Implementazione dei metodi illustrati
nel corso di
Fondamenti di Calcolo Numerico
A.A. 2016-2017

Pietro Pennestrì
Matr. 1694905

Indice

Capitolo 1

Soluzione di Equazioni non Linerari

1.1 Metodo di bisezione

1.1.1 Pseudocodice

Algorithm 1: Algoritmo di bisezione per soluzione un'equazione non lineare

Input: Funzione **f**, intervallo ricerca radice $[a, b]$, tolleranza `TOL`, numero max iterazioni `NMAX`
Condizioni: $a < b$, $f(a) < 0$ $f(b) > 0$ oppure $f(a) > 0$ e $f(b) < 0$
Output: Stima soluzione con una differenza da $f(x) = 0$ inferiore a `TOL`

Function *Bisezione (f,a,b,TOL,NMAX)*
  $N \leftarrow 1$;
  while $N < NMAX$ **do**
    $c \leftarrow \frac{a+b}{2}$;
    if $f(c) = 0$ *or* $(b-a)/2 < TOL$ **then**
      Output(c) ; `// Soluzione trovata`
      Stop
    end
    $N \leftarrow N + 1$; `// Incrementa contatore`
    if sign$f(c) =$ sign$f(a)$ **then**
      $a \leftarrow c$;
    else
      $b \leftarrow c$; `// Nuovo intervallo`
    end
  end
  Output(`Ricerca soluzione fallita: superato massimo numero iterazioni`);

1.1.2 Implementazione in Matlab

Main

```
%  @author  Pietro Pennestri'
%  @websit http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0
A=[5 2 7 4 9 6]
Aeven = A(1:2:end)
Aodd  = A(2:2:end)
```

Funzione accessoria

```
function [y] = f(x)
    y = x^3 - 2;
end
```

Procedura

```
function [ r,iter ] = bisezione( f, a, b, N, tol )
%-------------------------------------------------------
% Funzione che implementa il metodo di bisezione
%-------------------------------------------------------
% InpUTS
% f: funzione con l'equazione da risolvere
% a,b: estremi dell'intervallo di ricerca
% CRITERI DI ARRESTO
% N: numero massimo di iterazioni
% tol: tolleranza
% Outputs
% r: soluzione (approx)
% k: numero di iterazioni
%-------------------------------------------------------
    % Controlla se la radice coincide con uno degli
        estremi
    if ( f(a) == 0 )
        r = a;
    iter=0;
        return;
    elseif ( f(b) == 0 )
        r = b;
    iter=0;
        return;
    % controlla che l'intervallo sia opportuno
    elseif ( f(a) * f(b) > 0 )
        error( 'ERRORE: Scegliere un intervallo opportuno
            ' );
    end
```

```
28
29      % Inizia la ricerca della radice
30      for k = 1:N
31          % Trova il punto medio dell'intervallo
32          c = (a + b)/2;
33
34          % Decidi quale intervallo scegliere
35          %           [a, c] se f(a) e f(c) hanno segni
                opposti, o
36          %           [c, b] se f(c) e f(b) hanno segni
                opposti.
37
38          if ( f(c) == 0 )
39              r = c;
40              iter=k;
41              return;
42          elseif ( f(c)*f(a) < 0 )
43              b = c;
44          else
45              a = c;
46          end
47
48          %controlla convergenza
49          if ( b - a < tol )
50              if ( abs( f(a) ) < abs( f(b) ) && abs( f(a) )
                    < tol )
51                  r = a;
52                  iter=k;
53                  return;
54              elseif ( abs( f(b) ) < tol )
55                  r = b;
56                  iter=k;
57                  return;
58              end
59          end
60      end
61
62      error( 'Il metodo non converge!' );
63  end
```

1.1.3 Approssimazione di $\sqrt{2}$

Un'approssimazione di $\sqrt{2}$ si può ottenere risolvendo con il metodo di bisezione l'equazione

$$f(x) \equiv x^2 - 2 = 0 \, . \tag{1.1}$$

La radice positiva di tale equazione costituir la soluzione. La Tabella 1.1 riassume i risultati ottenuti adottando tale metodo per differenti valori del criterio di tolleranza ε. L'intervallo iniziale di ricerca $[0, 3]$

Tabella 1.1: Approssimazioni di $\sqrt{2}$ con il metodo della bisezione

Tolleranza (ε)	Valore (approx)	Iterazioni
10^{-2}	1.412109375	8
10^{-4}	1.414215087890625	14
10^{-6}	1.414212942123413	21
10^{-8}	1.4142135623842478	28
10^{-10}	1.4142135622969363	34

1.2 Metodo di Newton

1.2.1 Pseudocode

Algorithm 2: Algoritmo di Newton per soluzione di un' equazione non lineare

Input: Funzione **f**, derivata funzione $\mathbf{f'}$, x_0 soluzione tentativo, tolleranza `TOL`, numero max iterazioni `NMAX`

Output: Stima soluzione con una differenza da $f(x) = 0$ inferiore a `TOL`

```
1  Function Newton (f,f',TOL,NMAX)
2      while |δ| < TOL or N =NMAX do
3          f_0 ← f(x_0) ;                          // Calcola funzione
4          f'_0 ← f'(x_0) ;                        // Calcola derivata
5          if |f'_0| < TOL then
6              Output(Derivata funzione troppo piccola)
7              Stop
8          end
9          δ ← f(x_0)/f'(x_0) ;                    // Calcola correzione
10         x_0 ← x_0 − δ ;                         // Aggiorna soluzione
11         N ← N + 1 ;                             // Incrementa contatore
12     end
13     Output(Ricerca soluzione fallita:  superato massimo numero
       iterazioni);
```

Per l'implementazione Matlab si faccia riferimento a quella più generale relativa ai sistemi di equazioni non lineari (v. p. 35).

Capitolo 2

Sistemi lineari

2.1 Metodo di Eliminazione di Gauss

2.1.1 Pseudocode

Algorithm 3: Algoritmo di eliminazione di Gauss

Input: Matrice quadrata $\mathbf{A}$, vettore termini noti $\mathbf{b}$
Output: Vettore soluzione $\mathbf{x}$ del sistema di equazioni $\mathbf{Ax} = \mathbf{b}$

Function *Gauss (* $\mathbf{A}, \mathbf{b}, \mathbf{x}$*)*
  Eliminazione gaussiana
  for $k = 1$ **to** $n - 1$ **do**
    for $i = k + 1$ **to** n **do**
      $a_{ik} \leftarrow \frac{a_{ik}}{a_{kk}}$;
      for $j = k + 1$ **to** n **do**
        $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$;
      end
    end
  end
  Eliminazione in avanti
  for $k = 1$ **to** $n - 1$ **do**
    for $i = k + 1$ **to** n **do**
      $b_i \leftarrow (b_i - a_{ik}b_k)$ (Soluzione all'indietro)
    end
  end
  for $i = n$ **to** 1 **by** -1 **do**
    $s \leftarrow b_i$;
    for $j = i + 1$ **to** n **do**
      $s \leftarrow (s - a_{ij}x_j)$;
    end
    $x_i \leftarrow \frac{s}{a_{ii}}$;
  end

ddd

2.1.2 Implementazione Matlab

`Main`

```
%  @author  Pietro Pennestri'
%  @websit http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0

clc

A=[1 2 -4; 2 -1 1; 1 1 3]; % matrice dei coefficienti
b=[3 ; 5; 8]; % vettore termini noti
tic
[At,x] = gauss_np(A,b)
toc

disp('----------RISULTATI----------')
disp('Matrice dei coefficienti A')
fprintf([repmat('%f\t', 1, size(A, 2)) '\n'], A');
disp('Vettore dei termini noti b')
fprintf([repmat('%f\t', 1, size(b, 2)) '\n'], b');

disp('Vettore soluzione x')
fprintf([repmat('%f\t', 1, size(x, 2)) '\n'], x');
disp('Matrice triangolare')
fprintf([repmat('%f\t', 1, size(At, 2)) '\n'], At');


fprintf('Tempo esecuzione %10e \n', toc) ;
test=A*x-b;
disp('Verifica il risultato test=A*x-b')
disp('Matrice test')
fprintf([repmat('%f\t', 1, size(test, 2)) '\n'], test');
disp('-----------------------------------')
```

`Procedura`

```
function [At,x] = gauss_np(A, b)
% Risolvi sistema di equazioni  Ax = b
% con eliminazione gaussiana senza pivot
% A e' una matrice quadrata di ordine n
% b e' un vettore di ordine n
% x e' il vettore soluzione
```

```
7
8  [n, n] = size(A);      % Determina la dimensione della
       matrice A
9  [n, k] = size(b);      % Determina
10 x = zeros(n,k);       % Initialize x
11 dd = abs(diag(A));
12 M = min(dd);
13 eps=1e-24;
14 if(M<eps)
15      msg = 'Presenza di zeri sulla diagonale principale!';
16      error(msg)
17 end
18 for i = 1:n-1
19      m = A(i+1:n,i)/A(i,i); % multipliers
20      A(i+1:n,:) = A(i+1:n,:) - m*A(i,:);
21      b(i+1:n,:) = b(i+1:n,:) - m*b(i,:);
22 end;
23 At=A; % matrice triangolare.
24 % Soluzione all'indietro
25 x(n,:) = b(n,:)/A(n,n);
26 for i = n-1:-1:1
27      x(i,:) = (b(i,:) - A(i,i+1:n)*x(i+1:n,:))/A(i,i);
28 end
```

```
----------RISULTATI----------------
Matrice dei coefficienti A
1.000000 2.000000 -4.000000
2.000000 -1.000000 1.000000
1.000000 1.000000 3.000000
Vettore dei termini noti b
3.000000
5.000000
8.000000
Vettore soluzione x
3.000000
2.000000
1.000000
Matrice triangolare
1.000000 2.000000 -4.000000
0.000000 -5.000000 9.000000
0.000000 0.000000 5.200000
Tempo esecuzione 5.732533e-03
Verifica il risultato test=A*x-b
Matrice test
0.000000
0.000000
```

```
0.000000
---------------------------------
```

2.2 Metodo di Gauss-Seidel

Lo pseudocode del metodo di Gauss-Seidel è sviluppato immaginando di accedere ai singoli elementi degli array.

