Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

Progetto 1 Bis
Metodi del Calcolo Scientifico

Relazione di:
Matteo Cavaleri - 875050
Cristian Piacente - 866020

Anno Accademico 2023-2024

Indice

1 Introduzione **2**

2 Struttura della libreria implementata **3**

3 Analisi dei risultati **8**

3.1 Analisi del tempo di esecuzione di ogni solutore per ogni tolleranza per ogni matrice 15

3.2 Analisi dell'errore relativo di ogni solutore per ogni tolleranza per ogni matrice 18

3.3 Analisi dell'errore relativo di ogni solutore per ogni iterazione per ogni tolleranza per ogni matrice 22

4 Conclusioni **39**

Capitolo 1

Introduzione

Lo scopo del progetto 1_*bis* del corso di metodi del calcolo scientifico prevede di sviluppare una libreria di solutori iterativi per matrici simmetriche definite positive. Sono stati implementati i seguenti solutori: *Jacobi*, *Gauss - Seidel*, *Gradiente* e *Gradiente coniugato*.

Capitolo 2

Struttura della libreria implementata

Per evitare duplicazioni di codice e facilitare modifiche per possibili sviluppi futuri è stato scelto di implementare una classe astratta chiamata *AbstractIterator* e quattro oggetti custom che ereditano la classe sopra citata. Gli oggetti figlio sono: $Jacobi$, $Gauss-Seidel$, $Gradiente$ e $Gradiente\ coniugato$. Il codice è organizzato come segue:

```
progetto 1_bis
    |
    |-> matrici
    |   |
    |   |-> spa1.mtx
    |       spa2.mtx
    |       vem1.mtx
    |       vem2.mtx
    |
    |-> metodi_iterativi
    |   |
    |   |-> utils
    |   |   |
    |   |   |-> __init__.py
    |   |       utils.py
    |   |
    |   |-> __init.py
    |       AbstractIterator.py
    |
```

```
|-> report_experiment
|   |
|   |-> report.txt
|
|-> run.py
```

Di seguito verrà fornita una breve spiegazione dell'architettura sopra mostrata.

- *matrici*: contiene i file rappresentanti le matrici da analizzare. Ogni file fornito è stato modificato aggiungendogli la riga di header del formato .mtx. Senza l'header il metodo *scipy.io.mmread* utilizzato nel metodo *read* in utils lancia un'eccezione.

- *metodi_iterativi*:

  - *utils*: contiene il file *__init__.py* per far capire a Python che è un package e il modulo *utils.py* il quale contiene due metodi: *read* per leggere ed aprire la matrice in formato .mtx e *write_report_and_logs* che va a scrivere su *report.txt* il report del programma e mostra dei log durante l'esecuzione.

  - *__init__.py*: serve per far capire a Python che il folder *metodi_iterativi* è un package.

  - *AbstractIterator.py*: contiene la classe astratta e le classi custom. Ad *AbstractIterator* viene passato nel costruttore *matrice_A* e *tol*, volendo si può scegliere di customizzare il numero massimo di iterazioni passando anche *max_iter* che di default è settato a 20000. Su *max_iter* viene effettuato un controllo tale che se il numero passato è inferiore a 20000 viene ugualmente settato a 20000. Gli attributi salvati sono: *matrice_A*, $x\_perfect$, x, b, tol, k, *max_iter* e *b_norma*. k rappresenta il numero di iterazioni compiute sull'oggetto. Vengono implementati i metodi per calcolare il residuo, l'errore relativo, effettuare l'aggiornamento di x ad ogni iterazione, vedere se x converge e eseguire la risoluzione del sistema lineare. Le classi custom creano degli attributi specifici che servono solo per loro ed effettuano l'override del metodo di aggiornamento di x secondo le specifiche del solutore. Nello specifico la classe figlia Jacobi come ulteriore attributo calcola P^{-1} e effettua l'override del metodo di aggiornamento. La classe figlia Gauss-Seidel come attributo aggiuntivo ha P ovvero la matrice triangolare inferiore della matrice originale di partenza e effettua l'override del metodo di aggiornamento. La classe figlia Gradiente effettua solamente l'override del metodo di aggiornamento. Infine la classe figlia GradienteConiugato

crea un ulteriore attributo d il quale inizialmente è uguale al residuo. Viene inolte effettuato l'override del metodo di aggiornamento.

- *report_experiment*: contiene il file chiamato *report.txt* il quale è il resoconto di tutte le esecuzioni.
- *run.py*: questo file permette di eseguire l'esecuzione di tutti i metodi di aggiornamento per tutte le matrici considerando ogni tolleranza.

L'esecuzione del codice inizia dal modulo *run.py* (si può vedere il codice più significativo di esso nell'immagine 2.1) incominciando ad eseguire tutti i metodi iterativi per ogni istanza per ogni matrice. Nel ciclo for più interno possiamo vedere come ogni oggetto custom dopo essere creato richiama il metodo *solve*.
Per risolvere il sistema lineare, viene quindi chiamato il metodo *solve*, il quale inizia ad iterare fino a quando la soluzione converge oppure, il numero massimo di iterazioni è stato raggiunto. Nel corpo del ciclo while notiamo che il vettore che rappresenta le x viene aggiornato richiamando il metodo *update*. Questo metodo nella classe madre è astratto ma quando viene richiamto da un oggetto figlio utilizza la sua implementazione specifica 2.2. Per quanto riguarda le implementazioni specifiche dei metodi di aggiornamento, essi si possono visionare nelle figure 2.3 per Jacobi, 2.4 per Gauss-Seidel, 2.5 per il gradiente e 2.6 per il gradiente coniugato.

```python
write_report_and_logs(output_path, output_file, "              RESOCONTO DELLE ESECUZIONI\n")
for m in matrices:
    write_report_and_logs(output_path, output_file, "**************************************************\n")
    mat = str(m.split("\\")[-1])
    write_report_and_logs(output_path, output_file, "                    "+mat+"\n")
    write_report_and_logs(output_path, output_file, "**************************************************\n")
    for t in tols:
        mat = read(m)
        write_report_and_logs(output_path, output_file, "**************************************************\n")
        write_report_and_logs(output_path, output_file, "                tolleranza:"+str(t)+"\n")
        write_report_and_logs(output_path, output_file, "**************************************************\n")
        write_report_and_logs(output_path, output_file, "Algoritmo utilizzato: Jacobi\n")

        jacobi = Jacobi(mat, t)
        jacobi.solve()

        write_report_and_logs(output_path, output_file, "Algoritmo utilizzato: Gauss-Seidel\n")
        gs = GaussSeidel(mat, t)
        gs.solve()

        write_report_and_logs(output_path, output_file, "Algoritmo utilizzato: Gradiente\n")
        gr = Gradiente(mat, t)
        gr.solve()

        write_report_and_logs(output_path, output_file, "Algoritmo utilizzato: Gradiente congiugato\n")
        gr_con = GradienteConiugato(mat, t)
        gr_con.solve()
```

Figura 2.1: `run.py`

```python
class AbstractIterator(ABC):

    def solve(self):
        path_report = 'report_experiment/'
        file_report = 'report.txt'
        start = time.time() #incomincio a cronometrare
        while self.non_converge(): # se la soluzione converge smetto di iterare, altrimenti continuo
            self.x = self.update() # richiamo un determinato metodo iterativo tra i 4 possibili a seconda del tipo dell'oggetto
            self.k += 1
            if self.k > self.max_iter: # controllo se ho raggiunto il massimo numero di iterazioni
                write_report_and_logs(path_report, file_report, "!!! Numero massimo di iterazioni raggiunte !!!! il metodo non converge\n")
                break
        end = time.time() - start # smetto di cronometrare
        # report su quanto eseguito
        write_report_and_logs(path_report, file_report, "Numero di iterazioni: " + str(self.k) + "\n")
        write_report_and_logs(path_report, file_report, "Errore relativo: " + str(self.relative_error()) + "\n")
        write_report_and_logs(path_report, file_report, "tempo di esecuzione: " + str(round(end, 3)) + " (s)" + "\n")
        write_report_and_logs(path_report, file_report, "\n")
```

Figura 2.2: *AbstractIterator.py*

```python
def update(self):
    #         x(k) + P^-1 * r(k)
    return self.x + (self.diag_inv * self.get_residuo())
```

Figura 2.3: implementazione specifica del metodo di Jacobi

```
def update(self):
    #        x(k) +  risultato: P * y = r(k)
    return self.x + spsolve(self.triang_inf, self.get_residuo())
```

Figura 2.4: implementazione specifica del metodo di Gauss-Seidel

```
def update(self):
    a, b = self.get_a_b()
    alpha = a / b
    #       x(k)  +    alpha * r(k)
    return self.x + (alpha * self.get_residuo())
```

Figura 2.5: implementazione specifica del metodo del gradiente

```
def update(self):
    y = self.matrice_A.dot(self.d)
    #z = self.matrice_A.dot(self.d)
    alpha = (self.d.dot(self.get_residuo())) / (self.d.dot(y))
    self.x = self.x + (alpha * self.d)
    nuovo_res = self.get_residuo()
    w = self.matrice_A.dot(nuovo_res)
    beta = (self.d.dot(w)) / (self.d.dot(y))
    self.d = nuovo_res - (beta * self.d)
    return self.x
```

Figura 2.6: implementazione specifica del metodo del gradiente coniugato

Capitolo 3

Analisi dei risultati

Di seguito mostreremo i risultati ottenuti da tutte le esecuzioni dei solutori implementati. Guardando semplicemnte le tabelle 3.1, 3.2, 3.3 e 3.4 non siamo in grado di fare osservazioni elaborate. Ciò che possiamo affermare con certezza è che per ogni solutore e per ogni matrice al crescere delle iterazioni e al diminuire della tolleranza otteniamo un errore relativo sempre minore su una determinata matrice. Quest'osservazione, seppur veritiera, non fornisce una panoramica completa delle esecuzioni, pertanto abbiamo calcolato ultriori informazioni e le abbiamo inserite nei grafici sotto mostrati per essere il più chiari possibile.

matrice	**tolleranza**	**iterazioni**	**t. esec. (s)**	**err. rel.**
spa1.mtx	0.0001	115	0.051	0.0017708118132163107
spa1.mtx	1e-06	181	0.063	1.797924734135133e-05
spa1.mtx	1e-08	247	0.166	1.824978826804907e-07
spa1.mtx	1e-10	313	0.141	1.8524371366782517e-09
spa2.mtx	0.0001	36	0.133	0.0017667164062951787
spa2.mtx	1e-06	57	0.282	1.6667519292204132e-05
spa2.mtx	1e-08	78	0.307	1.572869932678427e-07
spa2.mtx	1e-10	99	0.297	1.4842717026807931e-09
vem1.mtx	0.0001	1314	0.215	0.0035503020437906226
vem1.mtx	1e-06	2433	0.402	3.540172346953882e-05
vem1.mtx	1e-08	3552	0.466	3.53976694659024e-07
vem1.mtx	1e-10	4671	0.560	3.5394587549435938e-09
vem2.mtx	0.0001	1927	0.285	0.004988120299885087
vem2.mtx	1e-06	3676	0.553	4.967230366467791e-05
vem2.mtx	1e-08	5425	1.036	4.965609860645763e-07
vem2.mtx	1e-10	7174	0.984	4.964185239985712e-09

Tabella 3.1: Risultati ottenuti dall'esecuzione di Jacobi su diverse matrici e tolleranze

matrice	**tolleranza**	**iterazioni**	**t. esec. (s)**	**err. rel.**
spa1.mtx	0.0001	9	1.382	0.01819346991107023
spa1.mtx	1e-06	17	2.482	0.0001299689162552027
spa1.mtx	1e-08	24	3.665	1.7097328155016426e-06
spa1.mtx	1e-10	31	4.379	2.2480878778515217e-08
spa2.mtx	0.0001	5	20.290	0.0025987945418217548
spa2.mtx	1e-06	8	32.308	5.141640652906039e-05
spa2.mtx	1e-08	12	49.125	2.7943220304959326e-07
spa2.mtx	1e-10	15	58.892	5.570740791690507e-09
vem1.mtx	0.0001	659	3.181	0.003516704346663496
vem1.mtx	1e-06	1218	4.834	3.5267950764847704e-05
vem1.mtx	1e-08	1778	4.864	3.5174579739478486e-07
vem1.mtx	1e-10	2338	6.646	3.5082423267510406e-09
vem2.mtx	0.0001	965	4.604	0.004970707700261808
vem2.mtx	1e-06	1840	8.577	4.941955175876987e-05
vem2.mtx	1e-08	2714	13.206	4.958371641672979e-07
vem2.mtx	1e-10	3589	15.879	4.94891283351377e-09

Tabella 3.2: Risultati ottenuti dall'esecuzione di Gauss-Seidel su diverse matrici e tolleranze

matrice	**tolleranza**	**iterazioni**	**t. esec. (s)**	**err. rel.**
spa1.mtx	0.0001	143	0.15	0.034612824950369644
spa1.mtx	1e-06	3577	4.168	0.0009680780107418594
spa1.mtx	1e-08	8233	8.312	9.81636688299858e-06
spa1.mtx	1e-10	12919	11.943	9.820388449511405e-08
spa2.mtx	0.0001	161	1.337	0.018141262472769173
spa2.mtx	1e-06	1949	18.786	0.0006694284158053819
spa2.mtx	1e-08	5087	52.149	6.865240653821337e-06
spa2.mtx	1e-10	8285	88.772	6.937814831497985e-08
vem1.mtx	0.0001	890	0.346	0.0027103595777984535
vem1.mtx	1e-06	1612	0.459	2.713397344710513e-05
vem1.mtx	1e-08	2336	0.616	2.6953379930276475e-07
vem1.mtx	1e-10	3058	0.852	2.713166880031993e-09
vem2.mtx	0.0001	1308	0.416	0.0038234939449269052
vem2.mtx	1e-06	2438	1.044	3.791533139667894e-05
vem2.mtx	1e-08	3566	1.129	3.809851776373111e-07
vem2.mtx	1e-10	4696	1.484	3.79876794668187e-09

