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

Alberici Federico - 808058
Bettini Ivo Junior - 806878
Cocca Umberto - 807191
Traversa Silvia - 816435

Anno Accademico 2019 - 2020

Indice

Introduzione

Lo scopo di questo progetto è studiare l'implementazione del metodo di Cholesky per la risoluzione di sistemi lineari con matrici sparse, simmetriche e definite positive, in ambienti di programmazione open source e confrontarla con l'implementazione MATLAB. Questo confronto viene eseguito su due sistemi operativi diversi, Windows e Linux.

Per ognuna delle matrici calcoliamo:

- Il tempo necessario per calcolare la soluzione x del sistema lineare Ax=b
- L'errore relativo tra la soluzione calcolata x e la soluzione esatta xe, trovata come soluzione del sistema Axe=B
- La memoria necessaria per risolvere il sistema, ovvero l'aumento della dimensione del programma in memoria da subito dopo aver letto la matrice a dopo aver risolto il sistema.

Per poter raggiungere questo obiettivo abbiamo deciso di confrontare MATLAB con gli ambienti open source C++, R e Python. È possibile trovare il listato dei codici alla seguente **repository**.

MATLAB

MATLAB è un ambiente per il calcolo numerico e l'analisi statistica scritto in C. Nella scrittura del codice, per prima cosa abbiamo importato le diverse matrici attraverso la funzione *mmread* nel formato *.mtx* (invece del formato MATLAB .mat) in modo da mantenere linearità con gli altri linguaggi usati.

```
% read the matrix
tic;
[A, row, col, entries] = mmread("../../data/" + name);
import_time = toc * 1000;
info_A = whos('A');
```

Attraverso la funzione *tic toc* abbiamo calcolato il tempo di esecuzione specifiche porzioni del codice di interesse, come il calcolo della decomposizione di Cholesky e della soluzione finale del sistema lineare. Dopo aver calcolato la soluzione esatta xe del sistema lineare A*xe = b, attraverso il comando Chol abbiamo eseguito la decomposizione di Cholesky.

```
% Cholesky decomposition
tic;
R = chol(A);
cholesky_time = toc * 1000;
info_R = whos('R');
```

La funzione *chol* è in grado di accorgersi se la matrice passata è definita positiva e simmetrica. Dopo aver determinato lo spazio in memoria occupato dalla matrice decomposta attraverso il metodo di Cholesky abbiamo calcolato la soluzione finale x, grazie alla quale abbiamo potuto trovare l'errore relativo e abbiamo infine misurato il tempo di risoluzione.

Abbiamo gestito eventuali eccezioni generate durante l'esecuzione del programma attraverso un *catch exception*, che ci ha permesso di accorgerci che MATLAB va in "*out of memory*" con matrici particolarmente grandi (con matrici di dimensione superiore a cfd2.mtx, la quale ha 123.440 righe e colonne).

C++

La decomposizione di Cholesky è stata effettuata sfruttando Eigen, libreria template per l'algebra lineare. Delle numerose funzionalità messe a disposizione sono stati usati i moduli Sparse per la gestione di matrici sparse e *SparseCholesky* per la decomposizione.

Il programma è compilabile tramite Makefile presente nella cartella dei sorgenti (./src/c++/) e va eseguito via riga di comando fornendo come parametro la cartella in cui sono inserite le matrici in formato Matrix Market (es. $./main.out ../../data) L'analisi viene eseguita su tutte le matrici .mtx presenti nella cartella data.

Il primo problema affrontato è stata l'importazione in memoria del formato .mtx. Nonostante esistano delle funzioni di import reperibili online (**MatrixMarket/mmio-c.html**) abbiamo deciso di scrivere un parser ad-hoc per matrici Matrix Market.

```
// Import
t1 = std::chrono::high_resolution_clock::now();
Eigen::SparseMatrix<double> spMatrix = readMatrix(mtxFile);
t2 = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
```

Figure 1: *Import con Eigen::SparseMatrix<double>readMatrix(std::string &filename)*

Non avendo trovato una funzione che permettesse di controllare lo spazio in memoria occupato da un oggetto, la dimensione delle matrici è stata calcolata empiricamente, studiandone la metodologia di memorizzazione. Eigen (così come la libreria utilizzata per il linguaggio Python) comprime la matrice sparsa utilizzando il formato Compressed Sparse Row/Column. Vengono mantenuti in memoria due array di interi (4 byte), contenti indici per la ricostruzione della matrice, e un array di double (8 byte), contenete i valori non-zero.

