Dipartimento di Informatica
Corso di Laurea Magistrale in Informatica
Esame di Metodi del calcolo scientifico

Calcolo di soluzioni di matrici simmetriche definite postive con Matlab e Python

Di Capua Marco - 878295
Gherardi Alessandro - 817084
Pallini Vincenzo - 907303

Anno Accademico 2022 - 2023

Indice

1 Introduzione

Il progetto si propone di analizzare l'implementazione di un solutore, il quale è stato scelto essere il metodo Cholesky, per risolvere sistemi lineari con matrici sparse, simmetriche e definite positive in ambienti di programmazione open source. L'obiettivo è confrontare queste implementazioni con MATLAB in termini di tempo, accuratezza, impiego della memoria, facilità d'uso e documentazione per determinare se l'ambiente a pagamento giustifichi il suo costo fornendo migliori prestazioni rispetto agli ambienti open source gratuiti. Le matrici sparse, che sono comunemente utilizzate in problemi di calcolo scientifico, hanno la caratteristica di possedere molti elementi uguali a zero. La presenza di questi elementi consente, in fase di computazione, di evitarne il calcolo consentendo di ridurre il costo computazionale del sistema di equazioni. Per esempio, il metodo Choleski, un'alternativa all'eliminazione di Gauss, può essere applicato a matrici simmetriche e definite positive per la loro fattorizzazione. In particolare, questo metodo consente di esprimere la matrice originale come il prodotto di due matrici triangolari semplificando in tal modo la risoluzione di sistemi di equazioni lineari. Il progetto, invece, intende confrontare il solutore base di MATLAB con almeno una libreria open-source su sistemi Windows e Linux, utilizzando matrici provenienti dalla SuiteSparse Matrix Collection.

2 Specifiche del calcolatore e versioni software

Per il seguente progetto è stato utilizzato un calcolatore con sistema operativo Windows 11, su cui è stata creata una partizione di 100 GB di memoria fissa nella quale è installato un sottosistema Linux Ubuntu. Il calcolatore in questione dispone delle seguenti componenti hardware:

- Scheda madre: Asus ROG STRIX B550-F GAMING ATX AM4
- Processore: AMD Ryzen 7 5800X 3.8 GHz 8-Core
- Memoria RAM: Crucial Ballistix DDR4-3600 CL16 da 32 GB (2x16 GB)
- Memoria fissa: SSD Sabrent Rocket 4.0 da 1 TB
- Memoria fissa: HDD Western Digital Blue 3.5" 7200 RPM da 2 TB

Per quanto riguarda i linguaggi di programmazione sono stati usati Matlab versione R2022a e Python versione 3.10. Al fine di consentire un confronto dei risultati ottenuti nei due ambienti di lavoro, sia sull'ambiente Windows che sull'ambiente Linux, sono state installate le stesse versioni di Matlab e Python.

2.1 Librerie Matlab

Matlab è un software di calcolo numerico e di analisi dotato di una vasta gamma di funzioni matematiche, algoritmi numerici e librerie per l'analisi dati, il che lo rende uno strumento potente per la risoluzione di problemi di calcolo scientifico. Matlab supporta la programmazione ad oggetti e può essere utilizzato per la risoluzione di problemi di calcolo scientifico tra cui la risoluzione di sistemi di equazioni lineari, attraverso l'utilizzo di funzioni di algebra lineare come la fattorizzazione di Cholesky.
Attraverso la funzione implementata da Matlab *matlab.codetools.requiredFilesAndProducts* è stato possibile risalire alle dipendenze dello script implementato, quindi ai possibili add-on utilizzati tra quelli installati e non sono emerse altre dipendenze se non quella a Matlab stesso. Da questo si evince che Matlab comprende all'interno della sua versione base i file e i prodotti necessari per risolvere matrici sparse, simmetriche e definite positive e per leggere la memoria utilizzata.

2.2 Librerie Python

Python è un linguaggio di programmazione ad alto livello, interpretato e dinamicamente tipizzato. Ciò significa che il codice sorgente Python viene eseguito direttamente senza la necessità di una fase di compilazione e che le variabili non devono essere esplicitamente dichiarate con un tipo. Python è multipiattaforma, il che significa che può essere eseguito su diversi sistemi operativi come Windows, macOS e Linux. Inoltre, Python è open source, il che implica che il suo codice sorgente è disponibile gratuitamente e può essere modificato e distribuito liberamente. A differenza di Matlab, Python non contiene nativamente metodi per risolvere matrici sparse, simmetriche e definite positive, perciò bisogna avvalersi di librerie per la soluzione del problema. Nello specifico è stata scelta la libreria *SciPy* versione 1.9.0. *SciPy* è una libreria open source di Python che fornisce funzioni e strumenti per lavorare con algoritmi matematici e scientifici. È costruita sulla base di *NumPy*, una libreria per la manipolazione di array multidimensionali, e fornisce funzionalità aggiuntive per l'ottimizzazione, l'integrazione, l'interpolazione, l'algebra lineare, le trasformate di Fourier, il trattamento dei segnali e delle immagini, e molto altro. Di questa libreria sono stati utilizzati diversi sotto pacchetti:

- ***scipy.sparse***: sotto pacchetto di *SciPy* che fornisce strutture dati e funzioni per lavorare con matrici sparse, ovvero matrici in cui la maggior parte degli elementi sono zero. Le matrici sparse sono utili per risparmiare memoria e migliorare l'efficienza computazionale quando si lavora con grandi matrici contenenti molti valori nulli, in quanto questi valori in fase di computazione non vengono calcolati.

- ***scipy.sparse.linalg***: sotto pacchetto di *SciPy.sparse* che fornisce funzioni di algebra lineare per lavorare con matrici sparse. Tra queste funzioni, è stata utilizzata *scipy.sparse.linalg.norm*, che calcola la norma di una matrice sparsa senza la necessità di calcolare i valori degli elementi nulli. La norma è una misura della "grandezza" o "lunghezza" di una matrice o di un vettore e può essere utilizzata per confrontare matrici o vettori o per misurare l'errore tra due soluzioni approssimate.

