Metodi del Calcolo Scientifico

Progetto 1 Bis

Biancini Mattia 865966
Gargiulo Elio 869184

Indice

1 Introduzione

La scelta di svolgere il progetto usando come linguaggio di programmazione Python è dettata dalla flessibilità e adattabilità che questo linguaggio offre.
Come librerie per la struttura dati delle Matrici e Vettori abbiamo adottato:

- **NumPy**: Fornisce degli strumenti molto efficienti per le operazioni tra matrici e la gestione della struttura dati di matrici e vettori densi
- **SciPy**: Fornice una struttura dati per gestire in memoria le matrici sparse in un metodo analogo a quello fornito da MatLab.

1.1 Struttura Generale

La struttura generale del progetto si compone di due principali file Python:

- **main.py**: File che contiene funzioni di importazione e caricamento delle matrici tramite `SciPy`, funzioni per la scrittura su file del risultati ottenuti dall'esecuzione dei metodi risolutivi, e l'utilizzo vero e proprio della libreria implementata in `methods.py`.
- **methods.py**: File che implementa le funzioni cuore della libreria. In particolare vi è l'implementazione dei quattro metodi risolutivi, funzioni supplementari che aiutano a verificare/semplificare operazioni dei metodi (es. controllo della dominanza diagonale per righe) e funzioni di debugging.

Il progetto si divide di due classi principali `main.py`, il punto di inizio del programma in cui è possibile importare le matrici sparse in formato `.mtx` tramite `SciPy` e `methods.py` che il vero e proprio cuore della libreria.

2 Struttura del Main

2.1 Caricamento e Utilizzo delle Matrici

In seguito le funzioni principali utilizzate per l'importazione delle matrici all'interno del progetto. Esse sono necessarie al fine di poter fornire alla libreria, e quindi ai metodi risolutivi, le matrici trattate.
Sono state scritte funzioni per il caricamento di matrici in modo casuale, oppure tramite scelta attraverso un indice.
Le funzioni per il caricamento casuale sono state omesse in relazione in quanto estremamente simili a quelle tramite indice.

```
    """
    FUNZIONE: load_matrices():

    DESCRIZIONE:
    Carica le matrici sparse contenute nella cartella "./matrix"
    nel progetto e le ritorna al chiamante

    """
def load_matrices():
    # Ottengo la directory
    matrix_dir = os.path.dirname(os.path.abspath(__file__))
    matrix_dir = os.path.join(matrix_dir, 'matrix')
    # Ottengo i file contenuti nella directory
    matrix_files = os.listdir(matrix_dir)
    matrices = []
    for x in matrix_files:
        # Se i file sono effettivamente matrici "Matrix Market"
        if x.__contains__('.mtx'):
            # Appendo le matrici
            matrices.append(os.path.join(matrix_dir, x))

    return matrices

    """
    FUNZIONE: import_matrix(matrices, index):

    DESCRIZIONE:
    Variante della funzione "import_random_matrix(matrices) che permette
    la selezione di una specifica matrice in posizione "index", nel caso sia
    un indice corretto. Altrimenti ritorna al chiamante una matrice casuale

    """
def import_matrix(matrices, index):
    # Se la lunghezza della lista e' 0 (vuota)
    if len(matrices) == 0:
        return
    # Se l'indice non e' valido
    if index >= len(matrices) or index < 0:
        print('This is invalid number ({}). A random Matrix will be choose.'.format(
    index))
        return import_random_matrix(matrices)
    # Seleziono una matrice casuale e la assegno a matrix
    matrix = sp.io.mmread(matrices[index])
    return matrix

"""
    FUNZIONE: get_matrix(index):

    DESCRIZIONE:
    Funzione principale che incapsula "import_matrix" con "load_matrices"
    parametro, al fine di importare e selezionare una matrice dato un indice
    "index" con un unica chiamata.

"""
def get_matrix(index):
    return import_matrix(load_matrices(), index)
```

2.2 Salvataggio dei Risultati Ottenuti

La funzione implementata permette la scrittura su un file *"risultati.txt"* di un'intera esecuzione del main. In particolare vengono scritti su file:

- **data**: tupla contentente i seguenti dati:
  - *Soluzione Ottenuta*
  - *Tempo Impiegato*
  - *Errore Relativo*
  - *Numero di Iterazioni*
- **method**: nome del metodo utilizzato per la risoluzione del sistema
- **tolerance**: valore di tolleranza usata per arrestare le iterazioni (convergenza)
- **matrix number**: numero della matrice trattata

```
def write_results_to_file(data, method, tolerance, matrix_number):
    # Verifica se il file "risultati.txt" esiste gia'
    file_name = "risultati.txt"
    file_exists = os.path.exists(file_name)
    # Se il file non esiste, crea un nuovo file "risultati.txt"
    if not file_exists:
        with open(file_name, "w") as file:
            file.write("====[ DATI ESECUZIONE METODI ]====\n\n")
    # Scrivi i dati nella tupla nel file
    with open(file_name, "a") as file:
        file.write('=== {} Method ===\n'.format(method))
        file.write('Matrice: {}\n'.format(matrix_number))
        file.write('Tempo Impiegato: {:.9f}\n'.format(data[1].total_seconds()))
        file.write('Errore Relativo: {}\n'.format(data[2]))
        file.write('Numero di Iterazioni {}/50000\n'.format(data[3]))
        file.write('Tolleranza: {}\n'.format(tolerance))
        file.write('X = {}\n\n'.format(data[0]))
```

