UNIVERSITA' DEGLI STUDI DI CAGLIARI

DIPARTIMENTO DI INGEGNERIA
ELETTRICA ED ELETTRONICA

Algoritmi iterativi per sistemi lineari ed equazioni non lineari

Tesina del corso di Calcolo Numerico 2

Anno Accademico
2010/2011

Studenti:
Gianluca Santini (38056)
Alessandro Varsi (38190)

Relatore:
prof. Giuseppe Rodriguez

Indice

Capitolo 1

Introduzione teorica

1.1 Metodi iterativi per la risoluzione di sistemi lineari

Un sistema di n equazioni lineari in n incognite si può rappresentare come

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ \qquad\qquad ... \\ \qquad\qquad ... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases} \tag{1.1}$$

Oppure in forma matriciale, più compatta

$$Ax = b \tag{1.2}$$

dove $A = (a_{ij})_{i,j=1}^{n}$ è la matrice dei coefficienti, $b = (b_1, ..., b_n)^T$ è il vettore dei termini noti e $x = (x_1, ..., x_n)^T$ è il vettore soluzione. Per risolvere un sistema lineare così definito si possono usare sia metodi diretti che metodi iterativi. Quest'ultimi sono oggetto della nostra trattazione, e se non diversamente specificato ci si riferirà sempre a metodi iterativi.

Un metodo iterativo è caratterizzato dalla risoluzione del sistema in un numero indefinito di passi, a differenza dei metodi diretti. Un metodo di questo tipo parte da un vettore iniziale $x^{(0)}$ e genera una successione di vettori $\left\{x^{(k)}\right\}$ con $k = 0, 1, 2, ...$ che al limite deve poter fornire un'approssimazione adeguata della soluzione x. Si dice che un metodo iterativo è globalmente convergente se per ogni vettore iniziale $x^{(0)} \epsilon \mathbb{R}^n$ si ha che

$$\lim_{k\to\infty} \left\| x^{(k)} - x \right\| = 0 \tag{1.3}$$

Un metodo si dice consistente se

$$x^{(k)} = x \Longrightarrow x^{(k+1)} = x \tag{1.4}$$

Questa è una condizione necessaria, ma non sufficiente, per la convergenza.

I metodi che analizzeremo sono lineari stazionari e del primo ordine. Essi possono essere scritti in forma generale come:

$$x^{(k+1)} = Bx^{(k)} + f \tag{1.5}$$

Si dice lineare perché la legge che lo esprime è di tipo lineare, stazionario perché B, la matrice di iterazione, e il vettore f non dipendono dall'indice di iterazione k. Infine è del primo ordine perché il vettore aggiornato dipende esclusivamente dal vettore dell'iterazione precedente.

Una condizione necessaria e sufficiente per la convergenza di un metodo iterativo della forma (1.5) è $\rho(B) < 1$, dove l'operatore ρ indica il raggio spettrale della matrice, ovvero l'autovalore di modulo massimo.

Nell'implementazione di un metodo iterativo occorre tenere conto di opportuni criteri di arresto che permettano di interrompere il calcolo. Nei casi studiati si sono utilizzati tre criteri di arresto. Il primo riguarda lo scarto tra due iterazioni successive e, data una tolleranza $\tau > 0$, può essere rappresentato come:

$$\left\| x^{(k)} - x^{(k-1)} \right\| \leq \tau \left\| x^{(k)} \right\| \tag{1.6}$$

Il secondo criterio riguarda il numero massimo consentito di iterazioni e si è dunque utilizzato un criterio del tipo $k > N_{max}$. Il terzo invece riguarda l'utilizzo del residuo[1] normalizzato, e può essere utilizzato in luogo del criterio (1.6). Questo criterio si può esprimere come:

$$\left\| r^{(k)} \right\| \leq \tau \left\| b \right\| \tag{1.7}$$

Il criterio (1.6) ha carattere generale, mentre il (1.7) vale solo per sistemi lineari.

1.1.1 Metodi di Jacobi e di Gauss-Seidel

Tramite la tecnica dello splitting additivo si può esprimere la matrice A con la relazione

$$A = P - N \tag{1.8}$$

dove la matrice di precondizionamento P è non singolare. Sostituendo questa relazione nel metodo (1.5) si ricava l'espressione:

$$x^{(k+1)} = P^{-1}Nx^{(k)} + P^{-1}b \tag{1.9}$$

Si può considerare il particolare splitting additivo $A = D - E - F$ dove:

$$D_{ij} = \begin{cases} a_{ii}, & i = j \\ 0, & i \neq j \end{cases}, E_{ij} = \begin{cases} -a_{ij}, & i > j \\ 0, & i \leq j \end{cases}, F_{ij} = \begin{cases} -a_{ij}, & i < j \\ 0, & i \geq j \end{cases} \tag{1.10}$$

Si parla di metodo di Jacobi quando $P = D$ e $N = E + F$, e sostituendo queste due relazioni nella (1.9) si ottiene la forma matriciale del metodo. In alternativa si può esprimere il metodo di Jacobi in forma di coordinate vettoriali, ovvero:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right], i = 1, ..., n \tag{1.11}$$

Questa formula consente di calcolare le componenti di $x^{(k+1)}$ in qualsiasi ordine e indipendentemente l'una dall'altra, e perciò il metodo di Jacobi è parallelizzabile.

[1] Il residuo è definito come $r^{(k)} = b - Ax^{(k)}$

Il metodo di Gauss-Seidel invece è caratterizzato dalla scelta $P = D - E$ e $N = F$. Sostituendo queste relazioni nella (1.9) si ricava la versione matriciale del metodo, altrimenti, analogamente al metodo di Jacobi, si può ottenere una forma vettoriale:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right], i = 1, ..., n \tag{1.12}$$

Si nota dalla formula stessa che questo metodo, a differenza del precedente, non è parallelizzabile.

Se la matrice A è a predominanza diagonale stretta, o irriducibilmente diagonalmente dominante, allora i metodi di Jacobi e di Gauss-Seidel convergono. Se A è simmetrica definita positiva il metodo di Gauss-Seidel converge.