Tabella 3.3: Risultati ottenuti dall'esecuzione del Gradiente per diverse matrici e tolleranze

matrice	**tolleranza**	**iterazioni**	**t. esec. (s)**	**err. rel.**
spa1.mtx	0.0001	49	0.057	0.020800035393044362
spa1.mtx	1e-06	134	0.259	2.5529093556190602e-05
spa1.mtx	1e-08	177	0.168	1.319840519707949e-07
spa1.mtx	1e-10	200	0.193	1.204618263824277e-09
spa2.mtx	0.0001	42	0.361	0.009822497323405508
spa2.mtx	1e-06	122	1.488	0.00011979846942653868
spa2.mtx	1e-08	196	1.635	5.586660596275471e-07
spa2.mtx	1e-10	240	2.872	5.3242303757412056e-09
vem1.mtx	0.0001	38	0.016	4.0827937374423823e-05
vem1.mtx	1e-06	45	0.016	3.7323397023038456e-07
vem1.mtx	1e-08	53	0.016	2.8318734444310648e-09
vem1.mtx	1e-10	59	0.018	2.191750599624398e-11
vem2.mtx	0.0001	47	0.016	5.7290189553797573e-05
vem2.mtx	1e-06	56	0.024	4.7429962834266885e-07
vem2.mtx	1e-08	66	0.024	4.299983510333221e-09
vem2.mtx	1e-10	74	0.016	2.2476273012012288e-11

Tabella 3.4: Risultati ottenuti dall'esecuzione del Gradiente coniugato per diverse matrici e tolleranze

Viste le premesse appena fatte, come prima operazione preliminare abbiamo calcolato l'indice di sparsità di ogni matrice visibile in tabella 3.

	spa1.mtx	spa2.mtx	vem1.mtx	vem2.mtx
Elementi totali	1000000	9000000	2825761	6765201
Indice di sparsità	0.182434	0.181478	0.004737	0.003137

Tabella 3.5: Indici di sparsità per le matrici prese in esame

*spa*1.*mtx* e *spa*2.*mtx* presentano indici simili (la prima è più sparsa dello 0.5% circa in più della seconda), pertanto ci aspettiamo tempi di esecuzione simili rapportati alle diverse dimensioni delle matrici. *vem*1.*mtx* e *vem*2.*mtx* sono più sparse rispetto le precedenti due matrici. *vem*2.*mtx* è più sparsa di *vem*1.*mtxt* del 33.77% circa pertanto ci aspettiamo un tempo d'esecuzione maggiore a parità di dimensione.

Dopo aver calcolato l'indice di sparsità andiamo a visualizzarlo con lo scopo di trovare ulteriori informazioni che ci potranno aiutare per le successive analisi.

In blu troviamo nella figura 3.1 gli elementi diversi da zero della matrice *spa*1.*mtx*, gli spazi bianchi indicano le entrate uguali a zero. Osservando l'immagine notiamo che non vi è alcun particolare pattern nella sparsità di questa

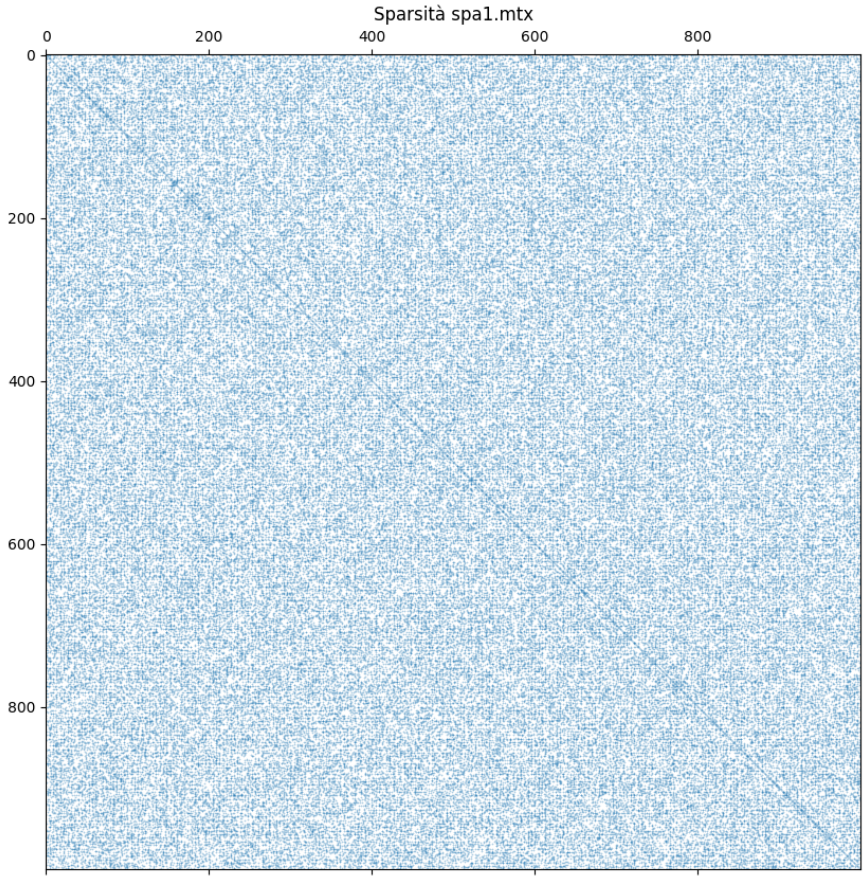

Figura 3.1: sparsità della matrice spa1.mtx

matrice ad eccezione che nella diagonale principale la maggioranza degli elementi non è nulla. Se non sapessimo l'indice di sparsità penseremmo che questa matrice abbia poche entrate uguali a zero. Tale osservazione è dovuta al fatto che la dimensione del marker utilizzato per visionare la matrice nonostante sia molto piccolo, tende a coprire le celle adiacenti e quindi a far sembrare meno sparsa la matrice di quello che è (per visualizzare questo grafico è stata utilizzata la funzione *spy* della libreria *matplotlib.pyplot* con marker = 0.1).

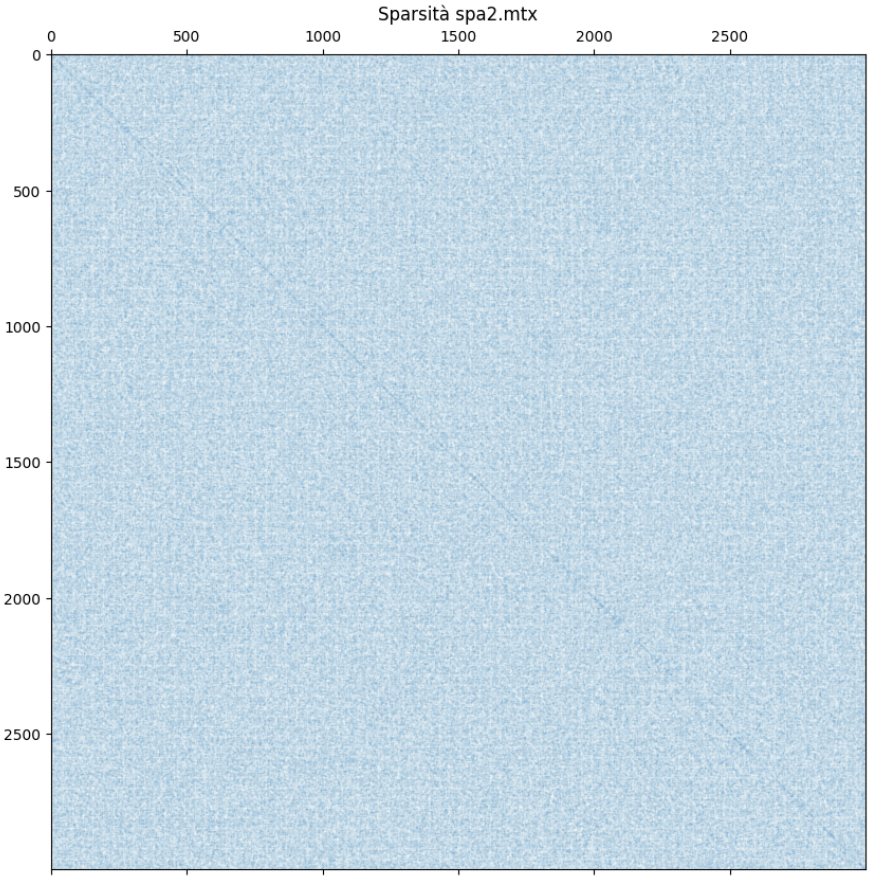

Figura 3.2: sparsità della matrice spa2.mtx

Nonostante sia stato utilizzato un marker ancora più piccolo (marker = 0.01), la matrice *spa*2.*mtx* visibile in figura 3.2 sembra sensibilmente meno sparsa della precedente; quest'osservazione non trova riscontro con l'indice di sparsità. Come nella matrice precedente anche in questa non vi è alcun pattern particolare per quanto riguarda la sparsità, se non che nella diagonale principale la maggioranza delle entrate non è nulla. Il motivo per cui le due matrici sembrano meno sparse di quello che sono è dovuto alle loro dimensioni che rientrano nell'ordine dei milioni.

La matrice *vem*1.*mtx* visibile in figura 3.3 è sensibilmente più sparsa delle precedenti ed è a dominanza diagonale.

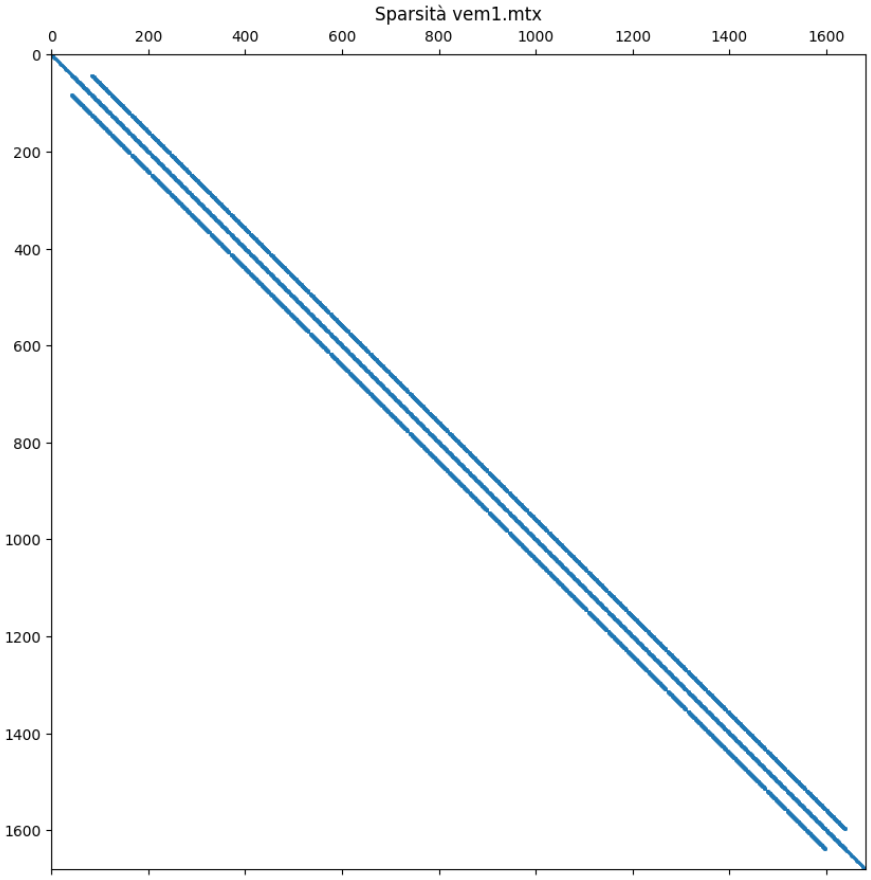

Figura 3.3: sparsità della matrice vem1.mtx

Come la precedente matrice anche *vem*2.*mtx* visibile in figura 3.4 risulta essere sparsa e a dominanza diagonale.

