Università degli Studi di Milano-Bicocca

Scuola di Scienze Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea Magistrale in Informatica

Progetto di Metodi del Calcolo Scientifico

MCS-progetto-1-alternativo

Anno Accademico 2024-2025

Relazione a cura di:

Oleksandra Golub

Matricola:

856706

Sommario

1. Introduzione

Lo scopo del progetto è realizzare una mini-libreria open-source per la soluzione di sistemi lineari **Ax=b** con matrici **simmetriche e definite positive (SPD)**, implementando da zero i quattro metodi iterativi classici:

- **Jacobi;**
- **Gauss-Seidel;**
- **Gradiente**;
- **Gradiente Coniugato**;

L'unica funzionalità richiesta alla libreria esterna è la gestione di vettori/matrici (struttura dati e operazioni elementari), **senza** usare routine già pronte per risolvere sistemi lineari. Il criterio d'arresto è sul **residuo relativo:**

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

con un ulteriore limite di iterazioni **maxIter ≥ 20000**.

La validazione prevede test con **x = [1,1,…,1]**, **b = Ax**, più valori di **tol** e report di **errore relativo, iterazioni e tempo** per ogni metodo.

1.1 Scelte tecnologiche

L'implementazione è in **Python**. Si usano:

- **NumPy** per operazioni vettoriali/matriciali di base (prodotti, somme, norme).
- **SciPy solo** per il caricamento delle matrici **Matrix Market (.mtx)** tramite **scipy.io.mmread**. Tutte le routine **Jacobi, Gauss-Seidel, Gradiente, Gradiente Coniugato** sono state **scritte da zero**, rispettando il vincolo della consegna che vieta l'uso di solutori integrati.

Nota sul formato .mtx: l'indicizzazione dei file è **1-based**; mmread gestisce automaticamente la conversione a **0-based**. È stato verificato con un test dedicato incluso nel repository.

1.2 Procedura di validazione

Per ogni matrice SPD fornita, si costruisce il sistema di prova con **x* = 1 e b = Ax***, poi si eseguono i quattro metodi partendo dal **vettore iniziale nullo**. La procedura è ripetuta per le tolleranze:

$$\texttt{tol} \in \{10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}\}$$

e, per ogni run, si riportano **numero di iterazioni, tempo di calcolo, errore relativo** tra la soluzione numerica e **x***.

1.3 Dati di test (.mtx)

Sono state utilizzate le quattro matrici sparse fornite in consegna:

- **spa1.mtx:** 1000×1000 sparsità ≈ **81.77%**
- **spa2.mtx:** 3000×3000 sparsità ≈ **81.87%**
- **vem1.mtx:** 1681×1681 sparsità ≈ **99.53%**
- **vem2.mtx:** 2601×2601 sparsità ≈ **99.69%**

Le matrici sono simmetriche e (numericamente) definite positive; il caricamento avviene da file **.mtx** e i test seguono la procedura standard descritta nella consegna.

2. Architettura della libreria

La libreria è una mini–toolbox **pure-Python** per la soluzione di sistemi lineari **Ax=b** con matrici **simmetriche e definite positive (SPD)**.

È composta da un **core con i solutori iterativi** e da **moduli di infrastruttura** (I/O delle matrici, generazione dei sistemi di test, salvataggio risultati e grafici). Le uniche dipendenze sono *NumPy* (operazioni vettoriali/matriciali) e *SciPy* limitatamente al **caricamento** dei file Matrix Market **.mtx**.

2.1 src/linear_solvers.py

È la classe **LinearIterativeSolver** che incapsula i 4 metodi iterativi (Jacobi, Gauss-Seidel, Gradiente, Gradiente Coniugato) e alcune utility comuni.

- **costruttore __init__(max_iter=20000, verbose=True)** imposta quante iterazioni al massimo può fare un metodo e se stampare messaggi.

- **_check_matrix_properties(A)** controlla che A sia utilizzabile dai metodi:
  - deve essere quadrata;
  - deve essere simmetrica (A uguale alla sua trasposta);
  - deve essere definita positiva (autovalori > 0).

  Se qualcosa non va, lancia un errore. Serve solo a evitare input sbagliati.

- **_compute_residual_norm(A, x, b)** calcola il **residuo relativo** usato come criterio d'arresto:

$$\frac{\|Ax - b\|_2}{\|b\|_2}$$

  È l'indicatore che usiamo per dire che siamo arrivati abbastanza vicino, perciò possiamo fermarci.

Regole comuni ai 4 metodi:

- **Punto di partenza:** se non si fornisce un x0, l'algoritmo parte da x^(0)=0.

- **Criterio di arresto:**
  - il residuo relativo scende sotto tol;
  - al raggiungimento di max_iter;
- **Controllo input:** per uniformità con la consegna, prima di iterare si verifica che A sia quadrata, simmetrica e definita positiva (SPD).
- **Output:** ciascun metodo restituisce un dizionario con solution, iterations, time, residual_norm, converged, method.

I quattro metodi:

- **jacobi(A, b, tol, x0=None)** implementa l'iterazione

$$x^{(k+1)} = D^{-1}\left(b - (L+U)x^{(k)}\right)$$

dove **A=D+L+U** con **D** diagonale (non nulla!), **L** triangolare inferiore stretta, **U** triangolare superiore stretta.

Costo per iterazione. Una moltiplicazione matrice–vettore e poche operazioni vettoriali: con matrici sparse il costo è **O(nnz(A))** a iterazione.

Convergenza. Non è garantita dal solo SPD. Una condizione sufficiente è la dominanza diagonale stretta (per righe/colonne) oppure, in generale

$$\rho(D^{-1}(L+U)) < 1$$

Nella pratica è spesso più lento di Gauss–Seidel e del Gradiente Coniugato, a parità di tolleranza.

- **gauss_seidel(A, b, tol, x0=None)** riformula l'iterazione come

$$(D+L)\,x^{(k+1)} = b - U\,x^{(k)}$$

risolvendo a ogni passo un sistema triangolare inferiore tramite **forward substitution** esplicita (aggiornamento riga per riga).

Costo per iterazione. Un prodotto matrice–vettore per **Ux^(k)** più la sostituzione in avanti: con matrici sparse il costo è **O(nnz(A))** a iterazione.
Convergenza. Per matrici SPD la convergenza è garantita e, a parità di tolleranza, richiede in genere meno iterazioni di Jacobi. L'aggiornamento è intrinsecamente sequenziale (meno adatto al parallelismo rispetto a Jacobi).

- **gradient_method(A, b, tol, x0=None)** è il metodo del gradiente per sistemi con **A** come SPD. Dato **x^(0)** (di default il vettore nullo), all'iterazione **k** si calcola il residuo:

$$r^{(k)} = b - Ax^{(k)}$$

e si sceglie il passo ottimo lungo la direzione del residuo:

$$\alpha_k = \frac{(r^{(k)},\, r^{(k)})}{(r^{(k)},\, Ar^{(k)})}$$

aggiornando:

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

Costo per iterazione. Un prodotto matrice–vettore **Ar^(k)**, due prodotti scalari e poche operazioni assiali vettoriali: con matrici sparse il costo è **O(nnz(A))** a iterazione.
Convergenza. Lineare e fortemente dipendente dal numero di condizione **κ(A)**; può presentare andamento a "zig-zag" quando **κ** è alto, richiedendo molte iterazioni. In pratica è raramente competitivo con Gradiente Coniugato su SPD.

- **conjugate_gradient(A, b, tol, x0=None)** rappresenta l'algoritmo classico per matrici SPD. Con inizializzazione:

$$r^{(0)} = b - Ax^{(0)},\; p^{(0)} = r^{(0)}$$

ad ogni iterazione **k=0,1…** si calcola:

$$\alpha_k = \frac{(r^{(k)},\, r^{(k)})}{(p^{(k)},\, Ap^{(k)})}, \quad x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}, \quad r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)},$$

$$\beta_k = \frac{(r^{(k+1)},\, r^{(k+1)})}{(r^{(k)},\, r^{(k)})}, \quad p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)}.$$

Costo per iterazione. Un solo prodotto matrice–vettore **Ap^(k)**, alcuni prodotti scalari e aggiornamenti vettoriali: con matrici sparse il costo è **O(nnz(A))** a iterazione.
Convergenza. In aritmetica esatta raggiunge la soluzione in **al più n iterazioni** con **n=dimA**. In pratica la velocità dipende da $\sqrt{\kappa(A)}$ ed è nettamente superiore ai metodi classici. Su SPD è lo standard de-facto per efficienza e robustezza.

