Elaborato di
Calcolo Numerico
Anno Accademico 2017/2018

August 29, 2018

August 29, 2018

Contents

1 Capitolo 4

1.1 Esercizio 4.1

Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di grado n in forma di Lagrange.
La forma della function deve essere del tipo: `y = lagrange( xi, fi, x)`

Il seguente codice Matlab implementa la function richiesta.

```
function y = lagrange(xi,fi,x)
    %y = lagrange(xi,fi,x) calcola il polinomio interpolante la coppia di
    %dati ascissa-ordinata.
    %INPUT:
    %xi = ascisse
    %fi = ordinate
    %x = vettore contenente i valori della x nel quale voglio calcolare il
    %polinomio
    %OUTPUT:
    %y = vettore contente i valori del polinomio interpolante le coppie
    %dei dati
    n = length(xi);
    L = ones(n,length(x));
    for i = 1:n
        for j = 1:n
            if(i~=j)
                L(i,:) = L(i,:).*(x-xi(j))/(xi(i)-xi(j));
            end
        end
    end
    y = 0;
    for i =1:n
        y = y +fi(i)*L(i,:);
    end
end
```

1.2 Esercizio 4.2

Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di grado n in forma di Newton.
La forma della function deve essere del tipo: `y = newton( xi, fi, x)`

Il seguente codice Matlab implementa la function richiesta.

```matlab
function y = newton(xi,fi,x)
    % y = newton(xi,fi,x)
    % Calcola il polinomio interpolante le coppie di dati (xi,fi)
    % sui punti del vettore x.
    % INPUT:
    % xi = vettore contenente le ascisse di interpolazione su cui calcolare la differenza divisa
    % fi = vettore contenente i valori assunti dalla funzione nei corrispondenti punti xi
    % x = vettore di punti sui quali valutare il polinomio interpolante
    % OUTPUT:
    % y = vettore contenente il valore del polinomio interpolante

    n = length(xi)-1; %grado del polinomio interpolante
    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
    y = fi(n+1)*ones(size(x));
    for i = n:-1:1
        y = (y.*(x-xi(i))+fi(i));
    end
end
```

1.3 Esercizio 4.3

Scrivere una function Matlab che implementi il calcolo del polinomio interpolante di Hermite.
La forma della function deve essere del tipo: `y = hermite( xi, fi, f1i, x)`

Il seguente codice Matlab implementa la function richiesta.

```
function y = hermite(xi,fi,f1i,x)
    %y = hermite(xi,fi,f1i,x) calcola il polinomio interpolante le coppie
    %di dati ascissa ordinata per mezzo del polinomio di Hermite
    %INPUT:
    %xi = ascisse
    %fi = ordinate
    %f1i = derivata prima
    %x = valori della x
        %OUTPUT:
    %y = vettore contenente il valore del polinomio interpolante calcolato
    % sulle x.

    n = length(xi)-1;
    xH = zeros(2*n+2,1); %ascisse interpolazione di Hermite, sono 2n+2
    xH(1:2:2*n+1) = xi;
    xH(2:2:2*n+2) = xi;
    fH = zeros(2*n+2,1); %vettore delle 2n+2 differenze divise
    fH(1:2:2*n+1) = fi;
    fH(2:2:2*n+2) = f1i;
    pHermGrade = length(xH)-1;
    for i = pHermGrade:-2:3
        fH(i) = (fH(i)-fH(i-2))/(xH(i)-xH(i-1));
    end
    for i = 2:pHermGrade
        for j = pHermGrade+1:-1:i+1
            fH(j) = (fH(j)-fH(j-1))/(xH(j)-xH(j-i));
        end
    end
    %Algoritmo di horner
    y = fH(pHermGrade+1)*ones(size(x));
    for i=pHermGrade:-1:1
        y = y.*(x-xH(i))+fH(i);
    end
end
```

1.4 Esercizio 4.4

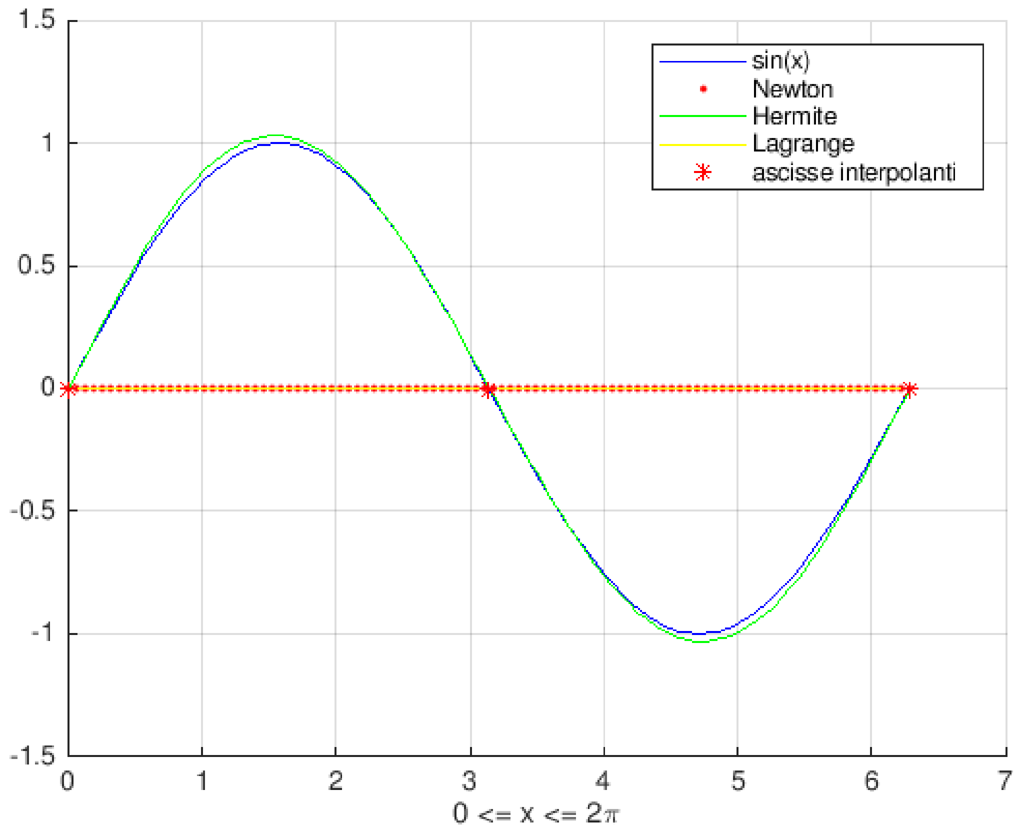
Utilizzare le functions degli esercizi precedenti per disegnare l'approssimazione della funzione $\sin(x)$ nell'intervallo $[0, 2\pi]$, utilizzando le ascisse di interpolazione $x_i = i\pi$, $i = 0, 1, 2$.

