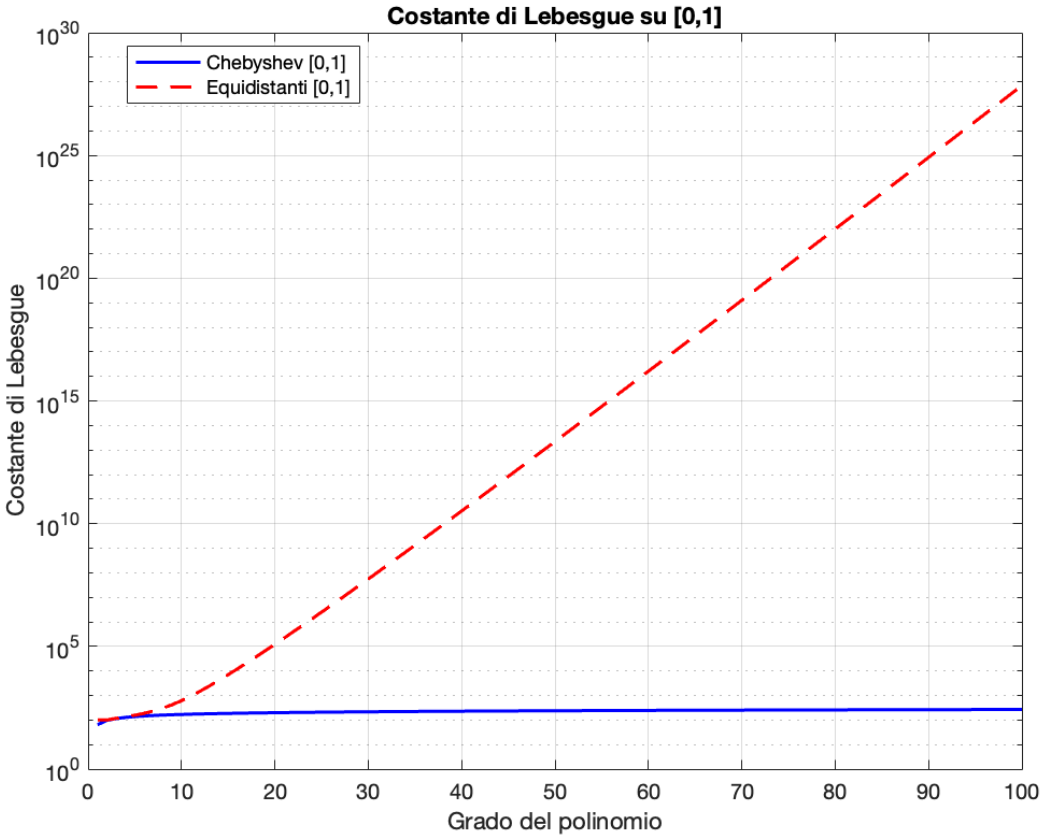
Esercizio 21

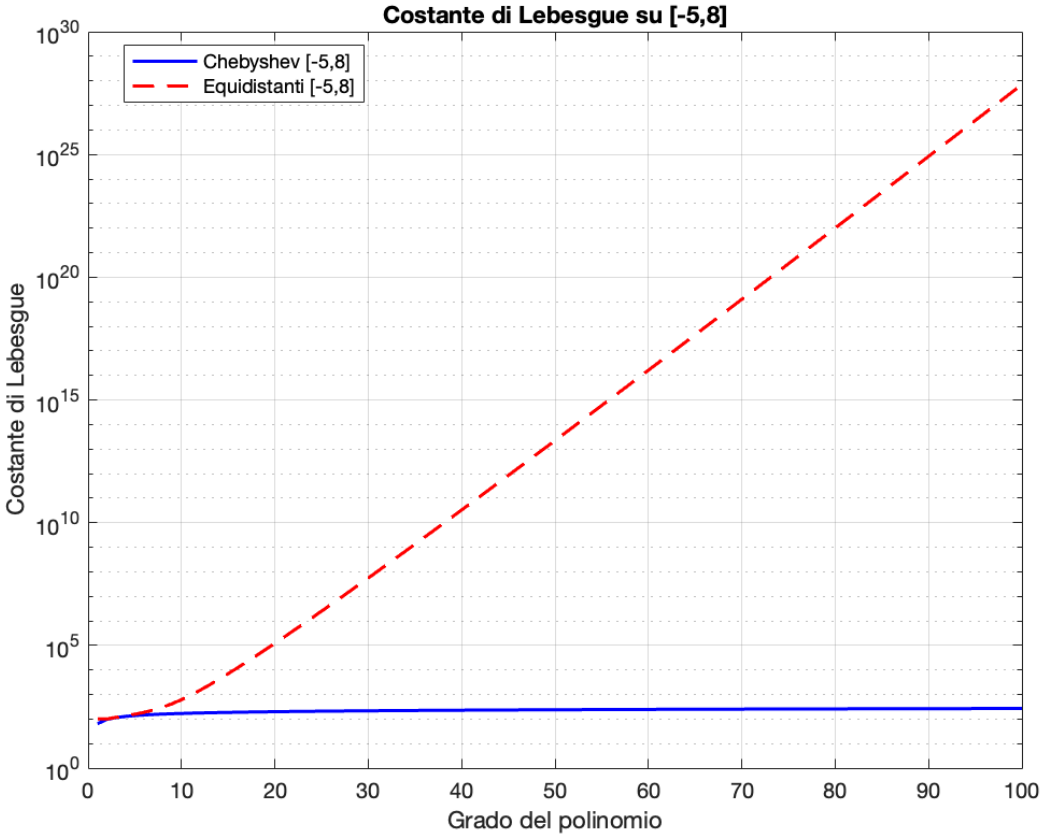
```
function x = chebyshev(n, a, b)
% chebyshev - Calcola le ascisse di Chebyshev per l'interpolazione polinomiale
%
%   x = chebyshev(n, a, b)
%
% Input:
%   n   - grado del polinomio interpolante (vengono calcolati n+1 nodi)
%   a   - estremo sinistro dell'intervallo
%   b   - estremo destro dell'intervallo
%
% Output:
%   x   - vettore (1 x n+1) contenente le ascisse di Chebyshev
   k = 0:n;
   x_cheb = cos((2*k + 1)*pi/(2*(n+1)));

   x = ((b - a) / 2) * x_cheb + (a + b) / 2;
end
```

Esercizio 22

```
nn = 1:100;
% Intervallo [0,1]
figure;
semilogy(nn, lebesgue(0, 1, nn, 1), 'b-', 'LineWidth', 1.5, ...
   'DisplayName', 'Chebyshev [0,1]');
hold on;
semilogy(nn, lebesgue(0, 1, nn, 0), 'r--', 'LineWidth', 1.5, ...
   'DisplayName', 'Equidistanti [0,1]');
hold off;
grid on;
xlabel('Grado del polinomio');
ylabel('Costante di Lebesgue');
title('Costante di Lebesgue su [0,1]');
legend('Location','best');
% Intervallo [-5,8]
figure;
semilogy(nn, lebesgue(-5, 8, nn, 1), 'b-', 'LineWidth', 1.5, ...
   'DisplayName', 'Chebyshev [-5,8]');
hold on;
semilogy(nn, lebesgue(-5, 8, nn, 0), 'r--', 'LineWidth', 1.5, ...
   'DisplayName', 'Equidistanti [-5,8]');
hold off;
grid on;
xlabel('Grado del polinomio');
ylabel('Costante di Lebesgue');
