# Metodi numerici con elementi di Programmazione

**A.A. 2013-2014**

## Introduzione al MatLab
## VII parte

Docente: Vittoria Bruni

Email: vittoria.bruni@sbai.uniroma1.it

Ufficio: Via A. Scarpa,
Pal. B, I piano, Stanza n. 16
Tel. 06 49766648

Ricevimento: Giovedì 14.00-15.00

Il materiale didattico è disponibile sul sito http://ingaero.uniroma1.it/ nella pagina dedicata al corso Metodi Numerici con elementi di Programmazione

Per consultazione: Getting Started with MatLab – The mathworks
www.mathworks.com

# Esercizio

Scrivere la funzione Matlab SOR_opt.m che implementi il metodo iterativo S.O.R.. La funzione deve ricevere in input la matrice **A** del sistema, il vettore **b** dei termini noti, il vettore dell'approssimazione iniziale **X0**, l'accuratezza **eps** richiesta alla approssimazione della soluzione prodotta e il numero massimo di iterazioni consentite **max_iter**.

Se il parametro di rilassamento non è dato in input, la funzione deve assegnare alla variabile **omega** il valore del parametro ottimo.

La funzione deve restituire in output, la approssimazione **X** della soluzione del sistema, il vettore **ERR** dell'errore massimo commesso ad ogni iterazione e il numero di iterazioni eseguite **iter**.

Se richiesto, la funzione deve restituire come variabile output il parametro di rilassamento **omega** usato.

# Esercizio

Scrivere la funzione Matlab SOR_opt.m che implementi il metodo iterativo S.O.R.. La funzione deve ricevere in input la matrice **A** del sistema, il vettore **b** dei termini noti, il vettore dell'approssimazione iniziale **X0**, l'accuratezza **eps** richiesta alla approssimazione della soluzione prodotta e il numero massimo di iterazioni consentite **max_iter**.

Se il parametro di rilassamento non è dato in input, la funzione deve assegnare alla variabile **omega** il valore del parametro ottimo.

La funzione deve restituire in output, la approssimazione **X** della soluzione del sistema, il vettore **ERR** dell'errore massimo commesso ad ogni iterazione e il numero di iterazioni eseguite **iter**.

Se richiesto, la funzione deve restituire come variabile output il parametro di rilassamento **omega** usato.

**???**

# Strutture dati

Cell permette di collezionare in un'unica variabile:

- oggetti di vario tipo (vettori, matrici, variabili numeriche o logiche, caratteri o stringhe)
- variabili dello stesso tipo ma di dimensione diversa

Variabili di tipo cell si definiscono tra parentesi graffe.

Le componenti si elencano una dopo l'altra e separate da virgole

C = {componente1, componente2, …. , componenteN};

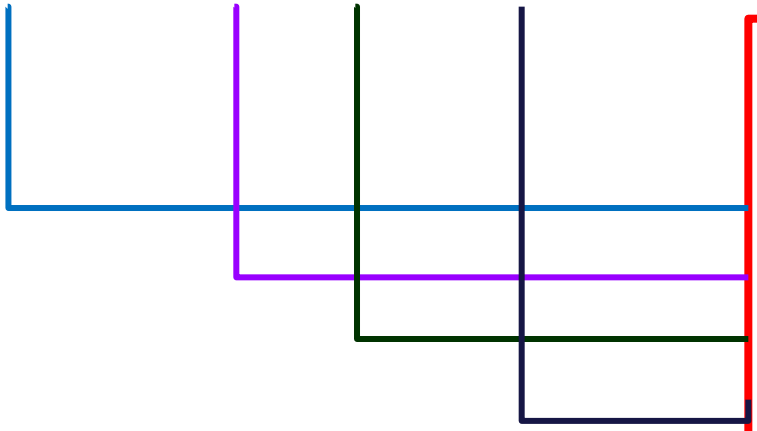
C{i} estrae la i-esima componente di C

C{i}(k) estrae (se esiste) il k-simo elemento della i-esima componente di C

# Strutture dati

Esempio:

```
>> C = {'ciao', [4 3 1 2],3,[1 7 2; 0 5 8; 1 0 9]};
```



definisce una variabile cell composta da 4 elementi:

- una stringa
- un vettore
- un numero
- una matrice

```
>> whos
Name        Size            Bytes  Class
  C         1x4               488  cell array
```

```
>>C{2} % estrae il secondo elemento della variabile di tipo cell C
ans =
    4    3    1    2
```

```
>>C{4}(2,3) %estrae l'elemento con indice di riga 2 e indice
              di colonna 3 della quarta componente di C
ans =
    8
```

# Strutture dati

Nota: una variabile di tipo Cell contiene copie di variabili e non puntatori a variabili.

