

Scuola di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Elaborato di Calcolo Numerico

Candidato: Giulio Fagioli (matricola 6006222)

A.A. 2018/2019

**Esercizio 1**

## 

## Verificare che, per h sufficientemente piccolo*:*

Soluzione

## Iniziamo con lo sviluppo di utilizzando il polinomio di Taylor al secondo ordine:

## 

## 

## Possiamo applicarlo a ed uno per sostituendo rispettivamenteed .

## 

## 

## 

## 

## Utilizziamo questi due sviluppi avendo quindi:

## 

## 

## 

## 

## 

## 

## 

## Eseguendo alcune semplificazioni si arriva ad:

## 

## 

## 

## 

# **Esercizio 2**

## 

## Quanti sono i numeri di macchina normalizzati della doppia precisione IEEE? Argomentare la risposta.

Soluzione

## Nella rappresentazione IEEE754 abbiamo:

## 1 bit per il segno;

## 53 bit per la mantissa;

## 11 bit per l’esponente;

## Inoltre abbiamo:

## Shift v=;

## Base ;

## Mentre per ( parte frazionaria della mantissa ), assumiamo la mantissa nella forma:

## 1,nel caso di numeri normalizzati

## 0, nel caso di numeri denormalizzati

## Il numero di bit riservati all’esponente (*e*) determina l’intervallo dei numeri rappresentabili.

## Possiamo calcolare quanti sono i numeri di macchina normalizzati attraverso la relazione:

## 

## 

**Esercizio 3**

## 

## Eseguire il seguente script Matlab:

format long e

n=75;

u=1e-300;

for i=1:n,u=u\*2;end,for i=1:n,u=u/2;end,u

u=1e-300;

for i=1:n,u=u/2;end,for i=1:n,u=u\*2;end,u

## 

## Spiegare i risultati ottenuti.

Soluzione

## Nel primo caso abbiamo prima la moltiplicazione seguita dalla divisione, in questo

## contesto non si presentano errori poiché *u* non assume un valore minore del

## valore minimo rappresentabile da matlab ( 2.225073858507201e-308 ).

## Differente è il secondo caso in cui effettuando prima le divisioni si arriva ad una condizione di underflow e il numero viene approssimato.

## Matlab in questo caso usa un sistema di recovery ( *gradual underflow* ) che diminuisce i bit della mantissa a favore dell'esponente.

## L'errore generato dal primo ciclo di divisioni viene propagato dal successivo ciclo di moltiplicazioni.

## Tabella rappresentativa dei risultati del primo ciclo:

|  |  |  |
| --- | --- | --- |
| **Iterazione** | **Moltiplicazione** | **Divisione** |
| 1.00000000000000e-300 | 3.77789318629572e-278 | 1.00000000000000e-300 |

## Tabella rappresentativa dei risultati del secondo ciclo:

|  |  |  |
| --- | --- | --- |
| **Iterazione** | **Moltiplicazione** | **Divisione** |
| 1.00000000000000e-300 | 2.96439387504748e-323 | 1.11991634220386e-300 |

# **Esercizio 4**

## 

## Eseguire le seguenti istruzioni Matlab:

## format long e

## a=1.111111111111111

## b=1.11111111111111

## a+b

## a-b

Soluzione

## Possiamo studiare il condizionamento di queste somme algebriche come segue:

## 

## Identifichiamo due casi significativi:

## 

## In questo caso si hanno addendi di segno concorde e k=1, questa operazione è sempre ben condizionata.

## 

## In questo caso il numero di condizionamento potrebbe essere grande, un’operazione di somma algebrica fra due numeri quasi opposti è un’operazione mal condizionata che da luogo al fenomeno chiamato *cancellazione numerica*.

**Esercizio** **5**

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

1. Metodo di Bisezione
2. Metodo di Newton
3. Metodo delle Secanti
4. Metodo delle Corde

Detta xi l’approssimazione al passo i-esimo, utilizzare come criterio di arresto:

essendo *tol* una opportuna tolleranza specificata in ingresso.

Soluzione

## Di seguito si riportano i codici delle funzioni relative ad i quattro metodi per la ricerca degli zeri di una funzione:

function [iterazioni,x] = metodoBisezione(f, a, b, tolx)

% Utilizzo: [iterazioni,x] = metodoBisezione(f, a, b, tolx, itmax)

% Calcola dell'approssimazione di una radice della funzione

% METODO: Bisezione

%

% Parametri:

% f: Funzione utilizzata

% a: Estremo sinistro dell'intervallo di condifenza iniziale

% b: Estremo destro dell'intervallo di condifenza iniziale

% tolx: Tolleranza prefissata

% Restituisce:

% iterazioni: numero di iterazioni eseguite

% x: Radice approssimata

x = inf;

fa = feval(f,a);

fb = feval(f,b);

if fa\*fb > 0

error("L'intervallo non contiene alcuna radice");

end

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

for i = 2:imax

xi = (a+b)/2;

fx = feval(f,x);

if abs(xi-x) <= tolx\*(1+abs(xi))

break;

elseif fa\*fx < 0

b = xi;

fb = fx;

else

a = xi;

fa = fx;

end

x = xi;

end

iterazioni = i;

end

function [iterazioni,x] = metodoNewton(f, f\_der, x0, tolx, itmax)

% Utilizzo: [iterazioni,x] = metodoNewton(f, f\_der, x0, tolx, itmax)

% Calcola dell'approssimazione di una radice della funzione

% METODO: Newton

%

% Parametri:

% f: Funzione utilizzata

% f\_der: Derivata della funzione designata

% x0: Punto di innesco

% tolx: Tolleranza prefissata

% itmax: Numero di iterazioni massime

% Restituisce:

% iterazioni: numero di iterazioni eseguite ( -1 nel caso di una non

% convergenza)

% x: Radice approssimata

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x = x0 - fx/f\_derx;

i = 0;

while (i<itmax) && (abs(x-x0)>tolx)

i = i+1;

x0 = x;

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x = x0 - fx/f\_derx;

end

if abs(x-x0) <= tolx \* (1 + x0) % Convergenza ottenuta

iterazioni = i;

else % Convergenza non ottenuta

warning("Il metodo (Newton) non converge");

iterazioni = -1;

end

end

function [iterazioni,x] = metodoCorde(f, f\_der, x0, tolx, itmax)

% Utilizzo: [iterazioni,x] = metodoCorde(f, f\_der, x0, tolx, itmax)

% Calcola dell'approssimazione di una radice della funzione

% METODO: Corde

%

% Parametri:

% f: Funzione utilizzata

% f\_der: Derivata della funzione designata

% x0: Punto di innesco

% tolx: Tolleranza prefissata

% itmax: Numero di iterazioni massime

% Restituisce:

% iterazioni: numero di iterazioni eseguite ( -1 nel caso di una non

% convergenza)

% x: Radice approssimata

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x = x0 - fx/f\_derx;

i = 0;

while (i<itmax) && (abs(x-x0)>tolx\*(1+x0))

i = i+1;

x0 = x;

fx = feval(f, x0);

x = x0 - fx/f\_derx;

end

iterazioni = i;

if abs(x-x0)> tolx\*(1+x0) % Convergenza non ottenuta

warning("Il metodo (Corde) non converge");

iterazioni = -1;

end

end

function iterazioni = metodoSecanti(f, f\_der, x0, tolx, itmax)

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x = x0 - fx/f\_derx;

i = 0;

while (i<itmax) && (abs(x-x0)>tolx\*(1+x0))

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) <= tolx\*(1+x0) % Convergenza ottenuta

iterazioni = i;

else % Convergenza non ottenuta

warning("Il metodo (Secanti) non converge");

iterazioni = -1;

end

end

**Esercizio 6**

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

## 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.

Soluzione

## Risultati per l'approssimazione della radice:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tolx** | **Bisezione** | **Newton** | **Corde** | **Secanti** |
| 0.1 | 5 | 2 | 3 | 3 |
| 0.01 | 8 | 3 | 6 | 4 |
| 0.001 | 11 | 3 | 1 | 4 |
| 0.0001 | 15 | 4 | 15 | 5 |
| 1e-05 | 18 | 4 | 19 | 5 |
| 1e-06 | 21 | 4 | 23 | 6 |
| 1e-07 | 25 | 5 | 27 | 6 |
| 1e-08 | 28 | 5 | 31 | 6 |
| 1e-09 | 31 | 5 | 36 | 6 |
| 1e-10 | 35 | 5 | 40 | 7 |
| 1e-11 | 38 | 5 | 44 | 7 |
| 1e-12 | 41 | 6 | 48 | 7 |

## Analizziamo i rispettivi costi:

## *Metodo di Bisezione*:

## Il metodo di bisezione effettua due valutazioni di funzione iniziali e una valutazione di funzione per ogni iterazione del metodo. L’ordine di convergenza è lineare verso radici semplici.

## *Metodo di Newton*:

## Questo metodo effettua due valutazioni di funzione iniziali e altrettante due valutazioni ad ogni iterazione. L’ordine di convergenza è quadratico verso radici semplici, per funzioni sufficientemente regolari.

## *Metodo delle Secanti e delle Corde*:

## Questi due metodi, come il metodo di Bisezione, effettuano due valutazioni di funzione iniziali e una valutazione di funzione per ogni iterazione del metodo. L’ordine di convergenza è comunque lineare.

## Come possiamo notare dalla tabella il metodo di Newton impiega il minor numero di iterazioni per convergere alla migliore approssimazione; seguito dal metodo delle secanti.

## I metodi di Bisezione e delle Corde, invece, impiegano molte piú iterazioni rispetto ai precedenti.

# **Esercizio 7**

## Calcolare la molteplicità della radice nulla della funzione .

## Confrontare, quindi, i metodi di *Newton*, *Newton modificato*, e di *Aitken*, per approssimarla per gli stessi valori di del precedente esercizio (ed utilizzando il medesimo criterio di arresto), partendo da .

## Tabulare e commentare i risultati ottenuti.

Soluzione

## Il codice relativo ad i metodi di *Newton*, *Newton modificato* e *Aitken* è il seguente:

function [iterazioni,x] = newton(f, f\_der, x0, tolx, itmax)

% Utilizzo [iterazioni,x] = newton\_mod(f, f\_der, x0, tolx, itmax)

% Calcola dell'approssimazione di una radice della funzione

% Metodo: Newton

% Parametri:

% f: funzione utilizzata

% f\_der: derivata della funzione utilizzata f

% x0: punto di innesco

% molteplicità: molteplicità nota della radice

% tolx: tolleranza prefissata

% itmax: numero massimo di iterazioni

% Restituisce:

% i: numero di iterazioni eseguite (-1 nel caso non converga)

% x: radice approssimata

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x = x0 - fx/f\_derx;

i = 0;

while (i<itmax) && (abs(x-x0)>tolx)

i = i+1;

x0 = x;

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x = x0 - fx/f\_derx;

end

if abs(x-x0)<=tolx

iterazioni = i;

else

% Non abbiamo raggiunto la convergenza entro itmax iterazioni

iterazioni = -1;

end

end

function [iterazioni,x] = newton\_mod(f, f\_der, x0, molteplicita, tolx, itmax)

% Utilizzo [iterazioni,x] = newton\_mod(f, f\_der, x0, tolx, itmax)

% Calcola dell'approssimazione di una radice della funzione

% Metodo: Newton Modificato

% Parametri:

% f: funzione utilizzata

% f\_der: derivata della funzione utilizzata f

% x0: punto di innesco

% molteplicità: molteplicità nota della radice

% tolx: tolleranza prefissata

% itmax: numero massimo di iterazioni

% Restituisce:

% i: numero di iterazioni eseguite (-1 nel caso non converga)

% x: radice approssimata

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x = x0 - fx/f\_derx;

i = 0;

while (i<itmax) && (abs(x-x0)>tolx)

i = i+1;

x0 = x;

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x = x0 - molteplicita\*fx/f\_derx;

end

if abs(x-x0)>tolx

% Non abbiamo raggiunto la convergenza entro itmax iterazioni

i = -1;

end

iterazioni = i;

end

function [iterazioni,x] = aitken(f, f\_der, x0, tolx, itmax)

% Utilizzo [iterazioni,x] = aitken(f, f\_der, x0, tolx, itmax)

% Calcola dell'approssimazione di una radice della funzione

% Metodo: Aitken

% Parametri:

