Elaborato Calcolo Numerico 2017-2018

Bertini Gabriele - matricola 5793664
Pratesi Mariti Lorenzo - matricola

11 agosto 2018

Indice

Capitolo 1

Errori ed aritmetica finita

Capitolo 2

Radici di una equazione

Capitolo 3

Sistemi lineari e non lineari

Capitolo 4

Approssimazione di funzioni

4.1 Esercizio 1

Prova

```
function y = lagrange(xi, fi, x)
% y = lagrange(xi,fi,x)
% Implementa il calcolo del polinomio interpolante
% di grado n in forma di Lagrange
% Input:
%  xi: vettore delle ascisse di interpolazione
%  fi: vettore dei valori della funzione su x
%  x: vettore dei punti in cui valutare il polinomio
%
% Output:
%  y: vettore dei valori del polinomio valutato sui punti x
   if isempty(x), error("Vettore x vuoto!"), end
   if length(xi)~=length(fi), error("Diversa lunghezza dei vettori xi e fi!"), end
    n=length(xi)-1;
    m=length(x);
    y=zeros(size(x));
    for i=1:m
        for j=1:n+1
            p=1;
            for k=1:n+1
                if j~=k
                    p=p*(x(i)-xi(k))/(xi(j)-xi(k));
                end
            end
            y(i)=y(i)+fi(j)*p;
        end
    end
return
end
```

4.2 Esercizio 2

Prova

```matlab
function y = newton(xi,fi,x)
% y = newton(xi,fi,x)
% Implementa il calcolo del polinomio interpolante
% di grado n in forma di Newton
% Input:
%  xi: vettore delle ascisse di interpolazione
%  fi: vettore dei valori della funzione su x
%  x: vettore dei punti in cui valutare il polinomio
%
% Output:
%  y: vettore dei valori del polinomio valutato sui punti x.
    if isempty(x), error('Vettore x vuoto!'), end
    if isempty(xi), error('Vettore xi vuoto!'), end
    if length(fi)~=length(xi), error('Vettori fi ed xi non compatibili!'), end
    dd = diffdivN(xi,fi);
    y = hornerGen(xi,dd,x);
return
end

function fi = diffdivN(xi,fi)
% f = diffdivN(xi,fi)
% Calcolo delle differenze divise per il polinomio di Newton
% Questo metodo prende in input:
%  xi: vettore delle ascisse
%  fi: vettore dei valori della funzione
%
% E restituisce:
%  fi: vettore contenente le differenze divise f[x0],f[x1,x2]...f[x0...xn]
    % Assumiamo che le ascisse siano tutte distinte.
    n = length(xi)-1;
    for j = 1:n
        for i = n+1:-1:j+1
            fi(i) = (fi(i)-fi(i-1))/(xi(i)-xi(i-j));
        end
    end
return
end

function p = hornerGen(xi,fi,x)
% p = hornerGen(xi,fi,x)
% Algoritmo di horner generalizzato
% Questo metodo prende in input:
%  fi: array dei coefficienti
%  xi: array degli zeri del polinomio
%  x: array di punti di valutazione del polinomio
% E restituisce la valutazione del polinomio
% pr-1(x)+fwr(x) nei punti x0
    n = length(xi)-1;
    p = fi(n+1)*ones(size(x));
    for k = 1:length(x)
        for i = n:-1:1
            p(k) = p(k)*(x(k)-xi(i))+fi(i);
        end
    end
return
end
```

