Progetto di Esperienze di Programmazione

Implementazione di metodi iterativi in JAVA
per matrici sparse grandi

Marco Telleschi

2020-04-10

Indice

1 Descrizione del Problema

1.1 Matrici sparse

In analisi numerica una *matrice sparsa* è una matrice i cui valori sono quasi tutti zero. Il numero di elementi uguali a zero diviso il numero totale di elementi è detto *sparsità* della matrice. Usando questa definizione una matrice si dirà sparsa quando la sua sparsità sarà maggiore di 0.5.
Memorizzando e manipolando matrici sparse su calcolatori, è buona norma usare algoritmi e strutture dati apposite. Le operazioni standard per matrici dense, infatti sarebbero lente e inefficienti sprecando potere di calcolo e memoria sui valori uguali a zero.
Tipicamente una matrice è memorizzata come un array bidimensionale. Ogni entry dell'array rappresenta un elemento a_{ij} della matrice ed è acceduto mediante gli indici i e j. Per una matrice $m \times n$, l'ammontare di memoria richiesto per la memorizzazione in questo formato è proporzionale a $m \times n$.
Nel caso delle matrici sparse, possiamo ridurre sostanzialmente la memoria occupata memorizzando solamente gli elementi diversi da zero. Ci sono differenti strutture dati che possono essere usate per portare notevoli risparmi di memoria. Il trade-off è portato da un accesso più difficoltoso agli elementi e da strutture addizionali che si rendono necessarie.

1.2 Sistemi lineari

Data una matrice $A \in R^{n \times n}$ e un vettore $b \in R^n$, si chiama *sistema lineare* un sistema di n equazioni della forma

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

dove $x = [x_1, x_2, ..., x_n^T]$ è detto *vettore delle incognite*. Con notazione più compatta il sistema (1) viene anche scritto

$$Ax = b.$$

Risolvere un sistema lineare significa calcolare, se esiste, un vettore x che soddisfa le (1).

La matrice A si dice *matrice dei coefficienti*, il vettore b *vettore dei termini noti*, la matrice $[A|b]$, ottenuta affiancando alla matrice A la colonna b, si dice *matrice aumentata*. Se $b = 0$ il sistema si dice *omogeneo*. Il sistema si dice *consistente* se ha almeno una soluzione.

1.3 Metodi iterativi per sistemi lineari

Per risolvere un sistema lineare come (1) oltre al metodo di Gauss sono disponibili anche i *metodi iterativi*, che sono particolarmente utili quando la matrice è di grandi dimensioni e sparsa.
Sia A non singolare, dove per matrice singolare si intende una matrice quadrata con determinante uguale a zero, e si consideri la decomposizione

$$A = M - N, \tag{2}$$

dove M è una matrice non singolare. Dalla (2), sostituendo nel sistema lineare dato, risulta

$$Mx - Nx = b,$$

cioè

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

Posto

$$P = M^{-1}N \quad \text{e} \quad q = M^{-1}b, \tag{3}$$

si ottiene il seguente sistema equivalente al sistema dato

$$x = Px + q. \tag{4}$$

Dato un vettore iniziale $x^{(0)}$, si considera la successione $x^{(1)}, x^{(2)}, ...$, così definita

$$x^{(k+1)} = Px^{(k)} + q, \quad k \geq 0 \tag{5}$$

La successione $x^{(k+1)}$ si dice *convergente* al vettore x^* e si indica con

$$x^* = \lim_{k\to\infty} x^{(k+1)},$$

se al tendere di k all'infinito le componenti di $x^{(k)}$ convergono alle corrispondenti componenti di x^*. Allora passando al limite nella (5) risulta

$$x^* = Px^* + q, \tag{6}$$

cioè x^* è la soluzione del sistema (4) e quindi del sistema dato.
La relazione (5) individua un *metodo iterativo* in cui, a partire da un vettore iniziale $x^{(0)}$, la soluzione viene approssimata usando una successione $\{x^{(k)}\}$ di vettori. La matrice P si dice *matrice di iterazione del metodo.*
Al variare del vettore iniziale $x^{(0)}$ si ottengono dalla (5) diverse successioni $\{x^{(k)}\}$, alcune delle quali possono essere convergenti e altre no. Un metodo iterativo è detto *convergente* se, qualunque sia il vettore iniziale $x^{(0)}$, la successione $\{x^{(k)}\}$ è convergente.

Teorema. *Il metodo iterativo* (5) *è convergente se e solo se* $\rho(P) < 1$.

dove $\rho(P)$ è definito come il *raggio spettrale* che equivale all'autovalore di modulo massimo della matrice. La condizione espressa da questo teorema è necessaria e sufficiente per la convergenza del metodo (5), ma in generale non è di agevole verifica. Conviene allora utilizzare, quando è possibile, una condizione sufficiente di convergenza di più facile verifica, come quella del seguente teorema.

Teorema. *Se esiste una norma matriciale* $\|.\|$ *per cui* $\|P\| < 1$*, il metodo iterativo* (5) *è convergente.*

In un metodo iterativo ad ogni iterazione il costo computazionale è principalmente determinato dall'operazione di moltiplicazione della matrice P per un vettore, che richiede n^2 operazioni moltiplicativa se la matrice A non ha specifiche proprietà. Se invece A è sparsa, per esempio ha un numero di elementi non nulli dell'ordine di n, la moltiplicazione di P per un vettore richiede molte meno operazioni moltiplicative. In questo caso i metodi iterativi possono risultare vantaggiosi rispetto a quelli diretti.