% f: funzione utilizzata

% f\_der: derivata della funzione utilizzata f

% x0: punto di innesco

% molteplicità: molteplicità nota della radice

% tolx: tolleranza prefissata

% itmax: numero massimo di iterazioni

% Restituisce:

% i: numero di iterazioni eseguite (-1 nel caso non converga)

% x: radice approssimata

i = 0;

x = x0;

vai = 1;

while (i<itmax) && vai

i = i + 1;

x0 = x;

fx = feval(f, x0);

f\_derx = feval(f\_der, x0);

x1 = x0 - fx/f\_derx;

fx = feval(f, x1);

f\_derx = feval(f\_der, x1);

x = x1 - fx/f\_derx;

x = (x\*x0-x1^2)/(x-2\*x1+x0);

vai = abs(x-x0)>tolx;

end

if ~vai

iterazioni = i;

else

% Non abbiamo raggiunto la convergenza entro itmax iterazioni

iterazioni = -1;

end

end

## 

## Confrontiamo risultati ottenuti utilizzando i vari metodi:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tolleranza** | **It Newton** | **x Newton** | **It Newton Mod** | **x Newton Mod** |
| 1.0000e-01 | 3.0000e+00 | 2.8789e-01 | 2.0000e+00 | 3.2423e-09 |
| 1.0000e-02 | 1.1000e+01 | 2.8805e-02 | 3.0000e+00 | 0.0000e+00 |
| 1.0000e-03 | 1.9000e+01 | 2.8838e-03 | 3.0000e+00 | 0.0000e+00 |
| 1.0000e-04 | 2.7000e+01 | 2.8870e-04 | 3.0000e+00 | 0.0000e+00 |
| 1.0000e-05 | 3.5000e+01 | 2.8903e-05 | 3.0000e+00 | 0.0000e+00 |
| 1.0000e-06 | 4.3000e+01 | 2.8935e-06 | 3.0000e+00 | 0.0000e+00 |
| 1.0000e-07 | 5.1000e+01 | 2.8968e-07 | 3.0000e+00 | 0.0000e+00 |
| 1.0000e-08 | 5.9000e+01 | 2.9001e-08 | 3.0000e+00 | 0.0000e+00 |
| 1.0000e-9 | 6.7000e+01 | 2.9034e-09 | 4.0000e+00 | NaN |
| 1.0000e-10 | 7.5000e+01 | 2.9066e-10 | 4.0000e+00 | NaN |
| 1.0000e-11 | 8.3000e+01 | 2.9099e-11 | 4.0000e+00 | NaN |
| 1.0000e-12 | 9.1000e+01 | 2.9132e-12 | 4.0000e+00 | NaN |

|  |  |  |
| --- | --- | --- |
| **Tolleranza** | **It Aitken** | **x Aitken** |
| 1.0000e-01 | 3.0000e+00 | 6.4929e-19 |
| 1.0000e-02 | 3.0000e+00 | 6.4929e-19 |
| 1.0000e-03 | 3.0000e+00 | 6.4929e-19 |
| 1.0000e-04 | 4.0000e+00 | 0.0000e+00 |
| 1.0000e-05 | 4.0000e+00 | 0.0000e+00 |
| 1.0000e-06 | 4.0000e+00 | 0.0000e+00 |
| 1.0000e-07 | 4.0000e+00 | 0.0000e+00 |
| 1.0000e-08 | 4.0000e+00 | 0.0000e+00 |
| 1.0000e-9 | 4.0000e+00 | 0.0000e+00 |
| 1.0000e-10 | 4.0000e+00 | 0.0000e+00 |
| 1.0000e-11 | 4.0000e+00 | 0.0000e+00 |
| 1.0000e-12 | 4.0000e+00 | 0.0000e+00 |

## Siamo nel caso di radici multiple poiché m=4, il problema dunque, si presenta mal condizionato. In questo caso il metodo di Newton ha una convergenza soltanto lineare.

## Utilizzando invece il metodo di Newton Modificato, con la molteplicità nota viene ripristinata la convergenza quadratica e il metodo di Newton Modificato converge verso la soluzione corretta con un solo passaggio.

## Per ultimo utilizziamo il metodo di accelerazione di Aitken, con il quale si ripristina la convergenza quadratica per il metodo di Newton con molteplicità non nota e radici multiple.

# **Esercizio 8**

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.

Soluzione

## Il codice della funzione palu(A) è il seguente:

function [LU, p] = palu(A)

% Utilizzo: [LU, p] = palu(A)

% Funzione che calcola la fattorizzazione LU della matrice A con

% pivoting parziale di A

% Parametri:

% A: la matrice da fattorizzare.

% Restituisce:

% LU: la matrice fattorizzata LU;

% p: vettore di permutazione

n=size(A,1);

p=(1:n);

for i=1:n-1

[mi, ki] = max(abs(A(i:n, i)));

if mi==0 % Controllo il caso in cui la matrice sia singolare

error('Errore: Matrice singolare');

end

ki = ki+i-1;

if ki>i

A([i ki], :) = A([ki i], :);

p([i ki]) = p([ki i]);

end

A(i+1:n, i) = A(i+1:n, i)/A(i, i);

A(i+1:n, i+1:n) = A(i+1:n, i+1:n) -A(i+1:n, i)\*A(i, i+1:n);

end

LU = A;

end

function [b]= sistemaLUpivot(A,b,p)

% [b]= sistemaLUpivot(A,b,p)

% Calcola la soluzione di Ax=b con A matrice LU con pivoting

% Parametri:

% A: Matrice matrice LU con pivoting (generata da palu)

% b: vettore colonna

% Restituisce:

% b: soluzione del sistema

P = zeros(length(A));

for i = 1:length(A)

P(i, p(i)) = 1;

end

b = tri\_inf\_unit(tril(A,-1)+eye(length(A)), P\*b);

b = triangSup(triu(A), b);

end

**Esercizio 9**

## 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)

Soluzione

## Il codice della funzione lusolve(LU,p,b) è il seguente:

function b = lusolve(LU,p,b)

% x = lusolve(A,b,p)

% Calcola la soluzione di Ax=b con A matrice LU con la tecnica del pivoting

% Parametri:

% A: Matrice matrice LU con pivoting (generata dalla funzione palu)

% b: vettore colonna

% Restituisce:

% b: soluzione del sistema

P = zeros(length(LU));

for i = 1:length(LU)

P(i, p(i)) = 1;

end

b = triangInfUnit(tril(LU,-1)+eye(length(LU)), P\*b);

b = triangSup(triu(LU), b);

end

function b = triangInfUnit(A, b)

% Utilizzo: b = tri\_inf\_unit(A, b)

% Calcola la soluzione del sistema Ax=b con A matrice triangolare inferiore a

% diagonale unitaria.

% Parametri:

% A: Matrice triangolare inferiore

% b: vettore dei termini noti

% Restituisce:

% b: soluzione del sistema

[n,m] = size(A);

if n~=m

error('Matrice non quadrata.')

end

if ~istril(A)

error('Matrice non triangolare inferiore')

end

if ~all(diag(A)==1)

error('Matrice non a diagonale unitaria')

end

for i=1:n

for j=1:i-1

b(i) = b(i) - A(i,j)\*b(j);

end

if A(i,i)==0 % Controllo se la matrice è singolare

error('Matrice singolare')

else

b(i) = b(i)/A(i,i);

end

end

end

function [b] = triangSup(A, b)

% Utilizzo: [b] = diagonale(A, b)

% Calcola la soluzione di Ax=b con A matrice triangolare superiore

% Parametri:

% A: Matrice triangolare superiore

% b: vettore colonna

% Restituisce:

% b: soluzione del sistema

for i=length(A):-1:1

for j=i+1:length(A)

b(i)=b(i)-A(i,j)\*b(j);

end

if A(i,i)==0 % Controllo se la matrice è singolare

error('Matrice singolare')

else

b(i)=b(i)/A(i,i);

end

end

end

# **Esercizio 10**

## Scaricare la function *cremat* al sito:

## http://web.math.unifi.it/users/brugnano/appoggio/cremat.m

## che crea sistemi lineari n × n la cui soluzione è il vettore x =(1 . . . n )T .

## Eseguire, quindi, lo script Matlab:

n = 10;

x = zeros(n,15);

for i = 1:15

[A,b] = cremat(n,i);

[LU,p] = palu(A);

x(:,i) = lusolve(LU,p,b);

end

## Confrontare i risultati ottenuti con quelli attesi, e dare una spiegazione esauriente degli stessi

Soluzione

## Si riporta le tabelle con i risultati del codice precedente:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Iterazione 1** | **Iterazione 2** | **Iterazione 3** | **Iterazione 4** | **Iterazione 5** |
| 1.0000e+00 | 1.0000e+00 | 1.0000e+00 | 1.0000e+00 | 1.0000e+00 |
| 2.0000e+00 | 2.0000e+00 | 2.0000e+00 | 2.0000e+00 | 2.0000e+00 |
| 3.0000e+00 | 3.0000e+00 | 3.0000e+00 | 3.0000e+00 | 3.0000e+00 |
| 4.0000e+00 | 4.0000e+00 | 4.0000e+00 | 4.0000e+00 | 4.0000e+00 |
| 5.0000e+00 | 5.0000e+00 | 5.0000e+00 | 5.0000e+00 | 5.0000e+00 |
| 6.0000e+00 | 6.0000e+00 | 6.0000e+00 | 6.0000e+00 | 6.0000e+00 |
| 7.0000e+00 | 7.0000e+00 | 7.0000e+00 | 7.0000e+00 | 7.0000e+00 |
| 8.0000e+00 | 8.0000e+00 | 8.0000e+00 | 8.0000e+00 | 8.0000e+00 |
| 9.0000e+00 | 9.0000e+00 | 9.0000e+00 | 9.0000e+00 | 9.0000e+00 |
| 1.0000e+01 | 1.0000e+01 | 1.0000e+01 | 1.0000e+01 | 1.0000e+01 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Iterazione 6** | **Iterazione 7** | **Iterazione 8** | **Iterazione 9** | **Iterazione 10** |
| 1.0000e+00 | 1.0000e+00 | 1.0000e+00 | 1.0000e+00 | 1.0000e+00 |
| 2.0000e+00 | 2.0000e+00 | 2.0000e+00 | 2.0000e+00 | 2.0000e+00 |
| 3.0000e+00 | 3.0000e+00 | 3.0000e+00 | 3.0000e+00 | 3.0000e+00 |
| 4.0000e+00 | 4.0000e+00 | 4.0000e+00 | 4.0000e+00 | 4.0000e+00 |
| 5.0000e+00 | 5.0000e+00 | 5.0000e+00 | 5.0000e+00 | 5.0000e+00 |
| 6.0000e+00 | 6.0000e+00 | 6.0000e+00 | 6.0000e+00 | 6.0000e+00 |
| 7.0000e+00 | 7.0000e+00 | 7.0000e+00 | 7.0000e+00 | 7.0000e+00 |
| 8.0000e+00 | 8.0000e+00 | 8.0000e+00 | 8.0000e+00 | 8.0000e+00 |
| 9.0000e+00 | 9.0000e+00 | 9.0000e+00 | 9.0000e+00 | 9.0000e+00 |
| 1.0000e+01 | 1.0000e+01 | 1.0000e+01 | 1.0000e+01 | 1.0000e+01 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Iterazione 11** | **Iterazione 12** | **Iterazione 13** | **Iterazione 14** | **Iterazione 15** |
| 1.0000e+00 | 9.9999e-01 | 9.9928e-01 | 1.0000e+00 | 9.8668e-01 |
| 2.0000e+00 | 2.0000e+00 | 2.0021e+00 | 2.0000e+00 | 2.0388e+00 |
| 3.0000e+00 | 3.0000e+00 | 2.9993e+00 | 3.0000e+00 | 2.9876e+00 |
| 4.0000e+00 | 4.0000e+00 | 4.0022e+00 | 4.0000e+00 | 4.0411e+00 |
| 5.0000e+00 | 4.9999e+00 | 4.9944e+00 | 5.0000e+00 | 4.8964e+00 |
| 6.0000e+00 | 6.0000e+00 | 6.0028e+00 | 6.0000e+00 | 6.0513e+00 |
| 7.0000e+00 | 7.0000e+00 | 7.0021e+00 | 7.0000e+00 | 7.0394e+00 |
| 8.0000e+00 | 8.0000e+00 | 8.0002e+00 | 8.0000e+00 | 8.0037e+00 |
| 9.0000e+00 | 9.0000e+00 | 9.0019e+00 | 9.0000e+00 | 9.0357e+00 |
| 1.0000e+01 | 1.0000e+01 | 1.0001e+01 | 1.0000e+01 | 1.0014e+01 |

## Nella colonna 13 e 15 si notano delle perturbazioni sui dati in uscita, queste perturbazioni sono dovute al condizionamento nelle matrici.