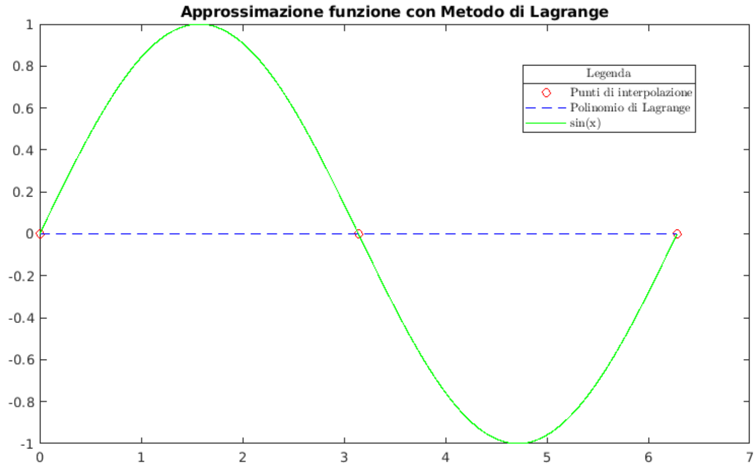
4.3 Esercizio 3

Prova

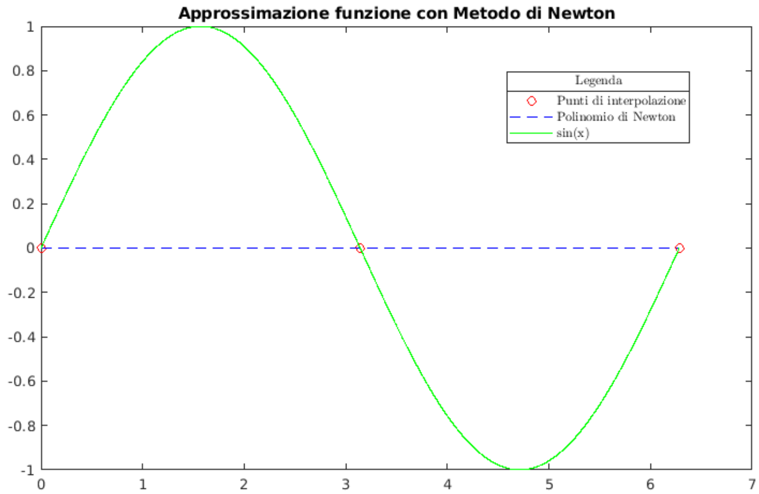
```
function y = hermite(xi,fi,fi1,x)
% y = hermite(xi,fi,f1i,x)
% Calcola il polinomio interpolante di Hermite
% Input:
%  xi: vettore delle ascisse di interpolazione
%  fi: vettore dei valori della funzione su x
%  fi1: vettore della derivata su x
%  x: vettore dei punti in cui valutare il polinomio
%
% Output:
%  y: vettore dei valori del polinomio valutato sui punti x.
    if isempty(x), error('Vettore x vuoto!'), end
    if isempty(xi), error('Vettore xi vuoto!'), end
    if length(xi)~=length(fi) || length(xi)~=length(fi1)
        error('Vettori xi, fi e fi1 di lunghezza diversa!')
    end
    n = length(xi)-1;
    xi = reshape([xi; xi], [], 1)';
    fi = reshape([fi; fi1], [], 1)';
    dd = diffdivH(xi,fi);
    y = hornerGen(xi,dd,x);
return
end

function fi = diffdivH(xi,fi)
% fi = diffdivH(xi,fi)
% Calcola differenze divise per polinomio di Hermite
% Questo metodo prende in input:
%  xi: vettore delle ascisse
%  fi: vettore dei valori della funzione
%
% E restituisce:
%  fi: vettore contenente le differenze divise f[x0],f[x0,x0],...,f[x0,...,xn]
    n = length(xi)-1; %2*m+1
    for i = n:-2:3
        fi(i) = (fi(i)-fi(i-2))/(xi(i)-xi(i-1));
    end
    for j = 2:n
        for i = n+1:-1:j+1
            fi(i) = (fi(i)-fi(i-1))/(xi(i)-xi(i-j));
        end
    end
return
end

function p = hornerGen(xi,fi,x)
% p = hornerGen(xi,fi,x)
% Algoritmo di horner generalizzato
% Questo metodo prende in input:
%  fi: array dei coefficienti
%  xi: array degli zeri del polinomio
%  x: array di punti di valutazione del polinomio
% E restituisce la valutazione del polinomio
% pr-1(x)+fwr(x) nei punti x0
    n = length(xi)-1;
    p = fi(n+1)*ones(size(x));
    for k = 1:length(x)
        for i = n:-1:1
            p(k) = p(k)*(x(k)-xi(i))+fi(i);
        end
    end
return
```

```
end
```

