Metodi del calcolo scientifico

Mini libreria per sistemi lineari

Volpato Mattia 866316 *
Andreotti Stefano 851596 †

Appello di Giugno 2024

Contents

1 Introduzione

1.1 Obiettivo

Lo scopo del progetto è la realizzazione di una mini libreria per la risoluzione di sistemi lineari, in particolare che implementi i metodi iterativi stazionari di Jacobi e di Gauss-Seider e i metodi iterativi non stazionari del Gradiente e del Gradiente coniugato.

1.2 Scelte implementative

Si è scelto di implementare la mini libreria in Python per 2 motivi principali: il primo è la popolarità del linguaggio, quindi la libreria può essere usata da molte persone mentre il secondo è la possibilità di creare un jupiter notebook, cioè un programma fatto da celle di testo e di codice che permettono una migliore interpretazione dei risultati e facilità di visualizzazione delle matrici.

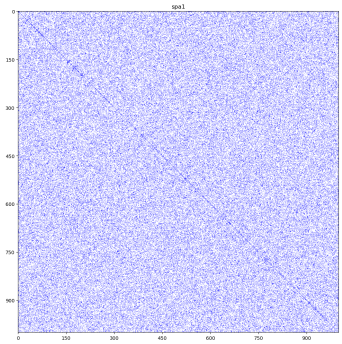
1.3 Descrizione matrici

Le matrici utilizzate per il progetto sono 4 diverse matrici sparse:

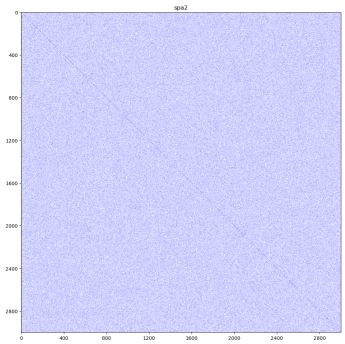
- Matrice spa1 1a
- Matrice spa2 1b
- Matrice vem1 2a
- Matrice vem2 2b

Nei grafici associati sono riportate le distribuzioni dei loro valori, si noti in particolare come spa1 e spa2 siano matrici sparse mentre vem1 e vem2 risultino essere matrici a bande. Di seguito si riportano alcune informazioni utili rispetto alle matrici:

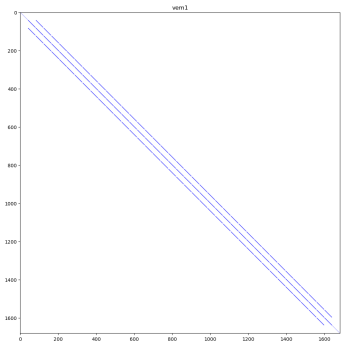
- Matrice spa1
  - Dimensioni: 1000x1000
  - Entrate non zero: 182264
  - Indice di sparsità: 18,23
- Matrice spa2
  - Dimensioni: 3000x3000
  - Entrate non zero: 161738
  - Indice di sparsità: 18,13
- Matrice vem1
  - Dimensioni: 1681x1681



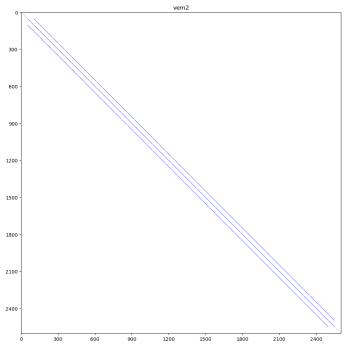
(a) Matrice spa1



(b) Matrice spa2



(a) Matrice vem1



(b) Matrice vem2

  - Entrate non zero: 13385
  - Indice di sparsità: 0,47

- Matrice vem2
  - Dimensioni: 2601x2601
  - Entrate non zero: 21225
  - Indice di sparsità: 0,31

1.4 Struttura

Per poter sfruttare i metodi iterativi serve calcolare un vettore dei termini noti b rispetto al quale calcolare il vettore soluzione x e un vettore soluzione iniziale x_0. La creazione del vettore b avviene tramite la risoluzione del sistema lineare $Ax = b$ dove il vettore x è un vettore di soli 1 (nella libreria è anche possibile scegliere di inizializzare il vettore con valori randomici). Successivamente il vettore x_0 è inizializzando con una lunghezza uguale alle righe della matrice A e con tutti i valori uguali a 0.

I metodi qui riportati condividono una struttura comune, si differenziano solo in base al metodo di calcolo della successiva iterata, infatti il criterio d'arresto usato per entrambi è il residuo scalato poichè quasi tutti i metodi lo calcolano durante l'aggiornamento e risulta computazionalmente più efficiente. Un altro criterio di arresto è il superamento di un numero massimo di iterazioni, durante i test è stato impostato a 20 000.
La struttura condivisa da ogni metodo è la seguente:

Algorithm 1 Metodi Iterativi

```
function SOLVER(Matrice A, vettore b)
    x = initialize_x_0
    k = 0
    residuo = b − Ax
    while check_termination = False do
        x, residuo = update_x(x, residuo)
        k+ = 1
        if k > max_iterations then
            iterations = k
            Nonconverge
    iteration = k
    Returnx
```

2 Metodi iterativi stazionari

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

$$A = P - N \tag{1}$$

Questi metodi sono chiamati stazionari poiché il calcolo dell'iterata successiva non dipende dall'iterazione. Nella libreria sono implementati i metodi di Jacobi e il metodo di Gauss-Seidel.

