UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

DIPARTIMENTO DI INFORMATICA

CORSO DI STUDIO IN
INFORMATICA

Sviluppo di una libreria Python per la risoluzione di sistemi lineari tramite metodi iterativi

Membro:
Lorenzo Erba
Matricola: 933012

Anno accademico:
2024/2025

Indice

1 Introduzione

1.1 Scopo del progetto

Il progetto consiste nell'implementazione di una libreria per la risoluzione di sistemi lineari, più precisamente tramite metodi iterativi applicati a matrici sparse e definite positive.

1.2 Componenti del gruppo

- Lorenzo Erba, matricola: 933012

1.3 IDE e linguaggio di programmazione

La libreria in questione è stata realizzata in linguaggio Python, utilizzando come ambiente di sviluppo Visual Studio Code. Le motivazioni a sostegno di queste scelte risiedono in:

- semplicità. Python è un linguaggio che si contraddistingue per la sua sintassi intuitiva e un paradigma di programmazione adatto per la prototipazione rapida. Analogamente VS Code offre un'interfaccia utente e un ambiente di sviluppo intuitivo garantendo rapidità nella scrittura del codice, grazie anche ai numerosi plug-in che possono essere utilizzati.
- open-source. Sia Python che VS Code sono due artefatti software di utilizzo pubblico e gratuito, offrendo accesso a numerose librerie altamente testate e alla possibilità di effettuare troubleshooting in modo rapido ed efficiente.
- conoscenza pregressa. Python è un linguaggio che è stato ampiamente esplorato e utilizzato in numerosi ambiti, soprattutto laddove era richiesta una prototipazione rapida ed efficiente, come la realizzazione di modelli di Machine Learning.

2 Specifiche

2.1 Specifiche del progetto

Il progetto richiede l'utilizzo di un linguaggio *open-source* per implementare i seguenti metodi iterativi:

- Jacobi
- Gauß-Seidel
- Gradiente
- Gradiente coniugato

È consentito l'utilizzo di librerie esterne per gestire le strutture dati e le operazioni relative a matrici, purché non vengano utilizzati metodi built-in come risolutori iterativi.

2.2 Richieste del progetto

È richiesto che la libreria presenti un'architettura logica e che sia ben strutturata, invece che essere una sequenza di funzioni indipendenti dalle altre. Inoltre dev'essere garantito che:

- i metodi iterativi devono partire da un *initial-guess* di elementi nulli (vettore con tutte le entrate pari a zero) e arrestarsi qualora la k-esima iterata x^k soddisfi
$$\frac{\|Ax^{(k)} - b\|}{\|b\|} < \text{tol} \tag{2.1}$$
dove **tol** rappresenta la tolleranza inserita dall'utente.
- i metodi iterativi, oltre a un controllo sulla soluzione x^k, devono tenere in considerazione il numero di iterazioni effettuate. Più precisamente tutte le routine devono arrestarsi (e segnalare che non è stata raggiunta la convergenza) se
$$k > maxIter$$
con **maxIter** un numero arbitrario non inferiore a 20000.
- il codice deve prendere in input:
  - una matrice A simmetrica e definita positiva
  - vettore dei termini noti b
  - vettore soluzione esatta x

– una tolleranza tol

e riportare su schermo le informazioni sui risultati ottenuti, in termini di:

– errore relativo tra la soluzione esatta e la soluzione approssimata

– numero di iterazioni effettuate

– tempo di calcolo richiesto

La matrice sparsa e definita positiva sopracitata fa riferimento a quanto contenuto nei file *spa1.mtx,spa2.mtx,vem1.mtx,vem2.mtx*, i quali contengono matrici SPD nel formato ".mtx". Si tratta di un formato standard in ASCII per lo scambio di matrici in ambito numerico, la cui descrizione è riportata nel seguente link:

`https://math.nist.gov/MatrixMarket/formats.html`

La risoluzione di sistemi lineari tramite i metodi sopracitati deve seguire la procedura standard:

- creare un vettore **x** che rappresenta la soluzione esatta del sistema

$$x = [1, 1, \ldots, 1]$$

- creare il vettore dei termini noti **b** tramite:

$$b = Ax$$

- calcolare la soluzione approssimata tramite l'esecuzione dei quattro metodi iterativi elencati precedentemente
- calcolare l'errore relativo, tra la soluzione esatta e quella approssimata, il numero di iterazioni e il tempo di esecuzione.

Questa procedura deve essere eseguita sempre sullo stesso calcolatore e per ogni valore dell'input **tol**. In particolare le tolleranze da considerare sono:

$$tol = [1e^{-4}, 1e^{-6}, 1e^{-8}, 1e^{-10}]$$

2.3 Specifiche dei metodi iterativi

I metodi iterativi stazionari sono una classe di algoritmi utilizzati per risolvere sistemi lineari della forma:

$$Ax = b$$

dove A è una matrice quadrata, x è il vettore delle incognite e b il vettore dei termini noti. A differenza dei metodi diretti (decomposizione di Cholesky, decomposizione LU...), i metodi iterativi costruiscono una sequenza di approssimazioni della soluzione, partendo da una stima iniziale x^0 e aggiornandola secondo una certa regola. L'obiettivo è che la sequenza x^k converga alla soluzione esatta x.
Alcuni metodi iterativi utilizzano la tecnica dello **splitting** di A, o decomposizione DLU, che prevede di scomporre la matrice A come:

$$A = D - L - U \tag{2.2}$$

dove:

- A è la matrice originale
- D è la matrice diagonale con tutte le entrate pari a zero tranne la diagonale principale, che coincide con quella di A
- L è la matrice triangolare inferiore stretta moltiplicata per -1, che possiede entrate pari a zero sulla diagonale principale e sopra di essa.
- U è la matrice triangolare superiore stretta moltiplicata per -1, che possiede entrate pari a zero sulla diagonale principale e sotto di essa.

Per esempio:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \underbrace{\begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix}}_{D} - \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ -a_{21} & 0 & 0 \\ -a_{31} & -a_{32} & 0 \end{bmatrix}}_{L} - \underbrace{\begin{bmatrix} 0 & -a_{12} & -a_{13} \\ 0 & 0 & -a_{23} \\ 0 & 0 & 0 \end{bmatrix}}_{U}$$

La tecnica di splitting di A offre diversi vantaggi, soprattutto dal punto di vista computazionale, poiché:

- essendo la matrice A sparsa, la decomposizione DLU preserva questa proprietà, ottimizzando la lettura/scrittura in memoria, in quanto è possibile memorizzare solo le entrate diverse da zero come (i, j, a_{ij}) con i,j rispettivamente l'indice di riga e colonna e a_{ij} il valore associato.
- la matrice D è diagonale, pertanto è possibile calcolare la matrice inversa invertendo le entrate sulla diagonale, costituendo un'operazione veloce e stabile numericamente.
- L, U sono triangolari strette, pertanto è possibile applicare tecniche di sostituzione *backward* e *forward*.

I metodi iterativi computano l'approssimazione della soluzione sino a quando non è soddisfatta una condizione di arresto, descritta dall'equazione (2.1). Questa disuguaglianza descrive il residuo normalizzato, ovvero quanto bene $x^{(k+1)}$ soddisfa l'equazione $Ax = b$. Allo stesso tempo, l'errore restituito dai metodi iterativi è calcolato come l'errore relativo dato dall'equazione:

$$ER = \frac{||x^{(k+1)} - x||}{||x||} \tag{2.3}$$

dove:

- $||x^{(k+1)} - x||$ è l'errore assoluto
- $||x||$ è la norma della soluzione esatta

2.4 Metodo di Jacobi

Il metodo di Jacobi è un metodo iterativo stazionario utilizzato per la risoluzione di sistemi lineari della forma

$$Ax = b$$

dove A è una matrice simmetrica e definita positiva. Questo metodo utilizza come concetto chiave la tecnica di **splitting** di A, scomponendo la matrice iniziale come descritto nell'equazione (2.2). L'idea di base del metodo di Jacobi prevede di partire da un vettore soluzione, *initial-guess*, $x^{(0)}$ e costruire una sequenza $x^{(1)}, x^{(2)}, \ldots$, di soluzioni approssimate, cercando di raggiungere la convergenza. Più precisamente $x^{(k+1)}$ viene calcolata come:

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b \tag{2.4}$$

o equivalentemente, ogni entrata del vettore soluzione coincide con la formula:

$$x_i^{(k+1)} = \frac{1}{D_{ii}} \left[b_i - \sum_{j=1, j \neq i} A_{ij} x_j^{(k)} \right] \tag{2.5}$$

Teorema 2.4.1. Se A è una matrice a dominanza diagonale, ovvero $\forall_{i=1\ldots N} \quad |a_{ii}| > \sum_{j=1, j \neq i} |a_{ij}|$, allora il metodo di Jacobi converge.

