Francesco Cavallini

Matricola: 920835
f.cavallini8@campus.unimib.it
Corso di studi di Informatica Magistrale

Università degli studi Milano Bicocca
Facoltà di informatica magistrale
Milano, Padiglione U24

Anno accademico 23/24

Easy Matrix
Metodi del Calcolo Scientifico

Relazione Progetto 1-Bis

Consegna:

“““

Si utilizzi un linguaggio di programmazione a vostra scelta: c++, fortran, java, pyton, etc. Lo scopo del progetto è quello di implementare una mini libreria che esegua i seguenti solutori iterativi, limitatamente al caso di matrici simmetriche e definite positive:
(1) metodo di Jacobi;
(2) metodo di Gauß-Seidel;
(3) metodo del Gradiente;
(4) metodo del Gradiente coniugato.

”””

Riferimenti :

Repository git-hub:	https://github.com/VR3ED/Easy_Matrix
Download libreria:	https://www.nuget.org/packages/easy_matrix/1.0.0

Sommario

0. Introduzione

0.1. Obbiettivi

Come definito nella pagina iniziale, l'obbiettivo del progetto è quello di implementare una mini libreria che esegua una serie di solutori iterativi, limitatamente al caso di matrici simmetriche e definite positive.

0.2. Scelte implementative

Si sceglie di suddividere lo sviluppo del progetto in 2 parti:

1. Implementazione della libreria
2. Paragoni con librerie open source ed analisi dei risultati.

Per lo sviluppo della prima parte si sceglie di usare il linguaggio C# utilizzando il framework .Net 8.0 per i seguenti motivi:

- **Facilità di pubblicazione libreria:**
  Microsoft offre la possibilità di pubblicare gratuitamente la propria libreria sviluppata usando il linguaggio C# sulla piattaforma NuGet. Da questa piattaforma la suddetta libreria potrà venire scaricata, incorporata ed utilizzata in qualsiasi progetto tramite un semplice click al seguente link: https://www.nuget.org/packages/easy_matrix/1.0.0

- **Gestione alta precisione:**
  Il linguaggio C# permette la gestione dei numeri a virgola mobile con il tipo decimal. Il tipo decimal, seppure permetta di rappresentare solo numeri più piccoli, ha la massima precisione tra i tre tipi a virgola mobile:

C# type/keyword	Approximate range	Precision
float	$\pm 1.5 \text{ x } 10^{-45}$ to $\pm 3.4 \text{ x } 10^{38}$	~6-9 digits
double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits
decimal	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \text{ x } 10^{28}$	~28-29 digits

Per lo sviluppo della seconda parte si sceglie di fare paragoni con performance di librerie esterne utilizzando il linguaggio python in quanto presenta un amplia gamma di librerie e (come vedremo) codice open-source che renderanno possibile il paragone con la libreria sviluppata al punto 1. Inoltre verrà utilizzato python anche per la formulazione di grafici per il paragone delle statistiche di performance tra libreria sviluppata e codice open source.

1. Parte-1: Sviluppo della libreria

1.1. Requisiti:

I requisiti che rispetteremo per lo sviluppo delle librerie sono i seguenti:

1. La libreria deve essere in grado di operare con ciascuno dei metodi iterativi sopra menzionati. Inoltre, in caso di utilizzo di librerie esterne, queste librerie devono fornire solamente la struttura dati di matrici e vettori, senza utilizzare i metodi relativi alla risoluzione dei sistemi lineari già implementati al suo interno.

2. I metodi iterativi devono partire da un vettore iniziale nullo e arrestarsi quando la k-esima iterata x(k) soddisfa la condizione:

$$\frac{||Ax^{(k)} - b||}{||b||} < tol$$

   Dove tol è una tolleranza assegnata dall'utente. Per eseguire i nostri test (che mostreremo nella parte-2) verranno usate $tol = [10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}]$. Alternativamente, laddove non si dovesse raggiungere la convergenza entro un numero massimo di iterazioni (maxIter, dove maxIter è un numero elevato a scelta (non inferiore a 20000)), la computazione del metodo iterativo dovrà cessare e restituire in output:

   convergence = false

3. La libreria deve avere un'architettura ben strutturata anziché essere una sequenza di funzioni indipendenti.

1.2. Le basi per lo sviluppo:

Nell'introduzione abbiamo accennato al fatto che tutte le matrici (A) che daremo in pasto agli algoritmi di libreria sono simmetriche e definite positive; abbiamo quindi che per ogni matrice (A) per essere **simmetrica e definita positiva** deve rispettare i seguenti criteri:

- La matrice (A) è simmetrica (ossia $A = A^t$)
- La matrice A è definita positiva (ossia per ogni vettore colonna $x \in \mathbb{R}^n \setminus \{0\}$, soddisfa la seguente condizione: $x^t \times A \times x > 0$

NOTA: Metodo di Cholesky

Il metodo di Cholesky permette di scomporre una matrice simmetrica e definita positiva (A) in

$$A = R \times R^t$$

Dove R ed R^t sono rispettivamente una matrice triangolare inferiore e superiore

Nella libreria, all'interno della classe AccurateMatrix (che serve per rappresentare una matrice in formato decimal), è stata infatti inserita una funzione apposta per verificare se le matrici lette in input sono simmetriche e definite positive, ma al posto che controllare se la matrice è definita positiva, si prova (tramite codice) invece ad applicare Cholesky, se non ci si riesce allora vuol dire che la matrice non è simmetrica e definita positiva:

```
public bool IsSymmetricPositiveDefinite()
{
    // Verify squared natrix
    if (rows != columns)
        return false;

    // Check for symmetry
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            if (matrix[i, j] != matrix[j, i])
                return false;
        }
    }

    create a copy of the matrix

    // Cholesky decomposition on copied matrix
    for (int i = 0; i < rows; i++)
    {
        decimal sum = 0;
        for (int k = 0; k < i; k++)
        {
            sum += tempMatrix[i, k] * tempMatrix[i, k];
        }

        decimal diagValue = tempMatrix[i, i] - sum;
        if (diagValue <= 0)
            return false;

        tempMatrix[i, i] = Sqrt(diagValue);
        for (int j = i + 1; j < rows; j++)
        {
            sum = 0;
            for (int k = 0; k < i; k++)
            {
                sum += tempMatrix[j, k] * tempMatrix[i, k];
            }
            tempMatrix[j, i] = (tempMatrix[j, i] - sum) / tempMatrix[i, i];
        }
    }

    return true;
}
```

Verifica se la matrice è simmetrica comparando $A[i,j]$ con $A[j,i]$ dentro un doppio loop

Verifica se l'elemento diagonale calcolato nella decomposizione di Cholesky è positivo, ossia prima si calcola $sum = \sum_{k=0}^{i-1} (A[i,k])^2$ poi si calcola il quadrato del valore diagonale: $R[i,i]^2 = A[i,i] - sum$ (se questo valore è >0 possiamo procedere)

Possiamo ora calcolare:

- $R[i,i] = \sqrt{diagValue}$
- $R[j,i] = \frac{A[j,i] - (\sum_{k=0}^{i-1} R[j,k], R[i,k])}{A[i,i]}$

In questo modo si calcola Cholesky per ogni iterazione i

all'interno della classe AccurateMatrix, oltre al metodo appena descritto sono ovviamente presenti i seguenti attributi il quale utilizzo è auto-esplicativo:

```
13 riferimenti
internal decimal[,] matrix { get; set; }
25 riferimenti
public int rows { get; set; }
20 riferimenti
public int columns { get; set; }
5 riferimenti
public int number_of_valorized { get; set; }
4 riferimenti
public string matrix_name { get; set; }
```

- Matrix: Conterrà la matrice target A in formato decimal
- Rows: Conterrà il numero di righe della matrice A
- Columns: Conterrà il numero di colonne della matrice A
- Number_of_valorized: indica il numero totale di valori diversi da 0 all'interno di A
- Matrix_name: stringa (non necessaria all'inizializzazione) usata per assegnare un nome ad ogni istanza di matrice

NOTA IMPLEMENTATIVA: Lettura delle matrici da file .mtx

In C# non esistono metodi per leggere un file .mtx ed assegnare i valori ad un tipo decimal, quindi, in verità, prima di definire se una matrice è simmetrica e definita positiva, la prima sfida implementativa è stata trovare una soluzione per permettere alla libreria di poter leggere i file. Questa sfida è stata poi superata utilizzando il multi-threading per leggere il file riga per riga, come fosse un file txt, e creare un parser custom che permettesse di convertire la stringa letta in un valore di tipo decimal (e poi memorizzare il valore nella cella corretta della matrice).

Questo metodo di lettura della matrice (che viene richiamato dal costruttore della classe AccurateMatrix) non verrà però approfondito in questa relazione in quanto si reputa essere fuori dallo scope della relazione stessa, ma si possono trovare più informazioni su questo metodo sulla repository di git-hub (qui)

Avendo quindi la possibilità di leggere file .mtx e verificare che le matrici lette siano simmetriche e definite positive il prossimo passo è quello dell'implementazione dei **solutori iterativi**.
Più nello specifico abbiamo che: un solutore iterativo è un metodo particolare utilizzato per risolvere sistemi di equazioni lineari (rappresentati dalle nostre matrici). Questo metodo genera una successione di approssimazioni (iterativamente) per trovare la soluzione del sistema lineare, migliorando iterazione per iterazione la stima di soluzione; appunto la caratteristica distintiva dei metodi iterativi è che la stessa operazione (o uno stesso insieme di operazioni) viene applicata ripetutamente a ogni iterazione per migliorare l'approssimazione. La convergenza dei metodi iterativi viene raggiunta quando viene raggiunta la soluzione esatta:

$$Ax = b$$

oppure quando (all'iterazione k) viene raggiunta un'approssimazione abbastanza vicina alla soluzione esatta:

$$\frac{||Ax^{(k)} - b||}{||b||} < tol$$

Ossia si calcola l'errore relativo $\left(ER = \frac{||Ax^{(k)}-b||}{||b||}\right)$ e si verifica che questo sia al di sotto di una certa soglia di tolleranza tol (come anche richiesto da consegna).