## 

## 

## 

## In dettaglio:

## 

## è l’errore relativo nei dati in uscita

## 

## e sono gli errori relativi sui dati di ingresso

## 

## indicabile con k(A) è il numero di condizionamento della matrice

## 

## Quando k(A) è piccolo si parla di problema ben condizionato, differente è il caso in cui k(A) >> 1, in questo caso invece si parla di problema mal condizionato.

## Possiamo notare questo mal condizionamento nelle colonne 13 e 15.

**Esercizio 11**

Scrivere una function Matlab che, data in ingresso una matrice A ∈ R m×n , con m ≥ 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)

Soluzione

## Si riporta il codice della funzione myqr(A) di seguito:

function QR = myqr(A)

% Utilizzo: [A] = myqr(A)

% Calcola la fattorizzazione QR della matrice A.

% Parametri:

% A: la matrice da fattorizzare

% Restituisce:

% QR: una matrice QR scritta con la parte significativa di R e la parte

% significativa dei vettori di Householder normalizzati con la prima

% componente unitaria

[m,n]=size(A); % Inizializzo m ed n con la grandezza di A

for i=1:n

alpha = norm(A(i:m, i), 2); % Rank della matrice

if alpha==0

error('La matrice non ha rank massimo');

end

if(A(i,i))>=0

alpha = -alpha;

end

v1 = A(i,i)-alpha;

A(i,i) = alpha;

A(i+1:m, i) = A(i+1:m, i)/v1;

beta = -v1/alpha;

A(i:m, i+1:n) = A(i:m, i+1:n) -(beta\*[1; A(i+1:m, i)])\*([1 A(i+1:m, i)']\*...

A(i:m,i+1:n));

end

QR = A;

end

# **Esercizio 12**

## Scrivere una function Matlab che, data in ingresso la matrice QR creata dalla function del precedente esercizio, ed il termine noto del sistema lineare Ax = b, ne calcoli la soluzione nel senso dei minimi quadrati:

## function x = qrsolve(QR,b)

Soluzione

## Viene mostrato il codice della funzione qrsolve(QR,b) di seguito:

function x = qrsolve(QR,b)

% x = qrsolve(QR,b)

% Calcola la soluzione di Ax=b con A matrice QR

% Parametri:

% A: Matrice QR

% b: vettore colonna

% Restituisce:

% b: soluzione del sistema lineare sovradeterminato

[m,n] = size(QR);

qtrasp=eye(m);

for i=1:n

qtrasp = [eye(i-1) zeros(i-1, m-i+1); zeros(i-1, m-i+1)' ...

(eye(m-i+1)-(2/norm([1; QR(i+1:m, i)], 2)^2)\*([1; QR(i+1:m, i)]...

\*[1 QR(i+1:m, i)']))]\*qtrasp;

end

x = triangSup(triu(QR(1:n, :)), qtrasp(1:n,:)\*b);

end

# **Esercizio 13**

Scaricare la function *cremat1* al sito: *http://web.math.unifi.it/users/brugnano/appoggio/cremat1.m* che crea sistemi lineari m × n, con m ≥ n, la cui soluzione (nel senso dei minimi quadrati) è il vettore x = (1 . . . n)^T

Eseguire, quindi, il seguente script Matlab per testare le function dei precedenti esercizi.

Soluzione

## Viene riportata la tabella con i risultati:

|  |  |  |  |
| --- | --- | --- | --- |
| **m** | **n** | **norma** | **norma(x-xx)** |
| 1 | 5 | 2.24748192067106e-14 | 5 |
| 2 | 5 | 1.75320351399483e-14 | 6 |
| 3 | 5 | 1.61223344547346e-14 | 7 |
| 4 | 5 | 5.37402993480137e-14 | 8 |
| 5 | 5 | 5.37054250957417e-15 | 9 |
| 6 | 5 | 8.62551123839374e-15 | 10 |
| 7 | 5 | 1.23034552927593e-14 | 11 |
| 8 | 5 | 4.37377147912526e-15 | 12 |
| 9 | 5 | 5.86740001550182e-15 | 13 |
| 10 | 5 | 6.14646579662655e-15 | 14 |
| 11 | 5 | 6.71294862268067e-15 | 15 |
| 12 | 6 | 1.54193791514084e-13 | 6 |
| 13 | 6 | 1.87233114247782e-14 | 7 |
| 14 | 6 | 6.49661685660984e-14 | 8 |
| 15 | 6 | 3.3083901009315e-15 | 9 |
| 16 | 6 | 9.03587868944547e-15 | 10 |
| 17 | 6 | 1.72567375746672e-14 | 11 |
| 18 | 6 | 1.04242788572733e-14 | 12 |
| 19 | 6 | 6.61771155244849e-15 | 13 |

|  |  |  |  |
| --- | --- | --- | --- |
| **m** | **n** | **norma** | **norma(x-xx)** |
| 20 | 6 | 1.11249665373964e-14 | 14 |
| 21 | 6 | 2.04293103816182e-14 | 15 |
| 22 | 6 | 8.25157905384712e-15 | 16 |
| 23 | 7 | 4.96224983026092e-14 | 7 |
| 24 | 7 | 1.06572158103885e-13 | 8 |
| 25 | 7 | 1.92872294697368e-14 | 9 |
| 26 | 7 | 1.96719490091217e-14 | 10 |
| 27 | 7 | 2.35241562019244e-14 | 11 |
| 28 | 7 | 2.94207006757785e-14 | 12 |
| 29 | 7 | 8.59114651514807e-15 | 13 |
| 30 | 7 | 1.65482829223001e-14 | 14 |
| 31 | 7 | 2.79714516348279e-14 | 15 |
| 32 | 7 | 1.0981671588864e-14 | 16 |
| 33 | 7 | 1.48570906318214e-14 | 17 |
| 34 | 8 | 1.26952825645372e-13 | 8 |
| 35 | 8 | 3.18082287270531e-14 | 9 |
| 36 | 8 | 4.76175677732291e-14 | 10 |
| 37 | 8 | 2.96080057903009e-14 | 11 |
| 38 | 8 | 3.43690692906712e-14 | 12 |

|  |  |  |  |
| --- | --- | --- | --- |
| **m** | **n** | **norma** | **norma(x-xx)** |
| 39 | 8 | 1.20032515914493e-14 | 13 |
| 40 | 8 | 1.3427733265506e-14 | 14 |
| 41 | 8 | 3.68787947797593e-14 | 15 |
| 42 | 8 | 1.83438948940332e-14 | 16 |
| 43 | 8 | 2.59785070850544e-14 | 17 |
| 44 | 8 | 2.1302689071571e-14 | 18 |
| 45 | 9 | 3.54361206919172e-14 | 9 |
| 46 | 9 | 6.76766312379184e-14 | 10 |
| 47 | 9 | 5.3532555080452e-14 | 11 |
| 48 | 9 | 1.02492287656922e-13 | 12 |
| 49 | 9 | 1.53580405866465e-14 | 13 |
| 50 | 9 | 5.34349545934035e-14 | 14 |
| 51 | 9 | 4.16272937617989e-14 | 15 |
| 52 | 9 | 2.76550293366644e-14 | 16 |
| 53 | 9 | 3.48348286554738e-14 | 17 |
| 54 | 9 | 2.89938308624056e-14 | 18 |
| 55 | 9 | 3.14224498798056e-14 | 19 |
| 56 | 10 | 9.68300345595875e-14 | 10 |
| 57 | 10 | 8.36283185233534e-14 | 11 |

|  |  |  |  |
| --- | --- | --- | --- |
| **m** | **n** | **norma** | **norma(x-xx)** |
| 58 | 10 | 6.53475423806255e-14 | 12 |
| 59 | 10 | 3.04761781435309e-14 | 13 |
| 60 | 10 | 8.78887916759733e-14 | 14 |
| 61 | 10 | 4.67845400655906e-14 | 15 |
| 62 | 10 | 2.23637611753954e-14 | 16 |
| 63 | 10 | 3.92567074798158e-14 | 17 |
| 64 | 10 | 2.89836261085267e-14 | 18 |
| 65 | 10 | 2.6052182275762e-14 | 19 |
| 66 | 10 | 3.70929151223012e-14 | 20 |

**Esercizio 14**

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

Soluzione

## Si riporta in seguito il codice per il calcolo del polinomio interpolante di Lagrange:

function y = lagrange(xi, fi, x)

% y = lagrange(xi, fi, x)

% Calcolo del polinomio interpolante per le coppie di dati (xi,fi) nei

% punti del vettore x, con il metodo di lagrange.

% Metodo utilizzato: Lagrange.

%

% Parametri:

% xi: ascisse di interpolazione

% fi: valori della funzione nelle ascisse di interpolazione

% x: ascisse dove valutare il polinomio, Metodo: lagrange

% Restituisce:

% y: valutazione delle ascisse in x

%

if length(xi) - length(fi) ~= 0

error("La lunghezza dei vettori xi e fi non é la stessa.")

end

if isempty(x)

error("Vettore x vuoto.")

end

n = length(xi)-1;

for i=1:n % Controllo che tutte le ascisse siano distinte

for j=i+1:n

if xi(i)==xi(j)

error("Le ascisse non sono tutte distinte");

end

end

end

y = zeros(size(x));

% per ogni ascissa in x

for i=1:length(x)

% calcolo del valore del polinomio

for k=1:n+1

lkn = 1;

% calcolo di lkn

for j=1:n+1

if (j~=k)

lkn = lkn.\*((x(i)-xi(j))/(xi(k)-xi(j)));

end

end

y(i) = y(i) + fi(k)\*lkn;

end

end

return

end

**Esercizio 15**

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

Soluzione

## 

## Si riporta in seguito il codice per il calcolo del polinomio interpolante di Lagrange:

function y = hermite(xi, fi, fi1, x)

% y = hermite(xi, fi, fi1, x)

% Funzione che calcola il polinomio interpolante per una data coppia di dati

% (xi,fi) nei punti del vettore x, utilizzando il metodo di Hermite

%

% Parametri:

% xi: ascisse di interpolazione

% fi: valori della funzione e la sua derivata nelle ascisse di interpolazione

% x: ascisse dove valutare il polinomio, con il metodo di hermite

% Restituisce:

% y: valutazione delle ascisse in x

%

if (length(xi) - length(fi) ~= 0) | (length(xi) - length(fi1) ~= 0)

error("La lunghezza dei vettori xi, fi e fi1 deve essere la medesima")

end

if isempty(x)

error("Il vettore x non deve essere vuoto")

end

for u=1:length(x)-1

if(x(u) == x(u+1))

fprintf("Ascisse uguali in posizione %d - %d \n",x(u),x(u+1));

error("Le ascisse devono essere distinte");

end

end

xi = reshape([xi; xi], [], 1)';

fi = reshape([fi; fi1], [], 1)';

n = length(xi)-1;

% Calcolo differenze divise

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

% Algoritmo di Horner

y = fi(n+1)\*ones(size(x));

for k=1:length(x)

for i=n:-1:1

y(k) = y(k)\*(x(k)-xi(i))+fi(i);

end

end

return

end

**Esercizio 16**

## Scrivere un programma che implementi efficientemente il calcolo di una spline cubica naturale interpolante su una partizione assegnata.

Soluzione

## Si riporta in seguito il codice per il calcolo di una spline cubica naturale:

function y = splineCubicaNaturale(xi, fi, x)

% Utilizzo: splineNaturale = naturalSpline(xi, fi, x)

% Funzione che calcola e valuta i valori della spline cubica naturale

% relativa alle coppie di dati assegnati.

