Confronto implementazioni Metodo di Cholesky
Attraverso l'utilizzo di ambienti differenti: Matlab, C++ e R

`https://github.com/nicoripa/MCS_Project/`

Lorenzo Rovida, Nicolò Ripamonti
l.rovida1@campus.unimib.it, n.ripamonti@campus.unimib.it
817151, 816171

Dipartimento di Informatica, Sistemi e Comunicazione, Universitá degli Studi di Milano-Bicocca, Milano, Italia

---

Abstract

Lo scopo di questo progetto è di studiare l'implementazione in ambienti di programmazione open source del metodo di **Cholesky** per la risoluzione sistemi lineari per matrici sparse, simmetriche e definite positive e successivamente confrontarli con l'implementazione nell'ambiente di programmazione di Matlab.

Immaginiamo che un'azienda abbia la necessità di munirsi di un ambiente di programmazione per risolvere sistemi lineari con matrici sparse e definite positive di grandi dimensioni utilizzando il metodo di Cholesky.

L'alternativa è tra software proprietario (Matlab) oppure open source e anche tra Windows oppure Linux.

La richiesta è quindi quella di confrontare, in ambiente Linux e Windows, e sulla stessa macchina, l'ambiente di programmazione Matlab con una libreria open source.

In altre parole bisogna porsi le seguenti domande:

1. È meglio affidarsi alla sicurezza di Matlab pagando oppure vale la pena di avventurarsi nel mondo open source?
2. È meglio lavorare in ambiente Linux oppure in ambiente Windows?

---

1. Introduzione

Lo scopo di questo progetto è quello di valutare l'implementazione, in ambienti di programmazione differenti, del metodo di Cholesky per la risoluzione di sistemi lineari per matrici sparse. La scelta degli ambienti, da parte del team, si è focalizzata su:

Matlab - C++ - R

Le matrici che sono state considerate per la realizzazione del progetto sono quelle del gruppo *SuiteSparse Matrix Collection* che colleziona matrici sparse derivanti da applicazioni di problemi reali (ingegneria strutturale, fluidodinamica, elettromagnetismo, termodinamica, computer graphics/vision, network e grafi).

In particolare le matrici simmetriche e definite positive che sono state analizzate sono le seguenti:

- **Flan_1565**
- **StocF-1465**
- **cfd2**
- **cfd1**
- **G3_circuit**
- **parabolic_fem**
- **apache2**
- **shallow_water1**
- **ex15**

Un sistema lineare si può rappresentare in forma matriciale come $A \cdot x = b$ dove:

- A è la matrice dei coefficienti del sistema ed è rappresentata da una delle matrici del gruppo *SuiteSparse Matrix Collection* elencate precedentemente.
- x è il vettore colonna delle incognite.

- b è il vettore colonna dei termini noti; b è scelto in modo che la soluzione esatta sia il vettore $xe = [111\ldots11]$ avente tutte le componenti uguali a 1, tale che $b = A \cdot xe$.

1.1. Matrici sparse

Le matrici sparse sono una classe di matrici con la caratteristica di contenere un significativo numero di elementi uguali a zero. Spesso il numero di elementi diversi da zero su ogni riga è un numero piccolo (per esempio dell'ordine di 10^1) indipendente dalla dimensione della matrice, che può essere anche dell'ordine di 10^8.

Le matrici sparse si possono memorizzare in modo compatto, tenendo solo conto degli elementi diversi da zero; per esempio, è sufficiente per ogni elemento diverso da zero memorizzare solo la sua posizione ij e il suo valore a_{ij}, ignorando gli elementi uguali a zero. Ciò consente di ridurre il tempo di calcolo eliminando operazioni su elementi nulli.

1.2. Calcolatore utilizzato

Per effettare i diversi calcoli è stato utilizzato un Surface Book 2 con le seguenti caratteristiche hardware:

- Intel Core i5-7300U @ 2.60GHz 2.71GHz
- CPU x64
- 8.00 GB RAM
- Intel HD Graphics 620

Il sistema operativo installato e utilizzato per Matlab, C++ e R è stato Windows 10, successivamente è stato incluso nel confronto lo stesso codice C++ lanciato su Ubuntu 20.04 LTS installato su WSL (Windows Subsystem Linux).