title('Costante di Lebesgue su [-5,8]');
legend('Location','best');

function ll = lebesgue(a, b, nn, type)
% lebesgue - Approssima la costante di Lebesgue per l'interpolazione
polinomiale
%
%   ll = lebesgue(a, b, nn, type)
%
% Input:
%   a, b    - estremi dell'intervallo
%   nn      - vettore contenente i gradi dei polinomi (es. 1:100)
%   type    - 0: ascisse equidistanti, 1: ascisse di Chebyshev
%
% Output:
%   ll      - vettore delle costanti di Lebesgue per ciascun grado in nn
   numPts = 10001;
   x = linspace(a, b, numPts);
   ll = nn;
```

```
    for i = 1:length(nn)
        if type == 0
            xi = linspace(a, b, nn(i));
        else
            xi = chebyshev(nn(i), a, b);
        end

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

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

function L = lagrangeBase(x, xi, idx)
% lagrangeBase - Calcola il polinomio di base L_j(x)
%
%   L = lagrangeBase(x, xi, idx)
%
% Input:
%   x   - vettore di punti in cui valutare la base
%   xi  - nodi
%   idx - indice del nodo corrispondente
%
% Output:
%   L   - polinomio di base
    L = ones(size(x));

    nLocal = length(xi) - 1;
    xii = xi(idx);

    xi_no_idx = xi([1:idx-1, idx+1:nLocal+1]);

    for k = 1:nLocal
        L = L .* (x - xi_no_idx(k)) / (xii - xi_no_idx(k));
    end
end
```