In notazione matriciale la formula per il calcolo del vettore soluzione è la seguente:

$$\mathbf{x}^{(k)} = (\mathbf{D} - \mathbf{L})^{-1} \left(\mathbf{U}\mathbf{x}^{(k-1)} + \mathbf{b} \right) \tag{2.1}$$

con $\mathbf{D}$, $-\mathbf{L}$ e $-\mathbf{U}$ matrice diagonale, triangolare inferiore e triangolare superiore di $\mathbf{A}$.

2.2.1 Pseudocodice

Algorithm 4: Algoritmo di Gauss-Seidel

Input: Matrice quadrata $\mathbf{A}$ di ordine N, vettore termini noti $\mathbf{b}$, soluzione iniziale $\mathbf{x}^o$, `TOL` criterio convergenza, `NMAX` numero massimo iterazioni

Output: Vettore soluzione $\mathbf{x}$ del sistema di equazioni $\mathbf{A}\mathbf{x} = \mathbf{b}$

Function *Gauss-Seidel (* $\mathbf{A}, \mathbf{b}, \mathbf{x}, \mathbf{x}^o$*,TOL,NMAX)*

Verifica condizione di convergenza del metodo ;
$k \leftarrow 1$;
while $k \leq NMAX$ **do**
  for $i = 1$ **to** N **do**
    $$x_i^{(k)} = \frac{1}{a_{ii}} \left[\sum_{j=1}^{i-1} \left(a_{ij} x_j^{(k)} \right) - \sum_{j=i+1}^{n} \left(a_{ij} x_j^{(k-1)} \right) + b_i \right]$$
  end
  if $\left| \mathbf{x}^{(k)} - \mathbf{x}^{k-1} \right| < TOL$ **then**
    Stop
  end
  $k \leftarrow k + 1$
end
Output(`Ricerca soluzione fallita: superato massimo numero iterazioni`);

2.2.2 Implementazione Matlab

Main

```
%  @author  Pietro Pennestri'
%  @websit http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0

clc

% A=[1 2 -4; 2 -1 1; 1 1 3]; % matrice dei coefficienti
A=[4 -1 -1 ;-2 6 1;-1 1 7]
D=diag(diag(A))
U=triu(A)-D
L=tril(A)-D
C=-inv(D)*(L+U)
out=eig(C)
n=length(out);
Nmax=1000;
tol=1e-4;


tt=[];
for i=1:n
    normaelemento=abs(out(i));
tt=[tt;
    normaelemento];
end
test=max(tt)
if (test<1)
    b=[3 ; 9; -6]; % vettore termini noti
    x0=[0;0;0]; % vettore di prima iterazioneNmax=100;
    tol=1e-4;
    tic
    [x,iter] = gauss_seidel(A,b,x0,Nmax,tol);
    toc

    disp('----------RISULTATI----------')
    disp('Matrice dei coefficienti A')
    fprintf([repmat('%f\t', 1, size(A, 2)) '\n'], A');
    disp('Vettore dei termini noti b')
    fprintf([repmat('%f\t', 1, size(b, 2)) '\n'], b');

    disp('Vettore soluzione x')
    fprintf([repmat('%f\t', 1, size(x, 2)) '\n'], x');

```

```
44      fprintf('Tempo esecuzione %10e \n', toc) ;
45      test=A*x-b;
46      disp('Verifica il risultato test=A*x-b')
47      disp('Matrice test')
48      fprintf([repmat('%f\t', 1, size(test, 2)) '\n'], test
            );
49      disp('-----------------------------------')
50  else
51      error('Raggio spettrale maggiore di 1')
52  end
```

Procedura

```
1   function [x,iter] = gauss_seidel(A, b, x0, itermax, tol )
2   % ------------------------------------------------------
3   % Funzione che implementa il metodo iterativo
4   % di Gauss-Seidel
5   %-------------------------------------------------------
6   % Inputs
7   % A, b: matrice e termine noto, rispettivamente
8   % x0 : soluzione iniziale
9   % tol : tolleranza calcoli
10  % itermax: massimo numero iterazioni
11  %
12  % Outputs
13  % x : vettore soluzione
14  % iter: numero delle iterazioni
15  %-------------------------------------------------------
16      n = length(b);
17      x=x0;
18      iter=0;
19      res=1+tol;
20      while (res>tol & iter<itermax)
21          for i=1:n
22              sum=0;
23              for j=1:n
24                  if(j~=i)
25                      sum=sum+A(i,j)*x(j);
26                  end
27              end
28              x(i)=(-sum+b(i))/A(i,i);
29          end
30          res=norm(A*x-b);
31          iter=iter+1;
32      end
33      if (res<tol)
34          disp('il metodo giunge a convergenza')
```

```
35          return;
36      else
37          error('il metodo non giunge a convergenza')
38      end
```

```
   Test Numerico Gauss-Seidel

A =
     4    -1    -1
    -2     6     1
    -1     1     7
D =
     4     0     0
     0     6     0
     0     0     7
U =
     0    -1    -1
     0     0     1
     0     0     0
L =
     0     0     0
    -2     0     0
    -1     1     0
C =
         0    0.2500    0.2500
    0.3333         0   -0.1667
    0.1429   -0.1429         0
out =
   -0.4295
    0.2820
    0.1474
test =
    0.4295
il metodo giunge a convergenza
Elapsed time is 0.005683 seconds.
----------RISULTATI----------------
Matrice dei coefficienti A
4.000000 -1.000000 -1.000000
-2.000000 6.000000 1.000000
-1.000000 1.000000 7.000000
Vettore dei termini noti b
3.000000
9.000000
-6.000000
Vettore soluzione x
1.000000
2.000001
```

```
-1.000000
Tempo esecuzione 7.185072e-03
Verifica il risultato test=A*x-b
Matrice test
-0.000002
0.000004
0.000000
-----------------------------------
```

2.3 Metodo di Jacobi

Lo pseudocode del metodo di Jacobi è sviluppato immaginando di accedere ai singoli elementi degli array.

In notazione matriciale la formula per il calcolo del vettore soluzione è la seguente:

$$\mathbf{x}^{(k)} = \mathbf{D}^{-1}(\mathbf{U} + \mathbf{L})\,\mathbf{x}^{(k-1)} + \mathbf{D}^{-1}\mathbf{b} \tag{2.2}$$

con $\mathbf{D}$, $-\mathbf{L}$ e $-\mathbf{U}$ matrice diagonale, triangolare inferiore e triangolare superiore di $\mathbf{A}$.

2.3.1 Implementazione Matlab

Main

```
1  %  @author  Pietro Pennestri'
2  %  @websit http://www.pennestri.me
3  %  @email pietro.pennestri@gmail.com
4  %  @version  1.0
5
6  clc
7
8  % A=[1 2 -4; 2 -1 1; 1 1 3]; % matrice dei coefficienti
9  A=[4 -1 -1 ;-2 6 1;-1 1 7]
10 D=diag(diag(A))
11 U=triu(A)-D
12 L=tril(A)-D
13 C=-inv(D)*(L+U)
14 out=eig(C)
15 n=length(out);
16 Nmax=1000;
17 tol=1e-4;
18
19 tt=[];
20 for i=1:n
21     normaelemento=abs(out(i));
22 tt=[tt;
```

```
23      normaelemento];
24  end
25  test=max(tt)
26  if (test<1)
27      b=[3 ; 9; -6]; % vettore termini noti
28      x0=[0;0;0]; % vettore di prima iterazioneNmax=100;
29      tol=1e-4;
30      tic
31      [x,iter] = jacobi(A,b,x0,Nmax,tol);
32      toc
33
34      disp('----------RISULTATI----------')
35      disp('Matrice dei coefficienti A')
36      fprintf([repmat('%f\t', 1, size(A, 2)) '\n'], A');
37      disp('Vettore dei termini noti b')
38      fprintf([repmat('%f\t', 1, size(b, 2)) '\n'], b');
39
40      disp('Vettore soluzione x')
41      fprintf([repmat('%f\t', 1, size(x, 2)) '\n'], x');
42
43      fprintf('Tempo esecuzione %10e \n', toc) ;
44      test=A*x-b;
45      disp('Verifica il risultato test=A*x-b')
46      disp('Matrice test')
47      fprintf([repmat('%f\t', 1, size(test, 2)) '\n'], test
            ');
48      disp('-----------------------------------')
49  else
50      error('Raggio spettrale maggiore di 1')
51  end
```

