Introduzione

Questa libreria fornisce un set completo di strumenti per risolvere grandi sistemi lineari sparsi utilizzando diversi metodi iterativi. I componenti principali della libreria sono organizzati in diversi moduli e classi Python, ognuno progettato per gestire diversi aspetti della manipolazione delle matrici, della risoluzione iterativa e dell'analisi delle prestazioni.

CAPITOLO 1

Struttura della Libreria

La libreria è strutturata attorno a tre moduli principali:

- *Executers*: Queste classi implementano i diversi metodi iterativi per risolvere sistemi lineari.
- *Utility*: Fornisce funzioni di supporto per la lettura delle matrici, il controllo delle proprietà delle matrici (come la positività definita e la simmetria), e la scrittura dei risultati su file.
- *matrixAnalysis*: Fornisce le funzioni per analizzare le matrici e visualizzare le loro proprietà; come il numero di condizionamento, il grafico di sparsità e la distribuzione dei valori contenuti nelle matrici.
- *solverRunner*: file da dove è possibile effettuare il lancio dei metodi iterativi, con la conseguente creazioni dei grafici di prestazione.

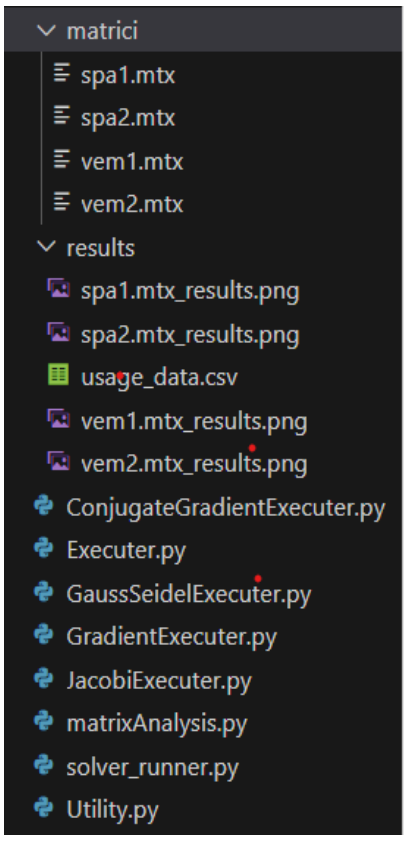

Figura 1 directory di progetto

Di seguito viene presentata una descrizione dettagliata dei moduli principali e delle loro interazioni:

Classe Astratta: Executer

La classe Executer funge da classe base astratta che definisce l'interfaccia comune per tutti i solver iterativi. Essa stabilisce la struttura di base necessaria per implementare i diversi risolutori, inclusi l'inizializzazione, i controlli di convergenza, e l'aggiornamento iterativo delle soluzioni. La classe memorizza attributi comuni, come:

- La matrice A (matrice dei coefficienti del sistema lineare)
- Il vettore soluzione x
- I livelli di tolleranza per la convergenza
- Il numero massimo di iterazioni
- Il vettore del termine noto b

La classe Executer implementa la funzione *method_execution(),* che rappresenta il ciclo principale del metodo iterativo. Questa funzione aggiorna iterativamente il vettore soluzione utilizzando una funzione di aggiornamento *update_function(),* che è specifica per ciascun solver.

Le classi derivate implementano solver iterativi specifici:

- *JacobiExecuter*: Implementa il metodo iterativo di Jacobi.
- *GaussSeidelExecuter*: Implementa il metodo iterativo di Gauss-Seidel.
- *GradientExecuter*: Implementa il metodo di discesa del gradiente.
- *ConjugateGradientExecuter*: Implementa il metodo del gradiente coniugato per risolvere matrici definite positive.

Ogni classe solver sovrascrive il metodo *update_function()* per fornire le regole di aggiornamento specifiche per il rispettivo metodo iterativo. Queste classi utilizzano la matrice A e la soluzione corrente x per calcolare il residuo e aggiornare la soluzione iterativamente fino a quando non si raggiunge la convergenza o il numero massimo di iterazioni preimpostato.

2. Classe Utility:

La classe Utility fornisce un insieme di metodi statici che supportano le operazioni principali richieste dalle classi solver.

Le funzionalità principali di questo modulo sono:

- Il caricamento delle matrici tramite le funzioni *get_matrix_path()* e *read()*.
- Il controllo delle proprietà delle matrici:

1. *check_size():* verifica che la matrice passata come parametro sia quadrata
2. *is_positive_definite():* verifica che la matrice passata come parametro sia definita positiva.
3. *is_symmetric():* verifica che la matrice passata come parametro sia simmetrica.

- La funzione *write_usage_csv()* scrive i risultati delle esecuzioni dei solver su un file CSV, registrando metriche come il numero di iterazioni, i residui e l'uso del tempo.

3. MatrixAnalysis:

Questo modulo contiene funzioni progettate per analizzare e visualizzare le proprietà delle matrici, particolarmente utili per effettuare considerazioni sui dati relativi alle performance dei solutori iterativi sulle relative matrici.

- *compute_condition_number():* Calcola e stampa il numero di condizione di una matrice sparsa data. Il numero di condizione è una metrica chiave che indica quanto una matrice sia sensibile alle operazioni numeriche, influenzando il comportamento di convergenza dei solver iterativi.
- *plot_sparsity_pattern():* Visualizza il pattern di scarsità di una matrice. Questo grafico ha lo scopo di comprendere la distribuzione degli elementi non nulli all'interno della matrice, che può influenzare la performance di risoluzione dei diversi risolutori.
- *plot_value_distribution():* Traccia la distribuzione dei valori all'interno di una matrice, distinguendo tra elementi nulli e non nulli.

4. Lancio dell'applicativo e Flusso di Lavoro:

Tramite il modulo *solverRunner* è possibile effettuare il lancio dell'applicativo, dove vengono applicati i diversi risolutori iterativi su ogni matrice nei diversi livelli di tolleranza, consentendo di trovare soluzioni che soddisfano criteri di accuratezza specifici. Facendo ciò, sarà poi possibile confrontare come le prestazioni dei diversi metodi differiscono per ogni matrice. Durante questa fase, per ogni combinazione di matrice e solver, vengono monitorati e registrati i dettagli delle iterazioni, il residuo finale e i tempi di calcolo. Lo script principale guida il processo complessivo richiamando funzioni e metodi dalla classe Utility per il caricamento delle matrici, l'esecuzione dei controlli e la scrittura dei risultati. Successivamente, le classi esecutrici vengono istanziate per ogni metodo solver utilizzando i dati delle matrici e risolvendo i sistemi in modo iterativo. Al termine dell'esecuzione, nella direcory *results* verranno creati i file .png contenenti i grafici relativi alle prestazioni dei risolutori per ogni diversa matrice e tolleranza.

Nelle immagini sottostanti vengono presentate le implementazioni dei solutori iterativi richiesti da specifiche:

```python
def update_function(self):
    # Compute the residual r(k) = b - A * x(k)
    r = self.b - self.matrix.dot(self.x)
    # Update x(k+1) = x(k) + D^-1 * r(k) (in-place update)
    np.multiply(self.diagonal_inv, r, out=r)  # r becomes D^-1 * r(k)
    self.x += r  # x(k+1) = x(k) + D^-1 * r(k)

    return self.x, r
```

Figura 2 Implementazione metodo di Jacobi

```python
def update_function(self):
    r = self.b - self.matrix.dot(self.x)  # Compute residual r(k) = b - A * x(k)
    y = spsolve(self.triang_inf, r)
    self.x += y
    return self.x, r
```

Figura 3 Implementazione metodo di Gaus-Seidel

```python
def update_function(self):
    # Compute residual r(k) = b - A * x(k)
    r = self.b - self.matrix.dot(self.x)

    # Compute A * r
    Ar = self.matrix.dot(r)

    # Compute r^T * r
    r_dot_r = np.dot(r, r)

    # Compute alpha = (r^T * r) / (r^T * A * r)
    r_Ar_dot = np.dot(r, Ar)  # r^T * (A * r)
    if r_Ar_dot == 0:  # Prevent division by zero
        alpha = 0
    else:
        alpha = r_dot_r / r_Ar_dot

    # Update x(k+1) = x(k) + alpha * r
    self.x += alpha * r

    return self.x, r
```

Figura 4 Implementazione metodo del gradiente

```python
def update_function(self):
    self.residual = self.b - self.matrix.dot(self.x)

    # Initialization of p(0)
    if self.r_old is None or self.p_old is None:
        p = self.residual
    else:
        # Compute beta
        beta = np.dot(self.residual, self.residual) / np.dot(self.r_old, self.r_old)
        # Update p(k) = r(k) + beta * p(k-1)
        p = self.residual + beta * self.p_old

    # Compute A * p
    Ap = self.matrix.dot(p)

    # Compute alpha = (r^T * r) / (p^T * A * p)
    alpha = np.dot(self.residual, self.residual) / np.dot(p, Ap)

    # Update x(k+1) = x(k) + alpha * p
    self.x = self.x + alpha * p

    self.r_old = self.residual
    self.p_old = p

    return self.x, self.residual
```

Figura 5 Implementazione metodo del gradiente coniugato

CAPITOLO 2

Analisi preliminare delle matrici

Come fase iniziale del progetto, sono state analizzate le proprietà delle matrici fornite, al fine di identificare le caratteristiche rilevanti che potrebbero influenzare le prestazioni dei risolutori durante l'elaborazione.

