Progetto di Calcolo Numerico

Flavio Casadei D.C. & Lorenzo Masetti

Indice

1 Risoluzione di sistemi lineari **2**
- 1.1 Files di uso comune . . . 2
- 1.2 Fattorizzazioni . . . 4
  - 1.2.1 Fattorizzazione LU . . . 6
  - 1.2.2 Fattorizzazione QR . . . 14
  - 1.2.3 Fattorizzazione LDL^T . . . 20
- 1.3 Metodi iterativi . . . 23
  - 1.3.1 Metodo di Jacobi . . . 24
  - 1.3.2 Metodo di Gauss-Seidel . . . 27

2 Equazioni non lineari **30**
- 2.1 Introduzione . . . 30
- 2.2 Metodo di bisezione . . . 30
- 2.3 Metodo di Newton . . . 35
- 2.4 Metodo di accelerazione di Aitken . . . 37
- 2.5 Metodo delle corde . . . 43
- 2.6 Metodo delle secanti . . . 46

3 Interpolazione di funzioni **50**
- 3.1 Interpolazione polinomiale . . . 50
- 3.2 Sistema di Vandermonde . . . 50
- 3.3 Interpolazione nella base di Newton . . . 52
- 3.4 Ascisse di Cebyshev . . . 58
- 3.5 Interpolazione nella base di Lagrange . . . 64
- 3.6 Neville . . . 68
- 3.7 Spline cubiche . . . 71

4 Il metodo dei minimi quadrati **84**

5 Approssimazione di integrali **91**
5.1 Newton Cotes 91
5.1.1 Accuratezza ed errore 97
5.1.2 Formule composite 97
5.2 Formule adattative 103
5.3 Metodo di Romberg 105

6 Ricerca dell'autovalore dominante **108**
6.1 Introduzione 108
6.2 Metodo delle potenze 108

Elenco delle figure

2.1 Metodo di Newton applicato alla funzione $y = e^x - 1$ 38
2.2 Metodo delle corde applicato alla funzione $y = e^x - 1$ 45
2.3 Metodo delle secanti applicato alla funzione $y = e^x - 1$ 49

3.1 Interpolazione della funzione seno con un polinomio di grado 4 57
3.2 Interpolazione della funzione seno con un polinomio di grado 8 58
3.3 Interpolazione funzione di Runge con ascisse a distanza costante (grado 6) 59
3.4 Interpolazione funzione di Runge con ascisse a distanza costante (grado 9) 60
3.5 Interpolazione funzione di Runge con ascisse a distanza costante (grado 12) 60
3.6 Interpolazione funzione di Runge con ascisse a distanza costante (grado 20) 61
3.7 Interpolazione funzione di Runge con ascisse di Cebyshev (grado 6) 62
3.8 Interpolazione funzione di Runge con ascisse di Cebyshev (grado 9) 63
3.9 Interpolazione funzione di Runge con ascisse di Cebyshev (grado 12) 63
3.10 Interpolazione funzione di Runge con ascisse di Cebyshev (grado 20) 64
3.11 Interpolazione della funzione di Runge con una spline su 7 punti 82
3.12 Interpolazione della funzione di Runge con una spline su 11 punti 82
3.13 Interpolazione della funzione di Runge con una spline su 19 punti 83

4.1 Minimi quadrati su una retta perturbata 89

4.2 Minimi quadrati su una parabola perturbata 90

Introduzione

In questa relazione vengono presentate le implementazioni in MATLAB degli algoritmi approfonditi durante il corso di calcolo numerico.

Ogni algoritmo viene presentato con una breve descrizione e con il listato della function Matlab corrispondente. Vengono inoltre forniti alcuni esempi del funzionamento ove questo si renda necessario.

Ogni listato contiene all'inizio un commento che viene visualizzato dall'help in linea del matlab e quindi descrive l'algoritmo all'utente tralasciando i particolari implementativi.

Degli algoritmi per l'interpolazione di funzioni non vengono forniti esempi non grafici in quanto una lista di numeri è molto meno leggibile di un grafico.

Capitolo 1

Risoluzione di sistemi lineari

In questa parte vengono presentati i metodi per la risoluzione di sistemi lineari del tipo $A\underline{x} = \underline{b}$ con le seguenti precondizioni sul problema:

$$A \in R^{n \times n} \quad \det(A) \neq 0 \quad \underline{b} \in R^n$$

la soluzione più immediata è quella di porre $\underline{x} = A^{-1}\underline{b}$ però questa operazione è costosissima e quindi sono stati sviluppati metodi più efficienti per la risoluzione del problema; vedremo due tipo di metodo: le fattorizzazioni, che scompongono la matrice A nel prodotto di matrici più "semplici" e i metodi iterativi, che *splittano* A nella differenza di due matrici e poi ricorsivamente operano su queste per ottenere la soluzione.

1.1 Files di uso comune

Nei nostri algoritmi per la risoluzione di sistemi lineari vengono comunemente ustilizzati due semplici algoritmi: un risolutore di sistemi triangolari superiori

---

$\underline{x}$=solveU(A,$\underline{b}$) - *Data A matrice triangolare superiore la funzione risolve il sistema lineare $A\underline{x} = \underline{b}$. La funzione se applicata ad una matrice non triangolare superiore, tratta la matrice come se lo fosse, ignorando i vari elementi che si trovano sotto la diagonale principale.*

CODICE: solveU.m

```
%SOLVEU
%x=solveU(A,b)
% Pre: A matrice triangolare superiore non singolare
%
% Questa funzione restituisce in x la soluzione di un
    sistema
% lineare di di n equazioni in n incognite
% la cui matrice dei coefficienti di dimensioni n x nè
    A
% e il cui vettore dei termini noti (di dimensione nè)
    b.
%
% Se A nonè una matrice triangolare superiore la
    procedura
% considera solo la parte triangolare superiore di A,
    come
% se la restante parte fosse composta di soli zeri.
%
% See also PLU, QRHOUSE
function x=solveU(A,b)
n=length(b);
x=zeros(n,1); x=x';
for i=n:-1:1
    somma=0;
    for j=i+1:n
        somma=somma+A(i,j)*x(j);
    end;
    x(i)=(b(i)-somma)/A(i,i);
end;
return
```

ed un risolutore di sistemi triangolari inferiori

$\underline{x}$=solveL(A,$\underline{b}$) - *Data A matrice triangolare inferiore la funzione risolve il sistema lineare $A\underline{x} = \underline{b}$. La funzione se applicata ad una matrice non triangolare inferiore, tratta la matrice come se lo fosse, ignorando i vari elementi che si trovano sopra la diagonale principale.*

CODICE: solveL.m

```
%SOLVEL
%x=solveL(A,b)
% Pre: A matrice triangolare inferiore non singolare
%
% Questa funzione restituisce in x la soluzione di un
    sistema
% lineare di di n equazioni in n incognite
% la cui matrice dei coefficienti di dimensioni n x nè
    A
% e il cui vettore dei termini noti (di dimensione nè)
    b.
%
% Se A nonè una matrice triangolare inferiore la
    procedura
% considera solo la parte triangolare inferiore di A,
    come
% se la restante parte fosse composta di soli zeri.
%
% See also PLU, QRHOUSE
function x=solveL(A,b)
n=length(b);
x=zeros(n,1); x=x';
for i=1:n
    somma=0;
    for j=1:i-1
        somma=somma+A(i,j)*x(j);
    end;
    x(i)=(b(i)-somma)/A(i,i);
end;
return
```

1.2 Fattorizzazioni

Come accennato precedentemente le fattorizzazioni vengono impiegate per risolvere un sistema con un costo asintotico minore rispetto a quello di

una "brutale" inversione di martice, e prendono spunto da alcune matrici "semplici", con le quali è immediato e poco costoso risolvere i sistemi lineari:

- Matrici diagonali
- Matrici triangolari
- Matrici ortogonali

l'idea di fondo delle fattorizzazioni è quella di scomporra la matrice A nel prodotto di matrici appartenenti alle categorie suddette in modo da facilitare e sveltire il calcolo della soluzione. Vediamo in dettaglio i casi "banali"

Matrici diagonali

$$A = \begin{pmatrix} a_1 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & a_n \end{pmatrix}$$

quindi il sistema diventa

$$\begin{cases} a_1 x_1 & = & b_1 \\ \dots & \dots & \dots \\ a_n x_n & = & b_n \end{cases}$$

con l'ovvia soluzione $x_i = b_i/a_i \quad \forall i = 1 \dots n$ che costa n *flops* (divisioni).

Matrici triangolari

$$A = \begin{pmatrix} a_{1\,1} & 0 & \dots & 0 & 0 \\ \vdots & \ddots & & & \vdots \\ \vdots & & a_{j\,j} & & 0 \\ \vdots & & & \ddots & \vdots \\ a_{n\,1} & \dots & \dots & \dots & a_{n\,n} \end{pmatrix}$$

ovviamente A è triangolare inferiore e vale la seguente condizione $a_{i\,j} = 0$ se $j > i$; per le triangolari superiori vale invece $a_{i\,j} = 0$ se $j < i$. In entrambi i casi $\det(A) = \prod_{i=1}^{n} a_{i\,i} \neq 0 \leftrightarrow a_{i\,i} \neq 0 \ \forall i = 1 \dots n$. Nel caso

delle triangolari inferiori l'equazione i-esima è $a_{i\,1}x_1 + \ldots + a_{i\,i} = b_i$ con $x_1, \ldots x_{i-1}$ determinate ai passi precedenti; nell'altro caso l'equazione i-esima è $a_{i\,i}x_i + \ldots + a_{i\,n}x_n = b_i$ con $x_{i+1}, \ldots, x_n$ determinati precedentemente. In entrambi i casi il costo dell'algoritmo è quadratico.

Matrici ortogonali

Se A é ortoginale allora $AA^T = A^TA = I$ quindi $A^{-1} = A^T$ ed il sistema $A\underline{x} = \underline{b}$ si risolve come $\underline{x} = A^T\underline{b}$. Il costo é circa $2n^2$ *flops* in quanto consta di n prodotti scalari, ognuno dei quali costa $2n-1$ *flops*: n moltiplicazioni e $n-1$ somme.

1.2.1 Fattorizzazione LU

La fattorizzazione LU consiste nello scomporre una matrice A nel prodotto du due matrici, L triangolare inferiore ed U triangolare superiore con la particolaritá che la matrice L ha sulla diagonale tutti 1; se esiste tale fattorizzazione è unica. Una volta fattorizzata A basta risolvere il seguente sistema lineare per ottenere la soluzione

$$\begin{cases} L\underline{y} &= \underline{b} \\ U\underline{x} &= \underline{y} \end{cases}$$

Se la matrice A è di ordine n allora l'algoritmo consiste di $n-1$ passi ognuno dei quali annulla gli elementi di una colonna (quella corrispondente al numero del passo) sotto la diagonale principale e lascia inalterato il "lavoro svolto" ai passi precedenti; tutto questo si ottiene premoltiplicando a sinistra A per delle matrici con struttura analoga alla L e chiamata matrici elementari di Gauss . Descrizione dell'algoritmo: con A^k viene indicata la matrice A al passo k-esimo e con $a^k_{i\,j}$ l'ememento in riga i,in colonna j ed al passo k della matrice A. L'algoritmo parte con

$$A = A^1 = \begin{pmatrix} a^1_{1\,1} & a^1_{1\,2} & \ldots & a^1_{1\,n} \\ a^1_{2\,1} & a^1_{2\,2} & \ldots & a^1_{2\,n} \\ \ldots & \ldots & \ldots & \ldots \\ a^1_{n\,1} & a^1_{n\,2} & \ldots & a^1_{n\,n} \end{pmatrix}$$

e ricava $A^2 = L^1 A$ dove adesso A^2 è

$$\begin{pmatrix} a^1_{1\,1} & a^1_{1\,2} & \dots & a^1_{1\,n} \\ 0 & a^2_{2\,2} & \dots & a^2_{2\,n} \\ \vdots & \vdots & \dots & \vdots \\ 0 & a^2_{n\,2} & \dots & a^2_{n\,n} \end{pmatrix}$$

si nota che adesso nelle righe inferiori alla prima compare l'indice 2 in alto e questo significa che l'algoritmo non modifica quello che era stato fatto ai passi precedenti e che via via si vanno a modificare le righe inferiori della matrice, lasciando inalterate le altre. Al passo i-esimo si ricava A^{i+1} come i prodotto $L^i A^i$ dove $A^i = L_{i-1} \cdots L_2 L_1 A$. Di queste matrici L_i ce ne sono esattamente $n-1$ e la loro forma è del tipo

$$L_i = I - \underline{g}_i \underline{e}_i^T$$

dove $\underline{e}_i^T$ indica nel modo usuale l'i-esimo vettore della base canonica di R^n trasposto e $\underline{g}_i$ indica l'i-esimo vettore di Gauss definito come

$$\underline{g}_i = \frac{1}{a^i_{i\,i}} (\underbrace{0 \dots 0}_{i}\ a^i_{i+1\,i}\ \dots\ a^i_{n\,i})^T$$

Se indichiamo con L^{-1} il prodotto $L_{i-1} \cdots L_2 L_1$ si ha proporio che $A = LU$, infatti si ha che l'inversa di L^i è $I - \underline{g}_i \underline{e}_i^T$ ed è triangolare inferiore con tutti 1 sulla diagonale, e quindi lo è anche il loro prodotto. L' algoritmo al passo i-esimo modifica la matrice nel seguente modo

$$\begin{cases} a^{i+1}_{k\,i} &= \frac{a^i_{k\,i}}{a^i_{i\,i}} & k = i+1 \dots n \\ a^{i+1}_{k\,j} &= a^i_{k\,j} - a^{i+1}_{k\,i} \cdot a^i_{i\,j} & j = i+1 \dots n \end{cases}$$

e poichè vengono annullati $n-i$ elementi sotto la diagonale, le loro posizioni possono essege occupare dalle componenti significative del vettore $\underline{g}_i$.

LU senza pivoting

Guardando attentamentela definizione di $\underline{g}_i$ si nota che viene effettuata una divisione e quindi un'ulteriore precondizione per la fattibilità del problema è che $a^i_{i\,i} \neq 0\ \forall i = 1 \dots n$ e questa condizione è equivalente a $\det(A_k) \neq 0\ \forall k = 1 \dots n$ dove con A_k viene indicato il minore principale di ordine k della matrice A.

[$\underline{x}$,A]=simpleLU(A,$\underline{b}$) - *Data la matrice A tale che* $\det A_k \neq 0 \quad k = 1 \ldots n$ *ed il vettore* $\underline{b}$*, fattorizza la matrice A nel prodotto LU, con L matrice triangolare inferiore con diagonale composta da tutti 1 ed U matrice triangolare superiore. Restituisce in* $\underline{x}$ *la soluzione del sistema* $LU\underline{x} = \underline{b}$ *e memorizza nella parte triangolare superiore di A la parte significativa di U, nella parte inferiore esclusa la diagonale invece inserisce la parte significativa (senza la diagonale) della matrice L.*

CODICE: simpleLU.m

```
%SIMPLELU
%[x,LU]=SIMPLELU(A,b)
%Pre: A matrice n x n tale che tutti i suoi minori
   principali
%     sono non singolari,
%     b vettore di lunghezza n
%
% La funzione risolve il sistema lineare di n equazioni
    in n
% incognite
% Ax=b con il metodo della fattorizzazione LU.
% Restituisce:
% - in x la soluzione del sistema,
% - in LU una matirce che contiene nella parte
   triangolare
%       superiore la matrice U
% e nella parte strettamente
%       triangolare inferiore gli elementi
   significativi
%       della matrice L (in pratica Lè una matrice che
    ha la
%       parte strettamente triangolare
%  inferiore uguale ad LU
%       e tutti 1 sulla diagonale principale).
% Le matrici L,U sono tali che A=L*U
%
```

```
% See also PLU, QRHOUSE
function [x,A]=simpleLU(A,b)
n=length(b);
for i=1:n-1
    for j=i+1:n
        if A(i,i)==0
            error('Matrice non fattorizzabile LU')
        end
        A(j,i)=A(j,i)/A(i,i);
        A(j,i+1:n)=A(j,i+1:n)-A(j,i)*A(i,i+1:n);
    end;
end;
L=tril(A,-1)+eye(n);
U=triu(A);
y=solveL(L,b);
x=solveU(U,y);
return
```

ESEMPIO di simpleLU.m

```
>> A

A =

     1     2     3     4
     5     6     7     8
     9    10    32   354
    65    78    98    54

>> b

b =

     1
     2
    54
     7
```

```
>> inv(A)*b

ans =

  -1.20731707317073
   2.38414634146343
  -1.14634146341463
   0.21951219512195

>> simpleLU(A,b)

ans =

  -1.20731707317073
   2.38414634146341
  -1.14634146341463
   0.21951219512195
```

Fattorizzazione LU con pivoting parziale

La possibilità che $\exists i | a^i_{i\,i} = 0\ i = 1 \ldots n$ non dà noia più di tanto, inquanto siamo sicuri che una colonna non può essere nulla ($\det(A) \neq 0$) e quindi con un piccolo accorgimento è possibile applicare la fattorizzazione LU ad ogni matrice non singolare. Infatti al passo i-esimo basta permutare l'eventuale elemento nullo con un altro non nullo e riflettere questa permutazione su tutta la riga (anche sul vettore dei termini noti) per ripristinare la situazione ideale. Esiste un accorgimento per migliorare il condizionamento del problema: ad ogni passo permutare l'elemento *pivot* ($a^i_{i\,i}$) con l'elemento di massimo modulo sulla stessa colonna, da qui viene il nome pivoting parziale. Per permutare una riga con un'altra basta moltiplicare la matrice A a sinistra per una matrice di permutazione che è la matrice identica con due righe scambiate, le quali sono quelle che volevamo scambiare in A. Questo procedimento viene applicato ad ogni passo ed in questo modo si ottiene una fattorizzazione $PA = LU$

[$\underline{x}$,A,$\underline{p}$]=plu(A,b) - *Data la matrice A con determinante non nullo e $\underline{b}$ vet-*

tore dei termini noti, la funzione fattorizza A come il prodotto di matrici LU con U triangolare superiore, L triangolare inferiore con elementi diagonali uguali ad 1 e risolve il sistema $LU\underline{x} = P\underline{b}$ *con P opportuna matrice di permutazione. La funzione restituisce anche il vettore* $\underline{p}$ *contenente i vari indici della permutazione finale delle righe di A. La matrice L viene posta nella parte triangolare inferiore di A meno la diagonale principale che deve contenere tutti 1, la matrice U viene posta nela parte triangolare superiore di A.*

CODICE: plu.m

```
%PLU
%[x,LU,p]=PLU(A,b)
%Pre: A matrice n x n non singolare, b vettore di
   lunghezza n
%
% La funzione risolve il sistema lineare di n equazioni
    in n
% incognite
% Ax=b con il metodo della fattorizzazione LU con
   pivoting
% parziale.
% Restituisce:
% - in x la soluzione del sistema,
% - in LU una matirce che contiene nella parte
   triangolare
%   superiore la matrice U
% e nella parte strettamente
%   triangolare inferiore gli elementi significativi
% della
%   matrice L (in pratica Lè una matrice che ha la
   parte
%   strettamente triangolare
% inferiore uguale ad LU e tutti 1
%   sulla diagonale principale).
% - nel vettore p la permutazione applicata alla
   matrice A.
```

```
%   La matrice P
% òpu essere ricostruita permutando le colonne
%   della matrice àidentit secondo
%l'ordine descritto dal
%   vettore p.
% Le matrici L,U,P sono tali che P*A=L*U
%
% See also SIMPLELU, QRHOUSE
function [x,A,p]=PLU(A,b)
n=length(b);
p=1:n; % vettore contenente la permutazione finale
for i=1:n-1
    [m,k]=max(abs(A(i:n,i)));
    % k e' l'indice del massimo elemento in modulo
    % la la sua posizione e' relativa alla dimensione
       del
    % vettore, quindi si deve aggiustare tale valore
    k=k+i-1;
    % esecuzione di vari swap su
    %%% righe matrice A
    tmp=A(i,:); A(i,:)=A(k,:); A(k,:)=tmp;
    %%% vettore dei termini noti
    tmp=b(i); b(i)=b(k); b(k)=tmp;
    %%% vettore permutazioni
    tmp=p(i); p(i)=p(k); p(k)=tmp;
    for j=i+1:n
       A(j,i)=A(j,i)/A(i,i);
       A(j,i+1:n)=A(j,i+1:n)-A(j,i)*A(i,i+1:n);
    end;
end;
L=tril(A,-1);
for c=1:n
     A(c,c)=1;
end
U=triu(A);
y=solveL(L,b);
x=solveU(U,y);
return
```