Costante di Lebesgue su [0,1]
Chebyshev [0,1]
Equidistanti [0,1]
Costante di Lebesgue
Grado del polinomio
10^{30}
10^{25}
10^{20}
10^{15}
10^{10}
10^{5}
10^{0}
0
10
20
30
40
50
60
70
80
90
100


Costante di Lebesgue su [-5,8]
Chebyshev [-5,8]
Equidistanti [-5,8]
Costante di Lebesgue
Grado del polinomio
10^{30}
10^{25}
10^{20}
10^{15}
10^{10}
10^{5}
10^{0}
0
10
20
30
40
50
60
70
80
90
100

Dal confronto tra le curve per nodi equidistanti e nodi di Chebyshev si nota che:

Le ascisse di **Chebyshev** producono una crescita molto più contenuta (sostanzialmente logaritmica) della costante di Lebesgue, in accordo con i noti risultati teorici che le identificano come nodi quasi ottimali per l'interpolazione polinomiale.

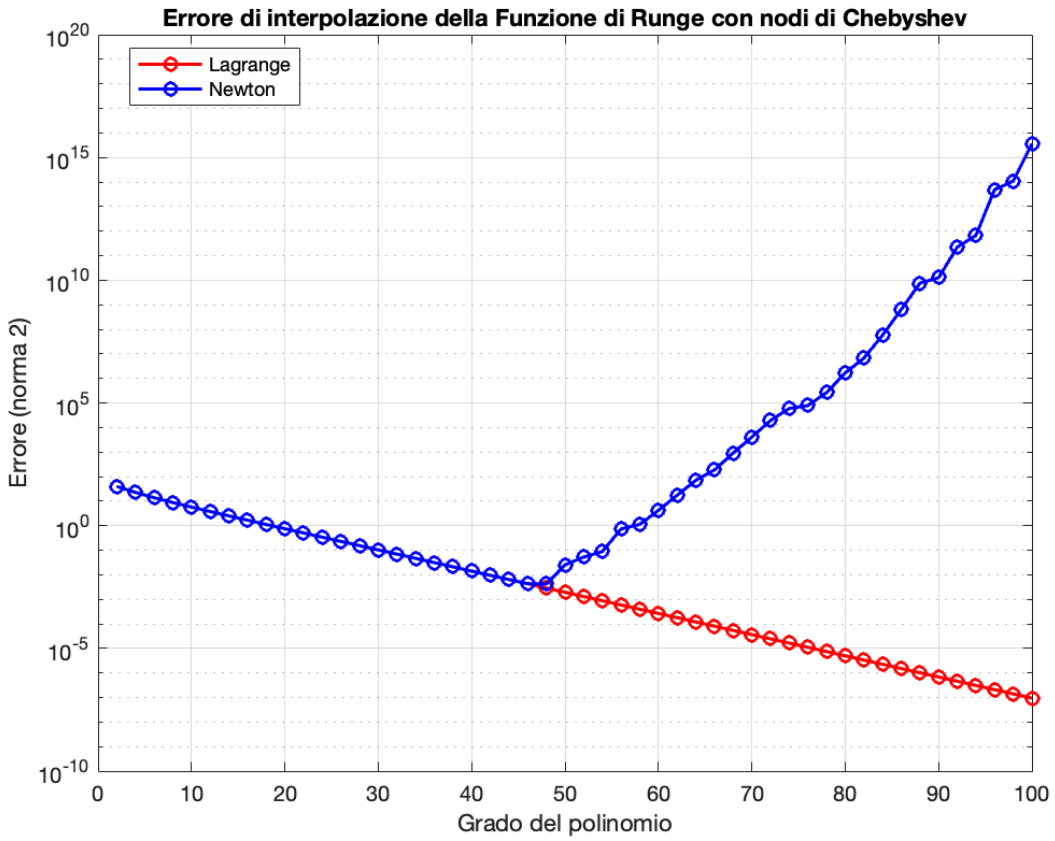
Le ascisse **equidistanti**, invece, mostrano un aumento più marcato (tendenzialmente esponenziale) della costante di Lebesgue al crescere del grado n. Questo conferma il fenomeno di Runge, ossia l'instabilità che si manifesta quando si utilizzano nodi equidistanti per polinomi di alto grado.

