Università degli Studi di Milano Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

Metodi del Calcolo Scientifico – Progetto 1

Algebra Lineare Numerica

Sistemi lineari con matrici sparse simmetriche e definite positive

Autori:

Giacomo Savazzi – 845372

Raffaele Cerizza – 845512

Andrea Assirelli - 820149

Anno Accademico 2021 – 2022

Sommario

Indice delle Figure

Introduzione

Lo scopo di questo elaborato è l'analisi dell'implementazione del metodo per la fattorizzazione di Cholesky per matrici simmetriche e definite positive, cioè con tutti e soli autovalori positivi (maggiori strettamente di zero). In particolare, si sono analizzate matrici sparse, cioè con molti elementi nulli, perché il metodo di Cholesky, in generale, permette di trattare in maniera più efficiente questa classe di matrici, a differenza della fattorizzazione LU che su matrici sparse è spesso soggetta al fenomeno del ***fill-in***, fenomeno che porta alla generazione, data una matrice sparsa, di due matrici triangolari ***L*** e ***U*** che la fattorizzano, la prima inferiore e la seconda superiore, tali da richiedere l'allocazione di spazio sufficiente a rappresentare l'intera matrice di partenza. Considerando il fatto che le matrici sparse vengono spesso rappresentate in memoria in modo efficiente salvandosi solo gli elementi non nulli e le loro posizioni, se al posto di una matrice sparsa si è costretti a memorizzare due matrici triangolari ***L*** e ***U*** che abbiano tanti elementi non nulli quanti sono gli elementi dell'intera matrice iniziale, l'occupazione di spazio aumenta di molto.

Nel nostro caso abbiamo messo a confronto l'implementazione del metodo di Cholesky offerta da ***MATLAB***, software per il calcolo scientifico molto efficiente, ma anche costoso, con l'implementazione dello stesso metodo offerta dalla libreria ***spam*** per il linguaggio ***R***, e l'implementazione offerta dalla libreria ***Eigen*** per il linguaggio ***C++***.

Le tre implementazioni sono state analizzate sotto i seguenti aspetti:

- Tempo necessario in secondi, data una matrice ***A***, per calcolare la sua fattorizzazione ***R*** tramite il metodo di Cholesky, e trovare successivamente la soluzione ***x*** del sistema $\boldsymbol{A} * \boldsymbol{x} = \boldsymbol{b}$, con ***b*** termine noto calcolato in precedenza (prima di considerare il tempo di esecuzione) nel seguente modo:
$$\boldsymbol{b} = \boldsymbol{A} * \boldsymbol{x_e}$$
con $\boldsymbol{x_e}$ vettore con tutte le componenti pari ad uno;
- L'errore relativo tra la soluzione esatta $\boldsymbol{x_e}$ e quella calcolata ***x***, definito nel seguente modo:
$$\boldsymbol{errore\ rel} = \frac{\left|\left|\boldsymbol{x} - \boldsymbol{x_e}\right|\right|_2}{\left|\left|\boldsymbol{x_e}\right|\right|_2}$$
- La memoria necessaria per eseguire la fattorizzazione di Cholesky e risolvere successivamente il sistema lineare, cioè l'aumento di memoria utilizzata dal programma che deriva dalla risoluzione del sistema;

Tutti questi aspetti sono stati analizzati in merito alla risoluzione di diverse matrici sparse, fatta su due sistemi operativi diversi, Windows e Linux, in esecuzione sulla stessa macchina. Le specifiche della macchina utilizzata per i test, e i sistemi operativi utilizzati, sono dettagliati nella sezione *Ambiente utilizzato*.

Fattorizzazione di Cholesky

Nel paragrafo precedente è stato specificato che l'obiettivo della presente relazione è confrontare diverse implementazioni ed esecuzioni della fattorizzazione di Cholesky. Risulta quindi opportuno descrivere brevemente la fattorizzazione di Cholesky. A questo scopo viene dedicato questo paragrafo.

La fattorizzazione di Cholesky è una particolare tipologia di fattorizzazione di matrici che si applica solo a matrici che presentino specifiche caratteristiche. In particolare, si richiede che le matrici siano:

- simmetriche (cioè tali che $\boldsymbol{A} = \boldsymbol{A^t}$);
- definite positive (cioè tali che gli autovalori siano tutti strettamente maggiori di zero).

Come tutte le fattorizzazioni di matrici, la fattorizzazione di Cholesky decompone una matrice in un prodotto tra matrici. In particolare, se $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{n}}$ e presenta le caratteristiche sopra citate, allora la fattorizzazione di Cholesky della matrice ***A*** è data da:

$$\boldsymbol{A} = \boldsymbol{R^t R},$$

dove $\boldsymbol{R} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{n}}$ è una matrice triangolare superiore in cui gli elementi sulla diagonale principale sono positivi[1] [1].

Inoltre se la matrice ***A*** è definita positiva, allora la matrice ***R*** appena specificata è unica [2]. Il costo computazionale della fattorizzazione di Cholesky è $\boldsymbol{O(n^3)}$, dove ***n*** è il numero di righe (o colonne) della matrice ***A*** [3].

Una volta decomposta la matrice ***A*** è possibile calcolare la soluzione di un sistema lineare attraverso i seguenti passi:

1) si calcola ***y*** tramite $\boldsymbol{R^t y = b}$ (sostituzione in avanti), dove ***b*** rappresenta il vettore dei termini noti;
2) si calcola ***x*** tramite $\boldsymbol{Rx = y}$ (sostituzione all'indietro);

[1] Alcune implementazioni della fattorizzazione di Cholesky decompongono la matrice ***A*** nel prodotto di due matrici triangolari inferiori $\boldsymbol{LL^t}$**.** Il procedimento è analogo e il risultato è equivalente.

Ambiente utilizzato

Per la raccolta dei risultati è stata utilizzata una macchina con le seguenti specifiche tecniche:

- *CPU: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 2095 Mhz, 4 Core(s), 4 Logical Processor(s)*
  Physical Memory (RAM): 8.00GB
  Disk: 154GB SSD
  Architecture: x64

Le specifiche degli ambienti Windows e Linux sono le seguenti:

- *OS Name: Microsoft Windows 10 Pro*
  Version: 10.0.19044 Build 19044
- *OS Name: Ubuntu 20.04.3 LTS*
  Kernel: Linux 5.4.0-1065-kvm

Inoltre, la dimensione della memoria *Swap* per i due ambienti è la seguente:

- *Windows Swap Memory (system managed): 10GB*
- *Ubuntu Swap Memory (user managed): 10GB*

MATLAB

MATLAB è un'abbreviazione di Matrix Laboratory. Si tratta di un ambiente per il calcolo numerico e l'analisi statistica. Questo ambiente è costruito utilizzando diversi linguaggi di programmazione. Tra questi vi sono C e l'omonimo linguaggio ***MATLAB***.

Più precisamente ***MATLAB*** è un software commerciale rilasciato per la prima volta nel 1984 da MathWorks, Inc. Il suo utilizzo richiede una licenza d'uso del software. Nonostante il software non sia open source, ad oggi è utilizzato da milioni di persone in tutto il mondo.

Il software ***MATLAB*** è utilizzabile su diversi sistemi operativi. Per quanto qui interessa è utilizzabile in particolare su Microsoft Windows e su Linux. Alcune delle caratteristiche principali del linguaggio di programmazione ***MATLAB*** sono le seguenti:

- è un linguaggio multi-paradigma. In particolare supporta: la programmazione imperativa; la programmazione procedurale; la programmazione funzionale; e recentemente anche la programmazione orientata agli oggetti;
- è un linguaggio interpretato e non compilato. Questo significa che le istruzioni vengono eseguite direttamente, senza necessità di una previa compilazione in linguaggio macchina;
- è un linguaggio debolmente tipizzato. Questo significa che non è necessario descrivere il tipo di una variabile in fase di dichiarazione;

MATLAB presenta una ricca collezione di funzioni *built-in*. L'utilizzo di queste funzioni non richiede la specifica importazione di librerie esterne all'ambiente ***MATLAB***. Si descrivono ora le funzioni più importanti utilizzate ai fini del presente progetto:

- ***Load***. Questa funzione consente di caricare file nel *workspace* di ***MATLAB***. In particolare questa funzione è stata utilizzata per caricare le matrici utilizzate nel progetto. Le matrici caricate nell'ambiente ***MATLAB*** presentano l'estensione .mat. Si tratta di un formato di dati tipicamente utilizzato da ***MATLAB***. La funzione ***load*** riconosce questo formato e carica i relativi dati in apposite variabili;
- ***Mldivide*** ("****"). La funzione ***mldivide*** consente di risolvere sistemi lineari. In particolare, data una matrice ***A*** e un vettore di termini noti ***b***, la sintassi usata tipicamente per la soluzione del sistema lineare è: ***x = A\b***. Questa funzione risolve il sistema lineare utilizzando diversi metodi a seconda delle caratteristiche della matrice ***A***. Ora, le matrici utilizzate per il presente progetto sono tutte: (*i*) quadrate; (*ii*) sparse; (*iii*) simmetriche a valori reali (hermitiane); e (*iv*) definite positive. E per le matrici con queste caratteristiche la funzione ***mldivide*** utilizza il metodo di Cholesky per la risoluzione dei relativi sistemi lineari;

- ***Spparms***. Questa funzione consente di stampare a schermo alcune informazioni relative alla risoluzione di sistemi lineari per matrici sparse. Tra le informazioni rilevanti mostrate da questa funzione si menzionano:
  - La tipologia di algoritmo utilizzata per risolvere il sistema lineare. In particolare l'algoritmo utilizzato è CHOLMOD per tutte le matrici. CHOLMOD è una particolare implementazione del metodo di Cholesky sviluppata da Timothy A. Davis [4].
  - Il numero di operazioni in virgola mobile utilizzate per la risoluzione del sistema lineare con il metodo di Cholesky.
  - Il numero di valori diversi da zero nella matrice triangolare originata dalla fattorizzazione di Cholesky.
  - Il picco di memoria (in MB) utilizzato durante la risoluzione del sistema lineare.
- ***Tic*** e ***toc***. Le funzioni ***tic*** e ***toc*** consentono di misurare il tempo trascorso nell'esecuzione di istruzioni. In particolare ***tic*** registra il tempo corrente, mentre ***toc*** misura il tempo trascorso successivamente. Le istruzioni considerate devono essere comprese tra la chiamata di ***tic*** e la chiamata di ***toc***. Ai fini di questo progetto queste funzioni sono state utilizzate per calcolare il tempo necessario a risolvere i sistemi lineari tramite il metodo di Cholesky;
- ***Norm***. Questa funzione si occupa di calcolare la norma di vettori e matrici. Ai fini del presente progetto è stata utilizzata per calcolare l'errore relativo tra la soluzione calcolata e la soluzione esatta. In particolare la norma utilizzata per il calcolo dell'errore relativo è stata la norma Euclidea;
- ***Condest***. Questa funzione calcola il numero di condizionamento di matrici sparse in norma 1. ***MATLAB*** utilizza di default la funzione ***condest*** ogni volta che viene richiesto di calcolare il numero di condizionamento di una matrice sparsa. Poiché le matrici oggetto del presente progetto sono tutte sparse, si è deciso di utilizzare direttamente questa funzione per il calcolo del numero di condizionamento delle matrici;

Per il calcolo della memoria utilizzata per la risoluzione dei sistemi lineari si è proceduto come segue:

- Anzitutto si è proceduto a calcolare la memoria occupata subito dopo il caricamento della matrice. In particolare per questo calcolo è stata implementata la funzione ***monitor_memory***. Più precisamente questa funzione: (*i*) calcola il numero di bytes occupati da ogni oggetto in memoria; (*ii*) effettua la somma di tutti i bytes così calcolati; e (*iii*) divide questa somma per 1048576 in modo da ottenere la memoria occupata in MB. Per l'implementazione di questa funzione si è fatto riferimento ai suggerimenti riportati dal team di supporto di MathWorks presso il seguente link:

https://it.mathworks.com/matlabcentral/answers/97560-how-can-i-monitor-how-much-memory-matlab-is-using;

- Successivamente si è proceduto a calcolare il picco di memoria utilizzato da ***MATLAB***. Questo picco rappresenta il più alto numero di MB occupati durante la fattorizzazione di Cholesky e la risoluzione del sistema lineare. Per calcolare questo picco è stato utilizzato l'output mostrato dalla funzione ***spparms*** illustrata sopra;
- Infine si è proceduto a calcolare la differenza tra la memoria occupata successivamente alla risoluzione del sistema lineare e la memoria occupata al momento del caricamento di una matrice;

MATLAB riceve una manutenzione frequente che viene accompagnata dal rilascio di *release* semestrali. Per il presente progetto è stato utilizzato ***MATLAB R2022a*** rilasciato nel marzo 2022. Infine ***MATLAB*** presenta una documentazione facilmente accessibile, completa ed estremamente chiara. Questo aspetto verrà approfondito meglio quando si procederà al confronto delle documentazioni dei diversi linguaggi utilizzati.

C++

C++ è ad oggi uno dei linguaggi di programmazione più utilizzati al mondo, data la sua efficienza e flessibilità. Essendo uno dei linguaggi di programmazione più diffusi sono disponibili una vasta quantità di librerie, segmenti di codice, documentazioni, compilatori, debuggers etc.

Scegliendo quindi ***C++*** come linguaggio, si ha la possibilità di scrivere qualsiasi software con la possibilità inoltre di interagire con ogni singolo componente hardware. Abbiamo quindi scelto di utilizzare la libreria open source ***Eigen*** per risolvere sistemi lineari con matrici sparse definite positive applicando il metodo di Cholesky.

La libreria ***Eigen*** mette a disposizione diversi solver per applicare il metodo di Cholesky. La nostra scelta è stata di utilizzare il modulo *PardisoLLT*, un solver basato sull'integrazione aggiuntiva delle classi *Intel MKL PARDISO* che applica su matrici sparse una fattorizzazione di Cholesky $\boldsymbol{LL^t}$. L'utilizzo di questo solver permette quindi di risolvere problemi lineari del tipo $\boldsymbol{A * x = b}$ tramite una fattorizzazione $\boldsymbol{LL^t}$ di Cholesky. La fattorizzazione $\boldsymbol{LL^t}$ è applicabile se e solo se la matrice reale è definita positiva e simmetrica.