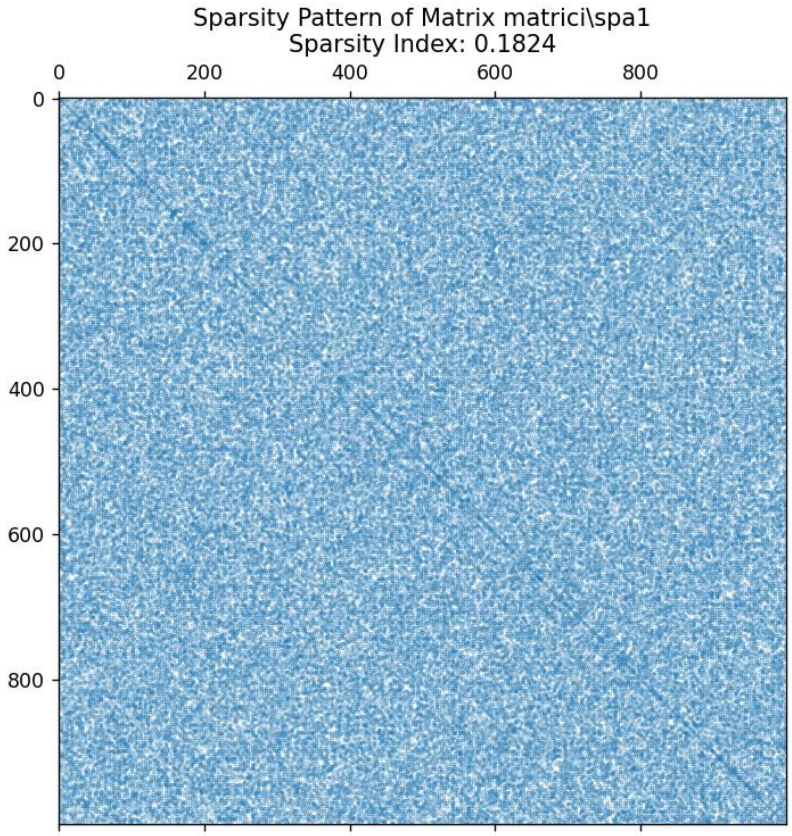

*Figura 6 visualizzazione sparsità **spa1***

L'immagine in Figura 6 mostra il grafico di sparsità della matrice **spa1**. In questa matrice, gli elementi non nulli sono distribuiti uniformemente senza un pattern specifico. L'indice di sparsità è 0.1824, indicando che il 18.24% degli elementi sono diversi da zero. La diagonale presenta una significativa costanza di elementi non nulli. La matrice spa1 ha un numero di condizionamento relativamente elevato, pari a 2.05 × 10^3 (2050).

La matrice spa2 raffigurata in Figura 7, al contrario, appare sensibilmente più densa rispetto alla spa1, anche se l'indice di sparsità è quasi identico, con la differenza attribuibile alla diversa scala della matrice. Il numero di condizionamento della matrice spa2 è 1.41 × 10^3 (1410), risultando significativamente inferiore rispetto a quello della matrice spa1.

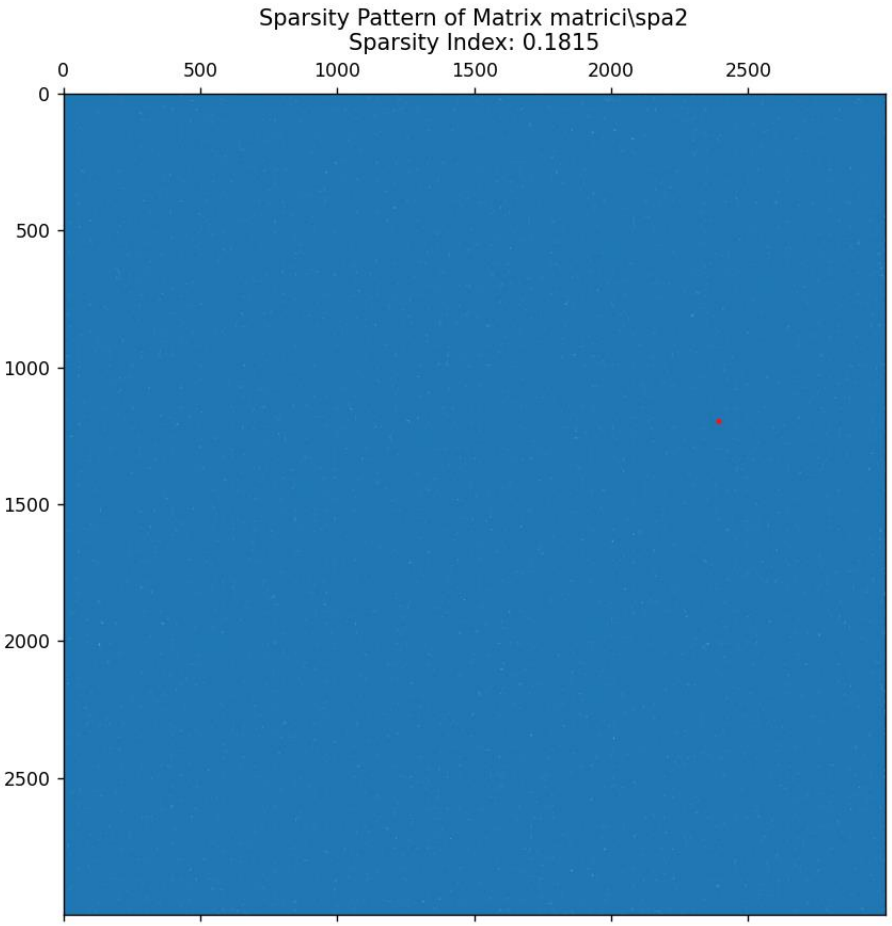

*Figura 7 visualizzazione sparsità **spa2***

Nei grafici in Figura 8 e Figura 9 è possibile visualizzare come la matrice vem2 presenta una dominanza diagonale e una struttura sparsa, con un indice di sparsità di 0.0031, che corrisponde allo 0.31% degli elementi non nulli. Risultati simili si osservano anche nella matrice vem1, che ha una struttura sparsa con un indice di sparsità leggermente maggiore e a dominanza diagonale. Il numero di condizionamento per la matrice vem1 è 3.25 × 10^2 (325), mentre per la matrice vem2 è 5.07 × 10^2 (507). Questi valori sono significativamente inferiori rispetto a quelli delle matrici spa1 e spa2, suggerendo che le matrici vem1 e vem2 sono più ben condizionate rispetto a quelle con numeri di condizionamento più elevati.