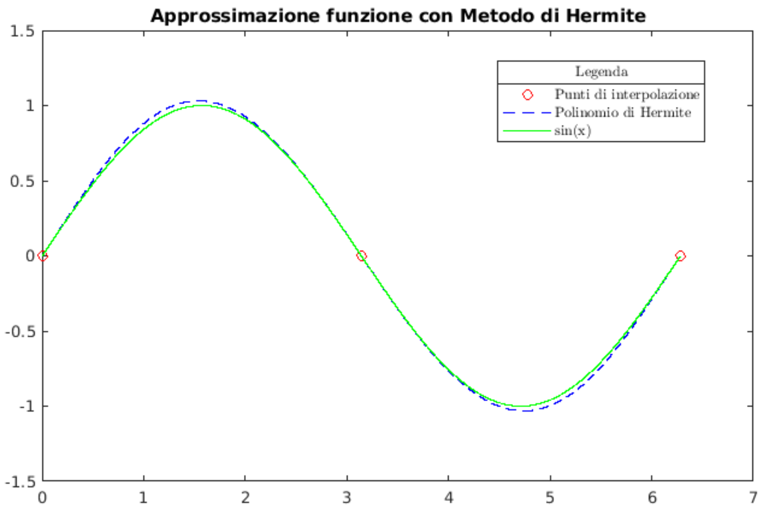
4.4 Esercizio 4



Prova



Prova



Prova

4.5 Esercizio 5

Prova

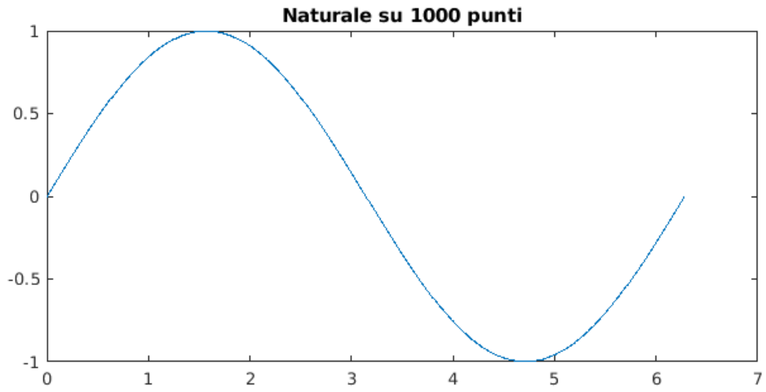
```
function y = spline3( xi, fi, x, tipo )
% y = spline3( xi, fi, x, tipo )
% Calcola e valuta i valori della spline cubica naturale o not-a-knot
% (rispettivamente per tipo uguale a 0 o 1) delle coppie di dati assegnati.
% Input:
%    xi: ascisse di interpolazione
%    fi: valori della funzione, valutati nelle ascisse xi
%    x: punti di valutazione della spline
%    tipo:
%        - nat identifica la spline naturale
%        - nak identifica la spline not-a-knot
%        - un valore diverso restituisce errore
% Output:
%    y: valutazione dei punti x calcolati sulla spline richiesta
    if tipo == "nat"
        mi = sistemaSplineNaturale(xi, fi);
        y = valutaSplineNaturale(xi, fi, mi, x);
    elseif tipo == "nak"
        mi = sistemaSplineNotAKnot(xi, fi);
        y = valutaSplineNotAKnot(xi, fi, mi, x);
    else
        error('Il tipo indicato non Ã¨ corretto.');
    end
    return
end

function fin = differenzeDivise(x, f)
% Input:
% - f: funzione;
% - x: vettore delle ascisse.
% Output:
% - fin: l'ultima differenza divisa calcolata, corrispondente a quella che
%        include tutte le ascisse.
```

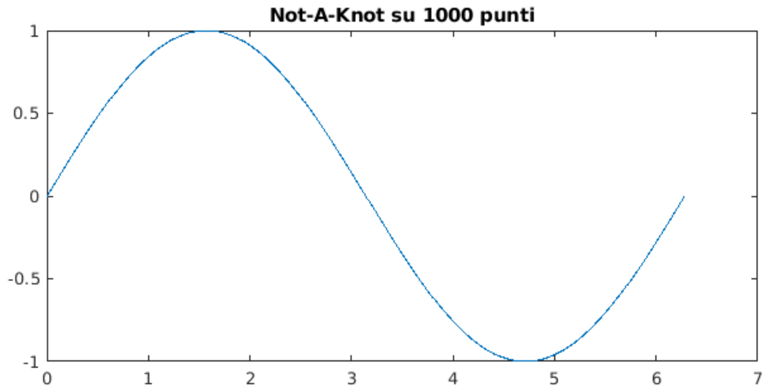
```
    n = length(x);
    for i = 1:n-1
        for j = n:-1:i+1
            f(j)=(f(j)-f(j-1))/(x(j)-x(j-i));
        end
    end
    fin = f(end);
end

function m = sistemaSplineNaturale( xi , fi )
% Input:
%    xi: ascisse di interpolazione
