Metodi iterativi - Progetto 3

Bianca Stan 816045

July 11, 2020

Contents

Chapter 1

Introduzione

Per “metodi iterativo” si intendono quelle tecniche per la risoluzione di sistemi lineari che approssimano iterativamente soluzioni sempre più accurate.
Essi si suddividono in due tipi: metodi stazionari - di più vecchia ideazione e meno efficienti - e quelli non stazionari. I primi prendono il loro nome dal fatto che, ad ogni iterazione, svolgono la stessa operazione sul vettore soluzione, indipendentemente dall'iterazione stessa. I secondi, invece, usano coefficienti diversi ad ogni iterazione per approssimare la prossima soluzione.

1.1 Criteri di arresto

Il ciclo iterativo può finire per vari motivi:

- superamento numero massimo iterazioni: se si impone un numero massimo di iterazioni, ci si ferma quando questo è raggiunto
- approssimazione soddisfacente: dato un valore di tolleranza, ci si ferma quando la soluzione trovata approssima quella esatta a meno di questo valore. Ossia, quando il residuo scalato:
$$\frac{\left\|\mathbf{b} - A\mathbf{x}^k\right\|}{\|\mathbf{b}\|}$$
diventa minore della tolleranza data.
- incremento tra un'iterata e la seguente: se
$$\frac{\left\|\mathbf{x}^{k+1} - \mathbf{x}^k\right\|}{\left\|\mathbf{x}^k\right\|}$$
è minore di una tolleranza data, allora l'algoritmo non migliora più l'approssimazione in modo soddisfacente e ci si ferma

Chapter 2

Progetto

Lo scopo del progetto è sviluppare una libreria che esponga quattro metodi iterativi per la risoluzione dei sistemi lineari. In particolare, è stato chiesto di implementare due metodi stazionari e due non stazionari.

2.1 Metodi stazionari

Per quanto riguarda questi metodi, sono stati implementati il metodo di **Jacobi** e quello di **Gauss-Seidel**. In entrambi i casi, si *splitta* la matrice A $\in \mathcal{R}^{nxn}$ in due matrici P ed N della stessa dimensione, in modo che A = P - N. A questo punto, sostituendo ad A la sua decomposizione, si ottiene la seguente uguaglianza:

$$x = P^{-1}Nx + P^{-1}b$$

, dal quale si ricava facilmente un metodo iterativo. Manipolando ulteriormente questa equazione (sostituzione di P - A ad N), si ottiene la seguente forma:

$$x^{k+1} = x^k + P^{-1}r^k$$

dove r^k è semplicemente $b - Ax^k$.
L'inversione della matrice P, generalmente un processo costoso in termini di elaborazione, è reso efficiente dal fatto che per il metodo di Jacobi P è semplicemente la matrice A a cui sono stati azzerati tutti i coefficienti non sulla diagonale principale. La sua inversa quindi è semplicemente una matrice che ha sulla diagonale principale l'inverso dei coefficienti di A nelle stesse posizioni, zero in tutte le altre posizioni.
Per il metodo di Gauss-Seidel, invece, dove la matrice P è la matrice triangolare inferiore ottenuta da A. Al posto di calcolare la sua inversa in modo diretto, viene invece calcolata la soluzione **y** del sistema $y = P^{-1}r^k$, che si risolve facilmente grazie alla sostituzione in avanti per merito della particolare struttura di P.
La convergenza di questi metodi è garantita per qualsiasi valore iniziale x^0 se esiste una scomposizione A = P - N, con P ed A simmetriche e definite positive e a patto che 2P - A sia a sua volta definita positiva e il modulo dell'autovalore di modulo massimo sia minore di 0. La convergenza risulta inoltre monotona rispetto alle norme dei vettori, e quindi si ha la garanzia che ad ogni iterazione ci si avvicina sempre più alla soluzione esatta.
Per verificare la convergenza dei metodi di Jacobi e Gauss-Seidel (senza dover calcolare

l'autovalore di modulo massimo), si può verificare che la matrice A sia a dominanza diagonale stretta per righe.
Entrambi i metodi hanno una complessita $O(n^2)$, dovuta al calcolo del prodotto tra la matrice A e il vettore soluzione x ad ogni iterazione (necessario per il calcolo del residuo) e, per quanto riguarda il metodo di Gauss-Seidel, anche alla risoluzione del sistema $y = P^{-1}r^k$.