Quindi se C ={A,B}, con A, B due generiche variabili, il contenuto di C non cambia se in seguito A e B sono modificate

Esempio:

```
>> A = [2 5 3 6];
>> B = [1 5; 8 9;3 6];
>> C = {A,B};
>> B = 1;
>> C{2}
ans =
    1    5
    8    9
    3    6
```

# Strutture dati

C può essere inizializzata con il comando C = cell(m,n) : C è composta da mxn matrici vuote

Celldisp(C): visualizza il contenuto di una cell

Cellplot(C): disegna il contenuto di una cell

Le celle possono essere annidate, cioè un elemento di una cell può essere esso stesso una cell.

Esempio: **C = cell(1,4);** **A = [1 2 3; 4 5 6];,** **v = [1 4 6];**

**C{1} = {A,v};**
**C{2} = 4;**
**C{4} = 'stringa';**

Per estrarre il contenuto della matrice A si digita il comando
C{1}{1}
*gli indici sono ordinati da sinistra verso destra, dalla cell più esterna a quella più interna*

# Alcune funzioni

varargin = variabile di tipo cell contenente le variabili di input di una funzione.

Si usa quando gli input di una funzione possono variare (esistono parametri di default)

*Oss*: l'uso di varargin richiede un ordine preciso delle variabili di input.
L'assegnazione dei valori di default deve essere compresa nel primo blocco di istruzioni della funzione combinata con il comando nargin.
nargin = numero delle variabili di input

# Alcune funzioni

Esempio:

```
function [R] = confronta(x,varargin)
% la funzione stabilisce se un vettore x ha elementi maggiori di % un valore
fissato T. Se T non viene dato in input, viene assegnato il
% valore di default 256. Lo output è la variabile logica R che vale 1 se
% esiste almeno un elemento in x che risulta maggiore di  T, 0 altrimenti.

if nargin == 0
   error('Attenzione: e'' necessario introdurre un vettore')
elseif nargin == 1
   T = 256;
elseif nargin == 2
   T = varargin{1};
else
    error('Troppe variabili di input!!!')
end

ind = find(x>T);
if isempty(ind),  R = 0, else R=1; end
```

# Alcune funzioni

OSS: per lo output si usa varargout combinata con nargout (numero delle variabili di output)

Esempio:

```
function [R,varargout] = confronta(x,varargin)
% la funzione stabilisce se un vettore x ha elementi maggiori di % un valore
fissato T. Se T non viene dato in input, viene assegnato il valore di
% default 256. Lo output è la variabile logica R che vale 1 se esiste almeno
% un elemento > T, 0 altrimenti. La seconda variabile di output, se richiesta,
% è il vettore contenente le posizioni degli elementi di % x > T
.....
.....
ind = find(x>T);
if isempty(ind),  R = 0, else R=1; end

if nargout == 2
   varargout{1} = ind;
elseif nargout>2
   error('Troppe variabili di output')
end
```

Come funzione precedente

# Esercizio

Tornando all' esercizio

```
function [X,ERR,iter,varargout] = SOR_opt(A,b,X0,eps,max_iter,varargin)
% function [X, ERR, iter] = SOR_opt(A,b,X0,eps,max_iter,varargin)
% Risolve un sistema lineare con metodo di sovra-rilassamento SOR
% eventualmente usando il parametro ottimo qualora questo non sia
% un parametro di input
% INPUT:
% A = matrice dei coefficienti del sistema
% B = vettore dei termini noti
% X0 = vettore dell'approssimazione iniziale
% eps = accuratezza della soluzione
% max_iter = numero massimo di iterazioni consentite
% varargin{1} = omega, parametro di rilassamento del metodo
% OUTPUT:
% X = vettore soluzione
% ERR = vettore dell'errore massimo per ogni iterazione
% iter = numero di iterazioni eseguite
% varargout{1} = valore del parametro di rilassamento
```

```
% controllo e assegnazione valori di default alle variabili di input
if nargin == 5
        L = tril(A);
        Minv = inv(L);
        U = triu(A,1);
        C_GS = -Minv*(U);
        rho_GS = max(abs(eig(C_GS)));
        if (rho_GS >= 1)
           error('Attenzione: ==>> rho di Gauss Seidel > 1')
        else
           omega = 2/(1+sqrt(1-rho_GS));
        end
elseif nargin == 6
        omega = varargin{1};
        if (omega<=0 | omega >= 2)
           error('il metodo SOR non converge: omega non e'' in (0,2)')
        end
        df = def_pos(A);
```

