Metodi del Calcolo Scientifico

Progetto 1

Bishara Giovanni 869532
Singh Probjot 869434

Indice

1 Introduzione

La scelta di svolgere il progetto usando come linguaggio di programmazione Python e' dettata dalla flessibilita' e adattabilita' che questo linguaggio offre.
Come librerie per la struttura dati delle Matrici e Vettori abbiamo adottato:

- **NumPy**: Fornisce degli strumenti molto efficienti per le operazioni tra matrici e la gestione della struttura dati di matrici e vettori densi
- **SciPy**: Fornice una struttura dati per gestire in memoria le matrici sparse in un metodo analogo a quello fornito da MatLab.

1.1 Struttura Generale

La struttura generale del progetto si compone di due principali file Python:

- **main.py**: File che contiene funzioni di importazione e caricamento delle matrici tramite `SciPy`, funzioni per la scrittura su file del risultati ottenuti dall'esecuzione dei metodi risolutivi, e l'utilizzo vero e proprio della libreria implementata in `methods.py`.
- **methods.py**: File che implementa le funzioni cuore della libreria. In particolare vi e' l'implementazione dei quattro metodi risolutivi, una funzione dispatcher per selezionare il metodo appropriato, e funzioni di supporto per le operazioni matematiche.

Il progetto si divide in due componenti principali: `main.py`, il punto di inizio del programma in cui e' possibile importare le matrici sparse in formato `.mtx` tramite `SciPy` e `methods.py` che e' il vero e proprio cuore della libreria, implementando i metodi risolutivi come funzioni indipendenti.

2 Struttura del Main

2.1 Caricamento e Utilizzo delle Matrici

In seguito le funzioni principali utilizzate per l'importazione delle matrici all'interno del progetto. Esse sono necessarie al fine di poter fornire alla libreria, e quindi ai metodi risolutivi, le matrici trattate.

```
def load_matrices(matrix_dir='matrix'):
    """Carica tutte le matrici .mtx dalla cartella specificata"""
    matrices = {}
    for file in os.listdir(matrix_dir):
        if file.endswith('.mtx'):
            name = file.split('.')[0]
            path = os.path.join(matrix_dir, file)
```

```
            matrices[name] = spio.mmread(path).tocsr()
            # Converti in formato sparso CSR
    return matrices
```

La funzione `load_matrices` si occupa di:

- Scansionare la directory specificata (default: 'matrix')
- Identificare tutti i file con estensione .mtx
- Caricare ogni matrice usando SciPy e convertirla in formato CSR (Compressed Sparse Row)
- Restituire un dizionario con i nomi delle matrici come chiavi e le matrici stesse come valori

```
def create_problem(A):
    """Crea il problema Ax=b con soluzione esatta x=[1,1,...,1]"""
    n = A.shape[0]
    x_exact = np.ones(n)
    b = A.dot(x_exact)
    return b, x_exact
```

La funzione `create_problem` si occupa di:

- Creare un vettore soluzione esatta x_{exact} composto da tutti 1
- Calcolare il vettore dei termini noti $b = Ax_{exact}$
- Restituire la coppia (b, x_{exact}) per il problema $Ax = b$

2.2 Esecuzione degli Esperimenti

La funzione `run_experiments` e' responsabile dell'esecuzione di tutti i metodi risolutivi su tutte le matrici caricate, per diverse tolleranze.

```
def run_experiments(matrices, tolerances):
    """Esegue tutti gli esperimenti e raccoglie i risultati"""
    results = {}

    for mat_name, A in matrices.items():
        print(f"\n{'=' * 40}\nProcessing matrix: {mat_name}\n{'=' * 40}")
        b, x_exact = create_problem(A)
        results[mat_name] = {}

        for tol in tolerances:
            print(f"\n-> Tolleranza: {tol:.0e}")
            current_results = {}

            # Lista dei metodi da testare
            methods = [
                'jacobi',
                'gauss_seidel',
                'gradient',
                'conjugate_gradient'
```

```
            ]

            for method in methods:
                start_time = datetime.now()
                x, iterations, time, error = solve_linear_system(
                    method=method,
                    A=A,
                    b=b,
                    x_exact=x_exact,
                    tol=tol
                )
                exec_time = (datetime.now() - start_time).total_seconds()

                current_results[method] = {
                    'iterations': iterations,
                    'time': exec_time,
                    'error': error,
                    'residual': np.linalg.norm(A.dot(x) - b) / np.linalg.norm(b)
                }

            results[mat_name][tol] = current_results

    return results
```

