Progetto di Metodi del Calcolo Scientifico

Relazione progetto alternativo 1

Autori:
Federica Ratti, 886158
Nicolò Molteni, 933190

{f.ratti15, n.molteni6}@campus.unimib.it

Corso di Laurea Magistrale in Informatica
Università di Milano-Bicocca
Anno Accademico 2024-2025

Indice

Capitolo 1

La libreria

La libreria sviluppata per il progetto assegnato consente la risoluzione di sistemi di equazioni lineari mediante metodi o solver iterativi studiati a lezione: il *metodo di Jacobi*, il *metodo di Gauss-Seidel*, il *metodo del Gradiente* e il *metodo del Gradiente Coniugato*.

1.1 Il linguaggio e le librerie utilizzate

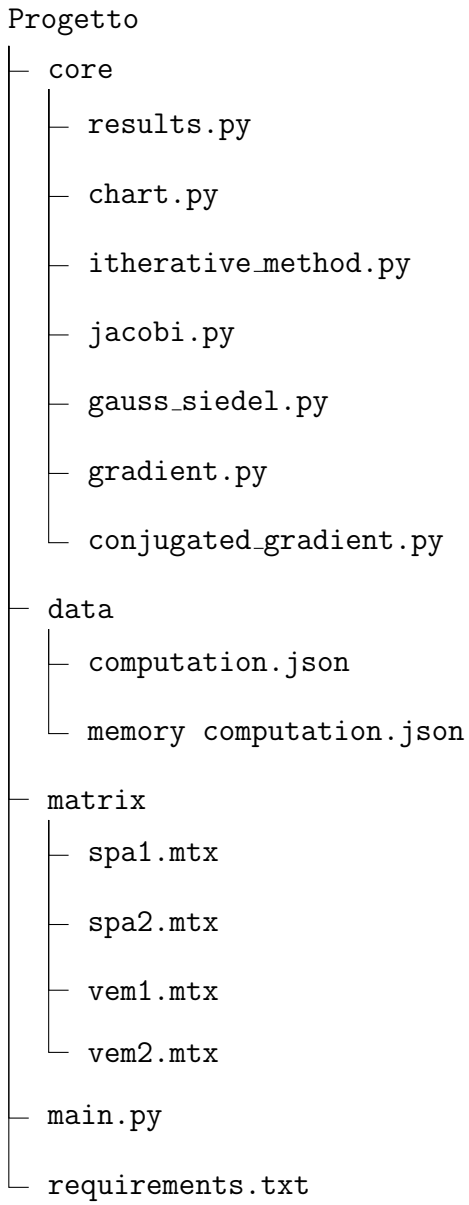
Il linguaggio scelto per lo sviluppo è `Python`, un linguaggio open source largamente adottato anche in ambito accademico e scientifico. Infatti, la sua semplicità sintattica e la grande disponibilità di librerie lo rendono un linguaggio ideale per lo sviluppo rapido di algoritmi numerici.

Le operazioni tra matrici, matrici e scalari o su singole matrici sono state implementate utilizzando la libreria `NumPy`, che consente di eseguire calcoli numerici in modo efficiente grazie a due caratteristiche fondamentali:

- le operazioni matriciali sono implementate in linguaggio `C`, garantendo prestazioni elevate;
- molte operazioni sfruttano OpenBLAS o MKL e permettono di vettorizzare o parallelizzate internamente le operazioni.

Per alcune funzionalità non presenti in `NumPy` è stata utilizzata anche `SciPy`, una libreria open source che estende `NumPy` con moduli specializzati per l'algebra lineare. Nel progetto, `SciPy` è stata impiegata ad esempio per: verificare la simmetria di una matrice e caricare matrici da file nel formato `.mtx` (Matrix Market). Anche `SciPy`, come `NumPy`, si basa su librerie ottimizzate scritte in `C`, `C++` e `Fortran`.

1.2 La directory del progetto



Qui sopra è possibile osservare il directory tree della libreria sviluppata. Il progetto è composto dalle cartelle: *matrix*, contiene i file `.mtx` delle matrici proposte; *data*, contiene i due file JSON con i dati relativi ai run di ciascun metodo iterativo; *core*, contiene le classi con la logica applicativa. Si possono osservare le 3 classi principali: la prima è la classe `chart`, la seconda è la classe `results` e la terza è la classe astratta `iterative_methods`.

Infine sono presenti due file: il primo, `requirements.txt` viene creato automaticamente e contiene le librerie necessarie per il corretto funzionamento del progetto. Il secondo `main.py` permette di eseguire le funzioni implementate dalla libreria. Di seguito è presente una breve introduzione delle classi principali e successivamente verrà discusso il file `main.py`.

1.2.1 La classe chart

La classe `chart` genera una serie di grafici utili per eseguire analisi su matrici e solver iterativi. Nel primo caso permette di generare grafici di densità delle matrici e percentuale di righe definite positive. Nel secondo caso è possibile generare grafici a barre in scala semi-logaritmica per visualizzare i risultati dei metodi iterativi applicati alle matrici, con diverse soglie di tolleranza. I grafici permettono di confrontare facilmente le performance in termini di numero di iterazioni, errore residuo, tempo di convergenza e memoria utilizzata.

È possibile creare grafici basati su un singolo solver, run oppure su tutti i run eseguiti finora. In questo secondo caso, i dati vengono caricati dai file nella directory `data` e i valori da visualizzare vengono calcolati usando la mediana, così da ridurre l'influenza di eventuali outlier creati ad esempio da fluttuazioni temporali.

All'interno dell'eseguibile `main.py` non sono stati utilizzati i metodi che riguardano l'analisi delle matrici, quest'ultimi metodi sono stati impiegati per generare grafici utili all'approfondimento del confronto eseguito.

1.2.2 La classe results

La classe `results` permette di rappresentare un risultato di un singolo run del solver iterativo. Ciascuna istanza di questa classe è composta da: il numero di iterazioni, l'errore al termine dell'esecuzione, la tolleranza, il tempo di convergenza del solver, le dimensioni della matrice che rappresenta il sistema e la quantità di memoria occupata. Quest'ultimo parametro viene impostato se e solo se è attiva l'analisi della memoria utilizzata.

La classe `results` contiene tutti i metodi necessari per impostare e leggere questi parametri, infine possiede un metodo per rappresentare il risultato sotto forma di stringa testuale.

1.2.3 La classe iterative_methods

La classe astratta `iterative_methods` è la componente principale di tutta la libreria perché definisce tutti i metodi che deve avere un solver iterativo. Nel dettaglio potremo osservare due tipologie di metodi:

- **metodi statici**, vengono ereditati dalle sottoclassi e non hanno necessità di essere ridefiniti. Tutti i metodi dichiarati statici definiscono un comportamento uguale per ciascun solver iterativo. I metodi statici sono:

  1. `load(path: str)`

     Carica in memoria una matrice a partire da un file `.mtx` specificato nel percorso `path`. Le matrici sono caricate in formato denso.

  2. `check_iteration(A: NDArray[np.float64], x_new: NDArray[np.float64], b: NDArray [np.float64])`

     Calcola il criterio di arresto del solver. Dati la matrice `A`, il vettore soluzione `x_new` al passo n e il termine noto `b`, il residuo viene calcolato secondo la Formula 1.1:

$$\frac{\|(\mathtt{A} \cdot \mathtt{x\_new}) - \mathtt{b}\|_2}{\|\mathtt{b}\|_2} \tag{1.1}$$

Se il valore è inferiore alla tolleranza fissata, l'algoritmo termina. È importante notare che il solver può anche interrompersi per il raggiungimento del numero massimo di iterazioni, anche se il criterio non è soddisfatto.

3. `evaluate_error(x_ex: NDArray[np.float64], x_new: NDArray[np.float64])`

   Calcola l'errore relativo tra la soluzione esatta `x_ex` e la soluzione calcolata `x_new` dopo la convergenza del solver iterativo usando la Formula 1.2:

$$\frac{\|\mathtt{x\_ex} - \mathtt{x\_new}\|_2}{\|\mathtt{x\_ex}\|_2} \tag{1.2}$$

   All'interno del progetto la soluzione esatta, `x_ex` è rappresentata dal vettore colonna composto da soli 1.

4. `converge(A: NDArray[np.float64], name: str)`

   Verifica se il metodo iterativo indicato dal nome `name` può convergere quando applicato alla matrice `A`. In caso di non convergenza restituisce un messaggio informativo da mostrare all'utente. Indipendente dal solver iterativo applicato per risolvere il sistema associato alla matrice viene verificato che la matrice sia quadrata e che le entrate sulla sua diagonale siano strettamente maggiori di 0.

5. `save_stats(res: Results, path: str, method_name: str, matrix_name: str, trace_memory: bool)`

   Salva i risultati (`res`) di un'esecuzione del solver in un file JSON situato nel percorso `path`. Occorre specificare anche il nome di: metodo iterativo e matrice, infine è importante specificare se è stato attivato il tracciamento della memoria.

- **metodi astratti**, vengono ereditati da ciascun solver e devono essere necessariamente ridefiniti in modo tale da renderli personalizzati per ciascun metodo:

  1. `solve(self, A: NDArray[np.float64], b: NDArray[np.float64], x_ex: NDArray[np.float64], n_max: int, toll: float, matrix_name: str, trace_memory: bool)`

     Implementa il metodo di risoluzione iterativa. Ogni solver deve fornire la propria implementazione del metodo `solve`.

  2. `name(self)`

     Restituisce il nome del solver iterativo che è stato utilizzato.

Tutti i solver implementati (*Jacobi, Gauss-Siedel, metodo del Gradiente* e *metodo del Gradiente Coniugato*) estendono la classe `iterative_solver`. Come già detto implementano tutti i metodi statici e dovranno ridefinire il metodo `solve` e `name`. Di seguito analizzeremo la loro implementazione nel dettaglio.