Modificare includendo un controllo per nargin<5 e sul determinante della matrice

```
        if df == 0
            % Costruzione matrice di iterazione
            L = tril(A,-1);
            D = diag(diag(A));
            Minv = inv(L+D/omega);
            U = triu(A,1);
            C_sor = -Minv*(U-(1-omega)*D/omega);
            % Verifica C.N.S. di convergenza per i metodi iterativi
            rhoCsor = max(abs(eig(C_sor)))
            if (rhoCsor >= 1)
                error('Attenzione: ==>> rho > 1, il metodo non converge')
            end
        end
elseif nargin > 6
        error('troppe variabili di input')
end
```

```
% controllo e assegnazione valori di default alle variabili di output
if nargout == 4
        varargout{1} = omega;
elseif nargout > 4
        error('troppe variabili di output')
end
% Calcolo delle dimensioni della matrice
dimA = size(A);, n = dimA(1);, X0 = X0(:)';
% Ciclo iterativo
err = 100;, iter = 0;, ERR = [];, X = X0; tic,
while (err>eps & iter<= max_iter)
    for i = 1:n
        V(i)=(-sum(A(i,1:i-1).*X(1:i-1))-sum(A(i,i+1:n).*X0(i+1:n))+b(i))/A(i,i);
        X(i)=omega * V(i) + (1 - omega)*X0(i);
    end
    err = norm(X-X0,inf);
    ERR = [ERR err];, X0 = X;, iter = iter + 1;
end, toc,
if (iter > max_iter) & (err>eps)
   fprintf('accuratezza richiesta non raggiunta dopo %7d iterazioni',max_iter)
end
```

```
function [df] = def_pos(A)
%function [df] = def_pos(A)
% stabilisce se la matrice quadrata A è simmetrica e definita positiva usando il
% criterio di Sylvester
%
% INPUT
% A = matrice quadrata
%
% OUTPUT
% df = variabile logica. df = 1 se A è simmetrica e definita positiva,
%                              0 altrimenti



% controlla se A è una matrice quadrata
[m,n] = size(A);
if m ~=n
   error('la matrice A deve essere quadrata!!!')
end
```

```
if A==A'
   i = 1;
   while (i<=m) & (det(A(1:i,1:i))>0)

     i = i+1;
   end

   if  i==m+1
     df = 1;
     disp('la matrice e'' simmetrica e definita positiva')
   else
     df = 0;
     disp('la matrice e'' simmetrica ma non e'' definita positiva')
   end
else
   disp('la matrice non e'' simmetrica')
   df = 0;
end
```

```
% controllo prima la dimensione e poi il
% determinante!!!
```

## Esempi

Non si da in input il parametro di rilassamento e si chiede come variabile di output (è il parametro ottimo)

```
>> [X, ERR, iter,omega] = SOR_opt(A,b,[0 0 0]',.5*10^-8,50);
Elapsed time is 0.000530 seconds.
>> omega

omega =

    1.0557
```

Il parametro di rilassamento è una variabile di input e lo si chiede anche come variabile di output

```
>> [X, ERR, iter,omega] = SOR_opt(A,b,[0 0 0]',.5*10^-8,50,1);
la matrice e' simmetrica e definita positiva
Elapsed time is 0.000535 seconds.
>> omega

omega =

     1
```

Si richiedono 5 variabili di output

```
>> [X, ERR, iter,omega,k] = SOR_opt(A,b,[0 0 0]',.5*10^-8,50,1);
la matrice e' simmetrica e definita positiva
??? Error using ==> SOR_opt
troppe variabili di output
```

Si danno 7 variabili di input

```
>> [X, ERR, iter,omega] = SOR_opt(A,b,[0 0 0]',.5*10^-8,50,1,10);
??? Error using ==> SOR_opt
troppe variabili di input
```

# Switch

Switch-case-Otherwise:

```
switch  nome_variabile (o espressione)
case valore1
        1° blocco di istruzioni
case valore2
        2° blocco di istruzioni
.....
otherwise
        ultimo blocco di istruzioni
end
```

Se il valore di nome_variabile è valore1 viene eseguito il 1° blocco di istruzioni;

Se è valore2, il 2° blocco di istruzioni, e così via,

altrimenti esegue l'ultimo blocco di istruzioni

# Switch

Esempio:

```
a = rem(b,3);        % resto della divisione per 3
switch  a
  case 0
      c = b/3;
  case 1
      c = (b-a)/3;
  otherwise
      c = b-a/2;
end
```