L'intervallo $[a, b]$ (sia $[0, 1]$ che $[-5, 8]$) non modifica il comportamento asintotico della costante: le due curve (Chebyshev ed equidistanti) si sovrappongono nelle rispettive scale, mostrando la stessa tendenza qualitativa, soltanto "traslata".

Esercizio 23

```
f = @(x) 1 ./ (1 + x.^2);
a = -5;
b =  5;
nCheb = 2:2:100;
normLagr = zeros(size(nCheb));
normNewt = zeros(size(nCheb));
x_eval = linspace(a, b, 10001);
f_eval = f(x_eval);
for i = 1:length(nCheb)
   n = nCheb(i);
   x_cheb = chebyshev(n, a, b);
   f_nodes = f(x_cheb);
   f_lag = lagrange(x_cheb, f_nodes, x_eval);
   f_newt = newton3(x_cheb, f_nodes, x_eval);
   normLagr(i) = norm(f_eval - f_lag, 2);
   normNewt(i) = norm(f_eval - f_newt, 2);
end
figure;
semilogy(nCheb, normLagr, 'r-o', 'LineWidth',1.5,
'DisplayName','Lagrange');
hold on;
semilogy(nCheb, normNewt, 'b-o', 'LineWidth',1.5,
'DisplayName','Newton');
hold off;
grid on;
legend('Location','best');
xlabel('Grado del polinomio');
ylabel('Errore (norma 2)');
```

```
title('Errore di interpolazione della Funzione di Runge con nodi di
Chebyshev');
```



L'interpolazione con **Lagrange** e nodi di Chebyshev mostra un errore in diminuzione all'aumentare del grado del polinomio, confermando la capacità di questi nodi di limitare il fenomeno di Runge.
Il metodo di **Newton**, invece, inizialmente segue lo stesso andamento, ma oltre un certo grado l'errore inizia a crescere, a causa di problemi di stabilità numerica legati al calcolo delle differenze divise.