1.4 Criterio di arresto

Poiché con un metodo iterativo non è ovviamente possibile calcolare in generale la soluzione con un numero finito di iterazioni, occorre individuare dei criteri per l'arresto del procedimento. Il criterio più comunemente usato è del tipo

$$\|x^{(k+1)} - x^{(k)}\| \leq tol.$$

dove *tol* indica una tolleranza prefissata, eventualmente combinato con una condizione sul numero massimo di iterazioni *max_iter* eseguite in modo da garantire comunque (anche in caso di non convergenza) la terminazione del

metodo. Per criteri basati sulla valutazione dell'errore assoluto e relativo in norma si osserva che se $\rho(P) < 1$ allora

$$x^{(k+1)} - x^{(k)} = x^{(k+1)} - x + x - x^{(k)} = (P - I_n)(x^{(k)} - x),$$

da cui

$$\|x^{(k)} - x\| \leq \|(P - I_n)^{-1}\|\|x^{(k+1)} - x^{(k)}\| \leq tol\|(P - I_n)^{-1}\|.$$

1.5 Metodi iterativi di Jacobi e Gauss-Seidel

Fra i metodi iterativi individuati da una particolare scelta della decomposizione (2) sono particolarmente importanti il metodo di Jacobi e il metodo di Gauss-Seidel, per i quali è possibile dare delle condizioni sufficienti di convergenza verificate da molte delle matrici che si ottengono scrivendo problemi differenziali.
Si consideri la decomposizione della matrice A

$$A = D - B - C$$

dove

$$d_{i,j} = \begin{cases} a_{ij} & \text{se } i = j \\ 0 & \text{se } i \neq j, \end{cases} \quad b_{ij} = \begin{cases} -a_{ij} & \text{se } i > j \\ 0 & \text{se } i \leq j, \end{cases} \quad c_{ij} = \begin{cases} 0 & \text{se } i \geq j \\ -a_{ij} & \text{se } i < j, \end{cases}$$

Scegliendo $M = D, \quad N = B + C$, si ottiene il *metodo di Jacobi*. Scegliendo $M = D - B, \quad N = C$, si ottiene il *metodo di Gauss-Seidel*. Per queste decomposizioni risulta $det(M \neq 0)$ se e solo se tutti gli elementi principali di A sono non nulli. Indicando con J la matrice di iterazione del metodo di Jacobi, dalla (3) si ha

$$G = (D - B)^{-1}C,$$

per cui la (5) diviene

$$x^{(k+1)} = Jx^{(k)} + D^{-1}b.$$

Nella pratica il metodo di Jacobi viene implementato nel modo seguente:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right], \quad i = 1, 2, ..., n. \tag{7}$$

In questo metodo quindi le componenti del vettore $x^{(k+1)}$ sostituiscono simultaneamente al termine dell'iterazione le componenti di $x^{(k)}$. Indicando con G la matrice di iterazione del medodo di Gauss-Seidel, dalla (3) si ha

$$G = (D - B)^{-1}C,$$

per cui la (5) diviene

$$x^{(k+1)} = Gx^{(k)} + (D - B)^{-1}b. \tag{8}$$

Per descrivere come il metodo di Gauss-Seidel viene implementato conviene prima trasformare la (8) così

$$\begin{aligned}(D - B)x^{(k+1)} &= Cx^{(k)} + b,\\ Dx^{(k+1)} &= Bx^{(k+1)} + Cx^{(k)} + b,\\ x^{(k+1)} &= D^{-1}Bx^{(k+1)} + D^{-1}Cx^{(k)} + D^{-1}b.\end{aligned}$$

Il metodo di Gauss-Seidel viene allora implementato nel modo seguente:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right], \quad i = 1, 2, ..., n. \tag{9}$$

La differenza fondamentale con il metodo di Jacobi è che qui per calcolare le componenti del vettore $x^{(k+1)}$ si utilizzano anche le componenti già calcolate dello stesso vettore. Quindi nell'implementazione del metodo di Jacobi è necessario disporre contemporaneamente, di entrambi i vettori $x^{(k+1)}$ e $x^{(k)}$, mentre per il metodo di Gauss-Seidel è sufficiente disporre di un solo vettore in cui si sostituiscono le componenti via via che si calcolano. In molte applicazioni il metodo di Gauss-Seidel, che utilizza immediatamente i valori calcolati nella iterazione corrente, risulta più veloce del metodo di Jacobi. Però esistono casi in cui risulta non solo che il metodo di Jacobi sia più veloce del metodo di Gauss-Seidel, ma anche che il metodo di Jacobi sia convergente e quello di Gauss-Seidel no. Un importante risultato che spesso permette di non utilizzare le più complesse condizioni per la verifica della convergenza dei metodi è il seguente teorema:

Teorema. *Se la matrice A è a predominanza diagonale in senso stretto (per righe o per colonne), il metodo di Jacobi e il metodo di Gauss-Seidel convergono.*

1.6 Il metodo iterativo SOR

Un terzo metodo iterativo, chiamato *Successive Over Relaxation (SOR) Method*, o comunemente *metodo del sovrarilassamento*, è una generalizzazione e un perfezionamento del metodo di Gauss-Seidel.
Per ogni metodo iterativo, nel trovare $x^{(k+1)}$ da $x^{(k)}$, ci spostiamo di una certa quantità in una particolare direzione da $x^{(k)}$ a $x^{(k+1)}$. Questa direzione è il vettore $x^{(k+1)} - x^{(k)}$, poiché $x^{(k+1)} = x^{(k)} + (x^{(k+1)} - x^{(k)})$. Se assumiamo che la direzione da $x^{(k)}$ a $x^{(k+1)}$ ci avvicini, non del tutto, alla soluzione corretta x, allora avrebbe senso spostarsi nella medesima direzione $x^{(k+1)} - x^{(k)}$, ma di una quantità maggiore.
Il perfezionamento del metodo di Gauss-Seidel prende la forma di una media pesata tra l'iterazione precedente e quella calcolata dal metodo.

$$x_i^{(k+1)} = \omega(x_i^{(k+1)})_{GS} + (1 - w)x_i^{(k)}$$

dove $(x_i^{(k+1)})_{GS}$ è l'iterazione di Gauss-Seidel e ω è il parametro di rilassamento. L'idea è quindi quella di scegliere un ω che accellererà la convergenza delle iterazioni verso la soluzione.
In termini matriciali, il metodo SOR può essere scritto come

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

dove le matrici D, $-L$ e $-U$ rappresentano rispettivamente la diagonale, la sottomatrice strettamente triangolare inferiore e la sottomatrice strettamente triangolare superiore.
Se $\omega = 1$, il metodo SOR si semplifica con il metodo di Gauss-Seidel. Un teorema (Kahan, 1958) mostra che SOR non converge se ω è fuori dall'intervallo $(0, 2)$. In generale non è possibile calcolare in anticipo il valore di ω che massimizzerà la convergenza di SOR.
In pratica, tipicamente utilizzeremo un computer per eseguire le iterazioni dei tre metodi trattati. Avremo quindi bisogno di un algoritmo implementabile per utilizzare per sistemi $n \times n$. Un possibile set di algoritmi per trovare l'elemento $x_i^{(k+1)}$ dati $x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}$ può essere:

Method	**Algoritmo per eseguire l'iterazione** $k+1$ **for** $i=1$ **to** n **do:**
J	$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right]$
GS	$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right]$
SOR	$x_i^{(k+1)} = (1-\omega) x_i^{(k)} + \frac{\omega}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right]$

2 Algoritmi di soluzione

2.1 Rappresentazione delle matrici sparse

I formati tradizionalmente usati per rappresentare matrici sparse possono essere divisi in due gruppi:

- Quelle che supportano operazione di modifica efficienti. Tipicamente usate per costruire le matrici.
- Quelle che supportano operazioni di accesso e matriciali efficienti.