Jacobi

```
        def solve(self, A: NDArray[np.float64], b: NDArray[np.float64],
              x_ex: NDArray[np.float64], n_max: int, toll: float,
                  matrix_name: str, trace_memory: bool) -> Results:

        # verifica la convergenza del metodo
        # forse meglio spostarla nel main
        conv, msg = self.converge(A)

        if not conv:
            raise ValueError(msg)

        # inizio a tracciare l'uso della memoria
        # da parte del metodo ma prima pulisco
        # gli stack allocati da python
        if trace_memory:
            tracemalloc.clear_traces()
            tracemalloc.start()

        m, n = np.shape(A)

        # estrazione della diagonale e calcolo
        # della sua inversa
        D = np.diag(np.diag(A))
        invD = np.diag(1 / np.diag(A))

        # calcolo la decomposizione LU
        # sottraendo D ad A
        B = D - A

        # creo il vettore che rappresenta la
        # soluzione iniziale del sistema
        x_0 = np.zeros(shape=(m, 1))
        x_new = x_0

        # definisco il numero d'iterazioni
        # per limitarle
        nit = 0

        start = time.time()

        # applico il metodo di jacobi per calcolare la
        # soluzione del sistema.
        while self.check_iteration(A, x_new, b) > toll and nit < n_max:
            x_old = x_new

            x_new = np.dot(invD, (b + np.dot(B, x_old)))
            nit = nit + 1

        # salvo il tempo necessario per il
        # calcolo della convergenza
        stop = time.time()
        elapsed_time = stop - start
```

```
52
53          # calcolo l'errore dell'ultimo run
54          err = self.evaluate_error(x_ex, x_new)
55
56          # ottengo l'uso di memoria da parte del
57          # metodo e reinizializzo
58          if trace_memory:
59              usage, peak = tracemalloc.get_traced_memory()
60              tracemalloc.stop()
61
62              # salvo le statistiche e genero il valore
63              # di ritorno della funzione
64              res = Results(nit=nit, err=err, tim=elapsed_time, tol=toll,
                    dim=m, mem=usage, mep=peak)
65
66          else:
67              res = Results(nit=nit, err=err, tim=elapsed_time, tol=toll,
                    dim=m)
68
69          if not trace_memory:
70              self.save_stats(res, "data/computation.json", "jacobi",
                    matrix_name, trace_memory)
71          else:
72              self.save_stats(res, "data/memory computation.json", "jacobi",
                     matrix_name, trace_memory)
73          return res
```

Codice 1.1: Implementazione del metodo di Jacobi.

Il metodo di Jacobi mostrato nella classe JacobiMethod è un'implementazione dell'algoritmo iterativo per la risoluzione del sistema lineare $Ax = b$. Il metodo è basato sulla scomposizione della matrice A nella somma $A = D + B$ dove D è la matrice contenente gli elementi sulla diagonale principale di A e $B = D - A$ è la parte rimanente della matrice, cioè i termini non diagonali che possono essere riscritti come la somma della matrice diagonale inferiore e superiore, $B = L+U$. Il metodo di Jacobi si può applicare quando la matrice A è strettamente a dominanza diagonale, condizione sufficiente ma non necessaria alla convergenza del metodo. L'algoritmo funziona nel seguente modo:

- Viene eseguito un test di convergenza in base al metodo iterativo applicato. Per il metodo di Jacobi si controlla se la matrice A in input è strettamente a dominanza diagonale e se è simmetrica e definita positiva. In caso contrario viene sollevata un'eccezione e l'esecuzione viene interrotta. Successivamente, dalla matrice A si estrae la diagonale D, di cui si calcola l'inversa, e si costruisce la matrice $B = D - A$, necessaria per l'aggiornamento iterativo della soluzione. Per la risoluzione con il metodo di Jacobi è importante che la diagonale della matrice sia invertibile: questa proprietà è garantita dai controlli effettuati in precedenza. L'invertibilità della diagonale è importante perché scrivendo la matrice come $A = D - (L + U)$, dove L è la parte triangolare inferiore e U quella superiore, il sistema $A\vec{x} = \vec{f}$ può essere riscritto nella forma iterativa: $\vec{x}^{(k+1)} = D^{-1}\left[(L + U)\vec{x}^{(k)} + \vec{f}\right]$. Se D non fosse invertibile allora il calcolo della soluzione $\vec{x}^{(k+1)}$ non sarebbe possibile.

Infine, vengono inizializzati: un vettore colonna nullo $\vec{x}^{(0)}$ come soluzione iniziale del sistema ed altre variabili ausiliarie (ad esempio numero massimo di iterazioni e tolleranza).

- Inizia l'esecuzione del ciclo iterativo per il calcolo della soluzione del metodo, ad ogni passo la nuova soluzione `x_new` viene aggiornata attraverso la formula `x_new = np.dot(invD, (b + np.dot(B, x_old)))`.
- Al termine del ciclo iterativo vengono misurati: il tempo di esecuzione totale, l'errore tra la soluzione esatta fornita in input e quella calcolata e opzionalmente la memoria occupata.Infine, viene costruito un oggetto Results contenente tutte le statistiche relative all'esecuzione (numero di iterazioni, errore, tempo, dimensione del sistema, uso e picco di memoria se richiesti), e tali informazioni vengono salvate su file JSON corrispondente.

Gauss-Siedel

```
        def solve(self, A: NDArray[np.float64], b: NDArray[np.float64],
            x_ex: NDArray[np.float64], n_max: int, toll: float,
                matrix_name: str, trace_memory: bool) -> Results:

        # verifica la convergenza del metodo
        # forse meglio spostarla nel main
        conv, msg = self.converge(A)

        if not conv:
            raise ValueError(msg)

        # inizio a tracciare l'uso della memoria
        # da parte del metodo ma prima pulisco
        # gli stack allocati da python
        if trace_memory:
            tracemalloc.clear_traces()
            tracemalloc.start()

        m, n = np.shape(A)

        # ricavo la matrice triangolare
        # inferiore mantenendo la
        # diagonale principale
        L = tril(A)

        # calcolo la matrice U
        # sottraendo L ad A
        B = A - L

        # creo il vettore che rappresenta la
        # soluzione iniziale del sistema
        x_0 = np.zeros(shape=(m, 1))
        x_new = x_0

        # definisco il numero d'iterazioni
        # per limitarle
        nit = 0

        start = time.time()

        # applico il metodo di jacobi per calcolare la
        # soluzione del sistema.
        while self.check_iteration(A, x_new, b) > toll and nit < n_max:
            x_old = x_new

            x_new = self.resolve_triang_inf(L, (b - np.dot(B, x_old)))
            nit = nit + 1

        # salvo il tempo necessario per il
        # calcolo della convergenza
        stop = time.time()
        elapsed_time = stop - start
```

```
52
53          # calcolo l'errore dell'ultimo run
54          err = self.evaluate_error(x_ex, x_new)
55
56          # ottengo l'uso di memoria da parte del
57          # metodo e reinizializzo
58          if trace_memory:
59              usage, peak = tracemalloc.get_traced_memory()
60              tracemalloc.stop()
61
62              # salvo le statistiche e genero il valore
63              # di ritorno della funzione
64              res = Results(nit=nit, err=err, tim=elapsed_time, tol=toll,
                    dim=m, mem=usage, mep=peak)
65
66          else:
67              res = Results(nit=nit, err=err, tim=elapsed_time, tol=toll,
                    dim=m)
68
69          if not trace_memory:
70              self.save_stats(res, "data/computation.json", "gauss-siedl",
                    matrix_name, trace_memory)
71          else:
72              self.save_stats(res, "data/memory computation.json", "gauss-
                    siedl", matrix_name, trace_memory)
73          return res
74
75      @classmethod
76      def resolve_triang_inf(cls, L: NDArray[np.float64], b: NDArray[np.
            float64]):
77          m, n = np.shape(L)
78          x_0 = np.zeros(shape=(m, 1))
79
80          # check che L sia davvero tril
81
82          x_0[0] = b[0] / L[0, 0]
83          for i in range(1, m):
84              x_0[i] = (b[i] - np.dot(L[i, 0:i - 0], x_0[0:i - 0])) / L[i, i
                    ]
85
86          return x_0
```

Codice 1.2: Implementazione del metodo di Gauss-Siedel.

Il metodo di Gauss-Siedel, presentato nell'Algoritmo 1.2, rappresenta un metodo iterativo simile a quello di Jacobi, per la risoluzione del sistema lineare $Ax = b$. L'idea è quella di raggiungere una convergenza più rapida rispetto al primo metodo, basandosi sull'utilizzo, ad ogni iterazione, dei valori già aggiornati della soluzione parziale. Il metodo può essere applicato se: la matrice A è strettamente a dominanza diagonale o è simmetrica definita positiva, SPD. Solo se A è così definita è possibile garantire la convergenza. Nel codice implementato:

- Viene prima eseguito un test di convergenza, verificando se la matrice A soddisfa le

proprietà richieste. Se questo non accade, viene sollevata un'eccezione, bloccando l'esecuzione dell'algoritmo. Viene inizializzata la matrice triangolare inferiore L e successivamente viene calcolata la matrice B=A-L che rappresenta invece la parte superiore. Si inizializza un vettore colonna nullo `x_0` come soluzione iniziale del sistema.

- Viene implementato il ciclo iterativo che verifica ad ogni passo se il metodo ha raggiunto la convergenza. Fino a quando ciò non accade viene aggiornata la soluzione tramite la risoluzione del sistema triangolare inferiore, attraverso la formula `x_new = self.resolve_triang_inf(L, (b - np.dot(B, x_old)))`.
- Al termine del ciclo iterativo, vengono calcolati il tempo di esecuzione, l'errore rispetto alla soluzione esatta e, se richiesto, la memoria occupata. I risultati vengono incapsulati nell'oggetto Results e salvati all'interno del file JSON corrispondente.