Il seguente codice Matlab contiene le chiamate alle funzioni degli esercizi precedenti:
y = *newton(xi, fi, x)*;
y = *lagrange(xi, fi, x)*;
y = *hermite(xi, fi, f1i, x)*;
calcolate fornendo in input le ascisse $0, \pi, 2\pi$ e la loro immagine attraverso f=$\sin(x)$. Nel caso di Hermite anche la loro immagine attraverso f'=$\cos(x)$.

```matlab
f = @(x) sin(x); % la funzione
xi = zeros(3,1); % vettore contenente le ascisse interpolanti la funzione
di = zeros(3,1); % vettore contenente i valori delle derivate
                 % della funzione relativa alle ascisse interpolanti
for j = 1:length(xi)
    xi(j) = (j-1)*pi;
    di(j) = cos(xi(j));
end
fi = f(xi);
x = linspace(0,2*pi);
pN = newton(xi,fi,x);
pL = lagrange(xi,fi,x);
pH = hermite(xi,fi,di,x);

figure;
hold on;
grid on;
fplot(f, [0, 2*pi], 'b');
plot(x, pN, 'r.', 'Markersize', 7);
plot(x, pH, 'g', 'Markersize', 7);
plot(x, pL, 'y', 'Markersize', 7);
plot(xi, fi, 'r*');
xlabel('0 <= x <= 2\pi');
legend('sin(x)', 'Newton', 'Hermite', 'Lagrange', 'ascisse interpolanti');
```

Nella figura sottostante é riportata l'approssimazione della funzione $\sin(x)$ tramite l'utilizzo delle funzioni di interpolazione:



Essendo $f_i = 0$ per tutte le ascisse x_i, sia il polinomio interpolante di Lagrange, che quello di Newton in realltá sono la retta $y = 0$.

1.5 Esercizio 4.5

Scrivere una function Matlab che implementi la *spline* cubica interpolante (naturale o *not-a-knot*, come specificato in ingresso) delle coppie di dati assegnate. La forma della function deve essere del tipo: `y = spline3( xi, fi, x, tipo)`

I seguenti codici Matlab, contengono la soluzione al problema dato:

```
function y = spline3(xi,fi,x,tipo)
    % y = spline3(xi,fi,x,tipo) calcola la spline cubica specificata dal
    % parametro tipo
    %INPUT:
    %xi = n+1 nodi di interpolazione
    %fi = valori della funzione calcolata in xi
    %x = vettore di valori nel quale voglio calcolare la spline
    %tipo = specifica il tipo di spline cubica tra naturale o not-a-knot
    %OUPTUT:
    %y = vettore contenente i risultati della valutazione nei punti di x
    phi = zeros(1,length(xi)-2);
    eps = zeros(1,length(xi)-2);
    for i=1:length(xi)-2
        hi = xi(i+1)-xi(i);
        h1 = xi(i+2)-xi(i+1);
        phi(i) = hi/(hi+h1);
        eps(i) = h1/(hi+h1);
    end
    diffDiv = diffDivise(xi,fi);
    if tipo == 0
        %cubica naturale
        mi = solveSplineNat(phi,eps,diffDiv);
        mi = [0;mi;0];
    else
        %cubica not a knot
        mi = solveSplineNaK(phi,eps,diffDiv);
    end
    s = createSpline(mi,fi,xi);
    y = evaluateSpline(s,xi,x);
end
```

```
function diffDiv = diffDivise(xi,fi)
    % fi = diffDivise(xi,fi) calcola le differenze divise
    % INPUT:
    % xi = nodi di interpolazione
    % fi = valori calcolati nei nodi di interpolazione
    % OUTPUT:
    % diffDiv = vettore contente le differenze divise
    n=length(xi)-1;

    for j=1:n-1
        for i=n+1:-1:j+1
            fi(i)=(fi(i)-fi(i-1))/(xi(i)-xi(i-j));
        end
    end

    diffDiv=fi(3:length(fi))';
end
```

```
function xi = solveSplineNat(b,c,xi)
    % solveSplineNat(b,c,xi) calcola la soluzione
    % del sistema lineare Ax = b dove A fattorizzabile LU
    % INPUT:
    % b = contiene le phi
    % c = contiene le eps
    % xi = contiene le differenze divise
    xi = 6*xi;
    a = ones(length(xi),1)*2;
    n = length(xi);
    for i = 1:n-1
        b(i) = b(i)/a(i);
        a(i+1) = a(i+1)- b(i)*c(i);
    end
    xi(2:n) = xi(2:n) - b(1:n-1) .* xi(1:n-1);
    xi(n) = xi(n)/a(n);
    for i = 1:n-1
        xi(n-i) = (xi(n-i)- c(n-i)*xi(n-i+1))/a(n-i);
    end
end
```

```
function xi = solveSplineNaK(phi,eps,xi)
    % mi = solveSplineNaK(phi,eps,xi) calcola la soluzione
    % del sistema lineare Ax = b dove A fattorizzabile LU
    % INPUT:
    % b = contiene le phi
    % c = contiene le eps
    % xi = contiene le differenze divise

    xi = [6*xi(1);6*xi;6*xi(length(xi))];
    n = length(eps)+1;
    a = zeros(n+1,1);
    b = zeros(n,1);
    c = zeros(n,1);
    a(1) = 1;
    c(1) = 0;
    b(1) = phi(1)/a(1);
    a(2) = 2-phi(1);
    c(2) = eps(1)-phi(1);
    b(2) = phi(2)/a(2);
    c(3) = eps(2);
    a(3) = 2-(b(2)*c(2));
    for i = 4:n-1
        b(i-1) = phi(i-1) / a(i-1);
        a(i) = 2 - b(i-1) * c(i-1);
        c(i) = eps(i-1);
    end

    b(n-1) = (phi(n-1)-eps(n-1))/a(n-1);
    a(n) = 2-eps(n-1)-b(n-1)*c(n-1);
    c(n) = eps(n-1);

    b(n) = 0;
    a(n+1) = 1;
    xi(2:n+1) = xi(2:n+1) - b(1:n) .* xi(1:n);
    xi(n+1) = xi(n+1)/a(n+1);
    for i = n:-1:1
        xi(i) = (xi(i)- c(i)*xi(i+1))/a(i);
    end
    xi(1) = xi(1)-xi(2)-xi(3);
    xi(n+1) = xi(n+1)-xi(n)-xi(n-1);
end
```

```
function s = createSpline(mi,fi,xi)
    % s = createSpline(mi,fi,xi) calcola i polinomi che costituiscono la
    % spline cubica
    % INPUT:
    % mi = derivata seconda della spline spline cubica calcolata in xi
    % xi = nodi di interpolazione
    % fi = valori della funzione calcolata in xi
    n = length(xi);
    s = sym('x',[n-1 1]);
    syms x;
    for i = 2:n
        hi = xi(i)-xi(i-1);
        ri = fi(i-1)-((hi^2)/6)*mi(i-1);
        qi = ((fi(i)-fi(i-1))/hi)-(hi/6) * (mi(i)-mi(i-1));
        s(i-1) = (((x-xi(i-1))^3)*mi(i)+((xi(i)-x)^3)*mi(i-1))/(6*hi);
        s(i-1) = s(i-1)+qi*(x-xi(i-1))+ri;
    end
end
```

```
function x = evaluateSpline(s,xi,x)
    % y = evaluateSpline(s,xi,x) valuta la spline s nei punti
    % contenuti in x
    % INPUT:
    % s = vettore contenente i polinomi che costituiscono la spline
    % xi = n+1 nodi di interpolazione
    % x = valori nel quale voglio valutare la spline
    n = length(xi) - 1;
    z = 1;
    k = 1;
    for i = 1:n
        isInt = 1;
        while k <= length(x) && isInt
            if x(k) >= xi(i) && x(k)<= xi(i+1)
                k = k+1;
            else
                isInt = 0;
            end
        end
        x(z:k-1) = subs(s(i),x(z:k-1));
        z = k;
    end
end
```