# Esercizio

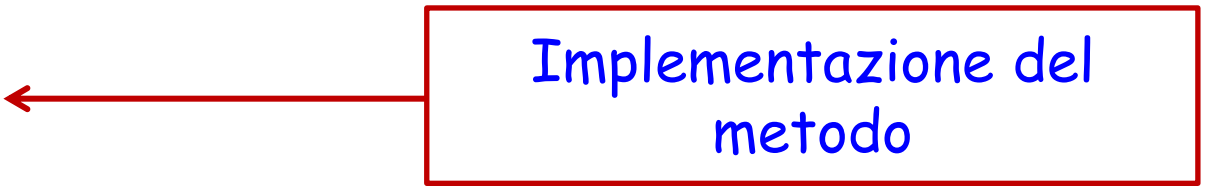
Modificare la funzione SOR_opt.m assegnando valori di default alle variabili X0, eps, max_iter e omega nel caso in cui queste non siano date in input.
La funzione restituisca in output la soluzione e, solo se richiesto, l'errore ad ogni iterazione, il numero di iterazioni eseguite e il valore del parametro di rilassamento.
Si usi opportunamente il comando switch-case-otherwise sia sulla variabile nargin che su naragout.

```
function [X,varargout] = SOR_opt_varargin(A,b,varargin)
.....
.....
if nargin<2
   error('I dati di input sono incompleti')
end
switch nargin
  case 2
          X0 = [0 0 0]';
          eps = 0.5*10^-5;
          max_iter = 100;
          L = tril(A);, Minv = inv(L);, U = triu(A,1);, C_GS = -Minv*(U);
          rho_GS = max(abs(eig(C_GS)));
          if (rho_GS >= 1)
             error('Attenzione: ==>> rho di Gauss Seidel > 1')
          else
             omega = 2/(1+sqrt(1-rho_GS));
          end
```

```
case 3
        X0 = varargin{1};
        eps = 0.5*10^-5;
        max_iter = 100;
        L = tril(A);, Minv = inv(L);, U = triu(A,1);, C_GS = -Minv*(U);
        rho_GS = max(abs(eig(C_GS)));
        if (rho_GS >= 1)
           error('Attenzione: ==>> rho di Gauss Seidel > 1')
        else
           omega = 2/(1+sqrt(1-rho_GS));
        end
 case 4
        X0 = varargin{1};
        eps = varargin{2};
        max_iter = 100;
        L = tril(A);, Minv = inv(L);, U = triu(A,1);, C_GS = -Minv*(U);
        rho_GS = max(abs(eig(C_GS)));
        if (rho_GS >= 1), error('Attenzione: ==>> rho di Gauss Seidel > 1')
        else, omega = 2/(1+sqrt(1-rho_GS));, end
```

```
case 5
        X0 = varargin{1};
        eps = varargin{2};
        max_iter = varargin{3};
        L = tril(A);, Minv = inv(L);, U = triu(A,1);, C_GS = -Minv*(U);
        rho_GS = max(abs(eig(C_GS)));
        if (rho_GS >= 1)
           error('Attenzione: ==>> rho di Gauss Seidel > 1')
        else
           omega = 2/(1+sqrt(1-rho_GS));
        end
 case 6
        X0 = varargin{1};
        eps = varargin{2};
        max_iter = varargin{3};
        omega = varargin{4};
        if (omega<=0 | omega >= 2)
           error('il metodo SOR non converge: omega non e'' in (0,2)')
        end
```

```
    if df == 0
        % Costruzione matrice di iterazione
        L = tril(A,-1);
        D = diag(diag(A));
        Minv = inv(L+D/omega);
        U = triu(A,1);
        C_sor = -Minv*(U-(1-omega)*D/omega);
        % Verifica C.N.S. di convergenza per i metodi iterativi
        rhoCsor = max(abs(eig(C_sor)))
        if (rhoCsor >= 1)
            error('Attenzione: ==>> rho > 1, il metodo non converge')
        end
    end
  otherwise
        error('Troppe variabili di input!!!')
end
...
...
```



```
if nargout >1
   switch nargout
            case 2
                    varargout{1} = ERR;
            case 3
                    varargout{1} = ERR;
                    varargout{2} = iter;
            case 4
                    varargout{1} = ERR;
                    varargout{2} = iter;
                    varargout{3} = omega;
            otherwise
                    error('Troppe variabili di input!!!')
   end

 end
```

# Esercizio

Scrivere uno script Matlab che dato un intero $n \geq 2$, costruisca le matrici $A_k$ così definite

$$a_{ij} = \begin{cases} n & se \ \ i = j \\ -1 & se \ \ |i-j| = k \\ 0 & altrimenti \end{cases}$$

$\forall$ k=1,2,…,n-1

# Strutture dati

E' possibile definire vettori a più dimensioni

N=2 A(i,j) (matrici) i = indice di riga j = indice di colonna di A

N=3 A(i,j,k) (vettore di matrici della stessa dimensione)

i = indice di riga j = indice di colonna della k-sima matrice

contenuta nella k-sima componente di A

Esempio: B = A(:,:,k) B è una matrice

a = A(i,j,k) a è l' elemento avente indice di riga i e indice di colonna j della k-sima matrice in A

In generale si può definire una struttura dati con N indici A(i,j,.....,k)

N indici

Per conoscere la dimensione di A si usa la funzione size

```
>> S = size(A);
```

S è un vettore di N elementi, se N sono gli indici di A

# Strutture dati

Esempio:

```
>> A = ones(6);
>> B = randn(6);
>> C = 10*ones(6);
>> I(:,:,1) = A;
>> I(:,:,2) = B;
>> I(:,:,3) = C;
>> size(I)
ans =
     6     6     3
>> I(:,:,3)
ans =
    10    10    10    10    10    10
    10    10    10    10    10    10
    10    10    10    10    10    10
    10    10    10    10    10    10
    10    10    10    10    10    10
    10    10    10    10    10    10
>> I(3,2,1)    % estrae l'elemento di posizione (3,2) nella prima
               %  componente di I
ans =
     1
```