%    fi: valori della funzione, valutati nelle ascisse xi
% Output:
%    m: coefficienti necessari a calcolare l'espressione della spline
%    cubica

    % calcolo delle phi, zi e diff_div (lunghi n-1)
    % n:= grado del polinomio interpolante
    n = length(xi)-1;
    phi = zeros(n-2, 1);
    zi = zeros(n-2, 1);
    dd = zeros(n-1, 1);
    hi = xi(2) - xi(1);
    hi1 = xi(3) - xi(2);
    phi(1) = hi/(hi+hi1);
    zi(1) = hi1/(hi+hi1);
    dd(1) = differenzeDivise(xi(1:3), fi(1:3));
    for i = 2:n-2
        hi = xi(i) - xi(i-1);
        hi1 = xi(i+1) - xi(i);
        phi(i) = hi/(hi+hi1);
        zi(i) = hi1/(hi+hi1);
        dd(i) = differenzeDivise(xi(i-1:i+1), fi(i-1:i+1));
    end
    dd(i+1) = differenzeDivise(xi(i:i+2), fi(i:i+2));
    % calcolo delle li ed ui che completano il sistema lineare della spline
    % naturale
    u(1) = 2;
    for i = 2:n-1
        l(i) = phi(i-1)/u(i-1);
        u(i) = 2-l(i)*zi(i-1);
    end
    % risoluzione del sistema lineare LU per trovare gli m
    % 1) L*y = 6*b
    % il vettore dd delle diffDiv viene sovrascritto
    dd(1) = 6*dd(1);
    dd(2:n-1) = 6*dd(2:n-1)-l(2:n-1)*dd(1:n-2);
    % 2) U*m = y
    m(n-1) = dd(n-1)/u(n-1);
    m(n-2:-1:1) = (dd(n-2:-1:1)-zi(n-2:-1:1) * m(n-1:-1:2))/u(n-2:-1:1);
end

function y = valutaSplineNaturale(xi, fi, mi, x)
    % Input:
    %    xi: ascisse di interpolazione
    %    fi: valori della funzione, valutati nelle ascisse xi
    %    mi: coefficienti necessari a calcolare l'espressione della spline
    %    x: punti in cui valutare la spline
    % Output:
    %    y: valutazioni nei punti x della spline.