Metodo del Gradiente

```
    def solve(self, A: NDArray[np.float64], b: NDArray[np.float64], x_ex:
        NDArray[np.float64], n_max: int, toll: float,
               matrix_name: str, trace_memory: bool) -> Results:

        # verifica la convergenza del metodo
        # forse meglio spostarla nel main
        conv, msg = self.converge(A, 'gradient')

        if not conv:
            raise ValueError(msg)

        m, n = np.shape(A)

        # creo il vettore che rappresenta la
        # soluzione iniziale del sistema
        x_old = np.zeros(shape=(m, 1))
        x_new = x_old

        # definisco il numero d'iterazioni
        # per limitarle
        nit = 0

        start = time.time()

        # applico il metodo del gradiente per calcolare la
        # soluzione del sistema.
        while self.check_iteration(A, x_new, b) > toll and nit < n_max:
            r = b - np.dot(A, x_old)
            k = np.dot(r.T, r) / np.dot(r.T, np.dot(A, r))

            # non applico il dot perche' e' prodotto
            # tra k (scalare) e vettore
            x_new = x_old + k * r
            x_old = x_new

            nit = nit + 1

        # salvo il tempo necessario per il
        # calcolo della convergenza
        stop = time.time()
        elapsed_time = stop - start

        # calcolo l'errore dell'ultimo run
        err = self.evaluate_error(x_ex, x_new)

        # ottengo l'uso di memoria da parte del
        # metodo e reinizializzo
        if trace_memory:
            usage = memory_usage(proc=os.getpid(), max_usage=True)

            # salvo nell'oggetto le
            # statistiche per il return
```

```
52              res = Results(nit=nit, err=err, tim=elapsed_time, tol=toll,
                    dim=m, mem=usage)
53          else:
54              res = Results(nit=nit, err=err, tim=elapsed_time, tol=toll,
                    dim=m)
55
56          if not trace_memory:
57              self.save_stats(res, "data/computation.json", "gradient",
                    matrix_name, trace_memory)
58          else:
59              self.save_stats(res, "data/memory computation.json", "gradient
                    ", matrix_name, trace_memory)
60          return res
```

Codice 1.3: Implementazione del metodo del Gradiente.

Il metodo del gradiente presentato nell'Algoritmo 1.3 rappresenta il codice implementato all'interno della libreria. L'obiettivo del metodo è di minimizzare una determinata funzione obiettivo. Il metodo del gradiente può essere applicato se: $A \in \mathbb{R}^{n \times n}$ è simmetrica e la funzione obiettivo $\phi : \mathbb{R}^n \to \mathbb{R}$ tale che $\phi(\vec{x}) = \frac{1}{2}\,\vec{x}^T \cdot A \cdot \vec{x} - (b^T \cdot \vec{x})$. Di conseguenza il gradiente di $\phi(\vec{x}) = A \cdot \vec{x} - \vec{b}$ conterrà tutte le derivate parziali della funzione. I punti stazionari di una funzione hanno derivata prima nulla, di conseguenza rappresentano il punto che minimizza la funzione obiettivo ed è la soluzione del sistema. La soluzione calcolata dal metodo iterativo si avvicina sempre di più ad ogni iterazione alla soluzione esatta, in particolare di avvicina di un determinato passo, α, verso il residuo, r, che rappresenta la direzione massima di discesa del gradiente. Ad ogni iterazione vengono nuovamente calcolati il passo e il residuo.

Di seguito è presente una breve descrizione del funzionamento del codice:

- Viene eseguito un test di convergenza in base al metodo iterativo applicato, in questo caso si controlla se la matrice A in input è simmetrica e definita positiva. Se la matrice in input non supera la validazione viene sollevata un'eccezione perché il metodo non convergerà per qualsiasi $\vec{x}$ iniziale. Di conseguenza `x_old` non può essere inizializzato come vettore colonna composto da soli 0. Successivamente vengono inizializzate alcune variabili come il numero di colonne e righe della matrice in input, il numero di iterazioni e la soluzione iniziale del sistema.
- Successivamente si verifica se il metodo iterativo è arrivato a convergenza. Si possono svolgere due scenari: il primo, la convergenza è rispettata e il ciclo termina. Il secondo, la convergenza non è rispettata, di conseguenza viene calcolato il residuo r e il passo ($\alpha =$ `k`). In ultimo viene aggiornata la soluzione `x_new` sfruttando il passo, il residuo e la soluzione calcolata all'iterazione precedente.
- Al termine del ciclo iterativo viene calcolato: il tempo di esecuzione, l'errore tra la soluzione esatta e quella calcolata e se necessario la memoria occupata. Infine viene creato l'oggetto `Results` e vengono salvate le statistiche all'interno del file JSON corretto.

Metodo del Gradiente Coniugato

```
        def solve(self, A: NDArray[np.float64], b: NDArray[np.float64],
            x_ex: NDArray[np.float64], n_max: int, toll: float,
                matrix_name: str, trace_memory: bool) -> Results:

        # verifica la convergenza del metodo
        # forse meglio spostarla nel main
        conv, msg = self.converge(A, 'conjugated-gradient')

        if not conv:
            raise ValueError(msg)

        # prelevo la size della matrice
        m, n = np.shape(A)

        # creo il vettore che rappresenta la
        # soluzione iniziale del sistema
        x_old = np.zeros(shape=(m, 1))
        x_new = x_old

        # calcolo il residuo iniziale
        r_old = b - np.dot(A, x_old)
        p_old = r_old

        # definisco il numero d'iterazioni
        # per limitarle
        nit = 0

        beta = 0

        start = time.time()

        # applico il metodo del gradiente coniugato per calcolare la
        # soluzione del sistema.
        while self.check_iteration(A, x_new, b) > toll and nit < n_max:
            k = np.dot(p_old.T, r_old) / np.dot(p_old.T, np.dot(A, p_old))
            x_new = x_old + (k * p_old)

            r_new = r_old - (k * (np.dot(A, p_old)))
            beta = np.dot((np.dot(A, p_old)).T, r_new) / np.dot(np.dot(A,
                p_old).T, p_old)

            p_new = r_new - (beta * p_old)

            x_old = x_new
            r_old = r_new
            p_old = p_new

            nit = nit + 1

        # salvo il tempo necessario per il
        # calcolo della convergenza
        stop = time.time()
```

```
51          elapsed_time = stop - start
52
53          # calcolo l'errore dell'ultimo run
54          err = self.evaluate_error(x_ex, x_new)
55
56          # ottengo l'uso di memoria da parte del
57          # metodo e reinizializzo
58          if trace_memory:
59              usage =  memory_usage(proc=os.getpid(), max_usage=True)
60
61              # salvo nell'oggetto le
62              # statistiche per il return
63              res = Results(nit=nit, err=err, tim=elapsed_time, tol=toll,
                    dim=m, mem=usage)
64          else:
65              res = Results(nit=nit, err=err, tim=elapsed_time, tol=toll,
                    dim=m)
66
67
68          if not trace_memory:
69              self.save_stats(res, "data/computation.json", "conjugated-
                    gradient", matrix_name, trace_memory)
70          else:
71              self.save_stats(res, "data/memory computation.json", "
                    conjugated-gradient", matrix_name, trace_memory)
72          return res
```

Il metodo del Gradiente Coniugato è una variante del metodo del Gradiente, si distingue per la gestione indipendente del calcolo del residuo e del passo. In particolare, per ciascuna direzione di discesa viene scelto il passo che minimizza la funzione obiettivo lungo quella direzione. Inoltre, una volta percorsa una direzione coniugata, il metodo non vi ritorna più nelle iterazioni successive. Questa caratteristica consente di evitare il comportamento di convergenza a *zig-zag* tipico del metodo del gradiente, dovuto alla dipendenza tra passo e direzione nei metodi classici. La convergenza a *zig-zag* spesso si verifica quando il condizionamento della matrice è particolarmente elevato. L'algoritmo introdotto è stato scritto nel seguente modo:

- Viene eseguito il test di convergenza che è del tutto identico al test eseguito per il metodo del gradiente. Dopodiché vengono istanziati: un vettore che rappresenta la soluzione, il residuo, il passo, il numero di iterazioni e il parametro β, simile al passo α, che viene impiegato per il calcolo della direzione.
- Successivamente si verifica se il metodo iterativo è arrivato a convergenza e si possono svolgere due scenari: il primo, la convergenza è rispettata e il ciclo termina. Il secondo, la convergenza non è rispettata, di conseguenza viene calcolato il residuo r, il passo ($\alpha =$ `k`) e il coefficiente β per costruire la nuova direzione di discesa. Infine, viene calcolata la nuova direzione di discesa p e aggiornata la soluzione `x_new` sfruttando il passo, la direzione di discesa e la soluzione calcolata all'iterazione precedente. È bene osservare che il numero di iterazioni del metodo del Gradiente Coniugato è deterministico, questa caratteristica è unica rispetto ai metodi iterativi analizzati.

- Come già accaduto per i metodi precedenti al termine del ciclo iterativo viene calcolato: il tempo di esecuzione, l'errore tra la soluzione esatta e quella calcolata e se necessario la memoria occupata. Infine viene creato l'oggetto `Results` e vengono salvate le statistiche all'interno del file JSON corretto.

1.2.4 Impiego della libreria

La libreria consente l'esecuzione di diversi metodi iterativi su matrici sparse, permettendo una personalizzazione estesa dei parametri di esecuzione. **N.B.** l'unico parametro obbligatorio è il path al file `.mtx` che contiene la matrice salvata in formato standard.

```
python3 main.py [opzioni]
```

Di seguito vengono elencate tutte le opzioni disponibili, con descrizione completa e comandi d'esempio.

Parametri da riga di comando

`--path, -p`

```
python3 main.py -p matrix/spa1.mtx matrix/vem1.mtx
```

Permette di specificare il percorso delle matrici su cui eseguire i metodi iterativi. Se non viene fornito, non verrà eseguito alcun metodo. La directory predefinita è `matrix/`, che contiene file come `spa1.mtx`, `vem2.mtx`, ecc.

Esempi:

- `python3 main.py -p matrix/spa1.mtx`
- `python3 main.py -p to/your/custom/matrix.mtx`

`--method, -m`

```
python3 main.py -p matrix/spa1.mtx -m jacobi
```

Specifica quali metodi iterativi applicare. Se non indicato, vengono eseguiti tutti nell'ordine: Jacobi, Gauss-Siedel, Gradiente e Gradiente Coniugato.

Esempi:

- `-m jacobi gauss-siedel`
- `-m gradient`

`--toll, -t`

```
python3 main.py -p matrix/spa1.mtx -t 1E-4
```

Imposta la tolleranza per l'arresto dell'algoritmo iterativo. Default: `1E-4`, `1E-6`, `1E-8`, `1E-10`. Il solver iterativo scelto viene eseguito con tutte le tolleranze. È possibile specificarne di differenti.

Esempi:

- `-t 1E-5`
- `-t 1E-4 1E-6 1E-10`
- `-t 1E-4 1E-7 1E-13`

`--niter, -ni`

```
python3 main.py -p matrix/spa1.mtx -ni 30000
```

Definisce il numero massimo di iterazioni per ciascun metodo. Default: `20.000` iterazioni.

