|  |  |
| --- | --- |
|  | Francesco Cavallini  Matricola: 920835  f.cavallini8@campus.unimib.it  Corso di studi di Informatica Magistrale |
| Università degli studi Milano Bicocca  Milano, Padiglione U24  Ottobre, 2024 |  |

|  |  |
| --- | --- |
|  | |
| Compressione di immagini tramite la DCT  Metodi del Calcolo Scientifico  Relazione Progetto 2  **Consegna:**  **“““**  Lo scopo di questo progetto è di utilizzare l’implementazione della DCT2 in un ambiente open source e di studiare gli effetti di un algoritmo di compressione tipo jpeg (senza utilizzare una matrice di quantizzazione) sulle immagini in toni di grigio.  **”””**  Riferimenti : | |
| Repository git-hub: | https://github.com/VR3ED/Easy\_Matrix |

Sommario

[0. Introduzione 3](#_Toc11)

[0.1. Obbiettivi 3](#_Toc12)

[0.2. Scelte implementative 4](#_Toc13)

[1. Parte-1: Sviluppo DCT & DCT2 5](#_Toc14)

[1.1. Requisiti tecnici: 5](#_Toc15)

[1.2. Le basi per lo sviluppo: cosa ci aspettiamo: 5](#_Toc16)

[1.3. Le basi per lo sviluppo: come è strutturato il codice: 6](#_Toc17)

[1.4. Utilss.py: Implementazione DCT: 8](#_Toc18)

[1.5. Utilss.py: Implementazione DCT2: 10](#_Toc19)

[1.6. Utilss.py: Metodi di libreria per DCT e DCT2: 12](#_Toc20)

[1.5. Implementazione GradientSolver 13](#_Toc21)

[1.6. Implementazione GradientSolver 15](#_Toc22)

[2. Parte-2: Analisi dei risultati 18](#_Toc23)

[2.1. Spiegazione risultati sulle matrici fornite 19](#_Toc24)

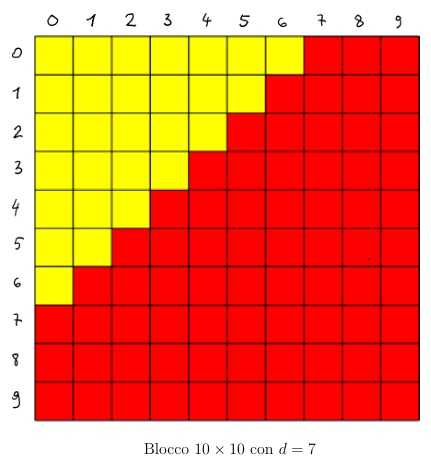
[2.2. Paragoni tra altre librerie ed OS differenti 28](#_Toc25)

# 0. Introduzione

## 0.1. Obbiettivi

Come definito nella pagina iniziale, l’obbiettivo del progetto è quello di creare un implementazione custom della DCT2. In particolare il lavoro è suddiviso in 2 parti in cui abbiamo:

* **Parte 1**: Implementare la DCT2 come spiegata a lezione in un ambiente open source a vostra scelta e confrontare i tempi di esecuzione con la DCT2 ottenuta usando la libreria dell’ambiente utilizzato (che si presuppone essere nella versione FFT)
* **Parte 2**: Creare un software con interfaccia grafica che permetta di suddividere l’immagine in blocchi quadrati di pixel di dimensioni partendo in alto a sinistra, scartando gli avanzi; poi, per ogni blocco eseguire le seguenti operazioni:
  + applicare la DCT2 (della libreria): ;
  + eliminare le frequenze con (assumendo che le frequenze partano da 0: se le elimino tutte, se elimino solo la più alta, cioè quella con. In sostanza bisogna eliminare i coefficienti in frequenza a destra della diagonale individuata dall’intero d, come esemplificato qui sotto:



abbiamo e . I coefficienti da eliminare sono indicati in rosso.

* applicare la DCT2 inversa all’array così modificato:
* arrotondare all’intero più vicino, mettere a zero i valori negativi e a 255 quelli maggiori di 255 in modo da avere dei valori ammissibili (1 byte);

Una volta fatto questo per ogni blocco si rimette insieme l’immagine mettendo assieme i blocchi ottenuti, mettendoli nell’ordine giusto e paragonare l’immagine ottenuta con l’originale di partenza.

## 0.2. Scelte implementative

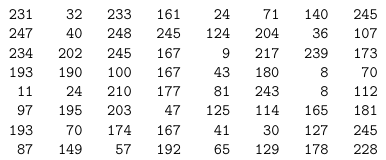
Si sceglie di utilizzare il linguaggio python per lo sviluppo di entrambe le parti del progetto, utilizzando visual studio code come ambiente di sviluppo. Le motivazioni dietro a questa scelta sono:

* Vasta disponibilità di librerie open source per i paragoni necessari ad altre librerieafici
* Vasta gamma di librerie per lo sviluppo di grafici
* Sintassi semplice ed intuitiva, renderà le fasi di analisi del codice più scorrevoli, e, allo stesso tempo, diversamente da C# (il linguaggio utilizzato per lo sviluppo del progetto 1) questo linguaggio è più adatto alla stesura di codice in ambito di calcolo scientifico.

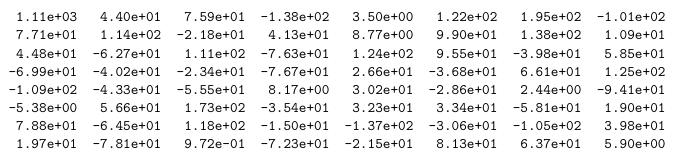
# 1. Parte-1: Sviluppo DCT & DCT2

## 1.1. Requisiti tecnici:

I requisiti necessari per lo sviluppo della prima parte sono quelli di prestare molta attenzione a come viene scalata la DCT2 (o la DCT). Come caso test è necessario verificare che il seguente blocchetto 8×8:



venga trasformato in questo modo dalla DCT2:



È inoltre necessario controllare per la DCT monodimensionale che la prima riga del blocchetto 8×8 precedentemente mostrato venga trasformata in:



## 1.2. Le basi per lo sviluppo: cosa ci aspettiamo:

Per questo progetto, analizzeremo le prestazioni della nostra implementazione della DCT2 rispetto a quella fornita da una libreria di riferimento, concentrandoci sui tempi di esecuzione e sulle complessità teoriche dei due approcci. In particolare, studieremo come i tempi di calcolo variano al crescere delle dimensioni della matrice fornita in input.

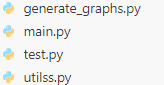
Ci si aspetta che la nostra implementazione manuale della DCT2 dovrebbe presentare una complessità temporale proporzionale a , mentre la versione basata su una libreria ottimizzata, tipicamente costruita utilizzando FFT, dovrebbe raggiungere una complessità . Questa differenza nasce dalle diverse strategie algoritmiche impiegate:

* la nostra implementazione si basa su una definizione diretta della DCT2,
* le librerie ottimizzate sfruttano algoritmi avanzati per ridurre il numero di operazioni necessarie.

Nel corso dell'analisi, confronteremo i tempi di esecuzione ottenuti per matrici di dimensioni crescenti, valutando in che misura i risultati sperimentali confermano le complessità teoriche attese.

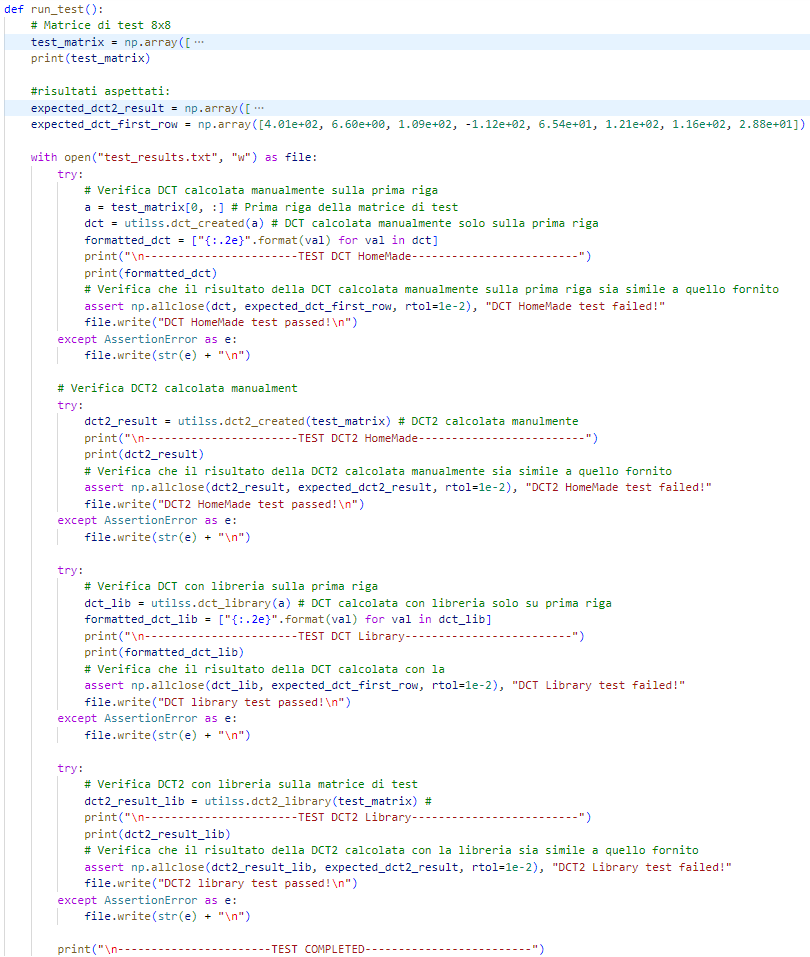
## 1.3. Le basi per lo sviluppo: come è strutturato il codice:

La parte 1 è stata sviluppata suddividendo il codice nelle seguenti classi:



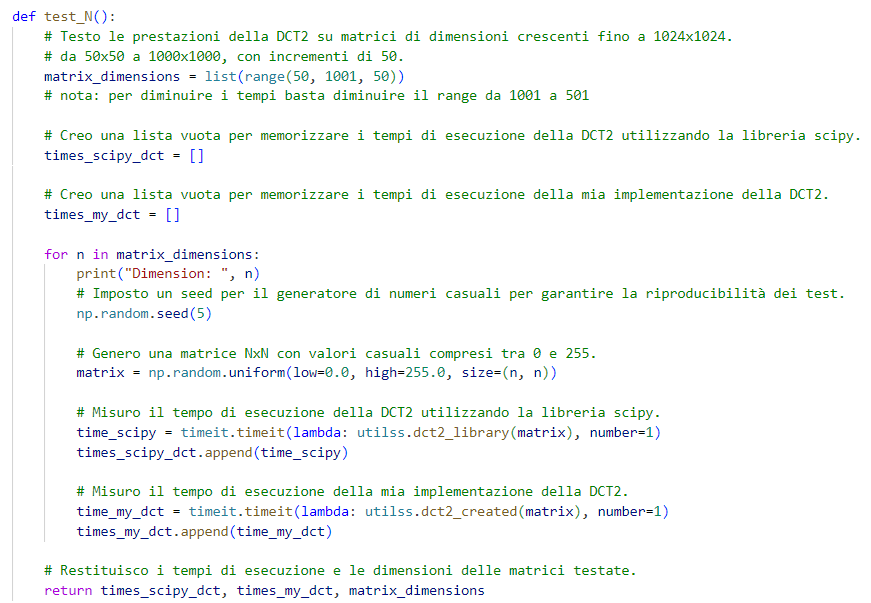
Dove abbiamo che:

test.py: Implementa i requisiti di controllo sull’accuratezza precedentemente accennati:



I commenti nel codice mostrato dovrebbero fornire tutte le informazioni necessarie per individuare tutti i test che vengono svolti per rispettare i requisiti tecnici imposti dalla consegna. Inoltre si può visualizzare che ogni test è inserito all’interno di un try-catch se il test concluderà con successo allora verrà scritto in un file txt che il test è stato completato correttamente, altrimenti, verrà scritto il messaggio di errore.

Sempre nel file tests.py viene poi implementato un altro metodo per misurare il tempo necessario per eseguire computazioni di DCT e DCT2 su matrici di grandezze progressivamente sempre più grandi (da ad ):



Anche qui abbiamo che i commenti sono abbastanza autoesplicativi, tutto quello che succede è l’esecuzione ricorsiva di metodi di DCT2 (creato a mano e di libreria) cambiando ad ogni iterazione la grandezza di . Ad ogni iterazione viene anche misurato il tempo di esecuzione di modo da poterlo graficare in seguito.

main.py: Avvia semplicemente i test definiti nel file test.py

generate\_graphs.py: Colleziona i timestamps di esecuzione forniti dai metodi in test.py e plotta un grafico per confrontare le prestazioni temporali della Trasformata Discreta del Coseno (DCT) e della Trasformata Discreta del Coseno bidimensionale (DCT2) tra due implementazioni: una fornita dalla libreria SciPy e una sviluppata manualmente. Nota che di questo file non verrà mostrato il codice in quanto non rilevante ai fini della relazione, ma verranno mostrati i grafici da essa prodotti.

utilss.py: Cuore del progetto, contiene infatti i due metodi creati manualmente per l’implementazione di DCT e DCT2. Per ciascuna di queste verrà dedicato un capitolo a parte che ne spiega l’implementazione.

Questo file contiene inoltre altri 2 metodi wrapper che servono a chiamare la versione da libreria (Scipy) delle implementazioni di DCT e DCT2.

## 1.4. Utilss.py: Implementazione DCT:

L’idea di base per la realizzazione di DCT è creare una trasformata matematica utilizzata per rappresentare un segnale discreto (dal dominio del tempo) come una somma di funzioni coseno (conversione al dominio della frequenza) con pesi opportuni. Si può usare poi questo principio per scartare le parti con minor quantitativo informativo per creare compressioni di segnali ed immagini. Inoltre DCT è un’operazione reversibile tramite l’operazione IDCT, tuttavia, la ricostruzione non è perfetta a causa di un troncamento delle frequenze che avviene durante la trasformazione DCT. Più nello specifico abbiamo che la DCT funziona secondo questo principio:

1. **Calcolo vettore :** Il calcolo di α è semplicemente una fase di normalizzazione necessaria per costruire la matrice di trasformazione della DCT seguendo la definizione teorica. Questo vettore contiene infatti i fattori di normalizzazione che dipendono dalla posizione (indice ) nella matrice di trasformazione:
   * Per : abbiamo che
   * Per : abbiamo che
2. **Calcolo DCT tramite matrice di trasformazione (T)**: Abbiamo che DCT () utilizza funzioni coseno con frequenze diverse per rappresentare il segnale, ossia abbiamo per ogni vettore che:

Dove Rappresenta il campione del segnale originale (nel dominio spazio/tempo)

Per semplicità implementativa possiamo però dividere questa trasformazione in:

1. **Calcolare la matrice di trasformazione T**: Ossia una matrice unicamente composta da vettori riga () che rappresentano funzioni coseno a diverse frequenze; ogni posizione singola di questa matrice è calcolata in questo modo:

Nota che la parte della formula applica la trasformazione cosinuoidale, utilizzando il vettore assicuriamo che la trasformazione sia ortogonale; ossia che:

* + - I vettori riga saranno mutuamente ortogonali: Il prodotto scalare tra due vettori riga e della matrice deve essere zero per . Questo garantisce che le frequenze diverse (associate a e ) siano indipendenti l'una dall'altra.
    - I vettori riga saranno normalizzati: Il prodotto scalare di ciascun vettore con sé stesso deve essere

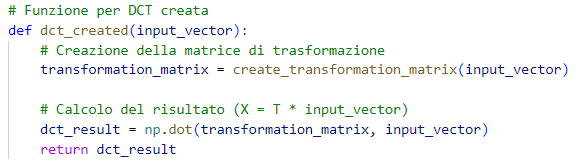
1. **Applicare la trasformazione al segnale originale**: ossia moltiplicare il segnale originale per il vettore di trasformazione:

Dove contiene i coefficienti della DCT, che rappresentano le ampiezze delle diverse frequenze.