2.1.1 Rappresentazione compressed sparse row

La rappresentazione *compressed sparse row* (CSR) o *compressed row storage* (CRS) o *Yale format* appartiene al secondo gruppo. Questo formato permette operazioni di accesso alle righe e di moltiplicazione matrice-vettore veloci. Il formato è in uso almeno dalla metà degli anni '60, con la sua prima descrizione completa che apparve nel 1967.
Il formato CSR memorizza una matrice $m \times n$ sparsa M utlizzando tre array mono-dimensionali che chiameremo `values`, `columnIndices` e `rowPointers`. Sia `nnz` il numero di elementi diversi da zero della matrice M.

- Gli array `values` e `columnIndices` sono di dimensione `nnz` e contengono rispettivamente i valori degli elementi diversi da zero e gli indici di quegli elementi.
- L'array `rowPointers` ha un elemento per riga e codifica l'indice della locazione di `values` dove inizia una data riga.

Per esempio, la matrice

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

è una matrice 4×4 con 4 elementi diversi da zero. Gli array saranno rispettivamente

```
values = [5,8,3,6]
columnIndices = [0,1,2,1]
rowPointers = [0,0,2,3,4]
```

Per estrarre una riga si definisce

```
rowStart = rowPointers[row]
rowEnd = rowPointers[row+1]
```

Quindi prendiamo parti di `values` e `columnIndices` che iniziano a `rowStart` e terminano a `rowEnd`.
Ad esempio per estrarre la riga 1 (la seconda) della matrice impostiamo `rowStart=0` e `rowEnd=2`. Quindi prendiamo le parti `values[0:2]=[5,8]` e `columnIndices[0:2]=[0,1]`. Quindi ora sappiamo che nella riga 1 abbiamo due elementi, rispettivamente alla colonna 0 e 1 con valore 5 e 8.
Si nota che in questo formato, il primo valore di `rowPointers` è sempre zero e l'ultimo è sempre `nnz`. Questo valore sarebbe quindi ridondante ma si mantiene per poter utilizzare la formula `rowPointers[i+1]-rowPointers[i]` per trovare la lunghezza di qualsiasi riga `i`. Inoltre il costo ridondante diventa insignificante per una matrice sufficientemente grande.

2.2 Applicazione dei metodi iterativi

Gli algoritmi illustrati nella sezione precedente trovano applicazione attraverso l'utilizzo di calcolatori in vari linguaggi di programmazione tipicamente su matrici sparse grandi.
I seguenti programmi MatLab prendono in input la matrice A, il vettore b ed una approssimazione x_{old} di x e restituiscono in output la nuova approssimazione x_{new} di x generata dal metodo corrispondente. Per il metodo di Gauss-Seidel x_{new} è sovrascritto direttamente in x_{old}.

```
function [x_new] = jacobi_mio(A,b,x_old)
n=length(b);
for k=1:n
    s=0;
    for j=1:k-1;
        s=s+A(k,j)*x_old(j);
    end
    for j=k+1:n
        s=s+A(k,j)*x_old(j);
    end
    x_new(k)=(b(k)-s)/A(k,k);
end
end
```

```
function [x_new] = gauss_seidel_mio(A,b,x_old)
n=length(b);
for k=1:n
    s=0;
    for j=1:k-1
        s=s+A(k,j)*x_new(j);
    end
    for j=k+1:n
        s=s+A(k,j)*x_old(j);
    end
    x_new(k)=(b(k)-s)/A(k,k);
end
end
```

Si implementa poi anche il criterio di arresto indicando un valore *tol* che rappresenta una tolleranza prefissata ed un valore *max_iter* che rappresenta il numero massimo di iterazioni. Nel caso che segue osserviamo il metodo di Jacobi che si arresta quando $\|x^{(k+1)} - x^{(k)}\|_\infty \leq tol$ o $k > max\_iter$.

```
function [x_new] = jacobi_solver(A,b,x_old,tol,max_iter)
err=+inf;
it=0;
while(err>tol && it<=max_iter)
    x_new=jacobi_mio(A,b,x_old);
```

```
        err=norm(x_new'-x_old, 'inf');
        x_old=x_new';
        it=it+1;
    end
    it
    end
```

Le implementazioni MatLab mostrate come esempio sono state seguite per l'implementazione dei tre metodi in JAVA. Si utilizza un costrutto while per applicare il criterio di arresto e al suo interno due costrutti for scorrono gli elementi della matrice per calcolare la sommatoria prevista dai metodi iterativi. Per ogni iterazione sulle righe viene quindi calcolato un elemento dell'approssimazione. Una volta che si esce dal ciclo principale all'interno di un array troviamo il risultato del metodo che viene restituito stampando il numero di iterazioni effettuate.

3 Scelte implementative

Mostriamo quindi tutte le scelte effettuate per l'implementazione in `JAVA` di quanto descritto sopra.

3.1 Implementazione del CSR format

Per implementare i tre array del formato `values`, `columnIndices` e `rowPointers` si utilizzano tre array JAVA rispettivamente uno di double per i valori e due di interi. Per praticità, si memorizza anche un intero `length` contenente la dimensione della matrice.
Le matrici vengono convertite in questo formato a partire dalla rappresentazione tradizionale *compact Matrix*, composta anch'essa da tre array che memorizzano rispettivamente i valori degli elementi diversi da zero e gli indici di riga e colonna.
Gli elementi vengono inseriti nella matrice, memorizzando i valori necessari, gradualmente, in modo da avere una maggiore flessibilità. Per fare ciò gli array della matrice vengono gestiti esplicitamente come array dinamici, raddoppiandone la dimensione ogni volta che il numero di elementi lo richiede.
Una volta che la memorizzazione è terminata, la matrice viene dichiarata *read only* ed è possibile procedere alla conversione nel formato più efficiente.