- ***scipy.linalg***: sotto pacchetto di *SciPy* che fornisce funzioni di algebra lineare per lavorare con matrici dense. Questo sotto pacchetto include funzioni per risolvere sistemi lineari, calcolare autovalori e autovettori, decomporre matrici, e molto altro.

Per quanto riguarda la metodologia risolutiva vera e propria è stata usata la funzione *spsolve* del sotto pacchetto *scipy.sparse.linalg*. Questa funzione prende in input una matrice quadrata A, un array b, ovvero la soluzione e produce in output un array x risolvendo l'equazione lineare $A * x = b$. Altre librerie di supporto utilizzate sono:

- ***NumPy*** 1.22.4: libreria essenziale poiché offre funzioni matematiche complete, routine di algebra lineare e gestione e creazione di matrici nonché la gestione e la manipolazione di grandi quantità di dati numerici in Python, grazie alla sua elevata efficienza computazionale.

- ***memory_profiler*** 0.61.0: libreria Python utilizzata per la profilazione della memoria, ovvero per analizzare l'uso della memoria di un programma al fine di individuare eventuali problemi di allocazione o rilascio di memoria o per monitorarne l'andamento mediante la funzione *memory_usage* che ne permette la lettura.

- ***Pandas*** 1.4.3: libreria open source per la gestione e l'analisi dei dati in Python, che fornisce strutture dati flessibili e potenti come i dataframe per la manipolazione di dati strutturati e non strutturati. Pandas è particolarmente utile per eseguire operazioni di preprocessing, pulizia, elaborazione e analisi di dati provenienti da fonti diverse, come file CSV, fogli di calcolo Excel, database SQL e molto altro.

- ***matplotlib*** 3.5.1: libreria open source per la visualizzazione dei dati in Python, che fornisce un'ampia gamma di funzionalità per la creazione di grafici, istogrammi, diagrammi a barre, diagrammi a dispersione, mappe di calore e molti altri tipi di visualizzazioni.

3 Matrici analizzate

Le matrici in analisi sono matrici simmetriche e definite positive. Questo tipo di matrici hanno la caratteristica di essere matrici quadrate, in cui gli elementi sono simmetrici rispetto alla diagonale principale, e definite positive, in quanto tutti i loro autovalori sono strettamente positivi. La presenza di matrici simmetriche positive semplifica la risoluzione di equazioni lineari, riduce la complessità computazionale e garantisce stabilità e accuratezza nelle soluzioni. Per queste ragioni, che semplificano la risoluzione di equazioni lineari, riducono la complessità computazionale e garantiscono stabilità e accuratezza nelle soluzioni, le matrici simmetriche positive hanno un ruolo rilevante nel campo del calcolo scientifico e vengono ampiamente utilizzate in una vasta gamma di applicazioni. Nel progetto sono state analizzate alcune matrici che fanno parte della **SuiteSparse Matrix Collection** che colleziona matrici sparse derivanti da applicazioni di problemi reali. Seguono le tabelle ordinate per peso:

- **ex15**: peso 555KB, dimensioni $6,867 \times 6,867$
- **shallow water1**: peso 2,263KB, dimensioni $81,920 \times 81,920$
- **cfd1**: peso 14,164KB, dimensioni $70,656 \times 70,656$
- **cfd2**: peso 23,192KB, dimensioni $123,440 \times 123,440$
- **parabolic fem**: peso 13.116KB, dimensioni $525,825 \times 525,825$
- **apache2**: peso 8,302KB, dimensioni $715,176 \times 715,176$
- **G3 circuit**: peso 13,883KB, dimensioni $1,585,478 \times 1,585,478$
- **StocF-1465**: peso 178,368KB, dimensioni $1,465,137 \times 1,465,137$
- **Flan 1565**: peso 292,858KB, dimensioni $1,564,794 \times 1,564,794$

Data la cospicua dimensione e peso di alcune tabelle che incrementano notevolmente i tempi di risoluzione richiesti, l'ambiente MATLAB è stato utilizzato per analizzare solo le seguenti 6 matrici:

- ex15
- shallow water1
- apache2
- G3 circuit
- cfd1
- cfd2

Mentre Python è stato utilizzato per analizzare solo queste 4 matrici:

- ex15
- shallow water1
- apache2
- G3 circuit

La scelta di analizzare un numero limitato di matrici è stata fatta per ottimizzare i tempi di calcolo e concentrarsi sulle matrici più rilevanti per il progetto.

4 Risultati estratti

In questa sezione verranno analizzati e confrontati i risultati ottenuti mediante tabelle e grafici. Verranno confrontate le capacità di Matlab e Python su Windows e Linux in relazione ai tempi di risoluzione della matrici, l'errore relativo ottenuto e la memoria utilizzata dal calcolatore.