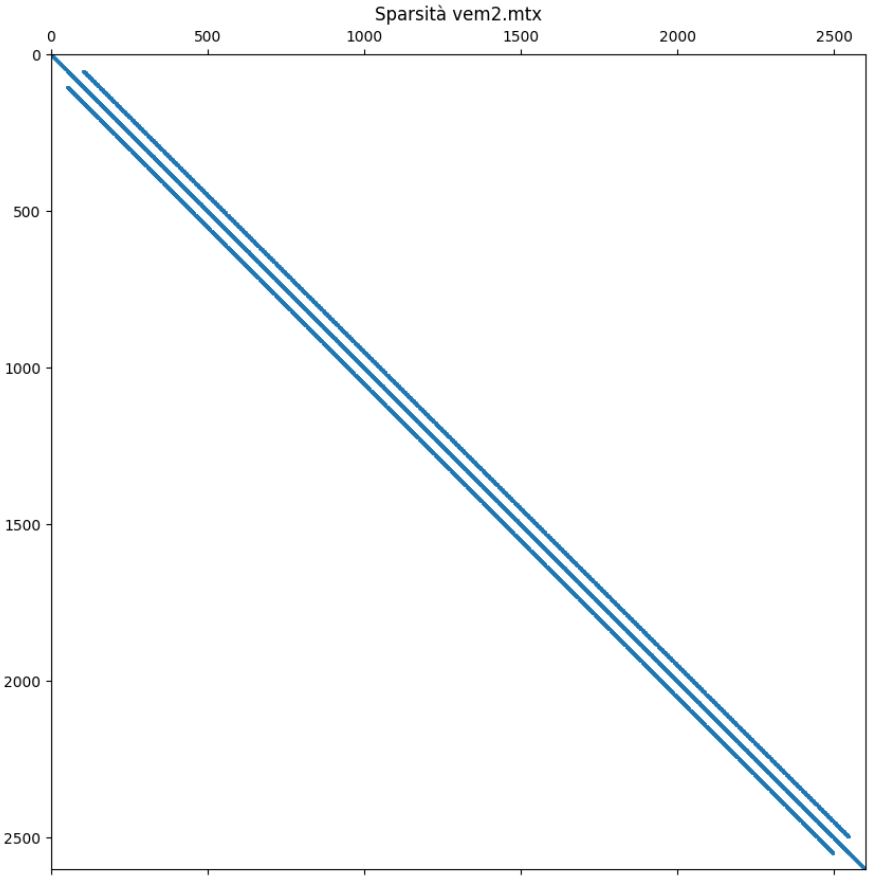

Figura 3.4: sparsità della matrice vem2.mtx

3.1 Analisi del tempo di esecuzione di ogni solutore per ogni tolleranza per ogni matrice

I grafici che verrano mostrati successivamente descrivono quanto tempo impiega ogni solutore a convergere verso la soluzione ottima data una specifica tollerenza su una determinata matrice. Per visualizzare il grafico nel modo più leggibile possibile, le tolleranze sono state rimappate come segue: $10^{-4} \rightarrow 0$, $10^{-6} \rightarrow 1$, $10^{-8} \rightarrow 2$, $10^{-10} \rightarrow 3$ e sull'asse y è stata usata la scala logaritmica. Osservando il grafico relativo alla matrice *spa*1.*mtx* in figura 3.5 notiamo come prevedibile che alla diminuzione delle tolleranza il tempo d'esecuzione di ogni solutore aumenta. Questo aumento di tempo è dovuto dal fatto che se vogliamo una soluzione più vicina a quella ottima dobbiamo iterare maggiormente per avvicinarci e quindi il calcolo computazionale sarà più lungo.

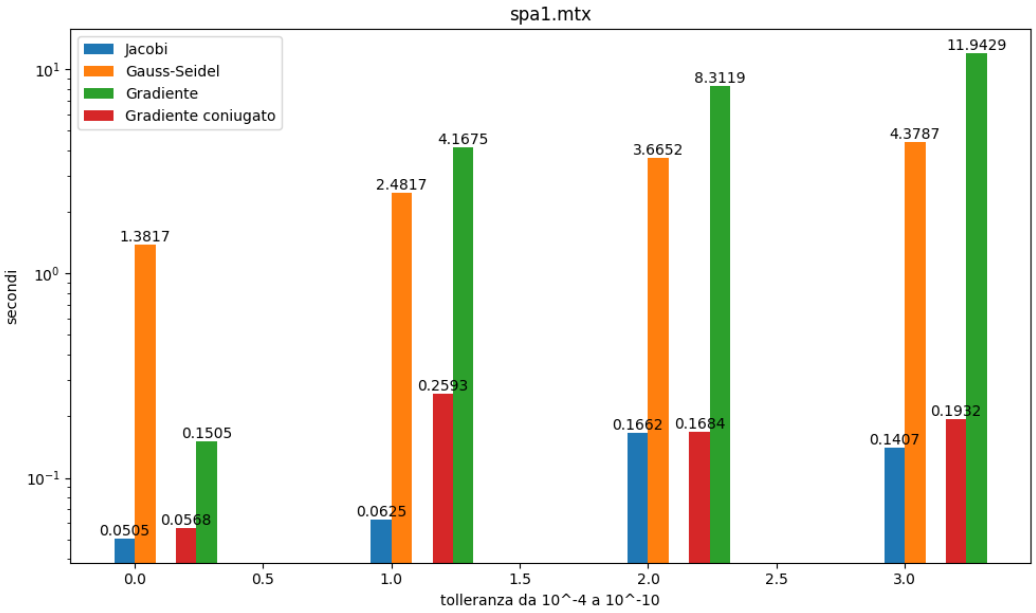

Figura 3.5: andamento del tempo d'esecuzione per ogni risolutore per ogni tolleranza per la matrice spa1.mtx

Per la matrice *spa*2.*mtx* visibile in figura 3.6 i solutori in ordine di velocità per le ultime tre tolleranze sono: Jacobi, Gradiente coniugato, Gauss-Seidel e gradiente. Per la prima tolleranza invece gradiente e Gauss-Seidel si scambiano di posizione. Paragonando i tempi d'esecuzione con la matrice precedentemente analizzata, notiamo che sono maggiori nonostante un indice di sparsità praticamente identico. La motivazione nella differenza dei tempi di esecuzione, trova risposta nel fatto che la prima matrice ha un milione di entrate, invece la seconda ne ha nove. Per la matrice *spa*1.*mtx* visibile in figura 3.7 i solutori in ordine di velocità sono: gradiente coniugato, Jacobi, gradiente e Gauss-Seidel. Possiamo osservare che i tempi d'esecuzione non subiscano cambiamenti significativi man man che si va incontro a tolleranze più stringenti. Da notare inoltre il metodo del gradiente coniugato che fornisce prestazioni di gran lunga maggiori rispetto agli altri solutori. Confrontando i tempi d'esecuzione della matrice presa in esame con le precedenti matrici notiamo che sono nettamente inferiori. Questo è dovuto al fatto che l'indice di sparsità è nettamente inferiore e la matrice è a dominanza diagonale. I solutori applicati alla matrice *vem*2.*mtx* visibile in figura 3.8 hanno comportamenti del tutto simili alla matrice appena discussa con la sola differenza che i tempi d'esecuzione sono leggermente maggiori fatta qualche eccezione non statisticamente significativa. L'aumento dei tempi d'esecuzione è imputabile alla dimensione della matrice che è circa 2.39 volte più grossa di quella precedentemente descritta nonostante, sia il 33% circa più sparsa.

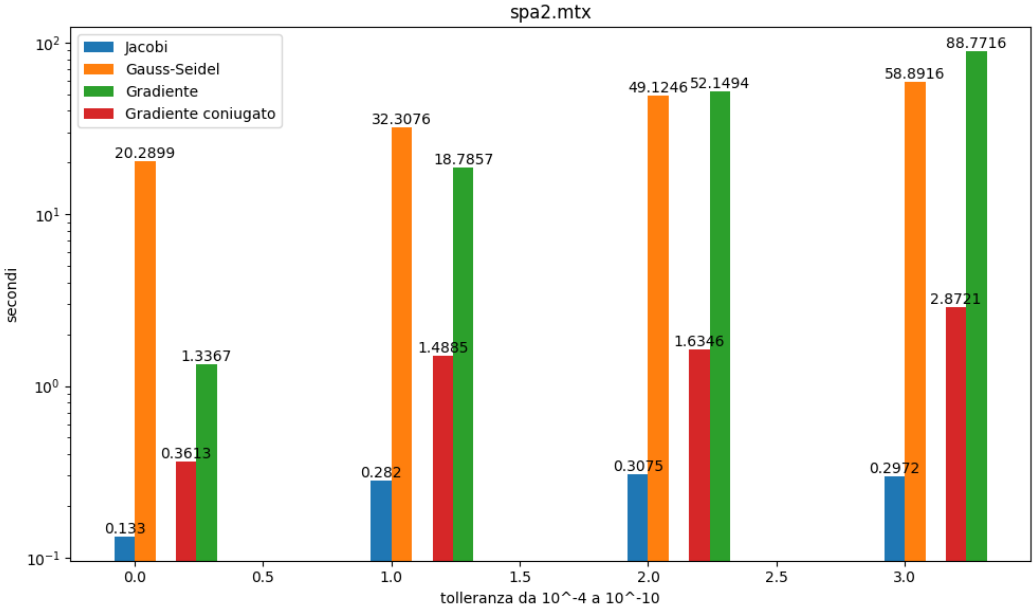

Figura 3.6: andamento del tempo d'esecuzione per ogni risolutore per ogni tolleranza per la matrice spa2.mtx

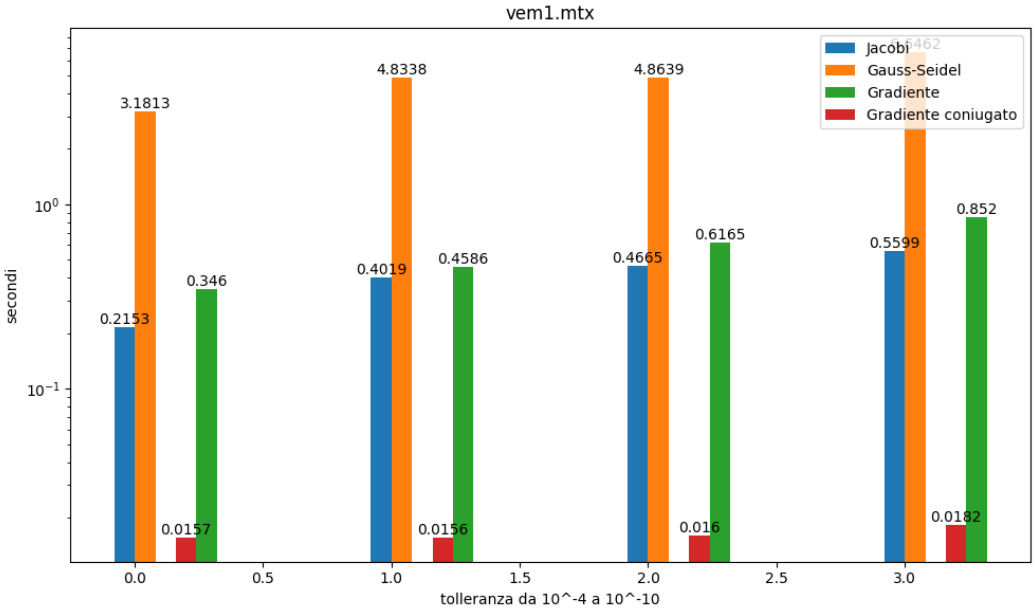

Figura 3.7: andamento del tempo d'esecuzione per ogni risolutore per ogni tolleranza per la matrice vem1.mtx

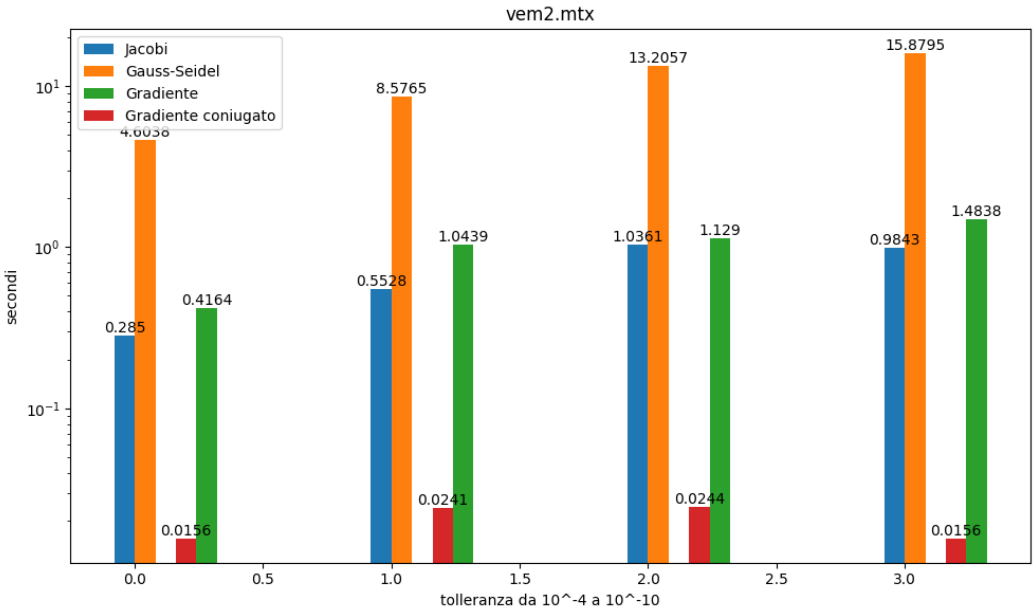

Figura 3.8: andamento del tempo d'esecuzione per ogni risolutore per ogni tolleranza per la matrice vem2.mtx

3.2 Analisi dell'errore relativo di ogni solutore per ogni tolleranza per ogni matrice

I grafici che verrano mostrati successivamente mostrano l'errore relativo all'ultima iterazione di ogni solutore data una specifica tolleranza su una determinata matrice. Le tolleranze sono state rimappate come nei grafici precedenti e l'asse y è anch'esso in scala logaritmica. Generalmente ci aspettiamo che al diminuire della tolleranza diminuisca anche l'errore relativo. Questa diminuzione è dovuta al fatto che se vogliamo una tolleranza più stringente itereremo un numero maggiore di volte e pertanto ci avvicineremo magggiormente alla soluzione ottima.

Osservando il grafico per la matrice *spa*1.*mtx* visibile in figura 3.9 notiamo una generale diminuizione dell'errore relativo al diminuire della tolleranza. Gradiente e Gauss-Seidel commettono un errore relativo maggiore rispetto a Jacobi e Gradiente coniugato. Al diminuire della tolleranza il Gradiente coniugato migliora sensibilmente il suo errore relativo.

