Università degli Studi di Milano Bicocca

Dipartimento di Informatica

Corso di Laurea Magistrale

Primo progetto di Metodi per il Calcolo Scientifico

ANNO ACCADEMICO 2024/2025

Farioli Alessio - 879217

Isceri Alessandro - 879309

Indice

1. Introduzione

1.1 Obiettivi del progetto

Il progetto realizzato ha come obiettivo la creazione di una libreria in linguaggio Python per la risoluzione di sistemi lineari mediante l'utilizzo dei seguenti metodi iterativi:

- **Metodo di Jacobi**.
- **Metodo di Gauss-Seidel**.
- **Metodo del Gradiente**.
- **Metodo del Gradiente Coniugato**.

1.2 Considerazioni Implementative dei Metodi Implementati

Tutti i metodi implementati si occupano di verificare che la matrice presa in input sia una matrice Simmetrica Positivamente Definita (**SPD**).

1.2.1 Jacobi

Il metodo di Jacobi converge se la matrice è a **dominanza diagonale**, dunque se tale condizione non venisse rispettata l'algoritmo potrebbe terminare al raggiungimento del numero massimo di iterazioni restituendo un risultato profondamente errato.

Ci si aspetta che Jacobi impieghi un numero abbastanza alto di iterazioni, ma la singola iterazione dovrebbe impiegare una quantità ridotta di tempo.

1.2.2 Gauss-Seidel

Similmente al metodo di Jacobi, anche il metodo di Gauss-Seidel converge se la matrice è a dominanza diagonale. Ciò vuol dire che anche in questo caso se la matrice non avesse tale proprietà il

metodo potrebbe divergere. Gauss-Seidel però, garantisce la convergenza anche nel caso in cui la matrice sia SPD. Siccome il codice implementato controlla sempre che la matrice in input sia SPD, il metodo convergerà sempre.

Ci si aspetta che Gauss-Seidel impieghi un numero più basso di iterazioni rispetto a Jacobi, ma la singola iterazione dovrebbe impiegare una quantità di tempo maggiore.

1.2.3 Gradiente

Il metodo del Gradiente converge sicuramente se la matrice su cui opera è SPD. Grazie alle proprietà del codice scritto quindi il metodo convergerà sempre.

Il metodo del Gradiente potrebbe impiegare un numero di iterazioni abbastanza alto, perchè, al contrario della sua variante (Gradiente Coniugato) tende ad oscillare molto prima di raggiungere la soluzione ottima.

In particolare, il metodo del Gradiente impiegherà molte iterazioni nel caso in cui gli autovalori della matrice siano molto diversi tra di loro.

1.2.4 Gradiente Coniugato

Anche il metodo del Gradiente Coniugato converge sicuramente se la matrice su cui opera è SPD. Grazie alle proprietà del codice scritto quindi il metodo convergerà sempre.

Il metodo del Gradiente Coniugato è un'evoluzione del metodo del Gradiente, nella quale si pone più attenzione alla scelta della direzione e del modulo dello spostamento lungo essa.

In particolare, dopo aver effettuato uno spostamento lungo una direzione, essa non verrà più modificata, in modo che si possa raggiungere convergenza in al più n iterazioni, con n = numero di righe della matrice. A causa di ciò ci si aspetta che il metodo del Gradiente Coniugato compi un numero estremamente ridotto di iterazioni per raggiungere convergenza.

1.3 Architettura della libreria

La libreria realizzata è composta principalmente da due cartelle:

- **src**: che contiene i files:
  - *solver.py*
  - *jacobi_solver.py*
  - *gauss_seidel_solver.py*
  - *gradient_solver.py*
  - *conjugated_gradient_solver.py*
  - *main.py*
  - *utils.py*
- **exe**: che contiene i files:
  - *exe_dense.py*
  - *exe_sparse.py*

In particolare, il file *solver.py* è una classe astratta, ovvero una classe non istanziabile, che presenta alcuni metodi ”vuoti” (*solve* e *solve_sparse*) ed altri metodi implementati.
I metodi implementati sono:

- Il costruttore: prende in input una matrice **A**, un vettore dei termini noti **b**, un parametro **tol** di tolleranza a un numero massimo di iterazioni **t** e istanzia l'oggetto. Inoltre controlla che la matrice A sia valida (SPD, quadrata) e che il numero di righe di A sia pari alla lunghezza del vettore b.
- Lo scheletro del metodo solve: per evitare di avere del codice duplicato è stato utilizzato il pattern architetturale **Template Method**. Questo pattern permette di definire nella classe astratta un metodo (in questo caso solve) che è parzialmente definito. Questo metodo, quando viene invocato, controlla che la matrice sia valida, se non lo è, ritorna un vettore soluzione

composto da zeri e un numero di iterazioni pari a -1 (segno che il metodo non può essere eseguito). Se invece l'oggetto è valido, controlla se la matrice è sparsa: se lo è, invoca il metodo *solve_sparse,* altrimenti invoca il metodo *solve_dense*. I metodi *solve_sparse* e *solve_dense* sono astratti: spetta alle classi che estendono la classe astratta implementarli affinché il codice funzioni correttamente.

Questo pattern architetturale permette di evitare di avere codice duplicato comune a tutti i metodi; inoltre permette di rendere la libreria facilmente estendibile: qualora si volesse aggiungere un nuovo metodo, basterà estendere la classe *solver.py* e implementare i metodi *solve_dense* e *solve_sparse* della nuova classe.

Il file *main.py* definisce una funziona **main** che permette all'utente di eseguire tutti i metodi implementati su una data matrice fornendo come input:

- la matrice stessa (**A**)
- il vettore soluzione esatto (**x**)
- il membro destro dell'equazione (**b**)
- il valore di tolleranza (**tol**)

Il metodo restituisce in output:

- **relative_errors**: un vettore contenente gli errori relativi, calcolati come:

$$\text{errore relativo} = \frac{||x - \tilde{x}||}{||x||}$$

- **elapsed_times**: un vettore contenente i tempi di esecuzione di ciascun metodo.
- **it_numbers**: un vettore contenente il numero di iterazioni effettuate da ciasun metodo.

In tutti i vettori restituiti dal metodo `main`, l'ordine degli elementi è il seguente:

- Posizione **0**: Jacobi
- Posizione **1**: Gauss-Seidel
- Posizione **2**: Gradiente

- Posizione **3**: Gradiente Coniugato

Infine, il file *utils.py* rende disponibili altre funzionalità implementate come il controllo che una matrice sia SPD, il risolutore di sistemi lineari triangolari inferiori (*solve_lower*) e la funzione per calcolare l'errore relativo della soluzione.

Per quanto riguarda la classe *solve_lower*, è emerso che l'utilizzo di matrici in formato sparso comportava un notevole rallentamento; di conseguenza, dopo alcune valutazioni, si è deciso di mantenerla esclusivamente in versione densa.

In pratica, il risolutore *gauss_seidel*, anche se sta lavorando su matrici in formato sparso, addensa la matrice ed esegue il metodo *solve* di *solve_lower* che lavora su matrici in formato denso.

Nonostante possa sembrare inefficiente, siccome la matrice su cui viene invocato *solve_lower* da Gauss-Seidel è sempre la matrice B = D - L, essa viene addensata una sola volta, e l'oggetto *triangular_solver* di classe *solve_lower* viene istanziato una sola volta, all'inizio del metodo di Gauss-Seidel, permettendo di ottenere delle ottime performance. In pratica di volta in volta viene solamente aggiornato il vettore dei termini noti b passato all'oggetto *solve_lower*.

Sono inoltre presenti due file eseguibili: uno lavora utilizzando le matrici in formato sparso, ed uno opera con matrici in formato denso. In questi file vengono eseguiti tutti e quattro i metodi con valori di tolleranza nell'insieme $\{10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}\}$ su quattro matrici note, ovvero *'spa1'*, *'spa2'*, *'vem1'*, e *'vem2'*, lette da dei file **mtx**. Si noti che per l'esecuzione su matrici dense le quattro matrici vengono addensate.

L'eseguibile, al termine dell'esecuzione, stampa a video gli errori relativi, i tempi di esecuzione e il numero di iterazioni sotto forma di vettori tridimensionali.

In particolare, le dimensioni dell'array indicano:

- **Prima dimensione** — identifica la matrice:
  - 0: `spa1`
  - 1: `spa2`
  - 2: `vem1`
  - 3: `vem2`
- **Seconda dimensione** — rappresenta la tolleranza:
  - 0: 10^{-4}
  - 1: 10^{-6}
  - 2: 10^{-8}
  - 3: 10^{-10}
- **Terza dimensione** — specifica il metodo utilizzato:
  - 0: Jacobi
  - 1: Gauss-Seidel
  - 2: Gradiente
  - 3: Gradiente Coniugato

Per produrre i grafici sono stati utilizzati due **notebook** python: il primo notebook esegue i quattro metodi su tutte le matrici (sia in formato denso che sparso) utilizzando le quattro tolleranze ed esporta i risultati in un file **JSON**.
Un secondo notebook ha il compito di leggere i risultati dal file JSON e produrre i grafici riportati nel capitolo 3.

1.4 Strumenti utilizzati

Per la creazione della libreria è stato utilizzato il linguaggio **Python** e le librerie open-source **Numpy**, per l'implementazione della maggior parte dei metodi, in particolare quelli che operano su matrici in formato denso, ed il suo superset **Scipy**, utilizzato per la gestione delle matrici sparse. Inoltre sono state utilizzate le librerie **Pandas** e **Matplotlib** per la creazione di grafici atti alla visualizzazione di errore normalizzato, numero di iterazioni e tempo impiegato dai metodi iterativi, per permettere una semplice ed intuitiva analisi dell'efficienza dei metodi.

Il sistema operativo sul quale è stato eseguito il codice è **Windows 11 Home**. Il computer utilizzato per l'esecuzione è un Modello **Acer Aspire A715-75G**, con processore **Intel Core i7-10750H** CPU @ 2.60GHz, 2592 Mhz.

2. Analisi delle Matrici

Prima di riportare i risultati ottenuti con l'esecuzione del codice, viene riportata un'analisi delle matrici su cui sono stati eseguiti i risolutori implementati.