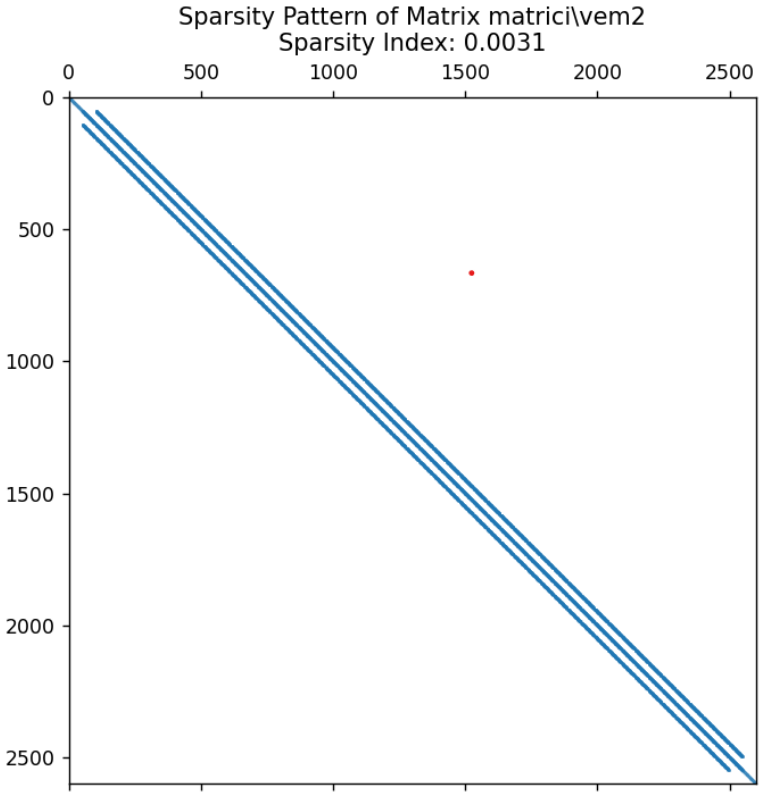

*Figura 8 visualizzazione sparsità **vem1***

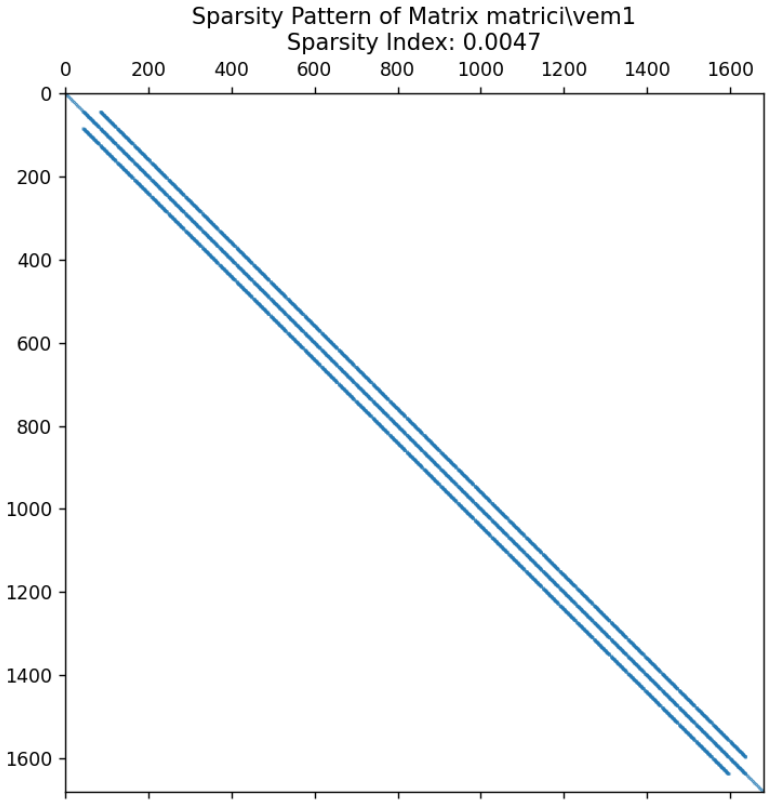

*Figura 9 visualizzazione sparsità **vem2***

CAPITOLO 3

Analisi dei risultati ottenuti

Come risultati forniti dall'applicativo sono presenti i dati relativi alle performance di ogni metodo iterativo su ogni matrice nei diversi livelli di tolleranza. Successivamente verranno discusse informazioni riguardanti il tempo d'esecuzione di ogni risolutore, il residuo finale raggiunto e la progressione del residuo nel numero di iterazioni richieste fino arrivare a convergenza.

Analisi del Tempo di Esecuzione dei Risolutori su Diverse Matrici e Tolleranze

Analizzando le prestazioni dei diversi metodi iterativi sulla matrice **spa1** in Figura 10, si osserva che il tempo di esecuzione di ciascun solutore tende ad aumentare con un grado di tolleranza più stringente. Questo aumento di tempo è giustificato dal fatto che un livello di precisione maggiore nella soluzione richiede un numero superiore di iterazioni per raggiungere la convergenza desiderata. Il metodo del Gradiente risulta essere il meno performante tra quelli considerati. Infatti, con la riduzione della tolleranza, il tempo necessario per raggiungere la convergenza aumenta sensibilmente, arrivando addirittura a triplicarsi. Al contrario, i metodi iterativi più efficienti per la matrice in questione sono il metodo Jacobi e il metodo del Gradiente Coniugato. Entrambi mostrano una tendenza relativamente stabile in termini di tempo di esecuzione al diminuire della tolleranza, con un aumento moderato nel test finale con una tolleranza pari a 1e-10. In questo scenario, Jacobi e il Gradiente Coniugato mostrano incrementi rispettivi del +0,23 e +0,20 nel tempo di esecuzione. Questo risultato evidenzia come entrambi i metodi siano capaci di gestire bene le richieste di precisione senza un drastico aumento del tempo di calcolo. Infine, il metodo di Gauss-Seidel mostra un comportamento intermedio: sebbene non sia il metodo più lento, il suo tempo di esecuzione raddoppia quando viene richiesta una tolleranza molto stretta, come 1e-10.

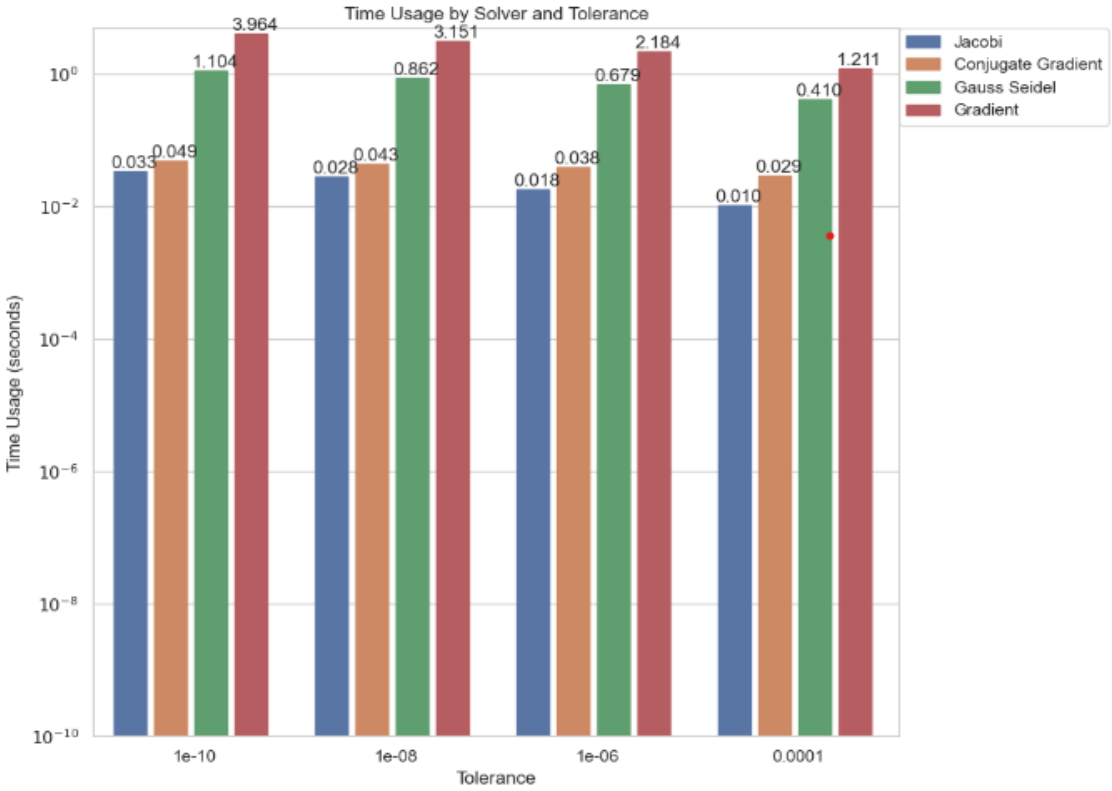

*Figura 10 tempistiche d'esecuzione sulla matrice **spa1***

Una simile ed equiparabile tipologia di risultati è possibile ottenerla quando vengono applicati i medesimi risultati alla matrice **spa2** visualizzabili in Figura 11. In questo caso, le tendenze di ogni risolutore sono pressoché identiche ai risolutori applicati alla matrice precedente, ma con una dilatazione dei tempi d'esecuzione per le medesime tolleranze. Questa considerazione è motivabile dal fatto che le due matrici hanno numero di condizionamento e indice di sparsità molto simili, differenziandosi unicamente dalla quantità di elementi effettivamente contenuti. La matrice **spa2**, infatti, contiene un numero nettamente superiore di elementi, aumentando così il tempo necessario per arrivare a convergenza di ogni metodo.

