

# **Metodi del calcolo scientifico**

Libreria per risoluzione di sistemi lineari con metodi iterativi

Volpato Mattia 866316 <sup>\*</sup>  
Andreotti Stefano 851596 <sup>†</sup>

Appello di Giugno 2024

---

<sup>\*</sup>m.volpato4@campus.unimib.it  
<sup>†</sup>s.andreotti7@campus.unimib.it

# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Obiettivo . . . . .	3
1.2	Struttura della libreria . . . . .	3
1.2.1	Scelte implementative . . . . .	3
1.2.2	Architettura . . . . .	3
1.3	Matrici utilizzate . . . . .	6
<b>2</b>	<b>Metodi iterativi stazionari</b>	<b>7</b>
2.1	Metodo di Jacobi . . . . .	7
2.1.1	Implementazione . . . . .	7
2.1.2	Risultati . . . . .	8
2.2	Metodo di Gauss-Seidel . . . . .	8
2.2.1	Implementazione . . . . .	9
2.2.2	Risultati . . . . .	9
<b>3</b>	<b>Metodi iterativi non stazionari</b>	<b>10</b>
3.1	Metodo di discesa del gradiente . . . . .	10
3.1.1	Implementazione . . . . .	10
3.1.2	Risultati . . . . .	10
3.2	Metodo di discesa del gradiente coniugato . . . . .	11
3.2.1	Implementazione . . . . .	11
3.2.2	Risultati . . . . .	12
<b>4</b>	<b>Statistiche per matrice</b>	<b>12</b>
<b>5</b>	<b>Conclusioni</b>	<b>15</b>

# 1 Introduzione

## 1.1 Obiettivo

Lo scopo del progetto è la realizzazione di una mini-libreria per la risoluzione di sistemi lineari (limitatamente al caso di matrici simmetriche e definite positive), in particolare che implementi:

- i *metodi iterativi stazionari* di **Jacobi** e di **Gauss-Seidel**;
- i *metodi iterativi non stazionari* del **gradiente** e del **gradiente coniugato**.

Tutti i metodi risolutivi verranno testate su quattro matrici in *formato sparso*, descritte nella sezione 1.3.

## 1.2 Struttura della libreria

Tutto il codice della libreria è disponibile in questa repository.

### 1.2.1 Scelte implementative

Si è scelto di implementare la libreria in **python** per tre motivi principali:

- la facilità d'uso del linguaggio e l'ampio supporto fornito dalla comunità, che mette a disposizione **librerie efficienti** e che limitano il principale lato negativo del linguaggio (ovvero le performance sul tempo di esecuzione);
- la popolarità del linguaggio, quindi il fatto che potenzialmente la libreria possa usata da molte persone;
- la possibilità di fare utilizzo dei *jupyter notebook*, cioè formato di file eseguibile e composto da celle di testo e di codice, che permettono una migliore interpretazione dei risultati e facilità di visualizzazione delle matrici.

### 1.2.2 Architettura

L'intera **architettura della libreria** è riportata nel grafico 1.

La libreria è stata sviluppata cercando di sfruttare il più possibile la struttura base dei *metodi iterativi*, i quali condividono una *base comune* e si differenziano solo per il metodo di calcolo della successiva iterata: infatti, il *criterio d'arresto* usato risulta sempre essere il **residuo scalato**, poichè quasi tutti i metodi lo calcolano durante l'aggiornamento, risultando quindi computazionalmente più efficiente. Inoltre, si impone anche un limite sul *numero massimo di iterazioni*, oltre il quale si considera la esecuzione interessata fallita.

Lo pseudocodice generico per un qualsiasi **metodo iterativo** (Algoritmo 1)

---

**Algorithm 1** Metodo Iterativo

---

```
function ITERATESOLVER(Matrix A, vector b)
     $x = \text{initialize\_x}_0(A)$ 
     $k = 0$ 
     $r = b - Ax$ 
    while not check_termination( $r, b$ ) do
         $x = \text{update\_x}(A, b, x)$ 
         $k = k + 1$ 
        if  $k > \text{MAX\_ITERATIONS}$  then
            State Convergence_Error
         $r = b - Ax$ 
    return  $x$ 
```

---

viene implementato nel metodo *solve* della classe padre astratta *Solver* come segue:

```

1 def solve(self, A:sp.sparse.csr_matrix, b:np.ndarray, support:any=None) ->
  np.ndarray:
2
3     x = self._initialize_x_0(A.shape[1])
4     k = 0
5     r = b - A.dot(x)
6
7     while not self._check_termination(r, b):
8         x, r, support = self._update_x(A, b, x, support)
9         k += 1
10
11     if k > self.max_iter:
12         self.iter = k
13         raise MaxIterationException(f"Max iteration reached: {k}")
14
15     self.iter = k
16     return x

```

L'utilizzo della variabile *support* permette di evitare calcoli ripetuti inutili nei metodi che necessitano delle strutture di supporto aggiuntive, come in **Jacobi** (sezione 2.1) e **Gauss-Seidel** (sezione 2.2), permettendo a ogni classe figlia di implementare solamente il metodo *update\_x* per il calcolo della iterata successiva (che in *Solver* è astratto).