2.2 Metodi non stazionari

Per quanto riguarda i metodi non stazionari, come sopra citato, lo spostamento nello spazio delle soluzioni è influenzato da un coefficiente specifico all'iterazione. Mentre per i metodi stazionari l'aggiornamento poteva essere visto come $x^{k+1} = x^k + P^{-1}r^k$, in questo caso l'aggiornamento viene fatto secondo una regola $x^{k+1} = x^k + \alpha_k P^{-1}r^k$, con α_k che varia ad ogni iterazione. In particolare, è stato richiesto di implementare i metodi del Gradiente e del Gradiente Coniugato.
Per quanto riguarda il primo, si parte da una funzione da $\mathcal{R}^n$ in $\mathcal{R}$ tale che

$$\phi(\mathbf{y}) = 0.5\mathbf{y}^t Ay - \mathbf{b}^t y$$

Nel caso di una matrice A simmetrica e definita positiva, questa è una forma quadratica il cui minimo corrisponde alla soluzione del sistema A$\mathbf{x} = \mathbf{b}$. Il nome del metodo deriva dal fatto che per calcolare questo si passa per il gradiente della funzione ϕ - procedimento che porta a risultati solo nel caso di A simmetrica e definita positiva.
Le approssimazioni del vettore soluzione, quindi, si riducono a vettori che si avvicinano ad ogni iterazione al minimo della funzione ϕ.
Come per qualsiasi algoritmo di ricerca del minimo, ci si muove procedendo con un certo passo (α_k) nella direzione opposta al gradiente, in quanto il gradiente di una funzione indica la direzione di massima crescita. La direzione in cui muoversi coincide con il residuo al passo k, perciò l'aggiornamento viene fatto nel seguente modo:

$$x^{k+1} = x^k + \alpha_k \mathbf{r}^k$$

Il coefficiente, invece, che per definizione varia ad ogni iterazione, è dato da $\alpha_k = \frac{(r^k)^t r^k}{(r^k)^t A\mathbf{r}^k}$.
La velocità di convergenza di questo metodo dipende dal numero di condizionamento della matrice; infatti più il numero di condizionamento è vicino a 1, più il "tragitto" tra il punto iniziale x^0 e la soluzione del sistema sarà diretto. Invece, per numeri di condizionamento molto grandi, la discesa verso il minimo procederà a zig zag.
Il metodo del gradiente coniugato è un miglioramento del precedente e punta ad eliminare questa discesa a zig zag.
Il modo in cui si procede è correggendo di volta in volta la direzione in cui ci si sposta affinché questa sia ottimale rispetto alle approssimazioni del vettore soluzione. Si ha quindi un aggiornamento della soluzione secondo

$$x^{k+1} = x^k + \alpha_k \mathbf{d}^k$$

dove $\alpha_k = \frac{(\mathbf{d}^k)^t \mathbf{r}^k}{(\mathbf{d}^k)^t A\mathbf{d}^k}$, $\mathbf{d}^{k+1} = \mathbf{r}^{k+1} - \beta_k \mathbf{d}^k$ ed infine $\beta_k = \frac{(\mathbf{d}^k)^t A\mathbf{r}^{k+1}}{(\mathbf{d}^k)^t A\mathbf{d}^k}$.
La convergenza, come per il metodo del gradiente, è garantita se A è simmetrica e definita positiva. Inoltre, però, c'è la garanzia che questo metodo converga in al più n iterazioni.