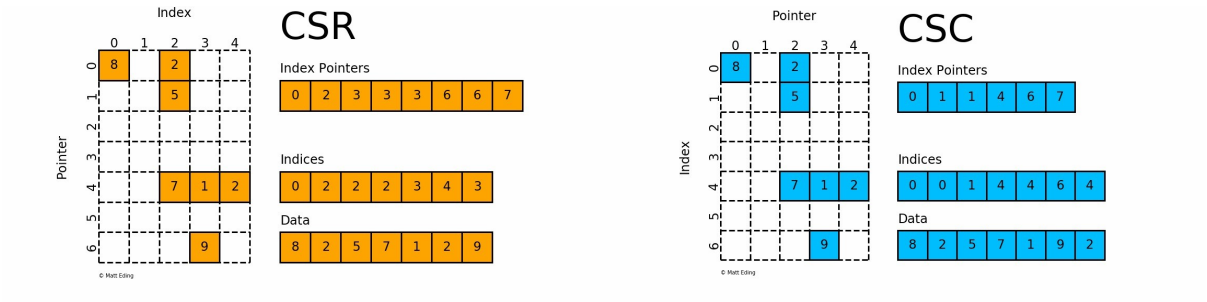


Figure 2: *formato CSR e CSC, matrici sparse*

Il calcolo per ricavare la dimensione della matrice è quindi il seguente:

$$MatrixSize = (Inner\_Pointers.size() + Indices.size()) * 4 + (Data.size() * 8)[byte]$$

A fronte di ulteriori strutture interne della libreria tale metodo permette di ottenere una buona approssimazione del costo in termini di memoria.

I passi dell'analisi sono gli stessi eseguiti in MATLAB, utilizzando opportunamente la sintassi C++ e la libreria Eigen. In particolare la decomposizione di Cholesky viene calcolata tramite la funzione

$$Eigen :: SimplicialLLT < Eigen :: SparseMatrix < double >> chol()$$

che riduce il fill-in applicando una permutazione simmetrica prima della fattorizzazione.

R

R è un linguaggio di programmazione e ambiente di sviluppo opensource disponibile per diversi sistemi operativi, tra i quali Linux e Windows.

Inizialmente il codice è stato scritto sfruttando la libreria Matrix, ma abbiamo riscontrato che essa non permette la lettura di matrici di dimensioni elevate, abbiamo deciso dunque di ricorrere alla libreria *spam*. Come riporta la documentazione ufficiale infatti questa libreria è veloce e scalabile, con il pacchetto di estensione *spam64* che abbiamo usato.

Importiamo la matrice con la funzione *read.MM*, che ci permette di salvarla in formato sparso. Con il comando *chol.spam* effettuiamo la decomposizione di Cholesky sulla matrice in esame. Essa deve essere simmetrica e definita positiva: queste due caratteristiche vengono verificate automaticamente dalla funzione stessa, che restituisce un errore nel caso in cui queste non siano rispettate, motivo per il quale è stato inserito un blocco try-catch.

```
# Cholesky decomposition
i2 <- Sys.time()
R <- tryCatch(
  {
    chol.spam(A)
  },
  error = function(e){
    print(name)
    print(e) # if matrix is not positive definite or symmetric, an error is signalled.
  }
)
if(inherits(R, "error")) next;  # skip to next matrix if error occours
```

La libreria in uso implementa anche la funzione *solve.spam* che, dato in input il risultato della funzione *chol.spam*, calcola direttamente il risultato del sistema lineare, combinando in maniera opportuna *backsolve* e *forwardsolve*.

Come in MATLAB e negli altri linguaggi di programmazione considerati, abbiamo calcolato l'utilizzo della memoria, il tempo necessario per la risoluzione del sistema lineare e l'errore relativo.

Python

In python, il calcolo della decomposizione di Cholesky può essere eseguito con la libreria *numpy* e *scipy* per matrici dense e con *scikits.sparse* per le matrici sparse. Il pacchetto *scikit-sparse* espande *scipy.sparse* ritornando le matrici in formato CSC.

Seppur il pacchetto *scikits.sparse* a detta dei creatori è usabile sia in ambiente Windows che Linux, in realtà ci sono dei problemi di installazione lato windows, come si può vedere nella schermata **issues** della repository ufficiale. Per installare il pacchetto su Windows infatti è stato necessario come prima cosa compilare attraverso *cmake* **suite sparse**, una potente libreria C/C++ che permette di eseguire operazioni su matrici sparse. Così facendo siamo stati in grado di generare le librerie della suite attraverso la build in release mode su Visual Studio. Una volta ottenuta la libreria abbiamo installato manualmente *scikits.sparse* per windows.

Le operazioni base eseguite dal programma sono le medesime effettuate negli altri linguaggi considerati. In particolare, la memoria occupata è stata calcolata empiricamente come visto in C++ poiché anche scikit sparse salva in memoria le matrici nello stesso formato di Eigen.