4.1 Risultati generali

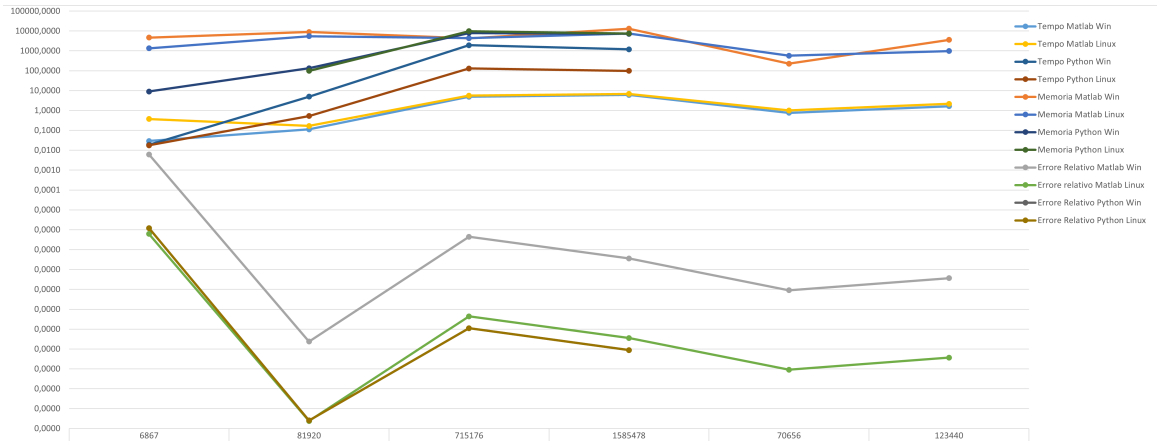

Fig. 1: Grafico comprensivo di tutte le misure

La Fig.1 mostra tutte le metriche estratte, ovvero tempo, memoria e errore relativo, per le 4 combinazioni possibili di Matlab e Python con Windows e Linux. La scala scelta per la rappresentazione è quella logaritmica, poiché gli ordini di grandezza delle metriche risultano essere molto diversi. Si può notare come, parlando di tempi, Matlab performi decisamente meglio di Python, anche se l'utilizzo di Python su Linux migliora i risultati rispetto che su Windows. Per quanto riguarda la memoria i risultati sono simili per tutti i casi, tranne per una piccola differenza per le matrici più piccole. L'errore relativo ha differenze minimali, tranne che per la combinazione Matlab-Windows che performa peggio, anche se il grafico fa pensare il contrario. Questo a causa della scala logaritmica che su valori piccoli tende a allargare le differenze.
In ogni caso il grafico, essendo le misure su ordini di grandezza differenti, risulta abbastanza complesso alla lettura, per questo si è deciso di andare ad analizzare i valori singolarmente.

4.2 Tempi

I tempi di calcolo, ovviamente, risultano essere un fattore essenziale da considerare quando si vuole risolvere problemi matematici. In particolare bisogna considerare che il solutore classico per sistemi lineari che calcola $A/b = x$ ha una complessità temporale che al peggio raggiunge un $O(n^3)$, con n dimensione dell'input. Questa considerazione è importante, poiché il tempo, pur essendo limato superiormente da un polinomio, potrebbe crescere molto nel caso di matrici molto grandi.

Dimensione	6867	81920	715176	1585478	70656	123440
Tempo Matlab Win (s)	0,03	0,11	4,92	6,06	0,76	1,63
Tempo Matlab Linux (s)	0,37	0,17	5,57	6,83	1,01	2,15
Tempo Python Win (s)	0,02	4,96	1930,59	1196,52		
Tempo Python Linux (s)	0,02	0,53	130,01	98,25		

Tabella 1: Tempi di calcolo in relazione alla dimensione

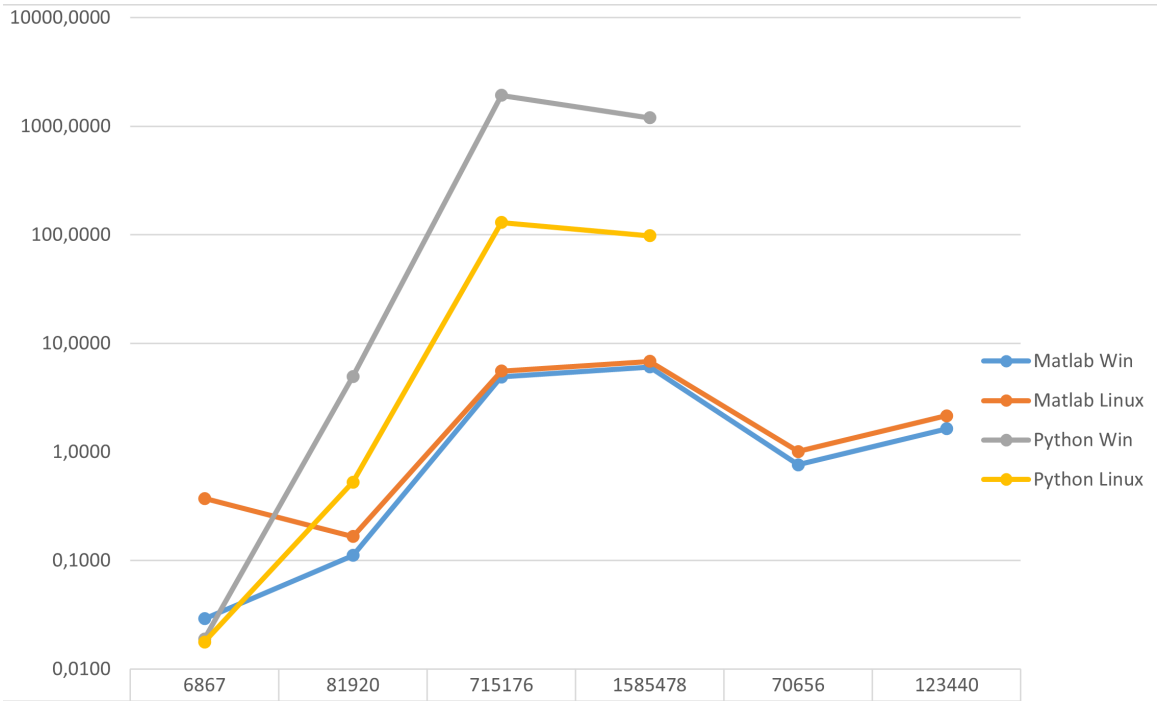

Fig. 2: Tempi di calcolo

Come si può notare dalla Fig.2 i tempi di calcolo tra Matlab e Python sono molto diversi man mano che aumentano le dimensioni delle matrici, infatti è stato necessario utilizzare una scala logaritmica a causa dei diversi ordini di grandezza dei dati. Matlab performa molto bene sia su Windows che sul sistema Linux, nell'ordine dei $10^0 - 10^1$ secondi, cosa che non si può dire di Python. Quest'ultimo compie i calcoli con tempi diversi in base al sistema operativo: su Linux si aggirano sull'ordine dei 10^2 secondi per le matrici più grandi, mentre su Windows crescono esponenzialmente fino a 10^3 secondi. Sulle matrici più piccole invece i tempi rimangono pressoché invariati (10^1 secondi) per tutte le combinazioni prese in esame.
Questi dati risultano essere rilevanti per una azienda, poiché i tempi di calcolo posso incidere fortemente sulla riuscita e sulla consegna in tempo di un lavoro richiesto. Questa prima analisi aiuta a far pendere l'ago della bilancia verso la combinazione Matlab-Linux, poiché unisce i tempi bassi di Matlab con il sistema gratuito Linux.

