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

Metodi del calcolo scientifico

Progetto 1 Bis

Authors of the paper:
Nicolò Nicholas Zagami 829888
Pietro Bressan 852260

A.A. 2023-2024

Indice

Capitolo 1

Introduzione

1.1 Scopo della relazione

L'obiettivo della relazione è quello di implementare una libreria contenente dei solutori iterativi per sistemi lineari, limitandoci al caso di matrici *simmetriche* e definite *positive*; i metodi iterativi implementati sono i seguenti:

- metodo di Jacobi
- metodo di Gauß-Seidel
- metodo del gradiente
- metodo del gradiente coniugato

1.2 Metodi iterativi

I metodi iterativi servono alla risoluzione di sistemi lineari del tipo:

$$A\underline{x} = \underline{b} \tag{1.1}$$

il processo di risoluzione è il seguente: a partire da un vettore iniziale $\underline{x}^{(0)}$ creiamo una sequenza di vettori $\underline{x}^{(k)}$ che si avvicinano sempre di più alla soluzione esatta $\tilde{\underline{x}}$ ad ogni iterazione.

Ognuno di questi algoritmi iterativi è caratterizzato da una quantità molto importante chiamata *tolleranza*; la quale funge da criterio di arresto e serve ad determinare quando la **soluzione approssimata** $\underline{x}^{(k)}$ è sufficientemente vicine alla **soluzione** $\tilde{\underline{x}}$.

1.3 Metodi iterativi stazionari

I metodi iterativi stazionari sfruttano la *procedura di splitting*: considerata una matrice $A \in \mathbb{R}^{n\times n}$, dividiamo A in modo da ottenere due matrici P e N, le quali rappresentano le componenti di A e possono essere scritte come:

$$A = P - N \tag{1.2}$$

I **metodi iterativi stazionari** si basano su due concetti:

1. P è una matrice la cui inversa P^{-1} è facilmente calcolabile; in caso contrario non avrebbe senso la decomposizione della matrice A poiché potremmo direttamente trovare la sua inversa A^{-1}.
2. Questi metodi vengono detti *stazionari* in quanto il vettore soluzione $\underline{x}$ calcolato all'iterata $k+1$ non dipende dalla $k-esima$ iterazione.

In questa categoria rientrano il metodo di *Jacobi* e *Gauß-Seidel*.

1.3.1 Metodo di Jacobi

Caratteristiche delle matrici

Nella sezione precedente abbiamo introdotto il metodo di splitting, il quale viene utilizzato dai metodi iterativi stazionari per la decomposizione della matrice A, adiamo ora ad analizzare le caratteristiche delle matrici P e N nel metodo di *Jacobi*:

P è la diagonale di A, una matrice facilmente **invertibile**:

$$P = \begin{pmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{pmatrix} \tag{1.3}$$

N è la matrice che si ottiene **mettendo a 0 le entrate sulla diagonale** di A e **cambiando il segno** di tutte le altre entrate:

$$N = \begin{pmatrix} 0 & -a_{1,2} & \cdots & -a_{1,n} \\ -a_{2,1} & 0 & \cdots & -a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n,1} & -a_{n,2} & \cdots & 0 \end{pmatrix} \tag{1.4}$$

Convergenza

Per verificare la convergenza del metodo di *Jacobi* è possibile fare riferimento ai coefficienti della matrice A:

Definizione 1 *Una matrice $A \in \mathbb{R}^{n\times n}$ è a dominanza diagonale stretta per righe se vale*

$$|a_{i,i}| > \sum_{j=1, j\neq i}^{n} |a_{i,j}| \tag{1.5}$$

Complessità computazionale

Per analizzare il **costo** della complessità computazione andiamo ad analizzare la $k-esima$ iterata del metodo di *Jacobi*, la quale si presenta nel seguente modo:

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha P^{-1} \underline{r}^{(k)} \tag{1.6}$$

1. Il costo di una matrice inversa solitamente è pari a $\mathcal{O}(n^2)$, trattandosi però di una **matrice diagonale** il costo si riduce a $\mathcal{O}(n)$
2. Il calcolo del residuo si riduce ad un'operazione **matrice per vettore**, concludiamo quindi che si tratta di un costo pari a $\mathcal{O}(n^2)$.

3. Il prodotto tra la matrice inversa e il residuo, come la somma finale hanno un costo pari a $\mathcal{O}(n)$.

La complessità finale risulta essere $\mathcal{O}(n^2)$.

1.3.2 Metodo di Gauß-Seidel

Il metodo di *Gauß-Seidel* è una variante del metodo di *Jacobi*, vediamo in cosa si differenzia.

Caratteristiche delle matrici

P è la parte triangolare inferiore di A:

$$P = \begin{pmatrix} a_{1,1} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \tag{1.7}$$

N è la parte triangolare superiore di A con **segno opposto** e **senza termini** sulla diagonale principale.

$$N = \begin{pmatrix} 0 & -a_{1,2} & \cdots & -a_{1,n} \\ 0 & 0 & \cdots & -a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix} \tag{1.8}$$

In questo caso P non è più una matrice diagonale, il calcolo della sua inversa risulta essere oneroso e poco stabile (si possono introdurre errori di approssimazione). Calcoliamo quindi $P^{-1}\underline{r}^{(k)}$ riscrivendo l'espressione come una nuovo sistema di equazioni lineari del tipo $P\underline{y} = \underline{r}^{(k)}$, che per via della struttura di P più semplice da risolvere. Reinterpretando l'espressione riusciamo a risparmiarci il calcolo della inversa di P.

Convergenza

La convergenza in questo caso è analoga a quella del metodo di *Jacobi* (1.3.1).