Il calcolo della decomposizione di Cholesky, poi, è stato eseguito con la funzione *cholesky*, la quale ritorna un risolutore factor. Quest'ultimo permette di risolvere il sistema lineare Ax = b come mostrato nella figura:

```
factor = cholesky(A)
x = factor(b)
```

Analisi dei risultati

Per l'analisi confrontiamo i grafici derivanti dai risultati ottenuti in output dalle varie esecuzioni dei linguaggi nei due sistemi operativi Linux e Windows. La raccolta dei dati è avvenuta su un unico pc provvisto di macchina virtuale per entrambi i sistemi operativi, in modo tale da avere una potenza di calcolo confrontabile statisticamente. Le specifiche tecniche sono le seguenti:

Windows x64
Processor: Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz, Number of cores: 4, Memory: Ram 11000 MB

Unix x64
Kernel: 5.3.0-53-generic, Processor: Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz, Number of cores: 4, Memory: Ram 11000 MB

I grafici scelti sono a linee con indicatori, dove sulle ascisse troviamo le matrici in ordine crescente per dimensione e sulle ordinate le tre grandezze richieste dal progetto:

- *chol_size*: memoria occupata dalla matrice decomposta con il metodo di Cholesky;
- *total_time*: somma del tempo necessario per la decomposizione di Cholesky e del tempo necessario per la risoluzione del sistema lineare data la matrice decomposta;
- *err*: errore relativo.

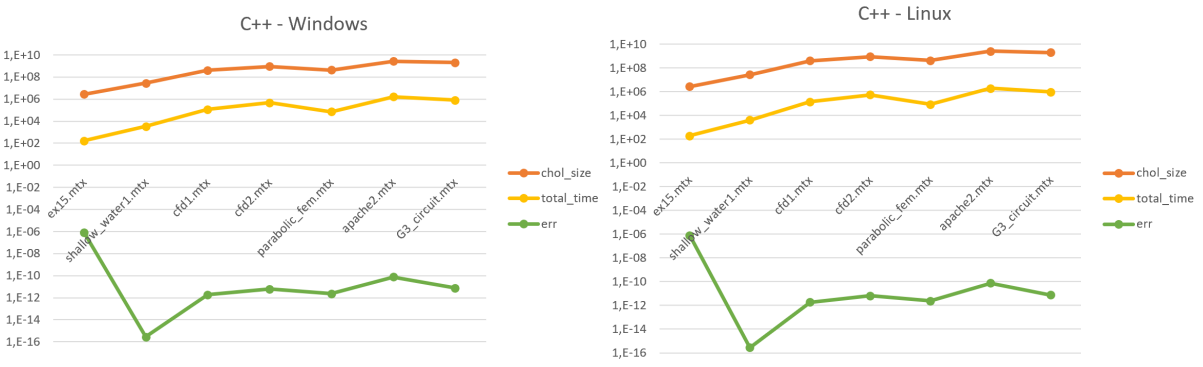
Nelle ordinate viene utilizzata una scala logaritmica poiché si hanno tre valori con ordini di grandezza diversi.

MATLAB

In entrambi i casi al crescere della dimensione della matrice aumenta il tempo di esecuzione e la memoria da queste occupata e si ha un andamento simile per l'errore relativo. È interessante evidenziare il picco verso il basso che si ottiene con la matrice *shallow_water1*, per la quale si ha l'errore relativo minore (ordine di grandezza 10^{-16}).

Con MATLAB non è stato possibile analizzare matrici di dimensioni superiori a *cfd2*, in quanto il programma ci ritorna un errore "*out of memory*".