%

% Parametri:

% xi: punti di interpolazione

% fi: valori della funzione, valutati nelle ascisse xi

% x: punti di valutazione della spline

%

% Restituisce:

% splineNaturale: valori di interpolazione della spline cubica naturale

%

n = length(xi);

splineNaturale = zeros(n, 1)';

widthI = xi(2 : n) - xi(1 : n - 1);

subDiag = (widthI(1 : end - 1)) ./ (widthI(1 : end - 1) + widthI(2 : end));

superDiag = (widthI(2 : end)) ./ (widthI(1 : end - 1) + widthI(2 : end));

divdiff = (fi(2 : n) - fi(1 : n - 1)) ./ widthI;

divdiff = 6 \* ((divdiff(2 : end) - divdiff(1 : end - 1)) ./ (xi(3 : end) - xi(1 : end - 2)));

m = sistemaSplineNaturale(subDiag, superDiag, divdiff);

[primaConst, secondaConst] = costantiIntegrazione(m, xi,fi, widthI);

k=2;

for j = 1 : length(x)

for i = 2 : length(xi)

if x(j) <= xi(i)

k = i;

break;

end

end

y(j) = (((x(j) - xi(k - 1)) ^ 3) \* m(k) + ...

((xi(k) - x(j)) ^ 3) \* m(k - 1)) / ...

(6 \* widthI(k - 1)) + secondaConst(k - 1) \* ...

(x(j) - xi(k - 1)) + primaConst(k - 1);

end

return

end

function m = sistemaSplineNaturale(subDiag, superDiag, divDiff)

% Utilizzo: m = sistemaSplineNaturale(subDiagCoeff, superDiagCoeff, divdiff)

% Funzione che ritorna i coefficienti necessari per il calcolo della spline

% cubica

%

% Parametri:

% subDiag: coeffiencenti della sotto diagonale della matrice

% superDiag: coeffiencenti della sopra diagonale della matri

% diffDiv: differenze divise

%

% Restituisce:

% m: coefficienti necessari a calcolare l'espressione della spline

% cubica

%

n = length(superDiag) + 1;

u(1) = 2;

l = zeros(1, n - 2);

m = zeros(1, n - 1);

for i = 2 : n - 1

l(i) = subDiag(i) / u(i - 1);

u(i) = 2 - l(i) \* superDiag(i - 1);

end

f(1) = divDiff(1);

for i = 2:n - 1

f(i) = divDiff(i) - l(i) \* f(i - 1);

end

m(n - 1) = f(n - 1) / u(n - 1);

for j = n - 2 : - 1 : 1

m(j) = (f(j) - superDiag(j + 1) \* m(j + 1)) / u(j);

end

m = [0 m 0];

return

end

function [primaConst, secondaConst] = costantiIntegrazione(m, xi, fi, valI)

% Utilizzo [ri, qi] = costantiIntegrazione(m, fi, xi, widthI)

% Funzione che ritorna le costanti di integrazione della spline.

%

% Parametri:

% m: fattore m calcolato

% fi: valori della funzione, valutati nelle ascisse xi

% xi: ascisse di interpolazione

% valI: valore dell'i-esimo intervallo

%

% Restituisce:

% primaConst: valore della costante prima integrazione

% secondaConst: valore della costane seconda integrazione

n = length(xi);

primaConst = zeros(1, n-1);

secondaConst = primaConst;

for i = 2 : n

primaConst(i - 1) = fi(i - 1) - (valI(i - 1) ^ 2) / 6 \* m(i - 1);

secondaConst(i - 1) = (fi(i) - fi(i - 1)) / ...

valI(i - 1) - valI(i - 1) / 6 \* (m(i) - m(i - 1));

end

return

end

**Esercizio 17**

## 

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

(Facoltativo)

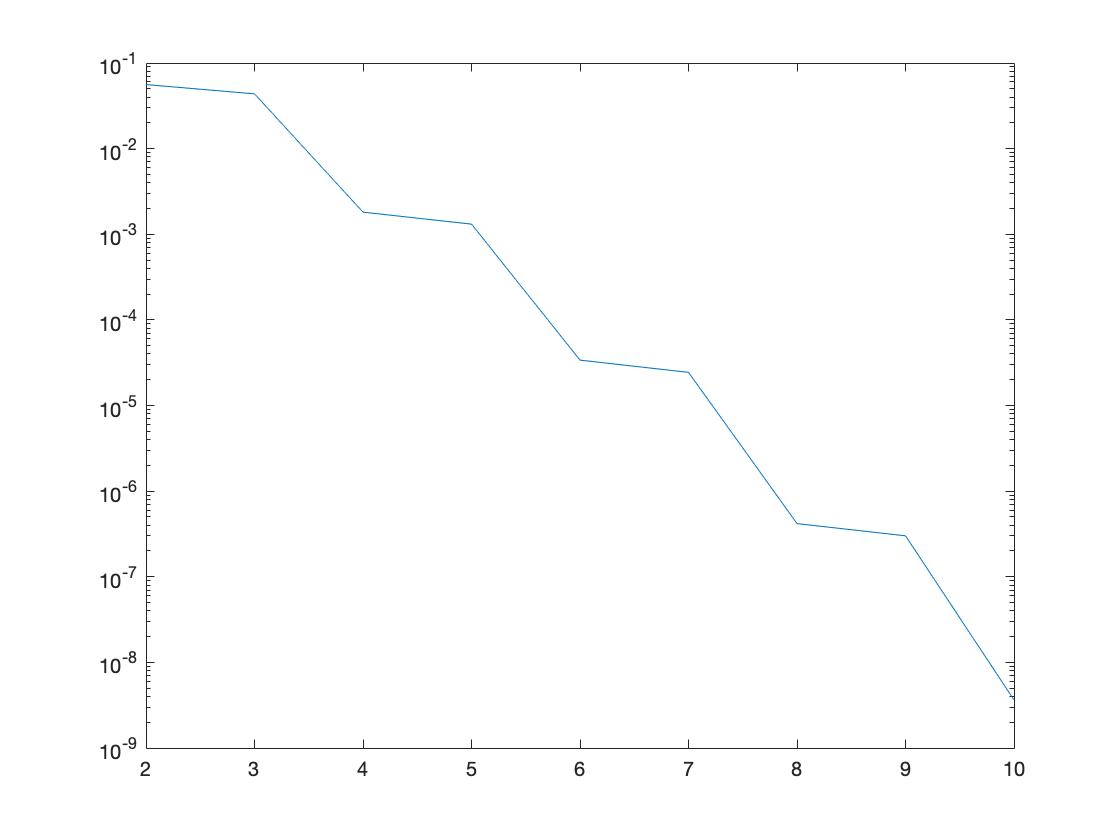
**Esercizio 18**

## 

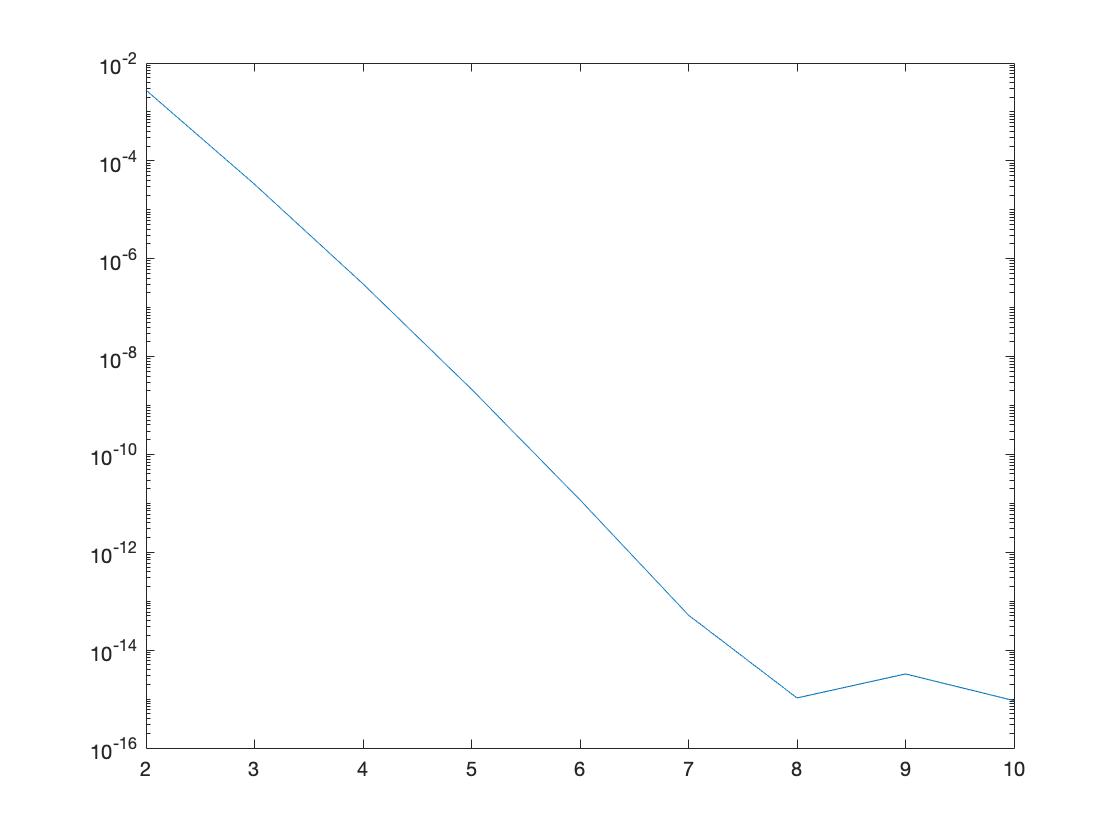
## Confrontare i codici degli esercizi 14–17 per approssimare la funzione f(x) = sin(x) sulle ascisse xi = iπ/n, i = 0, 1, . . . , n, per n = 1, 2, . . . , 10. Graficare l’errore massimo di approssimazione verso n (in semilogy), calcolato su una griglia uniforme di 10001 punti nell’intervallo [0, π].

Soluzione

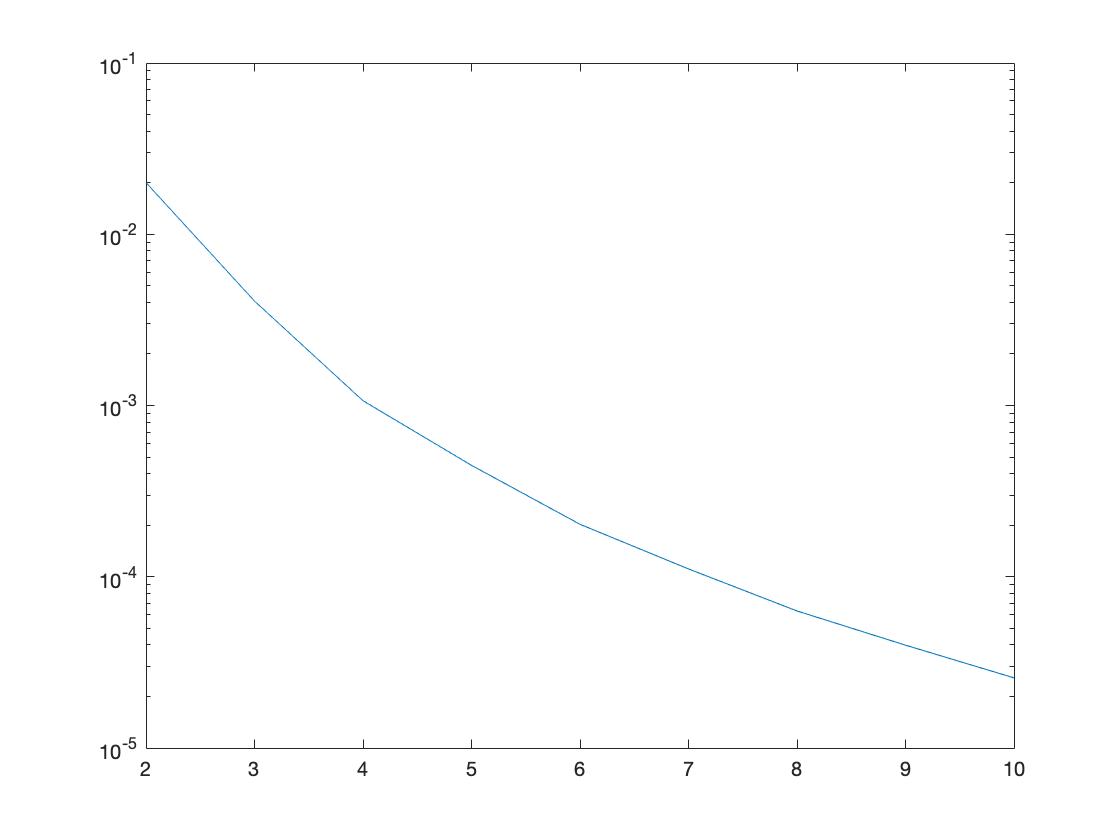
Per ottenere i seguenti grafici ho utilizzato il codice degli esercizi 14-16 riportati precedentemente. Sull’asse Y abbiamo i valori dell’errore massimo, mentre sull’asse X abbiamo i valori di *n.* Si mostrano in seguito i risultati ottenuti:



Valori dell’errore massimo ottenuto valutando la funzione *𝑓* (*𝑥*) utilizzando il Metodo di Lagrange.