2.1 SPY

Siccome le matrici sono salvate in formato sparso, inizialmente vengono riportati gli **Sparsity Pattern Plot** delle quattro matrici, per capire effettivamente il livello di densità delle matrici a prima vista.

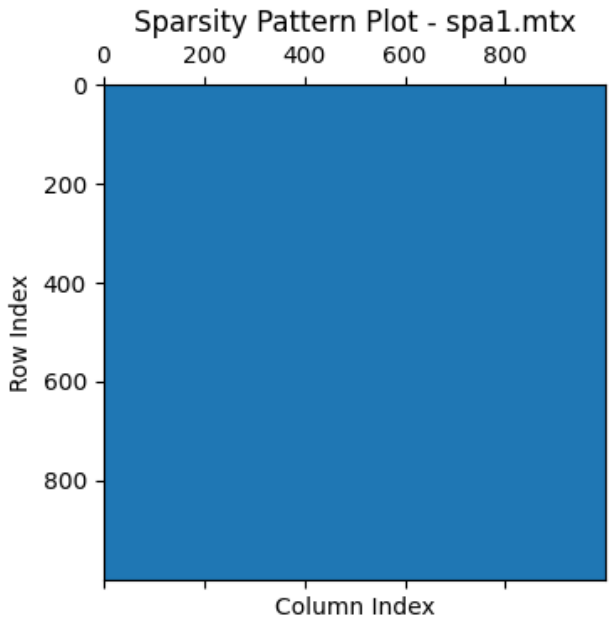

Figura 2.1: SPY di spa1.

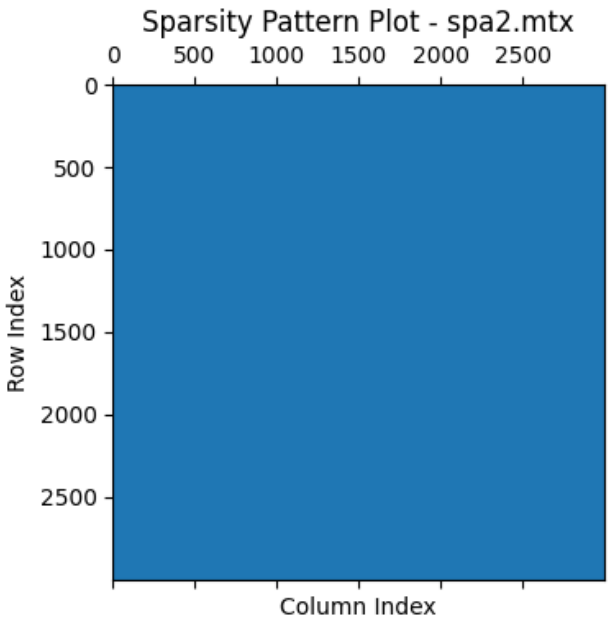

Figura 2.2: SPY di spa2.

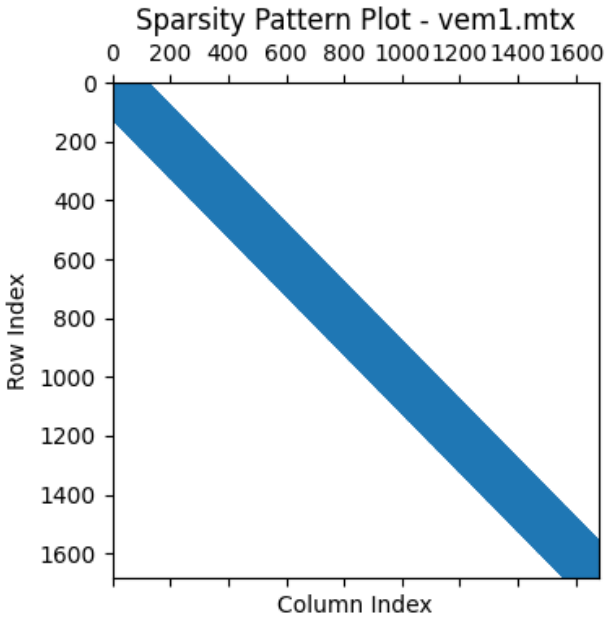

Figura 2.3: SPY di vem1.

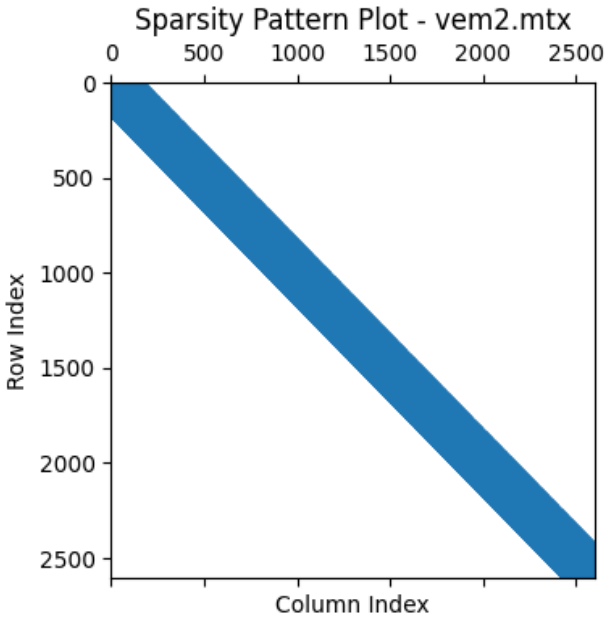

Figura 2.4: SPY di vem2.

Dai grafici 2.3 e 2.4 si nota che le matrici vem1 e vem2 hanno entry non nulle sulle posizioni vicine alla diagonale, mentre le prime matrici sembrano essere completamente dense.

Per analizzare ulteriormente il livello di densità delle matrici, sono state calcolati:

- Il numero totale di entries.
- Il numero di entries diverse da 0.
- Il livello di densità della matrice, calcolato come:

$$\text{Densità} = \frac{\text{Numero di elementi non nulli}}{\text{Numero totale di elementi}} = \frac{\text{Numero di elementi non nulli}}{n \times n}$$

- Il numero di Bytes occupati dalle matrici in formato denso.
- Il numero di Bytes occupati dalle matrici in formato sparso.

Di seguito vengono riportati i risultati:

	spa1	spa2	vem1	vem2
Number of Total Entries	1.000.000	9.000.000	2.825.761	6.765.201
Number of Non-Zeros entries	182.264	1.631.738	13.385	21.225
Density	0.1823	0.1813	0.0047	0.0031
Occupied Memory Dense (Bytes)	8.000.000	72.000.000	22.606.088	54.121.608
Occupied Memory Sparse (Bytes)	2.918.944	26.132.768	214.160	339.600

Da questa tabella si nota sia il risparmio di memoria ottenuto col formato sparso, in particolare per quanto riguarda le matrici vem1 e vem2, ma, soprattutto, il fatto che le prime due matrici non sono dense come poteva sembrare in prima battuta dalle figure 2.1 e 2.2, ma anzi hanno una quantità di entries ridotta. Questo succede perchè il Sparsity Pattern Plot è troppo piccolo per mostrare bene cosa accade in milioni di entries e dunque questa ulteriore analisi è stata rivelatrice di queste proprietà.

2.1.1 Analisi degli Autovalori

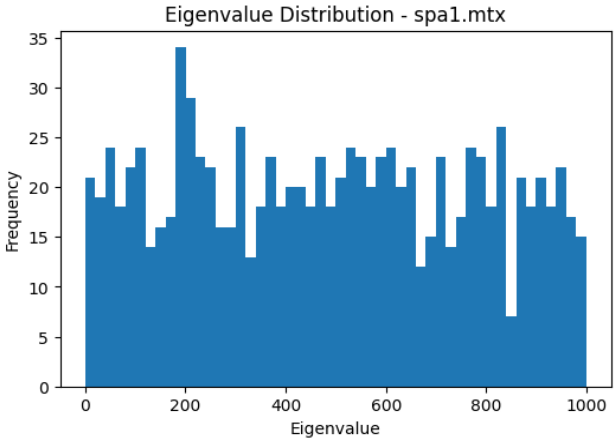

Figura 2.5: Distribuzione Autovalori spa1.

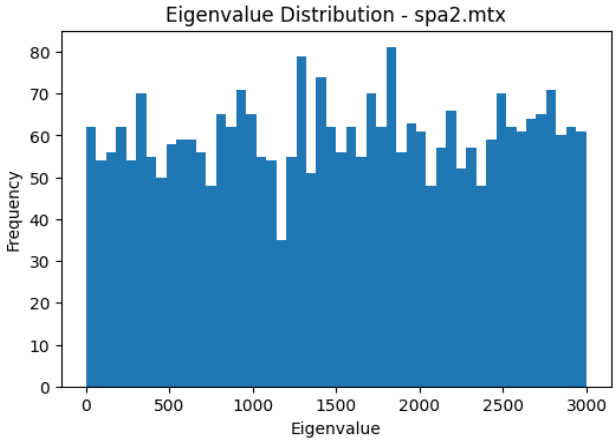

Figura 2.6: Distribuzione Autovalori spa2.

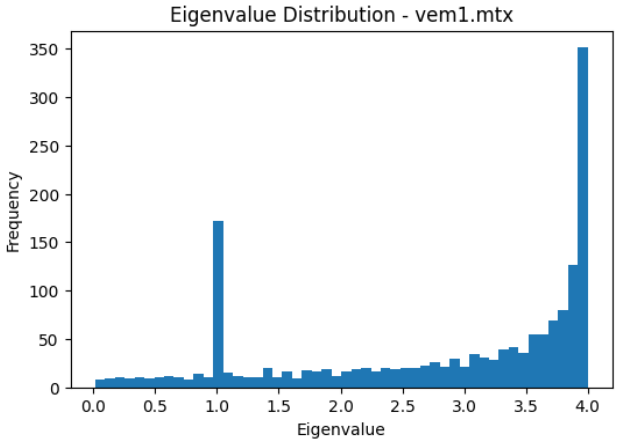

Figura 2.7: Distribuzione Autovalori vem1.

