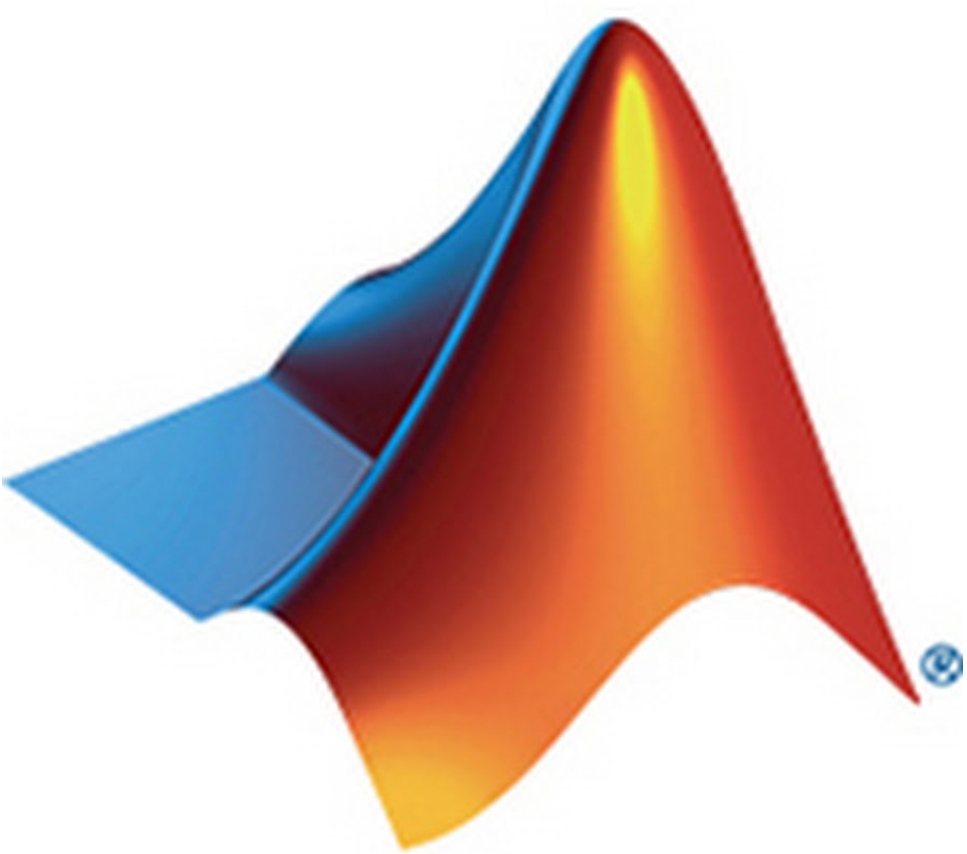
Elaborato Calcolo Numerico 2018-19

Bonechi Iacopo: *iacopo.bonechi@stud.unifi.it*
Caia Antonio: *antonio.caia@stud.unifi.it*
Taddei Dario: *dario.taddei@stud.unifi.it*

June 17, 2019



1 Esercizio

Verificare che per h sufficientemente piccolo vale:

$\frac{3}{2}f(x) - 2f(x-h) + \frac{1}{2}f(x-2h) = hf'(x) + O(h^3)$

$\frac{3}{2}f(x) - 2[f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 + O(h^3)] + \frac{1}{2}[f(x) - f'(x)2h + \frac{1}{2}f''(x)4h^2 + O(h^3)] = hf'(x) + O(h^3)$

$\frac{3}{2}f(x) - 2f(x) + 2f'(x)h - f''(x)h^2 + \frac{1}{2}f(x) - f'(x)h + f''(x)h^2 + O(h^3) = hf'(x) + O(h^3)$

$f'(x)h + O(h^3) = f'(x)h + O(h^3)$

2 Esercizio

Quanti sono i numeri di macchina normalizzati dalla doppia precisione IEEE? Argomentare la risposta:

Nella doppia precisione normalizzata si hanno *52 bit* per la mantissa, *1 bit* per il segno e *11 bit* per l'esponente. Non tutti i valori dell'esponente però possono essere usati:

- 0 è usato per la rappresentazione dei numeri denormalizzati
- 2047 è usato per esprimere valori speciali come *NaN, +Inf, -Inf*

I numeri di macchina normalizzati sono quindi:
$2 * 2^{52} * (2^{11} - 2) = 2^{64} - 2^{54} = 1.842872967520007 * 10^{19}$

3 Esercizio

Eseguire il seguente script di Matlab:

```
format long e
n=75;
u=1e-300;
%--Prima Parte--
for i=1:n
    u=u*2;
end
for i=1:n
    u=u/2;
end
%--Seconda Parte--
u=1e-300;
for i=1:n
    u=u/2;
end
for i=1:n
    u=u*2;
end
```

I risultati sono i seguenti:

$u_1 = 1.0 * 10^{-300}$
$u_2 = 1.119916342203863 * 10^{-300}$

Nella prima parte, effettuando le moltiplicazioni e poi le divisioni, non si incorre in nessun tipo di errore e u_1 risulta essere identica alla u iniziale.
Nella seconda parte invece, dove l'ordine viene invertito, u_2 presenta una differenza con quella iniziale. L'errore registrato è dovuto all'*underflow* e alla relativa recovery *gradual underflow*.
I numeri di macchina normalizzati infatti vengono rappresentati in binario nella forma $1.m * 2^{e-v}$, dove m è la mantissa memorizzata, v è il valore fissato di *shift* pari a 1023 ed e è l'esponente. Il più piccolo numero di macchina rappresentabile è b^{1-v}; se si volesse rappresentare valori più piccoli, senza incorrere in *underflow,* andrebbero *denormalizzati* i numeri, ossia sottrarre bit alla mantissa e aggiungerli all'esponente, consentendo quindi un più ampo range di numeri rappresentabili in cambio però di una minore precisione sui valori.
Questa riduzione di precisione comporta quindi un errore che si ripercuote su u_2 quando vengono effettuate le moltiplicazioni.

4 *Esercizio*

Eseguire le seguenti istruzioni Matlab:

```
format long e
a=1.111111111111111;
b=1.111111111111111;
a+b;
a-b;
```

I risultati sono i seguenti:
$a + b = 2.222222222222221$
$a - b = 8.881784197001252 * 10^{-16}$
Se si effettuano i calcoli in aritmetica esatta otteniamo:
$a + b = 2.222222222222221$
$a - b = 0.000000000000001 * 10^{-15}$
Confrontandoli con i risultati ottenuti in Matlab si osserva che mentre il risultato della somma è corretto, la sottrazione introduce un errore. La sottrazione in aritmetica finita è un operazione malcondizionata, di un fattore: $k = \frac{|a|+|b|}{|a+b|}$
Nel caso di numeri molto vicini tra di loro ($x_1 \approx -x_2$), il denominatore diventa estremamente piccolo facendo crescere rapidamente k, questo malcondizionamento da luogo al cosidetto fenomeno della cancellazione numerica.
Si ha quindi:
$k = 2.22222222222221 * 10^{15}$
$u = \frac{1}{2}2^{1-53} = 1.110223024625157 * 10^{-16} = \epsilon_x$
$|\epsilon_y| = k * |\epsilon_x| = 2.46716227694479 * 10$
Con un errore in uscita compatibile con i dati sperimentali.

5 Esercizio

Scrivere function Matlab distinte che implementino efficientemente i seguenti metodi per la ricerca degli zeri di una funzione:

- *Metodo di bisezione*
- *Metodo di Newton*
- *Metodo delle secanti*
- *Metodo delle corde*

Detta x_i l'approssimazione al passo i-esimo, utilizzare come criterio di arresto $|\Delta x_i| \leq tol \cdot (1 + |x_i|)$ essendo tol una opportuna tolleranza specificata in ingresso.

```
function [x,i]=bisezione(f ,a, b, tol)
%x=bisezione(f ,a, b, tol)
%f: funzione da analizzare
%a,b: estremi dell'intervallo in cui cercare la radice
%tol: parametro per determinare l'accuratezza sul valore di x
%Il metodo di bisenzione ricerca una radice in un intervallo [a,b]
%di una funzione f
if a>b
    error("intervallo invalido");
end
fa=feval(f,a);
fb=feval(f,b);
if fa*fb>0
    error("Non sono presenti zeri nell'intervallo");
end
x=(a+b)/2;
fx=feval(f,x);
imax=ceil(log2(b-a)-log2(tol));
for i=2:imax
    f1x=abs((fb-fa)/(b-a));
    if (abs(fx)<=tol*(1+abs(x))*abs(f1x))
        break;
    elseif((fa*fx)<0)
        b=x;
        fb=fx;
    else
        a=x;
        fa=fx;
    end
    x=(a+b)/2;
    fx=feval(f,x);
end
return
end
```

```
function [x,i]=corde(f,f1, x0, tol, itmax)
%x=corde(f,f1, x0, tol, itmax)
%f: funzione da analizzare
%f1: derivata di f
%x0: punto di innesco
%imax: numero massimo di iterazioni per trovare la radice
%tol: parametro per determinare l'accuratezza sul valore di x
%Il metodo delle corde ricerca gli zeri in una funzione
fx=feval(f,x0);
f1x=feval(f1,x0);
x=double(x0-fx/f1x);
i=0;
while i<itmax && abs(x-x0)>=tol*(1+abs(x))
    x0=x;
    fx=feval(f,x0);
    x=x0-fx/f1x;
    i=i+1;
end
if (i==itmax)
    disp("CORDE: valore della radice non sufficientemente preciso");
end
return
end
```

```
function [x,i]=newton(f, f1, x0, tol, itmax)
%x=newton(f,f1, x0, itmax, tol)
%f: funzione da analizzare
%f1: derivata della funzione f
%x0: punto di innesco
%imax: numero massimo di iterazioni per trovare la radice
%tol: parametro per determinare l'accuratezza sul valore di x
%Il metodo di Newton ricerca gli zeri in una funzione
fx=feval(f,x0);
f1x=feval(f1,x0);
x=double(x0-fx/f1x);
i=0;
while i<itmax && abs(fx)>=tol*(1+abs(x))*f1x && abs(x-x0)>=tol*(1+abs(x))
    x0=x;
    fx=feval(f,x0);
    f1x=feval(f1,x0);
    x=x0-fx/f1x;
    i=i+1;
end
if (i==itmax)
    disp("NEWTON: valore della radice non sufficientemente preciso");
end
return
end
```

```
function [x,i]=secanti(f,f1, x0, tol, itmax)
%x=secanti(f, x0, tol, itmax)
%f: funzione da analizzare
%f1: derivata funzione f
%x0: punto di innesco
%imax: numero massimo di iterazioni per trovare la radice
%tol: parametro per determinare l'accuratezza sul valore di x
%Il metodo delle secanti ricerca gli zeri in una funzione
fx=feval(f,x0);
f1x=feval(f1,x0);
x=x0-fx/f1x;
i=0;
while (i<itmax) && (abs(x-x0)>=tol*(1+abs(x)))
    i=i+1;
    fx0=fx;
    fx=feval(f,x);
    x1=(fx*x0-fx0*x)/(fx-fx0);
    x0=x;
    x=x1;
end
if abs(x-x0)>=tol*(1+abs(x))
    fprintf("SECANTI: valore della radice non sufficientemente preciso\n");
end
return
end
```