2.3 Utilizzo della Libreria

Nella funzione main vengono utilizzate le funzioni analizzate in precedenza al fine di permettere l'esecuzione dei metodi risolutivi. E' presente una prima parte concentrata al testing dove vengono provati, se *"solveall = True"*, tutti i metodi implementati della libreria in **methods.py**, con risultati e informazioni scritti su file *"risultati.txt"*. La seconda parte è un esempio di programma dove attraverso la console si possono specificare parametri (matrice, tolleranza e metodo di risoluzione) per poi ottenere i risultati desiderati. In seguito vi è riportata la parte di codice più importante, con l'esecuzione e salvataggio del risultati dei quattro metodi risolutivi.

```
# Carico le matrici
matrices = load_matrices()
tol = [1e-4, 1e-6, 1e-8, 1e-10]  # Lista di tolleranze per scrittura file
for i in range(len(matrices)):
    for j in range(4):
        # Estraggo la matrice i
        matrix = import_matrix(matrices, i)
        # Definisco il LinearSystemSolver, il costruttore offerto dalla
        # libreria implementata in methods.py
        solver = LinearSystemSolver(matrix, maxIteration, False, tol_index=j)
        result = solver.jacobi()  # Metodo di Jacobi
        write_results_to_file(result, 'Jacobi', tol[j], i)
        result = solver.gauss_seidel()  # Metodo di Gauss-Seidel
        write_results_to_file(result, 'Gauss-Seidel', tol[j], i)
        result = solver.gradient()  # Metodo del Gradiente
        write_results_to_file(result, 'Gradient', tol[j], i)
        result = solver.conjugate_gradient()  # Metodo del Gradiente Coniugato
        write_results_to_file(result, 'Conjugate Gradient', tol[j], i)
```

3 Struttura della Libreria

La struttura della libreria è composta da una classe chiamata **LinearSystemSolver** contenente:

- Il costruttore della classe
- Funzione per l'esecuzione del metodo di Jacobi
- Funzione per l'esecuzione del metodo di Gauss-Seidel
- Funzione per l'esecuzione del metodo del Gradiente
- Funzione per l'esecuzione del metodo del Gradiente Coniugato
- Funzioni di utilità

```
"""
    CLASSE: LinearSystemSolver

    DESCRIZIONE:
    La classe che contiene l'implementazione e quindi il cuore di tutta la libreria

    - Costruttore: __init__
    - Metodi Principali:
        - jacobi(self) -> tuple[Any, timedelta, float, int]
        - gauss_seidel(self) -> tuple[Any, timedelta, float, int]
        - gradient(self) -> tuple[Any, timedelta, float, int]
        - conjugate_gradient(self) -> tuple[Any, timedelta, float, int]

"""

class LinearSystemSolver:
    # Lista di tolleranze utilizzate
    tol = [1e-4, 1e-6, 1e-8, 1e-10]
```

3.1 Costruttore

```
"""
    FUNZIONE:  __init__(self, matrix, iteration, debug, tol_index=None, tol=0):

    DESCRIZIONE:
    Costruttore della classe LinearSystemSolver che necessita dei seguenti parametri:
        - matrix: la matrice su cui applicare un metodo
        - iteration: numero di iterazioni massimo per un metodo
        - debug: flag per abilitare il debug
        - tol_index: indice per selezionare una tolleranza, di default nessuno
        - tol: valore specifico di tolleranza, di default = 0

"""
def __init__(self, matrix, iteration, debug, tol_index=None, tol=0):
    # Assegnazione dei parametri
    self.prefix = '[DEBUG] '
    self.matrix = ss.csr_matrix(matrix)
    self.iteration = iteration
    self.debug = debug
    # Se non e' stato specificato un indice, ma un valore di tolleranza CORRETTO
    if tol_index is None and tol != 0:
        self.tol = tol  # Assegno il valore di tolleranza
    # Se non e' stato specificato un indice, ma un valore di tolleranza ERRATO
    elif tol_index is None or tol_index < 0 or tol_index > len(self.tol):
        self.tol = self.tol[randrange(len(self.tol))]
    # Altrimenti ho un indice specifico con cui selezionare la tolleranza
    else:
        self.tol = self.tol[tol_index]
    # Ottengo righe e colonne della matrice
    self.row, self.column = self.matrix.shape
    # Calcolo b = Ax, prodotto scalare tra la matrice e x = [1,...1] sol. esatta
    self.b = self.matrix.dot(_get_solution(self.column))
    # Setup per print che mostra tutta la soluzione
    np.set_printoptions(threshold=np.inf)
    # Setup di flags nel caso sia abilitato o meno il debug
    if debug:
        self.debug_jacobi = True
        self.debug_gauss_seidel = True
        self.debug_gradient = True
        self.debug_conjugate_gradient = True
    else:
        self.debug_jacobi = False
        self.debug_gauss_seidel = False
        self.debug_gradient = False
        self.debug_conjugate_gradient = False
    self.createLog()
```

Il costruttore si occupa di inizializzare alcuni parametri:

- Matrice, come matrice sparsa se è stata passata come tale (`self.matrix = ss.csr_matrix(matrix)`)
- Inizializzare la tolleranza `self.tol` da utilizzare
- Il vettore $b = Ax_{soluzione}$, dove $x_{soluzione}$ è un vettore di tutti 1
- Infine imposta i valori di `debug` nel caso in cui si stia eseguendo debug del codice

3.2 Metodi per la risoluzione del Sistema Lineare

Ognuno dei 4 metodi si compone di inizializzazione in cui vengono inizializzati:

- L'inizio dell'esecuzione del codice: `self._start_time()`
- Eventuali valori da inizializzare prima della prima iterazione, tra cui il vettore nullo di partenza x.