Vantaggi:

- Semplicità di implementazione.
- Parallelizzazione. Ogni valore del vettore soluzione, $x_i^{(k+1)}$, dipende solo dal valore precedente, $x_i(k)$, rendendo ideale l'implementazione su sistemi multi-core.

Svantaggi:

- Più lento di altri metodi iterativi.
- Richiede un numero di iterazioni elevate per raggiungere la convergenza se la matrice non è ben condizionata.

2.5 Gauß-Seidel

Il metodo di Gauß-Seidel è un metodo iterativo stazionario usato per trovare una soluzione approssimata a un sistema di equazioni lineari del tipo:

$$Ax = b$$

dove A è una matrice simmetrica e definita positiva. Analogamente al caso precedente, il metodo di Gauß-Seidel utilizza la tecnica di **splitting** della matrice A, ottenendo le matrici necessarie come indicato nell'equazione (2.2). L'idea è di partire da una stima iniziale $x^{(0)}$, *initial-guess*, per poi migliorarla iterativamente. Più precisamente la *k+1-esima* iterazione viene calcolata come:

$$x^{(k+1)} = (D - L)^{-1}Ux^{(k)} + (D - L)^{-1}b \tag{2.6}$$

o equivalentemente, ogni entrata del vettore soluzione coincide con l'equazione:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \right) \tag{2.7}$$

Nella pratica, invertire una matrice triangolare richiede un costo computazionale maggiore rispetto all'inversione di una matrice diagonale, come nel caso di Jacobi. Pertanto, il calcolo dell'equazione (2.6) si riduce alla risoluzione di un sistema triangolare inferiore della forma:

$$(D - L)x^{(k+1)} = Ux^{(k)} + b \tag{2.8}$$

richiedendo un costo computazionale pari a $O(n^2)$, nel caso di *forward substitution.*
Il teorema (2.4.1) preserva la sua validità anche nel caso del metodo di Gauß-Seidel.

Vantaggi: Il metodo di Gauß-Seidel spesso converge più rapidamente del metodo di Jacobi.

Svantaggi: Richiede operazioni sequenziali, il che lo rende meno parallelizzabile rispetto a Jacobi.

2.6 Metodo del gradiente

Il metodo del gradiente è un metodo iterativo non stazionario usato per risolvere sistemi della forma:

$$Ax = b$$

A differenza dei metodi iterativi stazionari, dove la matrice utilizzata per aggiornare la soluzione rimane costante, nei metodi non stazionari si adotta un procedimento che adatta direzione e passo ad ogni iterazione e l'informazione usata cambia dinamicamente, in modo da guidare meglio la discesa verso la soluzione. Il metodo si basa sull'idea di minimizzare una funzione obiettivo:

$$\phi(x) = \frac{1}{2}x^T Ax - b^T x \tag{2.9}$$

che presenta un unico minimo globale, essendo A definita positiva, dove quel minimo coincide con la soluzione del sistema $Ax = b$. La direzione di massima discesa della funzione ϕ, è data dal negativo del gradiente $\nabla\phi(x) = Ax - b = -r$ dove $r = b - Ax$ è il residuo. L'idea alla base consiste nel muoversi, ad ogni iterazione, lungo la direzione del gradiente negativo (lungo il residuo), cercando di avvicinarsi il più possibile al minimo di $\phi(x)$.
Quindi l'obbiettivo è calcolare:

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

dove:

- α_k rappresenta il passo, ovvero quanto ci spostiamo lungo la direzione del gradiente negativo,
- $\eta^{(k)}$ rappresenta la direzione di discesa, il gradiente negativo o r^k, il residuo al passo k.

Quindi il residuo al passo k lo possiamo esprimere come:

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

Volendo minimizzare $f(\alpha_k) = \phi(x^{(k)} + \alpha_k r^{(k)})$, è necessario calcolare la derivata prima di $f(\alpha_k)$ e porla uguale a zero. Pertanto il valore ottimo del passo lungo la direzione $r^{(k)}$, che minimizza $\phi(x)$, su quella retta è:

$$\alpha_k = \frac{{r^{(k)}}^T r^{(k)}}{{r^{(k)}}^T A r^{(k)}}$$

Teorema 2.6.1. Se A è una matrice a simmetrica e definita positiva, allora il metodo del gradiente converge.

Vantaggi:

- Semplicità concettuale, derivando da un'idea geometrica di discesa lungo il gradiente.

- Converge per matrici simmetriche e definite positive.
- Ottimale per sistemi di grandi dimensioni e sparsi, comune in problemi ingegneristici.

Svantaggi:

- Convergenza lenta, soprattutto se la matrice iniziale ha un numero di condizionamento elevato.
- La discesa segue ogni volta la direzione del gradiente, ma poiché questa cambia ad ogni iterazione e non è ortogonale alla precedente, il metodo può presentare un andamento a "zig-zag", con progressi lenti verso la soluzione. Questo aspetto viene risolto dal metodo del gradiente coniugato.

2.7 Metodo del gradiente coniugato

Il metodo del gradiente coniugato può essere considerato come una versione 2.0 del metodo del gradiente tradizionale. Questo metodo risolve il problema della convergenza a "zig-zag" e garantisce una convergenza, per matrici simmetriche e definite positive di dimensione n, in $O(n)$ iterazioni. Il metodo del gradiente coniugato cerca anch'esso di minimizzare la funzione obiettivo (2.9), ma invece di muoversi sempre nella direzione del gradiente, sceglie delle direzioni coniugate rispetto ad A, che non interferiscono tra loro, rendendo la discesa molto più efficiente. Pertanto, il residuo iniziale e la direzione iniziale risultano essere:

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

Il passo α_k viene calcolato come:

$$\alpha_k = \frac{{r^{(k)}}^T r^{(k)}}{{p^{(k)}}^T A p^{(k)}}$$

L'aggiornamento della soluzione equivale a:

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

Successivamente viene aggiornato il residuo con la seguente equazione:

$$r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)}$$

Infine viene calcolato il coefficiente β_k per poter aggiornare la nuova direzione coniugata $p^{(k+1)}$:

$$\beta_k = \frac{{r^{(k+1)}}^T r^{(k+1)}}{{r^{(k)}}^T r^{(k)}}$$

$$p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)}$$

Vantaggi:

- Convergenza veloce in $O(n)$ iterazioni.
- Adatto per matrici sparse in quanto richiede solo operazioni matrice-vettore.

Svantaggi: Più complesso di altri metodi, in quanto ogni iterazione richiede un numero maggiore di operazioni.

3 Implementazione

3.1 Repository della libreria

Lo sviluppo della libreria è avvenuto tramite l'utilizzo della piattaforma di versioning di GitHub ed è disponibile al seguente link.

3.2 Struttura della libreria

La libreria è stata sviluppata in Python tramite il supporto della libreria NumPy, in modo da sfruttare le strutture dati e operazioni matriciali necessarie a implementare i metodi iterativi richiesti. La libreria si compone di due classi principali:

- IterativeMethods.py. Si tratta della classe principale, il *core* della libreria. Fornisce la struttura dati e l'implementazione dei quattro metodi iterativi per poter risolvere sistemi lineari.
- Util.py. È una classe che ha lo scopo di incapsulare al suo interno una serie di metodi di supporto, che possono essere utili in più parti del codice, ma che non appartengono a una specifica logica di business.

3.3 IterativeMethods.py

La classe ItherativeMethods possiede i seguenti metodi pubblici:

- Costruttore parametrico.
- Jacobi.
- Gauß_Seidel.
- CG.
- CG_conj.
- singlePlotResult.
- groupPlotResult.

e i segeunti metodi privati:

- triangLower.
- getRelativeError.
- getErrorForExit.