6 Esercizio

Utilizzare le function del precedente esercizio per determinare una approssimazione della radice della funzione

$$f(x) = x - e^{-x}cos(\frac{x}{100})$$

per $tol = 10{-}i, i = 1, 2, ..., 12$*, partendo da* $x_0 = -1$*. Per il metodo di bisezione, utilizzare* $[-1, 1]$*, come intervallo di confidenza iniziale. Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare il relativo costo computazionale.*

Abbiamo preso in esame il numero di valutazioni di funzioni e la velocità di convergenza:
La radice della funzione che vogliamo individuare è $x_0 = 5.671374702931910816354075 71 * 10^{-1}$, con molteplicità pari a 1. Per ogni iterazione possiamo assumere i seguenti costi:

- *Bisezione*: 1 valutazione di funzione; velocità lineare.
- *Newton*: 2 valutazioni di funzione; velocità quadratica per funzioni sufficientemente regolari con radici semplici.
- *Corde*: 1 valutazione di funzione; velocità lineare.
- *Secanti*: 1 valutazione di funzione; velocità nell'ordine di 1.618.

Il metodo delle bisezioni e delle corde, sebbene abbiano un costo computazionale tra i più bassi, richiedono un numero di iterazioni molto maggiore rispetto a Newton e Secanti. Tra questi due metodi Newton ha un costo computazionale maggiore a causa delle due valutazioni di funzioni che esegue ad ogni iterazione. Il metodo delle Secanti risulta essere quindi il giusto compromesso tra velocità (superiore ai metodi lineari) e numero di iterazioni.

Tol	**i**	**Bisezione**	**i**	**Newton**	**i**	**Corde**	**i**	**Secanti**
10^{-1}	3	5.000000e-01	2	5.663058e-01	3	5.662928e-01	2	4.021809e-01
10^{-2}	6	5.625000e-01	3	5.671373e-01	4	5.671340e-01	6	5.495186e-01
10^{-3}	10	5.664062e-01	3	5.671373e-01	4	5.671340e-01	10	5.651742e-01
10^{-4}	14	5.671387e-01	4	5.671375e-01	5	5.671375e-01	15	5.670104e-01
10^{-5}	14	5.671387e-01	4	5.671375e-01	5	5.671375e-01	19	5.671232e-01
10^{-6}	14	5.671387e-01	4	5.671375e-01	6	5.671375e-01	23	5.671359e-01
10^{-7}	24	5.671375e-01	4	5.671375e-01	6	5.671375e-01	27	5.671373e-01
10^{-8}	24	5.671375e-01	5	5.671375e-01	6	5.671375e-01	31	5.671375e-01
10^{-9}	31	5.671375e-01	5	5.671375e-01	6	5.671375e-01	36	5.671375e-01
10^{-10}	34	5.671375e-01	5	5.671375e-01	7	5.671375e-01	40	5.671375e-01
10^{-11}	36	5.671375e-01	5	5.671375e-01	7	5.671375e-01	44	5.671375e-01
10^{-12}	36	5.671375e-01	5	5.671375e-01	7	5.671375e-01	48	5.671375e-01

Table 1: Tabella riassuntiva

7 Esercizio

Calcolare la molteplicità della radice nulla della funzione $f(x) = x^2 sin(x^2)$. *Confrontare, quindi, i metodi di Newton, Newton modificato, e di Aitken, per approssimarla per gli stessi valori di tol del precedente esercizio (ed utilizzando il medesimo criterio di arresto), partendo da* $x_0 = 1$. *Tabulare e commentare i risultati ottenuti.*

Abbiamo preso in esame il numero di valutazioni di funzioni e la velocità di convergenza:
La radice della funzione che vogliamo individuare è $x_0 = 0$, molteplicità pari a 4. Possiamo assumere quindi che:

- Newton: 2 valutazioni di funzione; velocità lineare per radici multiple.
- Aitken: 2 volte il costo di Newton, quindi 4 valutazione di funzione; velocità quadratica.
- NewtonMod: 2 valutazione di funzione; ripristina la convergenza quadratica ma richiede il calcolo della molteplicità della funzione.

Newton richiede un numero di passaggi molto superiore agli altri due metodi che invece richiedono più o meno lo stesso numero di iterazioni. Aitken effettua un numero di iterazioni maggiori rispetto a Newton modificato. Come suggerito dalla teoria Aitken e Newton modificato risultano essere più efficienti nel caso di funzioni con radici multiple ripristinando la convergenza quadratica.

Tol	**i**	**Newton**	**i**	**Newton Mod**	**i**	**Aitken**
10^{-1}	2	3.843179e-01	1	-2.179581e-01	2	3.261377e-04
10^{-2}	11	2.880514e-02	1	-2.179581e-01	3	6.492908e-19
10^{-3}	19	2.883766e-03	1	-2.179581e-01	3	6.492908e-19
10^{-4}	27	2.887023e-04	3	8.835818e-05	3	6.492908e-19
10^{-5}	35	2.890282e-05	4	0	4	0
10^{-6}	43	2.893546e-06	4	0	4	0
10^{-7}	51	2.896813e-07	4	0	4	0
10^{-8}	59	2.900084e-08	4	0	4	0
10^{-9}	67	2.903359e-09	4	0	4	0
10^{-10}	75	2.906637e-10	4	0	4	0
10^{-11}	83	2.909919e-11	4	0	4	0
10^{-12}	91	2.913205e-12	4	0	4	0

Table 2: Tabella riassuntiva

```
function [x,i] = aitken(f,f1, x0, tol, itmax)
% function x = aitken(f, x0, itmax, tol)
%f: funzione da analizzare
%f1: derivata della funzione f
%x0: punto di innesco
%imax: numero massimo di iterazioni per trovare x
%tol: parametro che stabilisce la tolleranza dell'errore su x
i=0;
x=x0;
cond=1;
while i<itmax && cond
    i=i+1;
    x0=x;
    fx=feval(f,x0);
    f1x=feval(f1,x0);
    x1=x0-fx/f1x;
    fx=feval(f,x1);
    f1x=feval(f1,x1);
    x=x1-fx/f1x;
    x=(x*x0-x1^2)/(x-2*x1+x0);
    cond=abs(fx)>tol*(1+abs(x))*abs(f1x) && abs(x-x0)>=tol*(1+abs(x));
end
if (cond)
    disp("AITKEN: valore della radice non sufficientemente preciso")
end
return
end
```

```
function [x,i]=newtonmod(f,f1, x0, tol, itmax, molt)
% function x=newton(f, x0, itmax, tol, molt)
%f: funzione da analizzare
%x0: punto di innesco
%imax: numero massimo di iterazioni per trovare x
%tol: parametro per determinare l'accuratezza sul valore di x
%molt: moltiplicità della radice cercata
%Il metodo di Newton modificato ricerca gli zeri in una funzione
%conoscendo la molteplicità della radice
fx=feval(f,x0);
f1x=feval(f1,x0);
x=x0-molt*(fx/f1x);
fx=feval(f,x);
f1x=feval(f1,x0);
i=1;
while i<itmax && abs(fx)>tol*(1+abs(x))*abs(f1x) && abs(x-x0)>=tol*(1+abs(x))
    i=i+1;
    x0=x;
    x=x0-molt*(fx/f1x);
    fx=feval(f,x);
    f1x=feval(f1,x);
end
if (i==itmax)
    disp("NEWTONMOD: valore della radice non sufficientemente preciso");
end
return
end
```

8 Esercizio

Scrivere una function Matlab che, data in ingresso una matrice A, restituisca una matrice, LU, che contenga l'informazione sui suoi fattori L ed U , ed un vettore p contenente la relativa permutazione, della fattorizzazione LU con pivoting parziale di A: function [LU,p] = palu(A)
Curare particolarmente la scrittura e l'efficienza della function.

```
function [LU,p] = palu(A)
%[LU,p]=palu(A)
%A: Matrice da fattorizzare
%Esegue la fattorizzazione LU sulla matrice passata e la
%restituisce insieme al relativo vettore delle permutazioni.
LU=A;
[m,n]=size(LU);
if m~=n
    error("la matrice non e' quadrata");
end
p = (1:n);
for i = 1:n-1
    [mi,ki]=max(abs(LU(i:n,i)));
    if mi==0
        error("la matrice e' singolare");
    end
    ki=ki+i-1;
    if ki>i
        LU([i ki], :) = LU([ki i], :);
        p([i ki])= p([ki i]);
    end
    LU(i+1:n, i)= LU(i+1:n,i)/LU(i,i);
    LU(i+1:n,i+1:n)= LU(i+1:n, i+1:n)- LU(i+1:n,i)*LU(i,i+1:n);
end
return
end
```

Abbiamo utilizzato una sola matrice per restituire la fattorizzazione LU.

