Università degli Studi di Milano Bicocca

DIPARTIMENTO DI INFORMATICA

CORSO DI STUDIO MAGISTRALE IN
INFORMATICA

Sviluppo di una Libreria per la Risoluzione di Sistemi Lineari tramite Metodi Iterativi

Membro:
Samuele Toniazzo
Matricola: 918799

Membro:
Emilio Daverio
Matricola: 918799

Anno accademico:
2023/2024

Indice

Capitolo 1

Informazioni Generali

1.1 Componenti del gruppo:

Il team di lavoro è composto dalle seguenti persone:

- Emilio Daverio (matricola: 918799)
- Samuele Toniazzo (matricola: 918624)

1.2 Obbietivo del progeto:

Lo scopo del progetto è quello di implementare una mini libreria che esegua i seguenti solutori iterativi, limitatamente al caso di matrici simmetriche e definite positive.

1.3 Ambiente di Sviluppo e Lingiuaggio:

Per la realizzazione di questo progetto (sia per la parte 1, sia per la parte 2), abbiamo scelto il linguaggio Python nell'ambiente di sviluppo IntelliJ IDEA. La scelta di Python è stata motivata da diversi fattori, tra cui i suoi notevoli vantaggi nell'ambito dell'elaborazione delle immagini e dello sviluppo di applicazioni desktop. In particolare, Python si distingue per la sua:

- Sintassi semplice e intuitiva, che facilita la scrittura, la lettura e la comprensione del codice, soprattutto quando si affrontano compiti complessi come l'elaborazione delle immagini.
- Natura intuitiva, che rende lo sviluppo del codice più efficiente e accessibile.
- Ampia gamma di librerie specializzate, dedicate all'elaborazione delle immagini e all'analisi dei dati. Queste librerie offrono strumenti potenti e già ottimizzati, evitando la necessità di svilupparli da zero.

Parte I

Risoluzione metodi

Capitolo 2

Introduzione al Progetto

2.1 Introduzione:

Come accennato, dobbiamo implementare una libreria e tale libreria deve essere sviluppata in un linguaggio di programmazione a scelta tra C++, Fortran, Java, Python, ecc., e deve eseguire i seguenti solutori iterativi per matrici simmetriche e definite positive:

1. Metodo di Jacobi
2. Metodo di Gauss-Seidel
3. Metodo del Gradiente
4. Metodo del Gradiente Coniugato

Per la gestione delle strutture dati e delle operazioni elementari tra matrici, è richiesto l'utilizzo di una libreria open-source come Eigen, Armadillo o BLAS/LAPACK. Alternativamente, se il linguaggio di programmazione lo permette, si possono utilizzare vettori e matrici già implementati al suo interno.

2.2 Aspettative:

Quello che ci aspettimo è che il progetto da noi sviluppato soddisfi i seguenti obiettivi chiave:

- **Implementazione Completa dei Metodi Iterativi:**
  La libreria deve essere in grado di operare con ciascuno dei metodi iterativi sopra menzionati. La libreria scelta come base deve fornire solamente la struttura dati di matrici e vettori, senza utilizzare i metodi relativi alla risoluzione dei sistemi lineari già implementati al suo interno.
- **Condizioni di Arresto:**
  I metodi iterativi devono partire da un vettore iniziale nullo e arrestarsi quando la k-esima iterata $x^{(k)}$ soddisfa la condizione:

$$\frac{\|Ax^{(k)} - b\|}{\|b\|} < \text{tol} \tag{2.1}$$

dove tol è una tolleranza assegnata dall'utente. In questo caso $tol = [10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}]$. Inoltre, devono esserci controlli sul numero massimo di iterazioni, arrestando le routine se $k > maxIter$, dove maxIter è un numero elevato a scelta (non inferiore a 20000).

- **Struttura della Libreria:**
  La libreria deve avere un'architettura ben strutturata anziché una sequenza di funzioni indipendenti. Questo approccio modulare facilita la manutenzione e l'estensibilità del codice.

2.3 Metodo Iterativo Stazionario:

Un metodo iterativo stazionario è un tipo di metodo utilizzato per risolvere sistemi di equazioni lineari. Questi metodi generano una successione di approssimazioni per la soluzione del sistema, migliorando iterativamente la stima iniziale. La caratteristica distintiva dei metodi iterativi stazionari è che la stessa operazione (o un insieme di operazioni) viene applicata ripetutamente a ogni iterazione per migliorare l'approssimazione. La convergenza dei metodi iterativi stazionari dipende spesso dalle proprietà della matrice A $Ax = b$.
Quello che è utile sapere è che per risolvere i metodi iterativi si può usare la tecnica dello **slitting**. Ovvero presa una matrice A possiamo suddividerla in due matrici P, N e avremo che:

$$A = N - P \tag{2.2}$$

con N, P due matrici quadrate, simmetriche e definite positive. I metodi iterativi si differenziano dai metodi diretti nei seguenti modi:

Metodi Diretti:

I metodi diretti cercano di risolvere il sistema lineare $Ax = b$ eseguendo una sequenza finita di operazioni che, in assenza di errori di arrotondamento, conducono esattamente alla soluzione. Esempi di metodi diretti includono:

- Eliminazione Gaussiana
- Fattorizzazione LU
- Fattorizzazione Cholesky
- ...

Nei metodi iterativi, bisogna tenere conto dell'errore assoluto e dell'errore relativo.