Chapter 3

Implementazione della libreria

3.1 Caratteristiche del hardware

L'elaboratore utilizzato è un portatile di marca HP, con sistema operativo Windows 10.
Il processore è un Intel Core i5 di settima generazione, con frequenza base 2.70GHz (turbo fino a 3.40GHZ). Ha a disposizione 128KB per quanto riguarda la cache di primo livello, 512KB per quella di secondo livello ed infine 3MB di cache di terzo livello.
Per quanto riguarda la memoria, invece, il PC è munito di 8GB di RAM (velocità 2133MHz).

3.2 Scelte di design

Si è scelto di implementare la libreria usando il linguaggio di programmazione Python.
Come libreria di appoggio per le operazioni matriciali si è utilizzato **numpy**.
La libreria espone tre metodi pubblici: uno per la lettura della matrice da file, uno per la risoluzione di sistemi manuale, e uno per la validazione del metodo che svolge in automatico tutti e quattro i metodi per poi stampare a schermo i dati sulla performance di ciascuno (numero di iterazioni, l'errore relativo della soluzione trovata rispetto a quella esatta fornita, il tempo di esecuzione e se il metodo è riuscito a convergere nel numero di iterazioni massimo impostato).
Per quanto riguarda invece la struttura interna, quella più interessante è senz'altro la funzione **solve_ls**, ossia quella che svolge effettivamente la risoluzione del sistema fornito.
Si è scelto di creare una sola funzione piuttosto che quattro separate per emulare lo stile programmativo delle librerie come scipy e numpy, che prendono in input il metodo di risoluzione desiderato quando ci sono più opzioni disponibili.
Data che il comando di **switch** non è disponibile in Python, si è dovuto utilizzare una sequenza di if-elif. Inizialmente, si creano tutti i vettori di supporto necessari ai metodi e il residuo al passo 0, valore comune a qualsiasi dei quattro metodi.
Vengono in seguito creati i vettori di supporto specifici (la matrice P per Jacobi e Gauss, i coefficienti y e d per il metodo del gradiente coniugato).
Se il metodo selezionato è uno di quelli stazionari, viene invocato il metodo
__check_diagonal_dominance__, che controlla se la matrice è a dominanza diagonale stretta per righe e, in caso negativo, solleva un warning per avvisare l'utente che la garanzia non è garantita. Il residuo viene calcolato poi fuori dall'if, dopo aver aggiornato

il vettore soluzione, e non in ciascun ramo dell'if per evitare ripetizione di codice.
Il vettore da sommare alla soluzione precedente viene calcolato, per i metodi stazionari, dalle funzioni di appoggio __update_[Nome Metodo], che prendono in input le operazioni necessarie ed elaborano separatamente gli **add-on** necessari. Per quanto riguarda Jacobi, questo è semplicemente un modo per rendere il codice consistente tra i metodi stazionari, mentre Gauss invece si appoggia ad un'ulteriore funzione (__forward_substitution__) che risolve il sistema $P^{-1}\mathbf{r}^k$ e quindi separare il codice porta ad un maggior livello di separation of concerns.
Per i metodi non stazionari, invece, i coefficienti alpha vengono ciascuno calcolato in una funzione a parte e poi l'add-on viene calcolato direttamente nel branch dell'if corrispondente, in quanto è semplicemente una moltiplicazione di vettore per scalare.
Nel caso del gradiente coniugato, il metodo di calcolo del coefficiente α ritorna un valore $\mathbf{y}$, che sarà necessario per l'aggiornamento della direzione di spostamento $\mathbf{d}$. Questo permette di efficientizzare il codice, evitando la ripetizione della costosa moltiplicazione matrice vettore necessaria per il calcolo di y.
Un'ulteriore efficientizzazione è stata fatta scegliendo di aggiornare la direzione di spostamento dopo il calcolo del residuo fatto fuori dall'if, in modo da non dover ripetere questo calcolo nella funzione di aggiornamento di $\mathbf{d}$.