Successivamente ognuno dei metodi si compone di un ciclo `for` eseguito al massimo `maxIteration` volte, valore passato all'instanziazione della classe, nel costruttore. All'interno del ciclo `for`:

- Avvengono gli aggiornamenti di costanti e vettori
- Avviene il controllo di Tolleranza tramite `self._isTolerate(self.x, self.b)`
  - Se $\frac{\|Ax-b\|}{\|b\|} < tol$ il ciclo for viene interrotto e viene eseguito il `return`

Alla fine dell'esecuzione del metodo viene calcolato il tempo di esecuzione del metodo e stampato con una precisione di $10^{-9}s$.
Tutti i metodi ritornano una quadrupla del tipo

$$\begin{bmatrix} result & time & \varepsilon_r & iteration \end{bmatrix}$$

in cui:

- *result* è il vettore x calcolato dal metodo
- *time* è il tempo impiegato per l'esecuzione del metodo
- ε_r è l'errore relativo
- *iteration* è il numero di iterazioni fatte

Ai fini del progetto è stato deciso di avere un numero massimo di iterazioni pari a **50000** in modo da poter permettere a tolleranze molto basse di poter raggiungere il risultato prima di raggiungere il numero massimo di iterazioni. Di seguito vi è riportato nel dettaglio il codice per ogni metodo, evidenziando le funzionalità e quindi le operazioni svolte.

3.2.1 Metodo di Jacobi

```
"""
    FUNZIONE:  jacobi(self) -> tuple[Any, timedelta, float, int]:

    DESCRIZIONE:
    Funzione che implementa il Metodo di Jacobi: x(k+1) = x(k) + P^(-1) r(k)
    Metodo iterativo (risoluzione del problema del fill-in) stazionario che segue una
    strategia di splitting, ovvero basato sulla decomposizione della matrice A = P -
    N.

    L'idea alla base del metodo di Jacobi e' quella di risolvere il sistema
    riscrivendo ciascuna
    equazione in modo che ogni variabile xi sia espressa in funzione delle altre
    variabili e
    dei valori della matrice e del vettore dati.

    Il metodo di Jacobi converge sicuramente se la matrice utilizzata e' a dominanza
    diagonale stretta per righe

    La matrice e' assunta simmetrica e definita positiva

"""
def jacobi(self) -> tuple[Any, timedelta, float, int]:
    # Per debugging
    if self.debug_jacobi:
        self.prefix = '[DEBUG-JACOBI] '
    # Definisco il vettore iniziale nullo come inizio del metodo iterativo
    self.x = _get_start_vector(self.column)
    # Ottengo il tempo iniziale di esecuzione
    self._start_time()
    # Ottengo P^(-1)
    reverse_P =  self._getReverseP_jacobi()
    # Controllo se la matrice e' a dominanza diagonale stretta per righe
    if not self._row_diagonal_dominance():
        print('[ALERT JB] La matrice fornita non e a dominanza diagonale stretta per
    righe -> Convergenza non assicurata.')
    # Ciclo for parte da i=0 fino a iteration-1
    for i in range(self.iteration):
        # Calcolo il residuo come r(k) = b - Ax(k)
        residual = self.b - self.matrix.dot(self.x)
        # Calcolo il passo successivo: x(k+1) = x(k) + P^(-1) r(k)
        self.x = self.x + reverse_P.dot(residual)
        # Se il debug e' attivo stampo l'interazione corrente
        if self.debug_jacobi:
            self._writeIteration(i)
        # Se arrivo alla tolleranza posso fermare l'iterazione
        if self._isTolerate(self.x, self.b):
            if self.debug_jacobi:
                self._writeTolerance()
            # Ottengo il tempo finale di esecuzione
            self._end_time()
            i += 1  # Incremento per contare l'ultima iterazione
            return self.x, self._total_time(), self.relative_error(self.x), i
    # Se arrivo a max iterazioni
    if self.debug and not self.debug_jacobi:
        self._writeSolution()
    # Ottengo il tempo finale di esecuzione
    self._end_time()
    return self.x, self._total_time(), self.relative_error(self.x), self.iteration
```

Poichè il metodo di **Jacobi** converge se una matrice è a **dominanza diagonale stretta per righe**, subito dopo aver registrato l'inizio dell'esecuzione del metodo viene effettuato un controllo, ovvero se la matrice è a dominanza diagonale stretta per righe. Siccome esistono matrici non a dominanza diagonale stretta per righe che possono convergere, è stato scelto di riportare all'utente un semplice avvertimento di convergenza non garantita.

All'interno del ciclo for avvengono i seguenti passaggi:

- Calcolo del residuo r come segue:

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

- Aggiornamento del vettore x:

$$x^{(i+1)} = x^{(i)} + P^{-1}r^{(i)}$$

  La Matrice P^{-1} viene calcolata tramite la funzione `_getReverseP_jacobi()` di cui si discute nella sezione 3.3.2.

- Controllo di tolleranza tramite la funzione `_isTolerate()` (sezione 3.3.1).

Infine il metodo termina tramite il verificarsi della condizione di tolleranza oppure raggiungendo il numero massimo di iterazioni.

