UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato di Calcolo Numerico

Metodo del gradiente coniugato

Anno Accademico 2021/2022

Candidati
Christian Marescalco
matr. M63001367
Davide Marcello
matr. M63001375

Contents

Chapter 1

Metodi per la risoluzione di sistemi lineari

1.1 Metodi diretti ed iterativi

Un sistema di n equazioni lineari in n incognite può essere rappresentato nella forma matriciale:

$$Ax = b$$

dove A è la matrice dei coefficienti, b il vettore dei termini noti.
Per la risoluzione di sistemi lineari è possibile utilizzare sia metodi **diretti** che **iterativi**. I metodi diretti calcolano la soluzione del sistema tramite una fattorizzazione della matrice A, fornendo una soluzione esatta (a meno di un errore di roundoff sul risultato) in un numero

finito di passi. La complessità resta in generale pari a $O(n^3)$ e non si riduce nel caso di matrici sparse (nel caso di sparsità non strutturata) a causa del *fill-in*, ossia il fenomeno in cui i valori nulli diventano diversi da zero durante l'esecuzione di un algoritmo. Per questo motivo risultano inadatti per problemi di grandi dimensione e sparsi.
Per sistemi con matrici sparse e di ordine elevato, un'alternativa migliore è quella dei metodi iterativi, in cui a partire da un vettore iniziale $x^{0)}$ viene generata una successione di vettori $x^{(k)}$ con k intero, che al limite deve fornire un'approssimazione adeguata della soluzione x. Un metodo iterativo è globalmente convergente se $\forall x^{(0)} \, \epsilon \, R^n$:

$$\lim_{k \to \infty} \|x^{(k)} - x\|$$

dove x è la soluzione esatta del sistema, trattandosi di vettori si parla di convergenza in norma.

I metodi iterativi stazionari e lineari sono rappresentabili come:

$$x^{(k+1)} = Bx^{(k)} + c$$

dove B è detta *matrice di iterazione*, il metodo si dice stazionario perchè la matrice di iterazione e il vettore c non dipendono dall'indice di iterazione k.
Per la costruzione di un metodo iterativo è necessario:

- Costruire la matrice B;
- Stabilire la convergenza della successione $x^{(k)}$;
- Scegliere la condizione iniziale $x^{(0)}$;
- Stabilire criterio di arresto del metodo;

Dopo aver costruito un metodo iterativo è opportuno domandarsi se la scelta di B è stata opportuna, cioè se dopo infinite iterazioni la soluzione ottenuta è realmente quella del sistema, ossia la convergenza ad x. Una condizione necessaria e sufficiente affinché il metodo converga alla soluzione del sistema per ogni scelta di $x^{(0)}$ è che il raggio spettrale (cioè il più grande autovalore in modulo) di B sia strettamente inferiore a 1:

$$\rho(B) < 1$$

Effettuando una comparazione tra i metodi diretti e quelli iterativi, si ottiene che i primi sono più robusti in quanto riescono ad ottenere una soluzione esatta, ma meno veloci in quanto i secondi richiedono un numero di iterazioni in generale dell'ordine $O(\#iter * n^2)$.
In presenza di matrici sparse, i metodi iterativi riescono a sfruttare la sparsità di A, lasciandola inalterata. Dunque un metodo iterativo è da preferire se:

- La matrice è sparsa, non strutturata e di grandi dimensioni;
- l'algoritmo fornisce una soluzione accurata in $\#iter << n$;

- non si vogliono alterare gli elementi della matrice;

Il problema dei metodi iterativi è dovuto dalla presenza dell'*errore di roundoff* e da problemi *malcondizionati*, che potrebbero rallentare di molto la convergenza dell'approssimazione della soluzione. Un possibile metodo per stabilire la velocità di convergenza del metodo è valutare $\rho(B)$, se questo è vicino a 0 allora il metodo è più veloce, invece è tanto più lenta quanto più è vicino ad 1.

1.2 Metodo del gradiente

E' un esempio di metodo iterativo per la minimizzazione di una funzione quadratica convessa. L'idea di base è quella di ricercare ad ogni passo la direzione di *massima discesa del gradiente* per raggiungere il minimo locale della funzione.

Si consideri A matrice simmetrica e definita positiva, a partire dal problema $Ax = b$, l'idea è calcolare la soluzione approssimativa dalla funzione:

$$E(x) = \|x - \tilde{x}\|_A^2 = \frac{1}{2}(x - \tilde{x})^T A(x - \tilde{x}) =$$

$$\frac{1}{2}\left(\tilde{x}^T - x^T\right)(Ax - A\tilde{x}) = \frac{1}{2}\left(\tilde{x}^T Ax - x^T Ax + x^T A\tilde{x} - \tilde{x}^T A\tilde{x}\right) =$$

$$\frac{1}{2}[\tilde{x}^T (Ax - b) - x^T (Ax - b)]$$

Poichè $\tilde{x}$ non è noto, posso effettuare una traslazione di E(x) di una quantità pari a $\frac{1}{2}x^T(Ax-b)$, ottenendo a meno del segno:

$$E(x) = \frac{1}{2}x^T Ax - x^T b$$

Essendo la matrice A simmetrica e definita positiva, la funzione E è un paraboloide ellittico convesso, dunque ammette un unico punto di minimo. Tale punto è quello in cui si annulla il gradiente:

$$\nabla E(x) = \frac{1}{2}(A + A^T)x - b = Ax - b = 0$$

Si può dunque scrivere la successione delle approssimazioni come metodo iterativo non stazionario:

$$x_{k+1} = x_k + \alpha_k d_k$$

Ad ogni passo k, si sceglie come direzione di ricerca $d_k = -\nabla E(x_k)$. Dunque la direzione di ricerca d_k è proprio il residuo del sistema lineare:

$$r_k = b - Ax_k$$

Per calcolare la lunghezza del passo α_k impongo che al passo k+1, per qualunque direzione d_k, la derivata di $E(x_{k+1})$ rispetto ad α si annulli, ossia impongo la condizione di stazionarietà rispetto ad α nel punto k+1 esimo:

$E(x_{k+1}) = \frac{1}{2}(x_k + \alpha \cdot d_k)^T A(x_k + \alpha \cdot d_k) - (x_k + \alpha \cdot d_k)^T b = E(x_k) + \frac{1}{2}(d_k)^T \cdot A \cdot (d_k) \cdot \alpha^2 - (d_k)^T \cdot r_k \cdot \alpha$

Imponendo la condizione di stazionarietà:

$$\frac{\partial E(x_{k+1})}{\partial \alpha} = (d_k)^T A(d_k) \cdot \alpha - (d_k)^T \cdot r_k$$

da cui segue:

$$\alpha_k = \frac{d_k^T r_k}{d_k^T A d_k}$$

Il metodo del gradiente, dunque, tenta di minimizzare il residuo r_k ad ogni iterazione, fino a trovare la soluzione ottima.

Posso descrivere l'algoritmo con il seguente pseudocodice:

Algorithm 1

$k \leftarrow 0$
$r0 = Ax_0 - b$
while $\|r_k\| > tol$ **do**
  $d_k = r_k = b - Ax_k$
  $a_k = \frac{d_k^T r_k}{d_k^T A d_k}$
  $x_{k+1} = x_k + \alpha_k d_k$
  $k = k + 1$

Dove si utilizza come criterio di arresto la norma del residuo ed una tolleranza impostata dall'utente.

Il problema di questo algoritmo sta nella lenta convergenza dovuta alla ridondanza delle direzioni scelte ad ogni iterazione, poichè si segue la direzione del gradiente. Infatti la convergenza risulta essere lineare:

$$\frac{\|x_{k+1} - \tilde{x}\|_A}{\|x_k - \tilde{x}\|_A} \leq \left(\frac{\mu_2(A) - 1}{\mu_2(A) + 1}\right)^2$$

dove $\mu_2(A) = \frac{\rho}{\lambda_{min}}$ *è l'indice di condizionamento spettrale.*

E' possibile fornire un'interpretazione geometrica di $\mu_2(A)$: ad una matrice A malcondizionata corrisponde un paraboloide ellittico molto allungato, mentre ad un $\mu_2(A)$ piccolo corrisponde un paraboloide più arrotondato.

1.3 Metodo delle direzioni coniugate

Rispetto al metodo del gradiente è possibile effettuare una scelta migliore delle direzioni di discesa. Ad ogni iterazione si minimizza la funzione E, agendo su sottospazi delle direzioni (scelte in precedenza) di dimensione via via crescente. Scrivendo la soluzione del sistema $\tilde{x}$, a partire da un punto x_0 arbitrario come:

$$\tilde{x} = x_0 + \alpha_0 v_0 + \alpha_1 v_1 + ... + \alpha_{n-1} v_{n-1}$$

dove i vettori v_i sono linearmente indipendenti e rappresentano le direzioni di ricerca.