Erore Assoluto:

L'errore assoluto misura la differenza tra la soluzione approssimata $x^{(k)}$ ottenuta dopo k iterazioni e la soluzione esatta x del sistema lineare. L'errore

assoluto può essere espresso come:

$$EA = ||x^{(k)} - x|| \tag{2.3}$$

dove:

- $x^{(k)}$ è il vettore delle soluzioni approssimate dopo k iterazioni
- x è il vettore delle soluzioni esatte.
- $||.||$ denota una norma vettoriale

Errore Relativo:

L'errore relativo misura l'errore assoluto rispetto alla grandezza della soluzione esatta. È utile per valutare l'errore in proporzione alla grandezza dei valori della soluzione. L'errore relativo può essere espresso come:

$$ER = \frac{\|x^{(k)} - x\|}{\|x\|} \tag{2.4}$$

dove:

- $||x^{(k)} - x||$ è l'errore assoluto.
- $||x||$ è la norma della soluzione esatta

Nei metodi iterativi, l'errore assoluto e relativo vengono spesso utilizzati come criteri di arresto per determinare quando fermare l'iterazione:

$$\frac{\|Ax^{(k)} - b\|}{\|b\|} < \text{tol}$$

Dove:

- $\|Ax^{(k)} - b\|$ è il residuo della soluzione approssimata dopo k iterazioni.
- $\|b\|$ è la norma del vettore dei termini noti.
- tol è una tolleranza assegnata dall'utente.

Figura 2.1: formula criterio d'arresto

2.3.1 Metodo di Jacobi:

Il metodo di Jacobi è un metodo iterativo stazionario utilizzato per risolvere sistemi lineari della forma $Ax = b$ dove A è una matrice n×n (simmetrica e definita positiva) e b è il vettore noto. L'idea di base del metodo di Jacobi è quella di risolvere ogni equazione del sistema per ciascuna delle incognite, esprimendo ogni incognita in termini delle altre incognite e dei termini noti,

e aggiornando ad ogni iterazione i valori calcolati. La formula che descrive questo metodo è:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) \tag{2.5}$$

Vantaggio:

Semplice da implementare e può essere parallelizzato. Quello che possiamo fare è partendo dalla matrice A possimo creare le due matrici P, N nel seguente modo:

$$A := \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix},$$

(a) mat_A_jacobi

$$P := \begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{bmatrix}.$$

(b) mat_P_jacobi

$$N := \begin{bmatrix} 0 & -a_{1,2} & \cdots & -a_{1,n} \\ -a_{2,1} & 0 & \cdots & -a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n,1} & -a_{n,2} & \cdots & 0 \end{bmatrix}.$$

(c) mat_N_jacobi

Figura 2.2: Matrici Jacobi

Svantaggio:

Può richiedere un gran numero di iterazioni per convergere se la matrice non è ben condizionata.

2.3.2 Metodo di Gauss-Seidel:

Il metodo di Gauss-Seidel è una variante del metodo di Jacobi e utilizza le informazioni più recenti disponibili per calcolare i nuovi valori di x_i. È simile al metodo di Jacobi, ma differisce nella formula di aggiornamento delle soluzioni. La formula di aggiornamento per il metodo di Jacobi è:

$$x^{(k+1)} = x^{(k)} + P^{-1} r^{(k)} \qquad \text{(Formula aggiornamento Jacobi)}$$

Nel metodo di Gauss-Seidel, la formula di aggiornamento utilizza i valori più recenti calcolati durante l'iterazione corrente:

$$x^{(k+1)} = x^{(k)} + y \qquad \text{(Formula aggiornamento Gauss)}$$

Vantaggio:

Spesso converge più rapidamente del metodo di Jacobi.

Svantaggio:

Non facilmente parallelizzabile come Jacobi.

2.3.3 Metodo del Gradiente:

Il metodo del gradiente fa parte dei metodi iterativi non stazionari. La principale differenza rispetto ai metodi stazionari è che, nei metodi stazionari, la matrice utilizzata per aggiornare la soluzione rimane costante ad ogni iterazione, e il parametro α rimane costante. Inoltre, la convergenza di questi metodi è tipicamente lenta e fortemente dipendente dalle proprietà della matrice. Nei metodi non stazionari, invece, la formula per aggiornare la soluzione può cambiare ad ogni iterazione (quello che cambia è il parametro α). La convergenza di questi metodi è generalmente più rapida rispetto ai metodi stazionari e può essere meno sensibile alle proprietà della matrice del sistema lineare. Il metodo del gradiente si basa sull'interpretazione della risoluzione di un sistema lineare come la ricerca del minimo di una funzione quadratica. La funzione da minimizzare è:

$$\varphi(y) = \frac{1}{2} y^T A y - b^T y \tag{2.6}$$

L'aggiornamento per il vettore x è dato da:

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} \tag{2.7}$$

dove

- $r^{(k)} = b - Ax^{(k)}$ è il residuo e α_k è calcolato come:
- $\alpha_k = \frac{r^{(k)T} r^{(k)}}{r^{(k)T} A r^{(k)}}$

Vantaggio:

Convergente per matrici simmetriche e definite positive.

Svantaggio:

Può presentare convergenza lenta se il numero di condizionamento della matrice è alto.

2.3.4 Metodo del Gradiente Coniugato:

Il metodo del gradiente coniugato è un miglioramento del metodo del gradiente, che evita la convergenza a zig-zag e garantisce la convergenza in un numero finito di passi per matrici simmetriche e definite positive. Le iterazioni del metodo del gradiente coniugato si basano sulle seguenti formule:

1. Residuo iniziale e direzione iniziale:

$$r^{(0)} = b - Ax^{(0)}, \quad p^{(0)} = r^{(0)} \tag{2.8}$$

2. Aggiornamento della soluzione:

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)} \tag{2.9}$$

3. Calcolo di α_k:

$$\alpha_k = \frac{r^{(k)T} r^{(k)}}{p^{(k)T} A p^{(k)}} \tag{2.10}$$

4. Aggiornamento del residuo:

$$r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)} \tag{2.11}$$

5. Calcolo di β_k:

$$\beta_k = \frac{r^{(k+1)T} r^{(k+1)}}{r^{(k)T} r^{(k)}} \tag{2.12}$$

6. Aggiornamento della direzione:

$$p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)} \tag{2.13}$$

Vantaggio:

Convergente in un numero finito di passi per matrici simmetriche e definite positive.

Svantaggio:

Ogni iterazione è più costosa computazionalmente rispetto ai metodi di Jacobi e Gauss-Seidel.

Capitolo 3

Descrizione Codice

3.1 Repository del Codice:

Il codice del progetto che stato sviluppato è disponibile si gitHub al seguente link: Link del Progetto 1 Bis. Nel repository Git nella cartella "idea" si trovano le due classe Support-py e main.py le quali servono per poter eseguire il codice.

