Metodi del calcolo scientifico

Libreria per risoluzione di sistemi lineari con metodi iterativi

Volpato Mattia 866316 *
Andreotti Stefano 851596 †

Appello di Giugno 2024

*m.volpato4@campus.unimib.it
†s.andreotti7@campus.unimib.it

Contents

1 Introduzione

1.1 Obiettivo

Lo scopo del progetto è la realizzazione di una mini-libreria per la risoluzione di sistemi lineari (limitatamente al caso di matrici simmetriche e definite positive), in particolare che implementi:

- i *metodi iterativi stazionari* di **Jacobi** e di **Gauss-Seidel**;
- i *metodi iterativi non stazionari* del **gradiente** e del **gradiente coniugato**.

Tutti i metodi risolutivi verranno testati su quattro matrici in *formato sparso*, descritte nella sezione 1.3. Per tutti i grafici riportati verrà applicata una **scala logaritmica** sull'asse delle ascisse.

1.2 Struttura della libreria

Tutto il codice della libreria è disponibile in questa repository.

1.2.1 Scelte implementative

Si è scelto di implementare la libreria in **python** per tre motivi principali:

- la facilità d'uso del linguaggio e l'ampio supporto fornito dalla comunità, che mette a disposizione **librerie efficienti** le quali limitano il principale lato negativo del linguaggio (ovvero le performance sul tempo di esecuzione);
- la popolarità del linguaggio, quindi il fatto che potenzialmente la libreria possa essere usata da molte persone;
- la possibilità di fare utilizzo dei *jupyter notebook*, un formato di file eseguibile composto da celle di testo e di codice, che permettono una migliore interpretazione dei risultati e facilità di visualizzazione delle matrici.

1.2.2 Architettura

L'intera **architettura della libreria** è riportata nel grafico 1.

La libreria è stata sviluppata cercando di sfruttare il più possibile la struttura base dei *metodi iterativi*, i quali condividono un *algoritmo comune* e si differenziano solo per il metodo di calcolo della successiva iterata: infatti, il *criterio d'arresto* usato risulta sempre essere il **residuo scalato**, poichè quasi tutti i metodi lo calcolano durante l'aggiornamento, risultando quindi computazionalmente più efficiente. Inoltre, si impone anche un limite sul *numero massimo di iterazioni*, oltre il quale si considera la esecuzione interessata fallita.

Lo pseudocodice generico per un qualsiasi **metodo iterativo** (Algoritmo 1)

Algorithm 1 Metodo Iterativo

```
function IterativeSolver(Matrix A, vector b)
    x = initialize_x_0(A)
    k = 0
    r = b − Ax
    while not check_termination(r, b) do
        x = update_x(A, b, x)
        k = k + 1
        if k > MAX_ITERATIONS then
            State Convergence_Error
        r = b − Ax
    return x
```

viene implementato nel metodo *solve* della classe padre astratta *Solver* come segue:

```python
def solve(self, A:sp.sparse.csr_matrix, b:np.ndarray, support:any=None) ->
    np.ndarray:

    x = self._initialize_x_0(A.shape[1])
    k = 0
    r = b - A.dot(x)

    while not self._check_termination(r, b):
        x, r, support = self._update_x(A, b, x, support)
        k += 1

        if k > self.max_iter:
            self.iter = k
            raise MaxIterationException(f"Max iteration reached: {k}")

    self.iter = k
    return x
```

L'utilizzo della variabile *support* permette di evitare calcoli ripetuti inutili nei metodi che necessitano delle strutture di supporto aggiuntive, come in **Jacobi** (sezione 2.1) e **Gauss-Seidel** (sezione 2.2), permettendo a ogni classe figlia di implementare solamente il metodo *update_x* per il calcolo della iterata successiva (che in *Solver* è astratto).

I metodi di **inizializzazione della soluzione** e di **controllo della terminazione** risultano essere condivisi da tutti i metodi iterativi, e di conseguenza sono implementati nella classe *Solver*. In particolare:

- *initialize_x_0* permette di inizializzare la *soluzione iniziale* in due maniere:
  - come il **vettore nullo** $\underline{0}$;
  - come un **vettore** inizializzato **casualmente** con valori reali compresi tra due bound;
- *check_termination* rappresenta il **criterio di terminazione**, il quale controlla che il rapporto tra la norma del **residuo** e la norma di $\boldsymbol{b}$ sia inferiore a una **tolleranza** piccola a piacere:

$$\frac{\|A\underline{x}^{(k)} - \underline{b}\|}{\|\underline{b}\|} < \epsilon \tag{1}$$

Nel caso venga superato il **numero massimo di iterazioni**, viene generata un'eccezione *MaxIterationExcpetion* che interrompe l'esecuzione; inoltre, è sempre possibile accedere al numero di iterazioni dell'ultima esecuzione attraverso l'attributo di classe *iter*.

Infine, il metodo di **Gauss-Seidel** richiede la risoluzione di un *sistema lineare triangolare inferiore* per il calcolo dell'**iterata successiva**: a tal fine è stata implementata la classe *TrilSolver*, che si occupa appunto di risolvere tali sistemi tramite la procedura *forward_substitution*. É stata anche fornita un'implementazione di tale procedura (*forward_substitution_naive*) che tuttavia, dovendo far uso dei cicli *for* di **python**, rende il metodo iterativo lento rispetto agli altri (che fanno uso solo di **operazioni tra vettori**) in maniera *unfair*. Per far fronte a questo problema, nel **benchmark** è stata utilizzata un'implementazione efficiente della *forward_substituion* presa dalla libreria **scipy**.