ESEMPIO di plu.m

```
>> A

A =

     1     2     3     4
     5     6     7     8
     9    10    32   354
    65    78    98    54

>> b

b =

     1
     2
    54
     7

>> plu(A,b)

ans =

  -1.20731707317072
   2.38414634146340
  -1.14634146341463
   0.21951219512195

>> inv(A)*b

ans =

  -1.20731707317073
   2.38414634146343
  -1.14634146341463
   0.21951219512195
```

1.2.2 Fattorizzazione QR

Lo scopo della fattorizzazione QR è quello di fattorizzare una matrice A nel prodotto di una matrice Q ortogonale e di una R triangolare superiore mediante un metodo analogo al precedente nel quale ad ogni passo si costruiscono delle matrici ortogonali dette matrici di Householder. Sia $P = I - 2\frac{vv^T}{v^Tv}$, $(v \neq 0)$ $P = P^T$ ed inoltre $P^2 = P^TP = I$ cioè è ortogonale. Considero $x \in R^n$, $x \neq 0$ impongo che Px sia un multiplo di e_1 cioè $Px = \alpha e_1$ con $\alpha \in R$ segue che

$$\big|\big|Px\big|\big|_2^2 = \big|\big|\alpha e_1\big|\big|_2^2 = \alpha^2\big|\big|e_1\big|\big|_2^2 = \alpha^2 = \big|\big|x\big|\big|_2^2 \rightarrow \alpha = \pm\big|\big|x\big|\big|_2$$

l' ultima uguaglianza vale perché P è una matrice ortogonale e moltiplicate un vettore per una matrice ortogonale non influisce sulla norma. Sia adesso $v = x + \beta e_1$ allora

$$Px = x(1 - 2\frac{v^Tx}{v^Tv}) - \beta 2\frac{v^Tx}{v^Tv}e_1$$

$2\frac{v^Tx}{v^Tv} = 1$ quindi $Px = -\beta e_1 = \alpha e_1 = \pm\big|\big|x\big|\big|$ per quanto riguarda il segno di β sia che il segno $+$ che il segno - vanno bene, però ci sino sue motivi che ci spingono a scegliere con attenzione.
Primo motivo: si deve notare che $v = x+\beta e_1$ e ci ricordiamo che la somma è un'operazione ben definita se i termini sono dello stesso segno

$$v = x + \beta e_1 = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} \beta \\ 0 \\ \dots \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 + \beta \\ x_2 \\ \dots \\ x_n \end{pmatrix}$$

quindi scegliamo come segno di β il segno di x_1.
Secondo motivo: se $x \neq \underline{0}$ allora la prima componente di v è non nulla e quindi possiamo dividere l'intero vettore per tale componente, in questo modo la prima componente diventa 1 e non è necessario memorizzarla. La struttura del vettore v viene alterata, ma fortunatamente non quelle della matrice P (infatti la matrice P determinata da v è la stessa determinata da αv.

Vettore di Householder

$\underline{v}$=house($\underline{x}$) - *Calcola il vettore di Householder associato al vettore $\underline{x}$ scalato secondo la prima componente. Ovviamente se $\underline{x} = \underline{0}$ restituisce $\underline{0}$*

CODICE: house.m

```
%HOUSE
%v=house(x)
%Pre: x vettore non nullo
% Questa funzione restituisce in v il vettore che
    genera la
% matrice di householder
% P=I -2*(vv')/(v'v) tale che P*x=a*e1 dove aè una
    costante
% reale ed e1è il
% primo vettore della base canonica.
%
% See also QRHOUSE
function v=house(x)
n=length(x);
nx=norm(x);
if nx==0
    v=zeros(n,1);
elseif x(1)>0
    temp=x(1)+nx;
    v(1)=-nx;
    v(2:n)=x(2:n)/temp;
else
    temp=x(1)-nx;
    v(1)=nx;
    v(2:n)=x(2:n)/temp;
end

return
```

ESEMPIO di house.m

Matrici di Householder

L' algoritmo parte con

$$A = \begin{pmatrix} a^0_{1\,1} & \dots & \dots & a^0_{1\,n} \\ a^0_{2\,1} & \dots & \dots & a^0_{2\,n} \\ \vdots & \dots\dots & & \vdots \\ a^0_{n\,1} & \dots & \dots & a^0_{n\,n} \end{pmatrix} = A^0$$

e al primo passo procede premoltiplicando a sinistra la matrice A per una matrice P^1 in modo che la prima colonna di A sia un multiplo di e_1, $\det(A) \neq 0 \;\rightarrow\; Ae_1 \neq \underline{0}$ ed esiste $v_1 \in R^n$ tale che

$$P^1 A e_1 = (I_n - 2\frac{v_1 v_1^T}{v_1^T v_1}) \begin{pmatrix} a^0_{1\,1} \\ a^0_{2\,1} \\ \vdots \\ a^0_{n\,1} \end{pmatrix} = \begin{pmatrix} a^1_{1\,1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

e questo v_1 è appunto il vettore di *Householder* o di riflessione associato alla prima colonna della matrice A. Dopo questa moltiplicazione la situazione diventa la seguente:

$$P^1 A^0 = A^1 = \begin{pmatrix} a^1_{1\,1} & a^1_{1\,2} & \dots & \dots & a^1_{1\,n} \\ 0 & a^1_{2\,2} & \dots & \dots & a^1_{2\,n} \\ \vdots & \vdots & \dots & \dots & \vdots \\ \vdots & \vdots & \dots\dots & & \vdots \\ 0 & a^1_{n\,2} & \dots & \dots & a^1_{n\,n} \end{pmatrix}$$

dove si nota che, a differenza della fattorizzazione LU, al primo passo anche la prima riga viene alterata.
Il secondo passo inizia considerando la matrice A^1 come triangolare a blocchi

$$A^1 = \left(\begin{array}{c|ccc} a^1_{1\,1} & a^1_{1\,2} & \dots & a^1_{1\,n} \\ \hline 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{array}\right)$$

con A_1 non singolare, ed esiste una matrice di riflessione P_2 che premoltiplicata ad A_1 trasforma la sua prima colonna in un multiplo di e_1 (con $n-1$

componenti); questa matrice non può essere moltiplicata per A^1 per incompatibilità di dimensione, quindi è necessario trovare una matrice ortogonale che sia moltiplicabile per A^1, che trasformi la prima colonna di A_1 nel modo desiderato e che non alteri il resto della matrice. Questa matrice esiste ed è definita come

$$P^2 = \left(\begin{array}{c|c} 1 & \bigcirc^T \\ \hline \bigcirc & \\ & P_2 A_1 \\ & \end{array}\right)$$

Gli altri passi sono identici, via via si lavora su sottomatrici più piccole tutte non singolari, si trova una matrice di riflessione opportuna e la si "inserisce" moltiplicata per la matrice corrente dentro una matrice più grande con tutti 1 sulla diagonale. In questo modo si ottiene $P^{n-1} \cdots P^1 A = R$ dove tutte le P sono ortogonali ed il loro pordotto vale Q^T, quindi $Q^T A = R \ \rightarrow \ A = QR$.

[$\underline{x}$,A]=QRhouse(A,$\underline{b}$) - *Data la matrice A ed il vettore $\underline{b}$ la funzione fattorizza A come il prodotto di due matrici QR con Q ortogonale ed R triangolare superiore, secondo il metodo di Householder e risolve il sistema $R\underline{x} = Q^T\underline{b}$. La matrice R viene memorizzata nella parte triangolare superiore di A, ed al posto della Q vengono memorizzate nella parte inferiore le parti significative dei vettori di Householder associati a ciascuna colonna della matrice originaria.*

CODICE: QRHouse.m

```
%QRHOUSE
%[x,A]=QRHOUSE(A,b)
%
%Pre: Aè una matrice quadrata non singolare n x n;
% b vettore di lunghezza n
%
% La procedura fornisce la fattorizzazione QR di una
    matrice
% A e risolve
%  il sistema lineare Ax=b; Qè una matrice
% ortogonale (Q'Q=I) e Rè
```

```
% triangolare superiore.
% In x viene restituita la soluzione del sistema
   lineare mentre
% in A viene
%restituita una matrice che contiene nella sua
% parte triangolare superiore
% la matrice R e nella parte
% strettamente triangolare inferiore le componenti
%  essenziali
% dei vettori di householder che generano la matrice Q
% See also HOUSE
function [x,A]=QRhouse(A,b)
n=length(b);
% beta = 2/(v' * v) con v vettore di houseolder
beta=zeros(1,n-1);
gamma=zeros(1,n);

for i=1:n-1
    % adesso il vettore v si trova in
    % A(i:n,i)
    A(i:n,i)=house(A(i:n,i))';
    beta(i)=2/([1;A(i+1:n,i)]'*[1;A(i+1:n,i)]);
    % calcola v'*y con y vettore colonna corrente
    for j=i+1:n
        gamma(j)=([1;A(i+1:n,i)]'*A(i:n,j));
    end
    % aggiorna la riga i-esima
    for t=i+1:n
        A(i,t)=A(i,t)-gamma(t)*beta(i);
    end
    % viene aggiornata la sottomatrice in
    % esame per colonne, moltiplicando per
    % una matrice di Householder
    for j=i+1:n
        % prodotto vettoriale con fattore moltiplicativo
        % aggiuntivo -gamma(k)*beta(i)
        for k=i+1:n
            A(k,j)=A(k,j)-gamma(j)*A(k,i)*beta(i);
```

```
        end
    end
end
% aggiornamento termine noto
x=b(:);
for i=1:n-1
    temp=beta(i)*[1;A(i+1:n,i)]'*x(i:n);
    x(i:n)=x(i:n)-temp*[1;A(i+1:n,i)];
end
x=solveU(A,x);
return
```

ESEMPIO di QRHouse.m

```
>> A

A =

3.0000000000 1.0000000000 0.5000000000 0.8000000000
1.5000000000 9.0000000000 0.5000000000 0.6000000000
3.2000000000 4.0000000000 12.7000000000 3.2000000000
0.7000000000 3.2000000000 4.1000000000 11.3000000000

>> b

b =

   0.50000000000000
   1.20000000000000
   3.00000000000000
   6.20000000000000

>>  inv(A)*b

ans =

  -0.01092762955907
   0.09772671951632
```

```
   0.08447073456824
   0.49102600234596

>> QRHouse(A,b)

ans =

  -0.01092762955907
   0.09772671951632
   0.08447073456824
   0.49102600234596
```

1.2.3 Fattorizzazione LDL^T

Se una matrice è simmetrica e definita positiva allora è fattorizzabile LU, però è possibile sfruttare la proprietà di essere simmetrica per ottenere una fattorizzazione *ad hoc*. Sia A s.d.p. allora A=LU con

$$L = \begin{pmatrix} 1 & & & \\ & 1 & & \bigcirc \\ & & \ddots & \\ & \times & & 1 \end{pmatrix}, U = \begin{pmatrix} u_1 & & & \\ & u_2 & & \times \\ & \bigcirc & \ddots & \\ & & & u_n \end{pmatrix}$$

$$U = \begin{pmatrix} u_1 & & & \\ & u_2 & & \bigcirc \\ & \bigcirc & \ddots & \\ & & & u_n \end{pmatrix} \begin{pmatrix} 1 & & & \\ & 1 & & \times \\ & \bigcirc & \ddots & \\ & & & 1 \end{pmatrix} \equiv D\hat{U}$$

$A = LU$, $A = L\hat{U} = A^T = \hat{U}^T D L^T$ (D è diagonale), $LD\hat{U} = \hat{U}^T D L^T = A$; questa fattorizzazione è unica e implica che $L = \hat{U}^T$, $\hat{U} = L^T$ e quindi che $A = LU \rightarrow A = LDL^T$; vediamo un algoritmo efficiente per calcolare la L e la D. Dato che A è simmetrica possiamo lavorare solo sulla parte triangolare inferiore considerando solo la L e la D e non la L^T quindi possiamo lavorare

con gli indici i,j tale che $i \geq j$. L'elemento $a_{i\,j}$ è calcolato come $a_{i\,j} = e_i^T A e_j = e_i^T (LDL^T) e_j = (e_i^T L) D (L^T e_j) = (l_{i\,1} \dots l_{i\,i} 0 \dots 0) D (l_{j\,1} \dots l_{j\,j} 0 \dots 0) =$

$$= (l_{i\,1} \dots l_{i\,i} \underbrace{0 \dots 0}_{n-i}) \begin{pmatrix} d_1 & & & & \\ & d_2 & & & \\ & & d_3 & & \\ & & & \ddots & \\ & & & & d_n \end{pmatrix} (l_{j\,1} \dots l_{j\,j} \underbrace{0 \dots 0}_{n-j}) =$$

$$= \sum_{k=1}^{\min(i,j)} l_{i\,k} d_k l_{j\,k} = \sum_{k=1}^{j} l_{i\,k} d_k l_{j\,k}$$

adesso si presentano due casi

1. $i = j$
2. $i > j$

Nel primo caso

$$a_{j\,j} = \sum_{k=1}^{j} l_{j\,k}^2 d_k = \sum_{k=1}^{j-1} l_{j\,k}^2 d_k + l_{j\,j}^2 d_j =$$

$$= (l_{j\,j} = 1) \; = \sum_{k=1}^{j-1} l_{j\,k}^2 d_k + d_j$$

e quindi

$$d_j = a_{j\,j} - \sum_{k=1}^{j-1} l_{j\,k}^2 d_k$$

Nel secondo caso invece

$$a_{i\,j} = \sum_{k=1}^{j-1} l_{i\,k} d_k l_{j\,k} + l_{i\,j} + d_j + l_{j\,j}$$

ed essendo $l_{j\,j} = 1$ si ha che

$$l_{i\,j} = \frac{a_{i\,j} - \sum_{k=1}^{j-1} l_{i\,k} d_k l_{j\,k}}{d_j}$$

R=LD(A) - *Data la matrice a non singolare e simmetrica definita positiva, la funzione esegue la fattorizzazione LDL^T della matrice data e restituisce nella parte strettamente triangolare inferiore di R le componenti significative della matrice L, e nella diagonale restituisce la diagonale della matrice D*

CODICE: ld.m

```
%LD
%R=LD(A)
%Pre: Aè una matrice simmetrica definita positiva.
%
% La funzione restituisce in L una matrice che riassume
% la fattorizzazione LDL^T di A.
% Nella parte strettamente triangolare inferiore di R
   si
% trovano
  le componenti di L e la dagonale principaleè
% uguale a quella
 % della matrice diagonale D.
% Le matrici L e D sono tali che
% A=L*D*L'
%
% See also PLU, QRHOUSE
function R=LD(A)
n=length(A);
d=zeros(n);
eta=zeros(n,n);
R=zeros(n,n);
for j=1:n-1
     som=0;
     for k=1:1:j-1
        eta(j,k)=A(j,k)*d(k);
                  som=som+A(j,k)*eta(j,k);
     end
    d(j)=A(j,j)-som;
    for i=j+1:1:n
                  som=0;
```

```
                    for  k=1:1:j-1
                    som=som+A(i,k)*eta(j,k);
                    end
            som=A(i,j)-som;
            som=som/d(j);
            A(i,j)=som;
    end
end
R=tril(A,-1);
for  i=1:n
    R(i,i)=d(i);
end
return
```

ESEMPIO di ld.m

1.3 Metodi iterativi

L'idea alla base di questi due metodi iterativi è quella di "splittare" la matrice A in $M - N$ con $\det(M) \neq 0$, quindi

$$(M - N)x = b, Mx = Nx + b \rightarrow x = M^{-1}Nx + M^{-1}b$$

. Ovviamente non viene calcolata M^{-1} poiché questa operazione è troppo costosa viene utilzzata la seguente formula: [1]

$$x^{k+1} = M^{-1}Nx^k + M^{-1}b$$
$$Mx^{k+1} = Nx^k + b$$

ed M deve essere di tipo semplice. Affinché il metodo converga ($x^{k+1} \rightarrow \overline{x}$ soluzione) è necessario che $(M^{-1}N)^{k+1}$ tenda a zero, perché $x^{k+1} - \overline{x} = (M^{-1}N)^{k+1}(x^0 - \overline{x})$. Da notare che a differenza delle fattorizzazioni, nei metodi iterativi compare anche l'errore di troncamento, in quanto la convergenza alla soluzione è di tipo asintotico e quindi è necessario troncare la procedura ad un certo punto.

[1] In questa sezione indicheremo con x^k il vettore al passo k-esimo, memtre con M^k (M matrice) indicheremo l'elevamento a potenza della una matrice M

1.3.1 Metodo di Jacobi

Nel metodo di Jacobi $A = L+D+U$ con L strettamente triangolare inferiore, D diagonale ed U strettamente triangolare superiore

$$\begin{cases} M &= D \\ N &= -L-U \end{cases} \quad \det(M) \neq 0 \leftrightarrow a_{i\,i} \neq 0 \; \forall i = 1 \ldots n$$

visto il precedente schema dei metodi iterativi si ha che

$$Dx^{k+1} = -(L+U)x^k + b$$

e procedendo riga per riga

$$a_{i\,i}x_i^{k+1} = b_i - \sum_{j=1, j\neq i}^{j=n} a_{i\,j}x_j^k$$

da cui si ricava

$$x_i^{k+1} = \frac{b_i - \sum_{j=1,j\neq i}^{j=n} a_{i\,j}x_j^k}{a_{i\,i}}$$

---

x=Jacobi(A,b,x_0,epsilon) - *Data la matrice A non singolare ed il vettore* $\underline{b}$ *la funzione risolve il sistema lineare* $A\underline{x} = \underline{b}$ *mediante il metodo iterativo di Jacobi partendo dal vettore di innesco* $\underline{x}_0$ *con una tolleranza epsilon; la condizione di arresto è basata sul metodo del residuo.*

CODICE: Jacobi.m

```
%JACOBI
% x=Jacobi(A,b,x0,epsilon)
% Pre: A non singolare, A(i,i) non nulli per i=1:n.
%      In realta' perche' il problema sia ben
   condizionato,
%      A deve
% essere  a diagonale dominante.
% La funzione restituisce in x la soluzione del sistema
    Ax=b,
% risolto con il metodo semiiterativo di Jacobi.
```

```
% Riceve in x0 il vettore di innesco e in epsilon la
    tolleranza
% desiderata.
%
% See also GAUSS_SEIDEL, DD
function x=Jacobi(A,b,x0,epsilon)
n=length(b);
x=x0;
r=b-A*x;

while norm(r)>epsilon
*norm(b)
    for i=1:n
        som=0;
        for j=1:n
            if i~=j
                som=som+A(i,j)*x(j);
            end
        end
        som=(b(i)-som)/A(i,i);
        x(i)=som;
    end
    r=b-A*x;
end
return
```

ESEMPIO di Jacobi.m

```
A =

     9     6     2
     3   -12     4
     7    15    -3

ż b

b =
```

```
     1
     2
     4

ż Jacobi(A,b,[1;2;3],1E-9)

ans =

   1.00000000008960
  -0.62500000007290
  -2.12500000015545

ż inv(A)*b

ans =

   1.00000000000000
  -0.62500000000000
  -2.12500000000000


ż A

A =

3.0000000000 1.0000000000 0.5000000000 0.8000000000
1.5000000000 9.0000000000 0.5000000000 0.6000000000
3.2000000000 4.0000000000 12.7000000000 3.2000000000
0.7000000000 3.2000000000 4.1000000000 11.3000000000

ż b

b =

   0.50000000000000
   1.20000000000000
   3.00000000000000
   6.20000000000000
```

```
ż  inv(A)*b

ans =

  -0.01092762955907
   0.09772671951632
   0.08447073456824
   0.49102600234596

ż Jacobi (A,b,[0.01;0.09;0.1;1],1E-9)

ans =

  -0.01092762960712
   0.09772671950406
   0.08447073461875
   0.49102600233408
```