1.1.2 Metodo del gradiente coniugato precondizionato

Sia A una matrice simmetrica definita positiva. Si consideri la forma quadratica

$$\phi(y) = \frac{1}{2} y^T A y - y^T b. \tag{1.13}$$

Tale funzione è minima nel punto in cui si annulla il suo gradiente

$$\nabla \phi(y) = \frac{1}{2} \left(A + A^T \right) y - b = Ay - b = 0 \tag{1.14}$$

Da qui si vede che la minimizzazione di (1.13) è equivalente alla soluzione di un sistema lineare nella forma (1.2). La soluzione si ricava utilizzando un metodo iterativo non stazionario della forma

$$x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)} \tag{1.15}$$

dove α_k è il passo mentre $d^{(k)}$ è la direzione di decrescita. Risolvendo $\frac{d}{d\alpha} \phi \left(x^{(k+1)} \right) = 0$ troviamo il valore ottimale di α_k

$$\alpha_k = \frac{\left(d^{(k)} \right)^T r^{(k)}}{\left(d^{(k)} \right)^T A d^{(k)}} \tag{1.16}$$

Nel metodo del gradiente coniugato, le direzioni di discesa sono scelte in modo da mantenere l'ottimalità rispetto alle direzioni precedenti. Un vettore $x^{(k)}$ è ottimale rispetto a una direzione p se e solo se:

$$p^T r^{(k)} = 0 \tag{1.17}$$

Per far sì che l'ottimalità del vettore k-esimo, rispetto ad una certa direzione, venga mantenuta anche dai vettori successivi, si pone $x^{(k+1)} = x^{(k)} + q$. Affinché $x^{(k+1)}$ sia ottimale rispetto a p è necessario che risulti:

$$0 = p^T r^{(k+1)} = p^T \left(r^{(k)} - Aq \right) = -p^T A q \tag{1.18}$$

In tal caso le direzioni p e q si dicono A-coniugate. Come direzioni di discesa si considerano, dunque, direzioni del tipo $p^{(k+1)} = r^{(k+1)} - \beta_k p^{(k)}$. Le direzioni $p^{(k+1)}$ e $p^{(k)}$ saranno A-coniugate se:

$$\beta_k = \frac{\left(p^{(k)} \right)^T A r^{(k+1)}}{\left(p^{(k)} \right)^T A p^{(k)}} \tag{1.19}$$

Si dimostra che con questa scelta di β_k la direzione $p^{(k+1)}$ è A-coniugata a tutte quelle precedenti. Operando questa scelta per le $d^{(k)}$, la formula (1.15) diventa:

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)} \tag{1.20}$$

Queste condizioni ci impediscono di far diminuire ulteriormente la $\phi(y)$ e dimostrano che questo metodo converge alla soluzione esatta con un numero finito di iterazioni.

A causa della propagazione degli errori, è preferibile accelerare il metodo precondizionando il residuo con una matrice P che sia una buona approssimazione della A, ma che sia facile da invertire. La scelta di un buon precondizionatore è molto importante; nel nostro caso si è scelto come precondizionatore $P = R^T R$, dove la matrice R è calcolata dalla fattorizzazione di Cholesky incompleta[2].

1.2 Metodi iterativi per la risoluzione di equazioni non lineari

Un'equazione non lineare può essere espressa come:

$$f(x) = 0 \tag{1.21}$$

Risolvere un'equazione di questo tipo significa trovare le radici, ovvero gli zeri, della funzione $f(x)$. L'algoritmo iterativo parte da un punto iniziale x_0 e genera una successione che, sotto opportune condizioni, converge alla soluzione dell'equazione. Nel nostro lavoro si sono analizzati tre diversi metodi per risolvere equazioni non lineari.

1.2.1 Metodo di bisezione

Supponendo che la $f(x)$ abbia una sola radice in un intervallo $[a, b]$ e che cambi segno all'interno di esso una sola volta, si valuta il punto intermedio $c = \frac{a+b}{2}$. Se la funzione cambia segno in $[a, c]$, allora la soluzione è in questo intervallo. Viceversa, si trova in $[c, b]$. Per valutare se la funzione cambia segno, si procede analizzando il prodotto $f(a) * f(c)$: se esso è negativo significa che la radice è in $[a, c]$, se no è nell'altro intervallo.

Dunque si aggiorna l'intervallo e si procede iterativamente ricalcolandone il punto intermedio e rivalutando il segno. Al termine delle iterazioni il punto intermedio potrà essere una approssimazione della soluzione con una certa accuratezza.

Nel caso in cui $f(c) = 0$, significa che la radice coincide con il punto c stesso. In questo caso l'algoritmo riconosce di aver trovato la soluzione e si ferma.

1.2.2 Metodo di Newton

Sia la $f(x)$ una funzione continua in $[a, b]$ con un'unica soluzione al suo interno, fissato un punto iniziale x_0, si considera la retta tangente alla funzione in quel punto. L'intersezione tra la tangente e l'asse delle ascisse viene considerata come la nuova approssimazione della soluzione. Si procede così iterativamente fino a che lo scarto tra due approssimazioni successive scende al di sotto di una determinata soglia τ.

Per valutare la tangente si ricorre al fascio proprio di rette con centro $(x_k, f(x_k))$:

$$y - f(x_k) = m(x - x_k) \tag{1.22}$$

Per selezionare tra tutte le rette la tangente alla funzione in x_k, bisogna imporre che m sia pari al valore della derivata della funzione in quel punto. Calcolando la sua intersezione con l'asse delle ascisse si ottiene il metodo iterativo di Newton:

[2] Per un approfondimento su questo argomento si veda il paragrafo 5.4.1 di [1]

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{1.23}$$

La formula (1.23) è ovviamente valida se la derivata in x_k è non nulla per tutti gli x_k della successione.

1.2.3 Metodo delle secanti

Poiché il calcolo della derivata nella formula (1.23) può risultare svantaggioso dal punto di vista computazionale, esistono metodi iterativi che, seppur partendo dalla teoria del metodo di Newton, approssimano il coefficiente angolare m in modo diverso rispetto all'uso della derivata. Per questo motivo si dicono metodi quasi-Newton, e sono caratterizzati dall'iterazione generica:

$$x_{k+1} = x_k - \frac{f(x_k)}{m_k} \tag{1.24}$$

Uno di questi è il cosiddetto metodo delle secanti. Il coefficiente angolare a ogni passo viene approssimato con quello della retta secante la funzione nei due punti di ascissa x_k e x_{k-1}:

$$m_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \tag{1.25}$$

Sostituendo il valore (1.25) nella (1.24) si ottiene il metodo delle secanti:

$$x_{k+1} = \frac{x_{k-1} f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})} \tag{1.26}$$

Poiché dipende da due valutazioni della funzione $f(x)$ occorre inizializzare il metodo con due punti iniziali, x_0 e x_1, mentre Newton richiedeva un unico punto iniziale.

1.2.4 Sistemi di equazioni non lineari

Un sistema di n equazioni non lineari in n incognite

$$\begin{cases} f_1(x_1, ..., x_n) = 0 \\ f_2(x_1, ..., x_n) = 0 \\ \vdots \\ f_n(x_1, ..., x_n) = 0 \end{cases} \tag{1.27}$$

può essere scritto nella forma:

$$F(x) = 0 \tag{1.28}$$

Considerando $F(x)$ come funzione vettoriale si può trattare il problema di sistemi di equazioni non lineari estendendo i casi già sviluppati per le equazioni non lineari monodimensionali.