Esercizio 24

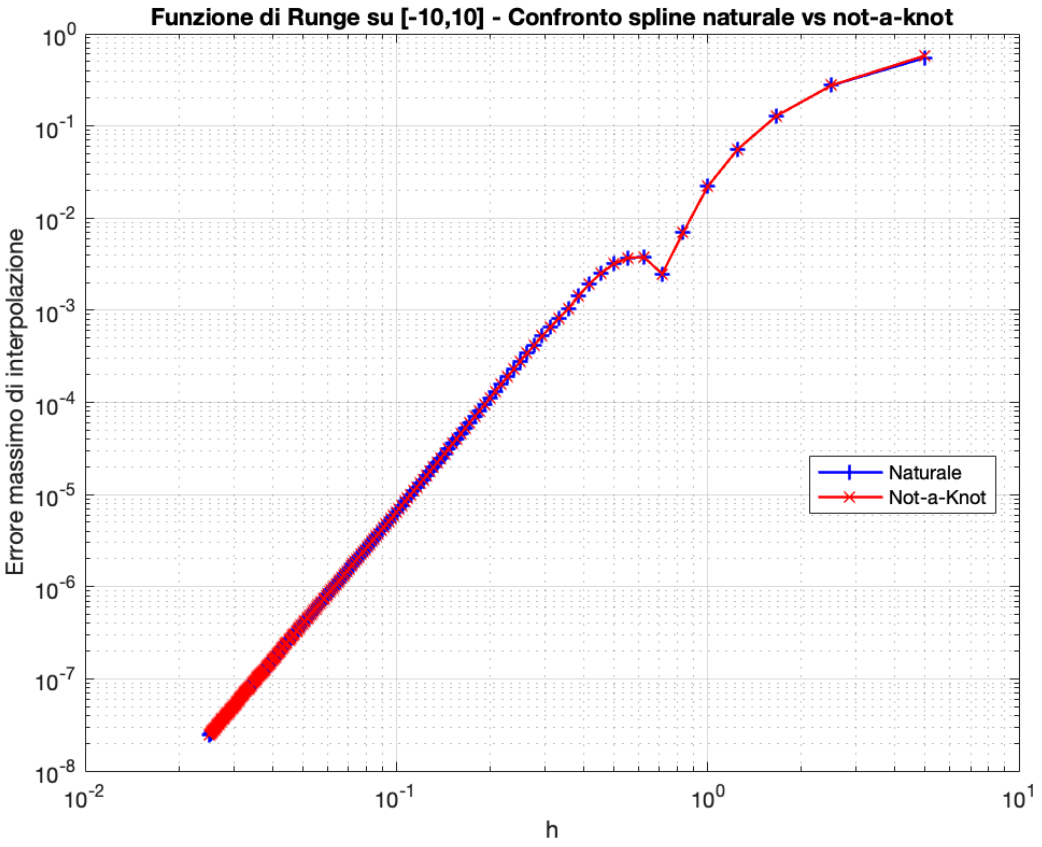
```
function YQ = spline0(X, Y, XQ)
% spline0 - Spline cubica naturale interpolante.
%
%   YQ = spline0(X, Y, XQ)
%
% Input:
%   X   - vettore delle ascisse (nodi), ordinato in modo crescente.
%   Y   - vettore dei valori corrispondenti.
%   XQ  - (opzionale) vettore dei punti in cui valutare la spline.
%
% Output:
%   YQ  - se vengono passati 2 argomenti, out è la struttura pp;
```

```
%         se vengono passati 3 argomenti, out contiene i valori della spline in
xq.
    if nargin < 2
        error('Numero insufficiente di argomenti.');
    end
    if length(X) ~= length(Y)
        error('I vettori X e Y devono avere la stessa lunghezza.');
    end
    X = X(:);
    Y = Y(:);
    n = length(X);

    if n < 2
        error('Occorrono almeno 2 punti per l''interpolazione.');
    end
    nseg = n - 1;

    h = diff(X);
    d = diff(Y) ./ h;

    N = n - 2;
    A = zeros(N, N);
    rhs = zeros(N, 1);

    for i = 1:N
        if i == 1
            A(i, i) = 2 * (h(1) + h(2));
            if N > 1
                A(i, i+1) = h(2);
            end
        elseif i == N
            A(i, i-1) = h(N);
            A(i, i) = 2 * (h(N) + h(N+1));
        else
            A(i, i-1) = h(i);
            A(i, i) = 2 * (h(i) + h(i+1));
            A(i, i+1) = h(i+1);
        end
        rhs(i) = 6 * (d(i+1) - d(i));
    end
    m = zeros(n, 1);
    if N > 0
        m(2:n-1) = A \ rhs;
    end
    coefs = zeros(nseg, 4);
    for i = 1:nseg
        a_i = (m(i+1) - m(i)) / (6 * h(i));
        b_i = m(i) / 2;
        c_i = d(i) - h(i) * (2*m(i) + m(i+1)) / 6;
        d_i = Y(i);
        coefs(i, :) = [a_i, b_i, c_i, d_i];
```

```
    end
    pp.form = 'pp';
    pp.breaks = X';
    pp.coefs = coefs;
    pp.pieces = nseg;
    pp.order = 4;
    pp.dim = 1;
    if nargin < 3
        YQ = pp;
    else
        YQ = ppval(pp, XQ);
    end
end
```