# Strutture dati

Esempio: Un' immagine a colori RGB si memorizza nella variabile I tale che

I(:,:,1) matrice componente del rosso
I(:,:,2) matrice componente del verde
I(:,:,3) matrice componente del blu

La dimensione di I è mxnx3 (mxn è la dimensione della immagine)

I = imread('nome_immagine.ext')
legge l' immagine contenuta nel file nome_immagine.ext
e la memorizza in una matrice I a 2 (livelli di grigio) o più dimensioni (colore); ext è l'estensione del file (jpg ,png,tiff, etc. )

imshow(I) visualizza un' immagine

# Esercizio

Scrivere uno script Matlab che dato un intero $n \geq 2$, costruisca le matrici $A_k$ così definite

$$a_{ij} = \begin{cases} n & se\ \ i = j \\ -1 & se\ \ |i-j| = k \\ 0 & altrimenti \end{cases}$$

$\forall$ k=1,2,…,n-1

---

```
N = input('inserisci la dimensione delle matrici n = ');
if n <=1, error('n deve essere >= 2')
else  n = round(n)  % mi assicuro che n sia intero
end
d = n*ones(1,n);

for k=1:n-1
    A(:,:,k) = diag(d);
    for i=1:n, for j=1:n, if abs(i-j)==k, A(i,j,k)=-1, end, end
    end
end
```

# Function MATLAB per la costruzione della tavola alle differenze divise

```
function [mat_diff] = diff_div(xnodi,fnodi)
% mat_diff = diff_div(xnodi,fnodi)
% Calcolo della tavola alle differenze divise a partire dai valori dei nodi
% e dalla funzione nei nodi (memorizzati nella prima colonna della
% matrice mat_diffdiv).
%
% Input:
%    xnodi: vettore delle coordinate dei nodi
%    fnodi: valori della funzione nei nodi
% Output:
%    mat_diff: tavola delle differenze divise (per colonne) nella prima
%                  colonna ci sono i valori fnodi

nnodi=length(xnodi);
% ordine massimo delle differenze divise
kord = nnodi-1;
```

# Function MATLAB per la costruzione della tavola alle differenze divise

```
% inizializzazione della tavola alle differenze divise la prima colonna
% contiene i valori della funzione nei nodi
mat_diff = zeros(nnodi,kord+1);
mat_diff(:,1) = fnodi';

% calcolo della tabella
for j = 1:kord, % indice per le colonne della tavola (ordine delle diff. divise)
  for i = 0:nnodi-1-j, % indice per le righe della tavola (nodi coinvolti)
   mat_diff(i+1,j+1) = (mat_diff(i+1,j)-mat_diff(i+2,j))/(xnodi(i+1)-xnodi(i+j+1));
  end
end
```

# Dal Command window

```
>> xnodi = [1.90 2.15 2.20 2.25];
>> fnodi = [-0.10536 0.13976 0.18232 0.22314];
>> [mat_diff] = diff_div(xnodi,fnodi);
>> disp(mat_diff)
  -0.10536000000000   0.98048000000000  -0.43093333333335   0.23695238095262
   0.13976000000000   0.85120000000000  -0.34799999999993                   0
   0.18232000000000   0.81640000000000                   0                   0
   0.22314000000000                   0                   0                   0
```

# Function MATLAB per la costruzione del polinomio di Newton alle differenze divise

```
function[pn,tavdif] = pol_difdiv(xnodi,fnodi,xeval)
% Polinomio interpolatore alle differenze divise:
% [pn,tavdif] = pol_difdiv(xnodi,fnodi,xeval)
% Input:
% xnodi: vettore delle coordinate dei nodi
% fnodi: valori nei nodi della funzione da interpolare
% xeval: vettore delle coordinate dei punti in cui calcolare
%il polinomio interpolatore
% Output:
% pn: vettore dei valori del polinomio interpolatore calcolato in xeval
% tavdif: tavola delle differenza divise calcolata usando la funzione
% diff_div.m

% controllo degli input
if length(xnodi)~=length(fnodi)
   error('il vettore dei nodi ha dimensione diversa dal vettore dei valori della funzione!!!')
end
```

# Function MATLAB per la costruzione del polinomio di Newton alle differenze divise

```
% Calcolo della tavola alle differenze divise
tavdif = diff_div(xnodi,fnodi);

% Calcolo del polinomio interpolatore di grado nnodi-1
nnodi = length(fnodi);
pn = tavdif(1,1); % primo elemento della tavola (f[x0])
pnod = 1; %inizializzazione dei polinomi nodali

for i=1:nnodi-1
        pnod = pnod.*(xeval-xnodi(i));
        pn = pn + pnod*tavdif(1,i+1);
end
```