Per ricavare, ad esempio, il metodo di Newton multidimensionale, si sviluppa la $F(x)$ in serie di Taylor troncata al primo ordine:

$$F(x) \simeq F\left(x^{(k)}\right) + F'\left(x^{(k)}\right)\left(x - x^{(k)}\right) \tag{1.29}$$

dove il simbolo $F'(x^{(k)})$ denota la matrice Jacobiana:

$$F'\left(x^{(k)}\right) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}\left(x_1^{(k)},\ldots,x_n^{(k)}\right) & \cdots & \frac{\partial f_1}{\partial x_n}\left(x_1^{(k)},\ldots,x_n^{(k)}\right) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}\left(x_1^{(k)},\ldots,x_n^{(k)}\right) & \cdots & \frac{\partial f_n}{\partial x_n}\left(x_1^{(k)},\ldots,x_n^{(k)}\right) \end{bmatrix} \tag{1.30}$$

Per calcolare le approssimazioni successive è quindi sufficiente utilizzare il metodo iterativo:

$$x^{(k+1)} = x^{(k)} - \left(F'(x^{(k)})\right)^{-1} F\left(x^{(k)}\right) \tag{1.31}$$

cioè calcolare lo zero della (1.29).

Capitolo 2

Implementazione degli algoritmi

2.1 Algoritmi per sistemi lineari

2.1.1 Algoritmi di Jacobi e Gauss-Seidel

Si sono implementate in MATLAB® dapprima le versioni matriciali dei due algoritmi, e poi le versioni vettoriali, espresse dalle formule (1.11) e (1.12). Di seguito per completezza vengono riportati i listati dei programmi che sono stati sviluppati.

Le versioni matriciali dei due metodi sono:

```
% Algoritmo di Jacobi in forma matriciale
% [x,k]jacobi_m(A,b,x0,tau,Nmax)

function [x,k]=jacobi_m(A,b,x0,tau,Nmax)

P=diag(diag(A));
N=P-A;
k=1;
x=(P\N)*x0+(P\b);
while((k<Nmax) && (norm(x-x0)>tau*norm(x)))
        x0=x;
        x=(P\N)*x0+(P\b);
        k=k+1;
end
```

```
% Algoritmo di Gauss Seidel in forma matriciale
% [x,k]=GaussSeidel_m(A,b,x0,tau,Nmax)

function [x,k]=GaussSeidel_m(A,b,x0,tau,Nmax)

P=diag(diag(A))+tril(A)-diag(diag(tril(A)));
N=P-A;
k=1;
x=(P\N)*x0+(P\b);

```

```
11 while ((k<Nmax) && (norm(x-x0)>tau*norm(x)))
12         x0=x;
13         x=(P\N)*x0+(P\b);
14         k=k+1;
15 end
```

Le versioni matriciali implementano direttamente le formule richiamate nel capitolo 1, e dunque non è stato necessario nessun particolare accorgimento in fase di scrittura degli algoritmi. Le versioni vettoriali invece sono:

```
1  % Algoritmo di Jacobi in forma vettoriale
2  % [x,k]=jacobi(A,b,x0,tau,Nmax)
3
4  function [x,k]=jacobi(A,b,x0,tau,Nmax)
5
6  k=0;
7  n=size(A);
8  n=n(1:1);
9  temp=x0;
10 x=x0;
11
12 for i=1:n
13         aux=0;
14         for j=1:n
15                 if j~=i
16                         aux=aux+(A(i,j))*temp(j);
17                 end
18         end
19         x(i)=(b(i)-aux)/(A(i,i));
20 end
21
22 k=k+1;
23
24 while ((k<Nmax)&&(norm(x-temp)>tau*norm(x)))
25         temp=x;
26         for i=1:n
27                 aux=0;
28                 for j=1:n
29                         if j~=i
30                                 aux=aux+(A(i,j))*temp(j);
31                         end
32                 end
33         x(i)=(b(i)-aux)/(A(i,i));   % Calcolo del successivo vettore x
34         end
35 k=k+1;
36 end
```

```
1  % Algoritmo di Gauss-Seidel in forma di coordinate.
2  % [x,k]=GaussSeidel(A,b,x0,tau,Nmax)
3
```

```
4  function [x,k]=GaussSeidel(A,b,x0,tau,Nmax)
5
6  k=0;
7  n=size(A);
8  n=n(1:1);
9  x=zeros(n,1);
10 for i=1:n
11          aux=0;
12          for j=1:i-1
13                  aux=aux+(A(i,j)*x(j));
14          end
15          temp=0;
16          for j=i+1:n
17                  temp=temp+(A(i,j)*x0(j));
18          end
19          x(i)=(b(i)-aux-temp)/(A(i,i));
20 end
21
22 k=k+1;
23
24 while((k<Nmax) && (norm(x-x0)>tau*norm(x)) )
25          x0=x;
26          x=zeros(n,1);
27          for i=1:n
28                  aux=0;
29                  for j=1:i-1
30                          aux=aux+(A(i,j)*x(j));
31                  end
32                  temp=0;
33                  for j=i+1:n
34                          temp=temp+(A(i,j)*x0(j));
35                  end
36                  x(i)=(b(i)-aux-temp)/(A(i,i));
37          end
38 k=k+1;
39 end
```

Come si può notare dai codici riportati, sia per Jacobi che per Gauss-Seidel, si è resa necessaria un'iterazione al di fuori del ciclo while principale, questo per poter implementare, seguendo la sintassi richiesta dal while, il criterio di arresto (1.6).

2.1.2 Algoritmo del gradiente coniugato precondizionato

Il codice del programma[1] contenente il metodo del gradiente coniugato precondizionato è riportato di seguito:

```
1  % Algoritmo del Gradiente Coniugato Precondizionato
2  % [x,k]=gcp(A,b,x0,tau,Nmax)
```

[1] Il programma è stato scritto basandosi sull'Algoritmo 5.4 di [1].

```
3
4  function [x,k]=gcp(A,b,x0,tau,Nmax)
5
6  sigma=1e-3;
7  R=cholinc(A,sigma); % Fattorizzazione di Cholesky incompleta
8  P=R'*R;  % Precondizionatore
9  x=x0;
10 r=b-A*x;
11 z=P\r;
12 p=z;
13 k=0;
14
15 while ((k<Nmax) && (norm(r)>tau*norm(b)))
16           s=A*p;
17           delta=p'*s;
18           alpha=(p'*r)/delta;
19           x=x+alpha*p;
20           r=r-alpha*s;
21           z=P\r;
22           beta=(s'*z)/delta;
23           p=z-beta*p;
24           k=k+1;
25 end
```

Il calcolo $s = Ap$ è stato introdotto, rispetto al metodo del gradiente coniugato visto in 1.1.2, al fine di ridurre il costo computazionale riutilizzando il valore di s laddove sarebbe stato richiesto il prodotto matrice-vettore.