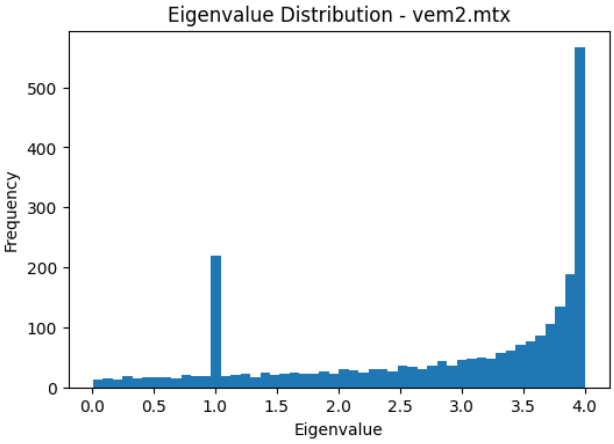

Figura 2.8: Distribuzione Autovalori vem2.

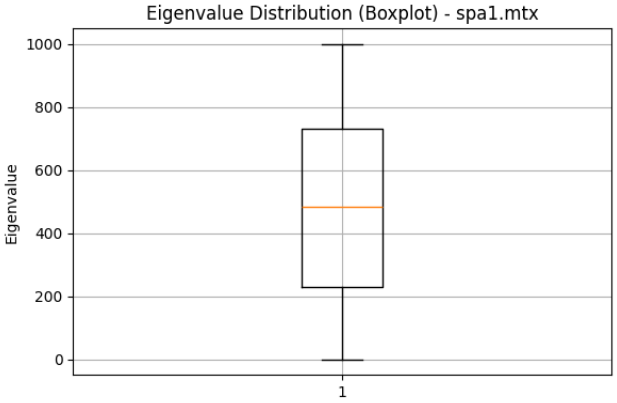

Figura 2.9: BoxPlot degli Autovalori di spa1.

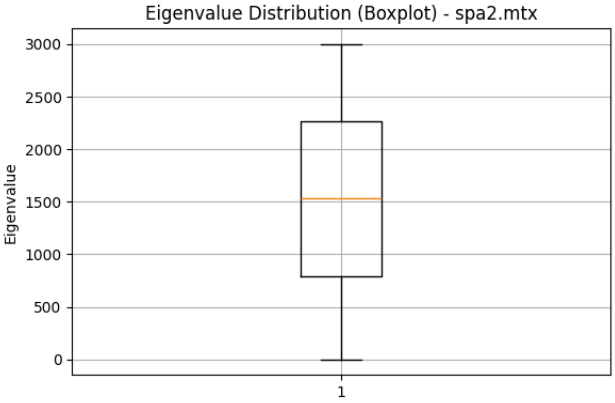

Figura 2.10: BoxPlot degli Autovalori di spa2.

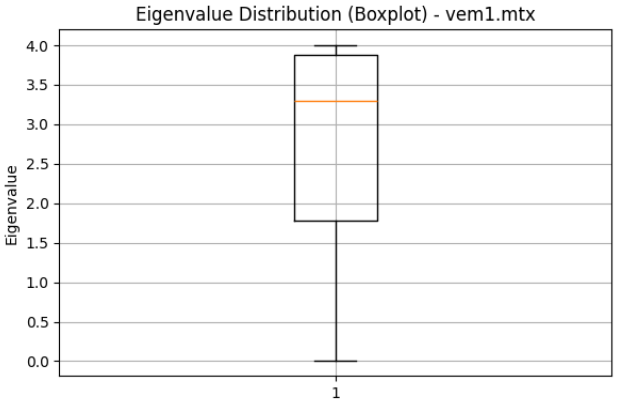

Figura 2.11: BoxPlot degli Autovalori di vem1.

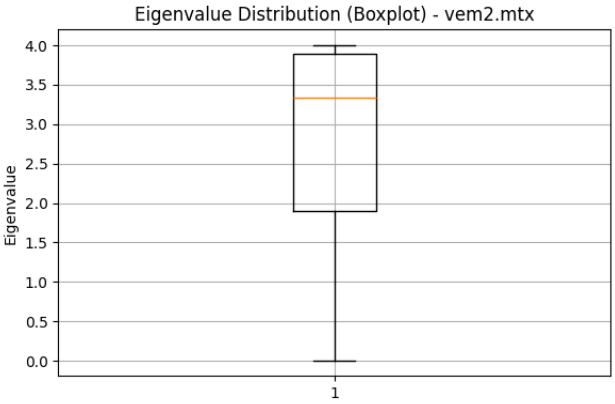

Figura 2.12: BoxPlot degli Autovalori di vem2.

Come si può notare dalle immagini che rappresentano la distribuzione degli autovalori delle quattro matrici, le matrici spa1 e spa2 presentano una distribuzione molto ampia di autovalori, con un rapporto elevato tra l'autovalore massimo e quello minimo. Questo elevato *numero di condizionamento* suggerisce che il metodo del **Gradiente** potrebbe incontrare difficoltà nella convergenza.

Al contrario, le matrici vem1 e vem2 presentano una distribuzione degli autovalori più contenuta e un rapporto tra autovalore massimo e minimo significativamente inferiore. Ci si aspetta, quindi, che il metodo del **Gradiente** riesca a convergere con maggiore facilità in questi casi.

2.1.2 Dominanza Diagonale

Abbiamo controllato se le matrici fossero a dominanza diagonale per assicurare la convergenza di Jacobi, ma nessuna delle quattro matrici prese in analisi è a dominanza diagonale, quindi la convergenza di Jacobi non è garantita.

3. Risultati e Grafici

In questo capitolo vengono riportati e commentati i grafici che sono stati prodotti dalle esecuzioni dei quattro metodi di risoluzione sulle matrici **spa1**, **spa2**, **vem1** e **vem2**.

3.1 Errori

Come prevedibile, gli errori relativi prodotti dai metodi che operano su matrici memorizzate in formato sparso sono uguali a quelli prodotti con i metodi che operano su matrici memorizzate in formato denso.
Di seguito vengono riportati i grafici prodotti.

3.1.1 Errori al variare della tolleranza

I seguenti grafici rappresentano l'**errore normalizzato** della soluzione restituita dai quattro metodi sulle quattro matrici al variare della tolleranza.

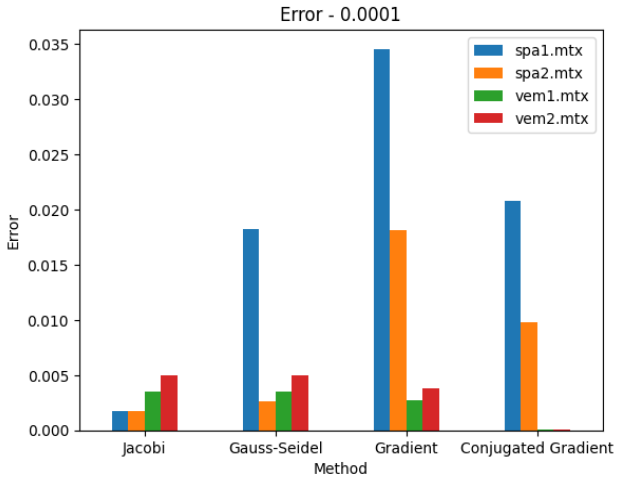

Figura 3.1: Errori con tolleranza di 10^{-4}.

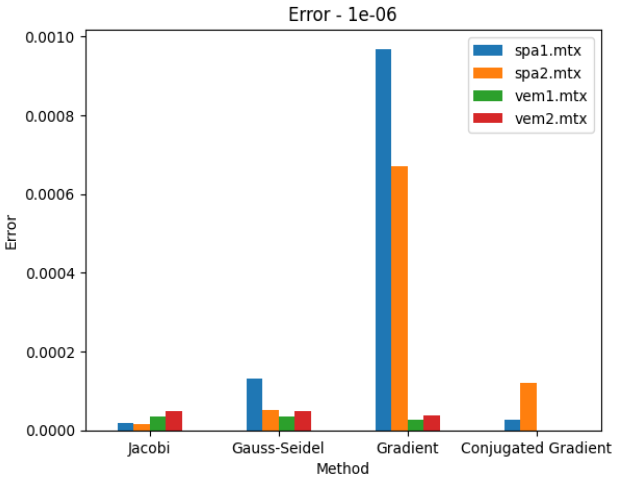

Figura 3.2: Errori con tolleranza di 10^{-6}.

Come si può notare dalle figure, il metodo del **Gradiente** presenta l'errore più alto, soprattutto al diminuire del valore della tolleranza. Il metodo del **Gradiente** e del **Gradiente Coniugato** sembrano avere più difficoltà sulle matrici spa1 e spa2, ovvero le matrici che presentano più entry

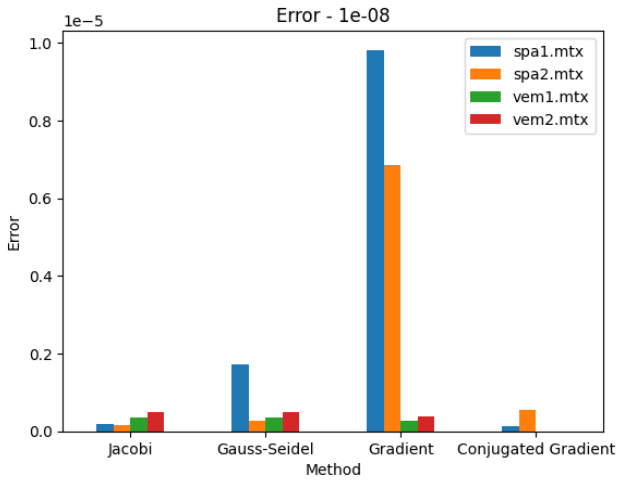

Figura 3.3: Errori con tolleranza di 10^{-8}.