Nel benchmark, il vettore $\underline{b}$ è stato creato appositamente in modo da ottenere una soluzione $\underline{x}$ di soli 1, come segue:

$$\underline{b} = A \cdot \underline{x} = A \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

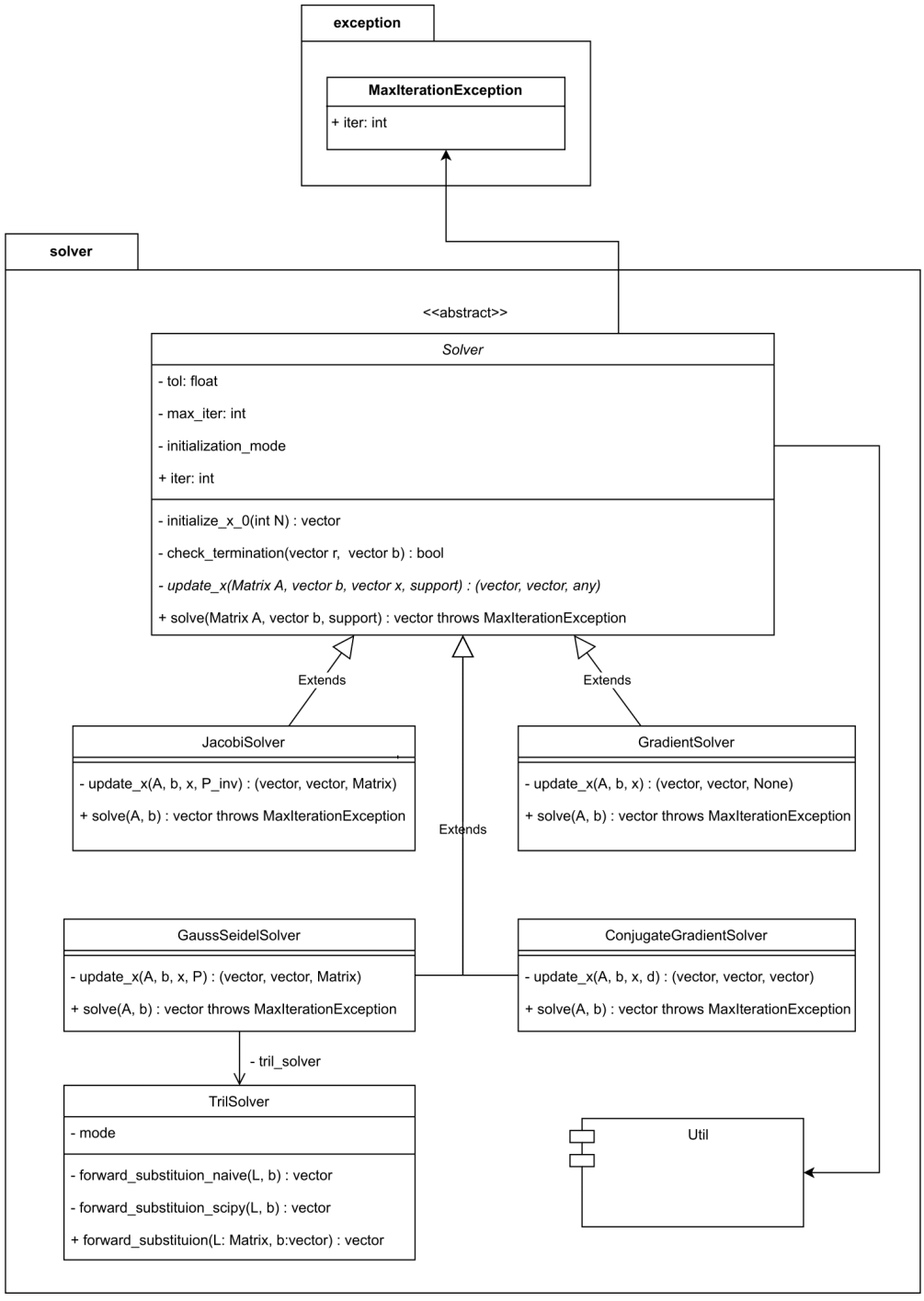


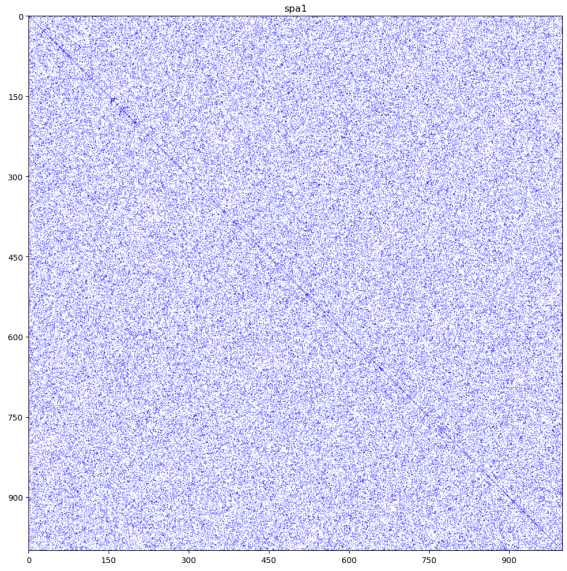
Figure 1: Architettura della libreria.