1.6 Esercizio 4.6

Scrivere una function Matlab che implementi il calcolo delle ascisse di Chebyshev per il polinomio interpolante di grado n, su un generico intervallo $[a, b]$.

La function deve essere del tipo: `xi = ceby( n, a, b )`

Il seguente codice Matlab implementa la function richiesta.

```
function xi = ceby(n,a,b)
    % xi = ceby(n,a,b)
    % Calcola le ascisse di Chebyshev per il polinomio
    % interpolante di grado n, trasformate in [a,b]
    % INPUT:
    % n =  numero di intervalli desiderati tra a e b che definiscono la
    % partizione delle ascisse interpolanti
    % a = estremo sinistro
    % b = estremo destro
    % OUTPUT:
    % xi = vettore ascisse di Chebyshev

    xi = cos( ((2*[0:n]+1)*pi) / (2*n+2) );
    xi = ((a+b)+(b-a)*xi)/2;
end
```

1.7 Esercizio 4.7

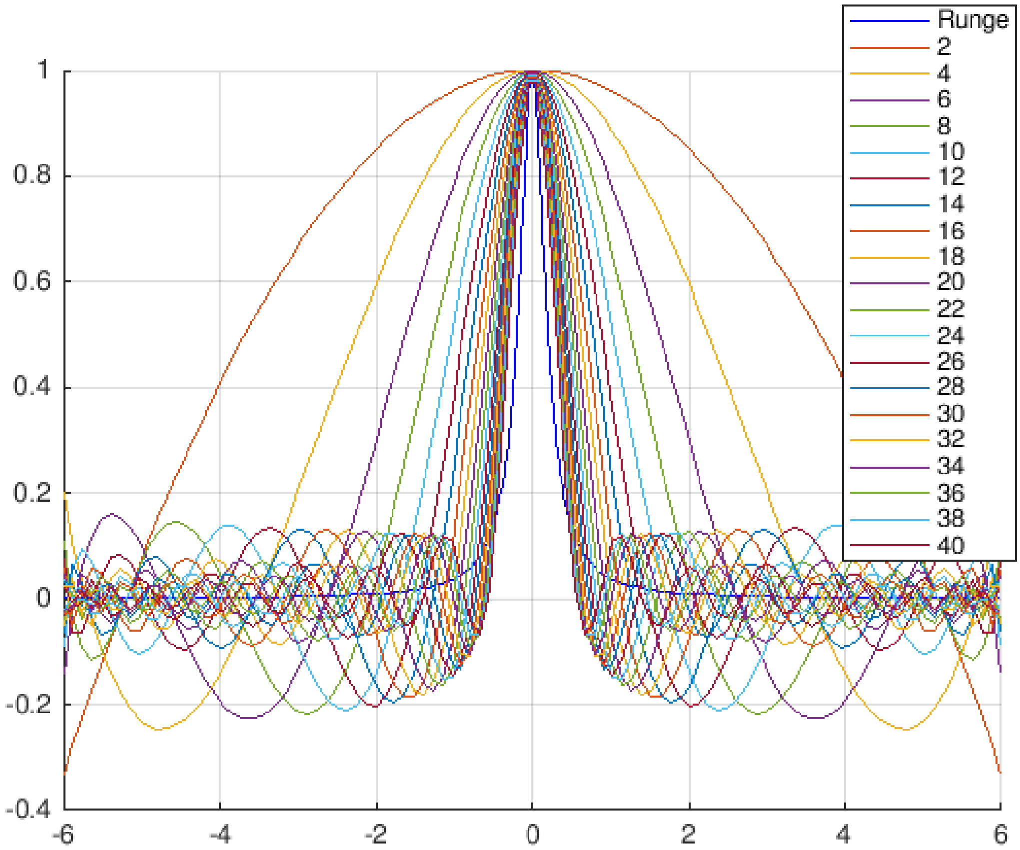
Utilizzare le function degli Esercizi 4.1 e 4.6 per graficare l'approssimazione della funzione di Runge sull'intervallo $[-6, 6]$, per $n = 2, 4, 6, \ldots, 40$. Stimare numericamente l'errore commesso in funzione del grado n del polinomio interpolante.