3.2.2 Metodo di Gauss-Seidel

```
"""
    FUNZIONE:  def gauss_seidel(self) -> tuple[Any, timedelta, float, int]:

    DESCRIZIONE:
    Funzione che implementa il Metodo di Gauss Seidel: x(k+1) = x(k) + P^(-1) r(k)

    Il metodo di Gauss Seidel e' una variante del metodo di Jacobi, dove vengono
    sfruttate
    le entrate del vettore x gia' calcolate, applicando per il calcolo di P^-1 la
    sostituzione in avanti.

    Come Jacobi, il metodo di Gauss Seidel converge sicuramente se la matrice
    utilizzata e' a
    dominanza diagonale stretta per righe

    La matrice e' assunta simmetrica e definita positiva

"""
def gauss_seidel(self) -> tuple[Any, timedelta, float, int]:
    # Per debugging
    if self.debug_gauss_seidel:
        self.prefix = '[DEBUG-GS] '
    # Definisco il vettore iniziale nullo come inizio del metodo iterativo
    self.x = _get_start_vector(self.column)
    # Ottengo il tempo iniziale di esecuzione
```

```
24      self._start_time()
25      # Calcolo la matrice triangolare inferiore P
26      P = self._getP_gauss()
27      if not self._row_diagonal_dominance():
28          print('[ALERT GS] La matrice fornita non e a dominanza diagonale stretta per
        righe -> Convergenza non assicurata.')
29      # Ciclo for parte da i=0 fino a iteration-1
30      for i in range(self.iteration):
31          # Se il debug e' attivo
32          if self.debug_gauss_seidel:
33              self._writeIteration(i)
34          # Calcolo il residuo come r(k) = b - Ax(k)
35          residual = self.b - self.matrix.dot(self.x)
36          # Ricavo y applicando la sostituzione in avanti
37          y = self._forward_substitution(P, residual)
38          # Aggiorno il valore della soluzione x(k+1) = x(k) + y
39          self.x = self.x + y
40          # Se arrivo alla tolleranza
41          if self._isTolerate(self.x, self.b):
42              if self.debug_gauss_seidel:
43                  self._writeTolerance()
44              # Ottengo il tempo finale di esecuzione
45              self._end_time()
46              i += 1  # Incremento per contare l'ultima iterazione
47              return self.x, self._total_time(), self.relative_error(self.x), i
48      # Se arrivo a max iterazioni
49      if self.debug and not self.debug_gauss_seidel:
50          self._writeSolution()
51      # Ottengo il tempo finale di esecuzione
52      self._end_time()
53      return self.x, self._total_time(), self.relative_error(self.x), self.iteration
```

Analogamente al metodo di **Jacobi** anche il metodo di **Gauss-Seidel** richiede la **Dominanza Diagonale Stretta per Righe** della matrice per assicurare la convergenza del metodo.
Il metodo di **Gauss Seidel** può essere visto come un miglioramento del metodo di **Jacobi**, dove vengono sfruttate le entrate del vettore $\mathbf{x}$ già calcolate nell'iterazione attuale, ovvero:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - a_{i,1}x_1^{(k+1)} - \cdots - a_{i,i-1}x_{i-1}^{(k+1)} - a_{i,i+1}x_{i+1}^{(k)} - \cdots - a_{i,n}x_n^{(k)} \right)$$

In particolare, l'implementazione del metodo vero e proprio è come segue:

- Calcolo della matrice triangolare inferiore P attraverso la funzione `_getP_gauss()`
- Nel ciclo:
  - Calcolo del residuo r come segue:

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

  - Calcolo della soluzione y attraverso `_forward_substitution()`:

$$P\mathbf{y} = \mathbf{r}^{(i)}$$

– Aggiornamento del vettore x:

$$x^{(i+1)} = x^{(i)} + y$$

– Controllo di tolleranza tramite la funzione `_isTolerate()` (sezione 3.3.1).

Infine il metodo termina tramite il verificarsi della condizione di tolleranza oppure raggiungendo il numero massimo di iterazioni.

3.2.3 Metodo del Gradiente

```
"""
    FUNZIONE:  gradient(self) -> tuple[Any, timedelta, float, int]:

    DESCRIZIONE:
    Funzione che implementa il Metodo del Gradiente: x(k+1) = x(k) + alpha(k) r(k)

    Il metodo del Gradiente e' un metodo iterativo non stazionario, dove lo scalare
    alpha dipendera' dalle iterazioni, invece di restare fisso. Esso si basa sulla
    ricerca di un punto di minimo per la risoluzione di sistemi lineari utilizzando
    il gradiente.

    La matrice e' assunta simmetrica e definita positiva

"""
def gradient(self) -> tuple[Any, timedelta, float, int]:
    # Per debugging
    if self.debug_gradient:
        self.prefix = '[DEBUG-G] '
    # Definisco il vettore iniziale nullo come inizio del metodo iterativo
    self.x = _get_start_vector(self.column)
    # Ottengo il tempo iniziale di esecuzione
    self._start_time()
    # Calcolo il residuo come r(k) = b - Ax(k)
    r = self.b - self.matrix.dot(self.x)
    # Calcolo alpha come r(k) trasposto r(k) / r(k) trasposto matrice r(k)
    alpha = (r.T.dot(r)) / (r.T @ self.matrix @ r)
    # Intero da i = 0 a iteration - 1
    for i in range(self.iteration):
        # Se il debug e' attivo
        if self.debug_gradient:
            self._writeIteration(i)
        # Aggiorno il valore x: x(k+1) = x(k) + alpha(k) r(k)
        self.x = self.x + alpha * r
        # Aggiorno r e alpha
        r = self.b - self.matrix.dot(self.x)
        alpha = (r.T.dot(r)) / (r.T @ self.matrix @ r)
        # Se arrivo alla tolleranza
        if self._isTolerate(self.x, self.b):
            if self.debug_gradient:
                self._writeTolerance()
            # Ottengo il tempo finale di esecuzione
            self._end_time()
            i += 1  # Incremento per contare l'ultima iterazione
            return self.x, self._total_time(), self.relative_error(self.x), i
    # Se arrivo a max iterazioni
    if self.debug and not self.debug_gradient:
```

```
44          self._writeSolution()
45      # Ottengo il tempo finale di esecuzione
46      self._end_time()
47      return self.x, self._total_time(), self.relative_error(self.x), self.iteration
```