3.2 Descrizione del Codice:

Il nostro progetto per la parte 1 è stato sviluppato nelle seguenti classi:

- support.py o LinearSolversSparse
- main.py

3.3 Classe support / LinearSolversSparse:

La classe LinearSolversSparse è progettata per risolvere sistemi di equazioni lineari sparse utilizzando diversi metodi iterativi. Questa classe è particolarmente utile quando si lavora con matrici di grandi dimensioni e sparse, dove i metodi diretti potrebbero non essere efficienti in termini di tempo e memoria. All'interno del costruttore della classe:

```
def __init__(self, A, b, tol=1e-6, max_iter=20000):
    self.A = csr_matrix(A)
    self.b = b
    self.tol = tol
    self.max_iter = max_iter
    self.x = np.zeros_like(b)
```

è inizializzata una matrice A e un vettore b, che rappresentano il sistema lineare $Ax = b$. Include anche la tolleranza tol per la convergenza e il numero massimo di iterazioni max_iter. La soluzione iniziale x viene impostata come un vettore di zeri con la stessa dimensione di b. Abbiamo deciso di rappresentare la matrice A come una matrice sparsa di tipo CSR (Compressed Sparse Row) per diversi vantaggi specifici nella gestione di matrici sparse, soprattutto in ambito computazionale. Le matrici sparse contengono principalmente

zeri, con solo pochi elementi non nulli. La rappresentazione CSR permette di memorizzare ed elaborare queste matrici in modo più efficiente. Inoltre, la libreria SciPy offre un supporto robusto e ottimizzato per le matrici sparse in formato CSR, rendendo le operazioni algebriche su queste matrici più veloci e meno dispendiose in termini di memoria. La classe LinearSolversSparse offre quattro metodi principali per risolvere il sistema lineare: Jacobi, Gauss-Seidel, Gradiente e Gradiente Coniugato.

3.3.1 Metodo jacobi(self):

Il primo metodo, jacobi, implementa l'algoritmo di Jacobi per la risoluzione del sistema lineare.

```
def jacobi(self):
    start_time = time.time()
    D_inv = 1.0 / self.A.diagonal()
    R = self.A - csr_matrix(np.diag(self.A.diagonal()))
    for k in range(self.max_iter):
        x_new = D_inv * (self.b - R.dot(self.x))
        if norm(self.A.dot(x_new) - self.b) / norm(self.b) <
        ↪  self.tol:
            end_time = time.time()
            exec_time = end_time - start_time
            print(f"Jacobi converged in {k+1} iterations and
            ↪  {exec_time:.2f} seconds")
            return x_new, k + 1, exec_time
        self.x = x_new
    end_time = time.time()
    exec_time = end_time - start_time
    print(f"Jacobi did not converge in {self.max_iter}
    ↪  iterations and {exec_time:.2f} seconds")
    return self.x, self.max_iter, exec_time
```

In questo metodo, si calcola l'inversa della diagonale della matrice A e si sottrae la diagonale da A per ottenere la matrice R. L'algoritmo itera poi aggiornando la soluzione x utilizzando la formula di Jacobi. Ad ogni iterazione, viene calcolato un nuovo vettore x e si verifica la convergenza confrontando la norma del residuo normalizzato con la tolleranza specificata. Se la soluzione converge entro il numero massimo di iterazioni, il metodo restituisce la soluzione trovata, il numero di iterazioni e il tempo di esecuzione. In caso contrario, il metodo indica che non è riuscito a convergere nel numero massimo di iterazioni specificato.

3.3.2 Metodo gauss_seidel(self):

Il secondo metodo, gauss_seidel, implementa l'algoritmo di Gauss-Seidel.

```
def gauss_seidel(self):
    start_time = time.time()
    L = csr_matrix(np.tril(self.A.toarray()))
    U = self.A - L
    D = self.A.diagonal()
    for k in range(self.max_iter):
        x_new = np.copy(self.x)  # Copia l'iterazione corrente
        for i in range(self.A.shape[0]):
            sum1 = L[i, :i].dot(x_new[:i])
            sum2 = U[i, i+1:].dot(self.x[i+1:])
            x_new[i] = (self.b[i] - sum1 - sum2) / D[i]
        if norm(self.A.dot(x_new) - self.b) / norm(self.b) <
        ↪  self.tol:
            end_time = time.time()
            exec_time = end_time - start_time
            print(f"Gauss-Seidel converged in {k+1} iterations
            ↪  and {exec_time:.2f} seconds")
            return x_new, k + 1, exec_time
        self.x = x_new
    end_time = time.time()
    exec_time = end_time - start_time
    print(f"Gauss-Seidel did not converge in {self.max_iter}
    ↪  iterations and {exec_time:.2f} seconds")
    return self.x, self.max_iter, exec_time
```

Questo metodo scompone la matrice A in una parte inferiore L e una parte superiore U. Durante ogni iterazione, aggiorna la soluzione x utilizzando la formula di Gauss-Seidel. Viene calcolato un nuovo vettore x e, se la norma del residuo normalizzato è inferiore alla tolleranza, il metodo converge e restituisce la soluzione, il numero di iterazioni e il tempo di esecuzione. Se il metodo non converge entro il numero massimo di iterazioni, viene indicato il fallimento della convergenza.