Valori dell’errore massimo ottenuto valutando la funzione *𝑓* (*𝑥*) utilizzando il Metodo di Hermite.



Valori dell’errore massimo ottenuto valutando la funzione *𝑓* (*𝑥*) utilizzando la Spline Cubica Naturale.

**Esercizio 19**

## Calcolare (numericamente) la costante di Lebesgue per i polinomi interpolanti di grado n = 2, 4, 6, . . . , 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

Soluzione

## Si mostrano in seguito la funzione per il calcolo della costante di Lebesgue:

function lebEqui = lebesgueEquidistanti(f,z,lebesgueConstanti)

% Utilizzo: lebesgueEquidistanti(f,z,lebesgueConstanti)

% Funzione che calcola la costante di lebesgue per polinomi

% interpolanti di grado n:2...40 su ascisse equidistanti

%

% Parametri:

% f: Funzione da approssimare

% z: punti di valutazione

% lebesgueConstanti: vettore per memorizzare le costanti calcolate

%

iteraz=0;

for n=2:2:40

xi = zeros (1 , n+1);

fxi = zeros(1,n+1);

for i = 1:n+1

xi(i) = -5+(i-1)\*(10/n);

end

for i = 1:n+1

fxi(i) = feval(f, xi(i));

end

fx=zeros(1, 10001); %valori nelle 10001 ascisse uniformi

lebesgueConstanti(n/2) = norm(lebesgue(xi),inf);

for k=1:10001

fx(k)=feval(f,z(k));

end

y=lagrange(xi, fxi,z);

err=max(abs(fx-y'));

iteraz = iteraz +1;

rst(iteraz,1) = n;

rst(iteraz,2)= err;

rst(iteraz,3)=lebesgueConstanti(n/2);

end

x=2:2:40

plot(x,lebesgueConstanti);

title('Costante di Lebesgue (Ascisse Equidistanti)');

colNames = {'n','errore','norma'};

tableResult = array2table(rst,...,

'VariableNames',colNames);

disp(tableResult);

end

function lebCheb = lebesgueCheby(f,z,lebesgueConstanti)

% Utilizzo: lebesgueEquidistanti(f,z,lebesgueConstanti)

% Funzione che calcola la costante di lebesgue per polinomi

% interpolanti di grado n:2...40 su ascisse equidistanti

%

% Parametri:

% f: Funzione da approssimare

% z:

% lebesgueConstanti: vettore per memorizzare le costanti calcolate

%

iteraz = 0;

for n=2:2:40

xi = zeros (1 , n+1);

fxi = zeros(1,n+1);

for i = 1:n+1

xi = chebyshev(n, -5, 5);

xi = sort(xi);

end

for i = 1:n+1

fxi(i) = feval(f, xi(i));

end

fx=zeros(1, 10001); %valori nelle 10001 ascisse uniformi

lebesgueConstanti(n/2) = norm(lebesgue(xi),inf);

for k=1:10001

fx(k)=feval(f,z(k));

end

y=lagrange(xi, fxi,z);

err=max(abs(fx-y'));

iteraz = iteraz +1;

rst(iteraz,1) = n;

rst(iteraz,2)= err;

rst(iteraz,3)=lebesgueConstanti(n/2);

end

x=2:2:40

plot(x,lebesgueConstanti);

title('Costante di Lebesgue (Ascisse di Chebyshev)');

colNames = {'n','errore','norma'};

tableResult = array2table(rst,...,

'VariableNames',colNames);

disp(tableResult);

end

function leb = lebesgue(puntiInterp)

## % Utilizzo: leb = lebesgue(puntiInterp)

## % Funzione per il calcolo della costante di lebesgue

## %

## % Parametri:

## % puntiInterp: punti di interpolazione

## %

## % Restituisce:

## % y: Costante di lebesgue

## npunti = length(puntiInterp);

## lin=zeros(10001,1);

## x=linspace(-5,5,10001);

## for j=1:10001

## k=0;

## for i=1:npunti

## val=abs(lagrangePol(puntiInterp,x(j),i));

## k=k+val;

## end

## lin(j,1)=k;

## end

## leb=lin;

## return

## end

## 

## 

## function valPol = lagrangePol(z,x,i)

## % Utilizzo: lagrangePol(z,x,i)

## % Funzione per il calcolo dell' i-esimo polinomio di Lagrange

## %

## % Parametri:

## %

## % z: punti di interpolazione

## % x: punto su cui effettuare la valutazione

## % i: indice del polinomio

## %

## % Restituisce:

## % valPol: Valore del polinomio in x

## 

## n = length(z); m = length(x);

## valPol = prod(repmat(x,1,n-1)-repmat(z([1:i-1,i+1:n]),m,1),2)/...

## prod(z(i)-z([1:i-1,i+1:n]));

## return

## end

## 

## 

## Tabella rappresentativa dell’errore massimo nell’approssimazione del polinomio di grado *n* con le*ascisse di Chebyshev***:**

|  |  |  |
| --- | --- | --- |
| **n** | **errore** | **norma** |
| 2 | 6.0060e-01 | 1.6667e+00 |
| 4 | 4.0202e-01 | 1.9889e+00 |
| 6 | 2.6423e-01 | 2.2022e+00 |
| 8 | 1.7084e-01 | 2.3619e+00 |
| 10 | 1.0915e-01 | 2.4894e+00 |
| 12 | 6.9216e-02 | 2.5957e+00 |
| 14 | 4.6602e-02 | 2.6867e+00 |
| 16 | 3.2614e-02 | 2.7664e+00 |
| 18 | 2.2492e-02 | 2.8371e+00 |
| 20 | 1.5334e-02 | 2.9008e+00 |
| 22 | 1.0359e-02 | 2.9587e+00 |
| 24 | 6.9484e-03 | 3.0118e+00 |
| 26 | 4.6349e-03 | 3.0608e+00 |
| 28 | 3.0782e-03 | 3.1063e+00 |
| 30 | 2.0616e-03 | 3.1487e+00 |
| 32 | 1.4017e-03 | 3.1885e+00 |
| 34 | 9.4933e-04 | 3.2260e+00 |
| 36 | 6.4075e-04 | 3.2613e+00 |
| 38 | 4.3121e-04 | 3.2948e+00 |
| 40 | 2.8946e-04 | 3.3267e+00 |

## 

## 

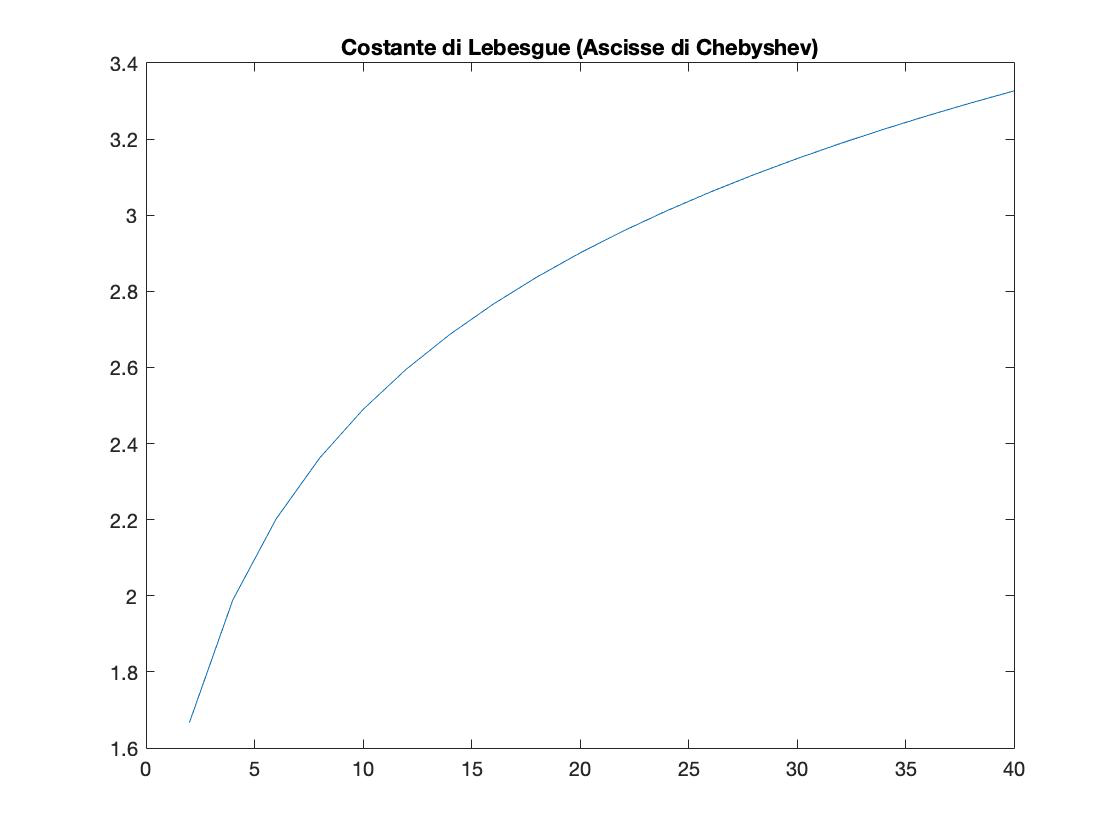
## 

## 

## 

## Tabella rappresentativa dell’errore massimo nell’approssimazione del polinomio di grado *n* con le*ascisse equidistanti***:**

|  |  |  |
| --- | --- | --- |
| **n** | **errore** | **norma** |
| 2 | 6.4623e-01 | 1.2500e+00 |
| 4 | 4.3836e-01 | 2.2078e+00 |
| 6 | 6.1695e-01 | 4.5493e+00 |
| 8 | 1.0452e+00 | 1.0946e+01 |
| 10 | 1.9157e+00 | 2.9900e+01 |
| 12 | 3.6634e+00 | 8.9325e+01 |
| 14 | 7.1949e+00 | 2.8321e+02 |
| 16 | 1.4394e+01 | 9.3453e+02 |
| 18 | 2.9190e+01 | 3.1714e+03 |
| 20 | 5.9822e+01 | 1.0987e+04 |
| 22 | 1.2362e+02 | 3.8671e+04 |
| 24 | 2.5721e+02 | 1.3785e+05 |
| 26 | 5.3817e+02 | 4.9651e+05 |
| 28 | 1.1314e+03 | 1.8038e+06 |
| 30 | 2.3883e+03 | 6.6011e+06 |
| 32 | 5.0590e+03 | 2.4309e+07 |
| 34 | 1.0749e+04 | 9.0009e+07 |
| 36 | 2.2901e+04 | 3.3489e+08 |
| 38 | 4.8907e+04 | 1.2512e+09 |
| 40 | 1.0467e+05 | 4.6924e+09 |



## Grafico rappresentativo dei valori assunti dalla costante di Lebesgue con ascisse di Chebyshev per polinomi interpolanti di grado n = 2,4,...,40.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## Grafico rappresentativo dei valori assunti dalla costante di Lebesgue con ascisse equidistanti per polinomi interpolanti di grado n = 2,4,...,40.

## Dai grafici ottenuti notiamo che la costante di Lebesgue ha una crescita esponenziale quando utilizziamo come ascisse di interpolazione ascisse equidistanti, mentre utilizzando le ascisse di Chebyshev la crescita della costante di Lebesgue è logaritmica.