Prima di eseguire il metodo del **Gradiente**, dato che esso è non stazionario, sono stati inizializzati i valori di $r^{(0)}$ e α_0 come segue:

$$r^{(0)} = b - Ax^{(0)}$$

$$\alpha_0 = \frac{(r^{(0)})^t r^{(0)}}{(r^{(0)})^t A r^{(0)}}$$

In seguito viene eseguito il ciclo for in cui si esegue l'aggiornamento di x, r e α e avviene il controllo di tolleranza, ovvero in successione si ha:

- $x^{(i+1)} = x^{(i)} + \alpha_i r^{(i)}$
- $r^{(i+1)} = b - Ax^{(i+1)}$
- $\alpha_{i+1} = \frac{(r^{(i+1)})^t r^{(i+1)}}{(r^{(i+1)})^t A r^{(i+1)}}$
- Controllo di tolleranza tramite la funzione `_isTolerate()` (sezione 3.3.1).

Infine il metodo termina tramite il verificarsi della condizione di tolleranza oppure raggiungendo il numero massimo di iterazioni.

3.2.4 Metodo del Gradiente Coniugato

```
1  """
2      FUNZIONE:  conjugate_gradient(self) -> tuple[Any, timedelta, float, int]:
3
4      DESCRIZIONE:
5      Implementa il Metodo del Gradiente Coniugato: x(k+1) = x(k) + alpha(k) d(k)
6
7      Il metodo del Gradiente Coniugato puo' essere visto come un miglioramento del
8      metodo del Gradiente dove si va a risolvere il problema della
9      convergenza a "zig-zag". Si vanno a cercare dei vettori ottimali che
10     non vengono piu' modificati lungo la direzione d.
11
12     La matrice e' assunta simmetrica e definita positiva
13
14 """
15 def conjugate_gradient(self) -> tuple[Any, timedelta, float, int]:
16     # Per debugging
17     if self.debug_conjugate_gradient:
18         self.prefix = '[DEBUG-CG] '
19     # Definisco il vettore iniziale nullo come inizio del metodo iterativo
20     self.x = _get_start_vector(self.column)
21     # Ottengo il tempo iniziale di esecuzione
22     self._start_time()
23     # Calcolo il residuo come r(k) = b - Ax(k)
24     r = self.b - self.matrix.dot(self.x)
25     # Assegno d = r
```

```
26      d = r
27      # Calcolo alpha come d(k) trasposto r(k) / d(k) trasposto matrice d(k)
28      alpha = (d.T.dot(r)) / (d.T @ self.matrix @ d)
29      # Intero da i = 0 a iteration - 1
30      for i in range(self.iteration):
31          # Se debug e' attivo
32          if self.debug_conjugate_gradient:
33              self._writeIteration(i)
34          # Aggiorno il valore x: x(k+1) = x(k) + alpha(k) d(k)
35          self.x = self.x + alpha * d
36          # Aggiorno il residuo
37          r = self.b - self.matrix.dot(self.x)
38          # Calcolo beta come d(k) trasposto matrice r(k+1) / d(k) trasposto matrice d(
    k)
39          beta = (d.T @ self.matrix @ r) / (d.T @ self.matrix @ d)
40          # Aggiorno d(k+1) = r(k+1) - beta(k) * d(k)
41          d = r - beta* d
42          # Aggiorno alpha come d(k) trasposto r(k) / d(k) trasposto matrice d(k)
43          alpha = (d.T.dot(r)) / (d.T @ self.matrix @ d)
44          # Se arrivo alla tolleranza
45          if self._isTolerate(self.x, self.b):
46              if self.debug_conjugate_gradient:
47                  self._writeTolerance()
48              # Ottengo il tempo finale di esecuzione
49              self._end_time()
50              i += 1
51              return self.x, self._total_time(), self.relative_error(self.x), i
52      # Se arrivo a max iterazioni
53      if self.debug and not self.debug_conjugate_gradient:
54          self._writeSolution()
55      # Ottengo il tempo finale di esecuzione
56      self._end_time()
57      return self.x, self._total_time(), self.relative_error(self.x), self.iteration
```

Il metodo del **Gradiente Coniugato** differisce rispetto al metodo del **Gradiente** per l'introduzione di una direzione ottimale d e di un coefficiente β utili ad evitare l'andamento a *"zig-zag"* tipico del metodo del Gradiente. Questo metodo permette di ottenere un limite superiore al numero di iterazioni necessarie per l'ottenimento della soluzione esatta pari alla dimensione della matrice di input, con una generale diminuzione di iterazioni.
Prima di entrare nel ciclo for sono stati inizializza i seguenti valori:

- $r^{(0)} = b - Ax^{(0)}$
- $d^{(0)} = r^{(0)}$
- $\alpha_0 = \frac{(d^{(0)})^t r^{(0)}}{(d^{(0)})^t A d^{(0)}}$

Successivamente nel ciclo for vengono aggiornati i valori di x, r, d, α e β e infine eseguito il controllo di tolleranza. In successione vi sono le seguenti operazioni:

- $x^{(i+1)} = x^{(i)} + \alpha_i d^{(i)}$
- $r^{(i+1)} = b - Ax^{(i+1)}$

- $\beta_{i+1} = \frac{(d^{(i)})^t A r^{(i+1)}}{(d^{(i)})^t A d^{(i)}}$
- $d^{(i+1)} = r^{(i+1)} - \beta d^{(i)}$
- $\alpha_{i+1} = \frac{(d^{(i+1)})^t r^{(i+1)}}{(d^{(i+1)})^t A d^{(i+1)}}$
- Controllo di tolleranza tramite la funzione `_isTolerate()` (sezione 3.3.1).