```

```
    if xi(1)>x(1) || xi(end)<x(end)
        error('I punti di valutazione non rientrano nel dominio della spline.');
    end
    if length(xi)~=length(fi)
        error('I vettori xi e fi devono avere la stessa lunghezza.');
    end
    % raccolgo tutti i sottoinsiemi di punti da valutare con le relative
    % functions, i punti x vegono ordinati
    mi = [0  mi  0];
    sort(x);
    y = zeros(length(x),1);
    lastIndex = 1;
    k = 2;
    for j = 1 : length(x)
        if x(j) <= xi(k-1)
            j = j + 1;
        else
            if j ~= lastIndex
                % calcolo la spline relativa al k-1Â° intervallo
                hi = xi(k) - xi(k-1);
                ri = fi(k-1) - ((hi^2)/6) * mi(k-1);
                qi = (fi(k) - fi(k-1))/hi - (hi/6) * ...
                    (mi(k) - mi(k-1));
                spline = @(x) (((x - xi(k-1)).^3) .* mi(k) + ((xi(k) - x).^3) .* mi(k-1))./(6*hi).
                  + qi .* (x - xi(k-1)) + ri;
                % calcolo le valutazioni della spline
                y(lastIndex : j-1) = spline(x(lastIndex : j-1));
                lastIndex = j;
            end
            k = k+1;
        end
    end
    % valutazione degli ultimi punti
    if j ~= lastIndex
        hi = xi(end) - xi(end-1);
        ri = fi(end-1) - ((hi^2)/6) * mi(end-1);
        qi = (fi(end) - fi(end-1))/hi - (hi/6) * (-mi(end-1));
        spline = @(x) (((x - xi(end-1)).^3) .* mi(end) + ((xi(end) - x).^3) .* mi(end-1))./(6*hi).
                  + qi .* (x - xi(end-1)) + ri;
        y(lastIndex:j-1) = spline(x(lastIndex:j-1));
    end

end

function m = sistemaSplineNotAKnot(xi, fi)
% Calcola i coefficienti m da applicare all'espressione della spline cubica
% not-a-knot.
% Input:
%    xi: ascisse di interpolazione
%    fi: valori della funzione, valutati nelle ascisse xi
% Output:
%    m: coefficienti necessari a calcolare l'espressione della spline
%    not-a-knot

    % calcolo delle phi, zi e diff_div
    % n:= grado del polinomio interpolante
    n = length(xi)-1;
    hi = xi(2) - xi(1);
    hi1 = xi(3) - xi(2);
    phi(1) = hi/(hi+hi1);
    zi(1) = hi1/(hi+hi1);
    diffDiv(1) = differenzeDivise(xi(1:3), fi(1:3));
```

```
    for i = 2:n-1
        hi = xi(i) - xi(i-1);
        hi1 = xi(i+1) - xi(i);
        phi(i) = hi/(hi+hi1);
        zi(i) = hi1/(hi+hi1);
        diffDiv(i) = differenzeDivise(xi(i-1:i+1), fi(i-1:i+1));
    end
    diffDiv(i+1) = differenzeDivise(xi(i:i+2), fi(i:i+2));
    % espando diffDiv a vettore di lunghezza n+1
    diffDiv = [6*diffDiv(1), 6*diffDiv, 6*diffDiv(end)];
    % i = 1
    u(1) = 1;
    % i = 2
    w(1) = 0;
    l(1) = phi(1) / u(1);
    u(2) = 2 - phi(1);
    % i = 3
    w(2) = zi(1) - phi(1);
    l(2) = phi(2) / u(2);
    u(3) = 2 - l(2)*w(2);
    % i = 4:n-1
    for i = 4:n-1
        w(i-1) = zi(i-2);
        l(i-1) = phi(i-1) / u(i-1);
        u(i) = 2 - l(i-1)*w(i-1);
    end
    % i = n
    w(n-1) = zi(n-2);
    l(n) = (phi(n-1) - zi(n-1)) / u(n-1);
    u(n) = 2 - zi(n-1) - l(n)*w(n-1);
    % i = n+1
    w(n) = zi(n-1);
    l(n+1) = 0;
    u(n+1) = 1;
    % 1) Ly = 6*diffDiv
    y(1) = diffDiv(1);
    for i = 2:n+1
        y(i) = diffDiv(i) - l(i-1)*y(i-1);
    end
    % 2) Um = y
    m = zeros(n+1, 1);
    m(n+1) = y(n+1) / u(n+1);
    for i = n:-1:1
        m(i) = (y(i) - w(i) * m(i+1)) / u(i);
    end
    m(1) = m(1) - m(2) - m(3);
    m(end)= m(n+1)-m(n)-m(n-1);
    return
end

function y = valutaSplineNotAKnot(xi, fi, mi, x)
% Valuta i punti della spline cubica not-a-knot nei punti assegnati x
% Input:
%    xi: ascisse di interpolazione
%    fi: valori della funzione, valutati nelle ascisse xi
%    mi: coefficienti necessari a calcolare l'espressione della spline
%    x: punti in cui valutare la spline
% Output:
%    y: valutazioni nei punti x della spline.

    if xi(1)>x(1) || xi(end)<x(end)
        error('I punti di valutazione non rientrano nel dominio della spline.');
```

```
    end
    if length(xi)~=length(fi)
        error('I vettori xi e fi devono avere la stessa lunghezza.');
    end
    % raccolgo tutti i sottoinsiemi di punti da valutare con le relative
    % functions, i punti x vegono ordinati
    sort(x);
    y = zeros(length(x),1);
    lastIndex = 1;
    k = 2;
    for j = 1 : length(x)
        if x(j) <= xi(k-1)
            j = j + 1;
        else
            if j ~= lastIndex
                % calcolo la spline relativa al k-1Â° intervallo
                hi = xi(k) - xi(k-1);
                ri = fi(k-1) - ((hi^2)/6) * mi(k-1);
                qi = (fi(k) - fi(k-1))/hi - (hi/6) * ...
                    (mi(k) - mi(k-1));
                spline = @(x) (((x - xi(k-1)).^3) .* mi(k) + ((xi(k) - x).^3) .* mi(k-1))./(6*hi).
                  + qi .* (x - xi(k-1)) + ri;
                % calcolo le valutazioni della spline
                y(lastIndex : j-1) = spline(x(lastIndex : j-1));
                lastIndex = j;
            end
            k = k+1;
        end
    end
    % valutazione degli ultimi punti
    if j ~= lastIndex
        hi = xi(end) - xi(end-1);
        ri = fi(end-1) - ((hi^2)/6) * mi(end-1);
        qi = (fi(end) - fi(end-1))/hi - (hi/6) * (-mi(end-1));
        spline = @(x) (((x - xi(end-1)).^3) .* mi(end) + ((xi(end) - x).^3) .* mi(end-1))./(6*hi).
                  + qi .* (x - xi(end-1)) + ri;
        y(lastIndex:j-1) = spline(x(lastIndex:j-1));
    end
end
```