1.3 Matrici utilizzate

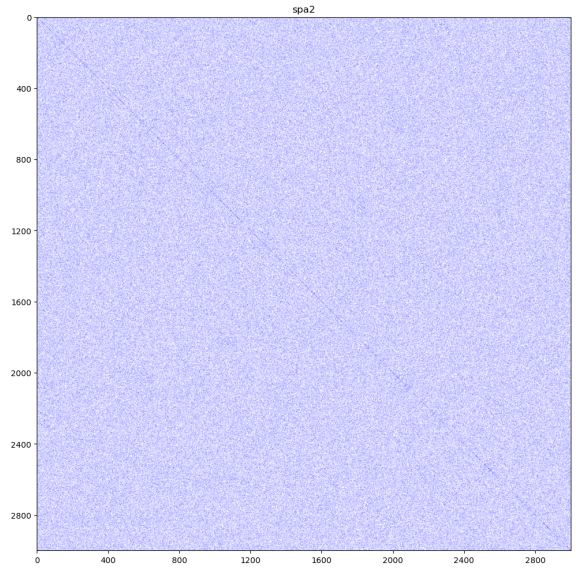
Al fine di testare la libreria sono state utilizzate quattro matrici in **formato sparso**:

- Matrice **spa1** (figura 2a):
  - Dimensione: 1000×1000
  - Entrate non zero: 182264
  - Indice di sparsità: 18.23%
- Matrice **spa2** (figura 2b)
  - Dimensione: 3000×3000
  - Entrate non zero: 161738
  - Indice di sparsità: 18.13%
- Matrice **vem1** (figura 3a)
  - Dimensione: 1681×1681
  - Entrate non zero: 13385
  - Indice di sparsità: 0.47%
- Matrice **vem2** (figura 3b)
  - Dimensione: 2601×2601
  - Entrate non zero: 21225
  - Indice di sparsità: 0.31%

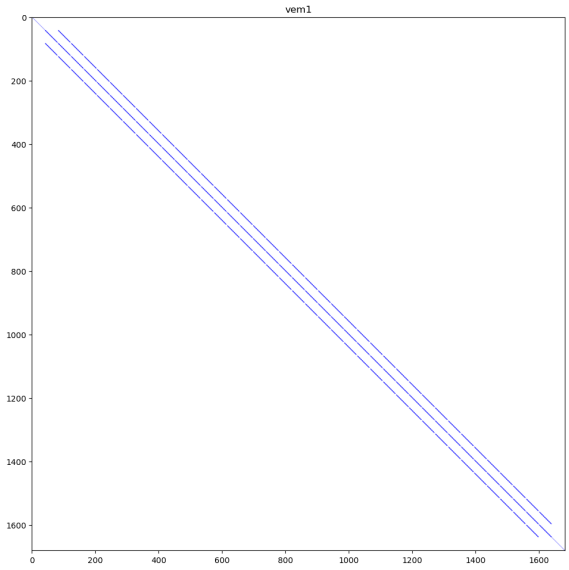
Nei grafici associati sono riportate le distribuzioni delle **entrate diverse da zero**; si noti in particolare come **spa1** e **spa2** siano *matrici sparse* mentre **vem1** e **vem2** risultino essere *matrici a bande*.



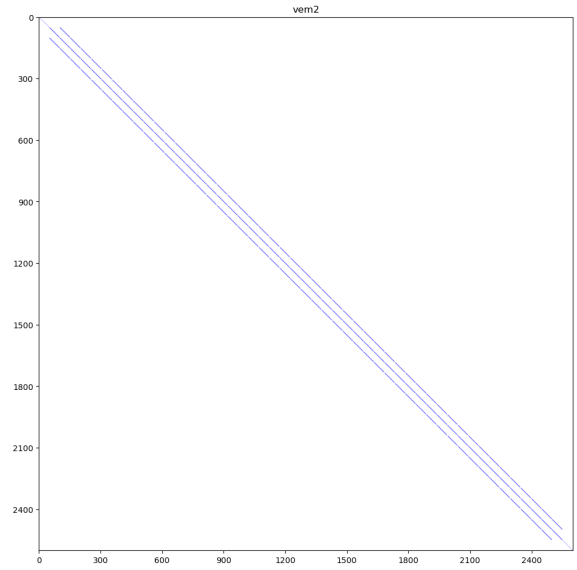
(a) Matrice **spa1**



(b) Matrice **spa2**



(a) Matrice **vem1**



(b) Matrice **vem2**

2 Metodi iterativi stazionari

I **metodi iterativi stazionari** usano una strategia basata sullo **splitting** per calcolare la soluzione approssimata $\underline{x}$. La decomposizione su cui si basano è:

$$A = P - N \tag{2}$$

Questi metodi sono chiamati *stazionari* poiché il calcolo dell'iterata successiva non dipende dall'iterazione corrente. Per questa categoria, nella libreria sono implementati i metodi di **Jacobi** (sezione 2.1) e di **Gauss-Seidel** (sezione 2.2).