In dettaglio sono stati utilizzati:

- MATLAB R2020a
- g++ (i686-posix-dwarf-rev0, Built by MinGW-W64 project) 8.1.0 (Ambiente Windows)
- RStudio (basato su R versione 3.6.2)
- g++ (Ubuntu 9.3.0-10ubuntu2) 9.3.0 (Ambiente Ubuntu)

2. Ambienti di sviluppo

Il confronto dei vari ambienti di programmazione sui sistemi operativi di Windows e Linux deve avvenire in termini di:

Tempo: Calcolo dei tempi di esecuzione del metodo di Cholesky;

Accuratezza: Calcolo dell'errore assoluto tra la soluzione esatta e la soluzione del sistema lineare;

Impiego della memoria: Quantità di memoria utilizzata per il calcolo del sistema lineare con il metodo di Cholesky.

2.1. Matlab

Versione: 9.7.0

Referenze: Sito - Documentazione

Matlab (abbreviazione di Matrix Laboratory) è un ambiente per il calcolo numerico e l'analisi statistica scritto in C, che comprende anche l'omonimo linguaggio di programmazione creato dalla MathWorks. MATLAB consente di manipolare matrici, visualizzare funzioni e dati, implementare algoritmi, creare interfacce utente, e interfacciarsi con altri programmi. Matlab è usato da milioni di persone nell'industria e nelle università per via dei suoi numerosi strumenti a supporto dei più disparati campi di studio applicati e funziona su diversi sistemi operativi, tra cui Windows, Mac OS, GNU/Linux e Unix.

2.2. Eigen, C++

Versione: 3.7.7, Novembre 2019

Referenze: Sito - Documentazione

Eigen è una libreria di modelli C++ per l'algebra lineare, ovvero che supporta il calcolo e la risoluzione di matrici, vettori, solutori numerici e algoritmi correlati.

Supporta matrici di qualsiasi dimensione, dalle piccole matrici di dimensioni fisse alle matrici dense arbitrariamente grandi, fino alle matrici sparse e il suo ecosistema offre molte funzionalità specializzate come l'ottimizzazione non lineare, le funzioni di matrice, un solutore polinomiale e molto altro.

"L'implementazione di un algoritmo su Eigen è come copiare semplicemente lo pseudocodice."

Eigen ha un buon supporto per il compilatore mentre si esegue una suite di test per garantire affidabilità e aggirare eventuali bug del compilatore. Eigen è anche standard C ++ 98 e mantiene tempi di compilazione molto ragionevoli.

Eigen è un software open source concesso in licenza con Mozilla Public License 2.0 dalla versione 3.1.1.

In particolare di Eigen sono state utilizzate le funzioni:

`SimplicialLDLT` Questa classe fornisce fattorizzazioni LDL^T Cholesky di matrici sparse che sono definite positive. La fattorizzazione consente di risolvere $AX = B$ dove X e B possono essere densi o sparsi. Al fine di ridurre il riempimento, viene applicata una permutazione simmetrica P prima della fattorizzazione in modo tale che la matrice fattorizzata sia PAP^{-1};

`solve()` Funzione utilizzata per trovare la soluzione di x.

2.3. Matrix, R

Versione: 1.2-18, Novembre 2018

Referenze: Sito - Documentazione

Il pacchetto di R *Matrix* fornisce un set di classi per matrici dense e sparse che estendono le classi base delle matrici già presenti in R. I metodi per un'ampia gamma di funzioni e operatori applicati agli oggetti di queste classi forniscono un accesso efficiente alla soluzione di sistemi lineari, matrici dense e matrici sparse.

Una caratteristica notevole del pacchetto è che ogni volta che una matrice viene fattorizzata, la fattorizzazione viene memorizzata come parte della matrice originale in modo che ulteriori operazioni sulla matrice possano riutilizzare questa fattorizzazione.