- **solve_all_methods(A, b, x_exact, tol)** che lancia in sequenza Jacobi, Gauss-Seidel, Gradiente e Gradiente Coniugato con gli stessi dati.
  In più, se c'è la soluzione esatta **x_exact** (nel nostro setup è il vettore tutto 1), calcola anche l'**errore relativo sulla soluzione** ‖x−x*‖/‖x*‖ e lo aggiunge all'output di ciascun metodo.

2.2 src/matrix_utils.py

Modulo di supporto per I/O delle matrici, generazione dei sistemi di test e formattazione dei risultati.

Non implementa solutori: prepara i dati per i metodi del file **linear_solvers.py**.

- **load_mtx_matrix(filepath)**

  Carica un file **Matrix Market** (.mtx) con scipy.io.mmread, che gestisce in automatico la conversione dagli indici 1-based del formato al 0-based di NumPy.

  Come output restituisce la matrice come array denso np.ndarray (conversione da sparse con .toarray() se necessario) e un dizionario con informazioni sulla matrice.

  Da notare: La scelta di convertire a denso semplifica il codice ma può richiedere memoria; qui è accettabile per le dimensioni in consegna.

- **check_matrix_properties(A, compute_eigenvalues=False)**

  Controlla proprietà strutturali utili ai solutori (quadratura, simmetria, definitezza positiva, calcola autovalori, salva min_eigenvalue e max_eigenvalue, il numero di condizione spettrale max/min ed eventuali problemi).

  Ritorna un dizionario con i risultati e un'eventuale lista di errori.

- **create_test_system(n, matrix_type='tridiagonal')**

  Crea una matrice A di dimensione n a scelta tra:
  - tridiagonal (SPD classica),
  - diagonal_dominant (matrice simmetrica resa diagonalmente dominante),
  - hilbert (simmetrica e definita positiva ma mal condizionata (utile per stress-test della convergenza).

Restituisce **A**, **b** e **x_exact**.

- **generate_standard_test_system(A)**

  Utility usata nella modalità di validazione: prende una **A** già caricata e genera lo stesso setup standard, ritornando **b** e **x_exact.**

- **print_matrix_info(A, name='Matrix')**

  Stampa a schermo: dimensioni, numero di non-zero, norma di Frobenius.

  Se **A** è quadrata, chiama **check_matrix_properties(..., compute_eigenvalues=True)** e mostra anche: simmetria, definitezza positiva, autovalori min/max e numero di condizione.

- **save_results_to_file(results, filepath, matrix_name, tolerance)**

  Scrive su un file di testo i risultati dei metodi per una certa tolleranza: convergenza sì/no, iterazioni, tempo, residuo e (se c'è) errore relativo sulla soluzione.

- **create_comparison_table(all_results)**

  Costruisce una tabella in testo: per ogni tolleranza elenca, metodo per metodo, convergenza, iterazioni, tempo ed errore relativo.

2.3 src/__init__.py

Questo file è l'inizializzatore del pacchetto: definisce l'API "pulita" esposta a chi usa la libreria (*es. from src import...)*.

Non contiene logica algoritmica: riesporta le classi e le funzioni principali (LinearIterativeSolver, utilità di I/O e di test), e fornisce metadati (versione e attore).

Il vettore pubblico è controllato da _**all**_, così gli import restano stabili e ordinati.

2.4 main.py

Rappresenta **l'entry point del progetto**: orchestria i test dei quattro solutori, gestisce gli input da riga di comando, produce l'output su schermo e salva i risultati su file.

Non implementa gli algoritmi numerici (che stanno in src/), ma coordina il workflow di validazione e confronto.

Uso: **python main.py**

Esistono tre modalità di esecuzione:

1. **Validazione (default)**
   Per ogni matrice .**mtx** trovata in **data/** (o per una matrice specificata con **-m**), costruisce il sistema con soluzione nota **x*=1** e **b=Ax***, avvia i quattro metodi con **x^(0)=0** e, per ciascuna tolleranza, riporta: numero di iterazioni, tempo, residuo relativo e (se disponibile) errore relativo sulla soluzione.

2. **Generica (utente fornisce A, b, x)**
   Con **-m A.mtx --b b.txt --x x.txt --single-tol 1e-6** legge una terna arbitraria **(A, b, x*)**. Esegue controlli di compatibilità (dimensioni) e una verifica diagnostica di coerenza **‖Ax*−b‖/‖b‖**. Poi lancia i quattro solutori con la tolleranza indicata.

3. **Matrice generata**
   Con **--test** (e opzionalmente **-n** e tipo in **create_test_system**) genera un sistema di prova (es. tridiagonale SPD), quindi segue lo stesso flusso della validazione.

Argomenti da riga di comando (principali):

- **-m/--matrix**: nome file .mtx da testare (cercato in **--data-dir**).
- **--test**: usa una matrice sintetica (default: tridiagonale).
- **-n/--size**: dimensione della matrice generata (default 100).
- **-t/--tolerances**: lista di tolleranze (default: 1e-4, 1e-6, 1e-8, 1e-10).
- **--b**, **--x**, **--single-tol**: attivano la **modalità generica**.
- **-o/--output**: cartella di destinazione dei risultati (results).
- **--data-dir**: cartella con i .mtx (data).

Funzioni di supporto:

- **test_single_matrix(path, tolerances, output_dir)**

  Carica **A** (con **load_mtx_matrix**), stampa info (**print_matrix_info**), costruisce **b=A*1** e lancia **LinearIterativeSolver.solve_all_methods** per ogni tolleranza. Stampa una tabella di confronto (**create_comparison_table**) e salva:
  - un summary complessivo per la matrice (*_summary.txt);

- un file di dettaglio per ciascuna tolleranza (*_tol_*.txt) con convergenza, iterazioni, tempo, residuo ed errore relativo.

- **test_generated_matrix(n, matrix_type, tolerances, output_dir)**

  Genera $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x^*}$ con **create_test_system** e segue lo stesso flusso di **test_single_matrix**.

- **test_generic_system(matrix_path, b_path, x_path, tolerance, output_dir)**

  Carica $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x^*}$ da file, verifica dimensioni e coerenza **b≈Ax*** , poi esegue i quattro metodi con la singola tolleranza indicata. Salva un file di risultati.

- **find_mtx_files(data_dir)**

  Elenca tutti i **.mtx** nella cartella indicata (usato dalla modalità di validazione "batch").

- **load_vector_from_file(filepath)**

  Carica **b** o **x*** da **.txt/.npy** con controlli d'errore basilari.

Logica del main():

Il main() interpreta gli argomenti, seleziona la modalità (generica, test, validazione singola, validazione batch), istanzia il solver con max_iter=20000 e verbose=False (set coerenti con la consegna), avvia i test e produce sia output su console sia file in results/.

In caso di errori (file mancanti, dimensioni incompatibili, ecc.) stampa messaggi esplicativi e termina in modo pulito.

Per ogni esperimento vengono sempre:

- stampate a schermo le metriche fondamentali (convergenza, iterazioni, tempo, residuo, errore relativo);
- salvati file testuali riutilizzabili per la relazione e per i grafici (timestamp nel summary, tabella di confronto formattata, un file per tolleranza).

2.5 plots/generate_plots.py

Questo script automatizza la creazione dei grafici a partire dai file di output prodotti da main.py (cartella results/). L'obiettivo è avere, con un solo comando, i grafici pronti.