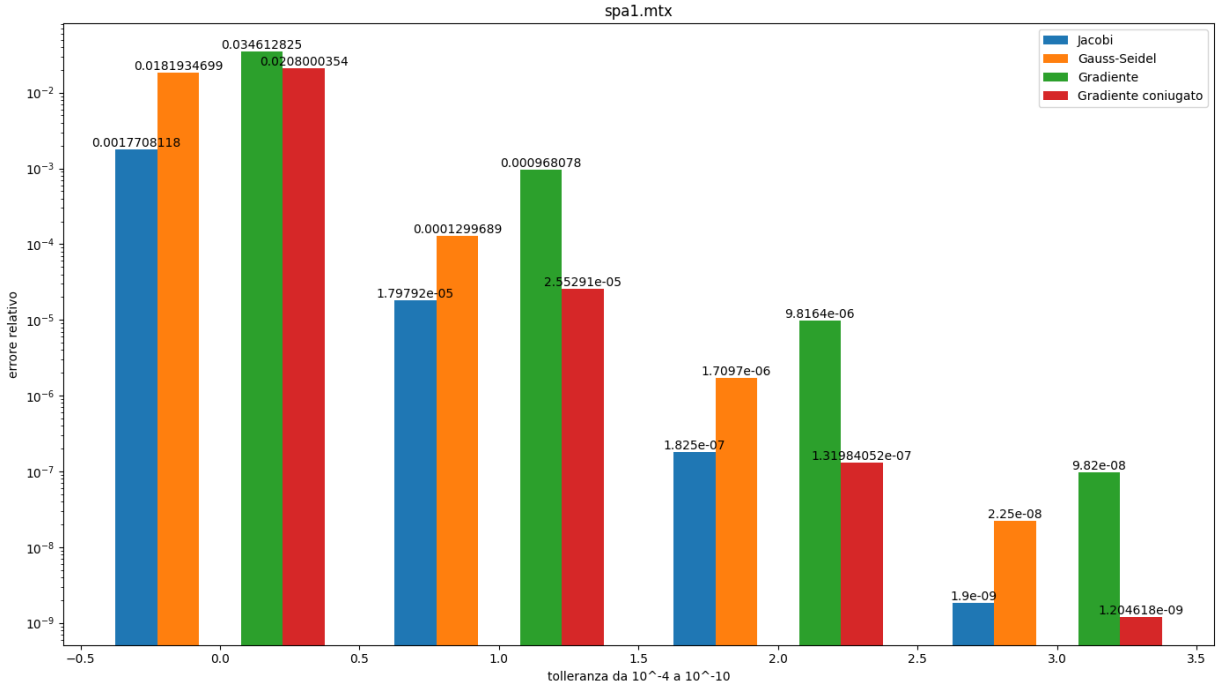

Figura 3.9: andamento dell'errore relativo per ogni risolutore per ogni tolleranza per la matrice spa1.mtx

Osservando il grafico per la matrice *spa*2.*mtx* visibile in figura 3.10 notiamo una generale diminuizione dell'errore relativo al diminuire della tolleranza. I due gradienti commettono un errore relativo maggiore rispetto a Jacobi e Gauss-Seidel. Al diminuire della tolleranza Jacobi è il solutore con il minor errore relativo.

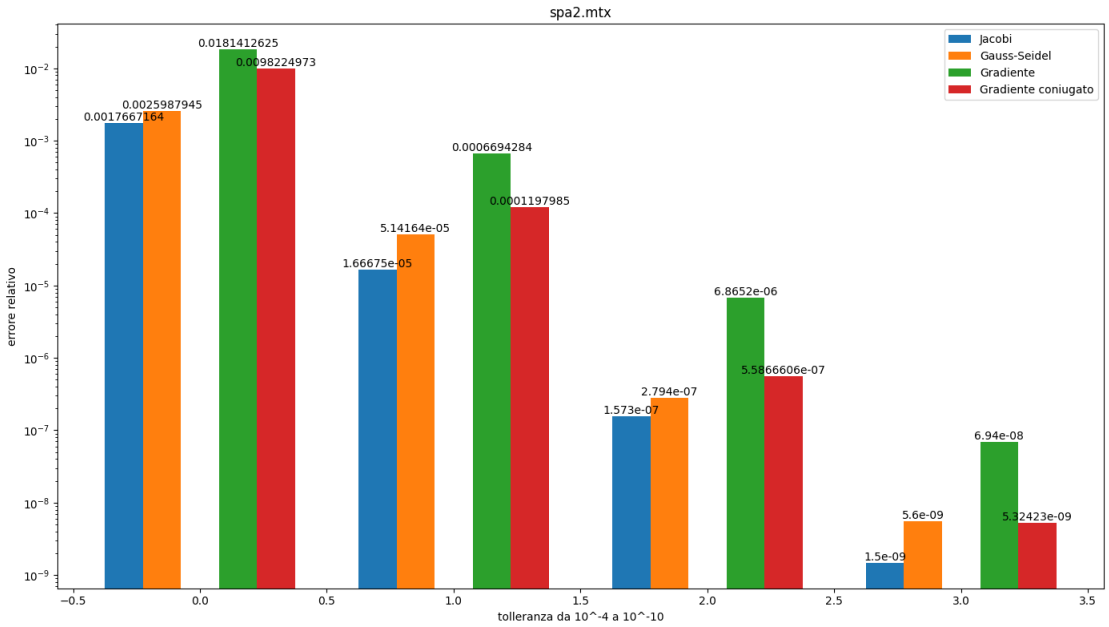

Figura 3.10: andamento dell'errore relativo per ogni risolutore per ogni tolleranza per la matrice spa2.mtx

Osservando il grafico relativo alla matrice *vem*1.*mtx* visibile in figura 3.11 notiamo che gli errori relativi tendono a diminuire al diminuire della tolleranza. Confrontando i vari errori relativi notiamo che non hanno differenze significative fatta eccezione per il Gradiente coniugato. Possiamo notare come il suo valore sia simile ai valori degli altri solutori eseguiti su una tolleranza inferiore di un ordine di grandezza.

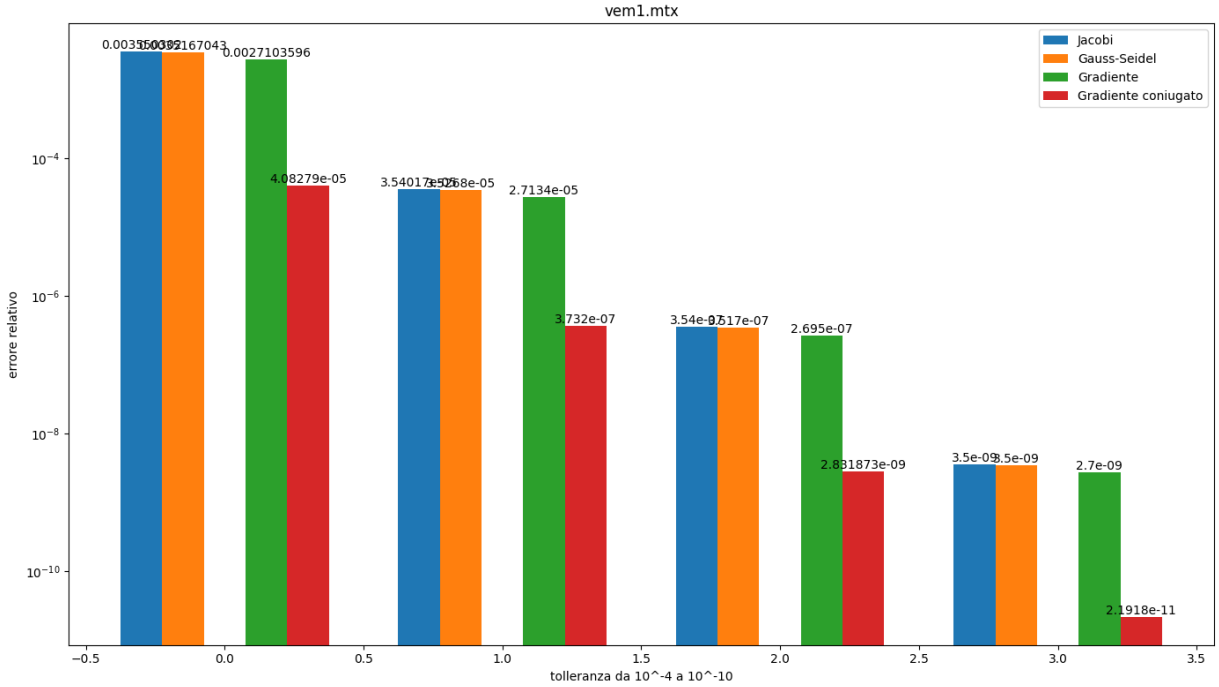

Figura 3.11: andamento dell'errore relativo per ogni risolutore per ogni tolleranza per la matrice vem1.mtx

Osservando il grafico relativo alla matrice *vem*2.*mtx* visibile in figura 3.12 notiamo che il comportamente è del tutto identico a quello della matrice *vem*1.*mtx*.

Generalmente quanato osservato per tutte le matrici è coerente con le considerazioni fatte all'inizio di questa sezione. Osservando tutti i grafici insieme notiamo che le prime due matrici hanno degli errori maggiori rispetto alle ultime due. Questo errore potrebbe essere spiegato dato dal fatto che le prime due matrici hanno un indice di sparsità maggiore rispetto a quello delle ultime due e dalla loro differente distribuzione.

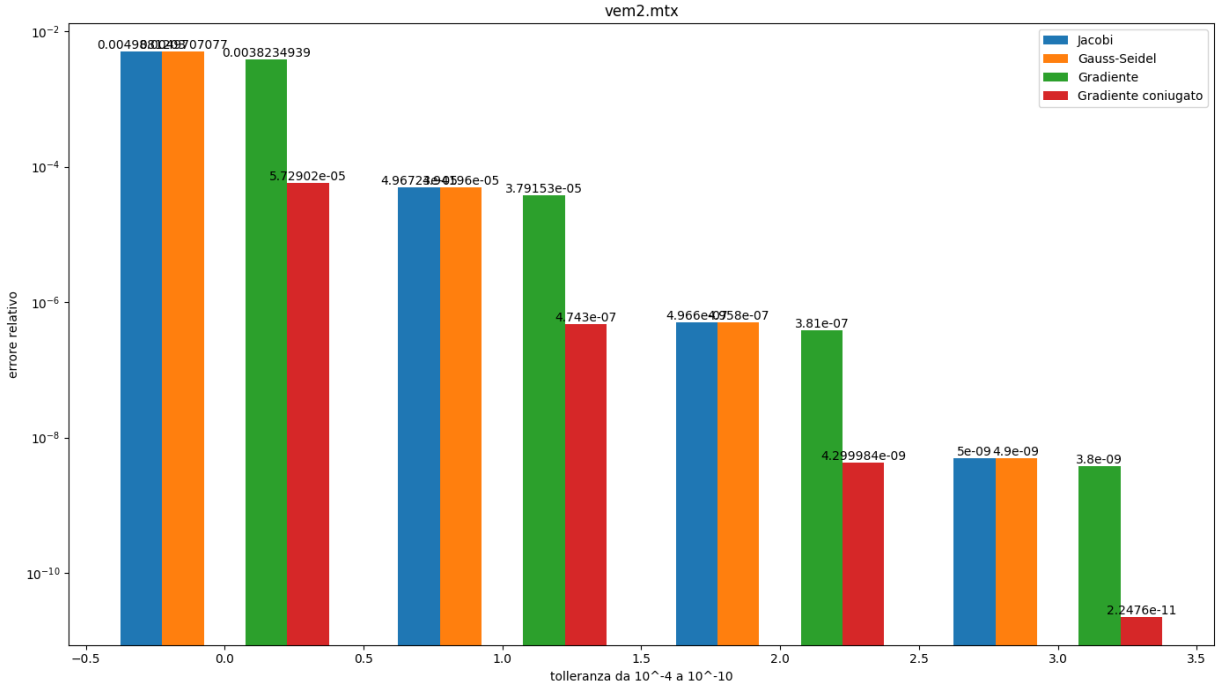

Figura 3.12: andamento dell'errore relativo per ogni risolutore per ogni tolleranza per la matrice vem2.mtx

3.3 Analisi dell'errore relativo di ogni solutore per ogni iterazione per ogni tolleranza per ogni matrice

I grafici che verranno mostrati successivamente fanno vedere l'andamento dell'errore relativo per ogni iterazione per uno specifico solutore su una specifica matrice con determinata tolleranza (per questioni di brevità verranno mostrati i grafici aventi tolleranza 10^{-10}; i grafici aventi tolleranze superiori non venogno mostrati in quanto sono inglobati in quelli con tolleranza minima, volendo si possono visionare sul jupyter notebook in allegato).
Da quanto è emerso nei grafici precedenti ci aspettiamo che si verifichi una diminuzione dell'errore relativo al crescere delle iterazioni.

- Jacobi
  - *spa*1.*mtx*: Osservando il grafico relativo alla matrice *spa*1.*mtx* visibile in figura 3.13 notiamo che fino all'iterazione 45 (guardando il grafico con tolleranza maggiore sul notebook si riesce a vedere meglio) l'erro-

re relativo decresce a zig-zag. Dall'iterazione 46 fino all'ultima invece abbiamo una diminuzione costante dell'errore relativo.