Gli array della matrice compact vengono ordinati utilizzando Quicksort, prima per riga e successivamente per colonna all'interno di ogni riga.
Si procede quindi alla conversione vera e propria. Scorrendo gli elementi, i vettori `values` e `columnIndices` vengono copiati e per ogni elemento si incrementa la rispettiva riga all'interno di `rowPointers`. Al termine si somma ad ogni elemento di `rowPointers` il precedente in modo da avere anche il numero di elementi diversi da zero nelle righe precedenti.
Per la matrice così convertita oltre che alcune operazioni di `get` vengono fornite una funzione di stampa e una di copia:

- La funzione `print` scorre le righe della matrice e per ognuna di esse stampa gli elementi diversi da zero, sfruttando le proprietà di `rowPointers`, e degli zeri nelle altre posizioni.
- La funzione `copy` alloca una nuova matrice CSR vi copia i tre array. Viene utilizzata per l'applicazione dei metodi iterativi.

Tra le funzioni getter risulta interessante quella per accedere un elemento. La funzione `getElement(int i, int j)` che scorre gli indici colonna della riga passata come parametro, indiduata mediante le proprietà di `rowPointers`, per verificare se ce n'è uno che corrisponde alla colonna dell'elemento richiesto.

3.2 Implementazione dei metodi iterativi

I metodi iterativi sono rappresentati come risolutori di sistemi lineari. Implementano l'interfaccia `LinearSystemSolver` che contiene metodi fondamentali quali:

- `Solver` che prendendo come parametro l'array dei termini noti risolve il sistema lineare.
- `DiagonallyDominant` che controlla la predominanza diagoanle della matrice.
- `subtract` che effettua la sottrazione tra gli elementi di due array.
- `norm` che calcola la norma infinito di un vettore.

Gli ultimi tre metodi vengono implementati all'interno di una classe astratta, essendo utilizzabili genericamente da uno dei tre metodi iterativi.

Per ognuno dei metodi iterativi trattati troviamo una classe specifica. Come costruttore si utilizza quello della classe astratta che effettua una copia della matrice necessaria per l'applicazione dei metodi.
La convergenza dei tre metodi viene assicurata solamente per matrici predominanti diagonali. Se la matrice, su cui si decide di operare, non rispettasse la suddetta proprietà la soluzione potrebbe non convergere.

3.2.1 Implementazione del metodo di Gauss-Seidel

Il metodo `Solver` lavora su una copia della matrice utilizzando l'array `b` dei termini noti, passato come parametro, e due array di ausiliari `x` e `xOld`. All'interno di un ciclo si scorrono, mediante costrutti `for`, gli elementi della matrice sommandone il prodotto per la rispettiva soluzione `x`, se è già stato calcolato, o `xOld`. Al termine dei cicli `for` sono state calcolate le sommatorie del metodo iterativo e si può procedere al calcolo di `x[i]`, con `i` indice della riga che stiamo trattando.
Una volta terminate le righe l'iterazione è completa e si può procedere alla successiva, se non è stato raggiunto il criterio di arresto.
Il ciclo `while` termina quando il numero di iterazioni ha raggiunto il massimo o $\|$`x-xOld`$\|$ è minore della tolleranza fissata. Il metodo JAVA restituisce l'array di double contenente l'approssimazione calcolata e stampa il numero di iterazioni che sono state necessarie.

3.2.2 Implementazione del metodo di Jacobi

Il metodo di Jacobi differisce dal metodo precedente solamente per il fatto che, all'interno del ciclo, ogni elemento della matrice viene moltiplicato, come previsto, solo per la rispettiva soluzione dell'iterazione precedente `xOld`.
Il metodo JAVA rimane inalterato in tutti gli altri aspetti.

3.2.3 Implementazione del metodo SOR

Il metodo SOR aggiunge al prorio costruttore il parametro ω, che verrà poi utilizzato nel calcolo di `x[i]`, con `i` indice che scorre le righe della matrice. Il parametro viene moltiplicato a `x[i]`, calcolato come nel metodo di Gauss-Seidel, e successivamente a questo prodotto viene sommato il calcolo dell'iterazione precedente moltiplicato per l'opposto del suddetto parametro.

4 Testing

4.1 Tecniche

Lo scopo principale per cui sono stati implementati i metodi iterativi e la rappresentazione secondo il metodo CSR è quello di operare in maniera efficiente su matrici sparse grandi. Nel testing quindi si è cercato quindi di sottolineare principalmente questo aspetto.
Si sceglie di lavorare a partire da una matrice identità I di dimensione n, con n parametro selezionabile per ogni test. Alla matrice I si somma una una matrice A con valori -0.27 sulla prima sovradiagonale e sulla prima sottodiagonale. Si ottiene quindi una matrice B formata come segue:

$$B = \begin{bmatrix} 1 & -0.27 & 0 & & & \dots & 0 \\ -0.27 & 1 & -0.27 & 0 & & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 & \dots & 0 \\ & & \ddots & \ddots & \ddots & & \\ 0 & \dots & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & & 0 & -0.27 & 1 & -0.27 \\ 0 & \dots & & & 0 & -0.27 & 1 \end{bmatrix}$$

Alla matrice B si somma quindi una terza matrice C contenente n elementi di valore casuale $0 \leq a \geq 0.59$ in posizione (i, j), con i e j indici anch'essi calcolati in maniera casuale. In questo modo la matrice I mantiene la predominanza diagonale e i metodi iterativi convergeranno.
La matrice viene allocata come detto prima con una rappresentazione *compact* e quindi convertita in formato *CSR*.
Ogni volta che un metodo viene applicato a una matrice si calcola il tempo di esecuzione con la funzione `System.nanoTime()`.