1.3.2 Metodo di Gauss-Seidel

Il metodo di Gauss-Seidel invece considera M=L+D N=-U ed ancora una volta si ha che $\det(M) \neq 0 \iff a_{i:i} \neq 0 \; \forall i$.
Lo schema del metodo di Gauss-Seidel è il seguente:

$$(L + D)x^{k+1} = (-U)x^k + b$$

e ricaviamo x^{k+1} nel seguente modo

$$\sum_{j=1}^{i} a_{i\,j} x_j^{k+1} = b_i - \sum_{j=i+1}^{n} a_{i\,j} x_j^k$$

$$\sum_{j=1}^{i-1} a_{i\,j} x_i^{k+1} + a_{i\,i} x_j^{k+1} = b_i - \sum_{j=i+1}^{n} a_{i\,j} x_j^k$$

$$x_i^{k+1} = \frac{b_i - \sum_{j=i+1}^{n} a_{i\,j} x_j^k - \sum_{j=1}^{i-1} a_{i\,j} x_j^{k+1}}{a_{i\,i}}$$

[x,passi]=GaussSeidel(A,b,x_0,epsilon,upper) - *Data la matrice A non singolare ed il vettore $\underline{b}$ la funzione risolve il sistema lineare $A\underline{x} = \underline{b}$ mediante il metodo iterativo di Gauss-Seidel partendo dal vettore di innesco $\underline{x}_0$ con una tolleranza epsilon; la condizione per l'arresto è basata sia sul criterio del residuo, sia sul numero massimo di iterazioni upper*

CODICE: GaussSeidel.m

```
%GAUSS_SEIDEL
% function [x,passi]=Gauss_Seidel(A,b,x0,epsilon,upper)
% Pre:  A non singolare. Perche' il problema sia ben
%       condizionato dev'essere A a diagonale dominante
%
% La function risolve il sistema lineare Ax=b con il
   metodo
% di Gauss Seidel.
% Riceve in x0 il vettore di innesco, in epsilon l'
   accuratezza
% desiderata, in  upper un limite superiore sul numero
   di
% passi da eseguire.
% Restituisce in x la soluzione del sistema e in count
   il numero
% di passi che sono  stati effettivamente necessari per
    arrivare
% alla soluzione. Se passi=upper si consiglia  di
   riprovare con
%un valore di upper piu' alto.
%
%See also DD, JACOBI

function [x,count]=GaussSeidel(A,b,x0,epsilon,upper)
n=length(b);
x=x0;
r=b-A*x;
count=0;
while norm(r)>=epsilon*norm(b);
```

```
    x_prec=x;
    for i=1:
        x(i)=(b(i)-A(i,i+1:n)*x(i+1:n)-A(i,1:i-1)*x(1:i
            -1))/A(i,i);
    end
    count=count+1;
    if ((norm(x-x_prec)<epsilon) | (count>=upper))
        return;
    end
    r=b-A*x;
end
return;
```

ESEMPIO di GaussSeidel.m

Capitolo 2

Risoluzione di equazioni non lineari

2.1 Introduzione

Il problema che ci poniamo in questo capitolo è il seguente: data una funzione

$$f : I \subseteq \mathbf{R} \to \mathbf{R}$$

cerchiamo le sue *radici*, ossia quei punti $\overline{x} \in \mathbf{R}$ tali che $f(\overline{x}) = 0$.

In questo caso, diversamente dai sistemi lineari, non abbiamo un criterio per decidere l'esistenza e l'unicità. Infatti si possono avere situazioni molto variegate: ad esempio le funzioni $f(x) = x^2 + 1$, $g(x) = x^2 - 1$, $h(x) = \sin(x)$ hanno rispettivamente 0, 2 e infinite soluzioni reali.

Quindi ci accontenteremo di determinare solo una di queste radici (se esiste).

2.2 Metodo di bisezione

Supponiamo che f sia una funzione continua nell'intervallo $[a, b]$ tale che $f(a)f(b) < 0$ cioè la funzione assume valori di segno opposto nei due estremi. Allora per il *teorema dell'esistenza degli zeri* $\exists c : a < c < b, f(c) = 0$.

Definiamo $x_1 = \frac{a+b}{2}$ allora si può avere uno dei seguenti tre casi:

1. $f(x_1) = 0$ allora abbiamo avuto fortuna: x_1 è una radice e l'algoritmo termina.

2. $f(a)f(x_1) < 0$ allora possiamo reiterare il procedimento sul nuovo intervallo $[a, x_1]$.

3. $f(b)f(x_1) < 0$ allora possiamo reiterare il procedimento sul nuovo intervallo $[x_1, b]$.

In un passo abbiamo in questo modo dimezzato l'ampiezza dell'intervallo di confidenza, quindi reiterando questo metodo è possibile trovare un intervallo di confidenza di ampiezza sempre più piccola.

Poiché $x_1 = \frac{a+b}{2}$ si ha $|\overline{x} - x_1| < \frac{b-a}{2}$. In generale $|\overline{x} - x_i| < \frac{b_i - a_i}{2}$. Sviluppando questa relazione si ottiene

$$\begin{aligned} |\overline{x} - x_i| < \frac{b-a}{2^i} &\leq \epsilon \\ b - a &\leq \epsilon 2^i \\ \frac{b-a}{\epsilon} &\leq 2^i \\ \log_2(b-a) - \log_2(\epsilon) &\leq i \\ i \geq \lceil \log_2(b-a) - \log_2(\epsilon) \rceil &= \nu \end{aligned} \tag{2.1}$$

In questo modo, fissata la accuratezza ϵ, viene fissato anche il numero massimo di iterazioni, ν. Nella nostra implementazione, epsilon viene dato dall'utente e ν viene calcolato secondo la formula (2.1) e restituito in nu.

Un altro problema che si presenta nell'implementazione dell'algoritmo è la condizione $f(x_i) = 0$ che è difficilmente verificabile in aritmetica finita e dev'essere più realisticamente sostituita con una condizione del tipo $|f(x_i)| < \delta$ dove δ soddisfa

$$\delta = \delta(\epsilon) \text{ tale che } |f(x_i)| < \delta \Longrightarrow |\overline{x} - x_i| < \epsilon \tag{2.2}$$

Tale δ risulta essere

$$\delta \approx \epsilon |f'(x_i)| \tag{2.3}$$

Tuttavia calcolare esplicitamente $f'(x_i)$ sarebbe più costoso del metodo di bisezione stesso. Quello che possiamo fare è approssimare $f'(x_i)$ con il rapporto incrementale che ha come estremi l'intervallo corrente $[a_i, b_i]$:

$$\delta \approx \epsilon \left| \frac{f(b_i) - f(a_i)}{b_i - a_i} \right| \tag{2.4}$$

Nell'esempio vediamo l'applicazione della procedura di bisezione alla funzione $y = \sin(x)$. L'intervallo di confidenza iniziale è $[2; 4]$ (osserviamo che la

funzione assume valori di segno opposto. In questo caso il metodo converge alla soluzione $\overline{x} = \pi$ in 49 passi mentre il limite fissato con la formula (2.1) è 51. La funzione sin è molto regolare quindi si ha $\delta = \epsilon$.

Nel caso che la radice non sia semplice e quindi la funzione sia molto schiacciata (nell'esempio $y = (x-5)^5$) ad un ϵ piccolo corrisponde un δ abbastanza grande: nell'esempio ad $\epsilon = 10^{-5}$ corrisponde un δ dell'ordine di 10^{-1}.

[x,nu,delta,passi]=bisez(f,a,b,epsilon) - *Applica il metodo di bisezione alla funzione f nell'intervallo [a,b] con accuratezza epsilon; precondizioni> f continua in [a,b] e* $f(a) \cdot f(b) < 0$. *Restituisce in x la radice trovata, in nu il limite superiore al numero di passi da eseguire, in delta l'accuratezza sulle y in passi il numero di passi effettuati.*

CODICE: bisez.m

```
%BISEZ
%[x,nu,delta,i]=bisez(f,a,b,epsilon)
%dove
%- f e' il NOME di una funzione f
%- a e b sono gli estremi di un intervallo tale che f(a
    )*f(b)<0
%- epsilon e' l'accuratezza desiderata
%
% applica il metodo di bisezione e restituisce in x una
    radice
% di f(x)
% In nu viene resttituito il limite superiore
% al numero di iterazioni affinche'
% il metodo sia attendibile con
% accuratezza epsilon; in delta viene restituita
% l'accuratezza sulle y
% See also NEWTON
function [x,nu,delta,i]=Bisez(f,a,b,epsilon)
nu=ceil(log2(b-a)-log2(epsilon));
%si pone un limite superiore al numero
%di iterazioni, dipendente dalla accuratezza
%desiderata (epsilon) e dall'intervallo iniziale [a,b]
```

```
fa=feval(f,a); fb=feval(f,b);
for i=1:nu
    x=(a+b)/2;
    fx=feval(f,x);
    delta=epsilon*abs((fb-fa)/(b-a));

    if abs(fx)<delta
        return
    elseif fa*fx<0
        b=x;
        fb=fx;
    else
        a=x;
        fa=fx;
    end
end
return
```

ESEMPIO di bisez.m

```
>> sin (2)

ans =

   0.90929742682568

>> sin(4)

ans =

  -0.75680249530793

>> [x,nu,delta,passi]=bisez('sin',2,4,1E-15)

x =

   3.14159265358979
```

```
nu =

    51


delta =

    1.000000000000000e-015


passi =

    49

>> pi

ans =

   3.14159265358979


>> type fm

function y=fm(x)
y=(x-5)^5;

>> [x,nu,delta,passi]=bisez('f_e',4,7,1E-5)

x =

   6.99999427795410


nu =

    19
```

```
delta =

   0.01096626883467


passi =

    19
```

2.3 Metodo di Newton

Consideriamo f di classe $C^{(1)}$, l'idea di base del metodo di Newton consiste nel partire da un punto iniziale x_0, trovare l'intersezione tra la retta tangente alla f in $(x_0, f(x_0))$ e l'asse delle ascisse e prendere questo punto come x_1 (approssimazione della radice al passo 1) e quindi reiterare il metodo che in forma compatta diventa

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad i = 0 \ldots \tag{2.5}$$

Il metodo per radici semplici ha convergenza quadratica e per radici multiple lineare, si tratta tuttavia di convergenza locale e questo é il principale svantaggio nell'utilizzo di questo metodo in luogo di quello di bisezione. Il punto di innesco x_0 deve essere scelto perciò in un opportuno intorno della radice (sconosciuta).

La procedura riceve in input la funzione della quale trovare la radice e la sua derivata, che deve quindi essere conosciuta per via analitica. Un approccio alternativo potrebbe essere quello di approssimare la derivata con un rapporto incrementale con incremento prossimo allo zero ma in questo caso sarebbe difficile determinare quanto piccolo debba essere l'incremento per ottenere un risultato sufficientemente accurato.

[x,passi]=Newton(f,df,x0,epsilon,upper) - *Applica il metodo di Newton alla funzione f, la cui derivata è df, con punto iniziale x_0, accuratezza epsilon e numero massimo di passi upper. Restituisce in x la radice (se trovata) e in*

passi, il numero di passi effettuati.

CODICE: Newton.m

```
%NEWTON
%[x,passi]=newton(f,df,x0,epsilon,upper)
%  Pre: f derivabile
%  Applica il metodo di Newton per il calcolo della
%  radice dell'equazione f(x)=0
%  f       - funzione
%  df      - derivata prima di f
%  x0      - punto di innesco
%  epsilon - tolleranza per la radice
%  upper   - numero massimo di iterazioni consentite
%
%  Restituisce in x la radice trovata e in passi il
   numero
%  di iterazioni che sono state necessarie per
   reggiungerla
%  See also BISEZ,  CORDE, SECANTI.

function [x,passi]=newton(f,df,x0,epsilon,upper)
count=0;
f0=feval(f,x0);
d=feval(df,x0);

while(count<upper) & (abs(f0)>abs(epsilon*d))
          count=count+1;
          d=feval(df,x0);
     x0=x0-(f0/d);
     f0=feval(f,x0);
end
x=x0;
passi=count;
return
```

ESEMPIO di Newton.m

```
>> type f_e

function y=f_e(x)
y=exp(x)-1;


>> [x,passi]=Newton('f_e','exp',4,1E-19,200)

x =

    1.692401153459279e-017


passi =

    10
```

Nella figura 2.1 vediamo un esempio del metodo di Newton applicato alla funzione $y = e^x - 1$ con punto di partenza $x_0 = 4$. Il metodo di Newton trova la radice $\overline{x} = 0$ con un errore dell'ordine di 10^{-17} in 10 passi. Il procedimento è illustrato in figura 2.1

2.4 Metodo di accelerazione di Aitken

Questo metodo serve per ripristinare la convergenza quadratica del metodo di Newton nel caso di radici multiple con molteplicitá sconosciuta; nel caso di molteplicità nota r si deve utilizzare questa variante del metodo precedente

$$x_{i+1} = x_i - r \cdot \frac{f(x_i)}{f'(x_i)} \quad i = 0 \ldots \tag{2.6}$$

Si può ossevare che questo metodo converge in una sola iterata se la funzione è del tipo $y = \alpha(x - \overline{x})^r$, infatti

$$x_{n+1} = x_n - \frac{r\alpha(x_n - \overline{x})^r}{r\alpha(x_n - \overline{x})^{r+1}} = x_n - x_x + \overline{x} = \overline{x}$$

[x,passi]=Newtonmolt(f,df,x0,r,epsilon,upper) - *Applica il metodo di*

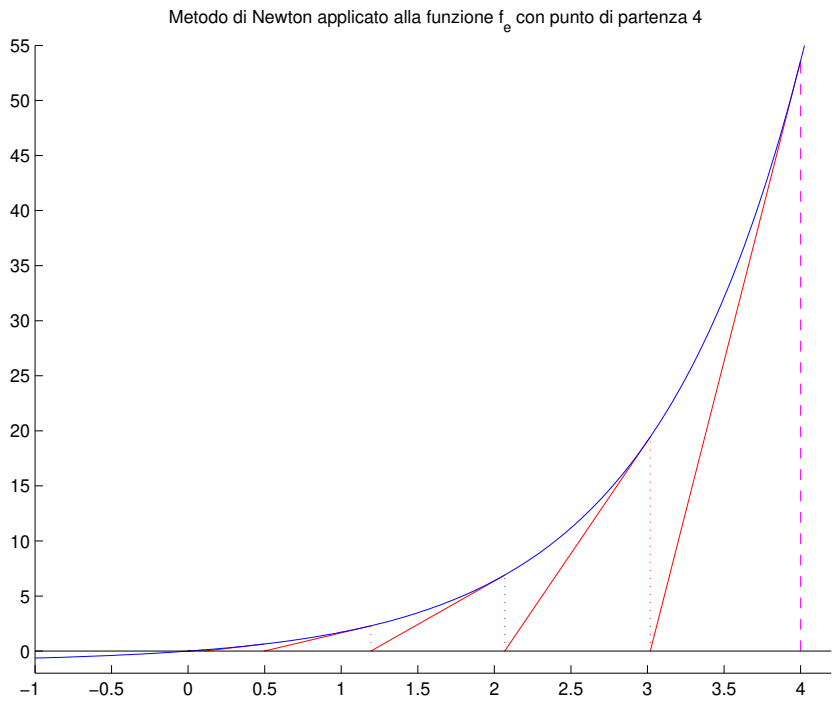

Figura 2.1: Metodo di Newton applicato alla funzione $y = e^x - 1$ con punto di partenza $x_0 = 4$. Converge alla radice $\overline{x} = 0$ in 10 passi.

Newton alla funzione f, la cui derivata è df, con punto iniziale x_0, molteplicità della radice r accuratezza epsilon e numero massimo di passi upper. Restituisce in x la radice (se trovata) e in passi, il numero di passi effettuati.

CODICE: Newtonmolt.m

```
%NEWTONMOLT
%[x,passi]=Newtonmolt(f,df,x0,r,epsilon,upper)
%  Pre: f derivabile
%  Applica il metodo di Newton per il calcolo della
%  radice (di molteplicita' nota r) dell'equazione f(x
   )=0
%  f          - funzione
%  df         - derivata prima di f
```

```
%  x0       - punto di innesco
%  r        - molteplicita' della radice
%  epsilon - tolleranza per la radice
%  upper    - numero massimo di iterazioni consentite
%
%  Restituisce in x la radice trovata e in passi il
   numero
%  di iterazioni che sono state necessarie per
   reggiungerla
%  See also BISEZ, CORDE, SECANTI.

function [x,passi]=Newtonmolt(f,df,x0,r,epsilon,upper)
count=0;
f0=feval(f,x0);
d=feval(df,x0);

while(count<upper) & (abs(f0)>abs(epsilon*d))
        count=count+1;
        d=feval(df,x0);
    x0=x0-r*(f0/d);
    f0=feval(f,x0);
end
x=x0;
passi=count;
return
```

ESEMPIO di Newtonmolt.m

```
function y=fm(x)
y=(x-5)^5;

>> type dfm

function y=dfm(x)
y=5*(x-5)^4;

>> [x,passi]=Newton('fm','dfm',6,1E-5,200)
```

```
x =

   5.00010633823966


passi =

    41

>> [x,passi]=Newtonmolt('fm','dfm',6,5,1E-5,200)

x =

     5


passi =

     1
```

Il metodo di accelerazione di Aitken, invece, utilizza questa approssimazione per l'approssimazione $\overline{x_n}$, note due approssimazioni precedenti che convergono linearmente, x_{n-1} e x_{n-2}:

$$\overline{x_n} \approx \frac{x_n^2 - x_{n-1}x_{n+1}}{2x_n - x_{n+1} - x_{n-1}} \tag{2.7}$$

In pratica il metodo di accelerazione si utilizza nel seguente modo:

- Si parte con x_0 e si generano x_1 e x_2 con il metodo di Newton.
- Con questi 3 punti si puó costruire $\overline{x_1}$ sfruttando la formula (2.7).
- Si pone $x_0 = \overline{x_1}$ e si generano x_1 e x_2
- Con questi nuovi valori si può costruire l'ulteriore approssimazione $\overline{x_2}$.
- Il procedimento si ripete iterativamente.

Come si vede dall'esempio, questo metodo è molto efficace nel caso di radici multiple. Applicandolo alla funzione $y = (x-5)^4$ trova la radice con una accuratezza pari a 10^{-14} in 3 passi contro i 134 del metodo di Newton.