Nella documentazione di ***Eigen*** è riportato che prima di applicare la fattorizzazione il solver applica una permutazione per ridurre il *fill-in*. Come già suggerito dal nome queste classi sono utilizzabili solamente su CPU Intel che permettono di utilizzare tutte le ottimizzazioni presenti. L'utilizzo di questo solver ci ha permesso di sfruttare tutte le potenzialità sia software offerte dal linguaggio di programmazione, che hardware.

La lettura delle matrici è stata effettuata tramite il metodo ***loadMarket***, fornito dalla classe SparseExtra, che permette di caricare in memoria una matrice sparsa in formato mtx (Matrix Market Exchange Formats) e salvarla come oggetto SparseMatrix<double>.

Eigen gestisce le matrici attraverso un formato "compressed", formato da 2 array di interi contenenti gli indici della matrice, e da un ulteriore array di double contenente i valori non nulli.

In qualsiasi momento ***Eigen*** permette di invocare la compressione su una matrice chiamando il metodo ***makeCompressed***.

Tutti i solver della libreria seguono lo stesso procedimento per l'applicazione del metodo di Cholesky su una matrice caricata in memoria. Viene innanzitutto definito il solver, su cui è invocato il metodo ***compute***, il quale in input richiede di fornire la matrice sparsa precedentemente caricata.

Il metodo ***compute*** invoca a sua volta due metodi, ***analyzePattern*** e ***Factorize***. ***AnalyzePattern*** si occupa di effettuare una decomposizione simbolica della matrice fornita che il metodo ***Factorize*** poi applicherà sulla matrice. Come suggerito nella documentazione di ***Eigen*** eseguire il

metodo ***compute*** è equivalente ad applicare il metodo ***analyzePattern*** prima e ***Factorize*** successivamente sulla matrice.

In particolare l'obiettivo del metodo ***AnalyzePattern*** è quello di riordinare gli elementi non nulli della matrice per far sì che lo step di fattorizzazione crei un minore *fill-in*. Chiamando il metodo ***solve*** è quindi possibile far computare al solver la soluzione del sistema lineare tramite decomposizione di Cholesky. La libreria ***Eigen*** non si occupa di controllare se effettivamente la matrice passata al solver rispetta le caratteristiche richieste, per esempio che la matrice sia definita positiva e simmetrica, ma lascia il compito allo sviluppatore.

Eigen e Intel MKL sono in continuo sviluppo, tant'è che le versioni utilizzate per questa analisi sono state recentemente aggiornate nel 2021. Nonostante la documentazione presente sia sufficiente per comprendere al meglio l'utilizzo della libreria, e le prestazioni fornite siano ottime, rimane comunque secondo noi una marcata difficoltà per l'installazione e utilizzo per chi non è solito all'impiego di ***C++*** come linguaggio di programmazione, in confronto sia a ***MATLAB*** che ***R***.

Linguaggio R

R è un linguaggio di programmazione e ambiente di sviluppo open source utilizzato principalmente per eseguire analisi statistica sui dati.

Sfruttando il package, sempre open source, ***spam***, siamo stati in grado di analizzare matrici sparse, e risolvere sistemi lineari che le riguardano in maniera semplice ed efficiente. In particolare, vista la dimensione molto elevata di alcune delle matrici da analizzare, è stato necessario fare uso di una sua estensione, chiamata ***spam64***, che permette la gestione veloce e scalabile di matrici sparse con anche più di $2^{31} - 1$ elementi non nulli.

La lettura delle matrici è stata fatta con la funzione ***read.MM***, sempre del package ***spam***, che permette il salvataggio della matrice in formato sparso, cioè memorizza solo il valore degli elementi non nulli e la loro posizione, senza sprecare tempo e memoria nel salvataggio degli elementi nulli. Abbiamo, infatti, notato che la gestione delle matrici con un formato classico, salvandoci tutti gli elementi, non era possibile per motivi di tempo e di spazio occupato.

Per quanto riguarda l'applicazione del metodo di fattorizzazione di Cholesky, si è fatto riferimento alla funzione ***chol.spam***: questa funzione, data in input una matrice simmetrica e definita positiva, ne calcola la matrice che la fattorizza secondo Cholesky applicando l'algoritmo di ***Ng and Peyton*** del 1993. È anche possibile specificare una strategia di pivoting, da seguire per la permutazione delle righe della matrice; le due strategie di pivoting supportate da questa funzione sono:

- ***Multiple Minimum Degree (MMD)***: algoritmo per la permutazione delle righe di una matrice, applicato prima della fattorizzazione di Cholesky con lo scopo di ridurre il numero di non-zeri nel risultato. Questo permette l'utilizzo di meno memoria per il salvataggio del risultato, e meno operazioni aritmetiche per la fattorizzazione;
- ***Reverse Cuthill-Mckee (RCM)***: algoritmo che permette di permutare una matrice sparsa in una matrice a bande, più semplice da risolvere.

È anche possibile richiedere all'algoritmo di fattorizzazione di non applicare alcuna strategia di pivoting. Facendo tre diverse prove con ogni matrice, una di fattorizzazione senza pivoting, una di fattorizzazione con strategia MMD, e una di fattorizzazione con strategia RCM, si è notato che la strategia MMD permette di ottenere, nel nostro caso, matrici di fattorizzazione più compatte, impiegando anche meno tempo. Per questo motivo è stata considerata solo questa strategia, e tutti i risultati riportati riguardano essa.

La funzione si occupa, inoltre, di verificare se la matrice passata è effettivamente simmetrica e definita positiva: se una di queste due condizioni non è verificata, la funzione ritorna una eccezione, gestibile tramite blocco ***try-catch***. Nel caso si volesse evitare il controllo sulla

simmetria, per guadagnare in efficienza, esiste l'opzione ***spam.cholsymmetrycheck*** che, se posta a FALSE, evita questo controllo.

La libreria offre anche la funzione ***solve.spam*** che, data in input la fattorizzazione della matrice di partenza, prodotta da ***chol.spam***, combinando in maniera opportuna le funzioni ***backsolve***, per la risoluzione di matrici triangolari inferiori, e ***forwardsolve***, per la risoluzione di matrici triangolari superiori, permette di ottenere la soluzione ***x*** del sistema lineare iniziale.

Per quanto riguarda la documentazione e manutenzione delle librerie ***spam*** e ***spam64***, come possiamo vedere dai siti ufficiali[2], abbiamo visto essere molto ben documentate e mantenute. Si consideri, infatti, che la loro ultima release risale a non molti mesi fa. Dai siti ufficiali è stato anche possibile scaricare la documentazione ufficiale, molto accurata, in formato pdf.

Per quanto riguarda la facilità di utilizzo di questo linguaggio e della libreria, a nostro parere ***R*** risulta di più difficile interpretazione e utilizzo rispetto a ***MATLAB*** per persone non avvezze alla programmazione ma, nel caso si siano già utilizzati linguaggi di programmazione a più basso livello come ***C++***, la situazione si semplifica. Unico problema, però, è che ***R*** presenta una sintassi molto specifica per certe operazioni, e non è così semplice traslare la conoscenza sintattica di altri linguaggi su di esso.

[2] CRAN - Package spam (r-project.org), CRAN - Package spam64 (r-project.org)

Risultati ottenuti

I dati raccolti e messi a confronto per valutare la migliore implementazione rispetto alle tre librerie e linguaggi analizzati sono:

- ***Numero di non zeri***: per ogni matrice analizzata è stato calcolato, tramite opportune funzioni di libreria, il numero di valori non nulli che questa contiene. Questo dato non è stato considerato per il confronto tra le tre librerie, ma come valore da porre sull'asse delle ascisse nei grafici;
- ***Memoria occupata***: la memoria occupata per la risoluzione del sistema lineare è stata calcolata in ambiente ***R*** e ***C++*** come dimensione della matrice $\boldsymbol{R}$ prodotta dalla fattorizzazione di Cholesky, più la dimensione del vettore ottenuto come risultato del sistema lineare risolto tramite $\boldsymbol{R}$: questa ci è sembrata essere una buona approssimazione della memoria effettivamente utilizzata, che comprende anche variabili temporanee di poco conto. Per quanto riguarda ***MATLAB*** invece, visto che la risoluzione del sistema lineare non avviene in due fasi (produzione della matrice di Cholesky e risoluzione con essa) come in ***R*** e ***C++,*** ma direttamente tramite operatore ***backslash /***, non si è potuta analizzare la dimensione della matrice prodotta dalla fattorizzazione, ma si è potuto vedere il picco di memoria utilizzata durante la risoluzione del sistema, picco che sicuramente comprende la dimensione della matrice di fattorizzazione e la dimensione del vettore risultato. Facendo una differenza tra la memoria prima dell'esecuzione, e il picco di memoria visto durante l'esecuzione, si è ottenuta una buona stima della memoria utilizzata, stima confrontabile con quelle ottenute per i linguaggi ***R*** e ***C++***;
- ***Tempo impiegato***: per i linguaggi ***R*** e ***C++***, il tempo impiegato è stato calcolato come somma tra il tempo necessario per la fattorizzazione, più il tempo necessario per la risoluzione del sistema lineare con la matrice ottenuta dalla fattorizzazione. In ***MATLAB***, il tempo impiegato è stato calcolato come tempo necessario per l'esecuzione del comando ***/*** sulla matrice considerata;
- ***Errore relativo***: per tutti e tre i linguaggi, l'errore relativo è stato calcolato facendo riferimento alla soluzione ***x*** ottenuta, e a quella esatta $\boldsymbol{x_e}$, in questo modo:

$$\boldsymbol{errore\ rel} = \frac{\left|\left|x - x_e\right|\right|_2}{\left|\left|x_e\right|\right|_2}$$

Per ogni linguaggio e sistema operativo considerato sono stati prodotti tre grafici: uno riguardante la memoria occupata, uno riguardante il tempo impiegato, e uno riguardante l'errore relativo. Questi grafici presentano sull'asse delle ascisse il numero di non zeri della matrice

considerata, e sull'asse delle ordinate la dimensione analizzata. Si noti che la scala dell'asse delle ordinate è logaritmica in base 10.

MATLAB

Cominciamo con la presentazione dei dati raccolti su ***MATLAB***. A questo proposito si segnala che utilizzando ***MATLAB*** non è stato possibile eseguire la fattorizzazione di Cholesky sulle matrici ***Flan_1565*** e ***StocF-1465***. Il motivo risiede nell'eccessivo uso di memoria RAM da parte dell'applicazione. In questo caso Linux ha proceduto all'uccisione dell'applicazione per evitare il crash del sistema; Windows invece ha semplicemente registrato un blocco del sistema.

I risultati raccolti sono i seguenti:

Windows	Numero Non Zeri	Numero Condizionamento	Memoria Iniziale (MB)	Picco di memoria (MB)	Memoria Utilizzata (MB)	Tempo Impiegato (s)	Errore Relativo
ex15	98671	1,4326E+13	3,1244	4,9000	1,7756	0,05	5,8617E-07
shallow_water1	327680	3,6280E+00	11,2601	48,1000	36,8399	0,28	2,3391E-16
cfd1	1825580	1,3351E+06	57,3805	406,8000	349,4195	1,37	9,2166E-14
cfd2	3085406	3,7285E+06	97,0314	788,5000	691,4686	2,65	3,5110E-13
parabolic_fem	3674625	2,1108E+05	124,1846	495,3000	371,1154	2,65	1,0376E-12
apache2	4817870	5,3169E+06	157,9513	1921,3000	1763,3487	7,89	4,0670E-11
G3_circuit	7660826	2,2384E+07	257,9913	2302,7000	2044,7087	12,12	3,2661E-12
StocF-1465	21005389	//	663,3975	//	//	//	//
Flan_1565	114165372	//	3,57E+03	//	//	//	//

Figura 1 - Risultati ottenuti da MATLAB su Windows

Linux	Numero Non Zeri	Numero Condizionamento	Memoria Iniziale (MB)	Picco di memoria (MB)	Memoria Utilizzata (MB)	Tempo Impiegato (s)	Errore Relativo
ex15	98671	1,4326E+13	3,1224	4,9000	1,7776	0,02	5,8617E-07
shallow_water1	327680	3,6280E+00	11,2585	48,1000	36,8415	0,25	2,3391E-16
cfd1	1825580	1,3351E+06	57,3789	406,8000	349,4211	1,34	9,2166E-14
cfd2	3085406	3,7285E+06	97,0298	788,5000	691,4702	2,85	3,5110E-13
parabolic_fem	3674625	2,1108E+05	124,1830	495,3000	371,1170	2,46	1,0376E-12
apache2	4817870	5,3169E+06	157,9459	1921,3000	1763,3541	9,27	4,0670E-11
G3_circuit	7660826	2,2384E+07	257,9858	2302,7000	2044,7142	14,47	3,2661E-12
StocF-1465	21005389	//	663,3969	//	//	//	//
Flan_1565	114165372	//	3,57E+03	//	//	//	//

Figura 2 - Risultati ottenuti da MATLAB su Linux

Procediamo con l'analisi dei risultati. Cominciamo con la memoria utilizzata. Il confronto fra i risultati ottenuti con ***MATLAB*** su Windows e su Linux è riportato nella Figura 3:

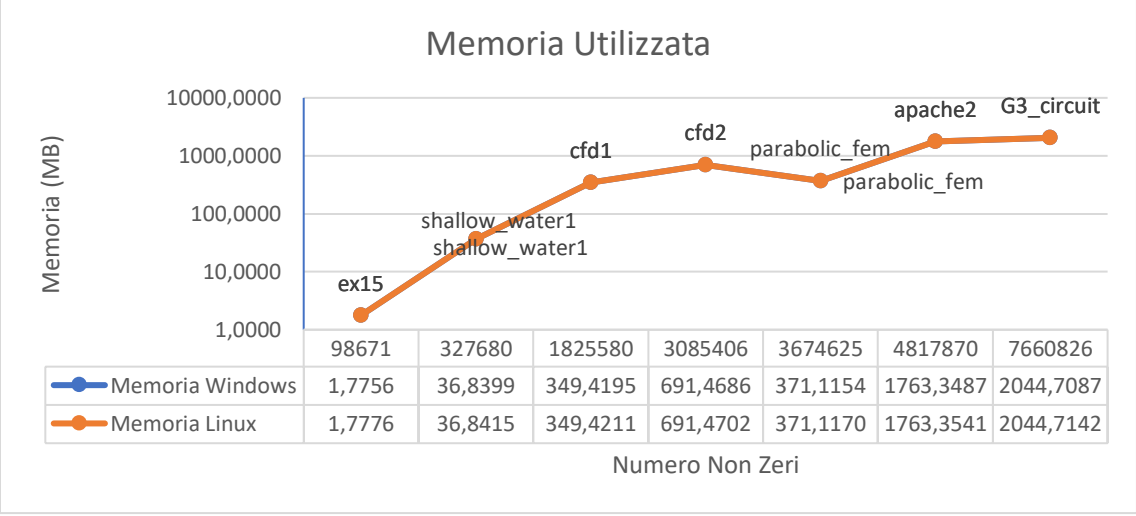


Figura 3 - Grafico memoria occupata da MATLAB

Da questo grafico si possono trarre le seguenti considerazioni:

- Anzitutto si può notare come non vi siano grosse differenze fra Windows e Linux. Linux sembra impiegare leggermente più memoria rispetto a Windows. Ma questa differenza è minima, trascurabile e sicuramente influenzata dall'arrotondamento dei risultati effettuato dalla funzione ***spparms***.
- La matrice che occupa meno memoria in valore assoluto è la più piccola, ovvero ***ex15***. La matrice che occupa più memoria in valore assoluto è la più grande, ovvero ***G3_circuit***. Questa banale osservazione sembra suggerire che l'aumento di memoria sia proporzionale alla dimensione della matrice di partenza. Tuttavia questa conclusione è facilmente smentita dai dati riportati in Figura 3. Pertanto occorre procedere a considerazioni più approfondite.
- L'aumento della memoria impiegata non è sempre proporzionale al numero di non zeri delle matrici originali. Questo è dovuto al fatto che il numero di non zeri della matrice ***R*** ottenuta tramite la fattorizzazione di Cholesky non è direttamente proporzionale al numero di non zeri della matrice di partenza ***A***. Quando il numero di zeri della matrice ***A*** che vengono convertiti in numeri diversi da zero nella matrice ***R*** è alto, si afferma che la fattorizzazione ha prodotto un alto ***fill-in*** [5].
- L'aumento della memoria impiegata è invece certamente influenzato in modo diretto dal ***fill-in*** generato dalla fattorizzazione delle matrici. A questo proposito viene mostrato in Figura 4 un prospetto che suggerisce quali siano le matrici che generano maggiore e minore ***fill-in***:

Matrice	**Numero iniziale Non Zeri**	**Numero Non Zeri fattorizzazione**	**Rapporto**
ex15	98671	2,27E+05	2,30E+00
shallow_water1	327680	2,36E+06	7,19E+00
cfd1	1825580	3,59E+07	1,97E+01
cfd2	3085406	7,38E+07	2,39E+01
parabolic_fem	3674625	3,61E+07	9,83E+00
apache2	4817870	1,77E+08	3,68E+01
G3_circuit	7660826	1,89E+08	2,47E+01
StocF-1465	21005389	//	//
Flan_1565	114165372	//	//

Figura 4 - Prospetto non zeri prima e dopo la fattorizzazione su MATLAB

In questo prospetto vengono indicati per ogni matrice: (*i*) il numero di non zeri della matrice ***A***; (*ii*) il numero di non zeri della matrice ***R*** ottenuta tramite la fattorizzazione di Cholesky (e calcolato tramite la funzione ***spparms***); e (*iii*) il rapporto tra i due numeri

precedenti. Questo prospetto suggerisce quali siano le matrici che generano più ***fill-in*** e quelle che ne generano meno. In particolare:

- le matrici che generano meno ***fill-in*** risultano ***ex15***, ***shallow_water1*** e ***parabolic_fem***;
- le matrici che generano più ***fill-in*** risultano ***apache2***, ***G3_circuit*** e ***cfd2***.

Queste considerazioni sul ***fill-in*** aiutano a comprendere perché la memoria impiegata per ***parabolic_fem*** sia minore rispetto a quella impiegata per ***cfd2***, nonostante la prima matrice sia più grande della seconda. E parimenti aiutano a comprendere perché la memoria impiegata per ***G3_circuit*** sia poco maggiore rispetto a quella impiegata per ***apache2***, nonostante la prima matrice sia molto più grande della seconda.

Passiamo ora all'analisi dei risultati relativi al tempo impiegato per la decomposizione delle matrici e per la risoluzione dei sistemi lineari. Il confronto fra i risultati ottenuti con ***MATLAB*** su Windows e su Linux è riportato nella Figura 5:

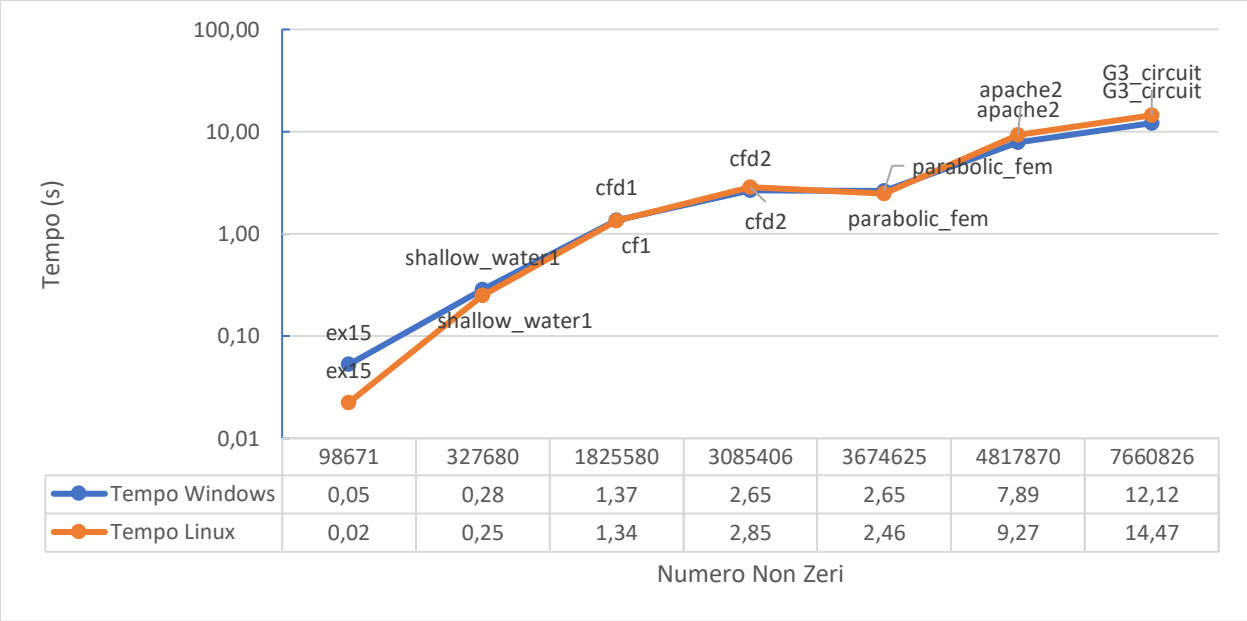


Figura 5 - Grafico tempo impiegato da MATLAB

Da questo grafico si possono trarre le seguenti considerazioni:

- Anzitutto si può notare come i tempi di esecuzione su Linux siano leggermente inferiori rispetto a quelli registrati su Windows per le matrici più piccole; mentre siano leggermente superiori per le matrici più grandi. Queste differenze sono da imputare al diverso modo con cui i sistemi operativi gestiscono l'allocazione e la liberazione della memoria utilizzata. Inoltre si può notare come il tempo di esecuzione registrato con ***parabolic_fem*** risulti inferiore a quello registrato con ***cfd2***, nonostante quest'ultima matrice sia più piccola (e con meno valori non nulli) rispetto alla prima. Questo risultato

è giustificato dal numero di operazioni *floating point* (in seguito: *flops*) eseguite per la fattorizzazione di Cholesky. In particolare l'output di ***spparms*** (sia su Windows che su Linux) riporta che:

- per la fattorizzazione di ***cfd2*** vengono eseguite 1.2671e+11 *flops*;
- per la fattorizzazione di ***parabolic_fem*** vengono eseguite 1.7111e+10 *flops*.

In questo quadro poiché la fattorizzazione di ***parabolic_fem*** richiede meno operazioni, il tempo complessivo di esecuzione risulta inferiore. Naturalmente anche in questo caso il minor numero di *flops* eseguite per ***parabolic_fem*** è correlato al basso ***fill-in*** prodotto dalla fattorizzazione.

Infine concludiamo con l'analisi dei risultati riguardanti gli errori relativi delle soluzioni calcolate. Il confronto fra i risultati ottenuti con ***MATLAB*** su Windows e su Linux è riportato nella Figura 6:

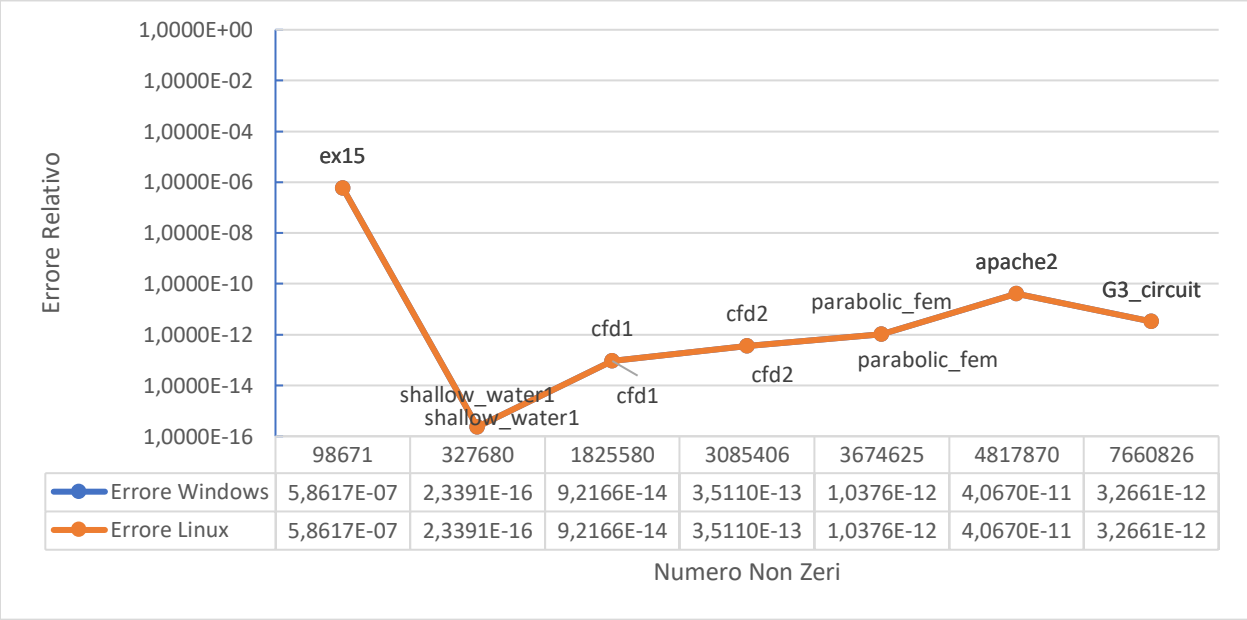


Figura 6 - Grafico errore relativo ottenuto da MATLAB

Da questo grafico si possono trarre le seguenti considerazioni:

- Anzitutto anche per l'errore relativo non vi sono state grandi differenze fra i risultati ottenuti su Windows e quelli ottenuti su Linux.
- Inoltre risulta evidente come l'errore maggiore si registri nella risoluzione del sistema lineare relativo alla matrice ***ex15***. Questo risultato non sorprende. Infatti la matrice ***ex15*** è la matrice con il numero di condizionamento più alto tra le matrici oggetto di questa relazione. E il numero di condizionamento della matrice influisce direttamente sull'errore relativo della soluzione del sistema lineare. In particolare:
  - L'errore relativo è definito come: $\frac{||x_e - x||}{||x_e||}$

dove x_e è la soluzione esatta e x è la soluzione approssimata.

- Se la matrice ***A*** è invertibile, allora:

$$Ax_e = b_e \equiv x_e = A^{-1}b_e$$
$$Ax = b \equiv x = A^{-1}b$$

dove b_e è il vettore dei termini noti ottenuto a partire dalla soluzione esatta x_e, e b è il vettore dei termini noti ottenuto a partire dalla soluzione approssimata x.

- L'errore relativo può allora essere riscritto come:

$$\frac{||x_e - x||}{||x_e||} \equiv \frac{||A^{-1}b_e - A^{-1}b||}{||A^{-1}b_e||}$$

- Infine vale la seguente disequazione:

$$\frac{||A||}{||A||}\frac{||A^{-1}|||b_e - b||}{||A^{-1}b_e||} \leq \big|||A|||A^{-1}|\big| \frac{||b_e - b||}{||b_e||}$$

dove $\big|||A|||A^{-1}|\big|$ è proprio il numero di condizionamento della matrice ***A***.

- In questo quadro risulta evidente che quanto più grande è il numero di condizionamento della matrice ***A*** tanto più grande è l'errore relativo risultante dal calcolo della soluzione del sistema lineare.

- Ancora, si può notare come l'errore minore si registri nella risoluzione del sistema lineare relativo alla matrice ***shallow_water1***. Anche questo risultato non sorprende. Infatti la matrice ***shallow_water1*** è la matrice con il numero di condizionamento più basso tra le matrici oggetto di questa relazione.
- Infine si può osservare come l'errore relativo registrato con la matrice ***G3_circuit*** risulti inferiore all'errore relativo registrato con la matrice ***apache2***, sebbene la prima matrice (*i*) sia più grande, (*ii*) abbia un maggior numero di non zeri e (*iii*) abbia un numero di condizionamento maggiore rispetto alla seconda. Il motivo è da ricercarsi nell'alto ***fill-in*** prodotto dalla fattorizzazione della matrice ***apache2***. Infatti se il ***fill-in*** è alto, allora vi saranno molti valori nulli della matrice ***A*** che diventano non nulli nella matrice ***R***. E questa differenza non può che riflettersi sull'accuratezza del calcolo in virgola mobile della soluzione del sistema lineare.

In conclusione: l'utilizzo di ***MATLAB*** su Windows e su Linux non presenta grandi differenze. Qualche leggera differenza si può notare in tema di tempi di esecuzione. In questo caso si può suggerire di utilizzare Linux nel caso di matrici di piccole dimensioni, e Windows nel caso di matrici di grandi dimensioni.