3.3.1 Costruttore parametrico

Il costruttore parametrico della classe IterativeMethods

```
def __init__(self, A, b, xe, x0, maxIter=20000):
    self.A = A
    self.b = b
    self.xe = xe
    if(maxIter < 20000 ):
        self.maxIter = 20000
    else:
        self.maxIter = maxIter
    self.x0 = x0
```

ha lo scopo di inizializzare un'istanza di questa classe, che possiede i seguenti attributi:

- A rappresenta la matrice associata al problema $Ax = b$.
- b rappresenta il vettore dei termini noti.
- xe esprime la soluzione esatta del sistema, necessaria a calcolare l'errore relativo.
- x0 rappresenta l'*initial-guess*, la soluzione iniziale al passo 0.
- maxIter esprime il massimo numero di iterazioni eseguibili (di default impostato a 20000).

3.3.2 Jacobi

Metodo che implementa l'algoritmo di Jacobi.

```
def jacobi(self, toll, info = True):
    util = Util()
    D = np.diag(np.diagonal(self.A))
    if(util.diagonalNotZero(D) is False):
        print("[WARN] La diagonale non è invertibile!")
        return False
    B = D - self.A
    D_inv = 1 / np.diagonal(D)

    xold = self.x0
    err = 1
    nit = 0
    stime = time.time()
    while err > toll and nit < self.maxIter:
        xnew = D_inv * (np.dot(B, xold) + self.b)
        err = self.__getErrorForExit(xnew)
```

```
            xold = xnew
            nit = nit + 1

        elapsed_time = (time.time() - stime) * 1000

        err = self.__getRelativeError(xnew)

        if(nit == self.maxIter):
            print("Convergenza non raggiunta!")

        if info:
            print("Jacobi")
            print("\tNumero di iterazioni:", nit)
            print("\tErrore:", err)
            print("\tTempo di esecuzione:", elapsed_time,
             ↪  "ms\n")

        return xnew, err ,nit, elapsed_time
```

Questo metodo prende in input il parametro **toll** rappresentante la tolleranza e un parametro opzionale **info**, di default *True*, per poter abilitare la visualizzazione su console di informazioni come il numero di iterazioni,errore relativo e tempo di esecuzione. Successivamente calcola la matrice diagonale D, sottrae la matrice A per ottenere la matrice B e calcola l'inverso di D. Se la matrice D non è invertibile viene stampato un messaggio di errore e il metodo restituisce *False*.
Inizialmente l'errore è pari a 1, per poter consentire almeno un'iterazione, e la soluzione al passo 0 coincide con l'*initial-guess*. A ogni iterazione si aggiorna la soluzione approssimata xnew, secondo l'equazione (2.4) e si controlla se si è raggiunta la convergenza confrontando il residuo normalizzato con la tolleranza. Se il metodo raggiunge il massimo numero di iterazioni supportate, viene stampato in console che la convergenza non è stata raggiunta.

3.3.3 Gauß_Seidel

Metodo che implementa l'algoritmo di Gauß-Seidel.

```
    def Gauss_Seidel(self,toll, info=True):
        D_L = np.tril(self.A)
        U = self.A - D_L
        xold = self.x0
        nit = 0
        err=1
        stime = time.time()
        while(err > toll and nit < self.maxIter):
```

```
            xnew =
             ↪  self.__triangLower(D_L,(self.b-np.dot(U,xold)))
            err = self.__getErrorForExit(xnew)
            xold = xnew
            nit = nit+1

        elapsed_time = (time.time()-stime)*1000
        err = self.__getRelativeError(xnew)

        if(nit == self.maxIter):
            print("Convergenza non raggiunta!")

        if info:
            print("Gauss_Seidel")
            print("\tNumero di iterazioni:", nit)
            print("\tErrore:", err)
            print("\tTempo di esecuzione:", elapsed_time,
             ↪  "ms\n")

        return xnew,err,nit,elapsed_time
```

Questo metodo prende in input il parametro **toll** rappresentante la tolleranza e un parametro opzionale **info**, di default *True*, per poter abilitare la visualizzazione su console di informazioni come il numero di iterazioni, errore relativo e tempo di esecuzione.
Successivamente calcola D_L che rappresenta (D-L) e sottrae ad A la matrice appena calcolata per ottenere U.
Inizialmente l'errore è pari a 1, per poter consentire almeno un'iterazione, e la soluzione al passo 0 coincide con l'*initial-guess*. A ogni iterazione si aggiorna la soluzione approssimata xnew, secondo l'equazione (2.6). Più precisamente viene chiamato il metodo

```
self.__triangLower(D_L,(self.b-np.dot(U,xold)))
```

per poter risolvere un sistema triangolare inferiore, e infine si controlla se si è raggiunta la convergenza, confrontando il residuo normalizzato con la tolleranza. Se il metodo raggiunge il massimo numero di iterazioni supportate, viene stampato in console che la convergenza non è stata raggiunta.

3.3.4 CG

Metodo che implementa l'algoritmo del gradiente.

```
def CG(self, toll, info = True):
    nit=0
    xold=self.x0
    err = 1
```

```
        stime = time.time()
        while(err > toll and nit < self.maxIter):
            residual = self.b - np.dot(self.A,xold)
            alpha =
             ↪  np.dot(residual.T,residual)/np.dot(residual.T,np.dot(self.A,r
            xnew = xold + alpha*residual
            err = self.__getErrorForExit(xnew)
            xold = xnew
            nit = nit+1

        elapsed_time = (time.time()-stime)*1000

        err = self.__getRelativeError(xnew)

        if(nit == self.maxIter):
            print("Convergenza non raggiunta!")

        if info:
            print("Gradiente")
            print("\tNumero di iterazioni:", nit)
            print("\tErrore:", err)
            print("\tTempo di esecuzione:", elapsed_time,
             ↪  "ms\n")
        return xnew,err,nit,elapsed_time
```

Questo metodo prende in input il parametro **toll** rappresentante la tolleranza e un parametro opzionale **info**, di default *True*, per poter abilitare la visualizzazione su console di informazioni come il numero di iterazioni,errore relativo e tempo di esecuzione.
Inizialmente l'errore è pari a 1, per poter consentire almeno un'iterazione, e la soluzione al passo 0 coincide con l'*initial-guess*. A ogni iterazione si calcola il residuo, il passo e si aggiorna la soluzione approssimata xnew, per minimizzare la funzione (2.9) e si controlla se si è raggiunta la convergenza confrontando la norma del residuo con la tolleranza. Se il metodo raggiunge il massimo numero di iterazioni supportate, viene stampato in console che la convergenza non è stata raggiunta.

3.3.5 CG_conj

Metodo che implementa l'algoritmo del gradiente coniugato.

```
    def CG_conj(self, toll, info = True):
        util = Util()
        if(util.spdCheck(self.A) is False):
            print("[WARN] La matrice in input non è SPD!")
            return False
```

```
        rold = self.b - (np.dot(self.A,self.x0))
        dold = rold.copy()
        err = 1
        nit = 0
        xold = self.x0
        stime = time.time()
        while err > toll and nit < self.maxIter:
            y_k = (np.dot(self.A,dold))
            alpha = (dold @ rold) / (dold @ y_k)

            xnew = xold + alpha * dold
            rnew = rold - alpha * y_k

            w_k = np.dot(self.A,rnew)
            beta_k = (dold @ w_k) / (dold @ y_k)
            dnew = rnew - beta_k * dold

            err = self.__getErrorForExit(xnew)
            xold = xnew
            rold = rnew
            dold = dnew
            nit += 1

        elapsed_time = (time.time() - stime)*1000

        err = self.__getRelativeError(xnew)

        if(nit == self.maxIter):
            print("Convergenza non raggiunta!")

        if info:
            print("Gradiente coniugato")
            print("\tNumero di iterazioni:", nit)
            print("\tErrore:", err)
            print("\tTempo di esecuzione:", elapsed_time,
             ↪  "ms\n")
        return xnew, err, nit, elapsed_time
```

Questo metodo prende in input il parametro **toll** rappresentante la tolleranza e un parametro opzionale **info**, di default *True*, per poter abilitare la visualizzazione su console di informazioni come il numero di iterazioni, errore relativo e tempo di esecuzione. Se la matrice in input non è SPD, il metodo stampa un messaggio di errore e restituisce *False*.
Inizialmente l'errore è pari a 1, per poter consentire almeno un'iterazione, e la soluzione al passo 0 è pari all'*initial-guess*. A ogni iterazione si calcola il

passo, si aggiorna la soluzione approssimata xnew, si aggiorna il residuo e si calcola il coefficiente β per poter aggiornare la direzione coniugata. Infine si controlla se si è raggiunta la convergenza confrontando la norma del residuo e la tolleranza. Se il metodo raggiunge il massimo numero di iterazioni supportate senza mai raggiungere la convergenza, viene stampato in console che la convergenza non è stata raggiunta.