Infine il metodo termina tramite il verificarsi della condizione di tolleranza oppure raggiungendo il numero massimo di iterazioni.

3.3 Altri Metodi Utili

3.3.1 Controllo della Tolleranza

```
"""
    FUNZIONE: _isTolerate(self, x, b):

    DESCRIZIONE:
    Funzione che si occupa di ricalcolare il residuo e verificare
    se la norma del residuo / la norma di b e' minore della tolleranza.
    Nel caso affermativo, e' il criterio di arresto delle iterazioni

"""
def _isTolerate(self, x, b):
    residual = self.matrix.dot(x) - b
    norm1 = np.linalg.norm(residual)
    norm2 = np.linalg.norm(b)
    return (norm1 / norm2) < self.tol
```

Questa funzione serve per controllare, ritornando *True* o *False*, se:

$$\frac{\|Ax - b\|}{\|b\|} < tol$$

L'esecuzione della funzione prevede:

- Calcolo del residuo: $residual = Ax - b$
- Calcolo della norma di $residual$ e di b rispettivamente salvate in $norm1$ e $norm2$
- Return della valutazione dell'espressione booleana:

$$\frac{norm1}{norm2} < tol$$

3.3.2 Matrice inversa di P - Metodo di Jacobi

```
"""
    FUNZIONE:  _getReverseP_jacobi(self):

    DESCRIZIONE:
    Funzione che si occupa di calcolare P^(-1), dove P e' la
    diagonale della matrice e P^(-1) la sua inversa, facilmente
    calcolabile dato che sara' composta dai reciproci degli
    elementi sulla diagonale

"""
def _getReverseP_jacobi(self):
    # Calcolo P^(-1)
    reverse_P = ss.diags([1 / self.matrix.diagonal()], [0])
    # Converto al formato sparso CSR nel caso non lo sia
    if not ss.isspmatrix_csr(reverse_P):
        reverse_P = reverse_P.tocsr()
    return reverse_P
```

Poichè la matrice P nel metodo di **Jacobi** è diagonale, la sua inversa sarà data dal reciproco degli elementi sulla diagonale di P. Sia $p_{i,i}$ il generico elemento diverso da 0 nella matrice P sulla diagonale. L'elemento generico nella matrice P^{-1} diverso da 0 è:

$$\frac{1}{p_{i,i}}$$

3.3.3 Matrice P - Metodo di Gauss Seidel

```
"""
    FUNZIONE:  _getP_gauss(self):

    DESCRIZIONE:
    Calcola la matrice P del metodi di Gauss Seidel, definita come la matrice
    triangolare
    inferiore della matrice matrix

"""
def _getP_gauss(self):
    P = np.tril(self.matrix.toarray())
    return P
```

Ritorna la matrice P del metodo di **Gauss Seidel**. Essa è definita come la parte triangolare inferiore della matrice iniziale *matrix*:

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

con $a_{i,j}$ componenti della matrice *matrix*.

3.3.4 Sostituzione in Avanti - Forward Substitution

```
"""
    FUNZIONE:  _forward_substitution(self, P, residual):

    DESCRIZIONE:
    Implementa l'algoritmo di sostituzione in avanti, al fine di evitare
    la computazione di P^-1 per l'implementazione di Gauss-Seidel

"""
def _forward_substitution(self, P, residual):
    # Dimensione del sistema
    n = P.shape[0]
    # Inizializzazione a tutti zeri di x
    x = _get_start_vector(n)
    # Controllo se il primo termine e' uguale a zero (determinante = 0)
    if P[0, 0] == 0:
        raise ValueError("[ERROR-GS] Il primo termine deve essere diverso da zero!")
    # Intero su n
    for i in range(n):
        # Se un elemento sulla diagonale e' zero (determinante = 0)
        if P[i, i] == 0:
            raise ValueError("[ERROR-GS] La matrice e' singolare, non puo' essere
    invertita.")
        # Calcolo x(i) come r(i) - P(i,:)*x / P(i,i)
        x[i] = (residual[i] - P[i, :].dot(x)) / P[i, i]
    # Ritorno la soluzione della sostituzione in avanti
    return x
```

Implementazione in funzione del metodo risolutivo diretto di sostituzione in avanti, necessario per la costruzione del metodo di **Gauss Seidel**. Affinchè la matrice triangolare inferiore passata in input sia risolubile, è stata controllata la presenza degli zeri sulla diagonale e sul primo elemento della prima riga.

3.3.5 Controllo Matrice a Dominanza Diagonale Stretta per Righe

```
"""
    FUNZIONE:  _row_diagonal_dominance(self):

    DESCRIZIONE:
    Funzione che si occupa di verificare se la matrice utilizzata
    e' a dominanza diagonale stretta per righe, ovvero se i valori sulla
    diagonale a(i,i) in abs di una matrice sono maggiori della somma dei abs
    dei valori della stessa riga escluso a(i,i)

"""
def _row_diagonal_dominance(self):
    for i in range(self.matrix.shape[0]):
        tot = 0
        elem = abs(self.matrix[i, i])
        for j in range(self.row):
            tot += abs(self.matrix[i, j])
        if elem <= tot:
            return False
    return True
```

Questa funzione serve a verificare che una Matrice $A \in \mathbb{R}^{n \times n}$ sia a Dominanza Diagonale Stretta per Righe.