Uso: **python tests/generate_plots.py**

Flusso generale:

1. **Caricamento dei risultati**: legge i file **results/{matrice}_tol_{toll}.txt** per le matrici **spa1, spa2, vem1, vem2** e per le tolleranze **1e-04, 1e-06, 1e-08, 1e-10**.
2. **Parsing**: da ciascun file estrae, per ogni metodo, converged, iterations, time, relative_error.
3. **Plot**: costruisce due figure riepilogative e le salva in **plots/**.

Funzioni principali:

- **parse_results_file_real(filepath)**

  Apre un singolo file di risultati (testo) e, tramite una regex, cerca il blocco di ciascun metodo:
  - convergenza ("Sì/No"),
  - numero di iterazioni,
  - tempo (secondi),
  - errore residuo,
  - errore relativo.

  Restituisce un dizionario.

  Assunzioni: il file ha il formato creato da **save_results_to_file**.

- **load_all_results_auto(results_dir="results")**

  Scorre in automatico i nomi **{matrice} × {tolleranza}** attesi, costruisce il percorso del file, chiama il parser e accumula tutto in una struttura nidificata. Stampa anche un breve riepilogo su quanti file sono stati trovati e letti con successo.

- **plot_all_matrices_comparison(results, output_dir="plots")**

  Crea una figura **2×2**, una sottotrama per ogni matrice. Sull'asse x c'è la **tolleranza** (scala log), sull'asse y il **numero di iterazioni** (scala log).
  Per ogni matrice traccia quattro curve (Jacobi, Gauss–Seidel, Gradiente, Gradiente Coniugato). L'asse x è **invertita** per leggere da sinistra a destra le tolleranze da più blanda (1e-4) a più severa (1e-10).

Salva il file come **plots/all_matrices_iterations.png** (curve iterazioni vs tolleranza).

- **plot_performance_all_tolerances(results)**

  Per **ogni tolleranza** genera una figura con **due grafici a barre** affiancati:
  - a sinistra, **iterazioni** per metodo e matrice (asse y in scala log);
  - a destra, **tempo (s)** per metodo e matrice (asse y in scala log);

  I file prodotti sono **plots/performance_tol_1e-04.png, ..._1e-06.png, ..._1e-08.png, ..._1e-10.png**.

- **main()**
  Esegue il flusso completo: carica i risultati, genera i due grafici e indica a video dove sono stati salvati.

Come leggere i grafici

- **Iterazioni vs tolleranza (2×2)**: a parità di matrice, confronta la **sensibilità alla tolleranza** e la **rapidità di convergenza** dei quattro metodi. Le curve più in basso sono migliori (meno iterazioni).
- **Riepilogo a barre (1e-06)**: confronta rapidamente **chi converge in meno iterazioni** e **chi impiega meno tempo reale** su ciascuna matrice.

2.6 tests/test_quick.py

Scopo: verifica veloce che la mini-libreria funzioni end-to-end su un caso piccolo e che i controlli sulle proprietà della matrice (quadrata, simmetrica, SPD) si comportino correttamente.

Uso: **python tests/quick_test.py**

- **test_basic_functionality()**
  Genera un sistema di prova A,b,x∗ con **create_test_system(n=10,'tridiagonal')**, dove **x*=1** e **b=Ax***.

  Istanziato il solver (**LinearIterativeSolver(max_iter=1000, verbose=True)**), esegue un test puntuale con Jacobi e poi il batch con tutti i metodi tramite **solve_all_methods(...)**.

  Stampa per ciascun metodo: convergenza, iterazioni, tempo, errore relativo.

- **test_matrix_properties()**
  Verifica positiva su una 3×3 tridiagonale simmetrica e SPD.
  Verifica negativa su una matrice non simmetrica: ci si attende un'eccezione dal controllo interno **_check_matrix_properties**.

Output atteso: tabella riassuntiva dei metodi sul caso 10×10 e messaggio conclusivo "TUTTI I TEST SUPERATI!" se non emergono errori.

2.7 tests/mtx_indexing_test.py

Scopo: validare che **scipy.io.mmread** gestisca correttamente la conversione dagli indici **1-based** (tipici del formato Matrix Market) agli indici **0-based** (NumPy/SciPy), e fare un controllo minimo di simmetria/consistenza su file reali.

Uso: **python tests/mtx_indexing_test.py**

- **test_mtx_indexing()**
  Crea un file .mtx temporaneo (piccolo, simmetrico) scritto **in 1-based**.
  Carica con mmread, converte a denso e confronta con la matrice attesa in 0-based usando np.allclose.
  Verifica aggiuntiva: simmetria della matrice caricata.

- **test_with_actual_matrix(matrix_path)**
  Carica una matrice reale (es. **data/spa1.mtx**), stampa tipo, dimensioni e nnz.
  Se la matrice è piccola, la converte a denso e verifica rapidamente la simmetria; se è grande, ispeziona alcuni elementi.

- **check_mtx_file_format(filepath)**
  Stampa le prime righe del file .mtx e legge la riga delle dimensioni/nnz per un controllo visivo del formato.

- **__main__**
  Esegue il test sintetico su file temporaneo e, a seguire, i controlli su alcune matrici reali (percorsi configurabili nell'elenco test_matrices).

Output atteso: messaggi che confermano la corretta conversione da 1-based a 0-based nel caso sintetico e l'esito dei controlli su file reali (shape, nnz, simmetria).

2.8 results/

Contiene gli output generati da main.py, che possono essere di tre tipi:

- **File "summary" per matrice**
  Per esempio: **spa1_summary.txt** contiene le info della matrice (dimensioni, sparsità, simmetria, ecc.) più una tabella di confronto tra metodi e tolleranze.

- **File per tolleranza (modalità validazione)**
  Per esempio: **spa1_tol_1e-06.txt**, **spa1_tol_1e-08.txt**.. per ogni metodo riporta Metodo, Convergenza, Iterazioni, Tempo, Errore residuo, Errore relativo.

- **File per tolleranza (modalità generica)**
  Per esempio: **spa1_generic_tol_1e-06.txt** identico ai file sopra ma ottenuto quando passi **--b** e **--x**.

Attenzione: i file in **results/** vengono creati/aggiornati automaticamente da **main.py**.

2.8 test_data/

Contiene gli input usati **solo** nella modalità generica:

- **x_*.txt:** vettori soluzione esatta (nel setup di questo progetto sono **tutti 1**).
- **b_*.txt:** termini noti coerenti, calcolati come **b = A @ x**.
- **test_x.txt, test_b.txt:** coppia di prova piccola.

Quando si lancia, ad esempio:
python main.py -m spa1.mtx --b test_data/b_spa1.txt --x test_data/x_spa1.txt --single-tol 1e-6, **main.py** carica **A** da **data/spa1.mtx** e **b** con **x** da **test_data/**..., verifica le dimensioni e poi esegue i 4 metodi su quella singola tolleranza.

Attenzione: i file in **test_data/** devono essere **preparati da noi** (o con uno script) e devono avere la **stessa dimensione** di A.

Esempio di alcuni script utilizzati:

spa1

python -c 'import numpy as np; from src import load_mtx_matrix; A,_=load_mtx_matrix("data/spa1.mtx"); x=np.ones(A.shape[0]); np.savetxt("test_data/x_spa1.txt", x); np.savetxt("test_data/b_spa1.txt", A@x)'

python main.py -m spa1.mtx --b test_data/b_spa1.txt --x test_data/x_spa1.txt --single-tol 1e-6

spa2

```
python -c 'import numpy as np; from src import load_mtx_matrix;
A,_=load_mtx_matrix("data/spa2.mtx"); x=np.ones(A.shape[0]);
np.savetxt("test_data/x_spa2.txt", x); np.savetxt("test_data/b_spa2.txt", A@x)'

python main.py -m spa2.mtx --b test_data/b_spa2.txt --x test_data/x_spa2.txt --
single-tol 1e-6
```

vem1

```
python -c 'import numpy as np; from src import load_mtx_matrix;
A,_=load_mtx_matrix("data/vem1.mtx"); x=np.ones(A.shape[0]);
np.savetxt("test_data/x_vem1.txt", x); np.savetxt("test_data/b_vem1.txt", A@x)'

python main.py -m vem1.mtx --b test_data/b_vem1.txt --x test_data/x_vem1.txt --
single-tol 1e-6
```

vem2

```
python -c 'import numpy as np; from src import load_mtx_matrix;
A,_=load_mtx_matrix("data/vem2.mtx"); x=np.ones(A.shape[0]);
np.savetxt("test_data/x_vem2.txt", x); np.savetxt("test_data/b_vem2.txt", A@x)'

python main.py -m vem2.mtx --b test_data/b_vem2.txt --x test_data/x_vem2.txt --
single-tol 1e-6
```

3. Risultati

In questo capitolo confrontiamo i quattro metodi implementati (Jacobi, Gauss–Seidel, Gradiente e Gradiente Coniugato) nella soluzione di sistemi lineari **Ax=b** con **A** simmetrica definita positiva (SPD).