Complessità computazionale

Come abbiamo visto nel metodo di *Jacobi* il calcolo del residuo ha un costo di $\mathcal{O}(n^2)$, per quanto riguarda il sistema lineare esso può essere risolto con il **metodo di risoluzione in avanti** (essendo una matrice triangolare), il quale ha una costo computazionale pari a $\mathcal{O}(n^2)$.

Come prima, il prodotto tra le due operazioni precedenti e la somma finale hanno un costo di $\mathcal{O}(n)$, quindi il costo complessivo è pari a $\mathcal{O}(n^2)$.

1.4 Metodi iterativi non stazionari

Se nei **metodi iterativi stazionari** costruiamo sequenze di vettori $\underline{x}^{(k)}$ come:

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha P^{-1}\underline{x}^{(k)} \tag{1.9}$$

Nei metodi **NON stazionari** dobbiamo considerare un fattore in più, ovvero il coefficiente α: il quale è fisso nei metodi *stazionari* per ogni iterazione, in questo caso invece dipende da k e cambia durante le iterazioni.

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha_k P^{-1}\underline{r}^{(k)} \tag{1.10}$$

In questa categoria rientrano il metodo del *gradiente* e il metodo del *gradiente coniugato*.

1.4.1 Metodo del gradiente

Concetti chiave

Il metodo del gradiente si basa su un'idea diversa rispetto ai metodi precedentemente illustrati: partiamo dalla definizione della funzione ϕ.

Definizione 2 *Supponendo che la matrice A sia **simmetrica** e **positiva** la funzione ϕ così definita:*

$$\phi(\underline{\mathbf{y}}) = \frac{1}{2}\underline{\mathbf{y}}^T A\underline{\mathbf{y}} + \underline{\mathbf{b}}^T\underline{\mathbf{y}} \tag{1.11}$$

è una forma quadratica della variabile vettoriale $\underline{y}$, la quale è analoga alla forma multidimensionale di una parabola con concavità verso l'alto.

Di conseguenza risolviamo il sistema lineare trovando il *punto minimo* della funzione ϕ attraverso il calcolo del **gradiente**:

$$\begin{aligned} \nabla\phi(\underline{y}) &= \frac{1}{2}(A^t + A)\underline{y} - \underline{b} \\ \nabla\phi(\underline{y}) &= A\underline{y} - \underline{b} \end{aligned} \tag{1.12}$$

Proseguo cercando il punto minimo della funzione ϕ, il che equivale a risolvere $A\underline{y} - \underline{b} = 0$.

Passo ricorsivo

La successione di vettori $\underline{x}^{(k)}$ può essere visto come un "percorso" verso il punto minimo della funzione ϕ.

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha_k\underline{d}^{(k)} \tag{1.13}$$

Tuttavia sapendo che il gradiente esprime la **direzione di massima crescita** consideriamo il suo opposto $-\nabla\phi(\underline{y})$ in quanto stiamo cercando il punto minimo; questa quantità coincide con $\underline{r}^{(k)}$, quindi definisco il passo ricorsivo come:

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha_k\underline{r}^{(k)} \tag{1.14}$$

dove α_k è:

$$\alpha_k = \frac{(\underline{\mathbf{r}}^{(k)})^t \underline{\mathbf{r}}^{(k)}}{(\underline{\mathbf{r}}^{(k)})^t \mathbf{A} \underline{\mathbf{r}}^{(k)}} \tag{1.15}$$

Convergenza

Per valutare la convergenza del metodo del gradiente consideriamo i suoi autovalori, più l'autovalore minimo λ_{min} è simile all'autovalore massimo λ_{max} e meno iterazioni saranno necessarie per la convergenza del metodo.

Costo computazionale

Con riferimento ai metodi precedenti sappiamo che il residuo ha un costo pari a $\mathcal{O}(n^2)$, α_k ha un costo complessivo di $\mathcal{O}(n^2)$ dato da un prodotto vettore matrice $Ar^{(k)}$.

Il costo complessivo risulta essere $\mathcal{O}(n^2)$.

1.4.2 Metodo del gradiente coniugato

Il metodo del gradiente coniugato può essere visto come un miglioramento del metodo del gradiente, in quanto riesce a porre rimedio alla convergenza a zig zag del metodo del gradiente.

Concetti chiave

Considerando le proprietà della funzione quadratica ϕ e della matrice descritta nel metodo del gradiente(1.4.1) allora diremo che:

$$\phi(\underline{x}^{(k)}) \leq \phi(\underline{x}^{(k)} + \lambda \underline{d}) \qquad \forall \lambda \in \mathbb{R} \tag{1.16}$$

Quindi l'alterazione della direzione $\underline{d}$ con un qualsiasi λ non permette di avvicinarsi al minimo della funzione ϕ.

Quindi con il metodo del gradiente coniugato introduciamo un nuovo concetto: il vettore $\underline{x}^{(k)}$ se ottimale non deve essere alterato lungo la direzione $\underline{d}$; poiché questo porterebbe al famoso *effetto zig zag*.

Definizione 3 *Un vettore è ottimale rispetto ad una direzione d quando:*

$$\underline{d} \underline{r}^{(k)} = 0 \tag{1.17}$$

Passo ricorsivo

Come nel metodo del gradiente, il passo ricorsivo è uguale a:

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha_k \underline{r}^{(k)} \tag{1.18}$$

Tuttavia il vettore $\underline{x}^{(k+1)}$ è ottimale rispetto alla direzione $\underline{r}^{(k+1)}$, ma $\underline{x}^{(k+2)}$ non lo è rispetto alla direzione $\underline{r}^{(k)}$, questo porta all'effetto *zig zag*; nel metodo del gradiente coniugato per sopperire a questo problema procediamo nel seguente modo:

calcoliamo α_k come:

$$\alpha_k = \frac{(\underline{\mathbf{r}}^{(k)})^t \underline{\mathbf{r}}^{(k)}}{(\underline{\mathbf{r}}^{(k)})^t \mathbf{A} \underline{\mathbf{r}}^{(k)}} \tag{1.19}$$

Una volta trovata la posizione, troviamo la nuova direzione con cui verrà costruita la nuova posizione:

$$\beta_k := \frac{(\underline{d}^{(k)})^t A \underline{r}^{(k+1)}}{(\underline{d}^{(k)})^t A \underline{d}^{(k)}} \tag{1.20}$$

Convergenza

Nel metodo del gradiente coniugato il numero di iterazioni è al più n, in quanto viene mantenuto il vettore ottimale $\underline{x}^{(k)}$ rispetto alle direzioni precedenti; abbattendo di fatto il numero di iterazioni necessarie.