Procedura

```
1   function [x,iter] = gauss_seidel(A, b, x0, itermax, tol )
2   %-----------------------------------------------------
3   % Funzione che implementa il metodo iterativo
4   % di Jacobi
5   %-----------------------------------------------------
6   % Inputs
7   % A, b: matrice e termine noto, rispettivamente
8   % x0 : soluzione iniziale
9   % tol : tolleranza calcoli
10  % itermax: massimo numero iterazioni
11  %
12  % Outputs
13  % x : vettore soluzione
14  % iter: numero delle iterazioni
```

```
15 %-------------------------------------------------------------------
16       n = length(b);
17       x=x0;
18       x_new=x0;
19       iter=0;
20       res=1+tol;
21       while (res>tol & iter<itermax)
22           for i=1:n
23               sum=0;
24               for j=1:n
25                   if(j~=i)
26                       sum=sum+A(i,j)*x(j);
27                   end
28               end
29               x_new(i)=(-sum+b(i))/A(i,i);
30           end
31           x=x_new;
32           res=norm(A*x-b);
33           iter=iter+1;
34       end
35       if (res<tol)
36           disp('il metodo giunge a convergenza')
37           return;
38       else
39           error('il metodo non giunge a convergenza')
40       end
```

```
   Test Numerico Jacobi

A =

     4    -1    -1
    -2     6     1
    -1     1     7


D =

     4     0     0
     0     6     0
     0     0     7


U =

     0    -1    -1
     0     0     1
     0     0     0


L =

     0     0     0
    -2     0     0
    -1     1     0


C =

         0    0.2500    0.2500
    0.3333         0   -0.1667
    0.1429   -0.1429         0


out =

   -0.4295
    0.2820
    0.1474


test =
```

```
    0.4295

il metodo giunge a convergenza
Elapsed time is 0.008528 seconds.
----------RISULTATI----------------
Matrice dei coefficienti A
4.000000 -1.000000 -1.000000
-2.000000 6.000000 1.000000
-1.000000 1.000000 7.000000
Vettore dei termini noti b
3.000000
9.000000
-6.000000
Vettore soluzione x
1.000004
1.999993
-1.000003
Tempo esecuzione 1.072181e-02
Verifica il risultato test=A*x-b
Matrice test
0.000027
-0.000052
-0.000035
-----------------------------------
```

2.4 Metodo di rilassamento S.O.R.

2.4.1 Pseudocodice

Algorithm 5: Algoritmo di rilassamento S.O.R.

Input: Matrice quadrata $\mathbf{A}$ di ordine N, vettore termini noti $\mathbf{b}$, soluzione iniziale $\mathbf{x}^o$, ω parametro, `TOL` criterio convergenza, `NMAX` numero massimo iterazioni

Output: Vettore soluzione $\mathbf{x}$ del sistema di equazioni $\mathbf{Ax} = \mathbf{b}$

Function *SOR(* $\mathbf{A}, \mathbf{b}, \mathbf{x}, \mathbf{x}^o$*, NMAX, TOL,* ω*)*
  Verifica condizione di convergenza del metodo ;
  $k \leftarrow 1$;
  while $k \leq NMAX$ **do**
    for $i = 1$ **to** N **do**
      $v_i^{(k)} = \frac{1}{a_{ii}} \left(-\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(k)} + b_i \right)$;
      $x_i^{(k+1)} = \omega v_i^{(k)} + (1 - \omega)\, x_i^{(k)}$
    end
  end
  if $\left|\mathbf{x}^{(k)} - \mathbf{x}^{k-1}\right| < TOL$ **then**
    Stop
  end
  $k \leftarrow k + 1$ Output(`Ricerca soluzione fallita: superato massimo numero iterazioni`);

2.4.2 Implementazione Matlab

`Main`

```
% @author Pietro Pennestri'
% @websit http://www.pennestri.me
% @email pietro.pennestri@gmail.com
% @version 1.0
clc
% A=[1 2 -4; 2 -1 1; 1 1 3]; % matrice dei coefficienti
A=[4 -1 -1 ;-2 6 1;-1 1 7]
D=diag(diag(A))
U=triu(A)-D
L=tril(A)-D
C=-inv(D)*(L+U)
out=eig(C)
n=length(out);
Nmax=1000;
tol=1e-4;
tt=[];
for i=1:n
```

```
18      normaelemento=abs(out(i));
19  tt=[tt;
20      normaelemento];
21  end
22  test=max(tt)
23  if (test<1)
24      b=[3 ; 9; -6]; % vettore termini noti
25      x0=[0;0;0]; % vettore di prima iterazioneNmax=100;
26      tol=1e-4;
27      omega=0.5;
28      tic
29      [x,iter] = sor(A,b,x0,Nmax,tol,omega);
30      toc
31
32      disp('----------RISULTATI----------')
33      disp('Matrice dei coefficienti A')
34      fprintf([repmat('%f\t', 1, size(A, 2)) '\n'], A');
35      disp('Vettore dei termini noti b')
36      fprintf([repmat('%f\t', 1, size(b, 2)) '\n'], b');
37
38      disp('Vettore soluzione x')
39      fprintf([repmat('%f\t', 1, size(x, 2)) '\n'], x');
40
41      fprintf('Tempo esecuzione %10e \n', toc) ;
42      test=A*x-b;
43      disp('Verifica il risultato test=A*x-b')
44      disp('Matrice test')
45      fprintf([repmat('%f\t', 1, size(test, 2)) '\n'], test
            ');
46      disp('-----------------------------------------')
47  else
48      error('Raggio spettrale maggiore di 1')
49  end
```

Procedura

```
1   function [x,iter] = gauss_seidel(A, b, x0, itermax, tol,
        omega)
2   % ------------------------------------------------------
3   % Funzione che implementa il metodo iterativo
4   % Rilassamento - SOR
5   %-------------------------------------------------------
6   % Inputs
7   % A, b: matrice e termine noto, rispettivamente
8   % x0 : soluzione iniziale
9   % tol : tolleranza calcoli
10  % itermax: massimo numero iterazioni
```

```
% omega: parametro per migliorare co
%
% Outputs
% x : vettore soluzione
% iter: numero delle iterazioni
%-------------------------------------------------------
    n = length(b);
    x=x0;
    iter=0;
    res=1+tol;
    while (res>tol & iter<itermax)
        for i=1:n
            sum=0;
            for j=1:n
                if(j~=i)
                    sum=sum+A(i,j)*x(j);
                end
            end
            v(i)=(-sum+b(i))/A(i,i);
            x(i)=omega*v(i)+(1-omega)*x0(i);
        end
        x0=x;
        res=norm(A*x-b);
        iter=iter+1;
    end
    if (res<tol)
        disp('il metodo giunge a convergenza')
        return;
    else
        error('il metodo non giunge a convergenza')
    end
```

```
   Test numerico SOR

A =

     4    -1    -1
    -2     6     1
    -1     1     7


D =

     4     0     0
     0     6     0
     0     0     7


U =

     0    -1    -1
     0     0     1
     0     0     0


L =

     0     0     0
    -2     0     0
    -1     1     0


C =

         0    0.2500    0.2500
    0.3333         0   -0.1667
    0.1429   -0.1429         0


out =

   -0.4295
    0.2820
    0.1474


test =
```

```
    0.4295

il metodo giunge a convergenza
Elapsed time is 0.011770 seconds.
----------RISULTATI----------------
Matrice dei coefficienti A
4.000000 -1.000000 -1.000000
-2.000000 6.000000 1.000000
-1.000000 1.000000 7.000000
Vettore dei termini noti b
3.000000
9.000000
-6.000000
Vettore soluzione x
0.999982
1.999983
-1.000000
Tempo esecuzione 1.480024e-02
Verifica il risultato test=A*x-b
Matrice test
-0.000056
-0.000066
0.000002
-----------------------------------
```

2.5 Fattorizzazione Cholesky

Se $\mathbf{A}$ è una matrice simmetrica definita positiva essa può essere fattoreizzata come segue

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T \tag{2.3}$$

con $\mathbf{L}$ matrice triangolare inferiore.