Esercizio 25

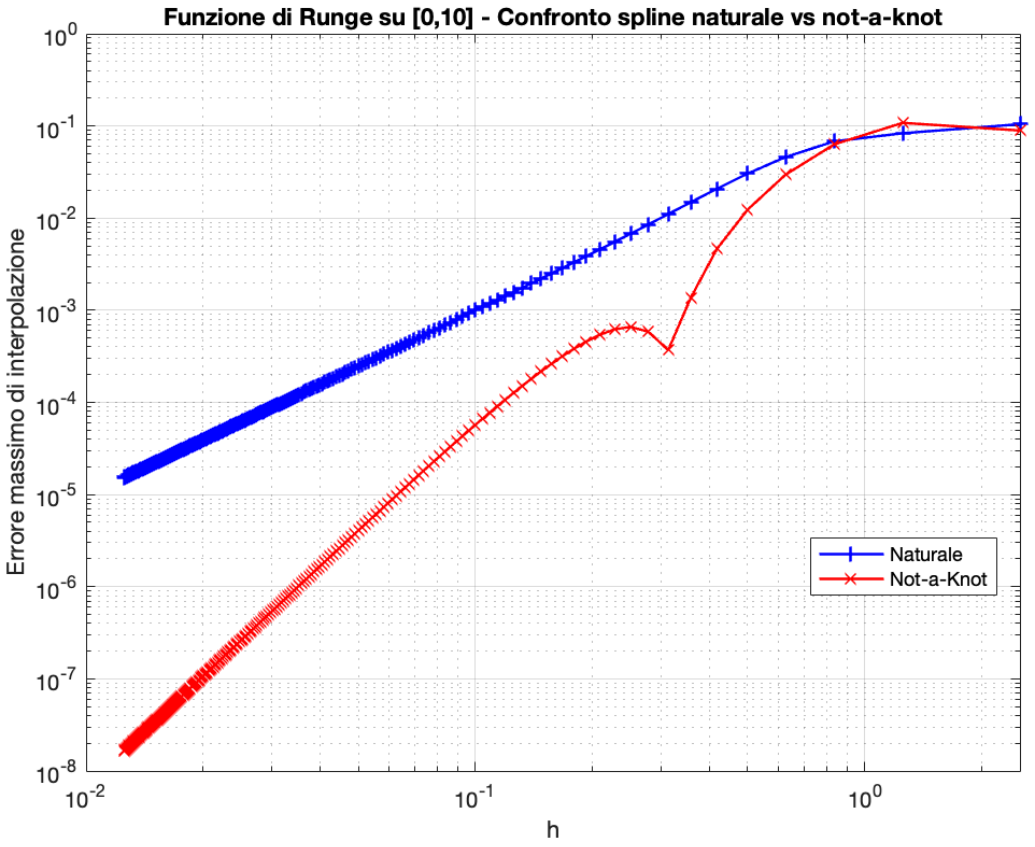
```
f = @(x) 1./(1 + x.^2);
a = -10;
b =  10;
xfine = linspace(a, b, 10001);
ffine = f(xfine);
n_values = 4:4:800;
err_nat = zeros(size(n_values));
err_nak = zeros(size(n_values));
hvals = zeros(size(n_values));
for k = 1:length(n_values)
    n = n_values(k);
    xi = linspace(a, b, n+1);
    fi = f(xi);
    snat = spline0(xi, fi, xfine);
    snak = spline(xi, fi, xfine);
    err_nat(k) = max(abs(ffine - snat));
    err_nak(k) = max(abs(ffine - snak));
    hvals(k) = (b - a) / n;
end
figure;
loglog(hvals, err_nat, 'b-+', 'LineWidth',1.2, 'MarkerFaceColor','b');
hold on;
loglog(hvals, err_nak, 'r-x', 'LineWidth',1.2, 'MarkerFaceColor','r');
hold off;
grid on;
xlabel('h');
ylabel('Errore massimo di interpolazione');
title('Funzione di Runge su [-10,10] - Confronto spline naturale vs
not-a-knot');
legend('Naturale','Not-a-Knot','Location','best');
```



Con il diminuire di h, l'errore di approssimazione delle due spline converge, diventando praticamente identico.

Esercizio 26

Il codice utilizzato per l'esercizio 26 è lo stesso di quello dell'esercizio precedente (25): l'unica modifica riguarda l'intervallo di interpolazione, impostato a $[0, 10]$ anziché $[-10, 10]$.



Osservando i risultati sul nuovo intervallo, si nota che:

La spline **Not-a-Knot** continua a offrire una decrescita dell'errore più rapida all'aumentare del numero di sottointervalli.

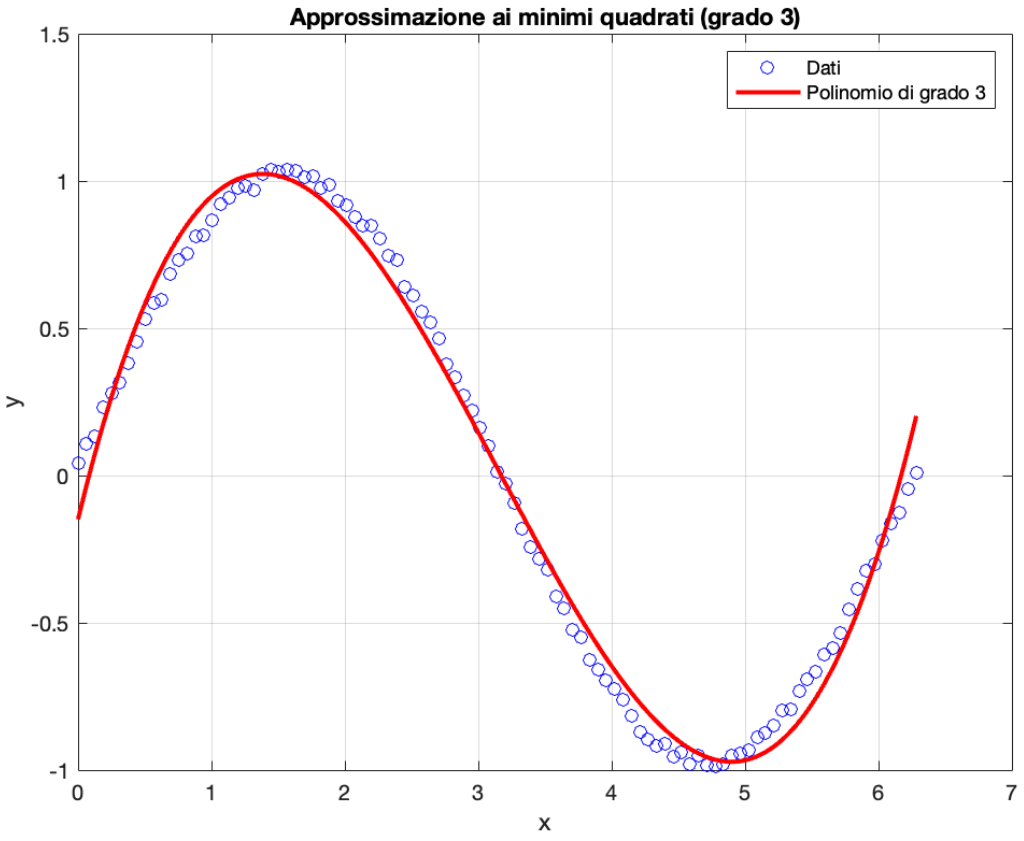
La spline **naturale** risulta invece meno accurata rispetto alla Not-a-Knot, e in questo intervallo l'errore appare persino più elevato di quello osservato nell'esercizio precedente (dove l'intervallo era più esteso).