## 

**Esercizio 20**

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

## Soluzione

## Si riporta la tabella rappresentativa dei massimi errori di approssimazione facendo uso delle ascisse di Chebyshev, relativa alla spline cubica naturale interpolante tali ascisse.

|  |  |
| --- | --- |
| **n** | **errore** |
| 0 | 0.00000000000000e+00 |
| 2 | 1.66801884123322e+00 |
| 4 | 2.36333087904880e+01 |
| 6 | 1.23813862105251e+01 |
| 8 | 1.37230832603283e+01 |
| 10 | 1.99696716295697e+00 |
| 12 | 8.72055877352552e+00 |
| 14 | 8.99444490393491e+00 |
| 16 | 1.15412817296127e+01 |
| 18 | 1.39494563894473e+01 |
| 20 | 1.66701539239381e+01 |
| 22 | 1.96949127548847e+01 |
| 24 | 2.29780622330274e+01 |
| 26 | 2.65460939278198e+01 |
| 28 | 3.03869404084138e+01 |
| 30 | 3.45044878860488e+01 |
| 32 | 3.88971574605510e+01 |
| 34 | 4.35652015167386e+01 |
| 36 | 4.85083907881600e+01 |
| 38 | 5.37266704542546e+01 |
| 40 | 5.92199678303044e+01 |

## Si riporta la tabella rappresentativa dei massimi errori di approssimazione facendo uso di ascisse equidistanti, relativa alla spline cubica naturale interpolante tali ascisse.

|  |  |
| --- | --- |
| **n** | **errore** |
| 0 | 0.00000000000000e+00 |
| 2 | 6.01194546811499e-01 |
| 4 | 2.79313407519679e-01 |
| 6 | 1.29300088354098e-01 |
| 8 | 5.60738528785616e-02 |
| 10 | 2.19738257495818e-02 |
| 12 | 6.90880143772588e-03 |
| 14 | 2.48286347571702e-03 |
| 16 | 3.74540283339586e-03 |
| 18 | 3.71799871804135e-03 |
| 20 | 3.18285764317361e-03 |
| 22 | 2.52965308897213e-03 |
| 24 | 1.92579236161883e-03 |
| 26 | 1.42704786366254e-03 |
| 28 | 1.03905328085696e-03 |
| 30 | 8.24362333267215e-04 |
| 32 | 6.55498681241262e-04 |
| 34 | 5.23708228635011e-04 |
| 36 | 4.21003570779233e-04 |
| 38 | 3.40837796214299e-04 |
| 40 | 2.77976540596248e-04 |

**Esercizio 21**

## 

## Uno strumento di misura ha una accuratezza di (in opportune unità di misura). I dati misurati nelle posizioni xi sono dati da yi , come descritto dalla seguente tabella:

## 

|  |  |  |
| --- | --- | --- |
| i | xi | yi |
| 0 | 0.010 | 1.003626 |
| 1 | 0.098 | 1.025686 |
| 2 | 0.127 | 1.029512 |
| 3 | 0.278 | 1.029130 |
| 4 | 0.547 | 0.994781 |
| 5 | 0.632 | 0.990156 |
| 6 | 0.815 | 1.016687 |
| 7 | 0.906 | 1.057382 |
| 8 | 0.913 | 1.061462 |
| 9 | 0.958 | 1.091263 |
| 10 | 0.965 | 1.096476 |

## 

## 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.

## 

## Soluzione

## Si mostrano in seguito la funzione per il calcolo del grado minimo nel senso dei minimi quadrati:

function gradoMinimo = calcolaGradoMinimo(n,xi,yi,tol)

% Utilizzo: gradoMinimo = calcolaGradoMinimo(n,xi,yi,tol)

% Calcola il grado minimo del polinomio nel senso dei minimi quadrati

%

% Parametri:

% n: Grado della matrice Vandermonde n\*n

% xi: Ascisse per la creazione della matrice di Vandermonde

% yi: Valore assunto dalla

% tol: tolleranza prefissata

%

% Restituisce:

% gradoMinimo: grado minimo calcolato nel senso dei minimi quadrati

## 

## for gradoMinimo=1:10

## V=vandermondeMatrix(n,gradoMinimo,xi);

## QR=myqr(V);

## a=qrsolve(QR,yi);

## if(norm(V\*a-yi,2)<tol)

## break;

## end

## end

## gradoMinimo=gradoMinimo-1;

## return

## end

## 

## 

## function VanderMatrix = vandermondeMatrix(n,m,xi)

## % Utilizzo: VanderMatrix = vandermonde(m,xi)

## % Funzione per la creazione della matrice di vandermonde V

## % partendo da un vettore xi

## %

## % Parametri:

## % n: numero righe di V

## % m: numero colonne di V

## % xi: elementi noti per la costruzione della matrice

## %

## % Resistuisce:

## % V: matrice di Vandermonde

## for i=1:n-1

## for j=i+1:n

## if(xi(i)==xi(j))

## error("Errore: Le ascisse non sono tutte distinte");

## end

## end

## end

## for i=1:n

## for j=1:m

## z=j-1;

## VanderMatrix(i,j)=xi(i)^z;

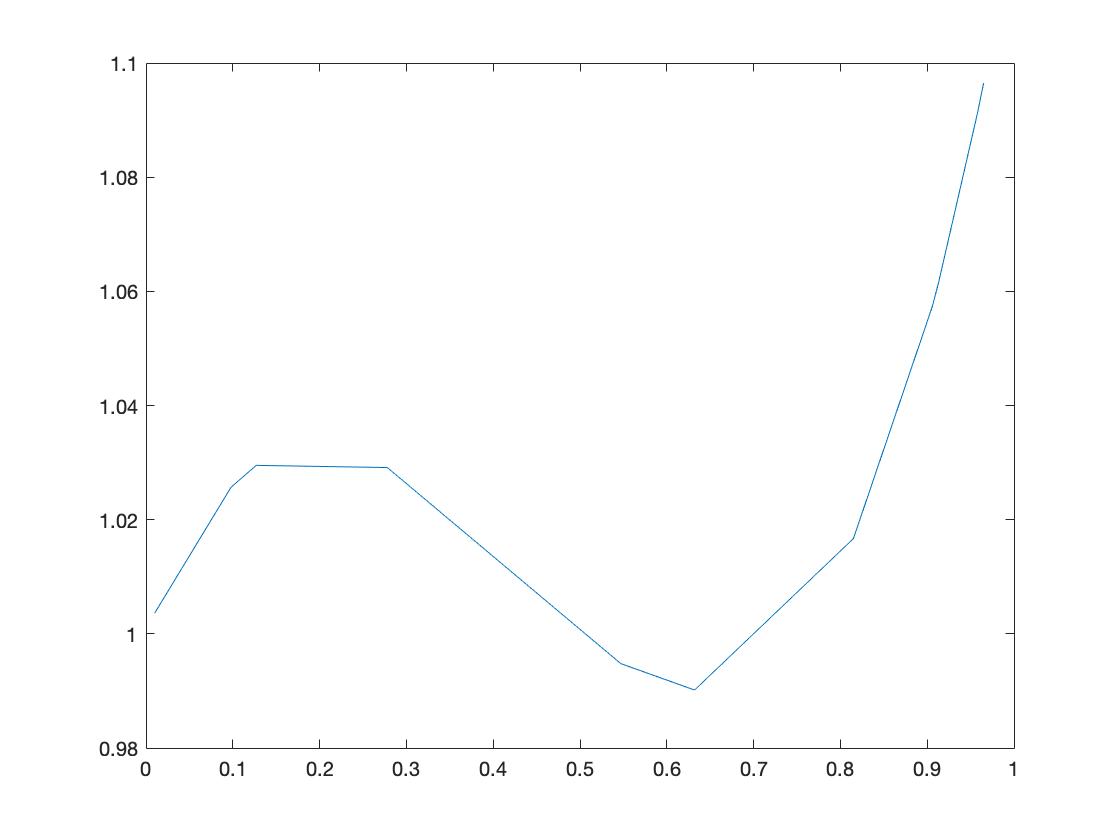
## end

## end

## return

## end

Il polinomio ha grado minimo 3 e il sottostante grafico riporta l’andamento delle coppie (xi,yi) delle misure sperimentali:



**Esercizio 22**

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

Soluzione

## Funzione che implementa la formula adattiva di Simpson:

function If = simpad( a, b, f, tol, fa, f1, fb )

% If = simpad( a, b, f, tol)

% Funzione che calcola ricorsivamente l'integrale della funzione data,

% in un intervallo specificato, utilizzando la formula adattiva di Simpson

%

% Parametri:

% a: estremo sinistro dell?intervallo

% b: estremo destro dell?intervallo

% f: funzione integranda

% tol: tolleranza prefissata ( Obbligatoriamente diversa da 0 )

% Restituisce:

% If: Approssimazione dell?integrale definito della funzione in un dato

% intervallo

if tol >= 0, error("Tolleranza specificata non corretta"), end

if a >= b, error("Intervallo di integrazione non corretto."), end

x1 = (a + b) / 2; % Calcolo del punto medio x1

if nargin <= 4 % Prima iterazione non ricorsiva

fa = f(a); % Valutazione della funzione nei punti a,b,x1

fb = f(b);

f1 = f(x1);

end

h = (b - a) / 6;

x2 = (a + x1) / 2;

x3 = (x1 + b) / 2;

f2 = f(x2);

f3 = f(x3);

I1 = h\*(fa+4\*f1+fb); % Formula di Simpson

If = .5\*h\*(fa+4\*f2+2\*f1+4\*f3+fb); % Formula di Simpson Adattiva

e = abs(If-I1)/15; % Calcolo dell?errore

if e>tol % Chiamata ricorsiva alla funzione Simpad con nuovi intervalli

If = simpad( a, x1, f, tol/2, fa, f2, f1) ...

+ simpad( x1, b, f, tol/2, f1, f3, fb);

end

end

## 

## Funzione che implementa la formula adattiva di Trapezi:

function If = trapad( a, b, f, tol, fa, fb )

% If = trapad( a, b, f, tol)

% Funzione che calcola ricorsivamente l'integrale della funzione data,

% in un intervallo specificato, utilizzando la formula dei Trapezi adattiva

%

% Parametri:

% a: estremo sinistro dell'intervallo

% b: estremo destro dell'intervallo

% f: funzione integranda

% tol: tolleranza prefissata ( Obbligatoriamente diversa da 0 )

% Restituisce:

% If: Approssimazione dell?integrale definito della funzione in un dato

% intervallo

if tol >= 0, error("Tolleranza specificata non corretta"), end

if a >= b, error("Intervallo di integrazione non corretto."), end

if nargin<=4 % Prima iterazione non ricorsiva

fa = f(a); % Valutazione della funziona in a e b

fb = f(b);

end

h = b-a;

x1 = (a+b)/2; % Calcolo del punto medio fra a e b

f1 = f(x1);

I1 = (h/2)\*(fa+fb); % Formula dei Trapezi

If = (I1+h\*f1)/2; % Formula dei Trapezi adattiva

e = abs(If-I1)/3;

if e>tol

If = trapad( a, x1, f, tol/2, fa, f1) ...

+ trapad( x1, b, f, tol/2, f1,fb);

end

end

**Esercizio 23**

Sapendo che:

Tabulare il numero dei punti richiesti dalle formule adattive dei trapezi e di Simpson per approssimare , utilizzate con tolleranze assieme ai relativi errori.

Soluzione

Nelle funzioni precedentemente esposte, è stata aggiunta una variabile globale di nome *count*, essa permette di tenere traccia del numero di punti necessari per la valutazione della funzione.

Per una migliore fruibilità della variabile, è stata usata la funzione :

assignin('base',’nPoint’,count)

Questa funzione permette di assegnare la variabile, nel nostro caso globale, countGlobal alla variabile ’nPoint’ presente nel workspace ’base’.

function If = trapad( a, b, f, tol, fa, fb )

% Utilizzo: If = trapad( a, b, f, tol)

% Calcola ricorsivamente l'integrale della funzione, nell'intervallo prescelto,

% usando la formula dei trapezi adattiva.