3.3 Codice

Listing 3.1: Library

```python
from enum import Enum
import numpy as np
import time
import pandas as pd
import warnings

#global variable
MAX_ITER = 30000

class Method(Enum):
    JACOBI = "JACOBI"
    GAUSS_SEIDEL = "GAUSS_SEIDEL"
    GRADIENT = "GRADIENT"
    CONJ_GRADIENT = "CONJ_GRADIENT"


def read_matrix(input_file):
    """
    Reads a matrix from file path. Matrix must be a Matrix Market format.
    Returns a ndarray

    Parameters
    ----------
    input_file: string

    Returns
    ----------
    matrix: ndarray
```

```python
    """

    ##checks for right input type and right format
    if (not type(input_file) == str) or (not input_file.endswith(".mtx")):
        raise Exception("Wrong file extension")

    file = open(input_file)

    #split line into individual strings
    rows, columns, nnz = file.readline().split()

    #convert to int
    total_rows = int(rows)
    total_columns = int(columns)
    nnz = int(nnz)

    #initialise empty matrix
    matrix = np.zeros(shape=(total_rows, total_columns))
    for line in file:
        #read line, split into individual strings, convert to numbers, append
        element = line.split()

        #-1 required since mtx indexing starts from 1
        row = int(element[0])-1
        column = int(element[1])-1
        value = float(element[2])

        matrix[row, column] = value

    return matrix

def solve_ls(matrix, b, tol, method = Method.JACOBI):
    """
    """
     # Type checking
    if method not in Method._member_names_ and not isinstance(method, Method):
        raise TypeError('Method not supported')

    if  method == Method.JACOBI or method.upper() == Method.JACOBI.value or \
    method == Method.GAUSS_SEIDEL or method.upper() == Method.GAUSS_SEIDEL.value:
        print("Checking convergence criteria")
        if not __check_diagonal_dominance__(matrix):
            warnings.warn("Convergence not guaranteed")

    #turn b into column array if not already in that shape
    if b.shape[1] != 1:
        np.reshape(b, (b.shape[1], b.shape[0]))
    #initialise first solution to 0 vector
    x = np.zeros((matrix.shape[0], 1))

    #initialise counter
    k = 0

    #get initial error
    residue = b - matrix @ x
```

```python
    #needed for conjugated gradient
    d = residue
    y = np.zeros_like(residue)

    error = np.linalg.norm(residue) / np.linalg.norm(b)

    diagonal_p = matrix.diagonal()
    inverse_diagonal = np.reciprocal(diagonal_p)


    #reshape into column array
    inverse_diagonal = np.reshape(inverse_diagonal, (inverse_diagonal.shape[0], 1))

    #create lower triangular matrix
    triangular_p = np.tril(matrix)

    while k < MAX_ITER and error > tol:
        if method == Method.JACOBI or method.upper() == Method.JACOBI.value:
            add_on = __update_jacobi__(x, residue, inverse_diagonal)
            x = x + add_on

        elif method == Method.GAUSS_SEIDEL or \
        method.upper() == Method.GAUSS_SEIDEL.value:
            add_on = __update_gauss__(residue, triangular_p)
            x = x + add_on

        elif method == Method.GRADIENT or \
        method.upper() == Method.GRADIENT.value:
            alpha = __gradient_alpha__(matrix, residue)
            x = x + alpha*residue

        elif method == Method.CONJ_GRADIENT or \
        method.upper() == Method.CONJ_GRADIENT.value :
            alpha, y = __conjugated_gradient_alpha__(x, matrix, residue, d)
            x = x + alpha*d

        residue = b - matrix @ x
        if method == Method.CONJ_GRADIENT or \
        method.upper() == Method.CONJ_GRADIENT.value:
            d = __update_conjugated_gradient_d__(matrix, residue, d, y)

        #update stopping criterion
        k += 1
        error = np.linalg.norm(residue) / np.linalg.norm(b)

    if k >= MAX_ITER:
        raise Exception("No convergence")
    return x, k

def validate(matrix, b, tol, exact_solution):
    iterations = np.zeros(shape=(4))
    errors = np.zeros_like(iterations)
    execution_time = np.zeros_like(iterations)
    convergent = np.ones_like(iterations).astype(bool)


    for method, i in zip(Method._member_names_, range(4)):
```

```
        print("Solving with... " + method)
        time_start = time.perf_counter()
        try:
            result, iterations[i] = solve_ls(matrix, b, tol, method=method)
            execution_time[i] = time.perf_counter() - time_start
            errors[i] = np.linalg.norm(exact_solution - result ) / \
            np.linalg.norm(exact_solution)
        except Exception:
            convergent[i] = False
    data = np.array([Method._member_names_, iterations, errors, \
    execution_time, convergent]).transpose()
    results = pd.DataFrame(data=data,
    columns = ["Method", "Iterations", "Relative error",\
     "Execution time (s)", "Convergence" ])
    print(results)


def __update_jacobi__(x, residue, p_1):
    #elementwise multiplication
    add_on = np.multiply(p_1, residue)
    return add_on

def __update_gauss__(residue, triangular_p):

    y = __forward_substitution__(triangular_p, residue)
    return y

def __forward_substitution__(matrix, b):

    x = np.zeros(shape = (matrix.shape[0], 1))
    if matrix[0,0] == 0:
        raise Exception("Unsolvable linear system")
    x[0] = b[0]/matrix[0,0]

    for i in np.arange(1, matrix.shape[0]):
        if matrix[i, i] == 0:
            raise Exception("Unsolvable linear system")
        x[i] = (b[i]-matrix[i, :] @ x) / matrix[i, i]
    return x

def __gradient_alpha__(matrix, residue):

    transposed_residue = residue.transpose()
    y = matrix @ residue
    a = transposed_residue @ residue
    b = transposed_residue @ y
    return a/b

def __conjugated_gradient_alpha__(x, matrix, residue, d):

    y = matrix @ d
    z = matrix @ residue


    #returns an array of array
    alpha = (d.transpose() @ residue) / (d.transpose() @ y)
    return alpha[0, 0], y
```

```python
def __update_conjugated_gradient_d__(matrix, residue, d, y):
    w = matrix @ residue

    #returns an array of an array
    beta = (d.transpose() @ w) / (d.transpose() @ y)

    return residue - beta[0, 0] * d

def __check_diagonal_dominance__(matrix):
    row_sum = np.sum(np.absolute(matrix), axis = 1).reshape(matrix.shape[0], 1)
    for i in matrix.shape[0]:
        row = row_sum[i] - matrix[i, i]
        if matrix[i, i] <= row:
            return False
    return True
```