La funzione `run_experiments` si occupa di:

- Iterare su tutte le matrici caricate
- Per ogni matrice, creare il problema $Ax = b$ con soluzione esatta nota
- Per ogni tolleranza specificata, eseguire tutti i metodi risolutivi
- Per ogni metodo, chiamare la funzione `solve_linear_system` del modulo `methods.py`
- Raccogliere e organizzare i risultati in una struttura dati gerarchica

2.3 Visualizzazione dei Risultati

La funzione `print_results` si occupa di visualizzare i risultati degli esperimenti in formato tabellare.

```
def print_results(results):
    """Stampa i risultati in formato tabellare"""
    for mat_name, mat_results in results.items():
        print(f"\n{'#' * 60}")
        print(f"RISULTATI PER LA MATRICE: {mat_name.upper()}")
        print(f"{'#' * 60}")

        for tol, tol_results in mat_results.items():
            print(f"\n > Tolleranza: {tol:.0e}")
            print("-" * 55)
            print(f"{'Metodo':<20} | {'Iterazioni':<10}
            | {'Tempo (s)':<10} | {'Errore Relativo':<15}")
            print("-" * 55)

            for method, data in tol_results.items():
                print(f"{method:<20} | {data['iterations']:<10}
```

```
                | {data['time']:<10.4f} | {data['error']:<15.2e}")
            print("-" * 55)
```

Questa funzione formatta e visualizza i risultati in modo chiaro e leggibile, organizzandoli per matrice, tolleranza e metodo. Per ogni combinazione, vengono mostrati:

- Il nome del metodo
- Il numero di iterazioni effettuate
- Il tempo di esecuzione in secondi
- L'errore relativo rispetto alla soluzione esatta

2.4 Utilizzo della Libreria

Nel blocco principale del programma (`if __name__ == "__main__"`), vengono utilizzate le funzioni analizzate in precedenza per eseguire gli esperimenti e visualizzare i risultati.

```
if __name__ == "__main__":
    # Configurazione
    matrix_dir = 'matrix'  # Cartella contenente le matrici
    tolerances = [1e-4, 1e-6, 1e-8, 1e-10]  # Lista di tolleranze da testare

    # Caricamento matrici
    print("Caricamento matrici...")
    matrices = load_matrices(matrix_dir)

    # Esecuzione esperimenti
    print("\nAvvio esperimenti...")
    start_time = datetime.now()
    results = run_experiments(matrices, tolerances)
    total_time = (datetime.now() - start_time).total_seconds()

    # Stampa risultati
    print("\n\n" + "=" * 60)
    print(f"RIEPILOGO FINALE - Tempo totale esecuzione: {total_time:.2f} secondi")
    print("=" * 60)
    print_results(results)
```

Il flusso di esecuzione del programma e' il seguente:

- Definizione dei parametri di configurazione (directory delle matrici e tolleranze)
- Caricamento delle matrici dalla directory specificata
- Esecuzione degli esperimenti per tutte le matrici, tolleranze e metodi
- Visualizzazione dei risultati in formato tabellare

Per ogni matrice e tolleranza, vengono eseguiti tutti e quattro i metodi risolutivi (Jacobi, Gauss-Seidel, Gradiente, Gradiente Coniugato) tramite la funzione `solve_linear_system` del modulo `methods.py`, che funge da dispatcher per selezionare il metodo appropriato.

3 Struttura della Libreria

La struttura della libreria e' composta da un insieme di funzioni per la risoluzione di sistemi lineari:

- Funzione dispatcher `solve_linear_system` per selezionare il metodo appropriato
- Funzione per l'esecuzione del metodo di Jacobi
- Funzione per l'esecuzione del metodo di Gauss-Seidel
- Funzione per l'esecuzione del metodo del Gradiente
- Funzione per l'esecuzione del metodo del Gradiente Coniugato
- Funzioni di utilitÃă per supportare i metodi risolutivi