[x,passi]=Aitken(f,df,x0,epsilon,upper) - *Applica il metodo di Newton con accelerazione di Aitken alla funzione f (la cui derivata è df) con punto iniziale x_0, accuratezza epsilon e numero massimo di passi upper. Restituisce in x la radice (se trovata) e in passi, il numero di passi effettuati.*

CODICE: Aitken.m

```
%AITKEN
%function [x,passi]=Aitken(f,df,x0,epsilon,upper)
%
% Trova la radice della funzione f con derivata df
% applicando il metodo di accelerazione di Aitken
% con punto iniziale x0, tolleranza epsilon e numero
% massimo di passi upper.
% Restituisce in x il valore della radice e in passi
% il numero di iterazioni eseguite.
function [x,passi]=Aitken(f,df,x0,epsilon,upper)

f0=feval(f,x0);
d=feval(df,x0);
x1=x0-(f0/d);
f1=feval(f,x1);

d=feval(df,x1);
x2=x1-(f1/d);
f2=feval(f,x2);

count=0;

while (count<upper) & abs(f2)>abs(epsilon*d)
    count=count+1;
    x0=(x1*x1-x0*x2)/(2*x1-x2-x0);
    f0=feval(f,x0);
    d=feval(df,x0);
```

```
    x1=x0-(f0/d);
    f1=feval(f,x1);
    d=feval(df,x1);
    x2=x1-(f1/d);
    f2=feval(f,x2);
end
x=x2;
passi=count+2;
return
```

ESEMPIO di Aitken.m

```
>> type fm

y=(x-5)^5;

>> type dfm

function y=dfm(x)
y=5*(x-5)^4;

>> [x,passi]=Newton('fm','dfm',6,1E-14,300)

x =

   5.00000000000010


passi =

   134

>> [x,passi]=Aitken('fm','dfm',6,1E-14,300)

x =

   4.99999999999993
```

```
passi =

     3
```

2.5 Metodo delle corde

Se ci troviamo abbastanza vicino alla soluzione $\overline{x}$ allora la pendenza della retta tangente il grafico della funzione non cambia di molto e quindi f' ci dice in qualche modo in che direzione dobbiamo guardare per trovare la soluzione. Quindi il metodo delle corde, calcola $f'(x_0)$ e lascia invariato questo valore usandolo al posto di $f'(x_i) \quad i = 1 \ldots$. Quindi il metodo diventa

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_0)}$$

La funzione ha comunque bisogno di conoscere f e f'. La valutazione di f' avverrà solo al primo passo.

Il costo del metodo delle corde è assimilabile a quello di bisezione perché ad ogni passo l'operazione predominante è una valutazione della f; il metodo è convergente di ordine 1, comunque scegliendo il punto di innesco adeguatamente si ottiene una convergenza più veloce. Nell'esempio tuttavia osserviamo che il metodo delle corde impiega 1978 iterazioni a trovare la radice della funzione $y = e^x - 1$ partendo da $x_0 = 4$ contro i 10 passi del metodo di Newton.

[x,passi]=corde(f,df,x0,epsilon,upper) - *Applica il metodo delle corde alla funzione f (la cui derivata è df) con punto iniziale x_0, accuratezza epsilon e numero massimo di passi upper. Restituisce in x la radice (se trovata) e in passi, il numero di passi effettuati.*

CODICE: corde.m

```
%CORDE
```

```
%[x,passi]=corde(f,df,x0,epsilon,upper)
%  Pre: f derivabile
%  Applica il metodo delle corde per il calcolo della
%  radice dell'equazione f(x)=0
%  f       - funzione
%  df      - derivata di f
%  x0      - punto di innesco
%  epsilon - tolleranza per la radice
%  upper   - numero massimo di iterazioni consentite
%
%  Restituisce in x la radice trovata e in passi il
   numero
%  di iterazioni che sono state necessarie per
   reggiungerla
%  See also BISEZ, NEWTON, SECANTI.
function [x,passi]=corde(f,df,x0,epsilon,upper)
count=0;
d=feval(df,x0);
while(count<upper) & (abs(feval(f,x0))>epsilon)
   count=count+1;
        x0=x0-(feval(f,x0)/d);
end
x=x0;
passi=count;
return
```

ESEMPIO di corde.m

```
>> type f_e

function y=f_e(x)
y=exp(x)-1;

>> [x,passi]=corde('f_e','exp',4,1E-19,10000)

x =

    1.097812158372279e-016
```

```
passi =

        1978
```

Nella figura 2.2 vediamo il grafico che illustra il comportamento del metodo delle corde applicato alla funzione $y = e^x - 1$ con punto di partenza $x = 2$. In questo caso il metodo ha bisogno di ben 250 iterazioni per convergere alla soluzione con un errore dell'ordine di 10^{-17}.

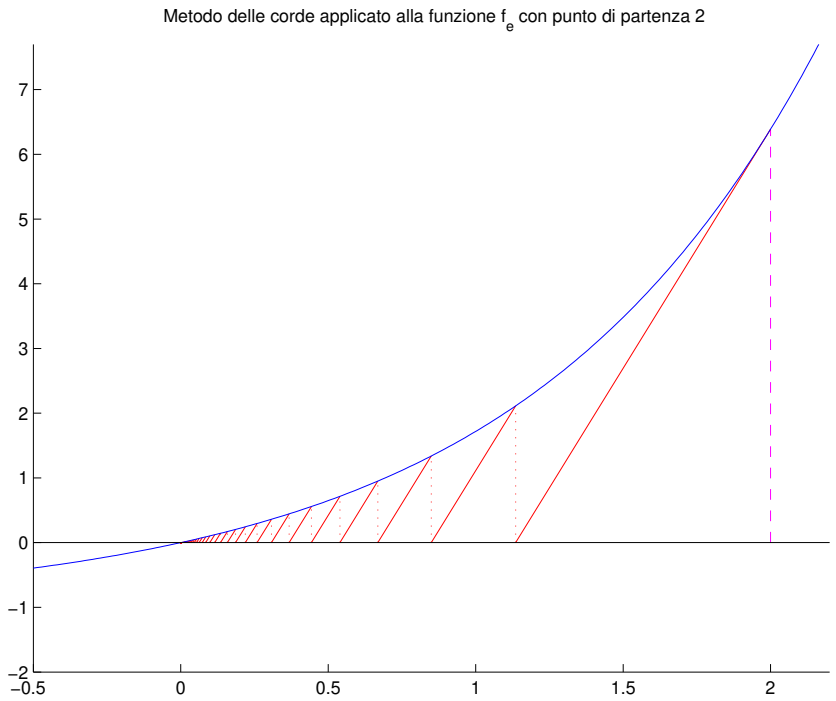

Figura 2.2: Metodo delle corde applicato alla funzione $y = e^x - 1$ con punto di partenza $x_0 = 2$. Converge alla radice $\overline{x} = 0$ in 250 passi.

2.6 Metodo delle secanti

Il metodo delle secanti è un "ibrido" tra il metodo di Newton ordinario e quello delle corde, poiché ad ogni passo né calcola la derivata prima della funzione, né tiene fissato un valore, ma approssima la derivata con la pendenza della retta secante il grafico della funzione passante per le due approssimazioni precedenti. In pratica x_1 viene calcolato con il metodo di Newton ordinario e successivamente il metodo diventa del tipo

$$x_{n+1} = \Phi(x_n, x_{n-1})$$

dove viene effettuata la seguente approssimazione

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

il secondo membro non è altro che il coefficiente angolare della retta secante il grafico passante per i punti $(x_n, f(x_n))$ e $(x_{n-1}, f(x_{n-1}))$; l'espressione del metodo ricapitolando diventa

$$x_{n+1} = x_n - \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

e il metodo ovviamente viene detto a passo multiplo.

$$\text{Ordine di convergenza} = \begin{cases} \frac{1+\sqrt{5}}{2} & \text{radici semplici} \\ 1 & \text{radici multiple} \end{cases}$$

Per la complessità valgono considerazioni analoghe al metodo della corde notando che $f(x_{n-1})$ è nota dal passo precedente e non deve quindi essere ricalcolata. Nell'esempio vediamo che il metodo delle secanti dà buoni risultati se applicato alla funzione $y = e^x - 1$ infatti converge alla radice in 13 passi.

[x,passi]=secanti(f,df,x0,epsilon,upper) - *Applica il metodo delle secanti alla funzione f (con derivata df) con punto iniziale x_0, accuratezza epsilon e numero massimo di passi upper. Restituisce in x la radice (se trovata) e in passi, il numero di passi effettuati.*

CODICE: secanti.m

```
%SECANTI
%[x,passi]=secanti(f,x0,epsilon,upper)
%  Pre: f derivabile
%  Applica il metodo delle secanti per il calcolo della
%  radice dell'equazione f(x)=0
%  f       - funzione
%  x0      - punto di innesco
%  epsilon - tolleranza per la radice
%  upper   - numero massimo di iterazioni consentite
%
%  Restituisce in x la radice trovata e in passi il
   numero
%  di iterazioni che sono state necessarie per
   reggiungerla
%  See also BISEZ, NEWTON, CORDE.

function [x,passi]=secanti(f,df,x0,epsilon,upper)
count=0;
x_old=x0;
y_old=feval(f,x_old);
d=feval(df,x0);
x_curr=x0-y_old/d;
y_curr=feval(f,x_curr);

while(count<upper) & (abs(y_curr)>epsilon)
        count=count+1;
        tmp_x=x_curr;
        tmp_y=y_curr;

        x_curr=tmp_x -((tmp_y*(tmp_x - x_old))/(tmp_y -
            y_old));
        y_curr=feval(f,x_curr);

        x_old=tmp_x;
        y_old=tmp_y;
end
x=x_curr;
passi=count;
```

return

ESEMPIO di secanti.m

```
>> type f_e

function y=f_e(x)
y=exp(x)-1;


>> [x,passi]=secanti('f_e','exp',4,1E-19,300)

x =

    1.230367509482608e-017


passi =

    13
```

Nel grafico di figura 2.3 è illustrato il comportamento del metodo delle secanti applicato alla funzione $y = e^x - 1$ con punto di innesco $x = 3$. Il metodo converge alla radice con un errore dell'ordine di 10^{-17} in 10 passi. Nella figura è disegnata in colore verde la retta tangente in x_0 ottenuta dal metodo di Newton ordinario.

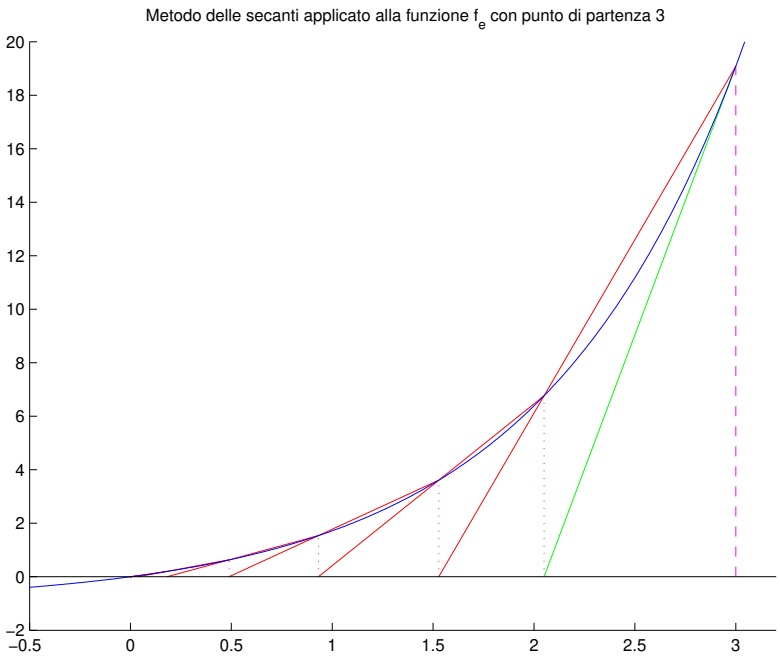

Figura 2.3: Metodo delle secanti applicato alla funzione $y = e^x - 1$ con punto di partenza $x_0 = 3$. Converge alla radice $\overline{x} = 0$ in 10 passi.

Capitolo 3

Interpolazione di funzioni

3.1 Interpolazione di funzioni mediante polinomi

Problema: data una funzione f ed $n+1$ ascisse distinte x_i trovare un polinomio di grado al più n che "interpola" la funzione nelle ascisse date. Più formalmente, date n+1 coppie (x_i, f_i) si cerca $p_n \quad \in \prod_n$ tale che $p_n(x_i) = f_i \; \forall i = 0 \ldots n$ dove con f_i indichiamo $f(x_i)$. Per determinare un polinomio di $\prod_n$ servono $n+1$ condizioni in quanto $\prod_n$ è uno spazio vettoriale di dimensione $n+1$ infatti

- $\forall p \in \prod_n \; p = \sum_{i=0}^{n} a_i x^i$
- i vari x^j sono linearmente indipendenti (principio di identità dei polinomi)

quindi i vari x^j sono una base.

3.2 Sistema di Vandermonde

Per trovare i coefficienti del polinomio basta risolvere un sistema lineare del tipo:

$$\begin{pmatrix} x_0^0 & \dots & x_0^n \\ x_1^0 & \dots & x_1^n \\ \dots & \dots & \dots \\ x_n^0 & \dots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}$$

la matrice contenente le x prende il nome di *matrice di Vandermonde* ed il numero di condizionamento associato alla matrice cresce più che esponenzialmente in n, quindi si cerca un approccio diverso alla risoluzione del problema. La costruzione di questa matrice ci permette comunque di concludere che tale polinomio interpolanate è unico perché la matrice è non singolare in quanto il suo determinante è $\prod_{j<i}(x_i - x_j)$ e noi abbiamo supposto che le varie x_i distinte.

V=Vandermonde(x) - *Costruisce la matrice di Vandermonde a partire dal vettore x delle ascisse di interpolazione, cioè contiene su ogni colonna le potenze successive degli elementi di x*

CODICE: Vandermonde.m

```
%VANDERMONDE
%V=Vandermonde(x)
%
% Restituisce la matrice di Vandermonde corrispondente
% al vettore x.
% See also VANDERINTERP, INTERPOLA
function V=Vandermonde(x)
n=length(x);
colonna=ones(n,1);
V=zeros(n,n);
tmp=x(:);
for i=1:n
    V(1:n,i)=colonna;
    colonna=colonna.*tmp;
end
return
```

---

p=VanderInterp(x,f) - *Restituisce i coefficienti del polinomio interpolante*

la f nelle ascisse contenute nel vettore x rispetto alla base canonica.

CODICE: VanderInterp.m

```
%VANDERINTERP
%p=VanderInterp(x,f)
%
% Calcola i coefficienti (nella base canonica) del
   polinomio
% interpolante i punti specificati

% nei vettori x (ascisse)
% e f (ordinate) risolvendo un sistema lineare.
% See also VANDERMONDE, INTERPOLA
function p=VanderInterp(x,f)
V=Vandermonde(x);
p=plu(V,f);
return
```

La procedura `VanderInterp` trova i coefficienti del polinomio costruendosi la matrice di Vandermonde V e risolvendo il sistema lineare $Vp = f$. Tuttavia *questo metodo dà luogo ad un sistema mal condizionato e pertanto la procedura VanderInterp non dà risultati attendibili.*

3.3 Interpolazione polinomiale nella base di Newton

Per ottenere un algoritmo efficiente per il calcolo del polinomio interpolante è necessario abbandonare la rappresentazione del polinomio interpolate nella base canonica ed adottare una base diversa per lo spazio $\prod_n$. Una possibile base è la base di Newton, così definita ricorsivamente:

$$\begin{cases} \omega_0(x) &= 1 \\ \omega_k(x) &= (x - x_{k-1})\,\omega_{k-1}(x) \end{cases}$$

In pratica si ha $\omega_k(x) = (x - x_0)(x - x_1)\ldots(x - x_{k-1})$ da cui segue che $\omega_k(x_j) = 0$ per $j < k$.

Si dimostra che $\{\omega_i(x)\}_{i=0\ldots n}$ è una base per $\prod_n$. Il polinomio interpolante di grado n interpolante una funzione può essere quindi espresso come

$$\sum_{i=0}^{n} c_i \omega_i(x)$$

I coefficienti sono dati dalle *differenze divise* così definite:

$$c_k = f[x_0, \ldots, x_k] = \sum_{i=0}^{k} \frac{f_i}{\prod_{j=0, j\neq i}^{k} (x_i - x_j)} \quad k = 0 \ldots n$$

Calcolo delle differenze divise

Il calcolo delle differenze divise segue un algoritmo fondato su due loro proprietà:

1. $f[x_i] = f_i$
2. $f[x_0, x_1, \ldots, x_{k+1}] = \frac{f[x_1,\ldots,x_{k+1}] - f[x_0,\ldots,x_k]}{x_{k+1} - x_0}$

La procedura per calcolare le differenze divise lavora su un vettore inizializzato a $(f_0, \ldots, f_n)$ e applica ripetutamente questa proprietà. Al primo passo si calcola $f[x_{n-1}, x_n]$ e lo si riscrive nel vettore in posizione n, e così via fino a $f[x_0, x_1]$ che viene salvato in posizione 2.

Alla fine del ciclo esterno il vettore conttiene $f[x_0], \ldots f[x_0, \ldots, x_k]$ che sono i coefficienti del polinomio interpolante.

---

v=Diffdiv(x,f) - *Calcola le differenze divise dato il vettore x delle ascisse di interpolazione e il vettore f dei corrispondenti valore della funzione da interpolare*

CODICE: DiffDiv.m

```
%DIFFDIV
%v=DiffDiv(x,f)
%
% La funzione calcola le differenze divise per le
    ascisse
```

```
% x e le corrispondenti ordinate f.
% x e f devono essere due vettori della stessa
   lunghezza
% che contengono rispettivamente le
% ascisse e le ordinate dei punti di interpolazione.
% Restituisce un vettore di lunghezza n-1 che contiene
% le differenze divise, cioe' i
% coefficienti del
% polinomio interpolante nella base di Newton.
% See also INTERPOLA, HORNER, CEBYSHEV
function v=DiffDiv(x,f)
n=length(x); v=f;
for i=1:(n-1),
    for j=n:(-1):(i+1),
        v(j)=(v(j)-v(j-1))/(x(j)-x(j-i));
    end;
end
```

Algoritmo di Horner

Dati i coefficienti $c_0 \dots c_n$ del polinomio rappresentato nella base di Newton, un metodo efficiente di calcolare il polinomio in un punto $\overline{x}$ è dato dall'algoritmo di Horner che corrisponde in pratica ad eseguire le operazioni raccogliendo i termini in questo modo.

$$c_0 + (\overline{x} - x_0)(c_1 + (x - x_1)(\dots + (x - x_n - 1)c_n)\dots)$$

Questo algoritmo ha una complessità di $O(3n)$ flops. La procedura Horner esegue l'algoritmo di Horner lavorando in parallelo su un vettore di ascisse di tabulazione e restituisce un vettore dei corrispondenti valori del polinomio in quei punti.

---

p=Horner(c,x,xtab) - *Calcola i valori del polinomio espresso in base di Newton con i coefficienti contenuti in c con ascisse di interpolazione contenute in x. In xtab riceve il vettore dei punti in cui calcolare il valore del polinomio. Restituisce i valori del polinomio nel vettore p.*

CODICE: Horner.m

```
%HORNER
%p=Horner(c,x,xtab)
%
% Calcola in parallelo i valori di un polinomio
% espresso nella base di Newton.
% Parametri in ingresso:
% c    - coefficienti del polinomio nella base di
%        Newton (differenze divise)
% x    - ascisse dei punti di interpolazione
%        (che definiscono la base di Newton)
% xtab - vettore delle ascisse nelle quali si
%         vuole calcolare il polinomio
%
% Restituisce in p un vettore della stessa lunghezza di
    xtab
% contenente i valori che
 il polinomio assume nei punti
% corrispondenti in xtab.
%
%See also INTERPOLA, DIFFDIV,CHEBYSHEV
function p=Horner(c,x,xtab)
n=length(c);
ntab=length(xtab);
uno=ones(1,ntab);
p=c(n)*uno;
for i=1:n-1
   p=c(n-i)*uno +(xtab-(x(n-i)*uno)).*p;
end
return
```

Interpolazione di una funzione

Per interpolare una funzione con un polinomio rappresentato nella base di Newton si calcolano i coefficienti del polinomio con la procedura `DiffDiv` alla quale si passano le ascisse di interpolazione e i corrispondenti valori della f. Poi si calcola il valore del polinomio nei punti necessari con la procedura `Horner`. La function Interpola riceve il nome di una funzione, il vettore delle

ascisse di interpolazione e un parametro (*steps*)che determina il numero di punti da tabulare.

