Università degli Studi di Milano-Bicocca

**Dipartimento di Informatica, Sistemistica e Comunicazione**

**Corso di Laurea Magistrale in Informatica**

**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 di studiare l’implementazione in ambienti di programmazione open source del metodo di Choleski per la risoluzione di sistemi lineari per matrici sparse, simmetriche e deﬁnite positive, e di confrontarli con l’implementazione di MATLAB, su due sistemi operativi diversi: Windows e Linux.

Per ognuna delle matrici occorre determinare:

* il tempo necessario per calcolare la soluzione x;
* l’errore relativo tra la soluzione calcolata x e la soluzione esatta xe;
* la memoria necessaria per risolvere il sistema, ovvero grosso modo l’aumento della dimensione del programma in memoria da subito dopo aver letto la matrice a dopo aver risolto il sistema.

Per poter compiere questo obbiettivo abbiamo deciso di confrontare MATLAB con gli ambienti open source C++, R e Python.

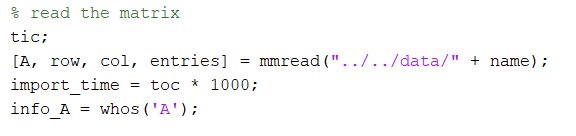
Negli ambienti open source, a differenza di MATLAB che si accorge da solo se la matrice che gli passate è simmetrica e deﬁnita positiva, per poter compiere gli stessi controlli abbiamo dovuto implementare diverse funzioni attraverso l’utilizzo di specifiche librerie.

È possibile trovare il listato dei codici alla seguente repository:

<https://gitlab.com/okamiRvS/cholesky-computing>

**MATLAB**

MATLAB è un ambiente per il calcolo numerico e l’analisi statistica scritto in C. Essendo il programma in grado di manipolare matrici, è riuscito da solo ad accorgersi quando una matrice passata è definita positiva e simmetrica.



Nella scrittura del codice, per prima cosa abbiamo importato le diverse matrici attraverso la funzione *mmread* nel formato *.mtx* in modo da mantenere linearità con gli altri linguaggi.

Attraverso la funzione *tic toc* abbiamo calcolato il tempo di esecuzione di tutto lo script e anche quello di calcolo dei vari elementi del codice.

Prima di eseguire la *decomposizione di Choleski*, abbiamo risolto il sistema in questo modo *b = A \* xe'*, poi attraverso il comando *Chol* eseguito la decomposizione.

Immagine che contiene tenendo, uomo, uccello, giocando

Descrizione generata automaticamente

Una volta eseguita la funzione *chol*, abbiamo calcolato la soluzione finale *x*,

grazie alla quale abbiamo potuto calcolare il tempo di risoluzione, l’errore relativo e la memoria occupata dal calcolo.

Abbiamo inserito anche un *catch exception* in modo da ottenere un commento quando si verifica un problema, questa cosa è stata molto utile in quanto ci siamo che per matrici che superano la grandezza della memoria eil programma va “*out of memory*”.

Infine, abbiamo riportato i risultati ottenuti in un file csv, grazie al quale abbiamo potuto realizzare un grafico che contenesse le tre richieste del progetto e le grandezze delle matrici. Teniamo a sottolineare che come richiesto dal progetto abbiamo eseguito i calcoli su due sistemi operativi differenti, Windows e Linux.

**C++**

La decomposizione di Cholesky è stata effettuata sfruttando Eigen, libreria template per 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 individuate.

Il primo problema affrontato è stata l’importazione in memoria dal formato .mtx. Nonostante esistano delle funzioni di import reperibili online ([*https://math.nist.gov/MatrixMarket/mmio-c.html*](https://math.nist.gov/MatrixMarket/mmio-c.html)) abbiamo deciso di scrivere un parser ad-hoc per matrici Matrix Market. Tale soluzione ha permesso di ottenere risultati migliori nei tempi di import rispetto agli altri linguaggi.