I metodi di **inizializzazione della soluzione** e di **controllo della terminazione** risultano essere condivisi da tutti i metodi iterativi, e di conseguenza sono implementati nella classe *Solver*. In particolare:

- *initialize\_x\_0* permette di inizializzare la *soluzione iniziale* in due maniere:
  - come il **vettore nullo**  $\underline{0}$ ;
  - come un **vettore** inizializzato **casualmente** con valori reali compresi tra due bound.
- *check\_termination* rappresenta il **criterio di terminazione**, il quale controlla che il rapporto tra la norma del **residuo** e la norma di **b** sia inferiore a una **tolleranza** piccola a piacere:

$$\frac{\|A\underline{x}^{(k)} - \underline{b}\|}{\|\underline{b}\|} < \epsilon \quad (1)$$

Nel caso venga superato il **numero massimo di iterazioni**, viene generata un'eccezione *MaxIterationException* che interrompe l'esecuzione; inoltre, è sempre possibile accedere al numero di iterazioni dell'ultima esecuzione attraverso l'attributo di classe *iter*.

Infine, il metodo di **Gauss-Seidel** richiede la risoluzione di un *sistema lineare triangolare inferiore* per il calcolo dell'**iterata successiva**: a tal fine è stata implementata la classe *TrilSolver*, che si occupa appunto di risolvere tali sistemi tramite la procedura *forward\_substitution*. È stata anche fornita un'implementazione di tale procedura (*forward\_substitution\_naive*) che tuttavia, dovendo far uso dei cicli *for* di **python**, rende il metodo iterativo ingiustamente lento rispetto agli altri (che fanno uso solo di **operazioni tra vettori**). Per far fronte a questo problema, nei **benchmark** è stata utilizzata un'implementazione efficiente della *forward\_substitution* presa dalla libreria **scipy**.

Nel benchmark, il vettore  $\underline{b}$  è stato creato appositamente in modo da ottenere una soluzione  $\underline{x}$  di soli 1, come segue:

$$\underline{b} = A \cdot \underline{x} = A \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