Listing 3.2: Testing

```python
import iterative
import numpy as np
import os
import time

for filename in os.listdir(os.getcwd()):
    if filename.endswith("vem2.mtx"):
        print("Matrix:" + filename)
        matrix = iterative.read_matrix(filename)
        x = np.ones(shape = (matrix.shape[0], 1))
        b = matrix @ x
        for tol in [10e-4, 10e-6, 10e-8, 10e-10]:

            print("\n\nTolerance: " + str(tol))
            iterative.validate(matrix, b, tol, x)
        print("\n\n")
time.sleep(10)
input("Press key to exit")
```

Chapter 4

Validazione

Tutte le matrici fornite sono definite positive e simmetriche. La validazione è avvenuta nel seguente modo:

- Per ciascuna matrice fornita:
  1. Creazione di un vettore soluzione $\mathbf{x}$ delle dimensioni appropriate per ogni matrice, con 1 in tutte le posizioni.
  2. Generazione del termine $\mathbf{b}$ attraverso il calcolo del prodotto $A\mathbf{x}$.
  3. Per ciascun valore di tolleranza presente nella lista $[10^{-4}; 10^{-6}; 10^{-8}; 10^{-10}]$:
     - invocazione del metodo validate, con la terna A, b, x precedentemente ottenuta e il relativo valore di tolleranza

Le matrici fornite sono quadrate e hanno lato rispettivamente:

- spa1: 1000
- spa2: 3000
- vem1: 1681
- vem2: 2601

4.1 Risultati

Lo script di test stampa a video la tolleranza usata per ogni validazione; il metodo di validazione si occupa invece di stampare le statistiche dei quattro metodi di risoluzione per i valori dati. I risultati ottenuti sono stati i seguenti:

4.2 Conclusioni

Tutti i metodi giungono a convergenza, per tutti i valori di tolleranza forniti. Questo era assicurato per quanto riguarda il metodo del Gradiente Coniugato, visto che il numero massimo di iterazioni scelto (30000) è minore di n, e come precedentemente detto il metodo giunge a convergenza in al più n iterazioni se A è simmetrica e definita positiva

```
          Method Iterations        Relative error   Execution time (s) Convergence
0         JACOBI       82.0  0.017581054332084643  0.06403020000000015        True
1   GAUSS_SEIDEL        5.0   0.21180807454199985  0.06287369999999992        True
2       GRADIENT       43.0   0.05103089460317297  0.06289950000000011        True
3  CONJ_GRADIENT       17.0  0.043159998047848765  0.05064669999999993        True


Tolerance: 1e-05
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error   Execution time (s) Convergence
0         JACOBI      148.0  0.00017845556047522263  0.10543820000000004        True
1   GAUSS_SEIDEL       13.0   0.0015418817074427207  0.12907180000000018        True
2       GRADIENT     1331.0    0.009307067852848305            1.6813785        True
3  CONJ_GRADIENT       99.0     0.00152348395159153  0.24080760000000012        True


Tolerance: 1e-07
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error   Execution time (s) Convergence
0         JACOBI      214.0   1.811403710752332e-06  0.14539500000000016        True
1   GAUSS_SEIDEL       20.0  2.0314114145444828e-05  0.20837269999999997        True
2       GRADIENT     5895.0   9.791807992091946e-05    7.777568799999999        True
3  CONJ_GRADIENT      147.0  1.4941664156301329e-05   0.3525089999999995        True


Tolerance: 1e-09
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error   Execution time (s) Convergence
0         JACOBI      280.0  1.8386557367193788e-08  0.18623469999999998        True
1   GAUSS_SEIDEL       28.0  1.4388327212144483e-07  0.28047920000000026        True
2       GRADIENT    10575.0   9.82553841529585e-07           14.0056689        True
3  CONJ_GRADIENT      188.0   1.526535207622451e-08   0.5195991000000006        True
```

Figure 4.1: Risultati con la matrice Spa1

```
          Method Iterations        Relative error   Execution time (s) Convergence
0         JACOBI       26.0  0.016271282719801564  0.23219690000000526        True
1   GAUSS_SEIDEL        4.0  0.010499150364997566   0.2817876999999953        True
2       GRADIENT       41.0  0.044954746095450124   0.5648987000000005        True
3  CONJ_GRADIENT       17.0     0.033909139642559  0.46010600000000323        True


Tolerance: 1e-05
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error  Execution time (s) Convergence
0         JACOBI       47.0  0.00015354763842225178  0.3638343000000006        True
1   GAUSS_SEIDEL        7.0  0.00019069675616587475  0.4386842999999985        True
2       GRADIENT      663.0    0.004945174389085545   7.560557199999998        True
3  CONJ_GRADIENT       90.0   0.0010252866199216605  2.0939267000000044        True


Tolerance: 1e-07
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error   Execution time (s) Convergence
0         JACOBI       67.0  1.8092600110131537e-06  0.46798249999999797        True
1   GAUSS_SEIDEL       10.0   3.804187255207967e-06   0.5925429999999992        True
2       GRADIENT     3507.0   6.795364665958029e-05   39.642027899999995        True
3  CONJ_GRADIENT      168.0   7.994728439602805e-06    3.943043099999997        True


Tolerance: 1e-09
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error  Execution time (s) Convergence
0         JACOBI       88.0  1.7073466753428223e-08  0.6120574000000119        True
1   GAUSS_SEIDEL       14.0  2.0549535626147797e-08  0.8011355000000009        True
2       GRADIENT     6681.0   6.915635904307981e-07          79.7759826        True
3  CONJ_GRADIENT      220.0  2.7200605724563428e-08   5.123248199999978        True
```