Costo computazionale

Nel metodo del gradiente coniugato ci sono diversi prodotti tra matrici e vettori i quali richiedono un costo di $\mathcal{O}(n^2)$; il quale risulta essere anche il costo totale.

Capitolo 2

Implementazione

La libreria è stata realizzata utilizzando il linguaggio Python (versione 3.8.8). La scelta è stata fatta considerando il già diffuso impiego di Python nel mondo del calcolo scientifico, oltre ad alcune funzionalità proprie del linguaggio che permettono di ottenere un algoritmo chiaro seppur conciso grazie ad un forte riutilizzo del codice.

2.1 Scopo della libreria

L'obiettivo della libreria è quello di implementare i 4 metodi iterativi che sono oggetto di questa analisi, facendo però attenzione a due aspetti fondamentali: efficienza in termini di operazioni svolte e di spazio, ed efficacia in termini di usabilità (l'ipotetico utilizzatore della libreria non deve essere solo in grado di eseguire i metodi ma deve anche poter avere controllo sull'esecuzione attraverso gli argomenti passati in input e i valori ritornati in output).

2.2 Tecnologie coinvolte

Per aiutarsi nella gestione delle matrici bidimensionali sono stati necessari i seguenti componenti:

- La libreria *Numpy* per la manipolazione di oggetti che rappresentano matrici e vettori.
- La libreria *Scipy* per la lettura di file in formato Matrix Market e per alcune operazioni ottimizzate sulle matrici.

2.3 Struttura

La libreria è composta da un unico modulo chiamato `linearsystemsolver`, al cui interno vengono definite le classi fondamentali per la risoluzione di sistemi lineari.

2.3.1 LinearSolver

rappresenta una generica procedura di risoluzione (Stazionaria o Non Stazionaria). Viene utilizzata come classe base da cui derivano le implementazioni vere e proprie delle due procedure. L'unica funzione membro della classe è

```
create_matrix_from_file(filename)
```

che, preso in input il percorso di un file in formato Matrix Market, ritorna un'oggetto ndarray rappresentante una matrice; la lettura viene fatta sfruttando il metodo `mmread`.

Dal momento che lo scopo della classe è solo quello di farci risparmiare codice, definendo una funzione utile per tutte le procedure, essa non è visibile all'esterno (privata).

2.3.2 StationarySystemSolver

`StationarySystemSolver` incapsula la logica dei metodi stazionari per la risoluzione di sistemi lineari: *Jacobi* e *Gauß-Seidel*.

Attraverso i seguenti metodi:

```
solve_with_jacobi(self, matrix, target=None, maxIters=20000, tol=1e-4)
```

```
solve_with_gaub_seidel(self, matrix, target=None, max_iters=20000, tol=1e-4)
```

Alcune considerazioni sul codice:

- Python calcola iterativamente il vettore risultato associato con la matrice fornita in ingresso.
- Di default i metodi sono configurati per utilizzare come vettore dei **termini noti** il **vettore nullo**, ma è possibile passare in input un vettore aggiuntivo target che verrà utilizzato come membro di destra nella risoluzione del sistema.
- Sempre in modo facoltativo è possibile controllare il numero massimo di iterazioni che il programma può eseguire prima di arrestarsi e ritornare la soluzione attraverso l'argomento `max_iters`. lo stesso può essere fatto per la tolleranza tramite il parametro `tol`.

I metodi ritornano un valore intero che rappresenta il numero di iterazioni totali che l'algoritmo ha svolto, e un valore booleano che indica se l'algoritmo stato fermato ($True$) o meno ($False$) prima di aver raggiunto il valore di tollerance richiesto:

```
return counter, True/False
```

Questa scelta nasce dalla necessità di ottenere questo dato in fase di rielaborazione dei dati e successivo plot dei grafici dei risultati.

Dal punto di vista implementativo i due metodi sopra citati seguono lo stesso procedimento per aggiornare il vettore risultato durante il passo iterativo (1.6)

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + P^{-1}\underline{r}^{(k)} \tag{2.1}$$

Ciò che cambia effettivamente è solo l'elemento $P^{-1}\underline{r}^{(k)}$, che nel caso di Jacobi corrisponde ad eseguire

$$P^{-1}\underline{\mathbf{r}}^{(k)} = \begin{bmatrix} \frac{1}{a_{1,1}} & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{2,2}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{a_{n,n}} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} \frac{r_1}{a_{1,1}} \\ \frac{r_2}{a_{2,2}} \\ \vdots \\ \frac{r_n}{a_{n,n}} \end{bmatrix}, \tag{2.2}$$

Cioè, corrisponde ad un'operazione di moltiplicazione punto a punto tra il vettore residuo $\underline{r}^{(k)}$ e il vettore individuato estraendo la diagonale da P^{-1}. Mentre per il metodo di Gauß-Seidel si tratta di trovare il vettore $\underline{y} = P^{-1}\underline{r}^{(k)}$ risolvendo il sistema triangolare inferiore $P\underline{y} = \underline{r}^{(k)}$. Dove P ricordiamo essere come illustrato a 1.7.