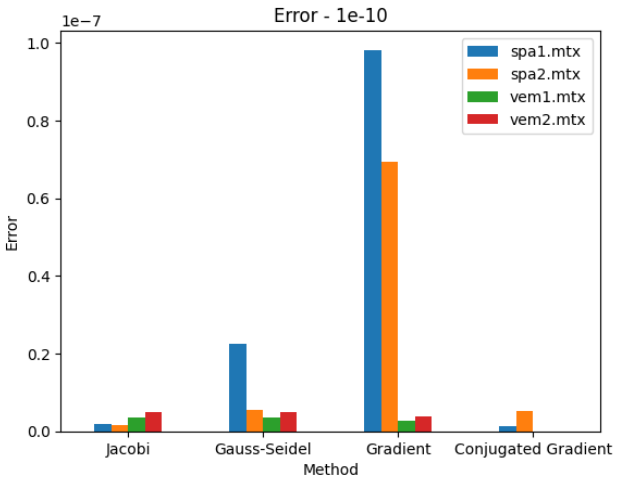

Figura 3.4: Errori con tolleranza di 10^{-10}.

diverse da 0 in percentuale, mentre il metodo di **Jacobi** sembra presentare errori più alti nel caso di matrici più sparse, coma vem1 e vem2. Infine, il metodo di **Gauss-Seidel** sembra produrre errori abbastanza bilanciati, con errori più alti solamente sulla matrice spa1.

3.1.2 Errori al variare della matrice

Di seguito vengono riportate delle immagini che rappresentano i grafici dell'errore normalizzato dei quattro metodi su una matrice, al variare della tolleranza, con asse x e y in scala logaritmica.

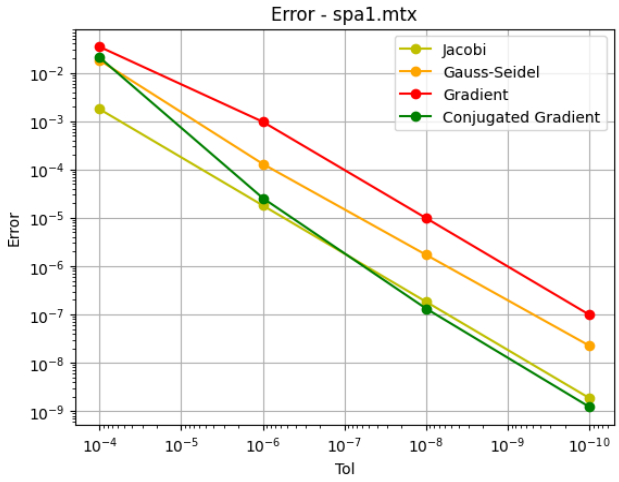

Figura 3.5: Errori su spa1.

Figura 3.6: Errori su spa2.

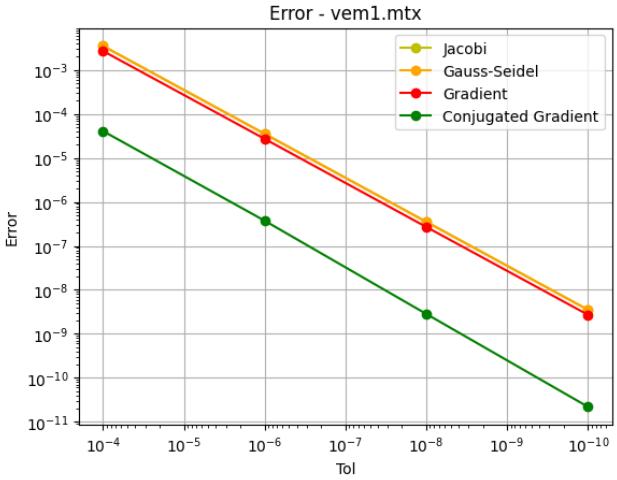

Figura 3.7: Errori su vem1.

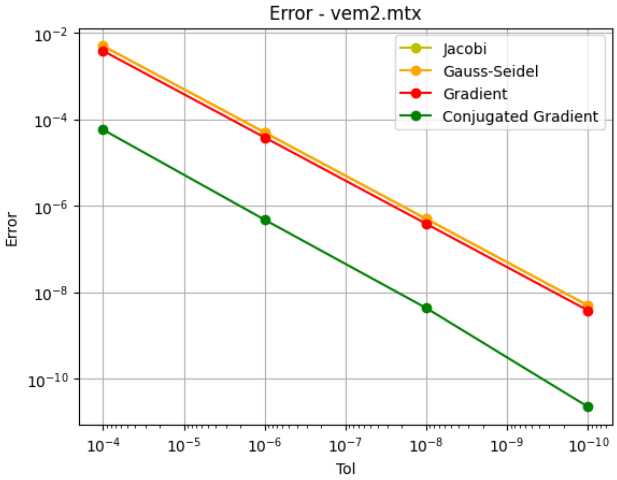

Figura 3.8: Errori su vem2.

Il metodo del **Gradiente Coniugato** su vem1 e vem2 presenta un errore di due ordini di grandezza inferiore rispetto agli altri metodi.

Per quanto riguarda gli errori su spa1 e spa2 si nota che i quattro metodi si comportano in maniera molto simile al variare della tolleranza, tranne per il metodo del **Gradiente Coniugato** su spa1 che migliora più rapidamente.

3.2 Numero di Iterazioni

Come per gli errori, e come prevedibile, anche il numero di iterazioni tra metodi che operano su matrici in formato sparso ed i metodi che operano su matrici in formato denso è uguale.

3.2.1 Numero di Iterazioni al Variare della tolleranza

Di seguito vengono riportati i grafici che rappresentano il numero di iterazioni di ciascun metodo, sulle quattro matrici al variare della tolleranza.

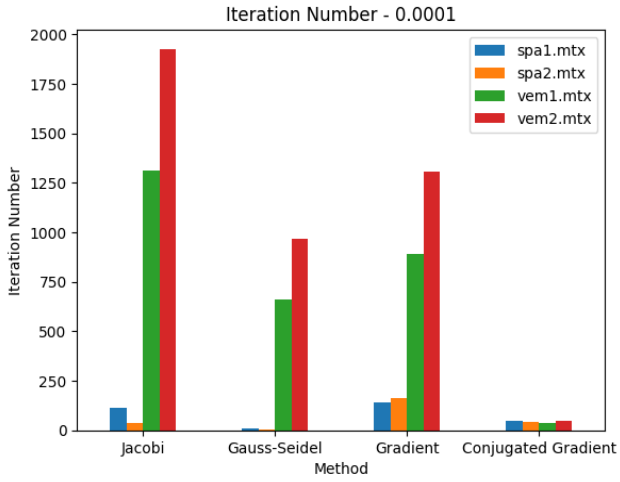

Figura 3.9: Numero di Iterazioni con tolleranza 10^{-4}.

Figura 3.10: Numero di Iterazioni con tolleranza 10^{-6}.

Dai grafici si nota che, come prevedibile, il metodo del **Gradiente Coniugato** presenta un numero di iterazioni estremamente basso anche al diminuire della tolleranza, soprattutto rispetto agli altri metodi.

Il metodo che tende ad avere performance peggiori invece pare essere il metodo del **Gradiente**, probabilmente perchè, come accennato precedentemente, esso tende ad oscillare (soprattutto sulle matrici spa1 e spa2, come citato in 2.1.1), e dunque soprattutto con valori di tolleranza più ridotti, esso dovrà eseguire un numero maggiore di iterazioni.

Il metodo di **Jacobi**, come previsto, ha prodotto un numero maggiore di iterazioni rispetto al metodo di **Gauss-Seidel**.

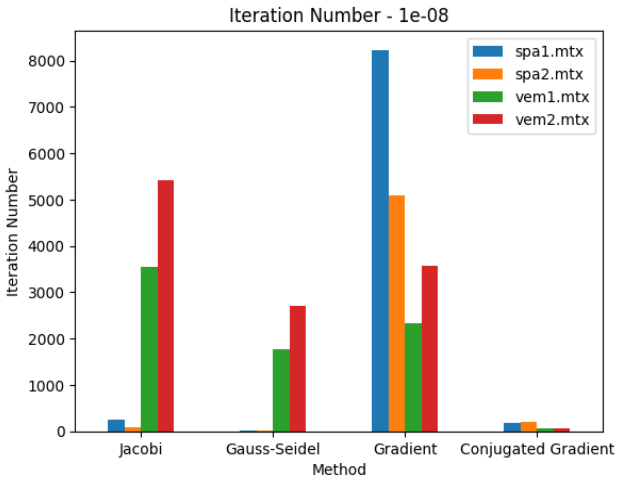

Figura 3.11: Numero di Iterazioni con tolleranza 10^{-8}.

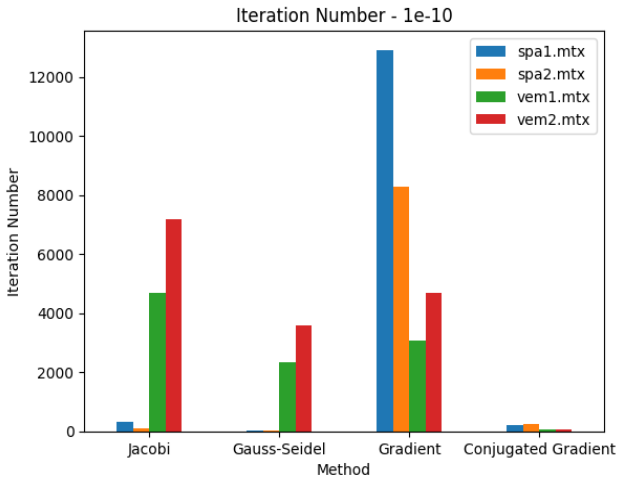

Figura 3.12: Numero di Iterazioni con tolleranza 10^{-10}.