La soluzione del sistema $\mathbf{A}\mathbf{x} = \mathbf{b}$ si ottiene come segue:

$$\mathbf{L}\mathbf{y} = \mathbf{b} \tag{2.4}$$

$$\mathbf{L}^T\mathbf{x} = \mathbf{y} \tag{2.5}$$

2.5.1 Pseudocodice

Algorithm 6: Algoritmo di Cholesky

Input: Matrice simmetrica definita positiva $\mathbf{A}$ di ordine N
Output: Matrice triangolare inferiore $\mathbf{L}$ tale che $\mathbf{LL}^T = \mathbf{A}$

Function *Cholesky(* $\mathbf{A}, \mathbf{L}$*)*
  $l_{11} \leftarrow \sqrt{a_{11}}$;
  for $i = 2$ **to** N **do**
    for $j = 1$ **to** N **do**
      $$l_{ij} \leftarrow \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right)$$
      $$l_{ii} \leftarrow \sqrt{a_{ii} - \sum_{k=1}^{j-1} l_{ik}^2}$$
    end
  end

Procedura

```
function [h]=cholesky(a)
%-----------------------------------------
% INPUT
% a: matrice simmetrica definita positiva
%
% OUTPUT
% l: matrice triangolare inferiore tale  che l*l'=a
%
%
%-----------------------------------------
test=isPositiveDefinite(a); % Controlla che la matrice
    sia simmetrica
%                                 positiva definita
n=size(a);
h(1,1)=sqrt(a(1,1));
for i=2:n,
for j=1:i-1,
s=0;
for k=1:j-1,
s=s+h(i,k)*h(j,k);
end
h(i,j)=1/h(j,j)*(a(i,j)-s);
end
h(i,i)=sqrt(a(i,i)-sum(h(i,1:i-1).^2));
end
return
```

Procedura accessoria

```
function x=isPositiveDefinite(A)
%Function to check whether a given matrix A is positive definite
%Author Mathuranathan for http://www.gaussianwaves.com
% Modified by Pietro Pennestri' for use in Cholesky algorithm
%Returns x=1, if the input matrix is positive definite
%Returns x=0, if the input matrix is not positive definite
%Throws error if the input matrix is not symmetric
    %Check if the matrix is symmetric
    [m,n]=size(A);
    if m~=n,
        error('A is not Symmetric');
    end
    %Test for positive definiteness
    x=1; %Flag to check for positiveness
    for i=1:m
        subA=A(1:i,1:i); %Extract upper left kxk submatrix
        if(det(subA)<=0); %Check if the determinent of the kxk submatrix is +ve
            x=0;
            break;
        end
    end
    if x
%       display('Given Matrix is Positive definite');
    else
        error('Given Matrix is NOT positive definite');
    end
end
```

2.5.2 Verifica dei Risultati

```
A =

     2    -1     0
    -1     2    -1
     0    -1     2

Elapsed time is 0.005613 seconds.
----------RISULTATI----------------
Matrice L
1.414214 0.000000 0.000000
```

```
-0.707107 1.224745 0.000000
0.000000 -0.816497 1.154701
Tempo esecuzione 6.205881e-03
Verifica il risultato [T]=[L]*[L]^t
Matrice T
2.000000 -1.000000 0.000000
-1.000000 2.000000 -1.000000
0.000000 -1.000000 2.000000
----------------------------------
```

2.6 Fattorizzazione LU

Sia **A** una matrice quadrata, la decomposizione LU con siste nel calcolo delle matrice trangolare inferiore L e triangolare superiore U tali che

$$\mathbf{A} = \mathbf{LU} \tag{2.6}$$

Nella versione semplificata, gli elementi delle due matrici si ottengono attraverso le formule

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} u_{kj} l_{ik} \tag{2.7}$$

e

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} u_{kj} l_{ik} \right) \tag{2.8}$$

Per evitare divisioni di elementi con lo zero si introduce una matrice di permutazione **P** che rende gli elementi sulla diagonale quelli massimi su una colonna.

In tal modo

$$\mathbf{PA} = \mathbf{LU} \tag{2.9}$$

La soluzione del sistema $\mathbf{Ax} = \mathbf{b}$ si riscrive nella forma

$$\mathbf{LUx} = \mathbf{b} \tag{2.10}$$

Il sistema si risolve in due fasi

$$\mathbf{Lz} = \mathbf{b} \tag{2.11}$$

e

$$\mathbf{Ux} = \mathbf{z} \tag{2.12}$$

2.6.1 Pseudocodice

Algorithm 7: Fattorizzazione LU (senza pivoting)

Input: Matrice quadrata **A** di ordine N
Output: Matrici **L** e **U**

Function *LU(* $\mathbf{A}, \mathbf{L}, \mathbf{U}$*)*
  for $k = 1$ **to** N **do**
    $l_{kk} \leftarrow 1$;
    for $j = k$ **to** N **do**
      $u_{kj} \leftarrow a_{kj} - \sum_{s=1}^{k-1} l_{ks} u_{sj}$;
    end
    for $i = k + 1$ **to** N **do**
      $l_{ik} \leftarrow \dfrac{\left(a_{ik} - \sum_{s=1}^{k-1} l_{is} u_{sk} \right)}{u_{kk}}$
    end
  end

2.6.2 Implementazione MATLAB

```
%  @author  Pietro Pennestri'
%  @websit http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0

% main_LU.m main che richiama la funzione LUdolittle.m

% matrice da fattorizzare
clc
A=[ 3  2  -1  2;
      -3  -4  2  1;
       6  2  -2  11;
      -6  -10  6  2]
 tic
 [ L,U] = LUdolittle( A );
 toc
 T=L*U;
disp('----------RISULTATI----------')
disp('Matrice L')
fprintf([repmat('%f\t', 1, size(L, 2)) '\n'], L');
disp('Matrice U')
fprintf([repmat('%f\t', 1, size(U, 2)) '\n'], U');
fprintf('Tempo esecuzione %10e \n', toc) ;
```

```
disp('Verifica il risultato [T]=[L]*[U]')
disp('Matrice T')
fprintf([repmat('%f\t', 1, size(T, 2)) '\n'], T');
disp('-----------------------------------------')
```

```
function [ L,U] = LUdolittle( A )
% ------------------------------------------------------
% Funzione che implementa il metodo di Dolittle
% per la fattorizzazione LU senza pivot
%-------------------------------------------------------
% Input: Matrice quadrata A non singolare
% Output:
% L,U: Matrici tali che A=LU
%-------------------------------------------------------
n=length(A);
U=zeros(n,n);
L=eye(n);
for k=1:n
    for m=k:n
        U(k,m)=A(k,m);
        if k>1
        U(k,m)=A(k,m)-dot(L(k,1:(k-1)),U(1:(k-1),m));
        end
    end
    for i=k+1:n
       L(i,k)=(A(i,k))/U(k,k);
        if k>1
        L(i,k)=(A(i,k)-dot(L(i,1:(k-1)),U(1:(k-1),k)))/U(k
            ,k);
        end
    end
    end
end
```

2.6.3 Verifica dei Risultati

```
A =

     3     2    -1     2
    -3    -4     2     1
     6     2    -2    11
    -6   -10     6     2

Elapsed time is 0.001110 seconds.
----------RISULTATI----------------
Matrice L
1.000000 0.000000 0.000000 0.000000
```

```
-1.000000 1.000000 0.000000 0.000000
2.000000 1.000000 1.000000 0.000000
-2.000000 3.000000 -1.000000 1.000000
Matrice U
3.000000 2.000000 -1.000000 2.000000
0.000000 -2.000000 1.000000 3.000000
0.000000 0.000000 -1.000000 4.000000
0.000000 0.000000 0.000000 1.000000
Tempo esecuzione 3.118932e-03
Verifica il risultato [T]=[L]*[U]
Matrice T
3.000000 2.000000 -1.000000 2.000000
-3.000000 -4.000000 2.000000 1.000000
6.000000 2.000000 -2.000000 11.000000
-6.000000 -10.000000 6.000000 2.000000
---------------------------------
```

Capitolo 3

Calcolo Autovalori

3.1 Metodo della Potenza

Algorithm 8: Metodo della potenza

Input: Matrice quadrata $\mathbf{A}$, stima autovettore $\mathbf{x^{(0)}}$, N numero max iterazioni, `tol` criterio tolleranza

Output: Stime dell'autovalore massimo λ e relativo autovettore $\mathbf{x}$

$k \leftarrow 0$
while $r > tol$ *and* $k < N$ **do**
  $k \leftarrow k + 1$
  $\mathbf{y}^{(k)} \leftarrow \mathbf{A}\mathbf{x}^{(k-1)}$
  $\lambda_k \leftarrow \|\mathbf{y}^{(k)}\|$
  $\mathbf{x}^{(k)} \leftarrow \mathbf{y}^{(k)}/\lambda_k$
end

3.1.1 Implementazione Matlab

`Main`

```
%  @author  Pietro Pennestri'
%  @website http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0

% main_potenze.m main che richiama la funzione potenze.m
A=[1 -1 2;-2 0 5;6 -3 6]
u0=[1;1;1]
tol=1.e-15 % tolleranza
N=200 % numero massimo di iterazione
tic
[lambda1,it2,iter] = potenze(A,u0,tol,N)
```

```
13 toc
```

Procedura

```
1  function [lambda1,it2,iter] = potenze(A,u0,tol,nmax)
2  % Metodo della potenza
3  % Parametri input
4  % A: Matrice quadrata
5  % u0: Stima iniziale autovettore
6  % tol: Parametro tolleranza
7  % nmax: Numero massimo di iterazioni
8  % Parametri output
9  % lambda1 : Stima autovalore
10 % it2: Stima autovettore
11 % iter: Numero iterazioni
12 it2=u0;
13 lambda1=0; % assegno valore arbitrario, che non entra
       nella successione
14 res=1+tol;
15 iter=0;
16 while (res>tol & iter<nmax)
17     lambda0=lambda1;
18     it1=it2;
19     it1=(it1)/norm(it1);
20     it2=A*it1;
21     it2=(it2);
22     lambda1=(it2(1))/(it1(1));
23     if (iter == 0)
24         iter=iter+1;
25     else
26         res=abs(lambda1-lambda0);
27         iter=iter+1;
28     end
29
30 end
31
32 if (res < tol)
33     return;
34 else
35       disp('-Errore---------')
36 end
37
38
39 end
```