Il problema può essere riformulato in questo modo all'iterazione k+1:

$$\min E(x_0 + \alpha_0 v_0 + \alpha_1 v_1 + ... + \alpha_k v_k) = minE(x_0 + V\alpha)$$

con $\alpha \, \epsilon \, R^{k+1}, V = [v_0, ..., v_k] \, \epsilon \, R^{n \cdot k+1}$. Calcolato $\tilde{\alpha} = argminE(x_o + V\alpha)$ ciò equivale a:

$$V^T \nabla E(x_o + V\tilde{\alpha}) = V^T r_k = 0$$

per ogni direzione v_i, con $i < k$, mantengo l'ottimalità del risultato parziale ottenuto x_k, imponendo che sia ortogonale a tutte le direzioni di V, che equivale alla seguente condizione di ortogonalità:

$$\nabla E(x_o + V\tilde{\alpha}) \, \epsilon \, Span\{v_0, ..., v_k\}^{\perp}$$

cioè ad ogni iterazione scelgo α in modo che sia ottimo rispetto a tutte le direzioni fino all'iterazione corrente.
Per la generica iterazione k, ciò accade se e solo se le direzioni v_j sono A-coniugate:

$$v_i^T A v_j = 0 : i \neq j, i, j < k + 1$$

Le lunghezze α_k sono scrivibili come:

$$\alpha_k = \frac{v_k^T \cdot r_k}{v_k^T A v_k}$$

Il problema di quest'algoritmo sta nella determinazione delle direzioni coniugate v_k, ma ha il vantaggio di poter terminare dopo **un numero finito di passi (al max n)**. A differenza del metodo del gradiente, il vettore x_{k+1} è reso ottimale non solo rispetto alla direzione di discesa r_k dell'iterazione corrente, ma anche rispetto a tutte

le precedenti direzioni, in questo modo risulta essere ottimale rispetto al sottospazio V.

Descrizione dell'algoritmo con il seguente pseudocodice:

Algorithm 2

$v_0, ..., v_{n-1} A - coniugati\, non\, nulli$
$k \leftarrow 0$
$r0 = Ax_0 - b$
while $\|r_k\| > tol\, and\, k < n$ **do**
  $r_k = b - Ax_k$
  $a_k = \frac{v_k^T r_k}{v_k^T A v_k}$
  $x_{k+1} = x_k + \alpha_k v_k$
  $k = k + 1$

1.4 Metodo del gradiente coniugato

Il metodo del gradiente coniugato utilizza l'approccio del metodo delle direzioni coniugate, andando a calcolare ad ogni iterazione k anche la direzione successiva v_{k+1}. La direzione di ricerca all'iterazione k è costruita come combinazione lineare dell'antigradiente in tale iterazione e della direzione di ricerca all'iterazione precedente:

$$v_{k+1} = r_{k+1} - \beta_k v_k$$

Bisogna stabilire il valore di β_k in modo che la direzione v_{k+1} sia A-coniugata rispetto alle altre. Essendo il passo ottimale ad ogni iterazione vale : $r_{k+1} \perp v_j \implies r_{k+1} \perp r_j \;\; j = 1, .., k-1$.

Dunque la direzione v_{k+1} è coniugata con tutte le direzioni fino alla k-1 esima, per poter ottenere che sia coniugata alla v_k calcolo β_k come:

$$\beta_k = \frac{v_k^T r_{k+1}}{v_k^T A v_k}$$

Il metodo, inoltre, è tale che:

$$Span\{v_0, ..., v_k\} = Span\{r_0, ..., r_k\} = Span\{r_0, Ar_0..., A^k r_0\}$$

Inoltre vale il seguente teorema per la convergenza:
Se m è il numero di autovalori distinti di A, allora il metodo CG converge alla soluzione in un numero di passi $k \leq m$.
Dal teorema si ottiene che il metodo dei gradienti coniugati, se si opera in *aritmetica infinita*, è un algoritmo diretto, come il precedente converge in n passi, ma poiché esso viene utilizzato con l'aritmetica a precisione finita, è detto metodo iterativo.
In presenza di errore di round off, l'algoritmo ha tre comportamenti differenti:

- Semiconvergenza o divergenza, dopo n iterazioni gli errori $E(x_{n+k})$ vanno a peggiorare, ottenendo un'approssimazione sbagliata;
- Convergenza, dopo n iterazioni gli errori $E(x_{n+k})$ convergono a zero;
- Convergenza numerica, dopo n iterazioni gli errori $E(x_{n+k})$ si

stabilizzano all'epsilon macchina;

La divergenza del metodo è causata dalle matrici malcondizionate $\mu(A) >> 1$. Dunque il residuo r_k può aumentare in presenza di matrici malcondizionate, come dimostra il seguente teorema:
Alla k-esima iterazione del metodo CG risulta:

$$\frac{\|b - Ax_k\|_2}{\|b - Ax_0\|_2} \leq \sqrt{\mu_2(A)} \frac{\|x_k - \tilde{x}\|_A}{\|x_0 - \tilde{x}\|_A} \tag{1.1}$$

A partire dalla 1.1, è possibile ottenere un limite superiore per l'errore al passo k, che mi garantisce la convergenza in aritmetica a precisione infinita:

$$\frac{\|x_k - \tilde{x}\|_A}{\|x_0 - \tilde{x}\|_A} \leq 2\Big[\frac{\sqrt{\mu_2(A)} - 1}{\sqrt{\mu_2(A)} + 1}\Big]^k$$

Dalla quale si ottiene una stima della velocità di convergenza, fissata una $TOL = 10^{-r}$:

$$2M^k = 2\Big[\frac{\sqrt{\mu_2(A)} - 1}{\sqrt{\mu_2(A)} + 1}\Big]^k \leq TOL$$

$$k \geq \frac{\log 2TOL}{\log M}$$

In aritmetica a precisione finita, la rapidità di convergenza dipende da μ:

- convergenza in un numero di passi $k \leq n$, se $\mu > 1$;
- convergenza numerica molto lenta, se $\mu >> 1$;

- semiconvergenza, se $\mu = \infty$ (in caso di problemi singolari);

L'algoritmo in pseudocodice:

Algorithm 3

$r0 = Ax_0 - b$
$v_0 = r_0$
while $k < n + 1$ **do**
  $a_k = \frac{v_k^T r_k}{v_k^T A v_k}$
  $x_{k+1} = x_k + \alpha_k v_k$
  $r_{k+1} = b - Ax_{k+1}$
  $\beta_k = \frac{v_k^T A r_{k+1}}{v_k^T A v_k}$
  $v_{k+1} = r_{k+1} - \beta_k v_k$
  $k = k + 1$

La complessità dell'algoritmo è data da: $O(iter \cdot n^2)$, dove n^2 sta ad indicare il prodotto matrice per vettore.

Supponendo l'aritmetica floating point, si sostituisce la condizione del while con $\|r_k\| > tol \|b\|$. In presenza di errori di roundoff, il metodo CG risulta essere instabile anche a piccole perturbazioni, rendendo le direzioni effettive non esattamente coniugate fra loro, si cerca dunque di mantenere il numero di iterazioni basso per evitare la propagazione dell'errore di roundoff.

La rapidità di convergenza dipende dall'indice di condizionamento, è possibile effettuare un **precondizionamento** in modo tale da migliorare l'indice di condizionamento e aumentare la velocità dell'algoritmo. Il sistema $Ax = b$ viene trasformato in uno equivalente con una matrice di precondizionamento (a sinistra) M simmetrica e definita positiva : $M^{-1}(Ax - b) = 0$. Scelgo M in modo che clusterizzi gli autovalori di

A in pochi e piccoli sottointervalli in prossimità di 1, per far convergere rapidamente l'algoritmo di CG. La matrice ideale sarebbe A^{-1}, ma poichè non è nota e richiede una complessità troppo elevata, si sceglie una basata sulla disposizione degli autovalori di A, facendo in modo che la matrice complessiva resti simmetrica e definita positiva. L'algoritmo viene modificato in questo modo:

Algorithm 4

$r0 = Ax_0 - b$
$z_0 = M^{-1}r_0$
$v_0 = z_0$
while $k < n + 1$ **do**
  $a_k = \frac{v_k^T r_k}{v_k^T A v_k}$
  $x_{k+1} = x_k + \alpha_k v_k$
  $r_{k+1} = b - Ax_{k+1}$
  $z_{k+1} = M^{-1}r_{k+1}$
  $\beta_k = \frac{v_k^T A z_{k+1}}{v_k^T A v_k}$
  $v_{k+1} = z_{k+1} - \beta_k v_k$
  $k = k + 1$

dove M è un generico precondizionatore, che resta fisso ad ogni iterazione. Una possibile scelta di M è $M = RR^T$, con R calcolata tramite fattorizzazione di Cholesky incompleta. Anche in questo caso la complessità dell'algoritmo è data da: $O(iter \cdot n^2)$, dove n^2 sta ad indicare il prodotto matrice per vettore.