Proprio per supportare la risoluzione di Gauß-Seidel la classe è stata dotata della funzione

```
def __forward_solve_triang(self, matrix, b):
```

che esegue la tecnica della Forward-Substitution per risolvere un sistema lineare caratterizzato da una matrice triangolare inferiore e ritorna il risultato.

Una volta capito che la differenza tra i metodi risiede nell'operazione di aggiornamento del vettore risultato, risulta pratico dotare la classe di un metodo che racchiude tutta la maggior parte della procedura iterativa che invece, come abbiamo visto, è praticamente la stessa. A tal proposito è stata creata la funzione:

```
__stationary_solve(self, matrix, factor, operation, target=None, maxIters=20000, tol=1e-4)
```

Quando si va a chiamare i metodi relativi a Jacobi e Gauss-Seidel, questi altro non fanno che invocare il metodo specializzandone il comportamento. Attraverso i parametri factor e operation possiamo controllare come verrà valutato l'elemento $P^{-1}\underline{r}^{(k)}$. Ciò è ancora una volta è reso possibile dal linguaggio utilizzato, che offre la possibilità di passare funzioni come argomenti di altre funzioni (**First Class Functions**).

2.3.3 NonStationarySystemSolver

Questa classe segue lo stesso principio della precedente, ma racchiude la logica dei metodi del *Gradiente* e del *Gradiente Coniugato*. Anche in questo caso ritroviamo due funzioni principali richiamabili dall'esterno:

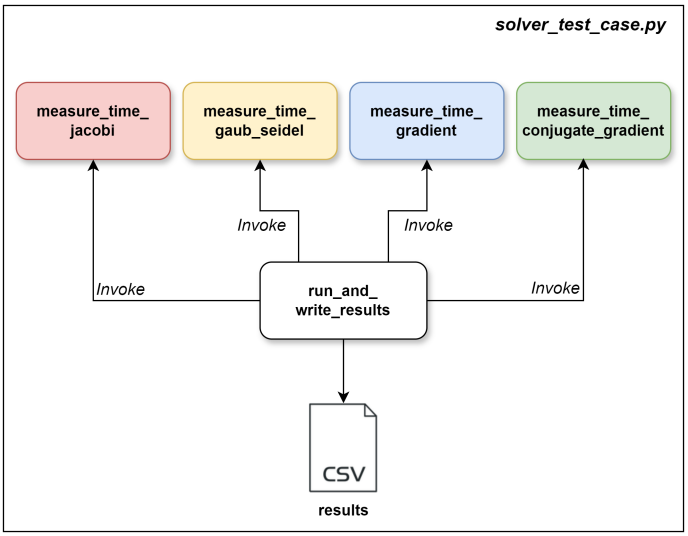
```
solve_with_gradient(self, matrix, target=None, maxIters=20000, tol=1e-4)
```

```
solve_with_conjugate_gradient(self, matrix, target=None, maxIters=20000, tol=1e-4)
```

dove i parametri di ingresso della funzione così come quelli ritornati in uscita sono gli stessi. Quindi l'utilizzatore anche in questo caso ha modo di controllare i parametri di esecuzione dei metodi, e ha modo di osservarne il comportamento. Seguendo questa architettura l'intero modulo si presenta come uno strumento standardizzato, dove le funzioni eseguibili presentano tutte la stessa firma (interfaccia), rendendone ancora più semplice e intuitivo l'utilizzo.

A differenza dei **metodi stazionari** però, non è possibile sfruttare delle caratteristiche comuni nei due metodi per ridurre le linee di codice: questo lo si deve al fatto che il *Gradiente Coniugato* non si presenta come una procedura non stazionaria alternativa al *Gradiente*, bensì si tratta di una miglioria algoritmo originale che richiede delle operazioni aggiuntive rispetto al passo iterativo (1.14).

2.4 Esecuzione dei test



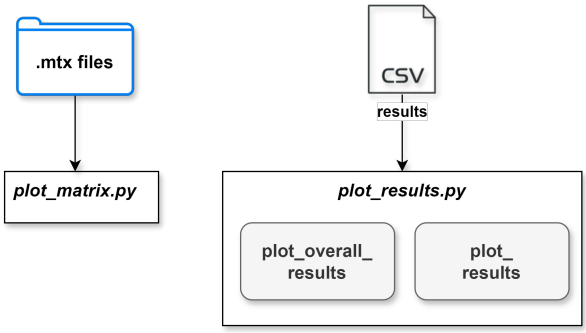
Per l'esecuzione dei test è stato creato un file `solver_test_case.py` il quale contiene una serie di funzioni utili alla misurazione dei tempi di esecuzione dei metodi e all'estrapolazione delle iterazioni necessarie alla convergenza; i nomi seguono la seguente struttura: `measure_time_METHOD_NAME`.

Ognuno di questi metodi viene invocato da un altro metodo chiamato `run_and_write_results` che produrrà un file `resuls.csv` con i dati necessari alla creazione dei grafici.

	Metodo (1)	Matrice (2)	Tolleranza (3)	Tempo (4)	Iterazioni (5)
1	Metodo	Matrice	Tolleranza	Tempo	Iterazioni
2	Jacobi	data/spa1.mtx	0.0001	0.01758079999126494	131
3	Jacobi	data/spa1.mtx	1e-06	0.026286000153049827	198
4	Jacobi	data/spa1.mtx	1e-08	0.03415660001337528	264
5	Jacobi	data/spa1.mtx	1e-10	0.042638600105419755	330
6	Gaub Seidel	data/spa1.mtx	0.0001	0.3326057998929173	12
7	Gaub Seidel	data/spa1.mtx	1e-06	0.5453755999915302	20
8	Gaub Seidel	data/spa1.mtx	1e-08	0.7360239999834448	27
9	Gaub Seidel	data/spa1.mtx	1e-10	0.9438264998607337	35
10	Gradiente	data/spa1.mtx	0.0001	1.587161700008437	6211
11	Gradiente	data/spa1.mtx	1e-06	2.7947269999422133	10907
12	Gradiente	data/spa1.mtx	1e-08	3.9377824999392033	15607
13	Gradiente	data/spa1.mtx	1e-10	5.036556999897584	20305
14	Gradiente coniugato	data/spa1.mtx	0.0001	0.07801700010895729	127
15	Gradiente coniugato	data/spa1.mtx	1e-06	0.1032356999348849	168
16	Gradiente coniugato	data/spa1.mtx	1e-08	0.12217430002056062	191
17	Gradiente coniugato	data/spa1.mtx	1e-10	0.13031999999657273	214