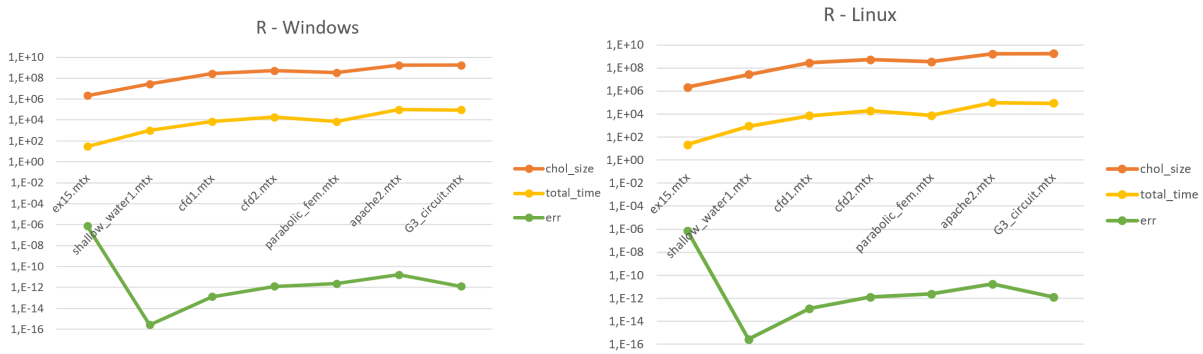
C++



Anche nel caso di C++ analizzando i grafici si possono notare andamenti simili. In Windows abbiamo una situazione abbastanza lineare per quanto riguarda la memoria occupata e il tempo, con una crescita dei due valori con l'aumentare della dimensione delle matrici, eccezione fatta per una lieve flessione con la matrice *parabolic_fem*.

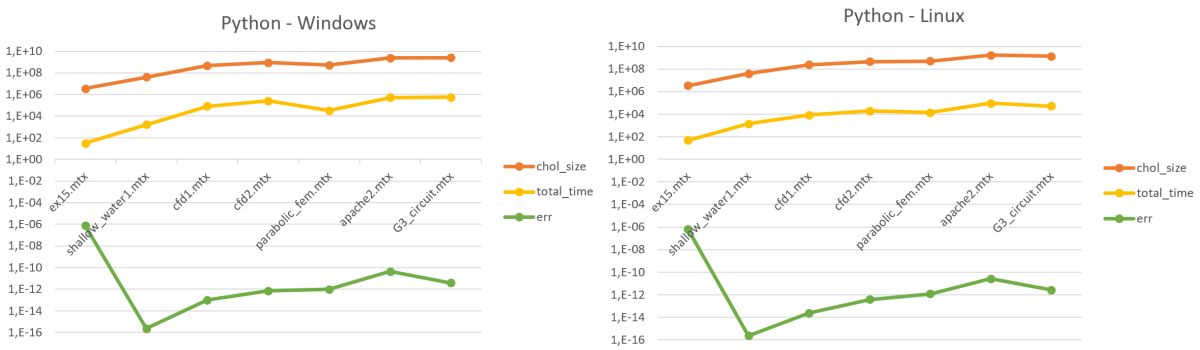
Lo stesso scenario è possibile osservarlo nel grafico riferito al sistema operativo Linux. In entrambi i casi l'errore relativo ha valori altalenanti, è interessante sottolineare che per entrambi i sistemi operativi si ha un picco verso il basso con la matrice *shallow_water1* (errore con ordine di grandezza 10^{-16}) mentre con la matrice *ex15* si ha un errore relativamente alto rispetto alle altre matrici (ordine di grandezza 10^{-6}).

R



In entrambi i casi al crescere della matrice cresce il tempo necessario per calcolare la soluzione finale e la memoria occupata, tranne che per una lieve flessione che si ottiene con la matrice *parabolic_fem*. Per quanto riguarda gli errori relativi si parte in entrambe i casi con un errore relativamente alto (ordine di grandezza 10^{-6}) per la matrice *ex15*, si ha poi un picco verso il basso con la matrice *shallow_water1* (raggiungendo l'ordine di grandezza 10^{-16}), mentre procedendo con matrici più grandi aumenta lievemente l'errore, tranne nel caso di *G3_circuit* dove si ha una lieve flessione.

Python



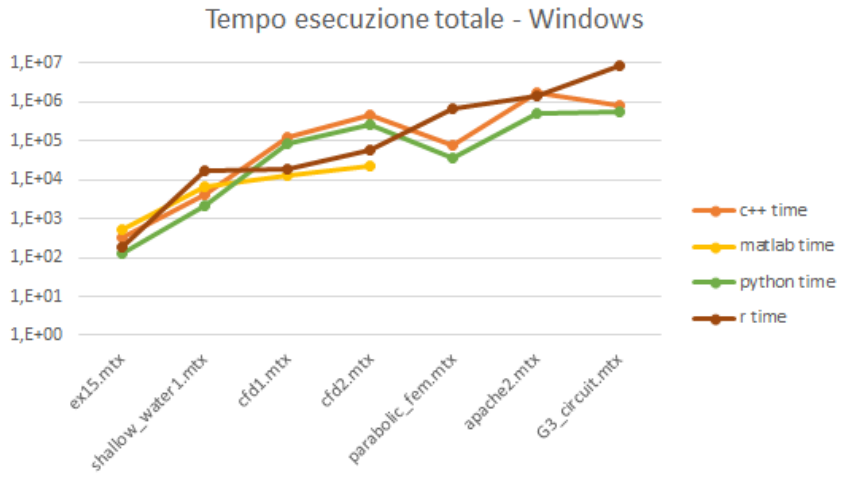
Al crescere della dimensione delle matrici aumenta la memoria occupata e il tempo di risoluzione, con una lieve flessione nel caso della matrice *parabolic_fem*. L'errore relativo ha un massimo con la matrice più piccola (*ex15*), raggiunge poi un minimo con la matrice shallow_water1 ed infine tende lievemente a crescere con l'aumentare della dimensione della matrice, con una flessione finale per la matrice *G3_circuit*.