Abbiamo quindi che tutti i metodi iterativi che sono stati implementati hanno lo stesso comportamento comune di base:

- Ciclare fino a raggiungere $Ax = b$ (oppure fino all'iterazione $maxIter$)
- All'interno del ciclo:
  - Calcolare un vettore delle incognite approssimato $x^{(k)}$ (la quale logica di calcolo cambia per ogni metodo iterativo) necessaria per calcolare l'errore relativo.
  - Verificare la condizione di uscita (ossia verificare se $ER < tol$) , in caso positivo interrompere il ciclo e mandare in output il vettore $x^{(k)}$ delle incognite approssimato (in modo da risolvere il sistema lineare come $Ax^{(k)} = b$)

Abbiamo, dunque, che per implementare tutti i solutori iterativi richiesti verranno sfruttare le proprietà del linguaggio di programmazione C# ed implementare una classe astratta (che non può essere istanziata) chiamata IterativeSolver con i seguenti attributi:

```
/// <summary>
/// Decimals matrix
/// </summary>
protected AccurateMatrix A;

/// <summary>
/// vecror of known terms
/// </summary>
protected decimal[] b;

/// <summary>
/// tollerance index. Up to 28-29 digits precision
/// </summary>
protected decimal tol;

/// <summary>
/// maximun number of iterations required
/// </summary>
protected int maxIter;
```

Inoltre la classe implementa al suo interno il metodo Solve() che descrive il comportamento generale di un solutore iterativo (appena descritto):

```
public decimal[] Solve()
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    int n = A.rows;
    decimal[] x = new decimal[n]; //vetto

    for (int k = 0; k < maxIter; k++)
    {
        #region  applicazione solver logic

        // Parallelizing the inner loop to calculate each element of x independently
        // tutte le iterazioni tranne la prima e l'ultima vengono eseguite in parallelo
        // questo perchè per la prima bisogna fare il setup e l'ultima bisogna testare la exit condition
        x = SolverLogic(0, x);
        Parallel.For(1, n-1, i =>
        {
            x = SolverLogic(i, x);
        });
        x = SolverLogic(n-1, x);

        #endregion

        // Check for the exit condition
        if (SolverExitCondition(x))
        {
            stopwatch.Stop();
            LogResults(true, tol, k, stopwatch.Elapsed);
            return x;
        }
    }

    LogResults(false, tol, maxIter, stopwatch.Elapsed);
    throw new Exception("Iterative method did not converge.");
}
```

Si cicla da 0 a maxIter, dove, per consegna, maxIter è un parametro determinato dall'utente per avere un numero di iterazioni massime

Per i da 0 a $n-1$ (dove n=numero di righe della matrice) si calcola una cella del vettore delle incognite approssimato, ossia si calcola $x[i]^{(k)}$ utilizzando la SolverLogic nativa di ogni classe figlia

Nella funzione SolverExitCondition si calcola l'errore relativo e lo si paragona con tol. Se questa funzione ritorna true allora il metodo iterativo converge e si ritorna in output il vettore $x^{(k)}$

Se entro MaxIter iterazioni il metodo non converge si ritorna in un eccezione per segnalare la non convergenza

Si vuole fare notare che il metodo precedentemente descritto non è astratto, il che vuol dire che tutte le classi solver figlie della classe IterativeSolver erediteranno anche questo metodo. Ma, invece, i metodi SolverLogic e SolverExitCondition sono stati **definiti astratti**, il che vuol dire che ogni classe figlia di IterativeSolver dovrà implementarli in maniera diversa.
Mantenendo questa struttura abbiamo quindi che ogni solutore eseguirà la stessa struttura base di IterativeSolver per eseguire il metodo Solve ma applicherà una logica di calcolo di soluzione approssimata $x^{(k)}$ diversa per ogni classe figlia.

NOTA IMPLEMENTATIVA: Perché PerSolverExitCondition è un metodo astratto?

Come abbiamo detto precedentemente, anche SolverExitCondition fa' parte della logica condivisa da tutti i solver iterativi, eppure lo abbiamo definito come metodo astratto, il che vuol dire che ogni classe avrà una definzione diversa per il metodo.
Sembrerebbe una contraddizione, invece è semplicemente stato necessario definire questo metodo in questo modo perché (nonostante tutte le condizioni di uscita rimarranno uguali per tutti i solutori, come definito da consegna) alcuni solutori hanno bisogno di eseguire delle operazioni di setup per la prossima iterazione all'interno di questo metodo.

Pertanto, nonostante tutte le exit condition di tutti i solutori rimarranno uguali, avremo che si potrebbero presentare delle piccole differenze all'interno dei metodi di SolverExitCondition per puri motivi tecnici di limiti di flessibilità del linguaggio C#

Una volta chiarita qual'è la struttura base di ogni solutore utilizza per trovare una soluzione (con il metodo Solve) possiamo ora passare alla descrizione vera e propria dei metodi SolverLogic e SolverExitCondition delle 4 classi che estendo il comportamento di IterativeSolver, ossia le classi:

- JacobiSolver
- GaussSeidelSolver
- GradientSolver
- ConjugateGradientSolver

1.3. Implementazione JacobiSolver

L'idea di base del metodo di Jacobi è quella di risolvere ogni equazione del sistema per ciascuna delle incognite, esprimendo ogni incognita in termini delle altre incognite e dei termini noti, aggiornando poi ad ogni iterazione i valori calcolati (x^k).
Abbiamo infatti che il metodo di Jacobi sfrutta una scomposizione di A (detta **slitting**) che permette di suddividere la matrice A in 2 matrici più facili da memorizzare:

$$A = P - N$$

dove:

- P: Matrice diagonale contenente gli elementi diagonali di A.
- N: Matrice contenente gli elementi non diagonali di A, cioè $N = P - A$.

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

(a) mat_A_jacobi

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

(b) mat_P_jacobi

$$N := \begin{bmatrix} 0 & -a_{1,2} & \cdots & -a_{1,n} \\ -a_{2,1} & 0 & \cdots & -a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n,1} & -a_{n,2} & \cdots & 0 \end{bmatrix}.$$

(c) mat_N_jacobi

In seguito a questa scomposizione si ha che il sistema $Ax = b$ può essere scritto come:

$$(P - N)x = b \quad \rightarrow \quad Px = b + Nx$$

Abbiamo infatti che la formula di per ottenere x in seguito alla scomposizione è:

$$x^{(k)} = \frac{\left(b + Nx^{(k-1)}\right)}{P}$$

dove:

- $x^{(k)}$: vettore delle incognite all'iterazione k.
- $x^{(k-1)}$: vecchio vettore delle incognite all'iterazione precedente $(k-1)$

Il che vuol dire che se vogliamo computare (per ogni iterazione k) ogni componente $x[i]$ allora possiamo aggiornarlo separatamente con questa formula:

$$x[i]^{(k)} = \frac{\left(b[i] - \sum_{j \neq i} A[i,j] \times x[j]^{(k-1)}\right)}{A[i,i]}$$

Nota che qui il segno è cambiato perché si stanno usando i valori di A non di N

dove:

- $A[i,i]$: Termine diagonale della matrice A, che corrisponde a $P[i,i]$.
- $A[i,j]$: Termine non diagonale. Che corrisponde a $-N[i,j]$

Se andiamo a visualizzare quindi l'implementazione del metodo SolverLogic all'interno della classe JacobiSolver possiamo vedere che implementerà esattamente la formula di aggiornamento di ogni componente $x[i]$ appena descritta:

```
public override decimal[] SolverLogic(int i, decimal[] x)
{
    init vector x and xOld

    decimal sigma = 0;
    for (int j = 0; j < base.A.rows; j++)
    {
        if (j != i)
        {
            //sommatoria di tutte le A[i,j] * x[j]^k
            sigma += A.matrix[i, j] * xOld[j];
        }
    }

    //formula per calcolo di jacobi
    x[i] = (b[i] - sigma) / A.matrix[i, i];

    //arrivato alla fine mi salvo il nuvo xOld
    if (i == A.rows-1)
    {
        Array.Copy(x, xOld, A.rows);
    }

    return x;
}
```

Calcolo della sommatoria

$$sigma = \sum_{j \neq i} A[i,j] \times x[j]^{(k-1)}$$

E salvataggio nella variabile 'sigma'

Calcolo aggiornamento $x[i]^{(k)}$:

$$x[i]^{(k)} = \frac{(b[i] - sigma)}{A[i,i]}$$

Aggiornamento della variabile 'xOld' che corrisponde a $x^{(k-1)}$ per la prossima iterazione

Nota che per permettere ad ogni $x[i]$ di avere il valore corretto di xOld salvato alla fine del calcolo $x[i-1]$ si salva questo valore come attributo della classe:

```
/// <summary>
/// static variable that memorizes xOld for each iteration
/// </summary>
private decimal[] xOld;
```

Siccome poi non abbiamo alcuna operazione aggiuntiva da fare per la prossima iterazione il metodo SolverExitCondition sarà semplicemente:

```
public override bool SolverExitCondition(decimal[] x)
{
    return NormAxMinusBFracNormB(x) < tol;
}
```

Dove il metodo NormAxMinusBFracNormB (inserito nella classe IterativeSolver come metodo di supporto) è semplicemente la formula $\frac{||Ax^{(k)} - b||}{||b||}$ come mostrato di seguito:

```
protected decimal NormAxMinusBFracNormB(decimal[] x)
{
    //calcola prodotto matrice A per vettore x
    decimal[] Ax = MatrixVectorMultiply(x);

    //calcola il residio b - Ax
    decimal[] AxMinusB = VectorsSubtraction(Ax, b);

    return Norm(AxMinusB) / Norm(b);
}
```