In particolare del pacchetto Matrix, sono state usate due funzioni:

`readMM()` Funzione utilizzata per leggere un file di tipo MatrixMarket. All'interno della funzione basta inserire il percorso del file;

`chol()` Funzione che calcola la fattorizzazione di Choleski di una matrice quadrata simmetrica e definita positiva. Gli argomenti che si possono inserire all'interno della funzione sono:

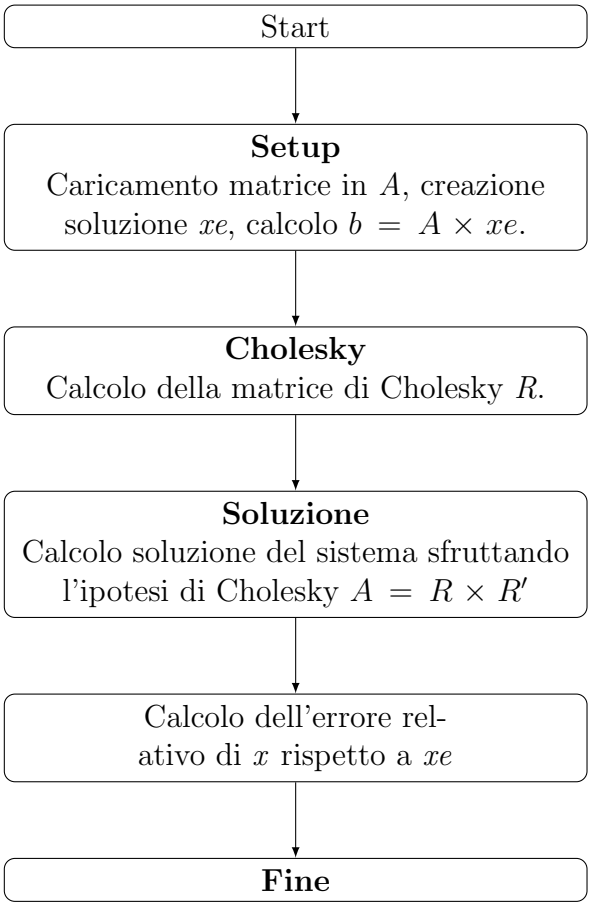
x Una matrice quadrata (sparsa o densa); se x non è definito positivo, viene segnalato un errore. per la nostra implementazione abbiamo utilizzato come argomento della funzione solamente la matrice;

pivot Valore logico che indica se si deve usare il pivot. Attualmente, questo non viene utilizzato per matrici dense.

cache Valore logico che indica se il risultato deve essere memorizzato nella cache; si noti che questo argomento è sperimentale e disponibile solo per alcune matrici sparse.

3. Implementazioni

L'implementazione del metodo è stata così suddivisa:



Le fasi segnate in grassetto verranno cronometrate dal sistema, da notare come l'ultima parte (il calcolo dell'errore) non venga inclusa nella misurazione, essa infatti non farà parte di un ipotetico codice definitivo (una volta scelto il linguaggio non avrà più senso calcolare l'errore, e nemmeno cronometrare le funzioni tra l'altro).

3.1. Matlab

L'implementazione su ambiente Matlab è stata effettuata tramite questo semplice script:

```
tic

%Leggo la matrice A
A = Problem.A;

%Creo soluzione che e' un vettore lungo righe di A con tutti 1
xe = ones(size(A,1), 1);

%Calcolo b dalla equazione Axe = b
b = A * xe;

disp('Time setup:');
toc
tic

%Calcolo Cholesky triangolare inferiore della matrice A
R = chol(A);

disp('Time Cholesky:');
toc
tic

%{
Siccome per ipotesi di Cholesky A = R * R',
allora risolvere Ax = b diventa R * R' * x = b,
quindi x = R \ (R' \ b)
%}

y = R' \ b;
x = R \ y;

disp('Time resolve:');
toc

%Calcolo errore fra x e xe
errore = norm(x - xe) / norm(xe);

disp('Errore: ');
disp(errore);
```

da notare l'utilizzo dei comandi `tic` e `toc` che vengono usati per calcolare il tempo che trascorre per effettuare determinati blocchi di codice.