2.2 Algoritmi per equazioni non lineari

2.2.1 Algoritmi di bisezione, Newton e secanti

Si sono implementati i metodi iterativi descritti in 1.2.

L'algoritmo di bisezione è:

```
1  % Algoritmo di Bisezione
2  % [x,k]=bisezione(fun,a,b,tau,Nmax)
3
4  function [x,k]=bisezione(fun,a,b,tau,Nmax)
5  fa=fun(a);
6  fb=fun(b);
7           if (fa*fb>=0)
8                   disp('La funzione non cambia segno in [a,b]')
9           else
10               k=0;
11                  c=(a+b)/2;
12                  fc=fun(c);
13                  while ((abs(b-a))>tau && (fc~=0) && (k<Nmax))
14                  k=k+1;
```

```
15                              if (fa*fc<0)
16                                      b=c;
17                                      fb=fc;
18                              else
19                                      a=c;
20                                      fa=fc;
21                              end
22                              c=(a+b)/2;
23                              fc=fun(c);
24                      end
25                      x=c;
26              end
```

L'algoritmo di Newton è:

```
1  % Algoritmo di Newton
2  % [x,k]=newton(fun,dfun,x0,tau,Nmax)
3
4  function [x,k]=newton(fun,dfun,x0,tau,Nmax)
5  temp=x0;
6  x=x0-(fun(x0))/(dfun(x0));
7  k=1;
8  while((k<Nmax)&&(abs(x-temp)>tau)&&(fun(x)~=0))
9       temp=x0;
10              x=x0-(fun(x0))/(dfun(x0));
11              x0=x;
12              k=k+1;
13 end
```

Infine l'algoritmo delle secanti è:

```
1  % Algoritmo Quasi-Newton: Secanti.
2  % [x,k]=secanti(fun,x0,x,tau,Nmax)
3
4  function [x,k]=secanti(fun,x0,x,tau,Nmax)
5
6  k=0;
7  while((k<Nmax)&&(abs(x-x0)>tau)&&(fun(x)~=0))
8               f=fun(x);
9               f0=fun(x0);
10              temp=x;
11              x=(x0*f-x*f0)/(f-f0);
12              x0=temp;
13              f0=f;
14              k=k+1;
15 end
```

2.2.2 Algoritmo di Newton multidimensionale

Si è implementato inoltre il metodo di Newton multidimensionale, riportato di seguito.

```
1  % Algoritmo di Newton per Sistema Non Lineare
2  % [x,k]=multi_newton(effe,jacob,x0,tau,Nmax)
3
4  function [x,k]=multi_newton(effe,jacob,x0,tau,Nmax)
5
6  k=0;
7  x=x0-inv(jacob(x0))*effe(x0);
8  temp=x0;
9  while((k<Nmax)&&(norm(x-temp)>tau))
10             temp=x;
11             x=x-inv(jacob(x))*effe(x);
12             k=k+1;
13 end
```

Capitolo 3

Test degli algoritmi

3.1 Test per i sistemi lineari

3.1.1 Test degli algoritmi di Jacobi e Gauss-Seidel

I quattro programmi riportati in 2.1.1 sono stati messi a confronto fra di loro, e anche con la funzione gmres di MATLAB. Un approfondimento sulla teoria del metodo GMRES è presente in [1]. Si è scelto come tolleranza τ il valore 10^{-12}. Il numero massimo di iterazioni è stato inizialmente fissato a 50. Di seguito sono quindi riportati i codici utilizzati per i test.

```
1  % Test degli algoritmi di Jacobi
2
3  n=100;
4  A=rand(n);
5  A=A-diag(diag(A));
6  A=A'*A;
7  s=abs(A)*ones(n,1);
8  k_dom=2;  % Fattore di predominanza
9  A=A+k_dom*diag(s);
10 e=ones(n,1);
11 b=A*e;
12 x0=zeros(n,1);
13 tau=1e-12;
14 Nmax=50;
15 tic
16 [x_vettoriale,k_vettoriale]=jacobi(A,b,x0,tau,Nmax);
17 k_vettoriale
18 errore_vettoriale=norm(x_vettoriale-e)
19 tempo_vettoriale=toc
20 tic
21 [x_matriciale,k_matriciale]=jacobi_m(A,b,x0,tau,Nmax);
22 k_matriciale
23 errore_matriciale=norm(x_matriciale-e)
24 tempo_matriciale=toc
```

```
% Test degli algoritmi di Gauss Seidel

n=100;
A=rand(n);
A=A-diag(diag(A));
A=A'*A;
s=abs(A)*ones(n,1);
k_dom=2;  % Fattore di predominanza
A=A+k_dom*diag(s);
e=ones(n,1);
b=A*e;
x0=zeros(n,1);
tau=1e-12;
Nmax=50;
tic
[x_vettoriale,k_vettoriale]=GaussSeidel(A,b,x0,tau,Nmax);
k_vettoriale
errore_vettoriale=norm(x_vettoriale-e)
tempo_vettoriale=toc
tic
[x_matriciale,k_matriciale]=GaussSeidel_m(A,b,x0,tau,Nmax);
k_matriciale
errore_matriciale=norm(x_matriciale-e)
tempo_matriciale=toc
```

```
% Test GMRES

n=100;
A=rand(n);
A=A-diag(diag(A));
A=A'*A;
s=abs(A)*ones(n,1);
k_dom=2;  % Fattore di predominanza
A=A+k_dom*diag(s);
e=ones(n,1);
b=A*e;
x0=zeros(n,1);
tau=1e-12;
Nmax=50;
tic
[x,flag,relres,iter]=gmres(A,b,[],tau,Nmax);
errore=norm(x-e)
iter
tempo=toc
```

La matrice A è stata creata in modo tale da essere simmetrica definita positiva e diagonalmente dominante al variare di un fattore di predominanza k_{dom}. Si sono effettuati test a dimensione n crescente e per k_{dom} decrescente.