1.4. Implementazione GaussSeidelSolver

L'idea di base del metodo di Gauss Seidel è la stessa di Jacobi, con la differenza che (al posto di risolvere ogni equazione del sistema per ciascuna delle incognite) si parte dalla prima riga in maniera uguale alla precedente, ma poi vogliamo prendere l'incognita già risolta alla prima riga e la portiamo alla equazione successiva nel sistema, continuando così a catena per tutte le righe dell'equazione.
Quello che succede, di fatto, è che anche qui abbiamo una scomposizione della matrice A in 2 matrici più facili da memorizzare:

$$A = P - N$$

dove:

- P: matrice invertibile che contiene gli elementi diagonali di A e la parte strettamente inferiore di A.
- N: matrice con coefficienti invertiti della parte strettamente superiore A.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

La scomposizione diventa:

$$P = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad N = \begin{bmatrix} 0 & -a_{12} & \cdots & -a_{1n} \\ 0 & 0 & \cdots & -a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Anche qui, in seguito a questa scomposizione si ha che il sistema $Ax = b$ può essere scritto come:

$$(P - N)x = b \quad \rightarrow \quad Px = b + Nx$$

Abbiamo infatti che la formula di per ottenere x in seguito alla scomposizione è:

$$x^{(k)} = \frac{\left(b + Nx^{(k-1)}\right)}{P}$$

Esattamente come prima, ma avendo però che le matrici P ed N sono differenti abbiamo che la formula di aggiornamento (per ogni iterazione k) di ogni componente $x[i]$, del vettore delle incognite, cambia, passando a questa formula:

$$x[i]^{(k)} = \frac{\left(b[i] - \left(\Sigma_{j<i} A[i,j] \times x[j]^{(k)}\right) - \left(\Sigma_{j>i} A[i,j] \times x[j]^{(k-1)}\right)\right)}{A[i,i]}$$

dove:

- La prima sommatoria ($j < i$) usa i nuovi valori aggiornati ($x[j]^{(k)}$) per le variabili già calcolate.
- La seconda sommatoria ($j > i$) usa i vecchi valori ($x[j]^{(k-1)}$) per le variabili ancora da calcolare.

Se andiamo a visualizzare quindi l'implementazione del metodo SolverLogic all'interno della classe GaussSeidelSolver possiamo vedere che implementerà esattamente la formula di aggiornamento di ogni componente $x[i]$ appena descritta:

```
public override decimal[] SolverLogic(int i, decimal[] x)
{
    decimal sigma = 0;
    for (int j = 0; j < A.columns; j++)
    {
        if (j != i)
            sigma += A.matrix[i, j] * x[j];
    }

    //formula calcolo gauss seidel
    x[i] = (b[i] - sigma) / A.matrix[i, i];

    return x;
}
```

Calcolo di entrambe le sommatorie con un solo ciclo:

$$sigma = \sum_{j \neq i} A[i,j] \times x[j]^{(?)}$$

Dove:

- La variabile $x[j]$ utilizza i nuovi valori $x[j]^{(k)}$ quando $j < i$, perché questi valori sono già stati aggiornati nel ciclo corrente.
- La variabile $x[j]$ utilizza i vecchi valori $x[j]^{(k-1)}$ quando $j > i$, perché questi valori non sono ancora stati aggiornati in questa iterazione.

Calcolo aggiornamento $x[i]^{(k)}$:

$$x[i]^{(k)} = \frac{(b[i] - sigma)}{A[i,i]}$$

Siccome poi non abbiamo alcuna operazione aggiuntiva da fare per la prossima iterazione il metodo SolverExitCondition sarà identico al precedente, ossia:

```
public override bool SolverExitCondition(decimal[] x)
{
    return NormAxMinusBFracNormB(x) < tol;
}
```

Dove il metodo NormAxMinusBFracNormB lo stesso in quanto inserito nella classe IterativeSolver dal quale entrambi questi solutori visti fin'ora ereditano.

1.5. Implementazione GradientSolver

Il metodo del gradiente si basa sull'interpretazione della risoluzione di un sistema lineare, come la ricerca del minimo di una funzione quadratica. La funzione da minimizzare infatti è:

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

Il minimo di $f(x)$ corrisponde alla soluzione del sistema $Ax = b$. Mentre la funzione di aggiornamento è:

$$x^{(k)} = x^{(k-1)} + \alpha^{(k-1)} r^{(k-1)}$$

Dove:

- $r^{(k-1)}$ è il residuo (ma negativo), ossia $r^{(k-1)} = b - Ax^{(k-1)}$
- $\alpha^{(k-1)} = \frac{r^{(k-1)T} r^{(k-1)}}{r^{(k-1)T} Ar^{(k-1)}}$

Più nello specifico, per arrivare risolvere il problema di minimizzazione, abbiamo che per ogni iterazione (prima del controllo della convergenza si verificano queste operazioni):

1. **Calcolo del residuo:** Si inizia con una stima iniziale del vettore delle incognite. Con questo si calcola il residuo $r = b - Ax$, che rappresenta l'errore assoluto tra il valore attuale Ax e il vettore della soluzione b.

2. **Direzione della discesa del gradiente**: Il residuo appena calcolato si usa come direzione di discesa, poiché r corrisponde al gradiente della funzione $f(x)$ valutato in x:
   $$-\nabla f(x) = b - Ax = r$$
   abbiamo che il gradiente punta nella direzione di crescita massima della funzione $f(x)$, quindi si ha che, per minimizzare la funzione, dobbiamo muoverci in direzione r

3. **Aggiornamento x:** Si cerca il miglior passo α lungo la direzione r minimizzando $f(x)$, ottenendom)::
   $$\alpha = \frac{r^T r}{r^T Ar}$$
   calcolato alpha si può poi aggiornare la soluzione:
   $$x = x + \alpha \cdot r$$
   abbiamo dunque che ogni aggiornamento x riduce $f(x)$ il più possibile lungo la direzione r scelta. Questo garantisce che il metodo scenda più velocemente possibile in quella direzione.

Se Consideriamo $x[i]$, l'i-esimo elemento del vettore x. La formula di aggiornamento può essere scritta come:

$$x[i]^{(k)} = x[i]^{(k-1)} + \alpha^{(k-1)} r[i]^{(k-1)}$$

Se andiamo a visualizzare quindi l'implementazione del metodo SolverLogic all'interno della classe GradientSolver possiamo vedere che implementerà esattamente la formula di aggiornamento di ogni componente $x[i]$ appena descritta:

```
public override decimal[] SolverLogic(int i, decimal[] x)
{
    //alpha initialization
    if (i == 0)
    {
        var Ax = MatrixVectorMultiply(x);
        r = VectorsSubtraction(b,Ax); //indichiamo b-Ax = r

        // Calcolare Ar = A * (Ax-b)
        Ar = MatrixVectorMultiply(r);

        // Calcolare alpha = (r^T * r) / (r^T * A * r)
        decimal rDotr = Dot(r, r);
        decimal rDotAr = Dot(r, Ar);

        // Evitare la divisione per zero
        if (rDotAr == 0)
        {
            alpha = 0;
        }

        alpha = rDotr / rDotAr;
    }

    x[i] += alpha * r[i];

    return x;
}
```

Calcolo del residuo: $-r = b - A \cdot x$

Calcolo di alpha: $\alpha = \frac{r^T r}{r^T Ar}$

Aggiornamento di $x[i]$, ossia:
$x[i] = x[i] + \alpha \cdot r[i]$

Nota che per tenere traccia dei valori del residuo e di alpha si definiscono questi come attributi delle classe, in modo che ogni chiamata del metodo SolverLogic per condivida gli stessi valori. Solo quando si inizia a calcolare un nuovo vettore $x^{(k+1)}$ (ce se ne accorge perché i==0) allora il valore viene sovrascritto:

```
/// <summary>
/// static variable that memorizes r for each iteration
/// </summary>
private static decimal[] r;

/// <summary>
/// static variable that memorizes Ar for each iteration
/// </summary>
private static decimal[] Ar;

/// <summary>
/// static variable that memorizes alpha for each iteration
/// </summary>
private static decimal alpha;
```

Siccome poi non abbiamo alcuna operazione aggiuntiva da fare per la prossima iterazione il metodo SolverExitCondition sarà identico al precedente, ossia:

```
public override bool SolverExitCondition(decimal[] x)
{
    return NormAxMinusBFracNormB(x) < tol;
}
```

1.6. Implementazione GradientSolver

Il metodo del gradiente coniugato è un miglioramento del metodo del gradiente, che evita la convergenza a zig-zag tramite l'utilizzo di una "guida" chiamata **passo**. Questo metodo garantisce la convergenza in un numero finito di iterazioni per matrici simmetriche e definite positive. Anche qui, quindi, abbiamo che la funzione che a funzione da minimizzare è:

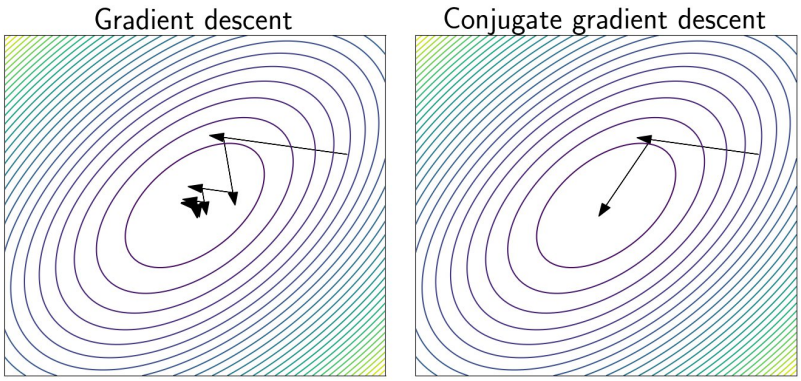
$$f(x) = \frac{1}{2}x^T Ax - b^T x$$

Il minimo di $f(x)$ corrisponde alla soluzione del sistema $Ax = b$. Mentre la funzione di aggiornamento è:

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

Dove:

- $\alpha^{(k-1)} = \frac{r^{(k-1)T} r^{(k-1)}}{r^{(k-1)T} Ap^{(k-1)}}$ (nota che $Ap^{(k-1)}$ è calcolato con p al posto che r)
- p è il passo del gradiente (vettore derivato da r) che serve ad indirizzare più precisamente r verso la soluzione del sistema



Più nello specifico, per arrivare risolvere il problema di minimizzazione, abbiamo i seguenti steps:

1. **Inizializzazione di residuo e passo:** Si inizia con una stima iniziale del vettore delle incognite $x^{(0)}$. Con questo si calcola:
   a. il residuo: $r^{(0)} = b - Ax^{(0)}$
   b. Il passo: $p^{(0)} = r^{(0)}$

2. **Aggiornamento di x**: Una volta calcolati r e p è possibile calcolare alpha:
   $$\alpha^{(k)} = \frac{r^{(k)T} r^{(k)}}{r^{(k)T} Ap^{(k)}}$$
   ed avendo alpha e p è possibile calcolare l'aggiornamento del vettore delle incognite:
   $$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$$

3. **Controllo convergenza** : Si controlla se la soluzione calcolata converge:
   $$\frac{||Ax^{(k+1)} - b||}{||b||} < tol$$

4. **Aggiornamento delle variabili per la prossima iterazione**: Abbiamo che dopo aver verificato se il vettore $x^{(k+1)}$ permette la convergenza, in caso negativo allora aggiorniamo le variabili per la prossima istanza:
    a. Residuo: $r^{(k+1)} = r^{(k)} - \alpha^{(k)} A p^{(k)}$ che è dimostrabile essere un modo più efficiente di calcolare $b - Ax^{(k+1)}$
    b. Beta: è un coefficiente di supporto per il calcolo di p. Infatti $\beta^{(k)} = \frac{r^{(k+1)T}\ r^{(k+1)}}{r^{(k)T}\ r^{(k)}}$
    c. Passo: $p^{(k+1)} = r^{(k+1)} + \beta^{(k)}\ p^{(k)}$

In ogni iterazione andremo a ripetere tutte le operazioni dalla 2 a alla 4.

Nota che se consideriamo $x[i]$, l'i-esimo elemento del vettore x. La formula di aggiornamento può essere scritta come:

$$x[i]^{(k+1)} = x[i]^{(k)} + \alpha^{(k)}\ p[i]^{(k)}$$

Diversamente dalle implementazioni precedenti, per realizzare questa serie di step in linguaggio C#, oltre che ad utilizzare l'implementazione del metodo SolverLogic all'interno della classe GradientSolver dovremo anche fare utilizzo del metodo costruttore della classe (in quanto, come abbiamo visto dal comportamento appena descritto, la parte iterativa del codice consisterà dallo step 2 allo step 4; lo **step 1** è puramente pensato per il setup iniziale)

```
public ConjugateGradientSolver(AccurateMatrix A, decimal[] b, decimal tol, int maxIter, decimal[]? x = null)
    : base(A, b, tol, maxIter)
{
    if(x == null)
    {
        //vettore delle incognite --> inizializzato a zero
        x = new decimal[A.rows];
    }

    //calcola prodotto matrice A per vettore x
    decimal[] Ax = MatrixVectorMultiply(x);

    //calcola il residio 0 = b - Ax
    r = VectorsSubtraction(b, Ax);

    //calcola passo 0
    p = (decimal[])r.Clone();

    //calcola r*r all'iterazione 0
    rDOTr = Dot(r, r);

    inizializzazione per non avere variabili = null
}
```

Se non viene fornito il vettore delle incognite $x^{(0)}$ allora viene inizializzato a 0

Calcolo del primo residuo:
$r^{(0)} = b - Ax^{(0)}$

Calcolo del primo passo:
$p^{(0)} = r^{(0)}$

Calcolo di $r^{(0)} \cdot r^{(0)}$ in quanto è una variabile utile sia per il calcolo di alpha che di beta, non avrebbe senso calcolarla 2 volte più tardi

Una volta fatto il setup delle variabili (corrispondente allo step 1 descritto sopra) possiamo vedere come il metodo metodo SolverLogic (all'interno della classe GradientSolver) implementi esattamente lo **step 2** (formula di aggiornamento di ogni componente $x[i]$) appena descritta:

```
public override decimal[] SolverLogic(int i, decimal[] x)
{
    if (i == 0)
    {
        Ap = MatrixVectorMultiply(p);
        alpha = rDOTr / Dot(r, Ap);
    }

    x[i] += alpha * p[i];

    r[i] -= alpha * Ap[i];

    return x;
}
```

Calcolo di alpha: $\alpha^{(k)} = \frac{r^{(k)T}\, r^{(k)}}{r^{(k)T}\, Ap^{(k)}}$

Nota che questa operazione viene eseguita una sola volta per ogni $x^{(k)}$

Aggiornamento di $x[i]^{(k+1)} = x[i]^{(k)} + \alpha^{(k)}\, p[i]^{(k)}$

Aggiornamento di $r[i]^{(k+1)} = r[i]^{(k)} - \alpha^{(k)} Ap[i]^{(k)}$

Nota che nella descrizione che abbiamo dato prima l'aggiornamento del residuo sarebbe da fare dopo il controllo della convergenza, noi in questo caso lo calcoliamo prima in quanto nel metodo SolverExitCondition (per scelta di design) non viene passato il parametro i quindi non sarebbe possibile calcolare $r[i]$ ma andrebbe calcolato ad ogni iterazione tutto r (che sarebbe un grosso spreco di risorse computazionali)

In fine, il metodo SolverExitCondition sarà incaricato, in questo caso, di completare lo step 3 e lo step 4:

```
public override bool SolverExitCondition(decimal[] x)
{
    if (NormAxMinusBFracNormB(x) < tol)
    {
        return true;
    }
    else
    {
        #region setup for next iteration
        decimal rDOTr_new = Dot(r, r);
        decimal beta = rDOTr_new / rDOTr;
        for (int i = 0; i < A.rows; i++)
        {
            p[i] = r[i] + beta * p[i];
        }

        rDOTr = rDOTr_new;
        #endregion

        return false;
    }
}
```

Step 3: Controllo della convergenza , si controlla che:

$$\frac{||Ax^{(k+1)} - b||}{||b||} < tol$$

Step 4: In caso di non convergenza si preparano le variabili per l'iterazione successiva, abbiamo infatti:

- Calcolo di beta: $\beta^{(k)} = \frac{r^{(k+1)T}\, r^{(k+1)}}{r^{(k)T}\, r^{(k)}}$
- Aggiornamento del passo: $p^{(k+1)} = r^{(k+1)} + \beta^{(k)}\, p^{(k)}$
- Aggiornamento della variabile rDOTr: $r^{(k+1)T}\, r^{(k+1)}$

Come si può notare, questo è il metodo che ha reso necessario rendere SolverExitCondition un metodo astratto, in quanto questa particolare calsse di solutore iterativo aveva bisogno di eseguire delle operazioni di setup per la prossima iterazione caso in cui non si raggiungesse la convergenza.

2. Parte-2: Analisi dei risultati

Dopo avere dettagliatamente descritto il funzionamento della libreria (e la teoria dietro la sua realizzazione) possiamo ora passare all'analisi dei risultati prodotti dall'esecuzione della libreria stessa.
Dividiamo infatti l'analisi in 2 parti:

1. Risultati di performance prodotti sulle 4 matrici fornite (spa1, spa2, vem1, vem2) e sui diversi indici di precisione richiesti ($tol = [10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}]$)
2. Risultati tra paragoni temporali della libreria appena sviluppata con altre librerie open source e paragoni su sistemi operativi diversi

Per realizzare i grafici che vedremo di seguito è stata aggiunta alla libreria una funzionalità di logging, che possiamo vedere anche all'interno del metodo Solve (all'interno della classe IterativeSolver):

```
public decimal[] Solve()
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    int n = A.rows;
    decimal[] x = new decimal[n]; //vettore delle soluzioni --> inizializzato a zero

    for (int k = 0; k < maxIter; k++)
    {
        applicazione solver logic

        // Check for the exit condition
        if (SolverExitCondition(x))
        {
            stopwatch.Stop();
            LogResults(true, tol, k, stopwatch.Elapsed);
            return x;
        }
    }

    LogResults(false, tol, maxIter, stopwatch.Elapsed);
    throw new Exception("Iterative method did not converge.");
}
```

Log risultati in caso di convergenza

Log risultati in caso di non convergenza

Dove questo metodo di logging permette semplicemente di salvare i risultati della convergenza / non convergenza e tempi di calcolo (per ogni matrice e per ogni solutore) in un file .CSV

```
protected void LogResults(bool converge, decimal tolerance, int iterations, TimeSpan timeSpent)
{
    string filePath = "results.csv";
    bool fileExists = File.Exists(filePath);

    using (StreamWriter writer = new StreamWriter(filePath, true))
    {
        // If file does not exist, write the header
        if (!fileExists)
        {
            writer.WriteLine("Matrix,Convergence,SolverType,PrecisionRequired,Iterations,TimeSpent(ms),OS");
        }

        string solverType = this.GetType().Name;
        string OS = RuntimeInformation.IsOSPlatform(OSPlatform.Linux) ? "Linux" : "Windows"; ;

        // Write the data
        writer.WriteLine($"{A.matrix_name},{converge},{solverType},{tolerance},{iterations},{timeSpent.TotalMilliseconds},{OS}");
    }
}
```

Avere questa funzionalità ha poi permesso di estrarre il file "results.csv" e fare delle piccole analisi in python (che saranno i grafi che vedremo di seguito).