In conclusione, Not-a-Knot si conferma più efficace nel contenere l'errore di interpolazione, anche quando si modifica l'intervallo.

Esercizio 27

```
rng(0);
xi = linspace(0, 2*pi, 101);
yi = sin(xi) + rand(size(xi)) *.05;
coeff = polyfit(xi, yi, 3);
disp('Coefficienti del polinomio di grado 3:');
disp(coeff);
xi_fit = linspace(0, 2*pi, 1000);
yi_fit = polyval(coeff, xi_fit);
```

```
figure;
plot(xi, yi, 'bo', 'MarkerSize', 6, 'DisplayName', 'Dati');
hold on;
plot(xi_fit, yi_fit, 'r-', 'LineWidth', 2, 'DisplayName', 'Polinomio di grado
3');
xlabel('x');
ylabel('y');
title('Approssimazione ai minimi quadrati (grado 3)');
legend show;
grid on;
```



Esercizio 28

```
gradi = [1, 2, 3, 4, 5, 6, 7, 9];
for n = gradi
   w = newtonCotesPesi(n);

   fprintf('Grado n = %d (n+1 = %d nodi):\n', n, n+1);
   for i = 1:length(w)
       fprintf('  w_%d = %s\n', i-1, rats(w(i)));
   end
   fprintf('--------------------------------\n');
end
```

```m
function w = newtonCotesPesi(n)
% newtonCotesPesi - Restituisce i pesi della formula di Newton-Cotes di grado n.
%
%   w = newtonCotesPesi(n)
%
% Input:
%   n   - grado compreso tra 1 e 9 con 8 escluso
%
% Output:
%   W   - pesi della formula di Newton-Cotes di grado n
   if (n < 1) || (n > 9) || (n == 8)
       error("Grado errato: n deve essere in [1..9] e diverso da 8.");
   end
   w = zeros(1, n+1);
   nodi = 0:n;
   for iNodo = nodi
       altriNodi = [nodi(1:iNodo), nodi(iNodo+2:end)];
       diffVals = iNodo - altriNodi;
       denom = prod(diffVals);
       pCoeffs = poly(altriNodi);
       pCoeffs = [pCoeffs ./ ((n+1):-1:1), 0];
       pVal = polyval(pCoeffs, n);
       w(iNodo+1) = pVal / denom;
   end
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	$\frac{95}{288}, \frac{125}{96}, \frac{125}{144}, \frac{125}{144}, \frac{125}{96}, \frac{95}{288}$
6	$\frac{41}{140}, \frac{54}{35}, \frac{27}{140}, \frac{68}{35}, \frac{27}{140}, \frac{54}{35}, \frac{41}{140}$
7	$\frac{1073}{3527}, \frac{810}{559}, \frac{343}{640}, \frac{649}{536}, \frac{649}{536}, \frac{343}{640}, \frac{810}{559}, \frac{1073}{3527}$
9	$\frac{130}{453}, \frac{1374}{869}, \frac{243}{2240}, \frac{5287}{2721}, \frac{704}{1213}, \frac{704}{1213}, \frac{5594}{2879}, \frac{243}{2240}, \frac{1374}{869}, \frac{130}{453}$

Esercizio 29