3.1 Funzione Dispatcher

```python
def solve_linear_system(method, A, b, x_exact, tol=1e-6, max_iter=20000):
    """
    Dispatcher che richiama il metodo iterativo specificato
    Args:
        method (str): Nome del metodo ['jacobi', 'gauss_seidel',
        'gradient', 'conjugate_gradient']
        A: Matrice del sistema
        b: Vettore dei termini noti
        x_exact: Soluzione esatta
        tol: Tolleranza
        max_iter: Numero massimo di iterazioni

    Returns:
        tuple: (soluzione, iterazioni, tempo, errore_relativo)
    """
    method = method.lower().replace('-', '_').replace(' ', '_')

    method_map = {
        'jacobi': jacobi,
        'gauss_seidel': gauss_seidel,
        'gs': gauss_seidel,
        'gradient': gradient,
        'grad': gradient,
        'conjugate_gradient': conjugate_gradient,
        'cg': conjugate_gradient,
        'gradiente': gradient,
        'gradiente_coniugato': conjugate_gradient
    }

    if method not in method_map:
        raise ValueError(
            f"Metodo {method} non valido. Scegli tra: \n"
            "-'jacobi'\n"
            "-'gauss_seidel'/'gs'\n"
            "-'gradient'/'grad'\n"
```

```
36              "-'conjugate_gradient'/'cg'"
37          )
38
39      return method_map[method](A, b, x_exact, tol, max_iter)
```

La funzione dispatcher si occupa di:

- Normalizzare il nome del metodo richiesto
- Mappare il nome del metodo alla funzione corrispondente
- Verificare che il metodo richiesto sia valido
- Richiamare la funzione appropriata con i parametri forniti

3.2 Metodi per la risoluzione del Sistema Lineare

Ognuna delle 4 funzioni di risoluzione accetta gli stessi parametri:

- `A`: La matrice del sistema
- `b`: Il vettore dei termini noti
- `x_exact`: La soluzione esatta (per calcolare l'errore)
- `tol`: La tolleranza per il criterio di arresto
- `max_iter`: Il numero massimo di iterazioni

Ogni funzione inizializza:

- Il tempo di inizio dell'esecuzione tramite `process_time()`
- Il vettore iniziale `x` (tipicamente un vettore nullo)
- Eventuali valori necessari per l'algoritmo specifico

Successivamente, ogni metodo esegue un ciclo `for` per un massimo di `max_iter` iterazioni, in cui:

- Vengono aggiornati i vettori e le costanti secondo l'algoritmo specifico
- Viene calcolato il residuo relativo $\frac{\|Ax-b\|}{\|b\|}$
- Se il residuo e' minore della tolleranza, il ciclo viene interrotto

Alla fine dell'esecuzione, ogni funzione calcola:

- Il tempo di esecuzione
- L'errore relativo rispetto alla soluzione esatta

Tutte le funzioni ritornano una tupla del tipo:

$$\begin{bmatrix} x & tempo & \varepsilon_r & iterazioni \end{bmatrix}$$

in cui:

- x e' il vettore soluzione calcolato
- *tempo* e' il tempo impiegato per l'esecuzione
- ε_r e' l'errore relativo rispetto alla soluzione esatta
- *iterazioni* e' il numero di iterazioni effettuate

Ai fini del progetto e' stato deciso di avere un numero massimo di iterazioni pari a **20000** in modo da permettere anche a tolleranze molto basse di raggiungere il risultato.

Di seguito e' riportato nel dettaglio il codice per ogni metodo, evidenziando le funzionalita' e le operazioni svolte.

3.2.1 Metodo di Jacobi

```
def jacobi(A, b, x_exact, tol, max_iter):
    """
    Metodo di Jacobi per sistemi lineari
    """
    x = np.zeros_like(b)
    D = sp.diags(A.diagonal())
    D_inv = sp.diags(1 / A.diagonal())
    R = A - D
    start_time = process_time()

    for k in range(max_iter):
        x_new = D_inv.dot(b - R.dot(x))
        residual = A.dot(x_new) - b
        rel_res = np.linalg.norm(residual) / np.linalg.norm(b)

        if rel_res < tol:
            break
        x = x_new

    error = np.linalg.norm(x - x_exact) / np.linalg.norm(x_exact)
    return x, k + 1, process_time() - start_time, error
```

Il metodo di **Jacobi** converge se una matrice e' a **dominanza diagonale stretta per righe**. L'implementazione segue questi passaggi:

- Inizializzazione del vettore soluzione x come vettore nullo
- Estrazione della matrice diagonale D e calcolo della sua inversa D^{-1}
- Calcolo della matrice resto $R = A - D$