3.3.3 Metodo gradient(self):

Il metodo gradient implementa l'algoritmo del Gradiente.

```
def gradient(self):
    start_time = time.time()
    r = self.b - self.A.dot(self.x)
    for k in range(self.max_iter):
        alpha = (r @ r) / (r @ self.A.dot(r))
        x_new = self.x + alpha * r
        if norm(self.A.dot(x_new) - self.b) / norm(self.b) <
        ↪  self.tol:
            end_time = time.time()
```

```
            exec_time = end_time - start_time
            print(f"Gradient Descent converged in {k+1}
             ↪  iterations and {exec_time:.2f} seconds")
            return x_new, k + 1, exec_time
        r = self.b - self.A.dot(x_new)
        self.x = x_new
    end_time = time.time()
    exec_time = end_time - start_time
    print(f"Gradient Descent did not converge in
     ↪  {self.max_iter} iterations and {exec_time:.2f}
     ↪  seconds")
    return self.x, self.max_iter, exec_time
```

In questo metodo, il residuo iniziale viene calcolato come la differenza tra b e il prodotto di A per x. Ad ogni iterazione, il metodo aggiorna la soluzione x utilizzando una direzione calcolata dal residuo. Il passo α viene determinato per minimizzare l'errore, e la nuova soluzione viene aggiornata di conseguenza. La convergenza è verificata confrontando la norma del residuo normalizzato con la tolleranza. Se la soluzione converge entro il numero massimo di iterazioni, il metodo restituisce la soluzione, il numero di iterazioni e il tempo di esecuzione. Se il metodo non converge, viene indicato il fallimento della convergenza.

3.3.4 Metodo conjugate_gradient(self):

Il metodo conjugate_gradient implementa l'algoritmo del Gradiente Coniugato.

```
def conjugate_gradient(self):
    start_time = time.time()
    r = self.b - self.A.dot(self.x)
    p = r
    for k in range(self.max_iter):
        r_dot_r = r @ r
        alpha = r_dot_r / (p @ self.A.dot(p))
        x_new = self.x + alpha * p
        r_new = r - alpha * self.A.dot(p)
        if norm(self.A.dot(x_new) - self.b) / norm(self.b) <
         ↪  self.tol:
            end_time = time.time()
            exec_time = end_time - start_time
            print(f"Conjugate Gradient converged in {k+1}
             ↪  iterations and {exec_time:.2f} seconds")
            return x_new, k + 1, exec_time
        beta = (r_new @ r_new) / r_dot_r
        p = r_new + beta * p
```

```
            r = r_new
            self.x = x_new
        end_time = time.time()
        exec_time = end_time - start_time
        print(f"Conjugate Gradient did not converge in
        ↪  {self.max_iter} iterations and {exec_time:.2f}
        ↪  seconds")
        return self.x, self.max_iter, exec_time
```

Inizialmente, viene calcolato il residuo e la direzione iniziale. Ad ogni iterazione, la soluzione x viene aggiornata utilizzando la direzione coniugata, e il passo α viene calcolato. Dopo ogni aggiornamento, il residuo viene aggiornato e viene calcolato il parametro β per aggiornare la direzione. La convergenza viene verificata confrontando la norma del residuo normalizzato con la tolleranza. Se la soluzione converge entro il numero massimo di iterazioni, il metodo restituisce la soluzione, il numero di iterazioni e il tempo di esecuzione. Se il metodo non converge, viene indicato il fallimento della convergenza.

3.3.5 Metodo load_matrix(file_path):

Infine, la funzione esterna load_matrix è utilizzata per caricare una matrice da un file.

```
def load_matrix(file_path):
try:
    # Legge la matrice dal file e la converte in formato
    ↪  csr_matrix
    matrix = scipy.io.mmread(file_path)
    return matrix.tocsr()
except ValueError as e:
    print(f"Error reading {file_path}: {e}")
    return None
```

Questa funzione legge il file della matrice e la converte in una matrice sparsa di tipo CSR. Se si verifica un errore durante la lettura del file, viene restituito un messaggio di errore.

3.4 Classe main:

La classe main fornisce una struttura per testare e confrontare diversi algoritmi di risoluzione di sistemi lineari su matrici sparse la funzione test_solver e utilizza la funzione plot_results per creare il grafico dove viene confrontato per ogni metodo il numero di iterazioni e il tempo impegato a risolvere quella matrice sparsa. Utilizza le librerie numpy e matplotlib per la gestione dei dati e la visualizzazione dei risultati e importa la classe support per utilizzare i me-

todi all'interno della classe LinearSolversSparse. in questa classe utilizzaimo le seguente librerie per:

- **numpy:** Utilizzato per operazioni numeriche, come la generazione di vettori casuali e la gestione delle matrici.
- **matplotlib.pyplot:** Utilizzato per la creazione di grafici che mostrano i risultati delle iterazioni e i tempi di esecuzione degli algoritmi di risoluzione.
- **support.LinearSolversSparse:** Contiene l'implementazione degli algoritmi di risoluzione di sistemi lineari sparsi.
- **load_matrix:** Utilizzata per caricare matrici sparse da file.

3.4.1 Metodo test_solver(matrix_file, tol_values):

La funzione test_solver è progettata per eseguire i test sui diversi algoritmi di risoluzione per una data matrice.

```
def test_solver(matrix_file, tol_values):
A = load_matrix(matrix_file)
if A is None:
    return {}
b = np.random.rand(A.shape[0])  # Vettore dei termini noti
 ↪  casuale
results = {}
iterazione = 1  # Inizializza la variabile iterazione

for tol in tol_values:
    print(f"Entrato nel for di test solver, iterazione
     ↪  {iterazione}\n")
    print(f"INIZIO iterazione numero {iterazione}\n")

    print("Inizio Jacobi")
    solver = LinearSolversSparse(A, b, tol=tol)
    results[f'Jacobi_tol_{tol}'] = solver.jacobi()
    print("Fine Jacobi \n")

    print("Inizio Gradiente")
    solver = LinearSolversSparse(A, b, tol=tol)
    results[f'Gradient_tol_{tol}'] = solver.gradient()
    print("Fine Gradiente \n")

    print("Inizio Gradiente Coniugato")
    solver = LinearSolversSparse(A, b, tol=tol)
    results[f'Conjugate_Gradient_tol_{tol}'] =
     ↪  solver.conjugate_gradient()
```

```
    print("Fine Gradiente Coniugato \n")

    print("Inizio Gauss-Seidel")
    solver = LinearSolversSparse(A, b, tol=tol)
    results[f'Gauss_Seidel_tol_{tol}'] = solver.gauss_seidel()
    print("Fine Gauss-Seidel \n")

    print(f"FINE iterazione numero {iterazione}\n")
    iterazione += 1  # Incrementa la variabile iterazione

return results
```