Il seguente codice Matlab implementa la soluzione al problema dato:

```
%Soluzione esercizio 7 capitolo 4
f = @(x) 1./(1+25 * x.^2); %funzione di Runge
a = -6;
b = 6;
n = 2;
xi = [];
err = zeros(20,1);
i = 1;
hold on;
grid on;
x = linspace(a,b);
fplot(f, [a,b], 'b','Markersize', 7);
while n <= 40
    xi = ceby(n,a,b);
    fi = f(xi);
    y = lagrange(xi,fi,x);
```

```
    err(i) = norm(f(x)-y, inf);
    plot(x,y);
    n = n+2;
    i = i+1;
end
legend('Runge','2','4','6','8','10','12','14','16','18','20','22','24','26
    ','28','30','32','34','36','38','40');
hold off
```

Il grafico seguente mostra i polinomi interpolanti di grado n, calcolati utilizzando come punti di interpolazione quelli corrispondenti alle n ascisse di Chebyshev. Si ricorda la funzione di Runge scritta come: $f(x) = \frac{1}{1+25x^2}$.



Abbiamo quindi calcolato l'errore al variare di n (con f = funzione di Runge e $p_n(x)$ = il suo polinomio interpolante di grado n) come segue:

$$||err|| \approx ||f(x) - p_n(x)||_{\text{inf}}$$

Nella seguente tabella si riportano gli errori calcolati:

n	$\|err\|$
2	0.9244
4	0.8717
6	0.8217
8	0.7757
10	0.7262
12	0.6866
14	0.6464
16	0.6025
18	0.5568
20	0.5291
22	0.5000
24	0.4696
26	0.4384
28	0.4067
30	0.3747
32	0.3427
34	0.3110
36	0.2855
38	0.2713
40	0.2570

Grazie alla scelta delle ascisse di Chebyshev come punti di interpolazione, l'errore diminuisce all'aumentare di n.

1.8 Esercizio 4.8

Relativamente al precedente esercizio, stimare numericamente la crescita della costante di Lebesgue.

I seguenti codici Matlab contengono il calcolo della costante di Lebesgue in funzione di n. La costante di Lebesgue è definita come segue:

$$\Lambda_n = ||\lambda_n|| \quad con\ \lambda_n(x) = \sum_{i=0}^{n} |L_{(i,n)}(x)|$$

```
%Soluzione esercizio 8 capitolo 4
a = -6;
b = 6;
n = 2;
constLeb = zeros(20,1);
i = 1;
while n <= 40
    xi = ceby(n,a,b);
    constLeb(i) = computeLeb(xi);
    n = n+2;
    i = i+1;
end
```

```
function y = computeLeb(xi)
    %y = computeLeb(xi) calcola la costante di Lebesgue relativa alle
    %ascisse in xi
    %INPUT:
    %xi = vettore delle ascisse
    %OUTPUT
    %y = costante di Lebesgue
    n = length(xi);
    x = linspace(xi(1),xi(end),10000);
    L = ones(n,length(x));
    sumL = 0;
    for i = 1:n
        for j = 1:n
            if(i~=j)
                L(i,:) = L(i,:).*(x-xi(j))/(xi(i)-xi(j));
            end
        end
        sumL = sumL + abs(L(i,:));
        y = norm(sumL,inf);
    end
end
```

Nella seguente tabella viene mostrata come varia la costante di Lebesgue in funzione di n. Come si può notare, grazie alle ascisse di Chebyshev, si ha una crescita logaritmica della costante al variare del grado n del polinomio:

n	$lebesgue$
2	1.2500
4	1.5702
6	1.7825
8	1.9416
10	2.0687
12	2.1747
14	2.2655
16	2.3450
18	2.4156
20	2.4792
22	2.5370
24	2.5900
26	2.6386
28	2.6843
30	2.7266
32	2.7662
34	2.8036
36	2.8391
38	2.8718
40	2.9022