`--nrun, -nr`

```
python3 main.py -p matrix/spa1.mtx -nr 5
```

Esegue ciascun metodo più volte (run multipli), è utile per evitare problemi (benchmark non corretti come il tempo di convergenza) dovuti a fluttuazioni occasionali. Tutti i risultati vengono comunque salvati in `computation.json` all'interno della cartella `data`. Il valore di default è `1` run per solver.

`--chart, -c`

```
python3 main.py -p matrix/spa1.mtx -c
```

Genera automaticamente grafici riassuntivi dei run eseguiti alla fine dell'esecuzione dei solver. Come default non viene *generato nessun grafico*.

`--stats, -s`

```
python3 main.py -p matrix/spa1.mtx -s
```

Genera grafici riassuntivi di tutti i run eseguiti utilizzando i dati salvati in `computation.json`. La funzione `stats` è molto utile per analisi dati su un grande numero di run.

`--tracememory, -tm`

```
python3 main.py -p matrix/spa1.mtx -tm
```

Traccia l'utilizzo di memoria per ciascun metodo e salva i dati in `memory_computation.json` all'interno della cartella data. Come default non viene eseguita alcuna analisi di memoria. Quando il `tracememory` è impiegato le statistiche raccolte non vengono salvate in `computation.json`.

`--verb, -v`

```
python3 main.py -p matrix/spa1.mtx -v
```

Abilita la stampa di messaggi informativi su terminale (setup iniziale, stato dei run, risultati parziali). Come default *non viene prodotto alcun messaggio informativo.*

N.B.

È possibile eseguire il programma specificando differenti parametri contemporaneamente. L'ordine con cui vengono definiti non è importante.

```
python3 main.py -p matrix/spa1.mtx -m jacobi gauss-siedel -v -s
```

Capitolo 2

Validazione dei metodi iterativi

I metodi iterativi introdotti nel capitolo precedente sono stati validati sfruttando il medesimo elaboratore con le seguenti caratteristiche:

Componenti	Caratteristiche
CPU	Ryzen 7 4700U
RAM	16GB DDR4 3200MHz
OS	Ubuntu 24.04

Tabella 2.1: caratteristiche dell'elaboratore impiegato nell'esecuzione dei vari solver iterativi presentati.

Ciascun solver è stato eseguito impiegando le matrici: *spa1, spa2, vem1* e *vem2* con diverse tolleranze (10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}) applicate all'errore tra soluzione esatta e calcolata. Nel caso in cui il metodo iterativo non converga entro la tolleranza specificata, è stato impostato un limite massimo di 20.000 iterazioni per ciascun metodo.

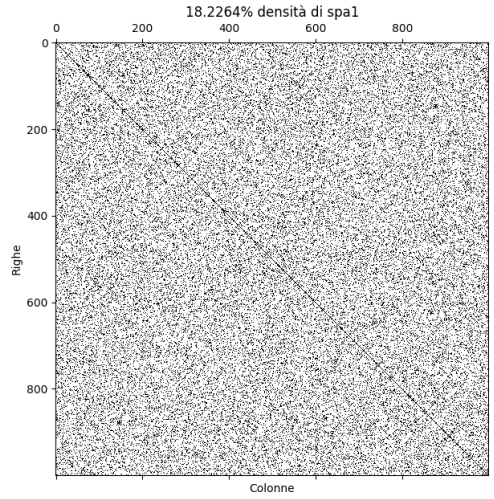
2.1 Descrizione delle matrici

Per eseguire una migliore analisi dei metodi iterativi è stato necessario osservare come fossero composte le matrici sfruttate in fase di validazione. In particolare sono state osservate:

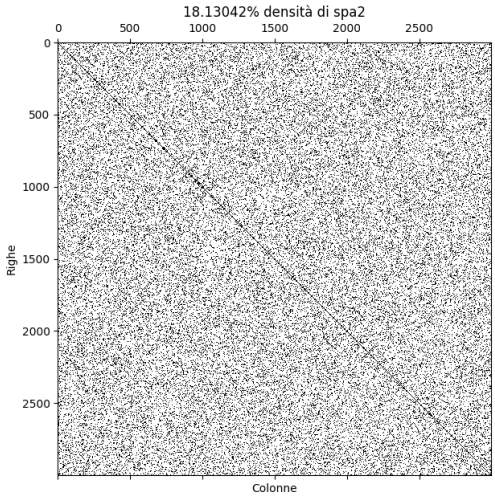
- la **densità delle matrici**, ovvero la percentuale di entry della matrice che contengono numeri maggiori di 10^{-10};
- la **densità** di **dominanza diagonale**. Una matrice quadrata è detta a dominanza diagonale se e solo se per ogni riga $i = 1, ..., n$ vale la disuguaglianza in Formula 2.1.

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \tag{2.1}$$

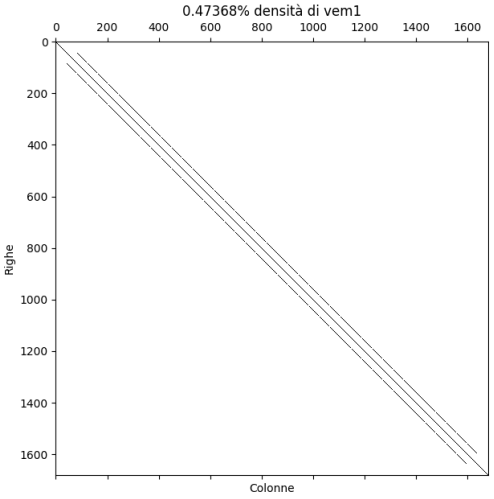
  In particolare per calcolare questa quantità è stato valutato il rapporto tra numero di righe che rispettano la definizione di dominanza diagonale e il numero totale di righe della matrice.



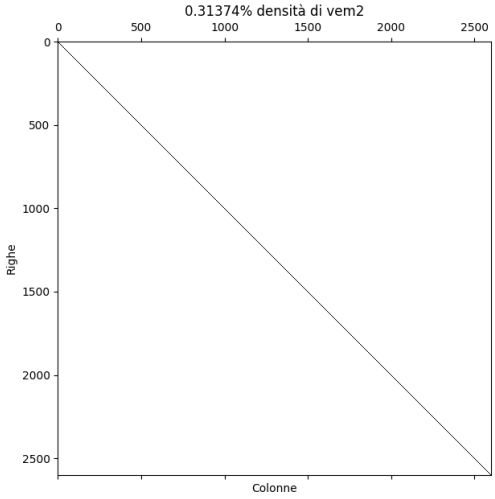
(a) Densità di spa1, la matrice ha un numero di entry con valore maggiore di 10^{-10} è pari al 18,23%



(b) Densità di spa2, la matrice ha un numero di entry con valore maggiore di 10^{-10} è pari al 18,13%



(c) Densità di vem1, la matrice ha un numero di entry con valore maggiore di 10^{-10} è pari al 0,47%



(d) Densità di vem2, la matrice ha un numero di entry con valore maggiore di 10^{-10} è pari al 0,31%

Figura 2.1: Densità delle matrici analizzate. Le matrici spa1 e spa2 presentano una densità del tutto comparabile, analogamente a quanto osservato per vem1 e vem2. Tuttavia spa1 e spa2 risultano molto più dense rispetto a vem1 e vem2.

La Figura 2.1 mostra chiaramente come spa1 e spa2 presentano una densità tra loro molto simile, così come accade per vem1 e vem2. Tuttavia spa1 e spa2 sono molto più dense rispetto a vem1 e vem2.

2.2 Il metodo di Jacobi

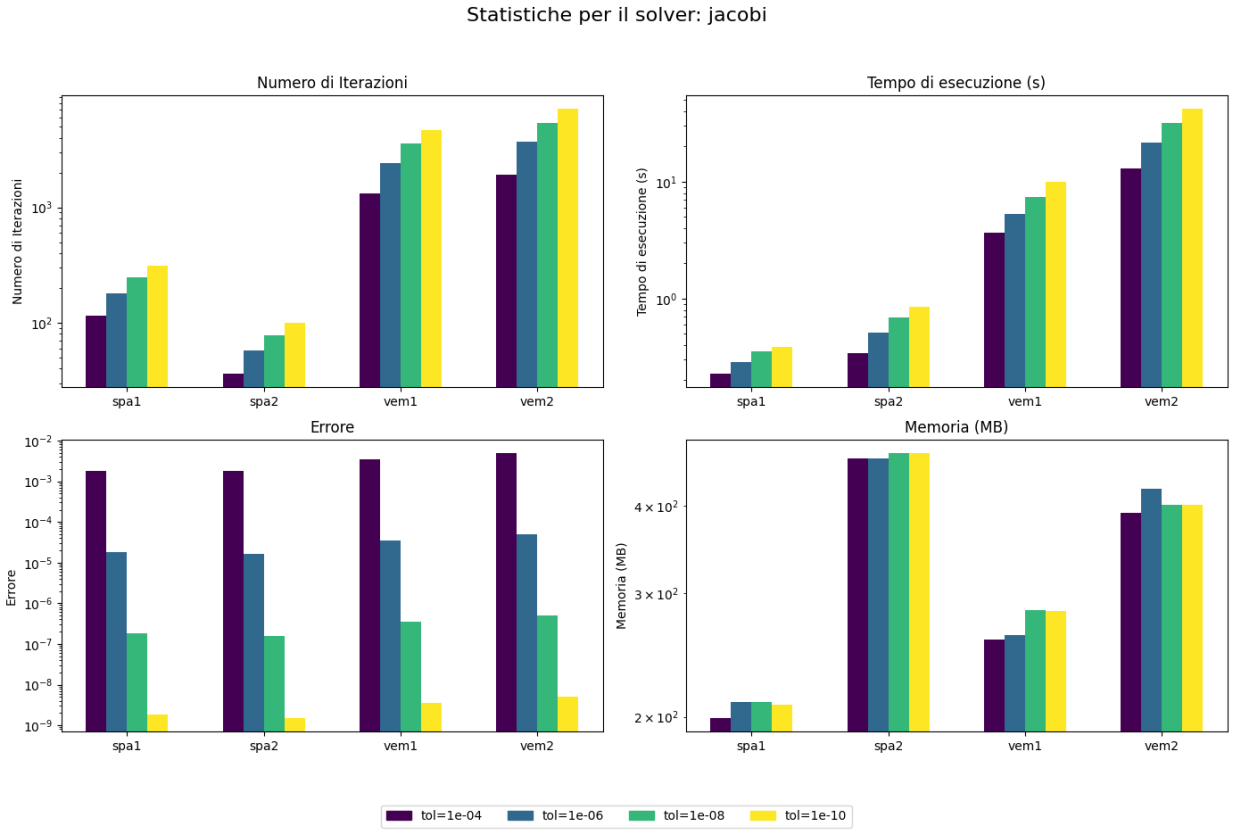


Figura 2.2: Figura riassuntiva dei run eseguiti su ciascuna matrice per ogni tolleranza con il metodo di Jacobi.