C++

In questa sezione sono riportati i risultati ottenuti dall'analisi delle matrici utilizzando ***C++*** sui due ambienti, Windows e Linux. I risultati sono riportati nelle tabelle seguenti:

Windows	Numero Non Zeri	Numero Condizionamento	Memoria Iniziale (MB)	Memoria Utilizzata (MB)	Tempo Impiegato (s)	Errore Relativo
ex15	98671	1,4326E+13	1,1914	12,9141	0,17	4,6243E-07
shallow_water1	327680	3,6280E+00	3,3008	61,1211	0,67	1,0505E-16
cfd1	1825580	1,3351E+06	20,6680	258,1090	1,79	1,3353E-14
cfd2	3085406	3,7285E+06	34,8789	454,9380	2,68	2,4273E-13
parabolic_fem	3674625	2,1108E+05	44,0742	486,8710	4,30	2,4493E-12
apache2	4817870	5,3169E+06	57,8750	1494,2700	10,25	6,1314E-13
G3_circuit	7660826	2,2384E+07	93,7344	1653,3000	12,11	9,9575E-14
StocF-1465	21005389	//	//	//	//	//
Flan_1565	114165372	//	//	//	//	//

Figura 7 - Risultati ottenuti da C++ su Windows

Linux	Numero Non Zeri	Numero Condizionamento	Memoria Iniziale (MB)	Memoria Utilizzata (MB)	Tempo Impiegato (s)	Errore Relativo
ex15	98671	1,4326E+13	1	22	0,15	5,8281E-07
shallow_water1	327680	3,6280E+00	2	69	0,68	1,0454E-16
cfd1	1825580	1,3351E+06	14	289	2,45	4,7909E-14
cfd2	3085406	3,7285E+06	24	491	4,24	2,6570E-13
parabolic_fem	3674625	2,1108E+05	42	498	5,76	2,5777E-12
apache2	4817870	5,3169E+06	55	1509	19,57	1,0652E-12
G3_circuit	7660826	2,2384E+07	88	1595	20,04	1,0835E-13
StocF-1465	21005389	//	//	//	//	//
Flan_1565	114165372	//	//	//	//	//

Figura 8 - Risultati ottenuti da C++ su Linux

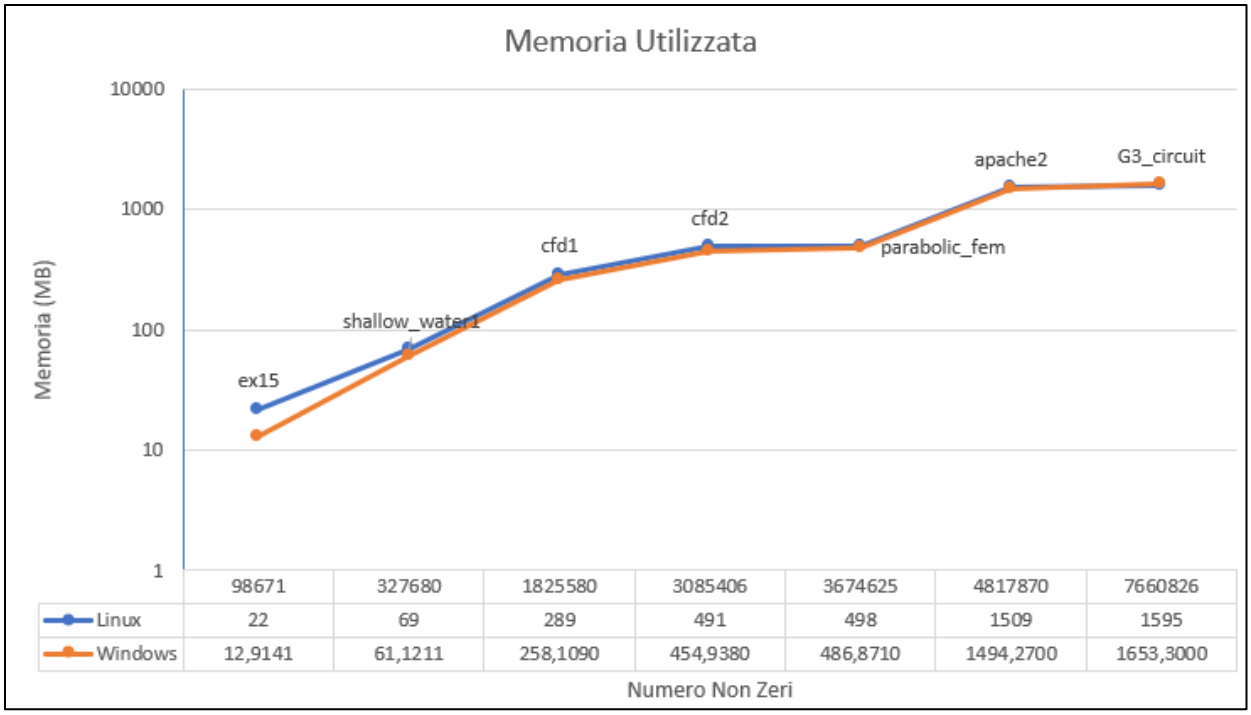


Figura 9 - Grafico memoria utilizzata da C++

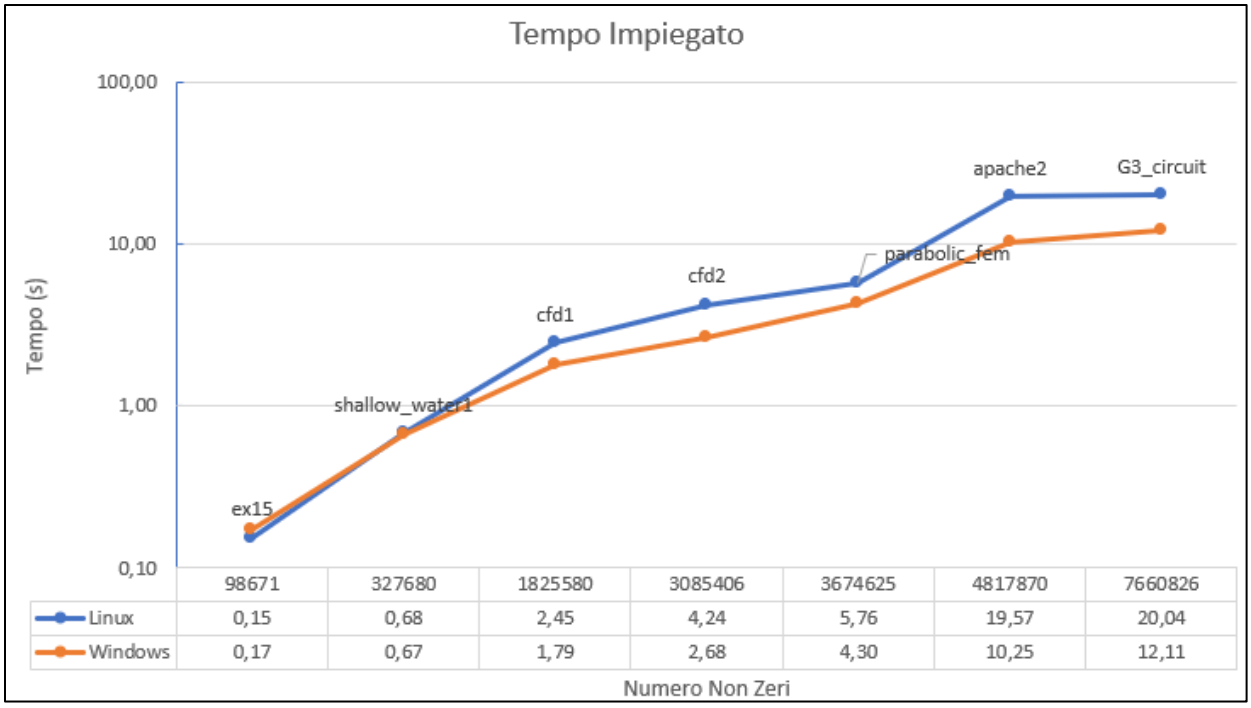


Figura 10 - Grafico del tempo impiegato da C++

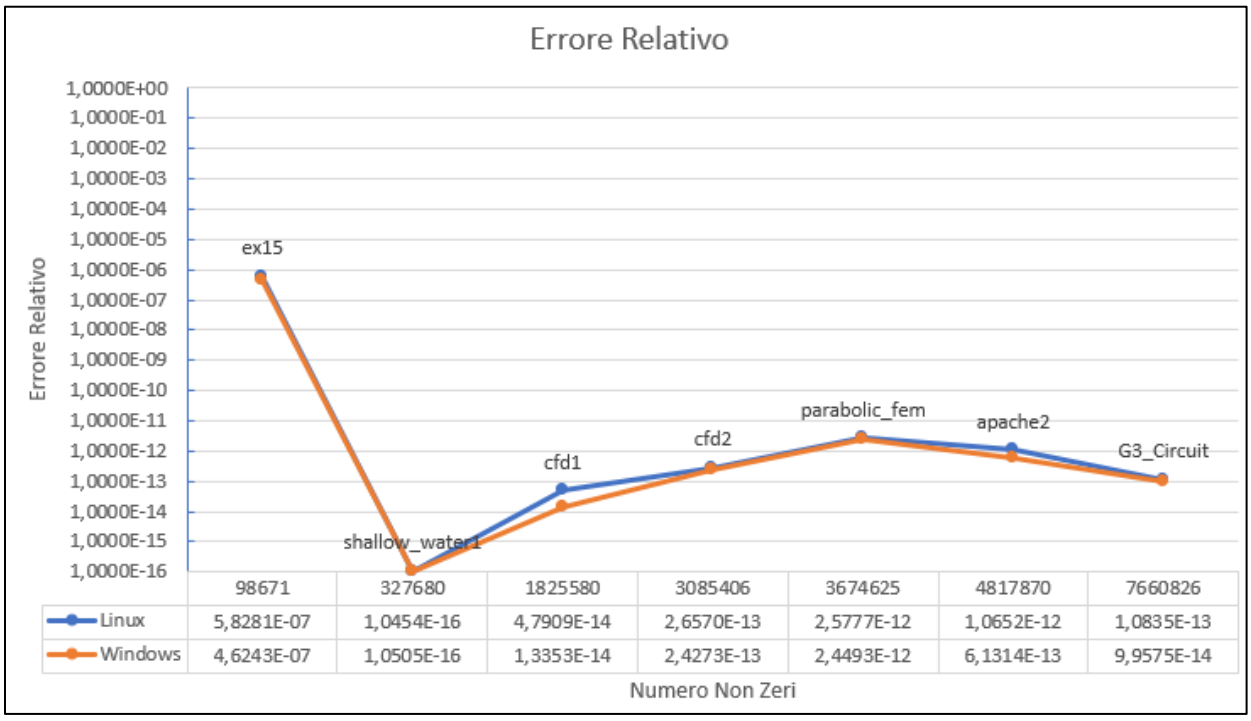


Figura 11 - Grafico errore relativo con C++

Osservando i grafici generati utilizzando le tabelle proposte precedentemente possiamo quindi notare che per i due ambienti software su cui è stata svolta l'analisi, l'utilizzo della memoria rimane pressoché invariato. Windows sembra gestire meglio la matrice ***ex15*** anche se la differenza non è così marcata rispetto ai risultati ottenuti su Linux.

In merito ai tempi necessari per eseguire l'analisi notiamo che per la matrice ***ex15*** il tempo è simile per entrambi i due ambienti, ma non si può dire lo stesso per le altre matrici di dimensioni maggiori dove Windows ha registrato tempi minori, in particolare su ***cfd2*** e ***apache2***, dovuti alle maggiori ottimizzazioni presenti compilando l'eseguibile con il compilatore *MSVC (Microsoft Visual C++).*

Per quanto riguarda i risultati ottenuti dal calcolo dell'errore relativo, i risultati sono coerenti con i rispettivi numeri di condizionamento delle matrici. Infatti ***ex15*** ha l'errore relativo più alto, ***shallow_water1*** quello più basso. Notiamo inoltre che Windows ha riportato valori più bassi rispetto a Linux per ***cfd1*** e ***apache2***.

Nel confronto tra Windows e Linux va inoltre considerata la semplicità dell'installazione delle componenti necessarie, che nel caso di Windows è risultata più chiara e immediata.

Nel complesso memoria e tempo necessario su entrambi gli ambienti seguono un andamento pressoché lineare, presentando solamente una leggera flessione tra ***cfd2*** e ***parabolic_fem***.

Considerando solamente i dati ottenuti risulta evidente, nonostante non sia così marcata la differenza, che l'ambiente Windows riesce ad analizzare meglio le matrici fornite rispetto a Linux, per questo motivo preferiamo consigliare l'utilizzo del primo rispetto al secondo sia in termini prettamente prestazionali che di usabilità.