3.2. R

La soluzione per ambiente R è data dal seguente script:

```
library(Matrix)

start_time_setup <- Sys.time()
A = readMM("../Matrici/ex15.mtx")

xe = rep(1, times = nrow(A))

b = A %*% xe
end_time_setup <- Sys.time()

start_time <- Sys.time()
R = chol(A)
end_time <- Sys.time()

Memory = object.size(chol(A))

start_time_solving <- Sys.time()
y = solve(t(R), b)

x = solve(R, y)
end_time_solving <- Sys.time()

xe = as.matrix(xe)

Errore = norm(x - xe) / norm(as.matrix(xe))

format(Errore, scientific = TRUE)

Time_Setup = end_time_setup - start_time_setup
Time_Chol  = end_time - start_time
Time_Solving = end_time_solving - start_time_solving
```

Una differenza sostanziale con il codice Matlab è data dalla presenza di una variabile che contiene la memoria utilizzata, molto utile per calcolare la memoria utilizzata dallo script (per gli altri ambienti abbiamo utilizzato tool del sistema operativo per controllare la memoria).

3.3. C++

Il file *script.cpp* è costruito sul seguente codice:

```
#include <iostream>
#include <Eigen/Dense>
#include <unsupported/Eigen/SparseExtra>
#include <Eigen/SparseCholesky>
#include <chrono>

using namespace std;
using namespace Eigen;
using namespace chrono;

int main(int argc, char** argv) {

  cout << endl;

  for (int i = 1; i < argc; ++i) {

    cout << "Cholesky con " << argv[i] << endl;
    //Salvo il tempo corrente
    steady_clock::time_point begin = steady_clock::now();

   //Dichiaro una matrice sparsa di double chiamata A
    SparseMatrix<double> A;

    //Carico su A la matrice *it corrente data dall'iteratore
    loadMarket(A, argv[i]);

    //Creo vettore riga lungo righe di A composto da soli uno
    VectorXd xe = VectorXd::Constant(A.rows(), 1);

    //Calcolo b come prodotto fra A e xe (selfadjointView<Lower>
    //specifica che la matrice A è definitia
    //solo inferiormente (essendo simmetrica))
    VectorXd b = A.selfadjointView<Lower>() * xe;

    steady_clock::time_point end = steady_clock::now();
```

```
    //Stampo il tempo trascorso
    cout << "Time Setup = "
         << duration_cast<nanoseconds>(end - begin).count()
         << "ns" << endl;

    begin = steady_clock::now();


    //Calcolo la Cholesky con la funzione SimplicialLDLT
    SimplicialLDLT<SparseMatrix<double>> chol(A);

    end = steady_clock::now();

    //Stampo il tempo trascorso
    cout << "Time Cholesky = "
         << duration_cast<nanoseconds>(end - begin).count()
         << "ns" << endl;

    begin = steady_clock::now();
    //Trovo la soluzione x con la funzione solve
    VectorXd x = chol.solve(b);
    end = steady_clock::now();

    //Stampo il tempo trascorso
    cout << "Time Resolve = "
         << duration_cast<nanoseconds>(end - begin).count()
         << "ns" << endl;


    //Calcolo errore relativo
    double relative_error = (x - xe).norm() / (xe).norm();

    cout << "Relative error = " << relative_error << endl;

    cout << endl << "****" << endl << endl;
  }
}
```

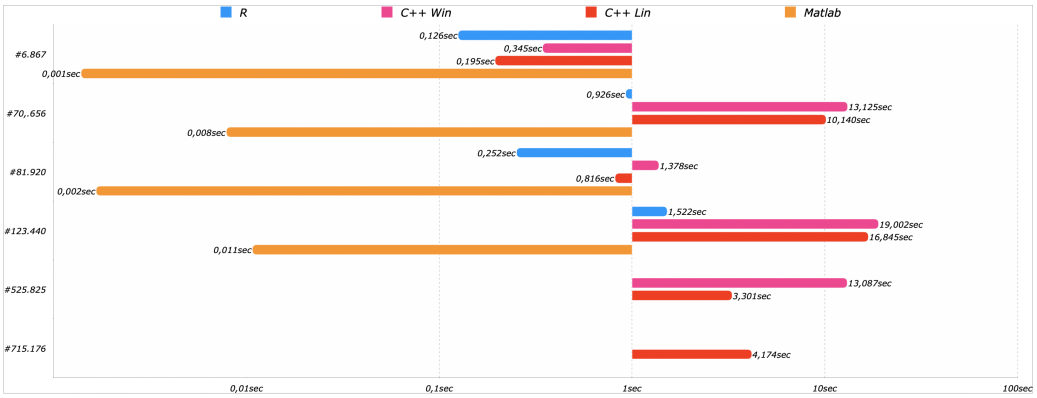
Come specificato in precedenza, questo script verrà lanciato sia su Windows che su WSL Ubuntu 20.04.