La funzione prende in ingresso il percorso del file della matrice (matrix_file) e un elenco di valori di tolleranza (tol_values). All'inizio, la funzione carica la matrice dal file specificato utilizzando la funzione load_matrix. Se la matrice non può essere caricata correttamente, la funzione ritorna un dizionario vuoto.
Successivamente, viene generato un vettore dei termini noti casuale b utilizzando np.random.rand, con una dimensione corrispondente al numero di righe della matrice A. Un dizionario results viene inizialmente creato per memorizzare i risultati dei test. La variabile iterazione viene utilizzata per tracciare il numero di iterazioni nel ciclo for.
All'interno del ciclo, per ciascun valore di tolleranza specificato, la funzione esegue i seguenti passaggi: inizia una nuova istanza della classe LinearSolversSparse con la matrice A, il vettore b e la tolleranza corrente tol. Successivamente, esegue i quattro metodi di risoluzione (jacobi, gradient, conjugate_gradient, gauss_seidel) e memorizza i risultati nel dizionario results con una chiave che include il nome del metodo e il valore di tolleranza. Dopo aver eseguito tutti i metodi per una data tolleranza, la funzione incrementa la variabile iterazione e continua con la successiva tolleranza.

3.4.2 Metodo plot_results:

La funzione plot_results è utilizzata per creare grafici che visualizzano i risultati delle iterazioni e i tempi di esecuzione per diversi metodi di risoluzione e valori di tolleranza.

```
def plot_results(results, matrix_file):
methods = ['Jacobi', 'Gauss_Seidel', 'Gradient',
 ↪  'Conjugate_Gradient']
tol_values = [1e-4, 1e-6, 1e-8, 1e-10]

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 12))

for method in methods:
    iterations = []
```

```
    execution_times = []
    for tol in tol_values:
        key = f'{method}_tol_{tol}'
        if key in results:
            solution, iters, exec_time = results[key]
            iterations.append(iters)
            execution_times.append(exec_time)
        else:
            iterations.append(np.nan)
            execution_times.append(np.nan)

    # Filtra i valori non positivi
    positive_tol_values = [tol for tol, iter in zip(tol_values,
    ↪  iterations) if iter > 0]
    positive_iterations = [iter for iter in iterations if iter
    ↪  > 0]
    positive_execution_times = [time for time, iter in
    ↪  zip(execution_times, iterations) if iter > 0]

    if positive_iterations:
        ax1.plot(positive_tol_values, positive_iterations,
        ↪  marker='o', label=method)
        ax2.plot(positive_tol_values, positive_execution_times,
        ↪  marker='o', label=method)

ax1.set_xscale('log')
ax1.set_yscale('log')
ax1.set_xlabel('Tolerance')
ax1.set_ylabel('Iterations')
ax1.set_title(f'Iterations vs. Tolerance for {matrix_file}')
if ax1.has_data():
    ax1.legend()
ax1.grid(True)

ax2.set_xscale('log')
ax2.set_yscale('log')
ax2.set_xlabel('Tolerance')
ax2.set_ylabel('Execution Time (seconds)')
ax2.set_title(f'Execution Time vs. Tolerance for
↪  {matrix_file}')
if ax2.has_data():
    ax2.legend()
ax2.grid(True)

plt.tight_layout()
```

```
plt.savefig(f'{matrix_file}_convergence.png')
plt.show()
```

Questa funzione prende in ingresso il dizionario dei risultati (results) e il nome del file della matrice (matrix_file). All'inizio, definisce un elenco di metodi di risoluzione (methods) e un elenco di valori di tolleranza (tol_values). Crea un layout di grafici con due subplot utilizzando plt.subplots, uno per le iterazioni e uno per i tempi di esecuzione.
Per ciascun metodo di risoluzione, la funzione estrae i dati dai risultati e li filtra per includere solo i valori positivi. Utilizza ax1.plot e ax2.plot per tracciare i dati filtrati sui rispettivi subplot, configurando le scale dei grafici, le etichette degli assi, i titoli e le legende. Infine, la funzione salva il grafico come file immagine e lo mostra a schermo utilizzando plt.savefig e plt.show.

3.4.3 Sezione main:

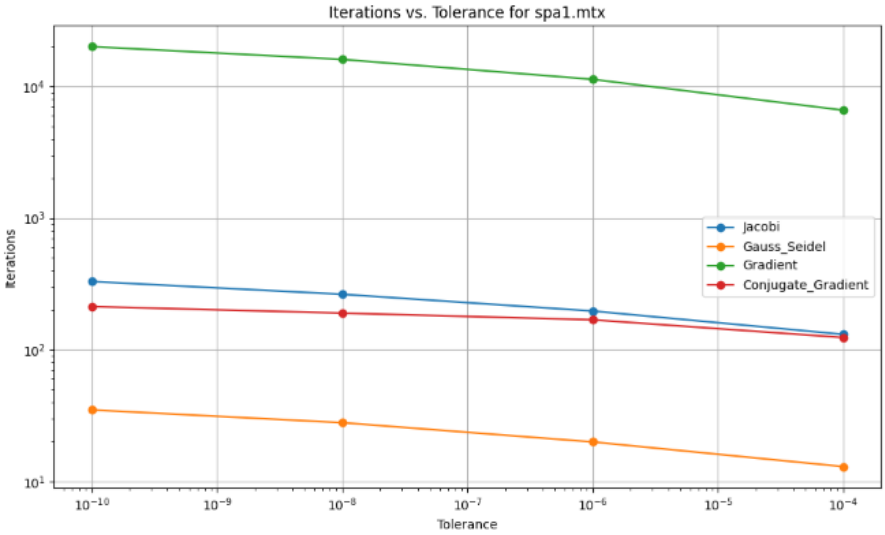
La sezione ___main___ gestisce l'esecuzione principale del programma. All'inizio, definisce i file di matrici da processare (matrix_files), che sono file in formato Matrix Market (estensione .mtx). Questi file contengono matrici sparse utilizzate nei test degli algoritmi di risoluzione. Inoltre, definisce i valori di tolleranza (tol_values) che saranno utilizzati per determinare la convergenza degli algoritmi.