4.3 Memoria

Anche la memoria utilizzata da un certo programma è un fattore rilevante, poiché strettamente legata al concetto di complessità temporale. Un buon algoritmo, infatti, dovrebbe cercare di utilizzare la quantità giusta di memoria mantenendo, al contempo, i tempi di calcolo accettabili. Spesso questa richiesta, però, è difficile da soddisfare, in quanto tendenzialmente spazio e tempo sono inversamente proporzionali. Infatti, in mancanza di tempo, si potrebbe utilizzare una enorme quantità di memoria per velocizzare il calcolo, viceversa si potrebbero allungare molto i tempi di calcolo in mancanza di memoria. Bisogna trovare il giusto *trade-off*. Queste considerazioni, benché interessanti, sono utili in casi nel quale tempo o spazio risultano avere un limite superiore stringente, cosa che non capita nel caso in esame.
Infatti, il calcolatore è stato in grado di performare senza limitazioni, poiché non limitato sul tempo e senza avere un limite stringente di memoria utilizzabile (32GB). La tabella e il grafico che seguono mostrano l'utilizzo della memoria:

Dimensione	6867	81920	715176	1585478	70656	123440
Memoria Matlab Win (Mb)	4636	8908	4288	13132	228	3580
Memoria Matlab Linux (Mb)	1340	5468	4368	7411	570	986
Memoria Python Win (Mb)	9	134	7939	7173		
Memoria Python Linux (Mb)	0	98	9717	7489		

Tabella 2: Memoria utilizzata in relazione alla dimensione

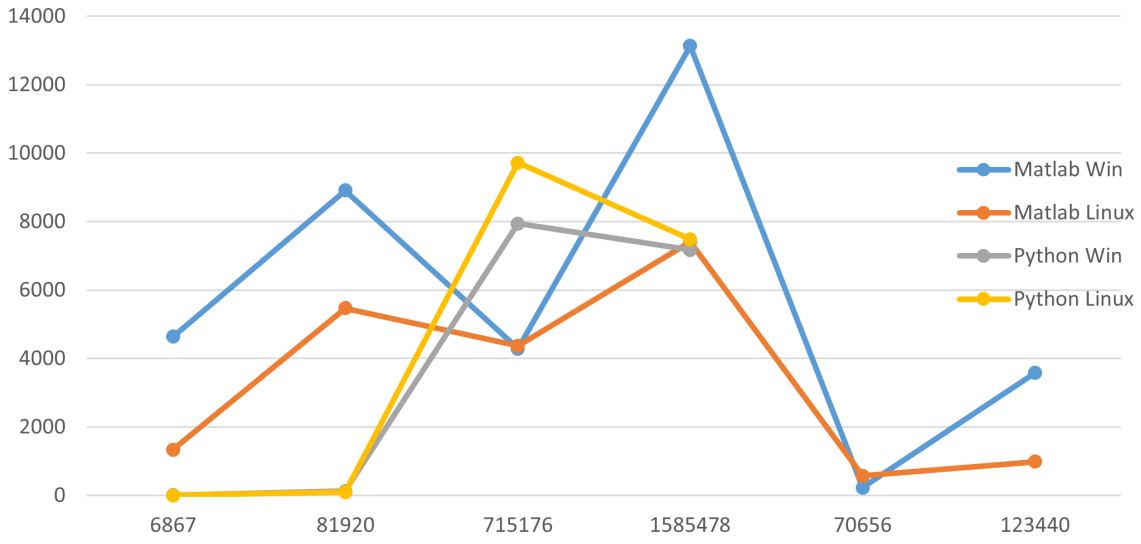

Fig. 3: Memoria utilizzata

Come si può notare in Fig.3 la memoria RAM utilizzata varia dal GB fino a un massimo di quasi 13 GB per la quarta matrice in Matlab-Windows. Partendo da Python si può vedere come sul sistema Linux ci sia un utilizzo leggermente maggiore di memoria, il che potrebbe indicare il motivo dei tempi più brevi di calcolo. Sfortunatamente, causa possibili errori della libreria di estrazione memoria, i dati sulle prime due matrici risultano essere molto bassi, forse a causa dei tempi di campionamento della memoria. Nella sottosezione seguente verrà analizzato più nel dettaglio l'utilizzo della memoria per Python.
Matlab, invece, crea due grafici molto simili, se non per un utilizzo più massivo di memoria per quanto riguarda Windows. Interessante è vedere che la dimensione della matrice non influenza troppo l'utilizzo di RAM, infatti la seconda matrice, sebbene più piccola, ha un valore più alto della terza. Questo perché Matlab possiede dei metodi nativi per capire il tipo di matrice passata e come gestirla al meglio. Importante è notare che la combinazione Matlab-Linux non supera mail gli 8GB di RAM utilizzata, quantità minima richiesta in un calcolatore di scarsa qualità.
Interessante è anche l'analisi della terza e quarta matrice. Matlab usa una quantità di memoria proporzionale alla dimensione della matrice, mentre Python il contrario. Questo fa pensare che Python sia meno "intelligente" nel gestire matrici del tipo specificato, il che spiegherebbe anche perché i tempi di calcolo sono maggiori.
Questa seconda analisi, quindi, fa mantenere il primato alla combinazione Matlab-Linux

4.3.1 Memoria su Python

La funzione utilizzata per il calcolo della memoria su Python è *memory profiler*. Essa è molto potente e permette di analizzare la memoria momento per momento, estraendo un numero di valori proporzionali al tempo totale di calcolo. A causa di questo, gli array di valori estratti risultavano

essere eterogenei per numero di campioni, quindi è stato necessario ridurli o allungarli a una quantità scelta, vicino a 800 nel caso, per ottenere un grafico leggibile:

- Se meno di 400 valori, ogni valore viene replicato un numero di volte pari all'intero inferiore della divisione: $800/Numero\_valori$
- Se più di 1600 valori, viene scelta una finestra di dimensione pari all'intero inferiore della divisione : $Numero\_valori/800$. Si cerca, poi, l'indice, in cui fermare la computazione. Questo valore è l'indice più vicino alla fine per cui $indice \equiv 0 \ mod \ dim\_finestra$. Poi la finestra viene fatta scorrere, senza sovrapposizioni, fino all'indice finale e vengono mediati i valori dentro la finestra. I valori rimasti fuori vengono inseriti in fondo.