L'obiettivo è valutare **efficienza** (iterazioni e tempo) e **accuratezza** (errore) al variare della tolleranza e della matrice.

3.1 Lettura dei risultati dal terminale

Di seguito vengono commentate le tabelle di confronto generate automaticamente dai file **results/*_summary.txt**. Per ogni matrice (**spa1**, **spa2**, **vem1**, **vem2**) sono riportati, al variare di **tol ∈ {10^(−4), 10^(−6), 10^(−8), 10^(−10)}**, **convergenza, iterazioni, tempo wall-clock e errore relativo.**

3.1.1 Matrice spa1.mtx

Pannello con le info della matrice:

```
Caricata matrice spa1.mtx
  Dimensione: 1000x1000
  Elementi non zero: 182264
  Sparsità: 81.77%
  Simmetrica: True

Informazioni Matrice spa1.mtx:
  Dimensione: 1000x1000
  Elementi non zero: 182264
  Norma di Frobenius: 1.7874e+04
  Simmetrica: True
  Definita positiva: True
  Autovalore minimo: 4.8798e-01
  Autovalore massimo: 9.9946e+02
  Numero di condizione: 2.0482e+03

Sistema generato:
  Soluzione esatta: x = [1, 1, ..., 1] (dimensione 1000)
  Termine noto: b = A * x
```

```
TABELLA DI CONFRONTO METODI ITERATIVI
================================================================================
Tolleranza   Metodo             Conv.  Iter.    Tempo (s)    Err. Rel.
--------------------------------------------------------------------------------
1e-04        Jacobi             Sì     115      0.034002     1.771281e-03
             Gauss-Seidel       Sì     9        0.696276     1.820594e-02
             Gradient           Sì     143      0.039021     3.457470e-02
             Conjugate Gradient Sì     49       0.010000     2.078976e-02
--------------------------------------------------------------------------------
1e-06        Jacobi             Sì     181      0.054004     1.797930e-05
             Gauss-Seidel       Sì     17       1.332070     1.299694e-04
             Gradient           Sì     3577     0.957618     9.680457e-04
             Conjugate Gradient Sì     134      0.026093     2.552909e-05
--------------------------------------------------------------------------------
1e-08        Jacobi             Sì     247      0.072157     1.824979e-07
             Gauss-Seidel       Sì     24       1.827774     1.709733e-06
             Gradient           Sì     8233     2.212606     9.816364e-06
             Conjugate Gradient Sì     177      0.033562     1.319840e-07
--------------------------------------------------------------------------------
1e-10        Jacobi             Sì     313      0.093518     1.852436e-09
             Gauss-Seidel       Sì     31       2.403620     2.248088e-08
             Gradient           Sì     12919    3.382639     9.820387e-08
             Conjugate Gradient Sì     200      0.037998     1.203111e-09
--------------------------------------------------------------------------------
```

Ordine delle iterazioni:

- **Gradiente Coniugato**:
  decine/centinaia di iterazioni (es. **49** iter a **1e-4**; **200** a **1e-10**).

- **Gauss–Seidel**:
  decine di iterazioni (es. **9** a **1e-4**; **31** a **1e-10**).

- **Jacobi**:
  centinaia di iterazioni (es. **115** a **1e-4**; **313** a **1e-10**).

- **Gradiente**:
  da centinaia a migliaia/decine di migliaia (es. **143** a **1e-4**, **12919** a **1e-10**).

Tempi:

- **Gradiente Coniugato**:
  è il più veloce (es. **0.01s** a **1e-4**; **0.379s** a **1e-10**).

- **Gauss–Seidel**:
  è lento nonostante le poche iterazioni (es. **0.696s** a **1e-4**; **2.403s** a **1e-10**), perché ogni iterazione fa sostituzione in avanti **sequenziale**.

- **Jacobi**:
  spesso abbastanza rapido (es. **0.034** a **1e-4**; **0.093** a **1e-10**) , ma il tempo cresce quando servono molte iterazioni.

- **Gradiente**:
  accumula tempo per il **numero enorme di iterazioni** (es. **0.039** a **1e-4**; **3.382** a **1e-10**).

Errori relativi:

- Scendono con la tolleranza!
- Per esempio, a 1e-10:
  - **Gradiente Coniugato ≈ 1.2e-09**
  - **Jacobi ≈ 1.85e-09**
  - **Gauss–Seidel ≈ 2.25e-08**
  - **Gradiente ≈ 9.8e-08.**
- Sono tutti numeri molto piccoli: siamo vicino ai limiti della precisione double.

3.1.2 Matrice spa2.mtx

Pannello con le info della matrice:

```
Caricata matrice spa2.mtx
  Dimensione: 3000x3000
  Elementi non zero: 1631738
  Sparsità: 81.87%
  Simmetrica: True

Informazioni Matrice spa2.mtx:
  Dimensione: 3000x3000
  Elementi non zero: 1631738
  Norma di Frobenius: 9.5801e+04
  Simmetrica: True
  Definita positiva: True
  Autovalore minimo: 2.1235e+00
  Autovalore massimo: 2.9984e+03
  Numero di condizione: 1.4120e+03

Sistema generato:
  Soluzione esatta: x = [1, 1, ..., 1] (dimensione 3000)
  Termine noto: b = A * x
```

```
TABELLA DI CONFRONTO METODI ITERATIVI
================================================================================
Tolleranza   Metodo             Conv.  Iter.    Tempo (s)    Err. Rel.
--------------------------------------------------------------------------------
1e-04        Jacobi             Sì     36       0.171706     1.766247e-03
             Gauss-Seidel       Sì     5        3.512159     2.598896e-03
             Gradient           Sì     161      0.501269     1.812965e-02
             Conjugate Gradient Sì     42       0.087575     9.821128e-03
--------------------------------------------------------------------------------
1e-06        Jacobi             Sì     57       0.267568     1.666756e-05
             Gauss-Seidel       Sì     8        5.543945     5.141641e-05
             Gradient           Sì     1949     6.158386     6.694229e-04
             Conjugate Gradient Sì     122      0.254864     1.197985e-04
--------------------------------------------------------------------------------
1e-08        Jacobi             Sì     78       0.367388     1.572870e-07
             Gauss-Seidel       Sì     12       8.309299     2.794322e-07
             Gradient           Sì     5087     15.858072    6.865240e-06
             Conjugate Gradient Sì     196      0.410797     5.586661e-07
--------------------------------------------------------------------------------
1e-10        Jacobi             Sì     99       0.466009     1.484273e-09
             Gauss-Seidel       Sì     15       10.355855    5.570739e-09
             Gradient           Sì     8285     26.227214    6.937813e-08
             Conjugate Gradient Sì     240      0.501360     5.324230e-09
--------------------------------------------------------------------------------
```

Ordine delle iterazioni:

- **Gradiente Coniugato**:
  decine/centinaia di iterazioni (es. **42** iter a **1e-4**; **240** a **1e-10**).

- **Gauss–Seidel**:
  decine di iterazioni (es. **5** a **1e-4**; **15** a **1e-10**).

- **Jacobi**:
  decine/centinaia di iterazioni (es. **36** a **1e-4**; **99** a **1e-10**).

- **Gradiente**:
  da centinaia a migliaia/decine di migliaia (es. **161** a **1e-4**, **8285** a **1e-10**).