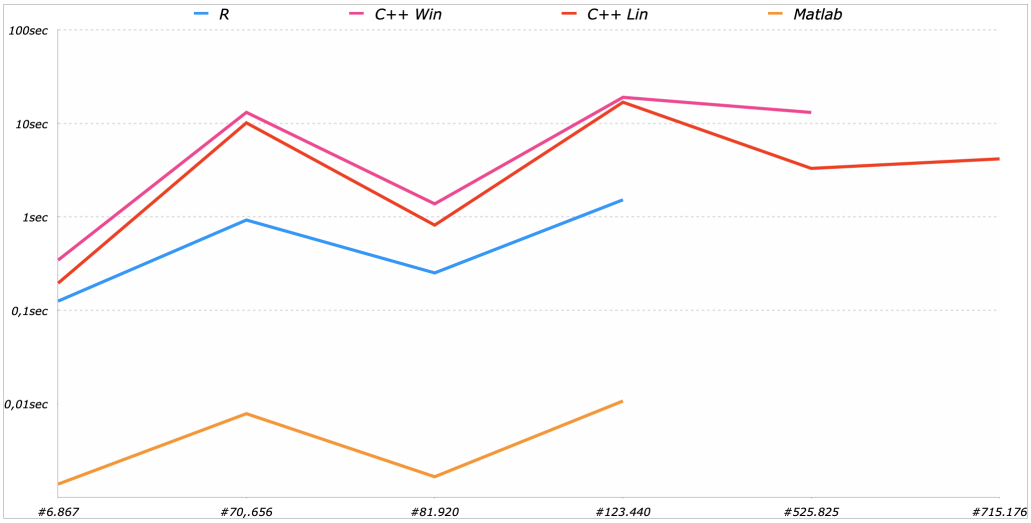
4. Elaborazioni e risultati

Nella sezione corrente verranno presentati i risultati delle varie elaborazioni su grafici. Essi conterranno valori per matrici fino a 715.176 righe (che sarebbe *apache2.mtx*) in quanto per matrici più grandi il calcolatore non riesce a concludere il calcolo.

4.1. Time setup

La prima parte dell'analisi ha riguardato il tempo di setup, come specificato all'inizio della sezione 3. In breve è il tempo di caricamento della matrice A, dell'inizializzazione di xe e del calcolo di b