- Ad ogni iterazione:
  - Calcolo del nuovo vettore x secondo la formula:
$$x^{(k+1)} = D^{-1}(b - Rx^{(k)})$$
  - Calcolo del residuo relativo e verifica del criterio di arresto
- Calcolo dell'errore relativo rispetto alla soluzione esatta

3.2.2 Metodo di Gauss-Seidel

```python
def gauss_seidel(A, b, x_exact, tol, max_iter):
    """
    Metodo di Gauss-Seidel per sistemi lineari
    """
    x = np.zeros_like(b)
    start_time = process_time()
    A = A.toarray()  # Per semplicitÃă di implementazione

    for k in range(max_iter):
        x_old = x.copy()
        for i in range(A.shape[0]):
            x[i] = (b[i] - A[i, :i] @ x[:i] - A[i, i + 1:] @ x_old[i + 1:])
            / A[i, i]

        residual = A.dot(x) - b
        rel_res = np.linalg.norm(residual) / np.linalg.norm(b)
        if rel_res < tol:
            break

    error = np.linalg.norm(x - x_exact) / np.linalg.norm(x_exact)
    return x, k + 1, process_time() - start_time, error
```

Analogamente al metodo di **Jacobi**, anche il metodo di **Gauss-Seidel** richiede la **dominanza diagonale stretta per righe** della matrice per assicurare la convergenza.

Il metodo di **Gauss-Seidel** puo' essere visto come un miglioramento del metodo di **Jacobi**, dove vengono sfruttate le entrate del vettore **x** gia' calcolate nell'iterazione attuale:

- Inizializzazione del vettore soluzione x come vettore nullo
- Conversione della matrice in formato denso per semplicita' di implementazione
- Ad ogni iterazione:
  - Aggiornamento di ogni componente x_i secondo la formula:
$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right)$$
  - Calcolo del residuo relativo e verifica del criterio di arresto
- Calcolo dell'errore relativo rispetto alla soluzione esatta

3.2.3 Metodo del Gradiente

```python
def gradient(A, b, x_exact, tol, max_iter):
    """
    Metodo del Gradiente per sistemi lineari
    """
    x = np.zeros_like(b)
    r = b - A.dot(x)
    p = r.copy()
    start_time = process_time()

    for k in range(max_iter):
        Ap = A.dot(p)
        alpha = r.dot(r) / (p.dot(Ap))
        x += alpha * p
        r_new = r - alpha * Ap

        rel_res = np.linalg.norm(r_new) / np.linalg.norm(b)
        if rel_res < tol:
            break

        beta = r_new.dot(r_new) / (r.dot(r))
        p = r_new + beta * p
        r = r_new

    error = np.linalg.norm(x - x_exact) / np.linalg.norm(x_exact)
    return x, k + 1, process_time() - start_time, error
```

Il metodo del **Gradiente** e' un metodo iterativo non stazionario, dove lo scalare α dipende dalle iterazioni. L'implementazione segue questi passaggi:

- Inizializzazione del vettore soluzione x come vettore nullo
- Calcolo del residuo iniziale $r^{(0)} = b - Ax^{(0)}$
- Inizializzazione della direzione di discesa $p^{(0)} = r^{(0)}$
- Ad ogni iterazione:
  - Calcolo del passo ottimale $\alpha_k = \frac{r^{(k)T} r^{(k)}}{p^{(k)T} A p^{(k)}}$
  - Aggiornamento della soluzione $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$
  - Aggiornamento del residuo $r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)}$
  - Calcolo del parametro $\beta_k = \frac{r^{(k+1)T} r^{(k+1)}}{r^{(k)T} r^{(k)}}$
  - Aggiornamento della direzione di discesa $p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)}$
  - Verifica del criterio di arresto
- Calcolo dell'errore relativo rispetto alla soluzione esatta

3.2.4 Metodo del Gradiente Coniugato

```python
def conjugate_gradient(A, b, x_exact, tol, max_iter):
    """
    Metodo del Gradiente Coniugato per sistemi lineari
    """
    x = np.zeros_like(b)
    r = b - A.dot(x)
    p = r.copy()
    rsold = r.dot(r)
    start_time = process_time()

    for k in range(max_iter):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x += alpha * p
        r -= alpha * Ap
        rsnew = r.dot(r)

        rel_res = np.sqrt(rsnew) / np.linalg.norm(b)
        if rel_res < tol:
            break

        p = r + (rsnew / rsold) * p
        rsold = rsnew

    error = np.linalg.norm(x - x_exact) / np.linalg.norm(x_exact)
    return x, k + 1, process_time() - start_time, error
```