3.1.2 Test numerico

```
A =

     1    -1     2
    -2     0     5
     6    -3     6


u0 =

     1
     1
     1


tol =

   1.0000e-15


N =

   200


lambda1 =

    5.0000


it2 =

    1.0164
    3.2525
    3.6590


iter =

    66

Elapsed time is 0.010656 seconds.
```

Capitolo 4

Sistemi di equazioni non lineari

4.1 Metodo di Newton-Raphson

4.1.1 Pseudocodice

Algorithm 9: Algoritmo di Newton-Raphson per soluzione di sistemi equazioni non lineari

Input: Sistema di equazioni non lineari $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, matrice Jacobiana $\mathbf{J}$, $\mathbf{x}_0$ soluzione tentativo, tolleranza `TOL`, numero max iterazioni `NMAX`

Output: Stima soluzione $\mathbf{x}_k$ con una differenza da $|\mathbf{f}(\mathbf{x}_k) - \mathbf{f}(\mathbf{x})|$ inferiore a `TOL`

Function *NewtonRaphson (*$\mathbf{f}$,$\mathbf{J}$,$\mathbf{x}_0$*, TOL,NMAX)*
  while $|\delta| <$ *TOL or* $N =$*NMAX* **do**
    $\mathbf{f}_0 \leftarrow \mathbf{f}(\mathbf{x}_0)$; `// Valuta funzioni`
    $\delta \leftarrow ||\mathbf{f}_0||$; `// Calcola norma vettore`
    $\mathbf{J}_0 \leftarrow \mathbf{J}(x_0)$; `// Calcola Jacobiano`
    if $\mathbf{J}_0$ *singolare* **then**
      Output(Jacobiana singolare)
      Stop
    end
    $\Delta\mathbf{x} \leftarrow \mathbf{J}_0^{-1}\mathbf{f}_0$; `// Calcola correzione`
    $\mathbf{x}_0 \leftarrow \mathbf{x}_0 - \Delta\mathbf{x}$; `// Aggiorna soluzione`
    $N \leftarrow N + 1$; `// Incrementa contatore`
  end
  Output(`Ricerca soluzione fallita: superato massimo numero iterazioni`);

4.1.2 Implementazione in Matlab

Main

```
%
% Soluzione di sistemi di equazioni non lineari mediante
    iterazione di Newton-Raphson
%
clc; clear;
% Soluzione di primo tentativo
x0=[0.5;-0.3];
str = sprintf('Soluzione di primo tentativo: %.1f  %.1f',
    x0);
disp(str);
% Criterio convergenza
tol=1e-6;
str = sprintf('Tolleranza: %.2e',tol);
disp(str);
% Massimo numero di iterazioni
itmax=100;
str = sprintf('Massimo numero di iterazioni: %i',itmax);
disp(str);
str= sprintf ( '\n\n-----Metodo Newton-Raphson-----\n\n');
disp(str);
% Metodo di Newton-Raphson per trovare la soluzione del
    sistema di
% equazioni non lineari
tic
[x,iterazioni,funzioni,residui,exitflag]=newtonraphson(
    @fun,@jacobian,x0,tol,itmax);
toc
str = sprintf('Vettore Soluzione = %.2e %.2e',x);
disp(str);
str = sprintf('Iterazioni = %.i',iterazioni);
disp(str);
str = sprintf('Residui della funzione = %.3e %.3e',
    funzioni);
disp(str);
str = sprintf('Massima norma dei residui = %.3e',residui)
    ;
disp(str);
fprintf('Tempo esecuzione %10e \n', toc) ;
```

Funzioni accessorie

```
function f=fun(x)
%
% INPUT
```

```
% x: Vettore delle incognite
%
% OUTPUT
% f: Vettore dei residui
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
f(1)= x(1)^2+x(2)^2-1;
f(2)= x(1)-x(2);
f=f'; % Vettore f in colonna
end
```

```
function J = jacobian(x)
%
% INPUT
% x: Vettore delle incognite
%
% OUTPUT
% J: Matrice Jacobiana
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
J(1,1)= 2*x(1);
J(1,2)= 2*x(2);
J(2,1)= 1;
J(2,2)= -1;
end
```

Procedura

```
function [x,iterations,f,resid,exitflag]=newtonraphson(
    fun,jacobian,x0,tol,itmax)
%
% INPUT
% fun : Function in cui vengono valutati gli elementi del
     vettore dei residui
% jacobian : Function in cui vengono valutati gli
    elementi della matrice jacobiana
% x0: Vettore soluzione di primo tentativo
% tol: Criterio convergenza
% itmax: Numero massimo di iterazioni
%
% OUTPUT
% x: Soluzione
% iterations: Numero di iterazioni eseguite
% f: Vettore dei reidui
% resid: Norma del vettore dei residui
% exitflag:  1 L'iterazione converge sul valore x, 0
%            0 Si e' superato il massimo numero di
    iterazioni
% Inizializza le variabili
```

```
18 resid=1000;
19 iterations=0;
20 while(resid>tol)
21     % Calcola matrice Jacobiana
22     J=feval(jacobian,x0);
23     % Calcola residui
24     f=feval(fun,x0);
25     % Calcola norma vettore dei residui
26     resid=norm(f);
27     % Trova il nuovo valore di x con il metodo di Newton-
           Raphson
28     x = x0-inv(J)*f; % inv(J)  J \
29     % Controllo sulla tolleranza
30     if(resid<tol)
31         exitflag=1;
32         return
33     end
34     % Controllo sul numero delle iterazioni
35     if(iterations>itmax)
36         exitflag=0;
37         return
38     end
39     % Assegan il nuovo valore come condizioni iniziali
           per la successiva
40     % iterazione
41     x0=x;
42     iterations=iterations+1;
43 end
44 end
```

4.1.3 Test numerico

```
Soluzione di primo tentativo: 0.5  -0.3
Tolleranza: 1.00e-06
Massimo numero di iterazioni: 100


----Metodo Newton-Raphson----


Elapsed time is 0.002181 seconds.
Vettore Soluzione = 7.07e-01 7.07e-01
Iterazioni = 7
Residui della funzione = 4.890e-12 0.000e+00
Massima norma dei residui = 4.890e-12
Tempo esecuzione 3.688650e-03
```

Capitolo 5

Approssimazione dati e funzioni

5.1 Interpolazione polinomiale di Lagrange

5.1.1 Pseudocodice

Algorithm 10: Interpolazione di Lagrange

Input: Dati due vettori $\mathbf{x} = \left\{ \begin{array}{cccc} x_1 & x_2 & \dots & x_{N+1} \end{array} \right\}$ e $\mathbf{y} \left\{ \begin{array}{cccc} y_1 & y_2 & \dots & y_{N+1} \end{array} \right\}$ di $N+1$ componenti. Si assume che non sia $y_i \neq y_j$ se $x_i = x_j$

Output: Coefficienti del polinomio interpolatore $\mathbf{c} = \left\{ \begin{array}{cccc} c_1 & c_2 & \dots & c_{N+1} \end{array} \right\}$ Il polinomio interpolatore sarà $f(x) = \sum_{i=1}^{N+1} L_i(x)$ dove $L_i(x) = c_i \prod_{\substack{j=1 \\ i \neq j}}^{N+1} (x - x_j)$

```
1  Function LagrangeInterpolazione (x,y,c)
2      for i = 1 to N + 1 do
3          product ← 1;
4          for j = 1 to N + 1 do
5              if i ≠ j then
6                  product ← product · (x_i − x_j)
7              end
8          end
9          c_i ← y_i / product
10     end
```