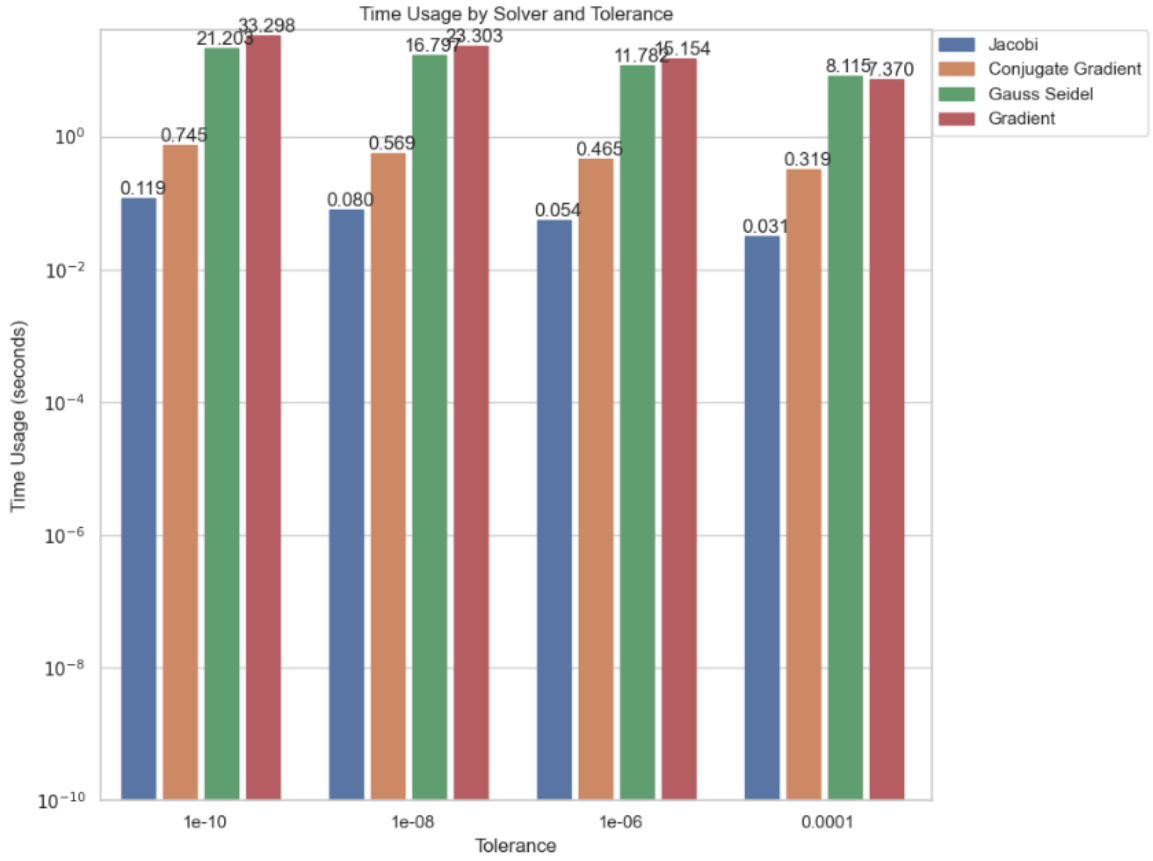

*Figura 11 tempistiche d'esecuzione sulla matrice **spa2***

I maggiori cambiamenti avvengono quando le diverse implementazioni vengono applicate nelle matrici **vem1** e **vem2**. In Figura 12 e Figura 13 è possibile notare come i tempi d'esecuzione siano nettamente inferiori rispetto alle precedenti, motivato dalla presenza di matrici a dominanza diagonale con basso indice di sparsità. Per questi due test il miglior risolutore trova luogo nel metodo del gradiente coniugato, rimanendo stabile al variare delle tolleranze ad un tempo di esecuzione pari a 0.002 per vem1 e 0.003 per vem2. Il risolutore meno performante invece è quello di Gaus-Seidel, il quale si presenta con tempistiche d'esecuzione nettamente maggiori rispetto agli altri metodi iterativi. È possibile notare come questo metodo subisce un ulteriore peggioramento nella matrice vem2. Risulta interessante notare come il metodo di Jacobi dimostri una notevole stabilità in questa grandezza nelle matrici testate, arrivando a performare nel caso peggiore tra tutte le matrici con una tempistica pari a 0.218 s. È possibile riscontra che nella matrice **vem2** le performance peggiorino in ogni punto rispetto a vem1, dato che la prima matrice contiene un numero molto maggiore di elementi. Nelle matrici vem1 e vem2 quindi, il risolutore più veloce è il gradiente coniugato, seguito da Jacobi, dal gradiente e dal metodo di Gaus-Seidel.