In questa maniera sono stati ottenuti array più simili per quantità di valori. La figura seguente mostra l'utilizzo di memoria in relazione al numero di campionamenti e quindi al tempo di calcolo:

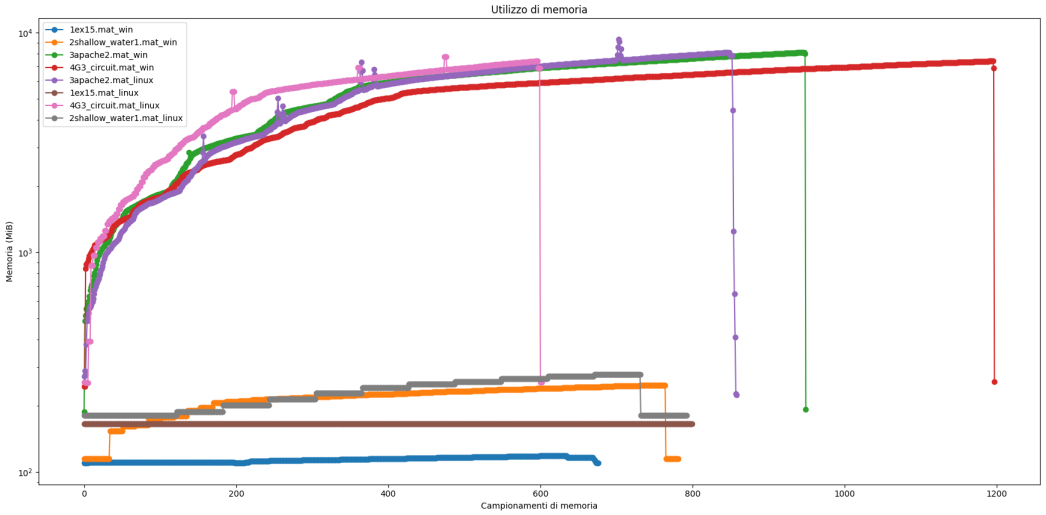

Fig. 4: Memoria Python

In Fig.4 si vede l'andamento della memoria durante tutta la computazione Python. I risultati sono gli stessi presentati nella sezione superiore ma con una lente di ingrandimento. Si può notare come, nella computazione su Linux, ci siano degli *spike* di memoria durante il calcolo della terza e quarta matrice (sul grafico viola e rosa). Questo dato risulta interessante poiché potrebbe essere il motivo della differenza di tempi tra Windows e Linux. Probabilmente, dato il fatto che Python è un linguaggio "nativo" per Linux o comunque l'installazione dello stesso è già implementata, Linux riesce a gestire meglio i calcoli con questo linguaggio di programmazione, migliorando la gestione della memoria e i tempi di calcolo.

4.4 Errore Relativo

Tra le misure scelte per l'analisi dei risultati è stato scelto l'errore relativo commesso nel calcolo della soluzione x rispetto alla soluzione aspettata xe. Infatti il termine b della equazione $Ax = b$ è stato scelto in maniera da ottenere come soluzione esatta $xe = [1111...]$. Scelto b, calcolato come $b = A * xe$, è stato calcolato il valore x. Trovata la soluzione x è stato possibile calcolare l'errore relativo tra x e xe, soluzione esatta, con la seguente formula:

$$errore\ relativo = \frac{\|x - xe\|_2}{\|xe\|_2}$$

Con $\|v\|_2$ norma euclidea del vettore. Questo valore indica quanto ci allontaniamo dalla soluzione esatta desiderata, o quanto meno ci mostra quanto sono precisi i vari solutori rispettivamente alle varie coppie linguaggio-sistema operativo. La tabella e il grafico che seguono mostrano gli errori relativi calcolati:

Dimensione	6867	81920	715176	1585478	70656	123440
Errore Relativo Matlab Win	6,19E-03	2,37E-12	4,40E-07	3,58E-08	9,00E-10	3,66E-09
Errore Relativo Matlab Linux	6,19E-07	2,37E-16	4,40E-11	3,58E-12	9,00E-14	3,66E-13
Errore Relativo Python Win	1,20E-06	2,54E-16	1,11E-11	8,75E-13		
Errore Relativo Python Linux	1,2E-06	2,54286E-16	1,11095E-11	8,74744E-13		