9 Esercizio

Scrivere una function Matlab che, data in ingresso la matrice LU ed il vettore p creati dalla function del precedente esercizio, ed il termine noto del sistema lineare $Ax = b$, ne calcoli la soluzione:
$function x = lusolve(LU, p, b)$
Curare particolarmente la scrittura e l'efficienza della function.

```
function x=lusolve(A,p,b)
%x=lusolve(LU,p,b)
%LU: matrice in ingresso fattorizzata LU
%p: vettore delle permutazioni
%b: vettore dei termini noti del Sistema
%lusolve risolve il sistema lineare LUx=b
b=b(p);
[m,n]= size(A);
k=length(b);
x=b(:);
if abs(m-n)>0 || abs(k-n)>0
    error("wrong input data");
end
for i=1:n
    x(i+1:n)=x(i+1:n)-A(i+1:n,i)*x(i);
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
return
end
```

10 Esercizio

Scaricare la function cremat al sito:
http://web.math.unifi.it/users/brugnano/appoggio/cremat.m
che crea sistemi lineari $n \times n$ la cui soluzione è il vettore $x = (1 ..... n)^T$
Eseguire, quindi, lo script Matlab:

```
format long e;
n = 10;
x = zeros(n,15);
u=2^(-52);
for j = 1:15
[A,b] = cremat(n,j);
k=cond(A);
fprintf("%e \n", k);
[LU,p] = palu(A);
x(:,j) = lusolve(LU,p,b);
end
```

k	Errore relativo	k	Errore relativo	k	Errore relativo	k	Errore relativo	k	Errore relativo	k	Errore relativo
	1.332268e-15		1.987299e-14		4.949596e-12		3.149709e-10		7.199294e-08		1.799836e-06
	4.218847e-15		2.553513e-14		7.220002e-12		4.591381e-10		1.049451e-07		2.623646e-06
	1.480297e-15		6.217249e-15		1.542322e-12		9.808717e-11		2.241982e-08		5.604994e-07
	8.881784e-16		1.332268e-14		3.820833e-12		2.430385e-10		5.555127e-08		1.388791e-06
10	2.131628e-15	10^3	2.717826e-14	10^5	7.702639e-12	10^7	4.899320e-10	10^9	1.119836e-07	10^{11}	2.799608e-06
	5.921189e-16		1.051011e-14		3.176422e-12		2.020411e-10		4.618050e-08		1.154520e-06
	3.806479e-16		6.724779e-15		2.092675e-12		1.330504e-10		3.041119e-08		7.602850e-07
	2.220446e-16		6.661338e-16		1.727507e-13		1.098655e-11		2.511223e-09		6.278100e-08
	1.973730e-16		5.131698e-15		1.476153e-12		9.388697e-11		2.145975e-08		5.364974e-07
	0.000000e+00		1.953993e-15		5.245582e-13		3.338663e-11		7.631213e-09		1.907816e-07

k	Errore relativo	k	Errore relativo
	2.249997e-04		1.831847e-02
	3.279851e-04		2.670308e-02
	7.006871e-05		5.704682e-03
	1.736145e-04		1.413492e-02
10^{13}	3.499824e-04	10^{15}	2.849401e-02
	1.443280e-04		1.175054e-02
	9.504414e-05		7.738070e-03
	7.848328e-06		6.389758e-04
	6.706818e-05		5.460392e-03
	2.384984e-05		1.941748e-03

Table 3: Variazione dell'errore relativo al crescere del numero di condizionamento

Per costruzione la soluzione $\mathbf{x}$ del sistema lineare A$\mathbf{x}$=$\mathbf{b}$ è sempre il vettore $\mathbf{x} = (1,2,3,4,5,6,7,8,9,10)^T$. Osservando i risultati ottenuti dal codice Matlab ci rendiamo conto tuttavia che è presente un errore introdotto inizialmente in ingresso dalla matrice A e dal vettore dei termini noti $\mathbf{b}$ e amplificato dal coefficiente di errore k. Sfruttando la funzione Matlab $k = cond(A)$ possiamo stabilire, durante le varie iterazioni, il coefficiente di condizionamento della matrice A.
Poiché l'errore in ingresso è rappresentato dal numero di macchina $u = \frac{1}{2} * 2^{1-52}$ si avrà che
$\frac{||\Delta x||}{||x||} \leq k * (\frac{||\Delta A||}{||A||} + \frac{||\Delta b||}{||b||})$
k cresce esponenzialmente ed ha un valore 10^i con *i=1, ..., 15.* Si può quindi osservare come al crescere di k

l'errore sui dati in x aumenti, interessando ad ogni passo una cifra significativa in più rispetto alla iterazione precedente.

11 Esercizio

Scrivere una function Matlab che, data in ingresso una matrice $A \in R$ $m \times n$, con $m \geq n = rank(A)$, restituisca una matrice, QR, che contenga l'informazione sui fattori Q ed R della fattorizzazione QR di A:
function QR = myqr(A)
Curare particolarmente la scrittura e l'efficienza della function.

```
1  function QR= myqr(A)
2  %QR=myqr(A)
3  %A: matrice da fattorizzare
4  %Esegue la fattorizzazione QR sulla matrice in ingresso
5  QR=A;
6  [m,n]=size(QR);
7  for i=1:n
8      alfa= norm(QR(i:m,i));
9      if alfa==0
10         error("la matrice QR non ha il rango massimo")
11     end
12     if QR(i,i)>=0
13         alfa=-alfa;
14     end
15     v1=QR(i,i)-alfa;
16     QR(i,i)=alfa;
17     QR(i+1:m,i)=QR(i+1:m,i)/v1;
18     beta=-v1/alfa;
19     QR(i:m,i+1:n)=QR(i:m,i+1:n)-(beta*[1;QR(i+1:m,i)])*...
20         ([1 QR(i+1:m,i)']*QR(i:m,i+1:n));
21 end
22 return
23 end
```

Il codice è stato ottimizzato prestando particolare attenzione all'ordine di esecuzione delle operazioni all'istruzione 20, viene svolto infatti prima il prodotto tra riga e matrice e successivamente il prodotto colonna per riga; questo evitare di dover fare il prodotto tra due matrici che risulterebbe più oneroso.
Per evitare il verificarsi di operazioni mal condizionate viene realizzato un controllo sul segno di α.
Per ottimizzare il costo computazionale, in termini di occupazione di memoria, viene utilizzato uno scalamento del vettore di Householder in modo che la prima componente sia 1, e quindi nota, questo ci permette di riscrivere la matrice in ingresso con la sua fattorizzazione QR.

12 Esercizio

Scrivere una function Matlab che, data in ingresso la matrice QR creata dalla function del precedente esercizio, ed il termine noto del sistema lineare $A\boldsymbol{x} = \boldsymbol{b}$*, ne calcoli la soluzione nel senso dei minimi quadrati:*
function x = qrsolve(QR,b)
Curare particolarmente la scrittura e l'efficienza della function.

```
function x = qrsolve2(QR,b)
% x=qrsolve(QR,b)
%QR: Matrice fattorizzata QR
%b: Vettore dei termini noti
%Risolve il sistema lineare QRx=b
[m,n] = size(QR);
x = b;
for i = 1:n
     v = [1; QR(i+1:m,i)];
     beta = 2/(v'*v);
     x(i:m) = x(i:m) -( beta*( v'*x(i:m) ) )*v;
end
x = x(1:n);
for i = n:-1:1
     x(i) = x(i)/QR(i,i);
     if i>1
        x(1:i-1) = x(1:i-1)-x(i)*QR(1:i-1,i);
     end
end
return
end
```

13 Esercizio

Scaricare la function cremat1 al sito:
http://web.math.unifi.it/users/brugnano/appoggio/cremat1.m
che crea sistemi lineari $m \times n$, con $m \geq n$, la cui soluzione (nel senso dei minimi > quadrati) è il vettore $x = [1 \ldots n]$. Eseguire, quindi, il seguente script Matlab per testare le function dei precedenti esercizi:

```
1  format long e
2
3  for n = 5:10
4      xx = [1:n]';
5      for m = n:n+10
6          [A,b] = cremat1(m,n);
7          QR = myqr(A);
8          x = qrsolve2(QR,b);
9          disp([m n norm(x-xx)])
10     end
11 fprintf("\n\n\n");
12 end
```

n	**m**	$\|\|\mathbf{x}-\hat{\mathbf{x}}\|\|$	**n**	**m**	$\|\|\mathbf{x}-\hat{\mathbf{x}}\|\|$	**n**	**m**	$\|\|x-\hat{\mathbf{x}}\|\|$	**n**	**m**	$\|\|\mathbf{x}-\hat{\mathbf{x}}\|\|$	**n**	**m**	$\|\|\mathbf{x}-\hat{\mathbf{x}}\|\|$	**n**	**m**	$\|\|\mathbf{x}-\hat{\mathbf{x}}\|\|$
5	5	1.556072e-13	6	6	5.387969e-14	7	7	2.751495e-14	8	8	9.405181e-14	9	9	7.025738e-14	10	10	6.126751e-14
5	6	2.147557e-14	6	7	1.288672e-14	7	8	5.360435e-14	8	9	2.270101e-14	9	10	7.118322e-14	10	11	1.234525e-13
5	7	1.295540e-14	6	8	2.605881e-14	7	9	9.272855e-15	8	10	4.137186e-14	9	11	5.895901e-14	10	12	6.826165e-14
5	8	2.491752e-14	6	9	1.700346e-14	7	10	2.152459e-14	8	11	3.244967e-14	9	12	6.920671e-14	10	13	2.702772e-14
5	9	5.277940e-15	6	10	1.374705e-14	7	11	2.651551e-14	8	12	2.895533e-14	9	13	2.216001e-14	10	14	4.810534e-14
5	10	1.166527e-14	6	11	2.100406e-14	7	12	1.875488e-14	8	13	1.401698e-14	9	14	1.492827e-14	10	15	3.951895e-14
5	11	9.678700e-15	6	12	1.282680e-14	7	13	1.041955e-14	8	14	1.209888e-14	9	15	4.352357e-14	10	16	5.550593e-14
5	12	4.022931e-15	6	13	1.057921e-14	7	14	2.360889e-14	8	15	3.775411e-14	9	16	2.845985e-14	10	17	2.355793e-14
5	13	5.626104e-15	6	14	5.621720e-15	7	15	1.645529e-14	8	16	1.512513e-14	9	17	1.653897e-14	10	18	1.929234e-14
5	14	5.388872e-15	6	15	8.358438e-15	7	16	7.695055e-15	8	17	1.154045e-14	9	18	2.178302e-14	10	19	2.189815e-14
5	15	4.720733e-15	6	16	6.620505e-15	7	17	1.575109e-14	8	18	1.360284e-14	9	19	2.634014e-14	10	20	1.879919e-14

Table 4: Errore assoluto QRsolve

La linea di codice 9 normalizza il vettore risultante dato dalla differenza tra le soluzioni in aritmetica esatta e quelle ottenute tramite le nostre funzioni. Possiamo vedere, dalla tabella, che l'errore sui dati in uscita rientra in un range accettabile (nell'ordine di 10^{-14}).
Alla luce di questi risultati, siamo ragionevolmente convinti che il nostro codice funzioni correttamente.