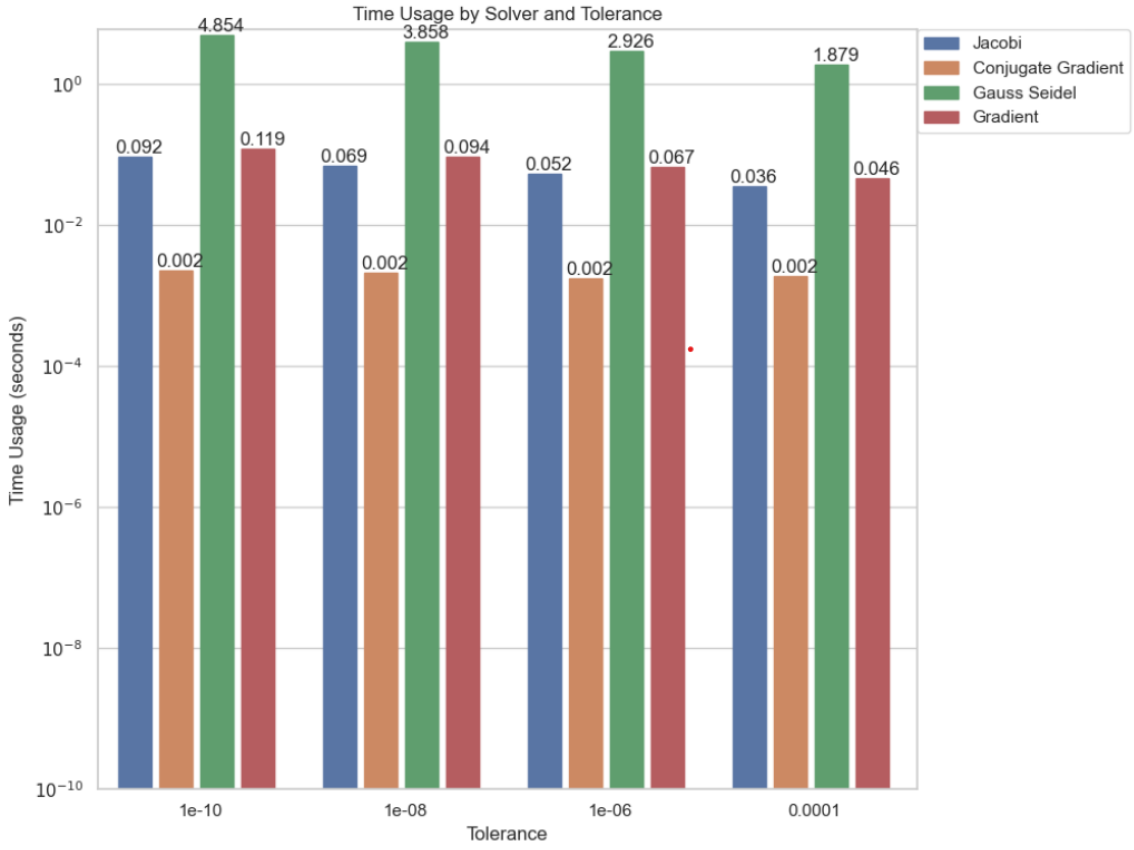

*Figura 12 tempistiche d'esecuzione sulla matrice **vem1***

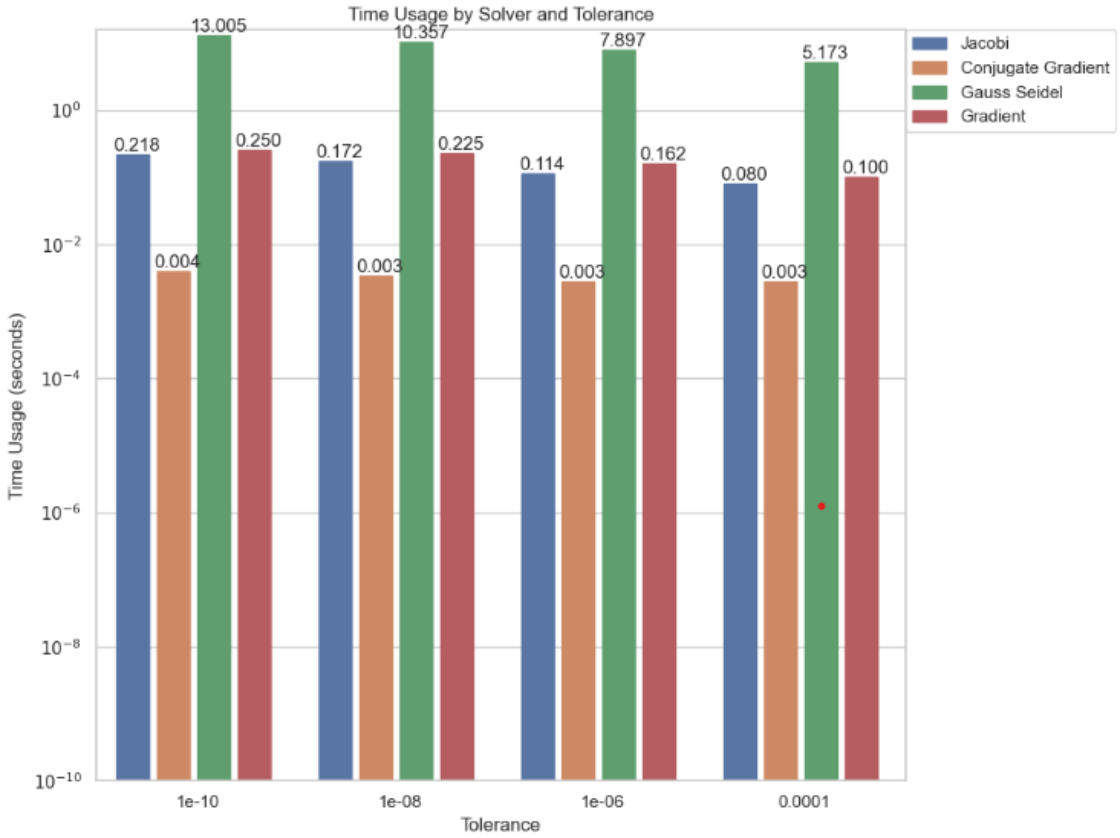

*Figura 13 tempistiche d'esecuzione sulla matrice **vem2***

Considerazioni

Dall'analisi delle tempistiche di esecuzione, è emerso che il tempo necessario per la convergenza di ciascun metodo dipende fortemente dalla tolleranza scelta e dalle proprietà della matrice. In particolare, per matrici come spa1 e spa2, che presentano una densità maggiore e un numero di condizionamento elevato, i metodi più efficienti risultano essere Jacobi e il Gradiente Coniugato. Entrambi i metodi mantengono tempi di esecuzione relativamente stabili al variare della tolleranza, evidenziando un buon equilibrio tra precisione e velocità. Al contrario, il metodo del Gradiente è stato identificato come il meno efficiente in termini di tempo di esecuzione, mostrando un incremento significativo del tempo necessario all'aumentare della tolleranza di precisione.

Per le matrici vem1 e vem2, caratterizzate da una struttura a dominanza diagonale e da un basso indice di sparsità, il metodo del Gradiente Coniugato si è distinto come il solutore più veloce e robusto, seguito dal metodo di Jacobi. Il metodo di Gauss-Seidel, invece, ha mostrato tempi di esecuzione notevolmente più lunghi per queste matrici.

Analisi del Progresso del Residuo in Funzione del Numero di Iterazioni

In questo paragrafo, analizziamo il comportamento dei vari risolutori in relazione all'andamento del residuo durante le iterazioni, fino al raggiungimento della convergenza. Per esaminare in dettaglio le performance, consideriamo il caso in cui la soluzione venga calcolata con una tolleranza molto bassa, pari a 1×10−10. Un aspetto comune a tutti i grafici di questo tipo è che il residuo segue una curva monotona decrescente continua. Ciò si verifica perché, ad ogni iterazione, il vettore soluzione viene aggiornato con un valore sempre più vicino alla soluzione esatta, fino a raggiungere la convergenza entro la tolleranza definita.

Nel grafico di Figura 14 che rappresenta l'andamento del residuo per la matrice **spa1**, si osserva chiaramente che il metodo del Gradiente richiede un numero di iterazioni significativamente maggiore rispetto agli altri risolutori, superando le 20.000 iterazioni. Al contrario, gli altri metodi convergono in meno di 300 iterazioni. Tra questi, il metodo iterativo che raggiunge la soluzione con il minor numero di iterazioni è quello di Gauss-Seidel, seguito dai metodi del Gradiente Coniugato e di Jacobi, che mostrano prestazioni quasi equivalenti. È possibile notare come entro le prime 100 iterazioni il residuo decresce repentinamente e in modo sincrono tra tutti i risolutori, per poi continuare con il medesimo trend unicamente per Gaus-Seidel e, di poco più lentamente, sia per Jacobi e il gradiente coniugato. La linea raffigurante il residuo del metodo

del gradiente invece arresta la sua discesa repentina e arriva a convergenza con un numero di iterazioni maggiore e quindi con una discesa leggera e graduale.

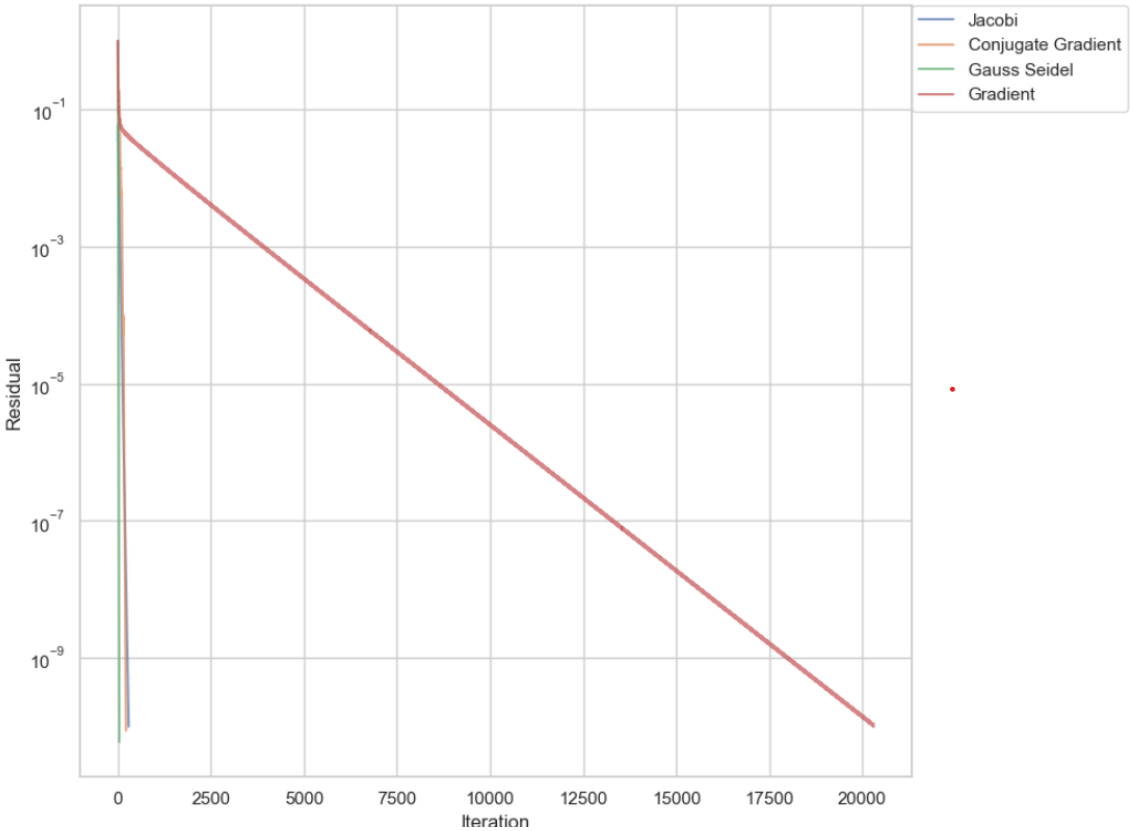

*Figura 14 andamento del residuo rispetto alle iterazioni per la matrice **spa1***

Un discorso simile può essere fatto per la matrice **spa2** rispetto alla matrice spa1. In questo grafico, si osserva che il numero di iterazioni necessarie al metodo di Jacobi per raggiungere la convergenza è significativamente inferiore rispetto a quello del metodo del Gradiente Coniugato. Questo rappresenta un cambiamento rispetto al caso della matrice spa1, dove il Gradiente Coniugato richiedeva un numero minore di iterazioni rispetto al metodo di Jacobi per convergere.