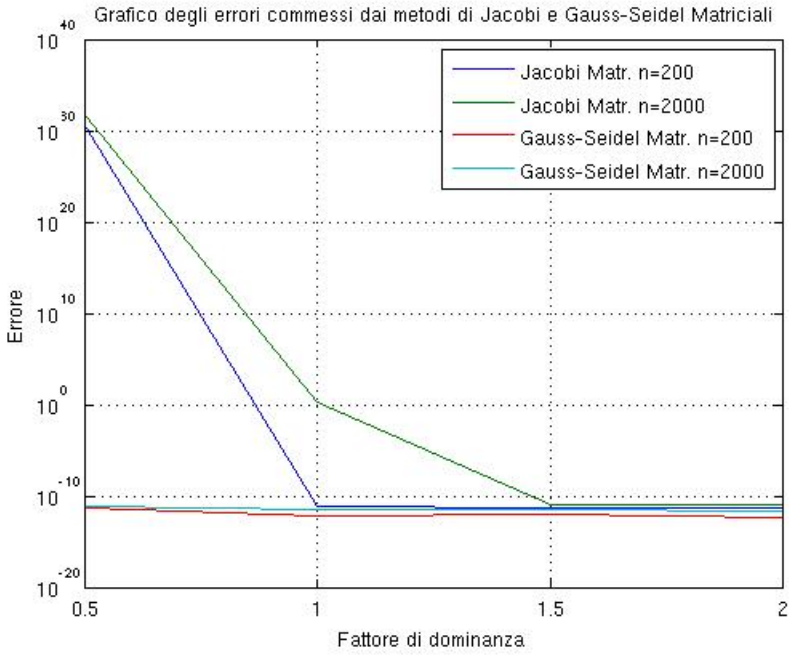

Figura 3.1: Andamento dell'errore in funzione di k_{dom} per i metodi di Jacobi e Gauss-Seidel matriciali

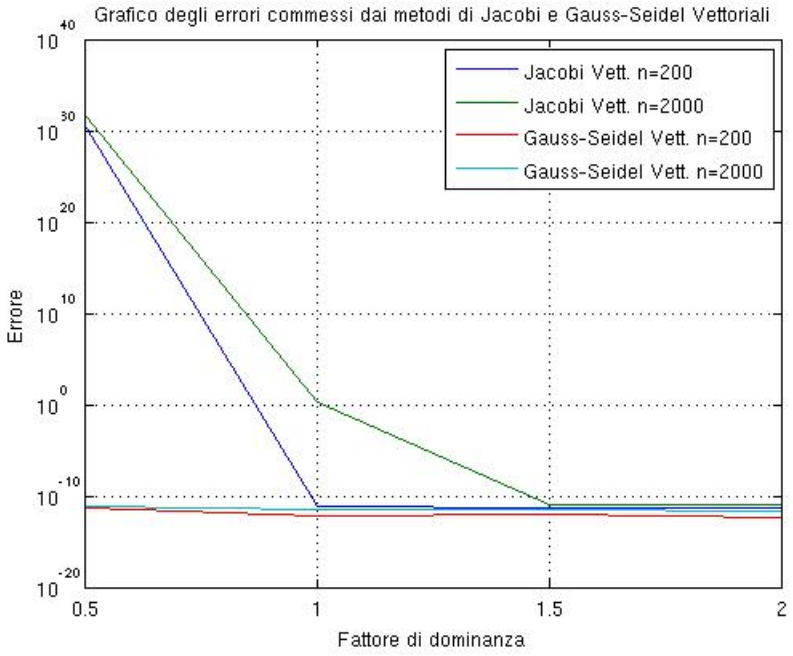

Figura 3.2: Andamento dell'errore in funzione di k_{dom} per i metodi di Jacobi e Gauss-Seidel vettoriali

I grafici 3.1 e 3.2 riportano l'errore in funzione del fattore di predominanza per i metodi di Jacobi e Gauss-Seidel in forma matriciale e vettoriale, rispettivamente. Si sono riportate le curve per determinati valori della dimensione n, in particolare 200 e 2000. Come si può vedere il metodo di Jacobi, sia matriciale che vettoriale, diverge al diminuire del fattore di predominanza, mentre il metodo di Gauss-Seidel presenta un errore piccolo indifferentemente dal valore di k_{dom}. Tutti i valori ricavati dai test sono riportati nelle tabelle della sezione 3.3: si è valutato il numero delle iterazioni k, l'errore e il tempo di calcolo. Nella tabella 3.4 vi sono i risultati ottenuti con un valore di k_{dom} pari a 2. Si è notato che, come ci si aspettava, tutti i metodi considerati convergono. Il metodo di Jacobi si mostra più lento di quello di Gauss-Seidel e, chiaramente, per entrambi le versioni matriciali sono più lente di quelle vettoriali. Il metodo GMRES, essendo ottimizzato, si rivela essere migliore degli altri. Tutte queste considerazioni sono applicabili anche ai test successivi.

La tabella 3.5 contiene i risultati per k_{dom} pari a 1.5. Abbiamo mantenuto inizialmente $N_{max} = 50$, come nel caso precedente, ma in questo modo non si è riusciti a osservare una convergenza per il metodo di Jacobi. Poiché il valore di k_{dom} è tale che la matrice sia ancora diagonalmente dominante, questo comportamento sarà da attribuire ad un N_{max} troppo piccolo. Infatti aumentando N_{max} a 100 siamo riusciti a ottenere convergenza anche per il metodo di Jacobi.

Nella tabella 3.6 sono riportati i risultati per k_{dom} pari a 1. Per lo stesso motivo riscontrato nel test precedente, si è provato ad aumentare N_{max} ai valori 200, 500, 1000, senza che il metodo di Jacobi convergesse. Abbiamo proceduto per tentativi adattando N_{max} in base alla dimensione n. Ci siamo comunque prefissati come limite massimo 2500: per questo valore abbiamo ottenuto convergenza sino a $n = 200$, mentre per valori superiori il metodo di Jacobi non convergeva più, comportamento osservabile chiaramente anche dai grafici. Il metodo di Gauss-Seidel, invece, si è rivelato insensibile alla diminuzione di k_{dom}, come detto in precedenza.

Infine, nella tabella 3.7 abbiamo inserito i valori ottenuti per $k_{dom} = 0.5$. In questo caso la matrice non è più diagonalmente dominante, quindi abbiamo riportato N_{max} a 100 sulla base del comportamento del metodo di Jacobi già osservato nel test precedente. E infatti, come già detto, non è convergente, come si può notare dai valori dell'errore. Viceversa, il metodo di Gauss-Seidel è ancora convergente in quanto la matrice è ancora simmetrica e definita positiva.