14 Esercizio

Scrivere un programma che implementi efficientemente il calcolo del polinomio interpolante su un insieme di ascisse distinte.
Abbiamo deciso di calcolare il polinomio interpolante con la forma di Newton, piuttosto che con la forma di Lagrange, perché generare il polinomio in modo incrementale si presta ad essere computazionalmente più efficiente.

```
function y = newton(xi,fi,x)
% y = Newton(xi,fi,x): Calcola il valore del polinomio interpolante la
% funzione sulle ascisse passate.
%xi: Ascisse di interpolazione
%fi: Valori assunti dalla funzione nelle ascisse di interpolazione
%x: Ascisse su cui si vuole calcolare il valore del polinomio
n=length(xi);
if n~= length(fi);
    error('Dati inconsistenti');
end
for i = 1 : n-1
    for j= i+1:n
        if xi(i)==xi(j)
            error('Ascisse non distinte');
        end
    end
end
f=difdiv(xi,fi);
y=f(n);
for i= n-1 : -1 : 1
    y=y.*(x-xi(i)) + f(i);
end
return
```

```
function f = difdiv(xi,fi)
% f = difdiv(xi,fi)
% xi= ascisse di interpolazione
% fi= valore della funzione nelle ascisse
% La fuzione calcola le differenze divise utilizzate per il calcolo del
% polinomio
f=fi;
n=length(xi);
for i = 1 : n-1
    for j = n : -1 : i+1
        f(j) = (f(j) - f(j-1))/(xi(j) - xi(j-i));
    end
end
return
```

15 Esercizio

Scrivere un programma che implementi efficientemente il calcolo del polinomio interpolante di Hermite su un insieme di ascisse distinte.

```
function ff = difdivherm(xx,ff)
% ff = difdivherm(x,fi,f1)
% x: Ascisse di interpolazione
% fi: Valori della funzione
% f1: Valori della derivata prima
% Funzione che calcola le differenze divise per il metodo di Hermite
n=length(xx)-1;
for i = n : -2 : 3
    ff(i) = (ff(i)-ff(i-2))/(xx(i)-xx(i-2));
end
for i = 2 : n
    for j = n+1 : -1 : i+1
        ff(j) = (ff(j)-ff(j-1))/(xx(j)-xx(j-i));
    end
end
return
```

```
function y = hermite(xi, fi, f1,  x)
% y = hermite(xi, fi, f1)
% xi: Ascisse di interpolazione
% fi: Valore della funzione
% f1: Valore della derivata
% x: Ascisse su cui valutare il polinomio
%  La funzione calcola il polinomio interpolantela funzione f
n=length(xi);
if n~= length(fi) || n~= length(f1)
    error('Dati inconsistenti');
end
for i = 1 : n-1
    for j= i+1:n
        if xi(i)==xi(j)
            error('Ascisse non distinte');
        end
    end
end
n=2*n -1;
xx(1:2:n)=xi;
xx(2:2:n+1)=xi;
ff(1:2:n)=fi;
ff(2:2:n+1)=f1;
ff=difdivherm(xx,ff);
y=ff(n);
for i= n-1 : -1 : 1
    y=y.*(x-xx(i)) + ff(i);
end
return
```

16 Esercizio

Scrivere un programma che implementi efficientemente il calcolo di una spline cubica naturale interpolante su una partizione assegnata.
Per garantire la massima efficienza abbiamo fattorizzato LU utilizzando esclusivamente le tre diagonali seguendo il seguente algoritmo:

SCOMPOSIZIONE LU PER UNA MATRICE TRIDIAGONALE $Ax = f$

$$\begin{pmatrix} b_1 & \varepsilon_1 & & & 0 \\ \varphi_2 & b_2 & .. & & \\ & .. & .. & .. & \\ & & .. & b_{n-1} & \varepsilon_{n-1} \\ 0 & & & \varphi_n & b_n \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ \ell_2 & 1 & & & \\ & \ell 3 & .. & & \\ & & .. & .. & \\ & & & \ell_n & 1 \end{pmatrix} \begin{pmatrix} \nu_1 & \varepsilon_1 & & & \\ & \nu_2 & \varepsilon_2 & & \\ & & .. & & \\ & & & \nu_{n-1} & \varepsilon_{n-1} \\ & & & & \nu_n \end{pmatrix}$$

PER DETERMINARE L, U:

$$\begin{aligned} \nu_1 &= b_1 \\ \varphi_k = \ell_k \nu_{k-1} &\Rightarrow \ell_k = \varphi_k \not{\nu}_{k-1} \\ b_k = \ell_k \varepsilon_{k-1} + \nu_k &\Rightarrow \nu_k = b_k - \ell_k \varepsilon_{k-1} \ , \qquad k = 2, ..., n \end{aligned}$$

PER RISOLVERE $Ly = f$:

$$\begin{aligned} y_1 &= f_1 \\ l_k y_{k-1} + y_k = f_k &\Rightarrow y_k = f_k - l_k y_{k-1}, \qquad k = 2, ..., n \end{aligned}$$

PER RISOLVERE $Ux = y$

$$\begin{aligned} \nu_n x_n = y_n &\Rightarrow x_n = y_n / \nu_n \\ \nu_k x_k + \varepsilon_k x_{k+1} = y_k &\Rightarrow x_k = (y_k - \varepsilon_k x_{k+1}) / v_k \ , \qquad k = n-1, .., 1 \end{aligned}$$

```
function s = spline3naturale(x, y, xx)
% s = spline3naturale(x, y, xx)
% Funzione che valuta le ascisse xx nella spline naturale interpolante la funzione passante
  per (x, y)
% x: ascisse di interpolazione
% y: valori assunti dalla funzione nei punti di interpolazione
% xx: punti in cui valutare la Spline
n=length(x);
hi= x(2:n) - x(1:n-1);
dd = (y(2:n) - y(1:n-1))./hi;

sup = (hi(2:end-1))./(hi(1:end-2)+hi(2:end-1));
infe = (hi(2:end-1))./(hi(2:end-1)+hi(3:end));
dd = 6*( (dd(2:end) - dd(1:end-1)) ./ (x(3:end)-x(1:end-2) ));

m = calcM(infe, sup, dd);
m = [0 m 0];

ri = zeros(1, n);
qi = zeros(1, n);
for i = 2:n
    ri(i-1) = y(i-1) - ((hi(i-1)^2)/6) * m(i-1);
    qi(i-1) = ( y(i)-y(i-1) )/ hi(i-1) - (hi(i-1)/6) * (m(i) - m(i-1));
end
s = evalSpline(ri, qi, xx, x, m, hi);
return
```

```
function m = calcM(inf, sup, b)
% m = calcM(inf, sup, b)
% Risolve il sistema tridiagonale ad hoc per ricavare
%il vettore m necessario alla valutazione di una spline cubic naturale
% inf: diagonale inferiore
% sup: diagonale superiore
% b: vettore delle differenze divise
n = length(b);
v(1) = 2;
l(1)=0;
for i = 2:n
    l(i) = inf(i-1)/v(i-1);
    v(i) = 2-l(i)*sup(i-1);
end
y(1) = b(1);
for j = 2:n
    y(j) = b(j)-l(j)*y(j-1);
end
m(n) = y(n)/v(n);
for k = n-1:-1:1
    m(k) = (y(k)-sup(k)*m(k+1))/v(k);
end
return
```

```
function s = evalSpline(ri, qi, xx, xi, m, hi)
% s = evalSpline(ri, qi, xx, xi, m, hi)
% Funzione che valuta una spline nei punti xx
% ri: vettore costanti di integrazione
% qi: vettore costanti di integrazione
% xx: ascisse in cui valutare il polinomio
% xi: ascisse di interpolazione
N = length(xx);
s = zeros(N,1)';
for j = 1 : N
    i = getRange(xi, xx(j));
    s(j) = ( ((xx(j)-xi(i-1))^3*m(i) + ((xi(i)-xx(j))^3)*m(i-1) )/ ...
        (6*hi(i-1)) ) + qi(i-1)*(xx(j)-xi(i-1)) + ri(i-1);
end
return
```

```
function i = getRange(xi, xx)
% i = getRange(xi, xx)
% Funzione che dato un punto xx stabilisce l'indice delle ascisse di interpolazione tra cui
  si trova
% xx: punto di valutazione
% xi: vettore delle ascisse di interpolazione
for i = 2 : length(xi)
    if xx <= xi(i)
        return;
    end
end
return
```

17 Esercizio

Scrivere un programma che implementi il calcolo di una spline cubica not-a-knot interpolante su una partizione assegnata.

```
function m = calcMnotaknot(diag, inf, sup, b)
% m = calcM(inf, sup, b)
% Funzione che calcola il vettore m necessario al calcolo della spline cubic naturale
% inf: diagonale inferiore
% sup: diagonale superiore
% b: vettore delle differenze divise
n = length(b);
v(1) = diag(1);
l(1)=0;
for i = 2:n
    l(i) = inf(i-1)/v(i-1);
    v(i) = diag(i)-l(i)*sup(i-1);
end
y(1) = b(1);
for j = 2:n
    y(j) = b(j)-l(j)*y(j-1);
end
m(n) = y(n)/v(n);
for k = n-1:-1:1
    m(k) = (y(k)-sup(k)*m(k+1))/v(k);
end
return
```

```
function s = spline3notaknot(x, y, xx)
% s = spline3ok(x, y, xx)
% Funzione che valuta le ascisse xx nella spline notaknot interpolante la funzione passante
  per (x, y)
% x: ascisse di interpolazione
% y: valori assunti dalla funzione nei punti di interpolazione
% xx: punti in cui valutare la Spline
n=length(x);
hi= x(2:n) - x(1:n-1);
dd = (y(2:n) - y(1:n-1))./hi;
sup = (hi(2:end-1))./(hi(1:end-2)+hi(2:end-1));
infe = (hi(2:end-1))./(hi(2:end-1)+hi(3:end));
dd = 6*( (dd(2:end) - dd(1:end-1)) ./ (x(3:end)-x(1:end-2) ));
dd = [dd(1); dd; dd(end)];

infe = [hi(1)/(hi(1)+hi(2)); infe ; 0];
sup = [0; sup; hi(end)/(hi(end-1) + hi(end))];

infe(end-1) = infe(end-1) - sup(end);
sup(2) = sup(2) - infe(1);

diag = 2*ones(n-2, 1);
diag = [1; diag; 1];
diag(2) = diag(2) - infe(1);
diag(end-1) = diag(end-1) - sup(end);

m = calcMnotaknot(diag, infe, sup, dd);
m(1) = m(1) - m(2) - m(3);
m(end) = m(end) - m(end-1) - m(end-2);

ri = zeros(1, n);
qi = zeros(1, n);
for i = 2:n
   ri(i-1) = y(i-1) - ((hi(i-1)^2)/6) * m(i-1);
   qi(i-1) = ( y(i)-y(i-1) )/ hi(i-1) - (hi(i-1)/6) * (m(i) - m(i-1));
end
s = evalSpline(ri, qi, xx, x, m, hi);
return
```