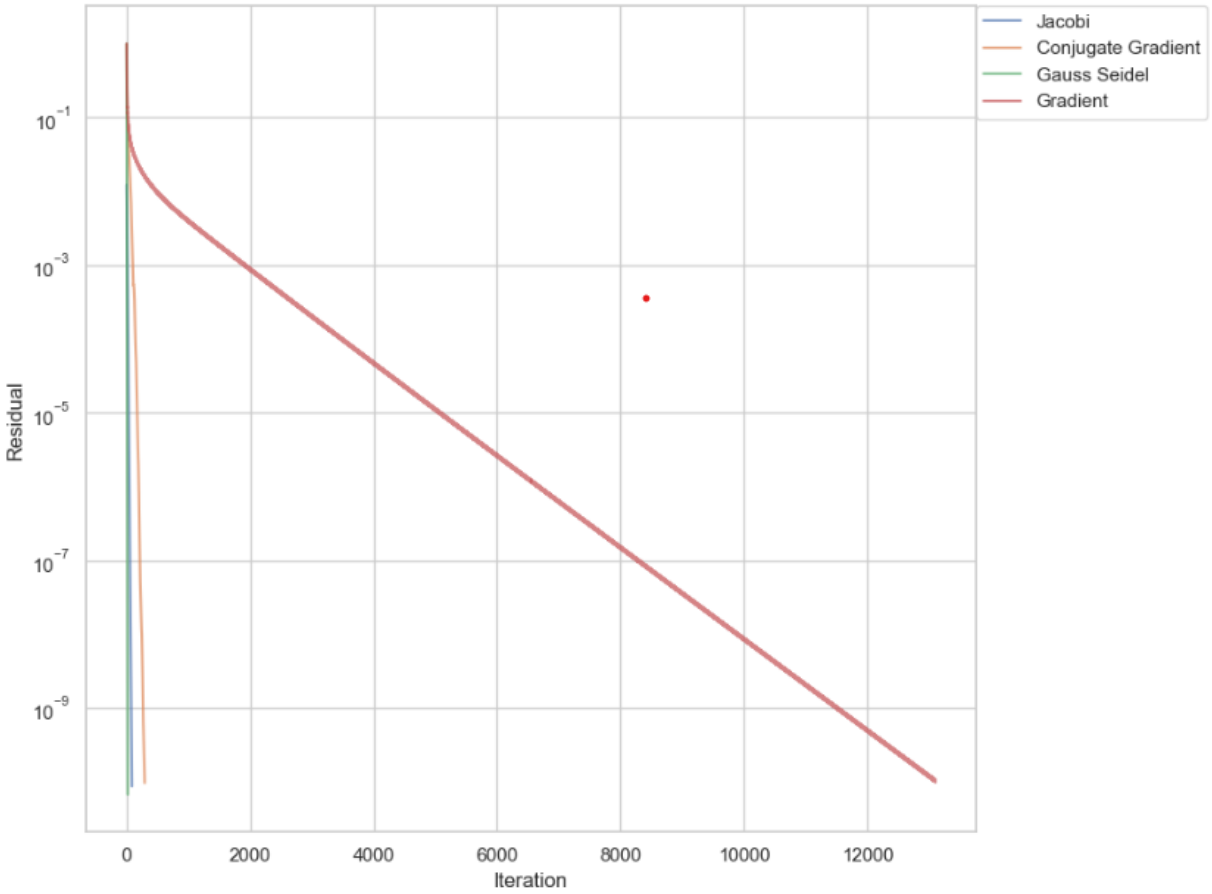

*Figura 15 andamento del residuo rispetto alle iterazioni per la matrice **spa2***

I comportamenti dei residui relativi per le matrici **vem1** (Figura16) e **vem2** (Figura17) si differenziano significativamente da quelli osservati per le matrici spa1 e spa2. In questo caso, l'unico risolutore che mostra una diminuzione rapida e costante del residuo fino alla convergenza è il metodo del Gradiente Coniugato. Questo è coerente con i tempi di esecuzione precedentemente analizzati, che indicano il Gradiente Coniugato come il metodo più efficiente per queste due matrici. I metodi di Gauss-Seidel e del Gradiente presentano un comportamento differente: dopo una breve e rapida discesa del residuo, simile in intensità a quella osservata per le matrici precedenti, la velocità di decrescita diminuisce significativamente con l'aumentare delle iterazioni. Il metodo di Gauss-Seidel si arresta intorno alle 2800 iterazioni, mentre il metodo del Gradiente prosegue oltre le 3700 iterazioni. Il metodo di Jacobi, invece, si conferma come il meno efficiente in termini di numero di iterazioni necessarie per raggiungere la convergenza, superando le 8000 iterazioni. A differenza di quanto osservato per le matrici spa1 e spa2, per i metodi diversi dal gradiente coniugato la curva del residuo ferma prematuramente (al di sotto delle 70 iterazioni) la propria fase di decrescita

'verticale' iniziale per stabilizzarsi e diminuire drasticamente in modo costante la propria intensità di discesa all'aumentare delle interazioni.

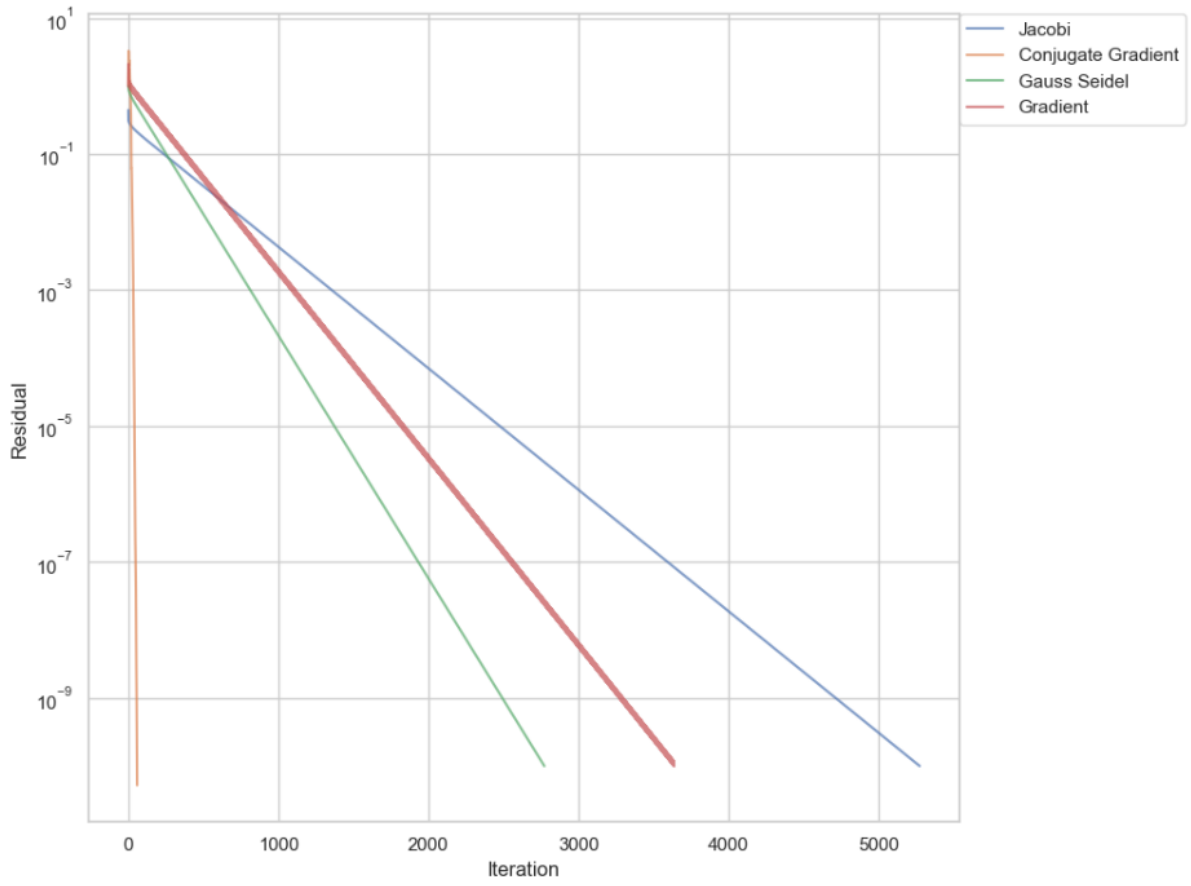

*Figura 16 andamento del residuo rispetto alle iterazioni per la matrice **vem1***

Come è possibile visualizzare in Figura 17, il discorso eseguito sul grafico relativo alla matrice vem1 è del tutto estendibile alla matrice **vem2**. L'unica differenza consiste nel numero di iterazioni richieste per arrivare a convergenza per tutti i metodi tranne il gradiente coniugato. Infatti, per i restanti risolutori, è necessario eseguire un numero maggiore di iterazioni.