3.3.6 SinglePlotResult

Metodo che consente la generazione di tre grafici aggregati con scale logaritmiche per visualizzare il comportamento, con diverse tolleranze, di uno specifico metodo iterativo in termini di:

- Errore relativo.
- Tempo d'esecuzione.
- Numero di iterazioni.

```
def singlePlotResult(self, method, err_list,
↪  elapsed_time_list, iterations_list,tol_list,file):

    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10,
    ↪  12))
    ax1.plot(tol_list, err_list, marker='d', label=method)
    ax2.plot(tol_list, elapsed_time_list, marker='d',
    ↪  label=method)
    ax3.plot(tol_list, iterations_list, marker='d',
    ↪  label=method)

    ax1.set_xscale('log')
    ax1.set_yscale('log')
    ax1.set_xlabel('Tolerance')
    ax1.set_ylabel('Error')
    ax1.set_title(f'Tolerance vs Error for {file}')
    if ax1.has_data():
        ax1.legend()
    ax1.grid(True)

    ax2.set_xscale('log')
    ax2.set_yscale('log')
    ax2.set_xlabel('Tolerance')
    ax2.set_ylabel('Execution Time (ms)')
    ax2.set_title(f'Tolerance vs Execution Time for
    ↪  {file}')
    if ax2.has_data():
        ax2.legend()
```

```
        ax2.grid(True)

        ax3.set_xscale('log')
        ax3.set_yscale('log')
        ax3.set_xlabel('Tolerance')
        ax3.set_ylabel('Iterations')
        ax3.set_title(f'Tolerance vs Iterations for {file}')
        if ax3.has_data():
            ax3.legend()
        ax3.grid(True)

        plt.subplots_adjust(hspace=0.7)
        plt.show()
```

Questo metodo prende in input:

- il nome del metodo iterativo,
- la lista degli errori relativi,
- la lista dei tempi d'esecuzione,
- la lista delle iterazioni,
- la lista delle tolleranze,
- il nome del file da cui proviene la matrice.

Il metodo provvede a generare un unico grafico composto da 3 sottoelementi per esprimere una relazione tra le tolleranze e i rispettivi parametri misurati durante l'esecuzione di un metodo specifico. La rappresentazione avviene in scala logaritmica dove ogni grafo possiede una legenda esplicativa.

3.3.7 GroupPlotResult

Metodo che consente la generazione di un grafo aggregato per ciascuno dei metodi iterativi implementati dalla libreria.

```
    def groupPlotResult(self,results,tol_list,methods,file):
        fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10,
        ↪  12))

        for m in methods:
            error_list = []
            time_list = []
            nit_list = []
            for tol in tol_list:
                key = f"{m}_{tol}"
```

```
                try:
                    xnew, err, nit, elapsed_time = results[key]
                except KeyError:
                    print(f"[WARN] Chiave mancante: {key}")
                time_list.append(elapsed_time)
                error_list.append(err)
                nit_list.append(nit)

            ax1.plot(tol_list, error_list, marker='d', label=m)
            ax2.plot(tol_list, time_list, marker='d', label=m)
            ax3.plot(tol_list, nit_list, marker='d', label=m)

        ax1.set_xscale('log')
        ax1.set_yscale('log')
        ax1.set_xlabel('Tolerance')
        ax1.set_ylabel('Error')
        ax1.set_title(f'Tolerance vs Error for {file}')
        if ax1.has_data():
            ax1.legend()
        ax1.grid(True)

        ax2.set_xscale('log')
        ax2.set_yscale('log')
        ax2.set_xlabel('Tolerance')
        ax2.set_ylabel('Execution Time (ms)')
        ax2.set_title(f'Tolerance vs Execution Time for
        ↪  {file}')
        if ax2.has_data():
            ax2.legend()
        ax2.grid(True)

        ax3.set_xscale('log')
        ax3.set_yscale('log')
        ax3.set_xlabel('Tolerance')
        ax3.set_ylabel('Iterations')
        ax3.set_title(f'Tolerance vs Iterations for {file}')
        if ax3.has_data():
            ax3.legend()
        ax3.grid(True)

        plt.tight_layout()
        plt.savefig(f"Plot_{file}.png")
        plt.show()
```

Questo metodo prende in input:

- lista di risultati (errori relativi, numero di iterazioni e tempi di esecuzione),
- dictionary delle tolleranze,
- lista dei metodi,
- nome del file da cui è stata letta la matrice

Il metodo provvede a calcolare, per ciascun metodo iterativo, il corrispondente single plot. La differenza principale, rispetto al metodo precedente, consiste nella presenza della lista **results**, che contiene un insieme misto di risultati, che differiscono per metodo e tolleranza associati. Pertanto è essenziale che il dictionary abbia come chiave la seguente struttura: <metodo>_<tol>, per esempio Jacobi_$1e^{-10}$ rappresenta la chiave relativa all'esecuzione di Jacobi con una tolleranza pari a $1e^{10}$. Ogni chiave del dizionario punta a una lista contenente i risultati dell'esecuzione del metodo identificato dall'identificatore. È consigliabile usare come risultato quanto restituito automaticamente dai metodi implementati nella libreria. Se la chiave non è presente nel dizionario, viene stampato in console un messaggio di errore e si procede a generare una nuova chiave. Analogamente al caso precedente, i grafici riportano i valori in scala logaritmica e in aggiunta, dopo la loro generazione, viene salvata un'immagine del grafico nel percorso relativo del progetto.

3.3.8 TriangLower

Metodo utilizzato per poter risolvere un sistema triangolare inferiore. Questo metodo viene invocato nel metodo di Gauß-Seidel per poter risolvere il sistema triangolare inferiore (2.8).

```
def __triangLower(self, L, b):
    M, N = L.shape
    x = np.zeros(M)

    x[0] = b[0] / L[0, 0]
    for i in range(1, N):
        x[i] = (b[i] - np.dot(L[i, :i], x[:i])) / L[i, i]

    return x
```

Il metodo prende in input una matrice L triangolare inferiore e un vettore b, che rappresentano il sistema $Lx = b$ da risolvere. Nel dettaglio, il metodo applica la tecnica della *forward substitution* che richiede un costo computazionale pari a $O(n^2)$ con n la dimensione della matrice L, e restituisce la soluzione x calcolata.

3.3.9 GetRelativeError

Questo metodo calcola l'errore relativo tra la soluzione esatta e quella approssimata, ottenuta dopo l'applicazione di un metodo iterativo. L'invocazione di questo metodo avviene in ogni metodo iterativo implementato nella libreria, nella riga successiva all'uscita dal ciclo delle iterazioni, in quanto l'errore relativo dev'essere calcolato sull'ultima soluzione iterata, quella considerata migliore e che, auspicabilmente, determina la convergenza.

```
def __getRelativeError(self,x_app):
    return np.linalg.norm(x_app-self.xe) /
    ↪  np.linalg.norm(self.xe)
```

Il metodo prende in input la soluzione approssimata e restituisce l'errore relativo calcolato come nell'equazione (2.3).

3.3.10 GetErrorForExit

Questo metodo calcola la norma del residuo, come indicato nell'equazione (2.1), per poter essere confrontata con la tolleranza scelta e stabilire pertanto una condizione di arresto.

```
def __getErrorForExit(self,xnew):
    return np.linalg.norm(np.dot(self.A, xnew) - self.b) /
    ↪  np.linalg.norm(self.b)
```

Il metodo prende in input la soluzione approssimata, xnew, e calcola il residuo normalizzato in modo da indicare quanto la soluzione corrente viola il sistema lineare $Ax = b$.