Prova

4.6 Esercizio 6

Prova

```
function xi = ceby(n, a, b)
% xi = ceby(n, a, b)
% Implementa il calcolo delle ascisse di Chebyshev
% per il polinomio di grado n su [a,b]
% Input:
%  n: grado del polinomio interpolante
%  a: estremo sinistro dell'intervallo
%  b: estremo destro dell'intervallo
%
% Output:
%  xi: ascisse di Chebyshev
    if n<=0
        error('Inserire numero maggiore di 0!')
    end
    xi = cos((2*[0:n]+1)*pi/(2*n+2));
    xi = ((a+b)+(b-a)*xi)/2;
```

```
    return
end
```

4.7 Esercizio 7

Prova

4.8 Esercizio 8

Prova

4.9 Esercizio 9

Prova

4.10 Esercizio 10

Capitolo 5

Formule di quadratura

5.1 Esercizio 1

Prova

```
function If = trapcomp(n, a, b, fun)
% If = trapcomp(n, a, b, fun)
% Calcola l'integrale della funzione, nell'intervallo prescelto,
% usando la formula dei trapezi composita.
% Input:
 %  n: intero positivo, indica num intervalli in [a,b]
 %  a: estremo sinistro
 %  b: estremo destro
 %  fun: funzione integranda
% Output:
 %  If: approssimazione dell'integrale definito della funzione
    if n <= 0, error('Numero intervalli non valido.'), end
    if a >= b, error('Intervallo di integrazione non valido.'), end
    x = linspace(a, b, n+1);
    f = feval(fun, x);
    If = ((b-a)/n)*(f(1)/2 + sum(f(2:n)) + f(n+1)/2);
end
```

5.2 Esercizio 2

Prova

```
function If = simpcomp(n, a, b, fun)
% If = simpcomp(n, a, b, fun)
% Calcola l'integrale della funzione, nell'intervallo prescelto,
% usando la formula di Simpson composita.
% Input:
 %  n: intero positivo pari, indica num intervalli in [a,b]
 %  a: estremo sinistro
 %  b: estremo destro
 %  fun: funzione integranda
% Output:
 %  If: approssimazione dell'integrale definito della funzione
    if mod(n,2) ~= 0
```

```
        error('Il numero di intervalli deve essere pari.')
    end
    if n <= 0
        error('Numero di intervalli non valido.')
    end
    if a >= b, error('Intervallo di integrazione non valido.'), end
    x = linspace(a, b, n+1);
    f = feval(fun, x);
    If = f(1) + f(n+1);
    If = (If + 4*sum(f(2:2:n)) + 2*sum(f(3:2:n-1)))*(b-a)/(3*n);
end
```

5.3 Esercizio 3

Prova

```
function If = trapad(a, b, fun, tol, fa, fb)
% If = trapad(a, b, fun, tol)
% Calcola ricorsivamente l'integrale della funzione, nell'intervallo prescelto,
% usando la formula dei trapezi adattiva.
% Input:
 %  a: estremo sinistro
 %  b: estremo destro
 %  fun: funzione integranda
 %  tol: tolleranza
% Output:
 %  If: approssimazione dell'integrale definito della funzione
    if a >= b, error('Intervallo di integrazione non valido.'), end
    if nargin <= 4
        fa = feval(fun, a);
        fb = feval(fun, b);
    end
    h = b - a;
    x1 = (a+b)/2;
    f1 = feval(fun, x1);
    I1 = (h/2)*(fa + fb);
    If = (I1 + h*f1)/2;
    err = abs(If - I1)/3;
    if err > tol
        If = trapad(a, x1, fun, tol/2, fa, f1) ...
           + trapad(x1, b, fun, tol/2, f1, fb);
    end
end
```

5.4 Esercizio 4

Prova

```
function If = simpad(a, b, fun, tol, fa, f1, fb)
% If = simpad(a, b, fun, tol)
% Calcola ricorsivamente l'integrale della funzione, nell'intervallo prescelto,
% usando la formula di Simpson adattiva.
% Input:
 %  a: estremo sinistro
 %  b: estremo destro
```

```
 %  fun: funzion integranda
 %  tol: tolleranza
% Output:
 %  If: approssimazione dell'integrale definito della funzione
    if a >= b, error('Intervallo di integrazione non valido.'), end
    x1 = (a+b)/2;
    if nargin <= 4
        fa = feval(fun, a);
        fb = feval(fun, b);
        f1 = feval(fun, x1);
    end
    h = (b - a)/6;
    I1 = h*(fa + 4*f1 + fb);
    f2 = feval(fun, (a+x1)/2);
    f3 = feval(fun, (x1+b)/2);
    If = .5*h*(fa + 4*f2 + 2*f1 + 4*f3 + fb);
    err = abs(If - I1)/15;
    if err > tol
        If = simpad(a, x1, fun, tol/2, fa, f2, f1)...
            + simpad(x1, b, fun, tol/2, f1, f3, fb);
    end
end
```