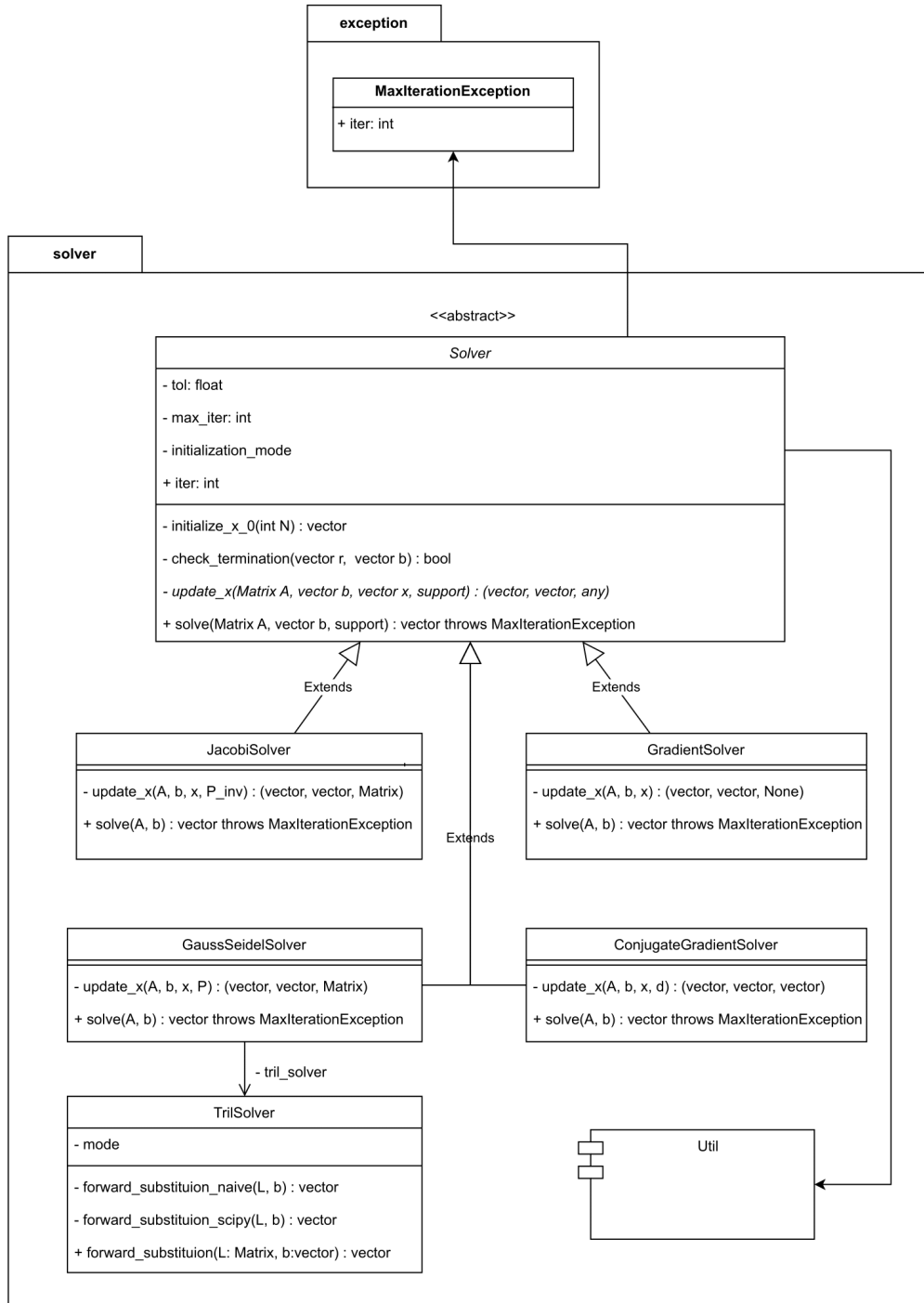


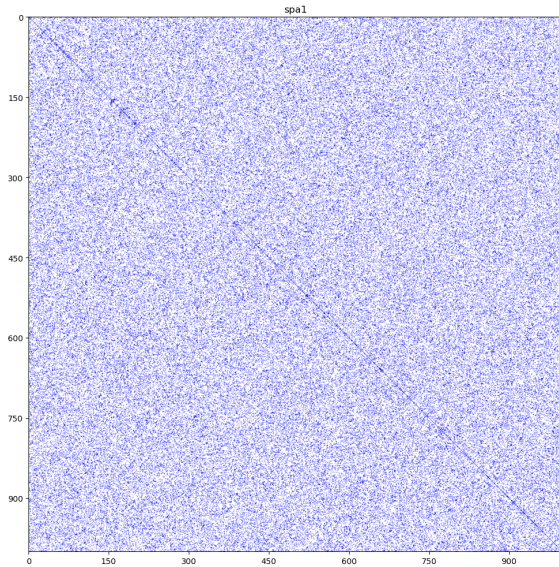
Figure 1: Architettura della libreria.

### 1.3 Matrici utilizzate

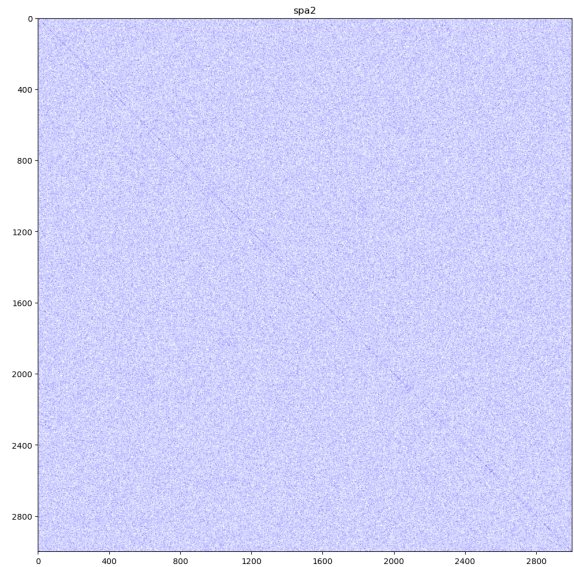
Al fine di testare la libreria sono state utilizzate quattro matrici in **formato sparso**:

- Matrice **spa1** (figura 2a):
  - Dimensione:  $1000 \times 1000$
  - Entrate non zero: 182264
  - Indice di sparsità: 18.23%
- Matrice **spa2** (figura 2b)
  - Dimensione:  $3000 \times 3000$
  - Entrate non zero: 161738
  - Indice di sparsità: 18.13%
- Matrice **vem1** (figura 3a)
  - Dimensione:  $1681 \times 1681$
  - Entrate non zero: 13385
  - Indice di sparsità: 0.47%
- Matrice **vem2** (figura 3b)
  - Dimensione:  $2601 \times 2601$
  - Entrate non zero: 21225
  - Indice di sparsità: 0.31%

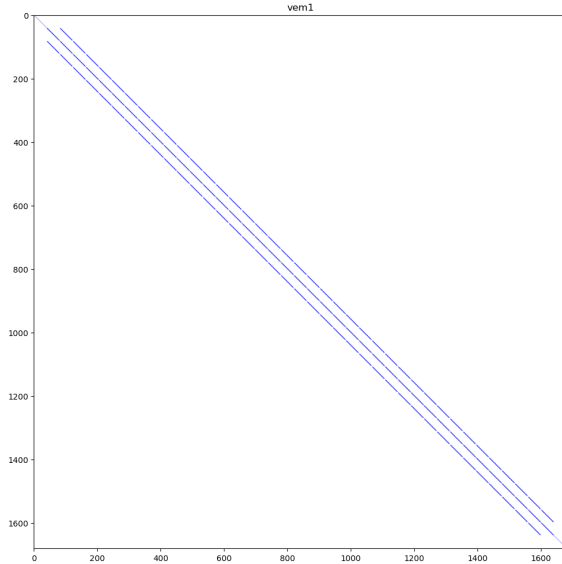
Nei grafici associati sono riportate le distribuzioni delle **entrate diverse da zero**; si noti in particolare come **spa1** e **spa2** siano *matrici sparse* mentre **vem1** e **vem2** risultino essere *matrici a bande*.



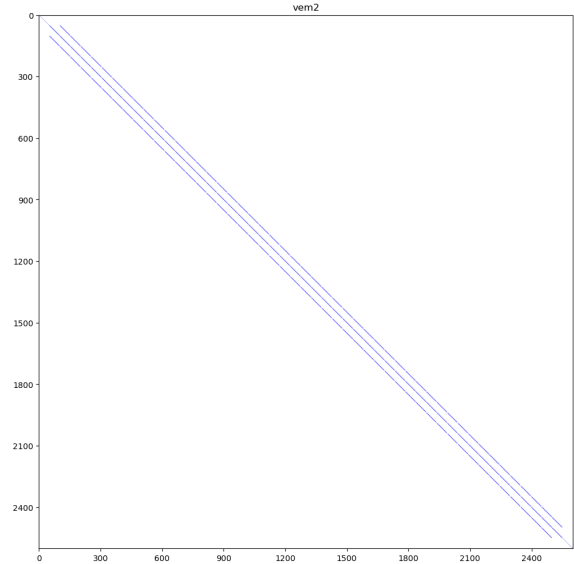
(a) Matrice **spa1**



(b) Matrice **spa2**



(a) Matrice **vem1**



(b) Matrice **vem2**

## 2 Metodi iterativi stazionari

I **metodi iterativi stazionari** usano una strategia basata sullo **splitting** per calcolare la soluzione approssimata  $\underline{x}$ . La decomposizione su cui si basano è:

$$A = P - N \quad (2)$$

Questi metodi sono chiamati *stazionari* poiché il calcolo dell'iterata successiva non dipende dall'iterazione. Per questa categoria, nella libreria sono implementati i metodi di **Jacobi** (sezione 2.1) e di **Gauss-Seidel** (sezione 2.2).

### 2.1 Metodo di Jacobi

Il **metodo di Jacobi** si basa sull'uso della matrice  $P^{-1}$  e del *residuo* per calcolare l'iterata successiva, dove  $P$  è la matrice diagonale costruita (appunto dalla diagonale) di  $A$ ; in particolare, la matrice  $P^{-1}$  si ottiene semplicemente dai reciproci degli elementi sulla diagonale di  $P$ . Da un punto di vista computazionale, la matrice  $P$  (e quindi anche  $P^{-1}$ ) rimane la stessa per tutta l'esecuzione del metodo e, di conseguenza, si calcola una sola volta. La  $k$ -esima iterata è calcolata da

$$x^{(k+1)} = x^{(k)} + P^{-1}(Ax^{(k)} - b) \quad (3)$$

Si noti che all'interno del calcolo dell'iterata è presente anche il calcolo *residuo*; di conseguenza, non sarà necessario calcolarlo nuovamente per verificare la convergenza **condizione di terminazione**.

#### 2.1.1 Implementazione

Il calcolo dell'iterata è implementato nella libreria come

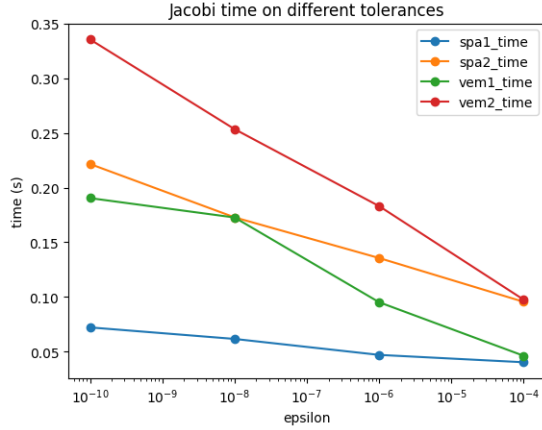
```

1 def _update_x(self, A:sp.sparse.csr_matrix, b:np.ndarray, x:np.ndarray,
2     P_inv:sp.sparse.csr_matrix) -> tuple[np.array, np.array, sp.
3         sparse.csr_matrix]:
4
5     r = b - A.dot(x)
6     x = x + P_inv.dot(r)
7
8     return x, r, P_inv

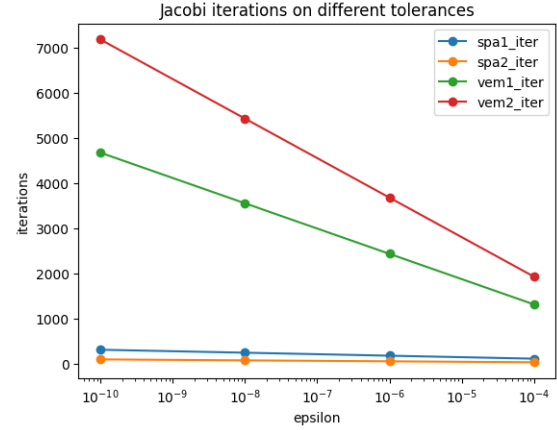
```

### 2.1.2 Risultati

I risultati possono essere verificati manualmente usando la classe implementata nel file *JacobiSolver.py* oppure più facilmente mediante il notebook **benchmark.ipynb**.



(a) **Jacobi** rispetto al **tempo di esecuzione**



(b) **Jacobi** rispetto al **numero di iterazioni**

Dall'analisi dei tempi in figura 4a si può notare un'ottima **efficienza** nei tempi di questo metodo. Una seconda osservazione può invece essere fatta rispetto all'immagine 4b, dove si vede chiaramente come la *struttura delle matrici* influenzi il **numero di iterazioni**: le *matrici sparse* **spa1** e **spa2** convergono dopo poche iterazioni, mentre le *matrici a bande* **vem1** e **vem2** impiegano molte più iterazioni. Ciò è dovuto con ogni probabilità alla stazionarietà del metodo; infatti questo fenomeno si verificherà anche nel metodo di **Gauss-Seidel**, mentre quando si analizzeranno i metodi non stazionari si noterà una grande differenza a livello di iterazioni sulle specifiche matrici.