18 Esercizio

Confrontare i codici degli esercizi 14–17 per approssimare la funzione $f(x) = sin(x)$ *sulle ascisse* $x_i = \frac{\pi i}{n}$, $i = 0..n$, *per* $n = 1, 2, \ldots, 10$. *Graficare l'errore massimo di approssimazione verso n (in semilogy), calcolato su una griglia uniforme di 10001 punti nell'intervallo [0,* π*].*

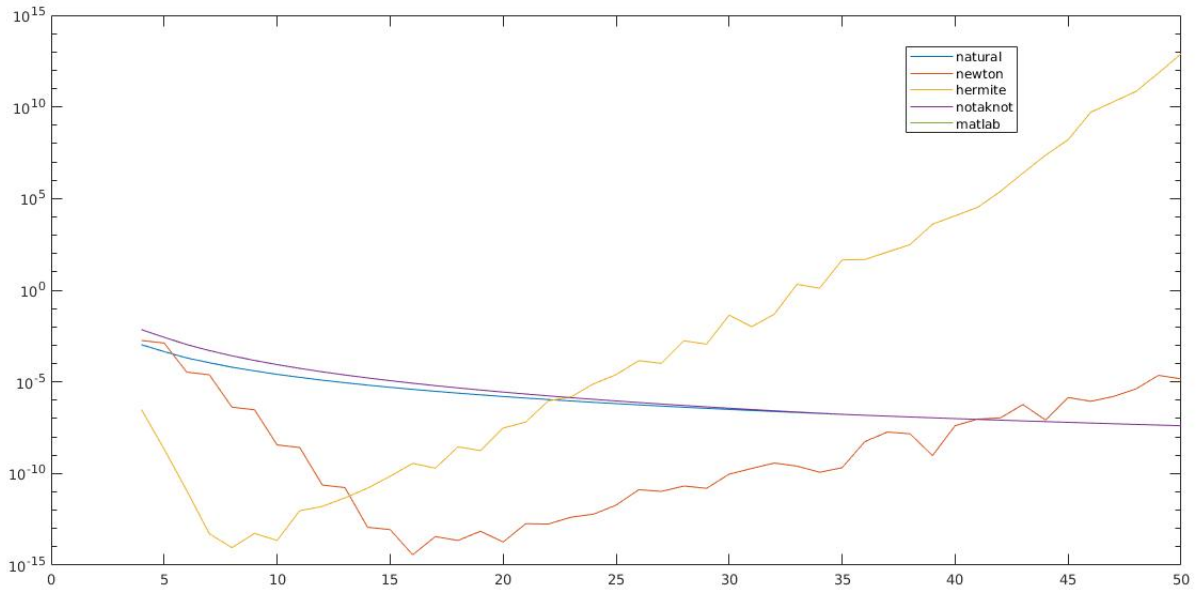


Figure 1: Massimo errore di interpolazione

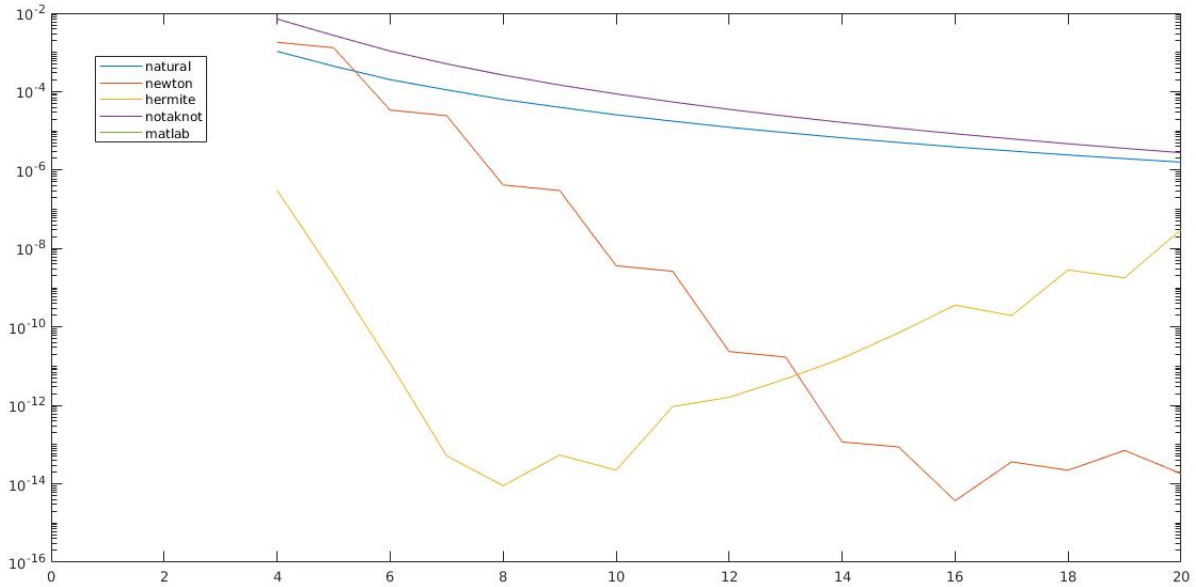


Figure 2: Dettaglio errore

Gli errori sono stati graficati con n>4 , in quanto la spline cubica not-a-knot necessita di almeno cinque punti per essere definita. Possiamo notare come la not-a-knot si comporti essattamente come la funzione spline di Matlab,

mentre la naturale risulta essere un approssimazione migliore per n piccoli per poi convergere come la not-a-knot al crescere di n.

19 Esercizio

Calcolare (numericamente) la costante di Lebesgue per i polinomi interpolanti di grado n=2,4 ... 40 sia sulle ascisse equidistanti che su quelle di Chebyshev (utilizzare 10001 punti equispaziati per valutare la funzione di Lebesgue). Graficare convenientemente i risultati ottenuti. Spiegare, quindi, i risultati ottenuti approssimando la funzione:
$f(x) = \frac{1}{1+x^2}$ *con* $x \in [-5, 5]$
utilizzando le ascisse equidistanti e di Chebyshev precedentemente menzionate (tabulare il massimo errore valutato su una gliglia 10001 punti equidistanti nell'intervallo [−5, 5]).

n	**Chebyshev**	**Equidistanti**
2	1.666667e+00	1.250000e+00
4	1.988854e+00	2.207824e+00
6	2.202215e+00	4.549341e+00
8	2.361857e+00	1.094565e+01
10	2.489430e+00	2.989995e+01
12	2.595678e+00	8.932490e+01
14	2.686715e+00	2.832107e+02
16	2.766353e+00	9.345315e+02
18	2.837132e+00	3.171357e+03
20	2.900825e+00	1.098666e+04
22	2.958723e+00	3.867132e+04
24	3.011793e+00	1.378515e+05
26	3.060778e+00	4.965070e+05
28	3.106262e+00	1.803783e+06
30	3.148712e+00	6.601060e+06
32	3.188509e+00	2.430881e+07
34	3.225963e+00	9.000917e+07
36	3.261336e+00	3.348870e+08
38	3.294847e+00	1.251203e+09
40	3.326682e+00	4.692429e+09

Figure 3: Andamento della costante di Lebesque al crescere del grado

n	Chebyshev	Equidistanti
2	6.005977e-01	6.462292e-01
4	4.020169e-01	4.383571e-01
6	2.642274e-01	6.169479e-01
8	1.708356e-01	1.045177e+00
10	1.091535e-01	1.915659e+00
12	6.921571e-02	3.663393e+00
14	4.660235e-02	7.194881e+00
16	3.261358e-02	1.439385e+01
18	2.249229e-02	2.919044e+01
20	1.533372e-02	5.982231e+01
22	1.035891e-02	1.236243e+02
24	6.948424e-03	2.572129e+02
26	4.634871e-03	5.381745e+02
28	3.078217e-03	1.131420e+03
30	2.061588e-03	2.388281e+03
32	1.401747e-03	5.058960e+03
34	9.493348e-04	1.074905e+04
36	6.407501e-04	2.290123e+04
38	4.312103e-04	4.890719e+04
40	2.894608e-04	1.046677e+05