Chapter 2

Implementazione degli algoritmi

2.1 Implementazione del metodo di discesa del gradiente

Di seguito è riportata l'implementazione in Matlab del metodo di discesa del gradiente. Per poter salvare la lista dei punti x_k raggiunti ad ogni iterazione ed il numero di iterazioni, si usa *lista_punti* e *kterm*. Se vengono impiegate un numero di iterazioni superiore ad *nmax* l'algoritmo termina con un warning.

```
function [xk,lista_punti,kterm] = discesa(A, b, x0, nmax, toll,lista_punti)
    xk = x0;
    rk = b-A*xk;
    k = 0;
    while ((norm(rk) >= toll) && (k < nmax))
        lista_punti{k+1,1} = xk;
        dk = rk;
        alphak = dk' * rk / (dk' * A * dk);
        xk = xk + alphak * dk;
        rk = b - A * xk;
        k = k+1;
    end
    kterm = k;
    if k == nmax
        warning('Convergenza non raggiunta in nmax iterazioni');
    end
end
```

Figure 2.1: Algoritmo di discesa del gradiente

2.1.1 Test dell'algoritmo

Per effettuare il test dell'algoritmo si utilizza il seguente script 2.2, in cui è stata utilizzata la funzione sprandsym() per poter generare in modo casuale matrici simmetriche e definite positive, andando a definire anche l'indice di condizionamento associato e la percentuale di densità di elementi della matrice (che in questo caso viene fissata al 100%). La tolleranza è fissata a 10^{-8}, mentre il numero di iterazioni massime a 1000.

Per analizzare le prestazioni dell'algoritmo si valuta l'andamento del residuo relativo rispetto alle iterazioni, andando a rappresentare la curva in scala logaritmica rispetto alle ordinate. In figura 2.3 e 2.4, sono mostrati gli andamenti.Si noti come all'aumentare dell'indice di condizionamento, la convergenza è sempre più lenta e richiede più iter-

azioni, inoltre il residuo tende ad oscillare sempre di più all'aumentare dell'indice di condizionamento.

```
import discesa.*


N = 100;
mu = 100000; %indice di condizionamento
A = full(sprandsym(N, 1, 1/mu, 1)) * 100;  %costruisco la matrice simmetrica e definita positiva
                                        %(dim, densità, 1/indice_condizionamento, definita positiva = 1)

%parametri
b = rand(N,1) * 100;
x0 = rand(N,1) * 100;       %oppure considero il vettore nullo come pos iniziale
nmax = 1000;
toll = 10^(-8);

%variabili per memorizzare dati
kterm = 0;
lista_punti = cell(nmax,1);


%algoritmo
[xk,lista_punti,kterm] = discesa(A, b, x0, nmax, toll,lista_punti);

%valutazione errore rispetto al metodo diretto
xt = A\b;
ea = norm(xk-xt);
er = norm(xk-xt)/norm(xt);
rk = b-A*xk;
```

```
%Andamento del residuo relativo norm(r)/norm(b)
res_vect = zeros(kterm,1);
for i = 1:kterm
    xc = lista_punti{i,1};
    rc = (norm(b-A*xc)/norm(b));
    res_vect(i) = rc;
end


%grafico del residuo
figure(1);
hold on
semilogy(1:kterm,res_vect,'blue');
xlabel('Numero di iterazioni')
ylabel('Residuo relativo')
legend(strcat('cond = ',int2str(mu)))
set(gca, 'YScale', 'log')
hold off
```