2.1. Spiegazione risultati sulle matrici fornite

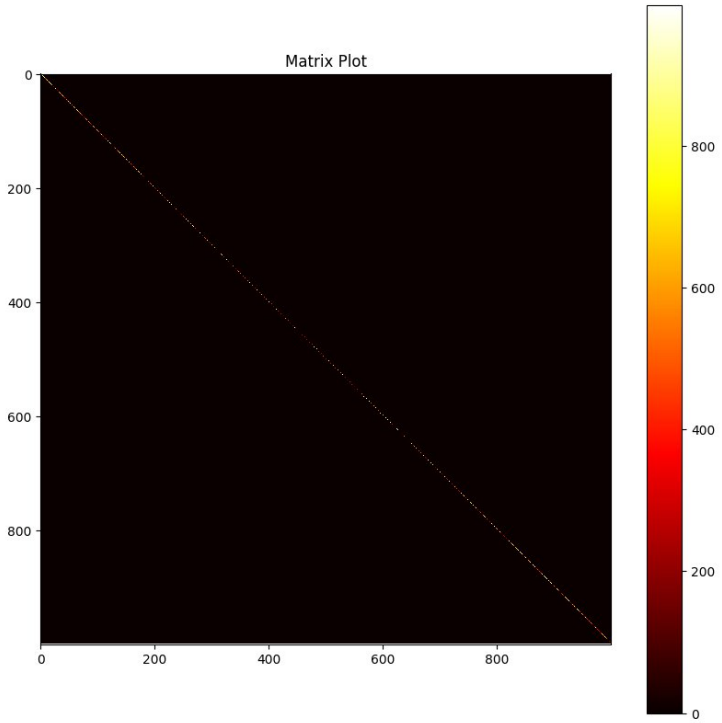
Questa sezione è dedicata ai risultati del progetto. Presenta un'analisi comparativa dei diversi metodi iterativi utilizzati per risolvere sistemi lineari sparsi. Tale analisi è illustrata mediante grafici che riportano sull'asse y:

- O il logaritmo in base 10 del numero di iterazioni (per raggiungere la convergenza)
- O il logaritmo in base 10 del tempo di esecuzione in ms (per raggiungere la convergenza)

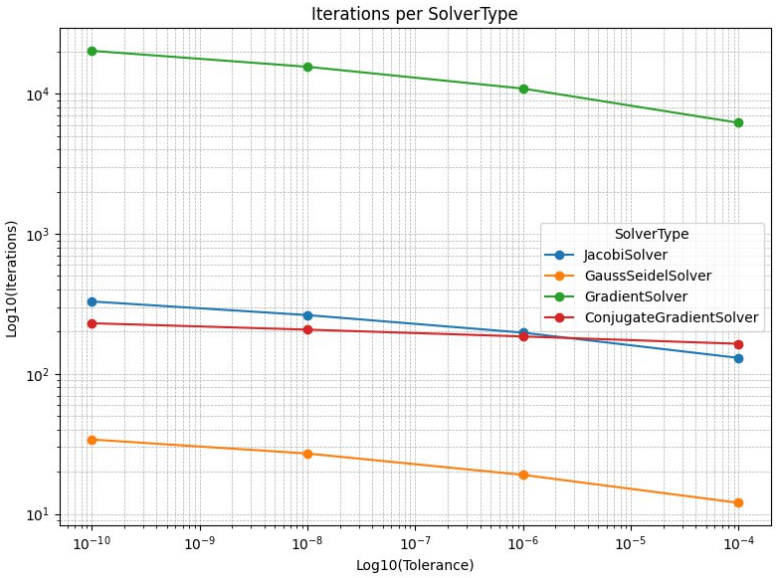
Si invece sull'asse x i differenti valori di tolleranza ($tol = [10^{-4},10^{-6},10^{-8},10^{-10}]$).

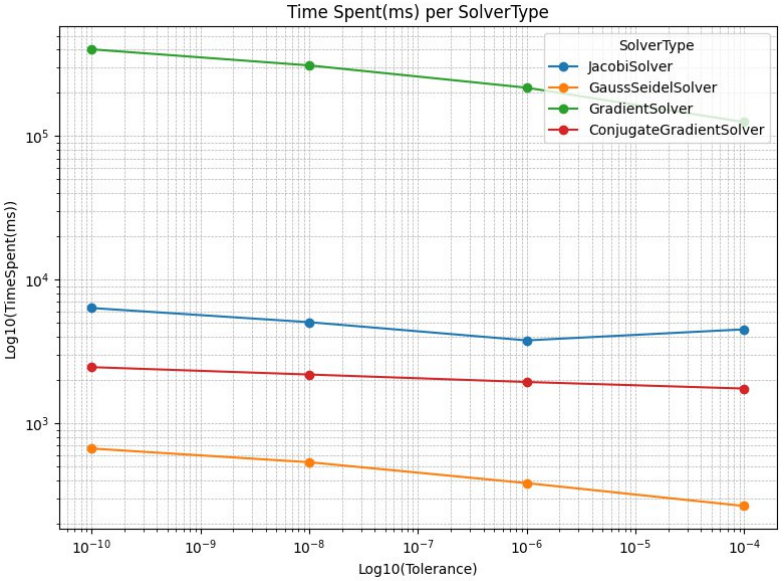
2.1.1. Spiegazione risultati spa1.mtx

La matrice in input "spa1.mtx" è una matrice sparsa a dominanza diagonale 1000*1000 con 182434 celle valorizzate:



Di seguito i risultati dell'elaborazione con i diversi solutori:





Abbiamo che:

- **Gradient Solver**: Non risulta particolarmente efficiente ne a livello di numero di iterazioni ne a livello di tempo impiegato per raggiungere la soluzione, risultando tra tutti i metodi il peggiore a risolvere questo sistema. Questo comportamento è probabilmente dovuto al

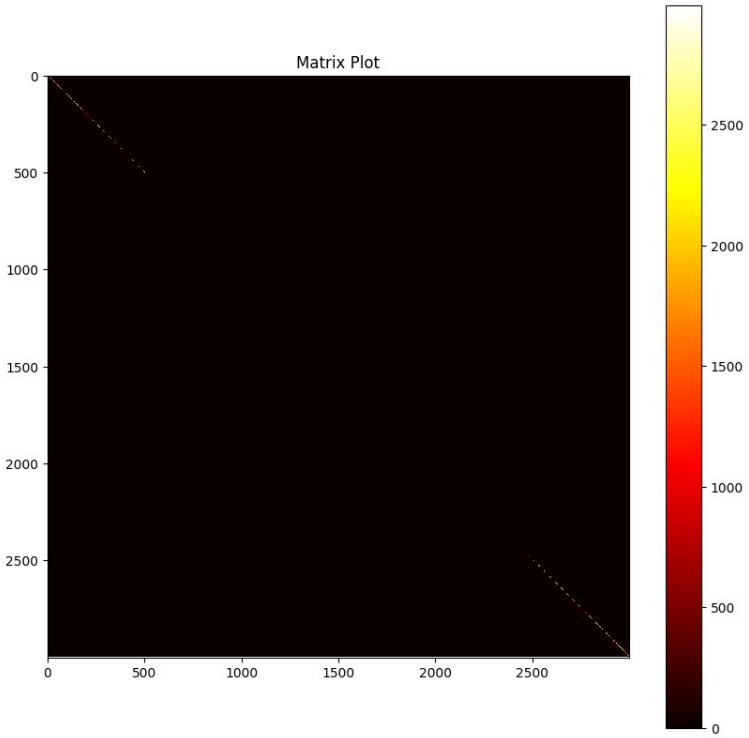
fatto che generalmente questo metodo è utile quando si ha una buona stima iniziale della soluzione (che in verità corrisponde al nostro caso, in quanto partiamo con un vettore delle soluzioni inizializzato a zero) e che la velocità di convergenza è lenta su sistemi mal condizionati (questa potrebbe essere la causa della lentezza di questo metodo, abbiamo infatti che se osserviamo i valori della scala di heatmap c'è un grossissima differenza tra il valore minimo 0, ed il valore massimo ~900).

- **Jacobi Solver & Conjugate gradient solver**: Hanno prestazioni molto simili sia a livello di iterazioni che a livello di tempo impiegato per raggiungere la soluzione anche se non risultato essere il più efficiente per motivi diversi:
    - **Jacobi solver**: presenta prestazioni leggermente peggiori (ma comunque non terribili) perchè è un algoritmo semplice che funziona bene se il sistema è diagonalmente dominante.
    - **Conjugate gradient solver**: presenta prestazioni leggermente migliori perché è un miglioramento dell'algoritmo del gradiente, inoltre generalmente funziona bene quando si ha sistemi grandi e sparsi (come nel nostro caso).

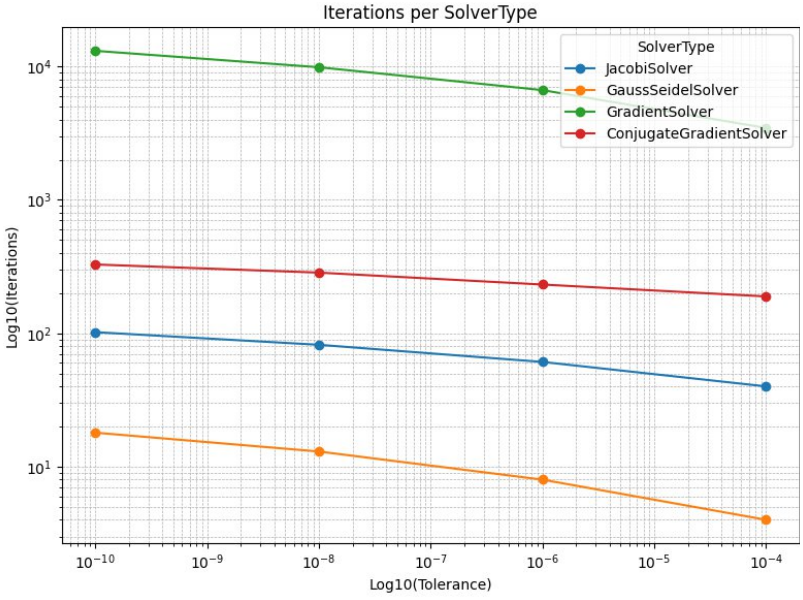
- **Gauss Seidel Solver**: presenta le performance migliori su questa matrice in quanto è un miglioramento al metodo di Jacobi (che funzionava già discretamente bene), generalmente converge sempre più velocemente rispetto al Jacobi e funziona bene su sistemi a dominanza diagonale, di conseguenza ha senso che i tempi di computazione siano i migliori.