Dall'analisi dei risultati di Figura 2.2 emerge che la tolleranza impostata non ha effetto significativo sull'occupazione di memoria dei metodi iterativi. Sebbene una tolleranza più piccola comporti un numero maggiore di iterazioni, quest'ultime non richiedono l'allocazione di strutture dati aggiuntive o di dimensioni superiori. Infatti, i metodi sono stati implementati per riutilizzare la memoria già allocata, sovrascrivendo le strutture dati esistenti durante l'esecuzione.

Il metodo di Jacobi si dimostra particolarmente efficiente in termini di tempo quando applicato alle matrici spa1 e spa2, che vengono risolte in meno della metà del tempo necessario per vem1 e vem2. È possibile trovare una giustificazione alle prestazioni descritte sfruttando il teorema di convergenza del metodo di Jacobi. Quest'ultimo converge sicuramente se la matrice che rappresenta il sistema è a dominanza diagonale, tuttavia se la condizione descritta non è verificata il solver potrebbe convergere ugualmente perché è una condizione necessaria ma non sufficiente. In fase di test tutte le matrici prese in considerazione (spa1, spa2, vem1 e vem2) sono risultate non essere a dominanza diagonale, tuttavia avere un maggior numero di righe che rispettano la proprietà di dominanza diagonale potrebbe facilitare la risoluzione

del sistema perché viene limitata la propagazione degli errori tra le iterazioni ed è garantita una certa stabilità numerica.

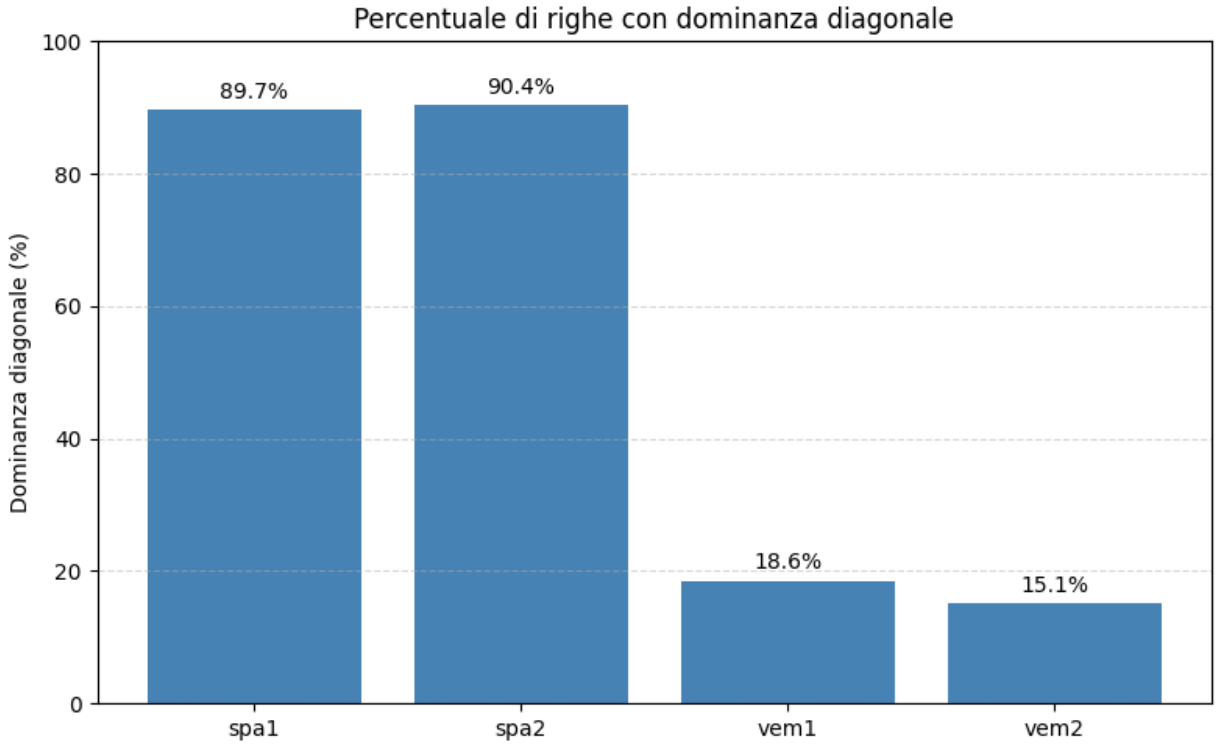


Figura 2.3: Percentuale di righe che rispettano la proprietà di dominanza diagonale. spa1 e spa2 hanno un numero di righe che rispetta la proprietà di dominanza diagonale che è circa 4.5 volte superiore rispetto a vem1 e vem2.

Osservando la percentuale di righe che sono a dominanza diagonale all'interno delle matrici analizzate Figura 2.3 è chiaro come spa1 e spa2 abbiano un numero di righe che rispettano la proprietà di dominanza diagonale che è circa 4.5 volte superiore rispetto a vem1 e vem2. Di conseguenza è facile intuire che le ottime prestazioni ottenute su spa1 e spa2 dal metodo di Jacobi dipendano strettamente da questa caratteristica descritta.

Un'ulteriore osservazione interessante può essere il numero di iterazioni necessario per ottenere la convergenza di spa1 e spa2. La seconda matrice analizzata ha circa il doppio delle righe rispetto a spa1 e intuitivamente impiega sicuramente più tempo per convergere ma, per ogni tolleranza, sfrutta un numero inferiore di iterazioni. Anche questo fatto può essere dovuto alle osservazioni che riguardano la quantità di righe che godono della proprietà di dominanza diagonale. In particolare, per la tolleranza massima, spa2 impiega circa $\frac{1}{3}$ delle iterazioni di spa1 nonostante abbia solo lo 0.6% di righe a dominanza in più rispetto a spa1. Un chiaro segnale dell'importanza di questa proprietà nell'efficienza del metodo di Jacobi.

2.3 Il metodo Gauss-Siedel

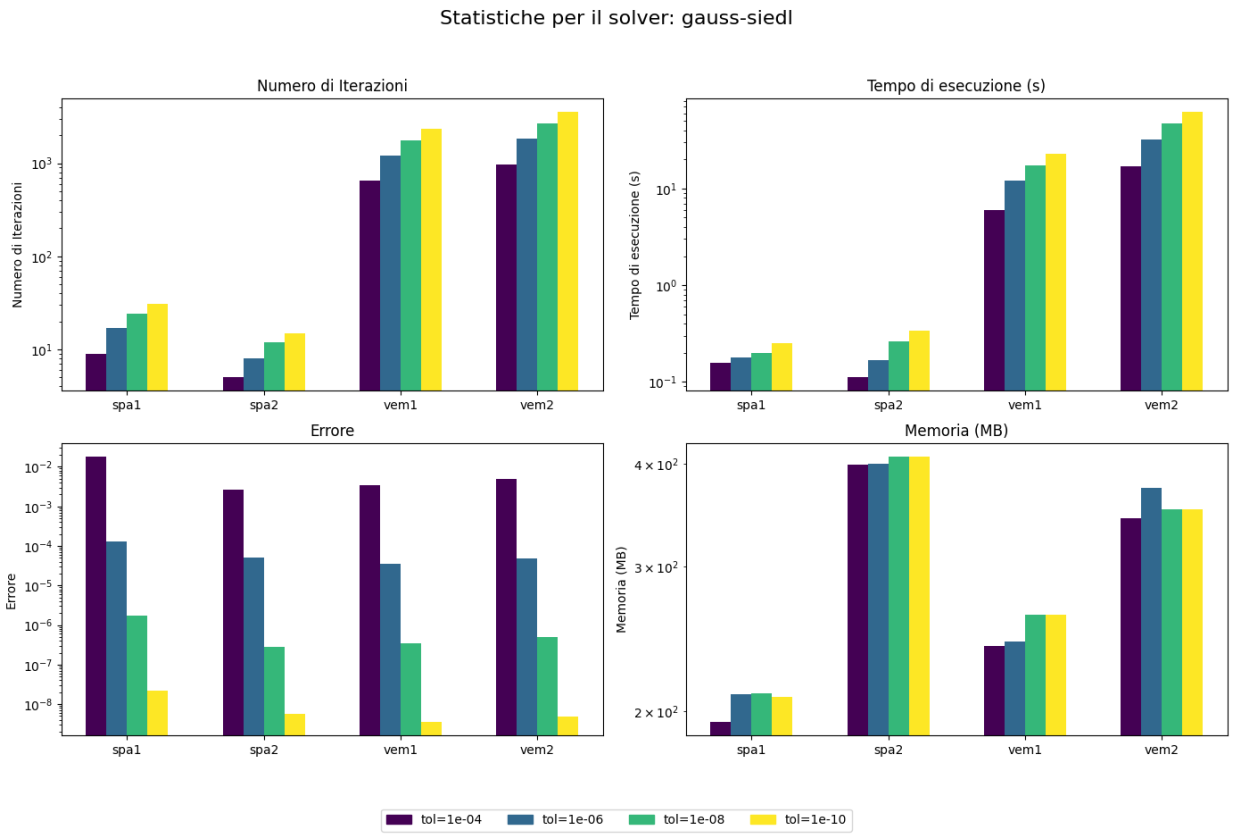


Figura 2.4: Figura riassuntiva dei run eseguiti su ciascuna matrice per ogni tolleranza con il metodo di Gauss-Siedel.

Il metodo di Gauss-Siedel è una variante del metodo di Jacobi, di conseguenza le affermazioni precedenti sono valide anche per questo solver iterativo. Tuttavia, rispetto a Jacobi il metodo di Gauss-Siedel ha prestazioni migliori in termini di tempo di convergenza e numero di iterazioni per le matrici spa1 e spa2. L'aumento delle performance descritte è dovuto al calcolo della soluzione tramite l'impiego di altri valori già calcolati durante la risoluzione del sistema mediante un metodo diretto.