Figure 2.2: Test e analisi dell'algoritmo di discesa del gradiente

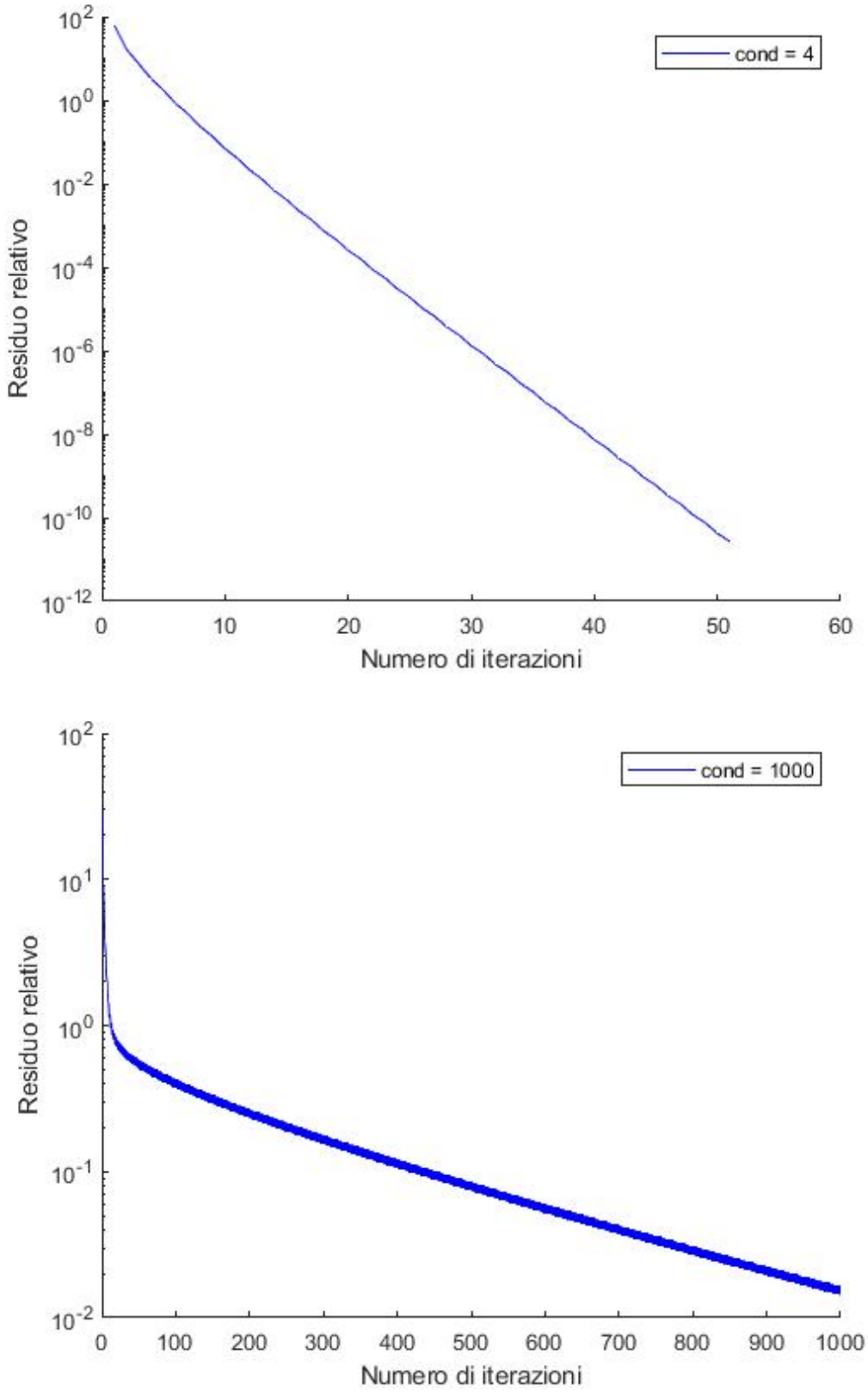

Figure 2.3: Andamento del residuo relativo per diversi indici di condizionamento

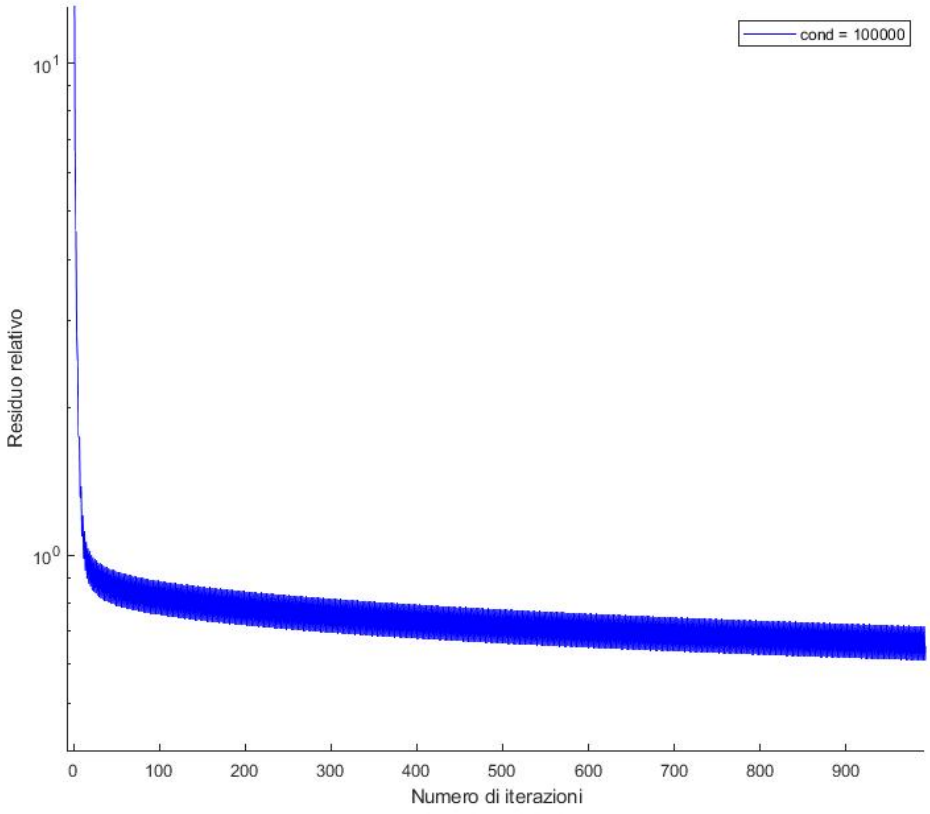

Figure 2.4: Andamento del residuo relativo per diversi indici di condizionamento

2.1.2 Visualizzazione grafica dell'algoritmo

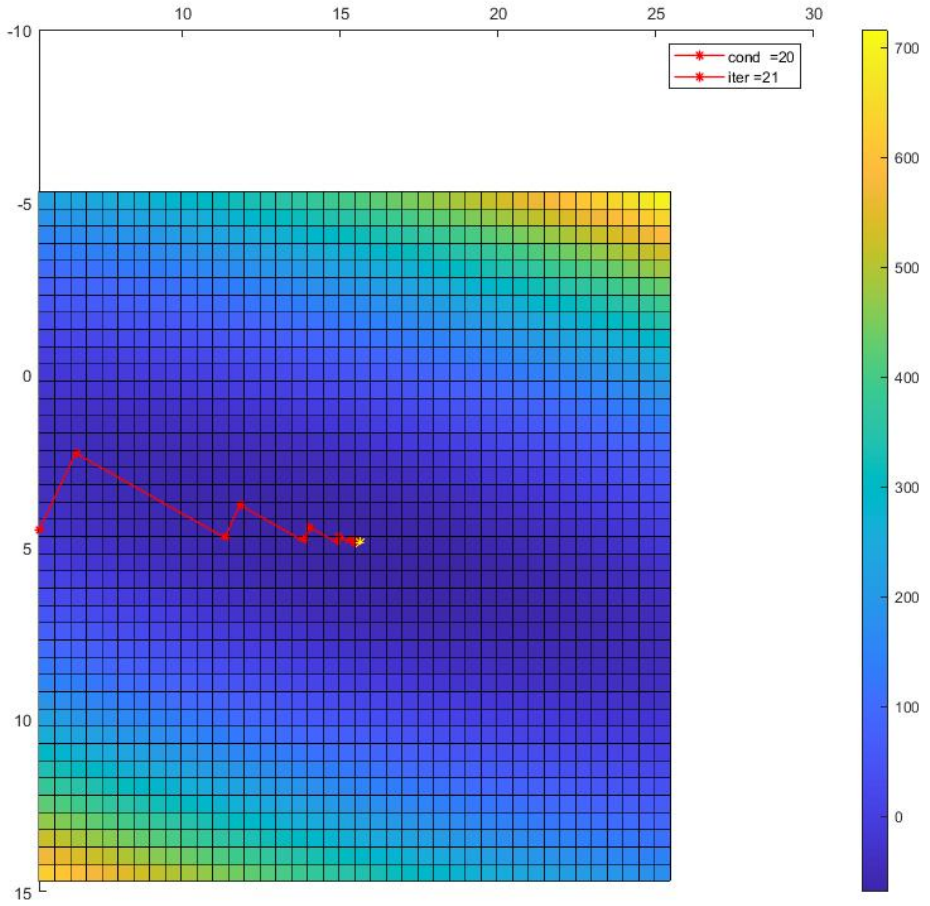

Figure 2.5: Visualizzazione grafica dell'algoritmo

Per effettuare una visualizzazione grafica dell'algoritmo in 3 dimensioni, si fissa la dimensione di A pari a 2. Si impone, poi, un numero di iterazioni massime pari a 50, per evitare overflow di memoria nel rendering del grafico. Si calcola il paraboloide ellittico convesso a partire da A e tramite la funzione meshgrid() si crea la griglia di punti, la quale è centrata rispetto al minimo della funzione. Per ogni iterazione

dell'algoritmo, si calcola la linea che congiunge x_{k-1} ad x_k e si effettua il plot. Infine, tramite la funzione surf() si visualizza la superficie del paraboloide. Il punto di minimo è mostrato in giallo, mentre le iterazioni x_k in rosso. Lo script è mostrato in figura 2.7.