2.1 Metodo di Jacobi

Il **metodo di Jacobi** si basa sull'uso della matrice P^{-1} e del *residuo* per calcolare l'iterata successiva, dove P è la matrice diagonale costruita (appunto dalla diagonale) di A; in particolare, la matrice P^{-1} si ottiene semplicemente dai reciproci degli elementi sulla diagonale di P. Da un punto di vista computazionale, la matrice P (e quindi anche P^{-1}) rimane la stessa per tutta l'esecuzione del metodo e, di conseguenza, si calcola una sola volta. La *k-esima iterata* è data da

$$x^{(k+1)} = x^{(k)} + P^{-1}(b - Ax^{(k)}) \tag{3}$$

Si noti che all'interno del calcolo dell'iterata è presente anche il calcolo *residuo*; di conseguenza, non sarà necessario calcolarlo nuovamente per verificare la **condizione di terminazione**.

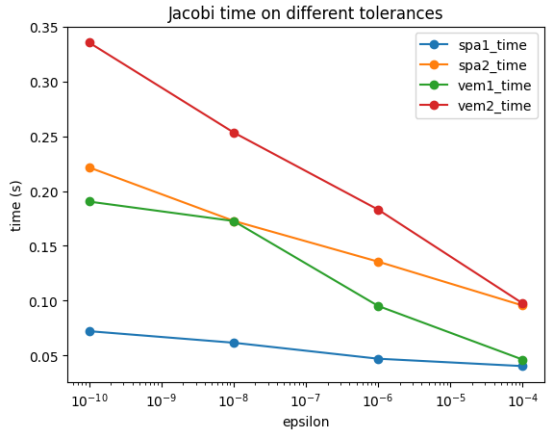
2.1.1 Implementazione

Il calcolo dell'iterata è implementato nella libreria come

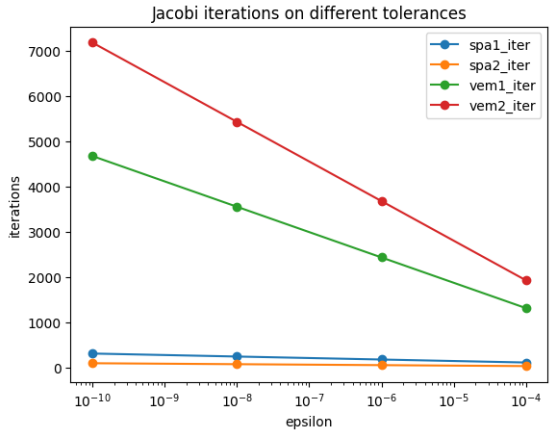
```
1 def _update_x(self, A:sp.sparse.csr_matrix, b:np.ndarray, x:np.ndarray,
2           P_inv:sp.sparse.csr_matrix) -> tuple[np.array, np.array, sp.
                sparse.csr_matrix]:
3
4     r = b - A.dot(x)
5     x = x + P_inv.dot(r)
6
7     return x, r, P_inv
```

2.1.2 Risultati

I risultati possono essere verificati manualmente usando la classe implementata nel file *JacobiSolver.py* oppure più facilmente mediante il notebook **benchmark.ipynb**.



(a) **Jacobi** rispetto al **tempo di esecuzione**



(b) **Jacobi** rispetto al **numero di iterazioni**

Dall'analisi della figura 4a si può notare un'ottima **efficienza** nei tempi di esecuzione di questo metodo. Una seconda osservazione può invece essere fatta rispetto all'immagine 4b, dove si vede chiaramente come la *struttura delle matrici* influenzi il **numero di iterazioni**: le *matrici sparse* **spa1** e **spa2** convergono dopo poche iterazioni, mentre le *matrici a bande* **vem1** e **vem2** impiegano molte più iterazioni. Ciò è dovuto con ogni probabilità alla stazionarietà del metodo; infatti questo fenomeno si verificherà anche nel metodo di **Gauss-Seidel**, mentre quando si analizzeranno i metodi non stazionari si noterà una grande differenza a livello di iterazioni sulle specifiche matrici.

Nella tabella 1 vengono confrontati i risultati ottenuti con le caratteristiche delle matrici (fissando la **tolleranza** a $\epsilon = 1e^{-}8$):

Matrix	N	Non-zero entry	Sparsity index	Time (s)	Iterations
spa1	1000	182264	0.182264	0.061566	248
spa2	3000	1631738	0.181304	0.172829	79
vem1	1681	13385	0.004737	0.172698	3553
vem2	2601	21225	0.003137	0.25356	5426

Table 1: Risultati delle esecuzioni del **metodo di Jacobi** per $\epsilon = 1e^{-}8$

2.2 Metodo di Gauss-Seidel

Il metodo di **Gauss-Seidel** è una variante del metodo di **Jacobi**, nel quale le matrici P e N sono rispettivamente la *triangolare inferiore* e la *triangolare superiore* (senza la diagonale principale) della matrice A.

Il calcolo della matrice P^{-1} si effettua tramite la risoluzione del sistema lineare $Py = r^{(k)}$, che è possibile risolvere applicando la procedura di *sostituzione in avanti*, essendo la matrice P triangolare inferiore.

La *k-esima* iterata è calcolata come:

$$x^{(k+1)} = x^{(k)} + y \tag{4}$$

Come già riportato nella sezione 1.2.2, nella libreria sono presenti due modalità di calcolare questa matrice: il primo è l'utilizzo della funzione *'naive'* implementata da noi, mentre il secondo è usare la più efficiente libreria **scipy**: nel benchmark, per consentire contronti *fair* con gli altri metodi, verrà utilizzata la versione di **scipy**.