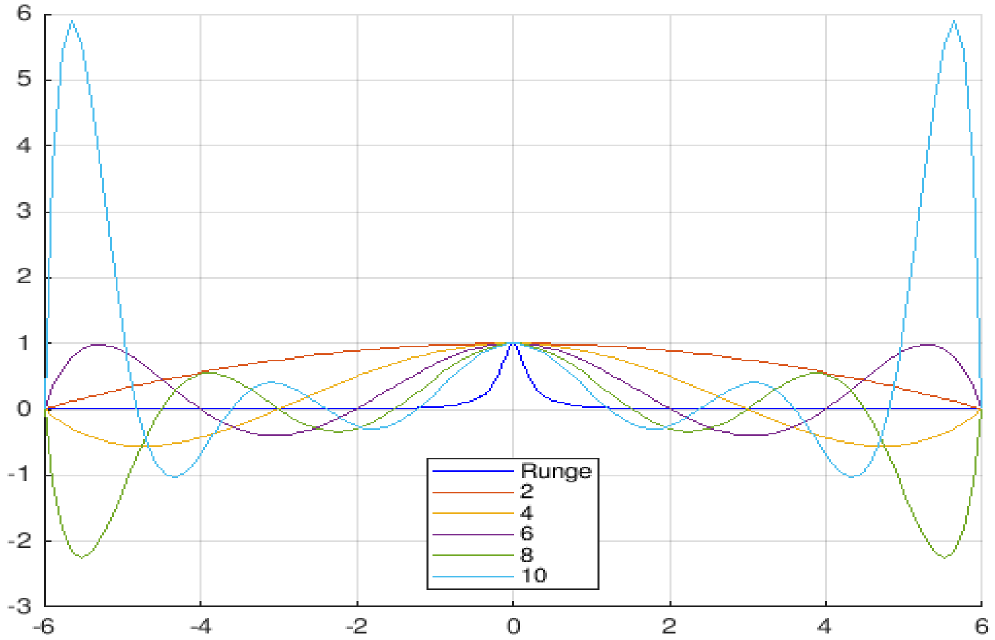
1.9 Esercizio 4.9

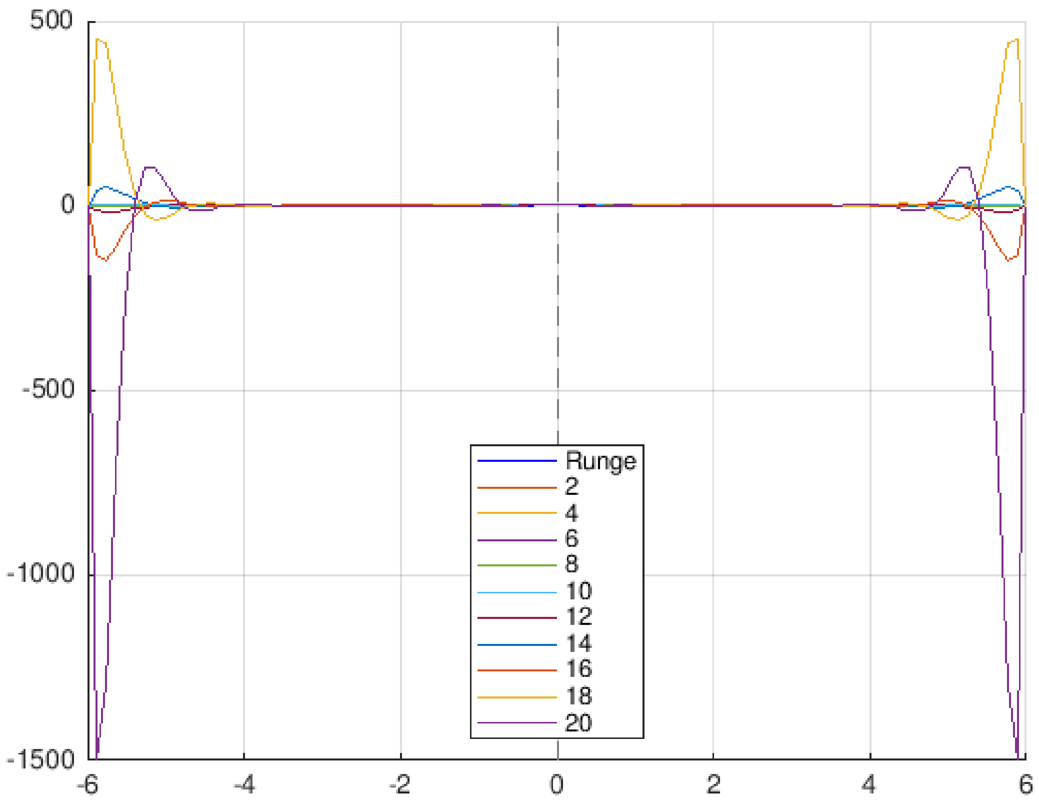
Utilizzare la function dell'Esercizio 4.1 per approssimare la funzione di Runge sull'intervallo $[-6, 6]$, su una partizione uniforme di $n + 1$ ascisse per $n = 2, 4, 6, \ldots, 40$. Stimare le corrispondenti costanti di Lebesgue.

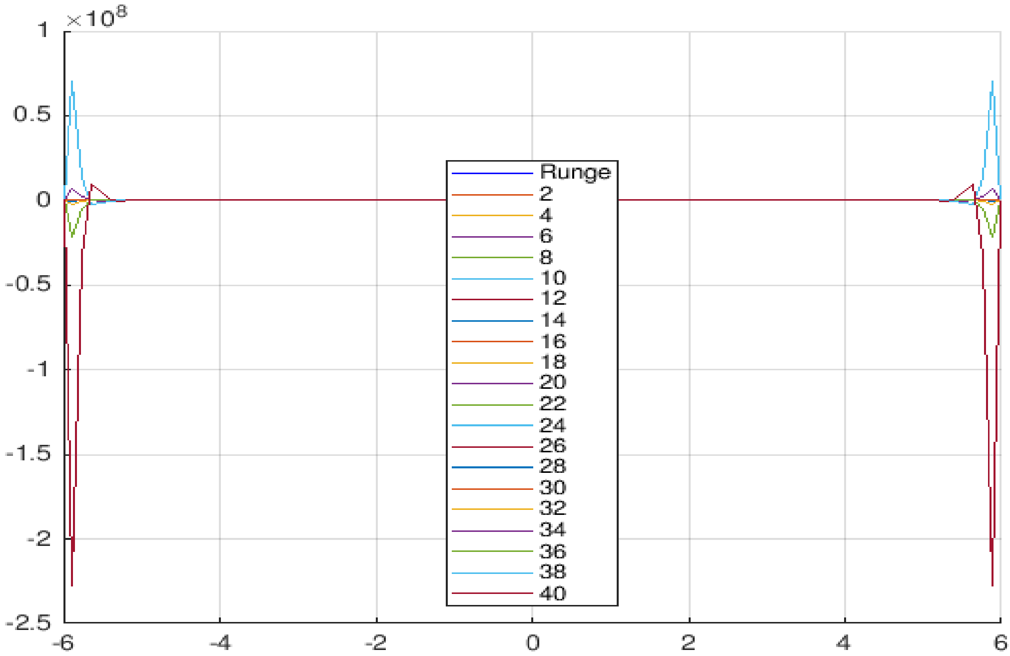
Il seguente codice Matlab contiene la soluzione al problema dato:

```matlab
%Soluzione esercizio 9 capitolo 4
f = @(x) 1./(1+25 * x.^2); %funzione di Runge
a = -6;
b = 6;
n = 2;
xi = [];
err = zeros(20,1);
constLeb = zeros(20,1);
i = 1;
hold on;
grid on;
x = linspace(a,b);
fplot(f, [a,b], 'b','Markersize', 7);
while n <= 40
    xi = linspace(a,b,n+1);
    fi = f(xi);
    y = lagrange(xi,fi,x);
    err(i) = norm(f(x)-y, inf);
    constLeb(i) = computeLeb(xi);
    plot(x,y);
    n = n+2;
    i = i+1;
end
legend({'Runge','2','4','6','8','10','12','14','16','18','20','22','24','
    26','28','30','32','34','36','38','40'},'Location','south');
hold off
```