Table 5: Andamento errore interpolazione al variare del grado

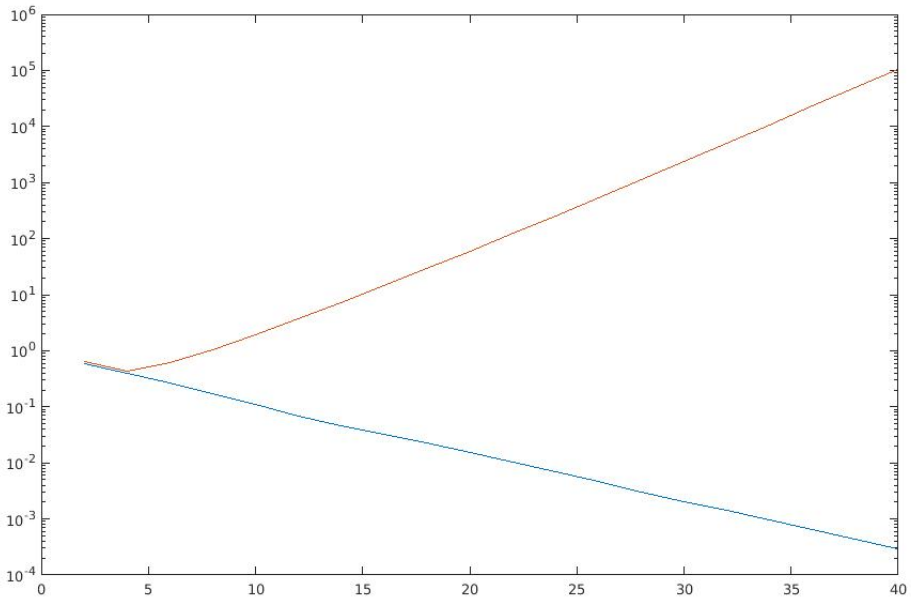


Figure 4: Andamento dell'errore di interpolazione al variare del grado

Il malcondizionamento dovuto ai dati in ingresso e l'errore che si commette nell'interpolare una funzione sono strettamente legati, come possiamo vedere infatti dalla seguente formula:
$||e|| \leq \alpha(1 + \Lambda)w(f, \frac{b-a}{n})$
La costante di Labesque, che esercita quindi una parte fondamentale per tenere sotto controllo l'errore, discende dalla omonima funzione, che altro non è che la sommatoria dei prodotti tra i rapporti delle distanze tra le ascisse;

chiamata base di Lagrange, è costituita da polinomi specifici ottenuti per costruzione sulle ascisse, di cui quest' ultime ne determinano i gradi massimi; la scelta di punti diversi influisce quindi sulla precisione che si desidera ottenere.
Sotto un altra forma l'errore commesso nell'interpolare la funzione si può esprimere anche come:

$$||e|| = f[x_0....x_n, x] * ||w_{n+1}||$$

ed è anch'esso strettamente legato alla scelta delle ascisse: il secondo termine infatti rappresenta la produttoria della distanza tra le ascisse di interpolazione al variare delle iterazioni; ancora una volta è quindi chiaro di come la scelta delle ascisse influenzi la precisione di approssimazione.
Alla luce di queste considerazioni si osserva dalla Tabella 1 che con la scelta delle ascisse equidistanti si ottiene un aumento esponenziale della costante di Labesque all'aumentare del grado del polinomio interpolante; la scelta delle ascisse di Chebyshev invece minimizza la seconda componente e tiene sotto controllo la costante.
Dal grafico oltretutto notiamo di come l'errore massimo segua l'andamento della costante e cresca irrimediabilmente per le ascisse Equidistanti e vada invece a diminuire con Chebyshev.

```
function error = interpError(f,a,b)
%f = errorInterpolation(f,a,b)
%f: funzione da approssimare
%a,b: intervallo di riferimento
%Calcola gli errori ottenuti tramite l'interpolazione utilizzando le
%ascisse equidistanti e di Chebyshev
x = linspace(a,b,10001);
ff= feval(f,x(1:end));
error = zeros(2,20);
i=1;
for n=(2:2:40)
    xi = cheby(n,a,b);
    fi = feval(f,xi);
    y = newton(xi,fi,x);
    error(1,i) = max(abs(ff - y));
    xi = equidist(n,a,b);
    fi = feval(f,xi);
    y = newton(xi,fi,x);
    error(2,i) = max(abs(ff - y));
    i=i+1;
end
return
```

```
function A = calcLeb(a,b)
%A = calcLeb(a,b)
%a,b: Range dell'intervallo
% Calcola le costanti di Labesque per ascisse equidistanti e di Chebyshev
A=zeros(20,2);
x = linspace(a,b,10001);
k=1;
for i = 2 : 2 : 40
    xi = cheby(i,a,b);
    A(k,1) = labesque(xi,x);
    xi = equidist(i,a,b);
    A(k,2)= labesque(xi,x);
    k=k+1;
end
return
```

```
function leb = labesque(xi,x)
%leb = labesque(xi,x)
%xi: ascisse di interpolazione
%x: ascisse su cui calcolare la costante di lebesque
%Calcola la costante di Labesque per le ascisse passate
n = length(xi);
y = zeros(size(x));
for i = 1:n
    li=1;
    for k = [1:i-1 i+1:n]
        li = li.*(x-xi(k))/(xi(i)-xi(k));
    end
    y= y + abs(li);
end
leb = max(y);
end
```

```
function x = equidist(n,a,b)
%x = equidist(n,a,b)
%n: grado interpolazione
%a,b: estremi intervallo
%Calcola il vettore contenente le ascisse di interpolazione
%equidistanti
x = zeros(1,n+1);
for i = 1 : n+1
    x(i)=a + (i-1)*((b-a)/(n));
end
return
```

```
function x = cheby(n,a,b)
%x = cheby(n,a,b)
%n: grado di interpolazione
%a,b: estremi intervallo
%Calcola il vettore delle  ascisse di interpolazione secondo il metodo di
%Chebyshev
if a>=b || n~=fix(n) || n<0
    error('dati errore');
end
x= cos((2*(0:n)+1)*(pi/(2*n+2)));
x= ((a+b)+ (b-a)*x)/2;
return
```

20 Esercizio

Con riferimento al precedente esercizio, tabulare il massimo errore di approssimazione (calcolato come sopra indicato), sia utilizzando le ascisse equidistanti che quelle di Chebyshev su menzionate, relativo alla spline cubica naturale interpolante f (x) su tali ascisse.

n	**Chebyshev**	**Equidistanti**
2	1.668019e+00	6.011945e-01
4	2.632667e+01	2.793134e-01
6	1.706418e+01	1.293001e-01
8	1.636027e+01	5.607385e-02
10	1.996967e+00	2.197383e-02
12	8.928629e+00	6.908801e-03
14	8.994445e+00	2.482863e-03
16	1.154128e+01	3.745403e-03
18	1.394946e+01	3.717999e-03
20	1.667015e+01	3.182858e-03
22	1.969491e+01	2.529653e-03
24	2.297806e+01	1.925792e-03
26	2.654609e+01	1.427048e-03
28	3.038694e+01	1.039053e-03
30	3.450449e+01	8.243623e-04
32	3.889716e+01	6.554987e-04
34	4.356520e+01	5.237082e-04
36	4.850839e+01	4.210036e-04
38	5.372667e+01	3.408378e-04
40	5.921997e+01	2.779765e-04

Figure 5: Errore massimo al variare del grado del polinomio

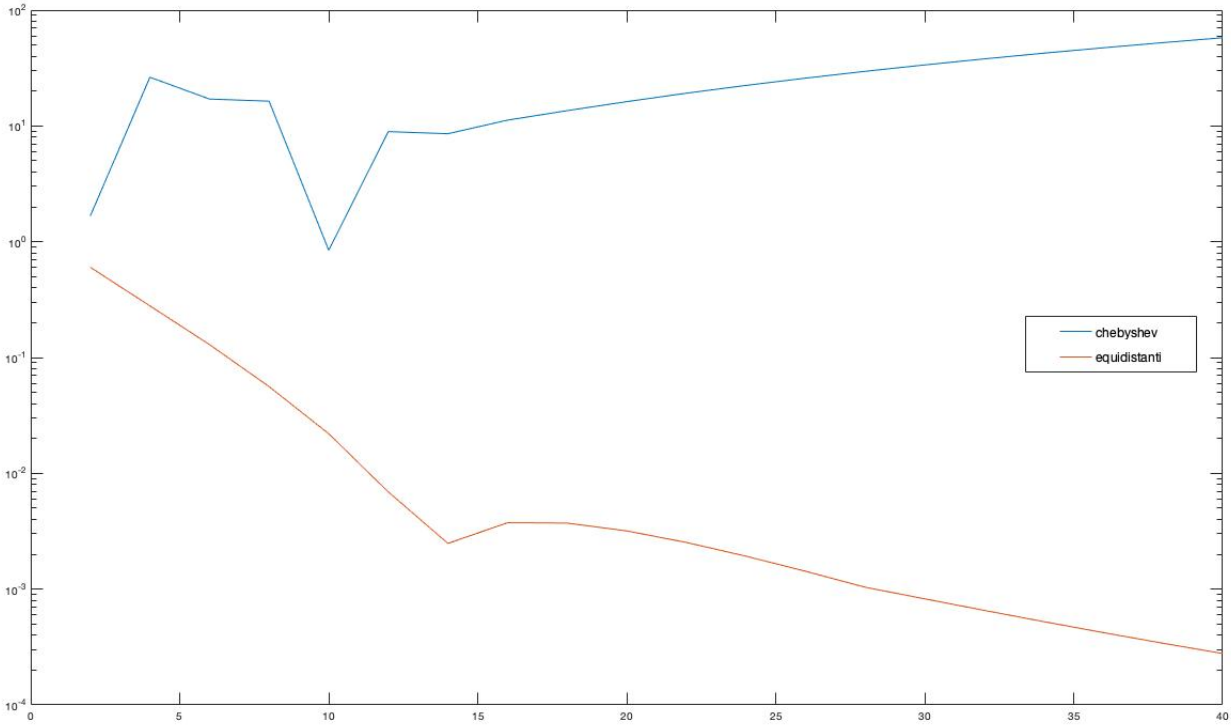


Figure 6: Errore equidistanti, Chebyshev

Alla luce dei nostri risultati, possiamo verificare che per il calcolo di una Spline cubica la scelta migliore sia quella delle ascisse equidistanti.

21 Esercizio

Uno strumento di misura ha una accuratezza di 10^{-6} *(in opportune unità di misura). I dati misurati nelle posizioni* x_i *sono dati da* y_i *, come descritto:*

	x_i	y_i
0	1.000000e-02	1.003626e+00
1	9.800000e-02	1.025686e+00
2	1.270000e-01	1.029512e+00
3	2.780000e-01	1.029130e+00
4	5.470000e-01	9.947810e-01
5	6.320000e-01	9.901560e-01
6	8.150000e-01	1.016687e+00
7	9.060000e-01	1.057382e+00
8	9.130000e-01	1.061462e+00
9	9.580000e-01	1.091263e+00
10	9.650000e-01	1.096476e+00

Calcolare il grado minimo, ed i relativi coefficienti, del polinomio che meglio approssima i precedenti dati nel senso dei minimi quadrati con una adeguata accuratezza. Graficare convenientemente i risultati ottenuti.