3.2.2 Numero di Iterazioni al variare della matrice

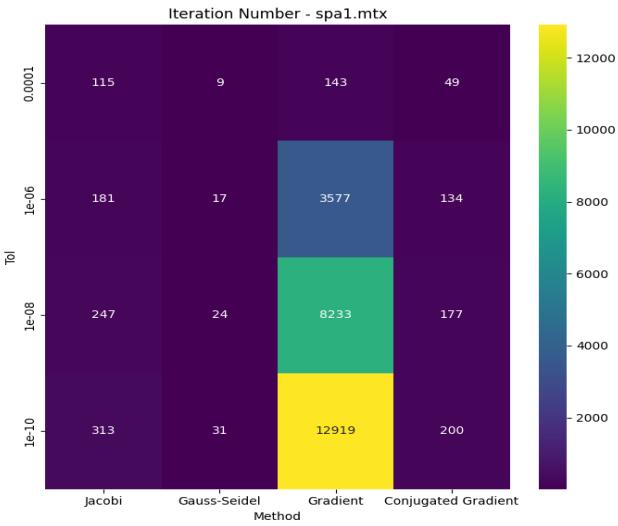

Figura 3.13: Numero di Iterazioni su spa1

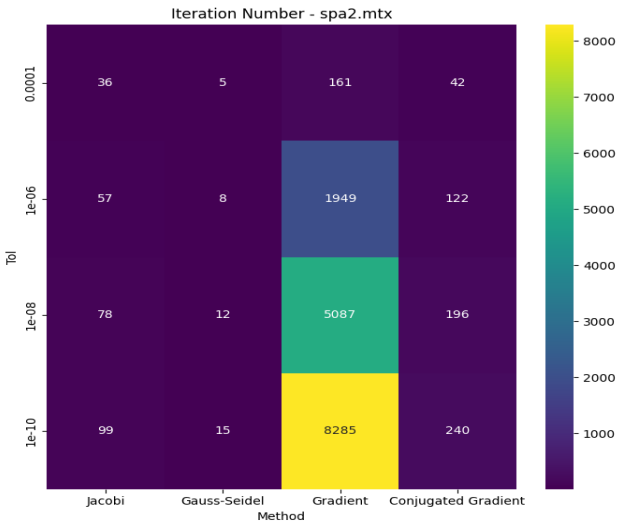

Figura 3.14: Numero di Iterazioni su spa2

Il metodo del **Gradiente Coniugato** si comporta molto bene sia su matrici sparse (vem1, vem2) che su matrici un po' più dense (spa1, spa2), **Jacobi** e **Gauss-Seidel** hanno avuto più difficoltà nel raggiungere convergenza sulle matrici più sparse, ed infine il metodo del **Gradiente** ha avuto iniziali difficoltà sulle matrici più sparse ma, al diminuire della tolleranza, ha prodotto un numero di iterazioni molto alto anche sulle matrici più dense.

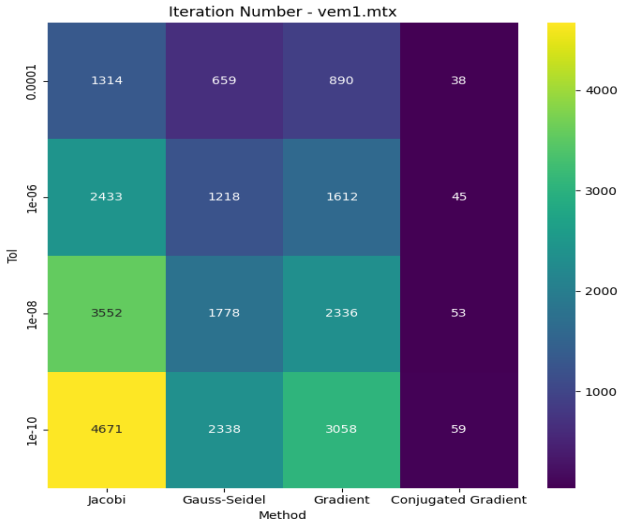

Figura 3.15: Numero di Iterazioni su vem1

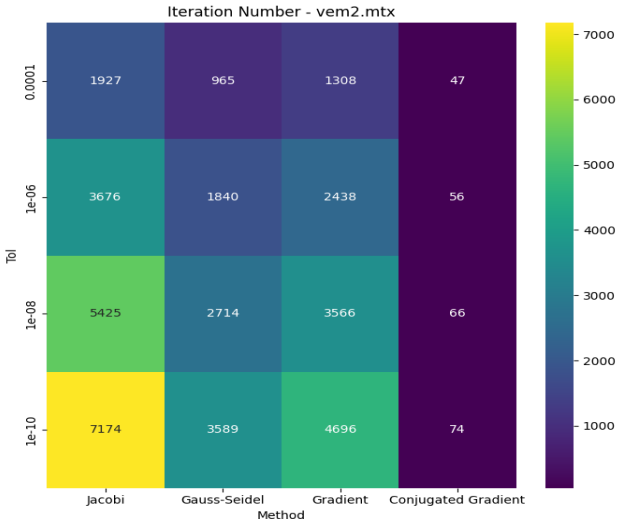

Figura 3.16: Numero di Iterazioni su vem2

3.3 Tempi di Esecuzione

Al contrario dell'errore relativo e del numero di iterazione, il tempo di esecuzione cambia se si decide di usare la versione "sparsa" dei metodi piuttosto che la versione "densa".

3.3.1 Al variare della tolleranza

Di seguito vengono riportati i grafici che riportano i tempi di esecuzione di ciascun metodo, sulle quattro matrici al variare della tolleranza, sia in formato denso che sparso.

Formato Denso

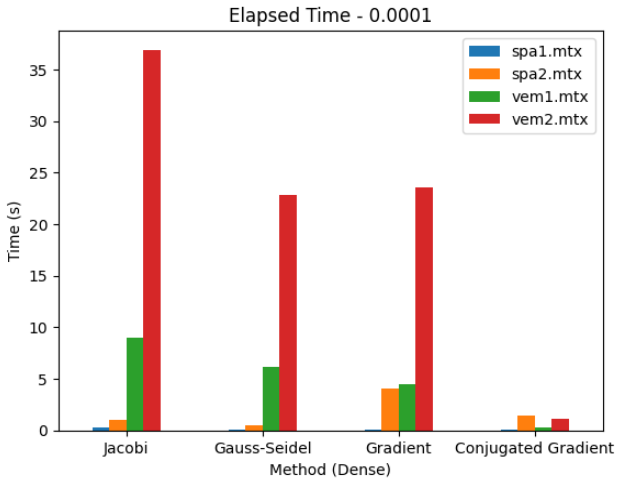

Figura 3.17: Tempi di Esecuzione su matrici dense con tolleranza 10^{-4}.

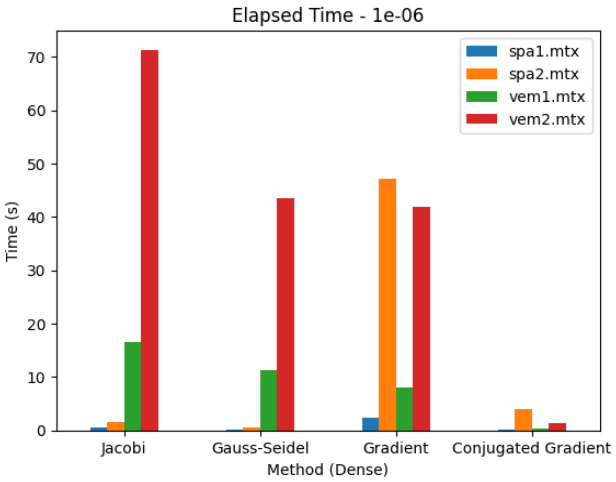

Figura 3.18: Tempi di Esecuzione su matrici dense con tolleranza 10^{-6}.

Il metodo del **Gradiente Coniugato** impega una quantità particolarmente bassa di tempo, probabilmente per il fatto che il numero d'iterazioni che compie è estremamente ridotto, come visto precedentemente.

I tempi di esecuzione di **Gauss-Seidel** e del Metodo del **Gradiente** sulla matrice vem2 sono molto simili, ma il metodo del **Gradiente** tende ad avere tempi molto più elevati sulla matrice spa2, soprattutto al diminuire della tolleranza.

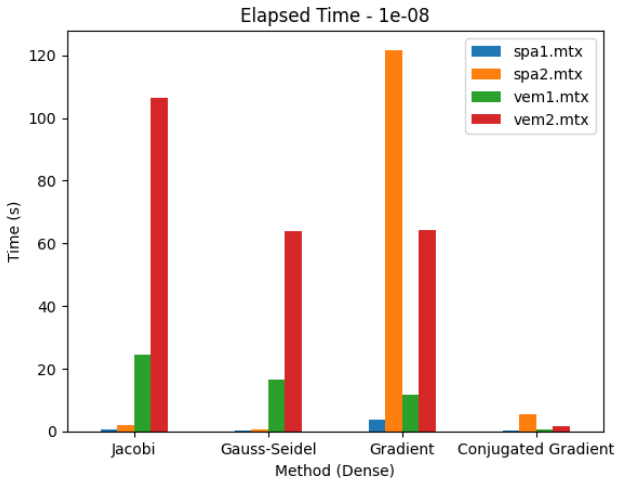

Figura 3.19: Tempi di Esecuzione su matrici dense con tolleranza 10^{-8}.

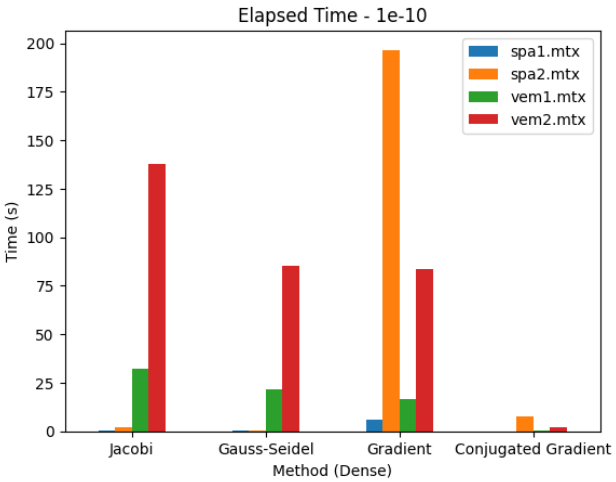

Figura 3.20: Tempi di Esecuzione su matrici dense con tolleranza 10^{-10}.

Jacobi è il metodo che presenta i tempi più elevati in linea di massima, fatta eccezione per la matrice spa2, dove il metodo del **Gradiente**, come detto in precedenza, impiega molto tempo per arrivare a convergenza.