Tempi:

- **Gradiente Coniugato**:
  è il più veloce (es. **0.088s** a **1e-4**; **0.501s** a **1e-10**).

- **Gauss–Seidel**:
  è lento nonostante le poche iterazioni (es. **3.512s** a **1e-4**; **10.356s** a **1e-10**), perché ogni iterazione fa sostituzione in avanti **sequenziale**.

- **Jacobi**:
  rapido e scalabile (es. **0.172** a **1e-4**; **0.466** a **1e-10**).

- **Gradiente**:
  accumula tempo per il **numero enorme di iterazioni** (es. **0.501** a **1e-4**; **26.227** a **1e-10**).

Errori relativi:

- Scendono con la tolleranza!
- Per esempio, a 1e-10:
  - **Gradiente Coniugato ≈ 5.32e-09**
  - **Jacobi ≈ 1.48e-09**
  - **Gauss–Seidel ≈ 5.57e-09**
  - **Gradiente ≈ 6.94e-08**
- Sono tutti numeri molto piccoli: correttezza fino a 8–9 cifre decimali.

3.1.3 Matrice vem1.mtx

Pannello con le info della matrice:

```
Caricata matrice vem1.mtx
  Dimensione: 1681x1681
  Elementi non zero: 13385
  Sparsità: 99.53%
  Simmetrica: True

Informazioni Matrice vem1.mtx:
  Dimensione: 1681x1681
  Elementi non zero: 13385
  Norma di Frobenius: 1.2527e+02
  Simmetrica: True
  Definita positiva: True
  Autovalore minimo: 1.2321e-02
  Autovalore massimo: 4.0000e+00
  Numero di condizione: 3.2464e+02

Sistema generato:
  Soluzione esatta: x = [1, 1, ..., 1] (dimensione 1681)
  Termine noto: b = A * x
```

```
TABELLA DI CONFRONTO METODI ITERATIVI
================================================================================
Tolleranza   Metodo             Conv.  Iter.    Tempo (s)    Err. Rel.
--------------------------------------------------------------------------------
1e-04        Jacobi             Sì     1314     1.309663     3.540381e-03
             Gauss-Seidel       Sì     659      145.454549   3.506973e-03
             Gradient           Sì     890      0.380865     2.704572e-03
             Conjugate Gradient Sì     38       0.013510     4.082793e-05
--------------------------------------------------------------------------------
1e-06        Jacobi             Sì     2433     2.583244     3.540073e-05
             Gauss-Seidel       Sì     1218     275.914331   3.526697e-05
             Gradient           Sì     1612     0.743079     2.713339e-05
             Conjugate Gradient Sì     45       0.014966     3.732340e-07
--------------------------------------------------------------------------------
1e-08        Jacobi             Sì     3552     3.929881     3.539766e-07
             Gauss-Seidel       Sì     1778     394.705513   3.517457e-07
             Gradient           Sì     2336     0.851060     2.695337e-07
             Conjugate Gradient Sì     53       0.014001     2.831873e-09
--------------------------------------------------------------------------------
1e-10        Jacobi             Sì     4671     4.693605     3.539467e-09
             Gauss-Seidel       Sì     2338     515.579566   3.508245e-09
             Gradient           Sì     3058     1.330840     2.713171e-09
             Conjugate Gradient Sì     59       0.018995     2.191750e-11
--------------------------------------------------------------------------------
```

Ordine delle iterazioni:

- **Gradiente Coniugato**:
  decine di iterazioni (es. **38** iter a **1e-4**; **59** a **1e-10**).

- **Gauss–Seidel**:
  centinaia/migliaia di iterazioni (es. **659** a **1e-4**; **2338** a **1e-10**).

- **Jacobi**:
  migliaia di iterazioni (es. **1314** a **1e-4**; **4671** a **1e-10**).

- **Gradiente**:
  da centinaia a migliaia di iterazioni (es. **890** a **1e-4**, **3058** a **1e-10**).

Tempi:

- **Gradiente Coniugato**:
  è il più veloce (es. **0.0135s** a **1e-4**; **0.0190s** a **1e-10**).

- **Gauss–Seidel**:
  è molto lento (es. **145s** a **1e-4**; **515s** a **1e-10**).

- **Jacobi**:
  abbastanza rapido (es. **1.31** a **1e-4**; **4.69** a **1e-10**), ma il tempo cresce quando servono molte iterazioni.

- **Gradiente**:
  piuttosto contenuto (es. **0.38** a **1e-4**; **1.33** a **1e-10**).

Errori relativi:

- Scendono con la tolleranza!
- Per esempio, a 1e-10:
  - **Gradiente Coniugato ≈ 2.19e-11** (11 cifre corrette)
  - **Jacobi ≈ 3.54e-09**
  - **Gauss–Seidel ≈ 3.51e-09**
  - **Gradiente ≈ 2.71e-09**
- Sono tutti numeri molto piccoli: nell'ordine di **10^(-9) e 10^(-11)**, vicino ai limiti della precisione double.

3.1.4 Matrice vem2.mtx

Pannello con le info della matrice:

```
Caricata matrice vem2.mtx
  Dimensione: 2601x2601
  Elementi non zero: 21225
  Sparsità: 99.69%
  Simmetrica: True

Informazioni Matrice vem2.mtx:
  Dimensione: 2601x2601
  Elementi non zero: 21225
  Norma di Frobenius: 1.5728e+02
  Simmetrica: True
  Definita positiva: True
  Autovalore minimo: 7.8892e-03
  Autovalore massimo: 4.0000e+00
  Numero di condizione: 5.0702e+02

Sistema generato:
  Soluzione esatta: x = [1, 1, ..., 1] (dimensione 2601)
  Termine noto: b = A * x
```

```
TABELLA DI CONFRONTO METODI ITERATIVI
================================================================================
Tolleranza   Metodo             Conv.  Iter.    Tempo (s)    Err. Rel.
--------------------------------------------------------------------------------
1e-04        Jacobi             Sì     1927     6.629710     4.968461e-03
             Gauss-Seidel       Sì     965      510.232157   4.951189e-03
             Gradient           Sì     1308     2.676783     3.811930e-03
             Conjugate Gradient Sì     47       0.060571     5.729017e-05
--------------------------------------------------------------------------------
1e-06        Jacobi             Sì     3676     12.999283    4.967034e-05
             Gauss-Seidel       Sì     1840     966.848848   4.941761e-05
             Gradient           Sì     2438     4.665881     3.791419e-05
             Conjugate Gradient Sì     56       0.071995     4.742996e-07
--------------------------------------------------------------------------------
1e-08        Jacobi             Sì     5425     19.190853    4.965608e-07
             Gauss-Seidel       Sì     2714     1403.482746  4.958370e-07
             Gradient           Sì     3566     6.778007     3.809851e-07
             Conjugate Gradient Sì     66       0.083406     4.299983e-09
--------------------------------------------------------------------------------
1e-10        Jacobi             Sì     7174     24.925650    4.964181e-09
             Gauss-Seidel       Sì     3589     1867.759789  4.948903e-09
             Gradient           Sì     4696     8.832831     3.798757e-09
             Conjugate Gradient Sì     74       0.096507     2.247624e-11
--------------------------------------------------------------------------------
```

Ordine delle iterazioni:

- **Gradiente Coniugato**:
  decine di iterazioni (es. **47** iter a **1e-4**; **74** a **1e-10**).

- **Gauss–Seidel**:
  centinaia/migliaia di iterazioni (es. **965** a **1e-4**; **3589** a **1e-10**).

- **Jacobi**:
  migliaia di iterazioni (es. **1927** a **1e-4**; **7174** a **1e-10**).

- **Gradiente**:
  migliaia di iterazioni (es. **1308** a **1e-4**, **4696** a **1e-10**).

Tempi:

- **Gradiente Coniugato**:
  è il più veloce (es. **0.06s** a **1e-4**; **0.096s** a **1e-10**).

- **Gauss–Seidel**:
  è molto lento (es. **510s** a **1e-4**; **1868s** a **1e-10**), perché ogni iterazione fa sostituzione in avanti **sequenziale**.