Figura 2.1: Struttura del file csv prodotto

2.5 Produzione dei grafici



Per produrre i grafici dei risultati e delle matrici sono stati utilizzati due file diversi `plot_matrix.py` e `plot_results.py`, il primo ci permette di visualizzare la composizione delle matrici attraverso il metodo `spy`.

Per quanto riguarda la produzione dei grafici dei risultati ci basiamo sul file csv appena prodotto, e abbiamo creato due metodi:

- `plot_results`: produce i grafici di ogni metodo per quanto riguarda tempi di esecuzione e numero di iterazioni
- `plot_overall_results`: produce due grafici aggregati per metodo e tolleranza, il primo sui tempi di esecuzione e il secondo sulle iterazioni

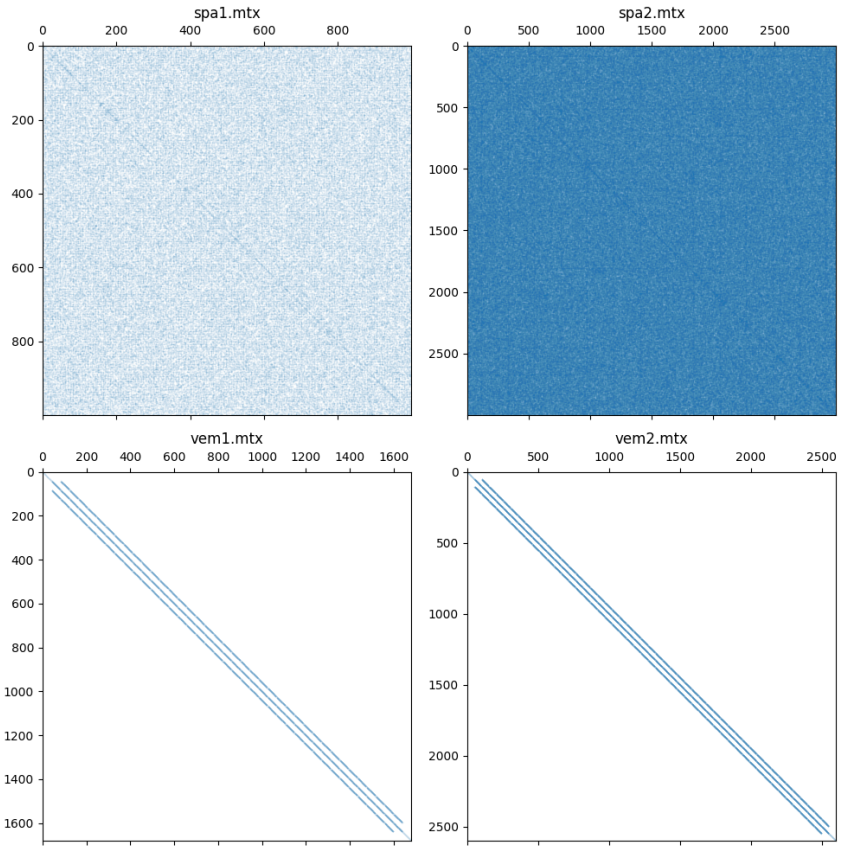
NB: questi metodi producono dei file .png sotto ad uno specifico percorso (../results).

Capitolo 3

Analisi dei risultati

3.1 Composizione delle matrici

Le strutture delle matrici utilizzate si presentano nel seguente modo:

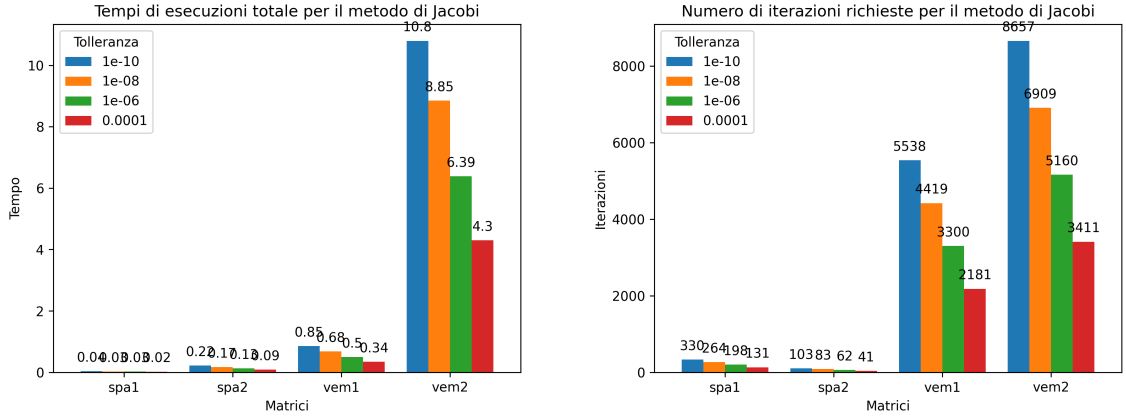


Matrice	Grandezza	Numero di condizionamento
spa1.mtx	1000x1000	2048.153813860841
spa2.mtx	3000x3000	1411.9678004487214
vem1.mtx	1681x1681	324.6439272932251
vem2.mtx	2601x2601	507.0222476111873

Tabella 3.1: Caratteristiche delle matrici usate nella relazione

Abbiamo perciò due matrici dense e due matrici a prevalenza diagonale.