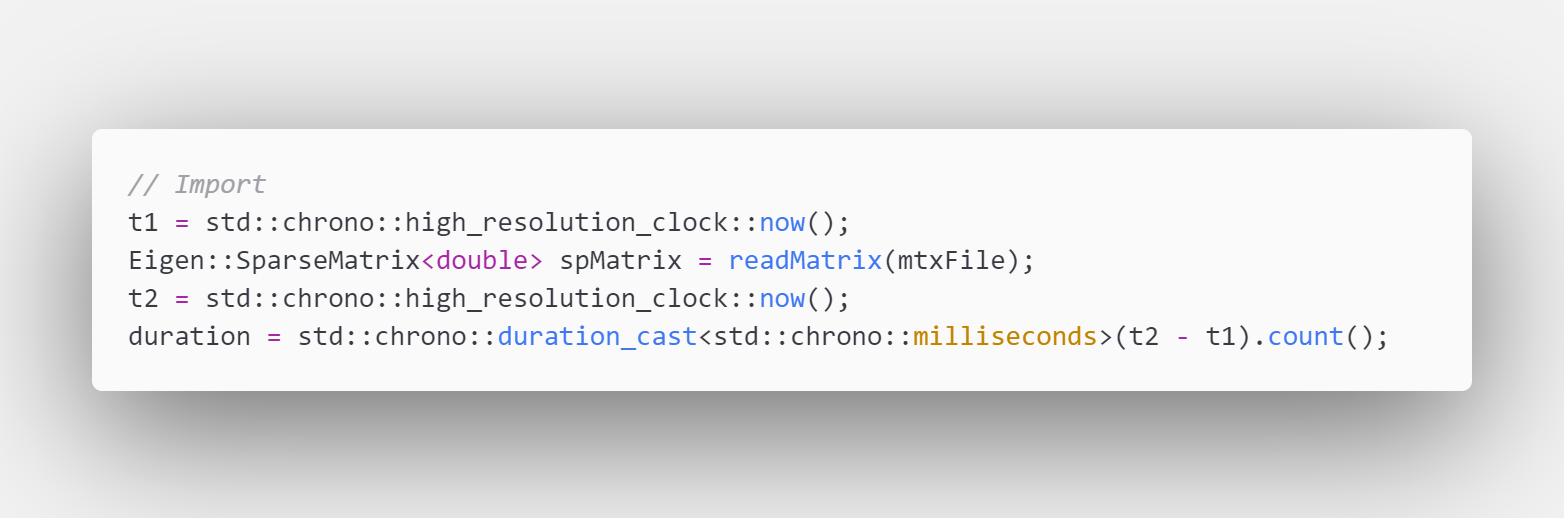


Figura 1: Import tramite funzione custom Eigen::SparseMatrix<double> readMatrix(std::string &filename);

C++ non dispone di una funzione per controllare lo spazio in memoria occupato da un oggetto. Dunque 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.