Per le matrici spa1 e spa2, il metodo di Jacobi ha prodotto un errore relativo più basso rispetto a Gauss-Seidel. Una possibile spiegazione risiede nel criterio di arresto usato che è basato sul residuo relativo di Formula 2.2:

$$\frac{\|Ax^{(k)} - b\|_2}{\|b\|_2} \tag{2.2}$$

Questo criterio non misura direttamente quanto $x^{(k)}$, la soluzione calcolata alla $k - esima$ iterazione sia vicino alla soluzione esatta `x_ex`. Di conseguenza, potrebbe accadere che Gauss-Seidel si fermi prima, anche se l'errore è ancora elevato.

2.4 Il metodo del Gradiente

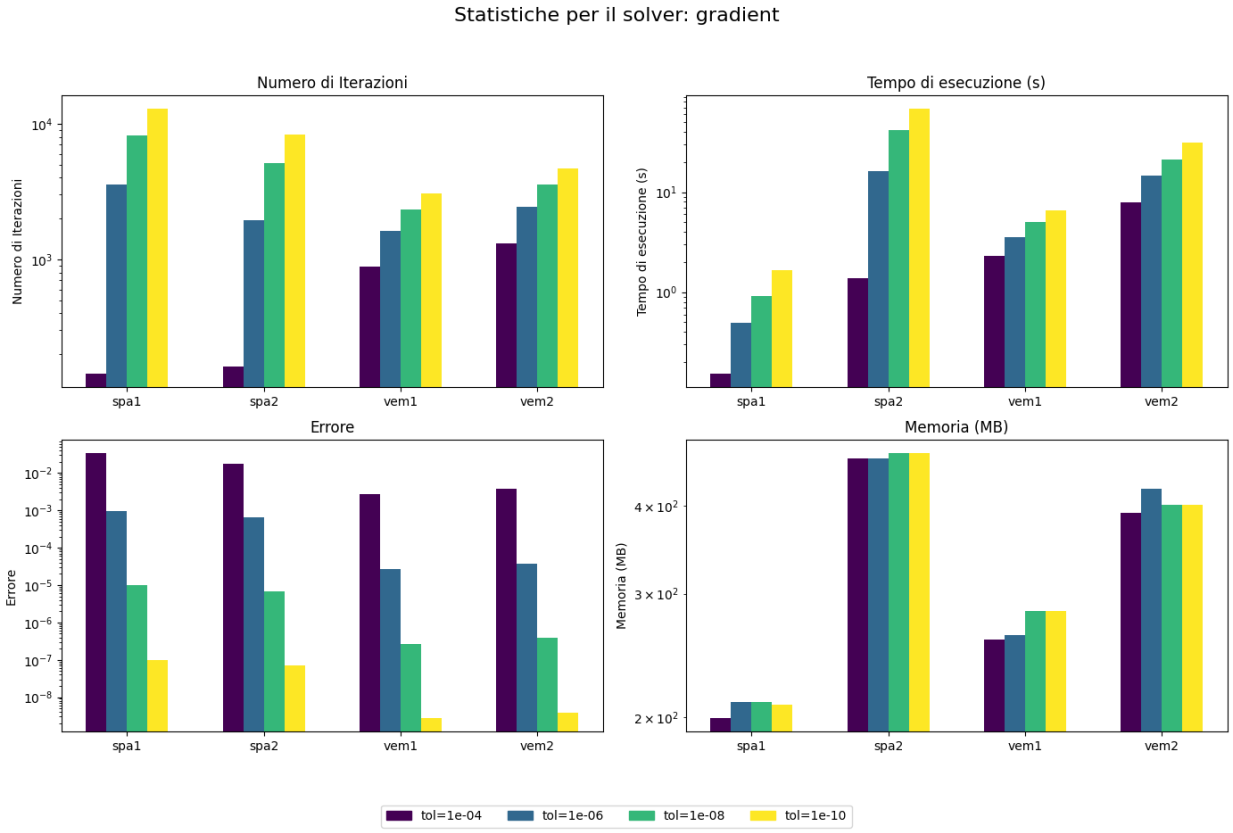


Figura 2.5: Figura riassuntiva dei run eseguiti su ciascuna matrice per ogni tolleranza con il metodo del Gradiente

Analizzando i risultati mostrati in Figura 2.5 si osserva che a differenza dei metodi studiati precedentemente (Jacobi e Gauss-Siedel), il metodo del gradiente di un grande numero di iterazioni e di tempo computazionale per convergere. Questo accade soprattutto su spa1 e spa2 che ottenevano ottimi risultati con i solver visti precedentemente. Il comportamento descritto è causato dal fatto che la convergenza del metodo del Gradiente non dipende dalla dominanza diagonale della matrice ma dal condizionamento della matrice. Quest'ultimo non è altro che un numero che misura quanto la soluzione della matrice, A, è sensibile rispetto a variazioni dell'input. Può essere definito in due modi differenti:

$$c(A) = \|A\|_2 \cdot \|A^{-1}\|_2 \tag{2.3}$$

La Formula 2.3 non viene applicata in numerica perché il calcolo dell'inversa è oneroso, di conseguenza è possibile valutare il numero di condizionamento sfruttando la formula Formula 2.4 se e solo se la matrice A è simmetrica e definita positiva.

$$c(A) = \frac{\lambda_{min}}{\lambda_{max}} \tag{2.4}$$

Dato che il metodo del gradiente converge se e solo se la matrice A è simmetrica e definita positiva allora si può impiegare la Formula 2.4 per calcolare il numero di condizionamento.

Matrice	Condizionamento
spa1	2048.15
spa2	1411.97
vem1	324.64
vem2	507.02

Tabella 2.2: Numeri di condizionamento per le matrici valutate. vem1 e vem2 dovrebbero essere matrici con una soluzione computabile con un errore inferiore rispetto a spa1 e spa2.

La Tabella 2.2 mostra che le matrici vem1 e vem2 hanno un numero di condizionamento molto più basso rispetto a spa1 e spa2. Dato che il metodo del Gradiente converge più facilmente su matrici ben condizionate il minor numero di iterazioni richiesto per vem1 e vem2 è giustificabile. Anche l'andamento dell'errore conferma questa osservazione, per ogni tolleranza considerata, il metodo del Gradiente trova soluzioni più accurate su vem1 e vem2.

Il tempo di convergenza più basso di spa1 è dovuto probabilmente alle sue dimensioni ridotte rispetto a spa2, vem1 e vem2. Le dimensioni di spa2 sono paragonabili a quelle di vem2, ma i risultati ottenuti sono peggiori. L'affermazione fatta va a dimostrare il collegamento tra metodo del gradiente e dipendenza dal condizionamento della matrice.

2.5 Il metodo del Gradiente Coniugato

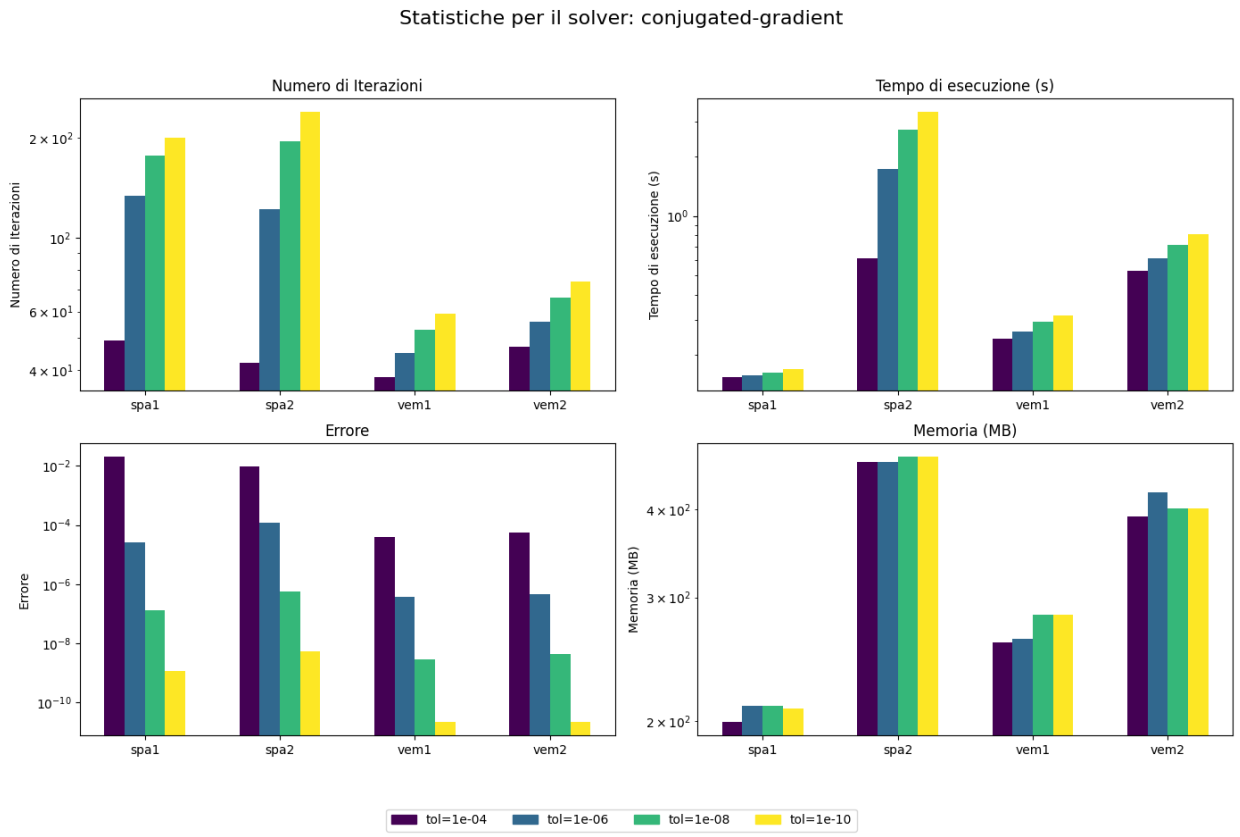


Figura 2.6: Figura riassuntiva dei run eseguiti su ciascuna matrice per ogni tolleranza con il metodo del Gradiente Coniugato.