---

pol=Interpola(f,x_interp,steps) - *Riceve il nome della funzione da interpolare, le ascisse di interpolazione e il parametro steps e disegna il grafico con la funzione ed il polinomio interpolante. Restituisce i valori del polinomio negli steps punti di tabulazione compresi tra min(x_interp) e max(x_interp).*

CODICE: Interpola.m

```
%INTERPOLA
%pol=Interpola(f,x_interp,steps)
% Questa funzione interpola una funzione con un
    polinomio
% (rappresentato nella base di Newton)
%
% Parameteri in ingresso
% f - nome della funzione da interpolare
% x_interp - Vettore delle ascisse di interpolazione
% steps    - numero di punti da valutare
%
% Restituisce i valori assunti dal polinomio
    interpolante
% su steps ascisse nell'intervallo di interpolazione.
%See also CEBYSHEV, HORNER, DIFFDIV
function [c,pol]=Interpola(f,x_interp,steps)
n=length(x_interp);
a=min(x_interp);
b=max(x_interp);

ordinate=zeros(1,n);
for i=1:n
    ordinate(i)=feval(f,x_interp(i));
end

x_tab=a:(b-a)/(steps-1):b;
```

```
y_tab=mfeval(f,x_tab);


c=DiffDiv(x_interp,ordinate);
pol=Horner(c,x_interp,x_tab);
```

return

Questi sono i grafici dei polinomi interpolanti la funzione sin calcolati con la procedura Interpola. In questo caso le ascisse sono scelte a distanza costante nell'intervallo ma questa non è la scelta ottimale, come faremo vedere in seguito con l'esempio di una funzione più "difficile". Tuttavia per la funzione sin i risultati sono buoni anche con le ascisse distribuite uniformemente nell'intervallo. I risultati sono discreti già con un polinomio di grado 4 (fig 3.1). Se il polinomio è di grado 8 l'approssimazione è ottima come si vede dalla figura 3.2.

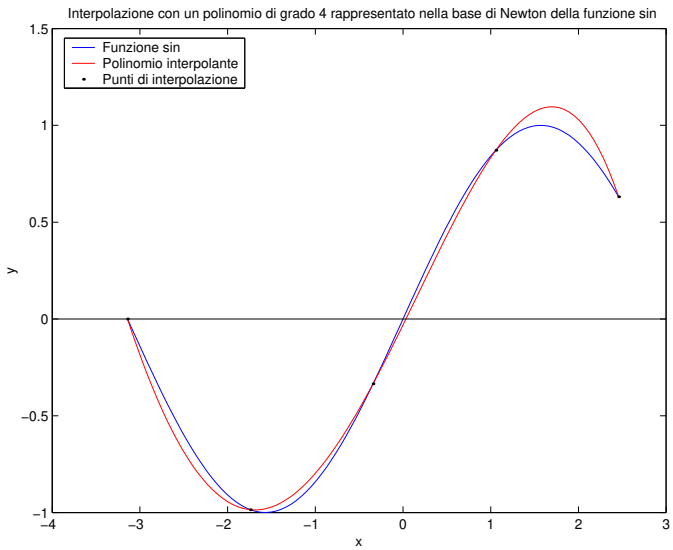

Figura 3.1: Interpolazione della funzione sin con un polinomio di grado 4 nell'intervallo $[-\pi, \pi]$. Ascisse a distanza costante

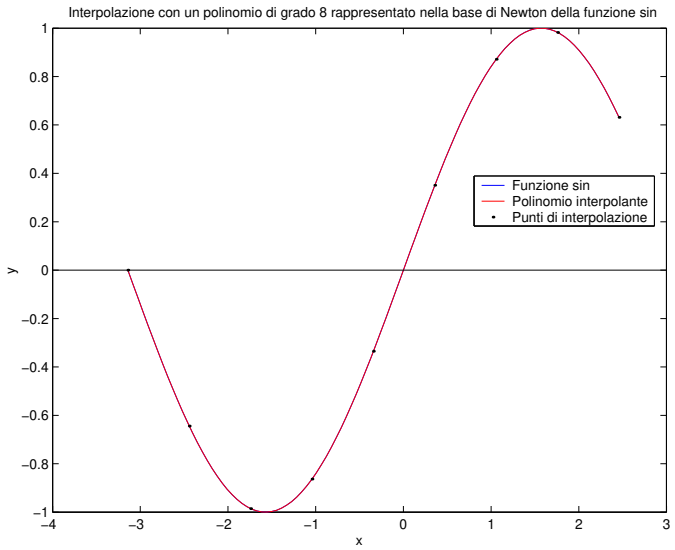

Figura 3.2: Interpolazione della funzione sin con un polinomio di grado 8 nell'intervallo $[-\pi, \pi]$. Ascisse a distanza costante

3.4 Ascisse di Cebyshev

Abbiamo visto nel caso della funzione sin che un polinomio di grado maggiore costituiva una approssimazione migliore della funzione. Ci aspetteremo quindi che all'aumentare del grado del polinomio interpolante questo approssimi sempre meglio la funzione, e che all'aumentare di n la famiglia di polinomi interpolanti tenda alla funzione.

Questo non è vero se scegliamo le ascisse a distanza costante nell'intervallo di approssimazione. Consideriamo ad esempio la *funzione di Runge* così definita

$$f(x) = \frac{1}{1+x^2} \tag{3.1}$$

nell'intervallo $[-5, 5]$. Vediamo i grafici dei polinomi interpolanti la f definita in 3.1 di grado crescente. Per $n = 6$ (cioè $h = 10/6 \approx 1.666667$) si ha il grafico in figura 3.3. L'approssimazione è accettabile. Proviamo ad aumentare il grado: con un polinomio di grado 9 (fig. 3.4) il risultato è ancora abbastanza buono. Se il polinomio è di grado 12 diventa evidete che l'approssimazione della funzione è pessima nel primo e nell'ultimo intervallo della partizione

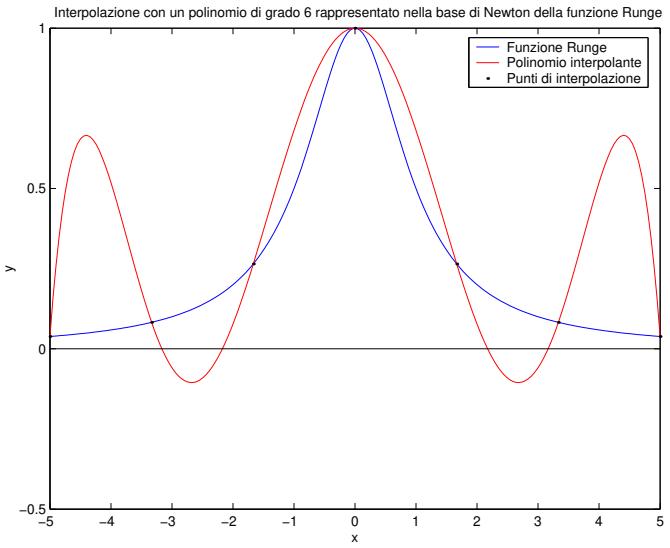

Figura 3.3: Interpolazione della funzione di *Runge* con un polinomio di grado 6 nell'intervallo $[-5;5]$. Ascisse a distanza costante

(fig. 3.5). Alzando il grado a 20 si ottiene il risultato disastroso della figura 3.6

La scelta delle ascisse di *Cebyshev* risolve questo problema. Le ascisse di Cebyshev sono definite in $[-1;1]$ da

$$x_k^{(i)} = \cos\left(\frac{2k+1}{i}\frac{\pi}{2}\right) \quad k = 0 \ldots i-1 \tag{3.2}$$

Nela (3.2) i è il grado del polinomio interpolante.

Per riportare le ascisse dall'intervallo $[-1;1]$ a un generico intervallo $[a;b]$ si usa questa formula, che trasforma $x \in [-1;1]$ in $X \in [a;b]$:

$$X = \frac{a+b}{2} + \frac{b-a}{2}x$$

---

x=Cebyshev(a,b,n) - *Restituisce n ascisse di interpolazione nell'intervallo* $[a;b]$. *Il grado del polinomio interpolante sarà quindi* $n-1$.

CODICE: Cebyshev.m

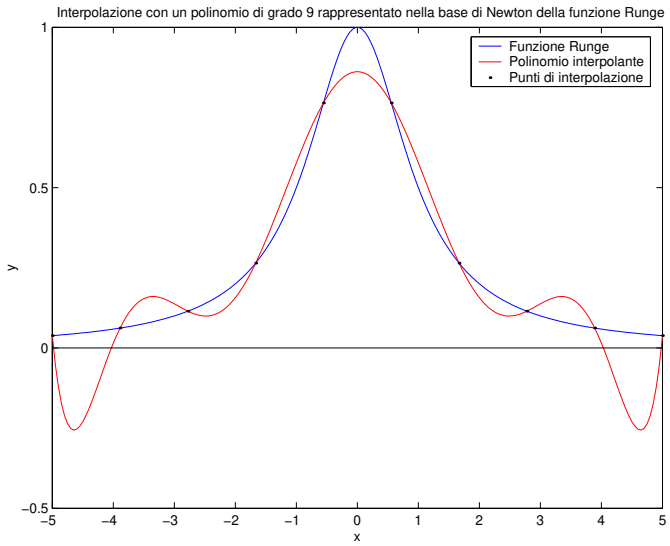

Figura 3.4: Interpolazione della funzione di *Runge* con un polinomio di grado 9 nell'intervallo $[-5; 5]$. Ascisse a distanza costante

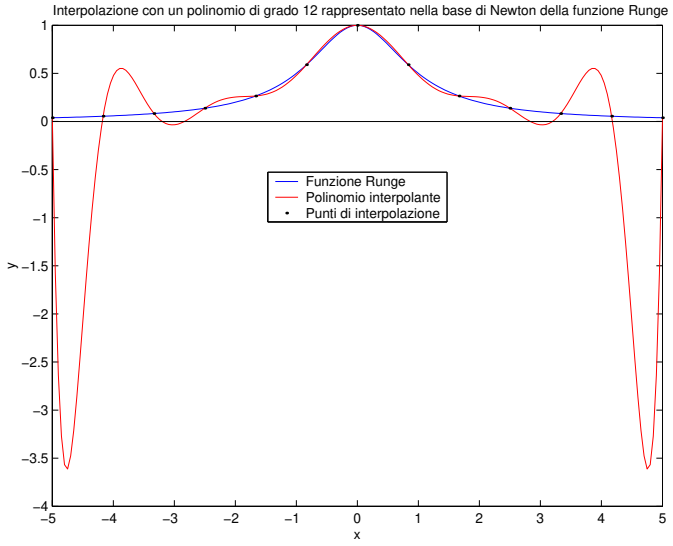

Figura 3.5: Interpolazione della funzione di *Runge* con un polinomio di grado 12 nell'intervallo $[-5; 5]$. Ascisse a distanza costante

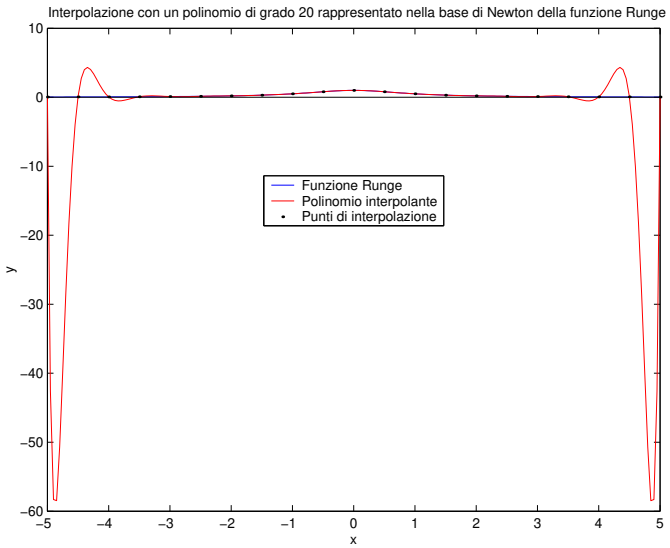

Figura 3.6: Interpolazione della funzione di *Runge* con un polinomio di grado 20 nell'intervallo $[-5; 5]$. Ascisse a distanza costante

```
%CEBYSHEV
%x=Cebyshev(a,b,n)
%
% La funzione restituisce le ascisse di interpolazione
    di
% Cebyshev nell'intervallo [a,b].
% Il parametro n determina il numero di ascisse di
% interpolazione e quindi il grado del polinomio
% interpolante (che sara' n-1).
% La funzione restituisce un vettore di lunghezza n
    contenente
% i punti di interpolazione.
% Le ascisse di interpolazione di Chebyshev nell'
    intervallo
% [-1, 1] sono della forma
% x(i)=cos(((2*i+1)/(n+1))*(pi/2)).
% Vengono riportate sull'intervallo [a,b] con la
    formula
```

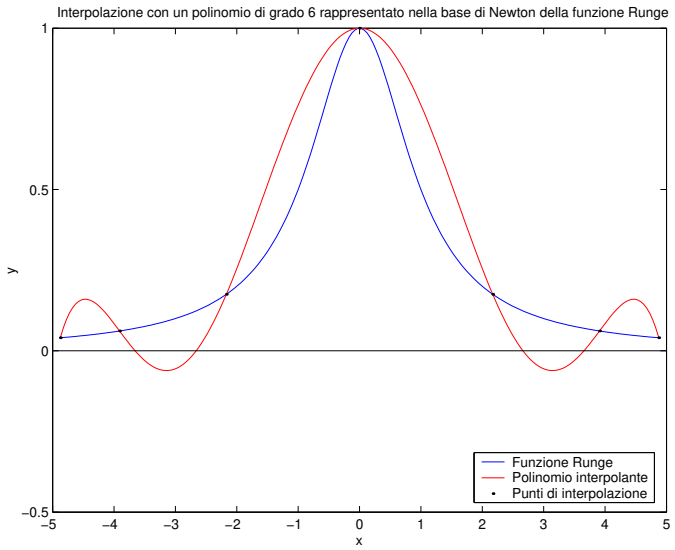

Figura 3.7: Interpolazione della funzione di *Runge* con un polinomio di grado 6 nell'intervallo $[-5; 5]$. Ascisse di Cebyshev

```
% X=(b-a)/2 *x + (a+b)/2
%
%See also INTERPOLA, HORNER, DIFFDIV

function x=Cebyshev(a,b,m)
n=m-1;
for i=0:n
    x(i+1)=cos(((2*i+1)/(n+1))*(pi/2));
end

x=((b-a)/2)*x+((a+b)/2)*ones(1,n+1);
return
```

Con le ascisse di Cebyshev i risultati ottenuti con la funzione di Runge sono infatti migliori e si può vedere che all'aumentare del grado del polinomio interpolante il polinomio tende effettivamente alla funzione. Nelle figure 3.7-3.10 si vedono i grafici dei polinomi interpolanti la funzione di Runge di grado 6,9,12,20 scegliendo le ascisse di interpolazione di Cebyshev

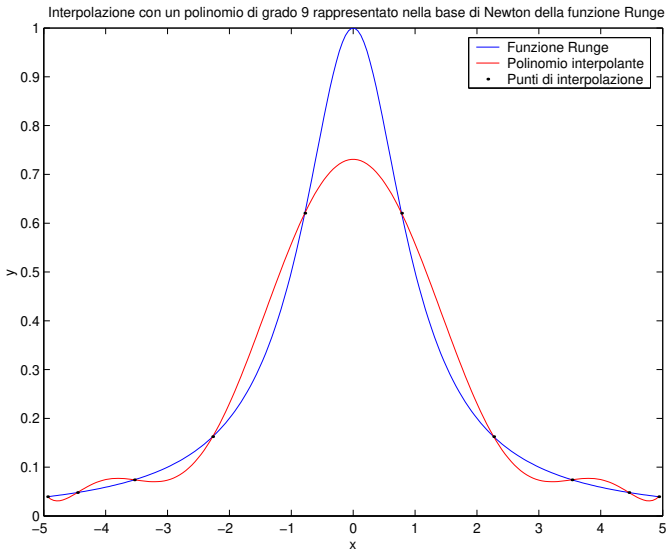

Figura 3.8: Interpolazione della funzione di *Runge* con un polinomio di grado 9 nell'intervallo $[-5; 5]$. Ascisse di Cebyshev

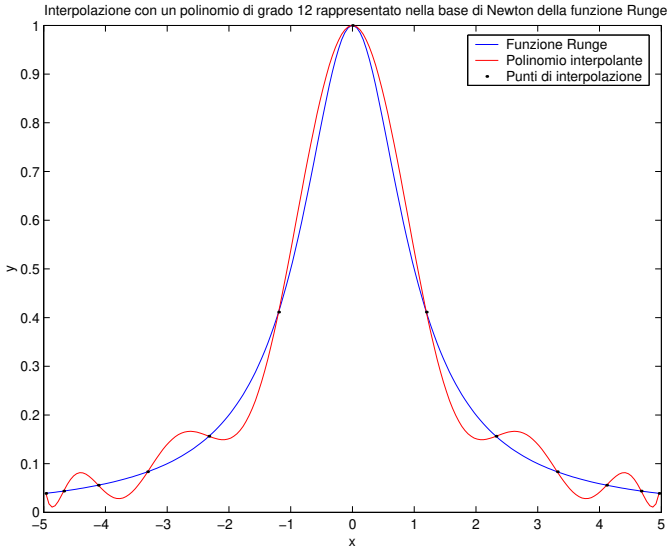

Figura 3.9: Interpolazione della funzione di *Runge* con un polinomio di grado 12 nell'intervallo $[-5; 5]$. Ascisse di Cebyshev

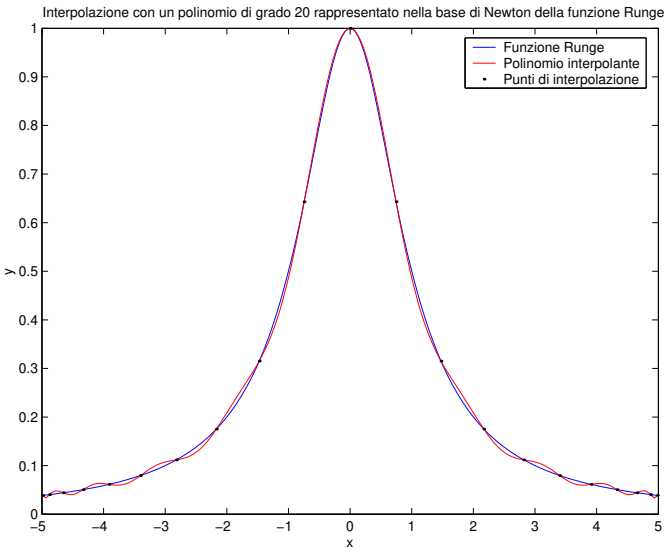

Figura 3.10: Interpolazione della funzione di *Runge* con un polinomio di grado 20 nell'intervallo $[-5; 5]$. Ascisse di Cebyshev

3.5 Interpolazione polinomiale nella base di Lagrange

Un'alternativa alla base di Newton che permette di calcolare il polinomio interpolante senza cadere in un problema mal condizionato consiste nel calcolare i coefficienti del polinomio interpolante nella base di Lagrange.

Gli elementi della base di Lagrange sono così definiti:

$$\mathbf{L}_{in}(x) = \frac{\prod_{\substack{j=0 \\ j\neq i}}^{n} (x - x_j)}{\prod_{\substack{j=0 \\ j\neq i}}^{n} (x_i - x_j)} \tag{3.3}$$

- Questo oggetto è ben definito se per $i \neq j$ si ha $x_i \neq x_j$ e questa è la condizione fondamentale che si richiede alle ascisse di interpolazione.
- $L_{in}(x) \in \Pi_n$.
- I polinomi $L_{in}(x)$ con $i = 0, \ldots, n$ sono esattamente $n + 1$ e si può dimostrare che costituiscono una base per Π_n.

- Il denominatore è una costante reale

Osserviamo anche che

$$\begin{cases} L_{in}(x_j) = 0 & j \neq i \\ L_{in}(x_i) = 1 \end{cases}$$

Vogliamo ora trovare i coefficienti del polinomio interpolante nella base di Lagrange

$$p_n(x) = \sum_{i=0}^{n} \beta_i L_{in}(x)$$

con le condizioni $p_n(x_k) = f_k \quad k = 0, \ldots, n$
ma

$$p_n(x_k) = \sum_{i=0}^{n} \beta_i L_{in}(x_k) = \beta_k L_{kn}(x_k) = \beta_k = f_k$$

quindi i coefficienti nella base di Lagrange coincidono con gli f_i.
In conclusione:

$$p_n(x) = \sum_{i=0}^{n} f_i L_{in}(x)$$

è il polinomio interpolante.