Dal Command Window:

```
>> xnodi = [1.90 2.15 2.20 2.25];
>> fnodi = [-0.10536 0.13976 0.18232 0.22314];
>> [pn,tavdif] = pol_difdiv(xnodi(1:3),fnodi(1:3),2);
>> pn
  pn =
    -8.47999999996919e-004
```

Calcolando il polinomio interpolatore usando tutti i nodi della tabella e valutandolo nel punto x = 2 si ha p3(2) =-0.00014

```
>> xnodi = [1.90 2.15 2.20 2.25];
>> fnodi = [-0.10536 0.13976 0.18232 0.22314];
>> [pn,tavdif] = pol_difdiv(xnodi,fnodi,2);
>> pn
  pn =
   -1.371428571418221e-004
```

# Esercizio

Modificare la funzione pol_difdiv.m in modo che restituisca come ulteriore variabile di output una stima dell'errore di troncamento qualora venga richiesto.
In tal caso, la funzione deve prevedere come ulteriore variabile di input un nodo aggiuntivo e il valore della funzione in questo punto.

# Esercizio

Scrivere una funzione matlab **diff_finite** che calcoli la tavola della differenze finite dato il vettore di nodi equidistanti e i valori della funzione in corrispondenza di essi.

Si scriva la funzione che calcoli il polinomio interpolatore di Newton alle differenze finite in avanti nel vettore di coordinate $xeval$ usando la funzione **diff_finite**.

## Soluzione

```
 function  [mat_diffin] = diff_finite(xnodi,fnodi)
%
% [mat_diffin] = diff_finite(xnodi,fnodi)
%
% Calcolo della tavola alle differenze finite a partire dai
% valori dei nodi e dalla funzione nei nodi (memorizzati nella
% prima colonna della matrice mat_diffin).
%
% Input:
% - xnodi: vettore delle coordinate dei nodi
% - fnodi: valori della funzione nei nodi
% Output:
% - mat_diffdiv: tavola delle differenze finite(per colonne)
%                nella prima colonna ci sono i valori fnodi
%
```

```
if numel(xnodi)~=numel(fnodi)
    error('il numero dei nodi non corrisponde al numero...
           dei valori della funzione')
end


if diff(xnodi,2)==zeros(1,length(xnodi)-2)

   nnodi=length(xnodi);
   % ordine massimo delle differenze finite
   kord = nnodi-1;
   % inizializzazione della tavola alle differenze finite
 % la prima colonna contiene i valori della funzione nei nodi
 mat_diffin = zeros(nnodi,kord+1);
 mat_diffin(:,1) = fnodi';
```

```
% calcolo della tabella
for j = 1:kord,
      mat_diffin(1:nnodi-j,j+1) = diff(mat_diffin(1:nnodi-j+1,j));
end

else
  error('i nodi devono essere equidistanti')
end
```

# Dal Command Window

```
>> xnodi=[-.5:.25:.25]

xnodi =

  -0.50000000000000  -0.25000000000000        0   0.25000000000000

>> fnodi = sin(pi * xnodi)

fnodi =

  -1.00000000000000  -0.70710678118655        0   0.70710678118655

>> [mat_diffin] = diff_finite(xnodi,fnodi)
```

```
mat_diffin =

  -1.00000000000000   0.29289321881345   0.41421356237309  -0.41421356237309
  -0.70710678118655   0.70710678118655                   0                   0
                  0   0.70710678118655                   0                   0
   0.70710678118655                   0                   0                   0
```

Valutando la funzione nel nodo *xeval = 1/6* si ha

```
>> [pn,tavdif] = pol_diffin(xnodi,fnodi,1/6)

pn =

    0.49697325920912

tavdif =

  -1.00000000000000   0.29289321881345   0.41421356237309  -0.41421356237309
  -0.70710678118655   0.70710678118655                   0                   0
                  0   0.70710678118655                   0                   0
   0.70710678118655                   0                   0                   0
```

mentre il valore vero è *0.5*

# Esercizio

Scrivere la funzione lagrange.m che, dato un vettore di nodi xnodi, un numero intero i e uno scalare xeval, valuti lo i-esimo polinomio della base di Lagrange nel punto xeval.

Scrivere la funzione interp_lagrange.m che, dati due vettori, xnodi e fnodi aventi la stessa lunghezza, e uno scalare xeval, valuti il polinomio interpolatore nella forma di Lagrange costruito sui dati xnodi e fnodi nel punto punto xeval

Le funzioni devono contenere opportuni controlli sulle variabili di input

# Esercizio

Scrivere la funzione matlab **vandermonde.m** che riceva in input un vettore $\mathbf{X}$ e un intero $N$ e costruisca la matrice di Vandermonde $\mathbf{V}$ delle componenti del vettore $\mathbf{X}$ (nodi). Il numero di colonne di $\mathbf{V}$ deve essere $N+1$.