Se andiamo a visualizzare quindi l’implementazione del metodo di creazione manuale della DCT all’interno del file utilss.py possiamo osservare che verranno applicati gli stessi principi teorici appena descritti:

Si decide di esportare il calcolo di e di in un altro metodo a parte

Applica la matrice di trasformazione al segnale originale:



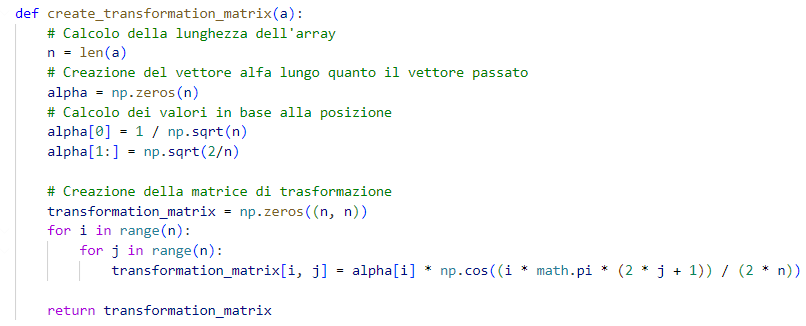
Come possibile osservare il calcolo di e di viene spostato in un metodo diverso (questo semplicemente per migliorare la leggibilità del codice) , questo metodo è il seguente:

Calcolo matrice T cella per cella:

Calcolo di con:

•

• (per i))



Svolgendo i calcoli nell’ordine di esecuzione abbiamo quindi:

1. Calcolo di
2. Calcolo di
3. Applicazione di al segnale originale

Questa serie di operazioni corrisponde quindi perfettamente con la teoria precedentemente descritta.

## 1.5. Utilss.py: Implementazione DCT2:

La DCT2 bi-dimensionale (2D) è un'estensione della DCT al dominio delle matrici. Il principio fondamentale della DCT2 è che concentra la maggior parte dell'informazione utile nei coefficienti a bassa frequenza (tipicamente in prossimità dell'angolo in alto a sinistra della matrice trasformata), rendendo possibile una significativa riduzione della dimensionalità del dato e una compressione efficace. Si ha infatti che la matrice risultante da dalla trasformazione trasfoallrisultatrasDCT2 () viene calcolata applicando separatamente DCT lungo le righe e le colone alla matrice originale, ossia:

Applicazione DCT sulle righe

Applicazione DCT sulle colonne

Dove:

* è il valore dell'elemento della matrice di input.
* e sono le dimensioni della matrice di input.
* Insieme a sono i 2 fattori di normalizzazione per le rispettive DCT.

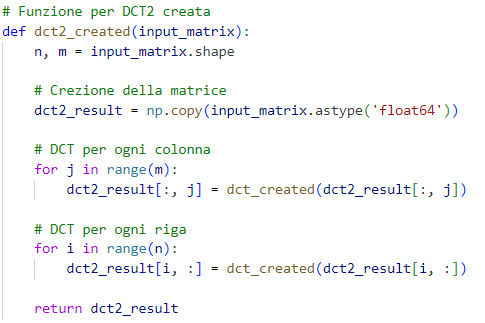
Essendo questa trasformazione direttamente derivata da DCT è facile capire che anche in questo caso è reversibile (mantenendo sempre il problema di ricostruzione perfetta a causa di un troncamento delle frequenze che avviene durante la trasformazione DCT).

È fondamentale evidenziare che la tecnica appena descritta, pur essendo efficace, non corrisponde a quella impiegata dal formato JPEG per comprimere le immagini. Questo perché alcune operazioni critiche potrebbero generare distorsioni e artefatti nella ricostruzione finale. Il formato JPEG, invece, utilizza strategie specifiche per superare tali problematiche, garantendo una compressione efficiente senza compromettere eccessivamente la qualità dell'immagine.

**NOTA**: DCT2 nel formato JPEG

Avendo quindi già spiegato il funzionamento della funzione DCT nel capitolo precedente diventa molto facile spiegare il codice generato per l’implementazione di DCT2

Se andiamo quindi a visualizzare quindi l’implementazione del metodo di creazione manuale della DCT2 (sempre all’interno del file utilss.py) possiamo osservare che verranno applicati gli stessi principi teorici appena descritti:



La DCT2 implementata nel codice antecedente segue una strategia "a separazione di variabili". Ossia viene sfruttata la linearità della trasformazione consentendo di ridurre il problema bidimensionale in **due trasformazioni monodimensionali** consecutive:

1. Prima si calcola la DCT lungo una dimensione (colonne).
2. Poi si calcola la DCT lungo l'altra dimensione (righe).

Questo approccio è computazionalmente efficiente perché riduce la complessità computazionale rispetto al calcolo diretto della DCT2, specialmente per matrici di grandi dimensioni. Abbiamo infatti che la definizione diretta della DCT2 per una matrice è:

Questo implica di avere due somme nidificate () su tutte le celle della matrice. Dove ogni somma richiede moltiplicazioni e somme per calcolare un singolo elemento ​. Per calcolare tutti i coefficienti della DCT2, la complessità totale diventa:

Invece, utilizzando il principio della scomposizione di due operazioni lineari abbiamo che:

* Per la trasformazione lungo le colonne: per una colonna sola richiede operazioni, quindi per tutte le colonne:
* Per la trasformazione lungo le righe: analogamente per una riga sola richiede operazioni, quindi per tutte le colonne:

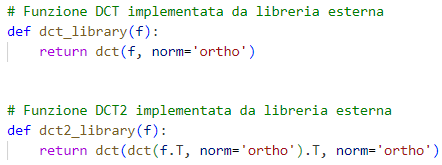
Avendo quindi che la complessità totale utilizzando questo metodo è:  
  
Avendo quindi che questa strategia è computazionalmente più efficiente (e rispecchia anche la complessità di implementazione che ci aspettavamo all’inizio)

**NOTA**: matrici quadrate

Nota che per entrambi i risultati finali se operiamo con una matrice quadrata allora i “” diventano “”

## 1.6. Utilss.py: Metodi di libreria per DCT e DCT2:

Come già anticipato si usano i metodi della libreria SciPy per avere un paragone con altre implementazioni di DCT e DCT2, di seguito vengono mostrati i metodi wrapper che chiamano le funzioni di libreria:



La funzione DCT2 prende in input gli stessi parametri della funzione DCT, ma viene richiamata due volte per creare la matrice bidimensionale sul trasposto della matrice fornita in input. Si da in pasto al metodo DCT2 la matrice trasposta appunto per creare un analogia con il codice scritto manualmente: qui, la trasposizione fa sì che la prima trasformazione venga applicata lungo le righe (tramite ), e la seconda lungo le colonne (invertendo con un altro ). Se non si usasse la trasposizione T nella funzione dct2\_library, la trasformazione sarebbe applicata solo lungo una dimensione della matrice (ad esempio, solo lungo le righe o solo lungo le colonne), anziché su entrambe le dimensioni.

**NOTA**: Coefficienti di normalizzazione

Si vuole far notare però che questa libreria implementa il fattore di scaling α in maniera analoga da come lo abbiamo appena visto dalle descrizioni teoriche che abbiamo dato nelle sezione [1.4](#_1.4._Utils) ma in maniera diversa da come abbiamo visto l’implementazione teorica della DCT.

Abbiamo infatti visto durante le lezioni teoriche che la formula per la DCT sarebbe:  
  
dove: (con vettore base della DCT)

* se allora (che possiamo trasformare quindi in moltiplicazione per )
* se allora (che possiamo trasformare quindi in moltiplicazione per )

A questi noi scegliamo di aggiungere aggiungela radice quadrata per fare normalizzazione, senza normalizzazione il prodotto scalare di un vettore base con sé stesso sarebbe proporzionale a N. La radice quadrata viene utilizzata per compensare questa dipendenza.

Ma se andiamo ad esplorare la [documentazione di SciPyci](https://docs.scipy.org/doc/scipy/reference/generated/scipy.fftpack.dct.html) (sotto dct type2 e norm=ortho) possiamo vedere che questa implementa il calcolo dei coefficienti di normalizzazione in maniera leggermente diversa, ma molto simile, a quella che abbiamo deciso di applicare:

* se allora (trasformabile quindi in moltiplicazione per )
* se allora (trasformabile quindi in moltiplicazione per )

## 1.7. Analsi dei risultati e conclusioi: on

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:

Il minimo di corrisponde alla soluzione del sistema . Mentre la funzione di aggiornamento è:

Dove:

* è il residuo

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

1. **Calcolo del residuo:** Si inizia con una stima iniziale del vettore delle incognite ​. Con questo si calcola il residuo , che rappresenta l'errore assoluto tra il valore attuale e il vettore della soluzione .
2. **Direzione della discesa del gradiente**: Il residuo appena calcolato si usa come direzione di discesa, poiché corrisponde al gradiente della funzione valutato in :

abbiamo che questo gradiente punta nella direzione di crescita massima della funzione , quindi si ha che, per minimizzare la funzione, dobbiamo muoverci in direzione

1. **Aggiornamento :** Si cerca il miglior passo lungo la direzione minimizzando , ottenendo:

calcolato alpha si può poi aggiornare la soluzione:

abbiamo dunque che ogni aggiornamento riduce il più possibile lungo la direzione scelta. Questo garantisce che il metodo scenda più velocemente possibile in quella direzione.