Figure 4.2: Risultati con la matrice Spa2

```
          Method Iterations         Relative error  Execution time (s) Convergence
0         JACOBI      755.0    0.03533256410819245  1.4078373000000113        True
1   GAUSS_SEIDEL      379.0    0.03511170647295883   8.753557899999976        True
2       GRADIENT      528.0   0.027211993589950353  1.7308645999999897        True
3  CONJ_GRADIENT       34.0  0.0005239171211776817  0.2438872000000174        True


Tolerance: 1e-05
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error   Execution time (s) Convergence
0         JACOBI     1874.0  0.00035329496150410684     3.120332100000013        True
1   GAUSS_SEIDEL      939.0  0.00035023882334969604   24.199687100000006        True
2       GRADIENT     1252.0  0.00026901347641016703     4.242305299999998        True
3  CONJ_GRADIENT       42.0  2.7571433745761622e-06  0.32097219999999993        True


Tolerance: 1e-07
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error  Execution time (s) Convergence
0         JACOBI     2993.0  3.5326428487991903e-06  5.2606237999999905        True
1   GAUSS_SEIDEL     1498.0  3.5220738969607597e-06          28.0310499        True
2       GRADIENT     1974.0  2.7039914759116454e-06   6.415838699999966        True
3  CONJ_GRADIENT       49.0   5.332624484072261e-08  0.3374360999999908        True


Tolerance: 1e-09
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations         Relative error  Execution time (s) Convergence
0         JACOBI     4112.0  3.532336402900715e-08   7.446562799999981        True
1   GAUSS_SEIDEL     2058.0  3.512846425574842e-08  52.504599799999994        True
2       GRADIENT     2696.0  2.721433257101434e-08   9.836978600000009        True
3  CONJ_GRADIENT       56.0  3.720192792896514e-10  0.5473835000000236        True
```

Figure 4.3: Risultati con la matrice vem1

```
          Method Iterations          Relative error  Execution time (s) Convergence
0         JACOBI     1053.0    0.04962636949646427  5.7632634000000005        True
1   GAUSS_SEIDEL      528.0    0.04942282048796491          18.2477308        True
2       GRADIENT      744.0    0.03809609827008548   8.083217600000001        True
3  CONJ_GRADIENT       42.0  0.0007327517370822417  1.0374701000000002        True


Tolerance: 1e-05
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error  Execution time (s) Convergence
0         JACOBI     2802.0  0.0004961211647882545  13.787771999999997        True
1   GAUSS_SEIDEL     1402.0  0.0004959521773189988  37.294423300000005        True
2       GRADIENT     1872.0  0.0003815736908137761  16.515670199999988        True
3  CONJ_GRADIENT       52.0  4.067723839667869e-06   0.957253399999999        True


Tolerance: 1e-07
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations          Relative error  Execution time (s) Convergence
0         JACOBI     4551.0  4.959786747965904e-06  20.122899900000007        True
1   GAUSS_SEIDEL     2277.0  4.950058490484862e-06          56.5596458        True
2       GRADIENT     3002.0  3.800243314971529e-06   25.64228539999999        True
3  CONJ_GRADIENT       60.0  6.310003904383341e-08  1.0620069999999942        True


Tolerance: 1e-09
Solving with... JACOBI
Solving with... GAUSS_SEIDEL
Solving with... GRADIENT
Solving with... CONJ_GRADIENT
          Method Iterations           Relative error  Execution time (s) Convergence
0         JACOBI     6299.0   4.971435034261786e-08  27.068095999999997        True
1   GAUSS_SEIDEL     3152.0   4.9406118772070773e-08  80.83205150000003        True
2       GRADIENT     4130.0   3.8197685682764127e-08 37.532838200000015        True
3  CONJ_GRADIENT       70.0   4.788316234893667e-10  1.2928534999999783        True
```