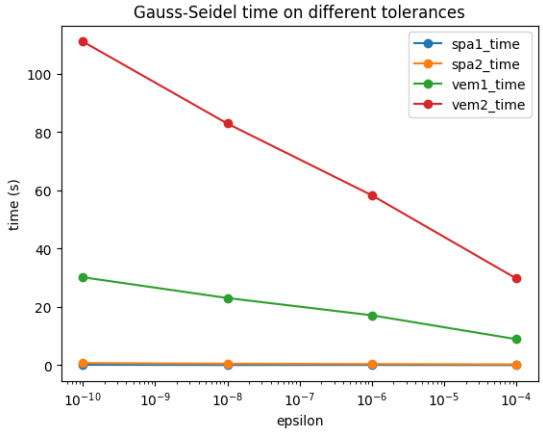
2.2.1 Implementazione

Il calcolo dell'iterata è implementato nella libreria come

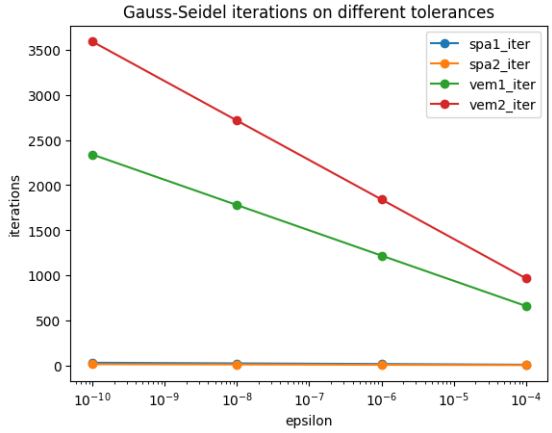
```
def _update_x(self, A:sp.sparse.csr_matrix, b:np.ndarray, x:np.ndarray, P:
    sp.sparse.csr_matrix) -> tuple[np.array, np.array, sp.sparse.csr_matrix
    ]:

     r = b - A.dot(x)
     y = self._tril_solver.forward_substitution(P, r)
     x = x + y

     return x, r, P
```

2.2.2 Risultati

I risultati possono essere verificati manualmente usando la classe implementata nel file *GaussSeidelSolver.py* oppure più facilmente mediante il notebook **benchmark.ipynb**.



(a) **Gauss-Seidel** rispetto al **tempo di esecuzione**



(b) **Gauss-Seidel** rispetto al **numero di iterazioni**

Come in **Jacobi**, con riferimento alla figura 5a si può notare come le *matrici sparse* **spa1** e **spa2** risultino convergere in molto meno tempo rispetto alle *matrici a bande* **vem1** e **vem2**; in maniera analoga, anche il **numero di iterazioni** dipende strettamente dalla struttura della matrice. Da questi dati si può pensare di dedurre che l'**efficienza** di **Gauss-Seidel** sia inversamente proporzionale alla **sparsità** di una matrice: ovvero, più una matrice è sparsa e più il metodo di **Gauss-Seidel** fa fatica a convergere.

Nella tabella 2 vengono confrontati i risultati con alle caratteristiche le matrici (fissando la **tolleranza** a $\epsilon = 1e^{-}8$):

Matrix	N	Non-zero entry	Sparsity index	Time (s)	Iterations
spa1	1000	182264	0.182264	0.09167	25
spa2	3000	1631738	0.181304	0.538953	13
vem1	1681	13385	0.004737	23.076995	1779
vem2	2601	21225	0.003137	82.959381	2715

Table 2: Risultati delle esecuzioni del metodo di **Gauss-Seidel** per $\epsilon = 1e^{-}8$

3 Metodi iterativi non stazionari

I **metodi iterativi non stazionari** si basano su una diversa formula di aggiornamento della soluzione approssimata, cioè

$$x^{(k+1)} = x^{(k)} + \alpha_k P^{-1} r^{(k)} \tag{5}$$

La differenza sta nel coefficiente α, che in questo non è stazionario ma bensì dipende dall'iterazione precedente.

3.1 Metodo di discesa del gradiente

Il **metodo del gradiente** interpreta una matrice *A simmetrica e definita positiva* e una funzione ι come un paraboloide del quale si vuole trovare il *punto di minimo*, che corrisponde alla soluzione del sistema lineare. Ne segue che la nuova iterazione può essere calcolata come

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} \tag{6}$$

Il calcolo di α_k risulta essere quello più oneroso:

$$\alpha_k = ((r^{(k)})^t r^{(k)})/((r^{(k)})^t A r^{(k)}) \tag{7}$$

3.1.1 Implementazione

Il calcolo dell'iterata è implementato nella libreria come

```
1 def _update_x(self, A:sp.sparse.csr_matrix, b:np.ndarray, x:np.ndarray, _:
      any=None) -> tuple[np.array, np.array, any]:
2     r = b - A * x
3     y = A * r
4 
5     a = r @ r
6     c = r @ y
7     x = x + a / c * r
8 
9     return x, r, _
```

3.1.2 Risultati

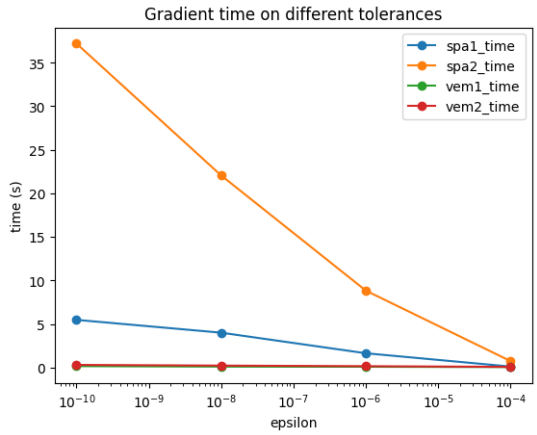
I risultati possono essere verificati manualmente usando la classe implementata nel file *GradientSolver.py* oppure più facilmente mediante il notebook **benchmark.ipynb**.