5.1.2 Implementazione in Matlab

Main

```
%  @author  Pietro Pennestri'
%  @websit http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0

clc
%y=sin(3*x)
x=[1 1.3 1.6 1.9 2.2];
y=[0.1411 -0.6878 -0.9962 -0.5507 0.3115];
tic
[c] = laginterp(x,y)
npoints=100
t= linspace(0,pi,npoints);
[l,f] = polilagrange(c ,x,1.5)
for i=1:npoints
    [l,f] = polilagrange(c ,x,t(i))
    fun(i)=f;
    funext(i)=sin(3*t(i));
end
plot(t,fun)
hold on
plot(t,funext)
hold on
scatter(x,y)
hold off

legend('Pol. Lagrange','sin(3x)','Dati')
toc
```

laginterp.m

```
function [ c ] = laginterp(x,y)
%-------------------------------------------------
% Funzione che calcola i coefficienti
% del polinomio di Lagrange
%-------------------------------------------------
% Input: vettori [x],[y]
% Output:[c] vettore coefficienti
% il polinomio f(x) di Lagrange e' ottenuto
% f(x)= sum(c_i * prod((x-x_j)))
%-------------------------------------------------
ns=size(x);
n=ns(2)-1;
for i=1:(n+1)
```

```
14      pr=1;
15      for j=1:(n+1)
16          if(i~=j)
17              pr=pr*(x(i)-x(j));
18          end
19      end
20      c(i)=(y(i))/(pr);
21  end
22  end
```

polilagrange.m

```
1   function [l,f] = polilagrange(c ,x,t)
2   % ---------------------------------------------------
3   % Funzione che valuta il polinomio di Lagrange f
4   % ed i polinomi accessori nell'ascissa t
5   %----------------------------------------------------
6   % Input: vettori [x],[c] | t
7   % Output:vettore [l] (valore dei polinomi accessori) f=
        sum(l)
8   %----------------------------------------------------
9   ns=size(x);
10  n=ns(2)
11  for i=1:n
12  p=1;
13  for j=1:n
14      if(i~=j)
15      p=p*(t-x(j));
16      end
17  end
18  l(i)=c(i)*p;
19  end
20  f=sum(l);
21  end
```

5.1.3 Esempio Numerico

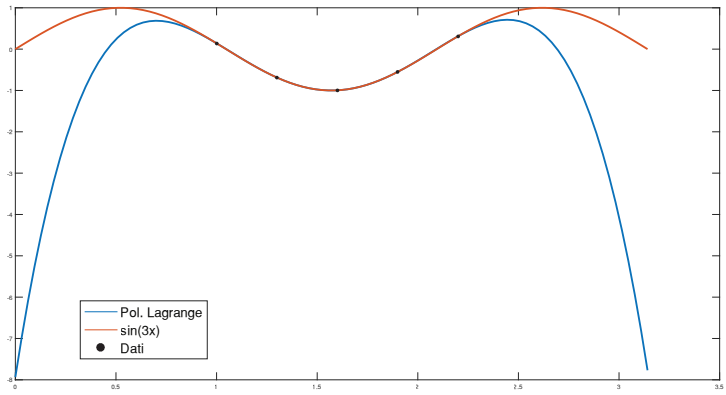

Figura 5.1: Approssimazione della funzione $\sin(3x)$ con polinomi di Lagrange su 5 punti

Capitolo 6

Integrazione

6.1 Metodo dei trapezi

6.1.1 Pseudocodice

Algorithm 11: Algoritmo di quadratura con formula dei trapezi

Input: Funzione $y = f(x)$ da integrare nell'intervallo $[a, b]$, numero N di subintervalli (pari)

Output: Stima $\int_a^b f(x)$

Function *Trapezio (f,a,b,N)*

$S = \sum_{i=1}^{N+1} f(x_i) - \left(\frac{f(a)+f(b)}{2}\right)$;

$\Delta x = \frac{b-a}{N}$;

`area` $\leftarrow S\Delta x$

Output(`area`);

6.1.2 Implementazione in Matlab

`Main metodo dei trapezi`

```
% @author Pietro Pennestri'
% @websit http://www.pennestri.me
% @email pietro.pennestri@gmail.com
% @version 1.0
%
% Integrazione con formula dei trapezi
n = 5                                   % Numero punti
a = -2                                  % Coordinata
    estremo sinistro intervallo integrazione
b = 2                                   % Coordinata estremo
     destro intervallo integrazione
```

```
10 tic
11 x = linspace(a, b, n) ;                  % Genera valori
       delle ascisse punti intervallo
12 y=computefun(x)                            % Calcola i
       valori della funzione nei punti dell'intervallo
13                                          %
14 sum_y = sum(y) - (y(1) + y(end)) ./ 2;   % Applica la
       formula dei trapezi
15 dx = (b - a) / (n - 1) ;                 % Calcola
       ampiezza intervalli elementari (base trapezio)
16 area = sum_y * dx;                       % Calcola area
17 toc
18 fprintf('Metodo dei trapezi \n') ;
19 fprintf('Estremi integrazione %10e %10e: \n', a,b) ;
20 fprintf('Numero intervalli %i \n', n) ;
21 fprintf('Stima integrale %10e: \n', area) ;
22 fprintf('Tempo esecuzione %10e \n', toc) ;
```

Funzione da integrare

```
1 function y = computefun(x)
2 n=length(x);
3 for i=1:n
4 y(i) = x(i)^2-4;
5 %y(i)=1./(1+x(i));
6 end
7 end % End
```

6.1.3 Esempio numerico

```
n =

     5


a =

    -2


b =

     2
```

```
y =

     0    -3    -4    -3     0

Elapsed time is 0.010722 seconds.
Metodo dei trapezi
Estremi integrazione -2.000000e+00 2.000000e+00:
Numero intervalli 5
Stima integrale -1.000000e+01:
Tempo esecuzione 2.029400e-02
```

6.2 Metodo di Cavalieri-Simpson

6.2.1 Pseudocodice

Algorithm 12: Algoritmo di quadratura con formula dei Cavalieri-Simpson

Input: Funzione $y = f(x)$ da integrare nell'intervallo $[a, b]$, numero N di subintervalli (pari)

Output: Stima $\int_a^b f(x)$

Function *Trapezio (f,a,b,N)*

$S = y_1 + 4y_2 + 2y_3 + 4y_4 + 2y_5 + \ldots + 2y_{N-1} + 4y_N + y_{N+1}$;

$\Delta x = \frac{b-a}{N}$;

area $\leftarrow S\frac{\Delta x}{3}$

Output(area);

Main metodo di Cavalieri-Simpson

```
%  @author  Pietro Pennestri'
%  @websit http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0
%
% Integrazione con formula di Simpson
clear all
n = 5
% Numero punti (dispari)
a = -2                                  % Coordinata
    estremo sinistro intervallo integrazione
b = 2                                  % Coordinata estremo
     destro intervallo integrazione
x = linspace(a, b, n)                   % Genera valori
    delle ascisse punti intervallo
```

```
13 y=computefun(x)                          % Calcola i valori
       della funzione nei punti dell'intervallo
14 tic
15 A=y(2:n-1);   % Vettore y con estremi esclusi
16 yeven = A(1:2:end); % Estrai elementi con indici dispari
17 yodd  = A(2:2:end); % Estrai elementi con indici pari
18 sum_y = (y(1)+2*sum(yodd)+4*sum(yeven)+y(n))    ;  %
       Applica la formula dei trapezi
19 dx = (b - a) / (n - 1) ;                  % Calcola
       ampiezza intervalli elementari (base trapezio)
20 area = sum_y * dx/3.;                        % Calcola area
21 toc
22 fprintf('Metodo di Simpson \n') ;
23 fprintf('Estremi integrazione %10e %10e: \n', a,b) ;
24 fprintf('Numero intervalli %i \n', n) ;
25 fprintf('Stima integrale %10e: \n', area) ;
26 fprintf('Tempo esecuzione %10e \n', toc) ;
```

Funzione da integrare

```
1 function y = computefun(x)
2 n=length(x);
3 for i=1:n
4 y(i) = x(i)^2-4;
5 %y(i)=1./(1+x(i));
6 end
7 end % End
```

```
n =

     5


a =

    -2


b =

     2


x =

    -2    -1     0     1     2
```

```
y =

     0    -3    -4    -3     0

Elapsed time is 0.001671 seconds.
Metodo di Simpson
Estremi integrazione -2.000000e+00 2.000000e+00:
Numero intervalli 5
Stima integrale -1.066667e+01:
Tempo esecuzione 3.901000e-03
```

6.3 Metodo di Romberg

6.3.1 Pseudocodice

Algorithm 13: Algoritmo di quadratura di Romberg

Input: Funzione $y = f(x)$ da integrare nell'intervallo $[a, b]$, numero positivo N intero

Output: Stima $\int_a^b f(x)$ e stima errore

Function *romberg (f,a,b,err,area,N)*
  for $j = 1, 2, \ldots, N$ **do**
    $m \leftarrow 2^{j-1}$;
    $h \leftarrow \dfrac{b-a}{m}$;
    $\mathbf{x} \leftarrow \{\ x_1 \quad x_2 \quad \ldots \quad x_{m+1}\ \}$;
    $\mathbf{f} \leftarrow \{\ f(x_1) \quad f(x_2) \quad \ldots \quad f(x_{m+1})\ \}$;
    $R_{j,1} \leftarrow \dfrac{h}{2}\left[f_1 + f_{m+1} + 2\sum_{i=2}^{m} f_i\right]$;
  end
  for $k = 2, \ldots, N$ **do**
    for $i = k, \ldots, N$ **do**
      $R_{i,k} \leftarrow \dfrac{4^{k-1} R_{i,k-1} - R_{i-1,k-1}}{4^{k-1} - 1}$
    end
  end
  $err \leftarrow |R_{N,N-1} - R_{N,N}|$;
  area$\leftarrow R_{N,N}$;
  Output(area);