Tabella 3: Errore relativo in relazione alla dimensione

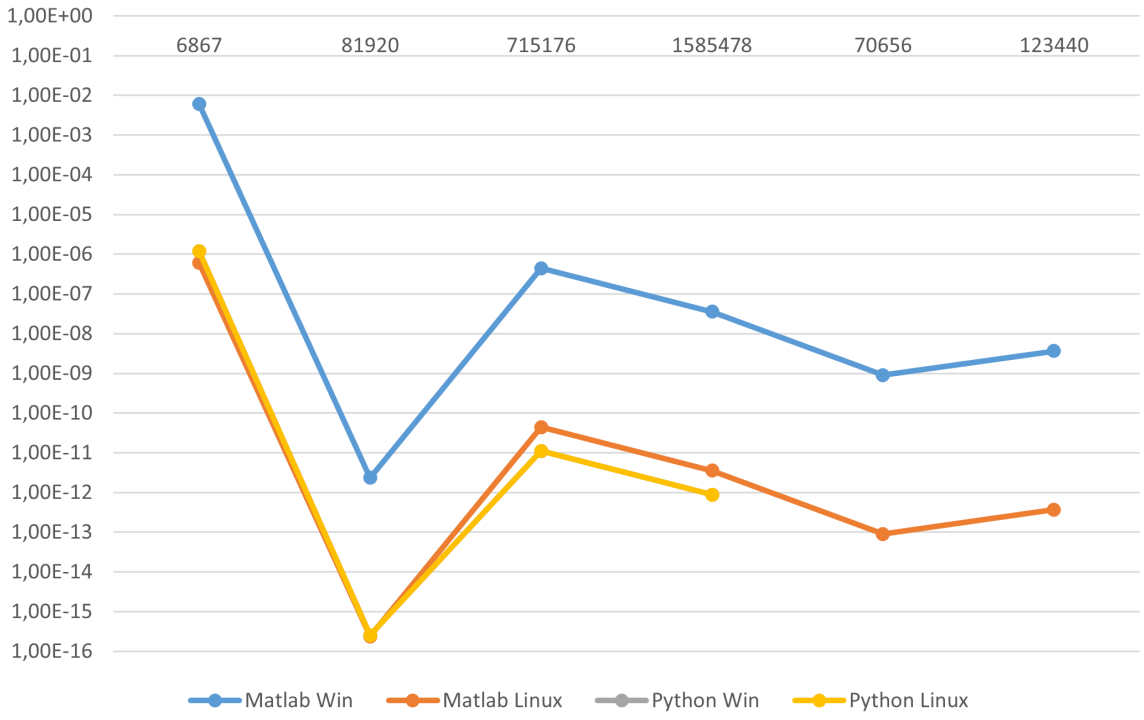

Fig. 5: Errore Relativo

Come si può vedere in figura 5 i risultati sono tutti molto simili, l'unica coppia per cui l'errore relativo è maggiore è Matlab-Windows. Le altre combinazioni producono errori al massimo dell'ordine di 10^{-6} fino a un minimo di 10^{-16} per la seconda matrice. Gli errori sono molto piccoli e poco rilevanti nella scelta finale. L'unica accortezza che possiamo evincere è che risulta preferibile computare con Matlab su Linux rispetto che su Windows. Questa analisi mantiene l'ago della bilancia sulla combinazione Matlab-Linux.

4.5 Considerazioni aggiuntive su costi, energia e consumo hardware

Spesso, quando si parla di calcolatori, è importante considerare anche l'assorbimento elettrico e il consumo dell'hardware relazionato al tempo di calcolo [1], questo perché alti tempi e manutenzione portano alti costi. Consideriamo i costi elettrici delle combinazioni Python-Linux e Matlab-Linux.

[1] Poiché memoria e errore elativo risultano essere più o meno standard per tutte e 4 le combinazioni

Linux e Python sono gratuiti, mentre Matlab costa 2150€ per una licenza perpetua o 860€ per una annuale. L'energia elettrica ad oggi viene pagata 0.234€/kWh dalle aziende. Il calcolatore preso in esame consuma 120W circa durante il calcolo. Supponiamo che una azienda debba calcolare X matrici al giorno e supponiamo che in media siano simili alla quarta matrice in tabella 1. Quindi i tempi di calcolo sono 6.83s per Matlab e 96.25s per Python. La licenza Matlab, installabile su 4 macchine, permette il calcolo contemporaneo per due computer [2], con ogni calcolatore che computa X/2 matrici. Per fare in modo che Python utilizzi lo stesso tempo totale di Matlab avremo bisogno di circa 28 calcolatori che computano X/28 matrici l'uno.
Fatto questo possiamo calcolare i costi come segue:

$$Costo\_totale = \frac{\frac{X \cdot Tempo}{3600} \cdot Consumo\_medio \cdot n\_Computer}{1000} \cdot 0.234$$

La tabella seguente riassume i costi in energia per calcolare X=2000 matrici al giorno con utilizzo totale di circa 2 ore:

	Tempo(s)	Matrici	n° pc	consumo(w)	Ore di calcolo	consumo(kWh)	costo(€)
Matlab	6,83	2000	2	240	1,897	0,455	0,107
Python	96,25	2000	28	3360	1,910	6,417	1,502

Tabella 4: Calcolo costi per 2000 matrici

Da questi costi giornalieri si può estrarre un grafico riassuntivo dell'aumento dei costi su giornate di lavoro, partendo da 0€ per Python e dal costo della licenza per Matlab come valore base. I valori vengono calcolati come: $Base + Costo\_giornaliero \cdot n\_giorni$