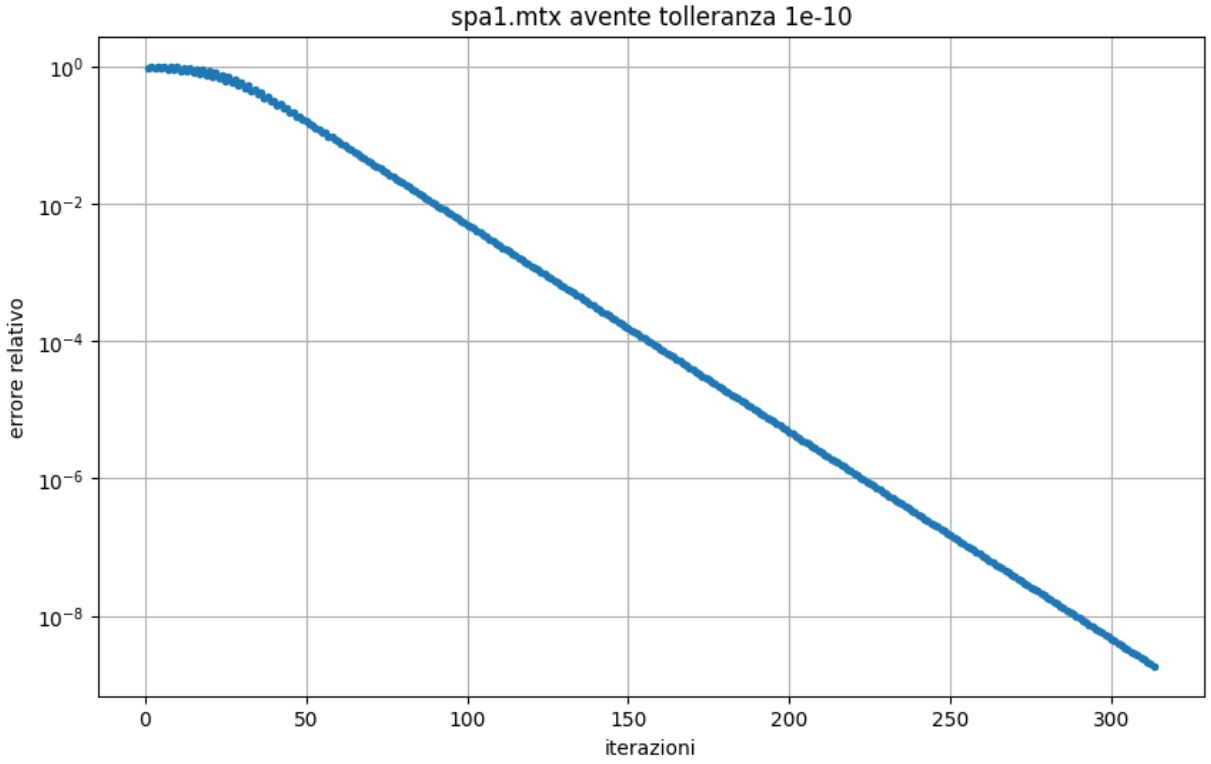

Figura 3.13: andamento dell'errore relativo per ogni iterazione con Jacobi sulla matrice spa1.mtx

– *spa*2.*mtx*: Osservando il grafico relativo alla matrice *spa*2.*mtx* visibile in figura 3.14 notiamo che fino all'iterazione 14 (guardando il grafico con tolleranza maggiore sul notebook si riesce a vedere meglio) l'errore relativo decresce a zig-zag. Dall'iterazione 15 fino all'ultima invece abbiamo una diminuzione costante dell'errore relativo.

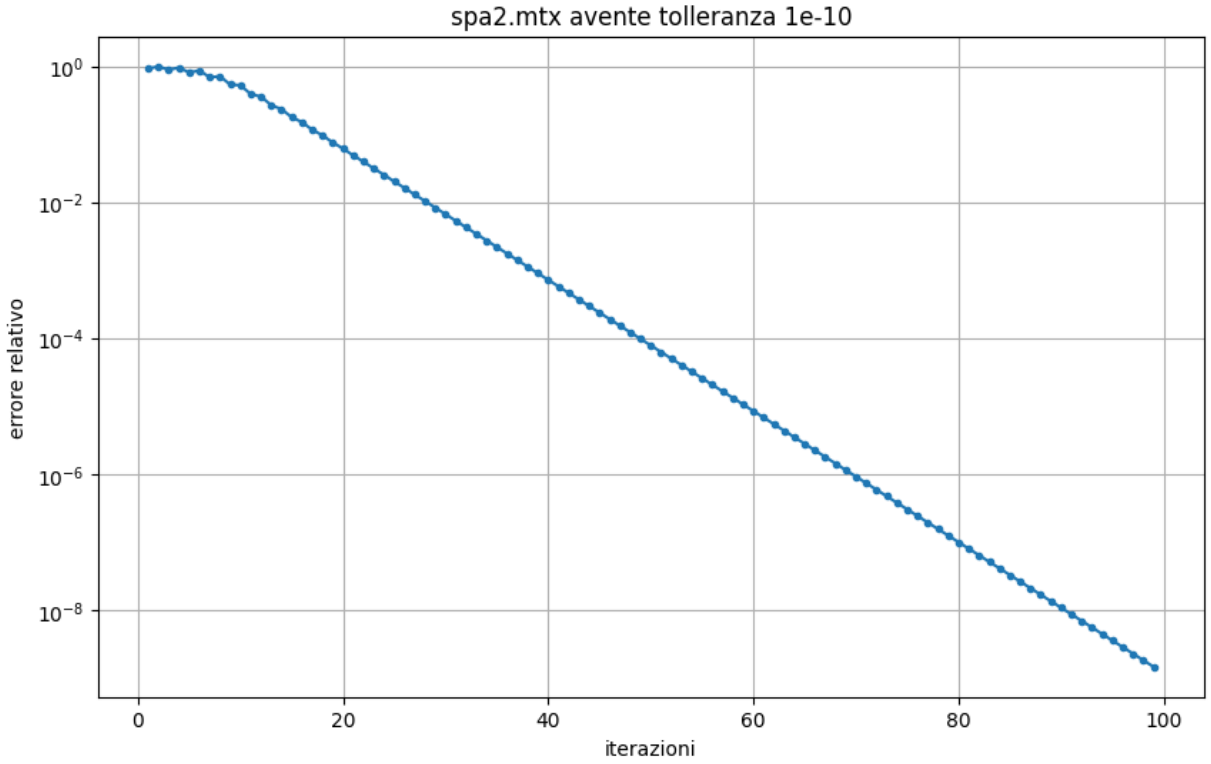

Figura 3.14: andamento dell'errore relativo per ogni iterazione con Jacobi sulla matrice spa2.mtx

– *vem*1.*mtx*: Osservando il grafico relativo alla matrice *vem*1.*mtx* visibile in figura 3.15 notiamo che fino all'iterazione 120 circa (guardando il grafico con tolleranza maggiore sul notebook si riesce a vedere meglio) l'errore relativo decresce a maggiormente che nelle iterazioni successive. Dalle successive iterazioni fino all'ultima invece abbiamo una diminuzione costante dell'errore relativo.

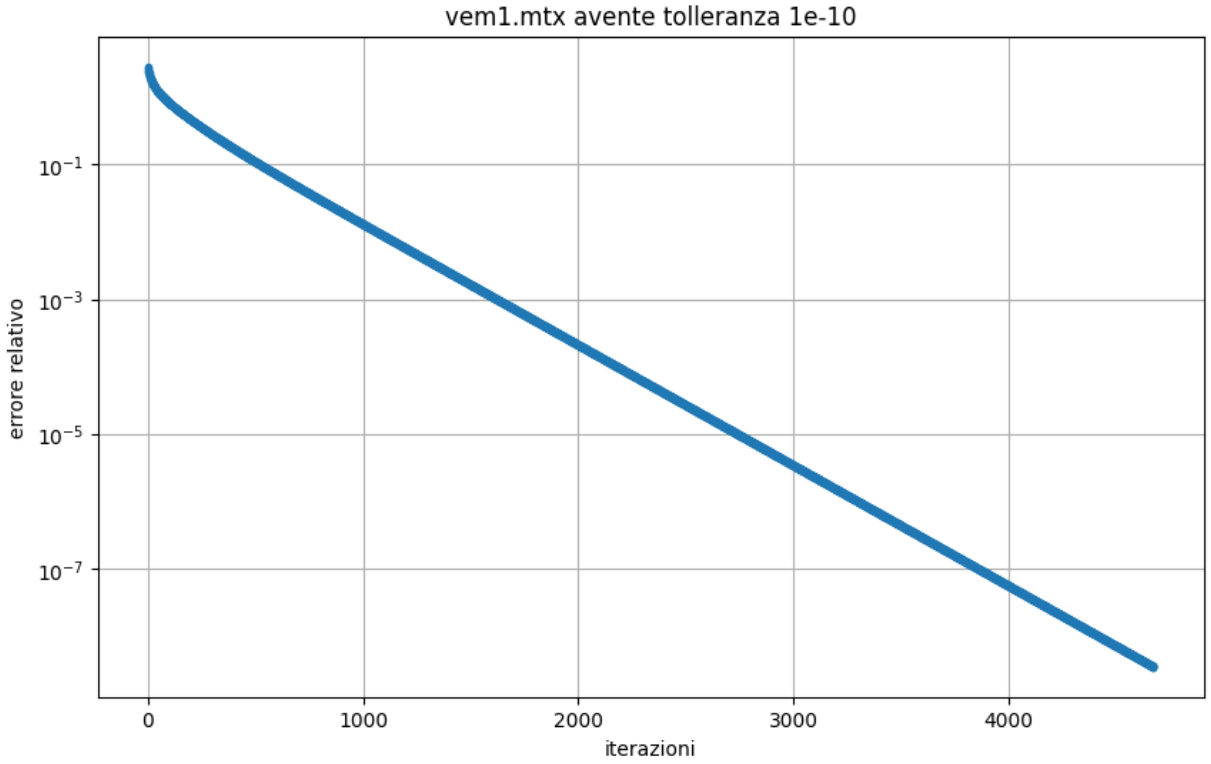

Figura 3.15: andamento dell'errore relativo per ogni iterazione con Jacobi sulla matrice vem1.mtx

– *vem2.mtx* Osservando il grafico relativo alla matrice *vem2.mtx* visibile in figura 3.16 notiamo che fino all'iterazione 125 circa (guardando il grafico con tolleranza maggiore sul notebook si riesce a vedere meglio) l'errore relativo decresce a maggiormente che nelle iterazioni successive. Dalle successive iterazioni fino all'ultima invece abbiamo una diminuzione costante dell'errore relativo.

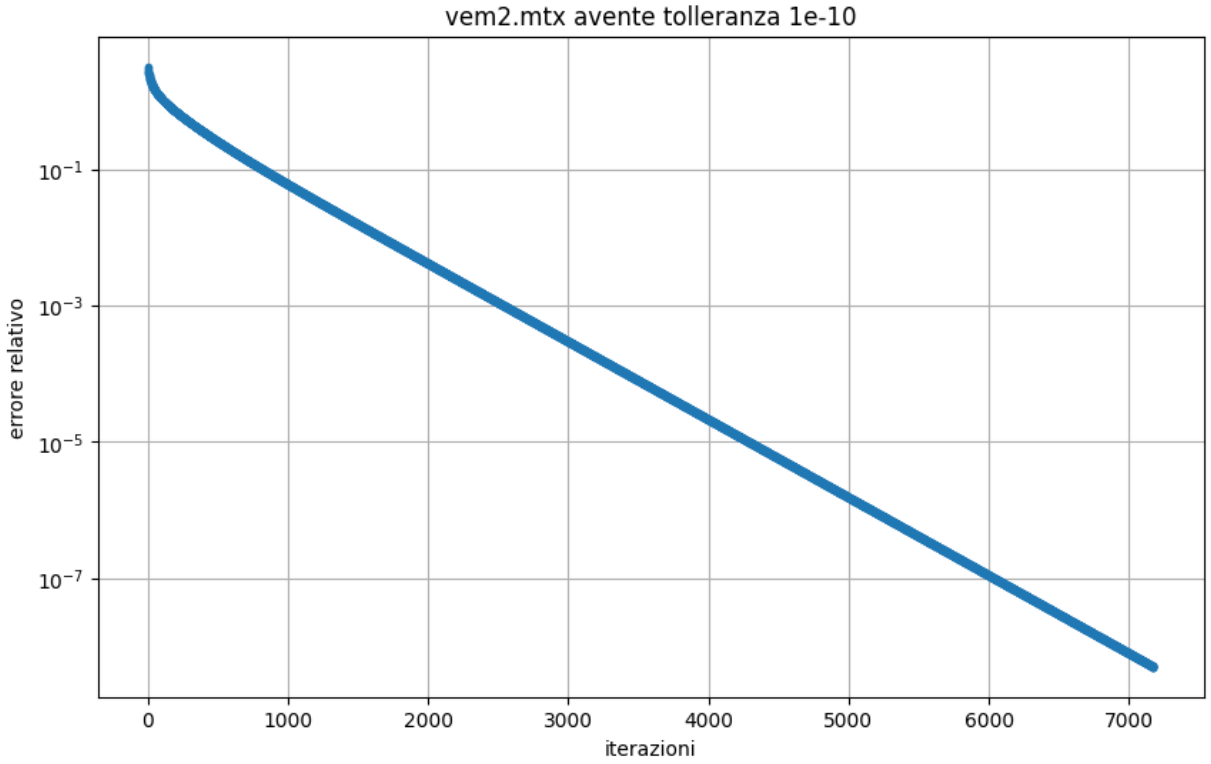

Figura 3.16: andamento dell'errore relativo per ogni iterazione con Jacobi sulla matrice vem2.mtx

- Gauss-Seidel
  - *spa*1.*mtx*: Osservando il grafico relativo alla matrice *spa*1.*mtx* visibile in figura 3.17 notiamo che fino all'iterazione 5 (guardando il grafico con tolleranza maggiore sul notebook si riesce a vedere meglio) l'errore relativo decresce a in maniera meno evidente che nelle iterazioni successive. Dalle successive iterazioni fino all'ultima invece abbiamo una diminuzione costante dell'errore relativo.