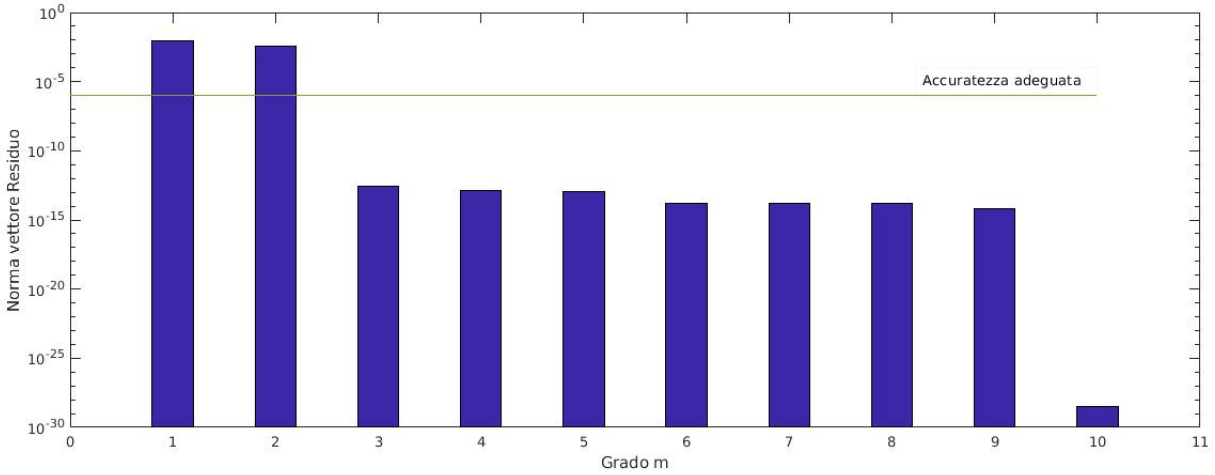


Figure 7: Andamento norma vettore residuo al variare del grado n

n											
1	1.002995e+00	5.812448e-02									
2	1.036730e+00	-2.155936e-01	2.695260e-01								
3	9.999999e-01	3.750012e-01	-1.250003e+00	1.000002e+00							
4	1.000000e+00	3.749964e-01	-1.249978e+00	9.999595e-01	2.205758e-05						
5	1.000000e+00	3.749983e-01	-1.249994e+00	1.000004e+00	-2.942213e-05	2.064508e-05					
6	1.000000e+00	3.749879e-01	-1.249858e+00	9.993716e-01	1.272307e-03	-1.194961e-03	4.216787e-04				
7	1.000000e+00	3.749906e-01	-1.249901e+00	9.996271e-01	5.615871e-04	-1.797733e-04	-2.997339e-04	2.022689e-04			
8	1.000000e+00	3.749907e-01	-1.249903e+00	9.996439e-01	5.013601e-04	-5.976081e-05	-4.345519e-04	2.821712e-04	-1.943227e-05		
9	1.000000e+00	3.749479e-01	-1.248865e+00	9.890000e-01	5.668738e-02	-1.673432e-01	2.917279e-01	-2.965100e-01	1.623091e-01	-3.695525e-02	
10	1.000016e+00	3.729016e-01	-1.196875e+00	4.195465e-01	3.352282e+00	-1.130427e+01	2.342325e+01	-3.025189e+01	2.375817e+01	-1.038937e+01	1.941249e+00

Table 6: Coefficienti polinomio

Come si può vedere dalla figura un grado pari a 3 è sufficientemente preciso da poter essere usato.

```
function b = miniSquare(xi,yi)
%b = miniSquare(xi,yi);
%xi: Punti
%yi: Valori misurati
%Restituisce le norme al quadrato del vettore residuo
%con grado da 1 a n
b=zeros(10,1);
for m = 1 : 10
    V = vander(xi,m);
    QR = myqr(V);
    a = qrsolve2(QR,yi');
    resid=(V*a) - yi';
    b(m,1)=norm(resid,2)^2;
end
return
```

```
function A = vander(xi,m)
% A = vander(xi,m)
%xi: Ascisse
%m: grado
%Crea una matrice di tipo Vandermonde per l'approssimazione ai minimi
%quadrati
A = zeros(length(xi),m+1);
for i = 1 : m+1
    A(:,i) = xi.^(i-1);
end
disp(A);
return
```

22 Esercizio

Scrivere due functions che implementino efficientemente le formule adattattive dei trapezi e di Simpson.

```
function I2=adasim(a,b,f,tol,fa,f1,fb)
%I2=adasim(a,b,f,tol)
%a,b: intervallo
%f: funzione
%tol: tolleranza
%Calcola un' approssimazione dell'integrale definito di f(x) da a a b con
%tolleranza tol (default 1e-5).
x1=(a+b)/2;
if nargin <= 4
    fa = feval(f,a);
    fb = feval (f,b);
    f1 = feval(f,x1);
    if nargin==3
        tol=1e-5;
    end
end
h=(b-a)/6;
x2=(a+x1)/2;
x3=(x1*b)/2;
f2=feaval(f,x2);
f3=feavl(f,x3);
I1=h*(fa+4*f1+fb);
I2=.5*h*(fa+4*f2+2*f1+ 4 *f3+fb);
e=abs(I2-I1)/15;
if e>tol
    I2=adasim(a, x1, f, tol/2,fa,f2,f1) + adasim(x1,b,f,tol/2,f1,f3,fb);
end
return
```

```
function I2 = adatrap(a, b, f, tol, fa, fb)
%I2 = adatrap(a, b, f, tol)
%a,b: intervallo
%f: funzione da  integrare
%tol: tolleranza
%Calcola un' approssimazione dell'integrale definito di f(x) da a a b con
%tolleranza tol (default 1e-5).
x1=(a+b)/2;
f1=feval(f,x1);
if nargin<=4
    fa=feval(f,a);
    fb=feval(f,b);
    if nargin==3
        tol=1e-5;
    end
end
h=(b-a)/2;
I1=h*(fa+fb);
I2=.5*h*(fa+2*f1+fb);
e=abs(I2-I1)/3;
if e>tol
    I2=adtrap(a,xi,f,tol/2,fa,f1)+adatrap(x1,b,f,tol/2,f1,fb);
end
return
```

23 Esercizio

Sapendo che:
$I(f) = \int_0^{atan(30)} (1 + tan(x)^2)dx = 30$
Tabulare il numero dei punti richiesti dalle formule adattive dei trapezi e di Simpson per approssimare I(f) con tolleranze tol $= 10^{-i}$ *, i = 2, . . . , 8, assieme ai relativi errori.*

	Trapezi		***Simpson***	
i	**Punti**	**Errore**	**Punti**	**Errore**
2	375	4.764530e-03	71	2.372538e-03
3	1181	5.737281e-04	113	6.165283e-04
4	3687	5.263955e-05	203	5.016671e-05
5	11883	5.454612e-06	371	2.220652e-06
6	37273	5.635538e-07	635	3.981788e-07
7	116747	5.255171e-08	1145	3.565320e-08
8	375793	5.507658e-09	2045	3.587672e-09

Table 7: Tabella Riassuntiva

```
%Script che calcola il numero di punti richiesti, e relativo errore
%utilizzando le formule adattive di Simpson e Trapezi
a = 0;
b = atan(30);
f = @(x)(1+tan(x).^2);
value = 30;
global globTrap
global globSim
globTrap=2;
globSim=2;
A = zeros(7,4);
k=1;
for i =2:8
    tol=10^-i;
    resuTrap = adatrap(a,b,f,tol) ;
    resuSimp = adasim(a,b,f,tol);
    A(k,1)= globTrap;
    A(k,2)=abs(value-resuTrap);
    A(k,3)=globSim;
    A(k,4)=abs(value-resuSimp);
    fprintf('i= {%d}  Trap: {%d} -- Simp: {%d}\n',i,globTrap,globSim);
    globTrap=2;
    globSim=2;
    k=k+1;
end
```

24 Esercizio

Scrivere una function che implementi efficientemente il metodo delle potenze.

```
function [l1,x1] = pot(A, tol, imax)
%[l1,x1] = pot(A, tol, imax)
%A: Matrice
%tol: tolleranza desiderata
%imax: numero di iterazioni massime
%La funzione calcola l'approssimazione dell'autovalore dominante e il corrispettivo
  autovettore
%della matrice A per una certa tolleranza  (default 1e-6)
[m,n]=size(A);
if m~=n
    error('Dati Inconsistenti');
end
if nargin <= 2
    if nargin <= 1
        tol=1E-6;
    else
        if tol>=0.1 || tol<= 0
            error('Tolleranza non valida');
        end
    end
    imax = ceil(-log10(tol))*n;
end
rand(0);
x = rand(n,1);
l1=0;
for k = 1:imax
    x1 = x/(norm(x)^2);
    x = A*x1;
    l0 = l1;
    l1 = x'*x1;
    err = abs(l1-l0);
    if err <= tol*(1+abs(l1))
        break;
    end
end
 if err > tol*(1+abs(l1))
        warning('Convergenza non ottenuta');
 end
 return
```

25 Esercizio

Sia data la matrice di Toeplitz

$$-\begin{pmatrix} 4 & -1 & & & & & & & -1 & \\ -1 & 4 & -1 & & & & & & & -1 \\ & -1 & 4 & -1 & & & & & & \\ & & -1 & 4 & -1 & & & & & \\ & & & -1 & 4 & -1 & & & & \\ & & & & -1 & 4 & -1 & & & \\ & & & & & -1 & 4 & -1 & & \\ & & & & & & -1 & 4 & -1 & \\ -1 & & & & & & & -1 & 4 & -1 \\ & -1 & & & & & & & -1 & 4 \end{pmatrix}$$

simmetrica in cui le extra-diagonali più esterne sono le none. Partendo dal vettore $u_0 = (1, \ldots, 1)$ *applicare il metodo delle potenze con tolleranza* $tol = 10^{-10}$ *per* $N = 10 : 10 : 500$*, utilizzando la function del precedente esercizio. Graficare il valore dell'autovalore dominante, e del numero di iterazioni necessarie per soddisfare il criterio di arresto, rispetto ad* N *. Utilizzare la function spdiags di Matlab per creare la matrice e memorizzarla come matrice sparsa.*

```
%Script ad-hoc esercizio 25
tol = 10e-10;
i = 1;
res = zeros(2, 50);
for N = 10:10:500
    diag = -1*ones(N,5);
    diag(:,3) = diag(:,3)*(-4);
    A = spdiags(diag, [-8 -1 0 1 8] , N, N);
    [l1, x1, k] = potenze(A, tol);
    res(1,i) = l1;
    res(2,i) = k;
    i = i+1;
end
figure
semilogy(10:10:500, res(2, :));
figure
semilogy(10:10:500,res(1,:));
```