Il metodo del Gradiente Coniugato è derivato dal metodo del Gradiente, di conseguenza le considerazioni fatte per il metodo del Gradiente valgono anche per il metodo del Gradiente Coniugato. Tuttavia è importante porre particolare attenzione alla riduzione degli errori, numero di iterazioni e tempo di convergenza, per ciascuna matrice presa in considerazione (spa1, spa2, vem1 e vem2).

Il miglioramento delle prestazioni è dovuto al fatto che il metodo del Gradiente coniugato una volta scelto il passo che minimizza la funzione obiettivo percorre la direzione coniugata e non vi ritorna più nelle iterazioni successive. Il metodo del Gradiente non compie l'operazione descritta, di conseguenza ad ogni iterazione prende in considerazione la direzione presa nell'iterazione precedente creando una convergenza a *"zig-zag"* che rallenta le prestazioni del solver in termini di iterazioni e tempo di convergenza. Inoltre aumenta l'errore.

Capitolo 3

Conclusioni

3.1 spa1

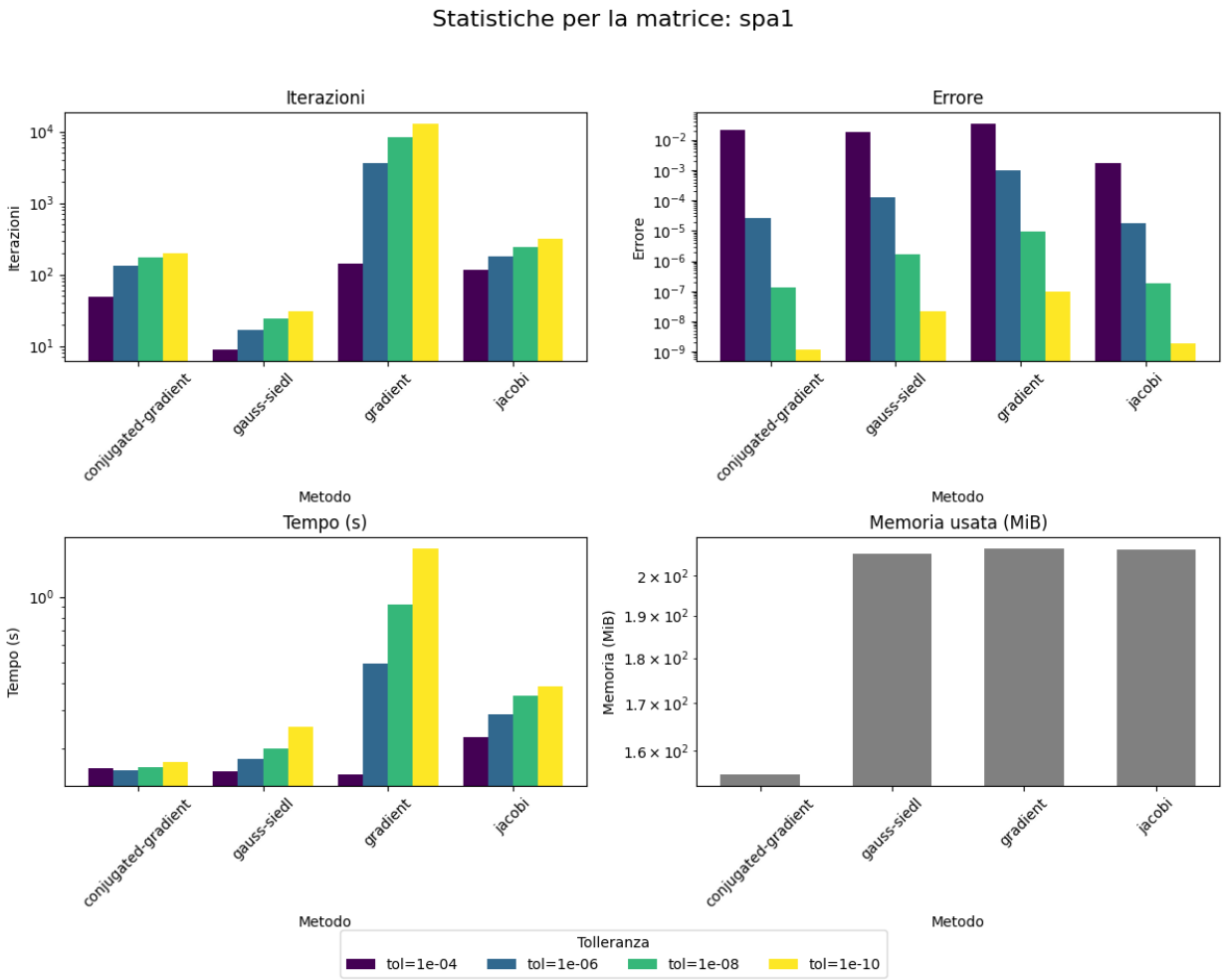


Figura 3.1: Prestazione dei solver iterativi rispetto alla matrice spa1, per ciascuna tolleranza.

Sfruttando le analisi effettuate nel capitolo precedente dalla Figura 3.1 si evince che Gauss-Siedel è il solver che riesce a convergere in un numero minore di iterazioni rispetto a tutti i metodi analizzati. Tuttavia, presenta un tempo di convergenza e un errore maggiore rispetto al metodo del Gradiente Coniugato.

Il tempo di convergenza del metodo del Gradiente Coniugato risulta inferiore di qualche decimo di secondo rispetto a Gauss-Seidel, nonostante la complessità temporale per iterazione sia $O(n^2)$ in entrambi i casi. Questa differenza può dipendere sia dalle ottimizzazioni implementate sia dalla dimensione della matrice. Nel primo caso, il metodo del gradiente coniugato sfrutta operazioni ottimizzate, come il prodotto matrice-vettore implementato nella libreria `NumPy`, che beneficia di algoritmi efficienti. Al contrario, Gauss-Seidel è un metodo iterativo sequenziale che si basa su un metodo diretto e non è facilmente parallelizzabile, limitando così le possibilità di ottimizzazione. Tuttavia, Gauss-Seidel non richiede di salvare una soluzione di appoggio per ogni iterazione, il che può ridurre l'uso di memoria, ma questo vantaggio non compensa la minore efficienza computazionale. Nel secondo caso, la matrice ha dimensione 10001000 e il metodo del Gradiente Coniugato potrebbe incontrare meno difficoltà nel convergere, nonostante l'elevato condizionamento, grazie alla capacità di riduzione dell'errore e all'efficienza delle operazioni numeriche coinvolte.

Il metodo di Jacobi riduce l'errore per le motivazioni già espresse in Sezione 2.3.

3.2 spa2

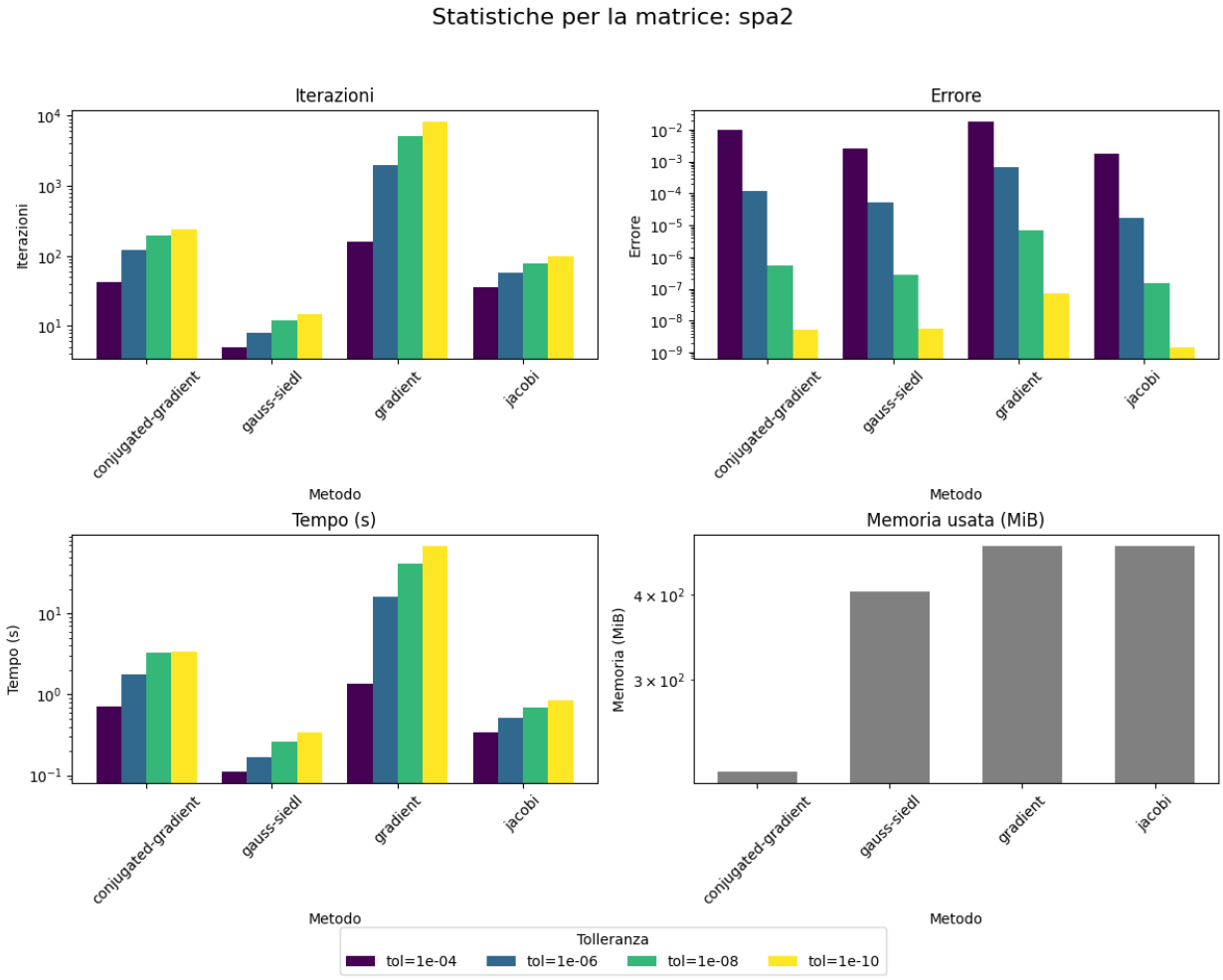


Figura 3.2: Prestazione dei solver iterativi rispetto alla matrice spa2, per ciascuna tolleranza.