Il metodo del **Gradiente Coniugato** e' una variante del metodo del Gradiente che utilizza direzioni di discesa coniugate rispetto alla matrice A. L'implementazione segue questi passaggi:

- Inizializzazione del vettore soluzione x come vettore nullo
- Calcolo del residuo iniziale $r^{(0)} = b - Ax^{(0)}$
- Inizializzazione della direzione di discesa $p^{(0)} = r^{(0)}$
- Memorizzazione di $r^{(0)T} r^{(0)}$ in $rsold$
- Ad ogni iterazione:
  - Calcolo del passo ottimale $\alpha_k = \frac{r^{(k)T} r^{(k)}}{p^{(k)T} A p^{(k)}}$
  - Aggiornamento della soluzione $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$
  - Aggiornamento del residuo $r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)}$
  - Calcolo di $r^{(k+1)T} r^{(k+1)}$ in $rsnew$
  - Verifica del criterio di arresto
  - Aggiornamento della direzione di discesa $p^{(k+1)} = r^{(k+1)} + \frac{rsnew}{rsold} p^{(k)}$
  - Aggiornamento di $rsold = rsnew$
- Calcolo dell'errore relativo rispetto alla soluzione esatta

3.3 Altre Funzioni Utili

In questa sezione vengono descritte le funzioni di supporto utilizzate dai metodi risolutivi per eseguire operazioni comuni o verificare condizioni specifiche.

3.3.1 Controllo della Tolleranza

Il controllo della tolleranza e' implementato direttamente all'interno di ciascun metodo risolutivo. La condizione di arresto e' basata sul calcolo del residuo relativo:

```
# Calcolo del residuo relativo
residual = A.dot(x) - b
rel_res = np.linalg.norm(residual) / np.linalg.norm(b)

# Verifica del criterio di arresto
if rel_res < tol:
    break
```

Il residuo relativo e' definito come:

$$\frac{\|Ax - b\|}{\|b\|}$$

Quando questo valore scende al di sotto della tolleranza specificata, il metodo ha raggiunto la convergenza e il ciclo di iterazioni viene interrotto.

3.3.2 Matrice inversa di P - Metodo di Jacobi

Nel metodo di Jacobi, la matrice P e' la matrice diagonale D estratta dalla matrice A. L'inversa di P viene calcolata come:

```
# Estrazione della matrice diagonale D
D = sp.diags(A.diagonal())
# Calcolo dell'inversa di D
D_inv = sp.diags(1 / A.diagonal())
```

Poiche' D e' una matrice diagonale, la sua inversa e' semplicemente una matrice diagonale con gli elementi reciproci degli elementi diagonali di D.

3.3.3 Matrice P - Metodo di Gauss-Seidel

Nel metodo di Gauss-Seidel, la matrice P e' implicitamente la parte triangolare inferiore della matrice A, inclusa la diagonale. Invece di costruire esplicitamente questa matrice, l'algoritmo implementa direttamente la formula di aggiornamento:

```
for i in range(A.shape[0]):
    x[i] = (b[i] - A[i, :i] @ x[:i] - A[i, i + 1:] @ x_old[i + 1:]) / A[i, i]
```

Questa implementazione equivale a risolvere il sistema $Px^{(k+1)} = b - (A - P)x^{(k)}$ tramite sostituzione in avanti.

3.3.4 Sostituzione in Avanti - Forward Substitution

La sostituzione in avanti e' implementata implicitamente nel metodo di Gauss-Seidel attraverso il ciclo che aggiorna sequenzialmente ogni componente del vettore soluzione:

```
for i in range(A.shape[0]):
    x[i] = (b[i] - A[i, :i] @ x[:i] - A[i, i + 1:] @ x_old[i + 1:]) / A[i, i]
```

Questo ciclo risolve il sistema triangolare inferiore $Lx = b$ componente per componente, utilizzando i valori gia' calcolati per le componenti precedenti.

3.3.5 Controllo Matrice a Dominanza Diagonale Stretta per Righe

Una funzione per verificare se una matrice e' a dominanza diagonale stretta per righe potrebbe essere implementata come segue:

```
def is_diagonally_dominant(A):
    """
    Verifica se la matrice A e' a dominanza diagonale stretta per righe
    """
    A = A.toarray() if hasattr(A, 'toarray') else A
    n = A.shape[0]

    for i in range(n):
        diagonal = abs(A[i, i])
        row_sum = sum(abs(A[i, j]) for j in range(n) if j != i)

        if diagonal <= row_sum:
            return False

    return True
```

Una matrice e' a dominanza diagonale stretta per righe se, per ogni riga, il valore assoluto dell'elemento diagonale e' maggiore della somma dei valori assoluti degli altri elementi della riga.