- **Jacobi**:
  diventa pesante (es. **6.63** a **1e-4**; **24.9** a **1e-10**) , ma il tempo cresce quando servono molte iterazioni.

- **Gradiente**:
  piuttosto contenuto (es. **2.67** a **1e-4**; **8.83** a **1e-10**).

Errori relativi:

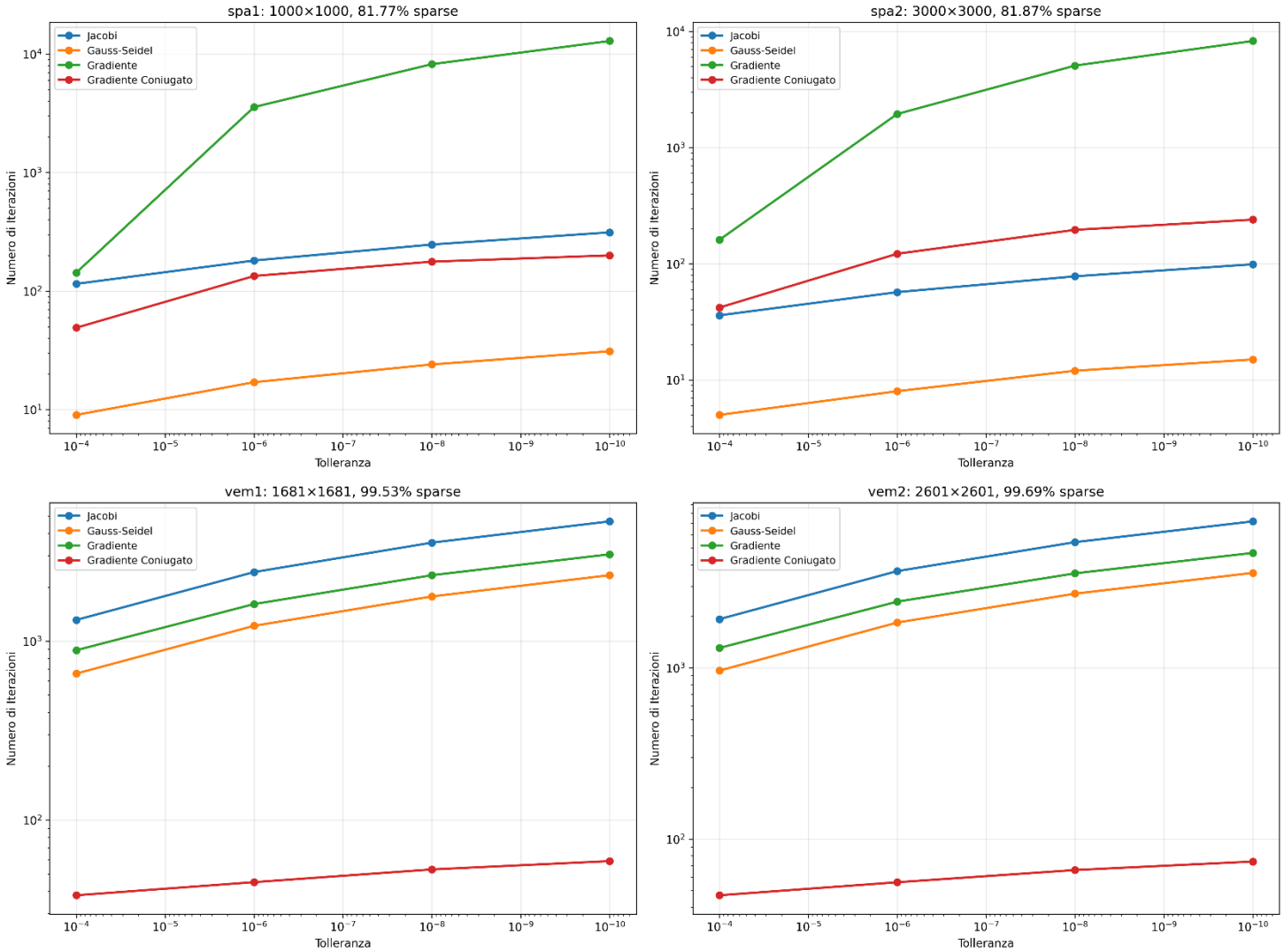
- Scendono con la tolleranza!
- Per esempio, a 1e-10:
  - **Gradiente Coniugato ≈ 2.25e-11** (11 cifre corrette)
  - **Jacobi ≈ 4.96e-09**
  - **Gauss–Seidel ≈ 4.95e-09**
  - **Gradiente ≈ 3.80e-09**
- Sono tutti numeri molto piccoli: nell'ordine di **10^(-9) e 10^(-11)**, vicino ai limiti della precisione double.

3.2 Iterazioni vs Tolleranza

Ogni pannello è una matrice (titolo = dimensioni e percentuale di zeri).

Sull'asse x c'è la **tolleranza** (in scala log, da 10^{-4} a 10^{-10}, con asse invertito: andando a destra la tolleranza si stringe).

Sull'asse y c'è il **numero di iterazioni** (log). Le quattro curve sono i 4 metodi citati precedentemente.



Osservazioni principali:

- **Il Gradiente Coniugato (CG) è nettamente il più efficace.**
  In tutti i casi la curva/colonna rossa è la più bassa: **pochi passi** per raggiungere la tolleranza, e tempi molto contenuti.
  L'aumento di iterazioni al ridursi della tolleranza è **moderato** rispetto agli altri metodi.

- **Il Gradiente (steepest descent) è il più sensibile alla tolleranza e alla condizione.**
  Le curve verdi crescono rapidamente stringendo la tolleranza (soprattutto su *spa1* e *spa2*): servono **migliaia** di iterazioni a 10^{-8} e 10 ^{-10}.
  È il comportamento tipico del metodo del gradiente quando il numero di condizione è alto.

- **Gauss–Seidel richiede meno iterazioni di Jacobi, ma il tempo può essere maggiore.**
  Nei pannelli e nel riepilogo a 10^{-6} le barre arancioni (GS) in **iterazioni** sono spesso più basse delle blu (Jacobi).
  Tuttavia, nella colonna **tempi** GS può risultare più lento perché l'aggiornamento è **intrinsecamente sequenziale** (forward substitution riga-per-riga) e, in questa implementazione, è meno **vettorializzato** di Jacobi.
  Quindi **meno iterazioni ≠ tempo minore**.

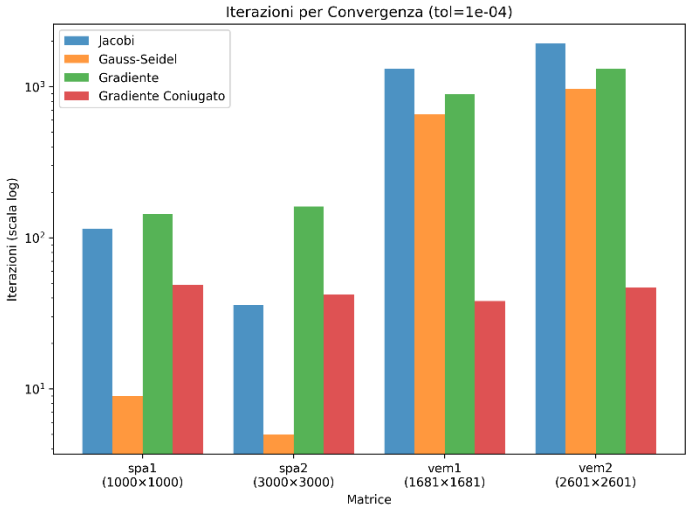
- **Effetto della dimensione/sparsità della matrice.**
  Passando da *spa1* (1000×1000) a *spa2* (3000×3000) e da *vem1* a *vem2* crescono sia iterazioni (moderatamente) sia tempi (marcatamente).
  Le matrici *vem* sono **molto più sparse** (≈99.5–99.7%) e questo aiuta i tempi dei metodi vettorializzati (Jacobi/CG) più di GS.