Linguaggio R

I risultati ottenuti dalla risoluzione dei sistemi lineari riguardanti le matrici analizzate in ambiente ***R*** in esecuzione su ***Windows*** e ***Linux*** sono i seguenti:

Windows	Numero Non Zeri	Numero Condizionamento	Memoria Iniziale (MB)	Memoria Utilizzata (MB)	Tempo Impiegato (s)	Errore Relativo
ex15	98671	1,4326E+13	1,2127	2,1832	0,04	7,2990E-07
shallow_water1	327680	3,6280E+00	4,2610	23,7433	2,17	2,7910E-16
cfd1	1825580	1,3351E+06	22,1907	265,1346	11,36	1,3030E-13
cfd2	3085406	3,7285E+06	37,5198	522,1456	32,12	1,2780E-12
parabolic_fem	3674625	2,1108E+05	46,2000	310,7171	11,38	2,3080E-12
apache2	4817870	5,3169E+06	60,6763	1593,3137	155,10	1,7220E-11
G3_circuit	7660826	2,2384E+07	98,2730	1677,3945	140,50	1,2910E-12
StocF-1465	21005389	//	257,9264	//	//	//
Flan_1565	114165372	//	1376,2448	//	//	//

Figura 12 - Risultati ottenuta da R su Windows

Linux	Numero Non Zeri	Numero Condizionamento	Memoria Iniziale (MB)	Memoria Utilizzata (MB)	Tempo Impiegato (s)	Errore Relativo
ex15	98671	1,4326E+13	1,2127	2,1842	0,03	7,2980E-07
shallow_water1	327680	3,6280E+00	4,2610	23,7532	1,64	2,7890E-16
cfd1	1825580	1,3351E+06	22,1907	266,1446	8,20	1,3010E-13
cfd2	3085406	3,7285E+06	37,5198	522,1336	23,35	1,2730E-12
parabolic_fem	3674625	2,1108E+05	46,2000	311,7271	7,77	2,3030E-12
apache2	4817870	5,3169E+06	60,6763	1594,3157	115,60	1,7200E-11
G3_circuit	7660826	2,2384E+07	98,2730	1678,3952	100,00	1,2890E-12
StocF-1465	21005389	//	257,9264	//	//	//
Flan_1565	114165372	//	1376,2448	//	//	//

Figura 13 – Risultati ottenuti da R su Linux

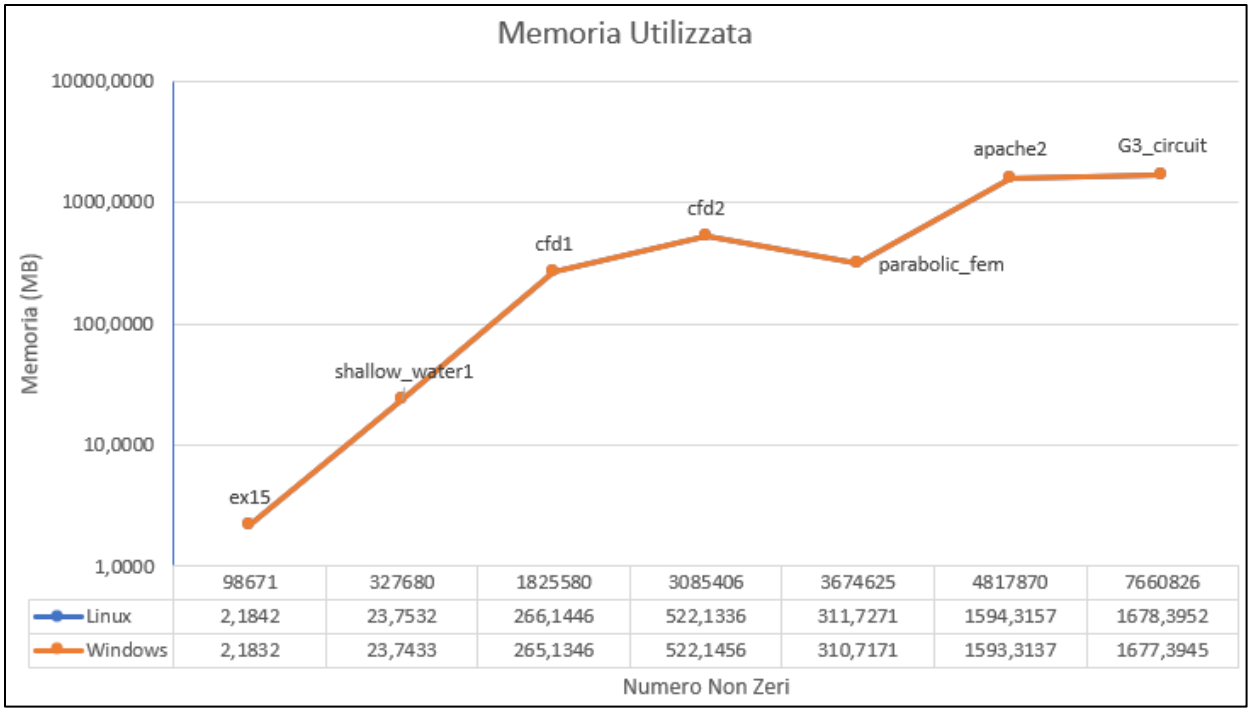


Figura 14 - Grafico memoria utilizzata

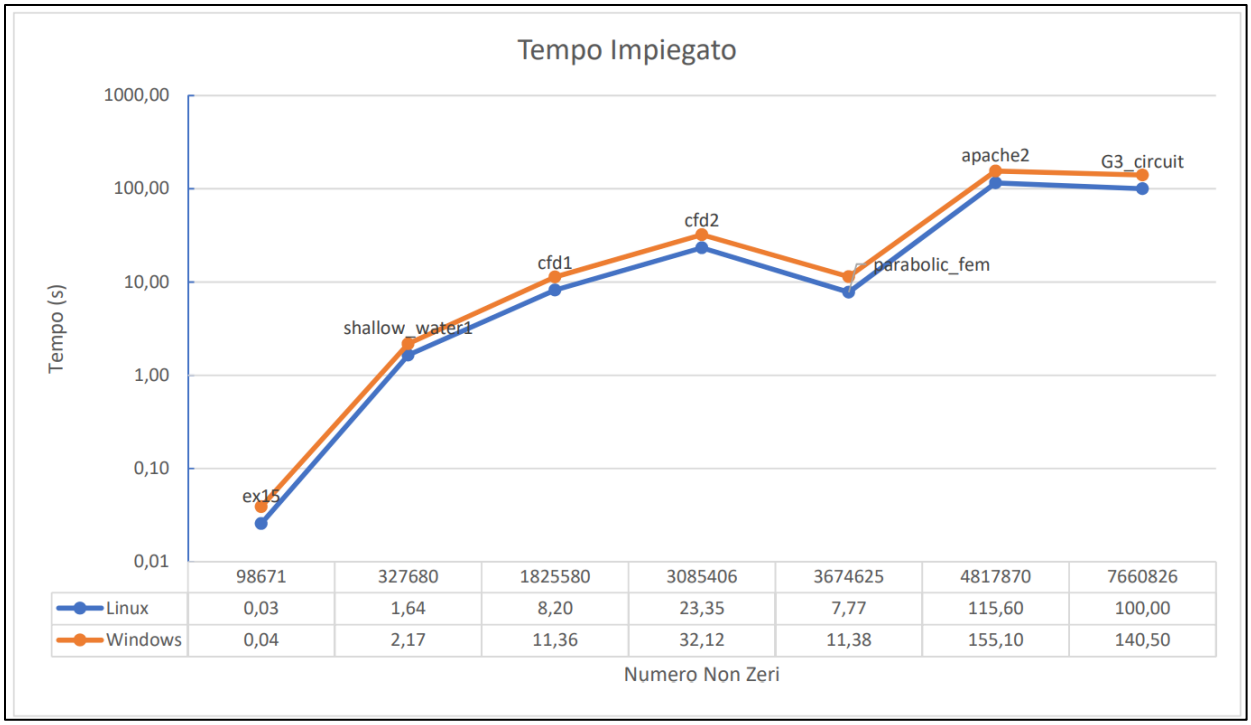


Figura 15 - Grafico tempo impiegato

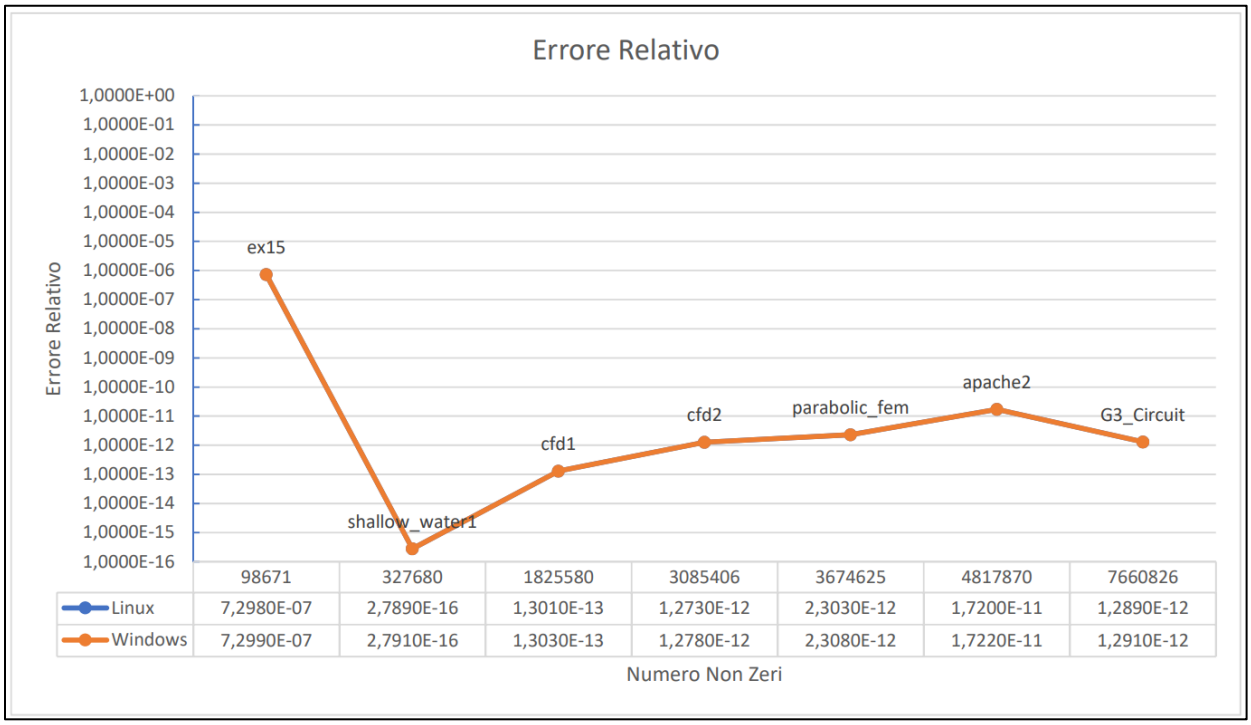


Figura 16 - Grafico errore relativo

Dal grafico in Figura 14 si noti come all'aumentare della dimensione della matrice aumenta anche la memoria occupata dalla matrice di fattorizzazione di Cholesky. L'andamento, però, non è propriamente lineare, infatti vediamo che:

- La matrice ***parabolic_fem***, anche se presenta un numero di non zeri superiore rispetto a ***cfd2***, presenta una matrice di fattorizzazione di Cholesky di dimensione molto inferiore. Questo probabilmente è dovuto al fatto che ***parabolic_fem*** risulta meglio condizionata rispetto a ***cfd2***, infatti presenta un numero di condizionamento sensibilmente inferiore. Inoltre, ***parabolic_fem*** risulta più sparsa rispetto a ***cfd2***, infatti queste due hanno un numero di non zeri non troppo differente, ma ***parabolic_fem*** presenta ***525.825*** righe e colonne, mentre ***cfd2*** solo ***123.440***;
- La matrice di fattorizzazione di ***apache2*** è di dimensioni molto simili alla matrice di fattorizzazione per ***G3_circuit***, anche se ***apache2*** risulta di molto più piccola;

Si è notato, quindi, che le matrici ***cfd2*** e ***apache2*** hanno prodotto molto *fill-in* nel caso di ***R***, su entrambe le piattaforme.

Per quanto riguarda il tempo impiegato, il grafico in Figura 15 presenta un andamento simile a quanto visto per la memoria occupata: si nota infatti che la matrice ***cfd2*** richiede più tempo rispetto a ***parabolic_fem***, anche se più piccola, e ***apache2*** richiede un tempo superiore rispetto a ***G3_circuit***, anche se più piccola.

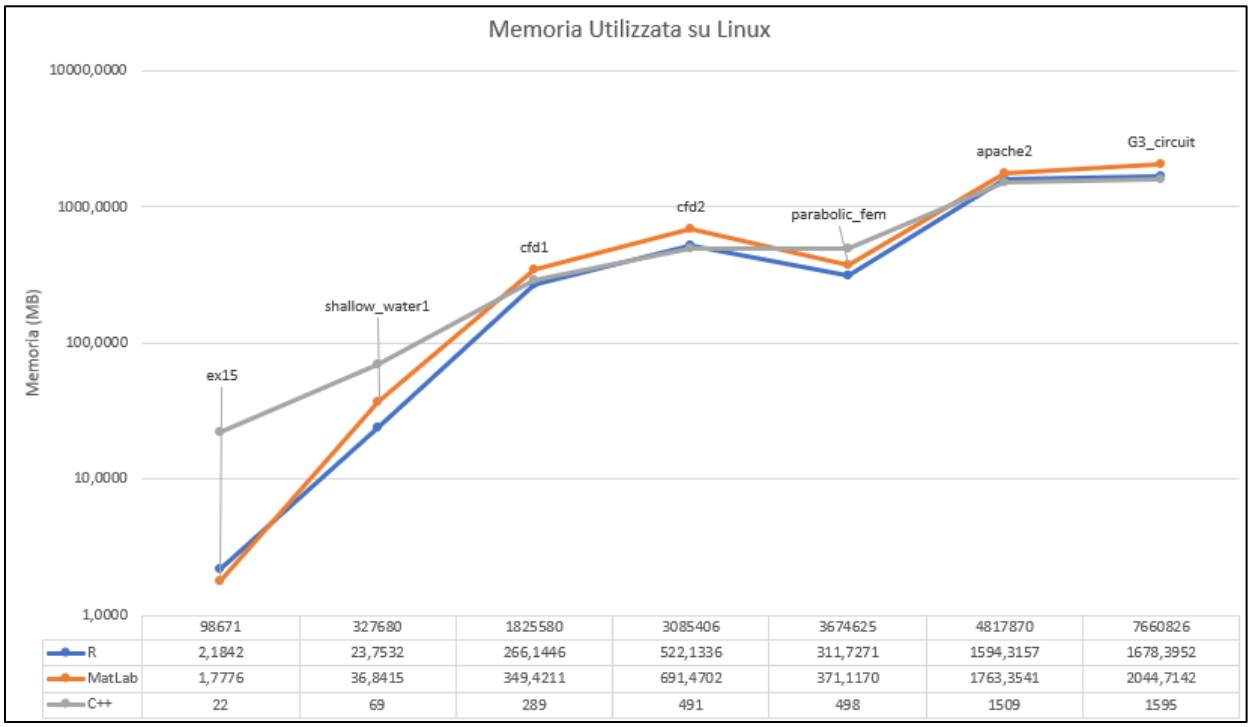
Per quanto riguarda l'errore relativo mostrato nel grafico in Figura 16, si è ottenuto un alto errore relativo per quanto riguarda ***ex15***, mentre per le altre matrici si è ottenuto un errore accettabile. L'alto errore relativo ottenuto su ***ex15*** è probabilmente dovuto al suo mal condizionamento, infatti si noti un numero di condizionamento per ***ex15*** molto alto, il più alto tra tutte le matrici. Invece, per quanto riguarda ***shallow_water1***, questa è quella che ha ottenuto l'errore relativo più basso, infatti si noti che è anche la matrice con numero di condizionamento più piccolo.

Per quanto riguarda il paragone tra i risultati ottenuti da ***R*** su Windows e Linux, questi hanno mostrato un errore relativo e una memoria occupata da ***R*** su Linux molto simili ai valori ottenuti da Windows, quindi valgono le stesse considerazioni fatte in precedenza. Quello che è sensibilmente cambiato è il tempo di esecuzione, mostrato nel grafico in Figura 15, infatti su Linux si sono osservati tempi sensibilmente inferiori che, per quanto riguarda ***R***, ci fanno preferire questo sistema operativo piuttosto che Windows.