É possibile notare come, per questi metodi, la situazione si ribalti rispetto ai **metodi stazionari**: le *matrici a bande* **vem1** e **vem2** risulatno molto più veloci a convergere di quelle *sparse* **spa1** e **spa2**; inoltre, il **numero medio di iterazioni** per matrice aumenta e sembra esserci leggermente meno correlazione tra il **numero di iterazioni** e il **tempo di esecuzione**.

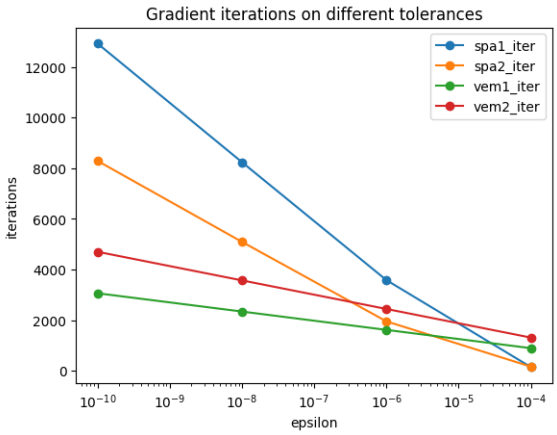
Nella tabella 3 vengono confrontati i risultati con le caratteristiche delle matrici (fissando la **tolleranza** a $\epsilon = 1e^{-}8$):

Matrix	N	Non-zero entry	Sparsity index	Time (s)	Iterations
spa1	1000	182264	0.182264	4.007221	8234
spa2	3000	1631738	0.181304	22.060645	5088
vem1	1681	13385	0.004737	0.109381	2337
vem2	2601	21225	0.003137	0.22854	3567

Table 3: Risultati delle esecuzioni del **metodo di discesa del gradiente** per $\epsilon = 1e^{-}8$



(a) **Gradiente** rispetto al **tempo di esecuzione**



(b) **Gradiente** rispetto al **numero di iterazioni**

3.2 Metodo di discesa del gradiente coniugato

Il **metodo del gradiente coniugato** è un'ottimizzazione del **metodo del gradiente** che pone rimedio al fenomeno dello *zig-zag*, dovuto alla grande differenza tra gli **autovalori minimo e massimo** della matrice: $\lambda_{min} << \lambda_{max}$. Per migliorare il metodo si definisce un **vettore ottimale** rispetto ad una direzione come:

$$d * r^{(k)} = 0 \tag{8}$$

In questo modo si ottiene un **vettore ottimale** per quella direzione che idealmente non verrà più modificato lungo la direzione d. Nella pratica la differenza rispetto al **metodo del gradiente** è nel calcolo di α_k, che diventa:

$$\alpha_k = ((d^{(k)})^t r^{(k)})/((d^{(k)})^t A d^{(k)}) \tag{9}$$

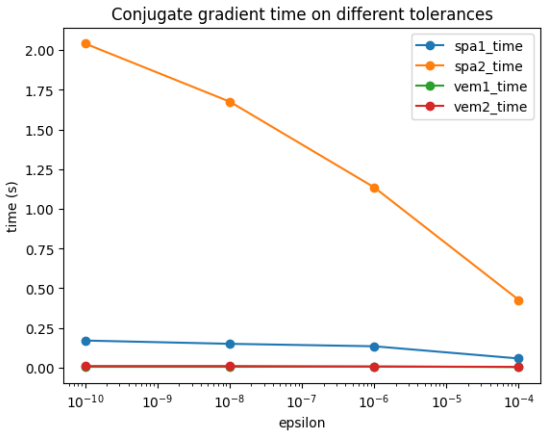
Mentre il calcolo dell'iterata successiva rimane

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)} \tag{10}$$

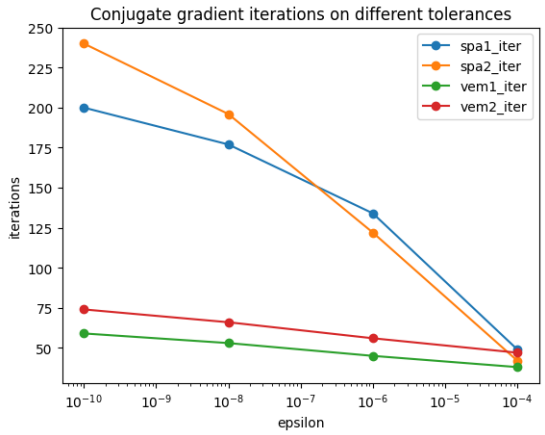
3.2.1 Implementazione

Il calcolo dell'iterata è implementato nella libreria come

```
1  def _update_x(self, A:sp.sparse.csr_matrix, b:np.ndarray, x:np.ndarray, d:
       np.array) -> tuple[np.array, np.array, np.array]:
2      r = b - A * x
3      y = A * d
4
5      alpha = (d @ r) / (d @ y)
6      x = x + alpha * d
7
8      r = b - A * x
9      w = A * r
10     beta = (d @ w) / (d @ y)
11     d = r - beta * d
12
13     return x, r, d
```



(a) **Gradiente coniugato** rispetto al **tempo di esecuzione**



(b) **Gradiente conigato** rispetto al **numero di iterazioni**

3.2.2 Risultati

I risultati possono essere verificati manualmente usando la classe implementata nel file *ConjugateGradientSolver.py* oppure più facilmente mediante il notebook **benchmark.ipynb**.