Il metodo di **Jacobi** e quello di **Gauss-Seidel** impiegano molto meno tempo sulle matrici più ”dense”, mentre tendono ad avere un tempo di esecuzione più alto sulle matrici ”sparse”.

La matrice su cui gli algoritmi impiegano più tempo è la matrice vem2, eccetto che per il caso di tolleranza ridotta. In tal caso la matrice che impiega più tempo è spa2 a causa del metodo del **Gradiente**, che tende ad oscillare.

Formato Sparso

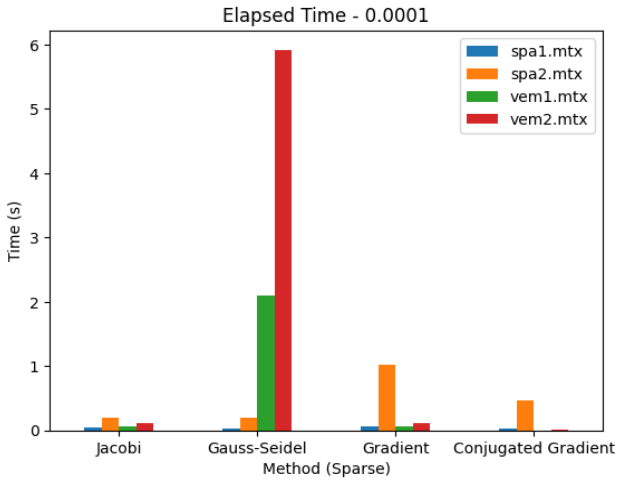

Figura 3.21: Tempi di Esecuzione su matrici sparse con tolleranza 10^{-4}.

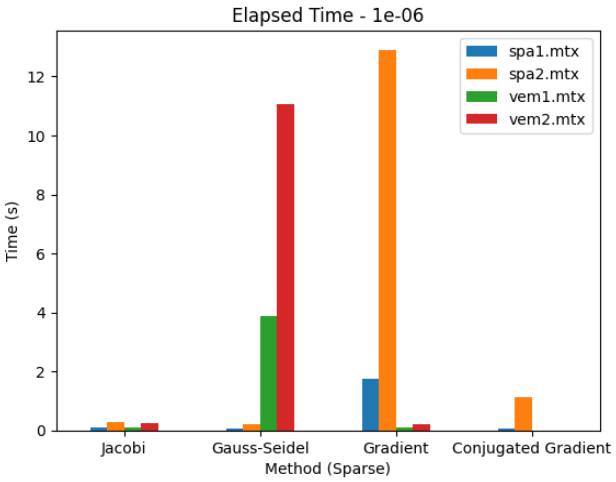

Figura 3.22: Tempi di Esecuzione su matrici sparse con tolleranza 10^{-6}.

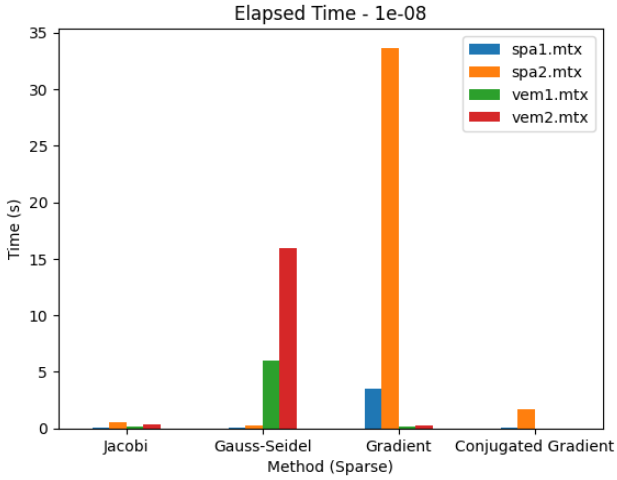

Figura 3.23: Tempi di Esecuzione su matrici sparse con tolleranza 10^{-8}.

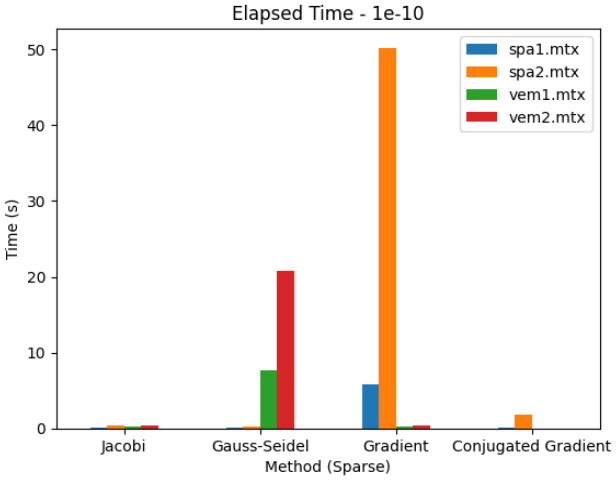

Figura 3.24: Tempi di Esecuzione su matrici sparse con tolleranza 10^{-10}.

Dai grafici si può notare che gli algoritmi che impiegano più tempo di esecuzione sono il metodo di **Gauss-Seidel** che impiega un tempo abbastanza ”costante” anche al variare della tolleranza, e il metodo del **Gradiente** che è particolarmente lento sulla matrice spa2, soprattutto al diminuire della tolleranza, mentre si comporta abbastanza bene sulle matrici più sparse.

Il metodo di **Jacobi** e il metodo del **Gradiente Coniugato** sono invece molto rapidi su tutte e quattro le matrici e su ogni tolleranza.

3.3.2 Al variare della matrice

Formato Denso

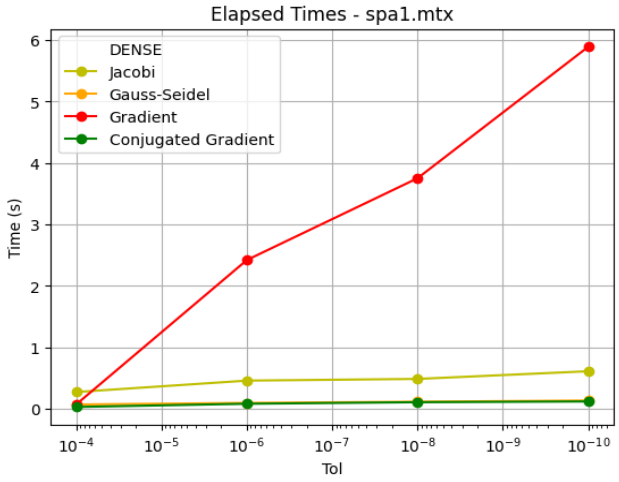

Figura 3.25: Tempi di Esecuzione su matrici dense su spa1.

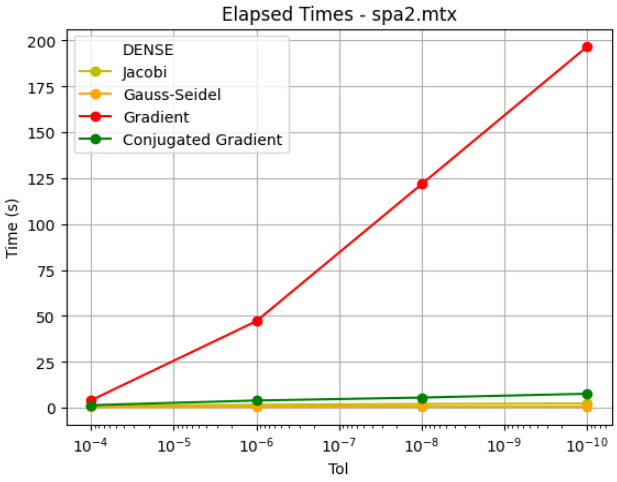

Figura 3.26: Tempi di Esecuzione su matrici dense su spa2.

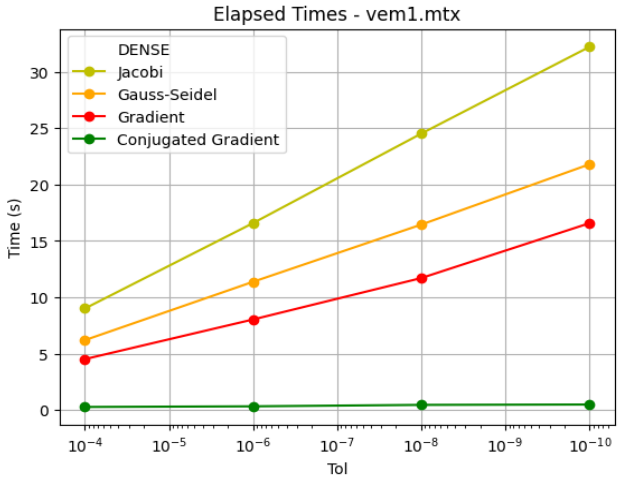

Figura 3.27: Tempi di Esecuzione su matrici dense su vem1.

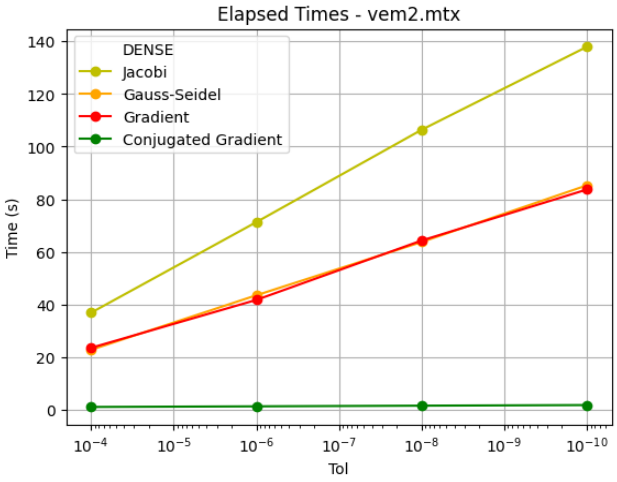

Figura 3.28: Tempi di Esecuzione su matrici dense su vem2.