3.4 Util.py

La classe Util non dispone di un costruttore parametrico in quanto non è caratterizzata da attributi/proprietà e possiede i seguenti metodi pubblici:

- CheckInstance.
- SymmetryCheck.
- SquareCheck.
- MatrixVectorSameLength.
- DiagonalNotZero.
- ReadFromFile.
- SpdCheck.
- IsDiagonallyDominant.

3.4.1 CheckInstance

Il metodo checkInstance viene invocato in molteplici metodi per poter determinare se la matrice A in ingresso è un oggetto istanza della classe **np.ndarray**. Questa condizione è necessaria affinchè la classe IterativeMethods funzioni correttamente, poichè fa uso di numerosi metodi implementati nella libreria NumPy.

```
def __checkInstance(self,A):
    return isinstance(A,np.ndarray)
```

Il metodo prende in input la matrice A e, tramite il metodo built-in **isinstance** di Python, restituisce lo stesso output della funzione invocata, ovvero:

- *True* se A è istanza di *np.ndarray*,
- *False* altrimenti.

3.4.2 SymmetryCheck

Il metodo symmetryCheck viene invocato nel metodo spdCheck e permette di determinare se la matrice A in input è simmetrica.
Una matrice A si dice simmetrica se:

- A è una matrice quadrata.
- Gli elementi simmetrici rispetto alla diagonale principale sono uguali. In altre parole se la matrice trasposta è la matrice stessa.

```
def symmetryCheck(self,A):
    return np.allclose(A,A.T) if self.__checkInstance(A)
    ↪  and self.squareCheck(A) else False
```

Il metodo prende in input la matrice A e in primis controlla se si tratta di un oggetto istanza della classe *np.ndarray*. Poiché Python esegue gli operatori logici tramite *lazy-evaluation*, se la matrice A non è istanza della classe indicata, non viene invocato il metodo squareCheck. Viceversa, se è istanza di *np.ndarray*, si controlla se A è una matrice quadrata (prima condizione affinché sia simmetrica). Se i controlli precedenti forniscono un esito positivo, si procede a controllare se ogni elemento di A e della sua trasposta (A.T) sono simili, mantenendo la tolleranza relativa (rtol) e assoluta(atol) di base. Più precisamente si controlla se $absolute(a-b) <= (atol + rtol * absolute(b))$ è vera element-wise. Se quanto descritto è verificato, viene restituito *True*, altrimenti *False*.

3.4.3 SquareCheck

Il metodo squareCheck implementa una serie di controlli per poter determinare se la matrice in input è quadrata, ovvero se una generica matrice $A \in R^{n \times n}$.

```
def squareCheck(self,A):
    return A.shape[0] == A.shape[1] if
    ↪  self.__checkInstance(A) is True else False
```

Il metodo prende in input la matrice A e controlla se è istanza di *np.ndarray*. Nel caso positivo provvede a restituire *True* se il numero di righe di A coincide con il numero di colonne, altrimenti restituisce *False*.

3.4.4 MatrixVectorSameLength

Il metodo matrixVectorSameLength controlla se il vettore dei termini noti b è della stessa dimensione della matrice A, in altre parole se b deriva dal sistema a cui è associata A, e quindi che A abbia tante righe quanti elementi contenuti in b.

```
def matrixVectorSameLength(self,A,b):
    return A.shape[0] == b.shape[0] if
    ↪  (self.__checkInstance(A) and
    ↪  self.__checkInstance(b)) is True else False
```

Il metodo prende in input la matrice A e il vettore b. Dopo aver controllato che A, b siano istanze di *np.ndarray*, si confrontano il numero di righe di A con la lunghezza di b. Se queste due dimensioni coincidono viene restituito *True*, altrimenti *False*.

3.4.5 DiagonalNotZero

Il metodo diagonalNotZero controlla se gli elementi sulla diagonale principale siano diversi da zero. Questo controllo è necessario prima di eseguire il metodo di Jacobi in quanto è richiesta l'inversione della matrice diagonale D, in modo da scongiurare eventuali divisioni per zero.

```
def diagonalNotZero(self,A):
    threshold = 1e-16
    return not np.any(np.abs(np.diag(A)) < threshold) if
    ↪  self.__checkInstance(A) else False
```

Il metodo prende in input la matrice A e, dopo aver controllato che l'oggetto sia istanza della classe *np.ndarray*, verfica che ogni elemento sulla diagonale sia, in modulo, maggiore di una determinata soglia, sotto la quale ogni qualsiasi valore viene considerato automaticamente zero. Se ogni condizione viene validata, il metodo restituisce *True* altrimenti *False*.

3.4.6 ReadFromFile

Il metodo ReadFromFile provvede a leggere il contenuto del file specificato in input (in formato ".mtx"), successivamente convertito in un oggetto della classe *np.ndarray*.

```
def readFromFile(self,path):
    return mmread(path).toarray()
```

Il metodo prende in input il percorso in cui si trova il file *.mtx* e provvede a leggerne il contenuto tramite il metodo *mmread* della libreria SciPy.

3.4.7 SpdCheck

Il metodo spdCheck controlla se la matrice A in input è simmetrica e definita positiva (Symmetric and Positive Definite). Questa è uan condizione sufficiente ma non necessaria affinché il metodo del gradiente raggiunga la convergenza, mentre invece è necessaria per il metodo del gradiente coniugato. Una matrice A si dice SPD se:

- A è simmetrica, ovvero A è uguale alla sua trasposta,
- A è definita positiva, ovvero se $x^T Ax > 0 \quad \forall x \in \mathbb{R}^n,\ x \neq 0$

Per verificare se una matrice è SPD è possibile:

- Controllare se tutti gli autovalori sono positivi oppure,
- eseguire la decomposizione di Cholesky. Se la matrice non è SPD Cholesky si arresta lanciando un'eccezione.

```
def spdCheck(self,A):
    if(self.__symmetryCheck(A)):
        try:
            cholesky(A)
            return True
        except:
            return False
    else:
        return False
```

Il metodo prende in input la matrice A e dopo aver controllato che sia simmetrica, esegue la decomposizione di Cholesky. Se durante l'esecuzione viene lanciata un'eccezione allora il metodo restituisce *False*, altrimenti se Cholesky termina senza errori, viene restituito *True*.

3.4.8 IsDiagonallyDominant

Il metodo isDiagonallyDominant controlla se la matrice A in input è a dominanza diagonale. Questa condizione è sufficiente, ma non necessaria, affinché il metodo di Jacobi e Gauß-Seidel raggiungano la convergenza. //Una matrice A si dice a dominanza diagonale se: $|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \quad$ per ogni $i = 1, \ldots, n$ ovvero se, in ogni riga, il valore assoluto dell'elemento sulla diagonale principale è maggiore o uguale alla somma dei valori assoluti di tutti gli altri elementi della stessa riga.

```
def isDiagonallyDominant(self,A):
    if(self.checkInstance(A)):
        for i, row in enumerate(A):
            s = sum(abs(v) for j, v in enumerate(row) if i
            ↪  != j)
            if s > abs(A[i][i]):
                return False
        return True
    else:
        return False
```

Il metodo prende in input la matrice A, e dopo aver controllato che sia un'istanza di *np.ndarray*, procede a verificare se A è dominante diagonalmente. In caso positivo il metodo restituisce *True*, altrimenti *False*.