Per ogni file di matrice, il programma esegue i test sui solver chiamando la funzione test_solver, che esegue i quattro metodi di risoluzione (Jacobi, Gradiente, Gradiente Coniugato, Gauss-Seidel) per ciascun valore di tolleranza. Dopo aver eseguito i test, i risultati vengono stampati. Infine, la funzione plot_results viene chiamata per visualizzare i risultati in grafici, permettendo di confrontare il numero di iterazioni e i tempi di esecuzione per ogni metodo di risoluzione e tolleranza.

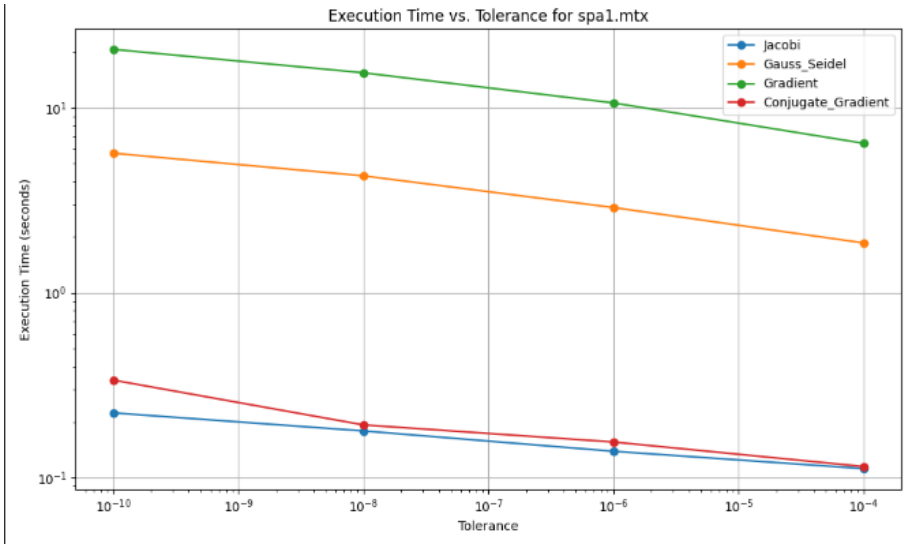
3.5 Risultati:

La sezione dei risultati del progetto include l'analisi e il confronto dei diversi metodi iterativi per la risoluzione di sistemi lineari sparsi. Questa analisi è presentata attraverso grafici che mostrano il numero di iterazioni e il tempo di esecuzione necessari per raggiungere la convergenza per vari valori di tolleranza. I grafici sono stati generati per ciascun file di matrice in formato Matrix Market (ad esempio, spa1.mtx, spa2.mtx, vem1.mtx, vem2.mtx).

3.5.1 Descriozione grafico spa1.mtx:



(a) iteration spa1



(b) time spa1

Figura 3.1: confronto spa1

Il primo subplot 3.1a mostra il numero di iterazioni necessarie per raggiungere la convergenza in funzione della tolleranza. L'asse delle x rappresenta la tolleranza su una scala logaritmica, mentre l'asse delle y rappresenta il numero di iterazioni, anch'esso su una scala logaritmica. Il comportamento dei metodi è il seguente:

- Gradiente: richiede un numero relativamente alto di iterazioni rispetto agli altri metodi. Sebbene il numero di iterazioni diminuisca con una tolleranza meno stringente, rimane comunque elevato.
- Jacobi e Gradiente Coniugato: rrichiedono un numero di iterazioni inferiore rispetto al gradiente, ma non sono i più efficienti.

- Gauss-Seidel: è il più efficiente in termini di numero di iterazioni necessarie per raggiungere la convergenza, richiedendo significativamente meno iterazioni rispetto agli altri metodi, specialmente per tolleranze più stringenti.

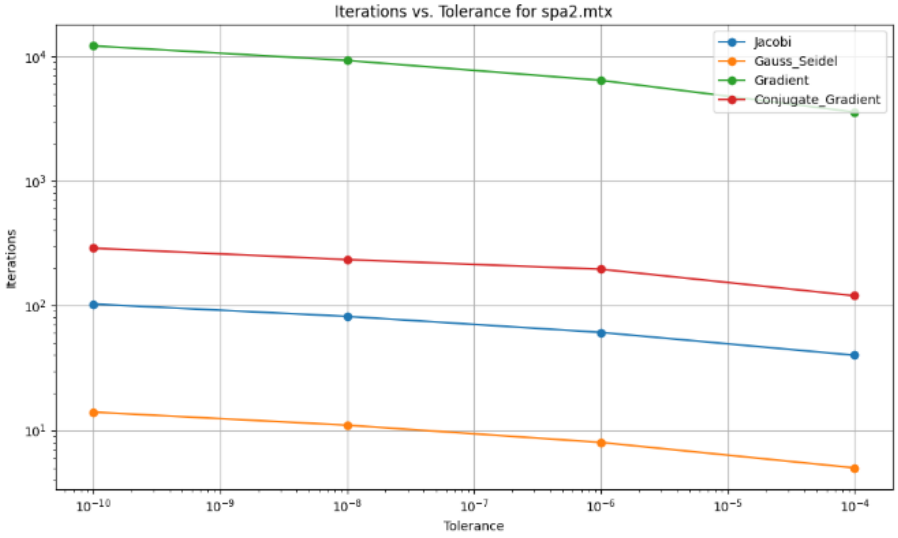
Mentre il secondo subplot 3.1b mostra il tempo di esecuzione necessario per raggiungere la convergenza in funzione della tolleranza. L'asse delle x rappresenta la tolleranza su una scala logaritmica, mentre l'asse delle y rappresenta il tempo di esecuzione in secondi, anch'esso su una scala logaritmica. Possiamo osservare che:

- Jacobi e Gradiente Coniugato: richiedono un tempo relativamente breve, risultando migliori rispetto agli altri metodi.
- Gradiente e Gauss-Sidel: questi due metodi richiedono un tempo elevato per raggiungere la convergenza.