Una volta allocata la matrice si fissa il vettore soluzione *xEsatto* con tutti componenti di valore *1*. Si procede al calcolo del vettore dei termini noti b come $A * xEsatto = b$. Con il vettore b calcolato si procede alla risoluzione del sistema mediante i metodi iterativi per valutare l'errore nel calcolo di x approssimato.
Per ogni iterazione del metodo vengono stamati:

- numero di iterazione;
- delta: differenza in norma infinito tra i vettori soluzioni di due iterazioni successive;
$$||x^{(k+1)} - x^{(k)}||$$
- errore vero: differenza in norma infinito tra vettore soluzione esatto e vettore soluzione dell'iterazione;
$$||xEsatto - x^{(k+1)}||$$
- rapporto tra le due stampe precedenti.
$$\frac{||xEsatto - x^{(k+1)}||}{||x^{(k+1)} - x^{(k)}||}$$

Al termine di ogni applicazione di ognuno dei metodi vengono stampati il numero di iterazioni svolte, il tempo di esecuzione e l'errore definito come segue:
$$||xEsatto - xCalcolato||$$
dove *xCalcolato* è il risulato del metodo iterativo.
Per osservare il variare dei tempi di esecuzione e del numero di iterazioni si può modificare il valore di n, nel `main` della classe di test. In questo modo si possono ottenere matrici di dimensioni diverse al fine di valutare l'efficienza dell'implementazione dei tre metodi iterativi.

4.2 Risultati

Procedendo al testing, nelle modalità illustrate nella sezione precedente, su matrici $n \times n$ si rilevano differenti risultati al crescere di n.

Valore di `n`	**Tempi per l'allocazione in secondi**
100	intorno a 0.004
1000	intorno a 0.010
10000	intorno a 0.052
100000	intorno a 0.200
1000000	intorno a 1.000
10000000	intorno a 9.000

Metodo iterativo	`n`	**Errore vero**	**It.**	**Tempi in secondi**
Jacobi	100	0.45809	72	intorno a 0.01
	1000	0.55443	95	intorno a 0.09
	10000	0.55337	91	intorno a 0.16
	100000	0.15076	93	intorno a 0.78
	1000000	4.55788×10^{-7}	94	intorno a 9.95
	5000000	4.99631×10^{-7}	94	intorno a 50.8
Gauss-Seidel	100	0.45809	41	intorno a 0.01
	1000	0.55443	53	intorno a 0.04
	10000	0.55337	51	intorno a 0.08
	100000	0.15076	52	intorno a 0.22
	1000000	2.39751×10^{-7}	52	intorno a 4.58
	5000000	1.90405×10^{-7}	53	intorno a 26.3
SOR	100	0.45809	30	intorno a 0.008
	1000	0.55443	33	intorno a 0.03
	10000	0.55337	34	intorno a 0.07
	100000	0.15076	34	intorno a 0.16
	1000000	2.94557×10^{-8}	35	intorno a 3.03
	5000000	3.44580×10^{-8}	35	intorno a 17.5

Dai risultati ottenuti possiamo notare una maggiore velocità in termini di numero di iterazioni e di tempo di esecuzione per il metodo di Gauss-Seidel rispetto al metodo di Jacobi e per il metodo SOR rispetto a Gauss-Seidel. In questo modo si trova conferma alle argomentazioni teoriche riportate in precedenza per motivare e descrivere i tre metodi.
Per testare la correttezza dei metodi è stato fatto un confronto dei risultati ottenuti con la semplice implementazione in Matlab della sezione 2.2 alla pagina 9.
La matrice, che segue, è stata scelta per questo test in modo che fosse abbastanza piccola da confrontare i risultati in modo pratico.

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 7 \end{bmatrix}$$

Le approssimazioni ottenute testando i metodi nei due linguaggi sono pressoché identiche e non si differenziano utilizzando un metodo iterativo al posto di un altro. Identico è anche il numero di iterazioni impiegato dai due linguaggi di programmazione, ma questo, al contrario, cambia a seconda del metodo utilizzato.

- Il metodo di Jacobi impiega 5 iterazioni per terminare.
- Il metodo di Gauss-Seidel impiega 2 iterazioni per terminare.
- Il metodo SOR impiega 67 iterazioni per terminare.

Nella tabella che segue si possono osservare i valori dei vettori restituiti.

Metodo iterativo	**Implementazione Matlab**	**Implementazione JAVA**
Jacobi	0.5000 -0.1875 0.3333 0.3036 0.2500 0.1111 1.0000 0.3333 0.5000 -0.0306	0.5 -0.1875 0.3333333333333333 0.30357142857142855 0.25 0.1111111111111111 1.0 0.3333333333333333 0.5 -0.03061224489795917
Gauss-Seidel	0.5000 -0.1875 0.3333 0.3036 0.2500 0.1111 1.0000 0.3333 0.5000 -0.0306	0.5 -0.1875 0.3333333333333333 0.30357142857142855 0.25 0.1111111111111111 1.0 0.3333333333333333 0.5 -0.03061224489795917
SOR	0.5000 -0.1875 0.3333 0.3036 0.2500 0.1111 1.0000 0.3333 0.5000 -0.0306	0.5 -0.1875 0.33333333333333337 0.3035714285714286 0.25 0.1111111111111111 1.0 0.33333333333333337 0.5 -0.030612244897959395