4 Analisi dei risultati

In questo capitolo verranno elencati e analizzati i risultati ottenuti dall'esecuzione dei seguenti metodi iterativi:

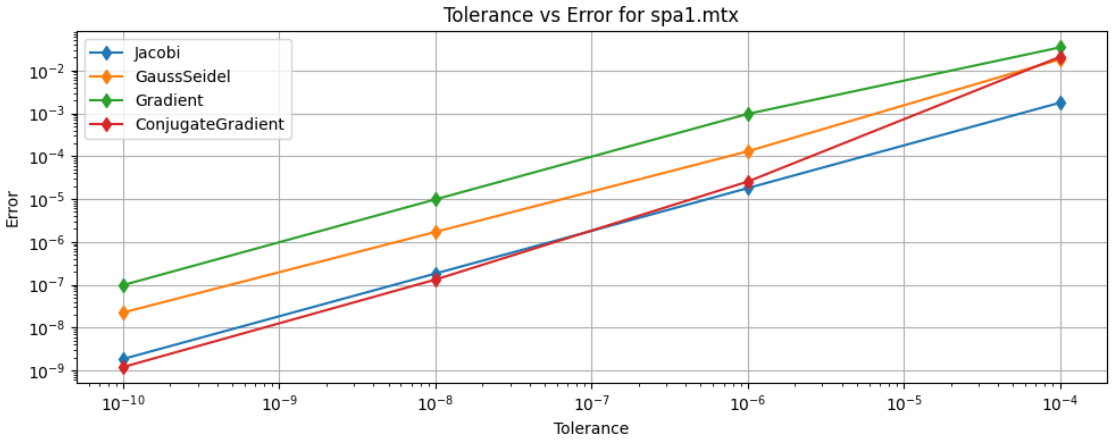
- Jacobi,
- Guaß-Seidel,
- metodo del Gradiente,
- metodo del gradiente coniugato,

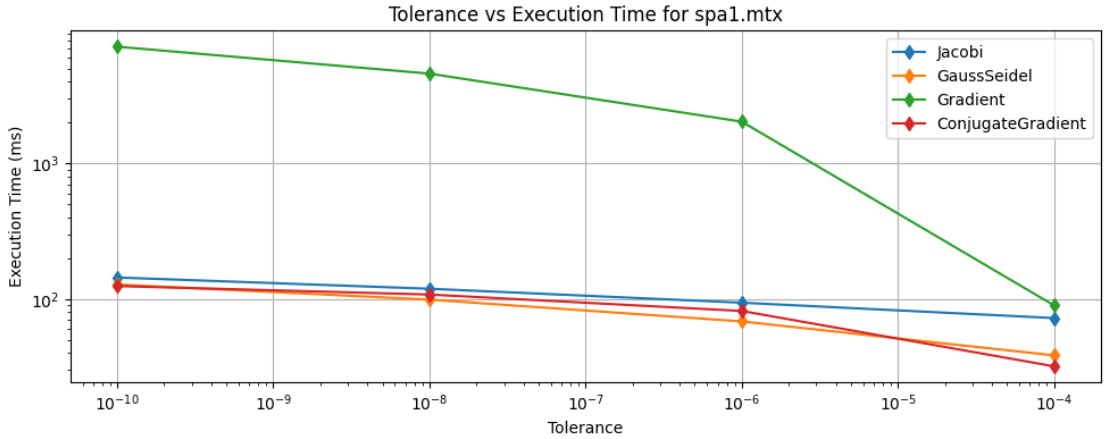
considerando le seguenti tolleranze $tol = [1e^{-4}, 1e^{-6}, 1e^{-8}, 1e^{-10}]$. Ciascuna combinazione metodo-tolleranza è stata applicata alle matrici contenute nei file *spa1.mtx, spa2.mtx, vem1.mtx, vem2.mtx*. I risultati sono mostrati attraverso dei grafici, tre per ciascun file, dove vengono riportati i valori relativi a:

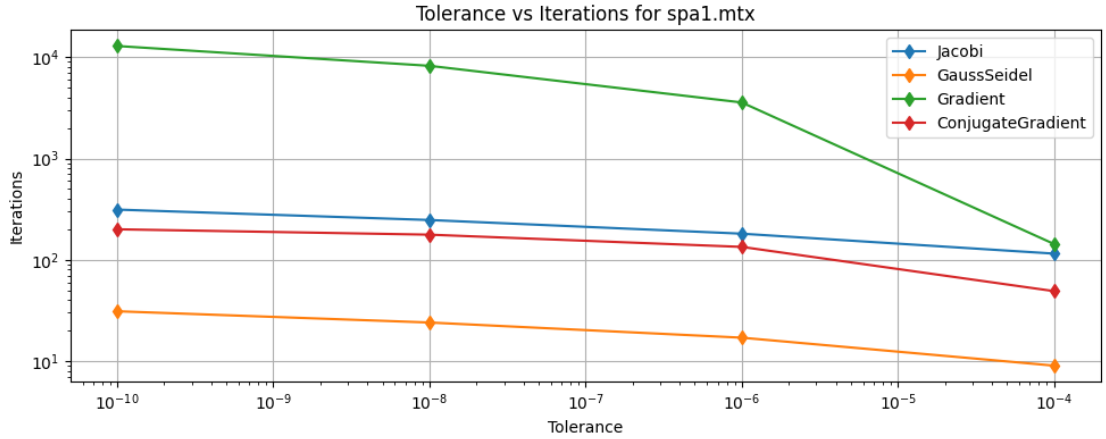
- errore relativo,
- numero di iterazioni,
- tempo d'esecuzione,

in relazione a diverse tolleranze. Ogni valore è stato riportato in scala logaritmica in modo da rendere il grafo indipendente da fattori di scala.

4.1 Risultati su spa1.mtx







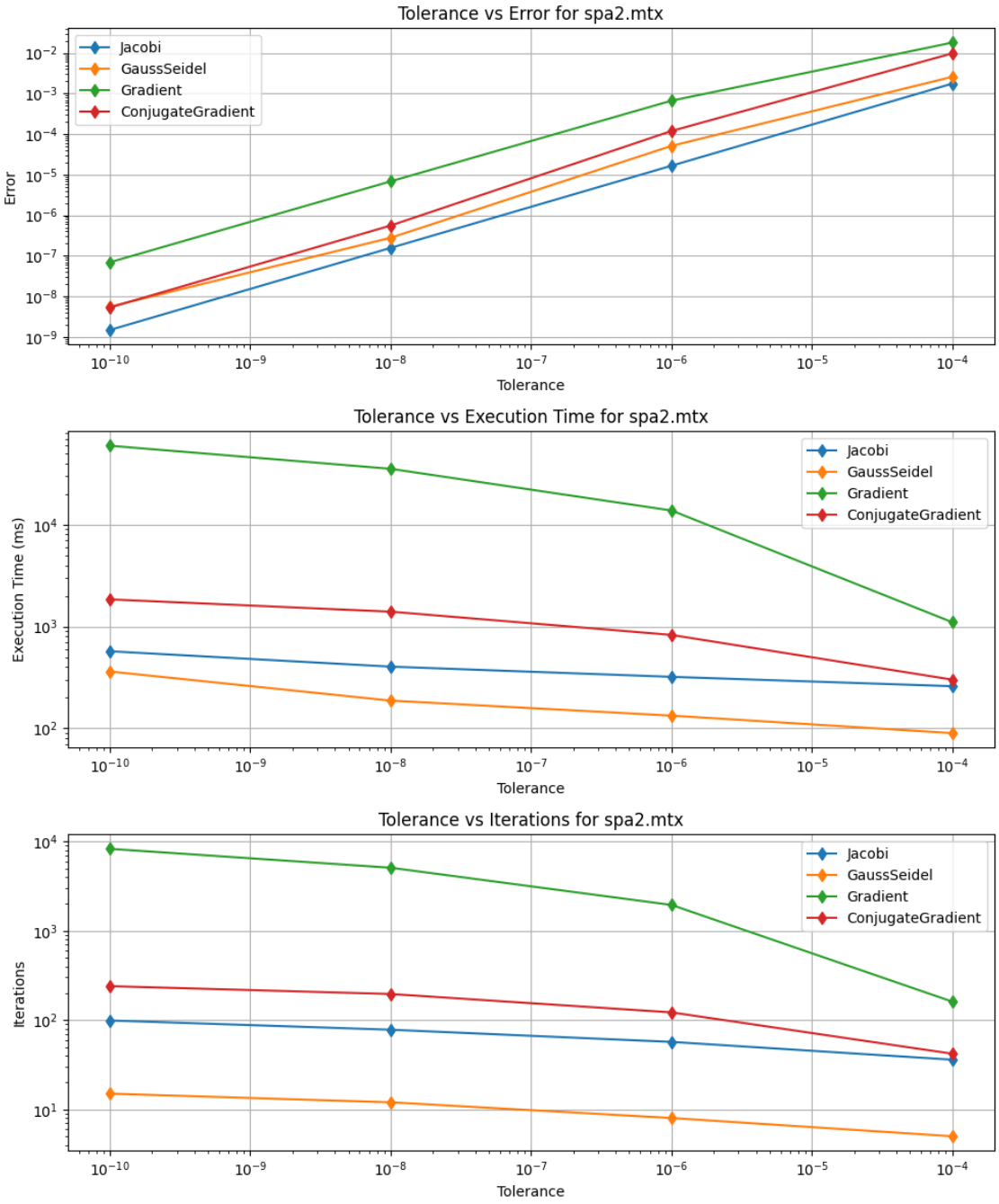
Il primo subplot descrive la relazione tra l'errore relativo (asse y) e la tolleranza (asse x), in riferimento al file *spa1.mtx*. Al suo interno è memorizzata, in forma sparsa, una matrice $A \in R^{1000\times1000}$. Ciascun metodo raggiunge la convergenza presentando errori relativi che si attestano su valori leggermente inferiori rispetto alla tolleranza associata. Pertanto, l'errore relativo e la tolleranza sono descritti da una funzione lineare, in quanto l'errore cresce/decresce linearmente al crescere/decrescere della tolleranza. Più precisamente, il metodo di Jacobi e gradiente coniugato presentano errori relativi leggermente inferiori rispetto ai metodi restanti, ma che risultano ugualmente efficienti.