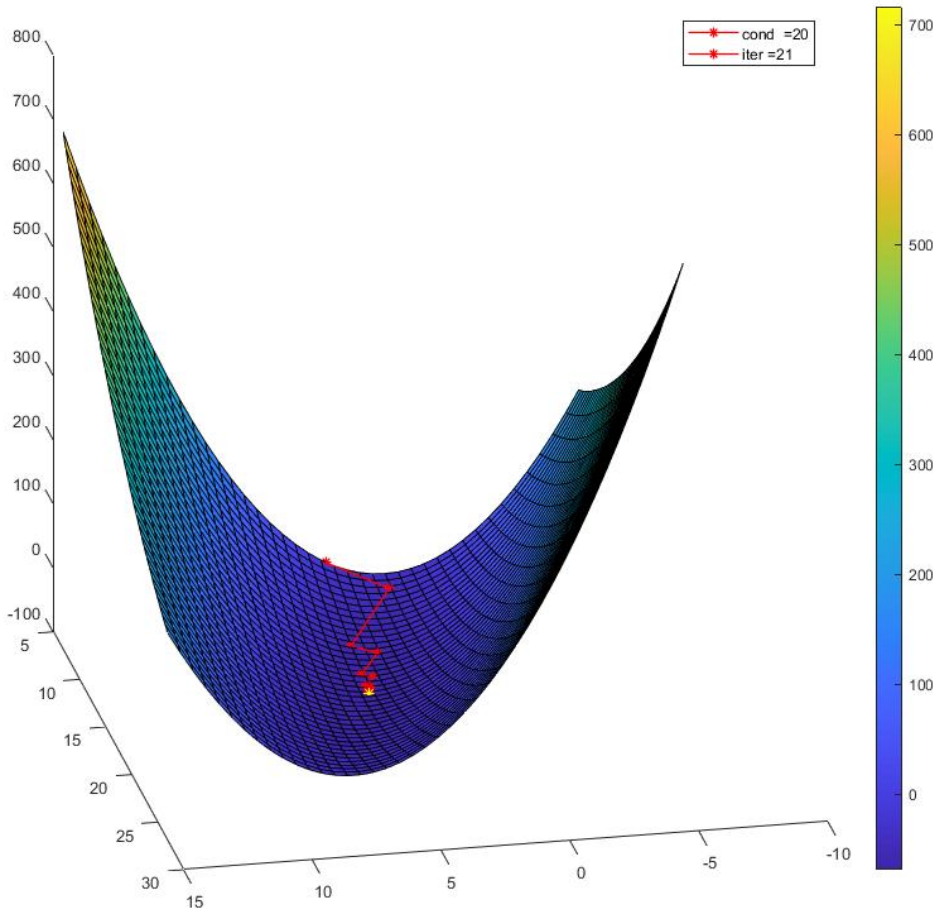

Figure 2.6: Visualizzazione grafica dell'algoritmo

```
%visualizzazione
syms x y
f = symfun(0.5*(A(1,1) * x^2 + (A(1,2)+A(2,1)) * x*y + A(2,2) * y^2) - b(1)*x - b(2)*y,[x y]);

%tuning dei parametri per la visualizzazione
camp = 0.01; %passo di campionamento
off = norm(x0); %scelgo offset
[X,Y] = meshgrid(-off+xt(1):camp:off+xt(1),-off+xt(2):camp:off+xt(2)); %griglia di punti


Z = 0.5*(A(1,1) * X.*X + (A(1,2)+A(2,1)) * X.*Y + A(2,2) * Y.*Y) - b(1)*X - b(2)*Y;
figure(1);
hold on


v = lista_punti{1,1};
xp = v(1);
yp = v(2);
zp = double(f(xp,yp));
for i = 2:kterm
    xp_n = xp;
    yp_n = yp;
    zp_n = zp;
    v = lista_punti{i,1};
    xp = v(1);
    yp = v(2);
    zp = double(f(xp,yp));
    pl = line([xp_n xp],[yp_n yp],[zp_n zp]);   %visualizzo le iterazioni dell'algoritmo
    pl.Marker = '*';
    pl.Color = 'red';
    pl.LineWidth = 1;
end
surf(X,Y,Z);

xp = xk(1);
yp = xk(2);
zp = double(f(xp,yp));
plot3(xp,yp,zp,'y*');
legend(strcat('cond  = ',int2str(mu)),strcat('iter = ',int2str(kterm)));
view(3);
hold off
```

Figure 2.7: Script Visualizzazione grafica dell'algoritmo

2.2 Implementazione del metodo del gradiente coniugato

Di seguito è riportata l'implementazione in Matlab del metodo dei gradienti coniugati, sia in versione standard che precondizionata, per quest'ultima si forniscono le matrici R1 ed R2 per poter calcolare la matrice di precondizionamento R.

```
function [xk,lista_punti,kterm] = gradienteconiugato(A, b, x0, nmax, toll,lista_punti)
    xk = x0;
    rk = b-A*xk;
    vk = rk;
    k = 0;
    while ((norm(rk) >= toll) && (k < nmax))
        lista_punti{k+1,1} = xk;
        alphak = vk' * rk / (vk' * A * vk);
        xk = xk + alphak * vk;
        rk = b - A * xk;
        betak = vk' * A * rk / (vk' * A * vk);
        vk = rk-betak*vk;
        k = k+1;
    end
    lista_punti{k+1,1} = xk;
    kterm = k+1;
    if k == nmax
        warning('Convergenza non raggiunta in nmax iterazioni');
    end
end
function [xk,lista_punti,kterm] = p_gradienteconiugato(A, b, x0, nmax, toll, R1, R2, lista_punti)
    R = R1*R2;
    xk = x0;
    rk = b-A*xk;
    zk = R\rk;
    vk = zk;
    k = 0;
    while ((norm(rk) >= toll) && (k < nmax))
        lista_punti{k+1,1} = xk;
        alphak = vk' * rk / (vk' * A * vk);
        xk = xk + alphak * vk;
        rk = b - A * xk;
        zk = R\rk;
        betak = vk' * A * zk / (vk' * A * vk);
        vk = zk-betak*vk;
        k = k+1;
    end
    lista_punti{k+1,1} = xk;
    kterm = k+1;
    if k == nmax
        warning('Convergenza non raggiunta in nmax iterazioni');
    end
end
```

Figure 2.8: Algoritmo dei gradienti coniugati

2.2.1 Test dell'algoritmo

Lo script per il test dell'algoritmo è molto simile a quello di discesa del gradiente 2.2. Essendo il metodo più veloce, si abbassa la tolleranza a $eps(\|b\|)$.

Nel caso precondizionato si utilizza come precondizionatore $R = R1R1^T$, con R1 dato dalla fattorizzazione di Cholesky incompleta tramite il comando ichol().

Dall'analisi dell'andamento dei residui relativi si nota che la convergenza del metodo precondizionato è molto più veloce di quello standard (tranne nel caso limite μ -> ∞, in cui non si ha la convergenza 2.11). Nei casi in cui $\mu >> 1$, si raggiunge la convergenza numerica 2.10, in cui l'algoritmo oscilla a livelli di accuratezza dell'epsilon macchina.