Di seguito i grafici mostrano i polinomi interpolanti di Lagrange al variare del grado N con $N = 2, 4, 6, \ldots, 40$.







Nella tabella è riportato come varia la *costante di Lebesgue*, al variare del grado n del polinomio interpolante. Come si può vedere, all'aumentare di n l'errore aumenta a causa della scelta delle ascisse equispaziate.

n	*lebesgue*
2	0.9342
4	0.8566
6	0.9846
8	2.2590
10	5.8960
12	16.3788
14	49.2750
16	147.6550
18	450.3933
20	$1.5025e+03$
22	$5.0066e+03$
24	$1.6654e+04$
26	$5.5282e+04$
28	$1.8307e+05$
30	$6.0468e+05$
32	$1.9918e+06$
34	$6.5422e+06$
36	$2.1426e+07$
38	$6.9960e+07$
40	$2.2774e+08$

1.10 Esercizio 4.10

Stimare, nel senso dei minimi quadrati, posizione, velocitá iniziale ed accelerazione relative ad un moto rettilineo uniformemente accelerato per cui sono note le seguenti misurazioni dele coppie $(tempo, spazio)$: $(1, 2.9)$ $(1, 3.1)$ $(2, 6.9)$ $(2, 7.1)$ $(3, 12.9)$ $(3, 13.1)$ $(4, 20.9)$ $(4, 21.1)$ $(5, 30.9)$ $(5, 31.1)$

La legge che descrive il fenomeno del moto rettilineo uniformemente accelerato si puó scrivere in forma polinomiale come segue:

$$y = s(t) = x_0 + v_0 t + a_0 t^2 \qquad \text{con } a_0 = \frac{1}{2}a$$

Il cui grado è n = 2. Il sistema ha soluzione se si ha almeno n+1 punti distinti. In questo caso il problema é ben posto poiché i punti distinti sono 5>3.
Si vuole quindi stimare nel senso dei minimi quadrati: posizione, velocità iniziale, ed accelerazione, che equivale alla risoluzione del sistema lineare sovradeterminato:

$$V\underline{a} = \underline{y}$$

con V matrice di tipo *Vandermonde* (la trasposta di una matrice di tipo Vandermonde), $\underline{a}$ vettore delle incognite e $\underline{y}$ il vettore dei valori misurati.
Tale sistema si risolve mediante fattorizzazione QR. La matrice V è scritta come segue:

$$V = \begin{bmatrix} x_0^0 & x_0^1 & \cdots & x_0^m \\ x_1^0 & x_1^1 & \cdots & x_1^m \\ \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^m \end{bmatrix}$$

```
%Soluzione esercizio 10 cap 4

%FORMULA MOTO RETTILINEO UNIFORMEMENTE ACCELERATO
%x(t) = x0+v0*t+1/2*a*t^2
%Il sistema ha soluzione se ho almeno n+1 punti distinti
%In questo caso il problema è ben posto poiche i punti distinti sono 5>3
value = 1;
V = zeros(10,3);
for i = 1:10
    V(i, 1:3) = value.^(0:2);
    r = mod(i,2);
    if r == 0
        value = value + 1;
    end
end
y = [2.9 3.1 6.9 7.1 12.9 13.1 20.9 21.1 30.9 31.1];
V = QRFatt(V);
[m,n] = size(V);
x = SolveLeastSquares(V,y,m,n);
```

Le soluzioni, calcolate, al problema dato sono :

$$x_0 = 1,\ v_0 = 1,\ a_0 = 1$$

2 Capitolo 6

2.1 Esercizio 6.1

Scrivere una function Matlab che generi la matrice *sparsa* $n \times n$, con $n > 10$

$$A_n = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, \text{ con } a_{ij} = \begin{cases} 4 \text{ se } i = j \\ -1 \text{ se } i = j \pm 1 \\ -1 \text{ se } i = j \pm 10 \end{cases}$$

Utilizzare,a questo fine, la function Matlab spdiags.

Il seguente codice Matlab risolve il problema dato utilizzando la funzione `spdiags`:

```
function A = createSparseMatrix(n)
    %A = createSparseMatrix(n) crea una matrice sparsa
    %INPUT
    %n = grado della matrice
    %OUTPUT
    %A = matrice sparsa n*n in cui:
    %a(i,j) = 4 se i=j
    %a(i,j) = -1 se i=j+-1
    %a(i,j) = -1 se i=j+-10
    if n <= 10
        error('n deve essere maggiore di 10');
    end
    e = ones(n,1);
    A = [-1*e 4*e -1*e -1*e -1*e];
    A = spdiags(A,[-1:1,-10, 10],n,n);
end
```