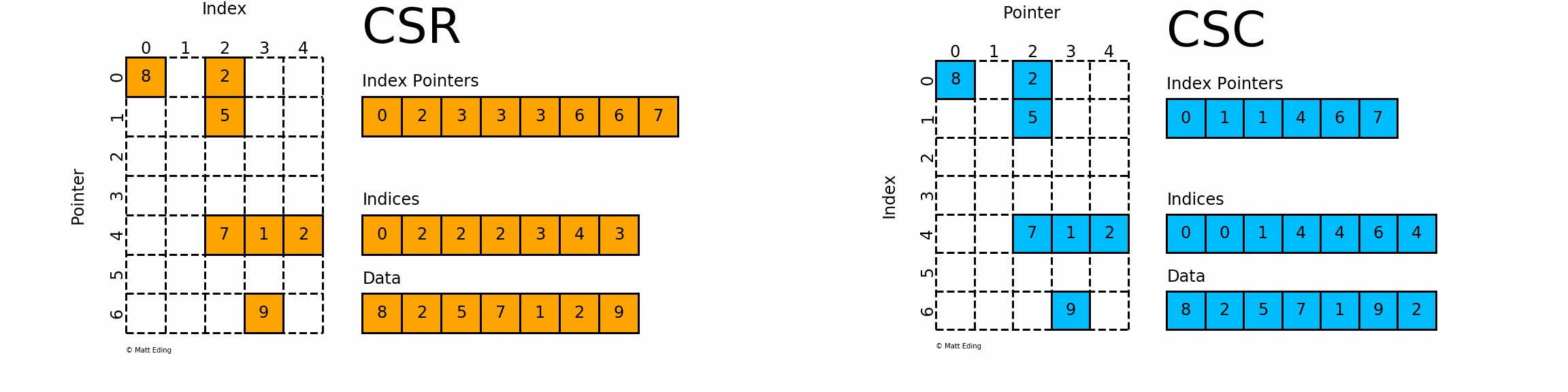


Figura 2: formato CSR e CSC

Il calcolo è quindi il seguente:

C:\Users\Federico\Downloads\CodeCogsEqn (2).png

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 utilizzati nell’analisi via 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.

I risultati sono riportati in un file csv.