Main - Metodo di Romberg

```
% @author Pietro Pennestri'
% @websit http://www.pennestri.me
% @email pietro.pennestri@gmail.com
% @version 1.0
```

```
%
% Integrazione con formula di Romber
clear all
n = 4
% Estremi di integrazione
a = -2
b = 2
tic
[R,area,err,h] = romber(@computefun,a,b,n);
toc
fprintf('Metodo di Romber \n') ;
fprintf('Estremi integrazione %10e %10e: \n', a,b) ;
fprintf('Numero righe tabella %i \n', n) ;
fprintf('Tabella di Romberg \n') ;
R
fprintf('Stima integrale %10e: \n',area ) ;
fprintf('Tempo esecuzione %10e \n', toc) ;
```

Procedura metodo di Romberg

```
function [R,quad,err,h] = romber(f,a,b,n)
%
%-------------------------------------------------------------------------
%ROMBER  Quadratura con il metodo di Romberg
% Inputs
%      f:      procedura per valutare funzione da integrare
%    a,b:    estremi intervallo integrazione
%     n :     massimo numero di righe nella tabella
% Return
%   R:      tabella di Romberg
%   quad    stima integrale
%   err     errore nella stima integrale
%   h       ampiezza intervallo impiegato
%
%
%-------------------------------------------------------------------------

% Prima colonna
%--------------------
R=zeros(n,n);
for j=1:n,
m=2^(j-1);
h=(b-a)/m;
x=linspace(a,b,m+1);
ff=f(x);
```

```
23 R(j,1)=(h/2)*(ff(1)+ff(end)+2*sum(ff(2:end-1))); %
       Applica Trapezi
24 end;
25 %-----------------------
26 % Colonne successive
27 %-----------------------
28 for k=2:n,
29 for i=k:n,
30 R(i,k)=(4^(k-1)*R(i,k-1)-R(i-1,k-1))/(4^(k-1)-1);
31 end;
32 end;
33 %
34 %% Stima Errore
35   err = abs(R((n),(n-1))-R(n,n));
36 % end
37 quad = R(n,n);
38 end % End
```

Funzione da integrare

```
1 function y = computefun(x)
2 n=length(x);
3 for i=1:n
4 y(i) = x(i)^2-4;
5 %y(i)=1./(1+x(i));
6 end
7 end % End
```

```
n =

     4


a =

    -2


b =

     2

Elapsed time is 0.048486 seconds.
Metodo di Romberg
Estremi integrazione -2.000000e+00 2.000000e+00:
Numero righe tabella 4
Tabella di Romberg
```

```
R =

         0         0         0         0
   -8.0000  -10.6667         0         0
  -10.0000  -10.6667  -10.6667         0
  -10.5000  -10.6667  -10.6667  -10.6667

Stima integrale -1.066667e+01:
Tempo esecuzione 5.873700e-02
```

Capitolo 7

Equazioni differenziali

7.1 Metodo di Eulero

7.1.1 Pseudocodice

Algorithm 14: Algoritmo di integrazione di Eulero

Input: Funzione $y' = f(t, y(t))$, condizione iniziale $y_0 = y(t_0)$, passo di integrazione h, N numero di passi

Output: Funzione integrale discreta $\mathbf{y} = \{ \; y_0 \quad y_1 \quad \dots \quad y_N \; \}$

for $i = 0, 1, 2, \dots, N-1$ **do**
  $t_i \leftarrow t_0 + ih$
  $y_{i+1} \leftarrow y_i + hf(t_i, y_i)$
end

Main - Metodo di Eulero

```
%  @author  Pietro Pennestri'
%  @websit http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0
%
% Integrazione equazioni differenziali
% con metodo di Eulero
clear all
close all
t0=0  % Tempo iniziale
t1=2  % Tempo finale
y0=1  % Valore iniziale
n=100 % Numero intervalli
tic
[y,t]=eulero1(@fun,n,t0,t1,y0);
toc
h=(t1-t0)/n;
```

```
for i=1:n+1
t(i)=t0+h*(i-1);
yprime(i)=fun(t(i),y(i));
end
V=[t',y']
yyaxis left
plot(t,y,'LineWidth',2);
ylabel('y')
yyaxis right
plot(t,yprime,'--','LineWidth',2)
ylabel('dy/dt')
grid on
title('Metodo Eulero - Curve integrale (y) e derivata (dy/dt)')
legend('y ','dy/dt','Location','northeast')
xlabel('t') % x-axis label
```

Metodo di Eulero

```
function [y,t]=euler1(fun,n,t0,t1,y0)
% fun : Funzione da integrare
% n : numero di passi di integrazione
% t_0, t_1 : Valore iniziale e finale del parametro indipendente
% y_0 : Valore iniziale
h=(t1-t0)/n; % Passo integrazione
t(1)=t0;
y(1)=y0;
for i=1:n
t(i+1)=t(i)+h;
y(i+1)=y(i)+h*fun(t(i),y(i));
end;
end  %End of function
```

Equazione differenziale da integrare $y' = \left(t^2 - y^2\right) \sin y$

```
function yprime=fun(t,y)
yprime=(t^2-y^2)*sin(y);
end % End of function
```

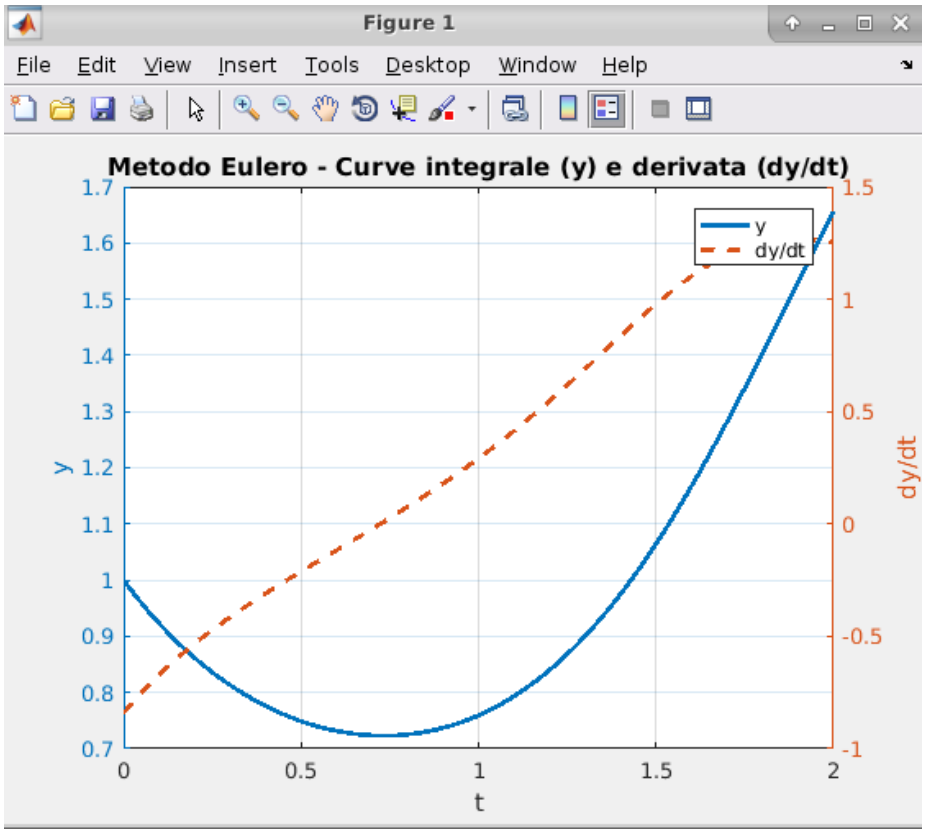

Figura 7.1: Metodo di Eulero

7.2 Metodo di Heun

7.2.1 Pseudocodice

Algorithm 15: Algoritmo di integrazione di Heun

Input: Funzione $y' = f(t, y(t))$, condizione iniziale $y_0 = y(t_0)$, passo di integrazione h, N numero di passi

Output: Funzione integrale discreta $\mathbf{y} = \{ \; y_0 \quad y_1 \quad \dots \quad y_N \; \}$

for $i = 0, 1, 2, \dots, N-1$ **do**

  $t_{i+1} \leftarrow t_i + h$

  $y_r \leftarrow y_i + hf(t_i, y_i)$

  $y_{i+1} \leftarrow y_i + \left(\frac{f(t_i, y_i) + f(t_i + h, y_r)}{2} \right) h$

end

Main - Metodo di Heun

```
%  @author  Pietro Pennestri'
%  @websit http://www.pennestri.me
%  @email pietro.pennestri@gmail.com
%  @version  1.0
%
% Integrazione equazioni differenziali
% con metodo di Heun
clear all
close all
clc
t0=0  % Tempo iniziale
t1=2  % Tempo finale
y0=1  % Valore iniziale
n=100 % Numero intervalli
tic
[y,t]=heun1(@fun,n,t0,t1,y0);
toc
h=(t1-t0)/n;
for i=1:n+1
t(i)=t0+h*(i-1);
yprime(i)=fun(t(i),y(i));
end
V=[t',y']
yyaxis left
plot(t,y,'LineWidth',2);
ylabel('y')
yyaxis right
plot(t,yprime,'--','LineWidth',2)
ylabel('dy/dt')
grid on
```

```
31 title('Metodo Heun - Curve integrale (y) e derivata (dy/
       dt)')
32 legend('y ','dy/dt','Location','northeast')
33 xlabel('t') % x-axis label
```