Nella tabella 1 vengono confrontati i risultati con le caratteristiche delle matrici (fissando la **tolleranza** a  $\epsilon = 1e-8$ ):

Matrix	N	Non-zero entry	Sparsity index	Time (s)	Iterations
spa1	1000	182264	0.182264	0.061566	248
spa2	3000	1631738	0.181304	0.172829	79
vem1	1681	13385	0.004737	0.172698	3553
vem2	2601	21225	0.003137	0.25356	5426

Table 1: Dati delle esecuzioni del **metodo di Jacobi**

## 2.2 Metodo di Gauss-Seidel

Il metodo di **Gauss-Seidel** è una variante del metodo di **Jacobi**, nel quale le matrici  $P$  e  $N$  sono rispettivamente la *triangolare inferiore* e la *triangolare superiore* (senza la diagonale principale) della matrice  $A$ .

Il calcolo della matrice  $P^{-1}$  si effettua tramite la risoluzione del sistema lineare  $Py = r^{(k)}$ , che è possibile risolvere applicando la procedura di *sostituzione in avanti*, essendo la matrice  $P$  triangolare inferiore.

La  $k$ -esima iterata è calcolata come:

$$x^{(k+1)} = x^{(k)} + y \quad (4)$$

Si sottolinea di nuovo che  $y$  è la soluzione del sistema  $Py = r^{(k)}$  e inoltre il residuo  $r$  è calcolato ad ogni iterazione.

Come già riportato nella sezione 1.2.2, nella libreria sono presenti due modalità di calcolare questa matrice: il primo è l'utilizzo della funzione *'naive'* implementata da noi, mentre il secondo è usare la più



efficiente libreria **scipy**: nel benchmark, per consentire confronti *fair* con gli altri metodi, verrà utilizzata la versione di **scipy**.

### 2.2.1 Implementazione

Il calcolo dell'iterata è implementato nella libreria come

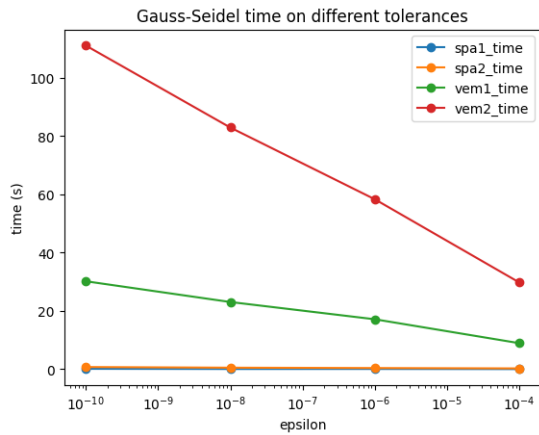
```

1 def _update_x(self, A:sp.sparse.csr_matrix, b:np.ndarray, x:np.ndarray, P:
  sp.sparse.csr_matrix) -> tuple[np.array, np.array, sp.sparse.csr_matrix
  ]:
2
3     r = b - A.dot(x)
4     y = self._tril_solver.forward_substitution(P, r)
5     x = x + y
6
7     return x, r, P

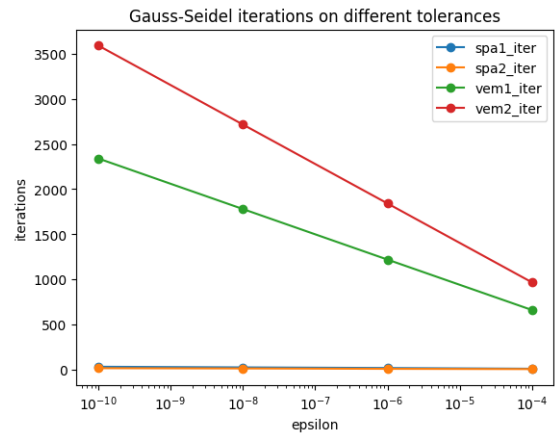
```

### 2.2.2 Risultati

I risultati possono essere verificati manualmente usando la classe implementata nel file *GaussSeidelSolver.py* oppure più facilmente mediante il notebook **benchmark.ipynb**.



(a) Gauss-Seidel rispetto al tempo di esecuzione



(b) Gauss-Seidel rispetto al numero di iterazioni

Con riferimento alla figura 5a si può notare come le *matrici sparse* **spa1** e **spa2** risultino convergere in molto meno tempo rispetto alle *matrici a bande* **vem1** e **vem2**; in maniera analoga, anche il **numero di iterazioni** dipende strettamente dalla struttura della matrice. Da questi dati si può pensare di dedurre che l'**efficienza** di **Gauss-Seidel** sia inversamente proporzionale alla **sparsità** di una matrice: ovvero, più una matrice è sparsa e più il metodo di **Gauss-Seidel** fa fatica a convergere.

Nella tabella 2 vengono confrontati i risultati con alle caratteristiche le matrici (fissando la **tolleranza** a  $\epsilon = 1e^{-8}$ ):

Matrix	N	Non-zero entry	Sparsity index	Time (s)	Iterations
spa1	1000	182264	0.182264	0.09167	25
spa2	3000	1631738	0.181304	0.538953	13
vem1	1681	13385	0.004737	23.076995	1779
vem2	2601	21225	0.003137	82.959381	2715

Table 2: Tabella delle esecuzioni del metodo di Gauss-Seidel

## 3 Metodi iterativi non stazionari

I **metodi iterativi non stazionari** si basano su una diversa formula di aggiornamento della soluzione approssimata, cioè

$$x^{(k+1)} = x^{(k)} + \alpha_k P^{-1} r^{(k)} \quad (5)$$

La differenza sta nel coefficiente  $\alpha$ , che in questo non è stazionario ma bensì dipende dall'iterazione precedente.