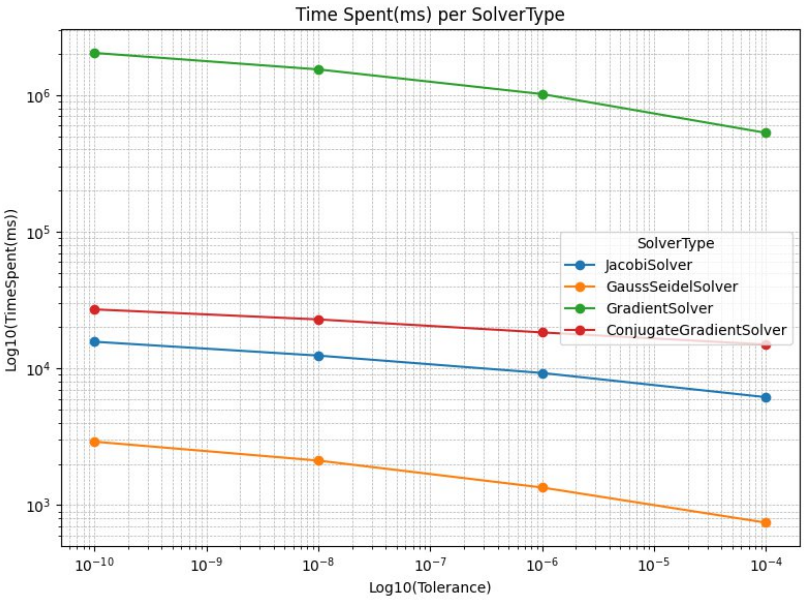
2.1.2. Spiegazione risultati spa2.mtx

La seguente matrice in si chiama "spa2.mtx" ed è una matrice sparsa a dominanza diagonale (anche se molti punti della diagonale risultano nulli) 3000*3000 con 1633298 celle valorizzate:



Di seguito i risultati dell'elaborazione con i 4 solutori implementati:





Abbiamo la matrice data in input per questa seconda elaborazione è una matrice molto simile alla precedente ma più grande, quindi i risultati sono abbastanza simili:

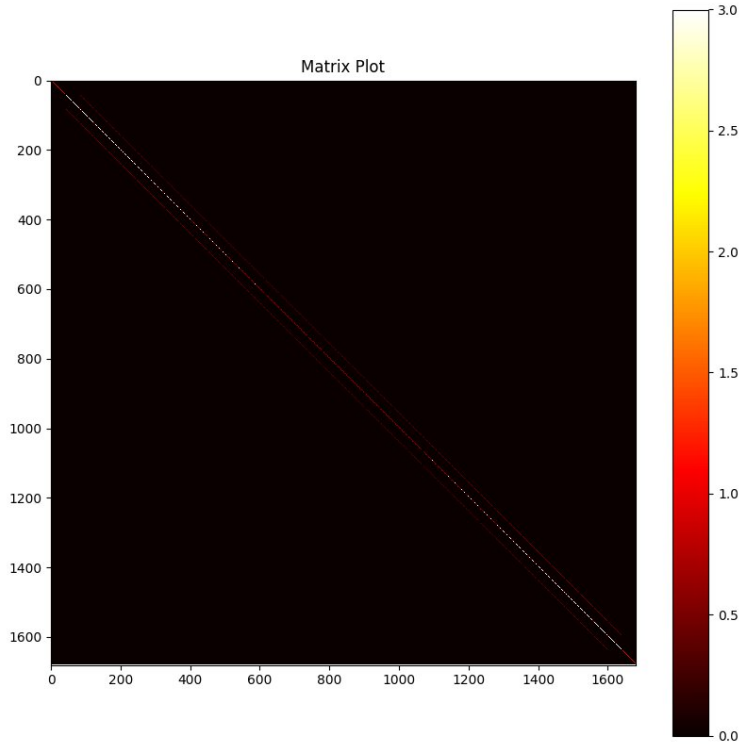
- **Gradient Solver**: Come prima non risulta particolarmente efficiente ne a livello di numero di iterazioni ne a livello di tempo impiegato per raggiungere la soluzione, risultando tra tutti i metodi il peggiore a risolvere anche questo sistema, per le stesse motivazioni date precedentemente.

- **Jacobi Solver & Conjugate gradient solver**: Hanno prestazioni molto simili sia a livello di iterazioni che a livello di tempo impiegato per raggiungere la soluzione; ma questa volta il gap tra questi 2 inizia a diventare più ampio:
    - **Jacobi solver**: presenta prestazioni leggermente migliori perché è un algoritmo semplice rispetto al gradiente coniugato, il che vuol dire che all'aumentare delle dimensioni del sistema ragionevolmente questo diventa più efficiente.
    - **Conjugate gradient solver**: questa volta presenta prestazioni leggermente peggiori in quanto il sistema è più grande e questo algoritmo richiede computazioni più complicate rispetto a Jacobi.

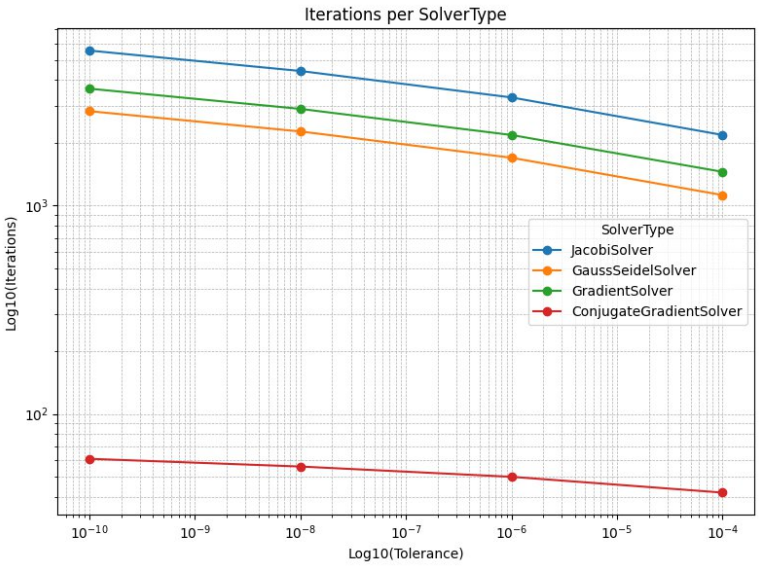
- **Gauss Seidel Solver**: rimane il migliore, a maggior ragione essendo un miglioramento di Jacobi anche qui all'aumentare delle dimensioni del sistema non si sente così tanta differenza quanta se ne sente con il gradiente coniugato.

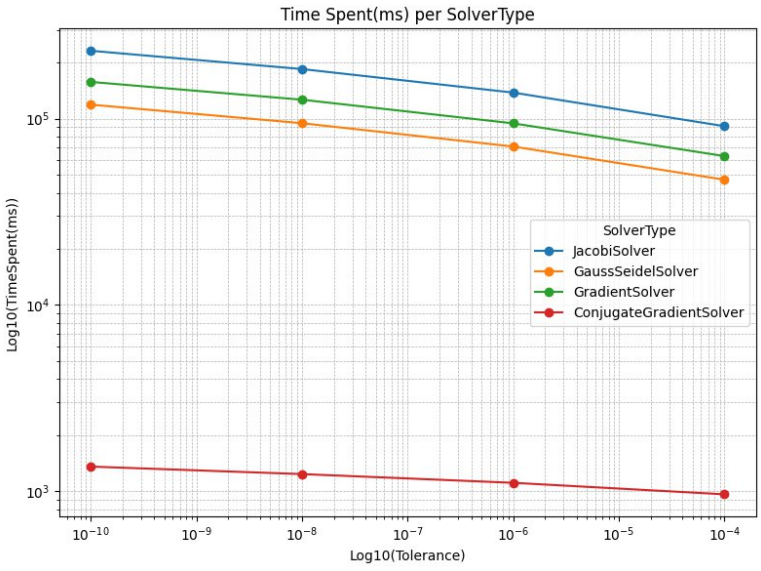
2.1.3. Spiegazione risultati vem1.mtx

La matrice seguente matrice si chiama "vem1.mtx" ed è una matrice sparsa a dominanza diagonale 1681*1681 con 13385 celle valorizzate (nota che a questa matrice è stato fatto il valore assoluto per stamparne la heatmap per meglio comprendere dove fossero i valori a zero, quindi, differentemente dalle precedenti, questa presenta anche valori negativi):



Di seguito i risultati:





Questa matrice, oltre ad avere numeri sulla diagonale presenta anche altre 2 righe parallele alla diagonale, possiamo comunque definirla a dominanza diagonale perchè i valori non nulli si trovano principalmente sulla diagonale principale o nelle vicinanze. Data questa differenza i risultati di performance sono cambiati:

- **Jacobi Solver**: Risulta diventare il peggiore sia a livello di numero di iterazioni sia a livello di tempi di calcolo. Ha senso le sue performance diminuiscano in quanto il sistema, è “meno diagonalmente dominante” rispetto a prima. Prima avevamo solo numeri sulla diagonale principale e raggiungeva performance discrete, ora che ci sono numeri anche su queste due altre linee parallele ha senso che le performance diminuiscano.