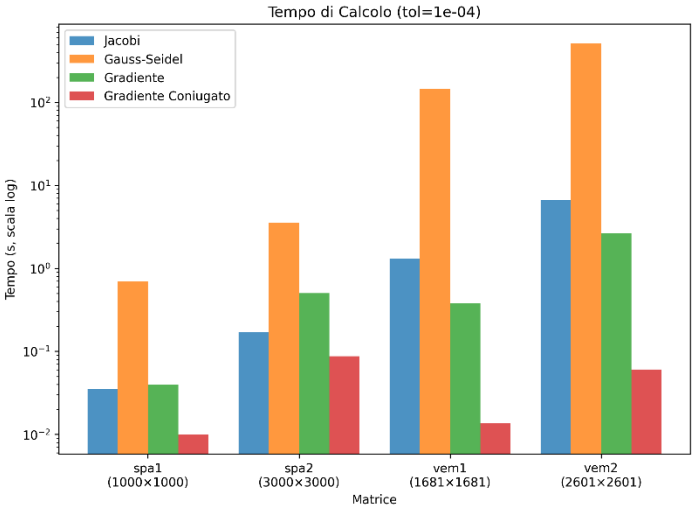
- **Tolleranza più stretta implica più iterazioni per tutti.**
  Le curve sono **crescienti** verso destra: l'ordine è quello atteso.
  L'incremento è **più dolce** per CG, **marcato** per Gradiente, **intermedio** per Jacobi/GS.

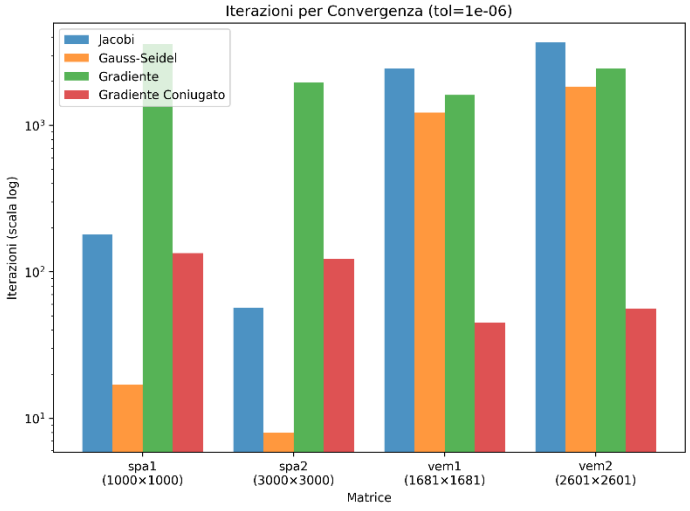
3.3 Riepilogo prestazioni

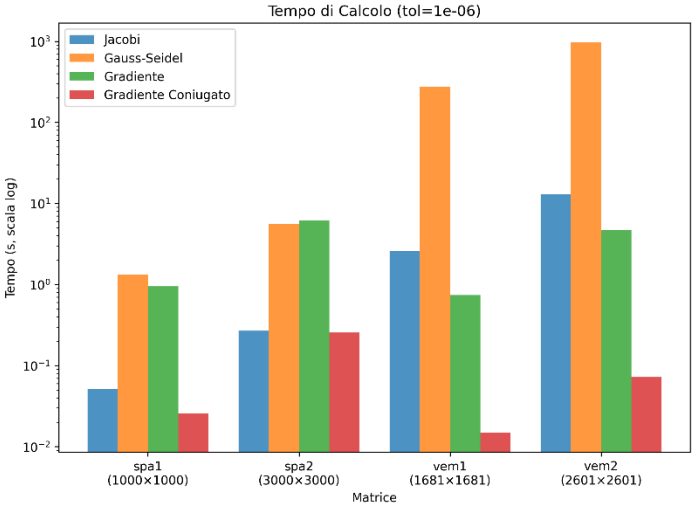
Sono le barre logaritmiche di **iterazioni** e **tempi** per un confronto immediato tra metodi e matrici, dove:

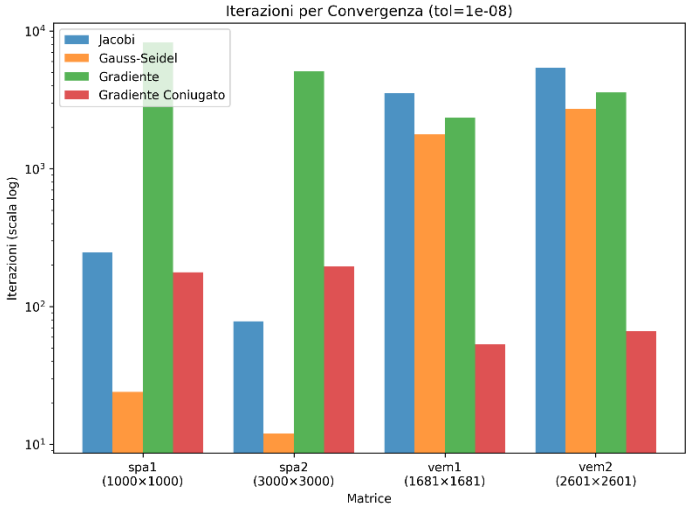
- A sinistra: **iterazioni** richieste dai metodi su ciascuna matrice (scala log).
- A destra: **tempi** wall–clock corrispondenti (scala log).

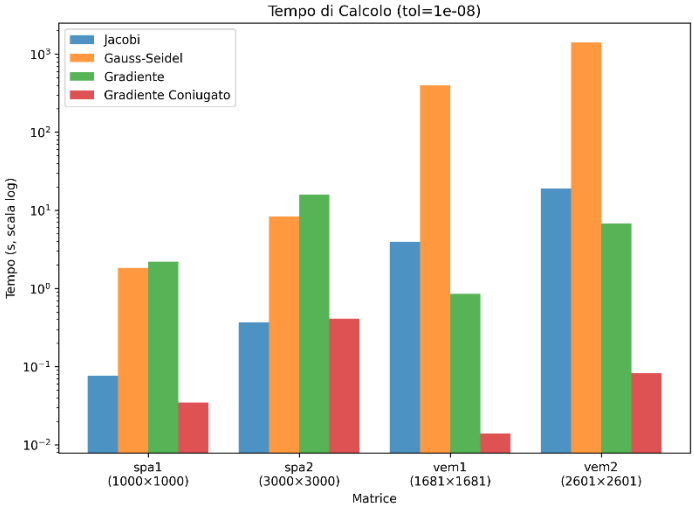


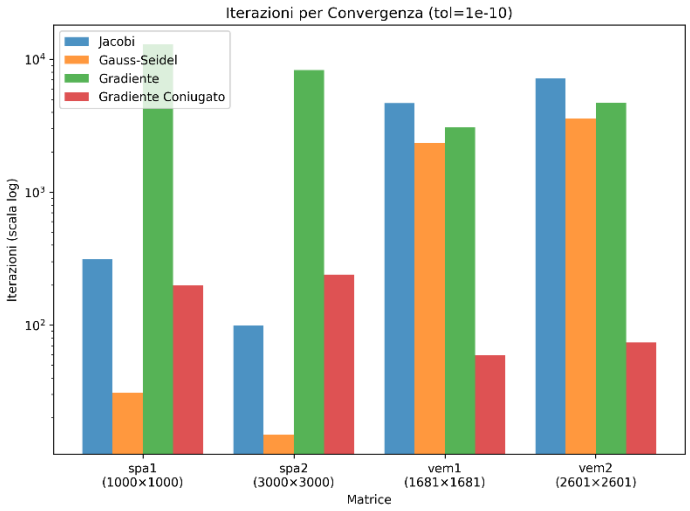


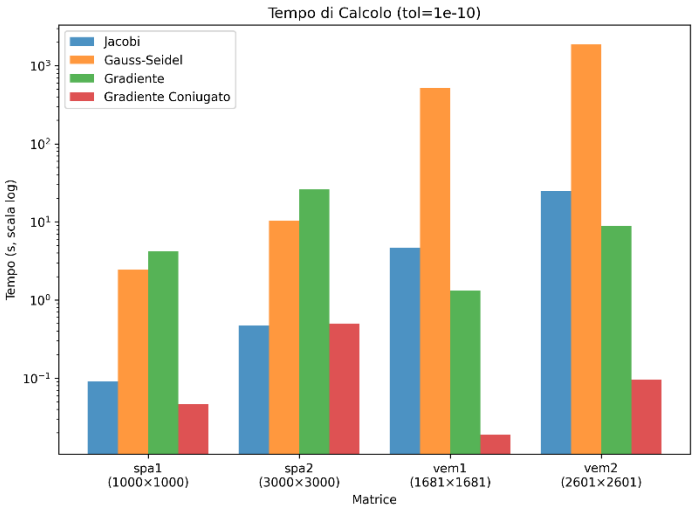












Osservazioni principali:

1) Iterazioni:

- **Gradiente coniugato** resta sempre nell'ordine delle **decine di iterazioni** su **tutte le matrici** e per **tutte le tolleranze**.
  L'incremento passando da 1e-4 a 1e-10 è **moderato**.
  Messaggio: è il **più robusto** all'inasprimento della tolleranza.