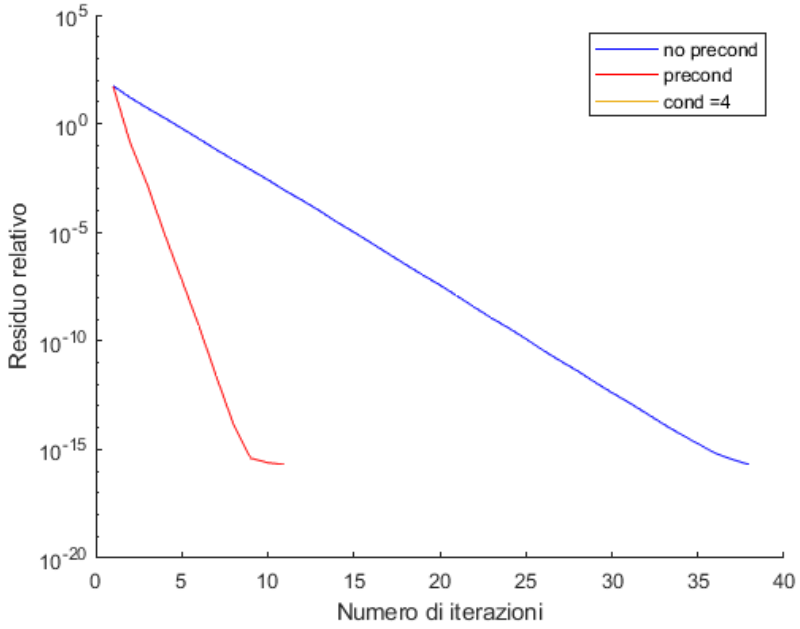

Figure 2.9: Andamento del residuo relativo per diversi indici di condizionamento

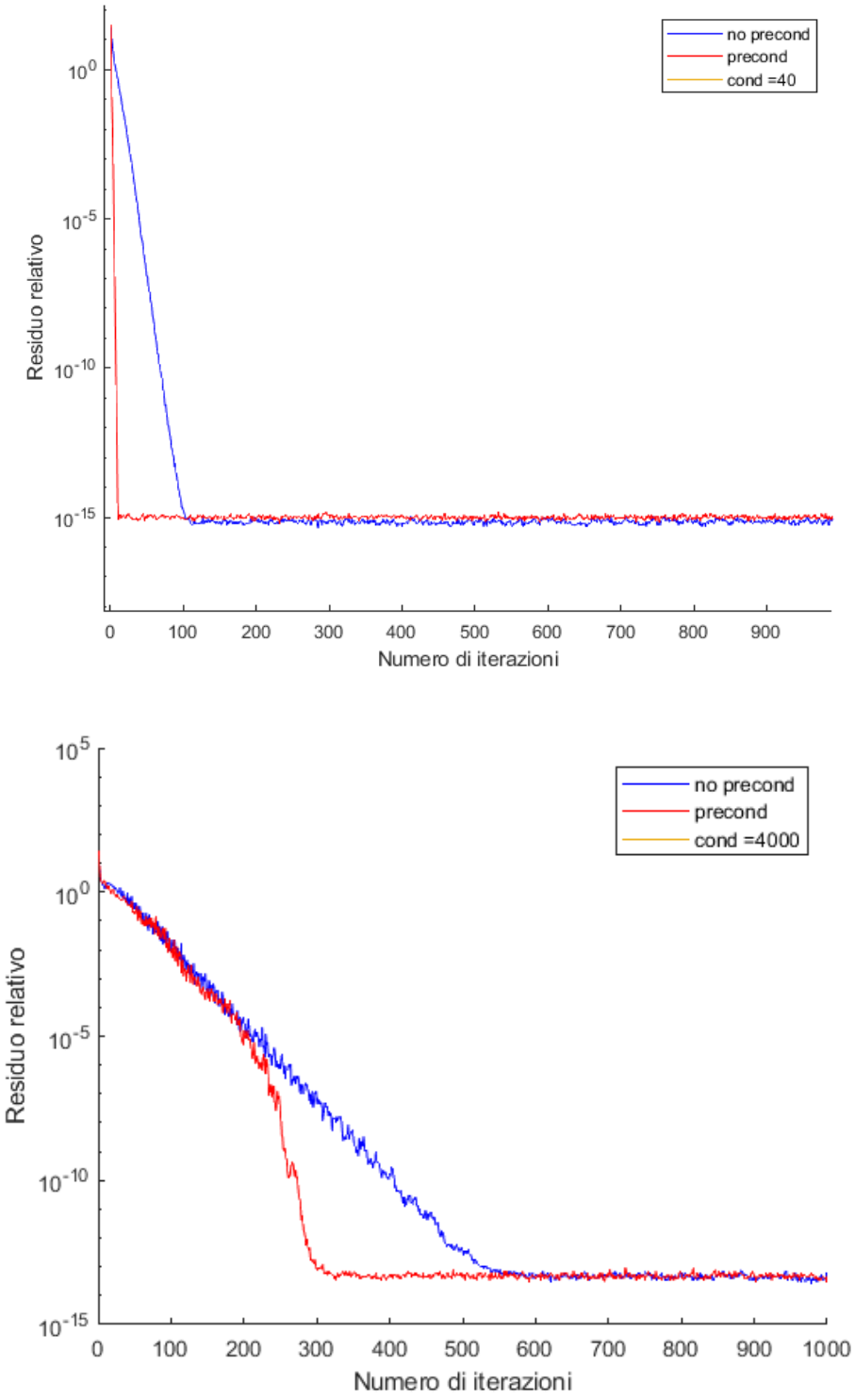

Figure 2.10: Andamento del residuo relativo per diversi indici di condizionamento

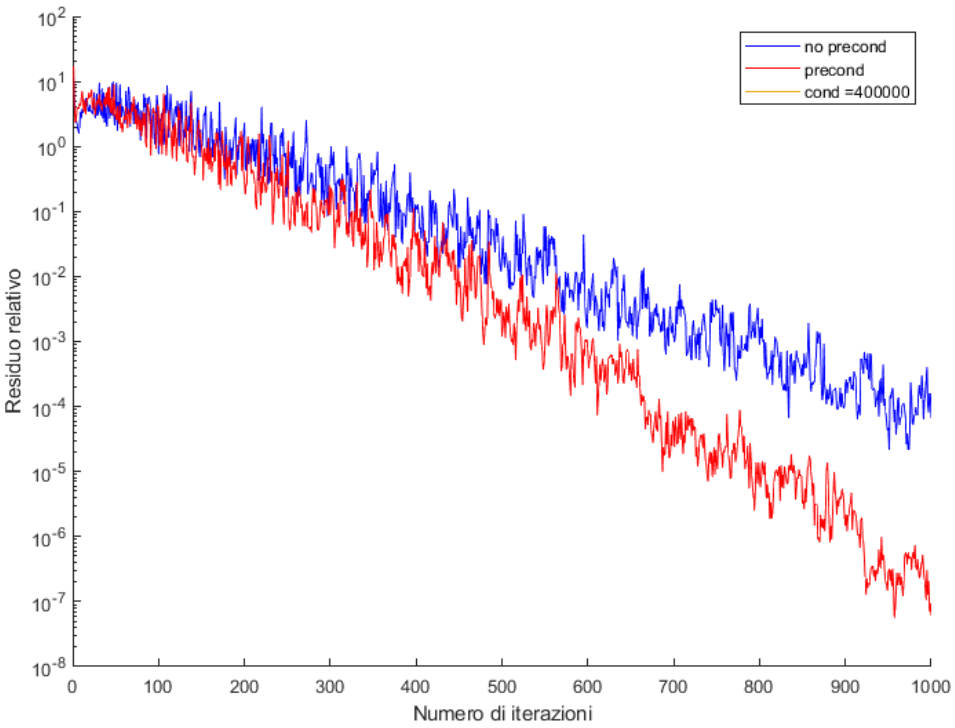

Figure 2.11: Andamento del residuo relativo per diversi indici di condizionamento

2.2.2 Visualizzazione grafica dell'algoritmo

Lo script per la visualizzazione è analogo al precedente. In rosso sono mostrate le iterazioni del CG standard, mentre in ciano le iterazioni dell'algoritmo di discesa. L'algoritmo di CG si avvicina molto più velocemente al punto di minimo, anche nel caso in cui non si raggiunge la convergenza.