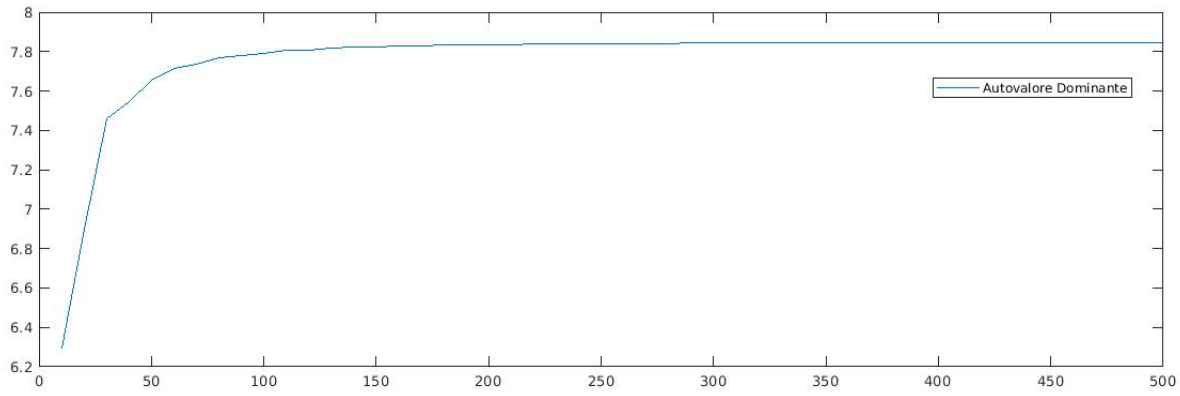


Figure 8: *Variazione autovalore dominante*

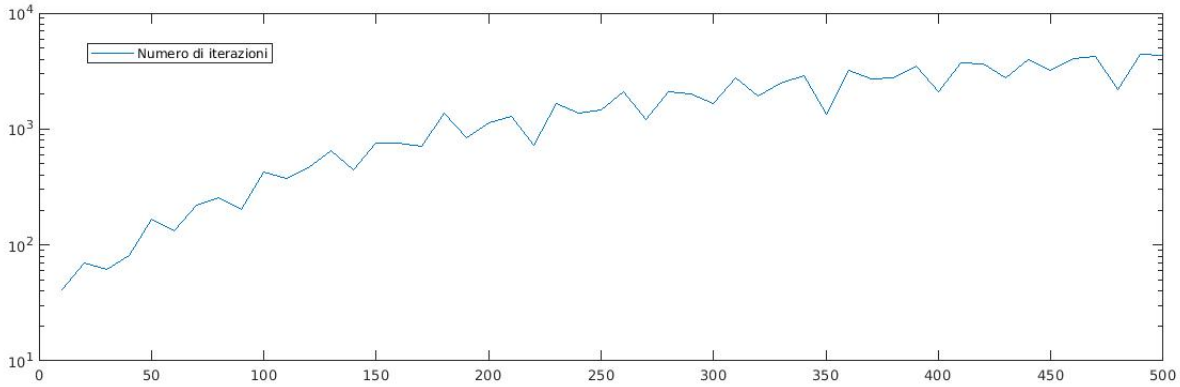


Figure 9: Numero iterazioni al variare di n

26 Esercizio

Scrivere una function che implementi efficientemente un metodo iterativo, per risolvere un sistema lineare, definito da un generico splitting della matrice dei coefficienti.

```
function x = splitting(b, A, nSolve, tol)
% x = splitting(b, matvec, nSolve, tol)
% b: vettore termini noti
% matvec: Costruisce la matrice
% nSolve: Funzione da passare (Jacobi o Seidel)
% tol: Tolleranza desiderata
% La funzione calcola il sistema lineare tramite lo splitting passato come
% funzione
n = length(b);
x = zeros(n,1);
imax = ceil(-log10(tol))*n;
tolb = tol*norm(b,inf);
for i = 1:imax
    r=A*x-b;
    nr = norm(r,inf);
    if nr < tolb
        fprintf("norma minore di tol");
        break;
    end
    u = nSolve(A,r);
    x = x-u;
end
if nr > tolb
    fprintf("Convergenza non raggiunta");
end
return
```

27 Esercizio

Scrivere le function ausiliarie, per la function del precedente esercizio, che implementano i metodi iterativi di Jacobi e Gauss-Seidel.

```
function b = jacobi(A,b)
% b = jacobi(A,b)
% b: vettore termini noti
% A: matrice
% Risolve il sistema diagonale per Jacobi
D = diag(A);
b=b./D;
return
```

```
function x = seidel(A,b)
%x = seidel(A,b)
%A: Matrice
%b: vettore termini noti
%Risolve il  sistema triangolare inferiore per Gauss-Seidel
n = length(b);
x = b;
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
return
```

28 Esercizio

Con riferimento alla matrice A_n definita nell'esercizio 25, risolvere il sistema lineare:

$$A_n x = \begin{pmatrix} 1 \\ . \\ . \\ . \\ 1 \end{pmatrix} \mathbb{R}^n$$

con i metodi di Jacobi e Gauss-Seidel, per $N = 10 : 10 : 500$, partendo dalla approssimazione nulla della soluzione, ed imponendo che la norma del residuo sia minore di 10^{-8} . Utilizzare, a tal fine, la function dell'esercizio 26, scrivendo function ausiliarie ad hoc (vedi esercizio 27) che sfruttino convenientemente la struttura di sparsità (nota) della matrice A_n . Graficare il numero delle iterazioni richieste dai due metodi iterativi, rispetto ad N , per soddisfare il criterio di arresto prefissato.

```
function [x,i]=splitting(b,matvec,msolve,tol)
%[x,i]=splitting(b,matvec,msolve,tol)
% b= vettore termini noti
%matvec= funzione per creare la matrice ad-hoc del esercizio 25
%msolve= funzione per risolvere il sistema lineare
%tol = tolleranza desiderata
%Funzione per la risoluzione ad-hoc di uno sistema lineare con splitting di
%Jacobi o Gauss-Seidel
n=length(b);
itmax= ceil(-log10(tol))*(n*n);
x=zeros(n,1);
tolb=10e-8;
for i=1:itmax
    r=matvec(x)-b;
    nr=norm(r);
    if nr<= tolb
        break;
    end
    u=msolve(r);
    x=x-u;
end
if nr>tolb
    fprintf('conv non raggiunta');end
return
```

```
function y= jacobi(x)
%y=jacobi(x)
%x: vettore termini noti
%Risolve un sistema diagonale per la matrice dell' esercizio 25
y=x/4;
return
```

```
function y=gaussSeidel(x)
%y=gaussnovec(x)
%x:vettore dei termini noti
%Risolve un sistema triangolare inferiore per la matrice dell' esercizio 25
n=length(x);
y=x;
y(1)=y(1)/4;
for i=2:n
    y(i)=(y(i)+y(i-1))/4;
end
for i=9:n
   y(i)=(y(i)+y(i-1))/4;
end
return
```

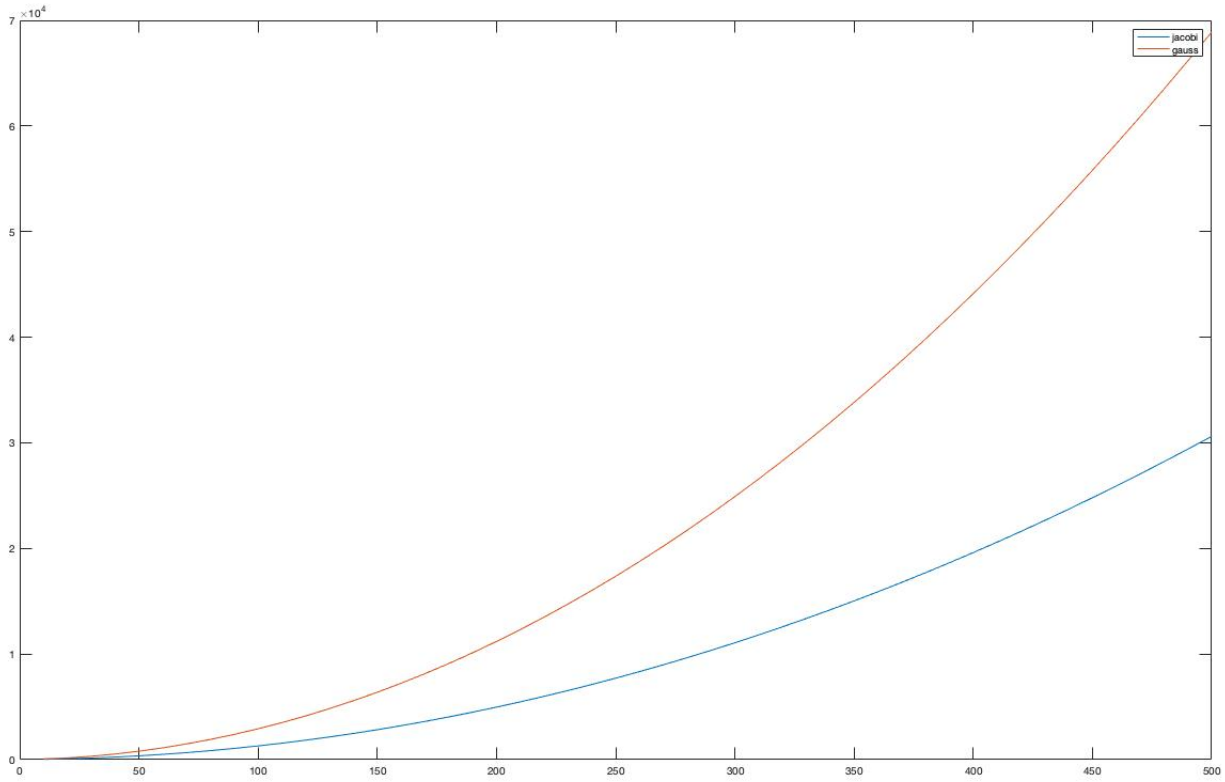


Figure 10: Numero iterazioni al variare della dimesione della matrice