3.1.2 Test dell'algoritmo del gradiente coniugato precondizionato

L'algoritmo da noi implementato è stato confrontato con la funzione pcg presente in MATLAB. Il codice del programma di test è riportato di seguito.

```
% Test degli algoritmi del Gradiente Coniugato Precondizionato

n=100;
dens=0.05;  %Densità elementi non nulli matrice sparsa
rcond=0.01; %Reciproco condizionamento
A=sprandsym(n,dens,rcond,1);
e=ones(n,1);
b=A*e;
x0=zeros(n,1);
tau=1e-12;
Nmax=50;
tic
[x_nostro,k_nostro]=gcp(A,b,x0,tau,Nmax);
errore_nostro=norm(x_nostro-e)
k_nostro
```

n	Metodo	Errore	k	Tempo
100	*gcp*	$1.9849 * 10^{-13}$	5	0.0093
	pcg	$2.0139 * 10^{-13}$	5	0.0096
200	*gcp*	$1.5324 * 10^{-11}$	6	0.0054
	pcg	$1.5326 * 10^{-11}$	6	0.0023
500	*gcp*	$4.1615 * 10^{-12}$	8	0.1107
	pcg	$4.1652 * 10^{-12}$	8	0.0505
1000	*gcp*	$1.1648 * 10^{-10}$	8	0.7348
	pcg	$1.1648 * 10^{-10}$	8	0.0561
2000	*gcp*	$6.0961 * 10^{-11}$	9	5.2901
	pcg	$6.0966 * 10^{-11}$	9	0.1062

Tabella 3.1: Risultati test gradiente coniugato precondizionato

```
16 tempo_nostro=toc
17 sigma=1e-3;
18 R=cholinc(A,sigma);
19 tic
20 [x_matlab,flag,relres,k_matlab,resvec]=pcg(A,b,tau,Nmax,R',R,x0);
21 errore_matlab=norm(x_matlab-e)
22 k_matlab
23 tempo_matlab=toc
```

Anche in questo caso si sono svolti diversi test per valori della dimensione n crescenti, riportati nella tabella 3.1. Come atteso, il metodo da noi implementato, non essendo ottimizzato, risulta più lento di quello implementato in MATLAB. Gli errori e le iterazioni ottenute sono comunque confrontabili.

Nelle figure 3.3 e 3.4 sono riportati gli andamenti della norma del residuo in funzione delle iterazioni, per dimensioni della matrice pari a 200 e 2000, rispettivamente. Come si può notare il residuo decresce all'aumentare di k, comportamento previsto dal momento che si tratta di un metodo convergente.

3.2 Test per le equazioni non lineari

Abbiamo scritto un file di test per valutare le prestazioni dei tre metodi implementati in 2.2.1. Il codice è riportato di seguito.

```
1  % Test dei metodi per la risoluzione di Equazioni Non Lineari
2
3  % Funzione e soluzione
4  fun=@(x) x-sin(x).^2-cos(x).^2;
5  dfun=@(x) 1;
6  sol=1;
7
8  tau=1e-8;
9  Nmax=100;
10
11 % Parametri
12 a=0;
13 b=50;
```

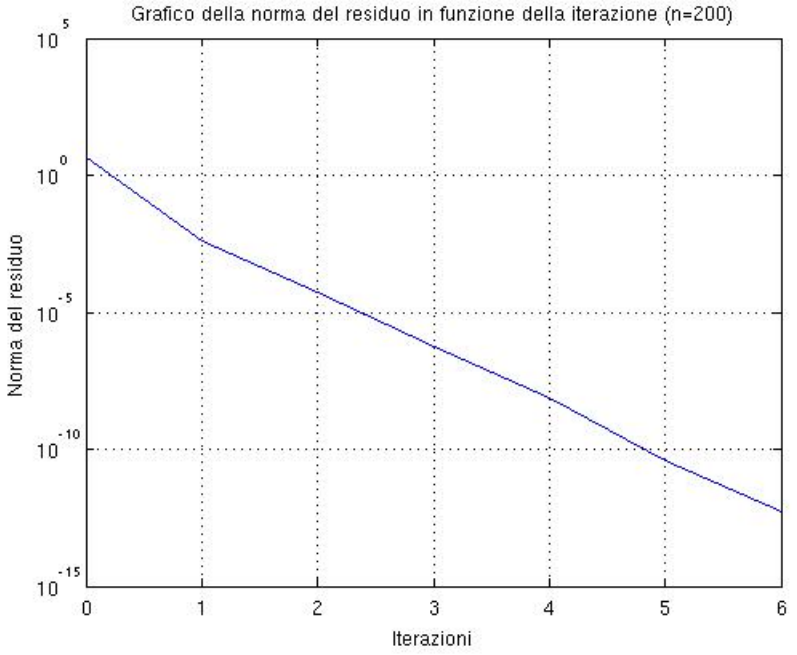

Figura 3.3: Andamento del residuo normalizzato in funzione di k ($n = 200$)

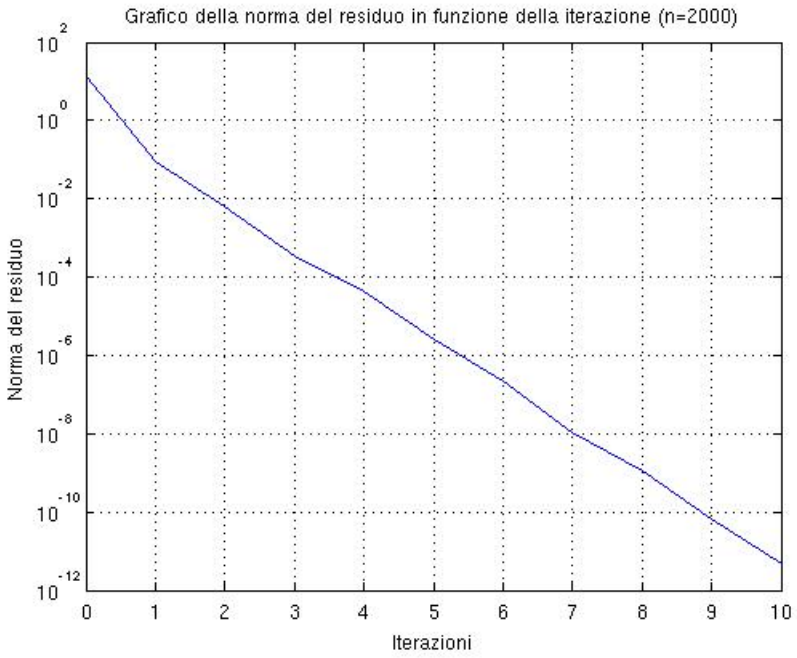

Figura 3.4: Andamento del residuo normalizzato in funzione di k ($n = 2000$)

	Bisezione	Newton	Secanti
Funzioni	Iterazioni / Errore	Iterazioni / Errore	Iterazioni / Errore
$e^x - 7$	29 — $2.9926*10^{-9}$	7 — 0	5 — $2.6201*10^{-14}$
$x - \sin^2 x - \cos^2 x$	29 — $3.7253*10^{-9}$	1 — 0	2 — 0
$(x-2)^5$	0 — 0	78 — $3.4509*10^{-8}$	0 — 0
$\tanh(x - 1.5)$	2 — 0	4 — 0	1 — 0