Confronti

Memoria occupata

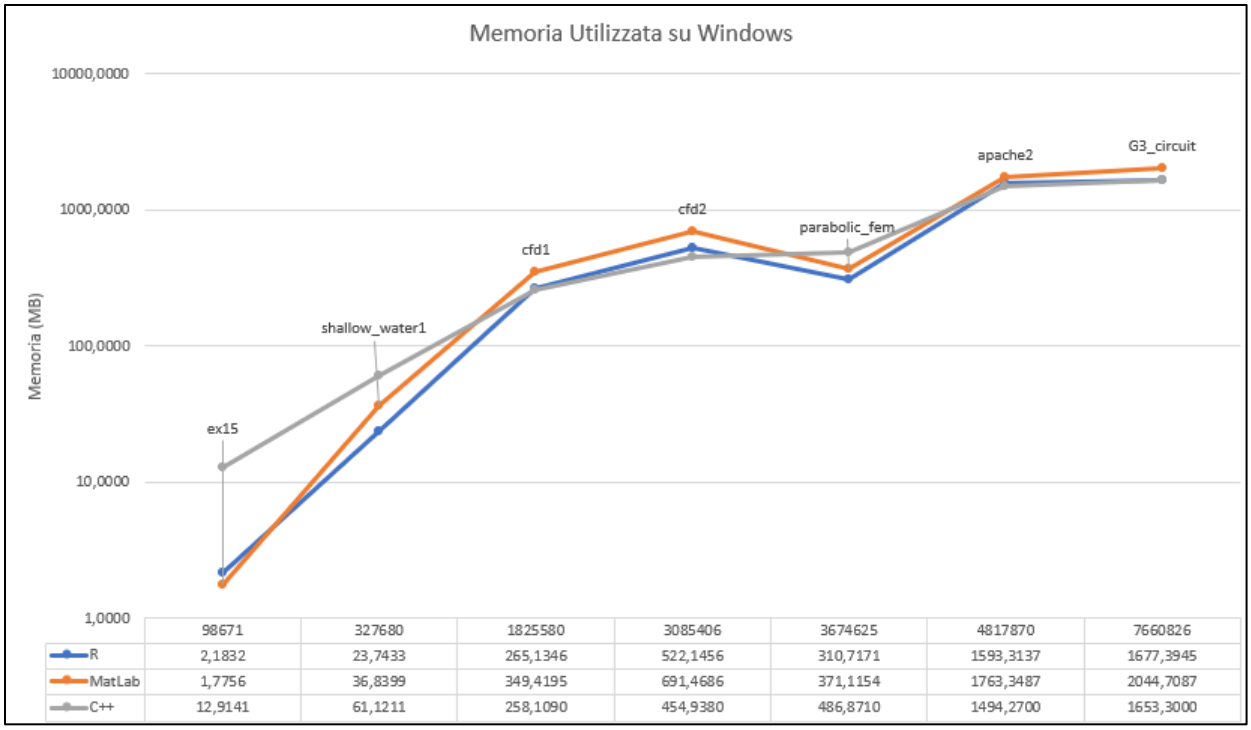
In Figura 17 e Figura 18 possiamo vedere rispettivamente i tre linguaggi messi a confronto per quanto riguarda l'impiego di memoria nella risoluzione del sistema lineare con Cholesky, rispettivamente su Linux e Windows.



	98671	327680	1825580	3085406	3674625	4817870	7660826
R	2,1842	23,7532	266,1446	522,1336	311,7271	1594,3157	1678,3952
MatLab	1,7776	36,8415	349,4211	691,4702	371,1170	1763,3541	2044,7142
C++	22	69	289	491	498	1509	1595

Figura 17 - Confronto rispetto alla memoria occupata su Linux

Per quanto riguarda Linux, possiamo vedere come ***MATLAB***, al crescere della dimensione della matrice considerata, sembri essere il linguaggio che utilizzata più memoria, a differenza di ***C++*** che, anche se utilizza molta memoria rispetto agli altri linguaggi sulle matrici piccole, sembra molto meglio ottimizzato su quelle di dimensione maggiore. Il linguaggio ***R***, invece, si comporta mediamente bene, perché ottimizza l'utilizzo di memoria sia sulle matrici più piccole che sulle più grandi.



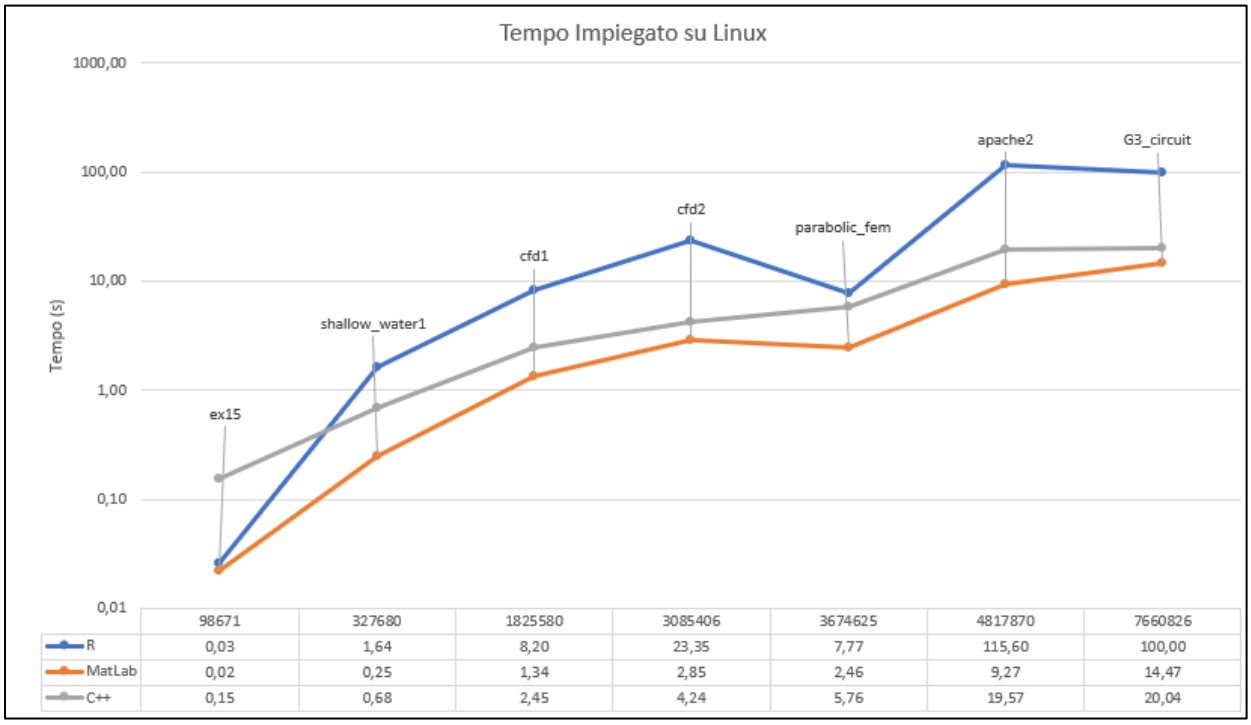
	98671	327680	1825580	3085406	3674625	4817870	7660826
R	2,1832	23,7433	265,1346	522,1456	310,7171	1593,3137	1677,3945
MatLab	1,7756	36,8399	349,4195	691,4686	371,1154	1763,3487	2044,7087
C++	12,9141	61,1211	258,1090	454,9380	486,8710	1494,2700	1653,3000

Figura 18 - Confronto rispetto alla memoria utilizzata su Windows

Per quanto riguarda la situazione su Windows, i risultati sono molto simili. Possiamo notare, in particolare, come in questo caso il linguaggio ***C++*** si comporti meglio rispetto al precedente confronto su Linux sulle matrici più piccole: questo probabilmente è dovuto al fatto che ***C++*** è più ottimizzato per Windows. Si nota, ancora, che ***MATLAB*** è il linguaggio con dispendio di memoria mediamente più alto e che comunque ***C++*** riporta ancora i valori più alti sulle matrici più piccole.

Tempo impiegato

In Figura 19 e Figura 20 possiamo vedere i tre linguaggi messi a confronto per quanto riguarda l'aspetto del tempo impiegato per la risoluzione del sistema lineare, risoluzione che prevede anche la fattorizzazione della matrice con Cholesky.



	98671	327680	1825580	3085406	3674625	4817870	7660826
R	0,03	1,64	8,20	23,35	7,77	115,60	100,00
MatLab	0,02	0,25	1,34	2,85	2,46	9,27	14,47
C++	0,15	0,68	2,45	4,24	5,76	19,57	20,04

Figura 19 - Confronto dei tre linguaggio rispetto al tempo impiegato su Linux

Per quanto riguarda Linux, possiamo vedere come il linguaggio ***R*** risulti nettamente il peggiore, a differenza di ***MATLAB*** e ***C++*** che risultano più paragonabili. Comunque, ***MATLAB*** è risultato il linguaggio migliore per quanto riguarda la velocità di fattorizzazione e risoluzione.

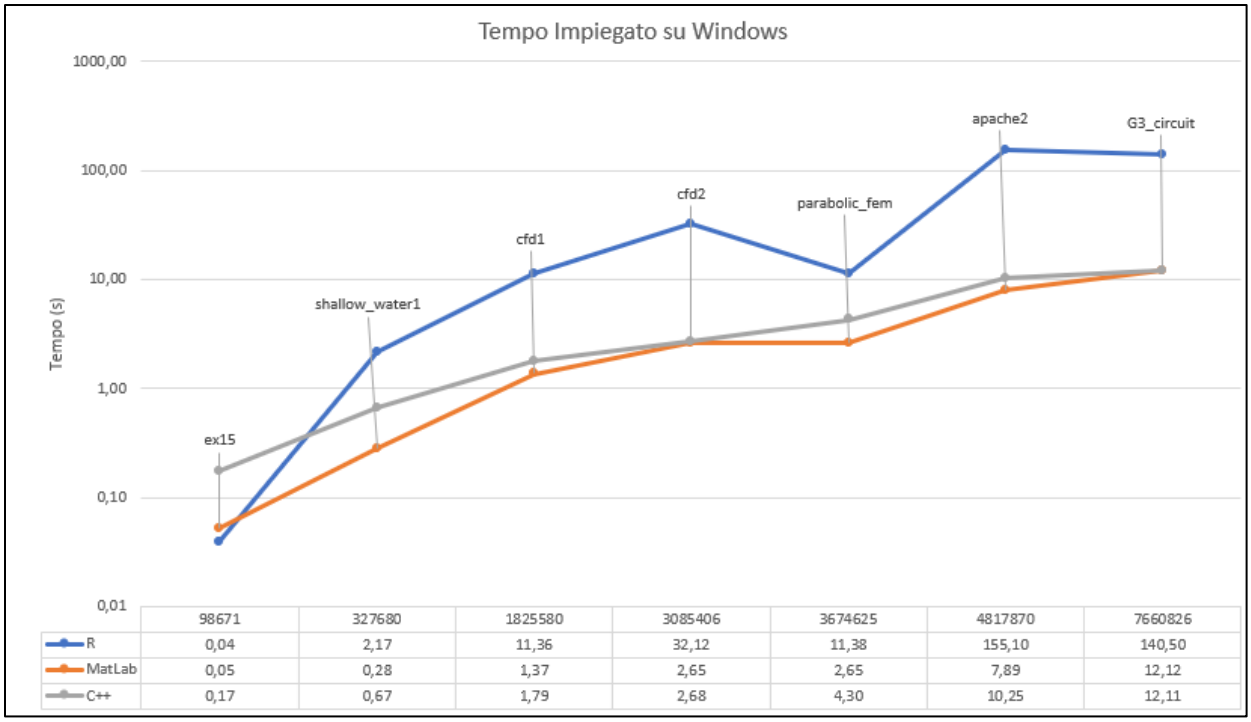
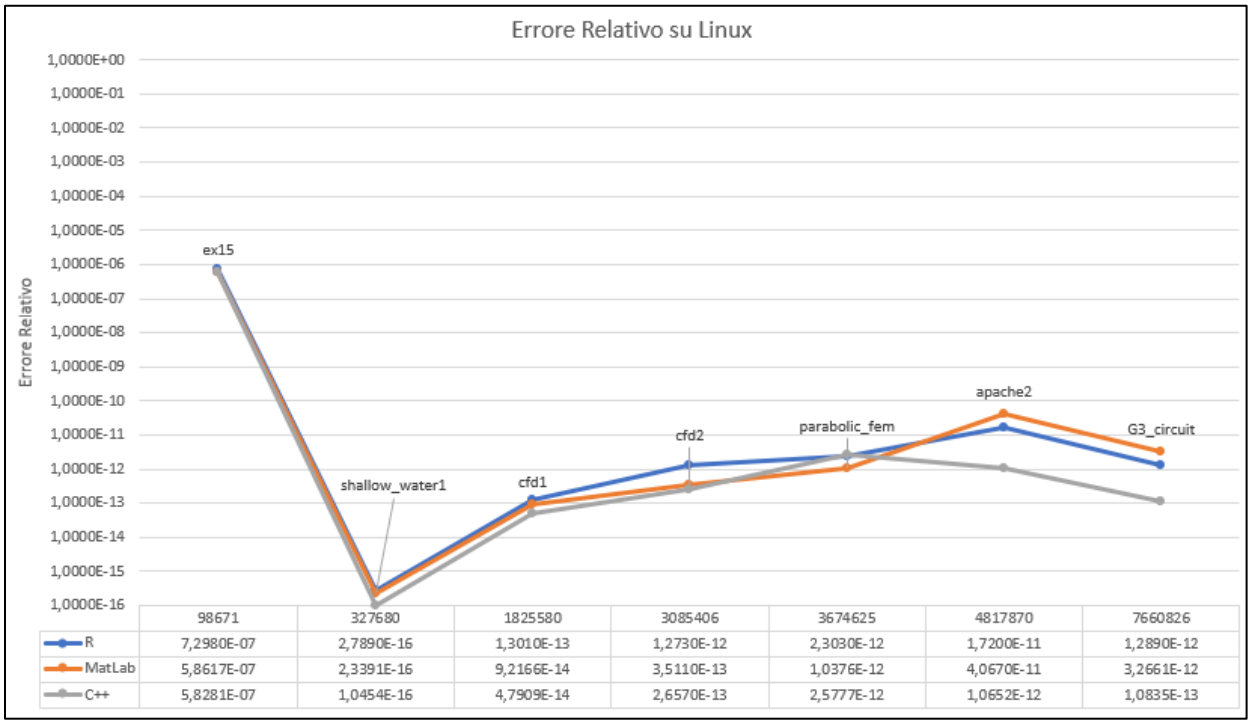


Figura 20 - Confronto dei tre linguaggi rispetto al tempo impiegato su Windows

Per quanto riguarda Windows, il confronto tra ***MATLAB*** e ***C++*** diventa più accesso, soprattutto perché ***C++*** risulta meglio ottimizzato su questa piattaforma. Anche se mediamente ***MATLAB*** risulta ancora il migliore per quanto riguarda la velocità di risoluzione e fattorizzazione, possiamo notare un significativo avvicinamento delle due curve. Per quanto riguarda il linguaggio ***R***, la sua situazione è ancora nettamente peggiore rispetto agli altri due.