### 3.1 Metodo di discesa del gradiente

Il **metodo del gradiente** interpreta una matrice  $A$  *simmetrica e definita positiva* e una funzione  $\iota$  come un paraboloide del quale si vuole trovare il *punto di minimo*, che corrisponde alla soluzione del sistema lineare. Ne segue che la nuova iterazione può essere calcolata come

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} \quad (6)$$

Il calcolo di  $\alpha$  risulta essere quello più oneroso:

$$\alpha_k = ((r^{(k)})^t r^{(k)}) / ((r^{(k)})^t A r^{(k)}) \quad (7)$$

#### 3.1.1 Implementazione

Il calcolo dell'iterata è implementato nella libreria come

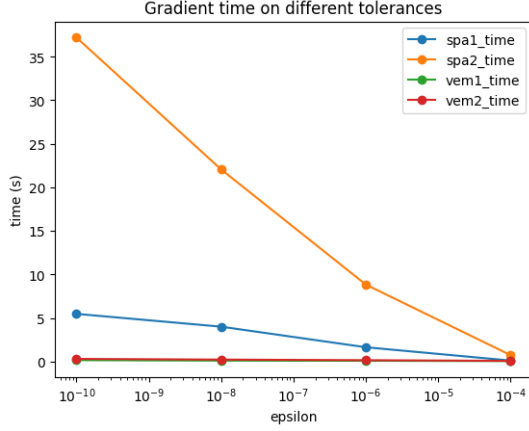
```
1 def _update_x(self, A:sp.sparse.csr_matrix, b:np.ndarray, x:np.ndarray, _:  
    any=None) -> tuple[np.array, np.array, any]:  
2     r = b - A * x  
3     y = A * r  
4  
5     a = r @ r  
6     c = r @ y  
7     x = x + a / c * r  
8  
9     return x, r, _
```

#### 3.1.2 Risultati

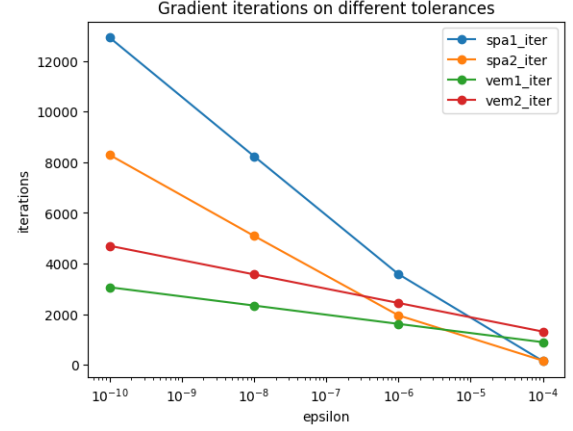
I risultati possono essere verificati manualmente usando la classe implementata nel file *GradientSolver.py* oppure più facilmente mediante il notebook **benchmark.ipynb**.

In figura 6a possiamo notare come le osservazioni fatte nei metodi stazionari siano valide, cioè in questo caso le soluzioni delle matrici a bande risultano molto più veloci a convergere al variare della tolleranza e notiamo soprattutto come una matrice abbastanza grande come spa2 (dimensione 3000x3000) impiega molto più tempo già solo con una tolleranza epsilon di 10e-6. Il secondo grafico riportato 6b rende più evidente la differenza di velocità in termini di iterazioni di convergenza rispetto ai metodi stazionari, infatti in questo caso le matrici a bande sembrano avere una crescita quasi lineare rispetto al numero di iterazioni mentre matrici sparse risultano essere più difficili da far convergere.

Nella tabella 3 vengono confrontati i risultati con le caratteristiche delle matrici (fissando la **tolleranza** a  $\epsilon = 1e^{-8}$ :



(a) Gradiente rispetto al tempo di esecuzione



(b) Gradiente rispetto al numero di iterazioni

Matrix	N	Non-zero entry	Sparsity index	Time (s)	Iterations
spa1	1000	182264	0.182264	4.007221	8234
spa2	3000	1631738	0.181304	22.060645	5088
vem1	1681	13385	0.004737	0.109381	2337
vem2	2601	21225	0.003137	0.22854	3567

Table 3: Tabella delle esecuzioni del metodo di discesa del gradiente

## 3.2 Metodo di discesa del gradiente coniugato

Il **metodo del gradiente coniugato** è una ottimizzazione del **metodo del gradiente**, poiché pone rimedio al fenomeno dello zig-zag, che accade quando gli **autovalori minimo e massimo** sono molto diversi:  $\lambda_{min} \ll \lambda_{max}$ . Per migliorare il metodo si definisce un **vettore ottimale** rispetto ad una direzione come:

$$d * r^{(k)} = 0 \quad (8)$$

In questo modo ho un **vettore ottimale** per quella direzione e idealmente non lo modificherei più lungo la direzione  $d$ . Nella pratica il cambiamento rispetto al gradiente è nel calcolo di  $\alpha$  che diventa:

$$\alpha_k = ((d^{(k)})^t r^{(k)}) / ((d^{(k)})^t A d^{(k)}) \quad (9)$$

Mentre il calcolo dell'iterata successiva rimane

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} \quad (10)$$

### 3.2.1 Implementazione

Il calcolo dell'iterata è implementato nella libreria come

```

1 def _update_x(self, A:sp.sparse.csr_matrix, b:np.ndarray, x:np.ndarray, d:
  np.ndarray) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
2     r = b - A * x
3     y = A * d
4
5     alpha = (d @ r) / (d @ y)
6     x = x + alpha * d
7
8     r = b - A * x
9     w = A * r
10    beta = (d @ w) / (d @ y)