Il secondo subplot descrive la relazione tra il tempo di esecuzione (asse y) dei vari metodi e la tolleranza (asse x) utilizzata in input. Tutti i metodi, eccetto il metodo del gradiente, posseggono un tempo di esecuzione pressoché costante, quasi indipendente dal valore della tolleranza, eccetto che per l'esecuzione con tolleranza pari a $1e^{-4}$ in cui il tempo di esecuzione tende a essere leggermente inferiore. Il metodo che più si discosta dagli altri è il metodo del gradiente, che possiede un tempo di esecuzione maggiore e meno costante, ma che decresce al diminuire della tolleranza. Questo comportamento rispetta le aspettative e coincide con quanto descritto precedentemente poiché il metodo del gradiente converge più lentamente muovendosi lungo la direzione del gradiente senza tenere conto delle direzioni passate.

Il terzo subplot descrive la relazione tra il numero di iterazioni (asse y) e la tolleranza (asse x). Analogamente al caso precedente, il metodo del gradiente richiede un numero di iterazioni maggiori rispetto agli altri metodi, proprio a causa della sua natura di discesa, spesso con un pattern a "zig-zag". Il metodo di Jacobi e gradiente coniugato richiedono un numero di iterazioni minori, mentre il metodo di Gauß-Seidel è il più veloce di tutti, soprattutto perché utilizza valori appena aggiornati, a differenza di Jacobi.

Considerazioni finali: Per la matrice *spa1* il metodo del gradiente risulta essere il peggiore sotto tutti e tre i punti di vista, mentre i restanti metodi risultano essere più "ottimali" in quanto raggiungono più rapidamente la convergenza e con errori relativi inferiori.

4.2 Risultati su spa2.mtx



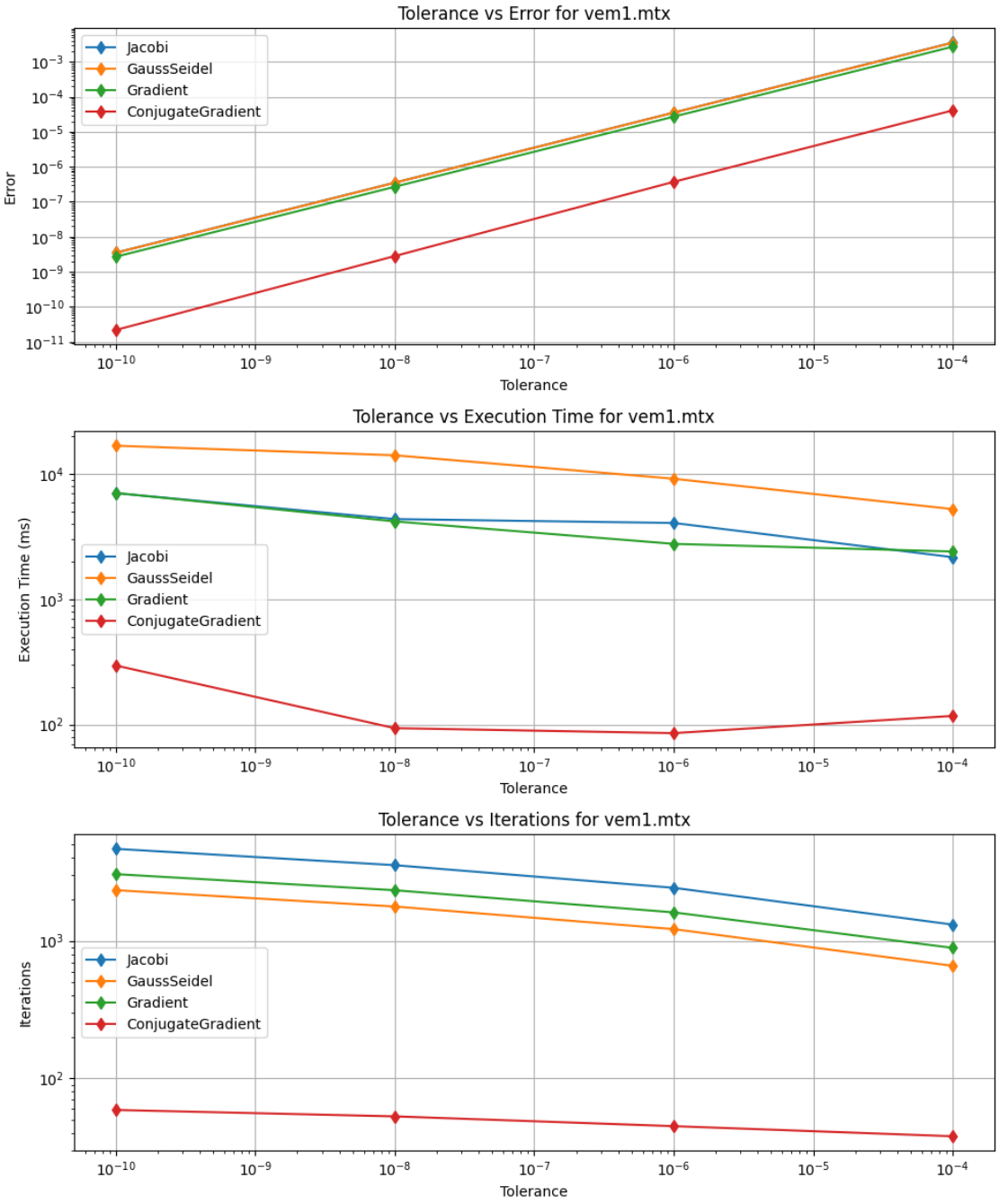
Il primo subplot descrive la relazione tra l'errore relativo (asse y) e la tolleranza (asse x), in riferimento al file *spa2.mtx*. Al suo interno è memorizzata, in forma sparsa, una matrice $A \in R^{3000\times3000}$. Ciascun metodo raggiunge la convergenza presentando errori relativi che si attestano su valori leggermente inferiori rispetto alla tolleranza associata. Pertanto, l'errore relativo e la tolleranza sono descritti da una funzione lineare, in quanto l'errore cresce/decresce linearmente al crescere/decrescere della tolleranza. Più precisamente, il metodo di Jacobi presenta errori relativi leggermente inferiori rispetto ai metodi restanti, ma che risultano ugualmente efficienti.

Il secondo subplot descrive la relazione tra il tempo di esecuzione (asse y) e la tolleranza (asse x) in input. In questo grafico è presente una distinzione più netta e visibile rispetto al medesimo subplot associato a *spa1.mtx*. È possibile osservare che il metodo del gradiente continua a essere l'algoritmo più lento. Successivamente si ha il metodo del gradiente coniugato, Jacobi e infine Gauß-Seidel che conferma di avere una struttura algoritmica tale per cui si hanno poche iterazioni (grafico successivo) e un tempo di esecuzione rapido.

Il terzo subplot descrive la relazione tra il numero di iterazioni (asse y) e la tolleranza (asse x). Rispetto al terzo subplot della matrice *spa1.mtx*, il metodo di Jacobi e del gradiente coniugato si scambiano di ruolo, dove Jacobi richiede minori iterazioni rispetto al gradiente coniugato. La situazione rimane invariata per il metodo del gradiente, il peggiore, e Gauß-Seidel, il migliore.