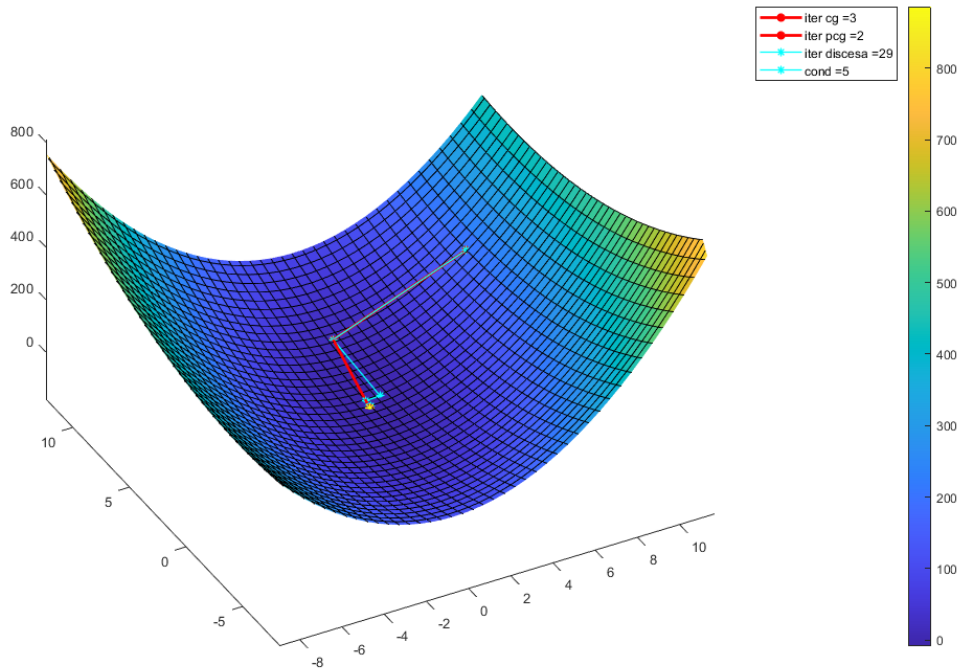

Figure 2.12: Visualizzazione grafica dell'algoritmo

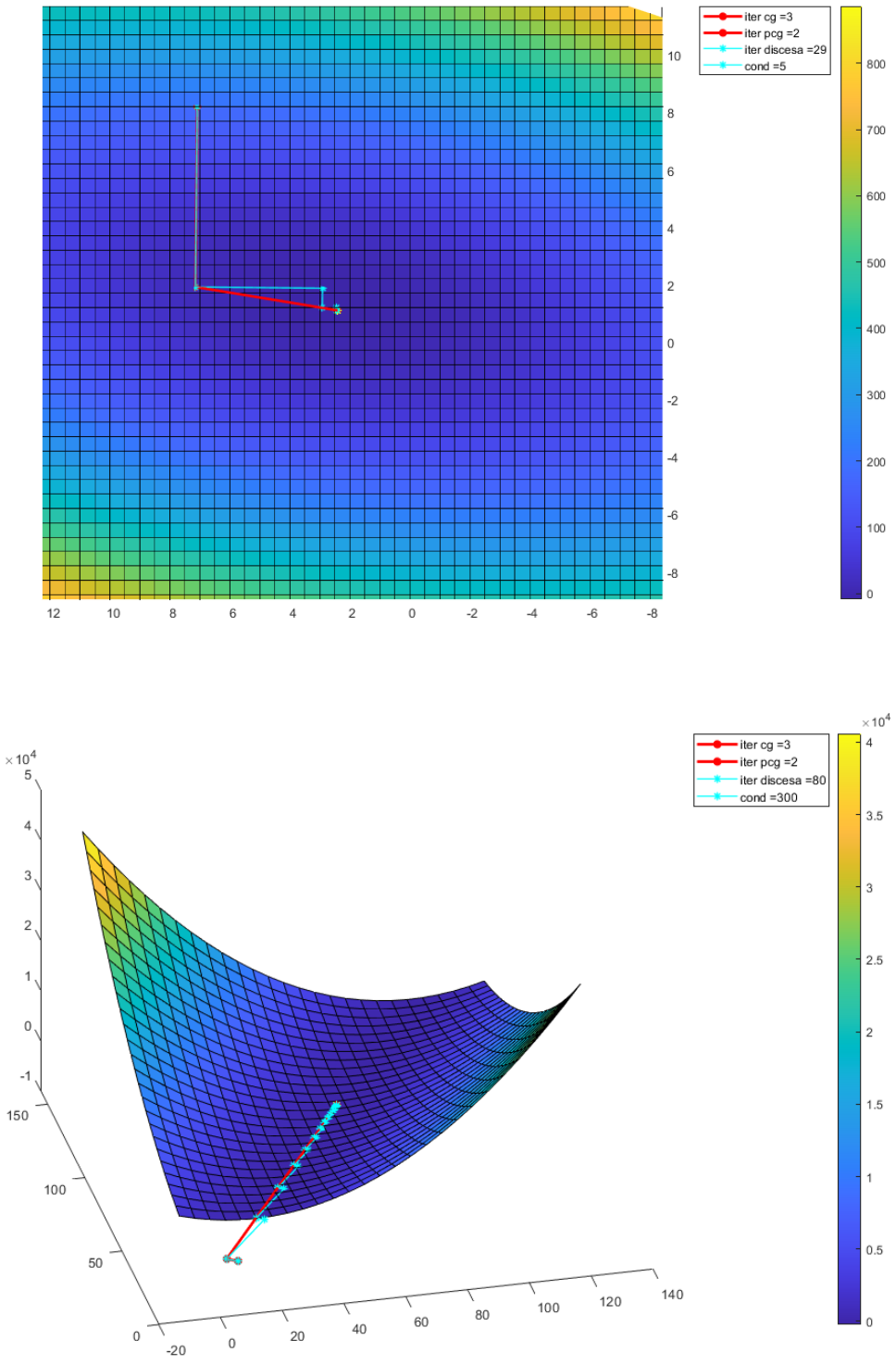

Figure 2.13: Visualizzazione grafica dell'algoritmo

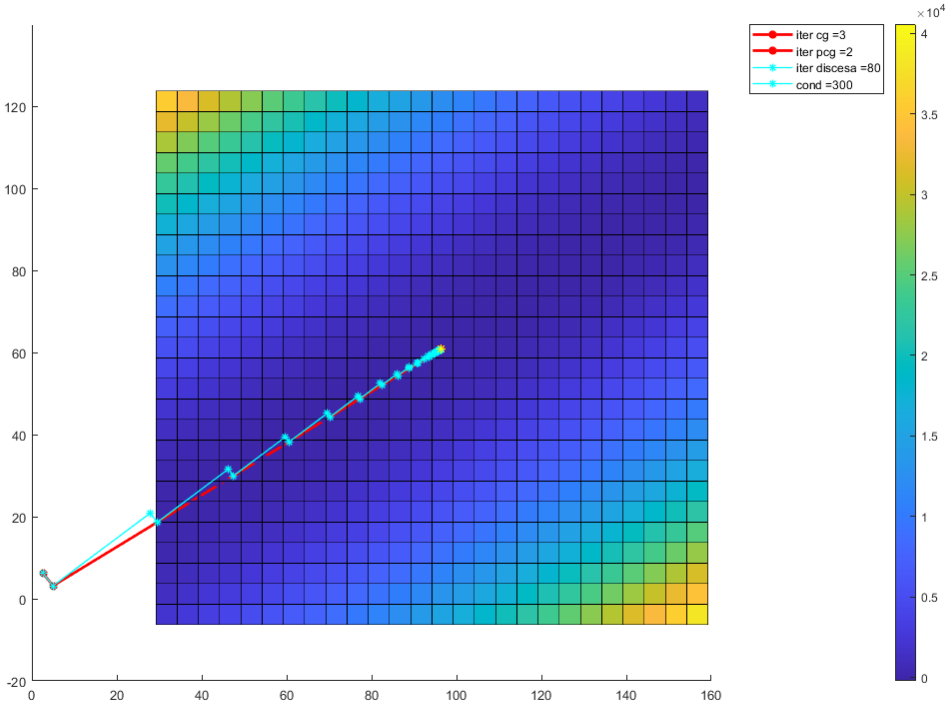

Figure 2.14: Visualizzazione grafica dell'algoritmo

2.3 Confronto tra gli algoritmi

Si è effettuato il confronto tra gli algoritmi precedentemente implementati al variare dell' indice di condizionamento e su matrici di dimensione pari a 100. Gli algoritmi sono stati confrontati con **pcg()** e **gmres()**, implementati da Matlab e precondizionati con R. Tutti gli algoritmi utilizzano le stesse condizioni iniziali e la stessa tolleranza (pari a 10^{-12}).

2.3.1 Andamento del residuo relativo

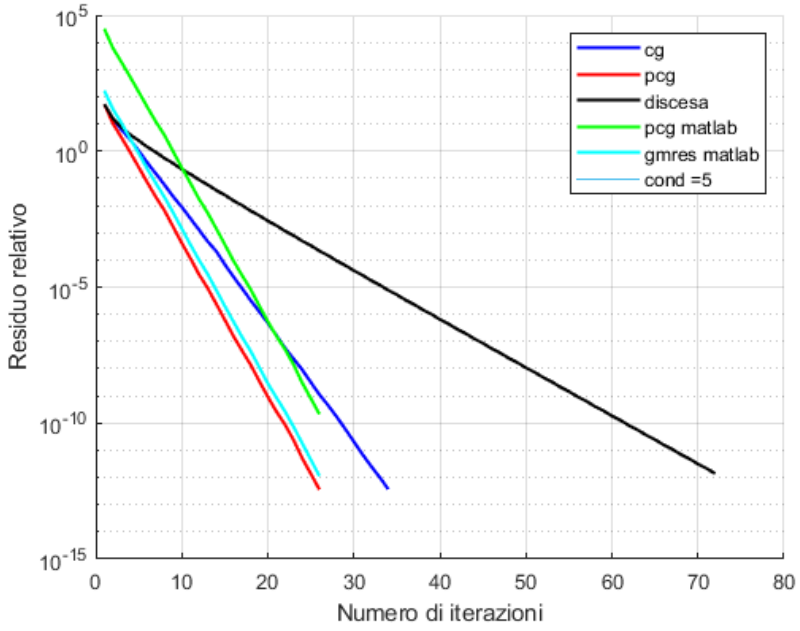

Figure 2.15: Andamento del residuo relativo per diversi indici di condizionamento

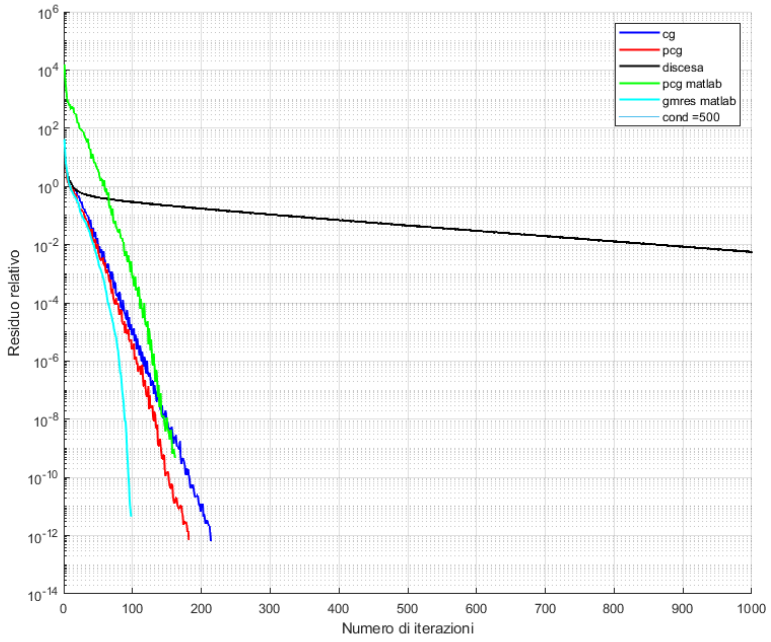

Figure 2.16: Andamento del residuo relativo per diversi indici di condizionamento

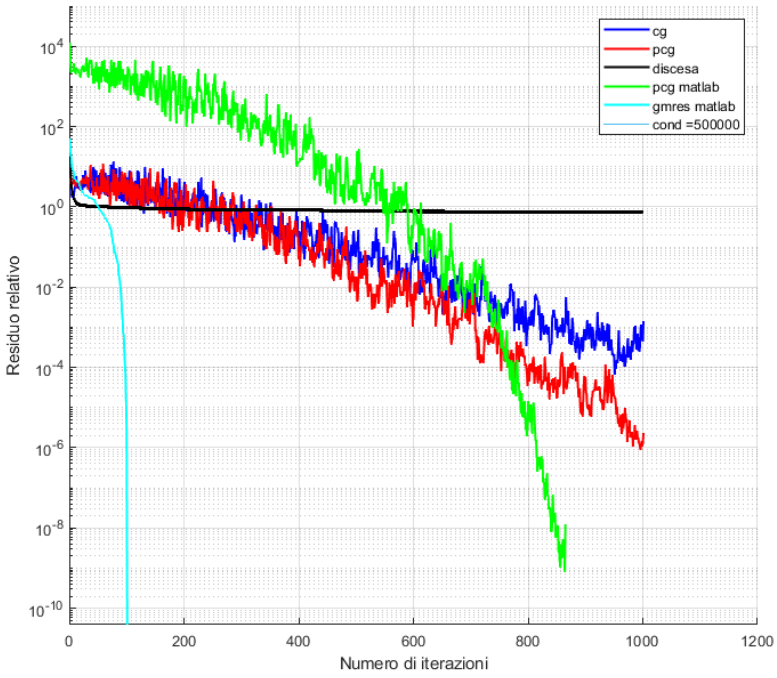

Figure 2.17: Andamento del residuo relativo per diversi indici di condizionamento

Il metodo migliore risulta essere gmres(), il quale ha una convergenza ripida anche nel caso di condizionamento molto elevato.