```

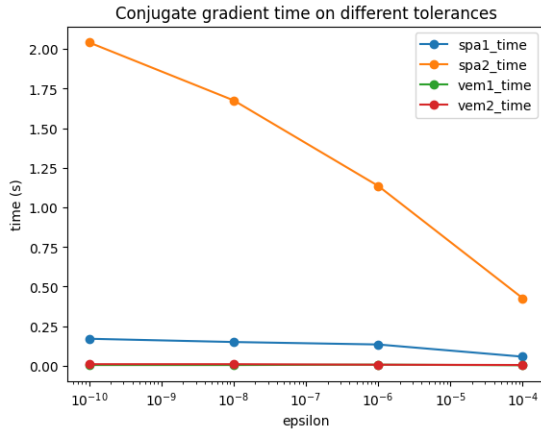
```

11     d = r - beta * d
12
13     return x, r, d

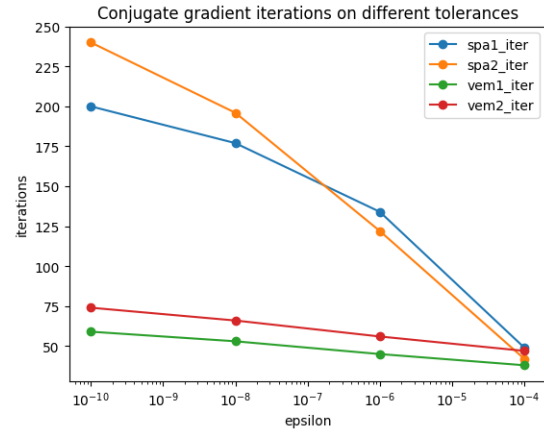
```

### 3.2.2 Risultati

I risultati possono essere verificati manualmente usando la classe implementata nel file *GradientSolver.py* oppure più facilmente mediante il notebook **benchmark.ipynb**. Questo metodo mantiene i vantaggi rispetto



(a) **Gradiente coniugato** rispetto al **tempo di esecuzione**



(b) **Gradiente coniugato** rispetto al **numero di iterazioni**

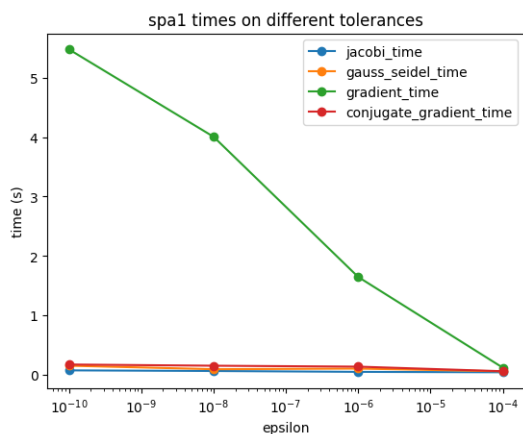
alle matrici a bande descritti al metodo del gradiente, è meglio notare invece come si migliora rispetto ad esso: in figura 7a si può notare come i tempi di esecuzione in generale siano sensibilmente più bassi, infatti si migliorano i tempi di ogni matrice, non solo di quelle a bande ed è proprio sulle matrici sparse che si nota la grande differenza. Il secondo confronto si può fare tra le figure 6b e 7b dove è facile notare come le iterazioni necessarie per la convergenza risultino essere molto minori questo perché come descritto prima introducendo il concetto di vettore ottimo rispetto ad una direzione non si ha l'effetto zig-zag di discesa del gradiente e quindi non si fanno iterazioni non necessarie. Nella tabella 4 vengono confrontati i risultati con le caratteristiche delle matrici (fissando la **tolleranza** a  $\epsilon = 1e^{-8}$ ):

Matrix	N	Non-zero entry	Sparsity index	Time (s)	Iterations
spa1	1000	182264	0.182264	0.150343	177
spa2	3000	1631738	0.181304	1.675488	196
vem1	1681	13385	0.004737	0.005208	53
vem2	2601	21225	0.003137	0.010405	66

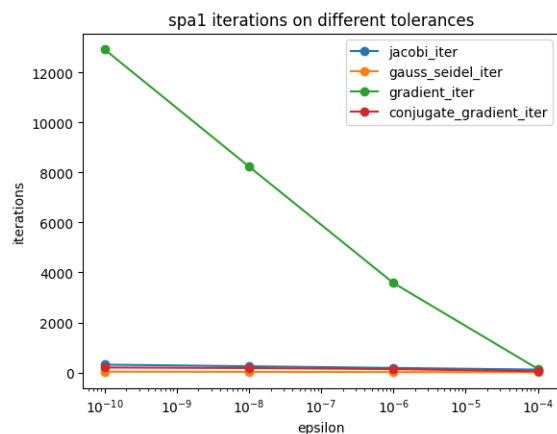
Table 4: Tabella delle esecuzioni del metodo di discesa del gradiente coniugato

## 4 Statistiche per matrice

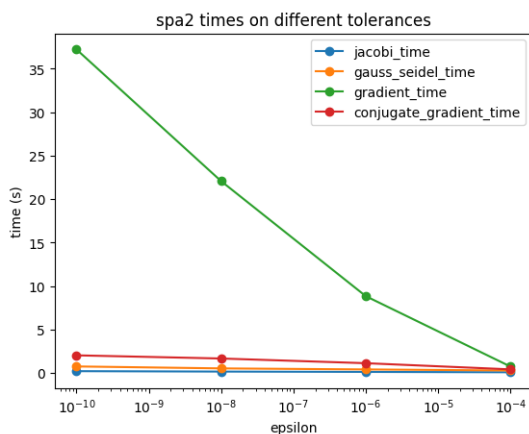
Di seguito riportiamo i grafici di ogni matrice rispetto alle iterazioni e al tempo di esecuzione per sottolineare come ogni metodo si è comportato con una stessa matrice.