Si generi un vettore $\mathbf{X}$ di $20$ elementi equispaziati nell'intervallo $[0,\ 1]$ e se ne calcoli la matrice di Vandermonde $\mathbf{V}$ associata di dimensione $20 \times 5$ usando la funzione **vandermonde.m**.

Confrontare i risultati usando la funzione predefinita di Matlab **vander.m**

```
function [V] = vandermonde(X,N)
% function [V] = vandermonde(X)
% calcola la matrice di Vandermonde dei nodi X(i).
%
% INPUT
% X = vettore di nodi
% N = numero di colonne +1 della matrice di Vandermonde
%
% OUTPUT
% V = matrice la cui k-esima colonna e' data da X.^k

dimX = size(X);
if size(dimX)>1
    error('La variabile di input deve essere un vettore!')
end

for k = 0:N
    V(:,k+1) = X.^k;
end
```

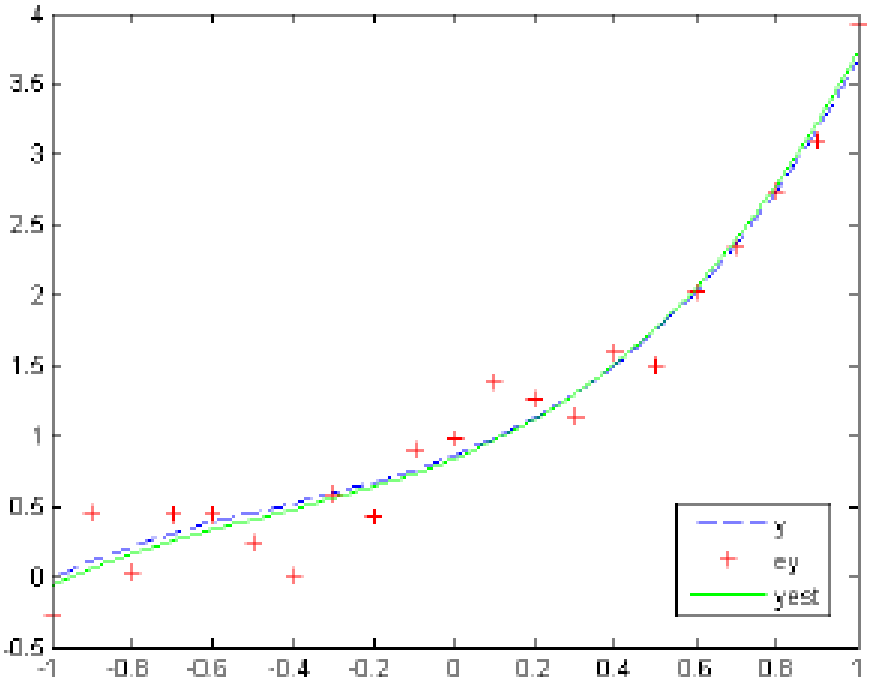
# Dal Command Window

```
>> x = linspace(0,1,20);
>> V = vandermonde(x,4);
>> V

V =

    1.0000         0         0         0         0
    1.0000    0.0526    0.0028    0.0001    0.0000
    1.0000    0.1053    0.0111    0.0012    0.0001
    1.0000    0.1579    0.0249    0.0039    0.0006
    1.0000    0.2105    0.0443    0.0093    0.0020
    1.0000    0.2632    0.0693    0.0182    0.0048
    1.0000    0.3158    0.0997    0.0315    0.0099
    1.0000    0.3684    0.1357    0.0500    0.0184
    1.0000    0.4211    0.1773    0.0746    0.0314
    1.0000    0.4737    0.2244    0.1063    0.0503
    1.0000    0.5263    0.2770    0.1458    0.0767
    1.0000    0.5789    0.3352    0.1941    0.1123
    1.0000    0.6316    0.3989    0.2519    0.1591
    1.0000    0.6842    0.4681    0.3203    0.2192
    1.0000    0.7368    0.5429    0.4001    0.2948
    1.0000    0.7895    0.6233    0.4921    0.3885
    1.0000    0.8421    0.7091    0.5972    0.5029
    1.0000    0.8947    0.8006    0.7163    0.6409
    1.0000    0.9474    0.8975    0.8503    0.8055
    1.0000    1.0000    1.0000    1.0000    1.0000
```