Errore relativo

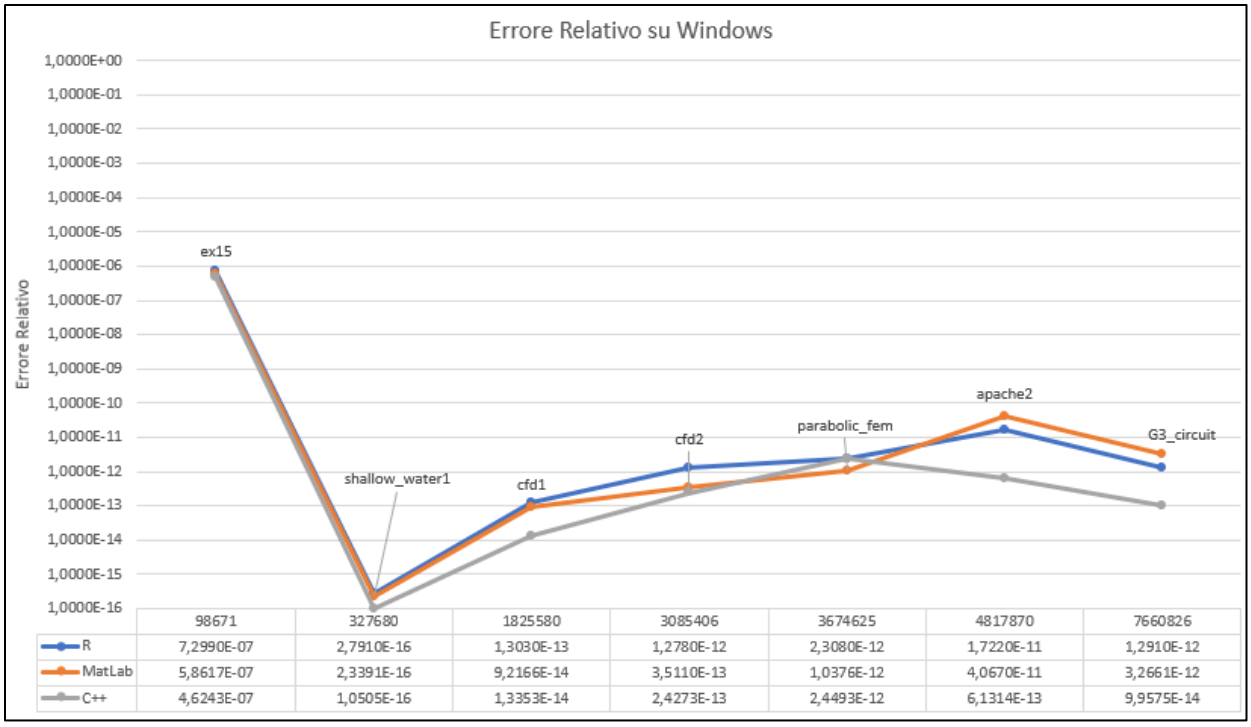
In Figura 21 e Figura 22 possiamo vedere i tre linguaggi messi a confronto rispetto alla metrica dell'errore relativo, calcolato rispetto alla soluzione esatta (vettore di tutti 1).



	98671	327680	1825580	3085406	3674625	4817870	7660826
R	7,2980E-07	2,7890E-16	1,3010E-13	1,2730E-12	2,3030E-12	1,7200E-11	1,2890E-12
MatLab	5,8617E-07	2,3391E-16	9,2166E-14	3,5110E-13	1,0376E-12	4,0670E-11	3,2661E-12
C++	5,8281E-07	1,0454E-16	4,7909E-14	2,6570E-13	2,5777E-12	1,0652E-12	1,0835E-13

Figura 21 - Confronto dei tre linguaggi rispetto all'errore relativo su Linux

Per quanto riguarda la situazione lato Linux, possiamo vedere come i tre linguaggi si sono comportati in maniera pressoché identica sulla matrice ***ex15***, ottenendo un alto errore relativo, dovuto al suo noto mal condizionamento. Mentre, per quanto riguarda le restanti matrici, il linguaggio che si comporta mediamente meglio è ***C++***, avendo ottenuto anche sulle due matrici più grandi un errore relativo significativamente più piccolo rispetto a quello ottenuto dagli altri due linguaggi.



	98671	327680	1825580	3085406	3674625	4817870	7660826
R	7,2990E-07	2,7910E-16	1,3030E-13	1,2780E-12	2,3080E-12	1,7220E-11	1,2910E-12
MatLab	5,8617E-07	2,3391E-16	9,2166E-14	3,5110E-13	1,0376E-12	4,0670E-11	3,2661E-12
C++	4,6243E-07	1,0505E-16	1,3353E-14	2,4273E-13	2,4493E-12	6,1314E-13	9,9575E-14

Figura 22 - Confronto dei tre linguaggi rispetto all'errore relativo su Windows

Per quanto riguarda la situazione lato Windows, non si notano grandi differenze rispetto a quanto analizzato su Linux, infatti, il linguaggio ***C++*** risulta mediamente il migliore.

Facilità d'uso

In merito alla facilità d'uso di ***MATLAB***, ***C++*** e ***R*** si possono svolgere le seguenti considerazioni:

- ***MATLAB*** è facile da usare quantomeno perché:
  - l'installazione dell'ambiente è semplice in quanto è sufficiente scaricare ed eseguire l'applicazione;
  - le funzioni offerte sono *built-in* e possono essere utilizzate senza richiedere l'importazione di librerie specifiche;
  - ***MATLAB*** è multi-paradigma e debolmente tipizzato: questo porta il linguaggio ad avere una sintassi semplice e intuitiva;
  - la piattaforma MATLAB Answers consente agli utenti di porre quesiti e formulare risposte. A questa piattaforma partecipano sovente anche dipendenti di MathWorks, Inc che forniscono risposte oltre che dettagli implementativi e spiegazioni teoriche;
  - ***MATLAB*** è mantenuto frequentemente.

  D'altro canto si segnala che l'utilizzo di ***MATLAB*** richiede in ogni caso una licenza d'uso del software. Inoltre sebbene la comunità di ***MATLAB*** sia molto attiva, risulta meno

numerosa rispetto a quella di alcuni software open-source. E questo porta per esempio ad avere meno codice disponibile in fonti aperte;

- Anche ***R*** è facile da usare. In particolare:
  - ***R*** è open source e quindi fruibile anche senza licenza d'uso del software;
  - anche l'installazione del linguaggio e dell'ambiente RStudio è semplice e immediata;
  - anche ***R*** è debolmente tipizzato. Quindi anche la sua sintassi è intuitiva;
  - anche ***R*** è mantenuto frequentemente.

  D'altra parte si segnala che in ***R*** solo alcune funzioni sono *built-in*: per l'utilizzo di molte funzioni è necessario importare librerie specifiche. Inoltre, l'esecuzione di alcune delle librerie necessarie in ambiente Linux richiede l'installazione di software aggiuntivo.
- ***C++*** è meno facile da usare. In particolare:
  - le funzioni della libreria ***Eigen*** richiedono di essere importate;
  - l'installazione di tutto quanto necessario ad eseguire un'applicazione ***C++*** può non essere banale;
  - ***C++*** è un linguaggio fortemente tipizzato: questo significa che il programmatore deve descrivere esplicitamente il tipo delle variabili in fase di dichiarazione. Eventuali errori possono provocare risultati inattesi e indesiderati.

  Tra i vantaggi di ***C++*** vi è la sua natura open source e la sua larga diffusione.

Documentazione

Infine confrontiamo i linguaggi ***MATLAB***, ***C++*** e ***R*** sotto il profilo della documentazione. A questo proposito si utilizzeranno tre parametri di confronto: (*i*) la reperibilità della documentazione; (*ii*) la completezza della descrizione delle funzioni offerte; e (*iii*) la chiarezza espositiva della documentazione.

- Reperibilità. La documentazione di ***MATLAB*** è reperibile sia online[3] sia tramite l'Help Center accessibile direttamente dal relativo ambiente di sviluppo. Anche la documentazione di ***R*** è accessibile sia online[4] sia tramite la sezione Help dell'ambiente di sviluppo RStudio. La documentazione di ***C++*** relativa alla libreria ***Eigen*** è invece accessibile solo online[5].
- Completezza. Le documentazioni di ***MATLAB*** e di ***R*** sono complete. Infatti entrambe queste documentazioni descrivono in dettaglio le funzioni offerte (con i rispettivi input e

[3] La documentazione di ***MATLAB*** è reperibile online all'indirizzo https://it.mathworks.com/help/matlab/.
[4] La documentazione di ***R*** è reperibile online all'indirizzo https://www.rdocumentation.org/.
[5] La documentazione della libreria ***Eigen*** di ***C++*** è reperibile online all'indirizzo https://eigen.tuxfamily.org/dox/.

output) e i relativi *package* (per quanto riguarda ***R***). La documentazione della libreria ***Eigen*** di ***C++*** è invece in parte incompleta. Infatti la documentazione di questa libreria descrive in modo completo le classi e le funzioni appartenenti ai domini applicativi più significativi. Tuttavia la documentazione delle classi e delle funzioni appartenenti alla sezione *unsupported modules* è a tratti molto carente.

- Chiarezza. In merito alla chiarezza delle documentazioni si possono svolgere le seguenti considerazioni:
    - La documentazione di ***MATLAB*** è chiara. Infatti a supporto della descrizione delle funzioni vengono presentati: (*i*) esempi applicativi; (*ii*) soluzioni di problemi tipici; (*iii*) funzioni utili correlate; *(iv*) eventuali alternative migliori; e (*v*) domande e risposte correlate presenti nella sezione Assistenza. D'altro canto si segnala anche che la natura proprietaria di ***MATLAB*** limita in taluni casi la chiarezza della documentazione disponibile pubblicamente. Ad esempio, la documentazione illustra in termini generali il funzionamento della funzione ***mldivide*** su matrici sparse. Tuttavia non è chiaro come questa funzione effettui permutazioni sulle matrici prima di procedere alla fattorizzazione.
    - Anche la documentazione di ***R*** è chiara. Infatti per ogni funzione e *package* vengono presentati: (*i*) esempi applicativi; (*ii*) funzioni utili correlate; e (*iii*) riferimenti bibliografici rilevanti.
    - La documentazione della libreria ***Eigen*** di ***C++*** è generalmente chiara in quanto presenta: (*i*) numerosi esempi applicativi; (*ii*) capitoli illustrativi dedicati a specifici argomenti; e (*iii*) alcuni temi generali di applicazione della libreria. Tuttavia occorre segnalare che a volte la descrizione dei parametri delle funzioni è molto sintetica. Inoltre la sezione relativa agli *unsopported modules* è poco documentata e quindi non sempre facilmente comprensibile.

Conclusioni

Riassumendo, possiamo dire che:

- Rispetto alla memoria impiegata nella fattorizzazione e risoluzione delle diverse matrici analizzate, ***MATLAB*** è risultato il peggiore mentre, i linguaggi ***R*** e ***C++*** risultano paragonabili, con piccolo vantaggio di ***C++***, soprattutto su Windows;
- Rispetto alla velocità di fattorizzazione e risoluzione del sistema lineare, il linguaggio ***R*** è risultato nettamente il peggiore, secondo ***C++*** con tempo accettabili, e primo con un certo scarto ***MATLAB***, che ha ottenuto tempi paragonabili su Linux e Windows. Comunque, è importante notare il fatto che ***C++*** su Windows ha mostrato tempi migliori rispetto a Linux, più vicini a quelli di ***MATLAB***;
- Rispetto alla grandezza dell'errore relativo, i tre linguaggi sono abbastanza paragonabili, anche se ***C++*** ha ottenuto risultati migliori sulle due matrici più grandi;

Tirando le somme, possiamo dire che il linguaggio migliore rispetto ai tre aspetti è sicuramente ***C++***, perché ha dimostrato un buon impiego della memoria, una accettabile velocità di esecuzione, e un errore relativo mediamente più basso rispetto agli altri due linguaggi, soprattutto sulle due matrici più grandi. In particolare, utilizzando questo linguaggio su Windows si possono sfruttare al meglio le sue ottimizzazioni, con risultati ancora migliori.

Comunque, nel caso in cui la metrica fondamentale da massimizzare fosse la velocità di esecuzione, a scapito della memoria occupata, allora è bene considerare ***MATLAB***, perché si è dimostrato il migliore rispetto al tempo impiegato. In particolare si può suggerire di preferire l'utilizzo di ***MATLAB*** su Linux per matrici di piccole dimensioni, e su Windows per matrici di grandi dimensioni.

Per quanto riguarda il linguaggio ***R*** occorre fare una precisazione. È vero che con questo linguaggio sono stati ottenuti risultati mediamente inferiori a quelli ottenuti con ***C++*** e ***MATLAB*** in tema di memoria, velocità di esecuzione ed errore relativo. Tuttavia è anche vero che da un lato ***R*** risulta più semplice da installare e utilizzare, e risulta meglio documentato rispetto a ***C++***; dall'altro lato ***R*** non richiede una licenza d'uso del software, a differenza di ***MATLAB***. In questo quadro allora si potrebbe pensare di preferire l'utilizzo di ***R*** qualora si voglia avere a disposizione uno strumento semplice da usare e che sia anche necessariamente open source. In questo caso è preferibile utilizzare Linux, dove si sono registrati tempi di esecuzione mediamente più bassi rispetto a Windows.