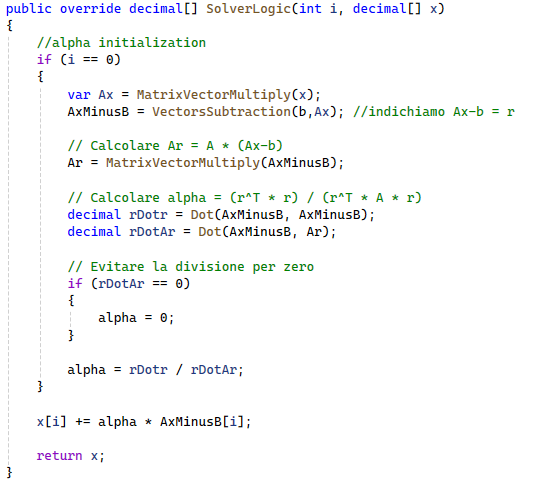
Se Consideriamo , l'i-esimo elemento del vettore . La formula di aggiornamento può essere scritta come:

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 appena descritta:

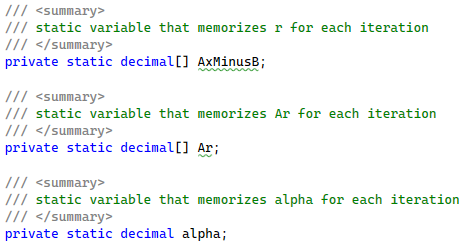
Calcolo del residuo:

Calcolo di alpha:

Aggiornamento di , ossia:



Nota che per tenere traccia dei valori del residuo e di alpha si definiscoefini 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 (ce se ne accorge perché i==0) allora il valore viene sovrascritto:



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



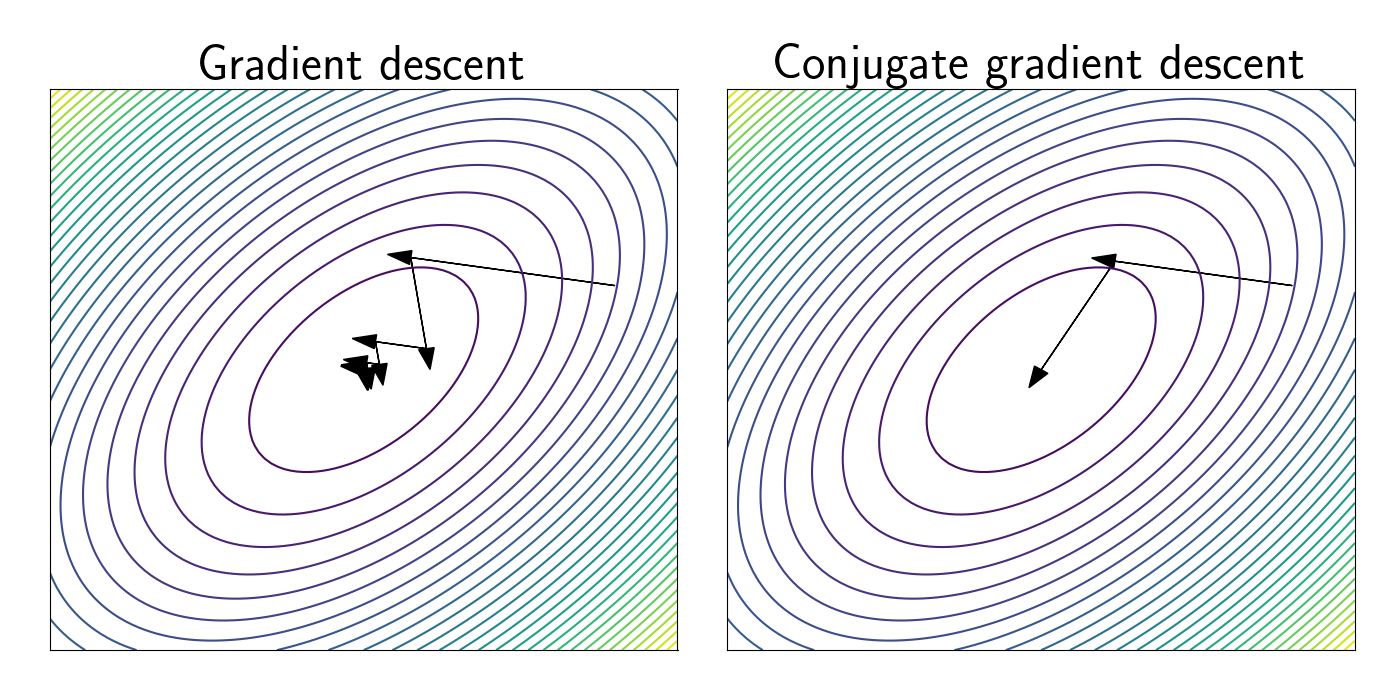
## 1.6. Implementazione GradientSolver

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 positive. **passo**. Questo metodo garantisce la convergenza in un numero finito di passi per matrici simmetriche e definite positive. Anche qui quindi abbiamo che la funzione che a funzione da minimizzare è:

Il minimo di corrisponde alla soluzione del sistema . Mentre la funzione di aggiornamento è:

Dove:

* (nota che è calcolato con al posto che )
* è il passo del gradiente (vettore derivato da ) che serve ad indirizzare più precisamente 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 ​. Con questo si calcola:
   1. il residuo:
   2. Il passo:
2. **Aggiornamento di** : Una volta calcolati e è possibile calcolare alpha:ento del

ed avendo alpha e a è possibile calcolare l’aggiornamento del vettore delle incognite:

1. **Controllo convergenza** : Si controlla se la soluzione calcolata converge:
2. **Aggiornamento delle variabili per la prossima iterazione**: Abbiamo che dopo aver verificato se il vettore permette la convergenza, in caso negativo allora aggiorniamo le variabili per la prossima istanza:
   1. Residuo: che è dimostrabile essere un modo più efficiente di calcolare
   2. Beta: è un coefficiente di supporto per il calcolo di .
   3. Passo:

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

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

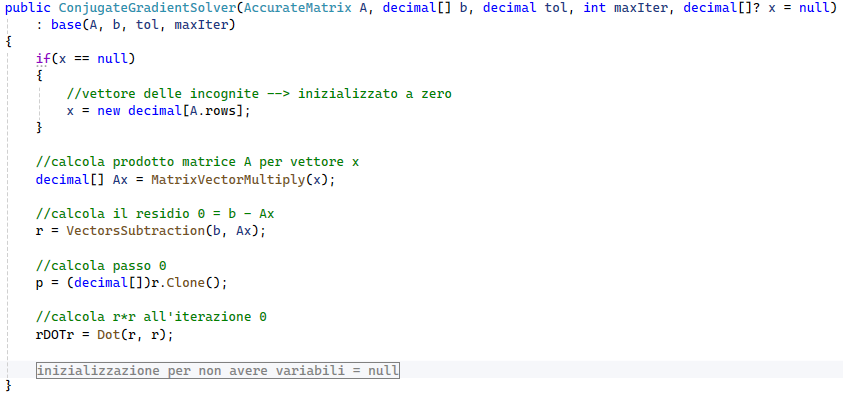
Diversamente dalle implementazioni precedenti, per realizzare questa serie di step in linguaggio C#, oltre che ad utilizzare l’implementazione del metodo SolverLogic all’interno della classe GradientSolver dovremo anche fare utilizzo del metodo costruttore della classe (in quanto, come abbiamo visto 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)

Calcolo di in quanto è una variabile utile sia per il calcolo di alpha che di beta, non avrebbe senso calcolarla 2 volte più tardi

Calcolo del primo passo:

Calcolo del primo residuo:

Se non viene fornito il vettore delle incognite allora viene inizializzato a 0



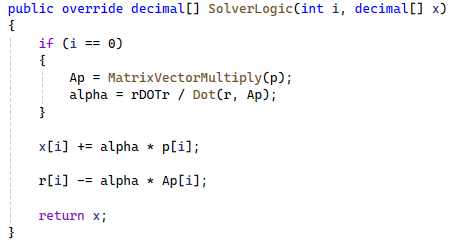
Una volta fatto il setup delle variabili (corrispondente allo step 1 descritto sopra) possiamo vedere come il metodo metodo SolverLogic (all’interno della classe GradientSolver) implementi esattamente lo **step 2** (formula di aggiornamento di ogni componente ) appena descritta:

Aggiornamento di

Calcolo di alpha:

Nota che questa operazione viene eseguita una sola volta per ogni

Aggiornamento di



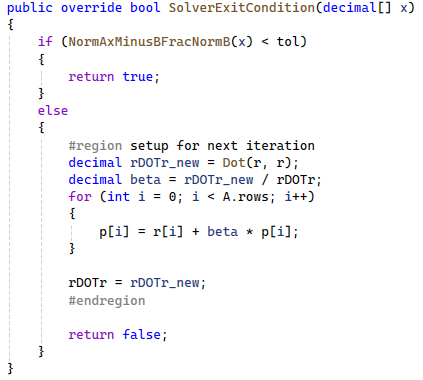
Nota che nella descrizione che abbiamo dato prima l’aggiornamento del residuo sarebbe da fare dopo il controllo della convergenza, noi in questo caso lo calcoliamo prima in quanto nel metodo SolverExitCondition (per scelta di design) non viene passato il parametro quindi non sarebbe possibile calcolare ma andrebbe calcolato ad ogni iterazione tutto (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:

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

* Calcolo di beta:
* Aggiornamento del passo:
* Aggiornamento della variabile rDOTr:

**Step 3**: Controllo della convergenza , si controlla che:



Come si può notare, questo è il metodo che ha reso necessario rendere SolverExitCondition un metodo astratto, in quanto questa particolare istanza di questo metodo 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

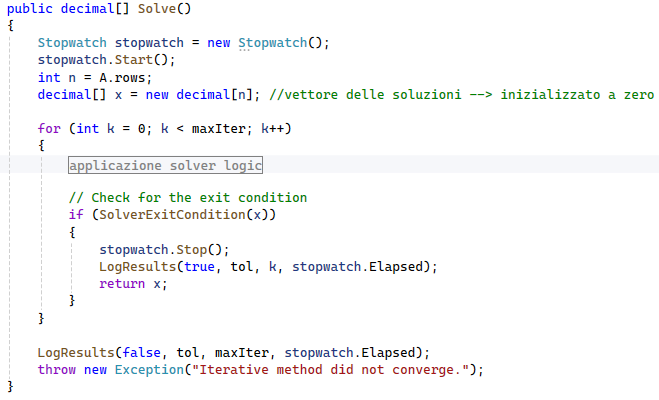
Dopo avere dettagliatamente descritto il funzionamento della libreria (e la teoria dietro la sua realizzazione) possiamo ora passare all’analisi dei risultati prodotti dall’esecuzione della libreria stessa.

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 ()
2. Risultati tra paragoni temporali della libreria appena sviluppata con altre librerie open source e paragoni su sistemi operativi diversi

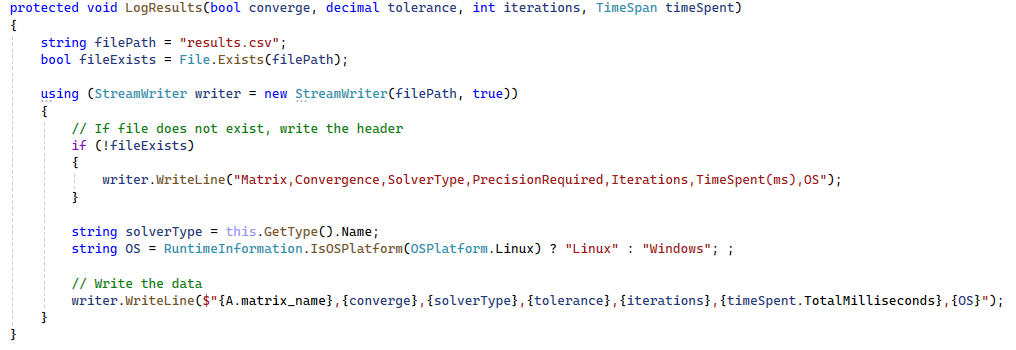
Per realizzare i grafici che vedremo di seguito è stata aggiunta alla libreria una funzionalità di logginglibreseguivedregraficrealiz che possiamo vedere anche all’interno del metodo Solve (all’interno della classe IterativeSolver):

Log risultati in caso di convergenza



Log risultati in caso di non convergenza

Dove questo metodo permette semplicemente di salvare i risultati in un file .CSV



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

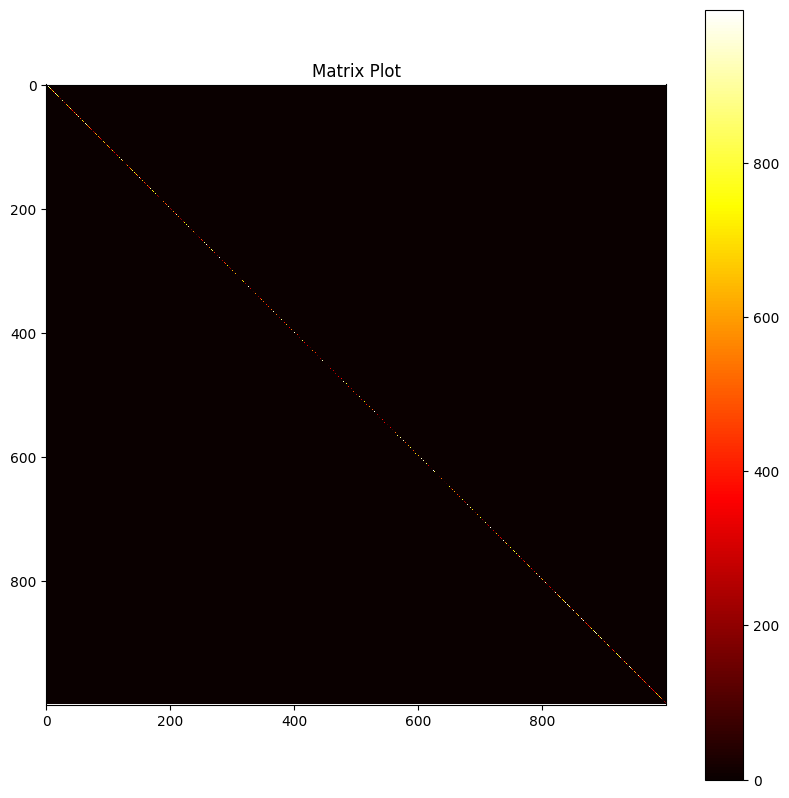
Questa sezione è dedicata ai risultati del progetto. Presenta un'analisi comparativa dei diversi metodi iterativi utilizzati per risolvere sistemi lineari sparsi. Tale analisi è illustrata mediante grafici che riportano:

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

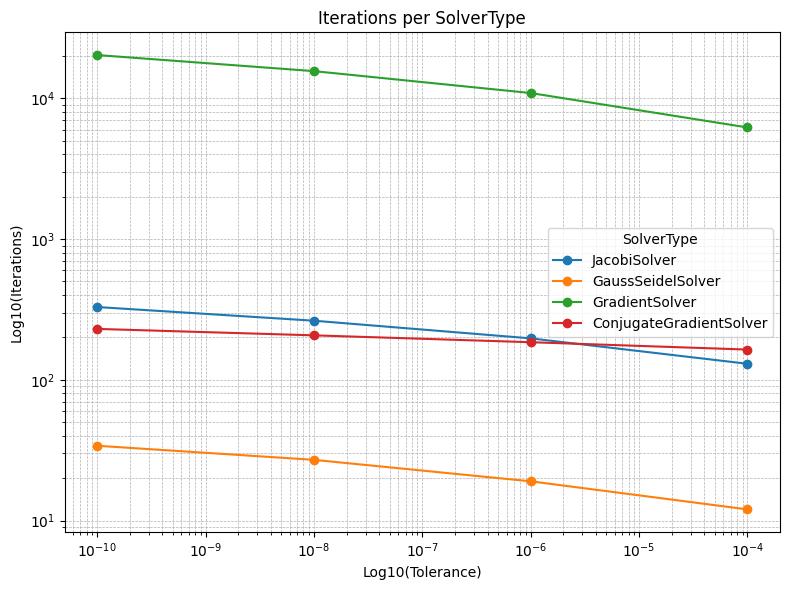
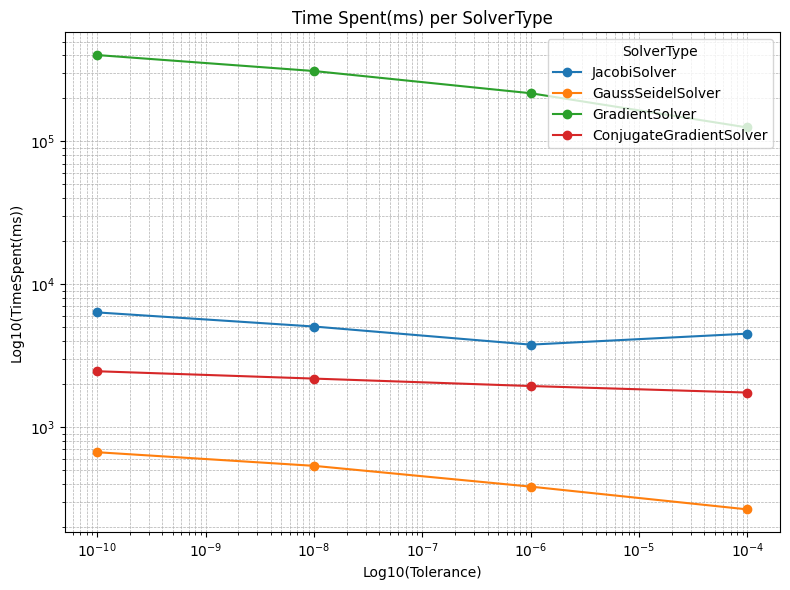
considerando differenti valori di tolleranza ().