Sulle matrici Spa1 e spa2 il metodo più lento è sicuramente il metodo del **Gradiente**, soprattutto al diminuire della tolleranza. Gli altri metodi mantengono dei tempi di esecuzione molto vicini allo zero. Per quanto riguarda le matrici vem1 e vem2, tutti i metodi incontrano difficoltà eccetto per il metodo del **Gradiente Coniugato**. In particolare, il metodo che impiega più tempo è **Jacobi**, mentre il metodo del **Gradiente** e di **Gauss-Seidel** hanno tempi simili.

Formato Sparso

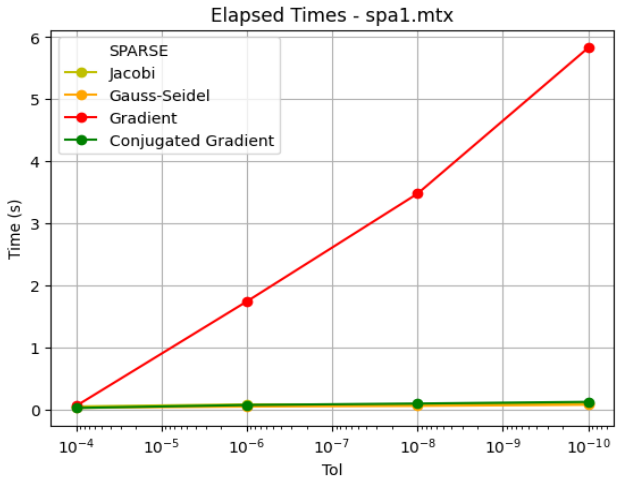

Figura 3.29: Tempi di Esecuzione su matrici sparse su spa1.

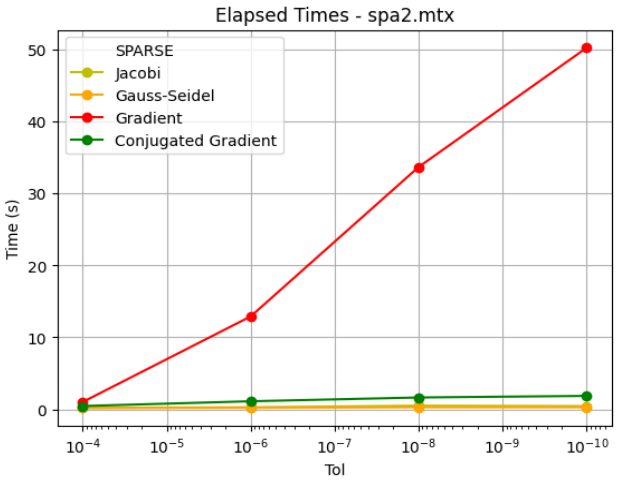

Figura 3.30: Tempi di Esecuzione su matrici sparse su spa2.

Sulle matrici spa1 e spa2 il metodo più lento è sicuramente il metodo del **Gradiente**, soprattutto al diminuire della tolleranza. Gli altri metodi mantengono dei tempi di esecuzione molto vicini allo zero.

Sulle matrici vem1 e vem2 il metodo più lento è sicuramente il metodo di **Gauss-Seidel**, soprattutto al diminuire della tolleranza. Gli altri metodi mantengono dei tempi di esecuzione molto vicini allo zero.

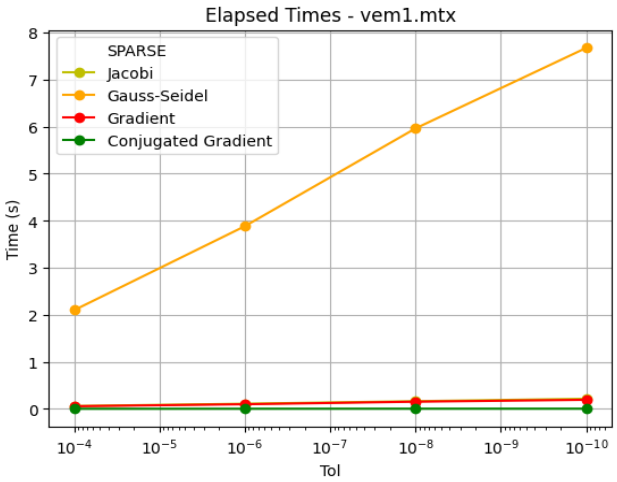

Figura 3.31: Tempi di Esecuzione su matrici sparse su vem1.

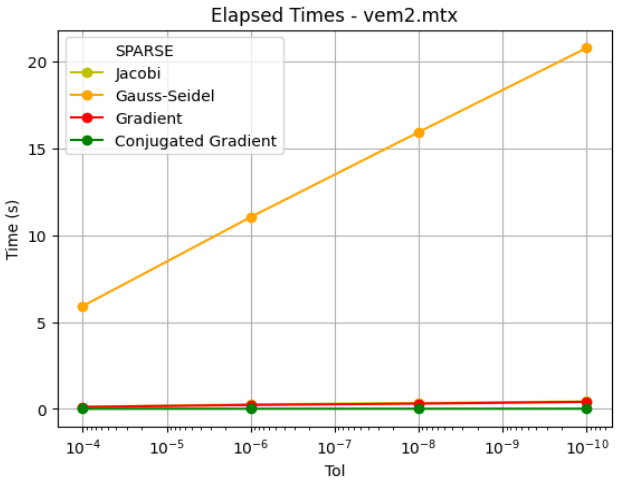

Figura 3.32: Tempi di Esecuzione su matrici sparse su vem2.

3.3.3 Confronto sparse e Dense

Di seguito sono riportati i grafici che confrontano i tempi di esecuzione dei metodi applicati a matrici in formato sparso rispetto a quelli applicati a matrici in formato denso, considerando una tolleranza pari a 10^{-10}:

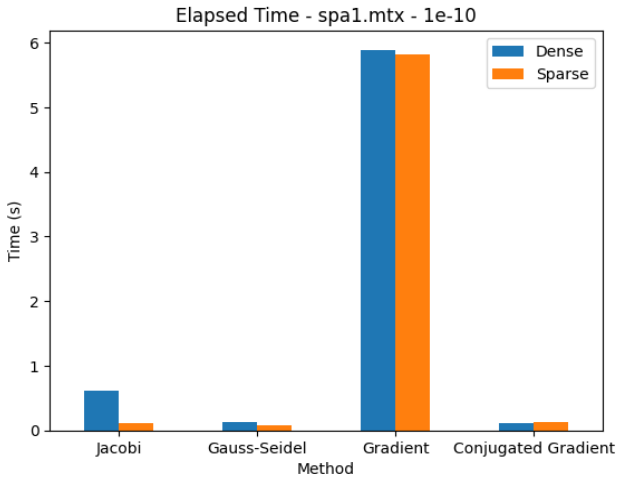

Figura 3.33: Tempo di Esecuzione di spa1.

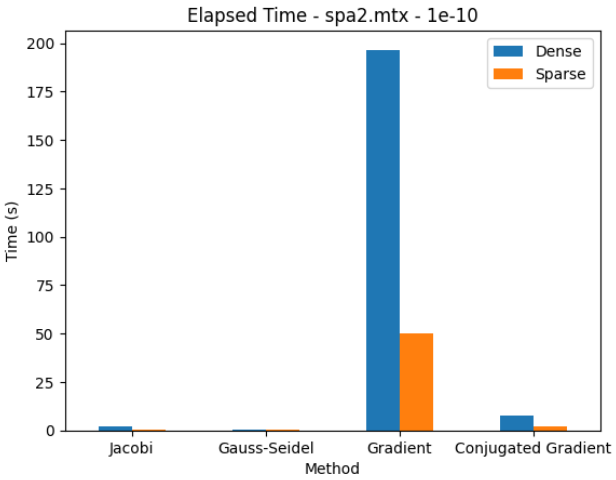

Figura 3.34: Tempo di Esecuzione di spa2.

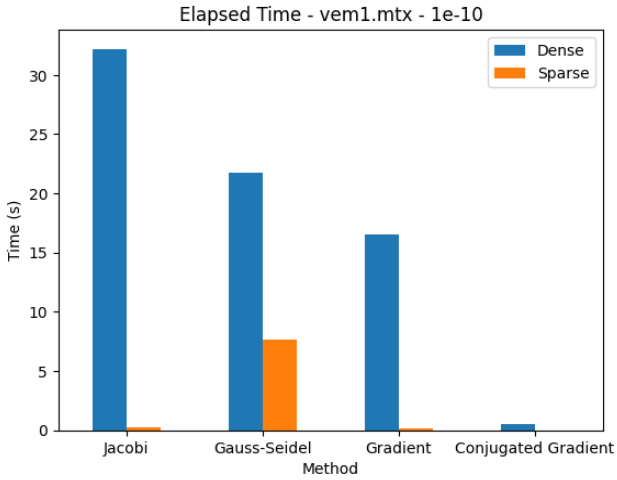

Figura 3.35: Tempo di Esecuzione di vem1.

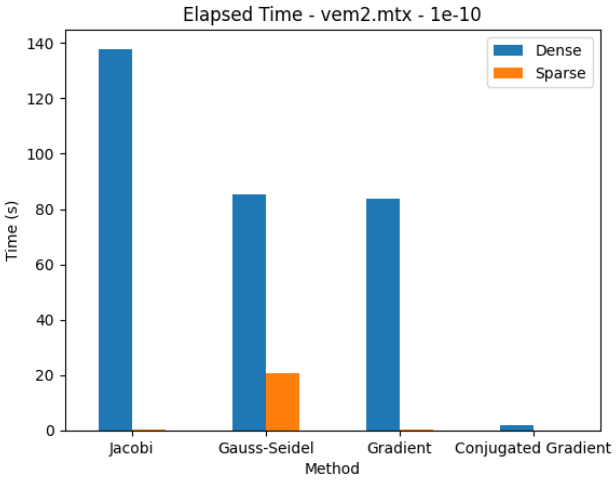

Figura 3.36: Tempo di Esecuzione di vem2.

Per quanto riguarda la matrice spa1, non si notano particolari differenze tra i metodi che lavorano su matrici in formato denso e quelli che lavorano su matrici in formato sparso.

Per quanto riguarda la matrice spa2, l'unico metodo che giova della rappresentazione in formato sparso, è il metodo del **Gradiente**, con un risparmio in termini di tempo pari a circa 150 secondi.