5 Codice JAVA

```
1  package matrix;
2
3  public class compactMatrix {
4    private double values[];
5    private int rows[];
6    private int columns[];
7    private int size;
8    private boolean write;
9
10   public compactMatrix() {
11     size = 0;
12     write = true;
13     values = new double[2];
14     rows = new int[2];
15     columns =  new int[2];
16   }
17
18   public void set(int i, int j, double value) {
19     if (!write) throw new IllegalArgumentException("Matrice
       read only!");
20     size++;
21     if (values.length < size) growUp();
22     values[size-1] = value;
23     rows[size-1] = i;
24     columns[size-1] = j;
25   }
26
27   private void growUp() {
28     double[] $values = new double[size*2];
29     int[] $rows = new int[size*2];
30     int[] $columns = new int[size*2];
31     System.arraycopy(values, 0, $values, 0, size-1);
32     System.arraycopy(rows, 0, $rows, 0, size-1);
33     System.arraycopy(columns, 0, $columns, 0, size-1);
34     values = $values;
35     rows = $rows;
36     columns = $columns;
37   }
38
39   public void makeReadOnly() { write = false; }
40
41   public int getNNZ() { return size;  }
42
```

```
43   public void quickSort(int begin, int end) {
44     rowQuickSort(begin, end);
45     int row = rows[0];
46     begin = 0;
47     for (int i=0; i<size; i++) {
48       if (rows[i] != row) {
49         end = i-1;
50         columnQuickSort(begin, end);
51         begin = i;
52         row = rows[i];
53       }
54       if ( i == size-1) {
55         end = i;
56         columnQuickSort(begin, end);
57       }
58     }
59   }
60
61   public void rowQuickSort(int begin, int end) {
62     if (begin < end) {
63       int partitionIndex = rowPartition(begin, end);
64       rowQuickSort(begin, partitionIndex-1);
65       rowQuickSort(partitionIndex+1, end);
66     }
67   }
68
69   public void columnQuickSort(int begin, int end) {
70     if (begin < end) {
71       int partitionIndex = columnPartition(begin, end);
72       columnQuickSort(begin, partitionIndex-1);
73       columnQuickSort(partitionIndex+1, end);
74     }
75   }
76
77   private int rowPartition(int begin, int end) {
78     int k = (int) (begin + (end - begin - 1) * Math.random())
      ;
79     swapElem(k,end);
80     int pivot = rows[end];
81     int i = (begin-1);
82     for (int j = begin; j<end; j++) {
83       if (rows[j] < pivot) {
84         i++;
85         swapElem(i,j);
86       }
```

```
87      }
88      swapElem(i+1,end);
89      return i+1;
90    }
91
92    private int columnPartition(int begin, int end) {
93      int k = (int) (begin + (end - begin - 1) * Math.random())
        ;
94      swapElem(k,end);
95      int pivot = columns[end];
96      int i = (begin-1);
97      for (int j = begin; j<end; j++) {
98        if (columns[j] < pivot) {
99          i++;
100         swapElem(i,j);
101       }
102     }
103     swapElem(i+1,end);
104     return i+1;
105   }
106
107   private void swapElem(int i, int j) {
108     int $row = rows[i];
109     int $col = columns[i];
110     double $val = values[i];
111     rows[i] = rows[j];
112     columns[i] = columns[j];
113     values[i] = values[j];
114     rows[j] = $row;
115     columns[j] = $col;
116     values[j] = $val;
117   }
118
119   public double getValue(int i) { return values[i]; }
120   public int getColumn(int i) { return columns[i]; }
121   public int getRow(int i) { return rows[i];  }
122
123   public int getSize() {
124     if (rows[size-1] > columns[size-1]) return rows[size
        -1]+1;
125     else return columns[size-1]+1;
126   }
127 }
```

```java
package matrix;
import java.util.Arrays;

public class CRSMatrix {
  private double[] values;
  private int[] columnIndices;
  private int[] rowPointers;
  private int length;

  public CRSMatrix(int n, int nnz) {
    if ((nnz/(n*n)) > 0.5) throw new IllegalArgumentException
   ("Matrice non sparsa!");
    this.values = new double[nnz];
    this.rowPointers = new int[n+1];
    this.columnIndices = new int [nnz];
    this.length = n;
  }

  public CRSMatrix(compactMatrix M) {
    this(M.getSize(), M.getNNZ());
    for (int i = 0; i<M.getNNZ(); i++) {
      values[i] = M.getValue(i);
      columnIndices[i] = M.getColumn(i);
      rowPointers[M.getRow(i)+1] += 1;
    }
    for (int i=1; i<=M.getSize(); i++)
      rowPointers[i] += rowPointers[i-1];
  }

  public double getElement(int i, int j) {
    if (i >= length  && j >= length) throw new
   IndexOutOfBoundsException();

    for (int k = rowPointers[i]; k < rowPointers[i+1]; k++) {
      if (columnIndices[k] == j)
        return values[k];
      if (columnIndices[k] > j)
        return 0;
    }
    return 0;
  }

  public CRSMatrix copy() {
    CRSMatrix M = new CRSMatrix(length, values.length);
    for (int i=0; i<rowPointers[length]; i++)
```

```
44        M.values[i] = values[i];
45      for (int i=0; i<length+1; i++)
46        M.rowPointers[i] = rowPointers[i];
47      for (int i=0; i<rowPointers[length]; i++)
48        M.columnIndices[i] = columnIndices[i];
49      return M;
50    }
51
52    public double getSummation(double[] xOld, int i) {
53      double sum = 0;
54      for (int k=rowPointers[i]; k<rowPointers[i+1]; k++) {
55        if (i != columnIndices[k]) {
56          sum += values[k]*xOld[columnIndices[k]];
57        }
58      }
59      return sum;
60    }
61
62    public double getSummation(double[] xOld, double[] x, int i
       ) {
63      double sum = 0;
64      for (int k=rowPointers[i]; k<rowPointers[i+1]; k++) {
65        if (columnIndices[k] < i)
66          sum += values[k]*x[columnIndices[k]];
67        if (columnIndices[k] > i)
68          sum += values[k]*xOld[columnIndices[k]];
69      }
70      return sum;
71    }
72
73    public double[] times(double[] x) {
74      double[] b = new double[length];
75      Arrays.fill(b, 0);
76      for (int i=0; i<length; i++) {
77        for (int k=rowPointers[i]; k<rowPointers[i+1]; k++) {
78          b[i] += values[k] * x[columnIndices[k]];
79        }
80      }
81      return b;
82    }
83
84    public int getLength() { return length; }
85    public double [] getValues() { return values; }
86    public int [] getColumns() { return columnIndices; }
87    public int [] getRows() { return rowPointers; }
```

```
88    public int getNNZ() { return rowPointers[length]; }
89 }
```

```java
package linear;

public interface LinearSystemSolver {
  double[] Solver(double[] b, double[] xEsatto);
  double[] subtract(double[] a, double[] b);
  public double norm(double[] a);

}
```

```java
package linear;
import matrix.CRSMatrix;

public abstract class AbstractSolver implements
    LinearSystemSolver {
  protected CRSMatrix A;

  protected AbstractSolver(CRSMatrix M) {
    // Faccio la copia della matrice
    A = M.copy();
  }

  public double[] subtract(double[] a, double[] b) {
    double[] tmp = new double[a.length];
    for (int i=0; i< a.length; i++)
      tmp[i] = a[i] - b[i];
    return tmp;
  }

  public double norm(double[] a) {
    double tmp = Math.abs(a[0]);
    for (int i=1; i<a.length; i++)
      if (Math.abs(a[i]) > tmp)
        tmp = Math.abs(a[i]);
    return tmp;
  }
}
```

```java
package linear;
import java.util.Arrays;
import matrix.CRSMatrix;

public class JacobiSolver extends AbstractSolver implements
    LinearSystemSolver {
  public static final int MAX_ITERATIONS = 200;

  public JacobiSolver(CRSMatrix A) { super(A); }

  public double[] Solver(double[] b, double[] xEsatto) {
    int iterations = 0;
    int n = A.getLength();
    double epsilon = 10e-8;
    double err = Double.POSITIVE_INFINITY;
    double sum, errVero, rapporto;
    double[] x = new double[n];
    double[] xOld = new double[n];
    for (int i=0; i<n; i++) { x[i]=0; xOld[i]=0; }

    System.out.println("it err errVero errVero/err");

    while (err>epsilon && iterations <= MAX_ITERATIONS) {
      for (int i=0; i<n; i++) {
        sum = A.getSummation(xOld, i);
        x[i] = (b[i]-sum)/A.getElement(i, i);
      }
      iterations++;
      err = norm(subtract(x, xOld));
      errVero = norm(subtract(xEsatto, x));
      rapporto = errVero/err;

      System.out.println(iterations+" "+err+" "+errVero+" "+
    rapporto);

      xOld = Arrays.copyOf(x, x.length);
    }
    System.out.println("Numero di iterazioni: "+iterations);
    return x;
  }
}
```

```java
package linear;
import java.util.Arrays;
import matrix.CRSMatrix;