Questo metodo mantiene i vantaggi visti nelle *matrici a bande* rispetto alle *matrici sparse* descritti nella sezione 3.1, portando netti miglioramenti sia dal punto di vista del **tempo di esecuzione** (figura 7a) che del **numero di iterazioni** (figura 7b). Infatti, come da attesa teorica, il **numero medio di iterazioni** per convergere risulta essere *molto minore* di quello del **gradiente** (figura 6b), ponendo rimedio al fenomeno dello *zig-zag*.

Nella tabella 4 vengono confrontati i risultati con le caratteristiche delle matrici (fissando la **tolleranza** a $\epsilon = 1e^{-}8$):

Matrix	N	Non-zero entry	Sparsity index	Time (s)	Iterations
spa1	1000	182264	0.182264	0.150343	177
spa2	3000	1631738	0.181304	1.675488	196
vem1	1681	13385	0.004737	0.005208	53
vem2	2601	21225	0.003137	0.010405	66

Table 4: Risultati delle esecuzione del **metodo di discesa del gradiente coniugato** per $\epsilon = 1e^{-}8$

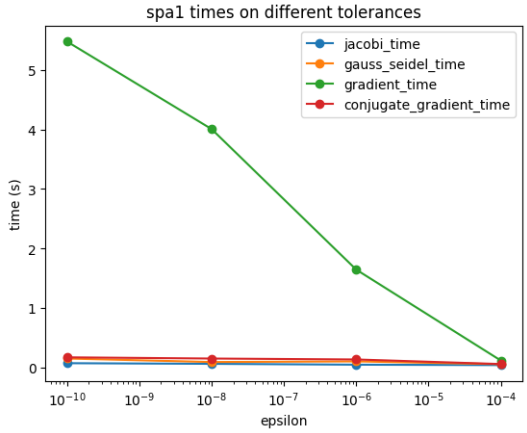
4 Risultati per matrice

In questa sezione riportiamo gli stessi risultati mostrati in precedenza raggruppati per *matrice* anzichè per *metodo iterativo*, al fine da avere una visione più chiara dei migliori metodi iterativi a seconda delle caratteristiche della matrice.

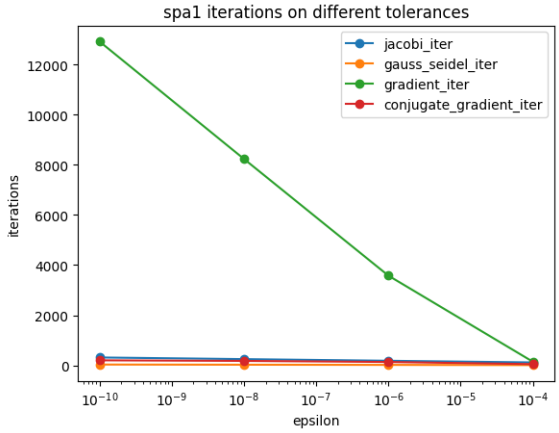
4.1 Matrici spa1 e spa2

Dalle figure 8a, 8b, 9a e 9b si osserva come i migliori **metodi iterativi** per questo tipo di *matrici sparse* siano quelli *stazionari* (ovvero **Jacobi** e **Gauss-Seidel**); nonostante questo, anche il **gradiente coniugato** riesce a raggiungere buone performance, mentre andrebbe evitato l'utilizzo del **gradiente**, decisamente meno efficiente.

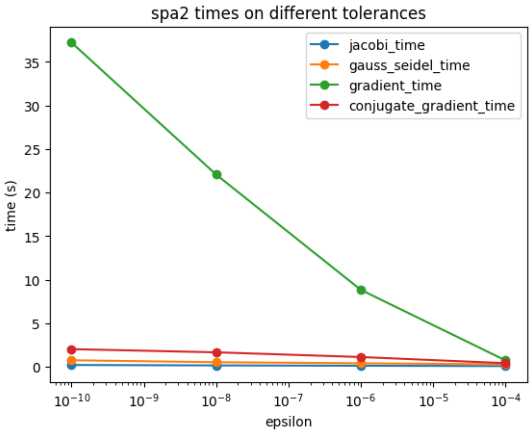
Risulta inoltre evidente una *forte corrispondenza* tra **tempi di esecuzione** e **numero di iterazioni**.