2.3.2 Valutazione delle prestazioni

Si è effettuato il confronto tra gli algoritmi precedentemente implementati e quelli forniti da Matlab al variare dell' indice di condizionamento e su matrici di dimensione pari a 100.
Le metriche di confronto utilizzate sono: l'errore assoluto e relativo, rispetto alla soluzione ottenuta con metodo diretto; il numero di iterazioni impiegato; il tempo di esecuzione.

μ	CG	PCG	GD	PCGM	GMRES
5	6.07e-12	9.564e-12	1.618e-11	9.553e-12	1.02e-11
50	6.014e-11	7.836e-11	2.022e-10	8.854e-11	2.295e-10
500	3.858e-10	2.314e-10	2.539	3.055e-10	6.54e-10
5000	9.212e-10	5.409e-10	3184	1.684e-10	9.804e-08
50000	8.903e-07	1.566e-08	59260	1.636e-08	4.231e-08

Table 2.1: Errore assoluto

μ	CG	PCG	GD	PCGM	GMRES
5	4.089e-13	6.443e-13	1.09e-12	6.435e-13	6.872e-13
50	5.82e-13	7.582e-13	1.956e-12	8.567e-13	2.22e-12
500	5.728e-13	3.436e-13	0.00377	4.536e-13	9.709e-13
5000	1.449e-13	8.504e-14	0.5006	2.649e-14	1.541e-11
50000	1.401e-11	2.465e-13	0.9325	2.575e-13	6.657e-13

Table 2.2: Errore relativo

μ	CG	PCG	GD	PCGM	GMRES
5	34	26	72	26	26
50	88	71	613	70	65
500	213	164	1000	147	95
5000	504	362	1000	289	100
50000	1001	708	1000	511	101

Table 2.3: Numero di iterazioni

μ	CG	PCG	GD	PCGM	GMRES
5	0.0006164	0.006614	0.0006104	0.003583	0.004319
50	0.0009152	0.01133	0.002695	0.002074	0.003927
500	0.001651	0.02464	0.004744	0.003615	0.005419
5000	0.0036	0.05553	0.005047	0.006352	0.006046
50000	0.008029	0.1117	0.005223	0.01164	0.006501

Table 2.4: Tempo di esecuzione

Chapter 3

Applicazione dell'algoritmo di discesa del gradiente

Di seguito è riportato un esempio applicativo del metodo di discesa del gradiente. L'obiettivo è predirre la potabilità dell'acqua in base a delle metriche di qualità.
A partire da un dataset di circa 3200 istanze di valori misurati per delle acque, ad esempio pH o concentrazione di altre sostanze, il classificatore effettua il training per fornire le predizioni sulla potabilità.
L'implementazione è in Python, e si utilizza un classificatore basato sulla *discesa stocastica del gradiente*, (**SGD**).

Importo le librerie

In [48]:
```
import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

from sklearn.linear_model import SGDClassifier
from sklearn.svm import SVC
from sklearn.metrics import precision_score,accuracy_score
```

Caricamento del dataset

In [ ]:
```
df = pd.read_csv('/content/drive/MyDrive/elabcn/water_potability.csv')

df.info()

test_df = pd.read_csv('/content/drive/MyDrive/elabcn/test.csv')
```

Preprocessing dei dati

In [50]:
```
df['ph'].fillna(value=df['ph'].median(),inplace=True) #inserisco le medie delle colonne dove ci sono i campioni nulli
df['Sulfate'].fillna(value=df['Sulfate'].median(),inplace=True)
df['Trihalomethanes'].fillna(value=df['Trihalomethanes'].median(),inplace=True)

test_df['ph'].fillna(value=df['ph'].median(),inplace=True) #inserisco le medie delle colonne dove ci sono i campioni nulli
test_df['Sulfate'].fillna(value=df['Sulfate'].median(),inplace=True)
test_df['Trihalomethanes'].fillna(value=df['Trihalomethanes'].median(),inplace=True)
```

In [51]:
```
X = df.drop('Potability',axis=1).values #elimino la colonna sulla potabilità dal dataset e la inserisco come etichetta
y = df['Potability'].values
```

In [52]:
```
df
```

Out[52]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
0	7.036752	204.890455	20791.318981	7.300212	368.516441	564.308654	10.379783	86.990970	2.963135	0
1	3.716080	129.422921	18630.057858	6.635246	333.073546	592.885359	15.180013	56.329076	4.500656	0
2	8.099124	224.236259	19909.541732	9.275884	333.073546	418.606213	16.868637	66.420093	3.055934	0
3	8.316766	214.373394	22018.417441	8.059332	356.886136	363.266516	18.436524	100.341674	4.628771	0
4	9.092223	181.101509	17978.986339	6.546600	310.135738	398.410813	11.558279	31.997993	4.075075	0
...	...	...	...	...	...	...	...	...	...	...
3267	4.668102	193.681735	47580.991603	7.166639	359.948574	526.424171	13.894419	66.687695	4.435821	1
3268	7.808856	193.553212	17329.802160	8.061362	333.073546	392.449580	19.903225	66.610937	2.798243	1
3269	9.419510	175.762646	33155.578218	7.350233	333.073546	432.044783	11.039070	69.845400	3.298875	1
3270	5.126763	230.603758	11983.869376	6.303357	333.073546	402.883113	11.168946	77.488213	4.708658	1
3271	7.874671	195.102299	17404.177061	7.509306	333.073546	327.459760	16.140368	78.698446	2.309149	1

3272 rows × 10 columns

Split del dataset e rescaling

In [53]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=101) #split del dataset in

scaler = StandardScaler()   #preprocessing dei dati: rescaling e centering rispetto alla media ed alla dev. std
scaler.fit(X_train)         #calcolo media e dev std
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)


test_case = test_df.drop('Potability',axis=1).values
test_case = scaler.transform(test_case)
```

Training sui classificatori

In [55]:
```
modelli =[("SVC", SVC()),
          ("SGDC", SGDClassifier())]

risultati = []
names = []
finalResults = []

for name,model in modelli:
    model.fit(X_train, y_train)
    model_results = model.predict(X_test)
    score = precision_score(y_test, model_results,average='macro')
    risultati.append(score)
    names.append(name)
    finalResults.append((name,score))

finalResults.sort(key=lambda k:k[1],reverse=True)
```

In [56]:
```
finalResults
```

Out[56]:
```
[('SVC', 0.6840374331550803), ('SGDC', 0.5008715923887932)]
```

Esempi di predizioni

In [57]:
```
res = []
modello_utilizzato = modelli[1][1]
for test in test_case:
  res.append(modello_utilizzato.predict([test])[0])

test_df.assign(Prediction = res)
```

Out[57]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability	Prediction
0	6.581878	272.982745	37169.444404	8.114731	416.083481	351.476839	15.129334	79.261026	4.201663	0	0
1	6.755146	231.260131	18536.698647	8.757133	342.548014	385.114648	13.888834	79.302436	5.162730	0	0
2	9.445130	145.805402	13168.529156	9.444471	310.583374	592.659021	8.606397	77.577460	3.875165	1	1
3	9.024845	128.098691	19859.676476	8.016423	300.150377	451.143481	14.770863	73.778026	3.985251	1	1

Bibliography

[1] C. T. Kelley, Iterative Methods for Linear and Nonlinear Equations, North Carolina State University.

[2] Il metodo del Gradiente Coniugato, 'docenti.unina.it/webdocenti-be/allegati/materiale-didattico/34054665'.