#### 2.1.1. Spiegazione risultati spa1.mtx

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



Di seguito i risultati dell’elaborazione:

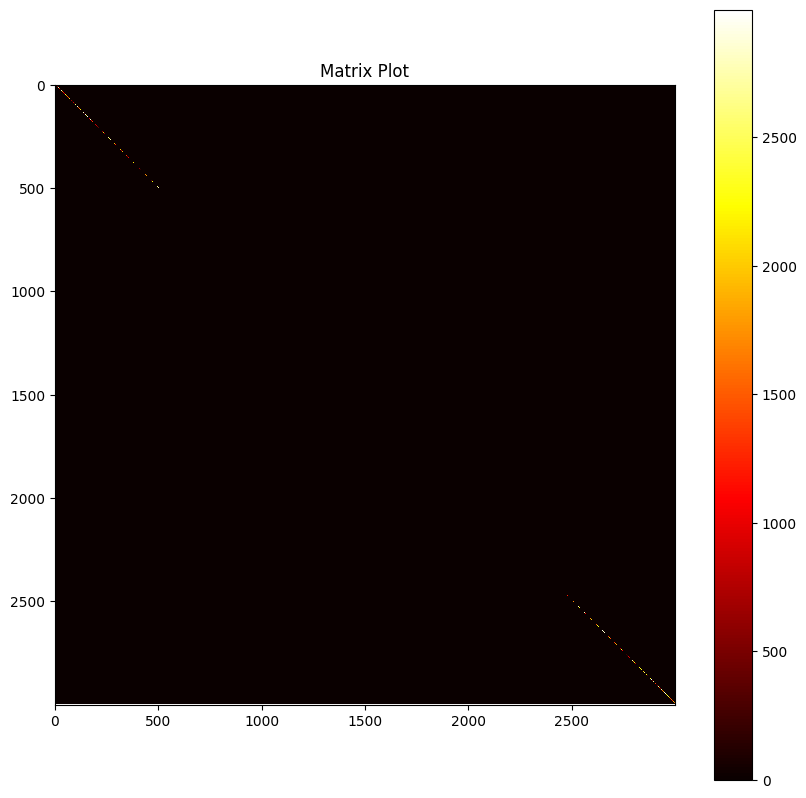


Abbiamo che:

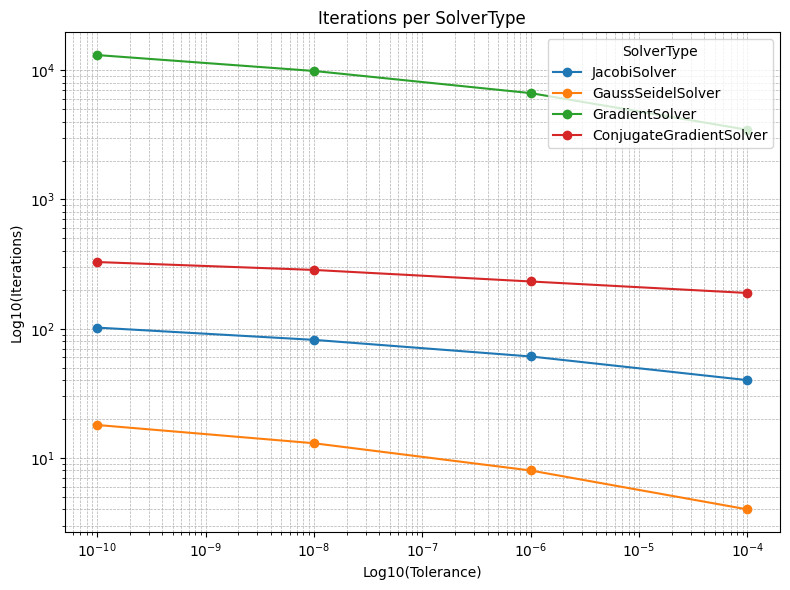
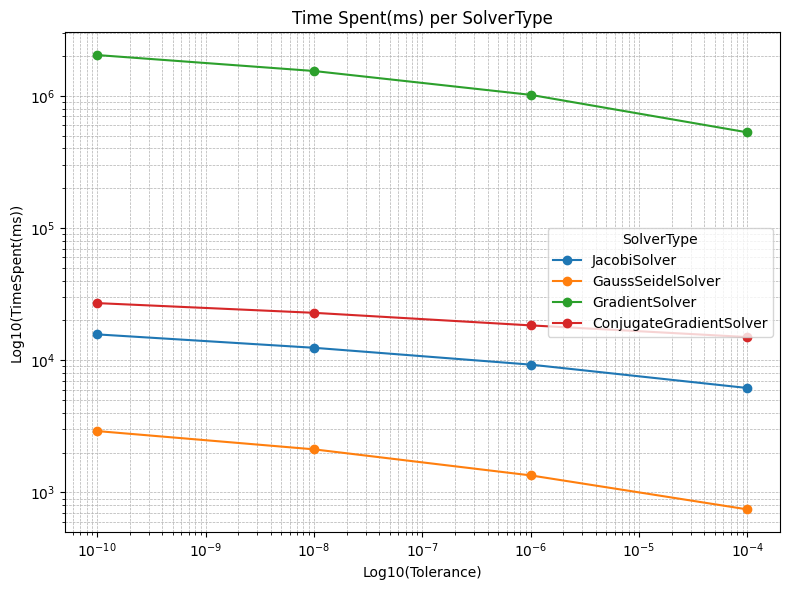
* **Gradient Solver**: Non risulta particolarmente efficiente ne a livello di numero di iterazioni ne a livello di tempo impiegato per raggiungere la soluzione, risultando tra tutti i metodi il peggiore a risolvere questo sistema. Questo comportamento è probabilmente dovuto al fatto che generalmente questo metodo è utile quando si ha una buona stima iniziale della soluzione e che la velocità di convergenza è lenta su sistemi mal condizionati.
* **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 perhcè è un miglioramento dell’algoritmo del gradiente, inoltre generalmente funziona bene quando si ha sistemi grandi e sparsi
* **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

#### 2.1.2. Spiegazione risultati spa2.mtx

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



Di seguito i risultati dell’elaborazione

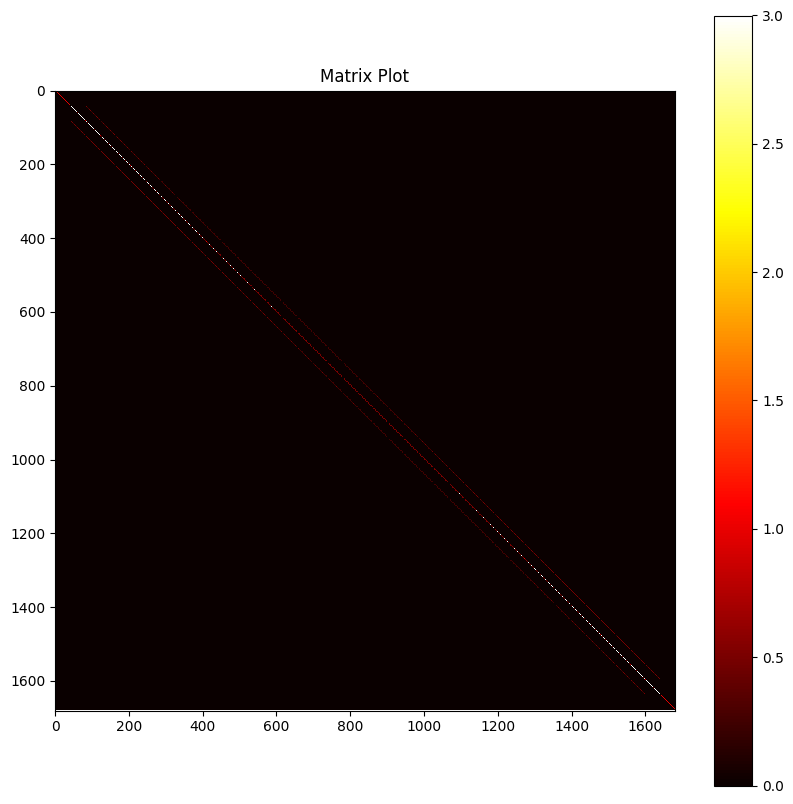


Abbiamo che è una matrice molto simile alla precedente ma più grande, quindi i risultati sono abbastanza simili:

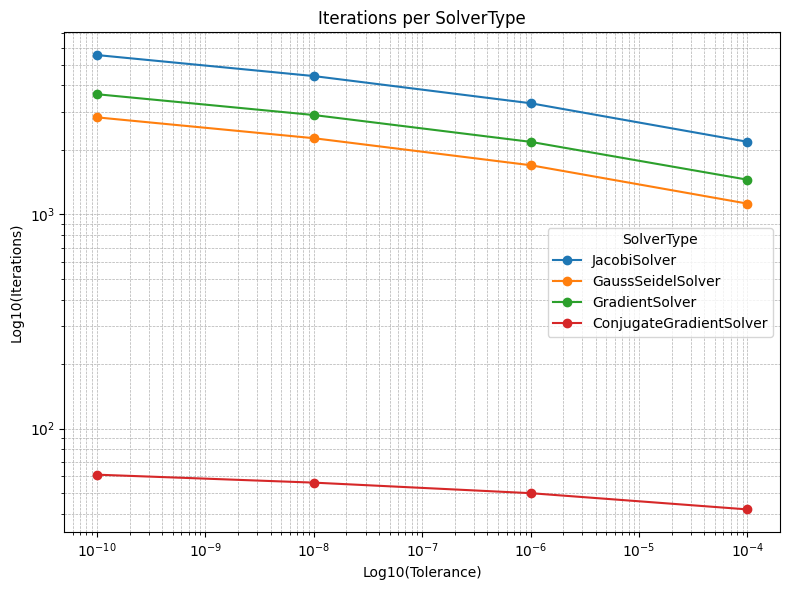
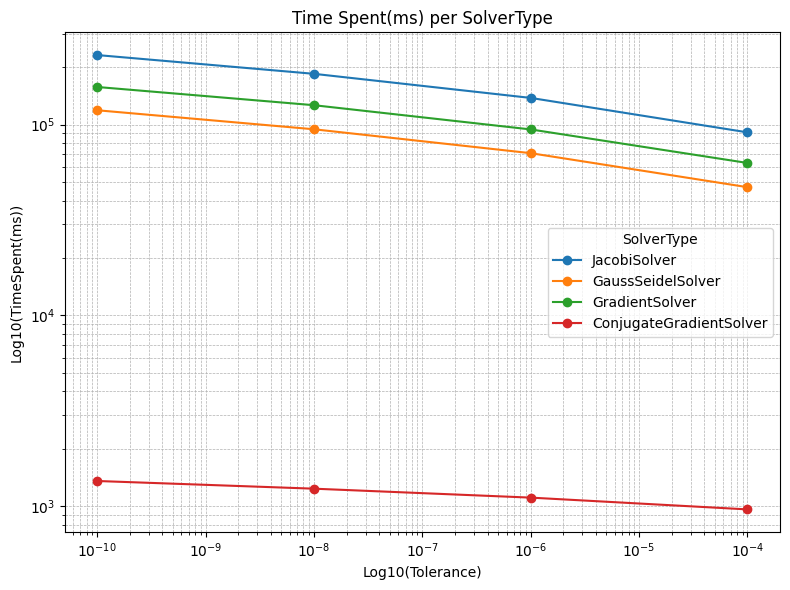
* **Gradient Solver**: Come prima non risulta particolarmente efficiente ne a livello di numero di iterazioni ne a livello di tempo impiegato per raggiungere la soluzione, risultando tra tutti i metodi il peggiore a risolvere anche questo sistema, per le stesse motivazioni date precedentemente.
* **Jacobi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



Di seguito i risultati:

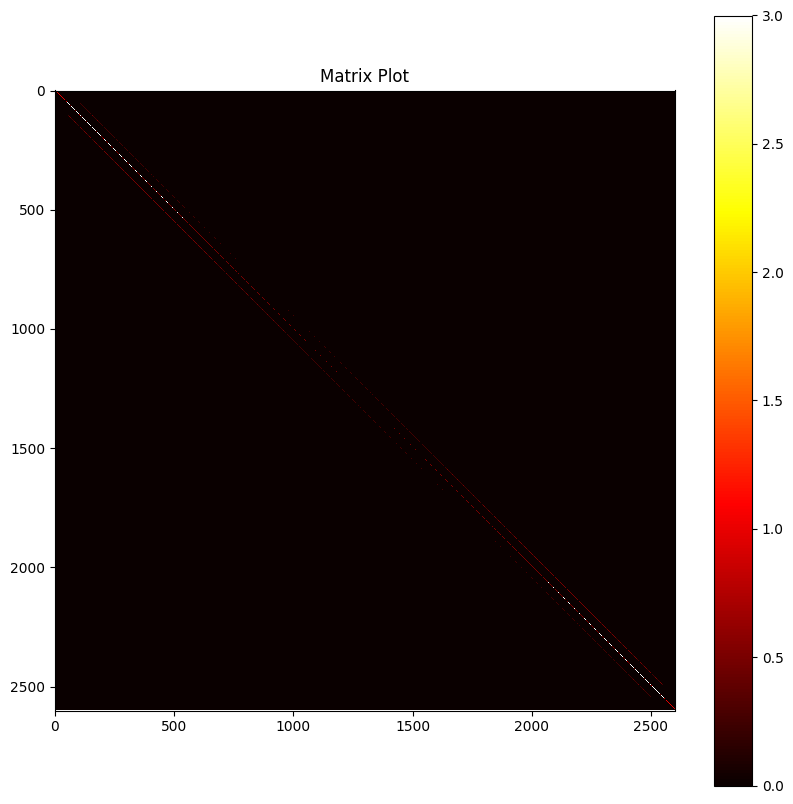


Questa matrice, oltre ad avere numeri sulla diagonale presenta anche altre 2 righe parallele alla diagonale, possiamo comunque definirla a dominanza diagonale perchè i valori non nulli si trovano principalmente sulla diagonale principale o nelle vicinanze. Data questa differenza i risultati di performance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

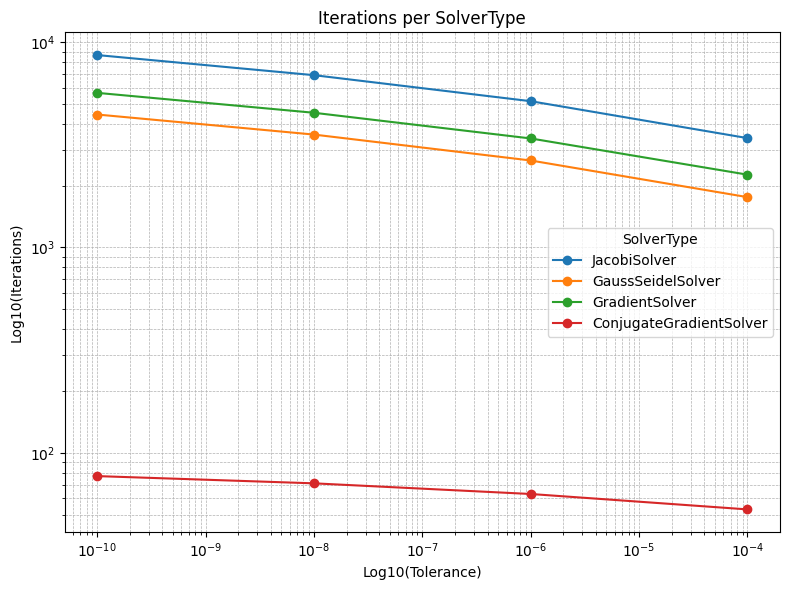
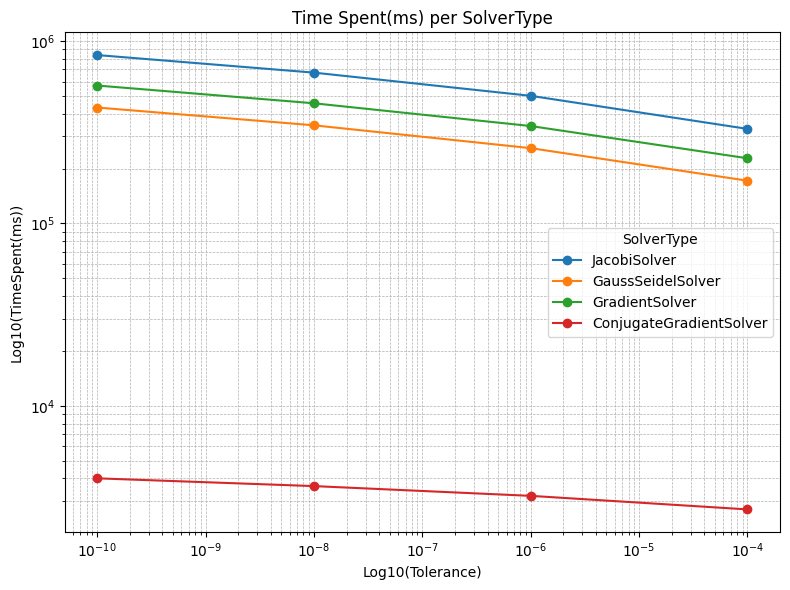
* **Jacobi Solver**: Risulta diventare il peggiore sia a livello di numero di iterazioni sia a livello di tempi di calcolo. Ha senso le sue performance diminuiscano in quanto il sistema, è “meno diagonalmente dominante” rispetto a prima. Prima avevamo solo numeri sulla diagonale e raggiungeva performance 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 da 0 a 3.5, mentre nelle 3 precedente 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):