Considerazioni finali: Rispetto al caso precedente, Gauß-Seidel si distingue maggiormente dagli altri metodi, risultando pertanto la scelta migliore tra tutte e quattro le tecniche. Il metodo del gradiente, invece, continua a rimanere il metodo con performance poco ottimali.

4.3 Risultati su vem1.mtx



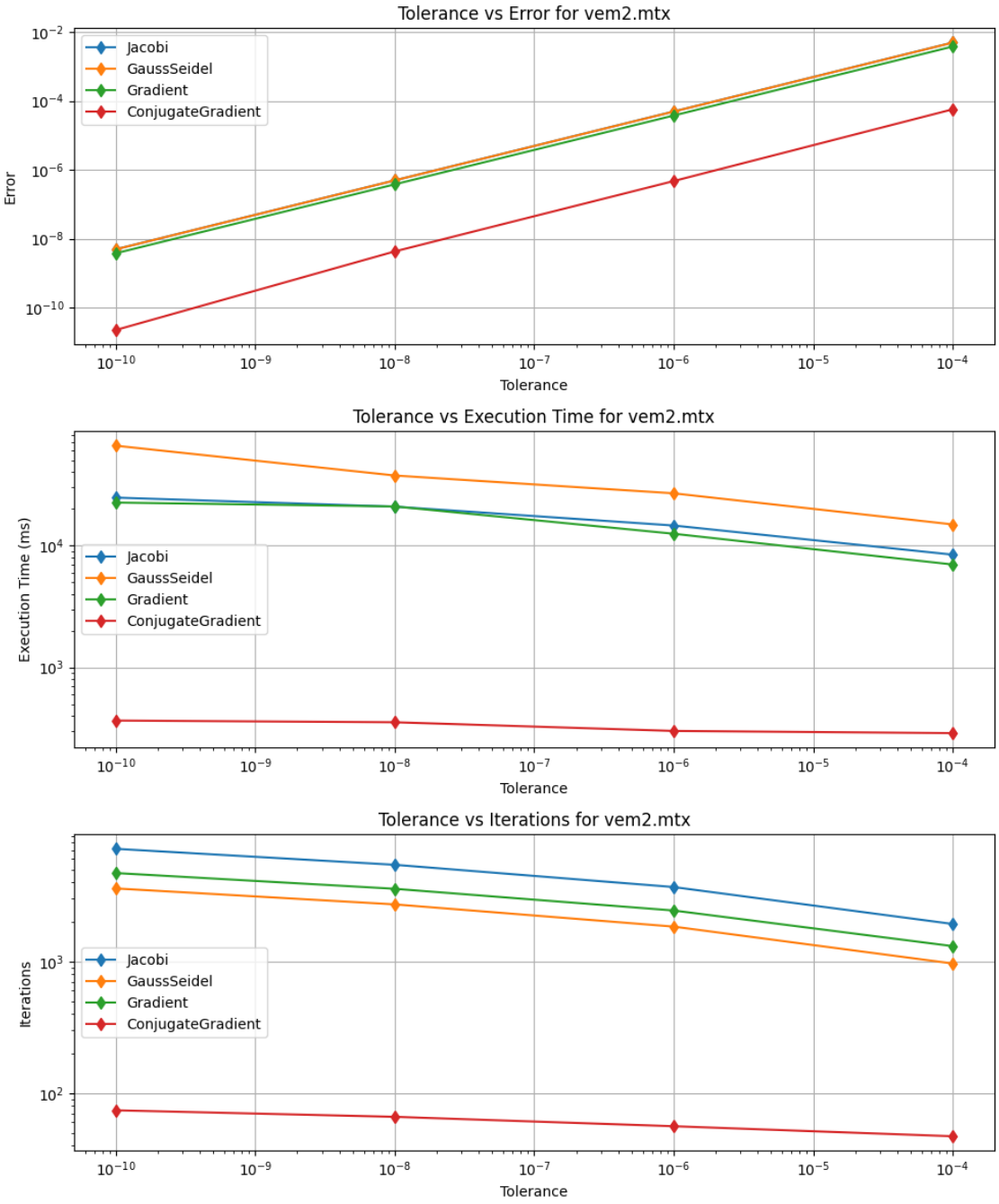
Il primo subplot descrive la relazione tra l'errore relativo (asse y) e la tolleranza (asse x), in riferimento al file *vem1.mtx*. Al suo interno è memorizzata, in forma sparsa, una matrice $A \in R^{1681 \times 1681}$. Ciascun metodo raggiunge la convergenza presentando errori relativi che si attestano su valori leggermente inferiori rispetto alla tolleranza associata. Pertanto, l'errore relativo e la tolleranza sono descritti da una funzione lineare, in quanto l'errore cresce/decresce linearmente al crescere/decrescere della tolleranza. Più precisamente, il metodo del gradiente coniugato presenta errori relativi inferiori rispetto ai metodi restanti, ma che risultano ugualmente efficienti.

Il secondo subplot descrive la relazione tra il tempo di esecuzione (asse y) e la tolleranza (asse x) in input. In questo grafico la situazione è completamente differente rispetto ai casi visti in precedenza. È possibile osservare che il metodo di Gauß-Seidel risulta essere il più lento, mentre il metodo del gradiente coniugato è il più veloce.

Il terzo subplot descrive la relazione tra il numero di iterazioni (asse y) e la tolleranza (asse x). Anche in questo subplot il metodo del gradiente coniugato risulta essere l'algoritmo che esegue meno iterazioni, mentre i restanti metodi eseguono un numero di iterazioni pressoché simile.

Considerazioni finali: Il metodo del gradiente coniugato si conferma il metodo migliore, sia come tempo di esecuzione che di errore relativo, pertanto, risulta essere la migliore scelta in caso di matrici simmetriche e definite positive come *vem1.mtx*

4.4 Risultati su vem2.mtx



Il primo subplot descrive la relazione tra l'errore relativo (asse y) e la tolleranza (asse x), in riferimento al file *vem2.mtx*. Al suo interno è memorizzata, in forma sparsa, una matrice $A \in R^{2601 \times 2601}$. Ciascun metodo raggiunge la convergenza presentando errori relativi che si attestano su valori leggermente inferiori rispetto alla tolleranza associata. Pertanto, l'errore relativo e la tolleranza sono descritti da una funzione lineare, in quanto l'errore cresce/decresce linearmente al crescere/decrescere della tolleranza. Più precisamente, il metodo del gradiente coniugato presenta errori relativi inferiori rispetto ai metodi restanti, ma che risultano ugualmente efficienti.

Il secondo subplot descrive la relazione tra il tempo di esecuzione (asse y) e la tolleranza (asse x) in input. Analogamente al caso precedente, il metodo del gradiente coniugato è il più veloce, possedendo un tempo d'esecuzione praticamente costante.

Il terzo subplot descrive la relazione tra il numero di iterazioni (asse y) e la tolleranza (asse x). Anche in questo subplot il metodo del gradiente coniugato risulta essere l'algoritmo che esegue meno iterazioni, mentre i restanti metodi eseguono un numero di iterazioni pressoché simile.

Considerazioni finali: Il metodo del gradiente coniugato si conferma il metodo migliore anche per *vem2.mtx*, sia come tempo di esecuzione che di errore relativo.

5 Conclusione

Il progetto sviluppato si poneva come obiettivo principale, oltre all'implementazione di alcuni metodi iterativi, l'esecuzione e l'analisi dei risultati ottenuti su alcune matrici fornite in input come test. I risultati hanno confermato le aspettative e le peculiarità di ciascun metodo, identificando il gradiente coniugato come il miglior algoritmo.
Quest'ultimo, infatti, rappresenta il miglior compromesso in termini di accuratezza (errore relativo) e velocità (tempo di esecuzione e numero di iterazioni) su ciascun file analizzato, in particolare sui file *vem1 e vem2*. Ciò non esclude, tuttavia, gli altri metodi, che si dimostrano validi risolutori in specifici casi d'uso, grazie alle loro ottime performance.
In conclusione, questo progetto conferma ancora una volta che non esiste un unico metodo ottimale e universale, ma che ciascun algoritmo offre prestazioni differenti a seconda del contesto di utilizzo. Ciò evidenzia l'importanza di un'analisi comparativa tra i vari metodi iterativi.