```
function [If, err] = composita(fun, a, b, k, n)
% composita - Applica la formula composta di Newton-Cotes di grado k
% per approssimare ∫[a,b] fun(x) dx.
%
%   [If, err] = composita(fun, a, b, k, n)
%
% Input:
%   fun - handle alla funzione da integrare, accetta input vettoriali.
%   a   - estremo di integrazione a < b
%   b   - estremo di integrazione b >= a
%   k   - grado della formula di Newton-Cotes chiusa (k+1 nodi).
%   n   - numero di sottointervalli (deve essere multiplo di k).
%         l'intervallo [a,b] è diviso in n parti di uguale ampiezza.
%
% Output:
%   If   - approssimazione dell'integrale di fun su [a,b].
%   err  - stima dell'errore di quadratura (basata su un semplice
%          confronto di Richardson, se possibile).
   if a > b
       error("Estremi intervallo non validi (a deve essere <= b).");
   end
   if k < 1
       error("Grado k errato: deve essere >= 1.");
   end
   tolleranza = 1e-3;
   ordineShift = 1 + mod(k,2);
   coeff = calcolaCoeffGrad(k);
   xx = linspace(a, b, n+1);
   fx = feval(fun, xx);
   h = (b - a) / n;
   IfO = h * sum( fx(1 : k+1) .* coeff(1 : k+1) );
   err = tolleranza + eps;
   while tolleranza < err
       n = n * 2;
       xx = linspace(a, b, n+1);
       fx(1 : 2 : n+1) = fx(1 : 1 : n/2 + 1);
       fx(2 : 2 : n)   = fun(xx(2 : 2 : n));
       h = (b - a) / n;
       IfN = 0;
       for i = 1 : (k+1)
           IfN = IfN + h * sum(fx(i : k : n)) * coeff(i);
       end
       IfN = IfN + h * fx(n+1) * coeff(k+1);
       fattore = 2^(k + ordineShift) - 1;
```

```
        err = abs(IfN - IfO) / fattore;
        IfO = IfN;
    end
    If = IfO;
end
function coef = calcolaCoeffGrad(n)
% calcolaCoeffGrad - Restituisce i coefficienti di Newton-Cotes
% di grado n in un vettore colonna di dimensione (n+1) x 1.
    if n <= 0
        error('Valore del grado di Newton-Cotes non valido (n > 0
richiesto).');
    end
    coef = zeros(n+1, 1);
    if mod(n,2) == 0
        for i = 0 : (n/2 - 1)
            coef(i+1) = calcolaCoeff(i, n);
        end
        coef(n/2 + 1) = n - 2 * sum(coef);
        coef((n/2)+1 : n+1) = coef((n/2)+1 : -1 : 1);
    else
        for i = 0 : (round(n/2,0) - 2)
            coef(i+1) = calcolaCoeff(i, n);
        end
        coef(round(n/2,0)) = (n - 2*sum(coef)) / 2;
        coef(round(n/2,0) + 1 : n+1) = coef(round(n/2,0) : -1 : 1);
    end
end
function cVal = calcolaCoeff(i, n)
% calcolaCoeff - Calcola un singolo coefficiente della formula
% di Newton-Cotes di grado n, corrispondente al nodo i.
    diffVect = i - [0 : i-1, i+1 : n];
    denom = prod(diffVect);
    p = poly([0 : i-1, i+1 : n]);
    p = [ p ./ ((n+1):-1:1), 0 ];
    num = polyval(p, n);
    cVal = num / denom;
end
```

Esercizio 30

```
I_esatto = (exp(3) - 1) / 3;
kVals = [1, 2, 3, 6];
n = 12;
fprintf('   k    Appross        Errore (If - esatto)   Err stima\n');
for k = kVals
   [If, errStima] = composita(@(x) exp(3*x), 0, 1, k, n);
   errVero = abs(If - I_esatto);
   fprintf(' %3d  %12.6f  %18.3e  %12.3e\n', k, If, errVero, errStima);
end
```

k	Approssimazione Integrale	Errore reale	Errore stimato
1	6.363916	$2.071e^{-3}$	$8.872e^{-4}$
2	6.361846	$5.390e^{-7}$	$1.153e^{-6}$
3	6.361847	$1.212e^{-6}$	$5.849e^{-7}$
6	6.361846	$1.580e^{-12}$	$3.124e^{-12}$