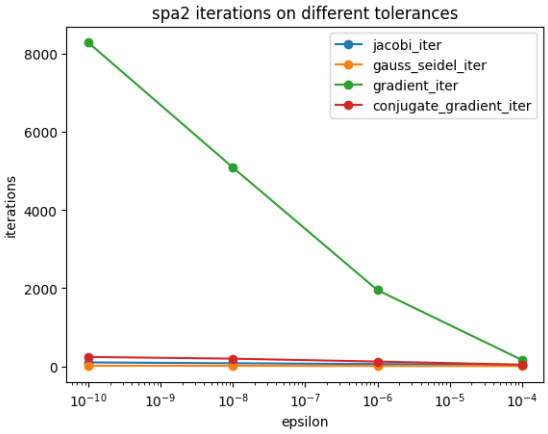
(a) *Tempi di esecuzione* su **spa1**



(b) *Numero di iterazioni* su **spa1**



(a) *Tempi di esecuzione* su **spa2**

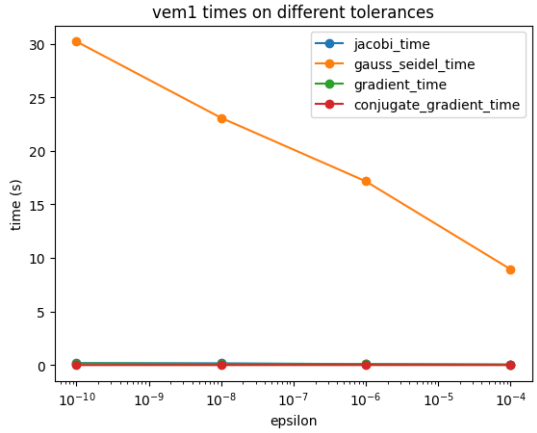


(b) *Numero di iterazioni* su **spa2**

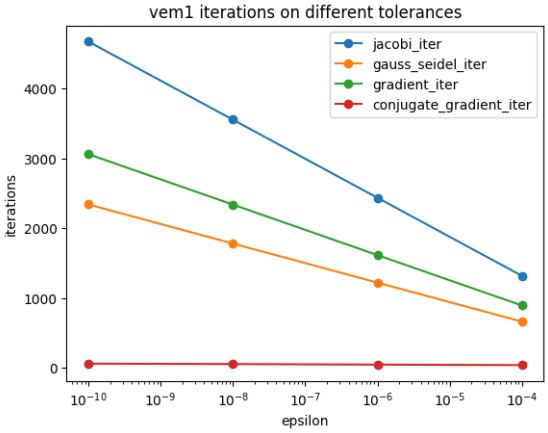
4.2 Matrici vem1 e vem2

Dall'analisi delle figure 10a, 10b, 11a e 11b, su questo tipo di *matrici a bande* vale un discorso simile a quello precedente, in cui però i metodi migliori diventano quelli **non stazionari**, mentre il metodo da evitare risulta essere **Gauss-Seidel**.

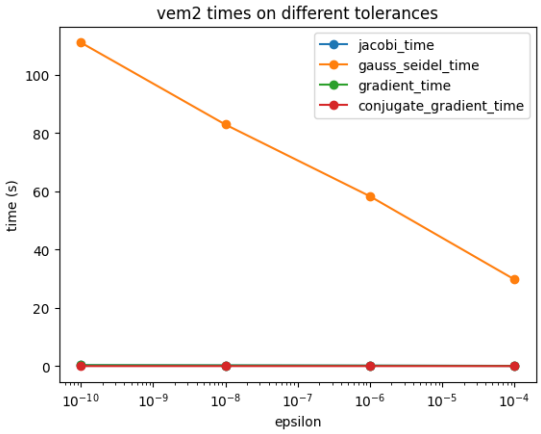
É possibile anche notare l'assenza di una corrispondenza immediata tra **tempi di esecuzione** e **numero di iterazioni**: con ogni probabilità, questo risulta essere una conseguenza della *forte sparsità* delle matrici, che viene sfruttata per eseguire *operazioni tra vettori ottimizzate*. Questo spiegherebbe in particolare la grande efficienza di **Jacobi** rispetto a **Gauss-Seidel**: infatti, l'aggiornamento della soluzione del primo coinvolge solo operazioni tra vettori e matrici, mentre nel secondo risulta necessario accedere ai singoli valori scalari nella procedura di *forward substitution*.



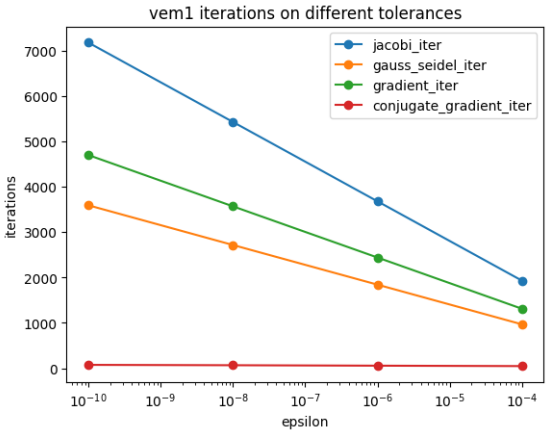
(a) *Tempi di esecuzione* su **vem1**



(b) *Numero di iterazioni* su **vem1**



(a) *Tempi di esecuzione* su **vem2**



(b) *Numero di iterazioni* su **vem2**

5 Conclusioni

In conclusione, è possibile constatare come ogni metodo iterativo abbia performance più o meno efficienti a seconda della *struttura* della matrice che si sta trattando; data una matrice risulta quindi importante scegliere il metodo più adatto alle sue caratteristiche, anche potenzialmente andando a investire un po' **di tempo di calcolo** sull'analisi degli aspetti più importanti della matrice considerata.