2.1 Metodo di Jacobi

Nel metodo di Jacobi si basa sull'uso della matrice P^{-1} e del residuo per calcolare l'iterata successiva, P è la matrice diagonale e in particolare la matrice P^{-1} indica la matrice diagonale di A nella quale ogni valore (sulla diagonale, gli altri sono 0) sono i reciproci. Computazionalmente la matrice P rimane la stessa per tutta l'esecuzione del metodo quindi la si calcola una sola volta.
La *k-esima* iterata è calcolata come:

$$x^{(k+1)} = x^{(k)} + P^{-1}(Ax^{(k)} - b) \tag{2}$$

Si nota che nel calcolo dell'iterata è presente il residuo, quindi non sarà necessario calcolarlo nuovamente per verificare la convergenza con il metodo del residuo scalato.

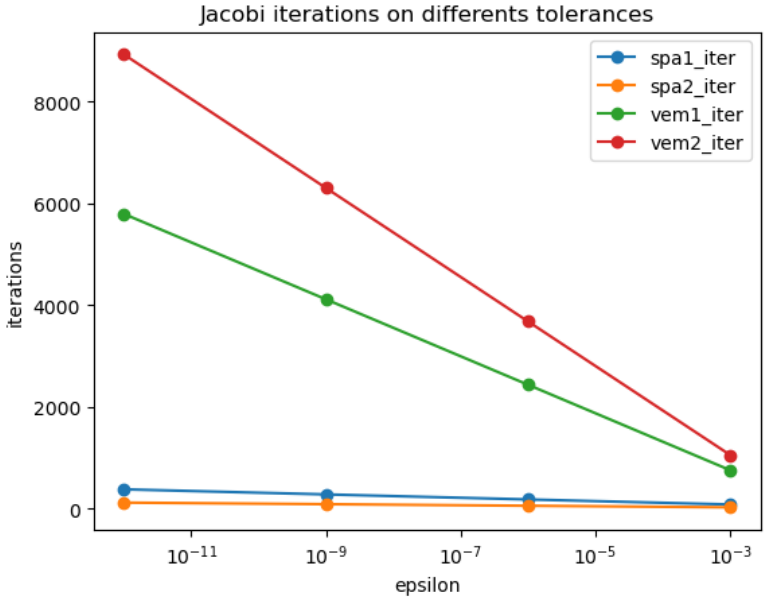


Figure 3: Risultati di Jacobi in base alle iterazioni

2.1.1 Risultati

Per ogni matrice descritta si è usato il metodo di Jacobi per la risoluzione. I risultati possono essere verificati manualmente usando la classe implementata nel file **JacobiSolver.py** oppure più facilmente mediante il notebook **linear_solver.py**.

In figura 4 possiamo notare sia la velocità di risoluzione, infatti la matrice con cui il metodo di Jacobi impiega più tempo a convergere, cioè vem2, è all'incirca mezzo secondo per una tolleranza di 10e-12 risultando quindi molto efficiente. Una seconda osservazione può invece essere fatta rispetto all'immagine 3 poiché in questa immagine si vede più chiaramente come la forma della matrice influenzi questo metodo: le matrici sparse spa1 e spa2 convergono dopo poche iterazioni, invece le matrici a bande vem1 e vem2 impiegano molte più iterazioni per convergere e ciò è dovuto alla stazionarietà del metodo (questo fenomeno si verificherà ancora nel metodo di Gauss-Seidel) infatti quando si analizzeranno i metodi non stazionari si noterà la grande differenza a livello di iterazioni che i metodi hanno gli uni con gli altri.
Di seguito 1 si riporta una tabella più precisa per quanto riguarda le esecuzioni:

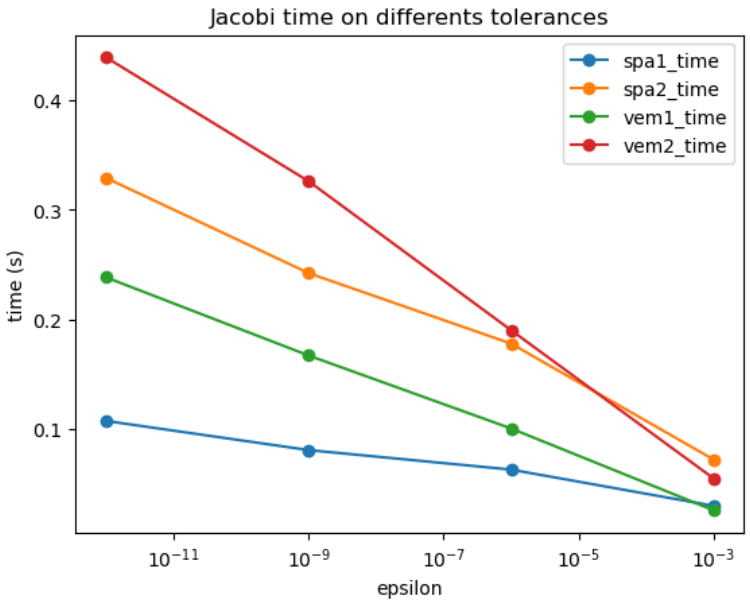


Figure 4: Risultati di Jacobi in base al tempo di esecuzione

Matrix	Dimension (N)	Non-zero	Sparsity index	Time (s)	Iteration
spa1	1000	182264	0.182264	0.080637	281
spa2	3000	1631738	0.181304	0.242089	89
vem1	1681	13385	0.004737	0.166903	4113
vem2	2601	21225	0.003137	0.325878	6300

Table 1: Tabella delle esecuzioni del metodo di Jacobi

Matrix	Dimension (N)	Non-zero	Sparsity index	Time (s)	Iteration
spa1	1000	182264	0.182264	0.131654	29
spa2	3000	1631738	0.181304	0.667321	15
vem1	1681	13385	0.004737	29.387556	2059
vem2	2601	21225	0.003137	106.721494	3153

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

2.2 Metodo di Gauss-Seider

Il metodo di Gauss-Seidel è una variante del metodo di Jacobi, questa volta 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} è possibile tramite la risoluzione del sistema lineare $Py = r^{(k)}$ che è possibile calcolare applicando la sostituzione in avanti dato che la matrice P è triangolare inferiore. Nella libreria sono presenti 2 modi di calcolare questa matrice. Il primo metodo è usare la funzione naive implementata nella libreria mentre il secondo è usare la soluzione tramite la libreria **Scipy** che risulta più efficiente, la scelta è effettuata mediante il file *Util.py* sotto la voce "$DEFAULT\_FORWARD\_SUBSTITUTION\_MODE$.
La *k-esima* iterata è calcolata come:

$$x^{(k+1)} = x^{(k)} + y \tag{3}$$

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

2.2.1 Risultati

Per ogni matrice descritta si è usato il metodo di GaussSeidel per il calcolo della soluzione approssimata. Anche in questo caso i risultati possono essere verificati manualmente usando la classe implementata nel file **GaussSeidelSolver.py** oppure più facilmente mediante il notebook **linear_solver.py** nell'apposita sezione.

In questo caso in riferimento alla figura 6 si può notare come le matrici sparse spa1 e spa2 risultino convergere in molto meno tempo rispetto alle matrici a bande vem1 e vem2. Le differenze a livello di iterazioni si fanno ancor più marcate che rispetto al metodo di Jacobi, infatti in figura 5 si vede come al variare della tolleranza le matrici sparse sembrano quasi non variare mentre le matrici a bande aumentano a dismisura il numero di iterazioni richieste.
Di seguito 2 si riporta una tabella più precisa per quanto riguarda le esecuzioni:

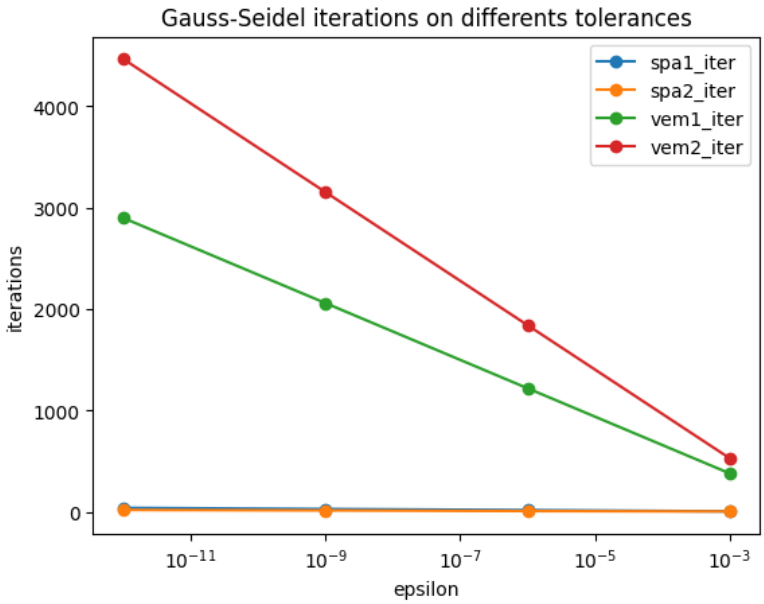


Figure 5: Risultati di Gauss-Seidel in base alle iterazioni

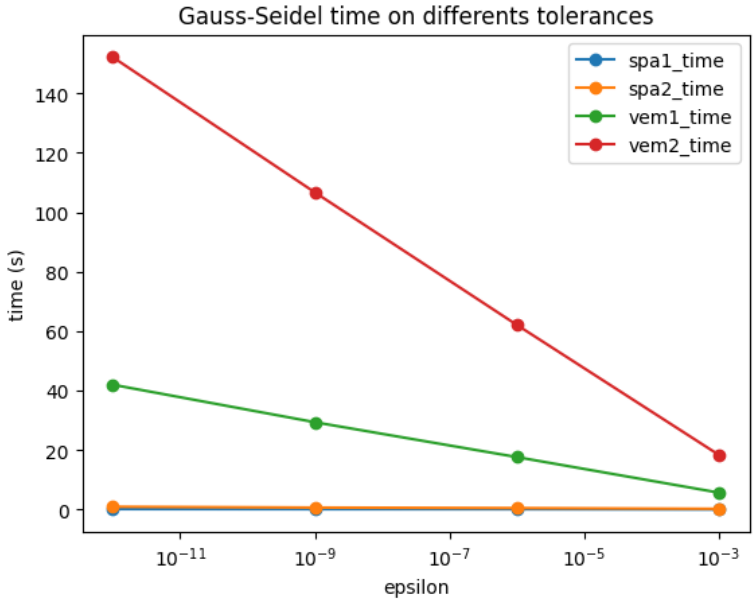


Figure 6: Risultati di Gauss-Seidel in base al tempo di esecuzione

3 Metodi iterativi non stazionari

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

$$x^{(k+1)} = x^(k) + \alpha_k P^{-1} r^(k) \tag{4}$$

La differenza si vede dal coefficiente α che in questo caso dipende dall'iterazione precedente anziché essere stazionario

3.1 Metodo di discesa del gradiente

Il metodo del gradiente interpreta una matrice A simmetrica e definita positiva e una funzione ι come una parabola della quale si vuole trovare il punto di minimo, il quale corrisponde alla soluzione del sistema lineare. Segue che la nuova iterazione può essere calcolata come

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} \tag{5}$$

Il calcolo di α risulta essere quello più oneroso:

$$\alpha_k = ((r^{(k)})^t r^{(k)})/((r^{(k)})^t A r^{(k)}) \tag{6}$$

3.1.1 Risultati

Per ogni matrice descritta si è usato il metodo del Gradiente per il calcolo della soluzione approssimata. I risultati possono essere verificati manualmente usando la classe implementata nel file **GradientSolver.py** oppure più facilmente mediante il notebook **linear_solver.py** nell'apposita sezione.

In figura 8 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 7 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. Di seguito 3 si riporta una tabella precisa rispetto ai dati delle esecuzioni:

3.2 Metodo di discesa del gradiente coniugato

Il metodo del gradiente coniugato è un miglioramento rispetto al metodo del gradiente sopracitato poiché pone rimedio al fenomeno dello zig-zag, cioè quando