2.2 Esercizio 6.2

Utilizzare il metodo delle potenze per calcolarne l'autovalore dominante della matrice A_n del precedente esercizio, con una approssimazione $tol = 10^{-5}$, partendo da un vettore con elementi costanti. Riempire, quindi, la seguente tabella:

n	*numero iterazioni effettuate*	*stima autovalore*
100		
200		
$\vdots$		
1000		

Di seguito sono riportati i codici implementati. La funzione *potenze* calcola sia l'autovalore dominante della matrice A, sia il numero di iterazioni, *numIt*, impiegate.

```
%Soluzione es 2 capitolo 6
tol = 10^-5;
numIt = zeros(10,1);
autVal = zeros(10,1);
index = 1;
for i = 100:100:1000
    A = createSparseMatrix(i);
    v = ones(i,1);
    [autVal(index),numIt(index)] = potenze(A,tol,v);
    index = index+1;
end
```

```
function [lambda,numIt] = potenze(A,tol,x0,maxit)
    %[lambda,numIt] = potenze(A,tol,[x0,maxit]) calcola l'autovalore
    %dominante della matrice A.
    %INPUT:
    %A = matrice
    %tol = tolleranza richiesta
    %x0 = vettore iniziale
    %maxit = numero massimo di iterazioni
    %OUTPUT:
    %lambda = autovalore domaninate
    %numIt = numero di iterazioni effettuate
    n = size(A,1);
    if nargin <= 2
        x = rand(n,1);
    else
        x = x0;
    end
    x = x/norm(x);
    if nargin <= 3
        maxit = 100*n*max(round(-log(tol),1));
    end
```

```
    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('Tolleranza richiesta non raggiunta');
    else
        numIt = i;
    end
end
```

Nella seguente tabella é possibile visualizzare i risultati ottenuti:

n	*numero iterazioni effettuate*	*stima autovalore*
100	167	7.8224
200	420	7.8803
300	638	7.8916
400	721	7.8949
500	743	7.8964
600	824	7.8974
700	893	7.8976
800	868	7.8967
900	795	7.8957
1000	775	7.8954

2.3 Esercizio 6.3

Utilizzare il metodo di Jacobi per risolvere il sistema lineare

$$A_n \mathbf{x} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

dove A_n è la matrice definita nell'Esercizio 6.1, con tolleranza $tol = 10^{-5}$, e partendo dal vettore nullo. Graficare il numero di iterazioni necessarie, rispetto alla dimensione n del problema, con n che varia da 100 a 1000 (con passo 20).

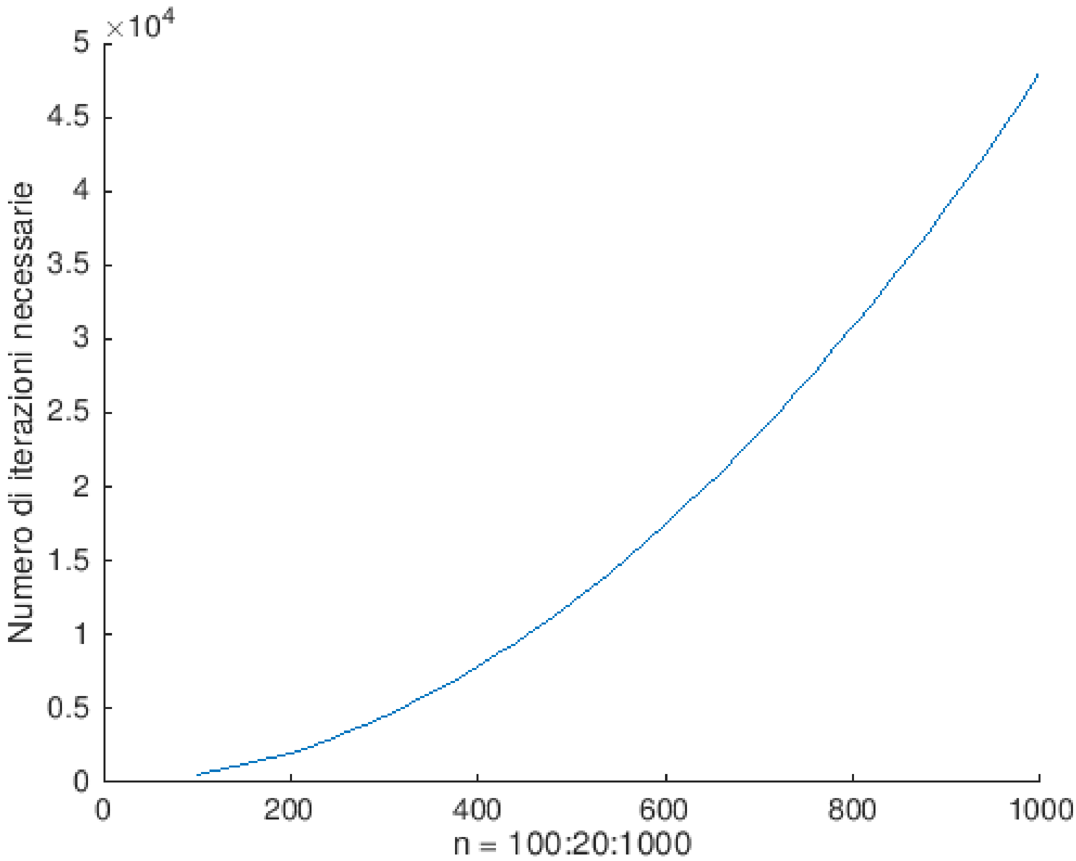
Di seguito si riportano i codici Matlab utilizzati. La funzione di *Jacobi* oltre ad effettuare il calcolo del vettore delle incognite della matrice A, restituisce anche il numero di iterazioni *numIt* impiegate e la norma infinito del residuo, al passo i-simo.

```
%Soluzione es 3 cap6
tol = 10^-5;
numIt = zeros(46,1);
index = 1;
for i = 100:20:1000
    A = createSparseMatrix(i);
    x0 = zeros(i,1);
    b = ones(i,1);
    [x,numIt(index)] = jacobi(A,b,tol,x0);
    index = index+1;
end
hold on;
plot(100:20:1000,numIt);
```

```
function [x,numIt,normRes] = jacobi(A,b,tol,x0)
    %[x,numIt,normRes] = jacobi(A,b,tol,[x0]) calcola la soluzione del sistema
    %lineare sparso Ax = b utilizzando il metodo iterativo di Jacobi con
    %tolleranza tol
    %INPUT:
    %A = matrice sparsa
    %b = vettore termini noti
    %tol = tolleranza richiesta
    %x0 = vettore iniziale
    %OUTPUT:
    %x = soluzione del sistema lineare sparso
    %numIt = numero di iterazioni effettuate
    n = length(b);
    if nargin <= 3
        x = zeros(n,1);
    else
        x = x0;
    end
```

```
    D = diag(A);
    maxit = 100*n*max(1,-log(tol));
    normRes = zeros(round(maxit),2);
    for i = 1:maxit
        r = b-(A*x);
        err = norm(r,inf);
        normRes(i,1) = i;
        normRes(i,2) = err;
        if err <= tol
            break;
        end
        x = x + r./D;
    end
    if err > tol
        warning('Tolleranza richiesta non raggiunta');
    else
        numIt = i;
    end
end
```

Graficamente si puó osservare il seguente risultato, al variare di n:



2.4 Esercizio 6.4

Ripetere una procedura analoiga a quella del precedente esercizio utilizzando il metodo di Gauss-Seidel.