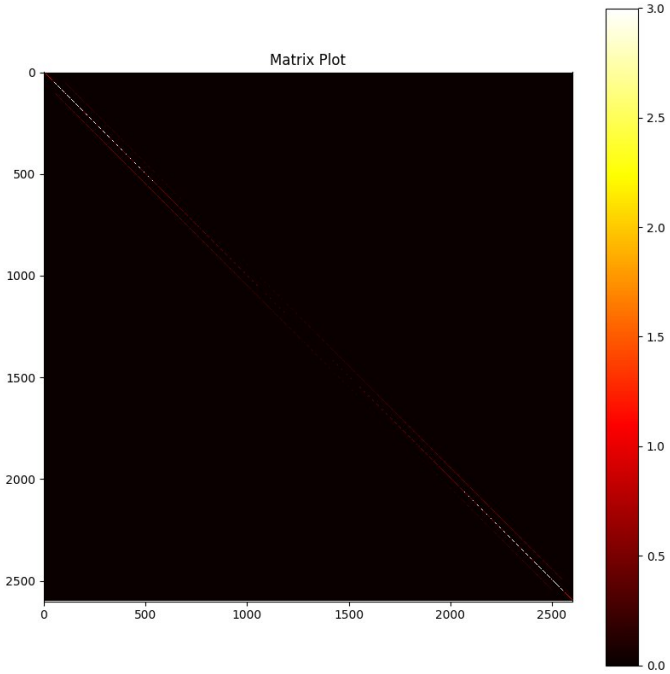
- **Gradient solver**: Secondo algoritmo meno performante, riesce a non essere l’ultimo a livello di performance probabilmente perchè, pur essendo più grande di spa1 abbiamo che questa matrice è molto meglio condizionata. Basta osservare la differenza della scala della heatmap per verificare che qui i numeri vanno solo da 0 a 3.5, mentre nelle 2 matrici precedenti vanno da 0 a valori molto più alti che portano ad avere un indice di condizionamento molto più alto.

- **Gauss Seidel Solver**: Essendo un derivato di Jacobi, al peggioramento delle performance di Jacobi peggiorano ance per questo algoritmo, che non si ritrova più, dunque, in prima posizione.

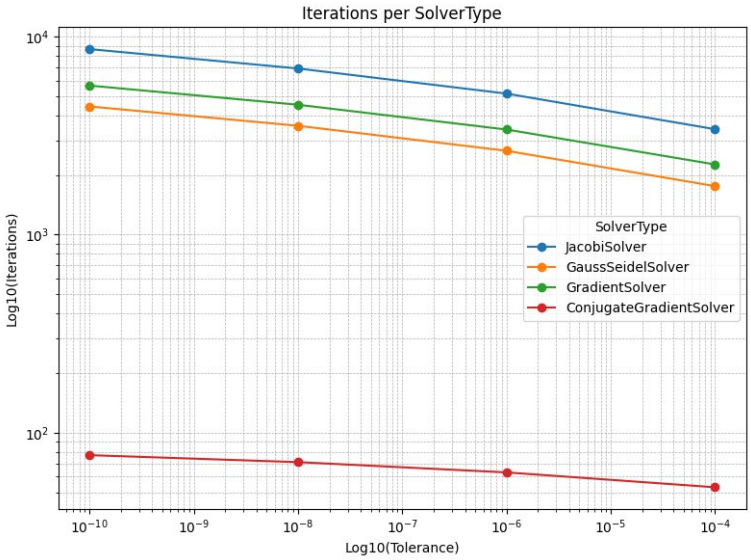
- **Conjugate Gradient solver**: Risulta, giustamente, essere il più efficiente in quanto, come abbiamo già specificato prima, questa matrice risulta essere molto meglio condizionata rispetto le precedenti migliorando le performance anche di questo solver, essendo questo un miglioramento dell’algoritmo del gradiente. Avere un numero di iterazioni molto più basso rispetto a tutti gli altri solutori porta inevitabilmente ad avere un tempo di elaborazione molto minore (anche se le singole iterazioni di questo metodo dovrebbero durare di più).

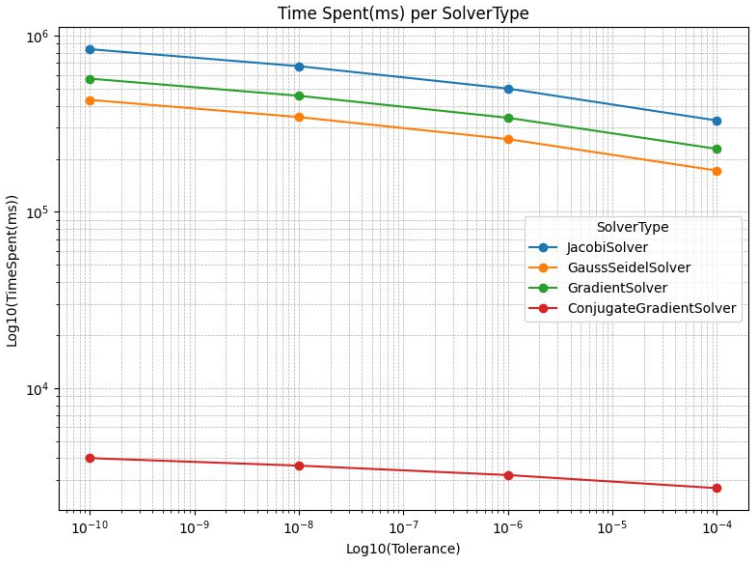
2.1.4. Spiegazione risultati vem2.mtx

La matrice in input si chiama "vem2.mtx" ed è una matrice sparsa a dominanza diagonale 2601*2601 con 21225 celle valorizzate (nota che a questa matrice è stato fatto il valore assoluto per stamparne la heatmap per meglio comprendere dove fossero i valori a zero, anche qui dunque sono presenti valori negativi):



Di seguito i risultati dopo l'elaborazione con i 4 solutori implementati:





Questa matrice è quasi identica alla precedente, solo più grande. Abbiamo infatti che i risultati ottenuti sono identici a quelli precedenti (ma ovviamente raggiunti aumentando il numero di iterazioni e tempo di elaborazione). Questo significa che tutte le conclusioni tratte precedentemente sono valide anche su questa matrice. Inoltre solidifica la validità dei risultati avendo due risultati strutturalmente identici per due matrici strutturalmente identiche.

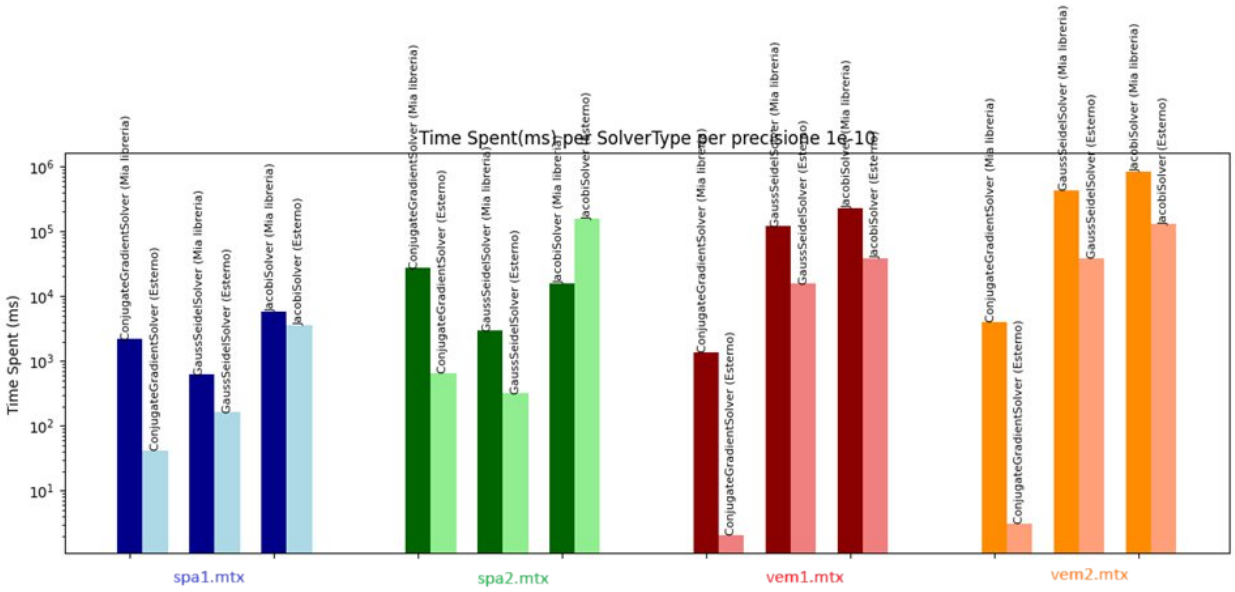
2.1.5. Conclusioni sul funzionamento teorico

Sulla base dei risultati ottenuti, è stato possibile confermare sia la piena funzionalità della libreria sviluppata sia la sua conformità ai principi teorici appresi. Abbiamo infatti che per ogni metodo utilizzato è consistente che il numero di iterazioni necessarie sia direttamente proporzionale al livello di precisione richiesto (come visualizzabile da tutti i grafici mostrati fin'ora). Abbiamo inoltre che le performance computazionali dei diversi solutori cambiano in maniera adeguata a seconda della matrice utilizzata.
Il progetto ha messo in luce l'importanza di condurre un'approfondita analisi comparativa dei metodi disponibili, evidenziando come tale valutazione sia fondamentale per identificare la soluzione più adatta in funzione delle diverse esigenze computazionali. Questo approccio non solo permette di ottimizzare le prestazioni, ma offre anche una guida pratica nella scelta degli strumenti più efficaci per affrontare problemi specifici, garantendo un equilibrio tra efficienza e precisione.