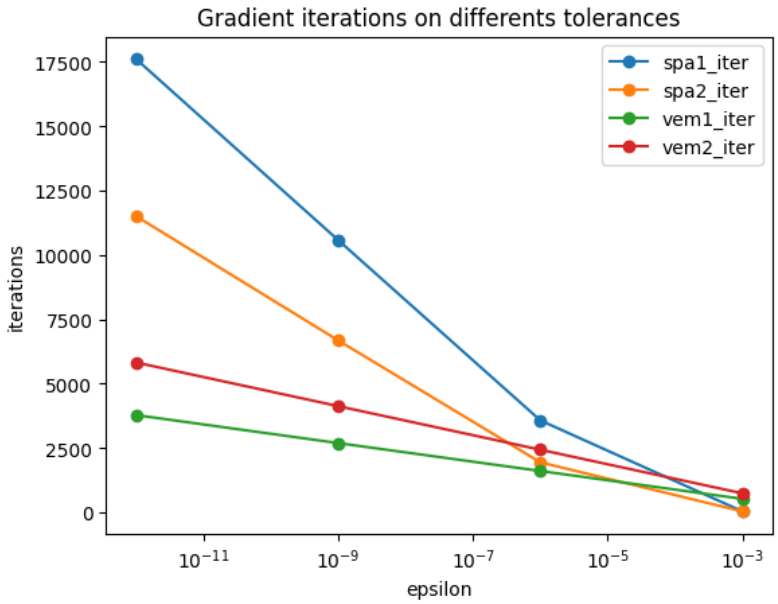


Figure 7: Risultati del Gradiente in base alle iterazioni

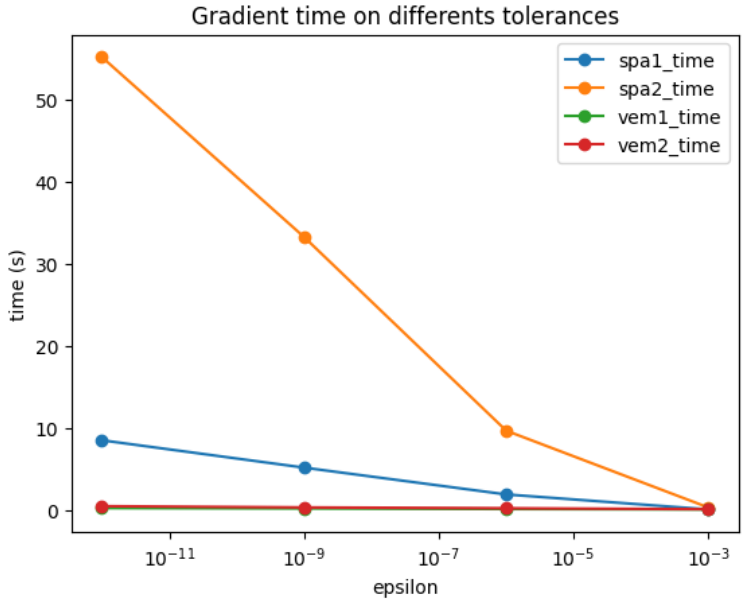


Figure 8: Risultati del Gradiente in base al tempo di esecuzione

Matrix	Dimension (N)	Non-zero	Sparsity index	Time (s)	Iteration
spa1	1000	182264	0.182264	5.132789	10576
spa2	3000	1631738	0.181304	33.244131	6682
vem1	1681	13385	0.004737	0.149694	2697
vem2	2601	21225	0.003137	0.283141	4131

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

$\lambda_{min} << \lambda_{max}$. Per migliorare il metodo si definisce un vettore ottimale rispetto ad una direzione come:

$$d * r^{(k)} = 0 \tag{7}$$

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

$$\alpha_k = ((d^{(k)})^t r^{(k)})/((d^{(k)})^t A d^{(k)}) \tag{8}$$

Mentre il calcolo dell'iterata successiva rimane

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} \tag{9}$$

3.2.1 Risultati

L'ultimo metodo implementato è quello della discesa del gradiente coniugato e come per gli altri si riportano i grafici inerenti al calcolo della soluzione approssimata. I risultati possono essere verificati manualmente usando la classe implementata nel file **ConjugateGradientSolver.py** oppure più facilmente mediante il notebook **linear_solver.py** nell'apposita sezione.

Questo metodo mantiene i vantaggi rispetto alle matrici a bande descritti al metodo del gradiente, è meglio notare invece come si migliora rispetto ad esso: in figura 10 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 7 e 9 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.
Di seguito 4 si riporta la tabella delle esecuzioni:

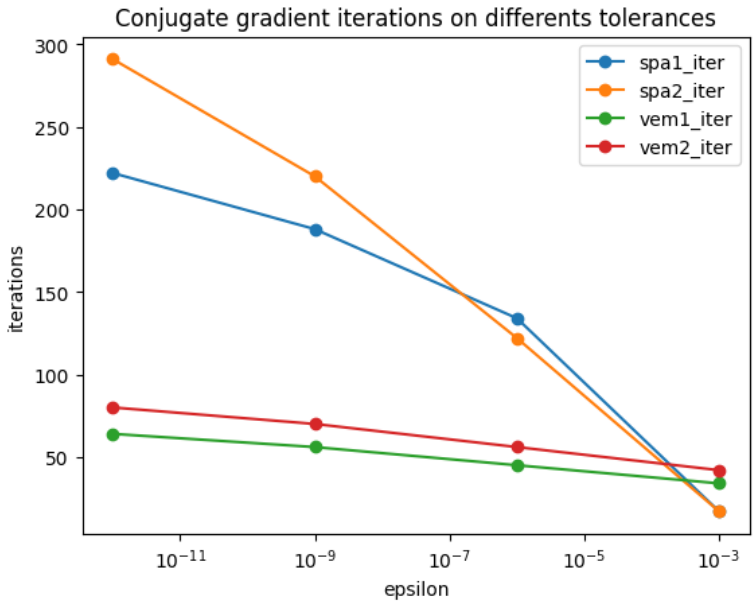


Figure 9: Risultati del Gradiente coniugato in base alle iterazioni

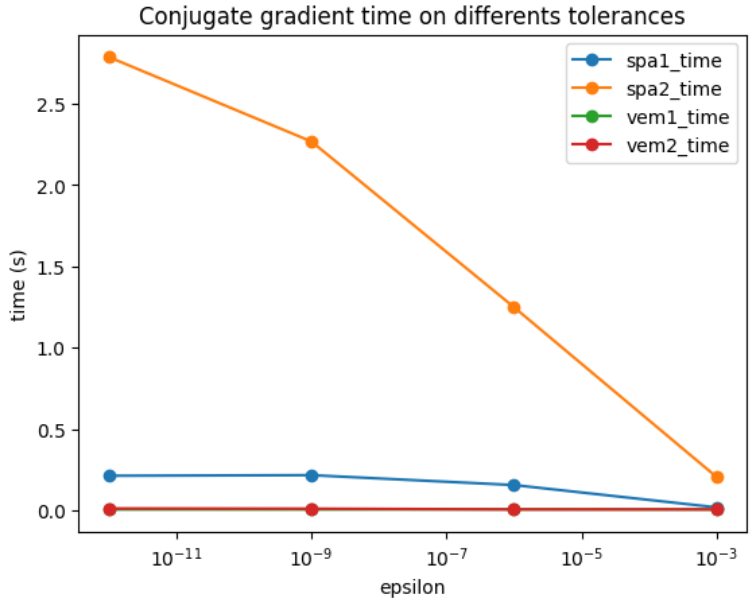


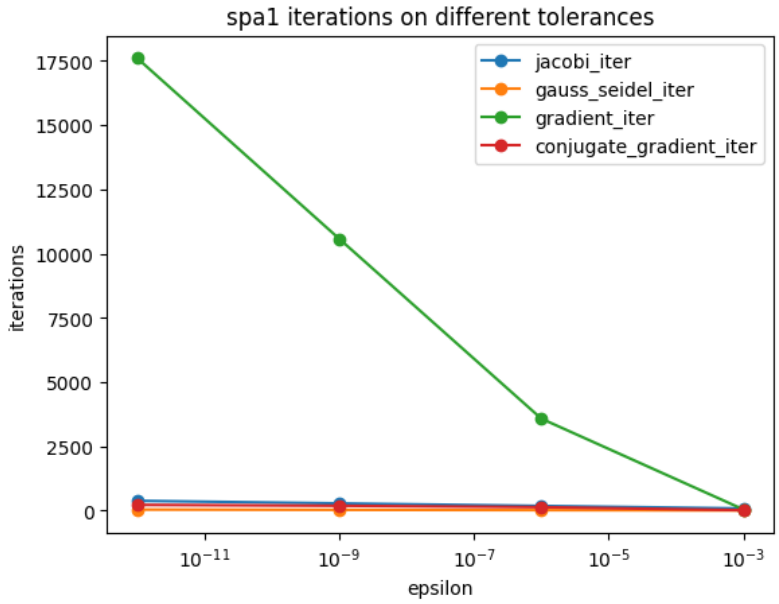
Figure 10: Risultati del Gradiente coniugato in base al tempo di esecuzione

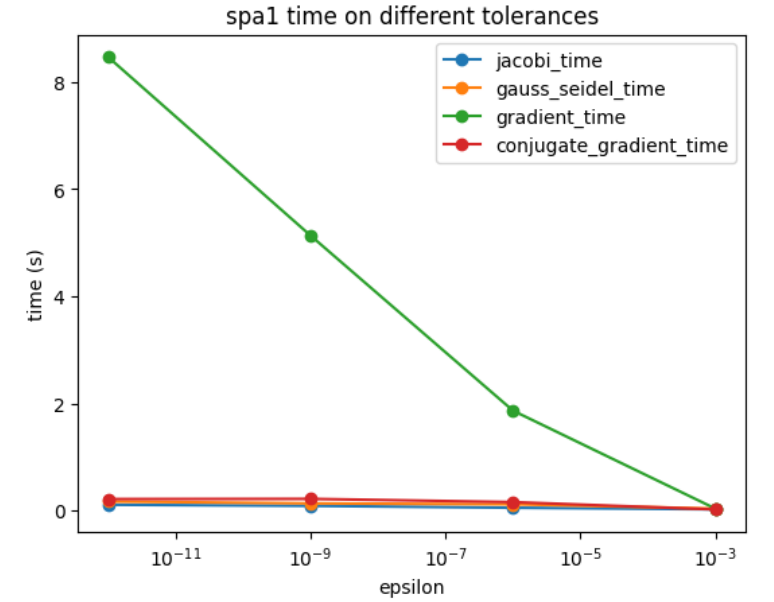
Matrix	Dimension (N)	Non-zero	Sparsity index	Time (s)	Iteration
spa1	1000	182264	0.182264	0.216183	188
spa2	3000	1631738	0.181304	2.267832	220
vem1	1681	13385	0.004737	0.005301	56
vem2	2601	21225	0.003137	0.010181	70

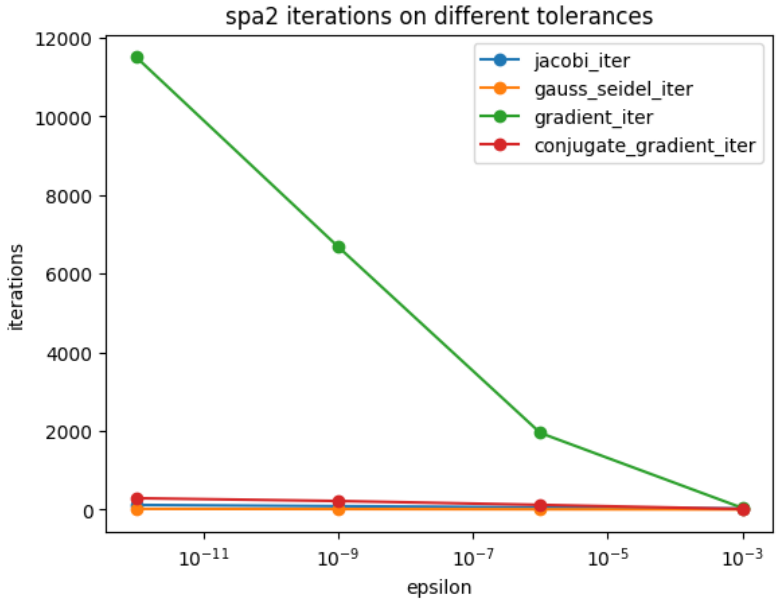
Table 4: Tabella delle esecuzioni del metodo di discesa del gradiente coniugato