5.5 Esercizio 5

Prova

Capitolo 6

Calcolo del Google pagerank. Risoluzione iterativa di sistemi lineari

6.1 Esercizio 1

Prova

```
function A = matriceSparsa(n)
% A = matriceSparsa(n)
% Genera la matrice quadrata sparsa nxn, con n maggiore di 10.
%
% Input:
%     - n: numero di righe/colonne della matrice quadrata sparsa
% Output:
%     - A: matrice quadrata nxn sparsa
    %n deve essere maggiore di 10
    if n <= 10
        error('n deve essere maggiore di 10.');
    end

    d = ones(n,1)*4;
    A = spdiags(d,0,n,n);

    d = ones(n,1)*(-1);
    A = spdiags(d,1,A);
    A = spdiags(d,-1,A);

    if n > 10
        A = spdiags(d,10,A);
        A = spdiags(d,-10,A);
    end
end
```

6.2 Esercizio 2

Prova

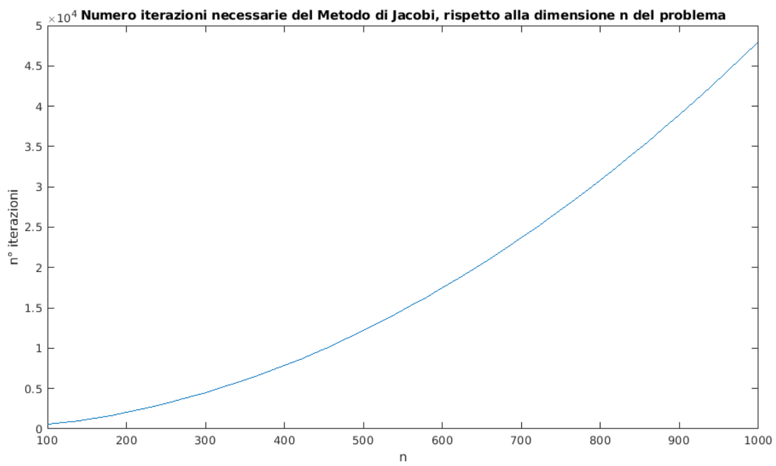
```
function [lambda, i] = potenze(A, tol, x0, maxit)
% [lambda, i] = metodoPotenze(A, tol, [x0, maxit])
% Restituisce l'autovalore dominante della matrice A e il numero di
% iterazioni necessarie per calcolarlo.
%
% Input:
%     - A: matrice utilizzata per il calcolo
%     - tol: tolleranza dell' approssimazione
%     - [x0]: vettore di partenza
%     - [maxit]: numero massimo di iterazioni
% Output:
%     - lambda: matrice quadrata nxn sparsa
%     - i: numero di iterazioni
    [m,n] = size(A);
    if m ~= n
        error('La matrice deve essere quadrata.');
    end
    if x0(:)==0
        error('Il vettore x0 non puo avere esclusivamente elementi nulli.');
    end % if da eliminare per calcolo con matrice

    if nargin <= 2
        x = rand(n, 1);
    else
        x = x0;
    end
    if nargin <= 3
        maxit = 100*2*round(-log(tol));
    end
    x = x0; % da eliminare per calcolo con matrice
    x = x / norm(x);
    lambda = inf;
    for i=1:maxit
        lambda0 = lambda;
        v = A * x;
        lambda = x' * v;
        err = abs(lambda - lambda0);
        if err <= tol
            break
        end
        x = v/norm(v);
    end
    if err > tol
        warning("Raggiunto maxit.");
    end
    return
end
```

6.3 Esercizio 3

Prova

```
function [x,i,nr] = jacobi(A, b, tol, x0, maxit)
% [x,i] = jacobi(A, b, tol, [xo, maxit])
% Restituisce la soluzione del sistema lineare Ax=b approssimata con il
% metodo di Jacobi e il numero di iterazioni eseguite.
% Input:
%     - A: matrice utilizzata per il calcolo
```