Conclusioni

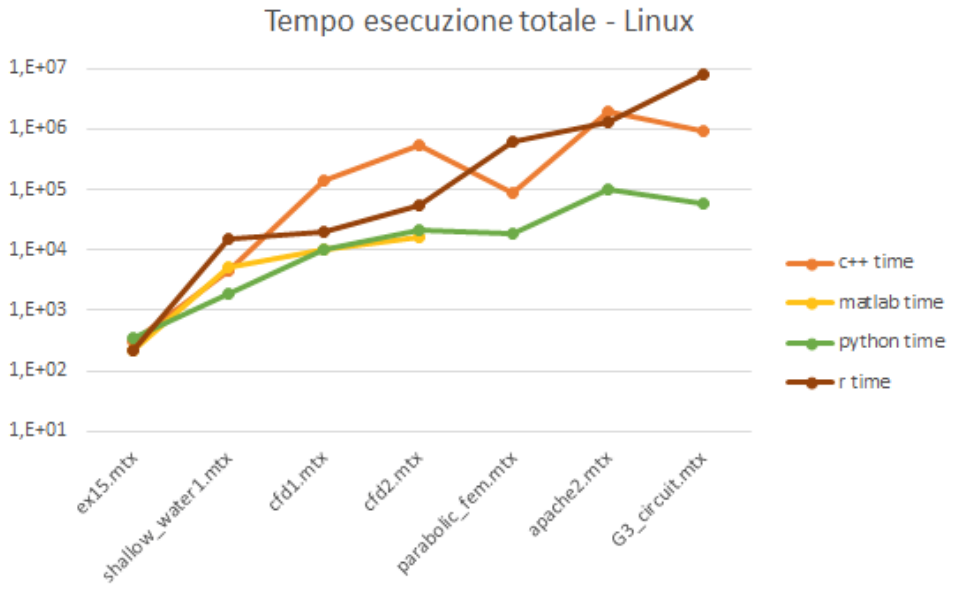
Alla luce di quanto abbiamo appena analizzato e mettendoci nell'ottica proposta all'inizio del progetto, ossia immaginare di dover scegliere per un'azienda un ambiente di programmazione in grado di risolvere con il metodo di Cholesky sistemi lineari con matrici sparse e definite positive di grandi dimensioni, dovendo decidere tra software proprietario (MATLAB) oppure open source e anche tra sistema operativo Windows oppure Linux, possiamo concludere quanto segue.

Come prima cosa è risultato evidente che per eseguire tutte le matrici proposte è necessario disporre di computer molto potenti, indipendentemente dal sistema operativo: non è stato possibile infatti eseguire i codici con le matrici *Flan 1565* e *StocF-1465*, rispettivamente di dimensioni 1.564.794 x 1.564.794 e 1.465.137 x 1.465.137.

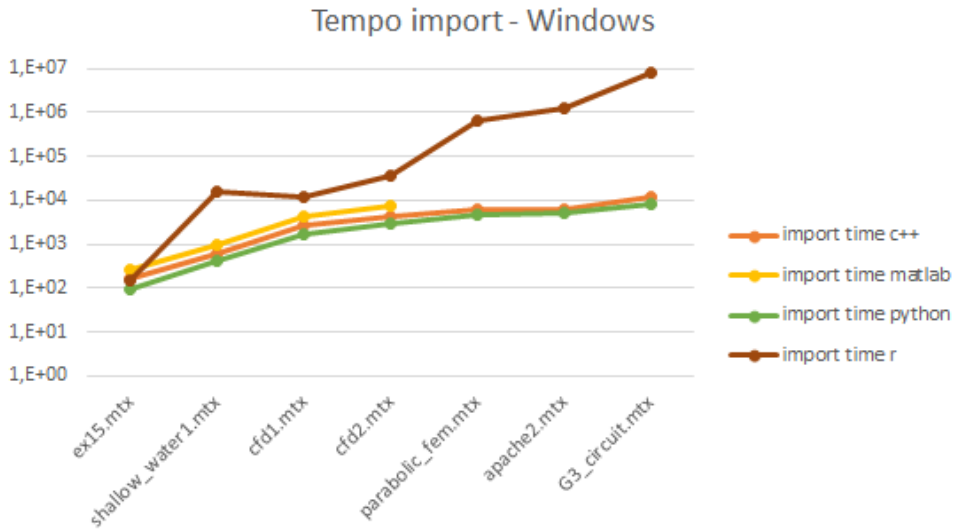
Nel sistema operativo Windows nessuno dei linguaggi analizzati dimostra di riuscire ad eseguire i calcoli in un tempo totale (calcolato come somma del tempo necessario per l'importazione della matrice e del tempo necessario per la decomposizione di Cholesky) particolarmente migliore rispetto agli altri.