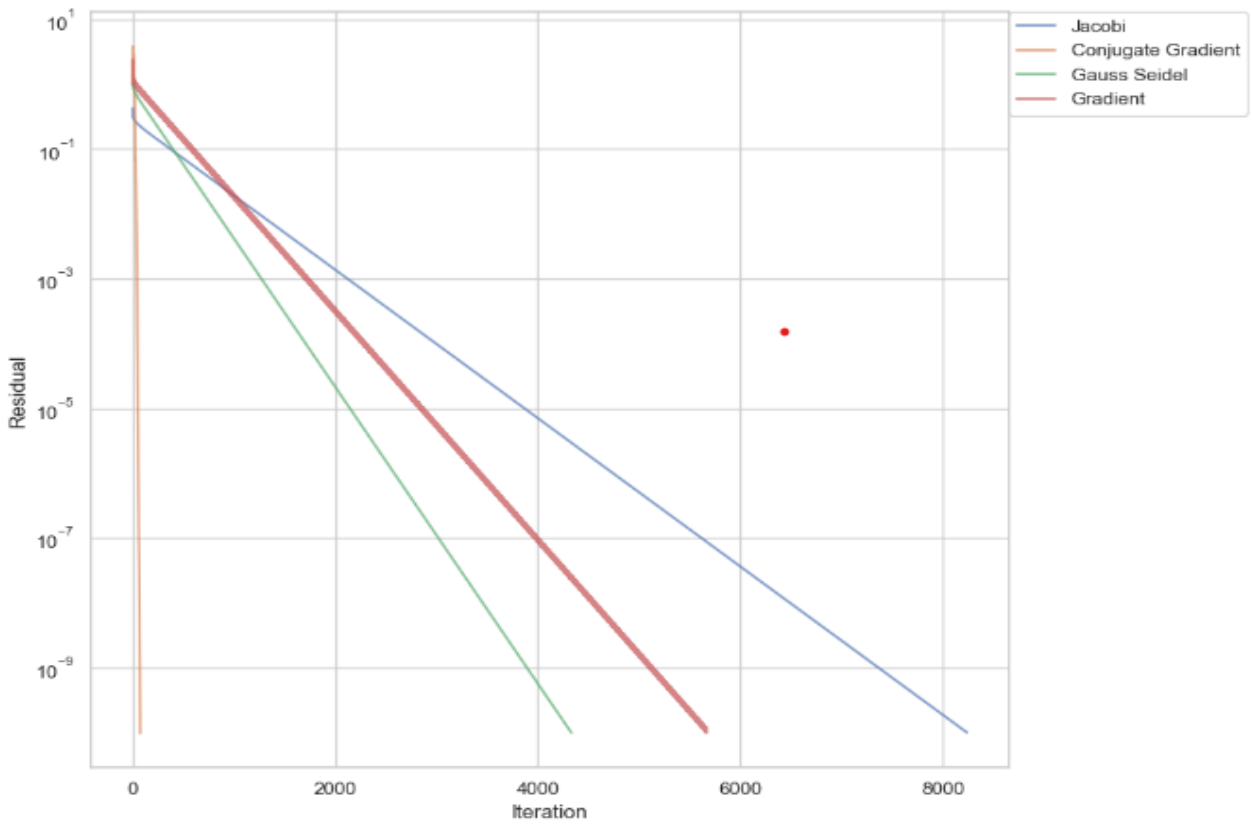

*Figura 17 andamento del residuo rispetto alle iterazioni per la matrice **vem2***

Considerazioni

L'analisi del progresso del residuo in funzione del numero di iterazioni ha rivelato comportamenti distinti per ciascun metodo iterativo. Per le matrici spa1 e spa2, il metodo del Gradiente richiede un numero significativamente maggiore di iterazioni per raggiungere la convergenza rispetto agli altri metodi, superando le 20.000 iterazioni. In questi casi, il metodo di Gauss-Seidel è risultato il più efficiente, raggiungendo rapidamente la convergenza con il minor numero di iterazioni. I metodi di Jacobi e del Gradiente Coniugato hanno mostrato performance simili, convergendo entro un numero relativamente ridotto di iterazioni.

Per le matrici vem1 e vem2, il metodo del Gradiente Coniugato ha nuovamente dimostrato la sua superiorità, convergendo con un numero minimo di iterazioni e con una discesa rapida e costante del residuo. Gli altri metodi hanno mostrato una riduzione iniziale rapida del residuo seguita da una decrescita più lenta e prolungata, con il metodo di Jacobi che ha richiesto il maggior numero di iterazioni per raggiungere la convergenza.

Analisi del residuo finale per le diverse matrici e tolleranze

In questa analisi valutiamo come ciascun metodo gestisce il residuo finale per tolleranze strette e più larghe. L'asse x rappresenta la tolleranza, che indica il criterio di arresto per i solutori iterativi, mentre l'asse y, su scala logaritmica, rappresenta i residui, che misurano l'errore tra la soluzione corrente e quella reale. L'obiettivo è confrontare la precisione e l'efficacia dei metodi in funzione della tolleranza e determinare se uno di essi si distingue in modo significativo rispetto agli altri.

Osservando il grafico relativo alla matrice **spa1 in** Figura 18, si nota che per tolleranze molto strette, come 1×10−10 e 1×10−8, i residui risultano estremamente bassi, prossimi a 1×10−10 o addirittura più bassi. Questo suggerisce che tutti i metodi utilizzati sono in grado di raggiungere soluzioni molto accurate quando la tolleranza è particolarmente rigida. Al contrario, con l'aumento della tolleranza, ad esempio a 1×10−6 e 1×10−4, i residui tendono ad aumentare. Questo poiché una tolleranza più alta permette all'algoritmo di fermarsi prima di raggiungere una soluzione di precisione maggiore, risultando quindi in un residuo più grande. Dal grafico si evince come il metodo di Gaus Seidel sia quello che ottenga un residuo inferiore nelle diverse tolleranze, seguito dal metodo del gradiente coniugato, Jacobi ed infine il gradiente.

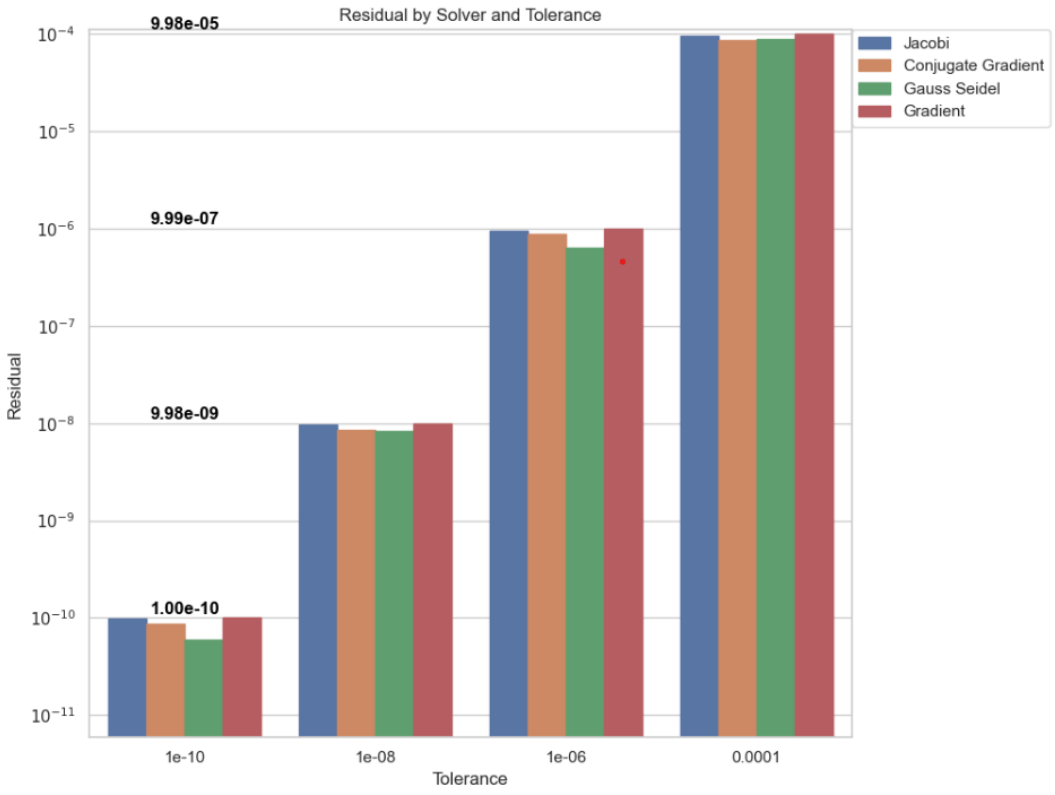

*Figura 18 residuo finale matrice **spa1***

Per la matrice **spa2**, in Figura 19 si osserva un andamento simile a quello della matrice precedente, con il pattern di prestazioni dei risolutori sostanzialmente invariato. Tuttavia, vi è un'inversione tra il metodo di Jacobi e il Gradiente Coniugato, che scambiano la loro posizione nelle classifiche di prestazione. Il metodo di Gauss-Seidel si conferma come il più performante aumentando il divario con gli altri risolutori specialmente per tolleranze diverse da 1x10-6.