3.3.6 Calcolo dell'Errore Relativo

L'errore relativo viene calcolato alla fine di ogni metodo risolutivo, confrontando la soluzione calcolata con la soluzione esatta:

```
error = np.linalg.norm(x - x_exact) / np.linalg.norm(x_exact)
```

L'errore relativo e' definito come:

$$\frac{\|x - x_{exact}\|}{\|x_{exact}\|}$$

Questo valore fornisce una misura della precisione della soluzione calcolata rispetto alla soluzione esatta.

3.3.7 Altre Funzioni

Altre funzioni di supporto che potrebbero essere utili includono:

```
def get_start_vector(n):
    """
    Crea un vettore iniziale nullo di dimensione n
    """
    return np.zeros(n)

def create_exact_solution(n):
    """
    Crea la soluzione esatta [1, 1, ..., 1] di dimensione n
    """
    return np.ones(n)
```

Queste funzioni semplificano la creazione di vettori iniziali e soluzioni esatte per i test.

3.4 Debug

Per facilitare il debug e l'analisi dei metodi risolutivi, e' possibile implementare funzioni di logging che stampano informazioni dettagliate durante l'esecuzione:

```
def print_iteration_info(method_name, iteration, x, residual, rel_res):
    """
    Stampa informazioni sull'iterazione corrente
    """
    print(f"[{method_name}] Iterazione {iteration}:")
    print(f"  x = {x}")
    print(f"  residuo = {residual}")
    print(f"  residuo relativo = {rel_res}")

def print_convergence_info(method_name, iterations, time, error):
    """
    Stampa informazioni sulla convergenza del metodo
    """
    print(f"[{method_name}] Convergenza raggiunta:")
    print(f"  iterazioni = {iterations}")
    print(f"  tempo = {time:.6f} secondi")
    print(f"  errore relativo = {error:.6e}")
```

Queste funzioni possono essere chiamate all'interno dei metodi risolutivi quando e' attivata la modalitÃă di debug, per fornire informazioni dettagliate sul progresso dell'algoritmo e sui risultati finali.

4 Risultati Ottenuti e Considerazioni

In seguito sono riportati in diverse tabelle i risultati ottenuti dall'esecuzione dei quattro metodi risolutivi ed alcune osservazioni generali sul comportamento di ogni metodo. Le tabelle sono suddivise per la matrice utilizzata, dove per ognuna vi e' riportato il nome della matrice. Inoltre, le tabelle sono state organizzate in sezioni per metodo risolutivo. Ogni tabella riporta le seguenti informazioni:

- *Tolleranza*
- *Tempo Impiegato in Secondi*
- *Errore Relativo*
- *Numero di Iterazioni*

4.1 Metodo di Jacobi

Il primo metodo analizzato e' il Metodo di **Jacobi**, che si comporta discretamente bene per le matrici di tipo *"spa"*, mostrando una crescita accettabile delle iterazioni e del tempo impiegato al diminuire della tolleranza, ma con un miglior errore relativo. Per le matrici di tipo *"vem"* vi e' una crescita importante delle iterazioni gia' a partire dalla tolleranza piÃź grande.

Matrice 0: "spa1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.049800000	1.90e-03	115
1e-06	0.075600000	1.93e-05	181
1e-08	0.093200000	1.96e-07	247
1e-10	0.131800000	1.99e-09	313

Matrice 1: "spa2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.153300000	2.21e-03	36
1e-06	0.229500000	2.08e-05	57
1e-08	0.316800000	1.96e-07	78
1e-10	0.435900000	1.85e-09	99

Matrice 2: "vem1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.056600000	3.55e-03	1314
1e-06	0.103200000	3.55e-05	2433
1e-08	0.148200000	3.55e-07	3552
1e-10	0.193300000	3.55e-09	4671

Matrice 3: "vem2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.109700000	4.98e-03	1927
1e-06	0.200400000	4.98e-05	3676
1e-08	0.297600000	4.98e-07	5425
1e-10	0.431600000	4.98e-09	7174

4.2 Metodo di Gauss-Seidel

Passando all'analisi dei risultati del metodo di **Gauss-Seidel** e' prevista una diminuzione generale delle iterazioni rispetto al metodo di **Jacobi**, ciÃš infatti si e' verificato mostrando un grosso decremento delle iterazioni complessive per le quattro matrici trattate, ma con un aumento, abbastanza importante per quanto riguarda le matrici *"vem"*, del tempo impiegato. Infine si puÃš notare come l'errore relativo sia aumentato rispetto a **Jacobi** se si considerano matrici *"spa"*.