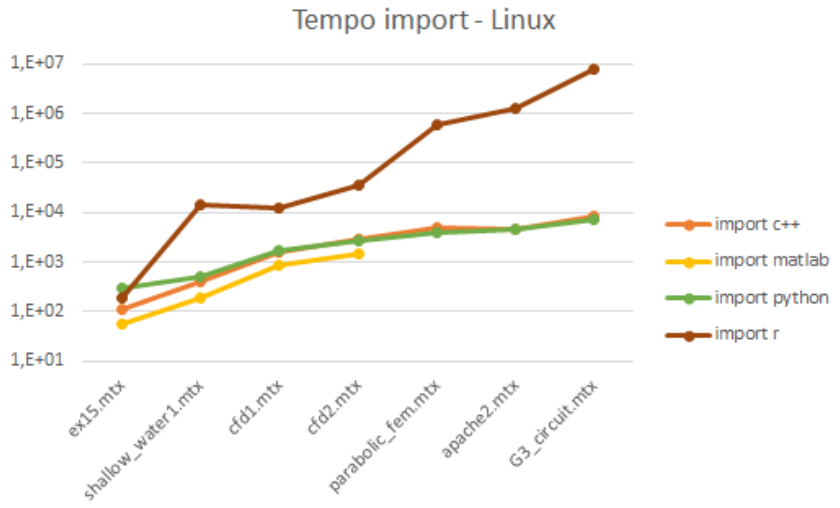
Per quanto riguarda Linux C++ e R risultano avere andamenti simili a Windows. Python, invece, mostra un'esecuzione molto più rapida rispetto a Windows rendendolo il migliore a tutti gli altri linguaggi utilizzati.



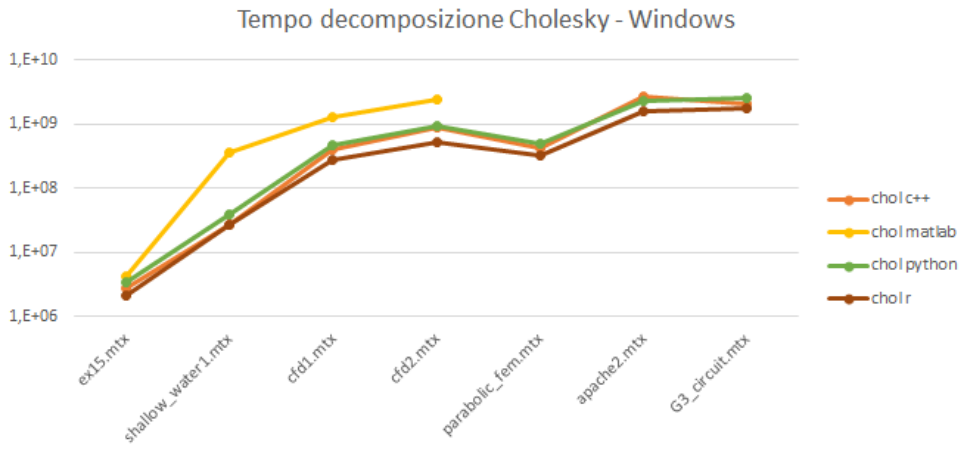
Esaminando però separatamente il tempo di import e il tempo della decomposizione di Cholesky, si può notare che su Windows R impiega molto più tempo per importare le matrici mentre è Python il più veloce.