%

% Input:

% a: estremo sinistro

% b: estremo destro

% f: funzione integranda

% tol: tolleranza prefissata

% Output:

% If: l’approssimazione dell’integrale definito della funzione

% Controlli di robustezza:

% - a deve essere minore di b

if a>=b

error("Intervallo di integrazione non corretto.")

end

global count;

if nargin<=4

fa = f(a);

fb = f(b);

count = 2;

end

h = b-a;

x1 = (a+b)/2;

f1 = f(x1);

count = count + 1;

I1 = (h/2)\*(fa+fb);

If = (I1+h\*f1)/2;

e = abs(If-I1)/3;

if e>tol

If = trapad( a, x1, f, tol/2, fa, f1) + trapad( x1, b, f, tol/2, f1, fb);

end

assignin('base',"nPoint",count)

end

function If = simpad( a, b, f, tol, fa, f1, fb )

% Utilizzo: If = simpad( a, b, f, tol)

% Funzione che calcola ricorsivamente l'integrale della funzione data,

% in un intervallo specificato, utilizzando la formula adattiva di Simpson

%

% Parametri:

% a: estremo sinistro dell?intervallo

% b: estremo destro dell?intervallo

% f: funzione integranda

% tol: tolleranza prefissata ( Obbligatoriamente diversa da 0 )

% Restituisce:

% If: Approssimazione dell?integrale definito della funzione in un dato

% intervallo

if tol <= 0, error("Tolleranza specificata non corretta"), end

if a >= b, error("Intervallo di integrazione non corretto."), end

x1 = (a + b) / 2; % Calcolo del punto medio x1

global count;

if nargin <= 4 % Prima iterazione non ricorsiva

fa = f(a); % Valutazione della funzione nei punti a,b,x1

fb = f(b);

f1 = f(x1);

count = 2; % Inizializzo count a 2 per i punti passati come arg.

end

h = (b - a) / 6;

x2 = (a + x1) / 2;

x3 = (x1 + b) / 2;

count = count + 3; % Aggiungo a count il calcolo dei punti x1,x2,x3

f2 = f(x2);

f3 = f(x3);

I1 = h\*(fa+4\*f1+fb); % Formula di Simpson

If = .5\*h\*(fa+4\*f2+2\*f1+4\*f3+fb); % Formula di Simpson Adattiva

e = abs(If-I1)/15; % Calcolo dell?errore

if e>tol % Chiamata ricorsiva alla funzione Simpad con nuovi intervalli

If1 = simpad( a, x1, f, tol/2, fa, f2, f1);

If2 = simpad( x1, b, f, tol/2, f1, f3, fb);

If = If1 + If2;

end

assignin('base',"nPoint",count);

end

## Tabella relativa alla funzione che implementa la formula adattiva di Simpson:

## 

|  |  |  |  |
| --- | --- | --- | --- |
| Toll. | Valore ottenuto | Punti necessari | Err. Relativo |
| 10^-2 | 30.0024 | 71 |  |
| 10^-3 | 30.0006 | 113 |  |
| 10^-4 | 30.0001 | 203 |  |
| 10^-5 | 30.0000 | 371 | 0 |
| 10^-6 | 30.0000 | 635 | 0 |
| 10^-7 | 30.0000 | 1145 | 0 |
| 10^-8 | 30.0000 | 2045 | 0 |

## 

## Tabella relativa alla funzione che implementa la formula adattiva dei Trapezi:

## 

|  |  |  |  |
| --- | --- | --- | --- |
| Toll. | Valore ottenuto | Punti necessari | Err. Relativo |
| 10^-2 | 30.0048 | 375 |  |
| 10^-3 | 30.0006 | 1181 |  |
| 10^-4 | 30.0001 | 3687 |  |
| 10^-5 | 30.0000 | 11883 | 0 |
| 10^-6 | 30.0000 | 37273 | 0 |
| 10^-7 | 30.0000 | 116747 | 0 |
| 10^-8 | 30.0000 | 375793 | 0 |

## La funzione *trapad*, che implementa la formula adattiva dei trapezi, nella prima iterazione usa i due punti passati come argomenti della funzione ‘a’ e ‘b’, in seguito calcola un nuovo punto medio di ‘a’ e ‘b’ chiamato ‘x1’.

## Nelle successive chiamate, verrà calcolato solamente un nuovo punto medio del nuovo intervallo.

## 

## La funzione *simpad*, che implementa la formula di Simpson adattiva, nella prima iterazione usa 3 punti ‘a’,’b’ e ‘x1’ che viene calcolato, essendo il punto medio di ‘a’ e ‘b’; inoltre vengono calcolati altri due punti ‘x2’ ed ‘x3’ rispettivamente punti medi fra ‘a’ ed ‘x1’ e tra ‘x1’ ed ‘b’.

## Nelle successive chiamate di funzione, solo i punti ‘x1’, ’x2’ ed ‘x3’ verranno calcolati nuovamente.

**Esercizio 24**

Scrivere una function che implementi efficientemente il metodo delle potenze.

Soluzione

## Il codice della function è il seguente:

function [lambda, i] = metodoPotenze(A, tol, x0, maxit)

% Utilizzo: [lambda, i] = metodoPotenze(A, [tol, [x0, [maxit]]])

% Restituisce l'autovalore dominante della matrice A e il numero di

% iterazioni necessarie per calcolarlo.

%

% Parametri:

% A: matrice utilizzata per il calcolo

% [tol]: tolleranza dell' approssimazione (specificata o omessa)

% [x0]: vettore di partenza (specificato o omesso)

% [maxit]: numero massimo di iterazioni

% Restituisce:

% lambda: matrice quadrata nxn sparsa

% i: numero di iterazioni

%

[m,n] = size(A); % Inizializzo m ed n con la dimensione della matrice

if m ~= n % Controllo se la matrice è una matrice quadrata

error('Errore: La matrice deve essere quadrata.');

end

if nargin <= 1

tol = 10^(-6);

elseif (tol >= 0.1) || (tol <= 0)

error("Errore: Tolleranza specificata non corretta");

end

if nargin <= 2 % Genero il vettore iniziale se non specificato

x=rand(n,1);

else

x = x0;

end

if nargin <= 3 % Inizializzo il numero di iterazioni massime se non specificate

maxit = ceil(-log(tol))\*n;

end

x = x/norm(x);

lambda = inf;

for i = 1:maxit

lambda0 = lambda;

v = A\*x;

lambda = x'\*v;

err = abs(lambda-lambda0); % Calcolo dell'errore

if err < tol\*(1+abs(lambda)) % Se il valore dell'errore è accettabile interrompo il ciclo

break;

end

x = v/norm(v);

end

if err > tol\*(1+abs(lambda)) % Se il valore dell'errore non è accettabile dopo le iterazioni termino

warning('la tolleranza richiesta non è stata raggiunta.');

end

end

function A = matrSparsa(n)

% Utilizzo: A = matrSparsa(n)

% Genera la matrice quadrata sparsa nxn, con n maggiore di 10.

%

% Parametri:

% n: numero di righe/colonne della matrice quadrata sparsa

% Restituisce:

% A: matrice quadrata nxn sparsa

%

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,9,A);

A = spdiags(d,-9,A);

end

end

Esercizio 25

## Sia data la matrice di Toeplitz simmetrica:

## 

## 

## in cui le extra-diagonali pi`u esterne sono le none.

## Partendo dal vettore u0 = (1, . . . , 1)> ∈ R N , 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.

Soluzione

## La function che implementa il metodo delle potenze è la seguente:

function [lambda, i] = metodoPotenze(A, tol, x0, maxit)

% Utilizzo: [lambda, i] = metodoPotenze(A, [tol, [x0, [maxit]]])

% Funzione che restituisce l'autovalore di modulo massimo della matrice A e il numero di

% iterazioni necessarie per il suo calcolo.

% Parametri:

% A: matrice utilizzata per il calcolo

% [tol]: tolleranza dell' approssimazione (specificata o omessa)

% [x0]: vettore di partenza (specificato o omesso)

% [maxit]: numero massimo di iterazioni

% Restituisce:

% lambda: matrice quadrata nxn sparsa

% i: numero di iterazioni eseguite

%

[m,n] = size(A); %Inizializzo m ed n con la dimensione della matrice

if m ~= n % Controllo se la matrice è una matrice quadrata

error('La matrice deve essere quadrata.');

end

if nargin <= 1 % Inizializzo la tolleranza se non è specificata

tol = 10^(-6);

elseif (tol >= 0.1) || (tol <= 0) % Controllo di robustezza sulla tolleranza

error("Tolleranza specificata non corretta");

end

if nargin <= 2 % Genero il vettore iniziale se non specificato

x=rand(n,1);

else

x = x0;

end

if nargin <= 3 % Inizializzo il numero di iterazioni massime se non specificate

maxit = ceil(-log(tol))\*n;

end

x = x/norm(x);

lambda = inf; % Inizializzo lambda con il valore infinito

for i = 1:maxit

lambda0 = lambda;

v = A\*x;

lambda = x'\*v;

err = abs(lambda-lambda0); % Calcolo dell'errore

if err < tol\*(1+abs(lambda))

break;

end

x = v/norm(v);

end

if err > tol\*(1+abs(lambda)) % Valore dell'errore non accettabile dopo le iterazioni

warning('la tolleranza richiesta non è stata raggiunta.');

end

end

function A = matrSparsa(n)

% A = matrSparsa(n)

% Genera la matrice quadrata sparsa nxn, con n maggiore di 10.

%

% Parametri:

% n: numero di righe/colonne della matrice quadrata sparsa

% Restituisce:

% A: matrice quadrata nxn sparsa

%

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,9,A);

A = spdiags(d,-9,A);

end

end

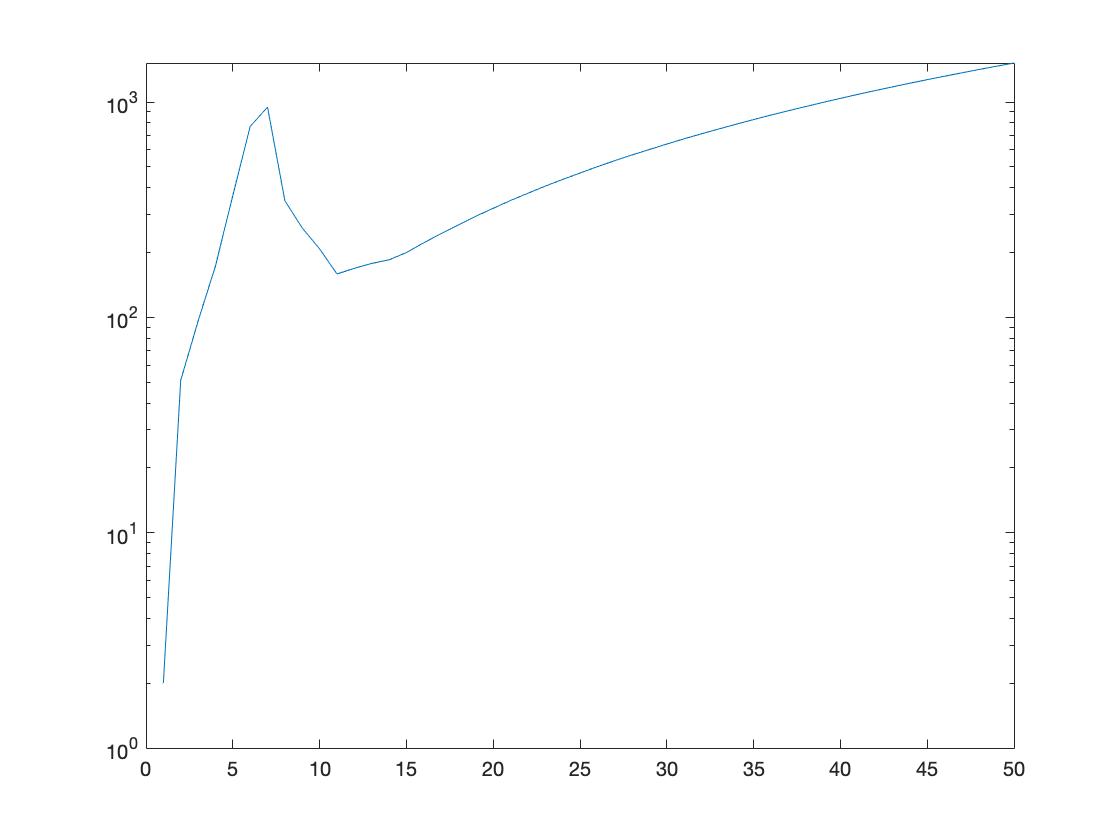


Grafico che rappresenta il numero di iterazioni effettuate.