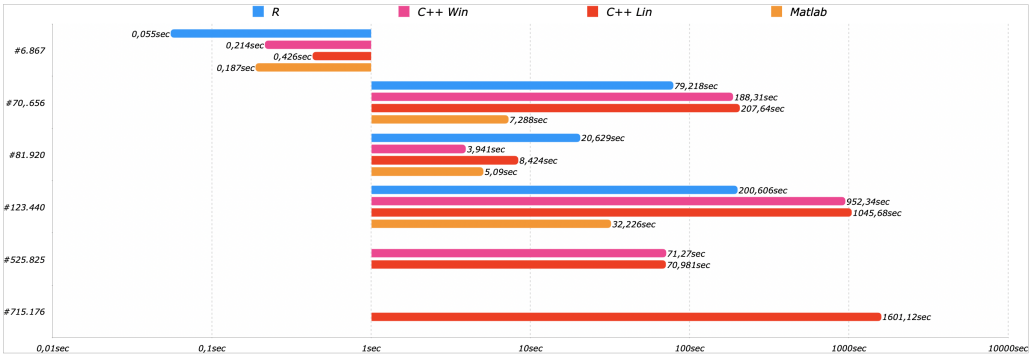


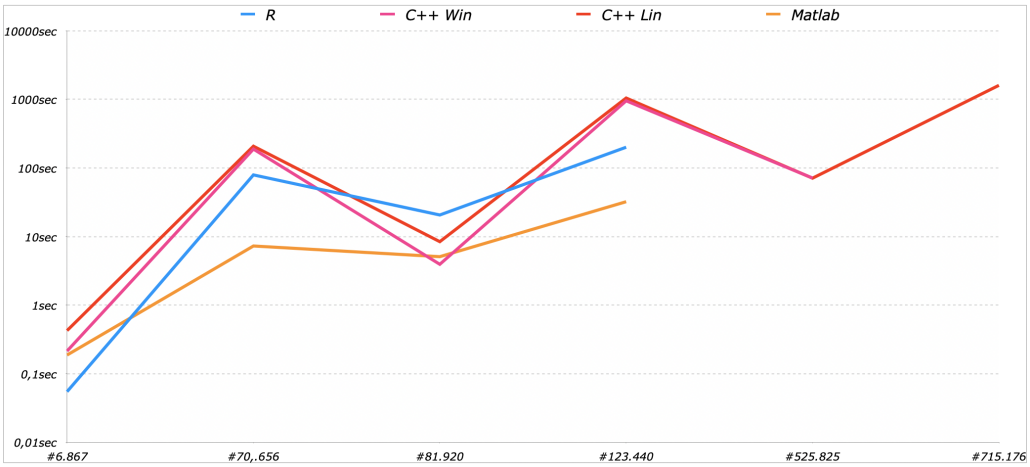


come si può facilmente osservare chi impiega il minor tempo è Matlab, a seguire (molto distanti) rispettivamente R, C++ su Win e C++ su Linux.

Da notare però come Matlab ed R non riescano a gestire matrici con numero di righe $\geq$ 525.825, cosa che C++ riesce a fare. Addirittura su ambiente linux il sistema riesce anche a gestirne una da 715.176.