3.2 Metodo di Jacobi



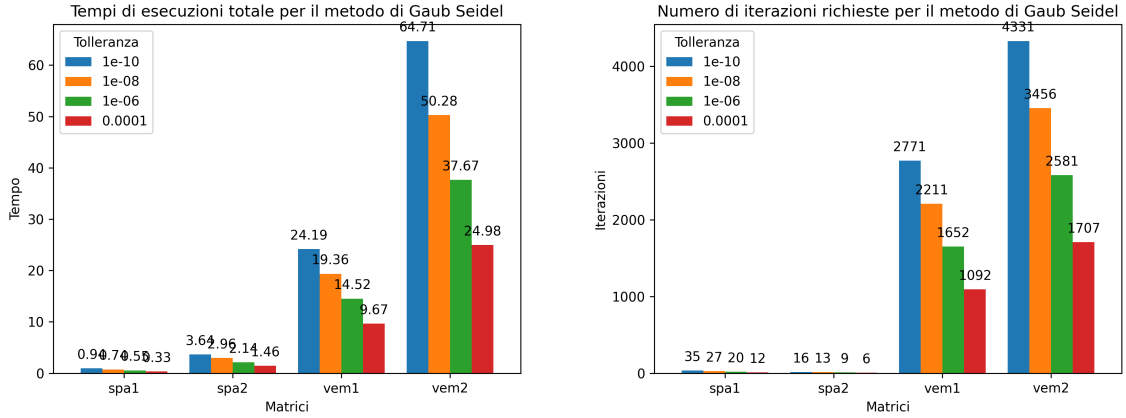
(a) Tempo di esecuzione

(b) Iterazioni

Figura 3.1: Risultati per il metodo di Jacobi

Come riportato dai grafici notiamo che i tempi di esecuzione sono più alti quando andiamo a considerare le matrici VEM; lo stesso succede con il numero di iterazioni dove però è possibile verificare una differenza molto netta.

3.3 Metodo di Gauß-Seidel



(a) Tempo di esecuzione

(b) Iterazioni

Figura 3.2: Risultati per il metodo di Gaub-Seidel

Anche nel metodo di *Gauß-Seidel* possiamo apprezzare una notevole differenza nei tempi di esecuzione: i quali crescono notevolmente nelle matrici VEM.

Le iterazioni invece rimangono elevate sempre con riferimento alle matrici VEM, tuttavia risultano essere meno se confrontate con il metodo di *Jacobi* (3.1b)

3.3.1 Jacobi vs Gauß-Seidel

Apprese quali sono le evidenze in termini di tempi di esecuzione e di numero di esecuzione per entrambi i metodi andiamo ad analizzare quali sono le implicazioni legate ai risultati.

La principale differenza la possiamo trovare nei componenti dei due metodi, infatti per quanto il passo iterativo possa essere ricondotto alla stessa formula (1.6) nel metodo di *Gauß-Seidel* utilizziamo il metodo di sostituzione in avanti per la risoluzione del sistema lineare, dato che invertire la matrice sarebbe troppo costoso.

L'iterazione del metodo di Jacobi:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{i,j} x_j^{(k)} \right) \tag{3.1}$$

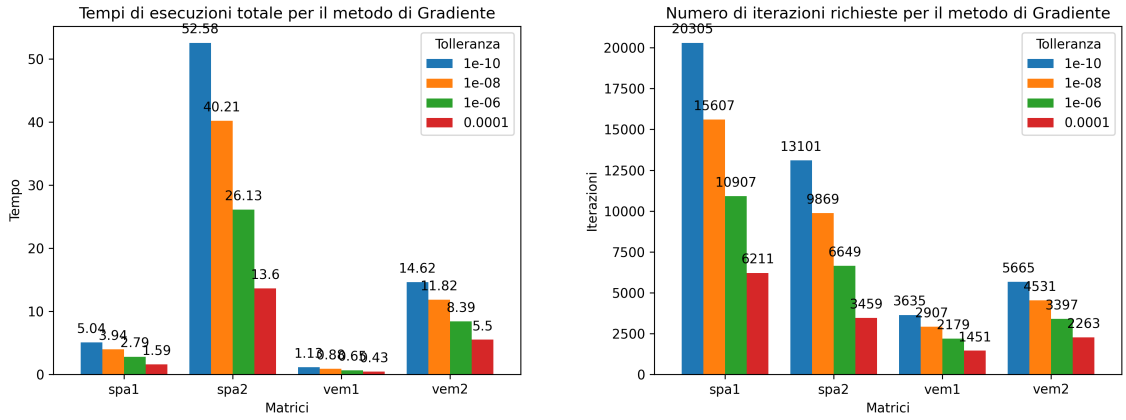
L'iterazione del metodo di Gauß-Seidel in componenti:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right) \tag{3.2}$$

In entrambi i casi la complessità è riconducibile a $\mathcal{O}(n^2)$, tuttavia il metodo di Jacobi per calcolare la soluzione $x^{(k+1)}$ interviene sulle componenti dell'iterata $x^{(k)}$ sfruttando quindi informazioni già ricavate precedentemente; mentre il metodo di *Gauß-Seidel* sfrutta anche le componenti dell'iterata corrente $x^{(k+1)}$.

Questo fattore porta il metodo di *Gauß-Seidel* ad avere un **numero minore di iterazioni richieste** per la convergenza ma **tempi di esecuzione nettamente maggiori**; il cui motivo può essere ricondotto alle operazioni aritmetiche più onerose.