```

```java
public class GaussSeidelSolver extends AbstractSolver
    implements LinearSystemSolver {
  public static final int MAX_ITERATIONS = 200;

  public GaussSeidelSolver(CRSMatrix A) {
    super(A);
  }

  public double[] Solver(double[] b, double[] xEsatto) {
    int iterations = 0;
    int n = A.getLength();
    double epsilon = 10e-8;
    double err = Double.POSITIVE_INFINITY;
    double sum, errVero, rapporto;
    double[] x = new double[n];
    double[] xOld = new double[n];
    for (int i=0; i<n; i++) { x[i]=0; xOld[i]=0; }

    System.out.println("it err errVero errVero/err");

    while (err>epsilon && iterations <= MAX_ITERATIONS) {
      for (int i=0; i<n; i++) {
        sum = A.getSummation(xOld, x, i);
        x[i] = (b[i]-sum)/A.getElement(i, i);
      }
      iterations++;
      err = norm(subtract(x,xOld));
      errVero = norm(subtract(xEsatto, x));
      rapporto = errVero/err;

      System.out.println(iterations+" "+err+" "+errVero+" "+
    rapporto);

      xOld = Arrays.copyOf(x, x.length);
    }
    System.out.println("Numero di iterazioni: "+iterations);
    return x;
  }
}
```

```java
package linear;
import java.util.Arrays;
import matrix.CRSMatrix;

public class SORSolver extends AbstractSolver implements
    LinearSystemSolver {
```

```
6   public static final int MAX_ITERATIONS = 200;
7   // parametro di rilassamento deve essere compreso tra 0 e 2
8   private double w;
9
10  public SORSolver(CRSMatrix A, double w) {
11    super(A);
12    this.w = w;
13  }
14
15  public double[] Solver(double[] b, double[] xEsatto) {
16    int iterations = 0;
17    int n = A.getLength();
18    double epsilon = 10e-8;
19    double err = Double.POSITIVE_INFINITY;
20    double sum, errVero, rapporto;
21    double[] x = new double[n];
22    double[] xOld = new double[n];
23    for (int i=0; i<n; i++) { x[i]=0; xOld[i]=0; }
24
25    System.out.println("it err errVero errVero/err");
26
27    while (err>epsilon && iterations <= MAX_ITERATIONS) {
28      for (int i=0; i<n; i++) {
29        sum = A.getSummation(xOld, x, i);
30        x[i] = ((1-w)*xOld[i]) + (w/A.getElement(i, i))*(b[i
    ]-sum);
31      }
32      iterations++;
33      err = norm(subtract(x,xOld));
34      errVero = norm(subtract(xEsatto, x));
35      rapporto = errVero/err;
36
37      System.out.println(iterations+" "+err+" "+errVero+" "+
    rapporto);
38
39      xOld = Arrays.copyOf(x, x.length);
40    }
41    System.out.println("Numero di iterazioni: "+iterations);
42    return x;
43  }
44 }
```

```java
package main;
import java.util.Arrays;
import java.util.Random;
import linear.GaussSeidelSolver;
import linear.JacobiSolver;
import linear.SORSolver;
import matrix.CRSMatrix;
import matrix.compactMatrix;

public class TestFinal {
  public static void main(String[] args) {

    Random R = new Random(1);
    double startTime, endTime, timeElapsed;
    int N = 1000000;
    startTime = System.nanoTime();
    compactMatrix I = new compactMatrix();
    for (int i=0; i<N; i++)
      I.set(i, i, 1.0);
    for (int i=1; i<N; i++) {
      I.set(i-1,i, -0.27);
      I.set(i,i-1, -0.27);
      int ii = (int) (N*R.nextDouble());
      int jj = (int) (N*R.nextDouble());
      double a = R.nextDouble()*0.59;
      I.set(ii, jj, -a);
    }
    I.makeReadOnly();
    I.quickSort(0, I.getNNZ()-1);

    CRSMatrix CCRS = new CRSMatrix(I);
    endTime = System.nanoTime();
    timeElapsed = endTime-startTime;
    System.out.println("Matrice allocata!");
    System.out.println("Tempo per l'allocazione: "+
   timeElapsed/1000000000+" secondi");

    double[] xEsatto = new double[N];
    Arrays.fill(xEsatto, 1);
    double[] b = new double[N];
    b = CCRS.times(xEsatto);

    //JACOBI
    System.out.println();
    JacobiSolver J = new JacobiSolver(CCRS);
```

```
45      System.out.println("Metodo di Jacobi: ");
46      startTime = System.nanoTime();
47      double[] x = J.Solver(b, xEsatto);
48      endTime = System.nanoTime();
49      timeElapsed = endTime-startTime;
50      System.out.println("Tempo di esecuzione: "+timeElapsed
        /1000000000+" secondi");
51      double errore = J.norm(J.subtract(xEsatto, x));
52      System.out.println("Errore: "+errore);
53
54      //GAUSS-SEIDEL
55      System.out.println();
56      GaussSeidelSolver G = new GaussSeidelSolver(CCRS);
57      System.out.println("Metodo di Gauss-Seidel: ");
58      startTime = System.nanoTime();
59      double[] y = G.Solver(b, xEsatto);
60      endTime = System.nanoTime();
61      timeElapsed = endTime-startTime;
62      System.out.println("Tempo di esecuzione: "+timeElapsed
        /1000000000+" secondi");
63      double errore = G.norm(G.subtract(xEsatto, y));
64      System.out.println("Errore: "+errore);
65
66      //SOR
67      System.out.println();
68      double w = 1.3;
69      SORSolver S = new SORSolver(CCRS, w);
70      System.out.println("Metodo del sovrarilassamento: ");
71      startTime = System.nanoTime();
72      double[] z = S.Solver(b, xEsatto);
73      endTime = System.nanoTime();
74      timeElapsed = endTime-startTime;
75      System.out.println("Tempo di esecuzione: "+timeElapsed
        /1000000000+" secondi");
76      double errore = S.norm(S.subtract(xEsatto, z));
77      System.out.println("Errore: "+errore);
78    }
79  }
```