Tabella 3.2: Risultati dei test con parametri "vicini" alla soluzione

	Bisezione	Newton	Secanti
Funzioni	Iterazioni / Errore	Iterazioni / Errore	Iterazioni / Errore
$e^x - 7$	33 — $2.6433*10^{-9}$	54 — $2.2204*10^{-16}$	77 — 0
$x - \sin^2 x - \cos^2 x$	33 — $1.9791*10^{-9}$	3 — 0	2 — 0
$(x-2)^5$	33 — $1.0477*10^{-9}$	95 — $3.7299*10^{-8}$	100 — $8.8365*10^{-6}$
$\tanh(x - 1.5)$	33 — $1.5134*10^{-9}$	3 — NaN	2 — ∞

Tabella 3.3: Risultati dei test con parametri "lontani" dalla soluzione

```
14 x0=50;
15 x1=51;
16
17 [x_bisezione,k_bisezione]=bisezione(fun,a,b,tau,Nmax);
18 [x_newton,k_newton]=newton(fun,dfun,x0,tau,Nmax);
19 [x_secanti,k_secanti]=secanti(fun,x0,x1,tau,Nmax);
20 k_bisezione
21 k_newton
22 k_secanti
23 errore_bisezione=abs(sol-x_bisezione)
24 errore_newton=abs(sol-x_newton)
25 errore_secanti=abs(sol-x_secanti)
```

Per praticità si sono effettuati test per tre diverse funzioni con soluzioni note, variando semplicemente fun, dfun e sol, e ripetendo le prove sia con parametri "vicini" alla soluzione, sia con parametri "lontani". Come parametri "vicini" si sono considerati: $[a, b] = [0, 4]$, $x_0 = 1$ e $x_1 = 2$. Come parametri "lontani" si sono presi invece: $[a, b] = [0, 50]$, $x_0 = 50$ e $x_1 = 51$.

Nella tabella 3.2 possiamo osservare come nel metodo di bisezione il numero di iterazioni sia sempre lo stesso a prescindere dalla funzione considerata: questo sottolinea che l'algoritmo lavora principalmente sull'intervallo. Da notare inoltre l'entità dell'errore per il metodo di Newton e quello delle secanti nel caso della seconda equazione: avendo usato come punto iniziale $x_0 = 1$, corrispondente alla soluzione stessa, i metodi hanno trovato subito il valore esatto. Questo si può notare anche per la terza equazione, in cui il metodo di bisezione trova subito la soluzione esatta perché il punto medio dell'intervallo considerato è proprio la soluzione, e il metodo delle secanti la trova perché essa coincide con il secondo punto iniziale x_1. Invece il metodo di Newton presenta un numero elevato di iterazioni, a causa della molteplicità non unitaria della soluzione.

Invece nella tabella 3.3 vi sono i risultati per parametri "lontani". Nella prima funzione si può notare come il metodo di Newton e quello delle secanti siano particolarmente sensibili alla vicinanza dei punti iniziali dalla soluzione, mentre il metodo di bisezione risulta meno influenzabile. Nella terza equazione si può notare che sia il metodo di Newton sia quello delle secanti hanno un alto numero di

iterazioni, e addirittura il metodo delle secanti termina prima di arrivare alla precisione richiesta con un numero di iterazioni massimo pari a 100: provando ulteriormente si è notato che la convergenza avviene alla 133° iterazione. Questi comportamenti sono legati sia alla lontananza che alla molteplicità della soluzione. Infine nell'ultima equazione si osserva che il metodo di bisezione converge, mentre il metodo di Newton e delle secanti presentano dei problemi attribuibili alla lontananza e alla pendenza nulla della funzione in quei punti iniziali. Entrambi escono dal ciclo quando la seconda condizione d'arresto presenta il valore assoluto di *Not-a-Number*: a quel punto in output si trovano i valori della soluzione dell'iterazione precedente, Not-a-Number per il metodo di Newton e ∞ per quello delle secanti. Questi comportamenti sono facilmente osservabili se vengono fatti eseguire i rispettivi algoritmi utilizzando il comando "pause" di MATLAB.

3.2.1 Test di un sistema non lineare 2x2

Abbiamo provato a risolvere un semplice sistema di equazioni non lineari 2x2, per testare l'efficacia del metodo di Newton multidimensionale da noi implementato.

Il sistema da risolvere era il seguente:

$$\begin{cases} x^2 + y^2 - 1 = 0 \\ x - y = 0 \end{cases} \tag{3.1}$$

La cui soluzione è:

$$\begin{cases} x = \frac{\sqrt{2}}{2} \simeq 0.7071 \\ y = \frac{\sqrt{2}}{2} \simeq 0.7071 \end{cases}$$

Per risolvere il sistema si è usato il seguente codice di test:

```
1  %Test sistema non lineare 2x2
2
3  f=@(x)[x(1)^2+x(2)^2-1;x(1)-x(2)];
4  jacob=@(x)[2*x(1)  2*x(2);1  -1];
5  x0=[0;4];
6  tau=1e-8;
7  Nmax=100;
8
9  [x,k]=multi_newton(f,jacob,x0,tau,Nmax)
```

Il test è arrivato alla soluzione corretta in 6 iterazioni, confermando la validità dell'implementazione effettuata.

3.2.2 Applicazione a una giunzione p-n

Oltre ai test già effettuati, si è voluto provare ad applicare i metodi di bisezione, Newton e secanti all'equazione non lineare che descrive l'andamento della corrente in un diodo:

$$I_I = \frac{V_V}{R_S} - \frac{1}{R_S}\frac{kT}{q}\ln\left(\frac{I_I}{I_0}\right) \tag{3.2}$$

dove $V_V = 0.6\,\mathrm{V}$, $R_S = 29.13\,\Omega$, $I_0 = 4.5 * 10^{-12}\,\mathrm{A}$ e $\frac{kT}{q} = 0.0259\,\mathrm{eV}$. Tutti e tre gli algoritmi, bisezione, Newton e secanti, hanno raggiunto un'approssimazione accettabile della soluzione, pari a $I_I = 2.6\,\mathrm{mA}$. Il metodo di bisezione ha impiegato 29 iterazioni, mentre sia quello di Newton che quello delle secanti hanno impiegato 8 iterazioni.