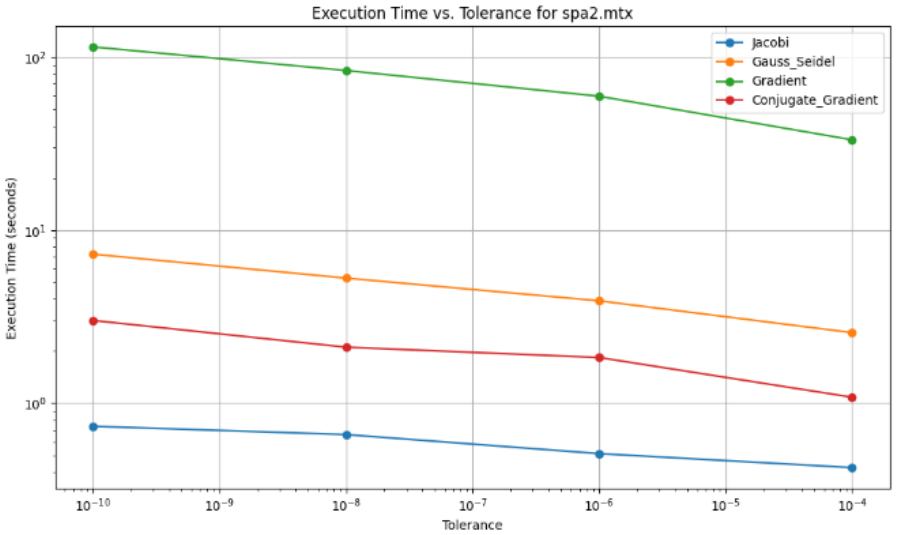
Interpretazione dei Risultati:

Dai grafici, si può osservare che la scelta ottimale per risolvere la matrice spa1 è rappresentata dai metodi di Jacobi o Gradiente Coniugato, poiché, sebbene richiedano un numero medio di operazioni, raggiungono la convergenza più rapidamente.

3.5.2 Descriozione grafico spa2.mtx:



(a) iteration spa2



(b) time spa2

Figura 3.2: confronto spa2

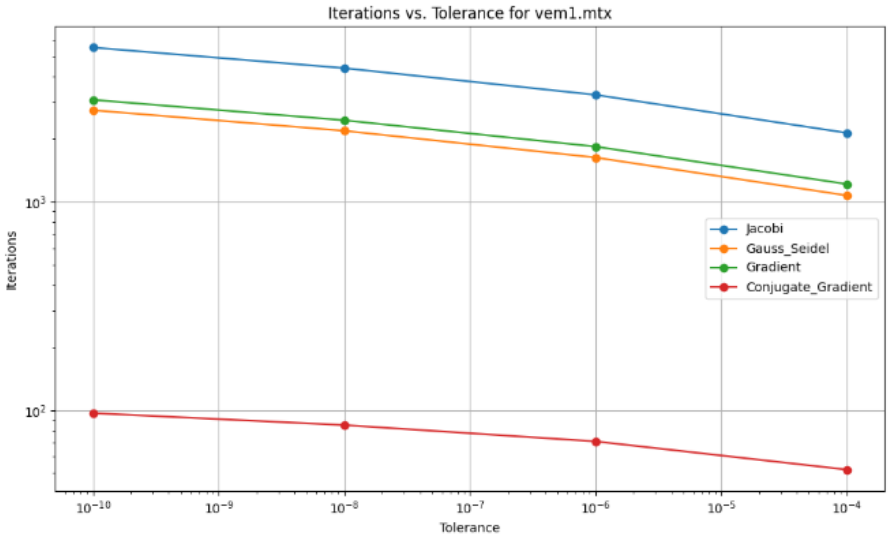
Dalla figura 3.2a possiamo notare che il metodo del gradiente, come nel caso di spa1, è il peggiore in quanto richiede un numero elevato di iterazioni. Al contrario, il metodo Gauss-Seidel rimane il migliore in termini di numero di iterazioni necessarie. Osservando il secondo grafico 3.2b, la situazione è leggermente diversa: il gradiente continua ad essere il peggiore, mentre il metodo Jacobi risulta essere il migliore. Per gli altri due metodi, il tempo di esecuzione è simile, con il gradiente coniugato che impiega leggermente meno tempo rispetto al Gauss-Seidel.

Interpretazione dei Risultati:

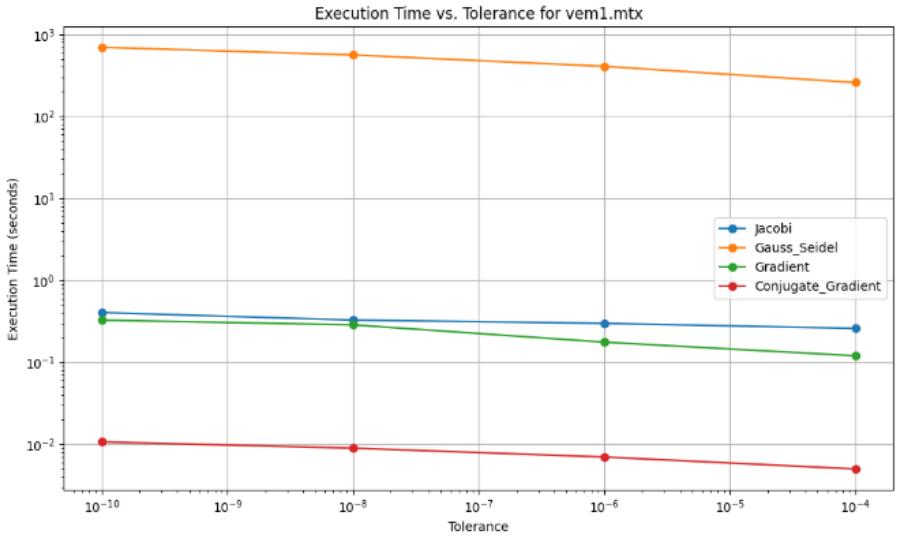
Dai risultati ottenuti per la matrice spa2, possiamo affermare che il metodo Jacobi è il migliore, poiché richiede un numero non eccessivo di iterazioni e

un tempo molto breve per raggiungere la convergenza.