Metodo di Heun

```
1  function [y,t]=heun1(fun,n,t0,t1,y0)
2  % fun : Funzione da integrare
3  % n : numero di passi di integrazione
4  % t_0, t_1 : Valore iniziale e finale del parametro
       indipendente
5  % y_0 : Valore iniziale
6  h=(t1-t0)/n; % Passo integrazione
7  t(1)=t0;
8  y(1)=y0;
9  for i=1:n
10 t(i+1)=t(i)+h;
11 sl=fun(t(i),y(i)); % Coeff. angolare estremo sx
12 yr=y(i)+sl*h;
13 sr=fun(t(i+1),yr); %Coeff. angolare estremo dx
14 y(i+1)=y(i)+h*(sl+sr)/2.;
15 end;
16 end %End of function
```

Equazione differenziale da integrare $y' = \left(t^2 - y^2\right) \sin y$

```
1  function yprime=fun(t,y)
2  yprime=(t^2-y^2)*sin(y);
3  end % End of function
```

7.3 Metodo di Runge

7.3.1 Pseudocodice

Algorithm 16: Algoritmo di integrazione di Runge

Input: Funzione $y' = f(t, y(t))$, condizione iniziale $y_0 = y(t_0)$, passo di integrazione h, N numero di passi

Output: Funzione integrale discreta $\mathbf{y} = \{ \; y_0 \quad y_1 \quad \dots \quad y_N \; \}$

for $i = 0, 1, 2, \dots, N-1$ **do**
  $t_{i+1} \leftarrow t_i + h$
  $t_h \leftarrow t_i + \frac{h}{2}$
  $y_h \leftarrow y_i + f(t_i, y_i) \frac{h}{2}$
  $y_{i+1} \leftarrow y_i + f(t_h, y_h) h$
end

Main - Metodo di Runge

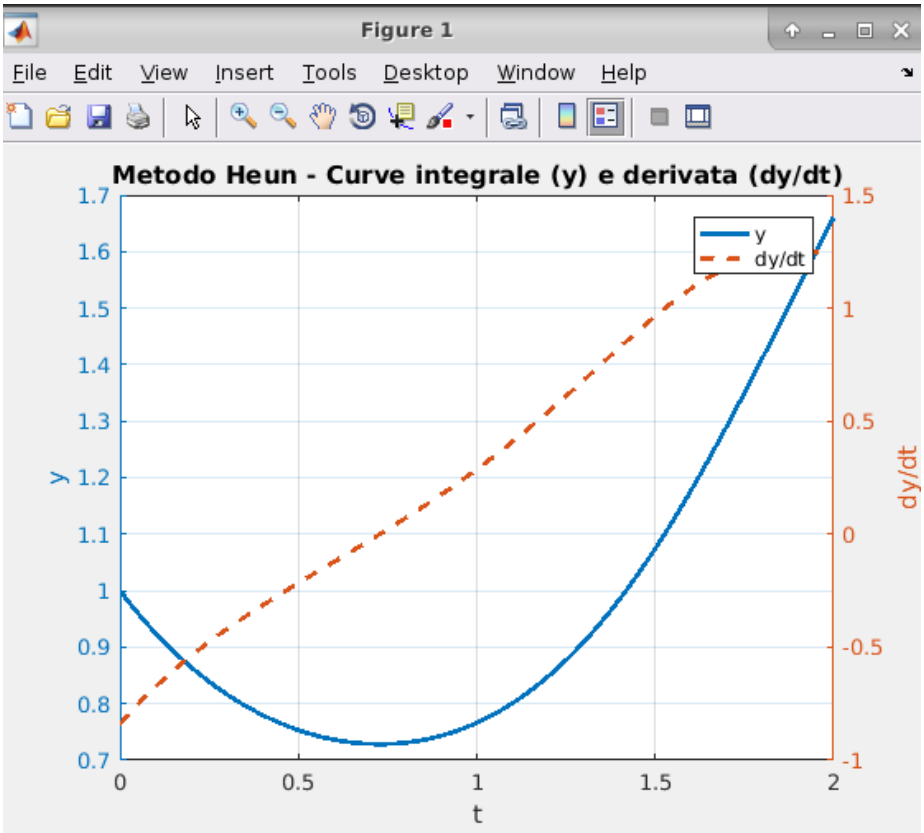

Figura 7.2: Metodo di Heun

```
% @author Pietro Pennestri'
% @websit http://www.pennestri.me
% @email pietro.pennestri@gmail.com
% @version 1.0
%
% Integrazione equazioni differenziali
% con metodo di Eulero
clear all
close all
t0=0 % Tempo iniziale
t1=2 % Tempo finale
y0=1 % Valore iniziale
n=100 % Numero intervalli
tic
[y,t]=euler1(@fun,n,t0,t1,y0);
toc
h=(t1-t0)/n;
for i=1:n+1
t(i)=t0+h*(i-1);
yprime(i)=fun(t(i),y(i));
end
V=['t','y']
yyaxis left
plot(t,y,'LineWidth',2);
ylabel('y')
yyaxis right
plot(t,yprime,'--','LineWidth',2)
ylabel('dy/dt')
grid on
title('Metodo Eulero - Curve integrale (y) e derivata (dy
    /dt)')
legend('y ','dy/dt','Location','northeast')
xlabel('t') % x-axis label
```

Metodo di Runge

```
function [y,t]=runge1(fun,n,t0,t1,y0)
% fun : Funzione da integrare
% n : numero di passi di integrazione
% t_0, t_1 : Valore iniziale e finale del parametro
    indipendente
% y_0 : Valore iniziale
h=(t1-t0)/n; % Passo integrazione
t(1)=t0;
y(1)=y0;
for i=1:n
t(i+1)=t(i)+h;
```

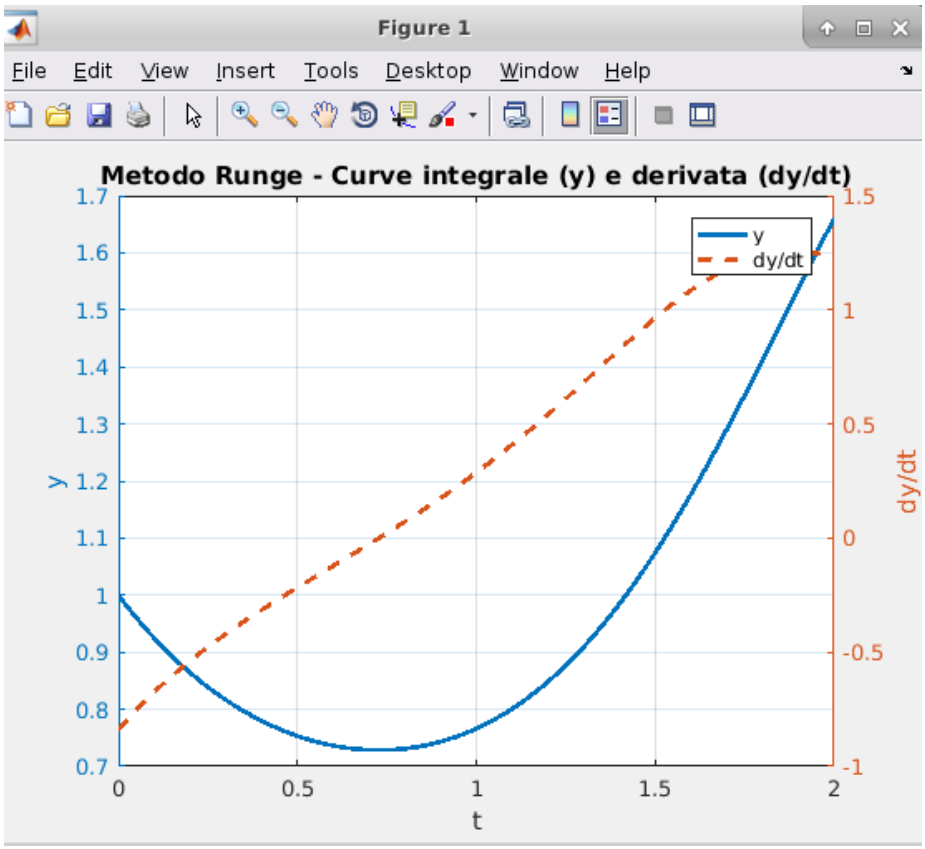

Figura 7.3: Metodo di Runge

```
y_h=y(i)+fun(t(i),y(i))*h/2.;
t_h=t(i)+h/2.;
s_h=fun(t_h,y_h); % Coeff. angolare punto mezzeria
y(i+1)=y(i)+h*s_h;
end;
end  %End of function
```

Equazione differenziale da integrare $y' = (t^2 - y^2)\sin y$

```
function yprime=fun(t,y)
yprime=(t^2-y^2)*sin(y);
end % End of function
```

7.4 Confronto tempi di calcolo

I tempi rilevati sono quelli riportati in Tabella 7.1.

Tabella 7.1: Tempi di esecuzione dei metodi di integrazione equazioni differenziali

Metodo	Tempo (sec)
Eulero	0.013506
Heun	0.016390
Runge	0.028113