3.4 Metodo del gradiente



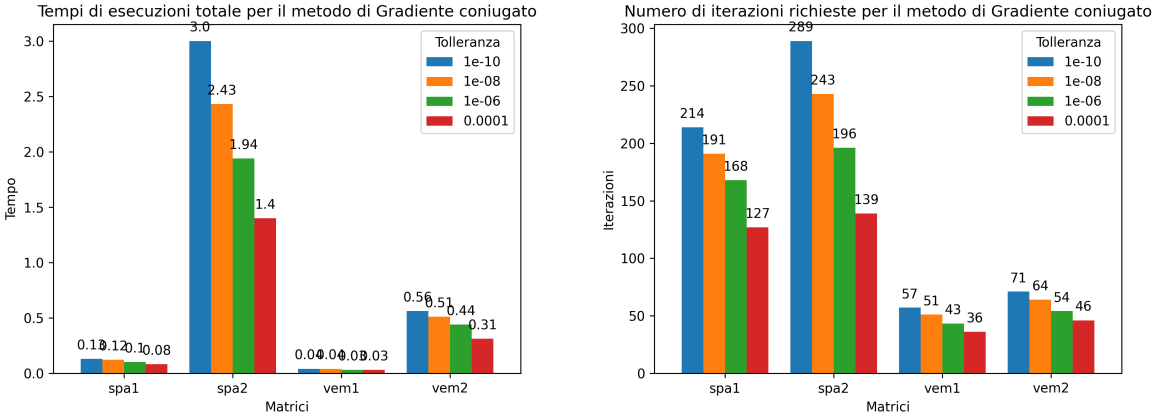
(a) Tempo di esecuzione

(b) Iterazioni

Figura 3.3: Risultati per il metodo del gradiente

Nel metodo del gradiente, contrariamente ai metodi iterativi stazionari, sono le matrici SPA a far crescere i tempi di esecuzione. A supporto dell'affermazione precedente è possibile apprezzare un elevato numero di iterazioni necessarie quando si considerano matrici SPA.

3.5 Metodo del gradiente coniugato



(a) Tempo di esecuzione

(b) Iterazioni

Figura 3.4: Risultati per il metodo del gradiente coniugato

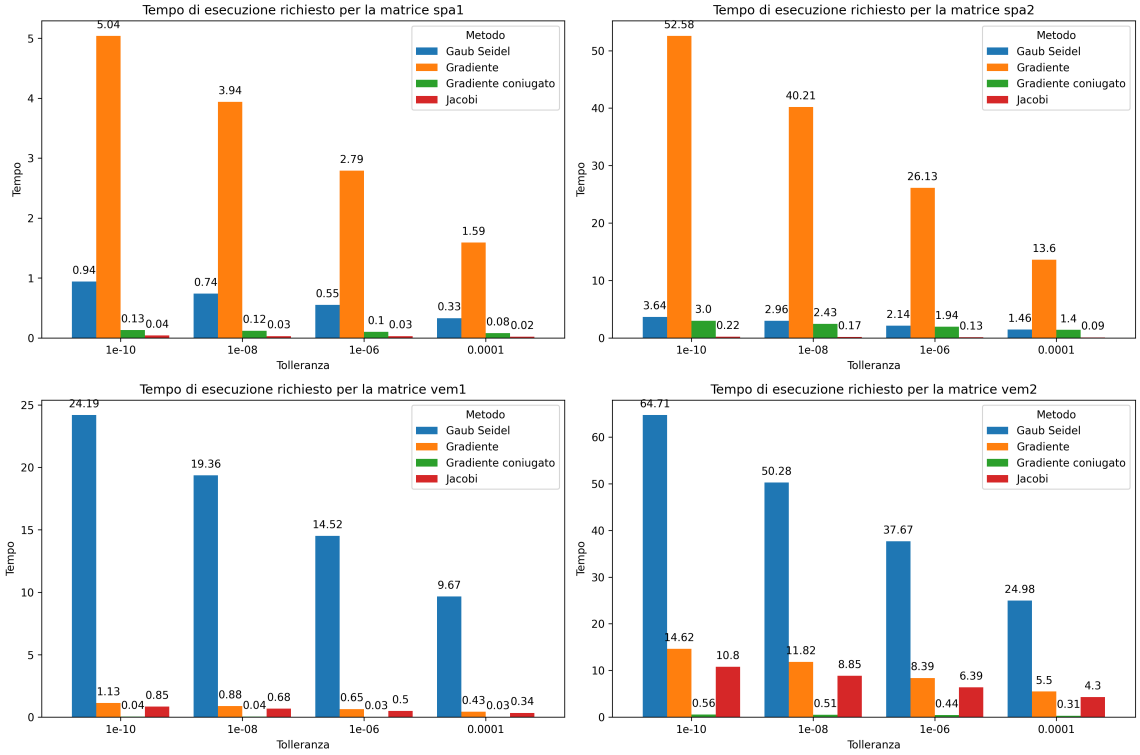
Anche nel metodo del gradiente coniugato troviamo una situazione analoga a quella precedente, dove le matrici SPA alzano notevolmente i tempi di esecuzione e il numero di iterazioni.