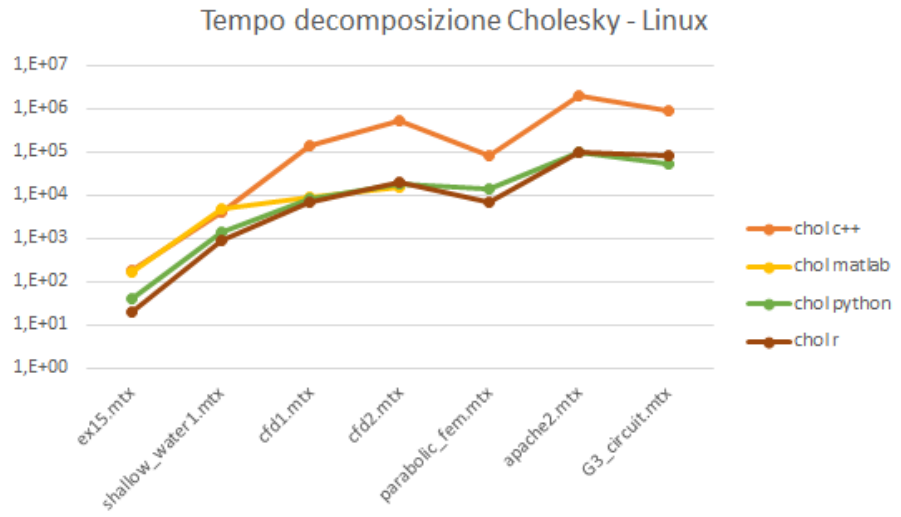
Anche su Linux è R ad avere il peggior tempo di import, mentre in questo caso è MATLAB il più veloce.



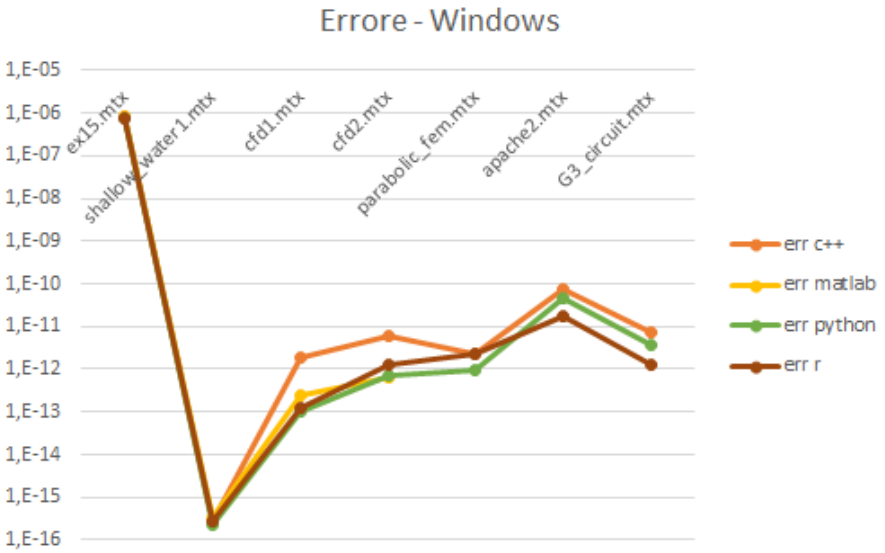
MATLAB è, invece, il più lento per quanto riguarda la decomposizione di Cholesky su Windows, mentre R in questo caso è il più veloce.



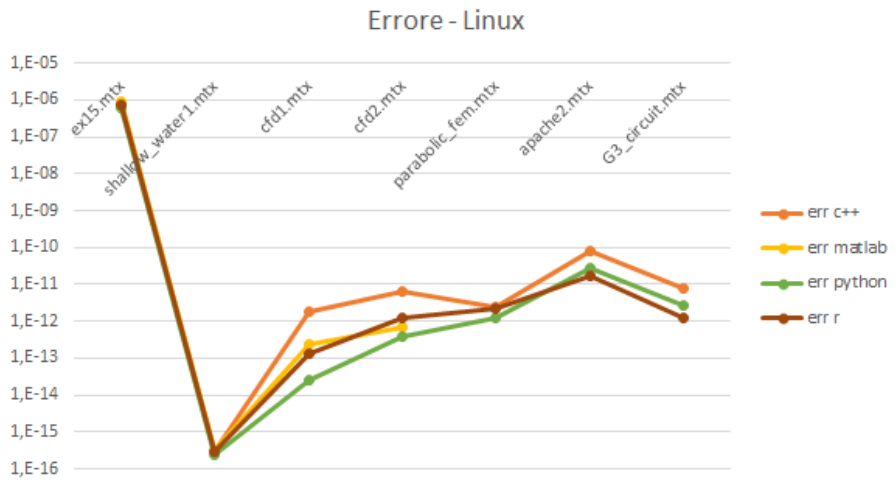
Nel complesso i vari linguaggi hanno tempi con ordini di grandezza minori rispetto a Windows. R e Python hanno i tempi migliori, con valori molto simili. E' importante evidenziare che C++ ha i risultati peggiori, ma migliori rispetto quelli di Windows.