In Figura 3.2 Gauss-Siedel è il solver migliore, seguito da Jacobi, che ha un ottimo comportamento per risolvere il sistema associato alla matrice spa2. Le buone prestazioni sono dovuto alle motivazioni introdotte in Sezione 2.3.

Il metodo del Gradiente Coniugato ha l'unico vantaggio di ridurre la memoria occupata. Il metodo di Jacobi come al solito riduce l'errore calcolato con il metodo di Gauss-Siedel per le motivazioni specificate nella Sezione 2.3.

3.3 vem1

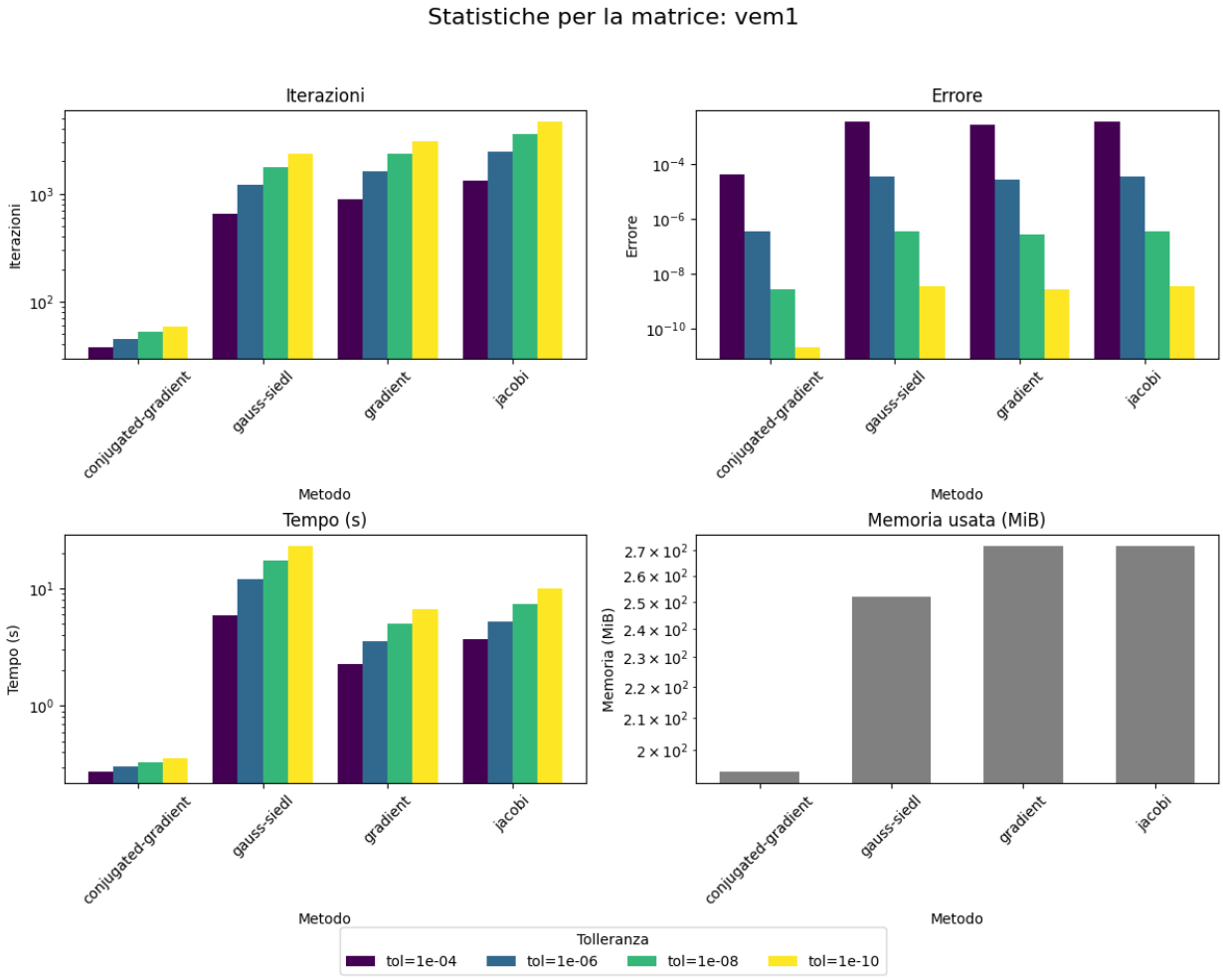


Figura 3.3: Prestazione dei solver iterativi rispetto alla matrice vem1, per ciascuna tolleranza.

La Figura 3.3 mostra che il metodo del Gradiente presenta prestazioni simili a quelle di Gauss-Seidel e Jacobi, nonostante la matrice vem1 abbia solo il 18.6% di righe a dominanza diagonale. Il metodo del Gradiente Coniugato si dimostra invece decisamente più efficiente in termini di iterazioni, errore, tempo di convergenza e memoria impiegata rispetto agli altri solver analizzati.

Rispetto al metodo del Gradiente, il Gradiente Coniugato mostra un comportamento molto più efficace, poiché il condizionamento di vem1 riportato nella Tabella 2.2 provoca nel metodo del gradiente una convergenza dell'errore a *zig-zag*, rallentandone il progresso verso la soluzione.

3.4 vem2

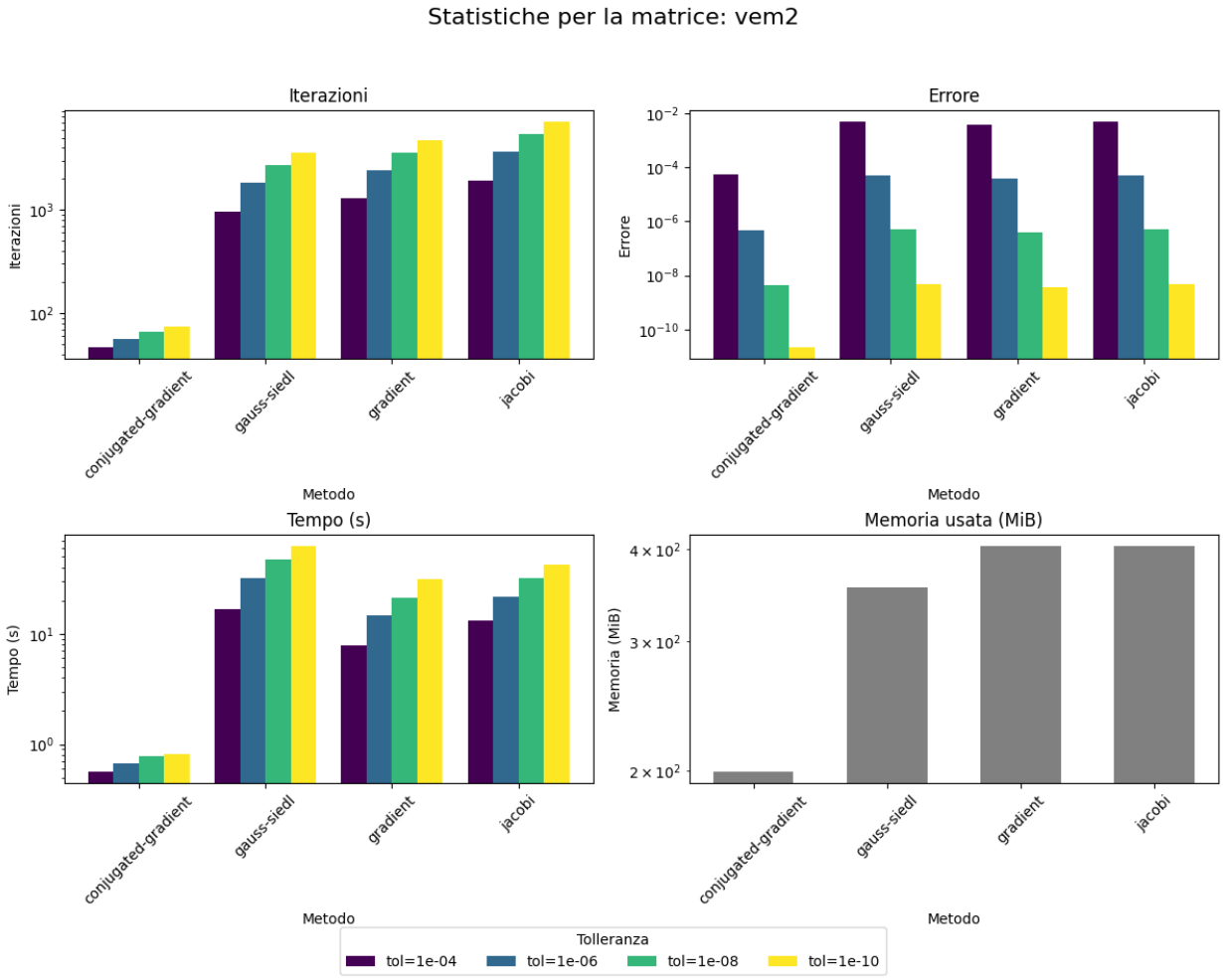


Figura 3.4: Prestazione dei solver iterativi rispetto alla matrice vem2, per ciascuna tolleranza.

Anche per la Figura 3.4 valgono le osservazioni fatte per la matrice vem1. Il metodo del Gradiente Coniugato ha prestazioni migliori. Il metodo del Gradiente aumenta il numero di iterazioni, questo è dovuto ad un maggior numero di condizionamento di vem1 rispetto a vem2, osservabile in Tabella 2.2.

3.5 Considerazioni finali

Dall'analisi effettuata si può affermare che, per matrici con un'elevata percentuale di righe a dominanza diagonale, i metodi iterativi di Jacobi e Gauss-Seidel risultano i più indicati, in particolare Gauss-Seidel è capace di ridurre il numero di iterazioni necessarie per la convergenza. Tuttavia, se è richiesta una maggiore precisione, il metodo di Jacobi è preferibile grazie al minor errore. Per matrici con una bassa percentuale di righe a dominanza diagonale il metodo del Gradiente Coniugato si dimostra il più adatto, grazie alle sue ottime prestazioni. Quest'ultimo metodo non risente dell'elevato condizionamento della matrice perché elimina la convergenza dell'errore a *zig-zag* tipica del metodo del Gradiente quando applicato a matrici con un alto numero di condizionamento.