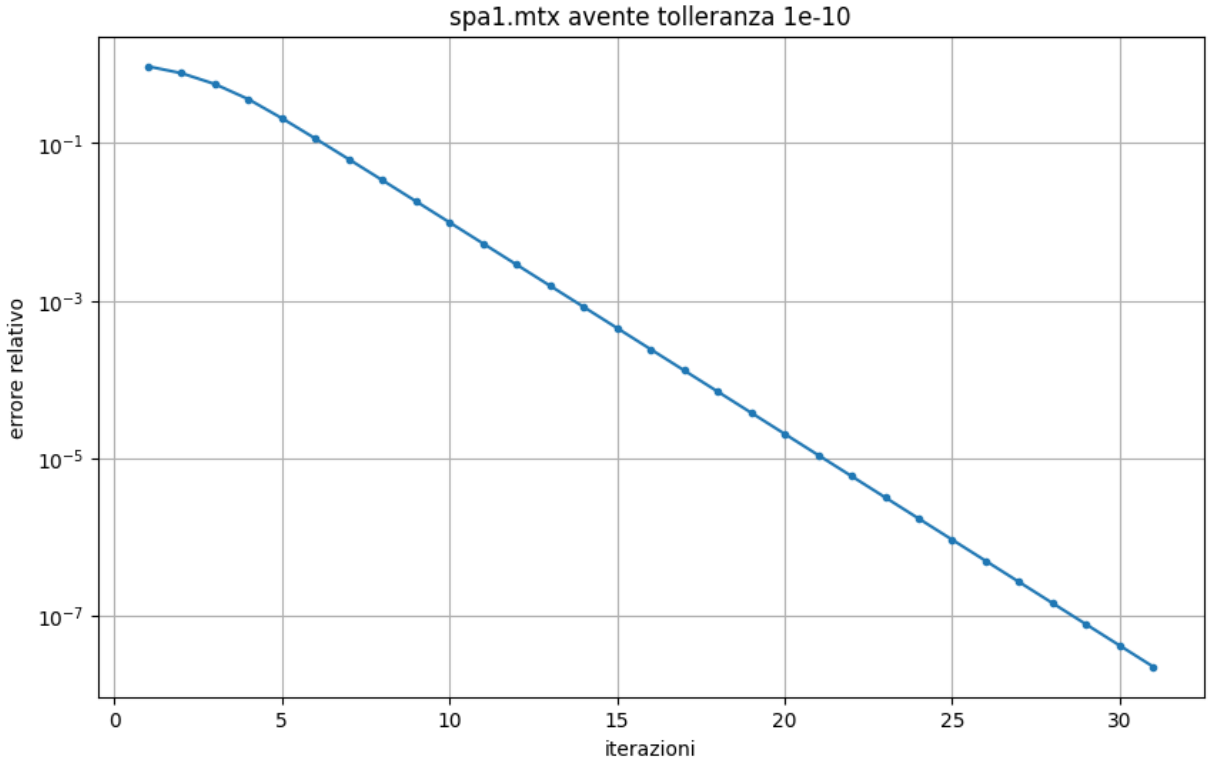

Figura 3.17: andamento dell'errore relativo per ogni iterazione con Gauss-Seidel sulla matrice spa1.mtx

- *spa*2.*mtx*: Osservando il grafico relativo alla matrice *spa*2.*mtx* visibile in figura 3.18 notiamo che vi è una diminuzione costante dell'errore relativo per tutte le iterazioni.
- *vem*1.*mtx*: Osservando il grafico relativo alla matrice *vem*1.*mtx* visibile in figura 3.19 notiamo che fino all'iterazione 100 circa (guardando il grafico con tolleranza maggiore sul notebook si riesce a vedere meglio) l'errore relativo decresce maggiormente che nelle iterazioni successive. Dalle successive iterazioni fino all'ultima invece abbiamo una diminuzione costante dell'errore relativo.

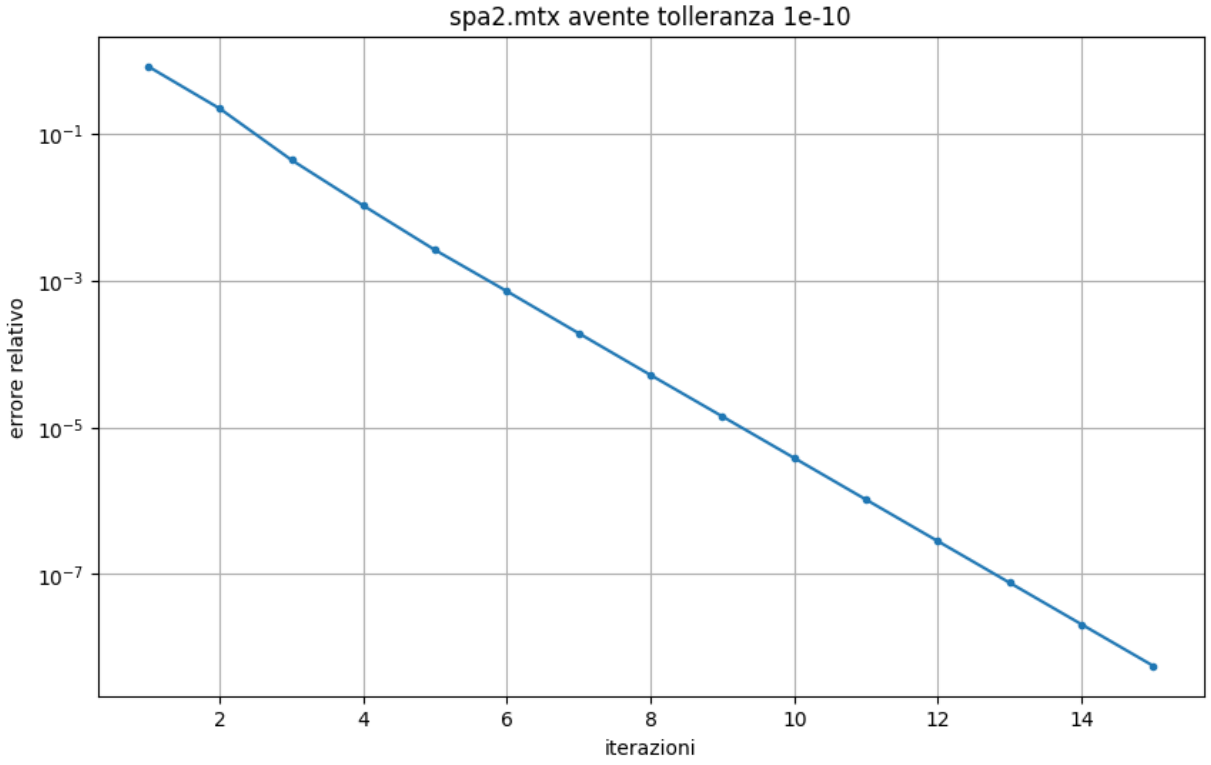

Figura 3.18: andamento dell'errore relativo per ogni iterazione con Gauss-Seidel sulla matrice spa2.mtx

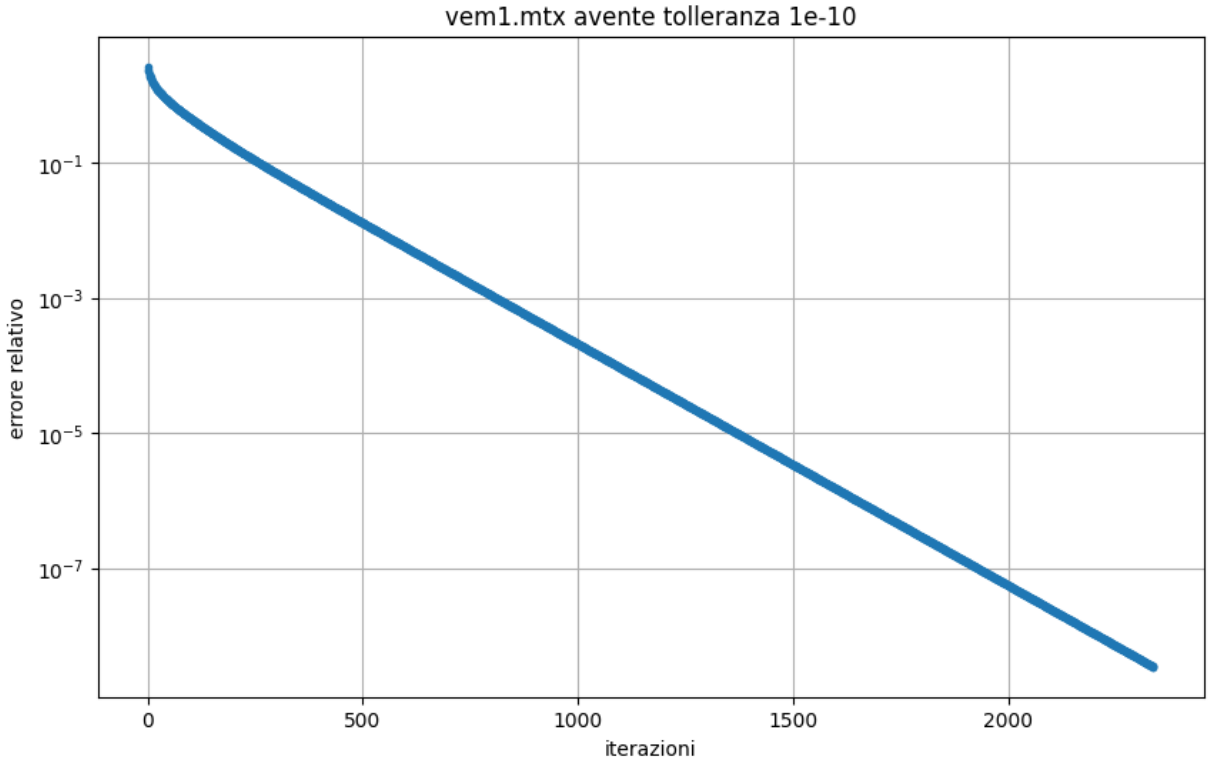

Figura 3.19: andamento dell'errore relativo per ogni iterazione con Gauss-Seidel sulla matrice vem1.mtx

– *vem*2.*mtx*: Il comportamente di *vem*2 è analogo a quello di *vem*1.

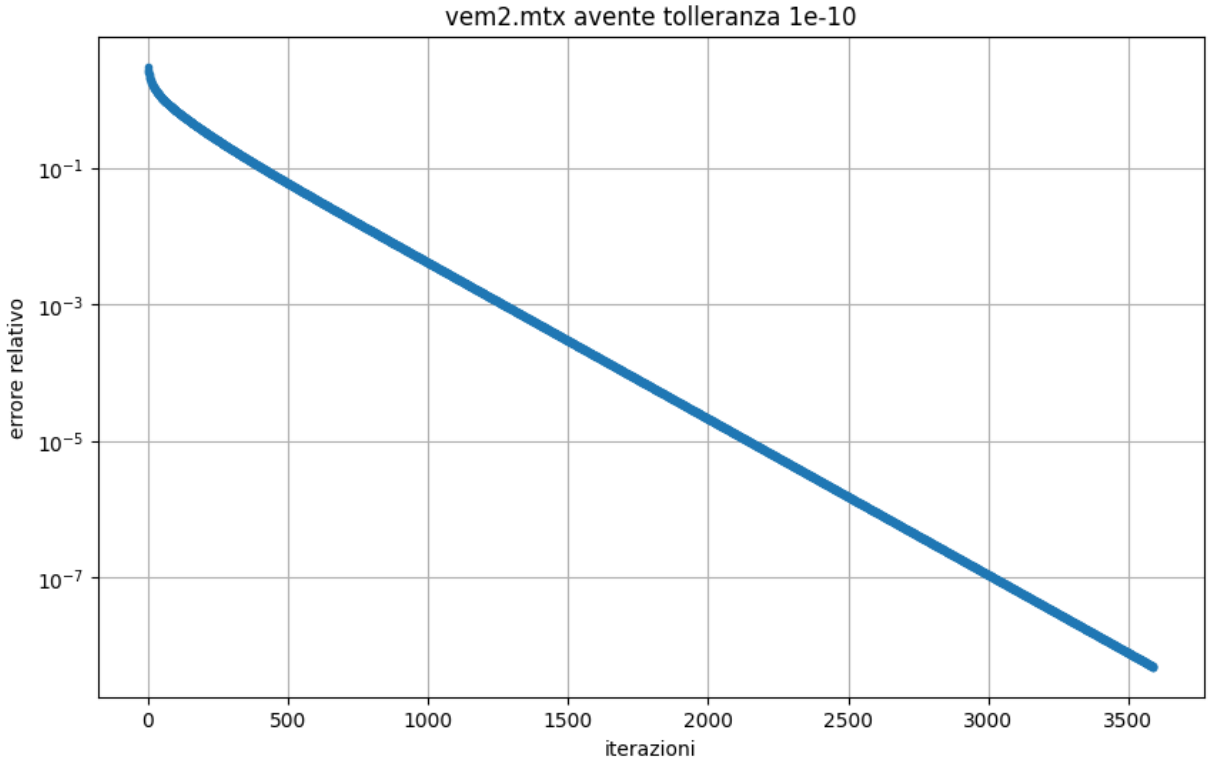

Figura 3.20: andamento dell'errore relativo per ogni iterazione con Gauss-Seidel sulla matrice vem2.mtx

- Gradiente
  - *spa*1.*mtx*: Osservando il grafico relativo alla matrice *spa*1.*mtx* visibile in figura 3.21 notiamo che fino all'iterazione 20 (guardando il grafico con tolleranza maggiore sul notebook si riesce a vedere meglio) l'errore relativo decresce notevolmente. Dalla 21 alla 40 questa decrescita rallenta. Dalle successive iterazioni fino all'ultima invece abbiamo una diminuzione costante dell'errore relativo.