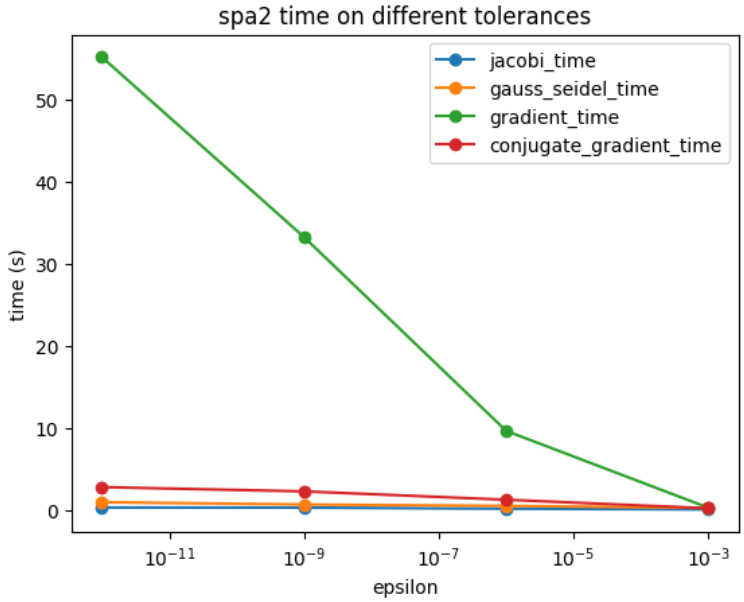
4 Grafici delle matrici

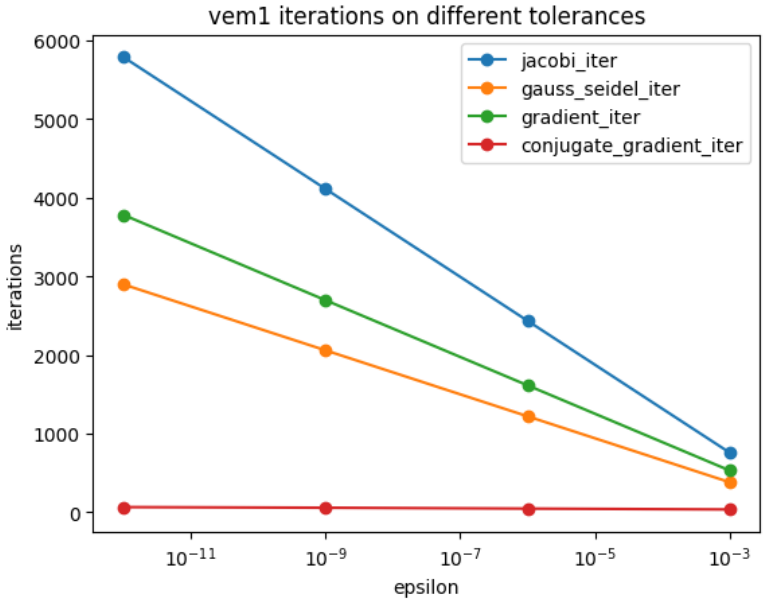
Di seguito riportiamo i grafici di ogni matrice rispetto alle iterazioni e al tempo di esecuzione per sottolineare come ogni metodo si è comportato con una stessa matrice.