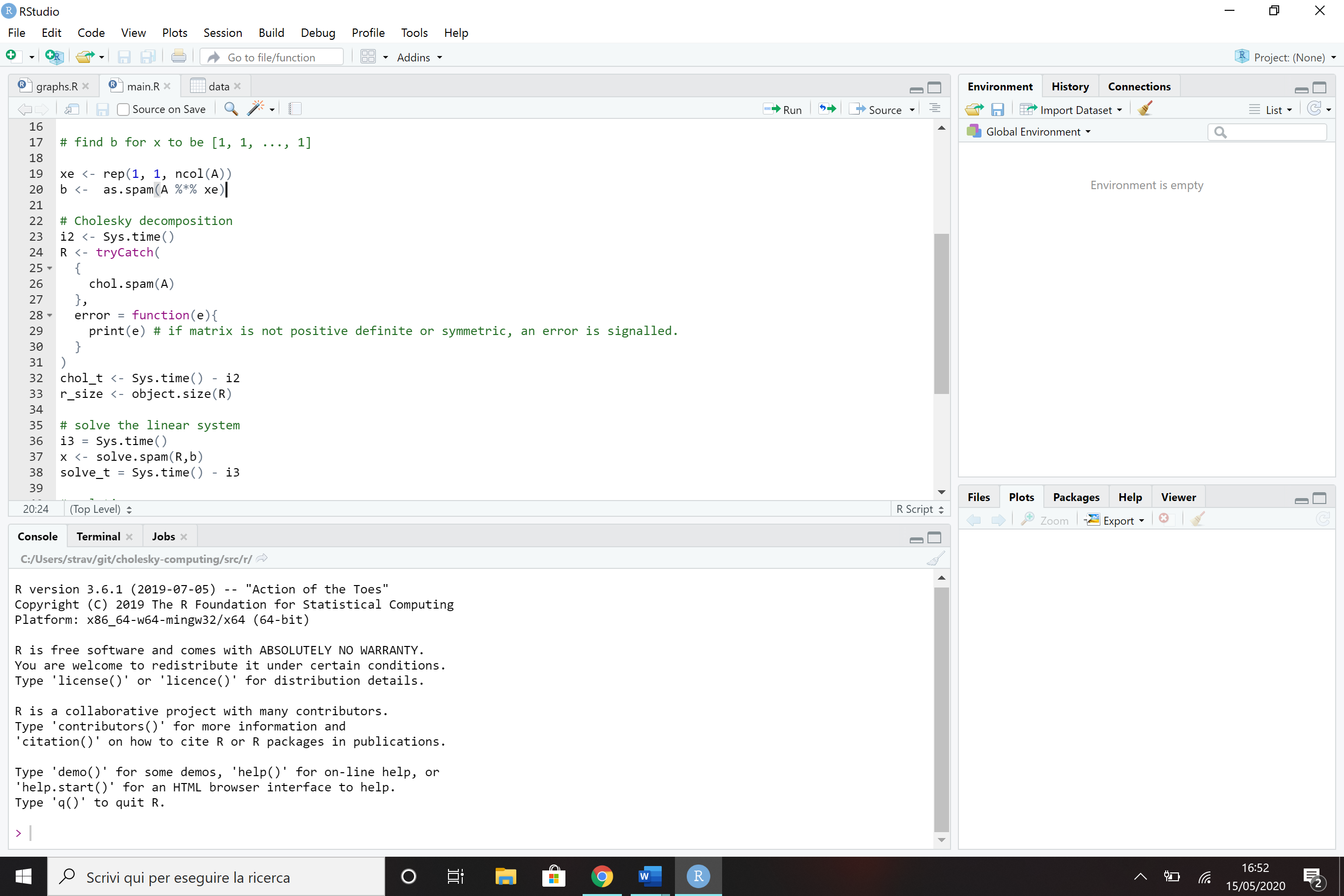
**R**

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, dopo aver riscontrato che con essa non era possibili leggere matrici particolarmente grandi, abbiamo deciso di ricorrere alla libreria *spam*. Come riporta la documentazione ufficiale, infatti, questa libreria è “veloce e scalabile, con il pacchetto di estensione *spam64*” e implementa la funzione che permette di effettuare la decomposizione di Choleksy.

Importiamo la matrice con la funzione *read.MM*, che ci permette di salvarla in formato sparso.

Grazie al 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 se queste non sono rispettate, motivo per il quale è stato inserito un try-catch.



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* (dato R risultato di *chol.spam*, calcoliamo *backsolve(R, forwardsolve(t(R), b))* ).

Abbiamo tracciato l’andamento della memoria usata per la decomposizione di Choleksy calcolando la dimensione della matrice originale e della matrice decomposta, tramite *object.size*, e abbiamo misurato il tempo necessario per le operazioni fondamentali di decomposizione e risoluzione con *sys.time*. Infine, è stato calcolato l’errore relativo, considerando come soluzione esatta il vettore con tutte le componenti pari a 1.

**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 quasi tutte 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](https://github.com/scikit-sparse/scikit-sparse/issues) ufficiale. Purtroppo, il materiale di riferimento è povero e poco chiaro, gli stessi creatori sottolineano che si tratta di un progetto “[rotto](https://github.com/scikit-sparse/scikit-sparse/issues/55#issuecomment-498811801)” per quanto concerne le dipendenze.

È stato possibile installare la libreria su una macchina virtuale con *os xubuntu*. Va fatto presente che un nuovo problema in “corso d’opera” si è manifestato in quanto sembra come se è stato eliminato il pacchetto dei requisiti.

Il calcolo è eseguito nel seguente modo, se A è una matrice sparsa, simmetrica e definita positiva, e b è una matrice o vettore (sparso o denso) allora il codice seguente risolve l’equazione Ax=b



**ANALISI DEI RISULTATI**

Per poter analizzare i risultati partiamo confrontando i grafici di esecuzione dei vari linguaggi nei due sistemi operativi.

L’esecuzione è avvenuta su un unico pc provvisto di Windows e di una macchina virtuale per eseguire Linux.

I grafici scelti sono a linee con indicatori, dove sulle ascisse troviamo le matrici ordinate in base alla loro grandezza e sulle ordinate troviamo le tre grandezze richieste dal progetto:

* *chol\_size* = memoria occupata dopo la risoluzione di x;
* *sol\_time* = tempo necessario per calcolare la soluzione x;
* *err* = errore relativo.

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

**MATLAB**

Analizziamo i grafici delle esecuzioni di MATLAB su Windows e Linux:

La prima differenza che possiamo notare tra le due esecuzioni è che mentre siamo riusciti ad eseguire la risoluzione del sistema lineare fino ad *Apache2* su Windows, a Linux ci siamo fermati *cfd2*.

Questo risultato potrebbe essere sicuramente influenzato dal fatto che per Linux abbiamo utilizzato una macchina virtuale, ma in ogni caso come vedremo più avanti abbiamo notato che MATLAB tende ad avere un utilizzo sproporzionato della *RAM* e ad andare facilmente “*out of memory*” a differenza di altri linguaggi *open source*.

Analizzando i dati ci accorgiamo che entrambe le esecuzioni al crescere delle matrici aumento il loro tempo di esecuzione e il loro spazio occupato, ma mentre su Linux si arriva al massimo ad occupare uno spazio di memoria nell’ordine di grandezza di 108 su Windows si occupa più memoria, circa nell’ordine di 1010. Questo fa sì che il tempo di esecuzione di Windows sia maggiore rispetto a quello di Linux.