Codici utilizzati

MATLAB

Il codice utilizzato per implementare il metodo di Cholesky su ***MATLAB*** è il seguente:

```
clear all, close all, clc;

% Caricamento matrici dalla cartella "matrixes".
matrixes = dir('matrixes\*.mat');

% Creazione delle variabili in cui salvare i risultati.
table_variables = {'name'; 'time (in seconds)'; 'relative error'; ...
    'initial memory usage (in MB)'; 'condition number'};
table_column_name = {};
table_column_time = [];
table_column_error = [];
table_column_initial_memory = [];
table_column_condition = [];

% Risoluzione di un sistem lineare con la fattorizzazione di Cholesky
% per ogni matrice.
for i=1:length(matrixes)

    % Caricamento della singola matrice A.
    matrix = load(['matrixes\', matrixes(i).name]);
    A = matrix.Problem.A;

    % Abilitazione della scrittura su file dell'output del terminale.
    diary on;

    % Stampa a schermo del nome della matrice.
    disp(matrix.Problem.name);

    % Calcolo della memoria occupata prima della risoluzione del sistema
    % lineare.
    memory_start = monitor_memory();
    memory_start

    % Calcolo del numero di condizionamento della matrice.
    k = condest(A);

    % Impostazione della soluzione esatta formata solo da 1.
    xe = ones(length(A),1);

    % Calcolo del vettore dei termini noti a partire dalla matrice A e
    % dalla soluzione esatta.
    b = A*xe;

    % Abilitazione della stampa a schermo di informazioni dettagliate
    % sulla risoluzione di sistemi lineari per matrici sparse.
    spparms('spumoni', 2);
```

```
    % Inizio del calcolo del tempo di risoluzione del sistema lineare.
    tic;

    % Risoluzione del sistema lineare con metodo di Cholesky.
    x = A\b;

    % Termine del calcolo del tempo di risoluzione del sistema lineare.
    time = toc;

    % Disabilitazione di diary.
    diary off;
    
    % Calcolo dell'errore relativo (come norma Euclidea) fra la soluzione 
    % calcolata e la soluzione esatta.
    error = norm(x-xe,2)/norm(xe,2);

    % Salvataggio dei risultati.
    table_column_name = [table_column_name, matrix.Problem.name];
    table_column_time = [table_column_time, time];
    table_column_error = [table_column_error, error];
    table_column_initial_memory = [table_column_initial_memory, memory_start];
    table_column_condition = [table_column_condition, k];

    % Rimozione delle variabili temporanee allocate.
    clear memory_start;
    clear k, clear xe, clear b, clear x, clear time, clear error;
end

% Apertura del file csv per memorizzare i risultati su disco.
fileID = fopen('matlab.csv', 'w');

% Scrittura dei nomi delle variabili nel file csv.
for i = 1:length(table_variables)
    if i == length(table_variables)
        fprintf(fileID, '%s', table_variables{i});
    else
        fprintf(fileID, '%s,', table_variables{i});
    end
end

% Scrittura dei risultati nel file csv.
fprintf(fileID, '\n');
for i = 1: length(matrixes)
    fprintf(fileID, '%s,%d,%d,%d,%d\n', table_column_name{i}, ...
        table_column_time(i), table_column_error(i), ...
        table_column_initial_memory(i), table_column_condition(i));
end
fclose(fileID);
```

Il codice utilizzato su ***MATLAB*** per calcolare la memoria utilizzata dal *workspace* dell'ambiente è invece il seguente:

```
% Questa funzione usa il comando WHOS per individuare gli elementi
% all'interno del BASE workspace. Dopodiché somma i byte di ciascun
% elemento e restituisce la somma in MB.
function [memory_in_use] = monitor_memory()
elements_in_memory = evalin('base','whos');
if size(elements_in_memory,1) > 0

    for i = 1:size(elements_in_memory,1)
        array_of_elements(i) = elements_in_memory(i).bytes;
    end
    memory_in_use = sum(array_of_elements);
    memory_in_use = memory_in_use/1048576;

else
    memory_in_use = 0;
end
```

C++

Di seguito sono riportati i codici utilizzati per applicare il metodo di Cholesky in ***C++***. Per semplificare l'utilizzo della libreria ***Eigen*** è stata creata una classe Framework che permette di caricare la matrice scelta, applicare Cholesky e mostrare i risultati ottenuti.

```cpp
#pragma once

#include <Eigen/Sparse>
#include <Eigen/PardisoSupport>
#include <unsupported/Eigen/SparseExtra>
#include <string>
#include <chrono>

using namespace Eigen;
using namespace std::chrono;

typedef SparseMatrix<double> spMatrix;

class Framework {

public:

    void printDBG(const char* error) { printf("[ERROR] %s\n", error); }

    bool isMatrixLoaded() { if (SparseMatrix.valuePtr()) return true; return false; }

    bool chol();

    void results();

    Framework(const char* fileName) {

        std::string input = fileName;

        if (input.empty()) { printDBG("Please provide a valid matrix file");  return; }

#if defined(_MSC_VER)
        MatrixName = input.substr(input.find_last_of("\\") + 1);
#elif defined(__GNUC__)
        MatrixName = input.substr(input.find_last_of("/") + 1);
        #endif
        loadMarket(SparseMatrix, input);
```

```cpp
    }

    const char* getMatrixName() { return MatrixName.c_str(); }

private:
    spMatrix    SparseMatrix;
    double      RelativeError;
    double      ElapsedTime;
    double      MatrixMemoryUsage;
    double      InitialMemoryUsage;
    std::string MatrixName;
};
```

```cpp
#include "Framework.h"

#if defined(_MSC_VER)
#include <Windows.h>
#include <Psapi.h>
#elif defined(__GNUC__)

#include "stdlib.h"
#include "stdio.h"
#include "string.h"

int parseLine(char* line) {
    // This assumes that a digit will be found and the line ends in " Kb".
    int i = strlen(line);
    const char* p = line;
    while (*p <'0' || *p > '9') p++;
    line[i - 3] = '\0';
    i = atoi(p);
    return i;
}

int getValue() { //Note: this value is in KB!
    FILE* file = fopen("/proc/self/status", "r");
    int result = -1;
    char line[128];

    while (fgets(line, 128, file) != NULL) {
        if (strncmp(line, "VmRSS:", 6) == 0) {
            result = parseLine(line);
            break;
        }
    }
    fclose(file);
    return result;
}
#endif
```

```cpp
bool Framework::chol() {

  //Esclusione memoria programma
#if defined(_MSC_VER)
  PROCESS_MEMORY_COUNTERS pmc;
  GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc));
  double programMemory = pmc.WorkingSetSize / (1024.0 * 1024.0);
#elif defined(__GNUC__)
  double programMemory = getValue() / 1024;
#endif

  // Inizializzazione Matrice Simmetrica
  spMatrix resultMatrix = SparseMatrix.selfadjointView<Eigen::Lower>();
  SparseMatrix.data().clear();
  resultMatrix.makeCompressed();

  // Memoria Iniziale Matrice
#if defined(_MSC_VER)
  GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc));
  double initialMemory = pmc.WorkingSetSize / (1024.0 * 1024.0);
#elif defined(__GNUC__)
  double initialMemory = getValue() / 1024;
#endif

  initialMemory -= programMemory;
  InitialMemoryUsage = initialMemory;

  // Vettori soluzione esatta e termini noti
  VectorXd xe = VectorXd::Ones(resultMatrix.rows());
  VectorXd b = resultMatrix * xe;

  //Inizializzazione Solver
  PardisoLLT<spMatrix>  solver;

  // Inizio timer Choleksy
  auto start = std::chrono::high_resolution_clock::now();
```

```
74
75      // Preparazione fattorizzazione matrice
76      solver.analyzePattern(resultMatrix);
77
78      // Fattorizzazione matrice
79      solver.factorize(resultMatrix);
80
81      // Calcolo soluzione
82      VectorXd x = solver.solve(b);
83
84      // Memoria Finale Matrice
85    #if defined(_MSC_VER)
86      GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc));
87      double finalMemory = pmc.WorkingSetSize / (1024.0 * 1024.0);
88    #elif defined(__GNUC__)
89      double finalMemory = getValue() / 1024;
90    #endif
91
92      // Calcolo memoria utilizzata Matrice fattorizzata
93      MatrixMemoryUsage = finalMemory - initialMemory;
94
95      // Fine timer Cholesky
96      auto end = std::chrono::high_resolution_clock::now();
97      std::chrono::duration<double> duration = end - start;
98
99      // Tempo esecuzione cholesky in secondi
100     ElapsedTime = duration.count();
101
102     // Calcolo errore relativo
103     RelativeError = (x - xe).norm() / xe.norm();
104
105     return true;
106   }
```

```
void Framework::results() {

    // Metodo per print dei risulati ottenuti

    std::cout << "=== " << getMatrixName() << " === "  << std::endl;
    std::cout << "[>] Elapsed Time: " << ElapsedTime << " s" << std::endl;
    std::cout << "[>] Relative Error: " << RelativeError << std::endl;
    std::cout << "[>] Matrix Initial Memory Usage: " << InitialMemoryUsage << " MB" << std::endl;
    std::cout << "[>] Matrix Final Memory Usage: " << MatrixMemoryUsage << " MB" << std::endl;
}
```

Il codice seguente riporta un esempio di utilizzo della classe Framework.

```cpp
#include <iostream>
#include "Framework.h"

#if defined(_MSC_VER)
#include <direct.h>
#define getcwd _getcwd
#elif defined(__GNUC__)
#include <unistd.h>
#endif

int main()
{
    std::cout << "[INFO] Please enter matrix file name (file must be inside a data folder): ";
    std::string fileName = "";

    std::cin >> fileName;

    while (fileName.empty() || fileName.find(".mtx") == std::string::npos) {
        std::cout << "[INFO] Please insert a valid mtx file: ";
        std::cin >> fileName;
    }

    char buff[FILENAME_MAX];

    if (getcwd(buff, FILENAME_MAX)) {

        std::string current_path = std::string(buff);

#if defined(_MSC_VER)
        current_path += "\\data\\" + fileName;
#elif defined(__GNUC__)
        current_path += "/data/" + fileName;
#endif

        Framework fChol = Framework(current_path.c_str());

        if (fChol.isMatrixLoaded()) {

            printf("[SUCCESS] Loaded Matrix\n");

            if (fChol.chol()) fChol.results();
        }
        else fChol.printDBG("File not found");
    }
    else std::cout << "[ERROR] Failed to parse current folder!" << std::endl;

#if defined(_MSC_VER)
    system("pause");
#elif defined(__GNUC__)
    std::cin.get();
#endif
    return 0;
}
```

Linguaggio R

Prima fase di installazione delle librerie necessarie, e loro import all'interno dell'ambiente.

```r
## Settaggio wd
allPackageInstalled = installed.packages()[, 1]
if (!("rstudioapi" %in% allPackageInstalled)) {
  install.packages("rstudioapi")
}
library(rstudioapi)
setwd(dirname(getSourceEditorContext()$path))
rm(allPackageInstalled)

## Installazione di tutte le librerie necessarie

source(paste(getwd(), "/PackageInstaller.R", sep = ""))

#### Import librerie necessarie

library(R.matlab)
library(spam)
library(spam64)
library(tictoc)
library(Matrix)
library(matrixcalc)
```

Seconda fase di definizione della funzione per la fattorizzazione di Cholesky e risoluzione del sistema lineare. La funzione esegue, inoltre, la stampa a video di tutte le metriche necessarie per l'analisi.

```
#### Funzione per risoluzione con Cholesky

myChol = function(matrix) {

  ## Calcolo numero di non zeri
  matrix_nonzero = Matrix::nnzero(matrix)
  print(paste("-> ", name, " - matrix non zeros: ", matrix_nonzero, sep = ""))

  ## Check simmetria perché, se non lo è, non posso fare la fattorizzazione
  if (isSymmetric.spam(matrix) == F)
    return ("Matrix not Symmetric!")

  ## Calcolo della soluzione esatta (tutti 1) e del termine noto associato
  xe = rep(1, nrow(matrix))
  b = matrix %*% xe

  print("-------------------------------------------------------")
  ## Esecuzione della decomposizione di Cholesky pivot MMD
  i = Sys.time()
  R = tryCatch(
    {
      chol.spam(matrix, pivot = "MMD")
    },
    error = function(e){
      ## Errore sollevato se la matrice non è definitiva positiva. Se non fosse
      ## stata simmetrica ci saremmo fermati al controllo di riga 50
      return (e)
    }
  )
  chol_time = difftime(Sys.time(), i, units = "secs")
  print(chol_time, digits = 4L)
  print(paste("-> ", name, " - chol time pivot MMD: ", chol_time, sep = ""))
  chol_size = object.size(R)
  print(chol_size, units = "MB", standard = "SI", digits = 4L)
  print(paste("-> ", name, " - chol size pivot MMD: ", chol_size, sep = ""))

  ## Risoluzione sistema lineare pivot MMD
  i = Sys.time()
  x = solve.spam(R,b)
  solve_time = difftime(Sys.time(), i, units = "secs")
  print(solve_time, digits = 4L)
  print(paste("-> ", name, " - solve time pivot MMD: ", solve_time, sep = ""))
  print(chol_time+solve_time, digits = 4L)
  print(paste("-> ", name, " - tot time pivot MMD: ", solve_time+chol_time, sep = ""))
  print("Total size: ")
  print(chol_size + object.size(x), units = "MB", standard = "SI", digits = 4L)

  ## Calcolo errore relativo
  rel_error = norm(xe - x, type = "2") / norm(xe, type = "2")
  print(rel_error, digits = 4L)
  print(paste("-> ", name, " - relative error pivot MMD: ", rel_error, sep = ""))
}
```

Questa funzione è stata poi eseguita su tutte le matrici, una alla volta, deallocando tra una esecuzione e l'altra tutta la memoria utilizzata, chiamando esplicitamente anche il Garbage Collector.

Riferimenti

[1] L. N. Trefethen e D. Bau III, Numerical Linear Algebra, SIAM, 1997.

[2] R. A. Horn e C. R. Johnson, Matrix Analysis, Cambridge University Press, 2013.

[3] A. Basu, S. Bensalem, M. Bozga, P. Bourgos, M. Maheshwari e J. Sifakis, «Component Assemblies in the Context of Manycore,» in *Formal Methods for Components and Objects*, 2011.

[4] T. A. Davis, «User Guide for CHOLMOD: a sparse Cholesky factorization and modification package,» 2019.

[5] R. W. Freund, «Large-Scale Matrix Computations,» in *Handbook of Linear Algebra*, Chapman & Hall/CRC, 2013.

[6] MathWorks Support Team. [Online]. Available: https://it.mathworks.com/matlabcentral/answers/97560-how-can-i-monitor-how-much-memory-matlab-is-using.