3.3 Appendice: tabelle dei risultati dei test di Jacobi e Gauss-Seidel

Sono di seguito riportate le quattro tabelle contenenti i risultati ottenuti durante i test degli algoritmi di Jacobi e Gauss-Seidel, commentate in sezione 3.1.1.

n	Risultati	Jacobi Vett.	Jacobi Matr.	G-S Vett.	G-S Matr.	GMRES
100	*Errore*	$1.9888*10^{-12}$	$1.9886*10^{-12}$	$3.2210*10^{-13}$	$3.2183*10^{-13}$	$1.0150*10^{-12}$
	k	41	41	14	14	11
	Tempo	0.0366	0.0509	0.0208	0.0135	0.0028
200	*Errore*	$4.2565*10^{-12}$	$4.2568*10^{-12}$	$5.1477*10^{-13}$	$5.1472*10^{-13}$	$1.8994*10^{-12}$
	k	41	41	14	14	10
	Tempo	0.0731	0.0548	0.0255	0.0288	0.0134
500	*Errore*	$4.2972*10^{-12}$	$4.2982*10^{-12}$	$8.7946*10^{-13}$	$8.7984*10^{-13}$	$4.2622*10^{-12}$
	k	42	42	14	14	9
	Tempo	0.4430	0.7231	0.0863	0.2537	0.0822
1000	*Errore*	$6.6113*10^{-12}$	$6.6115*10^{-12}$	$1.2736*10^{-12}$	$1.2758*10^{-12}$	$5.0427*10^{-12}$
	k	42	42	14	14	8
	Tempo	1.9533	5.0546	0.3711	1.7829	0.0325
2000	*Errore*	$9.7485*10^{-12}$	$9.7483*10^{-12}$	$1.8313*10^{-12}$	$1.8340*10^{-12}$	$3.7701*10^{-11}$
	k	42	42	14	14	7
	Tempo	7.7486	34.3803	1.2924	11.8578	0.1331

Tabella 3.4: Risultati test per $k_{dom} = 2$ e $N_{max} = 50$

n	Risultati	Jacobi Vett.	Jacobi Matr.	G-S Vett.	G-S Matr.	GMRES
100	*Errore*	$3.5579*10^{-12}$	$3.5575*10^{-12}$	$5.3719*10^{-13}$	$5.3707*10^{-13}$	$1.1471*10^{-12}$
	k	67	67	16	16	11
	Tempo	0.0470	0.0298	0.0289	0.0189	0.0046
200	*Errore*	$4.6319*10^{-12}$	$4.6320*10^{-12}$	$8.9559*10^{-13}$	$8.9631*10^{-13}$	$1.7743*10^{-12}$
	k	69	69	16	16	10
	Tempo	0.1190	0.0842	0.0236	0.0303	0.0054
500	*Errore*	$7.7228*10^{-12}$	$7.7228*10^{-12}$	$1.5595*10^{-12}$	$1.5589*10^{-12}$	$4.3315*10^{-12}$
	k	70	70	16	16	9
	Tempo	0.7416	1.1777	0.1223	0.3228	0.0750
1000	*Errore*	$8.4906*10^{-12}$	$8.4914*10^{-12}$	$2.2734*10^{-12}$	$2.2740*10^{-12}$	$1.8549*10^{-11}$
	k	71	71	16	16	8
	Tempo	3.2960	8.4754	0.4166	1.9863	0.0971
2000	*Errore*	$1.2992*10^{-11}$	$1.2990*10^{-11}$	$3.2537*10^{-12}$	$3.2542*10^{-12}$	$5.3113*10^{-11}$
	k	71	71	16	16	7
	Tempo	13.0869	58.9175	1.4520	13.5498	0.1291

Tabella 3.5: Risultati test per $k_{dom} = 1.5$ e $N_{max} = 100$

n	Risultati	Jacobi Vett.	Jacobi Matr.	G-S Vett.	G-S Matr.	GMRES
100	*Errore*	$4.8505*10^{-12}$	$4.8515*10^{-12}$	$4.2751*10^{-13}$	$4.2743*10^{-13}$	$1.6849*10^{-11}$
	k	1061	1061	20	20	10
	Tempo	0.4713	0.2172	0.0204	0.0174	0.1379
200	*Errore*	$6.9910*10^{-12}$	$6.9911*10^{-12}$	$7.2621*10^{-13}$	$7.2633*10^{-13}$	$2.5392*10^{-12}$
	k	2112	2112	20	20	10
	Tempo	3.2495	2.3382	0.0262	0.0405	0.0185
500	*Errore*	$3.5676*10^{-5}$	$3.5676*10^{-5}$	$1.3004*10^{-12}$	$1.3013*10^{-12}$	$5.8352*10^{-12}$
	k	2500	2500	20	20	9
	Tempo	26.4958	44.0315	0.1463	0.3571	0.0856
1000	*Errore*	0.0400	0.0400	$1.9159*10^{-12}$	$1.9162*10^{-12}$	$8.3958*10^{-12}$
	k	2500	2500	20	20	8
	Tempo	113.6179	298.6591	0.5204	2.4912	0.1225
2000	*Errore*	1.5938	1.5938	$2.7747*10^{-12}$	$2.7733*10^{-12}$	$7.3466*10^{-11}$
	k	2500	2500	20	20	7
	Tempo	453.9344	$2.0485*10^{3}$	1.8429	16.9417	0.1901

Tabella 3.6: Risultati test per $k_{dom} = 1$ e per $N_{max} = 2500$

n	Risultati	Jacobi Vett.	Jacobi Matr.	G-S Vett.	G-S Matr.	GMRES
100	*Errore*	$2.3175*10^{29}$	$2.3175*10^{29}$	$2.5268*10^{-12}$	$2.5265*10^{-12}$	$4.4460*10^{-12}$
	k	100	100	31	31	11
	Tempo	0.0435	0.0258	0.0220	0.0134	0.0796
200	*Errore*	$2.4297*10^{30}$	$2.4297*10^{30}$	$5.1937*10^{-12}$	$5.1946*10^{-12}$	$1.9919*10^{-12}$
	k	100	100	31	31	10
	Tempo	0.1701	0.1173	0.0430	0.0503	0.1359
500	*Errore*	$1.2732*10^{31}$	$1.2732*10^{31}$	$1.0994*10^{-11}$	$1.0196*10^{-11}$	$5.1249*10^{-12}$
	k	100	100	31	31	9
	Tempo	1.0595	1.7337	0.2132	0.5635	0.0143
1000	*Errore*	$2.6871*10^{31}$	$2.6871*10^{31}$	$1.5447*10^{-11}$	$1.5446*10^{-11}$	$6.4360*10^{-12}$
	k	100	100	31	31	8
	Tempo	4.5386	12.1654	0.8095	3.8645	0.1040
2000	*Errore*	$4.6413*10^{31}$	$4.6413*10^{31}$	$7.0654*10^{-12}$	$7.0668*10^{-12}$	$5.2139*10^{-11}$
	k	100	100	32	32	7
	Tempo	18.5202	82.4771	2.9228	26.6819	0.1334

Tabella 3.7: Risultati test per $k_{dom} = 0.5$ e per $N_{max} = 100$

Bibliografia

[1] Giuseppe Rodriguez, "*Algoritmi numerici*", Pitagora Editrice Bologna, 2008