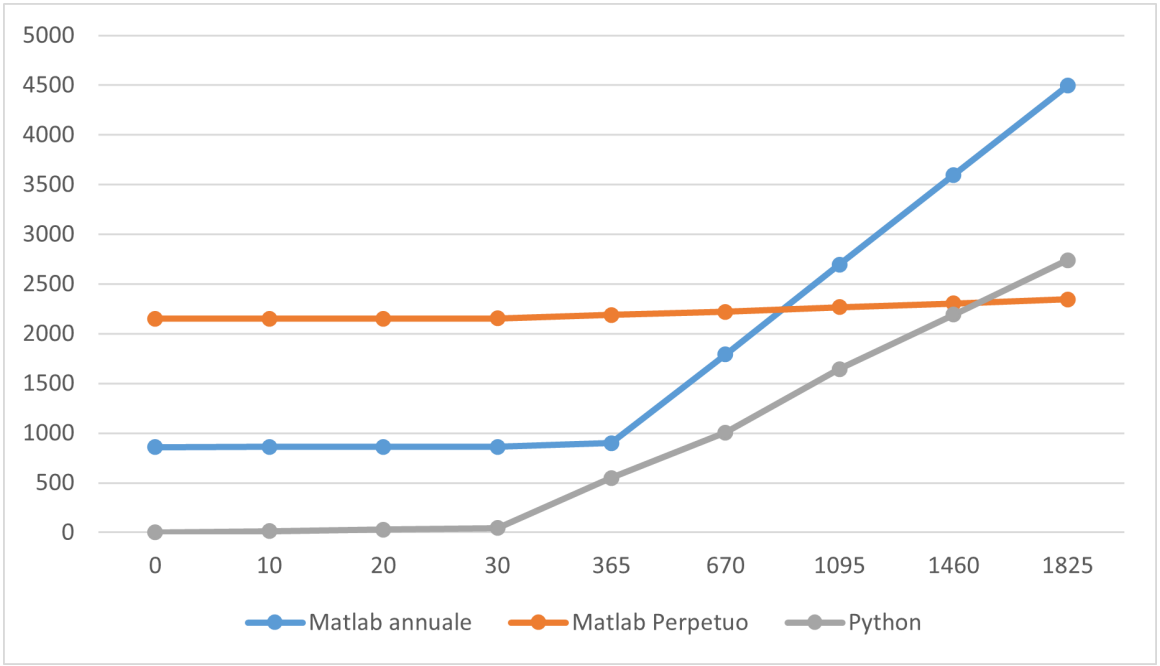

Fig. 6: Costi in base a giorni di utilizzo

Come si può notare in Fig.6, per brevi periodi di utilizzo, i costi base di Matlab superano ampliamenti i benefici al livello di tempo che esso può fornire. Infatti, come specificato, suddividendo

[2] Da license_agreements punto 1, paragrafo 1.1, sotto paragrafo (ii). Il file 'license_agreement.txt' si trova nella cartella di installazione Matlab

il lavoro di due computer con Matlab a 28 con Python si può pareggiare il costo temporale e azzerare il costo monetario. Per periodi più lunghi di tempo, ad esempio anni, la licenza annuale di Matlab risulta essere meno costosa rispetto a quella perpetua per i primi due anni, dopo di che aumenta. Il dato interessante risulta essere quello dei quattro anni: in questo punto i consumi di 28 calcolatori con Python pareggiano il costo dell'acquisto di una licenza perpetua più i consumi di due calcolatori con Matlab. Se prendessimo in esame la matrice alla colonna 3 della Tabella 1 (130s per Python e 5,5s per Matlab) il "punto di pareggio" sarebbe ai 3 anni.
Quindi, presupponendo di possedere una buona quantità di calcolatori, per periodi sotto i 4 anni, senza contare altri fattori, Python potrebbe essere una soluzione meno costosa. Importante risulta anche considerare che il calcolo di 2000 matrici giornaliere è un numero volutamente esagerato, al fine di raggiungere la tesi proposta. Inoltre, bisogna aggiungere che il calcolatore in uso consuma solo 120W, un consumo maggiore potrebbe diminuire il tempo nel quale i costi si pareggiano.
Questo risultato, in ogni caso, risulta interessante al livello teorico, però la realtà è ben diversa. L'utilizzo di più macchine di calcolo comporta anche più costi per la manutenzione e più probabilità di rottura. Inoltre, usando Matlab, potremmo usufruire dei 26 computer in meno per generare altre forme di guadagno, il che permetterebbe di coprire velocemente le spese per l'acquisto della licenza. La scelta, quindi, risulta ardua e specifica in base a vari fattori:

- Task: se le computazioni sono sporadiche meglio preferire Python considerando la perdita di un po' di tempo per computazione, se continue ed onerose meglio Matlab.
- Importanza del risultato: se il risultato risulta essere importante per continuare, per esempio, un processo produttivo, allora meglio Matlab per tempo ridotto del singolo calcolo.
- Disponibilità e costo dei calcolatori: numero di calcolatori e se questi sono di proprietà o virtual machine in affitto. Infatti se affitto una virtual machine potrò scegliere di utilizzare Matlab in quanto il costo della licenza è già compresa nella tariffa di affitto.
- Quantità di dati da elaborare: il numero di matrici da calcolare è inversamente proporzionale al tempo in cui i costi per Python raggiungono quelli di Matlab
- Disponibilità economica per acquisto e manutenzione hardware

E molti altri fattori. Per esempio una startup, con risorse limitate e che non necessita di avere i risultati in tempi brevi preferirà Python per i costi bassi. Una grande azienda, come può essere Microsoft o Apple, non avrà problemi a coprire i costi di varie licenze Matlab, ma avrà necessità di elaborare grandi quantità di dati velocemente.

4.6 Conclusioni

In conclusione si è cercato di analizzare le prestazioni di calcolo della soluzione di matrici simmetriche definite positive, su due diversi sistemi operativi, Windows e Linux, con due linguaggi di programmazione, Matlab a pagamento e Python open-source.
É stato implementato il risolutore base per ogni linguaggio, che computa $x = A/b$, con b scelto in maniera che risolvesse $A * xe = b$ con $xe = [1111...]$ soluzione esatta.
Sono stati scelte come misura per il confronto il tempo di calcolo, la memoria utilizzata e l'errore relativo commesso tra x e xe.
Ciò che è emerso mostra che a livello del sistema operativo la scelta è quasi identica per il linguaggio Matlab, se non per un maggiore errore relativo commesso su Windows, mentre si ha un vero e proprio un miglioramento computazionale per Python su Linux. Questo ha portato a scegliere Linux come migliore sistema operativo per il task esaminato.
Per quanto riguarda la scelta del linguaggio, l'analisi delle 3 misure ha dimostrato la superiorità di Matlab al livello di tempo di calcolo e un pareggio per memoria e errore relativo. Da ultima analisi sui costi, invece, si è evinto come diverse situazioni possono far cambiare la scelta, infatti essa dipende da vari fattori quali la quantità di dati da elaborare, il tempo disponibile, la disponibilità economica, quantità di calcolatori disponibili e altri. Una ridotta disponibilità economica può

portare a scegliere Python, nonostante i tempi lunghi, mentre una necessità di veloci risposte o un task urgente potrebbero far vertere su Matlab, nonostante i costi elevati.
In conclusione, per quanto riguarda il sistema operativo, Linux sembra essere una scelta migliore, sia al livello di gestione degli algoritmi, sia perché è open-source. Mentre, per la scelta del linguaggio di programmazione, la scelta dovrebbe essere ben ponderata in base ai costi di acquisto di una licenza Matlab, rispetto alla mole di dati e all'importanza del task da eseguire, considerando che Python risulta essere più lento ma gratuito.