Matrice 0: "spa1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.039300000	1.82e-02	9
1e-06	0.075900000	1.30e-04	17
1e-08	0.098200000	1.71e-06	24
1e-10	0.135400000	2.25e-08	31

Matrice 1: "spa2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.113500000	2.60e-03	5
1e-06	0.158700000	5.14e-05	8
1e-08	0.230800000	2.79e-07	12
1e-10	0.282600000	5.57e-09	15

Matrice 2: "vem1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	5.332500000	3.51e-03	659
1e-06	9.610700000	3.53e-05	1218
1e-08	13.109000000	3.52e-07	1778
1e-10	17.209300000	3.51e-09	2338

Matrice 3: "vem2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	13.048900000	4.95e-03	965
1e-06	25.149300000	4.94e-05	1840
1e-08	37.378400000	4.96e-07	2714
1e-10	49.375500000	4.95e-09	3589

4.3 Metodo del Gradiente

Il metodo del **Gradiente** risulta molto interessante rispetto ai primi due metodi risolutivi. Si puo' notare come ci sia stata un'inversione delle prestazioni rispetto a **Jacobi** e **Gauss-Seidel**. I primi due, infatti, mostravano degli ottimi risultati su matrici di tipo *"spa"* con un incremento delle metriche considerevole per le matrici di tipo *"vem"*, mentre per il metodo del **Gradiente** vi e' sostanzialmente l'opposto. Per le matrici di tipo *"spa"* risulta un incremento davvero elevato di iterazioni, tempo impiegato ed errore relativo al diminuire della tolleranza.

Matrice 0: "spa1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.009700000	2.08e-02	49
1e-06	0.025700000	2.55e-05	134
1e-08	0.033100000	1.32e-07	177
1e-10	0.039100000	1.20e-09	200

Matrice 1: "spa2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.078900000	9.82e-03	42
1e-06	0.227400000	1.20e-04	122
1e-08	0.380400000	5.59e-07	196
1e-10	0.495500000	5.32e-09	240

Matrice 2: "vem1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.001400000	4.08e-05	38
1e-06	0.001400000	3.73e-07	45
1e-08	0.001700000	2.83e-09	53
1e-10	0.001800000	2.19e-11	59

Matrice 3: "vem2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.001900000	5.73e-05	47
1e-06	0.002200000	4.74e-07	56
1e-08	0.002800000	4.30e-09	66
1e-10	0.006100000	2.25e-11	74

4.4 Metodo del Gradiente Coniugato

Infine, per l'analisi del metodo del **Gradiente Coniugato** e' attesa, come nel caso di **Gauss-Seidel**, una diminuzione delle iterazioni effettuate, in quanto si e' andato ad eliminare il problema del *zig-zag* del metodo del **Gradiente**. Questo avviene, riportando le miglior prestazioni sulle matrici di tipo *"vem"* rispetto agli altri metodi e discrete prestazioni per matrici di tipo *"spa"*, avvicinandosi ai metodi di **Jacobi** e **Gauss-Seidel**.

Matrice 0: "spa1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.009900000	2.08e-02	49
1e-06	0.024900000	2.55e-05	134
1e-08	0.036100000	1.32e-07	177
1e-10	0.037400000	1.20e-09	200

Matrice 1: "spa2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.078600000	9.82e-03	42
1e-06	0.235900000	1.20e-04	122
1e-08	0.415200000	5.59e-07	196
1e-10	0.445700000	5.32e-09	240

Matrice 2: "vem1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.001100000	4.08e-05	38
1e-06	0.001300000	3.73e-07	45
1e-08	0.001500000	2.83e-09	53
1e-10	0.001600000	2.19e-11	59

Matrice 3: "vem2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.002100000	5.73e-05	47
1e-06	0.002100000	4.74e-07	56
1e-08	0.002500000	4.30e-09	66
1e-10	0.004900000	2.25e-11	74