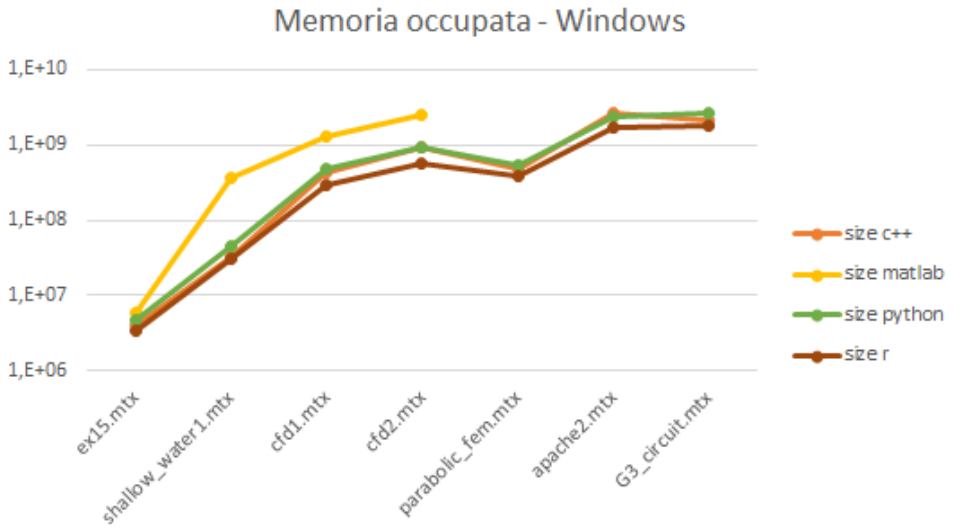
Per quanto riguarda l'accuratezza, in Windows tutti e 4 i linguaggi di programmazione hanno risultati simili, trovano l'errore più grande nella matrice *ex15* e quello più piccolo nella matrice *shallow_water1*.



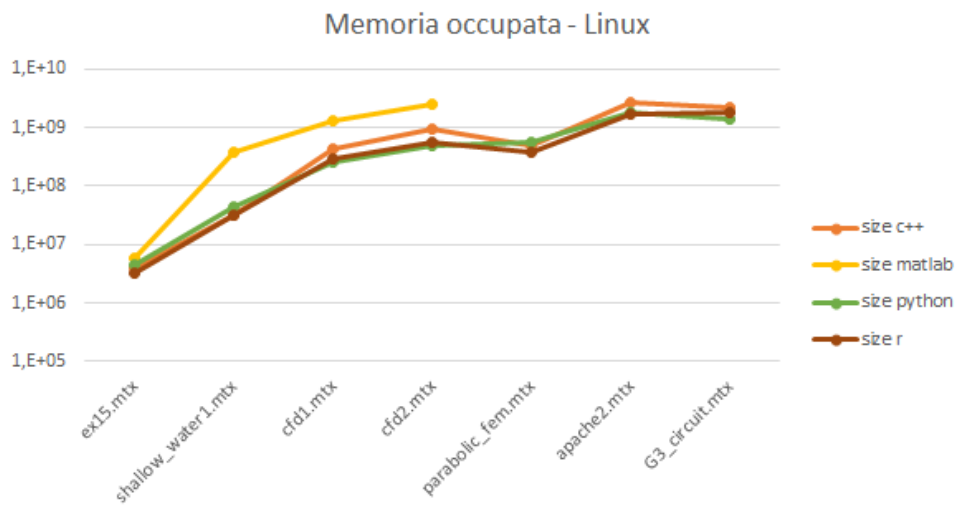
In Linux, si può notare che con Python si hanno errori lievemente di ordine di grandezza minore rispetto agli altri, mentre C++ ha una accuratezza ancora minore come Windows.



Per valutare la memoria occupata dal programma abbiamo sommato la dimensione della matrice importata e la dimensione della decomposizione di Cholesky. MATLAB è il linguaggio di programmazione che, in Windows, occupa più memoria, mentre Python, R e C++ hanno un andamento analogo, con risultati lievemente migliori per R.



Anche su Linux, il linguaggio che occupa più memoria è MATLAB, mentre Python, R e C++ hanno un andamento simile.



Per quanto concerne la facilità d'uso Python è stato il linguaggio che ha causato più difficoltà nell'installazione, sopratutto su sistema operativo Windows, a causa anche dalla poca documentazione reperibile, anche se nella scrittura del codice non si incontrano particolari difficoltà. In C++ è stata utilizzata la libreria Eigen, la quale è molto ben documentata e semplice da importare e utilizzare nel proprio programma. I codici in MATLAB e R sono i più immediati da scrivere, poiché nel primo caso non è necessario effettuare alcuna ricerca di librerie esterne, mentre nel caso di R si ha una documentazione ricca che permette di sfruttare al meglio le librerie disponibili.