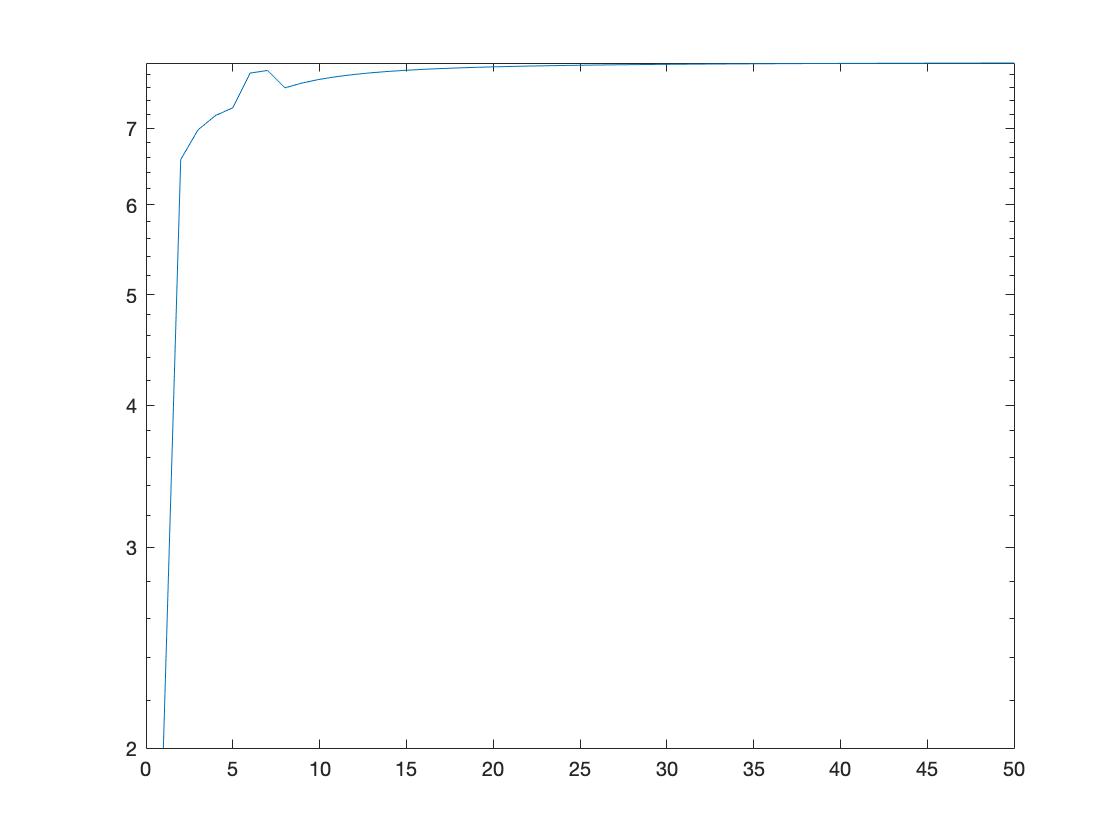


Grafico che rappresenta l’approssimazione dell’autovalore dominante.

**Esercizio 26**

## 

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

Soluzione

## Si riporta di seguito la function relativa alla risoluzione di un sistema lineare:

function [x,i,nr] = splittingGenerico(A, b, Msolve, tol, x0, maxit)

% Utilizzo: [x,i] = jacobi(A, b, [tol, [xo, [maxit]]])

% Restituisce la soluzione del sistema lineare Ax=b approssimata con il

% metodo utilizzato dalla funzione Msolve e il numero di iterazioni eseguite.

% Parametri:

% A: matrice utilizzata per il calcolo

% b: vettore dei termini noti

% Msolve: funzione che implementa il metodo di risoluzione

% [tol]: tolleranza dell' approssimazione (specificata o omessa)

% [x0]: vettore di partenza (specificato o omesso)

% [maxit]: numero massimo di iterazioni (specificato o omesso)

% Restituisce:

% x: soluzione approssimata del sistema

%

D = diag(A);

if ~all(D) % Controllo la diagona principale non sia nulla

error('La diagonale di A non deve avere elementi nulli');

end

n = length(b); %Inizializzo n con la lunghezza del vettore dei termini noti

if nargin <= 3

tol = 10^(-6);

elseif (tol >= 0.1) || (tol <= 0)

error("Tolleranza specificata non corretta");

end

if nargin <= 4 % Genero il vettore iniziale se non specificato

x=rand(n,1);

else

x = x0;

end

if nargin <= 5 % Inizializzo il numero di iterazioni massime se non specificate

maxit = ceil(-log(tol))\*n;

end

for i = 1:maxit

r = A\*x - b;

err = norm(r,inf);

nr(i) = err;

if err<=tol % Se l'errore è inferiore alla tolleranza termino il ciclo

break;

end

r = Msolve(A,r); % Risolvo il sistema con il metodo implementato dalla Msolve

x = x-r;

end

if err>tol % Se l

warning('La tolleranza richiesta non è stata raggiunta.');

end

end

function A = matrSparsa(n)

% A = matrSparsa(n)

% Genera la matrice quadrata sparsa nxn, con n maggiore eo uguale di 10.

%

% Parametri:

% n: numero di righe/colonne della matrice quadrata sparsa

% Restituisce:

% A: matrice quadrata nxn sparsa

%

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

**Esercizio 27**

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

Soluzione

## 

## La funzione splittingGenerico usa una di queste due funzioni per il calcolo di un approssimazione del sistema lineare Ax=b.

## Viene riportato il codice inseguito:

function y = MsolveGauss(M, r)

% y = MsolveGauss(M, r)

% Restituisce la soluzione del sistema lineare Mx=r.

% Ogni iterazione risolve un sistema triangolare inferiore

y=r;

n = length(r);

for i = 1:n

y(i) = y(i)/M(i,i);

y(i+1 : n) = y(i+1 : n) - M(i+1 : n,i)\*y(i);

end

end

function y = MsolveJacobi(M, r)

% y = MsolveJacobi(M, r)

% Restituisce la soluzione del sistema lineare Mx=r.

% Ogni iterazione risolve un sistema diagonale

y = r./diag(M);

end

**Esercizio 28**

## Con riferimento alla matrice risolvere il sistema:

## 

## 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 sparsit`a (nota) della matrice AN .

## Graficare il numero delle iterazioni richieste dai due metodi iterativi, rispetto ad N, per soddisfare il criterio di arresto prefissato.

Soluzione

## Viene riportato solo il codice relativo alla funzione prodMatVec, che è stato scritto ad-hoc per la matrice data:

function y = prodMatVec(A,x)

y=4\*x;

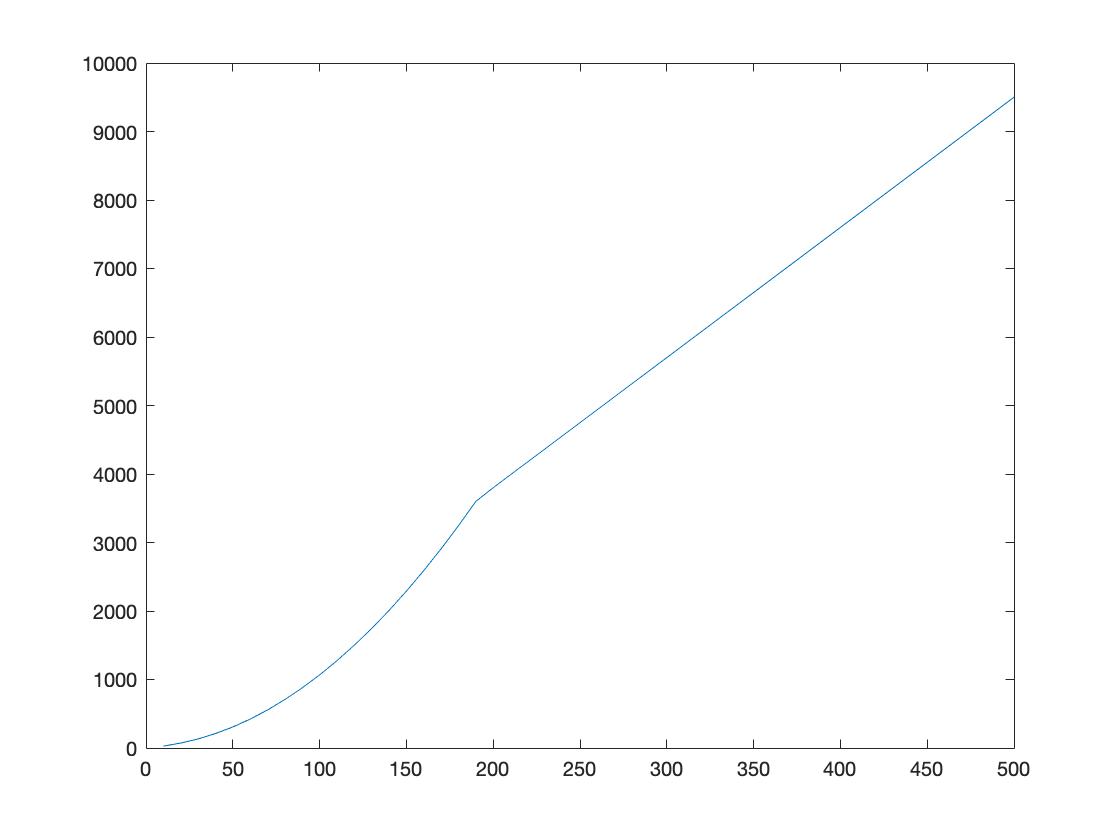
y(1:end-1)=y(1:end-1)-x(2:end);

y(2:end)=y(2:end)-x(1:end-1);

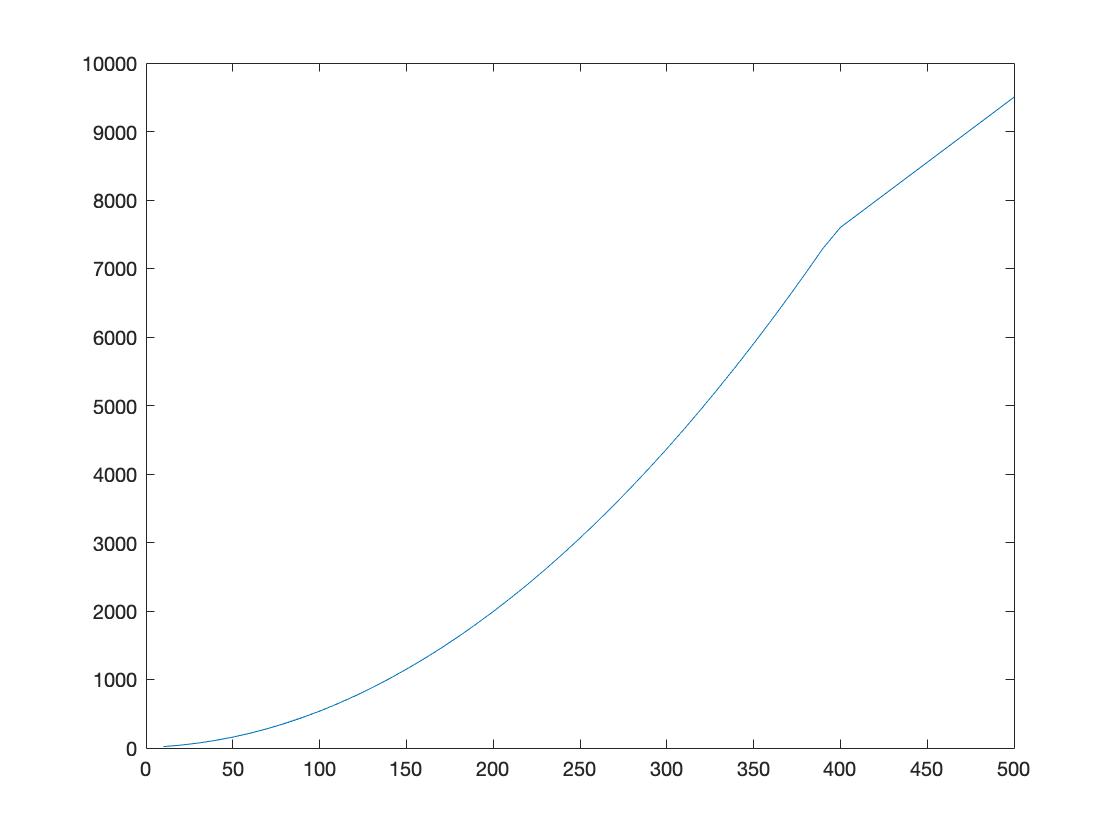
y(10:end)=y(10:end)-x(1:end-9);

y(1:end-9)=y(1:end-9)-x(10:end);

end



## Rappresenta il numero di iterazioni effettuate, con il metodo di Jacobi.



## Rappresenta il numero di iterazioni effettuate, con il metodo di Gauss-Seidel.