Di seguito i risultati:



Questa matrice è praticamente 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

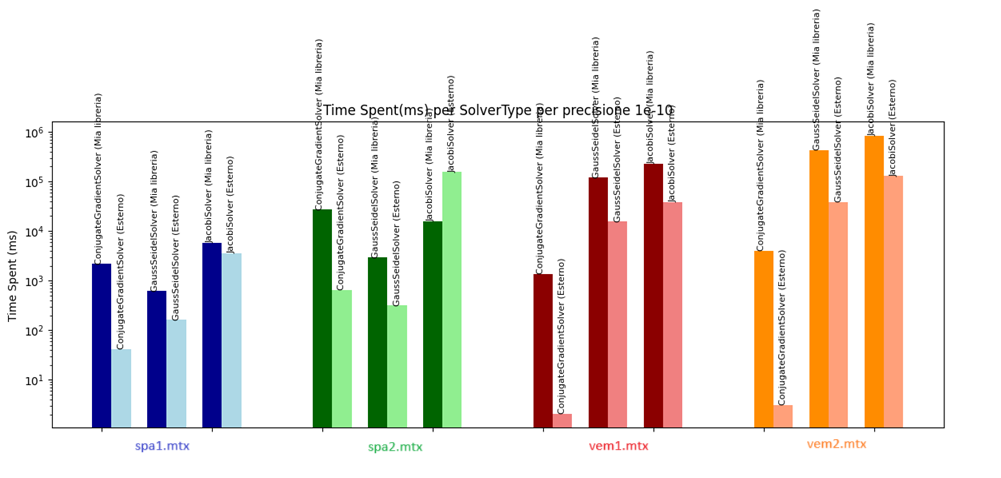
Sulla base dei risultati ottenuti, è stato possibile confermare sia la piena funzionalità della libreria sviluppata sia la sua conformità ai principi teorici appresi. Abbiamo infatti che per ogni metodo utilizzato è consistente che il numero di iterazioni necessarie sia direttamente proporzionale al 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

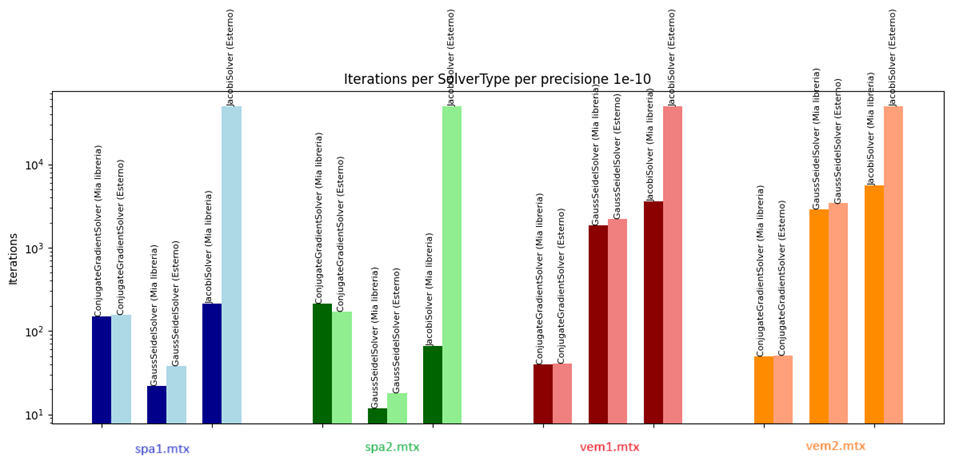
Si utilizza quest’ultima sezione della relazione per parlare dell’andamento dello sviluppo delle diverse versioni della libreria.

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 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](https://www.nuget.org/stats/packages/easy_matrix?groupby=ClientName)). Questo 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):



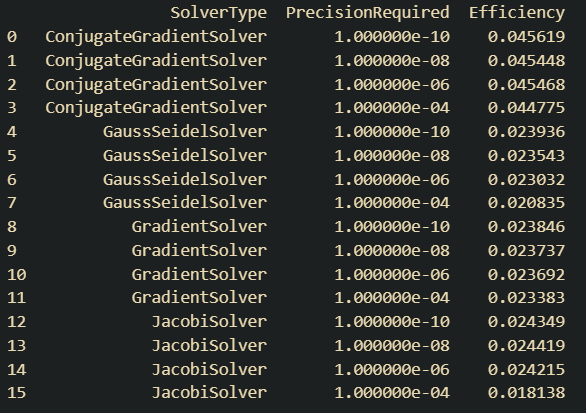
Questo è semplicemente dovuto al fatto che, appunto, ho dovuto implementare da zero molti metodi “basilari” che sicuramente altri linguaggi di programmazione / librerie implementano in maniera più ottimizzata. Ed è poi anche dovuto al fatto che il C# è un linguaggio molto pesante, in quanto il compilatore non fa’ altro che tradurre le istruzioni in CIL (un linguaggio simile al C) e poi dal CIL al linguaggio macchina.

Diventa però osservare che siccome la nostra libreria utilizza dati di tipo decimal richiede molte meno iterazioni rispetto che le stesse librerie open source del paragone precedente:



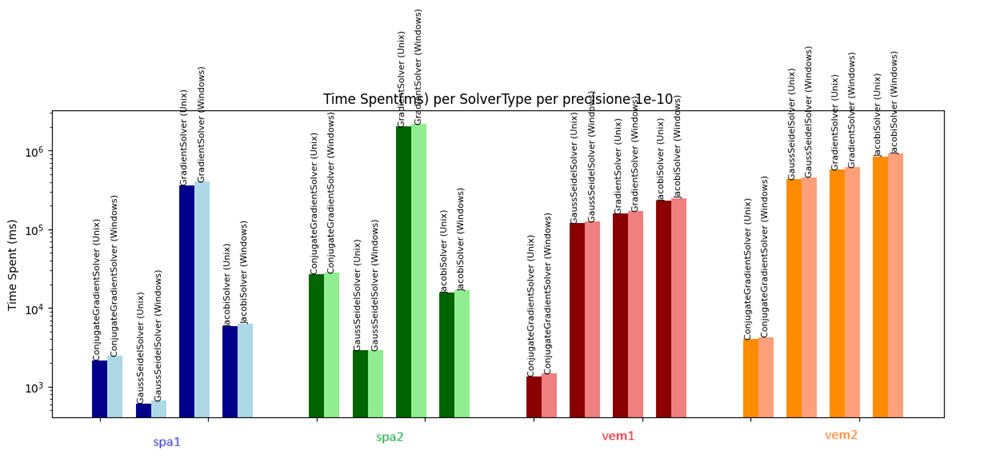
Questo succede perchè avendo a disposizione più cifre decimali ad ogni iterazione la nostra libreria arriverà sempre ad un risultato (anche se approssimato) più vicino alla soluzione corretta che rispetto ad altre librerie; a lungo termine questo porta ad avere un numero di iterazioni minore per arrivare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

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



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](https://www.deepin.org/index/en) 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.