Questo risultato è importante dal punto di vista implementativo: i coefficienti del polinomio interpolante sono noti, essendo gli f_i i dati del problema, l'algoritmo quindi deve solo calcolare il valore dei polinomi di base nelle ascisse di tabulazione.

In particolare la procedura `LagrangeInterpola` costruisce una matrice L che ha n righe (dove n è il grado del polinomio interpolante) e *steps* colonne dove con *steps* indichiamo il numero di punti di tabulazione; questa matrice contiene nella posizione (i, j) il valore di $\mathbf{L}_{in}$ nell'ascissa di tabulazione di posizione j. In simboli se indichiamo con $xtab$ il vettore che contiene i punti di tabulazione e con l_{ij} l'elemento di posto (i, j) della matrice L si ha $L \in \mathbf{R}^{(n\ x\ steps)}$ e

$$l_{ij} = \mathbf{L_{in}}(xtab_j)$$

Una volta costruita questa matrice, i valori assunti dal polinomio interpolante nelle ascisse di tabulazione si ottengono facilmente dal prodotto del vettore che contiene gli f_i e la matrice L. Infatti

$$p_n(xtab_j) = \sum_{i=0}^{n} f_i L_{in}(xtab_j)$$

e questo è proprio il valore assunto dalla posizione j del vettore fL se f è il vettore degli f_i. Da questo prodotto si ottiene un vettore di lunghezza *steps* se f è pensato come vettore colonna infatti le dimensioni di f e di L sono in questo caso $(1\ x\ n)$ e $(n\ x\ steps)$ quindi il risultato del prodotto avrà dimensioni $(1\ x\ steps)$.

La procedura `LagrangeInterpola` riceve una funzione da interpolare, il vettore delle ascisse di interpolazione e il numero di punti di tabulazione da calcolare. La matrice L è ottenuta richiamando la procedura ausilaria `Lag` che calcola in parallelo il valore del nominatore dei vari L_{in} nelle ascisse di tabulazione e poi dividendo tutti i suoi elementi per il valore del denominatore che dipende solo dalle ascisse si interpolazione e non dal punto in cui deve essere calcolato L_{in} (questi calcoli sono svolti dalla procedura `lag_denom`).

I valori del polinomio nei punti di tabulazione sono ottenuti dal prodotto del vettore *ordinate* (che contiene gli f_i) per la matrice L.

---

p=LagrangeInterpola(f,x,steps) - *Interpola la funzione f con un polinomio rappresentato nella base di Lagrange. Restituisce i valori assunti dal polinomio interpolante in steps punti equidistanti dell'intervallo [a,b] dove a=min(x) e b=max(x).*

CODICE: LagrangeInterpola.m

```
%LAGRANGEINTERPOLA
% LagrangeInterpola(f,x,steps)
% Questa funzione interpola una funzione con un
    polinomio
% (rappresentato nella base di Lagrange)
%
% Parameteri in ingresso
% f      - nome della funzione da interpolare
% x      - Vettore delle ascisse di interpolazione
% steps  - numero di punti da valutare
%
%See also CEBYSHEV, INTERPOLA
function p=LagrangeInterpola(f,x,steps)
n=length(x);
a=min(x);
```

```
b=max(x);
ordinate=mfeval(f,x);

xtab=a:(b-a)/(steps-1):b;

ytab=mfeval(f,xtab);


L=zeros(n,steps);
for i=1:n
    L(i,:)=Lag(i,xtab,x)/lag_denom(i,x);
end
p=zeros(1,steps);
ordinate=ordinate(:)';
p=ordinate*L;


function r=Lag(i,xtab,x)
n=length(x);
ntab=length(xtab);

uno=ones(1,ntab);
num=uno;
for j=1:n
    if i~=j
        num=num.*(xtab - x(j).*uno);
    end
end
r=num;

function denom=lag_denom(i,x)
n=length(x);
denom=1;
for j=1:n
    if i~=j
        denom=denom*(x(i) - x(j));
    end
end
```

Dato che il polinomio di grado n interpolante una funzione è unico, anche se i calcoli effettuati sono diversi, la procedura `LagrangeInterpola` trova gli stessi valori della procedura `Interpola`.

Valgono perciò le considerazioni esposte nel paragrafo 3.4 e quindi è conveniente passare anche a questa procedura un vettore x contenente le ascisse di Cebyshev.

3.6 Metodo di Neville per il calcolo del polinomio interpolante in un punto

Se ci interessa il valore del polinomio interpolante in un solo punto si può usare l'algoritmo di Neville che opera in modo simile al calcolo delle differenze divise e si basa sulla seguente formula

$$P_i(x) \equiv f_i$$

$$P_{i\,i+1\ldots j}(x) = \frac{P_{i\,i+1\ldots j-1}(x)(x - x_j) - P_{i+1\ldots j}(x)(x - x_i)}{x_i - x_j}$$

---

y0=Neville(x0,x,f) - *Applica il metodo di Neville per trovare il valore del polinomio interpolante le ascisse x e le ordinate f nel punto x0*

CODICE: Neville.m

```
%NEVILLE
% function y0=Neville(x0,x,f)
%
% Calcola il valore del polinomio interpolante la f
   sulle
% ascisse x nel punto x0 secondo lo schema di Neville.
% See also INTERPOLA, HORNER, CEBYSHEV
function y0=Neville(x0,x,f)
n=length(x);

for i=1:n
    for j=n:-1:i+1
        f(j)= f(j) + ( f(j) - f(j-1) ) ...
```

```
            *(x0 - x(j))/( x(j) - x(j-i));
    end
end
y0=f(n);
return
```

ESEMPIO di Neville.m

```
>> x=Cebyshev(0,5,5)

x =

   4.87764129073788   3.96946313073118
   2.50000000000000   1.03053686926882
   0.12235870926212

>> f=mfeval('exp',x)

f =

   1.0e+02 *

   1.31320551296239
   0.52956092707808
   0.12182493960703
   0.02802570044612
   0.01130159427733

>> x0=Neville(1,x,f)

x0 =

   2.67288785979018

>> exp(1)

ans =
```

```
2.71828182845905
```

3.7 Interpolazione mediante spline cubiche

Nel caso dell'interpolazione mediante polinomi abbiamo visto che sotto opportune condizioni per ogni funzione esiste una classe di polinomi interpolanti che la approssima con precisione sempre maggiore all'aumentare del grado del polinomio.

L'interpolazione mediante spline prevede un approccio diverso al problema dell'interpolazione. Le spline sono funzioni *polinomiali a tratti* che interpolano quindi la funzione tenendo fisso il grado del polinomio ma dividendo l'intervallo di interpolazione in n intervalli più piccoli e considerando un polinomio diverso per ognuno di questi sottointervalli.

In particolare consideremo le *spline cubiche* cioè quelle di grado 3.

Formalmente, data una partizione Δ dell'intervallo di interpolazione $[a, b]$,

$$\Delta = \{a = x_0 < x_1 < \ldots < x_n = b\}$$

$s_3(x)$ è una spline di grado 3 su Δ se:

1. $s_3(x)|_{[x_{i-1},x_i]} \in \prod_3$
2. $s_3(x) \in C^{(2)}([a, b])$

$s_3(x)$ si dice interpolante $f(x)$ se: $s_3(x_i) = f_i$ i=0,1,...,n.

Inoltre si può dimostrare che $S_{3,\Delta} = \{s_3(x) : s_3(x)$ è una spline cubica su $\Delta\}$ è uno spazio vettoriale di dimensione $n + 3$. La dimensione dello spazio $S_{3,\Delta}$ implica che servono $n+3$ condizioni per determinare univocamente una spline cubica; di queste $n + 1$ derivano dalle condizioni di interpolazione.

Abbiamo implementato il metodo delle *spline cubiche naturali* che prevede di scegliere queste due ulteriori condizioni.

$$s_3''(a) = s_3''(b) = 0$$

Per un calcolo efficiente della spline cubica naturale interpolante una funzione consideriamo la partizione

$$\Delta = \{a = x_0 < x_1 < \ldots < x_n = b\}$$

e definiamo

$$h_i = x_i - x_{i-1} \quad i = 2 \ldots n$$

La derivata di s_m è una spline di grado $m - 1$, quindi:

- $s_3'(x)$ è una spline di grado 2 su Δ
- $s_3''(x)$ è una spline di grado 1 (lineare) su Δ

Se indichiamo con I_i l'i-esimo intervallo

$$I_i = [x_{i-1}, x_i]$$

e con

$$M_i = s_3''(x_i), \quad i = 0, \ldots, n$$

ricavando l'equazione di $s_3''(x)|_{I_i}$ che è il segmento di retta che unisce i punti $(x_{i-1}; M_{i-1})$ e $(x_i; M_i)$ e integrando due volte per trovare l'espressione di $s_3(x)$ si ottengono i seguenti risultati:

$$s_3'(x)|_{I_i} = \frac{(x - x_{i-1})^2 M_i - (x_i - x)^2 M_{i-1}}{2h_i} + C_i \tag{3.4}$$

$$s_3(x)|_{I_i} = \frac{(x - x_{i-1})^3 M_i + (x_i - x)^3 M_{i-1}}{6h_i} + C_i(x - x_{i-1}) + D_i \tag{3.5}$$

dove

$$\begin{aligned} C_i &= \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}) \\ D_i &= f_{i-1} - \frac{h_i^2}{6} M_{i-1} \end{aligned}$$

Andiamo ora ad imporre la condizione di *continuità della derivata prima*. Infatti, come abbiamo detto, una spline cubica dev'essere, per definizione ,di classe $C^{(2)}$.

$$s_3'(x_i)|_{I_i} = s_3'(x_i)|_{I_{i+1}} \tag{3.6}$$

Otteniamo in questo modo $n - 1$ condizioni che permettono di ricavare gli M_i incogniti (dalla definizione di spline cubica *naturale* si ha $M_0 = M_n = 0$).

Facendo le opportune sostituzioni nella (3.6) si ottengono $n - 1$ equazioni della forma

$$h_i M_{i-1} + 2(h_i + h_{i+1}) M_i + h_{i+1} M_{i+1} = 6\delta_i$$

dove

$$\delta_i = \frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}$$

Il sistema da risolvere è il seguente

$$\begin{pmatrix} 2(h_1+h_2) & h_2 & & & 0 \\ h_2 & & & & \\ & \ddots & \ddots & \ddots & \\ & & & & h_{n-1} \\ 0 & & & h_{n-1} & 2(h_{n-1}+h_n) \end{pmatrix} \begin{pmatrix} M_1 \\ \vdots \\ \\ \\ M_{n-1} \end{pmatrix} = 6 \begin{pmatrix} \delta_1 \\ \vdots \\ \\ \\ \delta_{n-1} \end{pmatrix}$$

La matrice dei coefficienti di questo sistema è tridiagonale e simmetrica, a diagonale dominante, definita positiva e quindi non singolare, il problema è quindi ben posto.

Risoluzione di un sistema tridiagonale simmetrico

Con queste considerazioni teoriche abbiamo spostato il problema dell'interpolazione mediante spline cubiche alla risoluzione di un sistema lineare la cui matrice dei coefficienti è tridiagonale e simmetrica.

$$T = \begin{pmatrix} a_1 & b_2 & & \\ b_2 & a_2 & \ddots & \\ & \ddots & & b_n \\ & & b_n & a_n \end{pmatrix}$$

Per risolvere in maniera efficiente questo problema facciamo le seguenti osservazioni:

- Una matrice tridiagonale T può essere convenientemente rappresentata da due vettori: il primo, a, che contenga nelle posizioni $1..n$ gli elementi della diagonale di T e un altro, b, che contenga nelle posizioni $2..n$ gli elementi degli elementi al di sopra e al di sotto della diagonale.
- La fattorizzazione più conveniente è quella LDL^T

Con la fattorizzazione LDL^T, la matrice T viene fattorizzata in due matrici: L che ha gli elementi della diagonale uguali a 1 e la sottodiagonale

memorizzata nel vettore $(l_2 \dots l_n)$. La diagonale della matrice D è il vettore $(d_1 \dots d_n)$.

Svolgendo formalmente il prodotto LDL^T si ottiene la matrice

$$\begin{pmatrix} d_1 & d_1 l_2 & & & 0 \\ d_1 l_2 & d_2 + l_2^2 d_1 & & & \\ & \ddots & \ddots & \ddots & \\ & & & & d_{n-1} l_n \\ 0 & & & d_{n-1} l_n & (d_n + l_n^2 d_{n-1}) \end{pmatrix} \tag{3.7}$$

Dalla (3.7) si ricava l'algoritmo per ricavare i vettori d ed l.

$$d_1 = a_1$$
$$\begin{cases} l_i &= \frac{b_i}{d_{i-1}} \\ d_i &= a_i - b_i l_i \end{cases}$$

Questo algoritmo viene seguito nella procedura *solveTrid* per calcolare la fattorizzazione LDL^T in questo caso particolare.

Successivamente la procedura risolve il sistema $LDL^T m = 6\delta$ risolvendo tre sistemi successivi

$$\begin{cases} Ly &= 6\delta \\ Dz &= y \\ L^T m &= z \end{cases}$$

---

m=solveTrid(a,b,delta) - *Risolve un sistema lineare con matrice dei coefficienti tridiagonale e simmetrica. Riceve in a gli elementi della diagonale della matrice, nelle posizioni 2..n di b gli elementi adiacenti alla diagonale ed in delta il vettore dei termini noti. Restituisce in m la soluzione del sistema*

CODICE: solveTrid.m

```
%SOLVETRID
% function m=solveTrid(a,b,delta)
%
```

```
% La funzione risolve un sistema la cui matrice dei
% coefficienti sia tridiagonale e simmetrica.
% Il vettore a deve contenere gli elementi della
% diagonale principale.
% Il vettore b deve contenere nelle posizioni 2..n
% (dove n e' la lunghezza di a) gli elementi della
% sottodiagonale. Il valore di b(1) e' ininfluente.
% In pratica la matrice tridiagonale e' della forma
% | a(1) b(2)                |
% | b(2) a(2) b(3)           |
% ! ....                     |
% |                          |
% | ....          b(n) a(n)  |
%
% delta e' il vettore dei termini noti.
% La funzione restituisce il vettore soluzione del
   sistema,
% ottenuto con un adattamento dell'algoritmo di
% fattorizzazione LDL'.
% Questa procedura viene usata per il calcolo della
% spline cubica naturale.
%
% See also ld, splinecubnat
function m=solveTrid(a,b,delta)
n=length(a);
d=zeros(n,1);
l=d;
m=d;
y=d;
d(1)=a(1);
for i=2:n
    l(i)=b(i)/d(i-1);
    d(i)=a(i)-b(i)*l(i);
end

y(1)=delta(1);
for i=2:n
  y(i)=delta(i)-l(i)*y(i-1);
```

```
end


z=y(:)./d;


m(n)=z(n);
for  i=n-1:-1:1
  m(i)=z(i)-l(i+1)*m(i+1);
end
return
```

La procedura per calcolare i valori assunti da una spline cubica consiste nel calcolo del vettore h delle differenze tra due ascisse di interpolazione successive, quindi dei vettori che descrivono la matrice dei coefficienti T e del vettore dei termini noti δ. Si richiama quindi la procedura *solveTrid* per la risoluzione del sistema tridiagonale e si usa l'espressione (3.5) per calcolare i valori di s_3 nell'intervallo di tabulazione.

---

s=splinecubnat(f,x,steps) - *Riceve in f la funzione da interpolare, in x le ascisse di interpolazione e in steps il numero di punti dell'intervallo di interpolazione in cui tabulare la funzione. Restituisce i valori di s_3 in steps punti dell'intervallo di interpolazione.*

CODICE: splinecubnat.m

```
%SPLINECUBNAT
%function s=splinecubnat(f,x,steps)
%
% La funzione interpola la funzione f con una spline
    cubica
% naturale sulla partizione Delta data dagli elementi
    del
% vettore x. Pre: x dev'essere un vettore ordinato.
% Restituisce i valori che la spline assume in steps
    punti
% dell'intervallo [a,b] dove a=x(1) e b=x(n).
%
```

```
% See also SOLVETRID
function s=splinecubnat(f,x,steps)
n=length(x)-1;
a=x(1); b=x(n+1);
h=zeros(n+1,1);
F=h;
A=h;
m=h;
delta=h
for i=1:n
  h(i)=x(i+1)-x(i);
end

F=mfeval(f,x);

for i=2:n
  delta(i)=(F(i+1)-F(i))/h(i)-((F(i)-F(i-1))/h(i-1));
end
delta=6*delta;
for i=1:n-1
  A(i)=2*(h(i)+h(i+1));
end

m(1)=0;
m(n+1)=0;
m(2:n)=solveTrid(A(1:n-1),h(1:n-1),delta(2:n));
xtab=a:(b-a)/(steps-1):b;

for i=2:n+1
  c(i)=(F(i)-F(i-1))/h(i-1)-(h(i-1)/6)*(m(i)-m(i-1))
  d(i)=F(i-1)-h(i-1)^2/6*m(i-1);
end

count=2;
for i=1:steps
  ytab(i)=feval(f,xtab(i));
  if xtab(i)>x(count)
    count=count+1;
```

```
    end
    s(i)=((xtab(i)-x(count-1))^3)*m(count);
    s(i)=s(i)+((x(count)-xtab(i))^3)*m(count-1);
    s(i)=s(i)/(6*h(count-1));
    s(i)=s(i)+c(count)*(xtab(i)-x(count-1))+d(count);
end
return
```

Nei grafici delle figure 3.11-3.13 è illustrato il comportamento dell'interpolazione mediante spline cubiche naturali della funzione di Runge $y = 1/(1+x^2)$.

L'interpolazione mediante spline dà in questo caso risultati molto buoni, migliori anche dell'interpolazione polinomiale con ascisse di Cebyshev.

In questo caso la scelta delle ascisse di interpolazione a distanza costante è quella ottimale, perché con questa scelta viene sicuramente soddisfatto il seguente *requisito di quasi uniformità* che garantisce l'accuratezza della soluzione

$$\frac{\max h_i}{\min h_i} < k \quad \text{con } k \text{ costante indipendente da } n$$

Questo algoritmo può quindi essere migliorato considerando che le ascisse siano sempre a distanza costante (cioè $h_i = h \; \forall i$) ed evitando in questo modo alcuni calcoli.

La procedura *solveTridCost* risolve un sistema tridiagonale del tipo

$$\begin{pmatrix} 4h & h & & & 0 \\ h & & & & \\ & \ddots & \ddots & \ddots & \\ & & & & h \\ 0 & & & h & 4h \end{pmatrix} \begin{pmatrix} M_1 \\ \vdots \\ \\ \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} \delta_1 \\ \vdots \\ \\ \\ \delta_{n-1} \end{pmatrix} \tag{3.8}$$

---

m=solveTridCost(h,delta) - *Risolve un sistema lineare con matrice dei coefficienti della forma (3.8). Riceve in h l'elemento che determina univocamente la matrice ed in delta il vettore dei termini noti. Restituisce in m la soluzione del sistema*

CODICE: solveTridCost.m

```
%SOLVETRIDCOST
% function m=solveTridCost1111111111111(a,b,delta)
%
% La funzione risolve un sistema la cui matrice dei
    coefficienti
% sia tridiagonale e simmetrica ed in particolare della
     forma
% | 4h    h                    |
% | h     4h    h              |
% | ....                       |
% |                            |
% | ....           h     4h    |
%
% delta e' il vettore dei termini noti.
% La funzione restituisce il vettore soluzione del
    sistema,
% ottenuto con un adattamento dell'algoritmo di
% fattorizzazione LDL'.
% Questa procedura viene usata per il calcolo della
    spline
% cubica naturale.
%
% See also ld, splinecubnatCost
function m=solveTridCost(h,delta)
n=length(delta);
d=zeros(n,1);
a=4*h;
d(1)=a;
for i=2:n
     l(i)=h/d(i-1);
     d(i)=a-h*l(i);
end

y(1)=delta(1);
for i=2:n
   y(i)=delta(i)-l(i)*y(i-1);
end
```

```
z=y(:)./d;

m(n)=z(n);
for i=n-1:-1:1
  m(i)=z(i)-l(i+1)*m(i+1);
end
return
```

La procedura *splinecubnatCost.m* interpola una funzione con una spline cubica naturale calcolata su una partizione Δ di ascisse equidistanti su $[a, b]$.