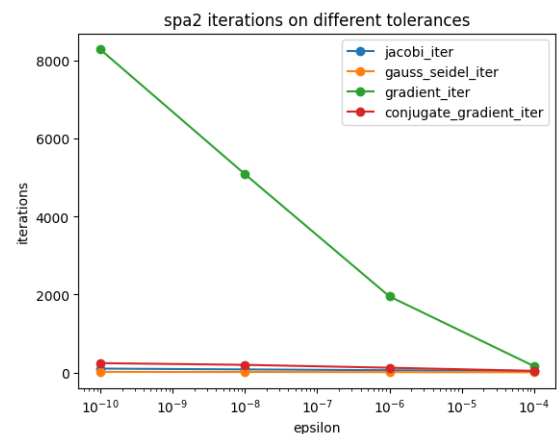
(a) Tempi di esecuzione su **spa1**



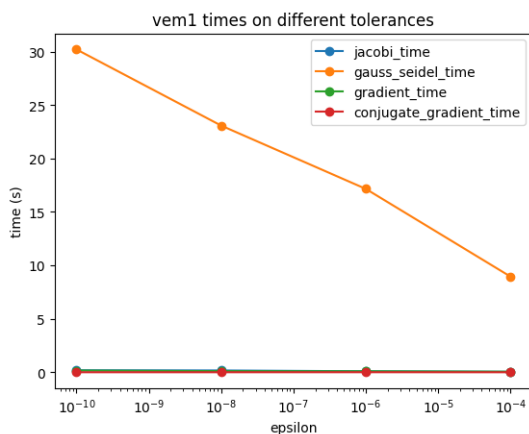
(b) Numero di iterazioni su **spa1**



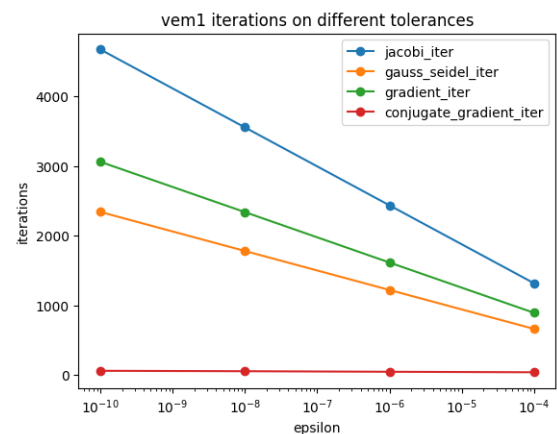
(a) Tempi di esecuzione su **spa2**



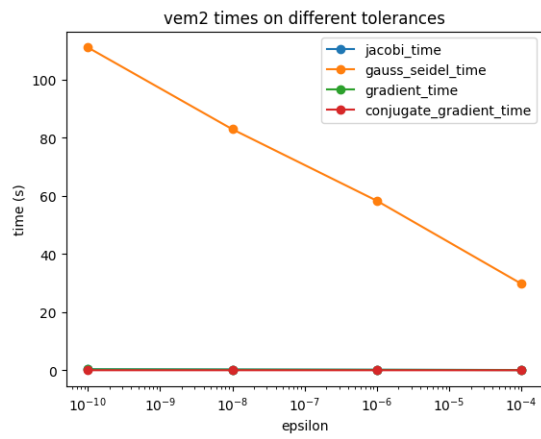
(b) Numero di iterazioni su **spa2**



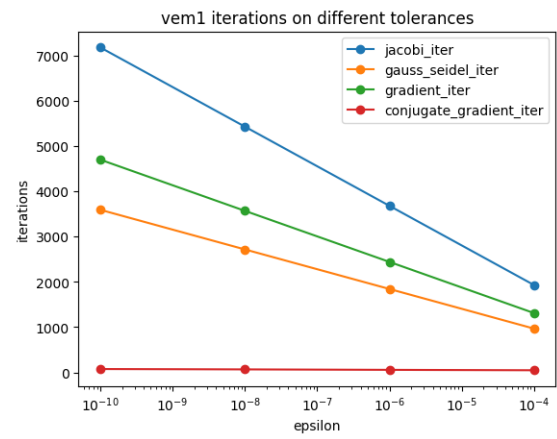
(a) Tempi di esecuzione su **vem1**



(b) Numero di iterazioni su **vem1**



(a) Tempi di esecuzione su **vem2**



(b) Numero di iterazioni su **vem2**

## 5 Conclusioni

In conclusione si vede come ogni metodo iterativo deve essere scelto in base alla matrice, infatti dei metodi esaminati non esiste un metodo univocamente migliore dell'altro ma è necessaria una breve analisi di ciò che si deve calcolare per scegliere al meglio il metodo di risoluzione. Tutti i file e i codici citati nella relazione sono disponibili su <https://github.com/sAndreotti/LinearSystemsResolution>.