- **Gauss–Seidel** richiede **meno iterazioni di Jacobi** (spesso 5–20× meno), ma le **iterazioni crescono parecchio** quando **si scende a 1e-8 e 1e-10**.

- **Jacobi** ha il numero di iterazioni **sempre superiore a Gauss–Seidel** e **molto più alto di Gradiente coniugato**.
  Cresce in modo marcato quando **la tolleranza si fa stringente**.

- **Gradiente "semplice"** è quello che **cresce di più**; su diverse matrici arriva a migliaia–decine di migliaia di passi per 1e-8 e 1e-10.
  Causa è l'andamento "zig-zag" e la dipendenza forte dal numero di condizione **κ(A)**.

2) Tempi:

- **Gradiente coniugato** è quasi sempre il **più veloce** in tempo assoluto, spesso con **due–tre ordini di grandezza in meno rispetto a Gauss–Seidel e al Gradiente**.
  Quasi sempre **più rapido anche di Jacobi** nonostante **faccia più operazioni per iterazione:** guadagna perché **chiude in poche iterazioni.**

- **Gauss–Seidel** pur con **poche iterazioni**, il **tempo esplode** su **matrici grandi/tolleranze strette**.
  Causa è la **forward substitution** che è intrinsecamente **sequenziale** e **meno cache-friendly** rispetto a un singolo mat-vec.
  Inoltre **ogni iterazione "pesa" di più**.

- **Jacobi** ha i **tempi spesso contenuti** (una sola mat-vec per iterazione e aggiornamenti indipendenti), ma **peggiora quando il numero di iterazioni cresce** (1e-8 e1e-10).

- **Gradiente** ha i tempi **intermedi/ampi** perché **accumula tantissime iterazioni**; anche quando ogni passo è economico, la somma diventa rilevante.

3) Differenze tra matrici:

- spa1 (1000×1000) e spa2 (3000×3000, ~81% di zeri):
  - all'aumentare della dimensione, i tempi di tutti i metodi crescono;
  - **Gauss–Seidel** e **Gradiente soffrono di più**, mentre **Gradiente coniugato** resta **competitivo**;

- vem1 (1681×1681) e vem2 (2601×2601, ~99.6% di zeri):
  - l'elevatissima sparsità mantiene i mat-vec leggeri, ma il numero di iterazioni richieste da Jacobi/Gauss–Seidel/Gradiente aumenta quando la tolleranza si stringe;
  - **Gradiente coniugato** rimane nell'ordine delle decine di passi e **domina** anche qui;

4) Effetto della tolleranza:

Passando da **1e-4 → 1e-6 → 1e-8 → 1e-10**:

- **Gradiente coniugato**: aumento **lieve** di iterazioni e tempi.

- **Jacobi/Gauss–Seidel**: **aumento marcato** delle iterazioni; i tempi diventano rapidamente alti, in particolare per **Gauss–Seidel**.

- **Gradiente**: crescita **forte** (fino a 10^3–10^4 iterazioni), con tempi che possono diventare impraticabili.

5) Da mettere in evidenza:

- **Gradiente coniugato** è **lo standard de-facto su SPD**: poche iterazioni, tempi minimi, scalabilità buona al crescere di n e alla richiesta di precisione.

- **Gauss–Seidel**: **pochi passi**, ma **costo per passo alto** e poco parallelismo → può risultare **molto lento** in wall-time su matrici grandi.

- **Jacobi**: semplice e **ben parallelizzabile**; tempo discreto a **tolleranze blande**, ma il conto di iterazioni lo penalizza quando si scende a 1e-8 e 1e-10.

- **Gradiente**: implementazione corta, ma **sensibile a κ(A)**; diventa **inefficiente** già a 1e-6 e 1e-8.

4. Conclusioni

L'analisi comparativa dei quattro metodi iterativi (Jacobi, Gauss–Seidel, Gradiente e Gradiente Coniugato) ha permesso di valutare in modo sistematico l'efficienza e l'accuratezza nella risoluzione di sistemi lineari con matrici simmetriche definite positive (SPD).

Dai risultati emersi possiamo trarre alcune considerazioni generali:

- **Gradiente Coniugato come metodo di riferimento**
  - È risultato sistematicamente il **più efficiente**: converge in poche decine/centinaia di iterazioni anche su matrici di grandi dimensioni e tolleranze strette.
  - I tempi di calcolo sono di gran lunga i **più bassi** (anche ordini di grandezza inferiori a Jacobi e soprattutto Gauss–Seidel).
  - Gli errori relativi raggiungono valori prossimi al limite della precisione double (circa **10^(-11)**), garantendo soluzioni numericamente molto accurate.

  Si conferma quindi lo **standard de facto** per matrici SPD.
- **Jacobi e Gauss–Seidel**
  - Entrambi convergono con affidabilità, ma presentano limitazioni:
    - **Jacobi** richiede **molte più iterazioni** di Gauss–Seidel, ma **è più parallelizzabile e veloce per singolo passo.**
    - **Gauss–Seidel**, pur richiedendo **meno iterazioni**, risulta **spesso molto lento** per via dell'implementazione **sequenziale** e del costo elevato per iterazione.
  - In pratica, il vantaggio teorico di Gauss–Seidel in termini di iterazioni non si traduce in un guadagno reale sui tempi.
- **Gradiente (steepest descent)**
  - Mostra una **forte dipendenza dal numero di condizione**: a tolleranze strette il numero di iterazioni cresce rapidamente fino a valori impraticabili (**10^3** e **10^4**).
  - I **tempi risultano medi o elevati**, a causa dell'accumulo delle iterazioni.
  - Resta un metodo concettualmente semplice, ma poco competitivo rispetto al Coniugato.

- **Effetti di dimensione e sparsità delle matrici**
  - **L'aumento della dimensione** comporta un **incremento dei tempi** per tutti i metodi, ma in misura molto diversa:
    - **Jacobi e Conjugate Gradient beneficiano della sparsità elevata** (vem1/vem2), **restando efficienti**.
    - **Gauss–Seidel scala male**, con **tempi che esplodono al crescere della dimensione**.
    - **Gradiente risente fortemente della tolleranza e del numero di condizione**, diventando **poco praticabile**.
- **Accuratezza**
  - Tutti i metodi raggiungono **errori molto bassi** (da 10^(-7) a 10^(-11)), sufficienti per la maggior parte delle applicazioni numeriche.
  - Il Coniugato si distingue perché mantiene errori sistematicamente più bassi, anche con poche iterazioni.

Conclusione generale

Il confronto ha mostrato che, pur essendo utili a fini didattici, Jacobi, Gauss–Seidel e Gradiente non sono competitivi per problemi di grandi dimensioni e tolleranze strette. Il **Gradiente Coniugato** emerge come metodo ottimale per matrici SPD: garantisce convergenza rapida, tempi ridottissimi e accuratezza elevata.

In prospettiva, l'adozione del Coniugato è la scelta naturale in contesti applicativi reali, mentre gli altri metodi restano strumenti preziosi per comprendere i concetti fondamentali di iterazione e convergenza.