---

s=splinecubnatCost(f,a,b,n,steps) - *Riceve in f la funzione da interpolare, in a e b gli estremi dell'intervallo di interpolazione, in n il numero di punti nella partizione, in steps il numero di punti dell'intervallo di interpolazione in cui tabulare la funzione. Restituisce i valori di s_3 in steps punti di $[a, b]$.*

CODICE: splinecubnatCost.m

```
%SPLINECUBNATCOST
%function s=splinecubnatCost(f,a,b,n,steps)
%
% La funzione interpola la funzione f con una spline
    cubica
% naturale sulla partizione Delta data da n ascisse
% a distanza costante nell'intervallo [a,b]
% Restituisce i valori che la spline assume in steps
    punti
% dell'intervallo [a,b].
%
% See also SOLVETRID
function s=splinecubnatCost(f,a,b,n,steps)
n=n-1;
h=(b-a)/n;
x=a:h:b;
m=zeros(n+1,1);
delta=zeros(n,1);
F=mfeval(f,x);
```

```
for i=2:n
   delta(i)=(F(i+1)-F(i))-(F(i)-F(i-1));
end
delta=6*(delta./h);

m(1)=0;
m(n+1)=0;
m(2:n)=solveTrid_cost(h,delta(2:n));
xtab=a:(b-a)/(steps-1):b;

for i=2:n+1
   c(i)=(F(i)-F(i-1))/h - (h/6)*(m(i)-m(i-1));
   d(i)=F(i-1)-h^2/6*m(i-1);
end

count=2;
for i=1:steps
   ytab(i)=feval(f,xtab(i));
   if xtab(i)>x(count)
      count=count+1;
   end
   s(i)=((xtab(i)-x(count-1))^3)*m(count);
   s(i)=s(i)+((x(count)-xtab(i))^3)*m(count-1);
   s(i)=s(i)/(6*h);
   s(i)=s(i)+c(count)*(xtab(i)-x(count-1))+d(count);
end
return
```

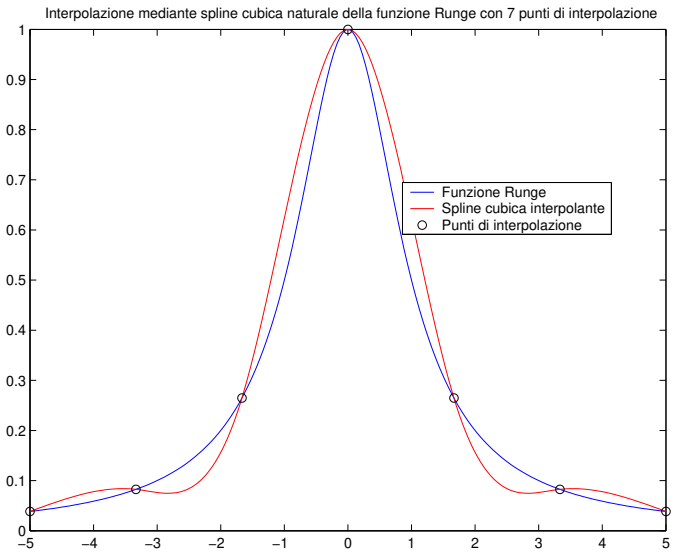

Figura 3.11: Interpolazione della funzione di *Runge* con una spline cubica naturale su una partizione contenente 7 punti nell'intervallo $[-5; 5]$.

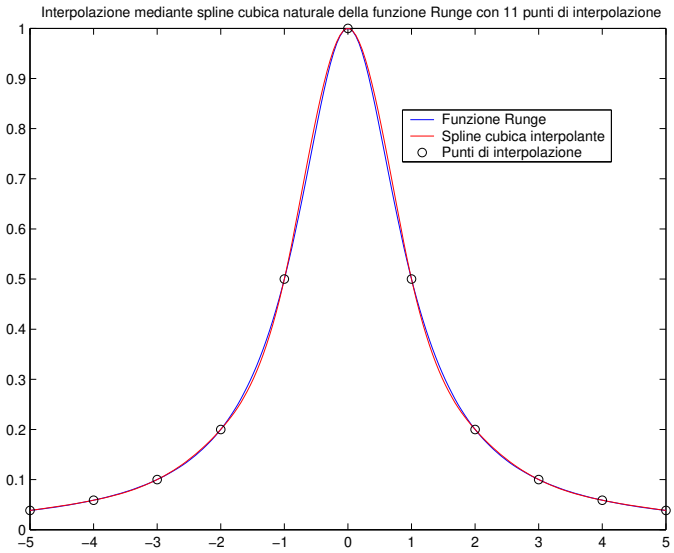

Figura 3.12: Interpolazione della funzione di *Runge* con una spline cubica naturale su una partizione contenente 11 punti nell'intervallo $[-5; 5]$.

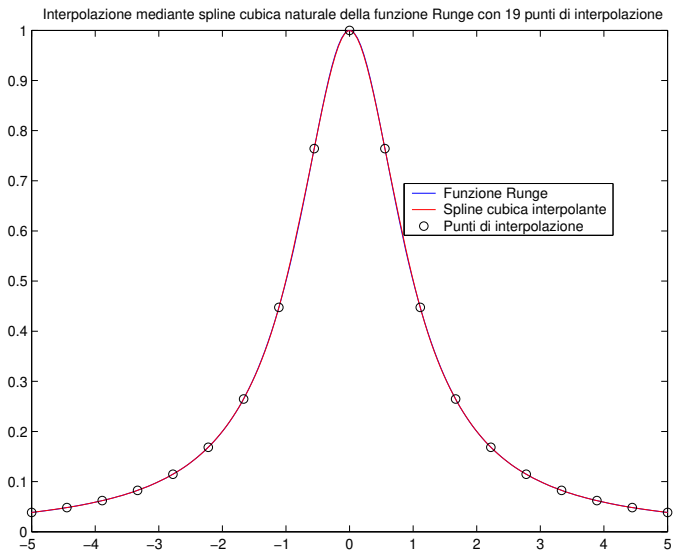

Figura 3.13: Interpolazione della funzione di *Runge* con una spline cubica naturale su una partizione contenente 19 punti nell'intervallo $[-5; 5]$.

Capitolo 4

Il metodo dei minimi quadrati per l'approssimazione polinomiale di funzioni

Il metodo dei minimi quadrati è un metodo numerico per trovare un vettore x che minimizza $\left|\left|Ax-b\right|\right|_2^2$ e può essere applicato all'interpolazione di funzioni mediante polinomi.
Supponiamo di aver effettuato dei campionamenti su un fenomeno mediante $m+1$ coppie (x_i, f_i) e supponiamo che il fenomeno cresca come un polinomio di grado n $(m \gg n)$ il problema è trovare il polinomio $p_n \in \prod_n$ che minimizza le distanze euclidee

$$\sum_{i=0}^{m}(p_n(x_i) - f_i)^2$$

dato che $p_n(x) = \sum_{j=0}^{n} a_j x^j$ il problema diventa il calcolo del polinomio che minimizza

$$\sum_{i=0}^{m}\left(\sum_{j=0}^{n} a_j x_i^j - f_i\right)^2$$

per trovare il polinomio non che resta che risolvere il classico sistema di Vandermonde e se tra le $m+1$ ascisse ce ne sono almeno $n+1$ distinte allora la matrice ha rango massimo. Il sistema da risolvere è del tipo $Va = f$ con V matrice di Vandemonde a vettore delle incognite (i coefficienti del polinomio) ed f vettore delle ordinate. La i-esima componente di $Va-f$ è $\sum_{j=0}^{n} a_j x_i^j - f_i$ e quindi $\sum_{i=0}^{m}\left(\sum_{j=0}^{n} a_j x_i^j - f_i\right)^2 = \left|\left|Va - f\right|\right|_2^2$

Fattorizzando QR la matrice V si ottiene

$$||Va - f||_2^2 = ||QRa - f||_2^2 = ||Ra - Q^T f||_2^2$$

con $R = \begin{bmatrix} R_1 \\ \bigcirc \end{bmatrix}$; chiamiamo $c = Q^T f = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$ con c_1 di dimensione $n+1$, quindi

$$Ra - c = \begin{bmatrix} R_1 a - c_1 \\ -c_2 \end{bmatrix} \quad ||Ra - c||_2^2 = ||R_1 a - c_1||_2^2 + ||c_2||_2^2$$

e questa è la norma da minimizzare; quindi si pone $a = R_1^{-1} c_1$

---

p=MinQuad(x,y,grado) - *Trova i coefficienti di un polinomio del grado immesso come parametro, che minimizza le distanze euclidee tra i dati sperimentali (x[i],y[i]) e i valori assunti dal polinomio nelle varie x[i].*

CODICE: MinQuad.m

```
%MINQUAD
%function p=MinQuad(x,y,grado)
%
% Calcola il polinomio del grado specificato
% date le misure sperimentali x e y con il
% metodo dei minimi quadrati.
function p=MinQuad(x,y,grado)
V=VandermondeRett(x,grado+1);
p=QR(V,y,2);
return
```

ESEMPIO di MinQuad.m

```
>> x=1:1:10;
>> MinQuad(x,x,1)

ans =

    0    1
```

```
>> type parabpert

function y=parabpert(x)
y=3*x^2+2*x+5+sin(x/10);

>> x=1:1:10;
>> y=mfeval('parabpert',x)

y =

   10.0998
   21.1987
   38.2955
   61.3894
   90.4794
  125.5646
  166.6442
  213.7174
  266.7833
  325.8415

>> MinQuad(x,y,2)

ans =

    5.0033    2.1035    2.9982


>> polyfit(x,y,2)

ans =

    2.9974    2.1114    4.9873
```

La procedura descritta sopra utilizza due procedure ausiliarie:

- VandermondeRett per riempire la matrice di Vandermonde
- OR per trovare il vettore di minima norma attraverso la fattorizzazione QR

V=VandermondeRett(x,n) - *Calcola la matrice di Vandermonde associata al vettore x e contenente n colonne*

CODICE: VandermondeRett.m

```
%V=VandermondeRett(x,n)
%
% Restituisce la matrice di Vandermonde corrispondente
% al vettore x. con n colonne
% See also VANDERINTERP, INTERPOLA

function V=VandermondeRett(x,n)
m=length(x);
colonna=ones(m,1);
V=zeros(m,n);
tmp=x(:);
for i=1:n
    V(1:m,i)=colonna;
    colonna=colonna.*tmp;
end
return
```

[x,R]=OR(A,b,2) - *Trova il vettore x che minimizza $||Ax-b||_2^2$. Utilizzando come terzo parametro 1 si ottiene la fattorizzazione QR della Matrice A e risoluzione del sistema lineare $Ax=b$*

CODICE: OR.m

```
%function [x,R]=OR(A,b,op)
% op 1 = fattorizza e risolvi
```

```
% op 2 = minimi quadrati

function [x,R]=OR(A,b,op)
[m,n]=size(A);
beta=zeros(1,m-1);
gamma=zeros(1,n);

for i=1:n-1
    % adesso il vettore v si trova in
    % A(i:n,i)
    A(i:m,i)=house(A(i:m,i))';
    beta(i) = 2/ ([1;A(i+1:m,i)]'* [1;A(i+1:m,i)]);
    % calcola v'*y con y vettore colonna corrente
    for j=i+1:n
        gamma(j)=([1;A(i+1:m,i)]'*A(i:m,j));
    end

    % aggiorna la riga i-esima
    for t=i+1:n
        A(i,t)=A(i,t)-gamma(t)*beta(i);
    end

    for j=i+1:n
        % prodotto vettoriale con fattore moltiplicativo
        % aggiuntivo -gamma(k)*beta(i)
        for k=i+1:m
            A(k,j)=A(k,j)-gamma(j)*A(k,i)*beta(i);
        end
    end
end
% aggiornamento termine noto
x=b(:);

for i=1:n-1
    temp=beta(i)*[1;A(i+1:m,i)]'*x(i:m);
    x(i:m)=x(i:m)-temp*[1;A(i+1:m,i)];
end
R=triu(A);
```

```
if m==n & op==1
    x=solveU(R,x);
elseif m~=n & op==2
    R1=R(1:n,1:n);
    c1=x(1:n);
    x=solveU(R1,c1);
end

return
```

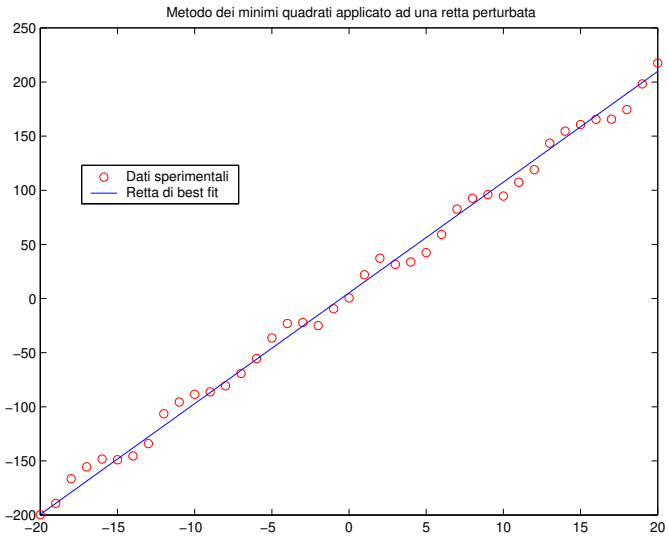

Figura 4.1: Applicazione del metodo dei minimi quadrati a dati sperimentali che crescono linearmente

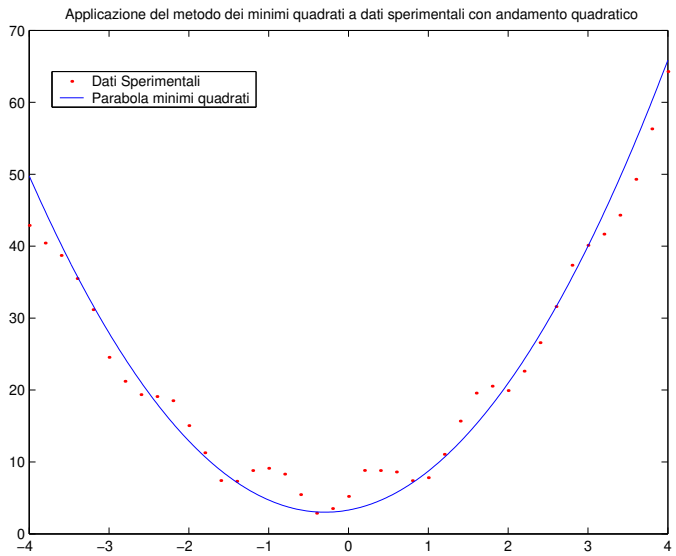

Figura 4.2: Applicazione del metodo dei minimi quadrati a dati sperimentali con andamento quadratico

Capitolo 5

Approssimazione di integrali

Il nostro problema è: data una funzione f integrabile su un intervallo [a,b], calcolare

$$I(f) \equiv \int_a^b f(x)dx$$

e se $f(x) \approx \phi(x)$ con $\phi(x) \in \prod_n$, essendo l'integrazione di polinomi semplice, il problema viene risolto calcolando

$$I(f) \equiv \int_a^b f(x)dx \approx \int_a^b \phi(x)dx$$

dove risulta

$$\int_a^b \phi(x)dx = \int_a^b \sum_{i=0}^{n} a_i x^i dx = \left[\sum_{i=0}^{n} a_i \frac{x^{i+1}}{i+1}\right]_a^b$$

scegliendo il polinomio interpolante la f come $\phi(x)$ il problema diventa quello di calcolare l'integrale del polinomio interpolante su determinate ascisse in [a,b].

5.1 Formule di quadratura di Newton Cotes

Per adesso consideriamo le ascisse di interpolazione equidistanti quindi $h = (b-a)/n$, $x_i = a + ih$, $i = 0 \ldots n$ quindi h é la distanza tra due ascisse contigue; per i nostri scopi viene utilizzata la base di Lagrange quindi

$$\int_a^b p_n(x)dx \equiv \int_a^b \sum_{i=0}^{n} f_i L_{i\,n}(x)dx = \sum_{i=0}^{n} f_i \int_a^b \prod_{j=0, j\neq i}^{n} \left(\frac{x - x_j}{x_i - x_j}\right) dx$$

(n,i)	1	2	3	4	5	6	7	8	k
1	1	1							2
2	1	4	1						6
3	1	3	3	1					8
4	7	32	12	32	7				90
5	19	75	50	50	75	19			288
6	41	216	27	272	27	216	41		840
7	5257	25039	9261	20923	20923	9261	25039	5257	120960

Tabella 5.1: I valori sulla riga i vanno divisi per $k[i]$ per ottenere i giusti coefficienti

dove $\sum_{i=0}^{n} f_i$ dipende dalla funziona f ed il resto dipende dalle ascisse di interpolazione (grado del polinomio ed ampiezza dell'intervallo).

$x = a+ih$ con i fissato e discreto, $x = a+th$ con t che varia con continuità su [0,n] e se $t = i$ ottengo x_i. Gli estremi di integrazione diventano quindi [0,n] ed il problema viene trasformato nel calcolo di

$$\sum_{i=0}^{n} f_i h \int_0^n \prod_{j=0, j\neq i}^{n} \left(\frac{a+th-(a+jh)}{(a+ih)-(a+jh)} \right) dt =$$

$$= h \sum_{i=0}^{n} f_i \left(\int_0^n \prod_{j=0, j\neq i}^{n} \frac{t-j}{i-j} dt \right) = I_n(f)$$

$dx = hdt$ e

$$I_n(f) = h \sum_{i=0}^{n} \alpha_i^x f_i \tag{5.1}$$

e quest'ultima formula prende il nome di *formula di quadratura di Newton-Cotes* con ascisse equidistanti e le varie α_i^n sono tabulate nella tabella (5.1) e non dipendono dalla f.