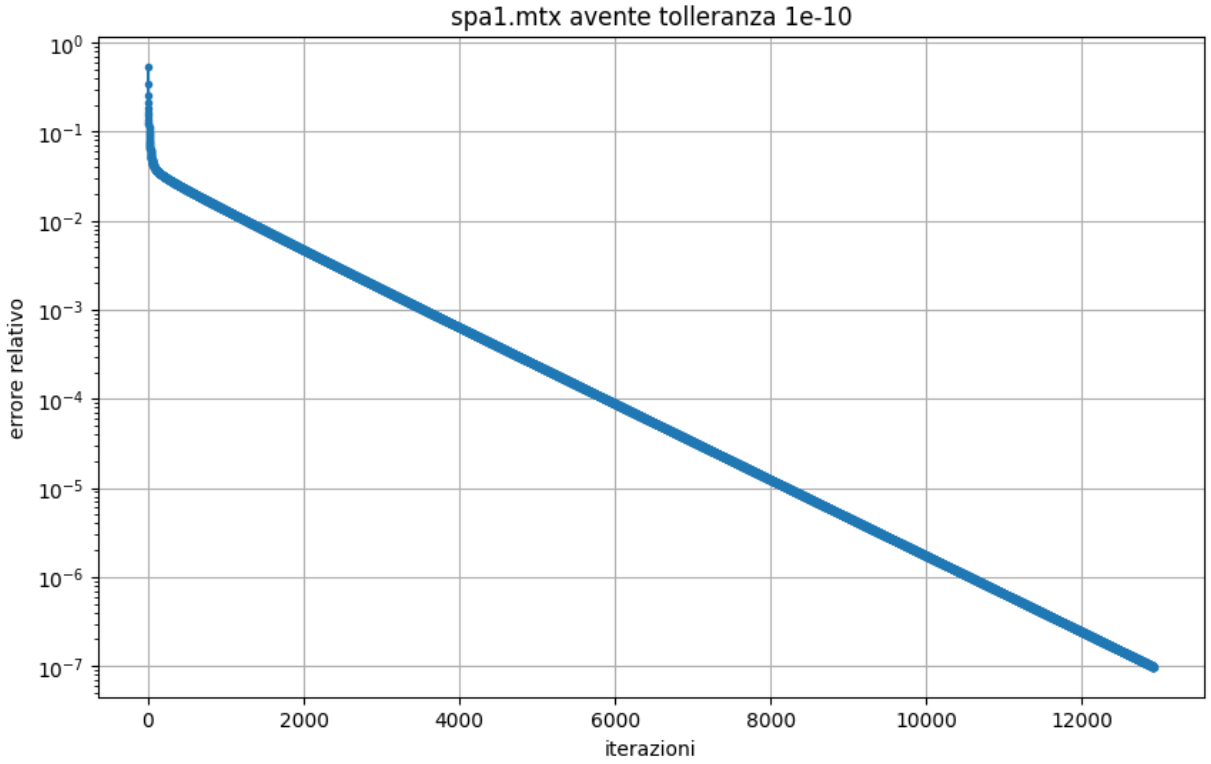

Figura 3.21: andamento dell'errore relativo per ogni iterazione con il Gradiente sulla matrice spa1.mtx

– *spa*2.*mtx*: Osservando il grafico relativo alla matrice *spa*2.*mtx* visibile in figura 3.22 notiamo che il suo comportamento è del tutto analogo a quello di *spa*1.*mtx*

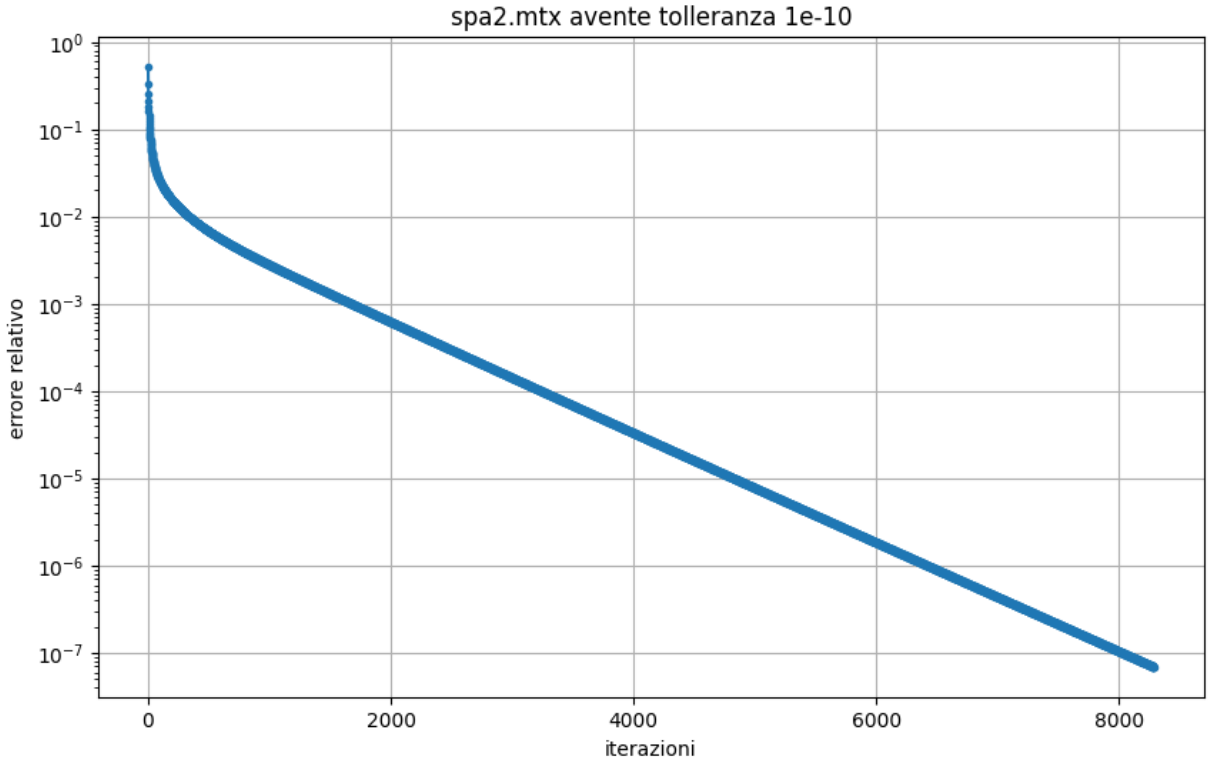

Figura 3.22: andamento dell'errore relativo per ogni iterazione con il Gradiente sulla matrice spa2.mtx

– *vem*1.*mtx*: Osservando il grafico relativo alla matrice *vem*1.*mtx* visibile in figura 3.23 notiamo che l'errore relativo decresce in maniera più marcata fino all'iterazione 50. Dall'iterazione successiva invece vi è una decrescita costante.

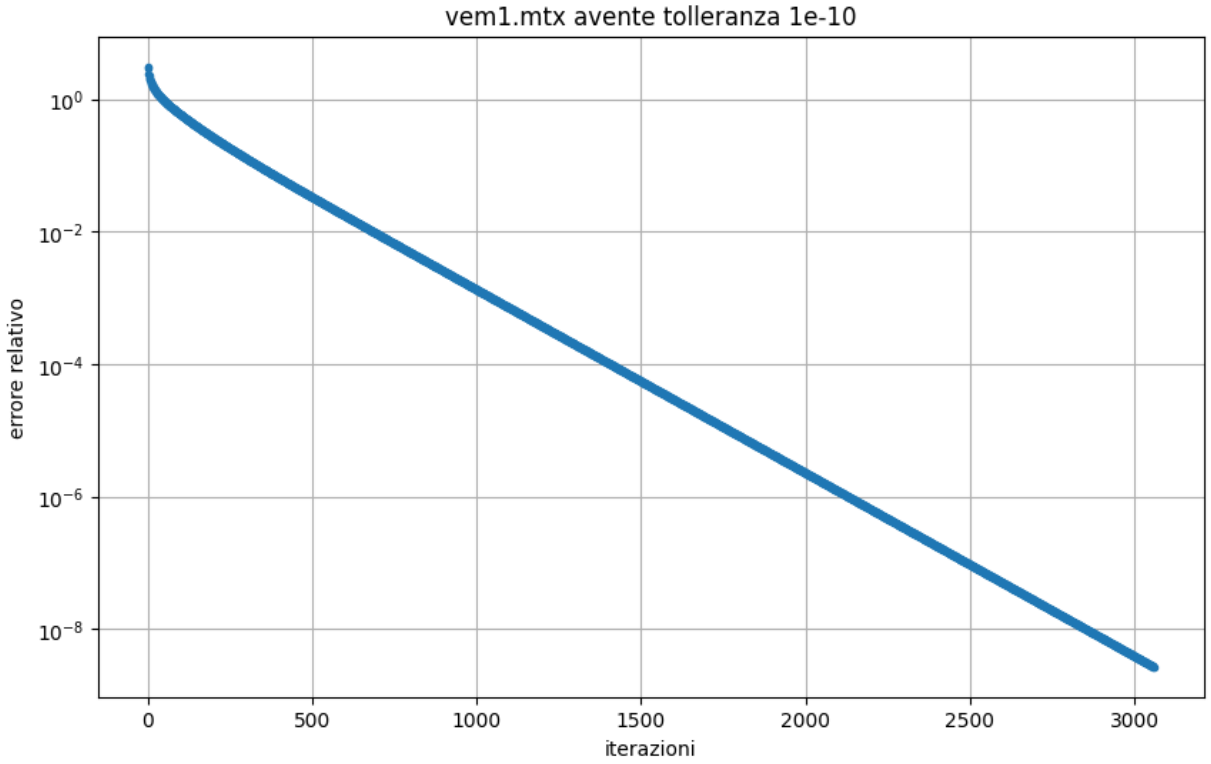

Figura 3.23: andamento dell'errore relativo per ogni iterazione con il Gradiente sulla matrice vem1.mtx

– *vem2.mtx*: Osservando il grafico relativo alla matrice *vem2.mtx* visibile in figura 3.24 notiamo che l'errore relativo decresce in maniera più marcata fino all'iterazione 100. Dall'iterazione successiva invece vi è una decrescita costante.

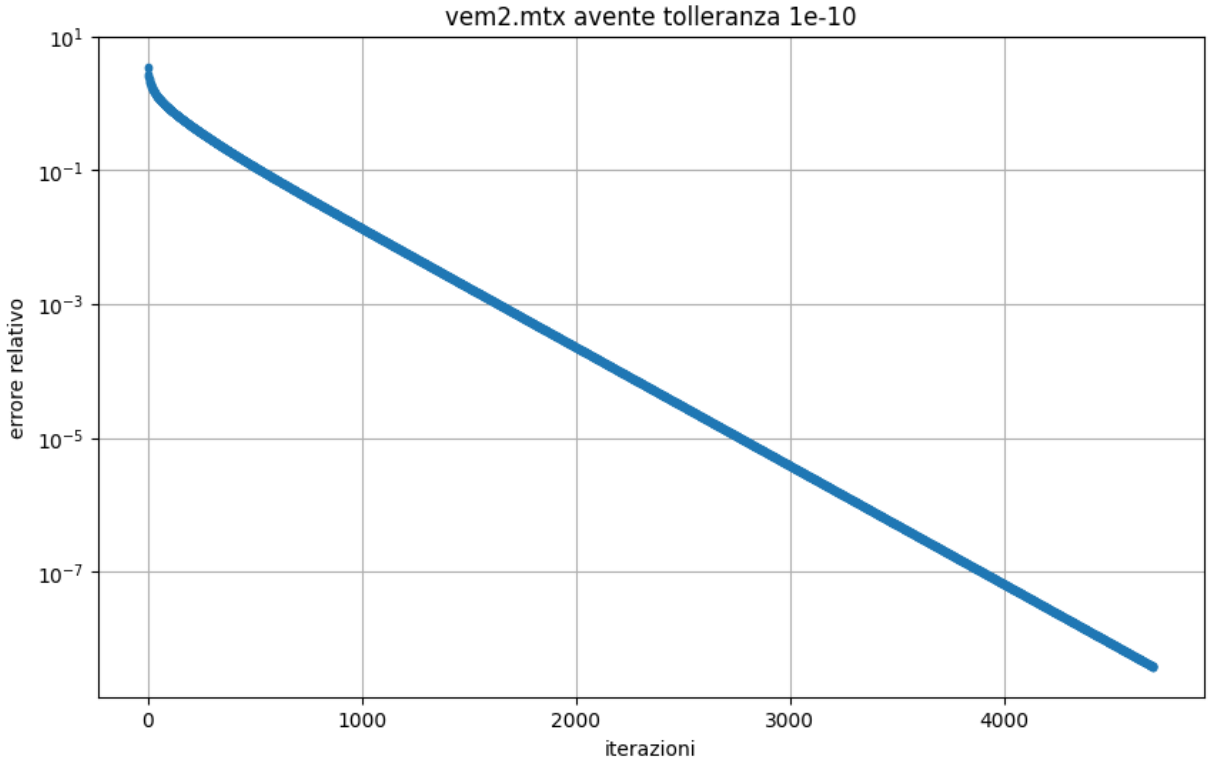

Figura 3.24: andamento dell'errore relativo per ogni iterazione con il Gradiente sulla matrice vem2.mtx

- Gradiente coniugato
  - *spa*1.*mtx*: Osservando il grafico relativo alla matrice *spa*1.*mtx* visibile in figura 3.25 notiamo che l'errore relativo decresce in maniera netta fino all'iterazione 15, dalla 16 alla 110 circa decresce abbastanza stabilmente, dalla 111 alla 130 vi è un vertiginoso calo dell'errore relativo, dalla 131 alla 160 circa non vi è una diminuzione. Successivamente vi è un nuovo calo repintino dell'errore .
  - *spa*2.*mtx*: Osservando il grafico relativo alla matrice *spa*2.*mtx* visibile in figura 3.26 notiamo che l'errore relativo tende a decrescere al crescere delle iterazioni senza nessun significativo stallo o decrescita improvvisa.