5 Listato del codice

5.1 Matlab

```
clear;
clc;

% Loading matrices
matrices = dir("./matrices/*.mat");
% Create file
file = ["Nome", "Dimensione", "Tempo(s)", "Memoria(Kb)", "Errore relativo",
            "NNZ", "Cond"];
writematrix(file, "report.csv", 'Delimiter', 'semi');
for matrix = matrices'
    name = convertCharsToStrings(matrix.name);
    matrix = load("./matrices/" + matrix.name);
    A = matrix.Problem.A;
    dim = size(A,1);
    % Solve linear system Ax=b
    % Exact solution
    xe = ones(size(A,1), 1);
    % Compute b
    b = A * xe;
    try
        profile clear;
        profile -memory on;

        tic
        x = solveSystem(A, b);
        time = toc;
        disp(time);

        profiler = profile('info');
        fNames = {profiler.FunctionTable.FunctionName};
        fRow = find(strcmp(fNames(:), 'solver>solveSystem'));
        mem = profiler.FunctionTable(fRow).TotalMemAllocated;

        % Group info
        notZero = nnz(A);
        %cond_ = condest(A);
        err = norm(x-xe) / norm(xe);

        res = [name, dim, time, mem, err, notZero];

        writematrix(res, "report.csv", 'WriteMode', 'append', 'Delimiter', 'semi');
    catch exception
        disp(exception.message);
        res = [name dim "N/A" "N/A" "N/A" "N/A"];
    end
end
function x = solveSystem(A, b)
    x = A \ b;
end
```

5.2 Python

```
import numpy as np
import scipy.io as sio
import time
import os
import csv
from scipy.sparse.linalg import norm
from scipy import linalg
from memory_profiler import memory_usage
import pandas as pd
import matplotlib.pyplot as plt

def solveSystem(A, b):
    from scipy.sparse.linalg import spsolve
    return spsolve(A, b)

def main():
    fields = ["Nome", "Dimensione", "Tempo(s)", "Memoria(MiB)", "Err",
              "NNZ", "Cond"]
    data = []
    mem_usage_data = []
    mem_check_times = [0.1, 0.1, 1, 1]
    i = 0

    for file in os.listdir("./matrices/"):
        if file.endswith(('.mat')):
            #caricamento matrice
            _name = file
            file = sio.loadmat("./matrices/"+file)
            A = file['Problem']['A'][0, 0]

            _nnz = A.nnz
            _dim = A.shape[1]

            xe = np.ones(A.shape[1])
            b = A * xe

            _cond, _error, _time, _mem = "N/A", "N/A", "N/A", "N/A"
            try:
                #risolvo sistema
                start = time.time()
                x = solveSystem(A, b)
                # controllo tempo e memoria
                _time = time.time() - start
                mem_usage = memory_usage(proc=(solveSystem,(A,b)),
                                         interval=mem_check_times[i])
                i=i+1
                _mem = max(mem_usage) - min(mem_usage)
                _opName = _name + "_win"

                mem_usage_data.append({"Nome": _opName, "Memoria": mem_usage})
                #errore
                _error = linalg.norm(xe - x) / linalg.norm(xe)
```

```
                #aggiungo risultati
                data.append([_name, _dim, _time, _mem, _error, _nnz, _cond])
            except Exception as e:
                print("Error: {}".format(e))
                data.append([_name, _dim, "N/A", "N/A", "N/A", _nnz, "N/A"])

    with open("./report.csv", 'w') as csvfile:
        csvwriter = csv.writer(csvfile)
        csvwriter.writerow(fields)
        csvwriter.writerows(data)

    # Creo un DataFrame per memory usage
    mem_usage_df = pd.DataFrame(mem_usage_data)
    mem_usage_df.to_csv("./memory_usage.csv", index=False)

if __name__ == '__main__':
    main()
```

5.3 Creazione grafici

```
import matplotlib.pyplot as plt
import pandas as pd
import math
import numpy as np

def expand(arr):
    repeat = math.floor(800/len(arr))
    new_arr=[val for val in arr for _ in range(repeat)]
    return new_arr

def shrink(arr):
    win = math.floor(len(arr)/800)
    b = False
    i = 0
    while b==False:
        print((len(arr)-i) % win)
        if (len(arr)-i) % win == 0:
            position_rest_equal_zero = len(arr)-i
            b=True
        else:
            i = i+1
    a = np.array(arr[0:position_rest_equal_zero])
    rest = np.array(arr[position_rest_equal_zero:len(arr)])
    shrinked_arr = np.average(a.reshape(-1, win), axis=1)
    if len(rest)!=0:
        new_arr = np.concatenate((shrinked_arr, rest), axis=0)
    else:
        new_arr = shrinked_arr
    return new_arr

def main():
    #Read csv
    mem_usage_win = pd.read_csv("./windows_memory_usage.csv")
```

```
    mem_usage_linux = pd.read_csv("./linux_memory_usage.csv")
    # Creare il grafico
    fig, ax = plt.subplots()
    for i, row in mem_usage_win.iterrows():
        mem_usage = [float(x) for x in row['Memoria'][1:-1].split(',')]
        if len(mem_usage)<400:
            mem_usage = expand(mem_usage)
        if len(mem_usage)>800:
            mem_usage = shrink(mem_usage)
        ax.plot(range(len(mem_usage)), mem_usage, label=row["Nome"], marker='o')
    for i, row in mem_usage_linux.iterrows():
        mem_usage = [float(x) for x in row['Memoria'][1:-1].split(',')]
        if len(mem_usage)<400:
            mem_usage = expand(mem_usage)
        if len(mem_usage)>800:
            mem_usage = shrink(mem_usage)
        ax.plot(range(len(mem_usage)), mem_usage, label=row["Nome"], marker='o')
    ax.set_xlabel("Campionamenti di memoria")
    ax.set_ylabel("Memoria (MiB)")
    ax.set_title("Utilizzo di memoria")
    ax.set_yscale("log")
    ax.legend()
    plt.tight_layout()
    plt.show()
    plt.savefig("./memory_usage.png")


if __name__ == '__main__':
    main()
```