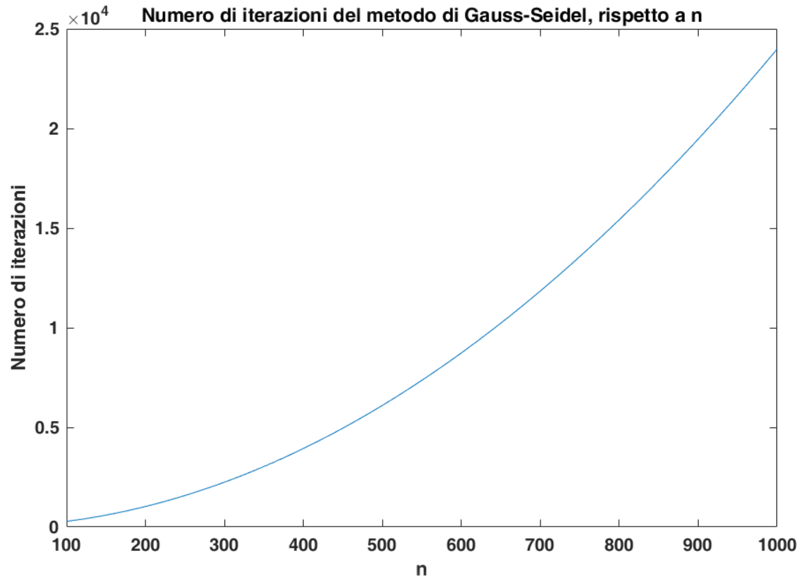
```
%     - b: vettore dei termini noti
%     - tol: tolleranza dell' approssimazione
%     - [x0]: vettore di partenza
%     - [maxit]: numero massimo di iterazioni
% Output:
%     - x: soluzione approssimata del sistema

    % - A non deve avere elementi nulli sulla diagonale
    D = diag(A);
    if ~all(D) % all(D) ritorna vero se tutti elementi !=0
        error('La diagonale di A non deve avere elementi nulli');
    end
    n = length(b);
    if nargin <= 3
        x = rand(n,1);
    else
        x = x0;
    end
    if nargin <= 4
        maxit = 100*n*round(-log(tol));
    end
    for i = 1:maxit
        r = A * x - b;
        nr = norm(r, inf);
        if nr <= tol
            break;
        end
        r = r./D;
        x = x - r;
    end
    if nr > tol
        warning('Raggiunto maxit.');
    end
    return
end
```



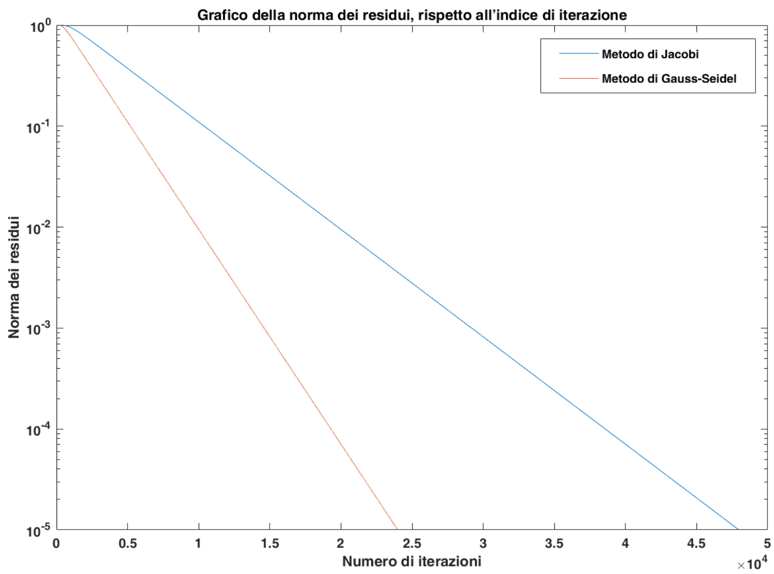
6.4 Esercizio 4

Prova

```
function [x,i,nr] = gauss_seidel(A, b, tol, x0, maxit)
% [x,i] = jacobi(A, b, tol, [xo, maxit])
% Restituisce la soluzione del sistema lineare Ax=b approssimata con il
% metodo di Gauss-Seidel e il numero di iterazioni eseguite.
% Input:
%     - A: matrice utilizzata per il calcolo
%     - b: vettore dei termini noti
%     - tol: tolleranza dell' approssimazione
%     - [x0]: vettore di partenza
%     - [maxit]: numero massimo di iterazioni
% Output:
%     - x: soluzione approssimata del sistema
    n = length(b);
    if nargin <= 3
        x = rand(n,1);
    else
        x = x0;
    end
    if nargin <= 4
        maxit = 100*n*round(-log(tol));
    end
    for i = 1:maxit
        r = A * x - b;
        err = norm(r,inf);
        nr(i) = err;
        if err<=tol
            break;
        end
        r = Msolve(A,r);
        x = x-r;
    end
    if err>tol
        warning('Non raggiunta tolleranza richiesta.');
    end
    return
end

function u = Msolve(M, r)
    % u = Msolve(M, r)
    % Restituisce la soluzione del sistema lineare Mx=r.
    u = r;
    n = length(u);
    for i = 1:n
       u(i) = u(i)/M(i,i);
       u(i+1:n) = u(i+1:n) - M(i+1:n,i)*u(i);
    end
    return
end
```



6.5 Esercizio 5



Prova