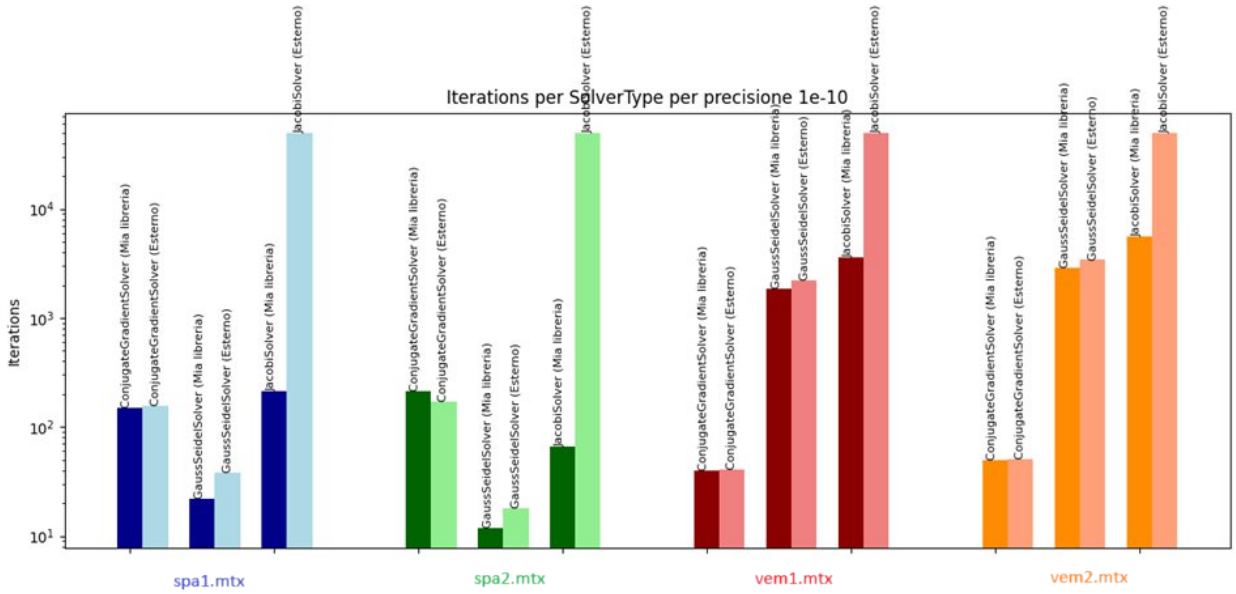
2.2. Paragoni tra altre librerie ed OS differenti

Si utilizza quest'ultima sezione della relazione per parlare dell'andamento dello sviluppo delle diverse versioni della libreria.

Abbiamo infatti che essendo questa libreria stata sviluppata in C# con gestione dei tipi decimal (che permettono di avere precisione fino a 29 cifre decimali) il background da cui partire per lo sviluppo era praticamente inesistente. In C# esistono altre librerie che permetto di operare con sistemi sotto forma matriciale, ma (oltre al fatto che queste hanno molte funzionalità deprecate o non funzionanti) non esiste alcuna altra libreria (ad eccezione di quella sviluppata e trattata in questa relazione) che permetta la gestione del tipo decimal. Tutte le altre librerie utilizzano il tipo double, che ha un livello di precisione molto più basso. (Si può infatti osservare che dal momento in cui è stata pubblicata la libreria ha riscontrato discreto successo, riscuotendo un discreto numero di download verificabile da questa pagina). La scarsità di risorse e librerie in c# ha portato a dover implementare da zero molti metodi che generalmente esistono come già forniti da altre librerie (come il metodo di lettura della matrice da file mtx di cui si è parlato nell'introduzione, la radice quadrata, e così molti altri). Di conseguenza abbiamo che paragonando le performance della nostra libreria con quelle di altre librerie python open-source si ha che le nostre performance temporali sono molto peggiori (la nostra libreria raggiunge tempi nell'ordine di grandezza dei 5 minuti, mentre le librerie open-source nell'ordine di grandezza dei 30 secondi):



Questo è semplicemente dovuto al fatto che, appunto, ho dovuto implementare da zero molti metodi "basilari" che sicuramente altri linguaggi di programmazione / librerie implementano in maniera più ottimizzata. Ed è poi anche dovuto al fatto che il C# è un linguaggio molto pesante, in quanto il compilatore non fa' altro che tradurre le istruzioni in CIL (un linguaggio simile al C) e poi dal CIL al linguaggio macchina.
Diventa però interessante osservare che siccome la nostra libreria utilizza dati di tipo decimal richiede meno iterazioni rispetto che le stesse librerie open source del paragone precedente per raggiungere la convergenza:



Questo succede perchè avendo a disposizione più cifre decimali ad ogni iterazione la nostra libreria arriverà sempre ad un risultato (anche se approssimato) più vicino alla soluzione corretta che rispetto ad altre librerie; a lungo termine questo porta ad avere un numero di iterazioni minore per arrivare alla convergenza. Il che ci porta a dire che se avessimo idealmente a disposizione un calcolatore molto potente sarebbe sempre consigliato usare questa libreria.

Di seguito vengono elencate le varie librerie open source utilizzate il confronto:

• **Jacobi**: https://gist.github.com/angellicacardozo/3a0891adfa38e2c4187612e57bf271d1

• **GaussSeidel**: https://stackoverflow.com/questions/5622656/python-library-for-gauss-seidel-iterative-solver

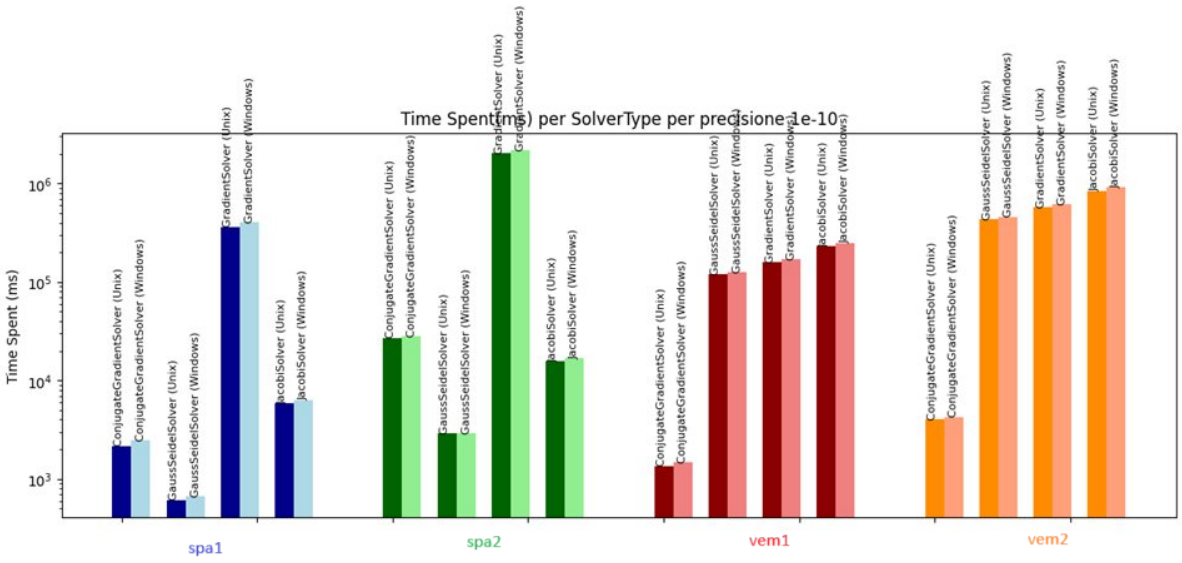
• **ConjugateGradient**: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.cg.html

Alternativamente possiamo anche osservare il tempo medio di elaborazione **di una sola iterazione** per ogni singola coppia “solver-precisione”:

	SolverType	PrecisionRequired	Efficiency
0	ConjugateGradientSolver	1.000000e-10	0.045619
1	ConjugateGradientSolver	1.000000e-08	0.045448
2	ConjugateGradientSolver	1.000000e-06	0.045468
3	ConjugateGradientSolver	1.000000e-04	0.044775
4	GaussSeidelSolver	1.000000e-10	0.023936
5	GaussSeidelSolver	1.000000e-08	0.023543
6	GaussSeidelSolver	1.000000e-06	0.023032
7	GaussSeidelSolver	1.000000e-04	0.020835
8	GradientSolver	1.000000e-10	0.023846
9	GradientSolver	1.000000e-08	0.023737
10	GradientSolver	1.000000e-06	0.023692
11	GradientSolver	1.000000e-04	0.023383
12	JacobiSolver	1.000000e-10	0.024349
13	JacobiSolver	1.000000e-08	0.024419
14	JacobiSolver	1.000000e-06	0.024215
15	JacobiSolver	1.000000e-04	0.018138

Da questa lista possiamo osservare che il tempo di iterazione medio, per ogni coppia “solver-precisione” risulta essere una frazione di millisecondo. Il che ci porta a confermare che l’utilizzo di questa libreria diventa consigliato anche quando si lavora con matrici piccole, in quanto non richiedo tempi di elaborazione giganti.

Per curiosità personale si decide anche di paragonare le performance della libreria quando utilizzata su una macchina linux (distro DeepinOS installato su bare hardware), comparandole con quelle di una macchina windows:



I risultati ottenuti sono benevolmente inaspettati. Essendo C# linguaggio proprietario Microsoft (compagnia owner di Windows) ci si aspettava di ottenere performance migliori sulla macchina

Windows. Invece, sembra che la struttura più light-weight di linux abbia portato comunque ad ottenere tempistiche migliori (ovviamente in questa sezione non è mostrato il numero di iterazioni in quanto rimangono le stesse indipendentemente dall'OS).

Concludendo possiamo dunque estrarre che i punti di forza della nostra libreria sono:

1. Possibilità di mantenere livelli di precisione molto alti
2. Mantenere un numero basso di iterazioni per arrivare a risolvere il problema
3. Possibilità di avere computazioni istantanee ed accurate su matrici piccole
4. Miglioramento generale delle performance computazionali se usata in ambiente linux.