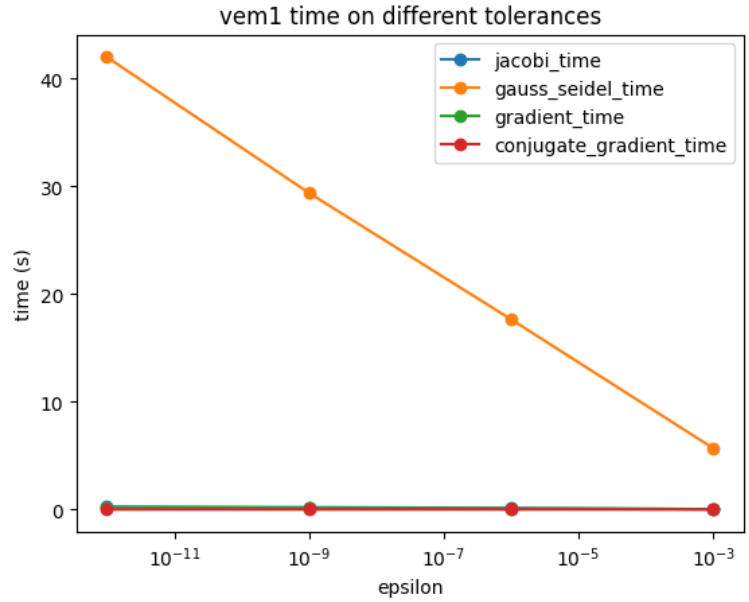


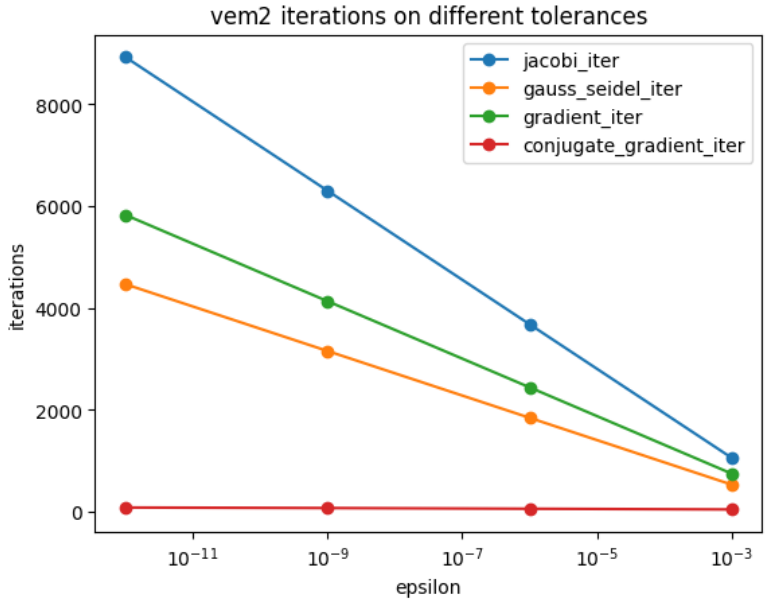

spa1 time on different tolerances
jacobi_time
gauss_seidel_time
gradient_time
conjugate_gradient_time
time (s)
0
2
4
6
8
10^{-11}
10^{-9}
10^{-7}
10^{-5}
10^{-3}
epsilon

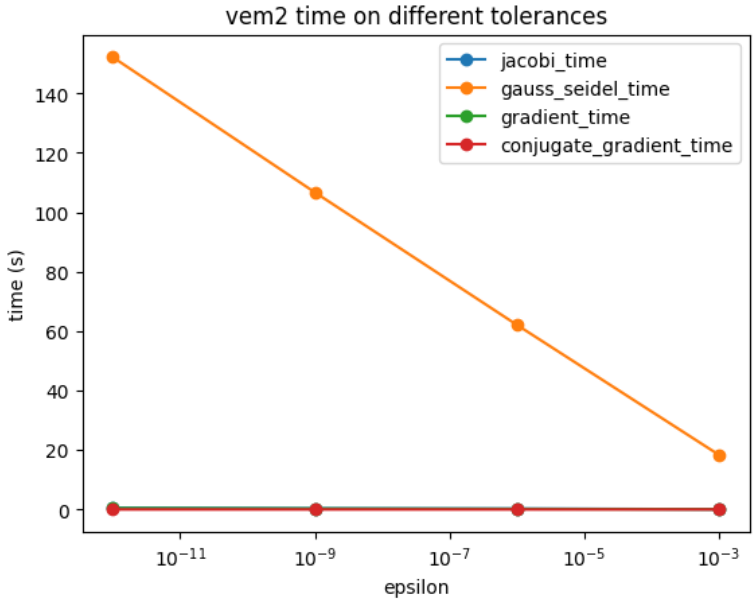

spa2 iterations on different tolerances
jacobi_iter
gauss_seidel_iter
gradient_iter
conjugate_gradient_iter
iterations
0
2000
4000
6000
8000
10000
12000
10^{-11}
10^{-9}
10^{-7}
10^{-5}
10^{-3}
epsilon


spa2 time on different tolerances
jacobi_time
gauss_seidel_time
gradient_time
conjugate_gradient_time
time (s)
0
10
20
30
40
50
10^{-11}
10^{-9}
10^{-7}
10^{-5}
10^{-3}
epsilon


vem1 iterations on different tolerances
jacobi_iter
gauss_seidel_iter
gradient_iter
conjugate_gradient_iter
iterations
0
1000
2000
3000
4000
5000
6000
10^{-11}
10^{-9}
10^{-7}
10^{-5}
10^{-3}
epsilon


vem1 time on different tolerances
jacobi_time
gauss_seidel_time
gradient_time
conjugate_gradient_time
time (s)
epsilon


vem2 iterations on different tolerances
jacobi_iter
gauss_seidel_iter
gradient_iter
conjugate_gradient_iter
iterations
epsilon



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`.

Andreotti Stefano e Volpato Mattia