Il codice Matlab utilizzato per realizzare il grafico é il seguente:

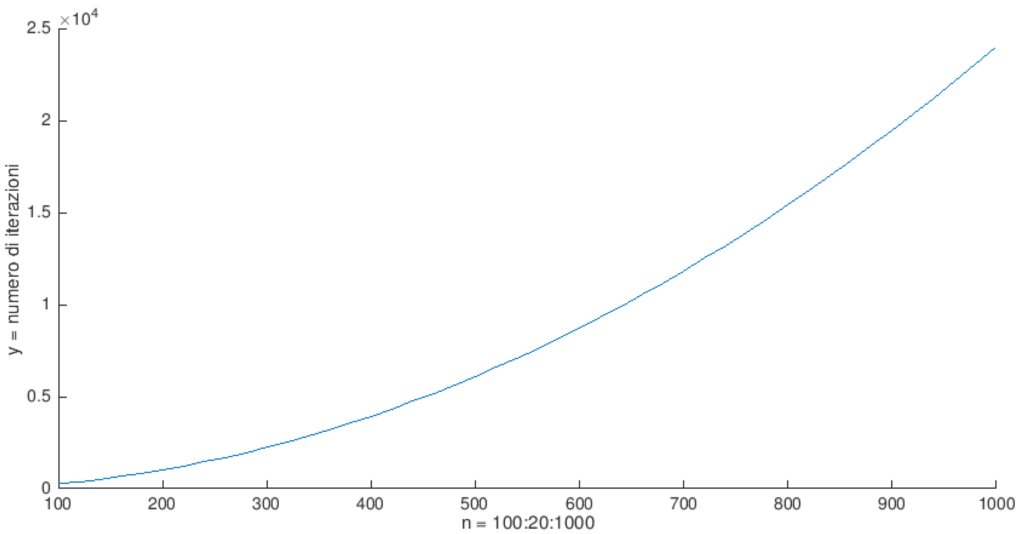
```
%Soluzione es 4 cap6
tol = 10^-5;
numIt = zeros(46,1);
index = 1;
for i = 100:20:1000
    A = createSparseMatrix(i);
    x0 = zeros(i,1);
    b = ones(i,1);
    [x,numIt(index)] = gaussSeidel(A,b,tol,x0);
    index = index+1;
end
hold on;
plot(100:20:1000,numIt);
```

```
function [x,numIt,normRes] = gaussSeidel(A,b,tol,x0)
        %[x,numIt,normRes] = gaussSeidel(A,b,tol,x0) risolve il sistema
            lineare Ax=b
        %con il metodo di Gauss-Seidel
    %INPUT:
    %A = matrice sparsa
    %b = vettore dei termini noti
    %tol = tolleranza richiesta
    %x0 = vettore iniziale
    %OUTPUT:
    %x = soluzione del sistema lineare sparso
    %numIt = numero di iterazioni
    n = length(b);
    if nargin <= 3
        x = zeros(n,1);
    else
        x = x0;
    end
    maxit = 100*max(1,-ceil(log(tol)))*n;
    err = inf;
    for i=1:maxit
        r = (A*x)-b;
        err = norm(r,inf);
        normRes(i,1) = i;
        normRes(i,2) = err;
        if err <= tol
            break;
        end
```

```
        r = trisolveInfGaussSeidel(A,r);
        x = x-r;
    end
    if err > tol
        warning('Tolleranza richiesta non raggiunta');
    else
        numIt = i;
    end
end
```

```
function x = trisolveInfGaussSeidel(A,b)
    %x = trisolveInfGaussSeidel(A,b)
    %La funzione restituisce la soluzione del sistema lineare Ax = b
    %INPUT:
    %A = matrice triangolare inferiore
    %b = vettore termini noti
    %OUTPUT:
    %x = vettore soluzione
    x = b;
    [m,n] = size(A);
    if m~=n
        error('Matrice non quadrata');
    end
    for i = 1:n
        x(i) = x(i)/A(i,i);
        x(i+1:n) = x(i+1:n)-x(i)*A(i+1:n,i);
    end
end
```

Grafico risultante:



2.5 Esercizio 6.5

Con riferimento al sistema lineare dell' Esercizio 6.3, con $n = 1000$, graficare la norma dei residui, rispetto all'indice di iterazione, generati dai metodi di Jacobi e Gauss-Seidel. Utilizzare il formato *semilogy* per realizzare il grafico, corredandolo di opportune *label*.

Il seguente codice Matlab é stato utilizzato per la risoluzione del problema:

```
%Soluzione esercizio 5 cap 6
tol = 10^-5;
A = createSparseMatrix(1000);
x0 = zeros(1000,1);
b = ones(1000,1);
[x,numIt,normRes] = jacobi(A,b,tol,x0);
[x2,numIt2,normRes2] = gaussSeidel(A,b,tol,x0);
hold on;
semilogy(normRes(:,1),normRes(:,2));
semilogy(normRes2(:,1),normRes2(:,2));
legend('Jacobi','Gauss-Seidel');
```

Il grafico seguente mostra la norma dei residui, rispetto all'indice di iterazione generati dai metodi di Jacobi (in azzurro) e Gauss-Seidel (in rosso):