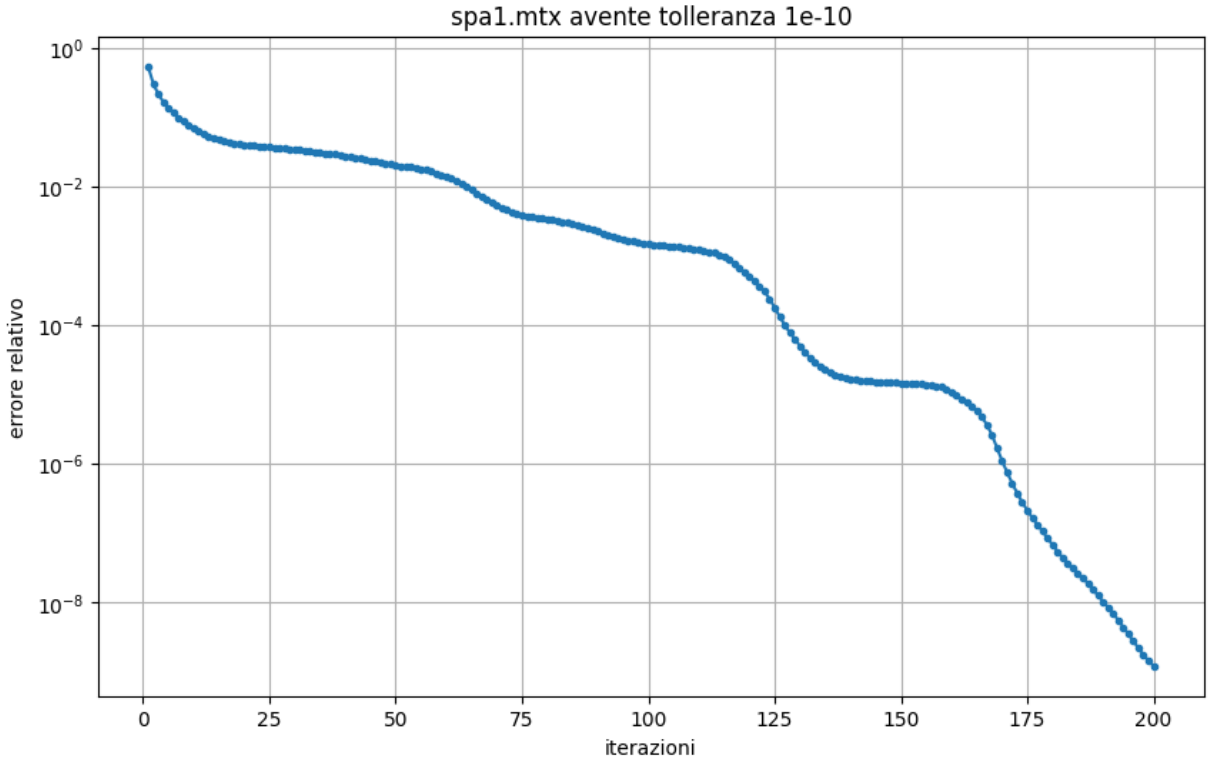

Figura 3.25: andamento dell'errore relativo per ogni iterazione con il Gradiente sulla matrice spa1.mtx

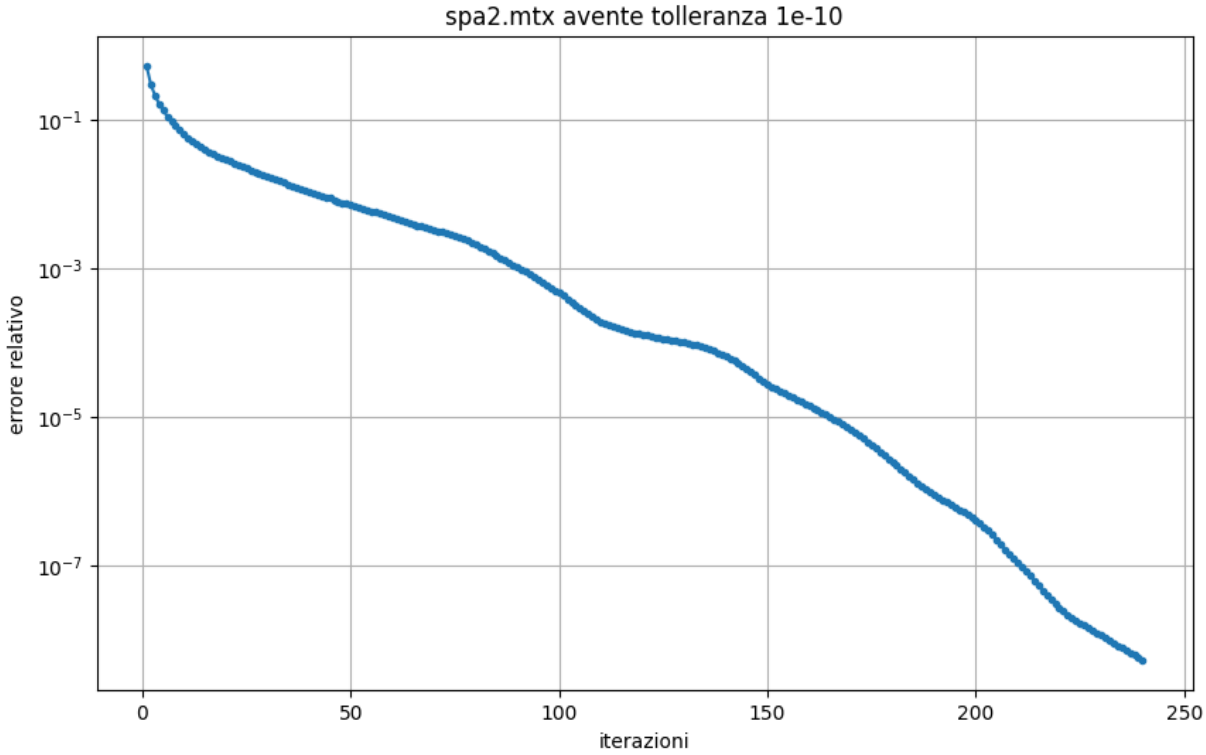

Figura 3.26: andamento dell'errore relativo per ogni iterazione con il Gradiente sulla matrice spa2.mtx

- *vem*1.*mtx*: Osservando il grafico relativo alla matrice *vem*1.*mtx* visibile in figura 3.27 notiamo che l'errore relativo tende a decrescere al crescere delle iterazioni molto lentamente fino all'iterazione 19. Dalla 20 in poi vi è una decrescita maggiore fino alla fine delle iterazioni.
- *vem*2.*mtx*: Osservando il grafico relativo alla matrice *vem*2.*mtx* visibile in figura 3.28 notiamo un comportamnte analogo a quello di *vem*1.*mtx* con la sola differenza che la decrescita più vertiginosa incomincia dall'iterazione 24.

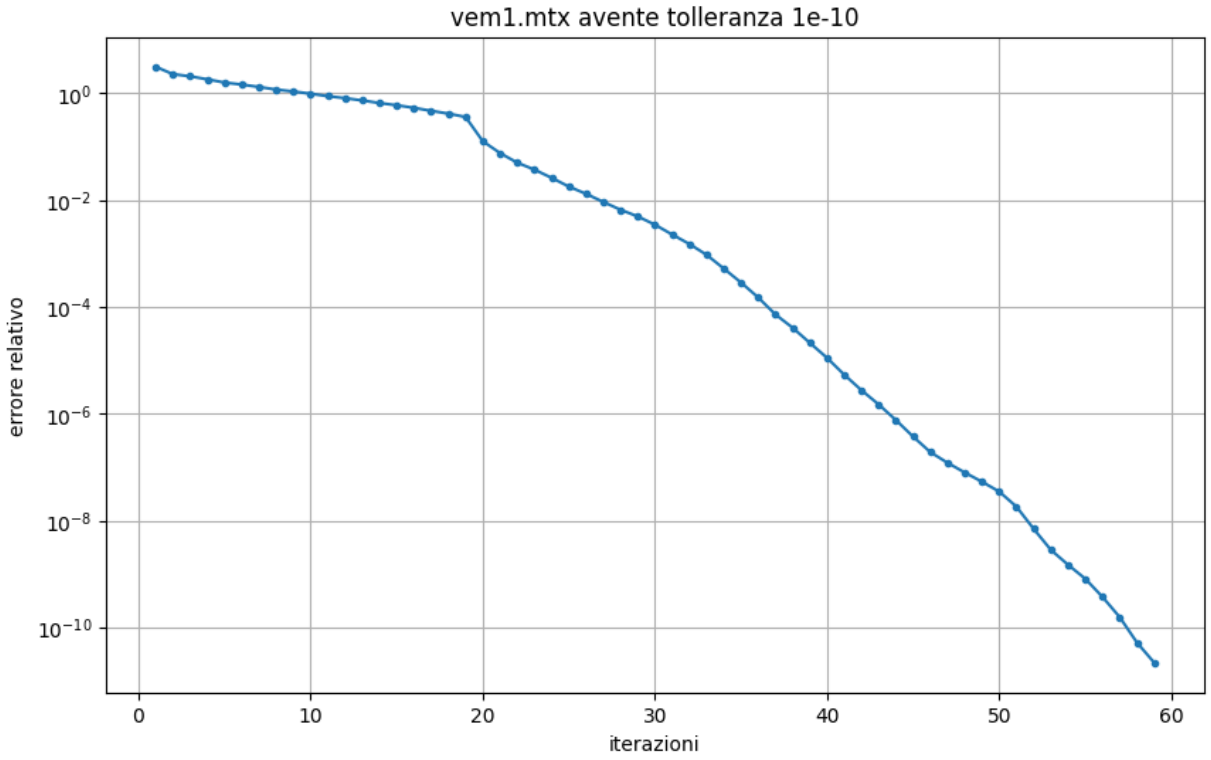

Figura 3.27: andamento dell'errore relativo per ogni iterazione con il Gradiente sulla matrice vem1.mtx

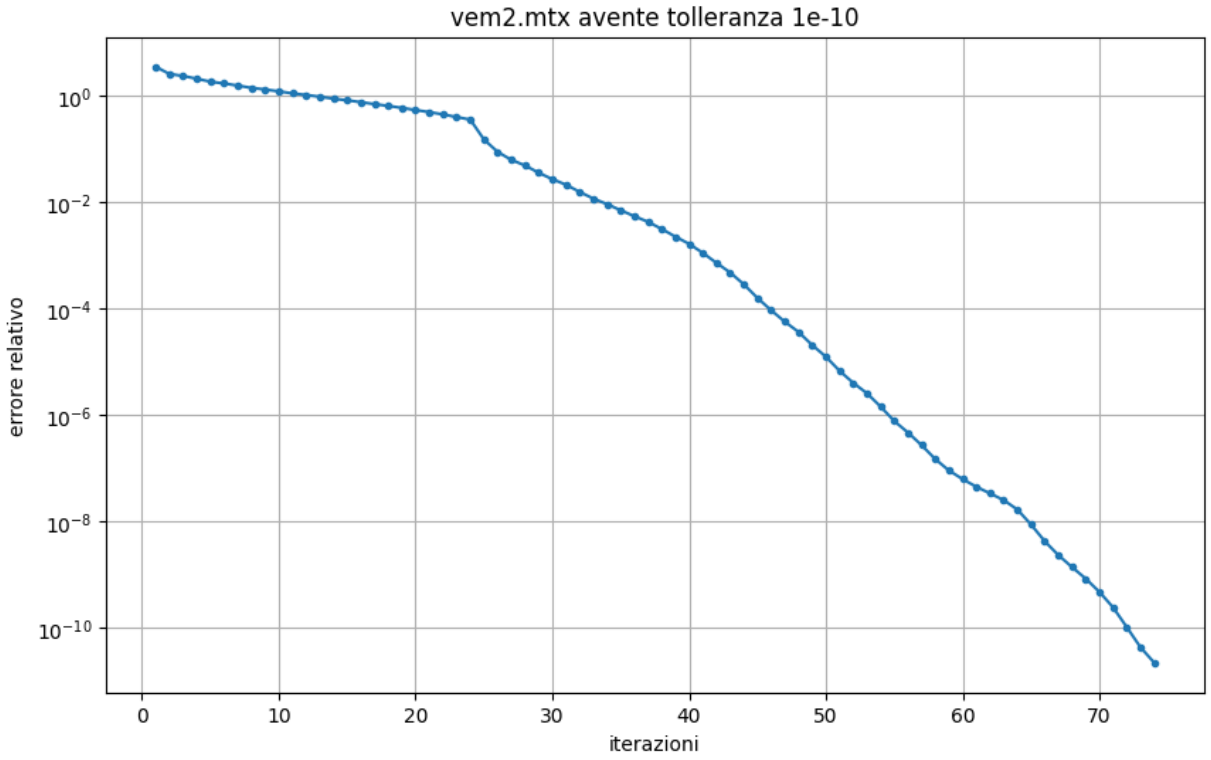

Figura 3.28: andamento dell'errore relativo per ogni iterazione con il Gradiente sulla matrice vem2.mtx

Capitolo 4

Conclusioni

Nella tabella sottostante riportiamo i migliori solutori per ogni matrice con tolleranza minima prendendo in considerazione separatamente errore relativo, iterazioni e tempo d'esecuzione.

	Spa1.mtx	Spa2.mtx	Vem1.mtx	Vem2.mtx
Tempo	Jacobi (0.141s)	Jacobi (0.297s)	Gr. co. (0.018s)	Gr. co. (0.016s)
Err. rel.	Gr. co. (1.20e-09)	Jacobi (1.48e-09)	Gr. co. (2.19e-11)	Gr. co. (2.24e-11)
Iterazioni	Gauss-Seidel (31)	Gauss-Seidel (15)	Gr. co. (59)	Gr. co. (74)

Su matrici sparse senza un pattern specifico ($spa1.mtx$, $spa2.mtx$) troviamo Jacobi come solutore più veloce, Gauss-Seidel come solutore che impiega meno iterazioni a convergere. Passando ora alle restanti due matrici sparse con dominanza diagonale notiamo che il gradiente coniugato è il migliore sotto tutti i punti di vista. L'ultimo punto che vogliamo analizzare è l'efficienza del codice. Le implementazioni sono state realizzate utilizzando funzioni di librerie già ottimizzate per impattare il meno possibile sulle performance. È dovoroso fare presente che in caso di necessità di alte performance - soprattutto in termini di tempo - Python non è il linguaggio adatto a farlo.