# Esercizio

Scrivere uno script matlab che generi un vettore $y$ contenente i valori di un polinomio di terzo grado in corrispondenza di nodi equidistanti di passo $0.1$ nell'intervallo $[-1, 1]$. Si generi il vettore $noise$, tale che $\|noise\|_2 = 1$, della stessa dimensione di $y$ usando la funzione **randn**. Sia $ey = y + noise$ (simulano i dati ottenuti in un esperimento numerico). Si stimi il polinomio di terzo grado che approssima i dati $ey$ nel senso dei minimi quadrati. Sia $yest$ il vettore stimato e lo si confronti con il vettore originale $y$

```
>> a = [0.74 0.97 1.1 0.86];
>> x = -1:.1:1;
>> y = polyval(a,x);
>> noise = randn(1,length(y));
>> noise = noise/norm(noise);
>> ey = y + noise;
>> aest = polyfit(x,ey,3);
>> disp([a;aest])
    0.7400    0.9700    1.1000    0.8600
    0.7166    0.9985    1.1751    0.8379
>> yest = polyval(aest,x);
>> figure
>> plot(x,y,'b--')
>> hold on
>> plot(x,ey,'r+')
>> plot(x,yest,'g')
>> legend('y','ey','yest')
>> norm((y-yest))^2
ans =
   0.0324
```



# Esercizio

La tabella seguente riporta le misure della densità relativa $\rho$ dell'aria a diverse altezze $h$.

| h (km) | 0 | 1.525 | 3.050 | 4.575 | 6.100 | 7.625 | 9.150 |
|---|---|---|---|---|---|---|---|
| $\rho$ | 1 | 0.8617 | 0.7385 | 0.6292 | 0.5328 | 0.4481 | 0.3741 |

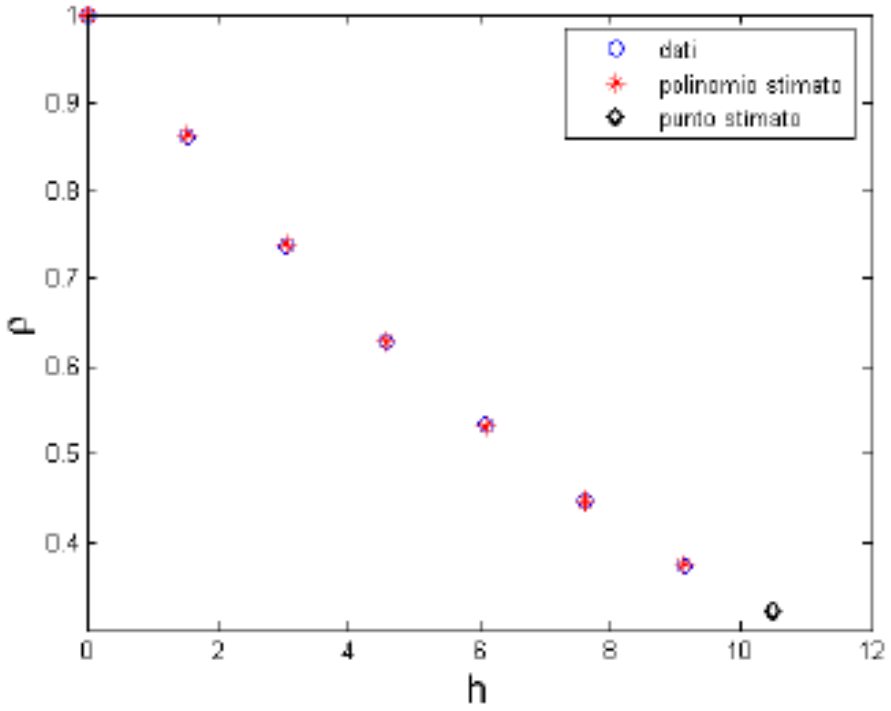
Si approssimi $\rho$ con un polinomio di secondo grado e si stimi il valore di $\rho$ in corrispondenza di $h = 10.5$ km.

## Soluzione

```
h = [0 1.525  3.050  4.575 6.100 7.625 9.150];
rho = [1 0.8617  0.7385  0.6292 0.5328 0.4481 0.3741];

a = polyfit(h,rho,2);
disp(a)

rho_pol = polyval(a,h);

rho_punto = polyval(a,10.5);

figure, plot(h,rho,'o')
hold on, plot(h,rho_pol,'r*')
hold on, plot(10.5,rho_punto,'kd')
xlabel('h')
ylabel('\rho')
legend('dati','polinomio stimato','punto stimato')
```

dal command window

```
>> esercizio_rhoaria
0.0028   -0.0934    0.9989
```



## In modo analogo

```
V = vandermonde(h,2);
H = V'*V;
B = V'*rho';

a = H\B;
disp(a)

a = fliplr(a')
rho_pol = polyval(a,h);

rho_punto = polyval(a,10.5);

figure, plot(h,rho,'o')
hold on, plot(h,rho_pol,'r*')
hold on, plot(10.5,rho_punto,'kd')
xlabel('h')
ylabel('\rho')
legend('dati','polinomio stimato','punto stimato')
```

Matrice dei coefficienti e vettore dei termini noti del sistema delle equazioni normali

dal command window

```
>> esercizio_rhoaria
    0.9989
   -0.0934
    0.0028
a =
    0.0028   -0.0934    0.9989
```