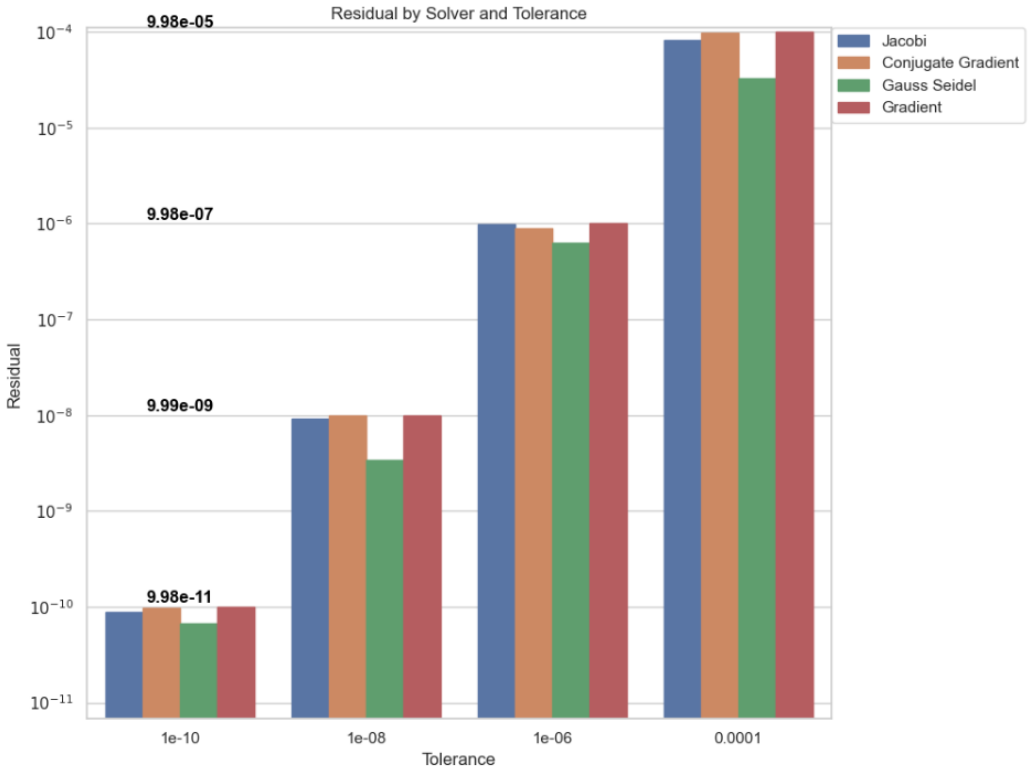

*Figura 19 residuo finale matrice **spa2***

Analizzando i risultati ottenuti sulla matrice **vem1** in Figura 20, emerge un quadro diverso rispetto agli altri casi. In questo scenario, il metodo più efficace risulta essere il Gradiente Coniugato, che offre il residuo più basso per tutte le tolleranze testate. Gli altri metodi mostrano prestazioni simili tra loro, con residui quasi identici.

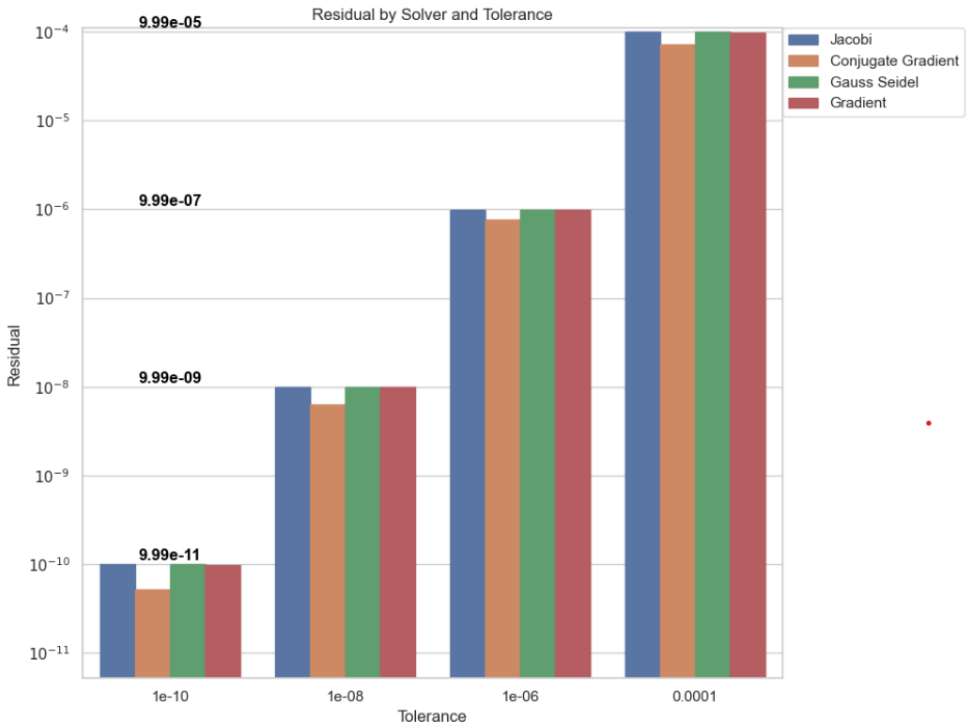

*Figura 20 residuo finale matrice **vem1***

Nella la matrice **vem2** raffigurata in Figura 21, il metodo del gradiente coniugato si conferma il metodo più performante ma è possibile notare come il gap con i residui calcolati dagli altri risolutori sia notevolmente dirotto, soprattutto per la tolleranza più stringente, indicando che non ci sia una sostanziale differenza con le soluzioni ottenute dagli altri solutori.

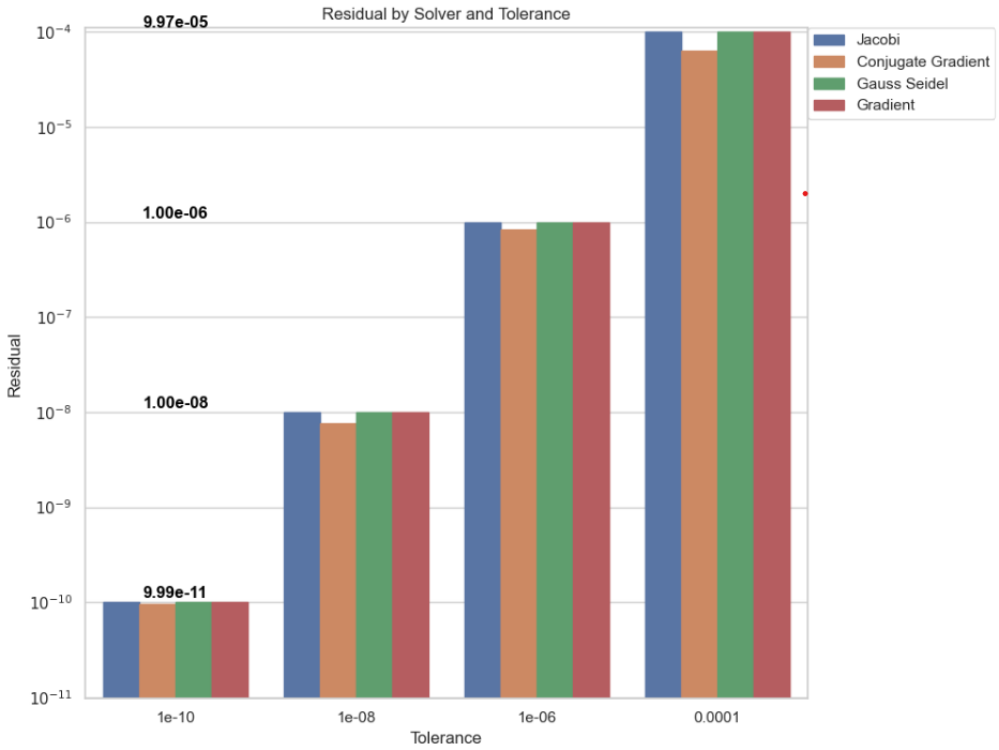

*Figura 21 residuo finale matrice **vem2***

Considerazioni

L'analisi del residuo finale per le diverse matrici e tolleranze ha fornito ulteriori informazioni sull'efficacia di ciascun metodo. In generale, tutti i solutori sono stati in grado di ottenere residui estremamente bassi con tolleranze molto strette, indicando una buona capacità di raggiungere soluzioni accurate. Tuttavia, con l'aumento della tolleranza, le differenze tra i metodi diventano più evidenti. Per le matrici spa1 e spa2, il metodo di Gauss-Seidel ha mostrato un residuo finale inferiore rispetto agli altri solutori, evidenziando una maggiore accuratezza. Per le matrici vem1 e vem2, il Gradiente Coniugato è risultato il metodo più performante, ottenendo residui finali significativamente più bassi rispetto agli altri metodi.

CAPITOLO 4

Conclusioni

I risultati delle analisi evidenziano che non esiste un solutore iterativo universalmente migliore per tutte le tipologie di matrici; l'efficacia di ciascun metodo dipende fortemente dalle caratteristiche strutturali della matrice e dalle condizioni di tolleranza impostate. Tuttavia, il metodo del Gradiente Coniugato si è dimostrato il più versatile, mostrando buone performance su tutte le matrici testate, specialmente su quelle a **dominanza diagonale**. Il metodo di Gauss-Seidel ha mostrato ottime performance in termini di numero di iterazioni e residuo finale ottenuto per **matrici più dense**, mentre i metodi di Jacobi e soprattutto del Gradiente hanno mostrato limitazioni in specifici scenari, quest'ultimo dimostrandosi in termine di performance il peggiore sotto ogni analisi effettuata. Pertanto, nella scelta del metodo iterativo più appropriato per risolvere sistemi lineari sparsi, è fondamentale considerare sia le proprietà delle matrici in gioco sia i requisiti di precisione e velocità richiesti dall'applicazione specifica.