Si riporta l'output per del test del metodo di Gauss-Seidel con una matrice 5000000×5000000.

```
Matrice allocata!
Tempo per l'allocazione: 4.469456066 secondi

Metodo di Gauss-Seidel:
it err errVero errVero/err
1 2.8099754517849838 3.8099754517849838 1.3558749950519209
2 1.4564504352229466 3.645356635881391 2.502904697421699
3 1.0280709569007673 3.2142049389877565 3.1264427006841338
4 0.7789005587036644 2.5708093743236233 3.3005617284474145
5 0.678500518338445 1.961569570835806 2.891036215623566
6 0.5329902437908017 1.464886409237594 2.7484300628447538
7 0.3997274248745626 1.069003238527321 2.6743304862376704
8 0.29871090581048354 0.7702923327168374 2.578721826800484
9 0.21900770971020656 0.5512846230066308 2.5171927679445476
10 0.158087872598619 0.3959870797770566 2.5048542514229255
11 0.11322388571949427 0.28462898498896094 2.5138598907841145
12 0.08077212694593117 0.20396622481711613 2.525205569411971
13 0.058110926799979534 0.1458552980171366 2.5099461675973127
14 0.04170121307884278 0.10415408493829381 2.49762722109244
15 0.029845462471846496 0.07430862246644732 2.4897795615177127
16 0.0213226132042319 0.05298600926221542 2.484967895571978
17 0.015216565995416431 0.037769443266798985 2.4821266032149425
18 0.010851645983326552 0.026917797283472433 2.4805266707770754
19 0.007735698119166301 0.019182099164306132 2.4796855912435016
20 0.005513223034972814 0.013668876129333318 2.479289526766029
21 0.003928804233782834 0.009740071895550484 2.4791441150968963
22 0.002795688678461583 0.006940503027704326 2.479132807704061
23 0.001994864444831546 0.00494563858287278 2.479185287845666
24 0.001421463006196610 0.003524175576676169 2.479259439966545
25 0.001012888537499812 0.002511287039176357 2.4793320747563734
26 7.217601373117333E-4 0.0017895269018646243 2.479392819517428
27 5.143147541635251E-4 0.0012752121477010991 2.479439171009371
28 3.664958074720559E-4 9.087163402290432E-4 2.4794726752729086
29 2.611631826923144E-4 6.475531575367288E-4 2.479496347307247
30 1.8610452717116033E-4 4.614486303655685E-4 2.47951319282617
```

```
31 1.3261826157418E-4 3.288303687913885E-4 2.479525556195421
32 9.450411305467199E-5 2.343262557367165E-4 2.4795349976053993
33 6.734398558450039E-5 1.669822701522161 2E-4 2.4795424372781416
34 4.7989638897405484E-5 1.1899263125481063E-4 2.479548377290329
35 3.4197676544334144E-5 8.479495471047649E-5 2.4795530948000994
36 2.4369470106311297E-5 6.042548460416519E-5 2.4795567708514095
37 1.73658428956891E-5 4.305964170847609E-5 2.4795595564880544
38 1.237501924045148E-5 3.0684622468024614E-5 2.4795615967789915
39 8.81852754552881E-6 2.1866094922495805E-5 2.479563034713477
40 6.284147926027828E-6 1.5581946996467977E-5 2.479564004521649
41 4.478131227170223E-6 1.1103815769297753E-5 2.4795646232802264
42 3.1911505593562595E-6 7.912665209941494E-6 2.4795649916115807
43 2.274038501060005E-6 5.638626708881489E-6 2.4795651904104252
44 1.6204974642164416E-6 4.018129244665047E-6 2.479565277572299
45 1.1547790887966869E-6 2.8633501558683605E-6 2.479565298374128
46 8.229045649343902E-7 2.0404455909339703E-6 2.4795652836081348
47 5.864081980000435E-7 1.4540373929339268E-6 2.4795652548735667
48 4.178790455888759E-7 1.0361583473450509E-6 2.479565219368429
49 2.977838584605763E-7 7.383744888844745E-7 2.479565187655153
50 2.1220309287528494E-7 5.261713960091896E-7 2.4795651603364175
51 1.512175731299692E-7 3.749538228792204E-7 2.4795651399388174
52 1.0775881786884156E-7 2.6719500501037885E-7 2.4795651093314213
53 7.678976920999503E-8 1.9040523580038382E-7 2.479565152483887
Numero di iterazioni: 53
Tempo di esecuzione: 27.883443372 secondi
Errore: 1.9040523580038382E-7
```

Bibliografia

[Gem] Luca Gemignani. *Lezioni di Calcolo Numerico*.

[Rob] Ornella Menchi Roberto Bevilacqua. *Appunti di Calcolo Numerico*.

Sitografia

[Str15] David M. Strong. *Iterative Methods for Solving $Ax = b$ - The SOR Method*. Accessed on 01-03-2020. Lug. 2015. URL: https://www.maa.org/press/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi-the-sor-method.

[Wik20] Wikipedia. *Sparse Matrix*. Accessed on 04-03-2020. Mar. 2020. URL: https://en.wikipedia.org/wiki/Sparse_matrix.

[Let] LetMeThink. *Jacobi Method*. Accessed on 15-02-2020. URL: https://letmethink.mx/posts/jacobi-method/.

[Mah] Sayan Mahapatra. *Sparse Matrix Representations*. Accessed on 02-03-2020. URL: https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/.

[Noe] Shirley Moore Noel Black. *Successive Overrelaxation Method*. Created by Eric W. Weisstein. Accessed on 21-04-2020. URL: https://mathworld.wolfram.com/SuccessiveOverrelaxationMethod.html.