Teorema 1 *Sia* $A \in \mathbb{R}^{n \times n}$. *A è a* ***Dominanza Diagonale Stretta per Righe*** $\iff$

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

Questo serve a garantire ai due metodi iterativi stazionari della libreria di verificare se la convergenza è garantita o meno.

3.3.6 Calcolo dell'Errore Relativo

```
"""
    FUNZIONE: relative_error(self, x):

    DESCRIZIONE:
    Funzione che si occupa di calcolare l'errore relativo tra la soluzione
    ottenuta x e la soluzione esatta x' attraverso la formula:
    norma(x - x') / norma(x)

"""
def relative_error(self, x):
    return np.linalg.norm(x - _get_solution(self.column)) / np.linalg.norm(x)
```

`get solution(self.column)` è una funzione che si occupa di ritornare un array di dimensione *self.column* di [1..1], ovvero la soluzione esatta.

Definizione Sia $\mathbf{x_h}$ la soluzione approssimata di un sistema lineare $A\mathbf{x} = \mathbf{b}$ e sia $\tilde{\mathbf{x}}$ la soluzione esatta, l'*errore relativo* è

$$e_{\text{rel}} = \frac{\|\mathbf{x_h} - \tilde{\mathbf{x}}\|}{\|\tilde{\mathbf{x}}\|}$$

dove $\|\cdot\|$ è una qualsiasi norma fra vettori.

3.3.7 Altre Funzioni

In seguito le altre funzioni utilizzate dalla libreria per una miglior gestione dei compiti nel codice. Sono state omesse le funzioni di debugging e scrittura di logs.

```
"""
    FUNZIONE:  _start_time(self):

    DESCRIZIONE:
    Funzione che si occupa di ottenere la data corrente.
    Utilizzata come tempo di riferimento iniziale per l'esecuzione
    di un metodo risolutivo

"""
def _start_time(self):
```

```
11      self.start = datetime.now()
12
13  """
14      FUNZIONE:  _end_time(self):
15
16      DESCRIZIONE:
17      Funzione che si occupa di ottenere la data corrente.
18      Utilizzata come tempo di riferimento finale per l'esecuzione
19      di un metodo risolutivo
20
21  """
22  def _end_time(self):
23      self.end = datetime.now()
24
25  """
26      FUNZIONE:  _total_time(self):
27
28      DESCRIZIONE:
29      Funzione che si occupa di calcolare il tempo totale di
30      elaborazione di un metodo
31
32  """
33  def _total_time(self):
34      return self.end - self.start
35
36  """
37      FUNZIONE:  _get_start_vector(dim):
38
39      DESCRIZIONE:
40      Funzione che si occupa di ritornare un array di dimensione dim di [0..0]
41      E' il vettore nullo
42
43  """
44  def _get_start_vector(dim):
45      return np.zeros(dim)
46
47  """
48      FUNZIONE: _get_solution(dim):
49
50      DESCRIZIONE:
51      Funzione che si occupa di ritornare un array di dimensione dim di [1..1]
52      E' la soluzione esatta
53
54  """
55  def _get_solution(dim):
56      return np.ones(dim)
```

3.4 Debug

Nella sezione di `debug` è stato possibile testare i metodi e averne una situazione della memoria ad ogni iterazione, in modo da verificare la corretta implementazione del metodo.
Ai fini del progetto è importante evidenziare come, tramite la funzione di debug, sia stato possibile raccogliere tutti i dati che successivamente riporteremo e analizzeremo.
All'interno del `main.py` è possibile eseguire tutti i metodi per tutte le matrici proposte con ogni tipo di tolleranza.

Invece in `methods.py` la funzione di debug si occupa di riportare per ogni iterazione il vettore x, la tolleranza se questa è soddisfatta e stampare la soluzione nel caso si raggiunga il massimo numero di iterazioni. In più, grazie alla libreria di SciPy è possibile, in quest'ultimo caso, verificare la corretta soluzione del sistema anche sapendo essere un vettore di solo 1.

4 Risultati Ottenuti e Considerazioni

In seguito sono riportati in diverse tabelle i risultati ottenuti dall'esecuzione dei quattro metodi risolutivi, ottenute dal file *"risultati.txt"*, ed alcune osservazioni generali sul comportamento di ogni metodo. Le tabelle sono suddivise per la matrice utilizzata, dove per ognuna vi è riportato il nome della matrice. Inoltre, le tabelle sono state organizzate in sezioni per metodo risolutivo. Ogni tabella riporta le seguenti informazioni:

- *Tolleranza*
- *Tempo Impiegato in Secondi*
- *Errore Relativo*
- *Numero di Iterazioni*

4.1 Metodo di Jacobi

Il primo metodo analizzato è il Metodo di **Jacobi**, che si comporta discretamente bene per le matrici di tipo *"spa"*, mostrando una crescita accettabile delle iterazioni e del tempo impiegato al diminuire della tolleranza, ma con un miglior errore relativo. Per le matrici di tipo *"vem"* vi è una crescita importante delle iterazioni già a partire dalla tolleranza più grande.