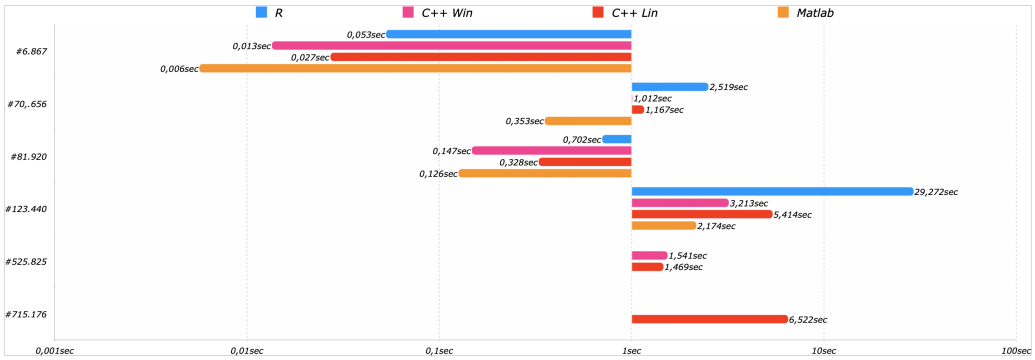
4.2. Time Cholesky

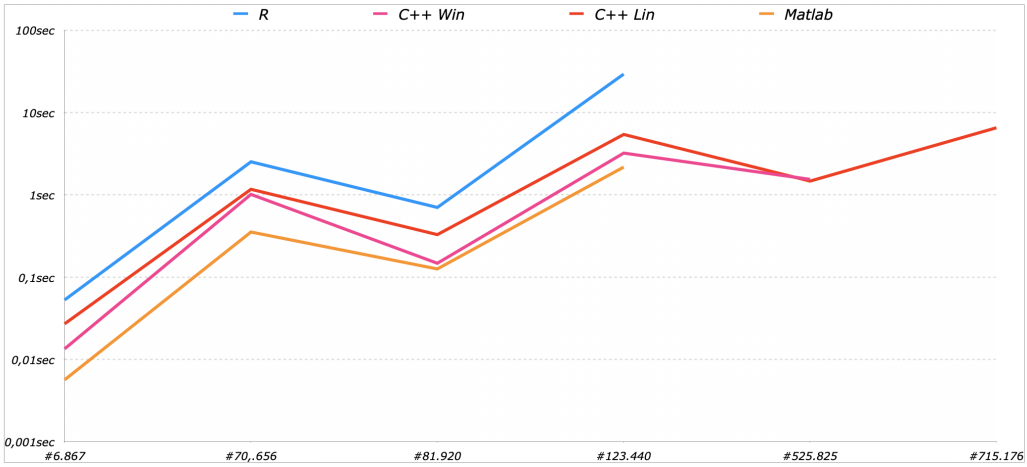




Questi grafici rappresentano il tempo forse più significativo, ovvero il tempo per il calcolo della matrice di Cholesky R. Matlab si conferma l'ambiente tendenzialmente più rapido, anche se la differenza non sembra essere così significativa (nonostante ci sia).