Figure 4.4: Risultati con la matrice Vem2

- in questo caso, le dimensioni delle matrici prese in esame (tutte simmetriche e definite positive) sono molto inferiori al numero massimo di iterazioni. Per i metodi stazionari, invece, si è controllata la dominanza diagonale stretta per righe prima di tentare di svolgere il sistema.
Si può notare, inoltre, come per le soglie di tolleranza più basse, c'è una marcata differenza tra il numero di iterazioni del metodo del Gradiente e quello del Gradiente Coniugato per tutte le matrici - questo fa pensare che il numero di condizionamento delle matrici sia molto maggiore di 1, causando il metodo del Gradiente ad esplorare molto di più lo spazio soluzioni prima di giungere ad una che soddisfa il criterio di arresto. Il caso lampante è quello della matrice Spa1, dove il metodo del Gradiente impiega più di 10 mila iterazioni, mentre quello del Gradiente Coniugato ne impiega meno di 200.
Così come si evidenzia in modo netto l'efficienza maggiore del metodo del Gradiente Coniugato per quanto riguarda i metodi non stazionari (risultato atteso visto che è un miglioramento del metodo del Gradiente), lo stesso accade anche per i metodi stazionari: infatti, il metodo di Gauss-Seidel è un miglioramento del metodo di Jacobi, e questo è immediatamente evidente a livello di numero di iterazioni necessarie per giungere ad una soluzione soddisfacente. Tuttavia, si può vedere come il tempo di esecuzione sia maggiore in tutti i casi per il metodo di Gauss-Seidel: infatti richiede anche la risoluzione di un sistema lineare secondario, che apporta un ulteriore operazione di costo $O(n^2)$.
La terza entrata della tabella delle statistiche riassuntive dei metodi è quella dell'errore relativo: questo infatti misura l'errore tra la soluzione approssimata e la soluzione trovata. Si può vedere che per quanto riguarda i metodi stazionari, non ci sono grandi differenze tra gli errori commessi. In tutti i casi si tratta dello stesso ordine di grandezza.
Questo non è vero per quanto riguarda i metodi non stazionari. Infatti, il metodo del Gradiente Coniugato è nettamente superiore, sia in tempi di esecuzione che in numero di iterazioni necessarie, ma anche per l'errore relativo - infatti per tutte le matrici, l'errore relativo del metodo del Gradiente Coniugato è un ordine di grandezza (o quasi, si veda la matrice Spa1, tolleranza 1^{-5} e rispettivamente 1^{-7}, dove pur essendo lo stesso ordine di grandezza, si hanno valori al limite).

- matrice spa1: il metodo che produce risultati migliori a livello di numero di iterazioni è il metodo di Gauss-Seidel; a livello di tempo di esecuzione, invece, quello di Jacobi è migliore per quanto riguarda i valori di tolleranza più bassi

- matrice spa2: anche per questa matrice si vede la velocità di convergenza (a livello di iterazioni) del metodo di Gauss-Seidel. Per tutti i valori di tolleranza, produce risultati più accurati in tempi nettamente minori. Questo fa pensare ad un numero di condizionamento particolarmente basso, data la difficoltà di convergenza dei metodi non iterativi che ne dipendono, specie per quanto riguarda il metodo del Gradiente.

- vem1: per questa matrice, il metodo del Gradiente Coniugato si mostra subito superiore a tutti gli altri, anche per il valore di tolleranza più basso. Infatti produce un risultato molto più accurato di tutti gli altri (si veda la tolleranza di 1^{-5}, dove si hanno addirittura due ordini di grandezza di differenza) in una frazione del tempo necessario per gli altri tre, e con un numero di iterazioni che resta nelle decine - mentre gli altri vanno nell'ordine delle migliaia.

- vem2: anche per questa matrice, come per quella precedente, valgono le stesse considerazioni. Tutti e tre i valori di paragone considerati sono nettamente migliori per quanto riguarda il metodo del Gradiente Coniugato. Per queste due matrici, quindi, si può supporre un buon numero di condizionamento (la convergenza col metodo del Gradiente, infatti, si ha sempre in meno iterazioni rispetto al numero di Jacobi).