Per quanto riguarda l’errore relativo in entrambi i casi si tende ad avere uno stesso andamento con risultati poco migliori in Windows. È interessante sottolineare il picco che entrambe le esecuzioni hanno nell’esecuzione della matrice *shallow\_water1*, dove in tutti e due i casi si ottiene l’errore più piccolo nell’ordine di grandezza di 10-16.

**C++**

Analizziamo i grafici delle esecuzioni di C++ su Windows e Linux:

Nell’esecuzione su C++ possiamo sottolineare un aspetto di cui parlavamo prima, con lo stesso computer e quindi a parità di caratteristiche e prestazioni di memoria, con un linguaggio open source siamo riusciti ad eseguire matrici anche più grandi di *Apache2*.

In questo caso notiamo andamenti abbastanza differenti sotto molti punti di vista delle due esecuzioni.

In Windows sotto l’aspetto della memoria occupata e del tempo abbiamo una situazione abbastanza lineare, che porta ad una crescita dei due fattori al crescere delle matrici, fatta eccezione per una lieve flessione con la matrice *parabolic\_fem*.

Nell’esecuzione di Linux è particolare sottolineare un picco che sia nella memoria utilizzata che nel tempo di esecuzione, fattore che porta ad avere anche un errore elevato.

Per quanto riguarda l’errore relativo in entrambi i casi abbiamo valori altalenanti, detto già di Linux, è interessante sottolineare il picco presente in Windows con la matrice *shallow\_water1* che produce un errore molto piccolo (misura di grandezza 10-16) e con la matrice *ex15* un errore abbastanza alto (10-3).

**R**

Analizziamo i grafici delle esecuzioni di R su Windows e Linux:

Confrontando i grafici delle due esecuzioni possiamo parlare di risultati “visivamente” simili, che variano solo nelle unità di grandezza.

In entrambi i casi al crescere della matrice cresce il tempo per calcolare la soluzione finale e la memoria occupata, tranne per una lieve flessione con la matrice *parabolic\_fem*.

Per quanto riguarda gli errori relativi, si parte in entrambe i casi con un errore “abbastanza alto” (10-6) con la matrice *ex15*, si ha poi un picco con la matrice *shallow\_water1* raggiungendo un valore molto basso (ordine di grande 10-16). Procedendo con matrici più grandi, aumenta lievemente l’errore, tranne nel caso di *G3\_circuit* dove abbiamo una live flessione.

**Python**

Abbiamo eseguito anche una risoluzione del sistema, ovviamente solo su ambiente Linux, tramite Python.

I risultati ottenuti sembrano apparentemente simili a quelli analizzati in precedenza di R.

Al crescere della grandezza delle matrici cresce la memoria occupata e il tipo di risoluzione, con una lieve flessione con la matrice *parabolic\_fem*.

L’errore relativo ha un massimo con la matrice più piccola (ex15), raggiunge poi un minimo con la matrice *shallow\_water1*, tende poi lievemente a crescere al crescere della matrice, trovando una flessione finale con la matrice *G3\_circuit*.

**CONCLUSIONI**

Alla luce di quanto abbiamo appena analizzato e mettendoci nell’ottica proposta all’inizio del progetto “Immaginate che la vostra azienda abbia la necessità di munirsi di un ambiente di programmazione per risolvere con il metodo di Choleski sistemi lineari con matrici sparse e deﬁnite positive di grandi dimensioni. L’alternativa `e tra software proprietario (MATLAB) oppure open source e anche tra Windows oppure Linux” possiamo concludere che:

* Per riuscire ad eseguire tutte le matrici proposte bisogna disporre di computer molto potenti,
* Dai risultati ottenuti con il nostro computer risulta più utile utilizzare codici open source piuttosto che MATLAB, ci garantisce un utilizzo migliore della memoria;
* C++ risulta avere dei risultati altalenanti, nonostante su Windows sembra utilizzi la memoria in modo più efficiente;
* R e Python sono i due linguaggi che risultano avere i risultati più lineari.