3.5.3 Descrizione grafico vem1.mtx:



(a) iteration vem1

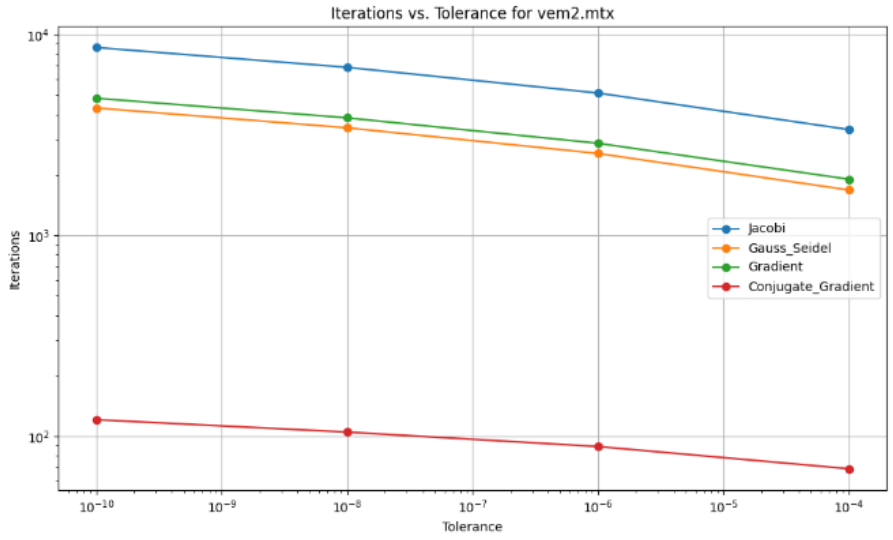


(b) time vem1

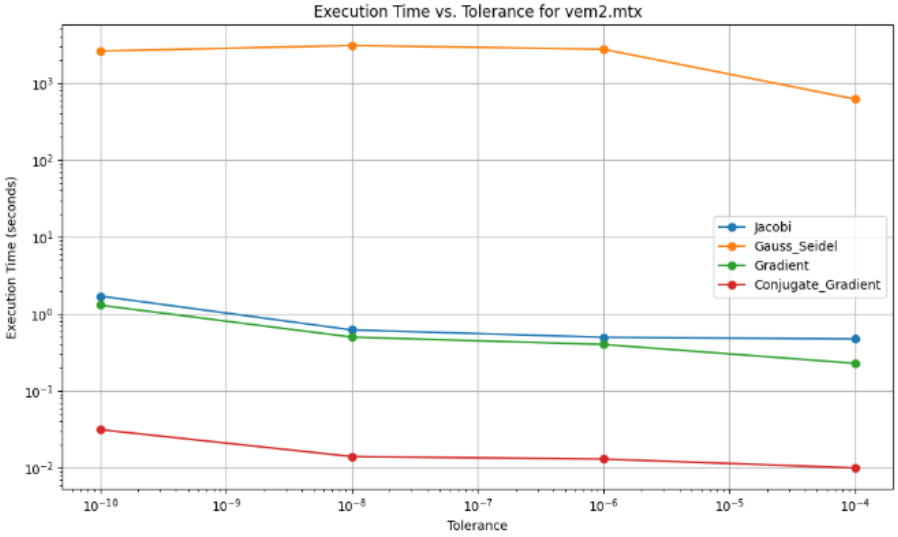
Figura 3.3: confronto vem1

Per quanto riguarda l'esecuzione del file vem1, ciò che colpisce immediatamente è l'enorme distacco tra la funzione del gradiente coniugato e le altre tre funzioni, come mostrato nell'immagine 3.3a. Infatti, il gradiente coniugato richiede un numero di iterazioni molto inferiore rispetto agli altri tre metodi, che necessitano di un numero elevato di iterazioni. Per quanto riguarda il tempo di esecuzione, la situazione è leggermente diversa, come si può notare dall'immagine 3.3b. Il gradiente coniugato mantiene comunque il tempo di esecuzione migliore. Anche i tempi di esecuzione del metodo di Jacobi e del gradiente sono accettabili, mentre il metodo Gauss-Seidel rimane il peggiore, con un tempo di esecuzione elevato.

3.5.4 Descrizione grafico vem2.mtx



(a) iteration vem2



(b) time vem2

Figura 3.4: confronto vem2

Per quanto riguarda la risoluzione di questa matrice, i risultati sono simili a quelli ottenuti per la matrice vem1.mtx (figura 3.3). Come si può osservare dall'immagine 3.4a, il metodo del gradiente coniugato è il migliore in termini di numero di iterazioni necessarie, e risulta anche il più efficiente in termini di tempo di esecuzione (figura 3.4b). Il metodo Gauss-Seidel è il peggiore in termini di tempo di esecuzione, mentre i metodi Jacobi e gradiente mostrano tempi comunque accettabili.

Parte II

Conclusioni Finali

Capitolo 4

Conclusione

In questo progetto abbiamo analizzato e confrontato diversi metodi per la risoluzione di matrici sparse, valutandone l'efficacia in termini di numero di iterazioni necessarie e tempo di esecuzione. I metodi considerati includono il Gradiente, il Gradiente Coniugato, Jacobi e Gauss-Seidel, applicati a diverse matrici di test.
In conclusione, il metodo del gradiente coniugato emerge come la scelta ottimale per la risoluzione di matrici sparse, grazie al suo bilanciamento tra efficienza computazionale e rapidità di convergenza. Tuttavia, l'applicazione specifica e le caratteristiche delle matrici possono influenzare la scelta del metodo più appropriato, suggerendo che un'analisi preliminare delle caratteristiche della matrice sia sempre consigliata.

Questo progetto ha evidenziato l'importanza di un'approfondita valutazione comparativa dei metodi, fornendo indicazioni preziose per la selezione del metodo più adeguato a seconda delle specifiche esigenze computazionali.