Matrice 0: "spa1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.025004000	0.0017708118132163102	115
1e-06	0.036205000	1.7979247341351328e-05	181
1e-08	0.045000000	1.8249788268049072e-07	247
1e-10	0.057003000	1.8524371366782515e-09	313

Matrice 1: "spa2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.089516000	0.001766716406295179	36
1e-06	0.137517000	1.6667519292204132e-05	57
1e-08	0.187386000	1.572869932678427e-07	78
1e-10	0.194018000	1.484271702680792e-09	99

Matrice 2: "vem1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.054511000	0.0035503020437906226	1314
1e-06	0.074509000	3.5401723469538816e-05	2433
1e-08	0.100600000	3.5397669465902384e-07	3552
1e-10	0.143018000	3.539458754943593e-09	4671

Matrice 3: "vem2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.090506000	0.004988120299885085	1927
1e-06	0.139013000	4.9672303664677925e-05	3676
1e-08	0.196015000	4.965609860645761e-07	5425
1e-10	0.251038000	4.964185239985714e-09	7174

4.2 Metodo di Gauss-Seidel

Passando all'analisi dei risultati del metodo di **Gauss-Seidel** è prevista una diminuzione generale delle iterazioni rispetto al metodo di **Jacobi**, ciò infatti si è verificato mostrando un grosso decremento delle iterazioni complessive per le quattro matrici trattate, ma con un aumento, abbastanza importante per quanto riguarda le matrici *"vem"*, del tempo impiegato. Infine si può notare come l'errore relativo sia aumentato rispetto a **Jacobi** se si considerano matrici *"spa"*.

Matrice 0: "spa1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.029000000	0.01819346991107113	9
1e-06	0.030016000	0.00012996891625549983	17
1e-08	0.036510000	1.709732816046525e-06	24
1e-10	0.044506000	2.2480878875168565e-08	31

Matrice 1: "spa2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.096609000	0.002598794541822277	5
1e-06	0.127209000	5.1416406529276493e-05	8
1e-08	0.165530000	2.794322033091081e-07	12
1e-10	0.178019000	5.570741072640405e-09	15

Matrice 2: "vem1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	1.421263000	0.0035167043466634864	659
1e-06	2.644127000	3.526795076484606e-05	1218
1e-08	3.497670000	3.517457973952495e-07	1778
1e-10	4.650932000	3.508242366381709e-09	2338

Matrice 3: "vem2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	4.353978000	0.004970707700261841	965
1e-06	8.41723400	4.941955175880287e-05	1840
1e-08	12.093371000	4.958371641950137e-07	2714
1e-10	16.102473000	4.948912858660209e-09	3589

4.3 Metodo del Gradiente

Il metodo del **Gradiente** risulta molto interessante rispetto ai primi due metodi risolutivi. Si può notare come ci sia stata un'inversione delle prestazioni rispetto a **Jacobi** e **Gauss-Seidel**. I primi due, infatti, mostravano degli ottimi risultati su matrici di tipo *"spa"* con un incremento delle metriche considerevole per le matrici di tipo *"vem"*, mentre per il metodo del **Gradiente** vi è sostanzialmente l'opposto. Per le matrici di tipo *"spa"* risulta un incremento davvero elevato di iterazioni, tempo impiegato ed errore relativo al diminuire della tolleranza.

Matrice 0: "spa1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.039589000	0.034612824950369735	143
1e-06	1.113897000	0.0009680780107461574	3577
1e-08	2.839827000	9.816366873577376e-06	8233
1e-10	4.017760000	9.820388475415228e-08	12919

Matrice 1: "spa2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.468621000	0.01814126247276902	161
1e-06	5.505509000	0.0006694284158037972	1949
1e-08	14.347439000	6.865240652697673e-06	5087
1e-10	21.259285000	6.937815012078666e-08	8285

Matrice 2: "vem1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.043028000	0.0027103595777947975	890
1e-06	0.073218000	2.713397344825607e-05	1612
1e-08	0.105509000	2.6953379149684406e-07	2336
1e-10	0.136509000	2.7131686294531124e-09	3058

Matrice 3: "vem2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.164015000	0.0038234939449239354	1308
1e-06	0.308033000	3.791533139770802e-05	2438
1e-08	0.217540000	3.809851830056432e-07	3566
1e-10	0.270936000	3.798772480516601e-09	4696

4.4 Metodo del Gradiente Coniugato

Infine, per l'analisi del metodo del **Gradiente Coniugato** è attesa, come nel caso di **Gauss-Seidel**, una diminuzione delle iterazioni effettuate, in quanto si è andato ad eliminare il problema del *zig-zag* del metodo del **Gradiente**. Questo avviene, riportando le miglior prestazioni sulle matrici di tipo *"vem"* rispetto agli altri metodi e discrete prestazioni per matrici di tipo *"spa"*, avvicinandosi ai metodi di **Jacobi** e **Gauss-Seidel**.

Matrice 0: "spa1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.025942000	0.020800035393044685	49
1e-06	0.075459000	2.5529093554004556e-05	134
1e-08	0.110094000	1.3198446196533585e-07	177
1e-10	0.113013000	1.2160866949314527e-09	200

Matrice 1: "spa2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.202126000	0.009822497323405248	42
1e-06	0.543581000	0.0001197984694267039	122
1e-08	0.933782000	5.586660596081778e-07	196
1e-10	1.039938000	5.324230565910014e-09	240

Matrice 2: "vem1.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.004003000	4.082793737440566e-05	38
1e-06	0.003999000	3.7323397023331895e-07	45
1e-08	0.004003000	2.831873451203778e-09	53
1e-10	0.005001000	2.191751450892725e-11	59

Matrice 3: "vem2.mtx"			
Tolleranza	Tempo Impiegato	Errore Relativo	Iterazioni
1e-04	0.011002000	5.72901895537801e-05	47
1e-06	0.013000000	4.742996283562162e-07	56
1e-08	0.0070000004	4.29998352800121e-09	66
1e-10	0.008001000	2.2476275108448333e-11	74