3.5.1 Gradiente vs Gradiente coniugato

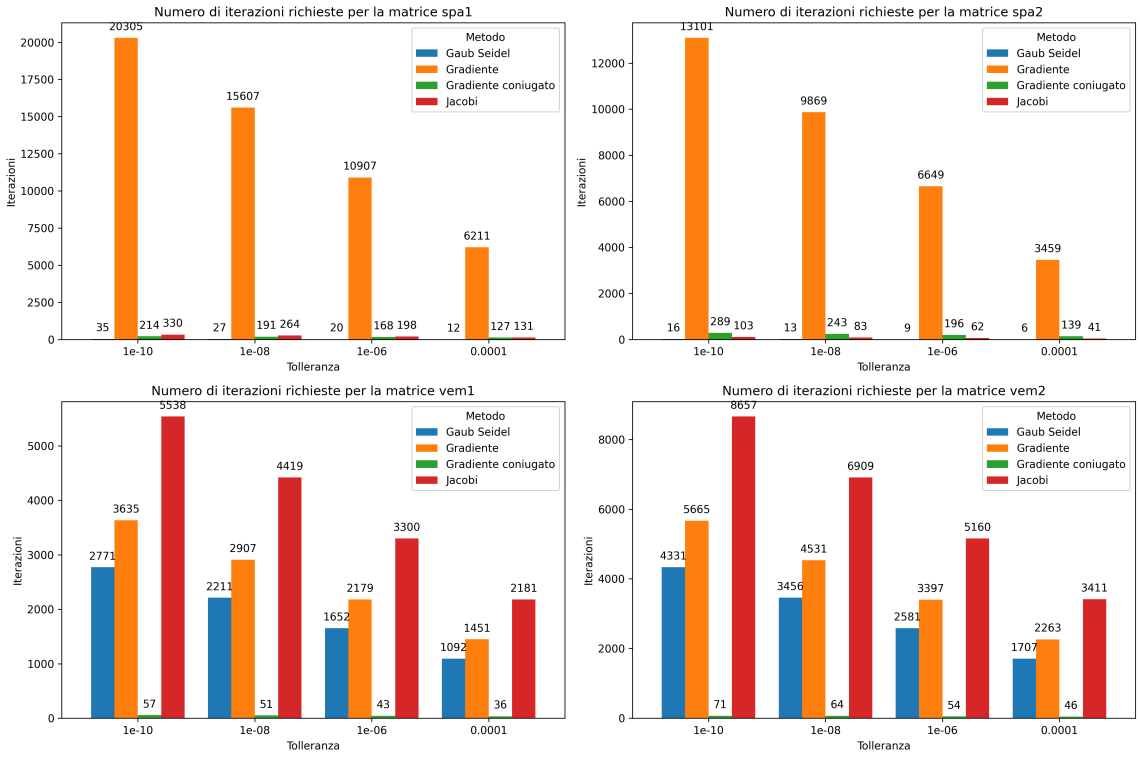
I dati raccolti indicano che sia in termini di **tempo** che di **iterazioni** il *metodo del gradiente* risulta essere il peggiore; quanto detto risulta essere scontato in quanto il metodo del *gradiente coniugato* rappresenta di fatto una versione migliorata del *metodo del gradiente*.

Infatti mantenendo il vettore ottimale $\underline{x}^{(k)}$ rispetto alle direzioni precedenti il numero di iterazioni necessarie alla convergenza viene abbattuto; lo stesso ragionamento avviene nei tempi di esecuzione: infatti sfruttando direzioni "coniugate" e non già esplorate abbattiamo i tempi di esecuzione del metodo.

3.6 Considerazioni generali



- **Grandezza delle matrici**: I tempi di esecuzione in tutti i metodi analizzati tendono a crescere al crescere della grandezza della matrice; lo stesso concetto non vale però per il numero di iterazioni.



- **Densità delle matrici**: Il numero di iterazioni non sembra influenzato dalla densità delle matrici,

tranne nel *metodo del Gradiente*; notiamo inoltre un numero di iterazioni più elevate quando consideriamo matrici a prevalenza diagonale in tutti gli altri metodi.

- **Efficienza temporale**: Il metodo *Gradiente Coniugato* emerge chiaramente come il metodo più efficiente in termini di tempo di esecuzione per tutte le matrici e tolleranze.

  Il metodo del *Gradiente* e di *Gauß-Seidel* invece sono i peggiori in termini di tempo ma in due tipi di matrici diverse; infatti il metodo del gradiente peggiora in modo drastico in presenza di matrici con densità maggiore (SPA) e meno con matrici a dominanza diagonale (VEM)

- **Numero di iterazioni**: anche in questo caso il metodo del *Gradiente Coniugato* eccelle con entrambi i tipi di matrici; il metodo del gradiente invece registra un numero elevato di iterazioni in presenza di matrici SPA.
  Il metodo di *Jacobi* peggiore in presenza di matrici VEM mantenendo però un numero più basso di iterazioni rispetto al metodo del *Gradiente*.

Capitolo 4

Conclusioni

Il presente lavoro ha affrontato l'implementazione e l'analisi di vari metodi iterativi per la risoluzione di sistemi lineari con matrici simmetriche e definite positive. Gli algoritmi esaminati includono il metodo di *Jacobi*, il metodo di *Gauß-Seidel*, il metodo del *gradiente* e il metodo del *gradiente coniugato*.

l'obiettivo principale era confrontare l'efficienza e l'efficacia di questi metodi, valutandone le prestazioni sia in termini di tempo di esecuzione sia di numero di iterazioni necessarie per raggiungere la convergenza. Dalle analisi emerge che il metodo del *gradiente coniugato* si distingue come il più **efficiente**, presentando **tempi di esecuzione inferiori** e un numero minore di **iterazioni** rispetto agli altri metodi; al contrario, il metodo del *gradiente* e il metodo di *Gauß-Seidel* risultano i **meno performanti** in termini di tempo di esecuzione, ma in contesti diversi.

In definitiva, il metodo del *gradiente coniugato* emerge come **il più adatto** per la risoluzione efficiente di sistemi lineari con le caratteristiche studiate, offrendo un compromesso ottimale tra tempo di esecuzione e numero di iterazioni.

Questo lavoro contribuisce a una migliore comprensione delle dinamiche dei metodi iterativi e fornisce una guida pratica per la scelta dell'algoritmo più appropriato in base alle specifiche caratteristiche delle matrici coinvolte.

Bibliografia

[1] Università degli Studi di Milano-Bicocca. *Note di Calcolo Scientifico*. 2024. URL: https://elearning.unimib.it/pluginfile.php/1655980/mod_resource/content/1/Note_calcolo_scientifico.pdf.