---

integrale=newtoccotes(f,a,b,m) - *Calcola $\int_a^b f(x)dx$ mediante la formula di quadratura di Newton Cotes mediante la formula di grado m.*

CODICE: newtoncotes.m

```
%function integrale=newtoncotes(f,a,b,m)
% m=grado della formula di Newton - cotes
% 1 = trapezi
% 2 = Simpson
% 3 = pulcherrima
% 4 = Milne
% 5 =  -
% 6 = Weddle
% 7 = -
% altro = non implementato
function integrale=newtoncotes(f,a,b,m)
newcottab= [ 1 1 0 0 0 0 0 0 ; ...
1 4 1 0 0 0 0 0; ...
1 3 3 1 0 0 0 0 ; ...
7 32 12 32 7 0 0 0 ; ...
19 75 50 50 75 19  0 0 ; ...
41 216 27 272 27 216 41 0 ; ...
5257 25039 9261 20923 20923 9261 25039 5257 ];
ns= [ 2 6 8 90 288 840 120960 ];
n=m+1;

h=(b-a)/m;
x=a:h:b;
F=mfeval(f,x);
integrale=0;
for i=1:n
    integrale=integrale+F(i)*newcottab(m,i);
end
integrale=integrale*(b-a)/ns(m);
return
```

ESEMPIO di newtoncotes.m

```
>> newtoncotes('sin',0,pi,1)

ans =

    1.923607162353882e-016
```

```
>> newtoncotes('sin',0,pi,2)

ans =

   2.09439510239320

>> newtoncotes('sin',0,pi,4)

ans =

   1.99857073182384

>> newtoncotes('sin',0,pi,7)

ans =

   2.00001086554154
```

Specializzando il metodo per $n = 1$ ed $n = 2$ si ottengono rispettivamente il metodo dei trapezi e quello di Simpson. Nel primo metodo la formula di quadratura diventa

$$I_1(f) = (b-a)/2(f_a + f_b)$$

e corrisponde all'area del trapezio con estremi $(a, 0)$ $(b, 0)$ (a, f_a) e (b, f_b). Il metodo di Simpson prende invece la seguente forma

$$I_2(f) = (b-a)/6(f_a + 4(f\frac{a+b}{2}) + f_b)$$

e corrisponde all'area del settore di parabola che passa per i punti (a, f_a) (b, f_b) e per $(\frac{a+b}{2}, f(\frac{a+b}{2}))$

---

x=trapezi(f,a,b) - *Calcola* $\int_a^b f(x)dx$ *mediante il metodo dei trapezi*

CODICE: trapezi.m

```
%TRAPEZI
%function [x]=trapezi(f,a,b)
% Calcola l'integrale di f in [a,b] con la formula
% dei trapezi.
function [x]=trapezi(f,a,b)
fa=feval(f,a);
fb=feval(f,b);

x=((b-a)/2)*(fa+fb);

return
```

ESEMPIO di trapezi.m

```
>> trapezi('sin',0,pi)

ans =

     1.923670693721790e-16

>> type parabola

function y=parabola(x)
y=x^2

>> trapezi ('parabola',0,3)

ans =

  13.50000000000000
```

---

x=Simpson(f,a,b) - *Calcola* $\int_a^b f(x)dx$ *mediante il metodo di Simpson*

CODICE: Simpson.m

```
%SIMPSON
%function [x]=Simpson(f,a,b)
% Calcola l'integrale di f in [a,b] con la formula
% di Simpson.
function [x]=Simpson(f,a,b)
fa=feval(f,a);
f1=feval(f,(a+b)/2);
fb=feval(f,b);
x=((b-a)/6)*(fa+4*f1+fb);
return;
```

ESEMPIO di Simpson.m

```
>> Simpson('Runge',0,1)

ans =

   0.78333333333333

>> atan(1)

ans =

   0.78539816339745


>> Simpson('parabola',0,3)

ans =

     9
```

Ci sono due proprietà fondamentali dei coefficienti α_i^n:

- se $n \leq 7$ allora $\alpha_i^n \geq 0$
- $1/n \sum_{i=0}^{n} \alpha_i^n = 1 \forall n$

5.1.1 Acuratezza e condizionamento del problema

Condizionamento del problema

Sia $h = (b-a)/n \ > 0$ (con $b > a$), in ingresso in luogo dei vari f_i abbiano degli $\tilde{f}_i$ che danno luogo a $I_n(\tilde{f})$ al posto di $I_n(f)$.

$$|I_n(\tilde{f}) - I_n(f)| = |h\sum_{i=0}^{n} \alpha_i^n(\tilde{f}_i - f_i) \leq \left(h\sum_{i=0}^{n}|\alpha_i^n|\right) max_i|\tilde{f}_i - f_i|$$

$(h\sum_{i=0}^{n}|\alpha_i^n|) = k$ è il numero di condizionamenmto del problema e $max_i|\tilde{f}_i - f_i|$ è l'errore che viene commesso sui dati in ingresso.

$$k = \begin{cases} (n \leq 7) & (b-a)/n\sum_{i=0}^{n}\alpha_i^n = b-a \\ (n > 7) & \gg b-a \end{cases}$$

quindi le formule di Newton Cotes non vengono utilizzate se $n > 7$.

Accuratezza

$$I(f) - I_n(f) = \int_a^b (f(x) - p_n(x))dx = \int_a^b f[x_0, \ldots, x_n, x]\omega_{n+1}(x) =$$

$$= c_n \frac{f^{n+k}(\xi)}{(n+k)!}h^{(n+k+1)}$$

con $\xi \in [a,b]$, f di classe C^{n+k}, $k = 1$ se n è dispari e 2 altrimenti e c_n dipende da n e non da f. Quindi se n è grande l'accuratezza aumenta, ma il problema diventa mal condizionato.

Il problema è che $h = (b-a)/n$ deve andare a 0, ma se n cresce il problema diventa mal condizionamento; per far andare h a 0 allora si lascia n invariato e si fa tendere $b-a$ a 0 e questa è la base delle formule composite di Newton Cotes.

5.1.2 Formule composite di Newton Cotes

Le formule composite di integrazione sfruttano la proprietà di linearità dell'operazione di integrazione ovvero suddividono l'intervallo [a,b] in n intervalli determinati dalle ascisse $a = x_0 < x_1 < \ldots < x_n = b$ e calcolano

$$\int_a^b f(x)dx = \sum_{i=0}^{n}\int_{x_{i-1}}^{x_i} f(x)dx$$

in questo modo è possibile utilizzare le formule di Newton Cotes fino al settimo grado su intervalli sempre più piccoli soddisfacendo i requisiti di accuratezza e di buon condizionamento del problema.

---

integ=newtoncotescomp(f,a,b,m,n) - *Calcola* $\int_a^b f(x)dx$ *mediante le formule compisite di Newton Cotes di grado m su n intervalli*

CODICE: newtoncotescomp.m

```
function integ=newtoncotescomp(f,a,b,m,n)
h=(b-a)/n;
x=a:h:b;
integ=0;
for i=1:n-1
    integ=integ+newtoncotes(f,x(i),x(i+1),m);
end
return
```

ESEMPIO di newtoncotescomp.m

```
>> newtoncotescomp('sin',0,pi,4,100)

ans =

   1.99950656036573

>> newtoncotescomp('sin',0,pi,2,300)

ans =

   1.99994516937386


>> newtoncotescomp('parabola',0,3,4,100)

ans =
```

```
   8.73269100000000

>> newtoncotescomp('parabola',0,3,7,100)

ans =

   8.73269100000000

>> newtoncotescomp('parabola',0,3,7,1000)

ans =

   8.97302699100000
```

Specializzando il metodo si ottengono le formule composite dei trapezi

$$I_{1\,n}(f) \equiv h(\frac{f_0}{2} + f_1 + \ldots + f_{n-1} + \frac{f_n}{2})$$

---

x=trapezi-comp(f,a,b,n) - *Calcola* $\int_a^b f(x)dx$ *mediante la formula composita dei trapezi con n intervalli*

CODICE: trapezicomp.m

```
% TRAPEZI-COMP
% function x=trapezi_comp(f,a,b,n)
% Calcola l'integrale di f in [a,b] con la formula
% composita dei trapezi su n intervalli.
function x=trapezi_comp(f,a,b,n)
h=(b-a)/n;
F=zeros(1,n);
ascisse=a:h:b;
for i=1:n
    F(i)=feval(f,ascisse(i));
end
somma=(F(1)+F(n))/2;
for i=2:n-1
```

```
    somma=somma+F( i );
end
x=somma*h;
return
```

ESEMPIO di trapezicomp.m

```
>> type parabola

function y=parabola(x)
y=x^2;


>> trapezicomp ('parabola',0,3,10)

ans =

   6.60150000000000

>> trapezicomp ('parabola',0,3,15)

ans =

   7.33600000000000

>> trapezicomp ('parabola',0,3,100)


ans =

   8.73313650000000

>> trapezicomp ('parabola',0,3,500)

ans =

   8.94612589199999
```

e di Simpson (con n pari)

$$I_{2\,n}(f) \equiv h/3(f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots + 2f_{n-1} + 4f_{n-1} + f_n)$$

---

x=Simpsoncomp(f,a,b,m) - *Calcola* $\int_a^b f(x)dx$ *mediante la formula composita di Simpson con* $2m+1$ *intervalli*

CODICE: Simpsoncomp.m

```
%SIMPSONCOMP
% function x=Simpsoncomp(f,a,b,m)
% Calcola l'integrale di f in [a,b] con la formula
% composita di Simpson su 2m+1 intervalli.
function x=Simpsoncomp(f,a,b,m)
n=2*m+1;
h=(b-a)/n;
F=zeros(1,n);
ascisse=a:h:b;
F=mfeval(f,ascisse);

somma=F(1)+F(n);
for i=2:n-1
    molt=2*(1+(1- mod(i,2)));
    somma=somma+(molt*F(i));
end
x=somma*(h/3);
return
```

ESEMPIO di Simpsoncomp.m

```
>> type parabola

function y=parabola(x)
y=x^2;


>> Simpsoncomp('parabola',0,3,10)
```

```
ans =

    7.7745

>> Simpsoncomp('parabola',0,3,20)

ans =

    8.3574

>> Simpsoncomp('parabola',0,3,100)

ans =

    8.8663

>> Simpsoncomp('parabola',0,3,300)

ans =

    8.9551

>> Simpsoncomp('parabola',0,3,500)

ans =

    8.9731

>> Simpsoncomp('sin',0,pi,10)

ans =

   1.98883637511258

>> Simpsoncomp('sin',0,pi,20)

ans =
```

```
   1.99706618390939

>> Simpsoncomp('sin',0,pi,30)

ans =

   1.99867416802724

>> Simpsoncomp('sin',0,pi,100)

ans =

   1.99987785760372
```

5.2 Formule adattative

Le formule composite con ascisse equidistanti sono ottime nel caso in cui la funzione sia abbastanza regolare, ma se la funzione presenta delle regioni in cui cresce velocemente e delle regioni in cui è quasi costante la scelta delle ascisse equidistanti non ottiene buoni risultati, sia per l'accuratezza sia per l'eccessivo numero di calcoli svolti nell' area dove la funzione è quasi costante. Abbiamo quindi bisogno di una "formula composita variabile" che si adatti alla forma della funzione aggiungendo più ascisse nella regione in cui la funzione cresce velocemente.

Per fare questo si utilizza una strategia ricorsiva che necessita di una stima dell'errore nel risultato ottenuto per decidere se arrestare l' algoritmo o continuare dividendo in due sotto intervalli l'intervallo di integrazione corrente ed operare ricorsivamente su questi ultimi.

La procedura `adaptiv` usa come metodo di base quello dei trapezi composito su due intervalli, $I_{1\,2}(f)$. A costo zero è possibile ricavarsi anche il valore di $I_1(f)$. Si può dimostrare che

$$|I(f) - I_{1\,2}(f)| \approx \frac{1}{3}|I_{1\,2}(f) - I_1(f)|$$

Quindi come criterio di arresto si impone che questa stima dell'errore sia minore di una tolleranza data ϵ. Se la condizione non è verificata si suddivide

l'intervallo corrente in due sottointervalli di uguale ampiezza e si richiama ricorsivamente la procedura con tolleranza $\epsilon/2$.

I=adaptiv(f,a,b,tol) - *Calcola* $\int_a^b f(x)dx$ *mediante la formula adattativa dei trapezi con tolleranza tol.*

CODICE: adaptiv.m

```
%ADAPTIV
%function I=adaptiv(f,a,b,tol)
% Calcola l'integrale di f su [a,b] con
% tolleranza tol mediante la formula adattativa
% dei trapezi
function I=adaptiv(f,a,b,tol,fa,fb)
if nargin==4
    fa=feval(f,a);
    fb=feval(f,b);
end
h=b-a;
x1=(a+b)/2;
f1=feval(f,x1);
I1=0.5*h*(fa+fb); % metodo dei trapezi
I=0.5*(I1+h*f1); % metodo dei trapezi composito su 2
    intervalli
err=abs(I-I1)/3;
 % stima dell'errore
if err>tol
    I=adaptiv(f,a,x1,tol/2,fa,f1)+adaptiv(f,x1,b,tol/2,
       f1,fb);
end
```

ESEMPIO di adaptiv.m

```
>> adaptiv('sin',0,pi,1E-5)

ans =
```

```
   1.99999307021164

>> adaptiv('Runge',0,1,1E-5)

ans =

   0.78539745067914

>> atan(1)

ans =

   0.78539816339745
```

5.3 Metodo di estrapolazione di Romberg

Se la funzione f è molto buona si ha che il metodo dei trapezi composito con ampiezza degli intervalli $h\, I_{1\,n}(f) \equiv T_h(f)$ ammette un sviluppo del tipo $\tau_0 + \tau_1 h^2 + \ldots \tau_r h^{2r} \ldots$ con i vari $\{\tau_i\}$ indipendenti da h e se $h = 0$ si ha che $T_0(f) = I(f)$. Poniamo $x = h^2$ e $g(x) = T_h(f)$. Non possiamo calcolare direttamente $g(0) = I(f)$ e non conosciamo la funzione $g(x)$, conosciamo però il valore che questa assume in $k+1$ punti distinti ed è quindi possibile ricavare il valore del polinomio $p_k(x)$ interpolante $g(x)$ nel punto $x = 0$ ed assumere questo valore come approssimazione di $g(0)$ e quindi dell'integrale. Il punto $x = 0$ non appartiene all'intervallo in cui si interpola la $g(x)$ e per questo si parla di estrapolazione. Scegliendo i vari h_i decrescenti secondo la formula $h_i = \frac{b-a}{2^{i-1}}$, $x_i = h_i^2$ si ottiene il metodo di Romberg, e le varie x_i sono vicine al punto 0 più delle h_i. Dovendo calcolare il valore del polinomio interpolante in un punto si applica lo schema di Neville opportunamente modificato tenendo conto che deve essere calcolato in $x = 0$ e la formula diventa

$$T_{i,j} = T_{i,j-1} + \frac{T_{i,j-1} - T_{i-1,j-1}}{4^j - 1}$$

e $T_{i,0} = I_{1\,h_i}(f)$ cioè il valore restituito dal metodo dei trapezi composito con numero di punti via via crescente.

---

int=romberg(f,a,b,k) - *Calcola* $\int_a^b f(x)dx$ *secondo il metodo di Romberg*

con k punti di interpolazione

CODICE: romberg.m

```
%ROMBERG
% function int=romberg(f,a,b,k)
% Calcola l'integrale di f su [a,b] con il metodo di
% estrapolazione di Romberg.
% k e' il numero di valutazioni del metodo dei trapezi
   che
% devono essere fatte per inizializzare il metodo.
function int=romberg(f,a,b,k)
t=zeros(k,1);
quattro=t;
n=1;
for i=1:k
    t(i)=trapezicomp(f,a,b,n);
    n=2*n;
end

% Calcola le potenze di quattro in un vettore.
q=4;
for i=1:k
    quattro(i)=q;
    q=q*4;
end

for i=1:k
    for j=k:-1:i+1
        t(j)=t(j)+(t(j)-t(j-1))/(quattro(j)-1);
    end
end
int=t(k);
```

ESEMPIO di romberg.m

```
>> romberg('sin',0,pi,5)
```

```
ans =

    1.9747

>> romberg('sin',0,pi,12)

ans =

    2.0000

>> romberg('Runge',0,pi,12)

ans =

   1.26248605700994

>> atan(pi)

ans =

   1.26262725567891
```

Capitolo 6

Ricerca dell'autovalore dominante

6.1 Introduzione

Sia $A \in \mathbf{C}^{n\,x\,n}$. Un vettore $v \in \mathbf{C}^n$, $v \neq \underline{0}$ si dice *autovettore* per A se

$$Av = \lambda v \quad \lambda \in \mathbf{C} \tag{6.1}$$

λ è detto *autovalore* per A. Dalla (6.1) si ricava il sistema

$$(A - \lambda I)v = \underline{0}$$

da cui segue che gli autovalori sono le radici del *polinomio caratteristico* $p_A(\lambda) = \det(A - \lambda I)$ cioè i valori per cui la matrice $A - \lambda I$ è singolare e quindi il sistema ammette soluzioni diverse da $v = \underline{0}$.

Ricavarsi gli autovalori in questo modo comporta il calcolo del determinante di una matrice che è una operazione altamente costosa. Si cercano quindi strade alternative per la ricerca degli autovalori.

Noi considereremo solo uno di questi metodi, che si applica ad un caso molto particolare e tuttavia molto frequente nelle applicazioni: la ricerca dell'autovalore dominante cioè di un autovalore maggiore in modulo di tutti gli altri.

6.2 Metodo delle potenze

Supponiamo che la matrice A abbia un autovalore dominante (e quindi semplice) λ_1 tale che

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$$

Il metodo delle potenze fornisce un metodo iterativo per approssimare l'autovalore dominante. Dato un vettore di innesco $u_0 \neq \underline{0}$ si definisce una successione di vettori in questo modo:

$$u_{k+1} = Au_k$$

Si può dimostrare che l'autovalore dominante è approssimato dalla seguente espressione

$$\sigma_k = \frac{u_k^* u_{k+1}}{u_k^* u_k} \approx \lambda_1$$

dove con u_k^* si indica il *trasposto coniugato* del vettore u_k.

Per evitare problemi di overflow o underflow si usa una versione normalizzata del metodo delle potenze, nella quale tutti i vettori u_k sono divisi per la loro norma in modo che la norma dei vettori così ottenuti sia sempre unitaria.

$$\begin{aligned} u_0 \quad & \text{(vettore di innesco)} \\ y_k &= \frac{u_k}{\left|\left|u_k\right|\right|_2} \\ u_{k+1} &= Ay_k \\ \sigma_{k+1} = \frac{y_k^* u_{k+1}}{y_k^* y_k} &= y_k^* u_{k+1} \end{aligned} \tag{6.2}$$

dove l'ultima eguaglianza deriva dal fatto che $y_k^* y_k = \left|\left|y_k\right|\right|_2 = 1$.

La procedura *Potenze* utilizza lo schema (6.2) usando come condizione di arresto $|\lambda_k - \lambda_{k-1}| < \epsilon$. Nell'esempio vediamo che effettivamente trova l'autovalore dominante con una accuratezza dell'ordine di 10^{-10} in 12 passi e che effettivamente la soluzione trovata differisce dal risultato dato dalla funzione di libreria del Matlab *eig* solo a partire dalla undicesima cifra decimale e quindi il risultato è esatto fino alla decima cifra come richiesto dal nostro ϵ.

[lambda,passi]=Potenze(A,y0,epsilon,upper) - *Riceve in A la matrice di cui si vuole calcolare l'autovalore dominante, in y0 un vettore di innesco (non nullo), in epsilon la accuratezza desiderata e in upper un limite superiore al numero di iterazioni.*
Restituisce in lambda l'approssimazione dell'autovalore dominante e in passi il numero di iterazioni effettuate.

CODICE: Potenze.m

```
%POTENZE
%function [lambda,passi]=Potenze(A,y0,epsilon,upper)
%
% Parametri in ingresso
% A  - matrice di cui si vuole calcolare l'autovalore
%      dominante
% y0 - vettore di innesco (non nullo!)
% epsilon - tolleranza
% upper - numero massimo di iterazioni
% Restituisce in lambda l'autovalore dominante e in
% passi il numero di iterazioni che sono state
   necessarie
% per arrivare al risultato.
function [lambda,passi]=Potenze(A,y0,epsilon,upper)
y0=y0(:);
y=y0/norm(y0);
lambdap=0;
lambda=1;
% si suppone che epsilon<1 quindi inizializzando
% in questo modo lambda e lambdap il corpo del
% while viene sicuramente eseguito la prima volta.
passi=0;
while (abs(lambdap-lambda)>epsilon) & (passi<upper)
     yp=y;
     lambdap=lambda;
     y=A*y;
     lambda=(conj(yp)'*y)/(conj(yp)'*yp);
     y=y/norm(y);
     passi=passi+1;
end
```

ESEMPIO di Potenze.m

```
>> A=[1 2 3 5
4 3 2 1
8 7 5 6
9 9 8 7]
```

```
A =

     1     2     3     5
     4     3     2     1
     8     7     5     6
     9     9     8     7


>> eig(A)

ans =

 19.41388824946578
 -1.55140196941206 + 0.95119045213462i
 -1.55140196941206 - 0.95119045213462i
 -0.31108431064165

>> [lambda,passi]=Potenze(A,[1 1 1 1],1E-10,200)

lambda =

  19.41388824945776


passi =

    12
```