Per quanto riguarda le matrici vem1 e vem2, si può notare che per tutti i metodi lavorare su matrici sparse è molto più efficiente che lavorare su matrici addensate. Ciò è particolarmente evidente nel metodo di **Jacobi**, che impiega un tempo elevato nella risoluzione del sistema tramite l'utilizzo di matrici dense, mentre impiega qualche secondo al massimo per risolvere lo stesso sistema tramite l'utilizzo di matrici sparse.

L'unico metodo che non mostra particolari differenze in termini di tempo è il metodo del **Gradiente Coniugato**; questo è probabilmente dovuto al fatto che il numero di iterazioni del metodo è molto basso su qualsiasi matrice.

3.4 Extra - Metodo del Gradiente Empirico

Durante l'implementazione del metodo del **Gradiente**, si è osservato che il metodo non mostrava convergenza; pensando a un possibile errore, si è tentata una modifica. È emerso che, invertendo erroneamente due istruzioni, il metodo forniva risultati molto buoni sulle matrici del progetto.

La prova è stata estesa ad alcune matrici costruite ad hoc con autovalori molto distanti tra loro, e anche in questo caso si è riscontrato un buon funzionamento.

Probabilmente si tratta di una circostanza fortuita, ma, per completezza, di seguito vengono riportati i risultati prodotti dal metodo del "**Gradiente Empirico**".

Errori

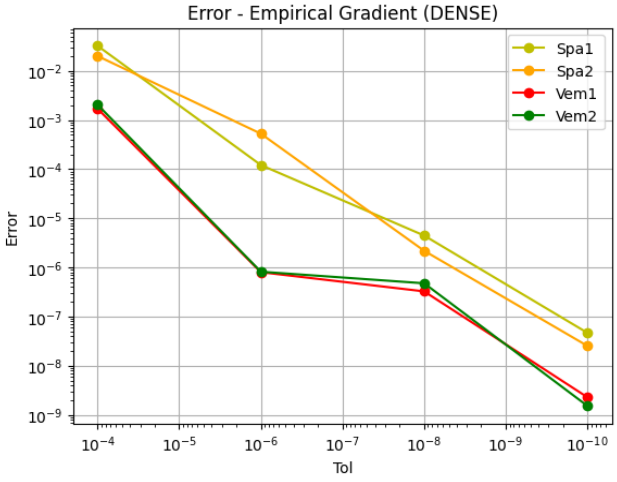

Figura 3.37: Errore su matrici in formato denso.

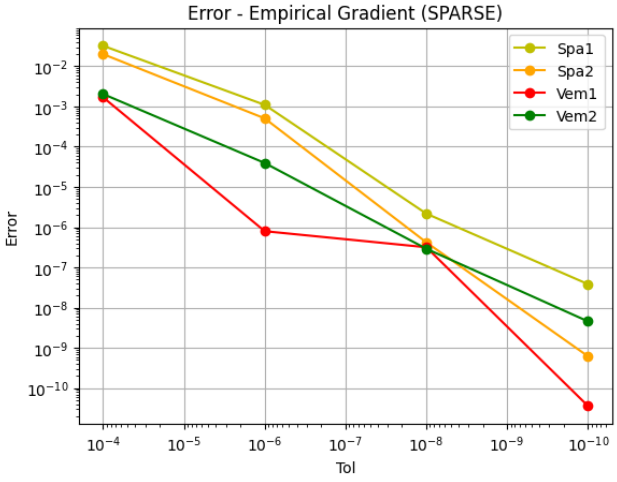

Figura 3.38: Errore su matrici in formato sparso.

Come prevedibile, gli errori di questo metodo diminuiscono al diminuire della tolleranza, sia quando il metodo opera su matrici in formato denso che quando il metodo opera su matrici in formato sparso.

Numero di Iterazioni

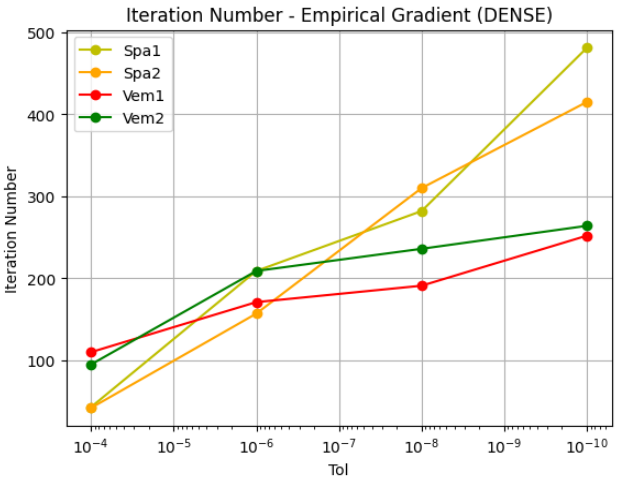

Figura 3.39: Numero di Iterazioni su matrici in formato denso.

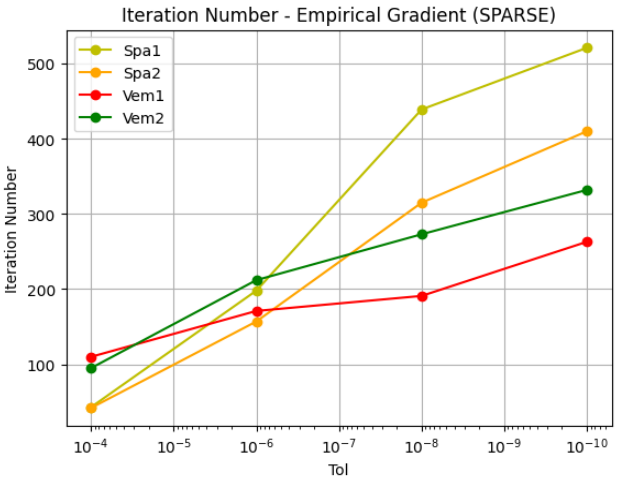

Figura 3.40: Numero di Iterazioni su matrici in formato sparso.

Il numero di iterazioni di questo metodo è paragonabile con quello del metodo del **Gradiente Coniugato**, ed è di gran lunga inferiore al numero di iterazioni del Gradiente tradizionale.

Si noti però che il numero di iterazioni varia leggermente ad ogni esecuzione, il che significa che il metodo non è **deterministico**.

Tempi di Esecuzione

Come si può notare dalle immagini, il tempo di esecuzione è estremamente ridotto. Infatti, il tempo peggiore è di solamente 10 secondi e viene riscontrato sulla matrice spa2 nel caso denso.

Questo tempo è estremamente minore di quanto era stato ottenuto con il metodo del **Gradiente** tradizionale.

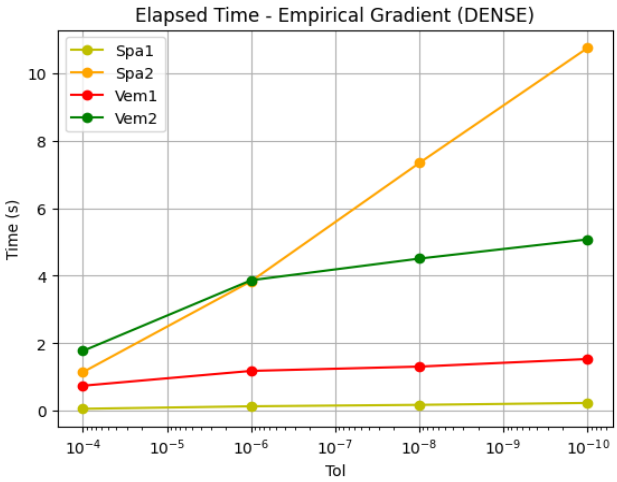

Figura 3.41: Tempo di Esecuzione su matrici in formato denso.

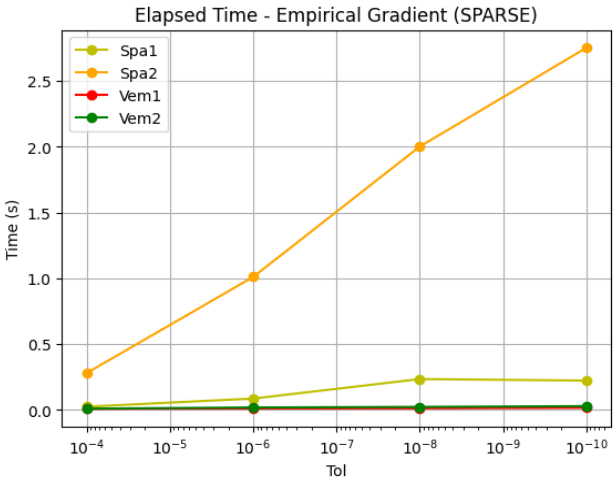

Figura 3.42: Tempo di Esecuzione su matrici in formato sparso.

4. Conclusioni

4.1 Breve Commento dei Risultati Ottenuti

Come prevedibile e come confermato dal capitolo 3, il metodo del **Gradiente Coniugato** è decisamente il metodo più veloce ed efficiente, con ottimi tempi di esecuzione sia su matrici in formato denso che sparso.

Il metodo di **Jacobi** ha presentato un numero di iterazioni maggiore del metodo di **Gauss Seidel**, come ci si aspettava, tuttavia le iterazioni del metodo di **Gauss-Seidel** sono state mediamente più veloci del metodo di **Jacobi**. Questo accade per come è stato implementato il metodo *solve_lower* discusso in 1.3.

Infine, il metodo del **Gradiente** ha mostrato un numero di iterazioni particolarmente elevato, e di conseguenza un tempo di esecuzione molto alto, dimostrandosi il metodo meno efficiente, soprattutto con valori di tolleranza molto bassi.

4.2 Sviluppi Futuri

La libreria è facilmente estendibile grazie alla sua struttura, descritta in 1.3, e dunque se si volessero aggiungere nuovi metodi iterativi basterebbe creare nuove specializzazioni della classe astratta *solver.py* che devono definire i metodi *solve_dense* e *solve_sparse* del nuovo metodo che si vuole implementare, seppur così verrebbero comunque sempre analizzate solo matrici SPD.