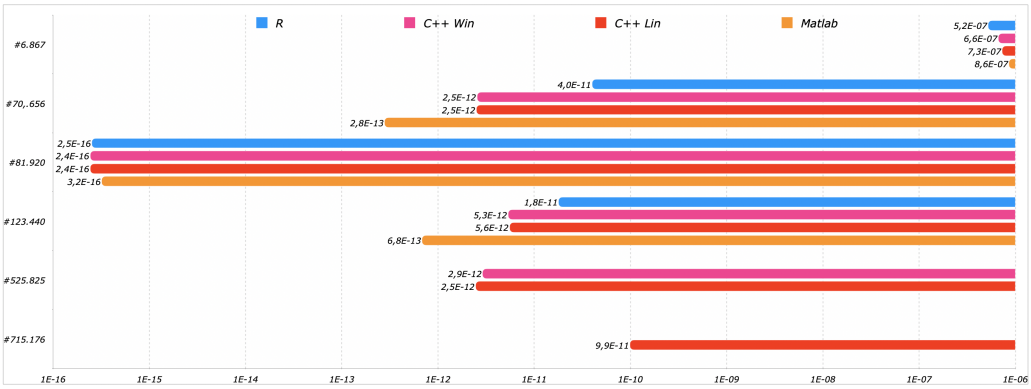
4.3. Time resolve

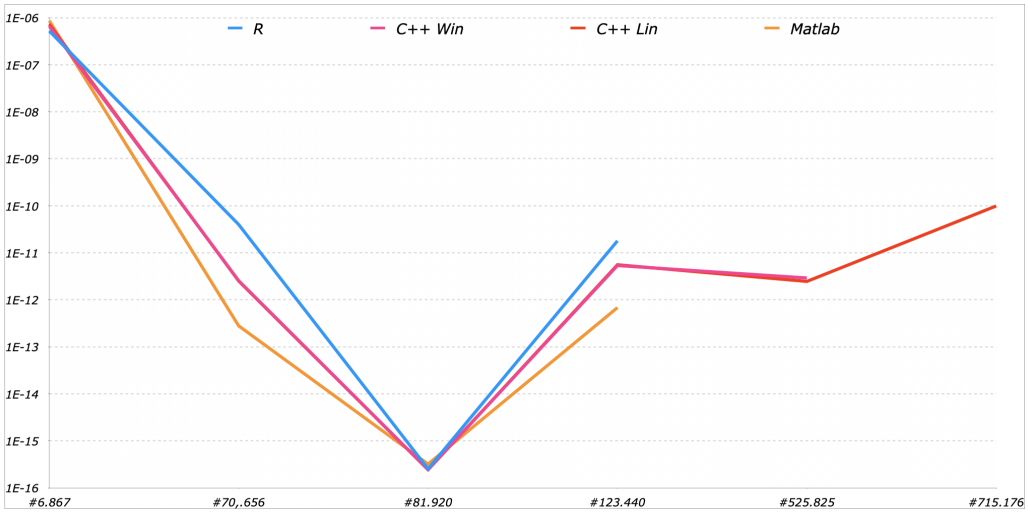




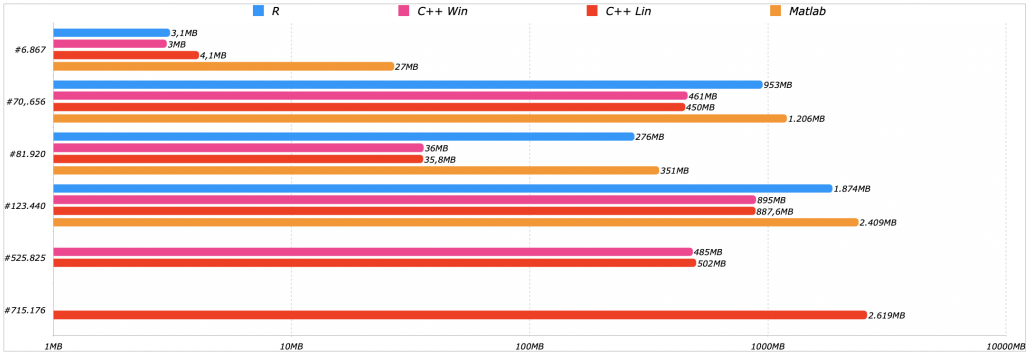
Questi grafici rappresentano il tempo impiegato per la risoluzione del sistema finale $R \times R' \times x = b$. Come si può notare i vari linguaggi sembrano avere un comportamento molto simile, e anche qui si osserva la leggera superiorità di Matlab rispetto ai suoi avversari, mentre R crolla con dei tempi più lunghi rispetto agli altri.

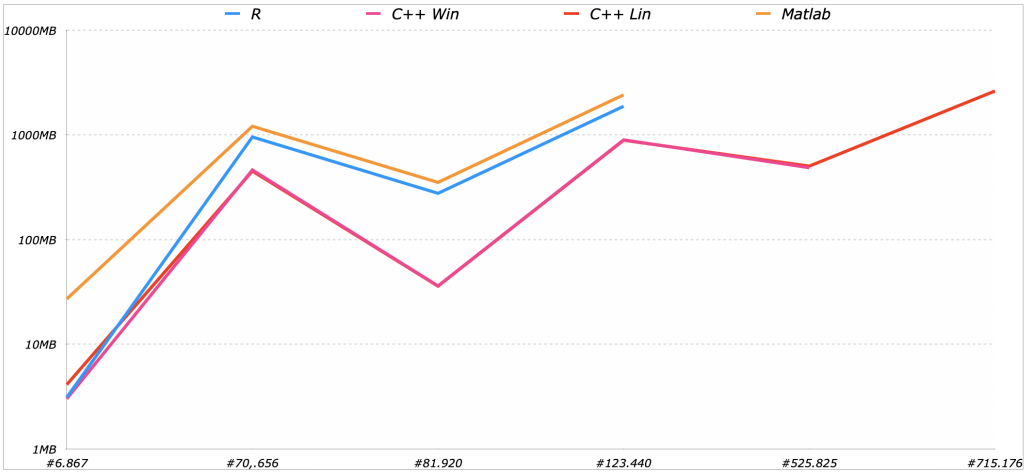
4.4. Errori relativi





Si passa ora ad analizzare da un altro punto di vista l'analisi delle diverse elaborazioni. L'errore relativo è un valore che determina con quanta precisione è stata calcolata la soluzione del sistema, in particolare più esso tende a 0 più il calcolo è stato preciso. Dalle diverse esecuzioni non si può stabilire univocamente quale sia l'ambiente migliore da questo punto di vista, forse Matlab potrebbe esserlo anche se non sempre è il più preciso.





Infine ecco la presentazione della memoria utilizzata. Questi risultati stravolgono in parte quello acquisito fino ad ora. Se infatti fino ad adesso Matlab si è rivelato il "migliore", soprattutto per quanto riguarda la velocità di esecuzione, per quanto riguarda la memoria esso è un vero e proprio *memory killer*, utilizza infatti quantitativi di memoria superiori rispetto agli altri, decisamente superiori rispetto alle implementazioni C++, infatti queste ultime riescono a gestire matrici di dimensioni maggiori senza incappare in problemi di *out of memory*.

Curioso notare come la memoria utilizzata da Matlab per la risoluzione di un sistema di 123.440 righe è dello stesso ordine di grandezza della memoria utilizzata da C++ per un sistema di 715.176 righe!